CS168: Modern Algorithmic Toolbox

Instructor: Greg Valiant; Notes: Adithya Ganesh

May 2, 2018

Contents

1 Lecture 3

Administrative updates:

- Mini project 1: due 11:59pm tomorrow
- Mini project 2: posted tonight (due in 8 days)

Core problem. How can we quickly find similar datapoints?

Two variations on this problem:

- One: given a dataset, search for similar pairs within the dataset.
- Two: given a new datapoint, quickly process the query and find points that are similar (nearest neighbor search problem).

Question. How do we define "similarity?"

Motivation / applications.

- Similarity for e.g. documents, webpages, source code. Search engines, for instance, perform a lot of deduplication to ensure that results are not repeated twice.
- Collaborative filtering (think Amazon / Netflix for recommendations). Idea: compute which individuals are similar, or compute which movies / items are similar.
- Machine learning via nearest neighbor search / similarity.

1.1 Similarity Metrics

Jacard Similarity. This is a notion that applies between sets / multi-sets S and T.

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

In the case of multisets - just count things with redundancy.

Example. Say $S = \{a, b, c, d, e\}$, and $T = \{a, e, f, g\}$, then the Jacard similarity is

$$J(S,T) = \frac{2}{7}.$$

Another context in which we can apply Jacard similarity is to consider the one-hot encoding vector S where S_i represents the number of times i appears.

Then

$$J(S,T) = \frac{\sum_i \min(S_i, T_i)}{\sum_i \max(S_i, T_i)}.$$

Tends to work well in practice especially for sparse data (for example, text and documents).

Euclidean distance. (between vectors in $\mathbb{R}^d$)

$$||x - y||_2 = D_{euc}(x,y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}.$$

One reason this is useful in that its rotationally invariant.

L_1 distance / Manhattan distance.

$$||x - y||_1 = \sum_{i=1}^{d} |x_i - y_i|.$$

Intuition - if walking in a grid, this is the distance no matter how you walk. Also - this is not rotationally invariant. Therefore, if you are ever using L_1 distance, make sure that the basis means something.

More broadly, we can define L_p metrics.

L_p metrics. The L_p distance is defined as

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

Note that there are many other notions of similarity.

- Cosine similarity (the angle between two vectors)
- Edit distance (Hamming)

Note that as $p \to \infty$, this converges to the max over i over the absolute difference in the components.

Note that there is a subtlety in the corner points (depending on how you define the limit).

Picture:

And note that you can define L_p metrics for p non-integer.

How do you decide between L_1 and L_2? You can reason about this by thinking about the Voronoi diagram of the vectors.

Definition. *Given points X in $\mathbb{R}^d$, and some metric D, the Voronoi diagram partitions space into regions (the set of all points that are closest to a single datapoint). Concretely, for $x \in X$, $part(X) = \left\{ y \in \mathbb{R}^d \text{ s. t. } D(x, y) = \min_{x' \in X} D(x', y) \right\}$.*

Story: John Snow (a doctor) figured out that the people who died of cholera were in a Voronoi cell corresponding to an infected well.

You can think about the Voronoi diagram for different similarity metrics. For L_2, they are going to be straight lines. Question: what do the partitions of the Voronoi diagrm look like for L_2 and L?

Area of math devoted to understanding the difference between different metrics: metric embeddings.

Natural question. How do you map one set of points in one metric to another set of points in a different metric, such that the original distances are equal to the new distances?

Concretely - given $x_1, \ldots, x_n \in \mathbb{R}^d$, can we find a function $f : \mathbb{R}^d \to \mathbb{R}^m$ such that $||x_i - x_j||_1 \approx ||f(x_i) - f(x_j)||_2$?

In many cases, the answer is yes, there exists a function that can do this.

For the rest of the class: let's return to the question of how we find similar objects. We will focus on Euclidean distance.

In two dimensions, Voronoi diagrams are straightforward to construct. Things get much harder in higher dimensions.

At a high level, the number of edges that a cell in the Voronoi diagram will have will scale exponentially with the number of dimensions you are in. Even storing the Voronoi diagram in memory will take, exponential space.

Example. $k-d$ trees (space partitioning data structure). Idea: Use a balanced binary tree that partitions space.

The idea is that each edge in our tree will correspond to a partition in space.

(TODO: Insert image).

How much space does it take to store this data structure? At each node, we only need to store the value of each point.

To find closest point to some new y. There are two steps:

- Go down the tree and figure out which region of space y would fall in $(\log N)$ operations.
- Go bac up, check each case.

Question: do we need to jump to other regions? In 1 dimension, we can just return the datapoint that is in that partition (guaranteed to be the closest point to y).

In higher dimensions - we might need to jump to adjacent regions. Going up the tree, we ask - is it possible that there is a point in the next partition that is closest to us than the next?

Rule of thumb. If dimension $d < 20$, works fairly well. This refers to the intrinsic dimensionality of the dataset (for example if the dataset is higher dimensional, but lies on a lower dimensional subspace), or if the number of points $> 2^d$.

k-d tree data structure.

Given a set of points - pick a dimension.

Given a set $S = \left\{x : x \in \mathbb{R}^d\right\}$.

- Pick a dimension / coordinate i.
- Compute the median of $\{x_i\}$.
- Partition: $S_1 = \{x \in S | x_i < m\}$, and $S_2 = \{x_i \geq m\}$.
- Recurse on S_1 and S_2 and store which dimension we are looking at.

How much back and forth do we need to do in the tree? The number of points that we'll need to check is going to be exponential in the dimension. This i sbecause the number of facets in a Voronoi diagram will tend to scale exponential in the dimension.

Runtime:

Logarithmic in the number of points

And exponential in the dimension of the points.

Does it make sense to sort a subset of the data? Yes, but then you have the question of sorting on which dimension.

2 Lecture 4

Review. Last time we talked about $k-d$ trees, which are binary trees that partition space in k dimensions.

The runtime to find a closest point to a new point y is:

$$\log(\#\text{ points}) \cdot \underbrace{\exp(\text{dim})}_{\text{number of partitions to look through}}$$

This is one example (broadly) of a phenomenon known as the "curse of dimensionality." For many geometric problems that we care about - the runtime will scale exponentially in the dimension that we're working with.

The kissing number. How many identical spheres can you place around a sphere such that all of them are touching the center sphere?

For $k = 2$, the kissing number is 6. For $k = 3$, the kissing number is 12. Note that in 5 dimensions, the kissing number is unknown! In general, the kissing number will scale exponentially in d.

Question. Why are proving these results so hard to prove? For certain dimensions, (for example dim = 8, it is fairly easy to show results (because of symmetry). But for other, the best packing strategies we know is based on random packing processes.

Note that sphere packing is a very relevant question to ask. You can think of radio stations and wanting to pack them as close together as possible without interference.

Problem: reduce the dimensionality while approximately preserving all pairwise distance.

Suppose you have $x_1, \ldots, x_n \in \mathbb{R}^d$. Suppose you have $y \in \mathbb{R}^d$, and you want to find the closest point in X. What are our options?

- Use $k-d$ tree, and pay $\log(\#\text{ points}) \cdot \exp(\text{dim})$.
- Brute force: $O(nd)$
- Dimensionality reduction + brute force (developed in this lecture).

We will describe a general recipe to perform dimensionality reduction, and this will apply for any similarity metric.

- Find easy / fast way to preserve distances in expectation.
- Repeat it a few times (independent repetitions; this will take you from being good in expectation to being good most of the time).

2.1 Dimensionality reduction for Jacard similarity

Consider Jacard similarity, defined earlier. We develop the "MinHash" technique that we can use to reduce dimesionality.

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

- Pick a uniformly random ordering of the universe U.
- Map set S to $f(S) =$ "min" element of S. w d

This is based on the following claim.

Claim. *For any sets S and T, we have*

$$\Pr(f(S) = f(T)) = J(S,T).$$

Proof. $f(S) = f(T)$ if and only if the first element is in the intersection. The denominator is the union. The result is clear from here. □

Concretely, suppose we repeat k times. This requires us to pick k random orderings. And then we can ask what fraction of k will satsify $f(T) = f(S)$?

What is the error relative to the true similarity? It is approximately $\frac{1}{\sqrt{k}}$.

Suppose we have independent random variables $X_1, \ldots, X_k \sim \text{Ber}(p)$. We want to know what is the standard deviation of their average?

$$\text{Var}\left(\frac{\sum_{i=1}^k X_i}{k}\right).$$

$$= \frac{1}{k^2}\,\text{Var}\left(\sum_{i=1}^k X_i\right)$$

$$= \frac{1}{k^2} \cdot k\,\text{Var}(X_1) \geq \frac{1}{k}$$

Hence the standard deviation is at most $\frac{1}{\sqrt{k}}$. Note: this is true in general; this is how you interpret the statistical significance of election polls.

2.2 Dimensionality reduction for Euclidean distance

- Choose a random d dimensional vector $r = (r_1, \ldots, r_d)$.

$$f_r(v) =< v, r >= \sum_{i=1}^d v_i r_i.$$

It turns out that if you pick two angles on the sphere, it doesn't end up being uniform! Similarity, if you uniformly pick the coordinates, the resulting distribution is not uniform. Instead, we will pick $r_i \sim \mathcal{N}(0,1)$, resulting in a uniform distribution that is rotationally invariant.

Fact. *Let $X \sim \mathcal{N}(\mu_1, \sigma_1^2)$, $Y \sim \mathcal{N}(\mu_2, \sigma_2^2)$. Then:*

$$X + Y \sim \mathcal{N}(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2).$$

Note that this is fairly unique to Gaussians - this isn't true for most distributions.

Claim. *For any two vectors $x, y \in \mathbb{R}^d$, we have*

$$\mathbb{E}[(f_r(x) - f_r(y))^2] = \mathbb{E}[||x - y||_2^2]$$

Proof. Note that

$$\begin{aligned}
\mathbb{E}[(f_r(x) - f_r(y))^2] &= \mathbb{E}\left\{\left(\sum_{i=1}^d r_i x_i - \sum_{i=1}^d r_i y_i\right)^2\right\} \\
&= \mathbb{E}\left\{\left(\sum_{i=1}^d r_i (x_i - y_i)\right)^2\right\} \\
&= \mathbb{E}\left\{\mathcal{N}(0, \sum_i (x_i - y_i)^2)\right\} \\
&= \mathrm{Var}\left\{\mathcal{N}(0, \sum_i (x_i - y_i)^2)\right\} \\
&= \sum_{i=1}^n (x_i - y_i)^2.
\end{aligned}$$

□

Fact. *If you repeat $l = \frac{\log n}{\epsilon^2}$ times, then with high probability, all $\binom{n}{2}$ pairwise distances are preserved to within a factor of $1 \pm \epsilon$.*

This transformation is referred to as the Johnson-Lindenstrauss transformation.

3 Lecture 5: Generalization

General question. How much data is enough?

Broadly, we can think about two different types of data analysis.

- Understanding dataset
- Goal of extrapolating beyond dataset (inference)

Consider the binary classification setting. We can broadly define it as follows:

- Suppose we have some datapoints $x_1, \ldots, x_n \in \mathbb{R}^d$.
- Known distribution D on $\mathbb{R}^d$.
- Ground truth label function $f : \mathbb{R}^d \to \{0, 1\}$.

Problem. Given $x_1, \ldots, x_n$ drawn independently from D, and labels $f(x_1), \ldots, f(x_n)$, our goal is to output $g : \mathbb{R}^d \to \{0, 1\}$ such that "$g \approx f$".

Namely, we want the generalization error to be low, defined this way:

$$\text{generalization error}(g) = \Pr_{x \sim D} \left[g(x) \neq f(x) \right].$$

Can also define the training error as the fraction of the training points on which g disagrees with the true labelling.

Claim. *For any function g, the expected training error is equal to the generalization error.* [1]

Question. Suppose we find g with training error 0. When does this imply that the generalization error is small?

Factors that influence this question:

- Amount of data (how faithful is the sample?)
- The complexity of the function (# of functions considered).
- Algorithm used to find g

This is often succinctly phrased as "does g generalize?" If answer is no, this implies that you've "overfit" the data.

First, we'll consider the "well-separated finite setting." Here, we will make two enormous assumptions. Assume that:

- The ground truth labelling function $f \in S = \{f_1, f_2, \ldots f_k\}$. That is, f belongs to a set of functions with k elements which is finite.
- All of these functions are well separated. No function in the class is similar to f. For all $f_i \in S$ with $f_i \neq f$, the generalization error of $f_i > \epsilon$. Note that this is sort of a silly assumption, but we will drop both.

Naive "algorithm:" return any g in our set S that has training error 0.
Theorem. *Given assumptions 1 and 2, if the number of datapoints $n > \frac{1}{\epsilon}(\log k + \log \frac{1}{\delta})$, then, with probability at least $1 - \delta$, g will generalize.*

Some comments: logarithmic function in k and $1/\delta$ is good, but inverse linear function in $\frac{1}{\epsilon}$ is kind of bad.

[1] To be clear: "training error" in this setting refers to datapoints that the functino has not necessarily been trained on. It might be clearer to say: the expected "empirical error" converges to the generalization error.

Proof. We will prove this in two parts.

- First, we will analyze the probability that we are "tricked" by a bad f_i.
- Next, we will union bound over all bad f_i's.

Consider a bad function f_i. The probability that we are tricked by this function is

$$\begin{aligned}\Pr\{\text{TrainingError}(f_i) = 0\} &= \prod_{j=1}^{n} \Pr_{x_j \sim D}(f_i(x_j) = f(x_j)) \\ &\leq (1-\epsilon)^n \\ &< e^{-\epsilon n}.\end{aligned}$$

The last inequality follows from the inequality $1 + x < e^x$. Proof from Taylor series / plot.

There are at most k "bad" functions. Hence we can apply a union bound, to obtain

$$\delta = \Pr(\text{output bad function}) \leq ke^{-\epsilon n}.$$

Now, the desired result directly follows from solving for failure probability $\frac{1}{\delta}$.

Note that we don't *really* need assumption 2. □

Results of this form are generally referred to as being in the "PAC" framework (probably appropximately correct).

Example. *Consider the set of linear classifiers in $\mathbb{R}^d$. This is defined by a vector $\mathbf{a} = (a_1, a_2, \ldots, a_d)$. Then the classifier is just*

$$f_a(x) = \text{sign}\left(\sum_{i=1}^{d} \mathbf{a} \cdot \mathbf{x}\right).$$

Intuition. Note that the vector $\mathbf{a}$ will be the normal vector to the hyperplane separating datapoints.

Also note that this is very general because if we don't want a hyperplane through the origin, we can just add another feature.

Claim. If we consider the set of linear classifiers, then the error still satisfies the generalization bound, with a few minor tweaks.

Theorem. *For linear classifiers, if the number of datapoints $n > \frac{C}{\epsilon}(d + \log\frac{1}{\delta})$, then, with probability $1 - \delta$, g will generalize.*

The intuition behind the proof here is that there are an exponential number of "important" directions in d dimensional space.

Important questions to consider:

- How do we find the optimal g?

- What if no function in S has error 0?
- What if you have fewer than the threshold of datapoints, what can you do?

4 Lecture 6: Regularization

Note on last part - it is very open ended, at the cutting edge of machine learning research (but don't feel obliged to write pages of analysis).

Punchline from last class. If you a have a set k different functions $\{f_1, \ldots, f_k\}$, then the "best" one will generalize if $n > O(\log k)$. If we are classifying in d dimensions - we can approximate by set of $\exp(d)$ linear functions. If $n > d$, expect generalization.

Regularization. A way to express a set of preferences over models. Such a scheme that will take both performance on training data as well as these preferences into account.

Example. *For example, we can consider L_2-regularized least squares. Let $x_1, \ldots, x_n \in \mathbb{R}^d$, and $y_1, \ldots, y_n \in \mathbb{R}$. In this setting, we want to minimize the following objective:*

$$\min_a f(a) = \sum_{i=1}^{n} (\langle x_i, a \rangle - y_i)^2 + \underbrace{\lambda ||a||_2^2}_{\textit{regularization term}} .$$

There are two broad types of regularization, explicit or implicit:

- Explicit regularization (e.g. L_2 regularization, preferring sparse vectors).
- Implicit regularization (algorithm itself has "preferences").

Key question. Why should we regularize; why wouldn't we just return the empirical risk minimizer?

Perspective. You always want roughly $n \approx d$, where generalization might not hold. If you have $n = 1,000,000$, then you want $d \approx 1,000,000$. Otherwise - it's sort of a "waste"; if d is truly 1000 in your dataset, try to construct additional features to learn a model in 1,000,000 dimensional space.

How should we construct additional features? Here are two approaches.

- *Polynomial embedding.* (For example - quadratic embedding.

  $$x = (x_1, x_2, \ldots x_d) \to f(x) = (x_1, \ldots, x_d, x_1^2, x_1x_2, x_1x_3, \ldots, x_d^2, 1) \in \mathbb{R}^{2d+1+\binom{d}{2}}.$$

  One simple setting in which you need quadratic features to fit a classifier is when you are fitting a circular decision boundary.

- *Random projection + non-linearity.* You really need the non-linearity, since otherwise you'll just be learning another linear function. Can choose $\sqrt{x}, x^2, \sigma(x)$, or most other "nice" nonlinearities (since they all roughly have similar properties).

Downsides of adding new features:

- One objection could be that working with $\approx d^2$ dimensional points is annoying. But this isn't an issue: you can "implicitly" work over the embedded points without computing the embedding. The area of math devoted to this is referred to as "kernelization" (usually covered in the context of SVMs).
- Real issue: if you need d^2 features, you generally need much more data.

Rule of thumb: if the coordinates actually have signifiance, the polynomial embedding preserves interpretability.

How should you think about regularization? There are two views: the Bayesian view, and the frequentist view.

- *Bayesian view.* Assume that the true model is drawn from some known "prior" distribution. This allows us to evaluate the "likelihood" / probability of a given candidate model.
- *Frequentist view.* Goal: argue that if true model has "nice structure," then can find it.

Bayesian approach to regularization. ("Gaussian prior") Suppose we have $x_1, \ldots, x_n \in \mathbb{R}^d$, assume that the true label $a^* \in \mathbb{R}^d$ is drawn by choosing each coordinate independently from $\mathcal{N}(0, 1)$.

Now, suppose that each label is set as $y_i = \langle x_i, a^* \rangle + z_i$, where noise $z_i \sim \mathcal{N}(0, \sigma^2)$. Now, given $x_1, \ldots, x_n, y_1, \ldots, y_n$, ask

$$\text{Likelihood}(a) = \Pr(a) \Pr(\text{data}|a).$$

Because we made strong assumptions on the label distributions, we can directly compute these probabilities. Hence

$$\begin{aligned}\text{Likelihood}(a) &= \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi}} \exp(-a_i^2/2) \prod_{i=1}^{n} \exp(-(\langle x_i, a \rangle - y_i)^2/2\sigma^2) \\ &\propto \exp(-||a||_2^2/2 - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (\langle x_i, a \rangle - y_i)^2)\end{aligned}$$

Maximum likelihood of a is equivalent to minimizing:

$$\sum_{i=1}^{n} (\langle x_i, a \rangle - y_i)^2 + 2\sigma^2 ||a||_2^2.$$

This derivation even told us how to set the regularization constant. Should be 2 times the variance of the noise.

Frequentist approach to sparsity / regularization. Consider a model a^* that is s-sparse (i.e. there are s nonzero coordinates).

Question: why do we care about sparse models?

- One answer is that lots of models are actually sparse. Think of the laws of physics.
- Other view: maybe the world is sloppy, and the best model might not be sparse. But what's most helpful for interpretability is fitting a sparse model.

Question: can we build a regularizer that lets us selectively find sparse models? The obvious choice is

$$f(a) = \sum_{i=1}^{n} \left(\langle x_i, a\rangle - y_i\right)^2 + \lambda ||a||_0,$$

where we are using the "0-norm" that computes sparsity.

Claim. If $n > O(s\log(d))$ then the sparsest model that fits the data is "correct."

The number of s-sparse d-dimensional function is just $\binom{d}{s}$, and there are about $\exp(s)$ sparse functions. So there are approximately $d^s \exp(s)$ sparse functions, and we need datapoints around the logarithm of this.

The problem with using sparse models is that it is not differentiable (so finding the minimizer is NP-hard).

So, we can note the following:

- l_0 regularization is great, but computationally intractable.
- Idea: use l_1 regularization as proxy for l_0.

Miraculously, the claim from before still holds for l_1 regularization (proved in the early 2000's, Candes in the stat / math department here).

5 Lecture 10

Definition. *A $n_1 \times n_2 \times \cdots \times n_k$ k-tensor is a set of $n_1 n_2 \ldots n_k$ numbers which interprets as being arranged in a k-dimensional hypercube.*

A 2-tensor is simply a matrix, with $A_{i,j}$ referring to the i,jth entry. You can refer to a specific element of a k-tensor via $A_{i_1,i_2,\ldots,i_k}$.

Note that tensors are very useful in physics, where they are viewed with more geometric intuition.

We can define a notion of rank of a tensor. Note that a matrix M has rank r if it can be written as $M = UV^T$, where U has r columns, and V has r columns.

We can write

$$M = \sum_{i=1}^{r} u_i v_i^T.$$

Here is an informal definition of tensor rank.

- A tensor is rank 1 if all rows of all matrices are multiples of each other.
- A tensor has rank k if it can be written as a sum of k rank 1 tensors.

We can define a tensor product as follows

Definition. *Given vectors $v_1, v_2, \ldots, v_k$ of lengths $n_1, \ldots, n_k$, the tensor product is denoted $v_1 \otimes v_2 \otimes \cdots \otimes v_k$ is the $n_1 \times n_2 \times n_k$ k-tensor A with entry*

$$A_{i_1,i_2,\ldots,i_k} = v_1(i_1) \cdot v_2(i_2) \cdots v_k(i_k).$$

Example. *For example, let*

$$v_1 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, v_2 = \begin{pmatrix} -1 \\ 1 \end{pmatrix}, v_3 = \begin{pmatrix} 10 \\ 20 \end{pmatrix}.$$

Then $v_1 \times v_2 \times v_3$ is a $3 \times 2 \times 2$ 3-tensor, that can be thought of as a stack of two 3×2 matrices:

$$M_1 = \begin{pmatrix} -10 & 10 \\ -20 & 20 \\ -30 & 30 \end{pmatrix}, M_2 = \begin{pmatrix} -20 & 20 \\ -40 & 40 \\ -60 & 60 \end{pmatrix}.$$

More formally, we can define the rank of a tensor as follows.
Definition. *A 3-tensor A has rank r if there exists 3 sets of r vectors, $u_1, \ldots, u_r, v_1, \ldots, v_r$ and $w_1, \ldots, w_r$ such that*

$$A = \sum_{i=1}^{r} u_i \otimes v_i \otimes w_i.$$

Let's go back to the motivation for SVD (the "Spearman experiment").

Suppose there are 1000 students. Construct a 1000×20 matrix where you administer 20 different school tests. He noticed that this is approximated by a rank 2 matrix. Namely, it is approximated by (1000×2) multiplied by 2×20. Question: to what extent is this decomposition unique?

If we can write

$$M = AB,$$

we can also write

$$M = (AX)(X^{-1}B).$$

Let's examine the differences between matrices and tensors.

- For matrices, the best rank-k approximation can be found by iteratively finding the best rank-1 approximation, and then subtracting it off. If uv^T is the best rank 1 approximation of M, then $rank(M - uv^T) = rank(M) - 1$.

  For k-tensors with $k \geq 3$, this is not always true. If $u \otimes v \otimes w$ is the best rank 1 approximation of 3-tensor A, it is possible that $rank(A - u \otimes v \otimes w) > rank(A)$.

- For matrices with entries in $\mathbb{R}$, there is no point in looking for a low-rank decomposition that involves complex numbers, because of $rank_{\mathbb{R}}(M) = rank_{\mathbb{C}}(M)$. For k-tensors, this is not always the case.
- With probability 1, if you pick the entries of an $n \times n \times n$ 3-tensor independently at random from the interval $[0, 1]$, the rank will be on the order of n^2. But we don't know how to describe any construction of $n \times n \times n$ tensors whose rank is greater than $n^{1.1}$ for all n.
- Computing the rank of matrices is easy (via SVD). Computing the rank of 3-tensors is NP-hard.
- If the rank of a 3-tensor is sufficiently small, then its rank can be efficiently computed, its low rank represnetation is unique, and can be efficiently recovered.

Theorem. *(Amazing theorem of tensors) Consider a 3-tensor A which has rank k. It can be writen as*

$$A = \sum_{i=1}^{k} u_i \otimes v_i \otimes w_i.$$

Claim: if $\{u_1, \ldots, u_k\}$, $\{v_1, \ldots, v_k\}$ are linearly independent, then can efficiently recover this factorization.

We can now discuss the tensor decomposition algorithm (Jenrich's Algorithm). Given an $n \times n \times n$ tensor $A = \sum_{i=1}^{k} u_i \otimes v_i \otimes w_i$ with $(u_1, \ldots, u_k), (v_1, \ldots, v_k), (w_1, \ldots, w_k)$ linearly independent, the following algorithm will output the lists of u's, v's, and w's.

- Choose random unit vectors $x, y \in \mathbb{R}^n$.
- Define the $n \times n$ matrices A_x, A_y, where A_x is defined as follows. Consider A as a stack of n $n \times n$ matrices. Let A_x be the weighted sum of these matrices, where the weight given to the ith matrix is x_i. Define A_y analogously.
- Compute the eigen-decompositions of $A_x A_y^{-1} = QSQ^{-1}$ and $A_x^{-1} A_y = Y^{-1} T Y^T$.
- With probability 1, the entries of diagonal matrix S will be unique, and will be inverses o the entries of T. The vectors $u_1, \ldots, u_k$ are the columns of Q corresponding to nonzero eigenvalues, and the vectors $v_1, \ldots, v_k$ will be the columns of Y, where v_i corresponds to the reciprocal of the eigenvalue to which u_i corresponds.
- Given the u_i's and the v_i's, we can now solve a linear system to find the w_i's, or imagine rotating the whole tensor A and repeating the algorithm to recover the w's.

Why does this work?

Claim. We have that

$$A_x = \sum_{i=1}^{k} \langle w_i, x \rangle u_i v_i^T; \qquad A_y = \sum_{i=1}^{k} \langle w_i, y \rangle u_i v_i^T.$$

We can see this using an SVD / eigendecomposition argument (see lecture notes).

Quick suggestions on where we might encounter tensors:

- Spearman experiment setting.
- NLP setting, where you have a tri-occurence 3 tensor for e.g. words.
- Social network tensor in terms of groups, not just pairs.
- Moment tensor. If you have n-dimensional data, then the covariance is $n \times n$. You can compute third-order and fourth-order moments fairly naturally.

6 Things to review

1. Review proof of simple PAC / generalization bound.
2. SVD intuition.

7 Key ideas