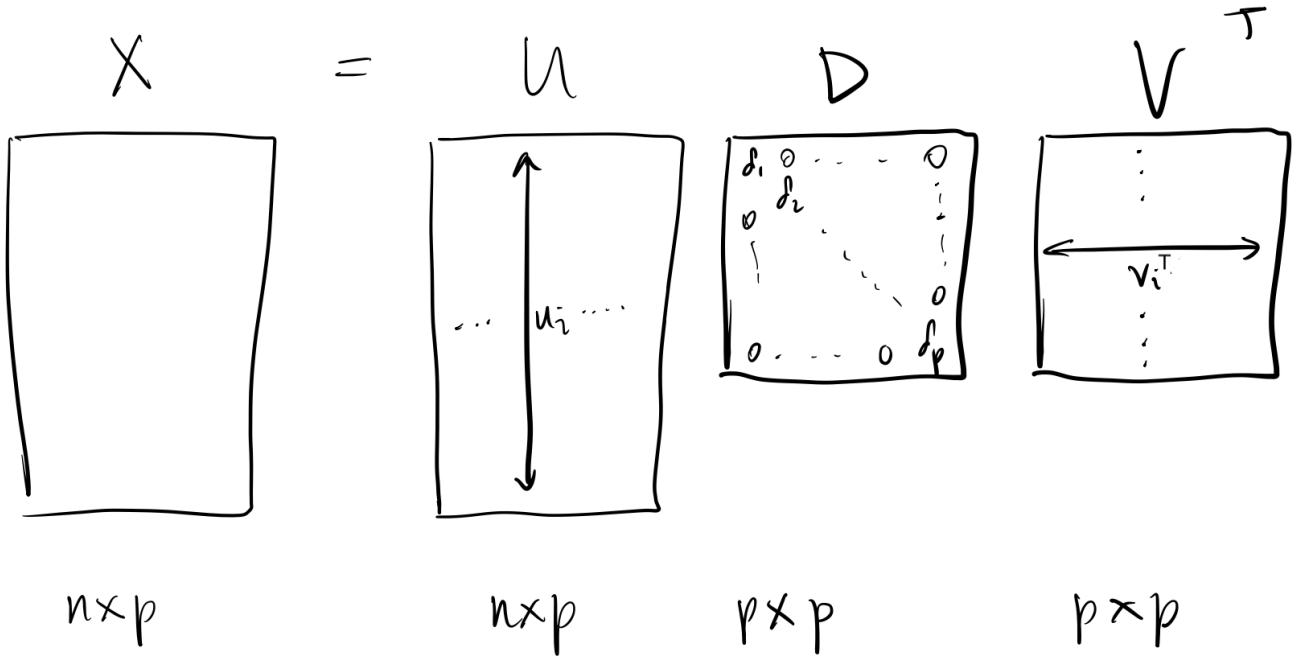
Advanced Machine Learning
Lecture Notes

Spring 2026

Cody Carroll

1 Singular Value Decomposition

Suppose we have a matrix X with n rows & p columns (without loss of generality, assume $n \geq p$). It turns out that every matrix X has a *singular value decomposition*:



where the following statements are true:

1. The columns of U: $u_1, ..., u_p$ are orthonormal vectors and are called the *left singular vectors* of X.
2. $U^T U = I_p$
3. The rows of V: $v_1, ..., v_p$ are orthonormal vectors and are called the *right singular vectors* of X.
4. $V^T V = I_p$ (i.e. the rows of V: $v_1, ..., v_p$ are orthonormal vectors)
5. The diagonal entries of D, $\{\delta_1, \ldots, \delta_p\}$, are nonnegative & called the *singular values* of X. Without loss of generality we can order them such that they are always nonincreasing:

$$\delta_1 \geq \delta_2 \geq \ldots \geq \delta_p \geq 0.$$

Note: If $n < p$, an SVD still exists, but now U is square and V is not.

Alternative form of the SVD:

If we carry out the matrix multiplication, we can formulate the decomposition as:

$$X = \sum_{i=1}^{d} \delta_i u_i v_i^T$$

Things to notice:

- We have expressed the design matrix X as a weighted sum of rank 1 matrices: $u_1 v_1^T, ..., u_p v_p^T$ with weights equal to the singular values, $\{\delta_1, \ldots, \delta_p\}$.
- Some of the singular values could be zero! If there are r nonzero singular values, then rank$(X) = r$.
- Recall that when X is a design matrix, then its rank is the dimension of the space in which variation of its data actually exists! If all the sample data points $x_1, ..., x_n$ (defined by the rows of the design matrix X) lie on a line, there is only one nonzero singular value. If all the sample data points lie on a plane, there are only two nonzero singular values. In general, if all of the points span a subspace of dimension r, there are r nonzero singular values.

Practical Question:

How do I find out what $U, D,$ & V are?
Consider $X^T X$:

$$X^T X = (UDV^T)^T (UDV^T) = V(D^T D)V^T$$

$$= V \begin{pmatrix} \delta_1^2 & 0 & \cdots & 0 \\ 0 & \delta_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \delta_p^2 \end{pmatrix} V^T$$

Does this structure/factorization look familiar?
Claim: V is the matrix of eigenvectors of $X^T X$ and $\delta_1^2, \ldots, \delta_p^2$ are the corresponding eigenvalues.
Proof:

$$X^T X v_1 = V(D^T D)V^T v_1$$

$$= VD^2 \begin{pmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_p^T \end{pmatrix} v_1$$

$$= VD^2 \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

$$= \begin{pmatrix} | & | & & | \\ v_1 & v_2 & \cdots & v_p \\ | & | & & | \end{pmatrix} \begin{pmatrix} \delta_1^2 & 0 & \cdots & 0 \\ 0 & \delta_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \delta_p^2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \delta_1^2 v_1$$

Same argument holds for $v_2, \ldots, v_p$.

Similar claim:

$$XX^T \text{ has an eigendecomposition } UD^2U^T.$$

Why do we care about the SVD? A few reasons:

1. Analysis using the SVD can lend insights and different perspectives into other ML methods (e.g. regression)

2. SVD can help with dimension reduction
   $\implies$ has a close tie to PCA (more on this later, but a sneak preview: row i of UD describes the coordinates of the cenetered sample point x_i^C in principal component space)

Geometric Interpretation of the SVD

Suppose we know that for an arbitrary vector w, we know $Xw = z$; i.e. multiplying w by X maps it to another vector z. Recall that orthonormal matrices represent rotation operations & diagonal matrices correspond to scaling operations. So if we break X down into its SVD components, we can consider the linear transformation induced by applying the matrix X on an arbitrary vector w as three consecutive transformations:

$$Xw = UDV^Tw$$

1. First multiplication: multiplication by V^T rotates w onto a new vector; call this $V^Tw = \tilde{w}$.

$$Xw = UD\tilde{w}$$

2. Second multiplication: multiplication by the diagonal matrix D rescales axes by the singular values, but does not change the direction of the vector. Call the transformed vector after this multiplication $D\tilde{w} = \tilde{\tilde{w}}$

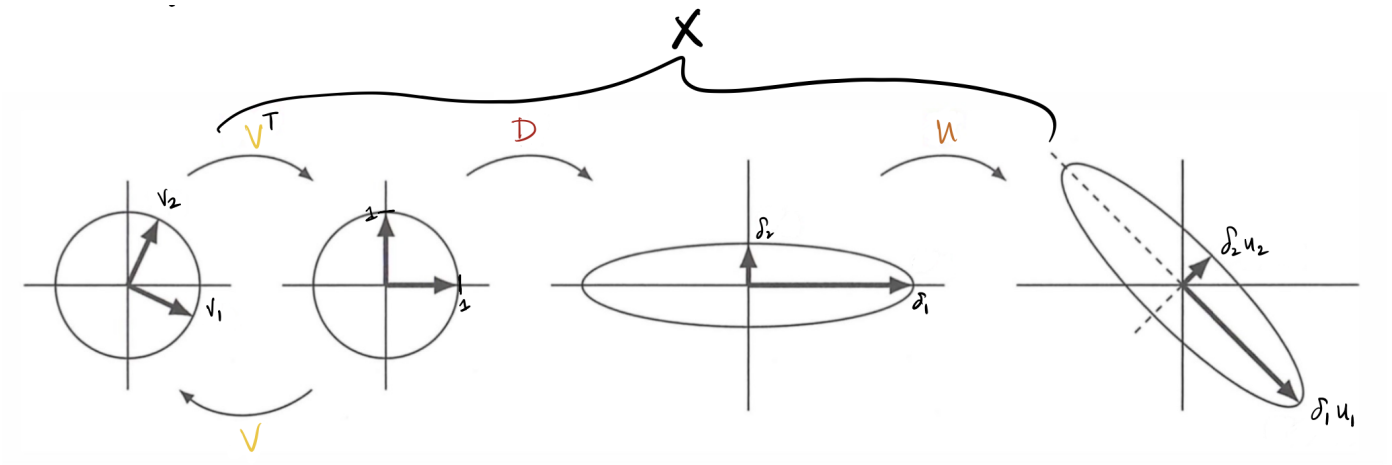
$$Xw = U\tilde{\tilde{w}}$$

(3) Second multiplication: multiplication by U rotates $\tilde{\tilde{w}}$ again; call this final vector z.

$$Xw = U\tilde{\tilde{w}} = z$$

A nice way of visualizing this is to consider transforming the basis vectors of V, i.e. choose $w = v_i$ for any $i = 1, ..., p$.

EX:



Intuition Building via SVD

Ex. OLS through SVD:

Consider the OLS setup

$$y = X\beta + \epsilon, \quad epsilon \sim N(0, \sigma^2 I_n).$$

Derive the OLS estimator $\hat{\beta}$ in terms of the SVD of X. You can assume X has full rank.

Sol:

Normal Eq: $(X^T X)\hat{\beta} = X^T y$

$$\implies \hat{\beta} = (X^T X)^{-1} X^T y$$

$$= (VD^2V^T)^{-1}(UDV^T)^T y$$

$$= (VD^2V^T)^{-1}(VDU^T)y$$

$$= VD^{-2}V^T VDU^T y$$

$$= VD^{-2}DU^T y$$

$$= VD^{-1}U^T y$$

How does this relate to multicollinearity & instability in $\hat{\beta}$?

Notice: $\hat{\beta} = VD^{-1}U^T y = \sum_{j=1}^{p} \delta_j^{-1}(u_j^T y)v_j$
so if δ_j is very close to zero, small changes in δ_j explode and result in very large changes in $\hat{\beta}$.

Question: In regression, how could regularization (e.g. ridge/lasso) stabilize the OLS estimate?

Exercise: For a given penalty parameter λ, calculate the $\mathcal{L}^2-$regularized (i.e. ridge) estimate $\hat{\beta}_\lambda$ in terms of the SVD of X.

Dimension Reduction via SVD

Idea: If at some point k, the remaining singular values, $\{\delta_j\}_{j>k}$, are very small in magnitude, their relative contribution to the weighted sum is very small:

$$X = \delta_1 u_1 v_1^T + \cdots + \delta_k u_k v_k^T + \delta_{k+1} u_{k+1} v_{k+1}^T + \cdots + \delta_p u_p v_p^T$$

If they aren't adding much, **who needs them?**

Let's zero them out.

Doing so results in a *low rank/rank-k approximation* of the data matrix X:

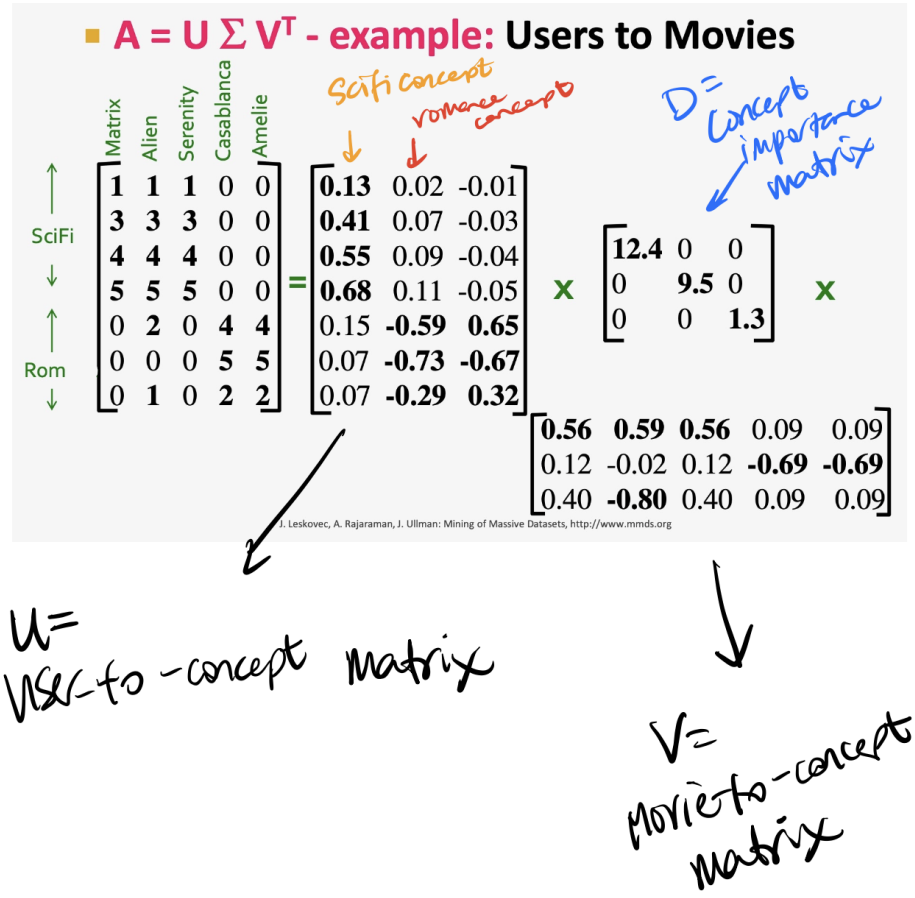
$$X \approx \delta_1 u_1 v_1^T + \cdots + \delta_k u_k v_k^T \quad (+0 + \cdots + 0)$$

$$= \underbrace{[u_1 \cdots u_k]}_{n \times k} \underbrace{\begin{pmatrix} \delta_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \delta_k \end{pmatrix}}_{k \times k} \underbrace{\begin{pmatrix} v_1^T \\ \vdots \\ v_k^T \end{pmatrix}}_{k \times p}$$

Call these truncated matrices U_k, D_k, V_k.

Instead of storing $n \times p$ values, now I only need to store $(n \times k) + (k \times k) + (k \times p) = (n + k + p)k$. This can matter a lot for high dimensional data, i.e. when p is very large!!

Example: Preview of Application to Recommender Systems



Reading Assignment:

Chapter 11.3 of *Mining of Massive Datasets*.

Principal Components Analysis

Suppose I observe a dataset consisting of n observations of p features:

$$\{(x_{i1}, x_{i2}, \ldots, x_{ip})\}_{i=1}^{n}.$$

We can arrange these in an $n \times p$ design matrix:

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix} = \begin{bmatrix} - & x_1 & - \\ - & x_2 & - \\ & \vdots & \\ - & x_n & - \end{bmatrix}.$$

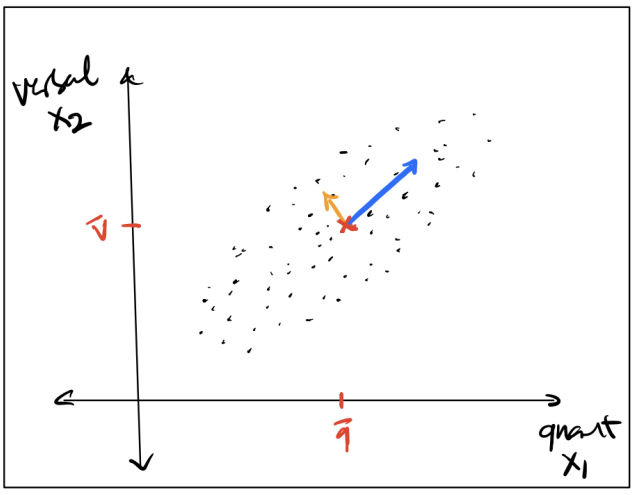
Let's call the i^{th} row $x_i \in \mathbb{R}^p$, & let $\bar{x}_j \in \mathbb{R}$ denote the mean of the j^{th} column:

$$\bar{x}_j = \frac{1}{n} \sum_{i=1}^{n} x_{ij}.$$

Then let $\bar{x} \in \mathbb{R}^p$ be the column vector w/ elements $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_p)^T$. The main idea of principal components analysis (PCA) is that sometimes the coordinate system we get from our features, i.e. the original $(x_1, x_2, \ldots, x_p)$-axes, is not the coordinate system which agrees or aligns most with the natural variation in our data.

Here's a simple example in $\mathbb{R}^2$:

Ex: Suppose n children took a standardized exam which had a quantitative subscore x_1, & a verbal subscore x_2. The data $\{(x_{i1}, x_{i2})\}_{i=1}^{n}$ are distributed roughly like this:



Despite the data coming to us in the form of quant & verbal scores, the directions in which the data vary seem to indicate other unobserved factors or "components." In this example, we might call them something like:

1) general aptitude as measured by the test &
2) the difference between quantitative & verbal aptitude.

Steps for Conducting PCA

Principal components analysis attempts to find these directions/new coordinates through the following steps:

1. Center each column vector to create the centered matrix

$$X^c = \{x_{ij}^c\}_{i=1,j=1}^{n,p}$$

where $x_{ij}^c = x_{ij} - \bar{x}_j$

Visually, this corresponds to moving the origin to the mean point of the data cloud. (The red X.)

2. Calculate the sample covariance matrix

$$S = \frac{1}{n-1} X^{cT} X^c.$$

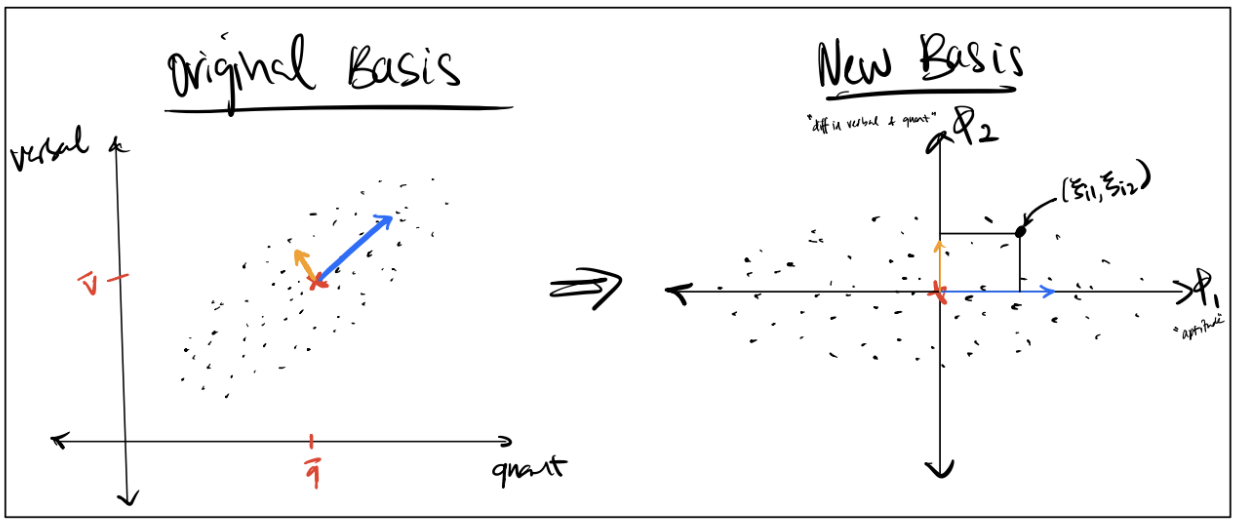
Notice this is a symmetric, positive semidefinite matrix, so we can...

3. Calculate the eigendecomposition:

$$S = \Phi \Lambda \Phi^T \quad \text{(AKA spectral decomposition)}.$$

The cols of $\Phi = [\phi_1 \ \phi_2 \ \dots \ \phi_p]$ give us the unit vectors in the directions of our new coordinate system. Here Λ is a diagonal matrix of the eigenvalues, that is,

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_p \end{bmatrix}.$$



4. Once the new directions/unit vectors $\phi_1, \ldots, \phi_p$ have been chosen, how can we compute the coordinates of a datapoint x_i with respect to our new basis? In other words, how can we find coefficients $\xi_{i1}, \ldots, \xi_{ip}$ such that

$$x_i \;=\; \bar{x} + \xi_{i1}\phi_1 + ... + \xi_{ip}\phi_p?$$

We can compute these coefficients using *Fourier's trick*. Centering and taking the dot product of both sides of the equation,

$$x_i - \bar{x} \;=\; \xi_{i1}\,\phi_1 \;+\; \cdots \;+\; \xi_{ip}\,\phi_p \tag{1}$$

with the vector ϕ_1, we obtain the coordinate in the first direction,

$$\langle x_i - \bar{x}, \phi_1 \rangle \;=\; \xi_{i1}\,\underbrace{\langle \phi_1, \phi_1 \rangle}_{=1} + \xi_{i2}\,\underbrace{\langle \phi_2, \phi_1 \rangle}_{=0} + \;\cdots\; + \;\xi_{ip}\,\underbrace{\langle \phi_p, \phi_1 \rangle}_{=0} \;\implies\; \xi_{i1} = \langle x_i - \bar{x}, \phi_1 \rangle,$$

and so on for $\xi_{i2} = \langle x_i - \bar{x}, \phi_2 \rangle,\ \ldots\ ,\ \xi_{ip} = \langle x_i - \bar{x}, \phi_p \rangle, \qquad i = 1, ..., n.$

Facts about PCA:

1. The coordinates of the i^{th} data point in the new basis are given by the inner product of the i^{th} row of X^c (i.e. $x_i^C \in \mathbb{R}^p$)) & the directions $\phi_1, \phi_2, \ldots, \phi_p$:

$$\xi_{i1} = \langle x_i - \bar{x}, \phi_1 \rangle$$

$$\vdots$$

$$\xi_{ip} = \langle x_i - \bar{x}, \phi_p \rangle$$

These are called the *principal component scores.*
They express the original data in terms of the new basis:

$$x_i = \bar{x} + \xi_{i1}\phi_1 + \xi_{i2}\phi_2 + \cdots + \xi_{ip}\phi_p$$

2. The sample variance of the j^{th} PC score is equal to the j^{th} eigenvalue of S:

$$\frac{1}{n-1}\sum_{i=1}^{n} \xi_{ij}^2 = \lambda_j, \quad \forall j = 1, \ldots, p.$$

3. Because the spectral decomposition arranges the eigenvalues along the diagonal in decreasing order, $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_p$, the basis vector ϕ_1 corresponds to the direction in which the data varies the most, followed by ϕ_2, then ϕ_3, & so on.

4. If we perform SVD on $X^c = UDV^T$, then the columns of V are the same as the eigenvectors of S. i.e. $\phi_j = v_j$ & $j = 1, \ldots, p$, and the rows of UD correspond to the coordinates of the original data points projected into principal component space.

5. The "total variation" of X (the summed variance of each of the p features) is equal to the trace of S:

$$\sum_{j=1}^{p}\left[\frac{1}{n-1}\sum_{i=1}^{n}(x_{ij} - \bar{x}_{\cdot j})^2\right] = \mathsf{tr}(S) = \lambda_1 + \lambda_2 + \cdots + \lambda_p,$$

& the ratio

$$\frac{\lambda_k}{\lambda_1 + \lambda_2 + \cdots + \lambda_p}$$

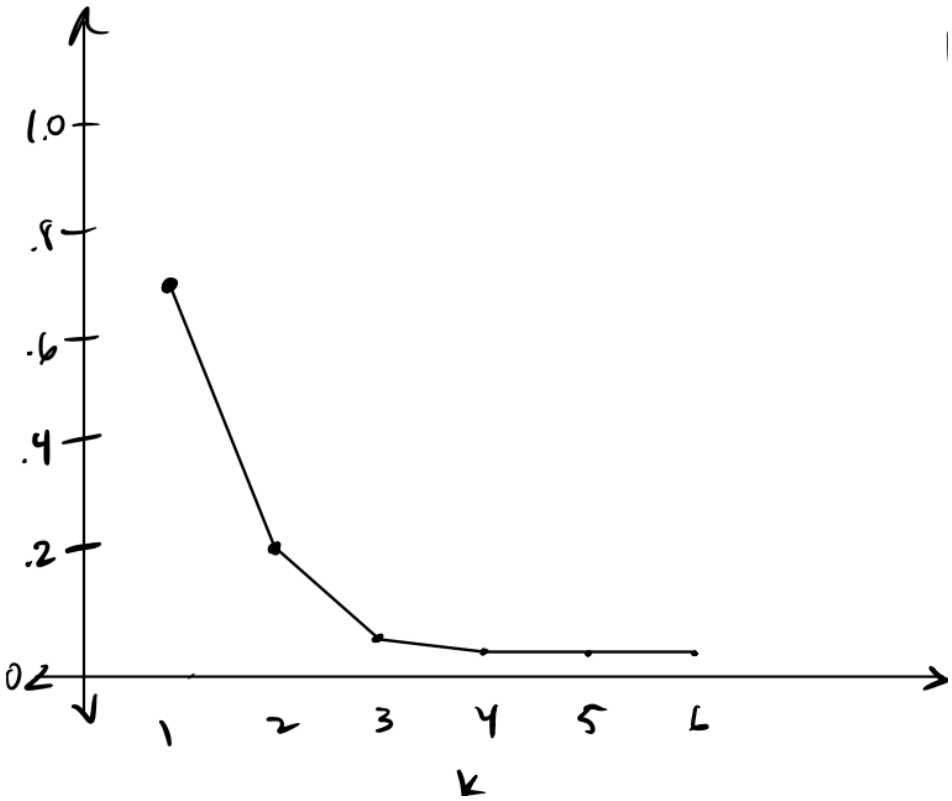
estimates the total variation in X that is explained by the k^{th} principal component.

6. Often we plot

$$\frac{\lambda_k}{\text{tr}(S)}$$

against k which shows how the eigenvalues decay. This is called a *scree plot.*

Ex: if $x_i \in \mathbb{R}^6$ we may have a scree plot like:

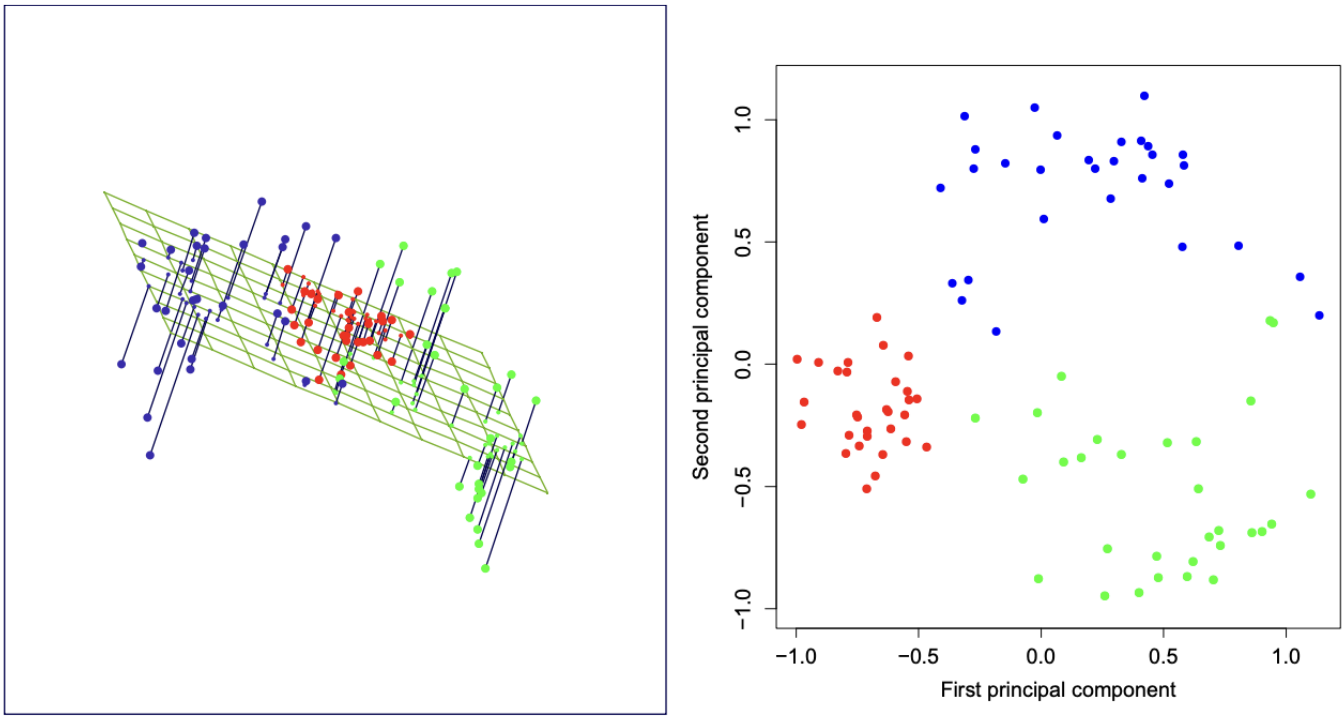


7. We can use scree plots to help decide a good k at which we can truncate the basis expansion of x_i without losing much variation/information:

$$x_i \approx \bar{x} + \sum_{j=1}^{k} \xi_{ij}\phi_j \quad \text{(rank } k \text{ approximation)}$$

Another example for data in $\mathbb{R}^3$:

Consider a dataset consisting of 3-dimensional vectors whose coordinates are noise-contaminated position measurements on a sphere (left). Principal components analysis gives us the optimal low dimensional approximation (here, 2-dimensional) to the sphere-generated data under linear projection. The right panel shows the projection of the data onto the first two principal components, where the coordinates in PC space are given by the rows of UD in the SVD of the centered data matrix.



Recreated from *Elements of Statistical Learning*.

Computation Ex:

Given a design/data matrix

$$X = \begin{bmatrix} 6 & -4 \\ -3 & 5 \\ -2 & 6 \\ 7 & -3 \end{bmatrix}$$

Let's use PCA to approximate the original 4 datapoints by their rank-1 approximations.

Steps:

1. Center the data.

$$X^c = \begin{bmatrix} 4 & -5 \\ -5 & 4 \\ -4 & 5 \\ 5 & -4 \end{bmatrix}$$

2. Calculate

$$S = \frac{1}{3} X^{cT} X^c = \tfrac{1}{3} \begin{bmatrix} 82 & -80 \\ -80 & 82 \end{bmatrix}$$

$$\Rightarrow \text{eigenvectors} : \phi_1 = \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix}, \quad \phi_2 = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$$

$$\lambda_1 = 54 \quad \lambda_2 = 2/3$$

3. Project onto ϕ_1 for the scores:

$$\xi_{11} = [4 \ -5] \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix} = 9/\sqrt{2}$$

$$\xi_{21} = [-5 \ 4] \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix} = -9/\sqrt{2}$$

$$\xi_{31} = [-4 \ 5] \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix} = -9/\sqrt{2}$$

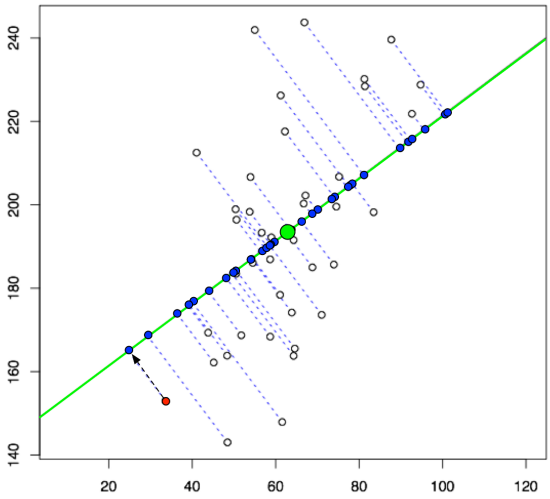
$$\xi_{41} = [5 \ -4] \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix} = 9/\sqrt{2}$$

4. Estimate x_1 by its rank 1 approximation.

$$\begin{bmatrix} 6 \\ -4 \end{bmatrix} = x_1 \approx \bar{X} + \frac{9}{\sqrt{2}} \phi_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix} + \frac{9}{\sqrt{2}} \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 2 + 9/2 \\ 1 - 9/2 \end{bmatrix} = \begin{bmatrix} 6.5 \\ -3.5 \end{bmatrix}$$

Seeing the connection to SVD

We can think about PCA as searching for the optimal basis whose first directions align maximally with the variation in our data. We can explore this idea by noticing that if the first axis of our coordinate system is well-aligned with the data, then the PC scores $\xi_{11}, ..., \xi_{n1}$, will have a high sample variance.



In this example, the first axis of our coordinate system is as well-aligned with the data as possible. As a result, the points in blue (the first PC scores, i.e. projections onto the first direction) are very spread out. This means that the numbers $\xi_{11}, ..., \xi_{n1}$ have a maximally high sample variance.

Mathematically to find this first direction that aligns most with the data, we are essentially searching for the for the unit vector w that maximizes the sample variance of the first PC scores. In other words, we search for w with unit norm that maximizes

$$\begin{aligned}
\frac{1}{n-1}\sum_{i=1}^{n}\xi_{i1}^2 = \frac{1}{n-1}\sum_{i=1}^{n}\langle x_i - \bar{x},\, w\rangle^2 &= \frac{1}{n-1}\sum_{i=1}^{n} w^T(x_i-\bar{x})(x_i-\bar{x})^T w \\
&= \frac{1}{n-1} w^T\Big(\sum_{i=1}^{n}(x_i-\bar{x})(x_i-\bar{x})^T\Big) w \\
&= \frac{1}{n-1} w^T X^{cT} X^c\, w \\
&= \frac{1}{n-1}\|X^c\, w\|^2,
\end{aligned}$$

where X^C is the $n \times p$ matrix whose ith row is $(x_i - \bar{x})^T$. Equivalently, we choose ϕ_1 to be the unit vector w that maximizes $\|X^c\, w\|$. But this is precisely the optimization problem that is solved by the singular value decomposition. If

$$X^c = UDV^T$$

is a singular value decomposition of X^c, then we can take v_1 to be the first column of V.

Next to find the second direction, we choose ϕ_2 to be the unit vector which maximizes $\|X^c\, w\|$ subject to the constraint that w is orthogonal to ϕ_1. But again, this optimization problem is solved by the SVD. We can take ϕ_2 to be the second column of V.

The remaining vectors $\phi_3, \ldots, \phi_p$ are chosen in the same way: ϕ_j is chosen to be the unit vector w which maximizes $\|X^c\, w\|$ subject to the constraints that phi_j must be orthogonal to each of the vectors $\phi_1, \ldots, \phi_{j-1}$. Again, this is precisely the optimization problem solved by the SVD. We can take ϕ_j to be the jth column of V. So, by computing the SVD of the matrix X^c, we have found the principal component vectors $\phi_1, \ldots, \phi_p$.

2 Matrix Factorization and Recommender Systems

Required Reading:

1. Watch this video `https://www.youtube.com/watch?v=zzTbptEdKhY`
2. Chapter 9 from "Mining of Massive Datasets."

The Utility Matrix

In a recommendation system there are two classes of entities, which we will refer to as users and items. Users have preferences for certain items, and these preferences must be teased out of the data. The data is represented as a utility matrix, giving for each user-item pair, a value that represents what is known about the degree of preference of that user for that item.

Example: In Table 1 we see an example utility matrix, representing users' ratings of movies on a 1-5 scale, with 5 the highest rating. Blanks represent the situation where the user has not rated the movie. The movie names are HP1, HP2, and HP3 for Harry Potter 1, 2, and 3, TW for Twilight, and SW1, SW2, and SW3 for Star Wars episodes 1, 2, and 3. The users are represented by capital letters A through D.

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	4		5	1			
B	5	5	4				
C				2	4	5	
D			3				3

Table 1: A utility matrix representing ratings of movies on a 1-5 scale

Most user-movie pairs have blanks, meaning the user has not seen/rated the movie. In practice, the matrix would be much sparser, with the typical user rating only a tiny fraction of all available movies. The goal of a recommendation system is to predict the blanks in the utility matrix. For example, would user A like the movie SW2? In most real systems the goal is to discover some entries in each row that are likely to be high, then take the top C of these items to display as recommendations.

2.1 Populating the Utility Matrix

Without a utility matrix, it is almost impossible to recommend items. Acquiring data from which to build a utility matrix is often difficult. There are two general approaches to discovering the value users place on items.

1. Ask users to rate items (aka *explicit rating*). Movie ratings are generally obtained this way, and some online stores try to obtain ratings from their purchasers. This approach is limited in its effectiveness, since generally users are unwilling to provide responses, and the information from those who do may be biased by the very fact that it comes from people willing to provide ratings.

2. Make inferences from users' behavior (aka *implicit rating*). If a user buys a product at Amazon, watches a movie on YouTube, or reads a news article, etc., then the user can be said to "like" this item, regardless of if they actually end up rating it or not.

2.2 Utility Matrix for Implicit Ratings

In the case of implicit ratings, the Utility Matrix has just one non-blank value, 1 means that the user likes the item (watch a movie / video, viewed or purchased a product).

If in Table 1 instead of having ratings (in scale 1-5) we had the information of whether a user watched a particular movie, our initial utility matrix would look like Table 2. In this case, Table 2 would represent an implicit utility matrix.

For the case of implicit ratings, we could also consider the matrix in Table 3, where blanks are replaced by 0's, but this *assumes missingness is truly reflective of negative ratings*. (Does this make sense? In what cases yes, in what cases no?)

To moderate that assumtion, another option is to sample 0s from the missing values while still leaving some blank (as shown in Table 4).

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	1		1	1			
B	1	1	1				
C				1	1	1	
D			1				1

Table 2: A utility matrix representing movies watched by users

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	1	0	1	1	0	0	0
B	1	1	1	0	0	0	0
C	0	0	0	1	1	1	0
D	0	0	1	0	0	0	1

Table 3: A utility matrix representing movies watched by users where all missing values are filled as negatives (0s).

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	1	0		1	1		0
B	1	1	1	0	0		0
C	0		1	1	1		0
D		0	1		0		1

Table 4: A utility matrix representing movies watched by users where a sample of missing values are filled as negatives

Matrix Factorization

Collaborative filtering recommends items by matching users with other users with similar interests. The items recommended to a user are those preferred by similar users. Collaborative filtering systems are usually categorized into two subgroups: memory-based and model-based. Memory-based methods memorize the utility matrix and make recommendations based on the relationship between the queried user and the rest of the utility matrix often using a distance and K-nearest neighbours. Model-based methods fit a parameterized model to the given utility matrix and then make recommendations based on the model. Here we introduce a model based approach Matrix Factorization.

The matrix factorization algorithm approximates the utility matrix as a product of two long, thin matrices. This view makes sense if there are a relatively small set of features of items and users that determine the rating.

We denote the utility matrix Y with elements $\{y_{ij}\}$, where $i \in 1, \ldots, n_u$ $j \in 1, \ldots, n_m$ and n_u is the number of users while n_m is the number of items. The decomposition has the equation $\hat{Y} = UV^T$ where U is $n_u \times K$ and V is $n_m \times K$. Every rating y_{ij} gets the prediction

$$\hat{y}_{ij} = u_i \cdot v_j$$

where u_i is the K-dimensional user embedding vector and v_j is the K-dimensional item embedding vector. Writing this out fully looks something like:

$$\begin{bmatrix} 5 & 1 & 4 & 5 & 1 \\ 5 & 2 & 1 & 4 & \\ 1 & 4 & 1 & 1 & 2 \\ 4 & 1 & 5 & 5 & 4 \\ 5 & 3 & 3 & & 4 \\ 1 & 5 & 1 & 1 & 1 \\ 5 & 1 & 5 & 5 & 4 \end{bmatrix} \approx \begin{bmatrix} u_{11} & u_{12} & \ldots & u_{1K} \\ u_{21} & u_{22} & \ldots & u_{2K} \\ u_{31} & u_{32} & \ldots & u_{3K} \\ u_{41} & u_{42} & \ldots & u_{4K} \\ u_{51} & u_{52} & \ldots & u_{5K} \\ u_{61} & u_{62} & \ldots & u_{6K} \\ u_{71} & u_{72} & \ldots & u_{7K} \end{bmatrix} \times \begin{bmatrix} v_{11} & v_{21} & v_{31} & v_{41} & v_{51} \\ v_{12} & v_{22} & v_{32} & v_{42} & v_{52} \\ \vdots & & & & \vdots \\ v_{1K} & v_{2K} & v_{3K} & v_{4K} & v_{5K} \end{bmatrix}$$

Here is a toy example in witch we have 7 users and 5 movies, and the choice of the number of parameters is dictated by $K = 2$.

$$\begin{bmatrix} 5 & 1 & 4 & 5 & 1 \\ 5 & 2 & 1 & 4 & \\ 1 & 4 & 1 & 1 & 2 \\ 4 & 1 & 5 & 5 & 4 \\ 5 & 3 & 3 & & 4 \\ 1 & 5 & 1 & 1 & 1 \\ 5 & 1 & 5 & 5 & 4 \end{bmatrix} \approx \begin{bmatrix} 0.2 & 3.4 \\ 3.6 & 1.0 \\ 2.6 & 0.6 \\ 0.9 & 3.7 \\ 2.0 & 3.4 \\ 2.9 & 0.5 \\ 0.8 & 3.9 \end{bmatrix} \times \begin{bmatrix} 0.0 & 1.5 & 0.1 & 0.0 & 0.7 \\ 1.3 & 0.0 & 1.2 & 1.4 & 0.7 \end{bmatrix}$$

We can gain insight into what these matrices represent by looking more closely at the estimated parameters (sometimes called "loadings") in terms of either the users or movies. U is the representation of users in some low dimensional space. For user i, the vector u_i is the representation of user i in terms of types of user preferences. It would have high magnitude loadings for underlying concepts which that user either extremely likes/dislikes (for example, "likes romance", "likes comedy", or more complicated and nuanced preferences). Similarly or an item j, v_j is the representation of the items in the same low dimensional space. An item will have a high magnitude loading if it is representative of the underlying concept ("has romance themes", "is comedic," etc.).

2.3 Computing similar items and similar users

The row vector $v_j = (v_{j1}, \ldots, v_{jK})$ can be interpreted as a learned feature vector for item j. If we now wanted to find items closely related to j we can look for items that v_ℓ that have small Euclidean distance to

v_j. That is, we want

$$\|v_j - v_\ell\|$$

to be small. We can identify similar users in the same fashion by using vectors u_i and u_k.

2.4 Loss Function

Many loss functions can work for this kind of model. The typical choice is the mean squared error (MSE) for observed values of the utility matrix. Let r_{ij} be 1 if the element y_{ij} of the utility matrix is non-empty and 0 otherwise. Then the MSE loss is:

$$E(u, v) = \frac{1}{N} \sum_{(i,j):r_{ij}=1} (y_{ij} - u_i v_j)^2$$

where N is the number of non-blank elements of the utility matrix Y.

2.5 Gradient descent with momentum

We can minimize the MSE using gradient descent.

Recall in gradient descent if we are trying to learn some weights w and we have some error function $E(w)$, at each iteration we update w as a small step into the direction of the negative gradient

$$w_{t+1} = w_t - \eta \nabla E(w_t)$$

where the parameter η is known as the learning rate.

A popular version of gradient descent is gradient descent with momentum. Instead of using the gradient directl, we can use a "moving average" of our gradients by incorporating a momentum term. Here is the modified gradient with momentum:

$$v_t = \beta v_{t-1} + (1 - \beta) \nabla E(w_t)$$

The updating equation for the model parameter becomes:

$$w_{t+1} = w_t - \eta v_t$$

2.6 Gradient descent for Matrix Factorization

We are trying to minimize mean square error:

$$E(U, V) = \frac{1}{N} \sum_{(i,j):r_{ij}=1} (y_{ij} - u_i \cdot v_j)^2$$

where N is the number of non-blank elements of Y and $u_i \cdot v_j = \sum_{s=1}^{K} u_{is} v_{js}$. Consider the derivative of E with respect to $u_{\ell k}$ or v_{jk}. Here i and ℓ are user indices, j is an item index, and k is an index for which dimension of the low-dim. embedding we're referring to.

Then the general from of GD equations look like:

$$\frac{\partial E}{\partial u_{\ell k}} = -\frac{2}{N} \sum_{j:r_{\ell j}=1} (y_{\ell j} - u_\ell \cdot v_j) \frac{\partial E}{\partial u_{\ell k}} \quad (1)$$

$$\frac{\partial E}{\partial v_{jk}} = -\frac{2}{N} \sum_{i:r_{ij}=1} (y_{ij} - u_i \cdot v_j) \frac{\partial E}{\partial v_{jk}} \quad (2)$$

To see why for (1), note that

$$\frac{\partial E}{\partial u_{\ell k}} = \frac{\partial (u_{ij} \cdot v_j)}{\partial u_{\ell k}} = \begin{cases} v_{jk}, & \text{if } \ell = i \\ 0, & \text{otherwise} \end{cases}$$

Therefore the gradient descent updating equation for (1) is:

$$u_{\ell k} \leftarrow u_{\ell k} + \frac{2\eta}{N} \sum_{j:r_{\ell j}=1} (y_{\ell j} - u_\ell \cdot v_j) v_{\ell k}$$

and a similar parallel argument gives for (2):

$$v_{jk} \leftarrow v_{jk} + \frac{2\eta}{N} \sum_{i:r_{ij}=1} (y_{ij} - u_i \cdot v_j) u_{ik}$$

For a full derivation, see the detailed notes below in "Details of Gradient Descent for Matrix Factorization."

2.7 Vectorizing gradient descent

For fast implementation of gradient decent, let's vectorize the updating equations.

Let R be the $n_u \times n_m$ matrix with elements $\{r_{ij}\}$. Let $\Delta = (Y - UV^T) \otimes R$, where the operation $\otimes$ represents elementwise matrix multiplication. Let $\frac{\partial E}{\partial U}$ and $\frac{\partial E}{\partial V}$ denote the matrix with elements $\frac{\partial E}{\partial u_{ik}}$ and $\frac{\partial E}{\partial v_{jk}}$ respectively. Then we can write the matrix of partial derivatives with the following equations:

$$\frac{\partial E}{\partial U} = -\frac{2}{N} \Delta \cdot V$$

$$\frac{\partial E}{\partial V} = -\frac{2}{N} \Delta^T \cdot U$$

Exercise: Verify that these equations are equivalent to equations 1 & 2 by showing that the element (i, k) from a matrix multiplication is the dot product of the i row on the first matrix and the k column of the second matrix.

2.8 Avoiding overfitting

We can add a regularization term to our error function.

$$\frac{1}{N} \sum_{(i,j):r_{ij}=1} (y_{ij} - u_i v_j)^2 + \lambda \left(\sum_{i=1}^{n_u} \sum_{k=1}^{K} u_{ik}^2 + \sum_{i=1}^{n_m} \sum_{k=1}^{K} v_{ik}^2 \right)$$

Where $N = \sum_{ij} r_{ij}$

Exercise. How would the gradient descent equations change? Derive the new updating equations with this regularization implemented.

Interview Questions

1. What types of recommender systems exist and can you describe them?
2. How would you evaluate a recommender system?
3. How would you design a recommender system for scientists on mendeley.com, a website for research articles?

4. What is the "cold start" phenomenon?
5. How would you design a recommendation system for a marketplace like Amazon?
6. How would you build a recommendation system for an entertainment platform similar to Netflix?

References

[1] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2nd edition, 2014.

3 Details of Gradient Descent for Matrix Factorization

Set Up:

Start with a utility matrix $Y \in \mathbb{R}^{n_u \times n_m}$.

EX: Ratings of items:

If there are n_u users and n_m items, then Y looks like the inside array of this table:

	Item 1	$\cdots$	Item j	$\cdots$	Item n_m
user 1	y_{11}	$\cdots$		$\cdots$	
$\vdots$		$\ddots$			
user i		$\cdots$	y_{ij}	$\cdots$	
$\vdots$				$\ddots$	
user n_u		$\cdots$		$\cdots$	$y_{n_u n_m}$

In most cases this matrix is really sparse. (Why?)

Goal for Matrix Factorization:

Find "skinny" matrices U & V such that $Y \approx UV^T$.

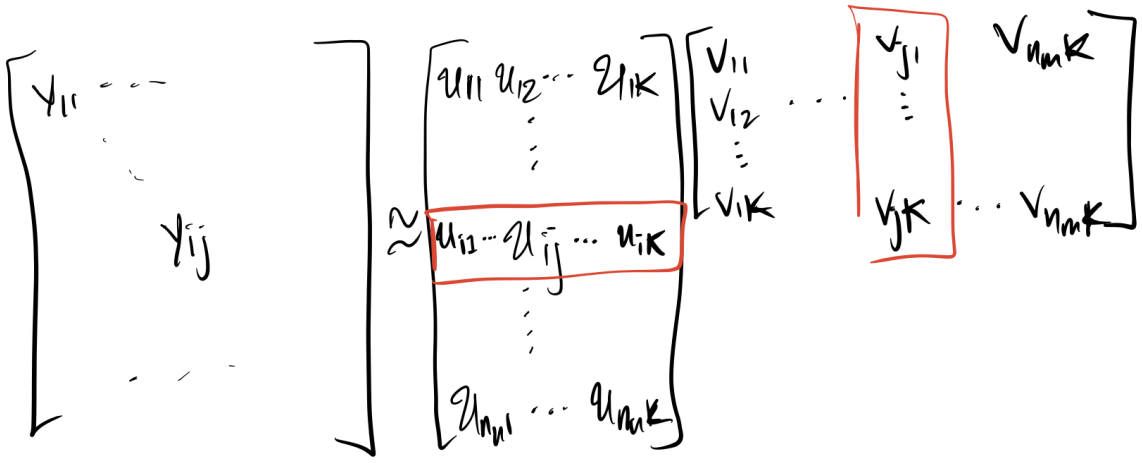
i.e:

$$Y \in \mathbb{R}^{n_u \times n_m}, \quad U \in \mathbb{R}^{n_u \times K}, \quad V^T \in \mathbb{R}^{K \times n_m}.$$

If K is small, then the matrix is "skinny" ... dimension reduction is happening!

Matrix Factorization Model:

$$\hat{y}_{ij} = (UV^T)_{ij} = \langle u_i, v_j \rangle = \sum_s u_{is} v_{js} \quad \leftarrow \text{dot product of } i\text{th row of } U \text{ and } j\text{th col of } V^T.$$



Parameters:

What are the parameters of this model? The elements of U & V.
How many parameters are there in this model? $(n_u + n_m) \times K$
How to find the optimal values of parameters?
Minimize some loss, denoted $E(U, V)$.
Let

$$r_{ij} = \begin{cases} 1 & \text{if } y_{ij} \text{ is not empty} \\ 0 & \text{else} \end{cases}$$

and $N = \sum_{i,j} r_{ij}$, i.e. the number of non-empty elements of Y.
Then we can define a squared error loss for the matrix factorization as:

$$E(U, V) = \frac{1}{N} \sum_{(i,j):r_{ij}=1} (y_{ij} - (UV^T)_{ij})^2.$$

Reminder: Useful Derivative Properties

Additivity:

$$(f + g)'(x) = f'(x) + g'(x).$$

Chain Rule for Composition:

$$(f \circ g)'(x) = f'(g(x)) \cdot g'(x).$$

Ex: $L(x) = (y - f(x))^2$

$$L'(x) = -2(y - f(x)) \cdot f'(x).$$

Back to Matrix Factorization

Our loss function for the MF model is:

$$E(U, V) = \frac{1}{N} \sum_{(i,j):r_{ij}=1} (y_{ij} - \sum_s u_{is} v_{js})^2.$$

Let's find the updating equations for elements of U & V, one at a time.
We will need:

$$\frac{\partial E}{\partial u_{ik}} \quad \text{and} \quad \frac{\partial E}{\partial v_{jk}} \quad \forall i, j.$$

Intermediate Result:

Recall $\hat{y}_{ij} = (UV^T)_{ij} = \sum_s u_{is}v_{js}$.
Then:

$$\frac{\partial \hat{y}_{ij}}{\partial u_{lk}} = \begin{cases} 0 & \text{if } \ell \neq i \\ v_{jk} & \text{if } \ell = i. \end{cases}$$

(i & ℓ are user indices.)

Illustrative Ex: If $K = 2$:

$$\hat{y}_{34} = u_{31}v_{41} + u_{32}v_{42}.$$

If we use $\ell = 5$, we see:

$$\frac{\partial \hat{y}_{34}}{\partial u_{51}} = \frac{\partial}{\partial u_{51}}(u_{31}v_{41} + u_{32}v_{42}) = 0 + 0 = 0.$$

If we use $\ell = 3$, we see:

$$\frac{\partial \hat{y}_{34}}{\partial u_{31}} = \frac{\partial}{\partial u_{31}}(u_{31}v_{41} + u_{32}v_{42}) = v_{41} + 0 = v_{41}.$$

So we see that $\frac{\partial \hat{y}_{ij}}{\partial u_{\ell k}} = v_{jk}$ only when $i = \ell$, if else, it's 0.

Back to gradient descent for the main loss function:

We know:

$$\frac{\partial E}{\partial u_{ik}} = \frac{1}{N} \sum_{(i,j):r_{ij}=1} \frac{\partial}{\partial u_{ik}}(y_{ij} - \hat{y}_{ij})^2.$$

$$= \frac{1}{N} \sum_{(i,j):r_{ij}=1} [-2(y_{ij} - \hat{y}_{ij}) \times \frac{\partial \hat{y}_{ij}}{\partial u_{\ell k}}].$$

$$= -\frac{2}{N} \sum_{j:r_{ij}=1} (y_{ij} - \hat{y}_{ij}) \cdot v_{jk}.$$

Therefore, the updating equation for the parameters of U is:

$$u_{\ell k} \leftarrow u_{\ell k} + \frac{2\eta}{N} \sum_{j:r_{\ell j}=1} (y_{\ell j} - \hat{y}_{\ell j})v_{jk}.$$

Exercise: Find the updating equation for v_{jk}.

A similar derivation for v_{jk} gives:

$$v_{jk} \leftarrow v_{jk} + \frac{2\eta}{N} \sum_{i:r_{ij}=1} (y_{ij} - \hat{y}_{ij})u_{ik}.$$

Vectorizing Gradient Descent for Matrix Factorization

Recall from above, the GD updating equations are:

$$u_{ik} \;\leftarrow\; u_{ik} + \frac{2\eta}{N} \sum_{j:r_{ij}=1} \big(y_{ij} - u_i \cdot v_j\big)\, v_{jk},$$

$$v_{jk} \;\leftarrow\; v_{jk} + \frac{2\eta}{N} \sum_{i:r_{ij}=1} \big(y_{ij} - u_i \cdot v_j\big)\, u_{ik},$$

& R is the matrix with $r_{ij} = 1$ if y_{ij} not empty, 0 else.

Then define Δ as

$$\Delta \;=\; \big(Y - U \cdot V^T\big) \;\otimes\; R.$$

where $\otimes$ represents elementwise matrix multiplication. **What is Δ_{ij}?**

$$\Delta_{ij} = \big(y_{ij} - u_i \cdot v_j\big)\, r_{ij}.$$

Define the gradient matrices:

$$\frac{\partial E}{\partial U} = \text{ the matrix w/ elements } \frac{\partial E}{\partial u_{ik}}, \qquad \frac{\partial E}{\partial V} = \text{ the matrix w/ elements } \frac{\partial E}{\partial v_{jk}}.$$

Claim:

$$\frac{\partial E}{\partial U} = -\frac{2}{N}\,\Delta\, V \quad \text{and} \quad \frac{\partial E}{\partial V} = -\frac{2}{N}\,\Delta^T\, U.$$

Proof: Let's show equality element-by-element.

$$\left(\frac{\partial E}{\partial U}\right)_{ik} = \frac{\partial E}{\partial u_{ik}} = -\frac{2}{N} \sum_j \Delta_{ij}\, v_{jk} \quad \text{(def of mat. mult.)}$$

Plug in Δ_{ij}:

$$\left(\frac{\partial E}{\partial U}\right)_{ik} = -\frac{2}{N} \sum_j \big(y_{ij} - u_i \cdot v_j\big)\, r_{ij}\, v_{jk} = -\frac{2}{N} \sum_{j:r_{ij}=1} \big(y_{ij} - u_i \cdot v_j\big)\, v_{jk}.$$

That's all! We have shown how to vectorize the gradient descent updates for matrix factorization. A parallel argument goes for $\dfrac{\partial E}{\partial V}$.

4 Neural Networks

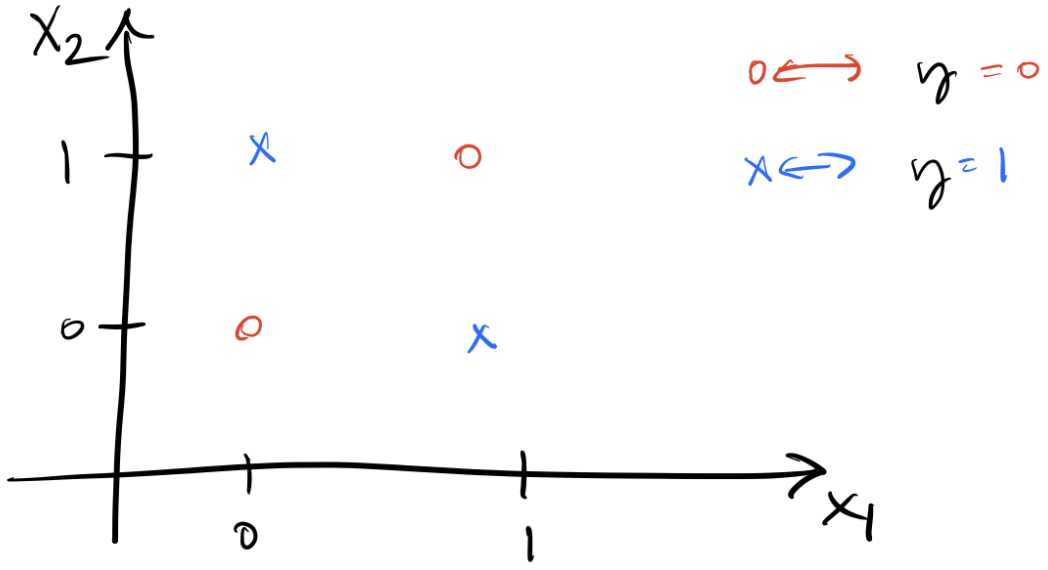
Learning Nonlinearities

Consider the XOR operator. In plain language, $y = \text{XOR}(x_1, x_2)$ lights up as TRUE (1) when exactly one of x_1 or x_2 is true, but not both. That is, if we have $x_1 \in \{0, 1\}$ and $x_2 \in \{0, 1\}$, $y = \text{XOR}(x_1, x_2)$ takes its value according to:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

The Classification Problem

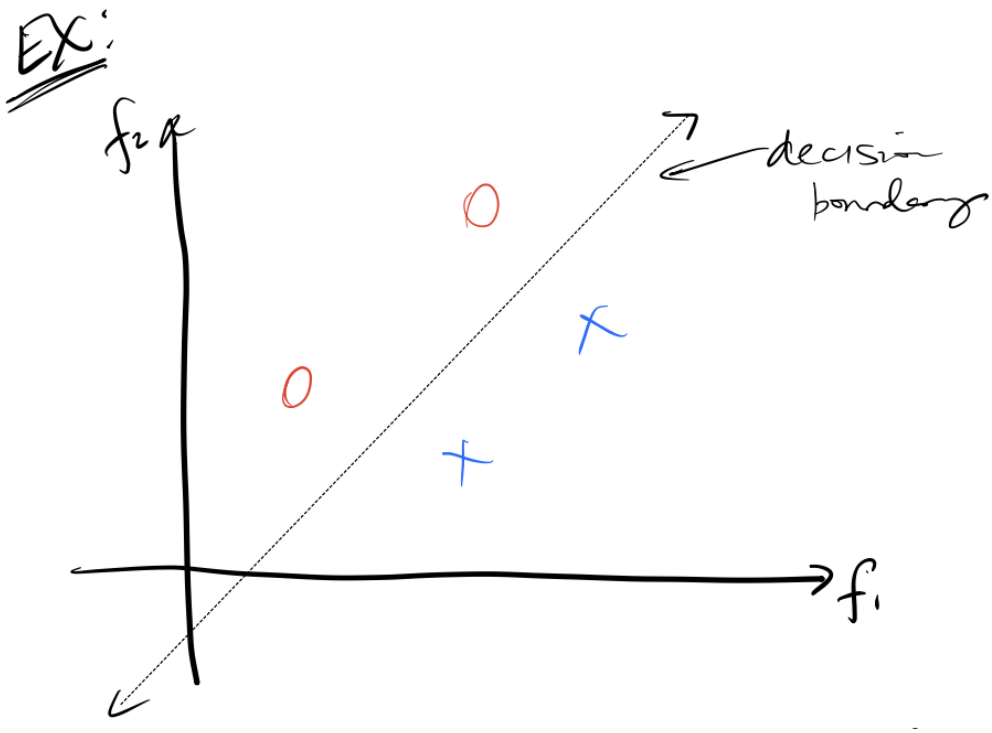
Since $y \in \{0, 1\}$, we can think of this setup in terms of a classification problem with the predictors x_1 and x_2.



The big problem:
We cannot come up with a decision boundary that is linear in x_1 and x_2 which separates the classes!!

Feature Engineering

One idea is to try feature engineering. That is, construct features $f_1 = \phi_1(x_1, x_2)$ and $f_2 = \phi_2(x_1, x_2)$ such that the classes are linearly separable in the space spanned by (f_1, f_2). We want to end up with something like:



But how should we pick $\phi_1(\cdot,\cdot)$ and $\phi_2(\cdot,\cdot)$?

Since this example is really simple, we can just think a bit and arrive through trial and error at some good choices.

Ex:

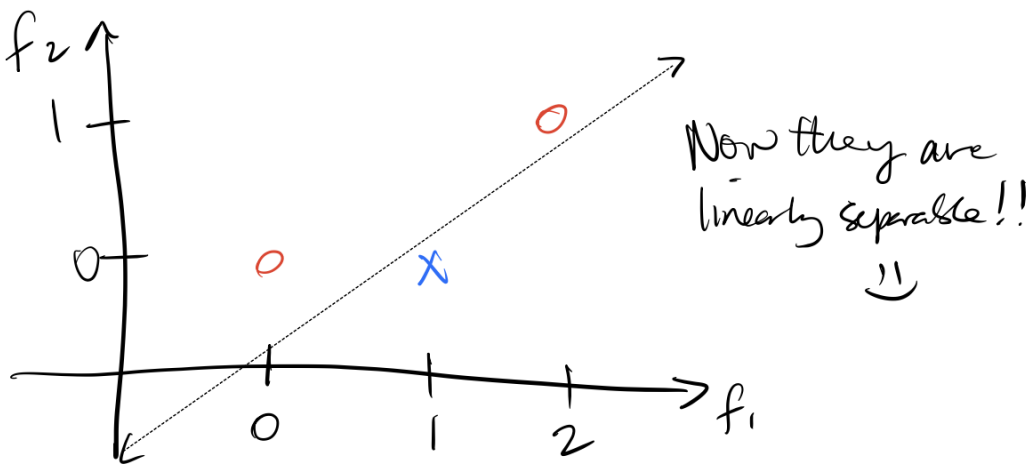
For example, take:

$$f_1 = \phi_1(x_1, x_2) = \max(x_1 + x_2, 0)$$
$$f_2 = \phi_2(x_1, x_2) = \max(x_1 + x_2 - 1, 0)$$

Then we have:

x_1	x_2	f_1	f_2	y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

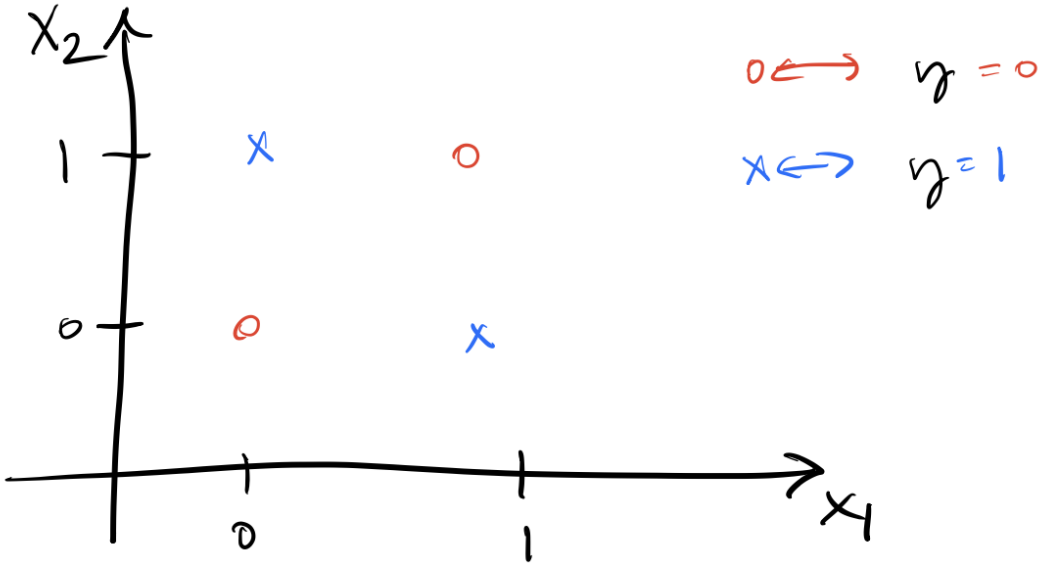
Visually, now we have:



Now they are linearly separable!! :)

Connection to Neural Networks/Representation Learning

This example of feature engineering also illustrates how neural networks work! Here is the idea:



If we take $W = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 1 & 1 \end{bmatrix}$ and the ReLU activation function,

$$\sigma(x) = ReLU(x) = x\mathbf{1}_{\{x>0\}}(x) = \max(x, 0)$$

then the constructed features are:

$$f_1 = \sigma(w_{10} + w_{11}x_1 + w_{12}x_2) = (x_1 + x_2)_+ = \phi_1(x_1, x_2)$$
$$f_2 = \sigma(w_{20} + w_{21}x_1 + w_{22}x_2) = (x_1 + x_2 - 1)_+ = \phi_2(x_1, x_2)$$

The last step is just to train a logistic regression on them and learn the $\hat{\beta}$ vector. Knowing the $\hat{\beta}$ coefficients is equivalent to knowing the decision boundary!

A Couple Things to Note

1. This problem was simple enough so that we could just guess the values of W. In general, we will have to learn them!

2. Sometimes classes are not linearly separable because of noise, sometimes because of a true structural nonlinearity (or both). In this case, we observe the values of $y = \text{XOR}(x_1, x_2)$ without noise, so we know it is a true structural nonlinearity. For structural nonlinearities, neural networks can help a lot; they can't do much about the noise problem, though.

3. The concept that the neural network is trying to learn how to represent the features f is one of the big ideas of neural networks. Sometimes you'll hear this referred to as "representation learning."

How to Fit a Neural Network: Backpropagation

In general, our neural network will be trying to minimize some **empirical loss function**. Let $\tilde{L}(\boldsymbol{y}, f(\boldsymbol{x}))$ denote the loss computed over an entire sample and $L(y_i, f(x_i))$ the loss for a single observation, so that

$$\tilde{L}(\boldsymbol{y}, f(\boldsymbol{x})) = \sum_{i=1}^{n} L(y_i, f(x_i)).$$

Stochastic gradient descent (SGD) helps us minimize this loss.
Consider a simple feed-forward neural network which looks like the following, where $\hat{y} = a$:

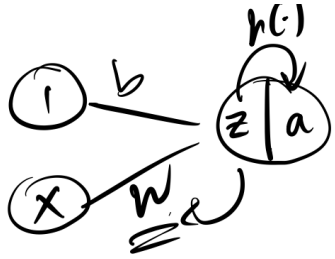


Figure 1: Simple neural network architecture

Notation

- **Output:** $\hat{y}$
- **Linear predictor:**

  $$z = wx + b,$$

  where w is the weight associated with x, the input/ predictor, and b is the bias/intercept term.
- **Activated linear predictor:**

  $$\begin{aligned} a &= h(z) \\ &= \text{"activated linear predictor"} \\ &= \text{hidden feature} \end{aligned}$$

  where h is the activation function.
- $\boldsymbol{x} = (x_1, ..., x_n)$

Role of Activation Functions

Activation functions are the mechanism by which neural networks learn nonlinear relationships. Without activation functions, a neural network would collapse to a single linear transformation.

By introducing nonlinearities between layers, activation functions allow neural networks to approximate a wider class of functions. In fact, under some mild regularity conditions, feed-forward neural networks with nonlinear activations are universal function approximators.

Beyond expressiveness, activation functions also influence optimization behavior. Their derivatives determine how gradients propagate backward through the network during training. Poor choices of activation functions can lead to vanishing or exploding gradients, which can slow or totally prevent learning.

Common Activation Functions

Several activation functions are commonly used in practice:

- **Sigmoid:** $h(z) = \sigma(z) = \dfrac{1}{1+e^{-z}}$
  - Maps inputs to $(0, 1)$ and was historically popular. However, sigmoid activations saturate for large $|z|$, leading to vanishing gradients.
- **Hyperbolic Tangent (tanh):** $h(z) = \tanh(z)$
  - Outputs values in $(-1, 1)$ and is zero-centered, improving optimization relative to sigmoid, but still suffers from saturation.
- **ReLU (Rectified Linear Unit):** $h(z) = \text{ReLU}(z) = \max(0, z)$
  - ReLU is computationally simple, avoids saturation for positive inputs, and enables faster training. For these reasons, it is the default activation in most modern deep networks.

SGD for a Single Observation

For a single observation (x, y), the gradient of the loss with respect to the weights is computed by the chain rule:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}.$$

Since

$$\hat{y} = a = h(z), \quad \text{and} \quad z = wx + b,$$

we have

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot h'(z) \cdot x$$

and so the stochastic gradient descent equation is

$$w \leftarrow w - \eta \cdot (\frac{\partial L}{\partial \hat{y}_i} \cdot h'(z_i) \cdot x_i).$$

Similarly, because

$$\frac{\partial z}{\partial b} = 1,$$

the gradient with respect to the bias is

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}_i} \cdot h'(z_i)$$

and the bias update is

$$b \leftarrow b - \eta \, \frac{\partial L}{\partial \hat{y}_i} \cdot h'(z_i)$$

SGD for the Entire Dataset (or Mini-Batch)

When using the entire dataset (or a mini-batch) of n observations, we sum the gradients:

$$w \leftarrow w - \eta \sum_{i=1}^{n} [\frac{\partial L}{\partial \hat{y}_i} \cdot h'(z_i) \cdot x_i]$$

$$b \leftarrow b - \eta \sum_{i=1}^{n} [\frac{\partial L}{\partial \hat{y}_i} \cdot h'(z_i)]$$

A More Complicated Network

Now consider a network with multiple predictors $x_1, x_2, \ldots, x_p$ for a given observation, i.e.,

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_p)^T,$$

a $p \times 1$ column vector of predictors.

(Or for a sample, $i = 1, ..., n$: $\boldsymbol{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ip})^T$.)

In a more complex (multi-layer) network the gradient of the loss with respect to a given weight involves contributions from multiple layers of dependence.

We will start in the context of a regression problem with a neural network with a single hidden layer with d neurons, producing an output $\hat{y}$:

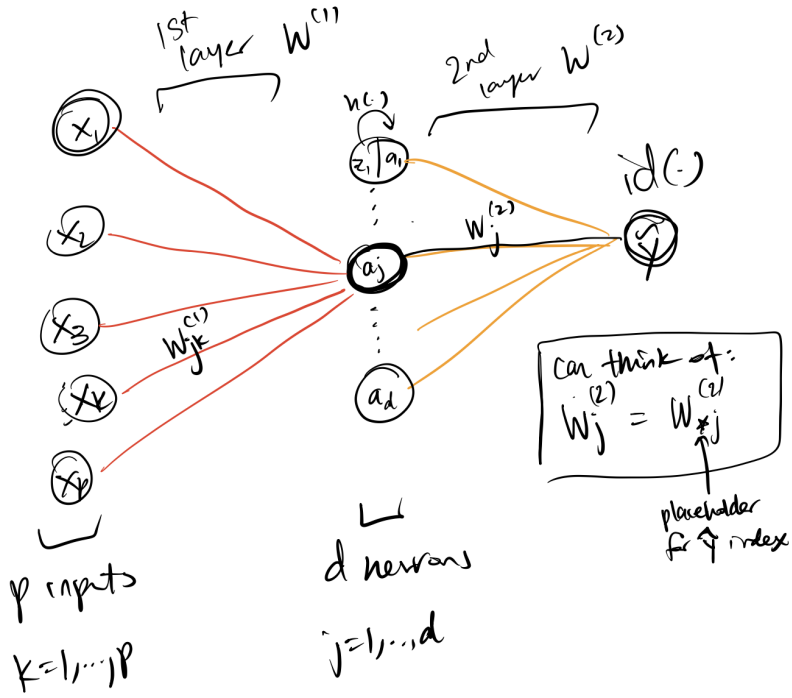


Figure 2: Simple neural network architecture

Here we have for the first layer:

$$\boldsymbol{z}^{[1]} = W^{[1]}\boldsymbol{x} + b^{[1]},$$

where $W^{[1]}$ is the first layer's weight matrix, $\boldsymbol{x}$ is the vector of input predictors, and b is the bias. Together these combine to get $\boldsymbol{z}^{[1]}$ the d-dimensional vector of unactivated linear predictors, which will pass through $h(\cdot)$ to get activated.

Let's try to update $w_{jk}^{[1]}$.
The NN equations are:

$$z_j^{[1]} = \sum_{k=1}^{p} w_{jk}^{[1]} x_k \; + \; b_j^{[1]}$$

$$a_j^{[1]} = h\big(z_j^{[1]}\big)$$

$$z^{[2]} = \sum_{j=1}^{d} w_j^{[2]} a_j^{[1]} \; + \; b^{[2]} \; = \; \hat{y}$$

Hence,

$$\hat{y} = z^{[2]}.$$

The gradient of L w.r.t. $w_{jk}^{[1]}$ looks like:

$$\frac{\partial L}{\partial w_{jk}^{[1]}} \; = \; \frac{\partial L}{\partial \hat{y}} \; \cdot \; \frac{\partial \hat{y}}{\partial a_j^{[1]}} \cdot \frac{\partial a_j^{[1]}}{\partial z_j^{[1]}} \; \cdot \; \frac{\partial z_j^{[1]}}{\partial w_{jk}^{[1]}}.$$

This simplifies to:

$$\frac{\partial L}{\partial w_{jk}^{[1]}} \; = \; \frac{\partial L}{\partial \hat{y}} \; \cdot \; w_j^{[2]} \cdot h'(z_j^{[1]}) \; \cdot \; x_k$$

Therefore for a single observation, the SGD updating equation looks like:

$$w_{jk}^{[1]} = w_{jk}^{[1]} - \eta \cdot [\frac{\partial L}{\partial \hat{y}_i} \; \cdot \; w_j^{[2]} \cdot h'(z_{ij}^{[1]}) \; \cdot \; x_{ik}]$$

or for the whole sample:

$$w_{jk}^{[1]} = w_{jk}^{[1]} - \eta \cdot \sum_{i=1}^{n} [\frac{\partial L}{\partial \hat{y}_i} \; \cdot \; w_j^{[2]} \cdot h'(z_{ij}^{[1]}) \; \cdot \; x_{ik}]$$

Similarly, the bias update for a

$$\begin{aligned} b_j &\leftarrow b_j - \eta \frac{\partial L}{\partial \hat{y}_i} \; \cdot \; \frac{\partial \hat{y}_i}{\partial a_j^{[1]}} \cdot \frac{\partial a_j^{[1]}}{\partial z_{ij}^{[1]}} \; \cdot \; \frac{\partial z_{ij}^{[1]}}{\partial b_j^{[1]}} \\ &= b_j - \eta \frac{\partial L}{\partial \hat{y}_i} w_j^{[2]} h'(z_{ij}^{[1]}) \end{aligned}$$

And the bias update for a whole sample is:

$$\begin{aligned} b_j &\leftarrow b_j - \eta \sum_{i=1}^{n} [\frac{\partial L}{\partial \hat{y}_i} \; \cdot \; \frac{\partial \hat{y}_i}{\partial a_j^{[1]}} \cdot \frac{\partial a_j^{[1]}}{\partial z_{ij}^{[1]}} \; \cdot \; \frac{\partial z_{ij}^{[1]}}{\partial b_j^{[1]}}] \\ &= b_j - \eta \sum_{i=1}^{n} [\frac{\partial L}{\partial \hat{y}_i} w_j^{[2]} h'(z_{ij}^{[1]})] \end{aligned}$$

Exercise

1. Continuing with the previous example, derive the SGD (or gradient descent) updating equations for $w_j^{[2]}$ and $b^{[2]}$.

5 Boosting

AdaBoost

Recall that, in general, boosting is an additive stagewise model, i.e.,

$$f(x) \;=\; \sum_{m=1}^{M} \alpha_m \, T_m(x).$$

Where $T_m(x) = \{-1, +1\}$ and the true labels $y \in \{-1, +1\}$.

Our prediction $\hat{y}$ is defined as:

$$\hat{y} \;=\; \text{sign}\big(f(x)\big).$$

It's useful for us to define the *margin* as:

$$m(x) \;=\; y \cdot f(x).$$

Claim: An obs (x, y) is classified correctly iff the margin is positive, $m(x) > 0$.

Proof: If $m(x) = y \cdot f(x) > 0$ then y and $f(x)$ have the same sign. Then if $m(x) > 0 \implies \hat{y} = \text{sign}\big(f(x)\big) \implies \hat{y} = y$.

If $y\, f(x) < 0 \implies \hat{y}$ is the opposite sign of y. $\implies \hat{y} \neq y$.

Things to notice:

1. $f(x) = 0$ is the decision boundary.
2. The margin can be thought of as like a residual for binary classification.
3. Loss functions for binary classification problem can be written in terms of the margin.

Claim: Log loss for binary classification (starting from the $y^* = \{0, 1\}$ logistic regression setup) can be written as:

$$L\big(y, f(x)\big) \;=\; \log\big[\,1 + e^{-\,y\, f(x)}\big],$$

if we switch to the convention of taking $y \in \{-1, +1\}$ and the classification rule is $\hat{y} = \text{sign}\big(f(x)\big)$.

Proof: Consider the $y^* = \{0, 1\}$ logistic regression scenario. Recall $f(x)$ is just the linear predictor in this case: $f(x) = a\, x + b$, and thus the soft prediction is,

$$\hat{y}^* \;=\; \text{sigmoid}\big(f(x)\big) \;=\; \frac{1}{1 + e^{-\,f(x)}}.$$

The binary log-loss in general takes the form:

$$L\big(y^*, \hat{y}^*\big) \;=\; -\Big[y^* \,\log\big(\hat{y}^*\big) \;+\; (1 - y^*)\,\log\big(1 - \hat{y}^*\big)\Big].$$

Here's the main idea we use to simplify this expression:

$$y^* = 1 \implies y = 1, \text{ and } \quad y^* = 0 \implies y = -1.$$

So in each case,

$$L\big(y^*, \hat{y}^*\big) \;=\; \begin{cases} -\log\big(\hat{y}^*\big), & \text{if } y^* = 1 \iff y = 1, \\ -\log\big(1 - \hat{y}^*\big), & \text{if } y^* = 0 \iff y = -1. \end{cases}$$

If $y = 1$,

$$-\log\big(\hat{y}^*\big) \;=\; -\log\Big[\big(1 + e^{-\,f(x)}\big)^{-1}\Big] \;=\; \log\big[\,1 + e^{-\,f(x)}\big].$$

And if $y = -1$,

$$-\log\big(1 - \hat{y}^*\big) \;=\; -\log\Big(1 - \frac{1}{1 + e^{-f(x)}}\Big) \;=\; -\log\Big(1 - \frac{e^{f(x)}}{e^{f(x)} + 1}\Big) =$$

$$-\log\Big(\frac{e^{f(x)} + 1 - e^{f(x)}}{e^{f(x)} + 1}\Big) \;=\; -\log\Big(\frac{1}{1 + e^{f(x)}}\Big) \;=\; \log\big[\,1 + e^{f(x)}\big] \;=\; \log\big[\,1 + e^{-\,y\,f(x)}\big].$$

Examples of Losses for Binary Classification

Log-loss $\Longrightarrow$

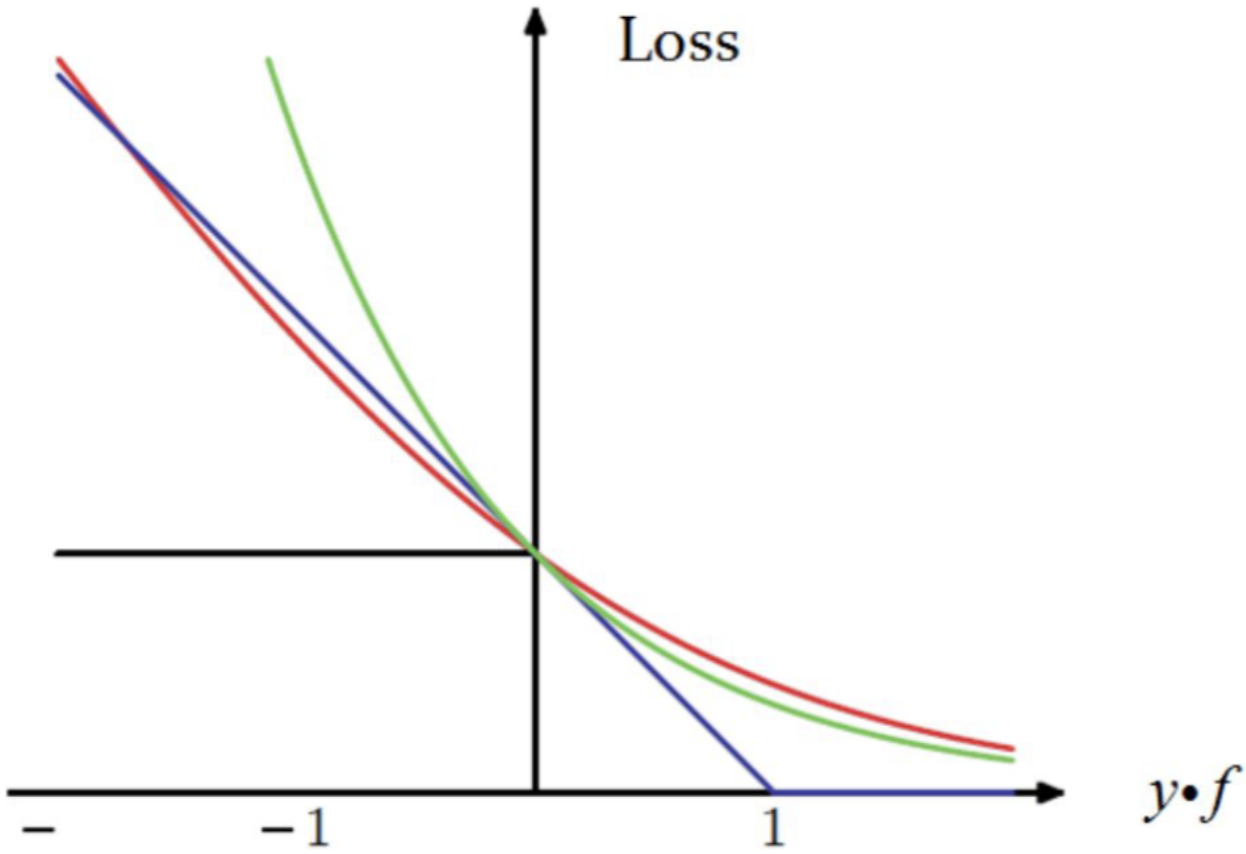
$$\ell\big(y, f(x)\big) \;=\; \log\big(1 + e^{-\,y\,f(x)}\big).$$

Exp. $\Longrightarrow$

$$L\big(y, f(x)\big) \;=\; e^{-\,y\,f(x)}.$$

Hinge $\Longrightarrow$

$$L^{H}\big(y, f(x)\big) \;=\; \max\big(0,\; 1 - y \cdot f(x)\big).$$



Notice:

- Loss functions penalize negative margins. How do each of these penalize? Look at the asymptotic behavior for negative margins:

$$\text{Exp.} \;\to\; e^{-m(x)} \quad \text{(exponential penalty)},$$

$$\text{LogLoss} \;\to\; \log\big(1 + e^{-\,m(x)}\big) \;\approx\; \text{linear for large negative margins},$$

$$\text{Hinge} \;\to\; \text{linear for negative margins.}$$

What is AdaBoost doing?

AdaBoost is:

1. optimizing exponential loss, and
2. fitting additive models in steps.

Consider the additive model:

$$f(x) \;=\; \sum_{m=1}^{M} \beta_m \, T_m(x).$$

Problem: Find $\beta_1, \ldots, \beta_m$ and $T_1, \ldots, T_m$ such that

$$\frac{1}{n}\sum_{i=1}^{n} L\big(y_i,\; f(x_i)\big) \;=\; \frac{1}{n}\sum_{i=1}^{n} L\Big(y_i,\; \sum_{m=1}^{n} \beta_m \, T_m(x)\Big)$$

is minimized. Finding all trees at once is intractable, so we work in stages.

Stage 1: Find β and T such that

$$\frac{1}{n}\sum_{i=1}^{n} L\big(y_i,\; \beta\, T(x_i)\big) \;=\; \frac{1}{n}\sum_{i=1}^{n} e^{-\,y_i\,\beta\,T(x_i)} \quad \text{is minimized.}$$

First,

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n} L\big(y_i,\; \beta\, T(x_i)\big) &= \frac{1}{n}\sum_{i=1}^{n} e^{-\,y_i\,\beta\,T(x_i)} \\ &= \frac{1}{n}\Big(\sum_{\text{margin}=1} e^{-\beta} \;+\; \sum_{\text{margin}=-1} e^{\beta}\Big).\end{aligned}$$

Define

$$\text{err} \;=\; \frac{\#\{\text{incorrectly classified points}\}}{n}.$$

and

$$Q(\beta) = \frac{1}{n}\sum_{i=1}^{n} L\big(y_i,\; \beta\, T(x_i)\big) \;=\; \text{err}\cdot\, e^{\,\beta} \;+\; (1-\text{err})\cdot\, e^{-\beta}.$$

Minimize with respect to β,

$$\frac{\mathrm{d}Q}{\mathrm{d}\beta} \;=\; \text{err}\, e^{\,\beta} \;-\; (1-\text{err})\, e^{-\,\beta} \;=\; 0.$$

$$\text{err}\, e^{\,2\beta} \;=\; 1-\text{err}. \quad\Longrightarrow\quad e^{\,2\beta} \;=\; \frac{1-\text{err}}{\text{err}}.$$

$$2\,\beta \;=\; \log\Big(\tfrac{1-\text{err}}{\text{err}}\Big) \quad\Longrightarrow\quad \beta^{*} = \beta_1 \;=\; \tfrac{1}{2}\,\log\Big(\tfrac{1-\text{err}}{\text{err}}\Big).$$

Here β_1 is the "amount of say" that the first tree has!

Stage m for $m > 2$:
At the $(m-1)^{th}$ stage, we have the current function:

$$f_{m-1}(x) \;=\; \sum_{j=1}^{m-1} \beta_j \, T_j(x).$$

We want to find T_m such that

$$\frac{1}{n}\sum_{i=1}^{n} L(y_i\ f_{m-1}(x_i)\ +\ \beta_m\, T_m(x_i)) = \frac{1}{n}\sum_{i=1}^{n} e^{-\,y_i\,[\,f_{m-1}(x_i)\,+\,\beta_m\,T_m(x_i)\,]}$$

is minimized.
We can rewrite:

$$\frac{1}{n}\sum_{i=1}^{n} e^{-\,y_i\,[\,f_{m-1}(x_i)\,+\,\beta_m\,T_m(x_i)\,]} = \frac{1}{n}\sum_{i=1}^{n} e^{-\,y_i\,f_{m-1}(x_i)}\,e^{-\,y_i\,\beta_m\,T_m(x_i)} \ =\ \frac{1}{n}\sum_{i=1}^{n} w_i\ e^{-\,y_i\,\beta_m\,T_m(x_i)}$$

where

$$w_i\ =\ e^{-\,y_i\,f_{m-1}(x_i)}.$$

Expanding this expression:

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n} w_i\ e^{-\,y_i\,\beta_m\,T_m(x_i)} &= \frac{1}{n}\Big[\sum_{i:\,y_i=T_m(x_i)} w_i\,e^{-\,\beta_m}\ +\ \sum_{i:\,y_i\neq T_m(x_i)} w_i\,e^{\,\beta_m}\Big] \\ &= \frac{1}{n}\left[(e^{\beta}-e^{-\beta})\sum_{i=1}^{n} w_i\mathbf{1}[y_i\neq T_m(x_i)] + e^{-\beta}\sum_{i=1}^{n} w_i\right]\end{aligned}$$

To minimize this, we want

$$\sum_{i=1}^{n} w_i\,\mathbf{1}[\,y_i\neq T_m(x_i)\,]$$

to be as small as possible. That is:

$$T_m^{*} = \arg\min_{T}\sum_{i=1}^{n} w_i\,\mathbf{1}[\,y_i\neq T(x_i)\,].$$

This is the tree that minimizes the weighted error.

The best β_m is, as before,

$$\beta_m\ =\ \tfrac{1}{2}\,\log\!\Big(\tfrac{1-\mathrm{err}_m}{\mathrm{err}_m}\Big),\quad \text{where}\quad \mathrm{err}_m\ =\ \frac{\sum_{i=1}^{n} w_i\,\mathbf{1}[\,y_i\neq T_m(x_i)\,]}{\sum_{i=1}^{n} w_i}.$$

When we update our weights for the next tree, we do:

$$w_i\ \leftarrow\ w_i\,\exp\!\big[-\,\beta_m\,y_i\,T_m(x_i)\big]\ =\ \begin{cases} w_i\,e^{-\,\beta_m}, & \text{if } y_i = T_m(x_i),\\ w_i\,e^{\,\beta_m}, & \text{if } y_i\neq T_m(x_i).\end{cases}$$

Showing equivalence to the α_m formulation:

We can redefine some quantities to show that this is totally equivalent to the α_m formulation. Define:

$$\alpha_m\ =\ 2\,\beta_m,$$

so that

$$\alpha_m = \log\Big(\frac{1-\text{err}_m}{\text{err}_m}\Big),$$

and redefine

$$w_i \;\leftarrow\; w_i \,\exp\big[\alpha_m \,\mathbf{1}(y_i \neq T_m(x_i))\big].$$

Then we're done.

Big Takeaway: Finally our fitted boosted tree looks like:

$$f(x) \;=\; \sum_{m=1}^{M} \beta_m \, T_m(x_i) \;=\; \tfrac{1}{2} \sum_{m=1}^{M} \alpha_m \, T_m(x_i) \quad \Longrightarrow \quad \hat{y} \;=\; \text{sign}\big(f(x)\big).$$

and since at the end of the day we are only using the sign of $f(x)$ for the hard prediction, multiplying by 2 doesn't change the final prediction $\hat{y}$.

Gradient Boosting

Gradient Boosting: General Framework (from AdaBoost)

- Gradient boosting can be viewed as a direct generalization of AdaBoost. Recall that AdaBoost constructs an additive model of the form

  $$f(x) = \sum_{m=1}^{M} \alpha_m T_m(x),$$

  where each weak learner $T_m(x)$ is chosen sequentially to reduce an exponential loss by reweighting the training observations. In this sense, AdaBoost fits an additive model by repeatedly correcting the current predictor's mistakes.

  Gradient boosting retains the stagewise, additive structure, but replaces the specific exponential loss used in AdaBoost with an arbitrary differentiable loss function. Rather than modifying observation weights, gradient boosting adds a multiple η of each new fitted tree T_m. Given training data $\{(x_i, y_i)\}_{i=1}^n$, the goal is to find a function $f(x)$ that minimizes the empirical risk

  $$\sum_{i=1}^{n} L\big(y_i, f(x_i)\big),$$

  where $L(y, f)$ is a chosen loss function.

- **Key idea of gradient boosting:**
  - Retains stagewise, additive structure of AdaBoost
  - Replaces the exponential loss with an arbitrary differentiable loss $L(y, f)$
  - Weight of new additive tree is governed by hyperparameter η instead of β_m
- **Empirical risk minimization:**
  - Given data $\{(x_i, y_i)\}_{i=1}^n$, minimize with respect to $f(x_i)$

    $$\sum_{i=1}^{n} L\big(y_i, f(x_i)\big)$$

  - Optimization is performed in function space, approximated by parameter space

- **Additive model construction:**
  - Initialize $f_0(x)$ to minimize the loss under constant predictions
  - Update sequentially:
$$f_m(x) = f_{m-1}(x) + \nu\, T_m(x)$$
  - $h_m(x)$: weak learner (typically a shallow decision tree)
  - $\nu > 0$: learning rate controlling step size
  - Smaller $\nu \Rightarrow$ slower learning

- **Pseudo-residuals:**
  - At iteration m, compute
$$r_{im} = -\left.\frac{\partial L\big(y_i, f(x_i)\big)}{\partial f(x_i)}\right|_{f=f_{m-1}}$$

- **Fitting the next weak learner:**
  - Fit $T_m(x)$ to the pseudo-residuals $\{r_{im}\}$
  - Each learner approximates the direction of steepest descent
  - Mirrors AdaBoost's "focus on mistakes," but via gradients instead of reweighting

- **Why decision trees are used:**
  - Naturally model nonlinearities and interactions
  - Shallow trees remain weak learners
  - Optimizing in function space becomes easy as shallow trees have few parameters

Ex: Gradient Boosting for Binary Classification

We're going to derive gradient boosting for binary classification. The loss function for binary classification for $y \in \{1, -1\}$ is

$$L(y, f) = \log\big(1 + e^{-y\, f(x)}\big).$$

For reference find the gradient boosting algorithm at the end of these notes.

Formula for f_0

Claim: The best constant for binary classification in this setting

$$f_0 = \log\Big(\frac{1+\bar{y}}{1-\bar{y}}\Big).$$

Proof: The best constant is the α that minimizes the following expression:

$$Q(\alpha) = \sum_{i=1}^{N} \log\big(1 + e^{-y_i\, \alpha}\big). \tag{1}$$

To find the optimal α we take the derivative of (1) with respect to α and set it equal to 0:

$$Q'(\alpha) = -\sum_{i=1}^{N} \frac{y_i}{1 + e^{y_i \alpha}} = 0. \tag{2}$$

Let N^+ be the number of training observations with $y = 1$ and N^- be the number of training observations with $y = -1$. Since y_i is either 1 or -1,

$$\sum_{i=1}^{N} \frac{y_i}{1 + e^{y_i \alpha}} \;=\; \frac{N^+}{1 + e^{\alpha}} \;-\; \frac{N^-}{1 + e^{-\alpha}} \;=\; 0.$$

So

$$\frac{N^+}{1 + e^{\alpha}} \;=\; \frac{N^-}{1 + e^{-\alpha}}.$$

By multiplying the right part of the equation by $\frac{e^{\alpha}}{e^{\alpha}}$ we observe $N^+ = N^- e^{\alpha}$. Finally we arrive at

$$\alpha = \log\big(\frac{N^+}{N^-}\big).$$

Note:

$$\log\Big(\frac{1 + \bar{y}}{1 - \bar{y}}\Big) \;=\; \log\big(\frac{N^+}{N^-}\big),$$

where $\bar{y}$ is the mean of the targets so $\bar{y} = \frac{N^+ - N^-}{N}$, with $N = N^+ + N^-$. Therefore:

$$\frac{1 + \bar{y}}{1 - \bar{y}} = \frac{N^+}{N^-}.$$

Formula for pseudo-residuals

The general formula for the pseudo-residual is:

$$r_i = -\Big[\frac{\partial L(y, f)}{\partial f}\Big]_{f = f_{m-1}(x_i), y = y_i}$$

First, since

$$L(y, f) = \log\big(1 + e^{-y\, f}\big)$$

and

$$\frac{\partial L(y, f)}{\partial f} = -\frac{y}{1 + e^{y\, f}}$$

the pseudo-residual r_i is:

$$r_i = \frac{y_i}{1 + e^{\,y_i\, f_{m-1}(x_i)}}.$$

Formula for the best constant per region

The next step is to find the optimal constant per region. That is

$$\beta_j = \arg\min_{\beta} L_{R_j}(\beta)$$

where

$$L_{R_j}(\beta) = \sum_{x_i \in R_j} L\big(y_i,\, f_{m-1}(x_i) + \beta\big) = \sum_{x_i \in R_j} \log\Big(1 + e^{-\,y_i \left[f_{m-1}(x_i) + \beta\right]}\Big)$$

In the same way as before, we take the derivative of $L_{R_j}(\beta)$ with respect to β and set it equal to 0. Call this function $G(\beta)$:

$$G(\beta) = L'_{R_j}(\beta) = -\sum_{x_i \in R_j} \frac{y_i}{1 + e^{\,y_i \left[f_{m-1}(x_i) + \beta\right]}}.$$

The equation $G(\beta) = 0$ does not have a closed form solution, but the optimal β can be approximated by

$$\beta_j \;\approx\; -\frac{G(0)}{G'(0)}.$$

Why?

Let's do a Taylor expansion at $\beta = 0$:

$$G(\beta) = G(0) + (\beta - 0)\, G'(0) + O(\beta^2).$$

So if we truncate it into a first-order Taylor expansion, we see that approximately ,

$$G(\beta) \;\approx\; G(0) \;+\; \beta\, G'(0) \;=\; 0.$$

Thus, an approximate solution to $G(\beta) = 0$ and therefore an approximate ideal constant for the region R_j is:

$$\beta_j \;\approx\; -\frac{G(0)}{G'(0)}.$$

Calculating β_j:

Note that the numerator:

$$-\,G(0) = \sum_{x_i \in R_j} \frac{y_i}{1 + e^{\,y_i\, f_{m-1}(x_i)}} = \sum_{x_i \in R_j} r_i.$$

Then turning our attention to the denominator we can show that:

$$G'(\beta) = \sum_{x_i \in R_j} \frac{\big(y_i\big)^2\, e^{\,y_i\,[\,f_{m-1}(x_i) + \beta\,]}}{\Big(1 + e^{\,y_i\,[\,f_{m-1}(x_i) + \beta\,]}\Big)^2} = |r_i|(1 - |r_i|).$$

So finally the optimal β for region R_j is:

$$\beta_j \approx -\,\frac{G(0)}{G'(0)} = \frac{\sum_{x_i \in R_j} r_i}{\sum_{x_i \in R_j} |r_i|\,\big(1 - |r_i|\big)}.$$

Gradient Tree Boosting Algorithm

---

procedure GRADIENT TREE BOOSTING

1: Initialize $f_0(x) = \arg\min_{\beta} \sum_{i=1}^{N} L\big(y_i, \beta\big)$

2: **for** $m = 1$ **to** M **do**

3: Compute the pointwise negative gradient of the loss function at the current fit:

$$r_i = -\Big[\frac{\partial L(y, f)}{\partial f}\Big]_{f=f_{m-1}(x_i), y=y_i} \quad \text{for } i = 1, 2, \ldots, n$$

4: Approximate the negative gradient by fitting a regression tree to the targets r_i, giving terminal regions R_{jm}, $j = 1, \ldots, J_m$.

5: Compute new predictions for every terminal node. For $j = 1, \ldots, J_m$ compute

$$\beta_{jm} = \arg\min_{\beta} \sum_{x_i \in R_{jm}} L\big(y_i, f_{m-1}(x_i) + \beta\big)$$

6: Update $f_m(x) = f_{m-1}(x) \; + \eta \sum_{j=1}^{J_m} \beta_{jm} \, \mathbf{1}\big(x \in R_{jm}\big)$

7: **end for loop**

8: **return** $f(x) = f_M(x)$

Practical Implementation

There are several modern implementations and modifications of classical gradient boosting. Common and popular ones include XGBoost, LightGBM, and CatBoost. Eacch boosting framework has its own strengths; XGBoost is well-rounded, LightGBM excels in large-scale datasets, and CatBoost simplifies categorical data handling.

While XGBoost, LightGBM, and CatBoost all implement gradient boosting with decision trees, their design choices lead to important practical differences. Understanding these differences helps explain when each method is most effective. A summary of the tradeoffs and ideal use cases is provided in the table below.

Feature	XGBoost	LightGBM	CatBoost
Tree Growth Strategy	Level-wise expansion	Leaf-wise growth	Symmetric (Oblivious) Trees
Training Speed	Slower than LightGBM	Fastest due to leaf-wise approach	Competitive but slower than LightGBM
Prediction Accuracy	Strong general-purpose performance	High accuracy	Best for categorical data
Handling Categorical Features	Requires manual encoding	Requires manual encoding	Built-in categorical handling
Overfitting Risk	Lower due to balanced growth	Higher due to deep trees	Lower due to feature permutations
GPU Support	Good acceleration	Best performance on GPUs	Decent support
Ease of Use	Moderate; requires tuning	Moderate; requires encoding	Easiest due to automatic categorical handlin
Best Use Cases	General-purpose ML	Large datasets requiring speed	Categorical-heavy datasets

Table 5: Comparison of XGBoost, LightGBM, and CatBoost