CSE 422A: Advanced Toolkit for Modern Algorithms

University of Washington

Andrew Chen

Winter 2026

Hello and welcome! This is my lecture notes on CSE 422A: Advanced Toolkit for Modern Algorithms. This course is a continuation of the typical undergraduate-level algorithms course. In this course, we do analysis on many “modern” algorithms. The professor is **Jerry Li**, and we meet TTh at **11:30 am** for lectures. There are no required textbooks for this course. However, as we frequently analyze modern era algorithms, there will be relevant research papers, for which I will link in their respective sections. Also note that theorem names might not necessarily be accurate; it's probably just whatever my textbook / professor said it is.

The goal of these lecture notes is to write **understandable** math. As the great Albert Einstein put it, ”If you can't explain it to a six year old, then you don't understand it yourself”. The hope is that anyone coming across these notes (like you!) will be able to at least take away the gist of these concepts. Should you find any errors in my mathematics, please contact me at zchen66@uw.edu

Contents

List of Definitions

List of Theorems

1 Lecture 03: Jan. 13th

summary

1.1 Metric data & similarity search

previously, data was hashed, but we didnt really care about the structure of the data points. But now, we do.

How do we measure how close data points are to each other?

Example 1.1. Points on a map. How far? There are many ways of doing this. One obvious one is Euclidean Distance, but on a map, strict Euclidean Distance might not represent how long it takes you (i.e., crossing a river vs. driving around it).

Example 1.2. Image simularity. How do we classify if one cat "looks like" another?

Example 1.3. Distances between strings. For example, DNA matching. Edit distance! (algos ptsd)

So mathematicians have thought about this for *quite* some time. Let's dive into some definitions.

Definition (Metric space)**.** A metric space is a pair (X, d), where X is some set of points, and $d : X \times X \to \mathbb{R}_{\geq 0}$ is some distance function that satisfies the following properties:

1. $d(x, y) = 0$ if and only if $x = y$
2. Symmetry: $d(x, y) = d(y, x)$
3. Triangle inequality: For all $x, y, z \in X$, $d(x, z) \leq d(x, y) + d(y, z)$

Always a good idea to have some examples after a definition.

Example 1.4. l_p-spaces, where $X = \mathbb{R}^d$. Here, we have

$$d(x, y) = ||x - y||_p = \left(\sum_{i=1}^{d} |x_i - y_i|^p \right)^{1/p}$$

Here, this is a metric space $\forall p \in [1, \infty]$.

Looking at specific values of p, we have $p = 2$ refers to the Euclidean space (your typical xy-plane).

More interestingly perhaps, $p = 1$ gives us what's known as the "Manhattan distance".

And most interestingly, we have $p = \infty$. We're allowed to do this because the expression above has a nice limit when we take $p \to \infty$. This gives us $d(x, y) = \max_{i=1,\ldots,d} |x_i - y_i|$

Example 1.5. String distances, where $X = \Sigma^*$. How do we measure "distance" between strings?

We've mentioned this above, but one possible way to measure this is with **Edit distance** !! Specifically, we define

$$d_{\text{edit}}(s_1, s_2) = \text{Minimum num of insertions and deletions to go from } s_1 \text{ to } s_2.$$

To compute edit distance, we use dynamic programming and achieve this in $O(mn)$. This was algos (CSE 421) ptsd.

But it turns out, there actually exists an algorithm $O(n^{1+\delta})$ that is able to get a $(1+\delta)$-approximation to edit distance. [Chakraborty, Das, Goldenburg, Kouchy, Sacks] at FOCS 2018.

There's another notion of distance known as the **Hamming distance**. We define this as

$$d_{\text{hamming}}(s_1, s_2) = \text{num of terminise replacements to get from } s_1 \text{ to } s_2$$

Example 1.6. Jaccard similarity, where $X = 2^\Omega$. This measures the similarity between different sets. For two sets S and T, we define

$$J(S, T) = \frac{|S \cap T|}{|S \cup T|}$$

Here, it's important to remark that the notion of "distance" is a bit different from the before cases such as edit distance. We have that a larger Jaccard simularity means *more similar sets.*

Because of this, we actually define, for this metric space,

$$d_j(S, T) = 1 - J(S, T)$$

Proposition 1.1 (Metric embeddings).
Given two metric spaces (X_1, d_1) and (X_2, d_2), is there a bijection $f : X_1 \to X_2$ such that

$$d_1(x, y) \approx d_2(f(x_1), f(x_2))$$

Okay that has nothing to do with the lecture, but just cool to point out.

This is too much math for a cs class, let's get into some computer science man! Let's take a look at the problem of the day:

Problem 1: Nearest neighbor problem.

Let (X, d) be a metric space, and let $D \subset X$ be a dataset. Preprocess D such that given a query point $q \in X$, quickly find $x \in D$ such that $d(x, q)$ is minimized for $x \in D$.

For today, we define $X = \mathbb{R}^k$ (all data are k-dimensional), and we familiarize ourselves with the L2 norm $d = ||\cdot||_2$.

There, of course, is an approximate version of this problem.
Approx-NN: Given parameter $\epsilon > 0$, output $x \in D$ such that

$$d(x, q) \leq (1 + \epsilon) \cdot \min_{x \in D} d(x, q)$$

Regardless, the overall goal here is that for $|D| = n$, we ideally want space $O(nk)$ and answer queries in time $O(\log n)$. For the scope of this course, we will be looking at two different approaches / methods to tackle both the exact and approximate versions of this problem.

1. **"Space partitioning" methods:** kd-trees, voronoid diagrams, etc.
2. **Random projection methods:** Johnson-Lindenstrauss projections, locality sensitive hashing, etc.

Let's dive in!

Method 1: Space partitioning – k-d trees [Bentley '75]

Idea: build a tree to partition $\mathbb{R}^k$:

- Leaf nodes: Single data point $x \in D$
- Non-leaf nodes: $i \in \{1, ..., k\}, v \in \mathbb{R}$, with (i, v)

Algorithm 1 Building a *kd*-tree

```
1: Maintain some set of data points which are "active". Initially: $D$.
2: Recursively: at node $v$ with active $D_v$
3: if $|D_v| = 1$ then
4:     $v$ is a leaf node with label $x \in D_v$
5: if $|D_v| > 1$ then
6:     pick some $i \in \{1, ..., k\}$
7:     let $m_v = \text{median}(x_i)$
8:     $v$ becomes $(i, m_v)$
9: recurse on the two split data sets
```

This is great so far... we have size $O(nk)$, and depth $O(\log n)$. We're definitely on the right track.

Let's now discuss look-up. The key idea for look up is looking within these pre-deterimined "cells", rather than brute-force linearly explore the distance between every point.

However, as it stands, there is a fundamental difficulty with the way we've constructed the tree. The problem is that the closest point *may not lie in the same cell as the point-of-interest*. So actually, once we find the point within the same cell, we would still have to recurse up the kd-tree to rule out all other points.

To formalize this, in the worst case, we may have to look up in $O(n)$ time – brute forcingly still checking every cell and every point.

But maybe ending on a hopeful note, in low-dimensions on average, look-up is in $O(2^k \cdot \log n)$, with the caveat that this is the case in $k \leq 20$. This is what we refer to as the **"curse of dimensionality"**.

Because it turns out in higher dimensions, points tend to be more similar to each other. This is why we tend to use approximation algorithms for higher-dimensional data.

2 Lecture 05: Jan. 20th

summary

2.1 Learning from data

We started very quickly with a reminder on Chernoff (style) bounds.

Let $X_1, ..., X_n$ be independent and identically distributed Random Variables with $\mathbb{E}[X_i] = \mu$. Then, we have

$$\Pr\left[\left|\frac{1}{n}\sum_{i=1}^{n} X_i - \mu\right| > t\right] \leq c_i \exp(-c_2 \cdot n \cdot t^2)$$

For some conntext, the content we are covering toady is very improtant to today's world in machine learning and supervised learning. It's all to do with data, of course.

Previously, for some dataset X, we've concerned ourselves with what we can learn about X itself. And today, we are going to learn about the process underlying X. This is a very basic problem in ML!

Example 2.1. Image classification (supervised learning). In this case, we're given a large dataset of images and their labels. Our goal would be to identify unlabeled images with accuracy.

$$f: \text{ new image} \to \{\text{"cat", "human"}\}$$

So how do we formalize this? We introduce the **Binary Classification Task**. We have

- D probability distribution over $\mathbb{R}^d$.
- Some (unknown) ground truth classifier $f : \mathbb{R}^d \to \{0, 1\}$
- Samples $X_1, ..., X_n \sim D$ that are independently selected
- Labels $y_1, ..., y_n$ with $y_i = f(x_i)$

Input: $\{(x_1, y_1), ..., (x_n, y_n)\}$
Output: Some learned predictor $g : \mathbb{R}^d \to \{0, 1\}$

Goal: $g \approx f$, where g "looks like" f on some unseen data. Let's make a few definitions.

Definition (Generalization Error Measurement)**.**

$$\text{GenError}(g) = \Pr_{X\sim D}[g(X) \neq f(X)] \approx \frac{1}{m}\#\{i : g(x'_i) \neq y'_i\}$$

Does this look familiar? How can we dig into this? Well, looking at the approximation, if we reframe the question, it may look like a Chernoff bound!

Definition (Training error)**.**

$$\text{TrainError}(g) = \frac{1}{n}\#\{i : g(x_i) \neq y_i, i = 1, ..., n\}$$

Here, n is the number of training data points. Importantly, note that

$$\text{TrainError}(g) \neq \text{GenError}(g)$$

Overfitting occurs when Training Error is small but Generalization Error is large.

Let's take a look at some examples now.

Example 2.2. Finite, well-separated hypotheses. Let

$$\mathcal{A} = \{f_1, ..., f_k\}$$

is a finite set of predictors. There are two settings at play:

1. (Realizable) ground truth $f \in \mathcal{A}$
2. (Agnostic) grouth truth is well-approximated by something in $\mathcal{A}$

Let's make a few assumptions:

1. We're in the realizable setting
2. $\forall f_i \neq f$, we have $\text{GenError}(f_i) \geq \epsilon$

We concern ourselves with a very simple question:

How large does n have to be to learn f?

Naively, our first approach could be to just output any $g \in \mathcal{A}$ where $\text{TestError}(g) = 0$. I claim that for any $\delta > 0$, let $n = \frac{1}{\epsilon}(\log k + \log \frac{1}{\delta})$, then

$$\Pr_{x_1,...,x_n}[g \neq f] \leq \delta$$

Proof. Let $f_i \in \mathcal{A}$, where $f_i \neq f$. Then,

$$\begin{aligned}\Pr_{x_1,...,x_n}[\text{TrainLoss}(f_j) = 0] &= \prod_{i=1}^{n} \Pr[f_j(x_i) = f(x_i)] \\ &\leq (1-\epsilon)^n \\ &\leq e^{-\epsilon n} \\ &\leq \frac{\delta}{k}\end{aligned}$$

From here, by union bound,

$$\Pr[\exists f_j \neq f \text{ s.t. } \text{TrainLoss}(f_j) = 0] \leq \sum_{f_j \neq f} \Pr[\text{TrainLoss}(f_j) = 0] \leq (k-1) \cdot \frac{\delta}{k} \leq \delta$$

□

Note here that assumption (2) is very powerful. What if the predictors are not well-separated?

... PAC ...

3 Leture 07: Jan. 27th

summary

3.1 Principal Component Analysis

Question: How do we find important distinguishing directions in data?

$$x_1, ..., x_m \in \mathbb{R}^d$$

Question: How do we find a "good" low-dimensional representation?

For both of the qusetions above, we COULD refer to the locality-sensitive hash (LSH) that we went over in the previous lecture. However, an issue with that is it *only* preserves distance, but what about other things? We've seen that in high dimensions, "distance" is a lossy form of data.

Let's look at an example.

Example 3.1. People eating.

	Kale	taco bell	sashimi	pop-tarts
Alice	10	1	2	7
Bob	7	2	1	10
Carol	2	9	7	3
Dave	3	6	10	2

How do we visualize this dataset the best?

Step 1: Center the data - $\mu = (5.5, 4.5, 5, 5.5)$

Step 2: Find two good directions: finding v_1, v_2 orthogonal such that

$$x - \mu \approx a_1 v_1 + a_2 v_2 \quad , x \in \{\text{Alice, Bob, Carol, Dave}\}$$

This gives us

$$v_1 = (3, -3, -3, 3) \qquad v_2 = (1, -1, 1, -1)$$

They are orthogonal. Trust me bro. More importantly, notice that v_2 is the direction that indicates "eating healthy food" (since we want kale and sashimi rather than taco bell and pop-tarts). Whereas v_1 is the direction that indicates "vegetarian" (since we only eat kale and pop-tarts).

Step 3: Center the data

$$\begin{aligned} \text{alice} - \mu &\approx v_1 + v_2 \\ \text{bob} - \mu &\approx v_1 - v_2 \\ \text{carol} - \mu &\approx -v_1 - v_2 \\ \text{dave} - \mu &\approx -v_1 + v_2 \end{aligned}$$

yay! we notice that based on the results of the principal components, who is vegetarian and who is health-conscious. Something that we may not have noticed based purely on the data.

Okay now how to we actually find the good orthogonal components? Let's generalize our procedure.

Step 1: Center the data

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

and set $x_i' = x_i - \mu$ for $i = 1, ..., n$ Without loss of generality, $x_i = x_i'$, so $\mu = 0$.

Step 2: Find a subspace $V \subseteq \mathbb{R}^d, \dim(V) = k$ such that $x_i \approx \text{proj}_V(x_i)$. Here, $k \ll d$.

More concretely,

$$V = \text{span}(\{v_1, ..., v_k\})$$

Take $v_1, ..., v_k$ to be orthonormal. The objective of our PCA is to find the argmax vectors $v_1, ..., v_k$ of the following expression:

$$\sum_{i=1}^{n}\sum_{j=1}^{k}\langle x_i, v_j\rangle^2 = \sum_{i=1}^{n}\|\text{proj}_v(x_i)\|_2^2$$

The goal here is to maximize the norm, as that would imply that the changes made to the data points are minimized.

Step 3: Suppose we've found the maximizers $v_1, v_2, ..., v_k$. These maximizers are known as *principal components.*

Now, we project our original vectors into the principal component space where

$$x_i \approx \text{proj}_{v_1,...,v_k}(x_i) = \sum_{i=1}^{k} c_i v_i$$

Remark. Principal components are not always unique, but they often are.

If the principal components are unique, then they have a nested structure, where the ith principal component space is $\{v_1, ..., v_i\}$.

As a side note, modern day algorithms almost always uses **singular value decomposition** to calculate the principal components.

4 Lecture 09: Feb. 3rd

summary

4.1 Singular Value decomposition

Recall from last lecture our discussion about symmetric matrices, where $A \in \mathbb{R}^{n \times n}$ is symmetric if $A_{ij} = A_{ji}$ for all i, j.

With these symmetric matrices, we can build a decomposition that gives us lots of spectral information about A, where

$$A = UDU^T$$

D is a diagonal matrix where entries are the eigenvalues of A, and each column U_i is an eigenvector of A. In other words, we can calculate

$$A = \sum_{i=1}^{n} \lambda_i u_i u_i^T$$

Notice that $u_i u_i^T$ is a rank 1 matrix, and each λ_i determines the "importance" of this matrix.

The issue with this is that thus far, we've required A to be symmetric. In today's lecture, we are going to generalize this for *all* matrices $A \in \mathbb{R}^{m \times n}$.

Theorem 4.1 (Singular Value Decomposition).
Let $A \in \mathbb{R}^{m \times n}$. Then A can be written as

$$A = USV^T$$

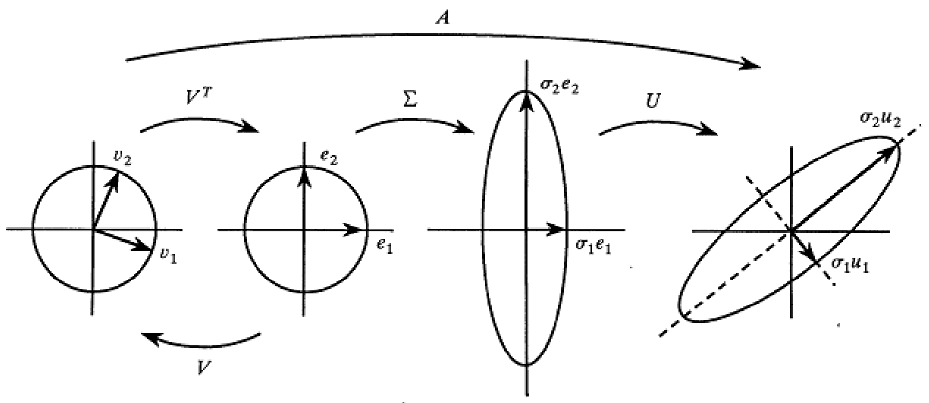
where:

- *$U \in \mathbb{R}^{m \times m}$ contains orthonormal vectors, we call the **left singular vectors** of A*
- *$S \in \mathbb{R}^{m \times n}$ contains only diagonal entries $\sigma_i \geq 0$ we call the **singular values** of A*
- *$V \in \mathbb{R}^{n \times n}$ form an orthonormal basis for $\mathbb{R}^n$, and we call the **right singular values** of A*

Equivalently,

$$A = \sum_{i=1}^{\min(n,m)} \sigma_i u_i v_i^T$$

To interpret this, let's call $M = u_i v_i^T$, and we multiply M by some vector x. Then, $Mx = \langle v_1, x \rangle \cdot u_i$.



The point is, SVD is the analog of the eigenvalue decomposition, just on $m \times n$ asymmetric matrices. Let's explore this connection further:

First, if A is symmetric, then we can get SVD from eigenvalues / eigenvectors. How do we make A symmetric? Consider

$$\begin{aligned} A^T A &= (USV^T)^T \cdot USV^T \\ &= VSU^T \cdot USV^T \\ &= VSI_{n\times n}SV^T \\ &= VS^2V^T \end{aligned}$$

Now doesn't this look a lot like the eigenvector decomposition that we're familiar with? Here, the eigenvalues of $A^T A$ are σ_i^2, and the eigenvectors are v_i.

Similarly, we have

$$AA^T = USU^T$$

giving us the same eigenvalues σ_i^2, and the eigenvectors are u_i.

*****

Recall that for PCA with data matrix X, then the top k PCs are the top k eigenvectors of $X^T X$. But as we've discussed just now, these are just the top k right singular vectors of X!

This means that given some $A \in \mathbb{R}^{m\times n}$, we can compute its SVD to find the singular vectors. How do we compute it fast? Well,

`np.linalg.svd(`A`)`

ts is overengineered because how much its used.

4.2 Low-rank approximation

How do we apply SVD? We can do what's called a *low-rank approximation.* But first, recall the definition:

Definition (Matrix rank)**.** A matrix $A \in \mathbb{R}^{m \times n}$ has rank 0 if $A =$ all zeroes.
A matrix $A \in \mathbb{R}^{m \times n}$ has rank 1 if $A = u \cdot v^T$

A matrix $A \in \mathbb{R}^{m \times n}$ has rank k if

$$A = \sum_{i=1}^{k} u_i v_i^T$$

and cannot be written as a sum of $k-1$ such matrices.

Remark. This is one of many equivalent definitions of rank. The following are also equivalent:

1. A has rank k
2. The largest set of linearly independent columns of A has size k
3. The largest set of linearly independent rows of A has size k
4. range(A) has dim(k)
5. A **has** k **nonzero singular values**

But why do we care about low-rank? Let's look at an example.

Example 4.1. Consider the following matrix.

$$\begin{pmatrix} 7 & & \\ & 8 & \\ & 12 & 6 \\ & & 2 \\ 21 & 6 & \end{pmatrix}$$

If we were told that this matrix was low-rank, how would we fill in the data? Well, we'd see that the third column is simply half the second, and then deduce that the first column is 7 times the third, giving us:

$$\begin{pmatrix} 7 & \mathbf{2} & \mathbf{1} \\ \mathbf{28} & 8 & \mathbf{4} \\ \mathbf{42} & 12 & 6 \\ \mathbf{14} & \mathbf{4} & 2 \\ 21 & 6 & \mathbf{3} \end{pmatrix}$$

In the real world, while usually not exactly low-rank, data is much often close to low-rank. So when we're "missing data" like the above example, we can find simple ways to approximate said data.

Question. Given A, find B which is low-rank, and which is "close" to A.

Definition (Frobenius norm)**.** For a matrix A, let

$$\|A\|_F^2 = \sum_{i,j} A_{ij}^2 = \sum_{i=1}^{\min(n,m)} \sigma_i^2$$

Goal. Given A, find B low-rank such that

$$\|A - B\|_F \leq \epsilon \cdot \|A\|_F$$

This preserves many of the multiplicative structure of A.

Theorem 4.2.
For any $A \in \mathbb{R}^{m\times n}$, for any B rank k, we have

$$\|A - A_k\|_F \leq \|A - B\|_F$$

where

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$$

Essentially, this is the $k-SVD$ of A, where we take the top k largest singular values of A along with the singular vectors.

5 Lecture 11: Feb. 10th

summary

5.1 Spectral Graph Theory

Previously, we discussed PCA, SVD, tensor decompositions, which are used as "analytic" methods for exploring data.

Today we'll explore the tie between combinatorial properties of graphs with the linear algebraic structure of matrices.

Let's recall a couple definitions.

Definition (Graph)**.** A graph G is a double of a set of vertices V and edges E, often denoted $G = (V, E)$.

Definition (Adjacency matrix)**.** For any graph G we can associate it with an $n \times n$ matrix A called the **adjacency matrix**, where

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Furthermore, a degree matrix D is defined where

$$D = \begin{pmatrix} \deg(1) & & & \\ & \deg(2) & & \\ & & \ddots & \\ & & & \deg(n) \end{pmatrix}$$

Definition (Laplacian)**.** For any graph G, its **Laplacian** is defined to be

$$L_G = D - A$$

Furthermore, the *normalized Laplacian* of G, denoted

$$\mathcal{L}_G = \frac{1}{d}(D - A)$$

if G is d-regular. More generally,

$$\mathcal{L}_G = D^{-\frac{1}{2}}(D - A)D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$$

We now explore some properties of Laplacians through some examples.

Example 5.1. If G has just a single edge (u, v), then

$$L_G = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \begin{pmatrix} 1 & -1 \end{pmatrix}$$

In general, laplacians compose linearly, so for a general graph G,

$$L_G = \sum_{e=(u,v)\in E} L_{u,v}$$

And to expand, for some vector x, we have

$$x^T L_{\{u,v\}} x = (x_u - x_v)^2$$

Combining the previous two facts, for some arb graph G, we have

$$x^T L_G x = x^T \left(\sum_{e=(u,v)\in E} L_{u,v} \right) x = \sum_{(u,v)\in E} (x_u - x_v)^2$$

Basically, this $x^T L_G x$ thing is pretty useful. Especially in mathematical physics! They even have a symbol *deltaf* for it. Also Ohm's law, the laplacian tells us the induced current!

The claim is that the laplacian is able to capture lots of interesting combinatorial properties of the associated graph.

Lemma 5.1.

For any graph G, L_G is positive semi-definite (non-negative eigenvalues).

Proof. Suppose x is an eigenvector with eigenvalue λ. Then,

$$x^T L_G x = x^T(\lambda x) = \lambda \|x\|_2^2$$

But also,

$$x^T L_G x = \sum_{(u,v)\in E} (x_u - x_v)^2 \geq 0 \quad \Rightarrow \quad \lambda \geq 0$$

□

Lemma 5.2.

For any graph G, if $\vec{1}$ is the all ones vector, then

$$L_G \vec{1} = \vec{0}$$

Proof.

$$L_G \vec{1} = \sum_{(u,v)\in E} L_{u,v} \vec{1} = \sum_{(u,v)\in E} (\vec{1}_u - \vec{1}_v) = 0$$

□

Corollary 5.2.1.

0 is an eigenvalue of L_G.

Let's arrange the eigenvalues of L_G in ascending order, where $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. Recall the multiplicity of an eigenvalue λ is the $\{\#i : \lambda_i = \lambda\}$.

Theorem 5.3.
The multiplicity of 0 *is exactly the number of connected components of* G*.*

Proof. Recall that the smallest eigenvalue of any PSD matrix M is

$$\min_{\|x\|_2=1} x^T M x$$

Let x_1 be argmin of the previous quantity. Then, the second smallest eigenvalue is

$$\min_{\|x\|_2=1 \quad x\perp x_1} x^T M x$$

This process can be perpetually repeated to find the j smallest eigenvalues.

Following this logic, we now claim that a PSD matrix has eigenvalue 0 with multiplicity k iff there exists k orthogonal (nonzero) vectors $x_1, ..., x_k$ such that

$$x_i^T M x_i = 0$$

So it suffices to show that if G has k connected components, there exists $x_1, ..., x_k$ non-zero orthogonal such that

$$x_i^T L_G x_i = 0 \quad \forall i = 1, ..., k$$

and no such collection of $k+1$ vectors can exist.

*****

We now show multiplicity $\geq$ # connected components. Recall

$$x^T L_G x = \sum_{(u,v)\in E} (x_u - x_v)^2$$

Let $S_1, ..., S_k$ be the connected components. Let

$$x_i = \begin{cases} 1 & \text{if } u \in S_i \\ 0 & \text{otherwise} \end{cases}$$

Then,

$$\sum_{i=1}^{k} \sum (u,v) \in E_i (x_u - x_v)^2$$

which implies

$$x_i^T L_G x_i = 0 \quad \text{for } i = 1, ..., k$$

Additionally, these vectors must be orthogonal since they're built from completely different bases.

*****

We now show # connected componetns $\geq$ multiplicity. Suppose that L_G had $k+1$ 0 eigenvalues with associated eigenvectors $x_1, ..., x_{k+1}$.

Then, we have

$$\langle x_i, x_j \rangle = 0 \; \forall i \neq j \quad \|x_i\|_2 = 1 \quad x_i^T L_G x_i = 0$$

For any vector x, we have

$$x^T L_G x = \sum_{i=1}^{k} \sum_{(u,v) \in E_i} (x_u - x_v)^2$$

In general if $x^T L_G x = 0$, then x must be constant on every connected component. This implies that x_i's are really just k-dimensional vectors (compress down by each connected component).

What we would need then, are $k+1$ orthogonal vectors in k dimensions. Clearly a contradiction. $\square$

Woo! Building off of this, nonzero eigenvalues all corresponds to the "connectiveness" of the graph as well.

The heuristic: if eigenvalues are small, then the graph is "barely" connected. And as eigenvalues get bigger, the more connected the graph is.