Principles of Data Analytics

Chinmay Hegde

July 8, 2017

Abstract

These are abridged lecture notes from the Spring 2017 course offering of "Principles of Data Analytics" that I offer at Iowa State University annually. This graduate level course offers an introduction to a variety of data analysis techniques, particularly those relevant for electrical and computer engineers, from an algorithmic perspective. Topics include techniques for classification, visualization, and parameter estimation, with applications to signals, images, matrices, and graphs. The material in these lecture notes are primarily adapted from the excellent online textbook, "Foundations of Data Science" (2016) by Blum, Hopcroft, and Kannan. Other resources used to construct these notes were "Advanced Algorithm Design" by Arora, "Theory Gems" by Madry, and "Convex Optimization" by Bubeck.

Contents

Chapter 1

Introduction

Overview of 525X

This course serves as an introduction to a a **variety** of **foundational principles** in data analytics for **engineering applications**.

Note that terms in **bold**:

- *Variety*: We will introduce a diverse collection of methods for solving challenges in data analysis.
- *Foundational principles*: For every method we study, we will emphasize understanding its fundamental properties: correctness, computational efficiency, potential ways to improve it, etc.
- *Engineering applications*: We will illustrate the efficacy of data analytics methods in the context of how they impact applications related to specific domains, with an emphasis on applications from electrical and computer engineering.

The science of data processing

Why focus on the *principles* of data processing? And why now?

As of today, data analytics is an off-the-charts "hot" topic. Virtually every scientific, engineering, and social discipline is undergoing a major foundational shift towards becoming more "analytics" oriented.

There are several reasons why this is the case; the foremost being that computing devices and sensors have pervasively been ingrained intro our daily personal and professional lives. It is easier and cheaper to acquire and process data than ever before. Hence, it is common to hear and read pithy newspaper headlines about data analytics, such as this:

> *"Why data is the new oil."* (Fortune Magazine, June 2016)

Of course, qualitative claims of this scale must always be taken with a pinch of salt.

It is undoubtedly true that a renewed focus on data-driven analysis and decision making has had considerable impact on a lot of fields. It is also equally true that important questions remain unanswered, and that much of the analysis tools used by practitioners are deployed as "black-boxes".

The only way to place data analytics on a solid footing is to build it bottom-up from the principles upwards; in other words, ask the same foundational questions, as you would for any other scientific discipline.

What this course is (not) about

This course addresses data analysis from a few different angles (while not exclusively dwelling upon any single one of them):

- machine learning
- algorithm design
- statistics
- optimization
- "big data" processing

Prior knowledge/experience in any of the above topics will be a plus, but not necessary; the course will be fairly self-contained.

Learning outcomes

At the end of 525X, we will have:

- become familiar with several commonly used methods for solving data analysis problems:
  - acquisition
  - compression
  - retrieval
  - classification
  - inference
  - visualization
- applied these methods in the context of several data models, including:
  - vectors
  - matrices
  - graphs
- obtained hands-on experience for applying these methods on real-world datasets.

Good luck!

Chapter 2

Vector representations of data

In several (most?) applications, "data" usually refers to a list of numerical attributes associated with an object of interest.

For example: consider meteorological data collected by a network of weather sensors. Suppose each sensor measures:

- wind speed (w) in miles per hour
- temperature (t) in degrees Fahrenheit

Consider a set of such readings ordered as tuples (w, t); for example: (4,27), (10,32), (11,47), ...

It will be *convenient* to model each tuple as a point in a two-dimensional vector space.

More generally, if data has n attributes (that we will call *features*), then each data point can be viewed as an element in a d-dimensional vector space, say $\mathbb{R}^d$.

Here are some examples of vector space models for data:

1. Sensor readings (such as the weather sensor example as above).
2. Image data. Every image can be modeled as a vector of pixel intensity intensity values. For example, a 1024×768 RGB image can be viewed as a vector of $d = 1024 \times 768 \times 3$ dimensions.
3. Time-series data. For example, if we measure the price of a stock over $d = 1000$ days, then the aggregate data can be modeled as a point in 1000-dimensional space.

Properties of vector spaces

Recall the two fundamental properties of vector spaces:

- Linearity: two vectors $x = (x_1, \ldots, x_d)$ and $y = (y_1, \ldots, y_d)$ can be *added* to obtain:

$$x + y = (x_1 + y_1, \ldots, x_d + y_d).$$

- Scaling: a vector $x = (x_1, \ldots, x_d)$ can be scaled by a real number $\alpha \in \mathbb{R}$ to obtain:

$$\alpha x = (\alpha x_1, \ldots, \alpha x_d).$$

Vector space representations of data are surprisingly general and powerful. Moreover, several tools from linear algebra/Cartesian geometry will be very useful to us:

1. Norms. Each vector can be associated with a *norm*, loosely interpreted as the "length" of a vector. For example, the ℓ_2, or *Euclidean*, norm of $x = (x_1, \ldots, x_d)$ is given by $\|x\|_2 = \sqrt{\sum_{i=1}^d x_i^2}$. The ℓ_1, or *Manhattan*, norm of x is given by $\|x\|_1 = \sum_{i=1}^d |x|_i$.
2. Distances. Vector spaces can be endowed with a notion of distance as follows: the "distance" between x and y can be interpreted as the norm of the vector $x - y$. For example, the ℓ_2, or Euclidean, distance between x and y is given by:

$$\|x - y\|_2 = \sqrt{\sum_{i=1}^d x_i^2}.$$

   One can similarly define the ℓ_1-distance, etc. The choice of distance will be crucial in several applications when we wish to compare how close two vectors are.

3. Similarities. (These are, in some sense, the opposite of distance.) Define the Euclidean *inner product* between vectors x and y as:

$$\langle x, y \rangle = \sum_{i=1}^d x_i y_i.$$

   Then, the *cosine* similarity is given by:

$$\text{sim}(x, y) = \frac{\langle x, y \rangle}{\|x\|_2 \|y\|_2}.$$

   The *inverse cosine* of this quantity is the generalized notion of *angle* between x and y.

Application: Document retrieval

Let us instantiate these ideas for a concrete warmup application in data analytics. For the purpose of this discussion, let a *document* refer to any collection of words in the English language (this could be a (literal) document, or a webpage, or a book, or just a phrase, etc.) Consider the following

> Problem: Given a database of n documents $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ and a query document D^*, find the document in the database that best matches D^*.

For example, if you were to build a search engine over a database of webpages, $\mathcal{D}$ is your webpage database, and any set of input keywords can be your query "document".

As with most problems in data analytics, the first (and most crucial) course of action is to decide a suitable *representation* of your data. Once we do that, then we can think of applying some combination of the aforementioned linear algebraic tools.

Stage 1: Modeling the database in a vector space

Can we somehow naturally model "documents" in English as "vectors" in some space?

There is no unique way to do this, but here is a first attempt. Let d denote the number of all words in the English language. (This is a very large number, and depending on how you define "English word", this can range from 10,000 to more than half a million.) Index all English words from 1 through d.

Convert every document into a d-dimensional vector x, by simply letting the j^{th} co-ordinate of x to be the **number** of occurrences of word j in the document. (This number is also sometimes called the "term-frequency".)

This gives us a set of n vectors $\{x_1, \ldots, x_n\}$, where each x_i is an element of ${\mathbb{R}_+}^d$, and represents the i^{th} document.

This has the deficiency of throwing out *context* in a document. For example, consider the following two (short) "documents":

> The quick brown fox jumped over the lazy dog.
>
> The lazy brown dog jumped over the quick fox.

will both have the **exact** same vector representation, even though they are clearly different documents. But let's assume for now that we won't face this ambiguity issue in actual applications.

Do the same to the query as well, so that we also obtain a query vector x^*.

Stage 2: Finding the best match

Once we have a vector representation of every document, as well as the query vector x^*, we now have to define what "best match" means. Again, there is no unique way to do this.

In text analysis, a common measure of "best" is the cosine similarity. More precisely, we calculate, for each i, the cosine similarity between the query vector with each other vector in the database:

$$\cos\theta_i = \frac{\langle x^*, x_i \rangle}{\|x^*\|_2 \|x_i\|_2},$$

and output the index i^* as __ :

$$i^* = \arg\max_i \cos\theta_i.$$

An improved approach

The above method works well, but the vector space representation is a bit brittle. Specifically, the definition on term frequency means that certain commonly occurring words ("the", "be", "to", "of", "and", etc) will constitute the dominant coordinate values of each document.

This heavily inflates the influence of the co-ordinates of the vector space corresponding to these words. However, generally these words are uninformative and their effect should be ideally minimized.

In order to do this coherently, we define a new vector space representation. For document i, we construct a vector x_i such that the j^{th} co-ordinate is given by:

$$x_i(j) = \text{tf}_i(j) \cdot \text{idf}(j).$$

The term $\text{tf}_i(j)$ is the same as before; it represents *term-frequency* and counts the number of occurrences of word j in document i.

The term $\text{idf}(j)$ is called the *inverse-document* frequency, and is defined as follows. Let n_j be the number of documents in the database which contain at least one occurrence of word j. Then,

$$\text{idf}(j) = \log \left(\frac{n_j}{n} \right)^{-1} = \log \frac{n}{n_j}.$$

Therefore, if a word occurred in every document, its idf value would be zero, and the word would not contribute to the similarity calculations.

Note that the idf value is the same for all documents, and is only a property of the database under consideration.

Concluding note

Much of the rest of this course will involve solving problems (more challenging) than the one above. The emphasis will primarily be on Stage 2: given the "right" vector space representation of a particular dataset, what mathematical tools and techniques are available to us for further analysis?

However, the first stage is where most of the ingenuity of data analysis arises. Indeed, choosing the "right" representation (or features) of the data is somewhat of an *art* and relies heavily on domain expertise.

(Note that there has been some preliminary, but very energetic, progress in the area of *deep learning*; in deep learning, the main objective is to automate the process of choosing the "right" representation for data.)

Chapter 3

Nearest neighbors

We discussed a simple document retrieval system using TF-IDF as the feature representation of the data, and cosine similarity as the measure of "goodness". This lecture, we will generalize this algorithm into something that we call as the *Nearest Neighbor* method.

The algorithm

Suppose that we have a database of data points $\{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}^d$, and a query data point $x_0 \in \mathbb{R}^d$. Also suppose that we are given a distance measure $d(\cdot, \cdot)$ that measures closeness between data points. Typical choice of distances $d(\cdot, \cdot)$ include the ℓ_2-distance (or ℓ_1-distance).

The nearest neighbor (NN) method advocates the following (intuitive) method for finding the closest point in the database to the query.

1. Compute distances $d_i = d(x_i, x_0)$ for $i = 1, 2, \ldots, n$.
2. Output $i^* = \arg\min_{i \in [n]} d(x_0, x_0)$.

Particularly simple! The above two steps form a core building block of several, more complicated techniques.

Efficiency

The NN method makes no assumptions on the distribution of the data points and can be generically applied; that's part of the reason why it is so powerful.

To process each given query point, the algorithm needs to compute distances from the query to n-points in d dimensions; for ℓ_1 or ℓ_2 distances, each distance calculation has a running time of $O(d)$, giving rise to an overall running time of $O(nd)$.

This is generally OK for small n (number of samples) or small d (dimension), but quickly becomes very large. Think of n being in the $10^8 - 10^9$ range, and d being of a similar order of magnitude. This is very common in image retrieval and similar applications.

Moreover: this is the running time incurred for *each* new query. It is typical to possess one large (training) database of points and make repeated queries to this set. The question now is whether we can improve running time if we were allowed to do some type of preprocessing to the database to speed up running time of finding the nearest neighbor.

Improving running time: the 1D case

Consider the one-dimensional case ($d = 1$). Here, the data points (and query) are all scalars. The running time of naive nearest neighbors is O(n). Can we improve upon this?

Yes! The idea is to use a divide-and-conquer strategy.

Suppose the data points are given by $\{x_1, x_2, \ldots, x_n\}$. We *sort* the data points in increasing order to obtain the (permuted version of the) data set $\{x_{\pi_1}, x_{\pi_2}, \ldots, x_{\pi_n}\}$. This takes $O(n \log n)$ time using MergeSort, etc.

Now, for each query point x_0, we simply perform *binary search*. More concretely: assuming that n is even, we compare x_0 to the median point. $x_{\pi_{n/2}}$. If $x_0 > x_{\pi_{n/2}}$, then the nearest neighbor to x_0 cannot belong to the bottom half $\{x_1, \ldots, x_{\pi_{n/2-1}}\}$. Else, if $x_0 < x_{\pi_{n/2}}$, then the nearest neighbor cannot belong to the top half $\{x_{\pi_{n/2+1}}, \ldots, x_{\pi_n}\}$. Either way, this discards half the number of points from the database. We recursively apply this procedure on the remaining data points.

Eventually, we will be left with a single data point x_j. We output $i^* = \pi^{-1} j$ as the index of the nearest neighbor in the (original) database.

Since the dataset size decreases by a factor 2 at each recursive step, the number of iterations is at most $\log_2 n$. Therefore, the running time of nearest neighbors for each query is $O(\log n)$.

So by paying a small additional factor ($O(\log n)$) in terms of pre-processing time, we can dramatically speed up running time per query. This kind of trick will often be used in several techniques that we will encounter later.

Extension to higher dimensions: kd-trees

The "binary search" idea works well in one dimension ($d = 1$). For $d > 1$, a similar idea called *kd-trees* can be developed. (The nomenclature is a bit strange, since "kd" here is short for "k-dimensional". But to be consistent, we will use the symbol d to represent dimension.) It's a bit more complicated since there is no canonical way to define "binary search" in more than one dimension.

Preprocessing

For concreteness, consider two dimensions ($d = 2$). Then, all data points (and the query point) can be represented within some bounded rectangle in the XY-plane.

We first sort all data points according to the X-dimension, as well as according to the Y-dimension.

Next, we arbitrarily choose a *splitting direction* along one of the two axes. (Without loss of generality, suppose we choose the X-direction.) Then, we divide the data set into two subsets according to the X-values of the data points by drawing a line perpendicular to the X-axis through the *median* value of the (sorted) X-coordinates of the points. Each subset will contain the same number of points if n is even, or differ by 1 if n is odd.

We recursively apply the above splitting procedure for the two subsets of points. At every recursive step, each of the two subsets will induce two halves of approximately the same size. One can imagine this process as an approximately balanced binary tree, where the root is the given input dataset, each node represents a subsets of points contained in a sub-rectangle of the original rectangle we started with, and the leaves correspond to singleton subsets containing the individual data points.

After $O(\log n)$ levels of this tree, this process terminates. Done!

There is some flexibility in choosing the splitting direction in each recursive call. One option is to choose the axis where the coordinates of the data points have the maximum variance. The other option is to simply alternate between the axes for $d = 2$ (or cycle through the axes in a round-robin manner for $d > 2$). The third option is to choose the splitting direction at random. Either way, the objective is to make sure the overall tree is balanced.

The overall running time is the same as sorting the n data points along each dimension, which is given by $O(dn \log n)$.

Making queries

Like in the 1D case, nearest neighbor queries in kd-trees can be made using a divide-and-conquer strategy. We leverage the pre-processed data to quickly discard large portions of the dataset as candidate nearest neighbors to a given query point.

Identical to the 1D case, the nearest neighbor algorithm starts at the root, looks at the splitting direction (in our above example, the initial split is along the X-direction), and moves left or right depending on whether the query point has its corresponding coordinate (in our above example, its X-coordinate) smaller than or greater than the median value. Recursively do this to traverse all the way down to one of the leaves of the binary tree.

This traversal takes $O(\log n)$ comparisons. Now, unlike the 1D case, there is no guarantee that the leaf data point is the true nearest neighbors (since we have been fairly myopic and only looked at each co-ordinate in order to make our decisions while traversing the tree.) So, we need to do some additional calculations to refine our estimate.

Declare the data point in the leaf as the *current estimate* of the nearest neighbor, and calculate the distance, Δ, between the current estimate and the query point. It is certainly true that the *actual* nearest neighbor cannot be further away than the current estimate. In other words, the true nearest neighbor lies within a circle of radius Δ centered at the query point.

Therefore, we only need to examine the sub-rectangles (i.e., nodes of the binary tree) that *intersect this circle*! All other rectangles (and points within those rectangles) are irrelevant.

Checking whether the circle intersects a given sub-rectangle is simple: since each rectangle is specified by a split that occurred along some X- (or some Y-) coordinate (say at value α), we only

have to check whether the difference between the splitting coordinate of the query point and α exceeds Δ. If yes, then there is no intersection; we can safely discard the sub-rectangle (and all its children). This intuition supports the divide-and-conquer strategy.

One can prove that the *expected* number of additional nodes that need to visit is given by $O(\log n)$ if the original data is generated from certain random distributions. However, in the worst case we may still need to perform $O(n)$ distance calculations, which implies a worst-case running time of $O(nd)$ (i.e., no better than vanilla nearest neighbors.)

Nevertheless, kd-trees are often used in practice, particularly for moderate dimensional problems. Most modern machine learning software packages (such as scikit-learn for Python) have robust implementations of kd-trees that offer considerable speedups over nearest neighbors.

Chapter 4

Modeling data in high dimensions

Recall that the running time of nearest neighbors / similarity search was given by $O(nd)$ where n is the number of data points and d is the dimensino.

The approach that we adopted in previous lectures for speeding up nearest neighbors was to preprocess the database and reduce the number of distance comparisons made to each given query point (i.e., essentially reduce n.)

An alternate approach is to reduce the *dimension* of the data somehow. If we were able to faithfully obtain a new, reduced-order representation of the given dataset (that preserves nearest neighbor information) then we could simply do vanilla nearest neighbors (or kd-trees, even) in the reduced space and get a considerable speedup.

Of course, it is at first unclear why such a procedure is even possible.

The curse of dimensionality

This is an umbrella phrase used to represent the idea that ordinary human intuition (that is trained to think in two or three dimensions) breaks down in higher dimensions. Basically, going to higher and higher dimensions *dramatically* changes fundamental geometric quantities (such as distances, surface areas, volumes, densities, etc), and in a very concrete sense, makes problems exponentially harder to solve.

Geometry in high dimensions

Let us instantiate this via some counter-intuitive results about geometry in high dimensions. Let $\|\cdot\|$ denote the Euclidean norm. Then, a *ball* in d-dimensions is the generalization of a circle in 2D; a ball of radius r centered at the origin is denoted as:

$$B_d(r) = \{x \in \mathbb{R}^d \mid \|x\| \leq r\}$$

Generally, $B_d(1)$ is called the *unit ball* in d dimensions centered at the origin.

The volume of $B_d(r)$ is given by:

$$\text{vol}(B_d(r)) = \frac{\pi^{d/2}}{\Gamma(d/2+1)} r^d,$$

where $\Gamma(d/2+1)$ is the Gamma function and is equal to $(d/2)!$ if d is even, and $d!!/2^{(d+1)/2}$ if d is odd, where !! denotes the product of all odd numbers up to a given number. (This constant is unimportant for now; let's just call it C_d for simplicity.)

We can view this as the higher-dimensional generalization of the notion of area of a circle (given by πr^2) or volume of a sphere (given by $\frac{4}{3}\pi r^3$). The main point is to realize that the volume of a d-dimensional ball of radius r is proportional r^d.

We now prove:

Counter-intuitive result #1: Most of the volume of a radius-r ball in d dimensions is concentrated near its surface.

Here, "surface" refers to the set of all points with Euclidean norm exactly equal to r.

This fact is relatively easy to prove. Consider a d-dimensional ball of radius r, and also consider a *slightly* smaller ball of radius $(1-\epsilon)r$, where $\epsilon > 0$ is some tiny constant (say, 0.01). We will show that the bulk of the bigger ball lies outside the smaller one.

To see this, Consider the ratio of their volumes:

$$\begin{aligned}\frac{\text{vol}(B_d(1-\epsilon)r)}{\text{vol}(B_d(r))} &= \frac{C_d(1-\epsilon)^d r^d}{C_d r^d} \\ &= (1-\epsilon)^d \\ &\leq \exp(-\epsilon d)\end{aligned}$$

where the last inequality follows from the Taylor series expansion of the exponential function.

This tells us that volume of the slightly-smaller-ball (relative to the bigger one) decays *exponentially* with the dimension d.

Example: setting $\epsilon = 0.01$ and $d = 1,000$, we infer that 99.99546% of the volume of the unit ball in $d = 1000$ dimensions is within a "shell" of width 0.01 close to the surface. Strange!

We also prove:

Counter-intuitive result #2: Most of the volume of a d-dimensional ball is concentrated near its equator.

The equator of a ball is defined as follows: **arbitrarily** choose a direction in d-dimensional space. Without loss of generality, suppose that we choose the first coordinate axis. Then, the equator is denoted as the set of all points whose first coordinate is zero:

$$E = \{x = (x_1, x_2, \ldots, x_d) \mid x_1 = 0\}.$$

i.e., the $d-1$-dimensional hyperplane perpendicular to the direction we choose.

We can prove that a thin "strip" near this equator:

$$S = \{x = \{x_1, x_2, \ldots, x_d\} \mid |x_1| \leq 0.03\}$$

contains over 99% of the volume of the sphere for $d = 10,000$. The proof follows from fairly basic (but messy) multi-variate integration. See Chapter 2 of the book for a formal proof.

Here is the interesting part: the above fact is true *no matter how we choose the equator*. E.g., if we chose the equator as a hyperplane perpendicular to (say) the 36th coordinate axis, then the above statement would hold for a strip close to this equator as well.

What's more, our choice need not be a direction aligned to any one coordinate axis; *any* direction is equally valid. Again, very strange!

Chapter 5

Dimensionality reduction

In this lecture, we propose an alternate way to speed up nearest neighbors (with Euclidean distances) using the concept of *dimensionality reduction*. Specifically, when d is large, we will demonstrate the existence of a mapping $f : \mathbb{R}^d \to \mathbb{R}^k$ that approximately preserves Euclidean distances:

$$\|f(x) - f(y)\| \approx \|x - y\|$$

for all data points x and y in the given dataset, where $\|\cdot\|$ denotes the Euclidean norm. In particular:

- the mapping f is *oblivious* to the dataset under consideration, and can indeed be constructed independently of the data.
- k can be much smaller than the dimension d, and only depends on the number of data points n.
- f is simple to construct (in fact, it is a linear mapping.)
- there is a nice inverse relationship between the quality of approximation and the dimensionality of the mapping: if you are willing to tolerate a bit of distortion,

Because f (approximately) preserves all Euclidean distances, one can transparently implement nearest neighbors on the transformed data points under the mapping (as opposed to the original data). Since the transformed data is of much lower dimension, the complexity of nearest neighbors can be alleviated to a considerable extent.

The Johnson-Lindenstrauss Lemma

The Johnson-Lindenstrauss lemma is a famous result in functional analysis. The original result was somewhat dense, but has been simplified over the years. We provide a self-contained proof.

Below, let $\|\cdot\|$ denote the Euclidean norm. We will prove the following:

Lemma: Let $X \subset \mathbb{R}^d$ be any set with cardinality n. Let $f(x) = \frac{1}{\sqrt{k}}Ax$, where A is a matrix of size $k \times d$ whose entries are independently sampled from the standard normal distribution, $\mathcal{N}(0, 1)$. Fix $0 < \epsilon < 1$. Then, provided $k > O(\frac{\log n}{\epsilon^2})$, the following statement holds with high probability:

$$(1 - \epsilon)\|x - y\|^2 \leq \|f(x) - f(y)\|^2 \leq (1 + \epsilon)\|x - y\|^2$$

for all $x, y \in X$.

The proof follows fairly elementary probabilistic arguments. Fix any $v \in \mathbb{R}^d$, and consider $z = \frac{1}{\sqrt{k}} Av$. Then, the ith coordinate of z,

$$z_i = \frac{1}{\sqrt{k}} \sum_{j=1}^{d} A_{i,j} v_j$$

is a weighted linear combination of independently sampled normal random variables. Therefore, each z_i itself is a normal random variable. Moreover, from properties of Gaussian distributions, we have, for each i:

$$E[z_i] = 0, \;\; \text{Var}(z_i) = \frac{1}{k} \sum_j v_j^2 = \frac{\|v\|^2}{k}$$

Therefore, the random variable $\bar{z}_i = \frac{\sqrt{k}}{\|v\|} z_i$ behaves like a normal r.v. with zero mean and unit variance. Hence, squaring and summing over i, the random variable:

$$\chi_k^2 = \sum_{i=1}^{k} \bar{z}_i^2 = \frac{k}{\|v\|^2} \sum_{i=1}^{k} z_i^2$$

behaves like a *chi-squared* random variable with k-degrees of freedom.

It is well-known that the chi-squared random variable is *tightly concentrated around its mean* for large enough k; in other words, the probability that χ_k^2 deviates from its mean by a certain percentage is exponentially decaying in k. More precisely:

$$E[\chi_k^2] = k,$$

and

$$Pr[|\chi_k^2 - k| > \epsilon k] \leq 2 \exp(-\frac{k}{4}(\epsilon^2 - \epsilon^3)).$$

The above statement can be derived by a straightforward application of Markov's inequality.

Plugging in the expression for χ_k^2 in the above deviation bound, the probability that $\frac{1}{k}\|Av\|^2$ deviates from $\|v\|^2$ by more than a fraction $(1 \pm \epsilon)$ is upper bounded by a "failure probability" term that is exponentially small in terms of k.

Now, this is true for any fixed v. We instantiate v with each of the difference vectors, $v = x_i - x_j$, between all pairs of data points. There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs, and we have $\binom{n}{2}$ such deviation inequalities, each with a tiny failure probability. By the union bound, the total failure probability is upper bounded by the sum of the individual probabilities:

$$\begin{aligned}
\eta &\leq \frac{n(n-1)}{2} 2 \exp(-\frac{k}{4}(\epsilon^2 - \epsilon^3)) \\
&\leq n^2 \exp(-\frac{k}{4}(\epsilon^2 - \epsilon^3)) \\
&= \exp(2 \log n - \frac{k}{4}(\epsilon^2 - \epsilon^3)).
\end{aligned}$$

We can ignore the ϵ^3 term for sufficiently small ϵ. Insert $k = O(\frac{\log n}{\epsilon^2})$ to get the desired result.

Random projections and nearest neighbors

For reasons clear from the construction, the mapping $f(x) = Ax/\sqrt{k}$ is called a *random linear projection*, or *random projection* for short.

The J-L Lemma motivates a simple way to accelerate the nearest neighbor algorithm. Given data points $X = \{x_1, x_2, \ldots, x_n\}$:

1. Compute random projections $\{f(x_i)\}, i = 1, 2, \ldots, n$.
2. Compute the random projection of the query point $f(x_0)$.
3. Compute Euclidean distances between the (projected) query point with the (projected) data points: $d_i = \|f(x_i) - f(x_0)\|^2$.
4. Output $i^* = \arg\min_i d_i$.

The running time of Steps 3-4 combined is given by $O(nk) = O(\frac{n \log n}{\epsilon^2})$ (since it is basically nearest neighbors in k dimensions). This can be significantly lesser than d, especially in situations where d is large.

Some care should be taken in terms of *correctness* of the algorithm; because distances are getting distorted, there is a possibility that the nearest neighbor in the projected space is different from that of the original space. There are more sophisticated nearest neighbor methods (e.g., Locality Sensitive Hashing) which take a more refined look at this issue, but we won't discuss them here.

Moreover, we glossed over the running times of Steps 1-2. Step 1 requires multiplying each d-dimensional data point by a $k \times d$ matrix. There are n such points. Therefore the total computational cost is $O(nkd) = O(nd\frac{\log n}{\epsilon^2})$ – bigger than $O(nd)$, but this is a one-time "training" cost.

Step 2 requires multiplying a single d dimensional point by a $k \times d$ matrix. This takes $O(kd) = O(d \log d/\epsilon^2)$. Again, not too bad as long as ϵ is manageable. If the inverse dependence on ϵ^2 is bothersome, then there exist alternative constructions of A that avoid this; see below.

Bottom line: if your data is very high-dimensional, a random projection into lower dimensional space is often a good, quick, *universal* pre-processing step to reduce computational burden. This is the reason why random projections can be used in conjunction with pretty much any other data analysis technique that relies on pairwise distance computations between a set of data points.

Concluding notes

Some points to keep in mind:

- The above works for nearest neighbors based on Euclidean distances. Analogous results can also be derived for *similarity* searches (based on inner-products).
- Gaussian random matrices are only a specific example for J-L to work, and a wide variety of other matrix constructions exist, several with much better running times. (Note that generating very large Gaussian random matrices poses a challenge in itself; better constructions exist.)
- Somewhat bafflingly, it is quite hard to build similar efficient dimensionality-reduction mappings that are suitable for preserving distances other than the Euclidean norm. (For example,

there is no principled way to considerably reduce data dimension if we want to do NN based on the ℓ_1-distance.) There seems to be something singular about the Euclidean distance that enables a fairly dramatic level of dimensionality reduction.

Chapter 6

Classification

We will now transition to **classification** problems, which constitute an extremely important class of problems in data analytics.

At a high level: a classification procedure, or *classifier* for short, processes a given set of objects that are *known* to belong to one of two (or more) classes. Then, it attempts to predict the class membership of a new object whose class membership is unknown. (The idea of processing a new, potential unseen object is key here.)

Classification problems arise in widespread applications ranging from automatic image tagging, to document categorization, to spam filtering, to sensor network monitoring, to software bug detection – you name it. It is one of the canonical pattern recognition problems. We will discuss a range of techniques for classification, starting with one that is particularly easy to implement.

k-nearest neighbors

We study the binary classification problem. Here, and below, we will assume as input:

- A set of *labeled* training samples: $X = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where each $x_i \in \mathbb{R}^d$ represent data vectors, each $y_i \in \{-1, 1\}$ represent class labels. (Here, the ± 1 don't represent numerical values; they could equally well be 0 or 1 with little change in the algorithms.)
- An *unlabeled* test data point $x \in \mathbb{R}$. Our goal is to output the label of x, denoted by y.

A reasonable hypothesis is that data points which have opposite labels are far away in the data space, and points with similar labels are nearby. Therefore, an obvious solution for the above problem can be obtained via nearest neighbors.

1. Compute distances $d(x, x_i)$ for each $i = 1, 2, \ldots, n$. Here, think of d as the Euclidean norm or the ℓ_1 norm. (The exact choice of distance is critical.)
2. Compute the nearest neighbor to x, i.e., find $i^* = \arg\min_i d(x, x_i)$.
3. Output $y = y_{i^*}$.

Therefore, this algorithm is a straightforward application of nearest neighbors. The running time (for each test point) is $O(nd)$.

While conceptually simple, there are several issues with such a nearest neighbors approach. First, it is notoriously sensitive to errors and outliers. For example, in an image recognition example, say we have a bunch of (labelled) cat and dog images, but some of the dog images are mislabeled as cats (and vice versa). Then, a new dog image that is closest to a mislabeled dog image will also get a wrong label assignment using the above algorithm.

The typical way to use this is via *k-nearest neighbors*. Find the k-nearest data points to the query x, look at their labels, and output the majority among the labels. For convenience of defining majority, k can be chosen to be an odd integer. (Even is fine as well, but 50-50 ties may occur and there should be a consistent rule to break them.) This helps mitigate outliers to some extent.

The running time of the k-NN algorithm is again $O(nd)$ for each query. This brings us to the second issue with NN (and k-NN): the *per-query* running time is quadratic (i.e., it scales with both n and d). This quickly becomes infeasible for applications involving lots of query points (e.g. imagine a spam classification algorithm trying to label millions of messages).

Occam's razor and linear separators

Every (deterministic) binary classification algorithm A, given a training data set X, automatically partitions the data space into two subsets – the set of points that will be labeled as $+1$ and the set of points that will be labeled as -1. These are called *decision regions* of the classifier. Really, the only information you need for classification are the decision regions (the role of training data is to help build these regions.)

The third issue with k-NN is that the decision regions are overly complex. The boundary of the two classes using NN classifiers is an irregular hypersurface that depends on the training data.

As a guiding principle in machine learning, it is customary to follow:

> Occam's Razor: Simpler hypotheses usually perform better than more complex ones.

One family of "simple" classifiers are *linear* classifiers, i.e., algorithms that try to lean decision regions with a straight line as the separating boundary. We will try to find such classifiers.

The perceptron algorithm

The perceptron algorithm was an early attempt to solve the problem of artificial intelligence. Indeed, after its initiation in the early 1950s, people believed that perfect classification methods were not far off. (Of course, that didn't quite work out yet.)

The goal of the perceptron algorithm is to learn a linear separator between two classes. The algorithm is simple and parameter-free: no excessive tuning is required. Later, we will see that the algorithm is an instance of a more general family of learning methods known as *stochastic gradient descent*.

In particular, the perceptron will output a vector $w \in \mathbb{R}$ and a scalar $b \in \mathbb{R}$ such that for each input data vector x, its predicted label is:

$$y = \text{sign}(\langle w, x_i \rangle + b).$$

Below, without loss of generality, we will assume that the learned separator w is homogeneous and passes through the origin (i.e., $b = 0$), and that the data is normalized such that $\|x\| \leq 1$. Geometrically, the boundary of the decision regions is the hyperplane:

$$\langle w, x \rangle = 0$$

and w denotes any vector normal to this hyperplane.

We will also assume that the training data is indeed perfectly separable. (Of course, in practice we are not guaranteed that this is the case. We will see how to fix this later.)

Input: Training samples $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$.

Output: A vector w such that $y_i = \text{sign}(\langle w, x_i \rangle)$ for all $(x_i, y_i) \in S$.

0. Initialize $w_0 = 0$.
1. Repeat:
   a. For each $(x_i, y_i) \in S$, if $\text{sign}(\langle w_{t-1}, x_i \rangle) \neq y_i$, update

      $$w_t \leftarrow w_{t-1} + y_i x_i$$

      and increment t.
   b. If no change in w_t after sweeping through the data, terminate.

While epoch $\leq 1, .., maxepochs$.

The high level idea is that each time a training sample x_i is mislabeled, we incrementally adjust the current estimate of the separator, w, in the direction of x_i to correct for it.

Analysis of the perceptron

We now discuss algorithm correctness and efficiency.

The per-iteration complexity of the algorithm is $O(nd)$: we need to sweep through each of the n data points x_i, compute the d-dimensional dot-product $\langle w, x_i \rangle$ and compare its sign with the true label.

The algorithm makes an update only when a training sample is mislabeled, and hence terminates only when every point is labeled correctly. So it might seem that the number of iterations can be very large. We will provide an upper bound for this.

For any given dataset and a separator w, define the *margin* as the minimum projection distance between any data point to the hyperplane $\langle w, x \rangle = 0$ i.e.,

$$\gamma = \arg \min_{i \in 1, \ldots, n} |\langle w, x_i \rangle|.$$

The optimal separator is defined as the vector w_* that maximizes γ over all valid separators. Without loss of genelarity, assume that the Euclidean norm of w_* is 1.

We prove that the perceptron will terminate after $1/\gamma^2$ updates.

First, consider the quantity $\langle w_t, w_* \rangle$, i.e., the dot-product between the current estimate and the optimal separator. If the label y_i is positive, then $\langle w_{t+1}, w_* \rangle = \langle w_t + x_i, w_* \rangle$, which by linearity of the dot-product is larger than $\langle w_t, w_* \rangle$ by at least γ (by definition of the margin). The same result holds if the label is negative; easy to check.

Therefore, after each update step, $\langle w, w_* \rangle$ increases by at least γ.

Next, consider the quantity $\|w_t\|^2 = \langle w_t, w_t \rangle$. After each update, this quantity changes to

$$\begin{aligned} \|w_{t+1}\|^2 &= \langle w_{t+1}, w_{t+1} \rangle \\ &= \langle w_t + x_i, w_t + x_i \rangle \\ &\leq \|w_t\|^2 + \|x_i\|^2 \\ &\leq \|w_t\|^2 + 1, \end{aligned}$$

since we have assumed that $\|x_i\| \leq 1$ for all i.

Putting the two together, we have, after T update steps:

$$\langle w_T, w_* \rangle \geq \gamma T$$

and

$$\|w_T\|^2 \leq T \rightarrow \|w_T\| \leq \sqrt{T}.$$

By the Cauchy Schwartz inequality and the fact that $\|w_*\| = 1$, we know that

$$\langle w_T, w_* \rangle \leq \|w_T\| \|w_*\| = \|w_T\|.$$

Therefore,

$$\gamma T \leq \sqrt{T} \rightarrow T \leq 1/\gamma^2.$$

Chapter 7

Kernel methods

The perceptron algorithm is simple and works well. However, it (provably) works only when the data is *linearly* separable. If not, its performance can be poor.

Datasets in applications can be far more complicated. Consider the case of two dimensional data ($d = 2$). We will consider a toy hypothetical example where the data corresponds to measurements that you collect from a pair of sensors observing a system at various instants. The goal is to classify the system state as properly functioning or not.

Two examples of non-separable acquired measurements for this application are as follows:

- XOR-distributed data: Let's say that the system is properly functioning if both sensor readings are consistent, i.e., they are both positive-valued, or both-negative valued. Therefore, the "+"-labeled examples all lie in the first and third quadrants, while the "-"-labeled examples all lie in the second and fourth quadrants. Here, the ideal classifier is given by:

$$y = \text{sign}(x_1 x_2)$$

- Radially-distributed data: Let's now say that the system is properly functioning if the Euclidean norm of the sensor readings is smaller than some amount. Therefore, the "+" examples all lie within a circle of radius R from the origin, and "-" examples lie outside this circle. Here, the ideal classifier is given by

$$y = \text{sign}(x_1^2 + x_2^2 - R^2)$$

In both cases, no linear separator of the form $y = \text{sign}(w_1 x_1 + w_2 x_2 + b)$ can separate the two classes. The perceptron algorithm would fail on such datasets.

The key idea in kernel methods is to map the given data to a *different* (typically, higher-dimensional) feature space:

$$x \to \phi(x)$$

using some nonlinear function $\phi(\cdot)$.

For example, in the above two dimensional examples, suppose we transformed each data point into *six*-dimensional space using the feature mapping:

$$x = (x_1, x_2) \to \phi(x) = (1, x_1, x_2, x_1 x_2, x_1^2, x_2^2).$$

Then we find that in the new feature space, both the XOR and the circle examples are perfectly separable. (Exercise: find a separating hyperplane for these two examples by explicitly writing out a suitable w for each case.) If we ran the perceptron on these new examples, it would perfectly work.

The kernel trick

Two immediate questions are:

> How to choose the feature mapping function ϕ?

This is tricky to answer, and is heavily dataset dependent. The short answer is to use *all* possible features that you can think of. (The risk here is that you might overfit.)

For example, given the data vector $(x_1, x_2, \ldots, x_d)$, common features include:

- the original features as is;
- a quadratic function of the features $x_1^2, x_2^2, \ldots, x_d^2, x_1x_2, x_2x_3, \ldots, x_ix_j, \ldots$. These are particularly useful for classifying datasets which have circles, ellipses, parabolas, or other conic surfaces as separating surfaces.
- any polynomial combination of the features $x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_d^{\alpha_d}$, where $\alpha_i \geq 0$ is an integer and $\sum_i^r \alpha^i \leq r$.
- nonlinear features of the co-ordinates, such as $\sin(x_i)$, $\cos(x_i)$, or any combinations of the above.
- other domain specific features. For example, in image classification applications, the input features correspond to pixel intensity values, and there are standard features (such as corners, edges, etc) that can be computed from the input pixel intensities.

The process of mapping input feature vectors x into a higher-dimensional space $\phi(x)$ is known as *lifting*. Lifting only increases separability if carefully designed. However, this leads to the second question:

> Is this procedure computationally efficient?

Even in simple cases, the computational complexity of lifting into a higher-dimensional space can quickly become untenable. In the above sensor examples, we created 6 features from $d = 2$ features. For general d-dimensional data, this number rises to $1 + 2d + \binom{d}{2}$ (the constant offset 1, the original features, squares of the original features, and all pairwise combinations.) This is already $O(d^2)$, and infeasible for d very large.

In general, if we wanted all degree r polynomial features, the dimension of the lifted feature space is $O(d^r)$. How do we deal with such large feature spaces?

A particularly simple solution can be achieved via the *kernel trick*. The key is to realize that in the higher-dimensional space, the only computations that we perform on the data is a bunch of dot-products. (For example, in the perceptron algorithm we only need to check a bunch of times the sign of $\langle w, x \rangle$) for different x.

Therefore, in most cases *we do not need explicitly compute the features* (however high-dimensional they are); all we need is a mechanism to compute dot products. The kernel trick is merely the

observation that for certain feature mappings, the dot-product in the feature space can be computed efficiently.

Formally: given a feature mapping $x \to \phi(x)$, the kernel inner product of a pair of vectors is given by:

$$K(x, z) = \langle \phi(x), \phi(z) \rangle$$

For *certain* feature mappings, the kernel inner product is efficiently computable. Consider the quadratic kernel:

$$(x_1, x_2, \ldots, x_d) \to (1, \sqrt{2}x_1, \ldots, \sqrt{2}x_d, x_1^2, \ldots, x_d^2, \ldots, \sqrt{2}x_i x_j, \ldots)$$

(The $\sqrt{2}$ factors are for convenience only and do not affect computation using perceptrons, etc. The net result is that the corresponding co-ordinates of the learned w vectors will be scaled by $1/\sqrt{2}$.)

The kernel inner product is:

$$K(x, z) = 1 + 2x_1 z_1 + \ldots 2x_d z_d + x_1^2 z_1^2 + \ldots x_d^2 z_d^2 + \ldots 2x_i x_j z_i z_j + \ldots$$

But one can check that this is equal to:

$$\begin{aligned} K(x, z) &= (1 + \sum_{i=1}^{d} x_i z_i)^2 \\ &= (1 + \langle x, w \rangle)^2. \end{aligned}$$

Therefore, even though the feature space is $O(d^2)$ dimensional, the kernel inner product can be computed using $O(d)$ operations.

Similarly, for polynomial features of degree r, one can compute the kernel inner product using the formula:

$$K(x, z) = (c + \langle x, z \rangle)^r.$$

Essentially, the kernel inner product $K(x, z)$ measures how similar x and z are in the lifted feature space, just as how the standard inner product (dot product) $\langle w, z \rangle$ measures how similar x and z are in the original space. Therefore, *algorithmically*, one can simply replace all occurrences of $\langle ;, \cdot \rangle$ by $K(\cdot, \cdot)$, and we are done!

We do not have to go through the intermediate, expensive step of writing down the transformation $\phi(\cdot)$. In other words, instead of choosing good feature spaces (specified by ϕ), we instead implicitly seek feature spaces by looking for good kernel inner products (specified by $K(\cdot, \cdot)$).

One example of a popular kernel inner product is given by the *Gaussian radial basis function*:

$$K(x, z) = \exp(-\frac{\|x - z\|_2^2}{\sigma^2})$$

where σ is the *bandwidth* parameter. This is an example of a kernel for which the implicit feature mapping ϕ is *infinite-dimensional*, yet the kernel inner product is straightforward to compute and requires $O(d)$ time.

Chapter 8

The kernel trick

Recall the main message of kernel methods: (i) if data is not linearly separable, one can make it so by "lifting" into a (very) high dimensional space and creating several new nonlinear features; (ii) however, this causes severe computational challenges; (iii) to resolve this, we use kernel inner products $K(x, z)$ (instead of standard inner products $\langle x, z \rangle$) wherever possible in order to perform efficient computations.

This idea is surprisingly powerful, and can be useful in most machine learning algorithms that primarily rely on inner products. The overall approach is sometimes call the *kernel trick*.

Choice of kernels

Choosing a "good" kernel for a given task can be a challenge. Not all functions $K(\cdot, \cdot)$ can be chosen as kernels. The main criteria that we need to maintain are:

- Efficient computation: $K(x, z)$ must be efficiently computable for any x, z.
- Symmetry: $K(x, z) = K(z, x)$ for all $x, z \in \mathbb{R}^d$.
- Dot product property: there exists a feature mapping ϕ such that K is equal to the dot product in the feature space, i.e., $K(x, z) = \langle \phi(x), \phi(z) \rangle$.

The first two are fairly easy to verify; the last one is not. A result in applied linear algebra, known as *Mercer's Theorem*, states that a function $K(\cdot, \cdot)$ is a legitimate kernel if for *any* set of n data points $x_1, x_2, \ldots, x_n$, the symmetric $n \times n$ matrix M formed using the formula:

$$M_{i,j} = K(x_i, x_j)$$

is positive-semidefinite; i.e., it has no negative eigenvalues.

However, this is not easy to check either since this has to hold for all subsets of n points in the data space. Bottom line: be careful while choosing kernels.

Popular choices include quadratic kernels, polynomial kernels, and Gaussian RBF kernels (that we discussed previously.) Each of these can be shown to be valid kernels using Mercer's Theorem, and each of them take $O(d)$ time to compute for d-dimensional vectors.

The kernel perceptron algorithm

Let us instantiate the kernel method for learning nonlinear separators using the perceptron algorithm.

Let us say that an oracle gives us the "right" kernel that implicitly maps the data into a higher-dimensional feature space, $x \to \phi(x)$. We need to learn a separator w in this feature space.

Input: Training samples $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$.

Output: A vector w such that $y_i = \text{sign}(\langle w, \phi(x_i)\rangle)$ for all $(x_i, y_i) \in S$.

0. Initialize $w_0 = 0$.
1. Repeat:
   a. For each $(x, y) \in S$, if $\text{sign}(\langle w_t, \phi(x))\rangle) \neq y$, update
   $$w_t \leftarrow w_{t-1} + y\phi(x).$$
   b. If no change in w_t after sweeping through the data, terminate.

   While epoch $\leq 1, .., maxepochs$.

There is a catch here, though. The above algorithm involves computations with $\phi(x_i)$; however, this can be expensive.

We resolve this using two observations:

- At each time instant, the estimate w_t is a *linear combination* of vectors $\phi(x_i)$, where x_i is some list of training vectors. This is a consequence of line 1b in the algorithm, since that is the way we constructed w_t. Call this list L. Therefore,
  $$w_t = \sum_{i \in L} \alpha_i \phi(x_i).$$
  The α_i can be positive or negative, depending on how many times the data point x_i in the list has appeared in one of the updates to w, and what its label is.
- Therefore, while checking for consistency of the sign in Line 1a, the dot product with $\phi(x)$ can be evaluated as follows:
  $$\begin{aligned}\langle w_t, \phi(x)\rangle &= \langle \sum_{i \in L} \alpha_i \phi(x_i), \phi(x)\rangle \\ &= \sum_{i \in L} \alpha_i \langle \phi(x_i), \phi(x)\rangle \\ &= \sum_{i \in L} \alpha_i K(x_i, x).\end{aligned}$$
  Therefore, the dot product in Line 1a can be replaced by the kernel inner product.

The above two observations tell us that *there is no need to ever compute $\phi(x)$ explicitly*; instead, we just store the list of data points whose linear combination synthesizes w_t at each time instant, update this list as necessary, and test for consistency by using the kernel trick.

The running time of the kernel perceptron, therefore, is similar to that of the original perceptron – $O(nd)$ – assuming that (i) computing the kernel inner product $K(\cdot, \cdot)$ is as easy as computing standard inner products, and (ii) the margin γ does not change much.

Multilayer networks

Kernels provide one approach to resolve the nonlinear separability problem in realistic datasets. There is a second way (in fact, which predate kernels).

Ignore kernels for a moment. We know that a single perceptron provides a single (linear separator, i.e., divides any given feature space into two half-spaces. For any given input data point, testing with w indicates which class the point belongs to. The decision boundary between the classes is a hyperplane.

Suppose that we now consider the output of *two* perceptrons acting on the same data. Therefore, for a given input x, we have outputs:

$$y^{(1)} = \text{sign}(\langle w^{(1)}, x \rangle),$$
$$y^{(2)} = \text{sign}(\langle w^{(2)}, x \rangle).$$

Now imagine a third perceptron w^3 *that combines (pools) these outputs*, i.e.,

$$y = \text{sign}(w_1^{(3)} y^{(1)} + w_2^{(3)} y^{(2)}).$$

Clearly, y is a nonlinear function of the input x. If we start drawing decision boundaries checking which x gave rise to positive y, and which ones gave rise to negative y, then the boundaries are extremely nonlinear depending on $w^{(1)}, w^{(2)}, w^{(3)}$.

Three extensions to this basic idea:

- Why stop at two perceptrons $w^{(1)}, w^{(2)}$? We can imagine any number of such "first-level" perceptrons.
- Why stop at a single pooling perceptron? We can imagine multiple such pooling combinations, corresponding to different outputs. We can call this set of perceptrons the "output layer", and the set of first-level perceptrons the "hidden layer".
- Lastly, why stop at a single hidden layer? We can repeat this idea several times, creating several hidden layers of perceptrons; each layer combining the outputs of the previous layer, as well as serving as inputs to the next. One can show that using a sufficient number of such hidden layers, one can reproduce *any* nonlinear separating surface.

A nice visualization of this idea can be found here.

http://playground.tensorflow.org

Try varying the number of perceptrons, hidden layers, etc for the various datasets (using the original feature space). Now try the same using kernels, and see what happens. Are there any benefits in using combinations of both?

Aside

The idea behind multi-layer perceptron-based classification methods is biologically inspired. A perceptron (at least functionally) behaves like a single *neuron*. At a very basic systems level, A neuron can be modeled as a bunch of dendrites (corresponding to input features), weights associated with each dendrite, and an axon (corresponding to the output). If an input "matches" the weights of

the neuron (i.e., the inner product is high) then the output is going to be high; else, it is going to be low. Learning the weights of the perceptron is akin to *training* the neuron according to how it reacts to most inputs.

A multi-layer network is a generalization of this idea, reflecting the fact that nervous systems (and brains) are not composed of isolated neurons, but rather consists of millions (or billions, or hundreds of billions) of neurons stacked together.

Of course, while all this conceptually makes some sense, the elephant in the room is *how* we can algorithmically learn the individual perceptrons. We won't discuss this in much more detail right now – that can be the focus of an entire course altogether – but later we will discuss some basic building blocks that constitute these learning techniques.

To an extent, this question of learning neural networks is as yet an unanswered mystery. The crisp algorithms and analysis that we provided for perceptron learning is absent for this more general problem, except for certain restrictive results.

The entire field of *neural network learning* attempts to provide answers to these questions. It began as a collection of biologically-inspired heuristic algorithms, and largely remains so. More currently, the advent of *deep learning* has shown that in fact despite the lack of methods can be used to give excellent results in a wide variety of engineering applications.

Chapter 9

Support Vector Machines

The issue with the classifiers that we have discussed so far – both with linear and nonlinear separators, with or without kernelization – is that they are *brittle*, and do not always provide robust separators.

For example, if the data is separable, the perceptron returns *a* separator (one among the infinitely many separators that exist.) How can we get the *best* separator?

To answer this question, we pose the perceptron as an optimization problem. Suppose the (training) data $D = \{(x_i, y_i)\}_{i=1}^n$ is separable via a separating hyperplane (specified by w) that passes through the origin. The only condition is that $y_i = \text{sign}(\langle w, x_i \rangle)$.

Therefore, the perceptron can be interpreted as solving a *feasible solution* to the problem:

$$\begin{aligned} \text{Find} \quad & w \\ \text{s.t.} \quad & y_i \langle w, x_i \rangle \geq 0, \quad i = 1, \ldots, n. \end{aligned}$$

(Above and henceforth, "s.t." is short for "subject to constraints").

However, recall that the margin of the separator is given by

$$\gamma = \min_i |\langle w, x_i \rangle| = \min y_i \langle w, x_i \rangle,$$

when the separator w is constrained to have unit Euclidean norm ($\|w\|$ = 1). The parameter γ represents a lower bound on the projection distance of each point to the separator. In other words, there is a "dead zone" of width γ around the optimal separator in which no data point can be present.

We would like this "dead zone" to be as large as possible. Therefore, to obtain the perceptron with the *highest possible margin*, we solve the optimization problem:

$$\begin{aligned} w^* = & \arg\max_{w,\gamma} \gamma \\ \text{s.t.} \quad & y_i \langle w, x_i \rangle \geq \gamma, \quad i = 1, \ldots, n, \\ & \|w\| = 1. \end{aligned}$$

Without going too deep into optimization theory, we note that the above problem is somewhat challenging. The objective function as well as the n data constraints are linear functions in the

variables w, γ, and linear functions are generally considered to be friendly for optimization purposes. The problem arises with the Euclidean norm constraint $\|w\| = 1$, which happens to be *non-convex*. Non-convex optimization problems are not easy to solve.

We resolve this issue as below using a simple change of variables. Let us write $\bar{w} = w/\gamma$. Then, $\|\bar{w}\| = 1/\gamma$, and maximizing γ is the same as minimizing $\|\bar{w}\|$, or equivalently, minimizing $\frac{1}{2}\|\bar{w}\|^2$. Therefore, we can drop the unit norm constraint, and rewrite the above optimization problem as:

$$\begin{aligned} w^* &= \arg\min_{\bar{w}} \frac{1}{2}\|\bar{w}\|^2 \\ \text{s.t.} \quad & y_i\langle \bar{w}, x_i\rangle \geq 1, \quad i = 1, \ldots, n. \end{aligned}$$

For simplicity of notation, we will just replace $\bar{w}$ by w as the variable in the optimization problem.

The solution is called the *support vector machine* (SVM), and the above problem is called the *primal SVM* problem. The reason for this name will be explained below when we discuss duality.

The primal SVM problem is an optimization problem with n linear constraints and a quadratic objective function in d variables. This is an example of a *convex* optimization problem. Specifically, such problems are called *quadratic programs* (QP) and the specific form can be solved fairly efficiently using QP solvers. We won't go too much further into the precise details of how to solve a QP – but the best running time is given by:

$$\min(O(nd^2), O(nd^2)).$$

SVMs and dual interpretation

The origin of the name "support vectors" is a bit involved. If you are not interested in some gory optimization details, skip ahead to the end of this section; if you are, then read on.

Let us step back a bit and consider a more general optimization problem with d-dimensional variables and n inequality constraints.

$$\begin{aligned} &\min_x f(x) \\ \text{s.t.} \quad & g_i(x) \geq 0. \end{aligned}$$

We will call this the *primal problem*, or P for short, and denote its optimum as x_P.

Define a vector $\alpha \in \mathbb{R}^n$. The *Lagrangean* of this problem is defined as the new function:

$$\mathcal{L}(x, \alpha) = f(x) + \sum_{i=1}^{n} \alpha_i g_i(x).$$

The α_i's are called the Lagrangean multipliers.

The quantity

$$F(x) = \max_{\alpha_i \leq 0} \mathcal{L}(x, \alpha)$$

is a function of x. Observe if that if x violates any one of the i primal constraints in P, then $g_i(x)$ is negative, and by choosing the corresponding α_i as an arbitrary large negative number, $F(x)$ can be made arbitrarily large. On the other hand, if x satisfies all the primal constraints, the value of $g_i(x)$

is non-negative; hence, the maximum value of $\mathcal{L}(x, \alpha)$ is obtained by setting all the α_i to zero and $F(x) = f(x)$.

Therefore, the optimal value of P can be re-written as:

$$\min_x \max_{\alpha_i \leq 0} \mathcal{L}(x, \alpha)$$

Now, consider a slightly different operation on $\mathcal{L}$, except that the "min" and "max" have been interchanged; first, we minimize $\mathcal{L}(x, \alpha)$ over x:

$$G(\alpha) = \min_x \mathcal{L}(x, \alpha)$$

and then maximize with respect to α, with the constraints that $\alpha_i \leq 0$:

$$\begin{aligned} &\max_\alpha G(\alpha) \\ \text{s.t.} \quad &\alpha_i \leq 0. \end{aligned}$$

Call the above problem the *dual problem*, or D for short, and denote its optimum as α_D. Therefore, we basically solve the optimization problem:

$$\max_{\alpha_i \leq 0} \min_x \mathcal{L}(x, \alpha).$$

By flipping the order of the "min" and "max", a natural question is whether the optimal answers will be same or different. However, two important results can be proved:

- **Result 1**: a property called *Slater's condition* asserts that for usual *convex* functions f and g, strong duality holds, i.e., the x (and corresponding α) obtained by the two procedures are indeed the same.
- **Result 2**: another property called *complementary slackness* says that the optimal multipliers α_i will be non-zero **only if** the corresponding constraints $g_i(x)$ hold with equality at the optimum, i.e., $g_i(x_P) = 0$.

Again, we won't go into details of why the above are true; take a convex optimization class if you are interested to learn more.

How does all this connect to SVMs?

Observe that the primal SVM problem is an instance of P. Let us write out the Lagrangean:

$$\mathcal{L}(w, \alpha) = \frac{1}{2}\|w\|_2^2 + \sum_{i=1}^n \alpha_i (y_i \langle w, x_i \rangle - 1).$$

Now, let us construct the dual. The inner "min" optimization can be solved by taking the derivative of $\mathcal{L}$ with respect to w and setting to zero. We get:

$$\begin{aligned} \nabla_w \mathcal{L}(w^*, \alpha) = 0, \quad &\text{i.e.,} \\ w^* + \sum_{i=1}^n \alpha_i y_i x_i = 0, \quad &\text{or} \\ w^* = -\sum_{i=1}^n \alpha_i y_i x_i. \end{aligned}$$

The overall optimization is completed by figuring out the solution to the dual problem, which is given by plugging in this value of w^* into $\mathcal{L}$ and maximizing over non-positive α. After a bit of algebra, we can simplify this to:

$$\begin{aligned} \max_{\alpha} \quad & -\sum_{i=1}^{n} \alpha_i + \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \langle x_i, x_j \rangle \\ \text{s.t.} \quad & \alpha_i \leq 0, \quad i = 1, 2, \ldots, n. \end{aligned}$$

This problem is called the *dual SVM* problem. Again, similar to the primal SVM, the objective function is a quadratic function of the dual variables α, while the constraints are linear functions of α.

The above insights let us make the following observations:

> The optimal separating hyperplane w is a linear combination of the data points x_i, weighted by the labels y_i and the corresponding dual variables.

and:

> Due to Result 2, the values of α_i are only non-zero if the corresponding data points x_i **exactly** achieve the margin, i.e., they are the ones closest to the decision boundary.

Therefore, the data points which induce non-zero dual multipliers α_i are the ones which influence the final separating hyperplane, and are called the *support vectors*. This is in fact the origin of the term "support vector machine".

Therefore, to perform a classification on a new, unseen data point x, we only need to store the set of support vectors, S, i.e., the non-zero α_i and the corresponding x_i. The predicted label of x is:

$$y = \text{sign}\left(\sum_{i \in S} \alpha_i \langle x, x_i \rangle\right).$$

Extensions

The convex optimization frameworks of the SVM (both the primal and the dual) leads to two interesting extensions:

1. The dual problem only involves inner products $\langle x_i, x_j \rangle$ between data points. This naturally motivates a kernel version of SVM by replacing the standard inner product with a kernel inner product $K(\cdot, \cdot)$:

   $$\begin{aligned} \max_{\alpha} & -\sum_{i=1}^{n} \alpha_i + \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j K(x_i, x_j), \\ \text{s.t.} \quad & \alpha_i \leq 0, i = 1, 2, \ldots, n. \end{aligned}$$

   Kernel SVMs are the robust analogue of kernel perceptrons, and depending on the choice of kernel used, can handle more complicated nonlinear decision boundaries. The corresponding prediction can be perfomed, similarly as above, by only using the support vectors:

   $$y = \text{sign}\left(\sum_{i \in S} \alpha_i K(x, x_i)\right).$$

2. Suppose the data is close to linearly separable, except for some errors and outliers. For such problems, we consider a *relaxed* version of the primal SVM problem:

$$\begin{aligned} \min_{w,s} \ & \frac{1}{2}\|w\|^2 + \lambda \sum_{i=1}^{n} s_i \\ \text{s.t.} \quad & y_i \langle w, x_i \rangle \geq 1 - s_i, \\ & s_i \geq 0. \end{aligned}$$

The variables $\{s_i\}$ are called *slack* variables, and a non-zero value of s_i implies that x_i violates the margin condition, i.e., it is allowed to cross over into the "dead zone" (or even the decision boundary). Of course, we don't want too many such points to violate the margin, so we penalize the sum of the s_i's in the objective function. Here, λ is a user-defined weight parameter that trades between the violating data points and the final margin. This formulation is called the *soft-margin* primal SVM, and the original one (without slack variables) is called the *hard-margin* primal SVM.

Chapter 10

Regression

Classification is a special case of a more general class of problems known as *regression*.

As before, we have training data points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ and there is some (unknown) functional relationship between x_i and y_i – except now, the label y_i can be any real number. (In classification, the labels were assumed to be ± 1). The goal in regression is to discover a function f such that $y_i \approx f(x_i)$.

In the simplest case, we assume a *linear* model on the function (i.e., the value y_i is a linear function of the features x_i). In this case, the functional form is given by:

$$y_i \approx \langle w, x_i \rangle, \; i = 1, \ldots, n.$$

where $w \in \mathbb{R}^d$ is a vector containing the *regression coefficients*. We need to figure out w from the data. Linear models are simple, powerful, and widely used.

Solving linear regression

The typical way to solve regression problems is to first define a suitable *loss function* with respect to the model parameters, and then discover the model that minimizes the loss. A model with zero loss can perfectly predict the training data; a model with high loss is considered to be poor.

The most common loss function is the *least-squares* loss:

$$L(f) = \frac{1}{2} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

For linear models, this reduces to $L(w) = \frac{1}{2}(y - \langle w, x \rangle^2$. For conciseness, we write this as:

$$L(w) = \frac{1}{2} \|y - Xw\|^2$$

the norm above denotes the Euclidean norm, $y = (y_1, \ldots, y_n)^T$ is an $n \times 1$ vector containing the y's and $X = (x_1^T; \ldots; x_n^T)$ is an $n \times d$ matrix containing the x's, sometimes called the "design matrix".

The gradient of $L(W)$ is given by:

$$\nabla L(w) = -X^T(y - Xw).$$

The above function $L(w)$ is a convex (in fact, quadratic) function of w. The value of w that minimizes this (say, w^*) can be obtained by setting the gradient of $L(w)$ to zero and solving for w:

$$\begin{aligned} \nabla L(w) &= 0, \\ -X^T(y - Xw) &= 0, \\ X^T X w &= X^T y, \text{ or} \\ w &= (X^T X)^{-1} X^T y. \end{aligned}$$

The above represents a set of d linear equations in d variables, and are called the *normal equations*. If $X^T X$ is invertible (i.e., it is full-rank) then the solution to this set of equations is given by:

$$w^* = \left(X^T X\right)^{-1} X^T y.$$

If $n \geq d$ then one can generally (but not always) expect it to be full rank; if $n < d$, this is not the case and the problem is under-determined.

Computing $X^T X$ takes $O(dn^2)$ time, and inverting it takes $O(d^3)$ time. So, in the worst case (assuming $n > d$), we have a running time of $O(nd^2)$, which can be problematic for large n and d.

Gradient descent

We discuss a different way to solve for the optimal model without inverting a system of linear equations using *gradient descent*. This is a very useful primitive in all types of prediction problems beyond linear regression.

If we want to minimize any function, a canonical way to do it is to start at a given estimate, and iteratively move to new points with lower functional value.

Similarly, the idea in gradient descent is to start at some estimate w_k, and iteratively adjust it by moving in the direction *opposite* to the gradient of the loss function at w_k, i.e.,

$$w_{k+1} = w_k - \alpha_k \nabla L(w_k).$$

The parameter α_k is called the step size, and controls how far we descend along the gradient. (The step size can be either constant, or vary across different iterations). If we have reached the optimal value of w then $\nabla F(w) = 0$ and there will be no further progress.

For the least-squares loss function, we have:

$$\begin{aligned} w_{k+1} &= w_k + \alpha_k X^T(y - Xw_k) \\ &= w_k + \alpha_k \sum_{i=1}^{n} (y_k - \langle w_k, x_i \rangle) x_i. \end{aligned}$$

It is known that gradient descent will converge eventually to a (local) minimizer of L; moreover, if L is nicely behaved (e.g. if it is convex), then it will in fact converge to a global minimizer. For the least-squares loss, this is indeed the case.

Observe that the per-iteration computational cost is $O(nd)$, which can be several orders of magnitude lower than $O(nd^2)$ for large datasets. However, to precisely analyze running time, we also need to get an estimate on the total number of iterations that yields a good solution.

There is also the issue of step-size: what is the "right" value of a_k? And how do we choose it?

Analysis

Some facts from linear algebra (stated without proof):

- A $n \times n$ matrix A is said to have an *eigenvalue* λ (in general, a complex number) if there is a vector $v \in \mathbb{R}^n$ such that $Av = \lambda v$.
- A $n \times n$ symmetric matrix A has n *real-valued* eigenvalues. They can be either positive or negative, and lie in some interval of the real line (l, L).
- If A is of the form $X^T X$, then all eigenvalues of A are non-negative and A is said to be *positive semi-definite* (psd).
- The eigenvalue of A that has maximum absolute value (say L) satisfies the relation:

  $$\|Ax\|_2 \leq |L|\|x\|$$

  for all $x \in \mathbb{R}^n$.
- If l and L are the minimum and maximum eigenvalues of a symmetric matrix A, then the eigenvalues of the matrix $I - \alpha A$ lie between $1 - \alpha L$ and $1 - \alpha l$.
- As a consequence of the above two facts, for any matrix A and any vector x, we have:

  $$\|(I - \alpha A)x\|_2 \leq \max(|1 - \alpha l|, |1 - \alpha L|)\|x\|_2$$

  Let us call this **Fact 1**.

We now can analyze gradient descent for the least squares loss function. From the definition of gradient, we know that for any two estimates w_1 and w_2:

$$\nabla L(w_1) - \nabla L(w_2) = X^T X(w_1 - w_2).$$

Let all norms below denote the Euclidean norm. Suppose that the optimal w is denoted as w^*; by definition, the gradient vanishes here and $\nabla L(w^*) = 0$. Suppose l and L are the minimum and maximum eigenvalues of $X^T X$. Since $X^T X$ is psd, they both are non-negative.

Then, consider the estimation error at iteration $k+1$:

$$\begin{aligned}
\|w_{k+1} - w^*\| &= \|w_k - \alpha_k \nabla L(w_k) - w^*\| \\
&= \|w_k - w^* - \alpha_k L(w_k)\| \\
&= \|w_k - w^* - \alpha_k (L(w_k) - L(w^*))\| \quad \text{(definition of } w^*\text{)} \\
&= \|w_k - w^* - \alpha_k (X^T X (w_k - w^*))\| \\
&= \|(I - \alpha_k X^T X)(w_k - w^*)\| \quad \text{(Fact 1)} \\
\|w_{k+1} - w^*\| &\leq \max(|1 - \alpha_k l|, |1 - \alpha_k L|) \|w_k - w^*\|.
\end{aligned}$$

The above inequality is nice, since it tells us that the error at iteration $k+1$ is at most ρ times the error at iteration k, where:

$$\rho = \max(|1 - \alpha_k l|, |1 - \alpha_k L|).$$

If $\rho < 1$, the estimate will converge *exponentially* to the optimum. The smaller ρ is, the faster the rate of convergence. We now choose α_k to make the above quantity to be as small as possible, and simple calculus gives us the value to be:

$$\alpha_k = \frac{2}{L + l}$$

and the convergence property becomes:

$$\|w_{k+1} - w^*\| \leq \frac{L - l}{L + l} \|w_k - w^*\|.$$

By induction on k, we can conclude that:

$$\|w_{k+1} - w^*\| \leq \left(\frac{L - l}{L + l}\right)^k \|w_1 - w^*\| = \rho^k \|w_1 - w^*\|.$$

We can initialize with $w_1 = 0$ (or really, anything else) – either way, the error decreases exponentially in k. In other words, only a *logarithmic* number of iterations are required to make the estimation error smaller than some desired target.

So, takeaway points:

1. Gradient descent converges very quickly to the right answer
2. provided the step size is chosen correctly; more precisely
3. the "right" step size is $2/(L + l)$ where L and l are the biggest and smallest eigenvalues of the design matrix $X^T X$.
4. In practice, one just chooses the step size by hand. In general it shouldn't matter very much as long as it isn't too large (and makes ρ bigger than 1); in this case, the answer will diverge.
5. Provided convergence occurs, the number of iterations required to push the estimation error below some desired parameter ε is given by:

   $$T = \log_{1/\rho}\left(\frac{\|w^*\|_2}{\varepsilon}\right).$$

   which can be much smaller than either n or d, depending on how small we set ε.

Chapter 11

Regression (contd)

We will discuss two extensions of gradient descent.

Stochastic gradient descent

One issue with gradient descent is the uncomfortable fact that one needs to compute the gradient. Recall that the gradient descent iteration for the least squares loss is given by:

$$w_{k+1} = w_k + \alpha_k \sum_{i=1}^{n} (y_i - \langle w_k, x_i \rangle) x_i$$

So, per iteration:

- One needs to compute the d-dimensional dot products
- by sweeping through each one of the n data points in the training data set.

So the running time is $\Omega(nd)$ at the very least. This is OK for datasets that can fit into memory. However, for extremely large datasets, not all the data is in memory, and even computing the gradient once can be a challenge.

A very popular alternative to gradient descent is *stochastic gradient descent* (SGD for short). This method is picking up in popularity since dataset sizes have exponentially grown in the last few years.

The idea in SGD is simple: instead of computing the full gradient involving all the data points, we *approximate* it using a *random* subset, S, of data points as follows:

$$w_{k+1} = w_k + \alpha'_k \sum_{i \in S} (y_i - \langle w_k, x_i \rangle) x_i.$$

The core idea is that the full gradient can be viewed as a weighted average of the training data points (where the i^{th} weight is given by $y_i - \langle w_k, x_i \rangle$), and therefore one can approximate this average by only considering the average of a *random* subset of the data points.

The interesting part of SGD is that one can take this idea to the extreme, and use a *single* random data point to approximate the whole gradient! This is obviously a very coarse, erroneous approximation of the gradient, but provided we sweep through the data enough number of times the errors will cancel themselves out and eventually we will arrive at the right answer.

Here is the full SGD algorithm.

Input: Training samples $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$.

Output: A vector w such that $y_i \approx \langle w, x_i \rangle$ for all $(x_i, y_i) \in S$.

0. Initialize $w_0 = 0$.
1. Repeat:
   a. Choose $i \in [1, n]$ uniformly at random, and select (x_i, y_i).
   b. Update:
   $$w_{k+1} \leftarrow w_k + \alpha_k(y_i - \langle w_k, x_i \rangle)x_i$$
   and increment k.

While epoch $\leq 1, .., maxepochs$.

The algorithm looks similar (structurally) to that of the perceptron. That is because the perceptron is essentially an SGD algorithm! (In the textbook, there is more details about precisely what loss function the perceptron is trying to minimize.)

We won't analyze SGD (bit messy) but will note that the step-size cannot be constant across all iterations (as in full GD). A good choice of decaying step size is the hyperbolic function:

$$\alpha_k = C/k.$$

One can see that the per-iteration cost of SGD is only $O(d)$ (assuming that the random selection can be done in unit-time). However, there is a commensurate increase in the number of iterations; suppose that after T iterations, the error drops to ε:

$$\|w_T - w^*\| \leq \varepsilon.$$

Then an upper bound on T is given by $O(1/\varepsilon)$.

In comparison with GD (running time of $O(nd \log(1/\varepsilon))$, SGD seems quite a bit faster $O(d/\varepsilon)$; however, if we want ϵ to be very close to zero (i.e., we seek the best possible model for the data) then the opposite is true.

Of course, keep in mind that obtaining the best possible model fit may not be the best thing to do, since the model could be over-fitted to the training data. So there are both computational as well as statistical reasons why you might want to choose SGD for regression problems.

SGD is very popular, and lots of variants have been proposed. The so-called "mini-batch" SGD trades off between the coarseness/speed of the gradient update in SGD versus the accuracy/slowness of the gradient update in full GD by taking small batches of training samples and computing the gradient on these samples. Other variants include Stochastic Average Gradients (SAG), Stochastic Mirror Descent (SMD), Stochastic Variance Reduced Gradients (SVRG) etc.

Logistic regression

We will now use regression algorithms to solve *classification* problems. Classification can be viewed as a special case of regression where the predicted values are binary class labels – $\{\pm 1\}$ (or $\{0, 1\}$). Indeed, one can solve classification in this manner by simply fitting linear (or nonlinear) regression models to the data points, and rounding off the predicted value to the closest label.

However, this is not conceptually very satisfying. Suppose we develop a linear model w (using regression) for a classification problem, and for a new data point x, we obtain a predicted value of $y = \langle w, x \rangle = -18950$. Do we round this to -1 or 1? (They are both basically the same distance away from y.)

The way to resolve this issue is as follows. Instead of finding a model f such that:

$$y_i = f(x_i),$$

we will instead find a model f such that:

$$P(y_i = 1 | x_i) = f(x_i).$$

Here, $P(y_i = 1|x_i)$ is the *probability* that the label of y_i is 1 given that the data point is x.

In a binary classification problem, the probability that the label of y_i is 0 is given by:

$$P(y_i = 0 | x_i) = 1 - f(x_i).$$

The two equations can be combined into one by using the general expression:

$$P(y_i | x_i) = (f(x_i))^{y_i} (1 - f(x_i))^{1-y_i}.$$

The above expression is also called the *likelihood* of y_i given x_i.

Given n independent data points, the likelihoods will multiply. Therefore, the likelihood of a set of n labels, $y \in \{\pm 1\}^n$, given a set of data points $x_1, x_2, \ldots, x_n$ is given by:

$$P(y|X) = \prod_{i=1}^{n} (f(x_i))^{y_i} (1 - f(x_i))^{1-y_i}.$$

Instead of products, it is easier to calculate sums – so we take logarithms on both sides. The (negative) log-likelihood is given by:

$$L(f) = -\sum_{i=1}^{n} y_i \log f(x_i) + (1 - y_i) \log(1 - f(x_i)).$$

The goal is to find the model f that minimizes the negative log likelihood (NLL) of the observed data.

A popular choice for f is the *logistic* function, where the probability function f is chosen as:

$$f(x) = \frac{1}{1 + e^{-\langle w, x \rangle}}$$

The function $\sigma(z) = 1/(1+e^{-z})$ is an S-shaped function called the *sigmoid*, and the logistic function is its multidimensional extension. Plugging this into the NLL function, we get:

$$L(w) = \sum_{i=1}^{n} l_i(w),$$

where

$$l_i(w) = -\left(y_i \log \frac{1}{1+e^{-\langle w, x_i \rangle}} + (1-y_i) \log \frac{e^{-\langle w, x_i \rangle}}{1+e^{-\langle w, x_i \rangle}} \right).$$

Solving logistic regression using GD

Despite its non-intuitive structure, the logistic function was carefully chosen; the sigmoid function satisfies the identity:

$$1 - \sigma(z) = \sigma(-z).$$

Moreover, the derivative of $\sigma(z)$ is:

$$\sigma(z)(1-\sigma(z)).$$

Using the two facts (and some calculus), one can derive the gradient update rule as:

$$w_{k+1} = w_k + \alpha \sum_{i=1}^{n} (y_i - \frac{1}{1+e^{-\langle w, x_i \rangle}}) x_i,$$

which, again, is remarkably similar to the update rules for linear regression as well as perceptron.

Chapter 12

Singular Value Decomposition

We will develop a third (and better) way to solve linear regression problems using the *Singular Value Decomposition*. This is a very powerful numerical method that is useful for all sorts of data analysis problems, including data visualization and compression (which we will discuss next class).

Arrange the data points as the $n \times d$ matrix $X = [x_1^T; x_2^T; \ldots x_n^T]$. Let r be the rank of this matrix. In general, r can be as large as $\min(n, d)$.

Then, one can show that X can be written as:

$$\begin{aligned} X &= \sum_{i=1}^{r} \sigma_i u_i v_i^T \\ &= U \Sigma V^T, \end{aligned}$$

where $U = [u_1, u_2, \ldots, u_r]$ is an $n \times r$ matrix, $\Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r)$ is a *diagonal* $r \times r$ matrix, and $V = [v_1, v_2, \ldots, v_r]$ is a $d \times r$ matrix.

Moreover, the factors U, Σ, V possess the following properties:

1. U and V are *orthonormal*. The columns of U, $u_i \in \mathbb{R}^n$ are of unit norm and are pairwise orthogonal: $\|u_i\|_2 = 1$ and $\langle u_i, u_j \rangle = 0$ for $i \neq j$. The vectors u_i are called the *left singular vectors* of X.
2. Similarly, the columns of V, $v_i \in \mathbb{R}^d$ (alternately, the rows of V^T) also are unit norm and pairwise orthogonal: $\|v_i\|_2 = 1$ and $\langle v_i, v_j \rangle = 0$ for $i \neq j$, The vectors v_i are called the *right singular vectors* of X.

The above conditions imply that $U^T U = V^T V = I_{r \times r}$, where $I_{r \times r}$ is the identity matrix with r rows and columns.

4. Σ is diagonal, and the diagonal elements are all *positive*. These are called the *singular values* of X. It is typical to arrange the singular values in decreasing order:

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0.$$

Any matrix X can be factorized according to the above specifications. This is called the singular value decomposition (or SVD) of X.

The above definition is sometimes called the "skinny SVD". Other times, (assuming $n > d$), one fills out additional singular values $\sigma_r, \sigma_{r+1}, \ldots, \sigma_n$ that are equal to zero, and additional corresponding right and left singular vectors, so that the expression for X becomes $X = \sum_{i=1}^{n} \sigma_i u_i v_i^T$. This is called the "full" SVD. But it should be clear that there is no effect of the extra zero terms. There are several other equivalent definitions.

It is interesting (although not too hard to show using linear algebra) that *any* real-valued matrix can be decomposed into 3 factors having the exact structure as described above. It is even more interesting that there exist efficient *algorithms* to find these factors. But we will hold off on algorithms for a bit and discuss why the SVD is important.

Properties of the SVD

Two key properties of the SVD are as follows:

1. If $A = U\Sigma V^T$ is a square matrix ($n = d$) and full-rank, then its inverse is given by: $A^{-1} = V\Sigma^{-1}U^T$.
2. (Best rank-k approximation.) Given a matrix $X = U\Sigma V^T$, the *best* rank-k approximation of X:
$$X_k = \arg\min_B \|B - X\|_F^2, \;\; \text{rank}(B) = k,$$
i.e, the rank-k matrix X_k that is closest to X in terms of the Frobenius norm is given by:
$$X_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T.$$
3. The i^{th} right singular vector v_i is the i^{th} eigenvector of $X^T X$; precisely, one can show that $X^T X v_i = \sigma^2 v_i$.

Again, the above facts are not too hard to prove (see Chap 3 of the textbook).

The second property offers a natural mechanism to *compress* a set of data points. Instead of storing the data in terms of an $n \times d$ matrix (nd numbers total), one can instead simply store the vectors $[u_1, \ldots, u_k]$, $v_1, \ldots, v_k$ and σ_1 ($nk + dk + k$ numbers total). For $k \ll n, d$ this can be significant; moreover, one is guaranteed that the *reconstructed* data X_k using these vectors is the best possible representation in terms of the Frobenius norm.

Here, we are leveraging the fact that the singular values are arranged in decreasing order. The first singular value (and corresponding right and left singular vectors) u_1, σ_1, v_1 are the most important components of the data; followed by u_2, σ_2, v_2; and so on. Including more components into the expression for X_k gets closer to the data, and therefore the parameter k controls the tradeoff between the size of the representation versus reconstruction error.

Solving linear regression

The first property gives us a closed-form expression for solving linear regression.

Recall that that the expression for the optimal linear regressor (assuming that $n > d$ and $X^T X$ is full-rank) is given by:

$$w_{\text{opt}} = (X^T X)^{-1} X^T y.$$

However, $X^T X = V \Sigma^T U^T U \Sigma V^T = V \Sigma^2 V^T$, and $X^T = V \Sigma U^T$. Simplifying, we get:

$$w_{\text{opt}} = V \Sigma^{-1} U^T y.$$

Why would this for be beneficial over explicitly computing the inverse of $X^T X$ and multiplying? Primarily, the inverse could be poorly conditioned, whereas computing the SVD is (usually, depending on the method used) numerically stable.

The power method

There are various algorithms of computing the SVD, all of which take cubic ($\min(O(n^d, nd^2))$ running time. This is cubic since it is a degree-3 polynomial in the parameters n and d. Not too efficient if n and d are large.

However, for several applications we will only be interested in obtaining the first few singular vectors and singular values of the matrix. For such situations, a quick (and algorithmically efficient) way to *approximate* the SVD is via the third property listed above. The first right singular vector v_1 satisfies the invariant:

$$X^T T v_1 = \alpha v_1.$$

Therefore, repeatedly multiplying v_1 by $X^T X$ shouldn't change the value. This intuition is leveraged in the so-called *Power method* for computing the SVD.

Input: Data $X = \{x_1, \ldots, x_n\}$

Output: Top singular vectors and values (u_1, v_1, σ_1)

0. Initialize $z_0 = 0$, $t = 0$
1. Repeat until convergence:
   a.
   $$z_{t+1} = X^T X z$$
   b.
   $$v_1 = z_{t+1} / \|z_{t+1}\|_2.$$
   c.
   $$\sigma_1 = \|X v_{t+1}\|, u_1 = X v_1 / \sigma_1$$
   d. Increment t.

The running time of the above algorithm can be analyzed as follows:

- per-iteration, the main cost is multiplying with the matrices X and X^T; this can be done in (nd) time. For *sparse* data matrices, i.e., matrices with lots of zeros in them, this can further be sped up. This is one of the main reasons the power method is so widely used.

- the total number of iterations can be upper bounded by $\log(nd)\frac{\sigma_1}{\sigma_1-\sigma_2}$, i.e., inversely proportional to the *gap* between the first and second singular values. Provided this gap isn't too large, the power-method

The power method described above returns the top singular vectors and value. Likewise, the top-k singular vectors and values can be obtained using a similar technique known as the block power method. Several other efficient SVD algorithms (with various properties) are also known, and fast SVD methods are an active ongoing area of research in the algorithms community.

Chapter 13

Nonnegative matrix factorization (NMF)

PCA is well-established and some version of PCA has been around for close to a century now. However, there are several other techniques for exploratory data analysis with all kinds of acronyms - CCA, LDA, NMF, DL, etc - which we won't cover in great detail here. But let us give one example.

One drawback of PCA is that the principal directions and/or scores are not particularly interpretable. Imagine, for example, computing the principal components of a collection of *images*. Image data are typically positive-valued data since they represent light intensity values. If we interpret the principal directions, i.e., the right singular vectors of the data matrix:

$$X = \sum_i \sigma u_i v_i^T$$

as "building blocks" of the data, then each data point (image) can be viewed as a synthesis of some combination of these principal directions.

However, singular vectors are orthogonal by definition, and therefore all their coordinates cannot be all nonnegative. In other words, we are trying to represent positive-valued data (images) in terms of building blocks that are not necessarily image-like.

A different type of decomposition

Let us focus on positive-valued data. Instead of the SVD, we will attempt to factorize any given data matrix as:

$$X = WH$$

where W is a $n \times r$ *coefficient* matrix and H is an $r \times d$ *basis* matrix. The restriction that we will impose is that H (and consequently, W) are both positive-valued. If this is possible, then such a factorization is called nonnegative matrix factorization (NMF).

(Aside: There are several types of matrix factorization techniques – beyond SVD/eigenvalue decomposition and NMF – that impose various types of assumptions on the factors.)

In general, computing the NMF of a matrix is hard (unlike computing the SVD, which supports efficient algorithms). In fact, polynomial-time algorithms are known only when the data matrix satisfies certain restrictive conditions, and the majority of NMF methods are heuristic. However, the methods work fairly well, and the net advantage is that the basis elements (rows of H) are (typically) more interpretable.

We describe a heuristic method to perform NMF. Given a data matrix X and a rank parameter r, we will (try to) solve the optimization problem:

$$\begin{aligned} &\min_{W,H} \|X - WH\|^2 \\ &\text{s.t. } W \geq 0,\ H \geq 0. \end{aligned}$$

where $W \in \mathbb{R}^{n \times r}$ and $H \in \mathbb{R}^{r \times d}$.

The positivity constraints on W and H are *convex*. Convex constraints are typically easy to handle. The difficulty lies in the objective function: it is *bilinear* in the variables W and H, and therefore is non-convex overall.
In order to (try to) solve the problem, we will adopt a technique known as *alternating minimization*.

The basic idea is that conditioned on one of the variables, the optimization problem becomes convex in the other variable, and can be solved using techniques such as gradient descent. For example, at the k^{th} estimate, if we fix $W = W_k$, then the problem of estimating H reduces to:

$$\begin{aligned} &\min_{H} \|X - W_k H\|^2 \\ &\text{s.t. } H \geq 0. \end{aligned}$$

This subproblem can be solved using *projected gradient descent*. Repeat the following update step until convergence:

$$H \leftarrow P_+(H - \eta W_k^T (X - W_k H)).$$

In words, the algorithm proceeds by descending along the the gradient of the objective function, followed by projecting onto the set of positive matrices (i.e., set all the negative entries of the matrix to 0.) Eventually, we get an estimate of H which we will call H_{k+1}.

Now, if we fix $H = H_{k+1}$, the problem of estimating W reduces to:

$$\begin{aligned} &\min_{W} \|X - WH_k\|^2 \\ &\text{s.t. } W \geq 0. \end{aligned}$$

To solve this subproblem, repeat until convergence:

$$W \leftarrow P_+(W - \eta'(X - WH_k)H^T).$$

Eventually, we will converge to an estimate that we denote as W_{k+1}. We use this to obtain H_{k+2} and so on. Alternate between the two subproblems sufficiently many times.

There is no guarantee that this will produce the global optimum of the bilinear optimization problem (and it is often the case that it will get stuck at a local optimum.)

The per-iteration cost of each subproblem in the above algorithm is dominated by the process of multiplying an $n \times d$ matrix with a $d \times r$ or with an $r \times n$ matrix, which incurs a running time of $O(ndr)$. However, analyzing the number of iterations is incredibly challenging.

Connections with clustering

There are some subtle connections between NMF and *clustering* algorithms. Suppose that we try to solve the above problem, but *also* constrain the rows of W to be indicator vectors (i.e., zeros everywhere except a single nonzero entry that is equal to 1.) Then, the product WH is nothing but the basis vectors in H listed in some order with possible duplicates, and the objective function:

$$\|X - WH\|^2$$

measures the (cumulative) error between the data points and the corresponding basis vectors.

For this special situation, finding the NMF solution is the same as finding a set of r *representative* vectors in the data space (H), and the corresponding mapping between each data point to one of these representatives (W). We call this special problem k*-means clustering*, and will discuss it in detail later.

Applications

NMF has been used in several applications, including:

- Face images: Given a collection of centered face images, NMF gives a "parts"-based decomposition of each image, where the basis vectors (semantically) correspond to different portions of a human face.
- Topic models: Given a dataset of n documents (represented by d-dimensional TF-IDF features), NMF produces a *topic* model for each document. The idea is that if we decompose the document matrix X as:

  $$X = WH$$

  then each document is a linear combination of a small number of *topics*, represented in TF-IDF format as the rows of H. This is a natural interpretation, since any given document only consists of a small number of semantically meaningful "topics". A short demo of this idea is given as a Python example at: https://medium.com/towards-data-science/improving-the-interpretation-of-topic-models.

Chapter 14

Clustering

We will move onto a different type of unsupervised learning problem known as *clustering*.

The high-level goal in clustering is as follows: given a set of *unlabeled* data points belonging to two or more classes, can we automatically figure out the class labels? Imagine, for example, an unordered set of senator voting records, and automatically trying to figure out the party identities from their voting records alone.

Clustering algorithms are numerous and widely studied. We will study a couple of representative and popular methods.

k-means

Below, $\|\cdot\|$ denotes the Euclidean norm.

Suppose we are given a set of data points $X = \{x_1, x_2, \ldots, x_n\}$. We are also given an integer parameter $k > 1$. Our aim is to produce a partition of the data, i.e., disjoint subsets (called *clusters*) $S_1, \ldots, S_k$ such that

$$X = S_1 \bigcup S_2 \bigcup \ldots S_k$$

as well as a set of k *cluster centers* $\mu_1, \ldots, \mu_k \subset \mathbb{R}^d$ such that the following objective function is minimized:

$$F(\{S_1, \ldots, S_k\}, \{\mu_1, \ldots, \mu_k\}) = \sum_{j=1}^{k} \sum_{x_i \in S_j} \|x_i - \mu_j\|^2.$$

The above objective function minimizes the sum of (squared) distances from each data point to its nearest cluster center, and is sometimes called the *k-means objective function*.

Notice that this is an optimization problem involving both discrete and continuous variables. In general, solving such "mixed-integer" optimization problems is very challenging and there are no black-box solution approaches (typical approaches such as gradient descent or convex relaxation are difficult to define.) Let us see if we can somehow overcome this issue.

Warmup: When the clusters are known

First, suppose that an oracle provided us the "right" cluster labels, i.e., we know the true $S_1, \ldots, S_k$, and need to only optimize over the cluster centers. However, observe that the objective function decouples into k separate terms, each term of the form:

$$\sum_{x_i \in S_j} \|x_i - \mu_j\|^2$$

that only involves the optimization variable μ_j. This problem admits a closed form solution, since some elementary algebra shows that the above term is equal to:

$$\left(\sum_{x_i \in S_j} \|x_i - \mu\|^2 \right) + |S_j| \|\mu - \mu_j\|^2$$

where μ is the mean of the data points within the cluster. Clearly the above term is minimized when $\mu_j = \mu$. Therefore, the optimal cluster center μ_j is given by:

$$\mu_j = \mu = \frac{1}{|S_j|} \sum_{x_i \in S_j} x_i.$$

Of course, this method works only if we somehow got the "right" clusters (which is the whole point of clustering in the first place.) In general, there are k^n possibilities of choosing the cluster labels, so cycling through all possibilities would take exponential running time.

Lloyd's algorithm

So instead of guessing $S_1, \ldots, S_k$, we will instead *alternate* between estimating the cluster labels and the cluster centers.

Given the (estimate of) labels, we know how to estimate the centers (as described above.) Given the (estimate of) the centers, we can estimate the cluster labels by simply mapping each data point to the *nearest* cluster center in the data space. This can be done using the nearest neighbor algorithms that we discussed in the early lectures.

This type of *alternating* procedure is called *Lloyd's algorithm*, which was originally proposed for a similar problem in data compression known as *vector quantization*. The full algorithm is as follows.

0. Initialize cluster centers $\{\mu_1, \mu_2, \ldots \mu_k\} \subset \mathbb{R}^d$.
1. For each i, assign x_i to the nearest cluster center:

   $$j^* = \arg\min_{j \in [k]} \|x_i - \mu_j\|.$$

   This induces a disjoint partition of the data into subsets $S_1, \ldots, S_k$.
2. For each subset S_j, update the cluster centers:

   $$\mu_j = \frac{1}{|S_j|} \sum_{x_i \in S_j} x_i.$$

3. Go to Step 1; repeat until there is no decrease in clustering objective function F.

Initialization

There are both minor and major issues in the above description of k-means. One minor issue is that clusters might become empty if a given center does not get assigned any data point. But this can be handled by arbitrarily initializing new cluster centers.

The major issue in k-means is initialization. Indeed, it is known that k-means can get stuck at arbitrarily bad local optima. Imagine, for example, a very simple data set with $n = 4$ points in $d = 2$ dimensions:

$$X = \{(-x, 0.5), (-x, -0.5), (x, 0.5), (x, -0.5)\}$$

where x is some very large positive number. Clearly, the "right" clusters here are:

$$S_1 = \{(-x, 0.5), (-x, -0.5)\},\ S_2 = \{(x, 0.5), (x, -0.5)\}.$$

with optimal cluster centers $(-x, 0)$ and $(x, 0)$.

However, suppose we are careless and initialize $\mu_1 = \{(0, 0.5)\}$ and $\mu_2 = (0, -0.5)$. Then, Step 1 gives us cluster labels $1, 2, 1, 2$ for the four data points respectively, and Step 2 gives us back the μ_i's that we started with. So the algorithm terminates in 1 step, but has clearly converged to the wrong answer.

To resolve this, several "smart" initialization strategies for k-means have been proposed. A commonly used initialization procedure is the k-means++ method by Arthur and Vassilvitskii (2007), who propose picking k initial points that are far away from each other. More precisely, the algorithm does the following:

0. Pick μ_1 as a data point $x \in X$ chosen uniformly at random, set $T = \mu_1$.
1. For $i = 2, \ldots, k$, pick μ_i as $x \in X$ chosen randomly with probability proportional to $\min_{\mu_j \in T} \|x - \mu_j\|$, update $T \leftarrow T \bigcup \mu_i$.

Interestingly, this type of initialization comes with provable quality guarantees; one can show that just the initialization itself is a pretty good way to cluster the data (without running subsequent iterations of k-means), and further updates with Lloyd's algorithm can only improve the quality. But we won't go into further details here.

Chapter 15

Clustering (contd)

The k-means algorithm is useful for clustering data that is *separable*, i.e., the clusters are compact and belong to well-separated parts of the data space.

On the other hand, data clusters can often be non-separable. Imagine, for example, data points that lie on two concentric circles. Here, the "natural" clusters are the two clusters, but k-means will not give any useful information.

Here, we discuss a new algorithm for clustering non-separable data which can resolve this issue. We call it *spectral clustering*, and it is one among several data analysis techniques that leverages ideas from algorithmic *graph* theory.

Graphs, etc.

First, some basics. Recall that a graph $G = (V, E)$ consists of a set of *nodes* V, and edges between nodes E. The nodes can represent any collection of objects, and edges can represent relations defined on these objects. Typically, we denote the number of nodes as n, and the number of edges as m.

e.g. Nodes can be people in a social network, and edges can represent friendships. Alternately, nodes can be spatial locations in a road network, and edges can represent road segments. Alternately, nodes can be IP addresses, and edges can represent communication links. You get the picture.

Depending on whether the edges have *arrows* (directions) or not, the graph can be directed or undirected. (If the relation is symmetric, the graph is undirected; and if not, it is directed.)

Optionally, edges can have *weights*; stronger relations can be represented using heavier weights, and vice versa.

Every graph with n nodes can be written down using an $n \times n$ matrix known as an *adjacency matrix* A. Suppose the graph G is unweighted. Then, the adjacency matrix A is such that $A_{ij} = 1$ if there is an edge from node i to node j, and $A_{ij} = 0$ otherwise. If G is weighted, then A_{ij} can be a nonnegative real number that represents the edge weight between i to j.

The *degree* of a node in an undirected graph is the number of edges connected to that node. If we store all the degrees in a length-n vector of integers $\deg(V)$, then it is not hard to see that:

$$\deg(V) = A1_n,$$

where 1_n is the all-ones vector of size $n \times 1$. The exact same definition of $$ can be used for a weighted graph; in this case, $\deg(V)$ is a vector of real numbers.

(If a graph is directed, then we can define the *in-degree* of a node as the number of incoming edges, and the *out-degree* as the number of outcoming edges. But we will defer these notions for later.)

The *graph Laplacian* of G is defined as the $n \times n$ matrix:

$$L = D - A,$$

where $D = \text{diag}(\deg(V))$.

The graph Laplacian has several interesting properties. Observe that for *any* graph:

$$\begin{aligned} L1_n &= D1_n - A1_n \\ &= \deg(V) - A1_n \\ &= 0. \end{aligned}$$

In words, the Laplacian matrix of any graph always has an eigenvalue equal to zero, with the corresponding eigenvector being the all-ones vector.

In fact, it is not very hard to prove that any graph Laplacian only has *non-negative* eigenvalues, i.e., it is positive semidefinite. Consider any vector $v \in \mathbb{R}^n$. Let $\text{nbr}(i)$ denote the set of neighbors of node i. Then, the vector Lv is such that:

$$\begin{aligned} (Lv)_i &= \sum_j L(i,j)v_j \\ &= \deg(v_i)v_i - \sum_{j \in \text{nbr}(j)} v_j \\ &= \sum_{j \in \text{nbr}(i)} (v_i - v_j). \end{aligned}$$

Therefore, the quantity $v^T Lv$ is given by:

$$\begin{aligned} v^T Lv &= \sum_{i=1}^n v_i \left(\sum_{j \in \text{nbr}(i)} (v_i - v_j) \right) \\ &= \sum_i \sum_{j \in \text{nbr}(i)} v_i^2 - v_i v_j \\ &= \frac{1}{2} \sum_i \sum_{j \in \text{nbr}(i)} 2v_i^2 - 2v_i v_j \\ &= \frac{1}{2} \sum_{(i,j) \in E} (v_i - v_j)^2 \\ &\geq 0. \end{aligned}$$

The above derivation is true no matter how we choose v. Suppose we set v to be any eigenvector of L; then the above derivation shows that the corresponding eigenvalue is non-negative. In general, the set of eigenvalues of the Laplacian are also known as the *spectrum* of the graph.

The above derivation also shows another interesting fact. Suppose the graph contained two disconnected components (pieces), and v is any vector that is *piecewise constant*, e.g., v_i is $+1$ if i belongs to the first component, and -1 if i belongs to the second component. In that case, $v^T Lv = 0$ as well. Therefore, the graph will have the two smallest eigenvalues as zero, with the first zero corresponding to the all-ones vector, and the second zero being the above piecewise constant vector.

In general, the number of zero eigenvalues denotes the number of connected components in the graph.

Spectral clustering

OK, how does all this relate to clustering? Suppose we are given a dataset X containing data points from two clusters, but we don't know the cluster memberships of the points and wish to deduce them automatically.

The high level idea is to denote *data points* as nodes in a hypothetical graph G. The goal is to construct a graph G with a suitable set of edges such that (i, j) is an edge only if i and j belongs to the same cluster. Of course, since we do not know the clusters *a priori*, we do not have access to the true edges!

However, we can *guess* that nearby data points can belong to the same cluster. Nearness can be captured via any similarity measure, e.g., using a user-defined kernel matrix. If we were lucky and the graph only contained intra-cluster edges, there would be exactly two pieces to the graph and the second eigenvector would be piecewise constant across these two pieces.

In reality, there might be extra spurious edges that connect points belonging to different clusters. These could be modeled as "noise"; but as long as the noise is small the clusters can still be deduced.

Overall, the spectral clustering algorithm works as follows. Suppose we have a dataset containing n data samples $\{x_1, \ldots, x_n\}$.

1. Construct a graph G with n nodes.
2. Construct the adjacency matrix of G using weights $W_{ij} = K(x_i, x_j)$ for $i \neq j$.
3. Construct a graph Laplacian $L = D - W$.
4. Construct the eigenvector v corresponding to the *second* smallest eigenvalue of L.
5. Partition the nodes into S_1 and S_2 such that $S_1 = \{v : v > 0\}$ and $S_2 = \{v : v < 0\}$.

The above algorithm involves computing the kernel weights (which has a computational cost of $O(n^2 d)$) and performing the eigenvalue decomposition (which has a computational cost of $O(n^3)$). Therefore, the overall running time scales as:

$$\max(O(n^2 d), O(n^3)).$$

Chapter 16

Clustering (contd)

Both the k-means and spectral clustering algorithms assume that we *know* the number of clusters in the data. However, in several exploratory data analysis problems, this is not known beforehand.

Indeed, clusters can manifest themselves at multiple scales. Imagine, for example, trying to cluster unlabeled genetic data of several flora and fauna, automatically. At a very coarse scale, there are two clusters. However, at the next we can think of clustering the flora/fauna according to genus, species, and so on.

A third type of clustering approach, known as *Hierarchical clustering* resolves this issue in a completely different way. The idea is to greedily form data clusters based on local correlations between the data points. This type of clustering is extensively used in data analysis problems encountered in biology and bioinformatics.

Hierarchical clustering

There are two types of hierarchical clustering methods. Both are conceptually very simple:

1. Top-down clustering: we model *all* data points as belonging to one large cluster, and recursively partition the data to incrementally create additional clusters of smaller sizes, each sub-cluster containing data points that are nearby one another.
2. Bottom-up clustering: we initialize each data point as its own cluster, and recursively merge clusters that are nearby each other to form bigger clusters; this is repeated until all points eventually coalesce into one cluster.

The second type of hierarchical clustering is more common; we will exclusively use this approach (which is also known as *agglomerative clustering*.) Mathematically, if $X = \{x_1, \ldots, x_n\}$, then the algorithm proceeds as follows:

- Initialize the n clusters as singleton sets $C_1 = \{x_1\}, \ldots, C_n = \{x_n\}$.
- Until one cluster remains, repeat:

merge clusters C and C' such that the distance $d(C, C')$ is minimized over all pairs of clusters C, C'.

Cluster distances

In the above algorithm, we haven't defined how we define the notion of distance $d(\cdot, \cdot)$ *between* clusters. There are three ways to do so, and different choices lead to different clustering algorithms:

1. We define the cluster distance as the *minimum* distance between a point belonging to one cluster and a point belonging to the second cluster, i.e.,
$$d(C, C') = \min_{x_i \in C} \min_{x_j \in C'} \|x_i - x_j\|.$$
If this notion of cluster distance is used, then the associated hierarchical clustering algorithm is known as *single-linkage* clustering. Single-linkage tends to produce "chain"-like clusters in the output.

2. We define the cluster distance as the *maximum* distance between a point belonging to one cluster and a point belonging to the second cluster, i.e.,
$$d(C, C') = \max_{x_i \in C} \max_{x_j \in C'} \|x_i - x_j\|.$$
If this notion of cluster distance is used, then the associated hierarchical clustering algorithm is known as *complete-linkage* clustering. Complete-linkage tends to produce compact clusters of roughly equal diameters.

3. We define the cluster distance as the *average* distance between a point belonging to one cluster and a point belonging to the second cluster, i.e.,
$$d(C, C') = \frac{}{|C||C'|} \sum_{x_i \in C} \sum_{x_j \in C'} \|x_i - x_j\|.$$
If this notion of cluster distance is used, then the associated hierarchical clustering algorithm is known as *average-linkage* clustering. In bioinformatics and genetics, this is sometimes called the UPGMA (unweighted pair group method with arithmetic mean) algorithm.

In the above description, we have used the Euclidean norm as a measure of distance between individual data points, but really any distance measure can be used here instead.

Dendrograms

A nice way to visualize hierarchical clustering methods is via a *dendrogram*. A dendrogram is a *binary tree* with n leaves (typically arranged at the bottom), with leaf nodes representing data points (singleton clusters). Each merge operation can be represented by merging nodes into a supernode. There are $n - 1$ merge operations, and therefore the root (denoted at the top of the dendrogram) will eventually comprise all n data points.

The clustering at any intermediate iteration can be immediately deduced by cutting the branches of the tree at a fixed horizontal level; the remaining connected components of the pruned tree indicates the clusters at that given iteration.

Chapter 17

Nonlinear dimensionality reduction

The next few lectures will deal with challenges where the size of the available data (either in terms of data dimension d or number of samples n) is very large.

Dimensionality reduction is an umbrella term used to handle cases where d is large. The idea is to somehow decrease the dimension of the data, while preserving (to the extent possible) the essential information preserved in the data. (Here, the meaning of "essential information" depends on the specific scenario being studied.)

Advantages of dimensionality reduction include:

- Data compression.
- Accelerating downstream data processing tasks (nearest neighbor classification, etc.).
- Data visualization. It is much more convenient to plot/visualize/explore data that is two- or three-dimensional.

For the above reasons, dimensionality reduction methods are often used as pre-processing techniques in large-scale data analysis problems.

Preliminaries

Note that both PCA as well as random projections (e.g., Johnson-Lindenstrauss) can be viewed as dimensionality reduction techniques. However, both are *linear* dimensionality reduction techniques; in random projections, we linearly project each data sample onto random vectors, while in PCA, we linearly each data sample onto the principal directions.

However, linear projections are not appropriate in cases where the data points lies on a *curved* surface in the data space. Imagine, as a toy example, a large set of data points in 3 dimensions ($d = 3$) lying on the surface of a sphere. Suppose we wish to reduce the dimensionality to $d = 2$, which we can do using for, e.g., computing the top two principal components.

However, since the data surface is nonlinear, projecting the data along any given 2D plane in the 3D space irrevocably *distorts* the geometry of the data points. Therefore, new methods are required.

We have already (extensively) discussed one class of methods to deal with nonlinear data sets: *kernel methods*. Fortuitously, most of the analysis methods we have discussed have kernelized variants. For example, we can do a kernel PCA instead of PCA. These usually involve figuring out a way to express PCA computations in terms of inner products of the data points, and replacing these with general (kernel) inner products.

Below, we discuss a few ways to extend this type of idea beyond just kernel methods.

Multidimensional Scaling

The high level idea is as follows. Consider a data matrix $X = [x_1, x_2, \ldots, x_n]^T$. Instead of the data matrix, suppose we only consider the *inner product* matrix of size $n \times n$:

$$P = XX^T.$$

Given only this matrix, what can we do? Observe that since $X = U\Sigma V^T$, we have:

$$P = U\Sigma^2 U^T.$$

So there is no chance of recovering the principal *directions* of X from P alone. However, we can calculate the principal *scores*; this is because the eigenvectors of P are precisely the original left singular vectors, and the eigenvalues are precisely the square of the original singular values.

Kernel PCA would replace the matrix P with a more general matrix K, where

$$K_{ij} = \langle \phi(x_i), \phi(x_j) \rangle,$$

is an arbitrary kernel inner product, and proceed as above.

However, let us now suppose that we do not even compute inner products, but merely have access to pairwise *distances* between data points, i.e., we consider the $n \times n$ matrix of (squared) distances Δ_{ij} such that:

$$\Delta_{ij} = \|x_i - x_j\|^2.$$

Problems like this arise in several instances in visualization. For example, one could have a network of n sensors whose geographical positions are unknown; however, one can measure (or estimate) pairwise distances between sensors via recording received signal strengths (RSS), or some other kind of range measurements. The goal is to visualize the map of these sensors.

What can do with the distance matrix? Observe that:

$$\|x_i - x_j\|^2 = \langle x_i, x_i \rangle + \langle x_j, x_j \rangle - 2\langle x_i, x_j \rangle.$$

Therefore,

$$\Delta_{ij} = P_{ii} + P_{jj} - 2P_{ij}.$$

This is a set of n^2 variables on the left, and n^2 variables on the right. With some algebra, we can solve for P_{ij} to get:

$$P_{ij} = -\frac{1}{2}\left(\Delta_{ij} - \frac{1}{n}\sum_{i=1}^{n}\Delta_{ij} - \frac{1}{n}\sum_{j=1}^{n}\Delta_{ij} + \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\Delta_{ij}\right).$$

In concise form, if I is the identity matrix and $\mathbf{1}$ is the all-ones vector of length n, then:

$$P = -\frac{1}{2}(I - \frac{1}{n}\mathbf{1}\mathbf{1}^T)\Delta(I - \frac{1}{n}\mathbf{1}\mathbf{1}^T).$$

The above operation that maps Δ to P is called *centering*. From the centered matrix of distances, we perform an eigendecomposition to reconstruct the principal scores of the data.

Therefore, the above derivation has shown that we can recover the principal scores of the matrix just from the pairwise distances of the data points alone!

In order to visualize the data points, we can plot the top 2 or 3 principal scores as a scatter plot, just like we would do in PCA.

The above procedure is called *multidimensional scaling* (or MDS). While the above derivation holds for pairwise *Euclidean* distances between data points, one may also perform MDS for non-Euclidean distances, or in general replace the distance matrix by a different matrix that captures a generic form of (dis)similarity between data points.

Chapter 18

Isomap

In MDS, we followed two basic steps:

1. Compute (or obtain in some other way) the matrix of pairwise Euclidean distances between data points.
2. Center this matrix.
3. Perform an eigendecomposition.

Overall, the algorithm takes $O(n^3)$ time for n data points, provided the distance matrix is given.

The above intuition can be extended in various ways. One modification of MDS, called *Isomap*, was the first of several approaches developed for *manifold learning*, that came about in the early 00's.

Manifold models

The high level idea of Isomap is that pairwise Euclidean distances between data points do not always reflect the "true" (intrinsic) distance. Imagine, for example, data points lying on a semicircle in 2D space:

$$x_i = (cos(i\delta), sin(i\delta))$$

for $i = 1, 2, \ldots, n = \pi/\delta$.

The Euclidean distance between x_1 and x_n is 2, but the "true" distance is best measured by traversing along the underlying curve (the semicircle), which in this case equals π. Therefore, there is a distortion of $\pi/2 \approx 160\%$ in our estimate of the "true" distance vs. the estimated distance between the data points.

Observe that this type of distortion only appears when the underlying hypersurface – in our case, the semicircle – from which the data points arise is nonlinear.

Such a hypersurface is called a *manifold*, and can be used to model lots of datasets. In general, if there are k independent degrees of freedom that control the data samples, then the set of data points can be modeled by a k-dimensional manifold. Examples of manifolds include images of a pendulum

swinging from one end to another (one degree of freedom – the angle of the pendulum), or images of an object rotating in space (three angular degrees of freedom.)

Mathematically, we can write:

$$x_i = f(\theta_i), \;\; i = 1, 2, \ldots, n,$$

where we don't immediately have access to θ_i, but would like to measure distances between data points as $\theta_i - \theta_j$.

Manifold learning

In general, how do we compensate for this distortion? The high level idea is that longer distances along a curve can be approximated by adding up a sequence of smaller distances, provided that the curve is smooth enough (which is more or less the same idea as in Riemannian integration.)

The way to fix this issue is via a clever graph-based technique.

1. First, we construct a graph with n nodes (representing data points), edges connecting nearby data points (e.g., all pairs of data points whose Euclidean distance is below some threshold value ε), and edge weights corresponding to the corresponding Euclidean distances. The hypothesis is that the edge distances are sufficiently small that they are roughly equal to the true distances.
2. Next, we compute shortest paths between all pairs of data points in the graph. There are numerous algorithms for computing shortest paths on a graph – Dijkstra's algorithm, Floyd-Warshall, etc – we can use any one of them. This produces a distance matrix Δ, which represents an approximation to the set of "true" parameter distances between data points. The original paper by Tenenbaum, De Silva, and Langford in 2000 precisely analyzes the approximation error vs. number of samples vs. smoothness of the underlying manifold.
3. Finally, we feed this distance matrix into MDS, and compute the top few principal scores (as in PCA) to visualize the data.

Isomap is surprisingly powerful. In particular, it can be effectively used to explore unlabeled/unorganized sets of images that can be modeled as a function of a small number of unknown parameters (e.g., a bunch of aerial images of a region from a unmanned aerial vehicle (UAV); here, each image is a function of the spatial location of the UAV).

Chapter 19

Random walks

In this lecture, we will develop several tools for data analysis based on concepts from graph theory. One powerful class of tools involve *random walks* on graphs. At a high level, a random walk over a graph is a random process that starts at a given vertex, selects a neighbor to visit next, and iterates. Random walks are widely applied in situations ranging from physics (Brownian motion) to stochastic control to web analytics to the text suggestion engine on your smartphone.

The term "random walk" is mainly used in the context of graphs; however, random walks are widely used in several branches of data analysis and statistics that don't necessarily speak the same language. In fact, there is a one-to-one correspondence between terms in different fields:

- Random walks $\leftrightarrow$ Markov chains
- Nodes $\leftrightarrow$ states
- Undirected graphs $\leftrightarrow$ time reversible
- strongly connected $\leftrightarrow$ persistent

etc.

Basics

Consider a graph $G = (V, E)$ consisting of n nodes. A random walk is a process that can be parameterized by two quantities:

1. An initial distribution p_0 over the n nodes. This represents how likely we are to begin at any node in the graph. Since p_0 is a distribution, observe that:

$$p_0(i) \geq 0, \quad \sum_{i=1}^{n} p_0(i) = 1.$$

   i.e., we can write down this distribution as an n-dimensional vector with non-negative entries that sum up to 1.

2. A transition probability matrix M of size $n \times n$. If X_t denotes the random variable recording the state at time t, then
$$M_{ij} = \text{Prob}(X_{t+1} = v_j | X_t = v_i).$$
This value is positive iff there is an edge between i and j; else it is zero. Since the rows of M denote conditional probabilities, we have:
$$\sum_j M_{ij} = 1.$$

By the expression for conditional probability, we can show that the probability of state i at time t, denoted by p_t, satisfies the recursion:

$$p_t = M^T p_{t-1}.$$

A key concept in random walks is the *stationary distribution*, which is defined by:

$$p_* = \lim_{t \to \infty} p_t.$$

Of course, this limit needs to exist for this quantity to be well defined. But if does, then the stationary distribution satisfies the fixed point equation:

$$p_* = M^T p_*,$$

or equivalently,
$$(I - M^T) p_* = 0.$$

One can think of $I - M^T$ as the Laplacian matrix of the (weighted) graph underlying the random walk, and p_* being the eigenvector of the Laplacian matrix whose eigenvalue is zero.

The PageRank algorithm

The reason for Google's early success was the development of the *PageRank* algorithm for web search. It happens to be a nice application where random walks arise in an unexpected manner.

The setup is standard: the user enters a query, the search engine finds all pages relevant to this query, and returns the list of these pages *in some order*. The key here is how to choose the ordering.

The first two steps are conceptually doable. The simplest idea (and one that was used in the mid 90's) was to represent each website as a document, perform a nearest neighbor procedure over the database of websites to a given query (similar to what we did in Lecture 1), and return the k nearest neighbors for some k. Of course, there are several issues – how to choose the features, how to do kNN efficiently etc but the basic idea is the same.

The last part is hard. Suppose we have several matches. How do we ensure that the most relevant matches are shown first? How to even define "most relevant"?

The core idea of PageRank is to defer the question of relevance to the structure of the web itself. One can imagine the web as some gigantic directed graph. The high level idea is that websites that have a

high number of incoming links (edges) are most likely authoritative, and hence should be assigned a high "relevance score".

(A second, and more subtle, idea in PageRank is that the relevance score of each website is *constant*, i.e., decoupled from the *semantic content* of the query. This is a bit harder to justify, but we will punt on this for now.)

Attempt 1

So, if the web has n pages, the goal is to produce a score vector $v \in \mathbb{R}^n$ that measures the relevance of each page. What are useful "relevance scores"? A first attempt is just simply count the incoming edges. If A is the adjacency matrix of the web, then the relevance score for all webpages is to declare the score of each webpage in terms of *in-degree*:

$$v(i) = \sum_{j=1}^{n} A_{ij} \quad \text{or,}$$
$$v = A^T \mathbf{1}_n,$$

where $\mathbf{1}_n$ is the all-ones vector.

This works, but is susceptible to manipulation since there could be webpages that simply consist of links to a large number of other webpages, and therefore are not very informative.

Attempt 2

There are two ways of fixing this. One way is to inversely weight the contribution of each incoming edge by the *out-degree* of the source of the edge. The second way is to use a recursive definition, and weight each incoming edge by the *score* of the source of the source of the edge. Overall, we get:

$$v(i) = \sum_{j=1}^{n} A_{ij} \frac{1}{d_j} v(j),$$

or:

$$v = A^T D^{-1} v,$$

where D is a diagonal matrix consisting of the out-degrees of each node in the graph. If any node in the graph has zero outgoing edges, we will add self-loops to all nodes so that the above vector is well-defined.

The above expression for v motivates the following interpretation: the stationary distribution of the random walk over the web graph with transition matrix $W = D^{-1}A$. Imagine a random surfer starting from a fixed page, and randomly clicking links to traverse the web. Intuitively, the limiting distribution of this random walk is going to be concentrated in webpages with a large number of incoming links.

Final algorithm

The PageRank algorithm makes an extra assumption: instead of a "random surfer" model, it assumes the "bored surfer" model where there is a small probability α that the surfer can stop clicking and teleport to a page chosen uniformly at random. In equations, we get:

$$v(i) = (1-\alpha)\sum_{j=1}^{n} A_{ij}D^{-1}v(j) + \alpha\frac{1}{n}.$$

or:

$$v = \left((1-\alpha)A^T D^{-1} + \frac{1}{n}\mathbf{1_n}\mathbf{1_n}^T\right)v.$$

Convince yourself that:

$$G = \left((1-\alpha)A^T D^{-1} + \frac{1}{n}\mathbf{1_n}\mathbf{1_n}^T\right)$$

is a valid transition matrix for a random walk. This is sometimes called the "Google matrix" and empirically the value of α is set to be 0.15.

Chapter 20

Random walks and electrical networks

This lecture explores the connections between random walks and *electrical networks*. More precisely, we will see how natural electrical quantities such as potential differences, currents, and resistances have analogous interpretations in the analysis of random walks on undirected graphs.

Below, we will assume that our underlying graph $G = (V, E)$ is an undirected graph with unit edge weights. Moreover, the graph is connected so that the stationary distribution is unique and well-defined.

Quantities of interest in random walks

A key quantity of interest in random walks is that of *convergence*, i.e., how quickly the random walk converges to the stationary distribution. Three quantities are of interest:

- the *hitting time* from u to v, $H_{u,v}$, is the expected number of hops required to traverse from node u to node v in the graph. The hitting time is *not symmetric* – it can take on average longer to reach from u to v than vice versa.
- the *commute time* between u and v, $C_{u,v}$, is the expected number of hops required to traverse a "round trip" from node u to node v back to node u. The commute time is given by $C_{u,v} = H_{u,v} + H(v, u)$. Unlike the hitting time, the commute time is a symmetric quantity.
- the *cover time* for u, C_u, is the expected number of hops starting from u to visit all nodes in the graph.

The hitting time, commute time, and cover time, encode key structural information about the edges of the graph. A well-connected graph (i.e., lots of paths from every node to every other node) will have low values of hitting, commute, and cover times in general. Several further examples are given in Chapter 5 of the textbook.

The hitting time $H_{u,v}$ can be estimated via the following argument (which is a bit hand-wavy, but can be made rigorous with some effort). Every path from u to v must pass through a *neighbor* of u. Therefore, the expected number of hops from u to v satisfies the relation:

$$H_{u,v} = 1 + \text{average}(\{H_{w,v} : w \text{ is a neighbor of } u\})$$

or,

$$H_{u,v} = 1 + \frac{1}{d_u} \sum_{(u,w) \in E} H_{w,v},$$

where d_u is the degree (number of neighbors) of u. For fixed v, this is a set of $n-1$ linear equations in $n-1$ variables, and can be solved by standard methods.

Connections to electrical networks

Now, consider the following alternate setup. Instead of the graph $G = (V, E)$, let us imagine a network of $|E|$ unit resistors connected according to the edges of G. Recall the electrical laws governing this network:

1. Kirchhoff's current law: the sum of currents into any node equals zero.
2. Kirchhoff's voltage law: the sum of potential differences around any loop in the network equals zero.
3. Ohm's law: the current through any edge equals the potential difference across the edge divided by the resistance of the edge.

We will show that familiar electrical quantities have connections with the quantities defined above. To see this, let us conduct a very different thought experiment. Given the above electrical network, at each node x in the network, inject d_x units of current (where d_x is the degree of x). Therefore, in total we are injecting:

$$\sum_{x \in V} d_x = 2|E|$$

units of current in this network. The above expression follows since the sum of degrees of nodes in a graph equals twice the number of edges in the graph, according to the familiar Handshaking Lemma from graph theory.

By Kirchhoff's Law, we cannot just inject current; we have to remove it somewhere. Let us remove all of the $2|E|$ units of current from node v. By principle of superposition of electric currents, this action is equivalent to the following: we inject d_x units of current into every node x other than v, and remove $2|E| - d_v$ units of current from v.

Now, the above injection of current induces potential differences. Since potentials are equivalent up to global shifts, we can set the "ground voltage" $\phi_v = 0$ without loss of generality. Further, since current flows from higher potential to lower potential, the potentials ϕ_x for every other node in the network is positive.

Now, let us apply Kirchhoff's current law to u. We have:

$$\begin{aligned} d_u &= \sum_{(u,w)\in E} \text{current}(u \to w) \\ &= \sum_{(u,w)\in E} \frac{\phi_u - \phi_w}{1} \text{ (Ohm)} \\ &= \sum_{(u,w)\in E} \phi_u - \sum_{(u,w)\in E} \phi_w \\ &= d_u \phi_u - \sum_{(u,w)\in E} \phi_w. \end{aligned}$$

Since $\phi_v = 0$, we can write everything in terms of potential differences $\phi_{x,v} = \phi_x - \phi_v = \phi_x$ for any x in the graph. Therefore,

$$\phi_{u,v} = 1 + \frac{1}{d_u} \sum_{(u,w)\in E} \phi_{w,v}.$$

Curiously, we observe that the above system of linear equations governing ϕ_u is *exactly the same* as those governing $H_{u,v}$. In other words, the hitting time can be interpreted as the potentials induced by setting up an appropriate set of current flows in a given graph, i.e.,

$$\phi_{u,v} = H_{u,v}.$$

It is interesting that the quantity on the left is purely obtained through electrical laws, while the quantity on the right is purely a statistical object that arises in the analysis of random walks.

One more curiosity. Call the above "Scenario 1". Now, construct "Scenario 2" as follows. It is the same as above, except that

i. we *extract* d_x units of current out of each node in the graph.

ii. to conserve current, we inject $\sum_x d_x = 2|E|$ units of current into the graph into node u.

Therefore, the analysis of Scenario 2 is identical to the one above, except that the role of v before is now played by u, and that all directions of current are reversed. Therefore, by a similar reasoning as above, we have the new potential difference between u and v given as:

$$\phi'_{u,v} = H_{v,u}.$$

Now, we apply principle of superposition again to add up Scenario 1 and Scenario 2. All currents and potentials are linearly added up. Therefore, the resultant potential difference between u and v is:

$$\phi''_{u,v} = \phi_{u,v} + \phi'_{u,v} = H_{u,v} + H_{v,u}$$

and the only currents in and out of the network being $2|E|$ units of current into u and $2|E|$ units of current out of v.

Therefore, by Ohm's Law, the *effective resistance* between u and v is given by

$$\frac{\phi''_{u,v}}{\text{current}(u \to v)} = \frac{H_{u,v} + H_{v,u}}{2|E|} = \frac{C_{u,v}}{2|E|}.$$

Therefore, the *commute time* between u and v is nothing but the effective resistance between u and v multiplied by the total number of edges in the network.

Applications

Long story short: several tools/techniques from electrical analysis can be used to quickly reason about random walks. Here is a simple toy example known as *Gambler's Ruin* which is often encountered in economics. Suppose a gambler enters a casino and places an indefinitely long sequence of bets worth \$1. He wins an extra \$1 at each step if he wins the bet and loses the \$1 otherwise. Suppose there is an equal probability of either event happening.

The gambler decides to go home either if he wins $N =$\$1 million, or goes broke. The question is: if the gambler starts off with \$i, how long is he expected to stay in the casino?

We model this problem as an instance of a random walk, where the nodes (states) represent the amount of money (in dollars) that the gambler currently possesses. The walk starts off at i, and terminates at either 0 or $N = 1,000,000$. The graph in this case is simple: it is a chain graph with $N + 1$ nodes and N edges. Instead of computing transition matrices and solving for conditional probabilities, one can quickly identify the quantities of interest as the *hitting times* between i and 0, or i and N. By invoking the connections made above, one can solve it using ordinary electrical circuit theory.

Chapter 21

Streaming algorithms

Thus far, all the techniques that we have discussed assume that the data to be analyzed *fits into memory* and that we are free to compute/manipulate it as we please.

This is not always the case, particularly in the era of *big data*. In applications such as surveillance, climate monitoring, and communications, a variety of data sources – sensors, video cameras, network routers – are in "always on" mode, and continuously generate data of various modalities. Within a short period of time, the amount of data gathered can run into tera- or even peta-bytes, and representing and storing such data can be a severe challenge. Traditional analytics approaches (regression, classification, etc.) can be prohibitive.

What if we were asked to process a large corpus of data, but were only given enough space to *store* a tiny fraction of it? This is the type of problem addressed via *streaming algorithms*. Precisely, the data is assumed to be in the form of a stream of elements:

$$x_1, x_2, x_3, \ldots, x_n,$$

where for convenience it is assumed that the data can be well-represented by some finite (countable) space, e.g., 64-bit floating point numbers. However, the amount of memory available to the data processor is much smaller than the number of data points say $O(\log^c n)$, where c is some tiny constant. Think of n being 10^{20} or similar, while the memory availability being in the order of a few kilobytes.

Further, assume that new samples are arriving at a very fast rate, so we do not have the luxury to wait very long for our analysis method to finish. Therefore, in addition to small memory, the *per-item* processing time of each element is also very small. In particular, it is not possible to touch data points more than once. (This automatically rules out iterative methods such as gradient descent, which revisits the available data points several times.)

At first glance, this kind of data analysis task looks impossible; how can we perform meaningful calculations on the data in such resource constraint situations? Fortunately, a couple of key ideas come into play:

1. *Sketching*. Instead of storing all of the data, we construct *summaries* or *sketches* of the data that support efficient analysis/inference via post-processing.
2. *Sparsity*. Often, the data is sparse and only a small fraction of its entries are significant/relevant.

3. *Randomness*. Lastly, we will use a powerful idea often used in algorithms for large-scale data analysis: we will let our methods be *randomized*. (We have already seen the power of randomization in speeding up algorithms, cf. stochastic gradient descent over vanilla gradient descent.)

A warmup: the missing element problem

Here is a toy problem that illustrates the idea of a sketch.

Let us say we have a stream of m numbers:

$$a_1, a_2, \ldots, a_n$$

where each a_i is a distinct number in $[0, n]$. Therefore, from the pigeon hole principle there is exactly one element from $[0, n]$ that is missing from the stream. How do we efficiently identify that element?

The simple way is to initialize an all-zero counter array for each element i (say $c[i]$), and increment $c[i]$ whenever we observe an element i in the stream. Our final estimate of the missing element is given by that i^* for which $c[i^*] = 0$.

However, this is inefficient (since the array consumes $O(n)$ space.) Can we build a smarter algorithm that uses lesser memory? *Yes*. Instead of a counter, we will simply maintain the running sum:

$$S_i = \sum_{j=1}^{i} a_i.$$

This can be done in a streaming fashion: we initialize S_0 as zero, and at each appearance of a new element a_i for $1 \leq i \leq n$, we update:

$$S_i \leftarrow S_{i-1} + a_i.$$

Finally, we output:

$$i^* = \frac{n(n+1)}{2} - S_n.$$

The above algorithm is certifiably correct for all inputs (since i^* does not appear in the final expression for S.) Moreover, since S_n can be at most $n(n+1)/2$, the maximum amount of space required to store S_n is $\log(n(n+1)/2) = O(\log n)$ bits of memory. Done!

The above algorithm is simple, but illustrates a basic idea in streaming algorithms. The quantity S_i is the *sketch* of the stream; it summarizes the contents of the stream into one number that is sufficient to solve the entire problem. The algorithm post-processes the sketch (in this case, a simple subtraction) in order to produce the desired answer. Moreover, the sketch can be stored using only logarithmic amount of memory.

The distinct elements problem

Let us now discuss a more realistic problem that is encountered in web-based applications. Say, for example, the data elements x_i represent a stream of IP addresses pinging a server, and the goal is to

detect the number of distinct IP addresses in the stream (or the number of "uniques"; this quantity is often of interest to advertisers.)

As above, let us suppose that n is very large; so is m, the number of all potential IP addresses (also called the "namespace").

The naive algorithm stores a count array $c[i]$ for each i in the namespace, and increment $c[i]$ each time element i is observed in the stream. At any given time we can simply count the number of nonzero entries of c (also called the "ℓ_0 norm" of c) and declare that to be the number of unique elements. However, this is inefficient for large m (since it requires at least $O(m)$ bits of memory.)

A (somewhat) different method stores a *list* of all items seen thus far. Each item in the namespace can be represented using $O(\log m)$ bits, and therefore the overall algorithm requires $O(n \log m)$ bits. This is very inefficient for large n.

A far better, and particularly nice, algorithm by Durand and Flajolet proposes the following *sketching*-based method. The idea is to not store the count vector $c[i]$, but rather a sketch of c as follows. This technique uses randomization.

First, we will solve a somewhat simpler *decision* version of the problem: suppose we guess the number of distinct elements as T, we will design an algorithm that will tell us (with high probability) whether or not the true answer is within some fraction (say 90 % smaller or larger) than T. If such an algorithm can be constructed, then we can simply call this algorithm with some geometric series of guesses between 0 and m, and one of these instances will give us a "yes" answer.

The algorithm proceeds as follows. Fix some parameter $T > 0$ to be specified soon. The algorithm selects a random subset $S \subset \{1, 2, \ldots, m\}$ such that:

$$\text{Prob}(i \in S) = 1/T.$$

independently for each i. Then, we maintain

$$X_S = \sum_{i \in S} c[i].$$

Intuitively, X_S aggregates the counts of all the elements of S seen in the stream. Since S is chosen randomly, if $T = 3$ (say) then on average $1/3$ of the set of distinct elements falls into S. In general if α is the number of distinct elements of the stream, then by independence, roughly α/T of these elements fall into S and are tracked by X_S.

This gives us a way to *guess* α. If $\alpha \gg T$ then most likely X_S is going to be nonzero. If $\alpha \ll T$ then most likely X_S is going to be zero. This can be formalized as follows. Let p be the probability that $X_S = 0$. This can only happen if each of the distinct elements of the stream fails to fall into S. The failure probability is $1 - 1/T$ and there are α such elements, so that:

$$p = (1 - 1/T)^\alpha \approx e^{-\alpha/T}.$$

Therefore, by simple algebra:

- if $\alpha > (1 + \varepsilon)T$ then $p < 1/e - \varepsilon/3$.
- if $\alpha < (1 + \varepsilon)T$ then $p > 1/e + \varepsilon/3$.

Of course, we have no way to precisely measure this probability p from only one counter X_S – it could be high or low depending on the specific trial. Therefore, we now use a commonly used trick in randomized algorithms. We repeat the above procedure several times in parallel, i.e., we select k copies $S_1, S_2, \ldots, S_k$, and store counters

$$X_{S_1}, X_{S_2}, \ldots, X_{S_k}$$

as above, and let Z as the total number of counters that are zero. By linearity of expectation, we have:

$$E[Z] = kp,$$

where p, as we calculated above, is the probability that any one counter is zero. Suppose we set $k = O(\frac{1}{\varepsilon^2}\log(\frac{1}{\eta}))$. By Chernoff's inequality on the tail probability of binomial random variables, we get that with probability $1 - \eta$:

- if $\alpha > (1+\varepsilon)T$ then $Z < k/e - \varepsilon$.
- if $\alpha < (1-\varepsilon)T$ then $Z > k/e - \varepsilon$.

Therefore, the overall algorithm will look at the number of zero counters, and compare with k/e. If this number is within $k/e \pm \varepsilon$, the algorithm will output YES, else will output NO.

Therefore, we have produced a randomized algorithm that, with probability $1 - \eta$, will tell us whether or not the number of distinct elements in the stream is within a fraction $1 + \pm\varepsilon$ of the true answer. This is the decision version of the distinct elements problem. To obtain the estimation version (i.e., guess the actual number of distinct elements), we run the above module several times in parallel with parameters

$$T = 1, (1+\varepsilon), (1+\varepsilon)^2, \ldots, n.$$

The number of such T's is $\log_{1+\varepsilon} n \approx \log n/\varepsilon$. Therefore, the total amount of memory required by the entire algorithm is given by:

$$O(\frac{\log n}{\varepsilon}\frac{1}{\varepsilon^2}\frac{1}{\eta}).$$

Chapter 22

Streaming algorithms (contd)

We now discuss the *frequent elements* problem in data streams. The goal is to efficiently identify which items occur the most frequently. Applications of this problem arise in web analytics, monitoring, surveillance, etc.

Finding the majority element

Let us start simple. Suppose there are n voters in an election with m candidates. We observe the votes in a stream:

$$x_1, x_2, x_3, \ldots, x_n.$$

How do we determine who wins the *majority* vote (>50%), if such an element exists?

The naive way is to form a count array $c[i]$ of size m, and update counts each time we observe a vote for a particular candidate. The total space requirement is given by $O(m \log n)$. In the end, we look at the maximum of this array and output the corresponding index.

Is there a more efficient algorithm? Yes – provided we weaken the requirements a little. Let us say we are satisfied with one of two scenarios:

- if there is a candidate with more than 50% of the vote, then the algorithm has to identify that candidate.
- if there is not, then the algorithm can report any one candidate.

Here is a particularly nice algorithm called *Boyer's method* to solve this problem:

1. Initialize single counter c to 1, and store $cand = x_1$.
2. For each subsequent $x_i,\ i \geq 2$:
    a. If $x_i = cand$, increment c.
    b. If $x_i \neq cand$ and $c > 0$, decrement c.
    c. If $x_i \neq cand$ and $c = 0$, update $cand = x_i$ and increment c.

3. Output $cand$ as the majority element.

The above algorithm only requires $\log n + \log m$ memory; $\log n$ bits to store c, and $\log m$ bits to store cand. Again, c here can be viewed as the *sketch* (or summary) of the stream that lets us solve the problem.

Why should this work? The high-level idea is that if there is a majority element MAJ, then every occurrence of a *non* majority-element can be matched with an occurrence of a majority-element. The counter c keeps track of how many matches have happened; if there indeed exists a majority-element, then it has occurred more than 50% of the time, c will be bigger than 0, and the eventual value of $cand$ will necessarily be MAJ.

Note that if there exists no majority element in the stream, then the algorithm could output any arbitrary element and $cand$ could be arbitrarily far from being the majority. In general there is no easy way to detect whether we are in Scenario 1 or 2. Therefore, in principle one has to go over the stream once again and count the occurrences of $cand$ to confirm whether it indeed is the majority element.

Heavy hitters

One can modify this algorithm to not just detect the majority element, but also to detect a *set* of most frequently occurring elements in a stream.

Let us define a *heavy hitter* as any element that that occurs with frequency greater than εn for some parameter $\varepsilon > 0$. (The case $\varepsilon = 0.5$ corresponds to the majority element problem described above.) The goal is to detect all heavy hitters, if they exist.

An algorithm by Misra and Gries proposes the following approach:

1. Maintain a list of $\lceil 1/\varepsilon \rceil$ candidates + counters $f_1, f_2, \ldots$. Initialize everything to null.
2. For each new element (say s) encountered in the stream:
    a. If s already occurs in the list of candidates, increment its corresponding counter.
    b. If s does not occur in the list of candidates and the list is not full (i.e., there are less than $\lceil 1/\varepsilon \rceil$ candidates) then add to the list and increment its counter by 1.
    c. If s does not occur in the list (which is full) then decrement all counters by one. If any counter becomes zero, then eliminate the candidate from the list.

The above algorithm generalizes the idea that we used in detecting the majority element. Moreover, one can show the following correctness property:

- Each counter reports a list of candidates-counts (s, f_s) such that if the *true* count of s is given by c_s, then the estimated count f_s satisfies:

$$c_s - \varepsilon n \leq f_s \leq c_s.$$

- Moreover, if any element t does not occur in the list, then necessarily its count c_t satisfies:

$$c_t \leq \varepsilon n.$$

The above two properties tell us that all heavy hitters (if they exist) will definitely be in the output of the algorithm, and all elements that are not in the output cannot be heavy hitters. There can never be false negatives; however, there *could* be false positives (if there are no heavy hitters in the stream then the output can be arbitrary.)

As a nice by-product, this algorithm also gives us a way to approximate count values up to some additive error εn.

Higher frequency moments

Distinct elements/heavy hitters/frequent elements all involve computing/testing for some properties of the count vector of a stream.

A more general class of problems involve estimating frequency *moments*. Higher order moments such as variance, kurtosis, etc give us important statistics about streams. They also enable efficient estimation of space/memory requirements in all kinds of database applications.

The simplest example is that of the second moment. Consider a stream with m items containing frequency counts $c_1, c_2, \ldots, c_m$. We already know that $c_1 + c_2 + \ldots c_m = n$. In order to estimate the variance of frequency counts, the goal is to efficiently calculate:

$$F_2 = c_1^2 + c_2^2 + \ldots c_m^2.$$

Once again, naive method: estimate all the c_i's by keeping count of each element. This takes $O(m \log n)$ space — bad!

Alternately, we could use the above heavy hitters algorithm (by Misra-Gries) and *approximate* the value of each count by obtaining the quantities $f_1, f_2 \ldots, f_(1\varepsilon)$. This is certainly space-efficient. However, since each count is only approximated up to an error εn, the total error in estimating F_2 can be as high as $m\varepsilon^2 n^2$. This can be very bad unless ε is very, very small – which means we need lots of counters in Misra-Gries to get good estimates.

Here is a better algorithm by Alon, Matias, and Szegedy. It uses *randomness* in a very clever manner (similar to how we used it to solve the distinct elements problem.)

Suppose we pre-select a set of i.i.d. random Bernoulli variables:

$$a_1, a_2, \ldots, a_m$$

such that:

$$Pr(a_i = 1) = Pr(a_i = -1) = 1/2.$$

Now, we maintain the quantity:

$$Z = a_1 c_1 + a_2 c_2 + \ldots a_m c_m.$$

Note that this can be maintained in a streaming fashion; we initialize $Z = 0$ and each time we see element i, we update $Z \leftarrow Z + a_i$.

At the end of the stream, we output $F_2 = Z^2$. That's it!

Why should this be a good idea? Observe that Z is a random variable (since a_i are chosen at random). Moreover, the expected value of Z^2 is:

$$\begin{aligned}
E[Z^2] &= E[(a_1c_1 + a_2c_2 + \ldots a_mc_m)^2] \\
&= E[\sum_i a_i^2c_i^2 + 2\sum_{i>j} a_ia_jc_ic_j] \\
&= E[\sum c_i^2 + 2\sum_{i>j} a_ia_jc_ic_j] \\
&= \sum c_i^2 + 2\sum_{i>j} E[a_ia_j]c_ic_j \\
&= \sum c_i^2,
\end{aligned}$$

since $E[a_ia_j] = E[a_i]E[a_j] = 0$.

Therefore, Z^2 is an unbiased estimator of F_2. However, since it is only a single estimator, it can have a high variance. Fortunately, via some tedious but straightforward calculation, one can estimate the variance of Z^2 as no greater than $2F_2^2$. Therefore, we can simply keep $k = O(1/\varepsilon^2)$ copies of Z in parallel (with a different set of random a_i's for each Z) and in the end output the average of all the $Z_i's$. A simple application of Chebyshev's inequality shows that this new average of the $Z_i's$ is sufficient to estimate F_2 up to $(1 \pm \varepsilon)$ accuracy.