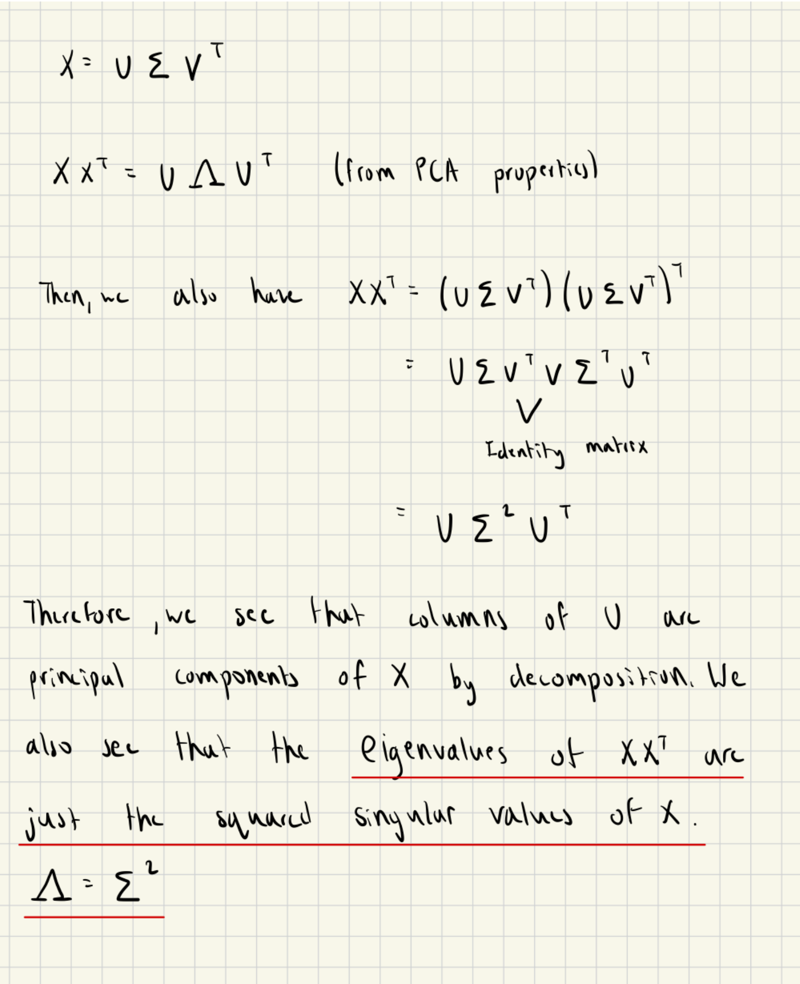
1 SVD and PCA [35 Points]

Relevant materials: Lectures 10, 11

Problem A [3 points]: Let X be a $N \times N$ matrix. For the singular value decomposition (SVD) $X = U\Sigma V^T$, show that the columns of U are the principal components of X. What relationship exists between the singular values of X and the eigenvalues of XX^T?

Solution A:



$X = U \Sigma V^T$

$XX^T = U \Lambda U^T$ (from PCA properties)

Then, we also have $XX^T = (U\Sigma V^T)(U \Sigma V^T)^T$

$$= U \Sigma \underbrace{V^T V}_{\text{Identity matrix}} \Sigma^T U^T$$

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

Therefore, we see that columns of U are principal components of X by decomposition. We also see that the eigenvalues of XX^T are just the squared singular values of X.

$\Lambda = \Sigma^2$

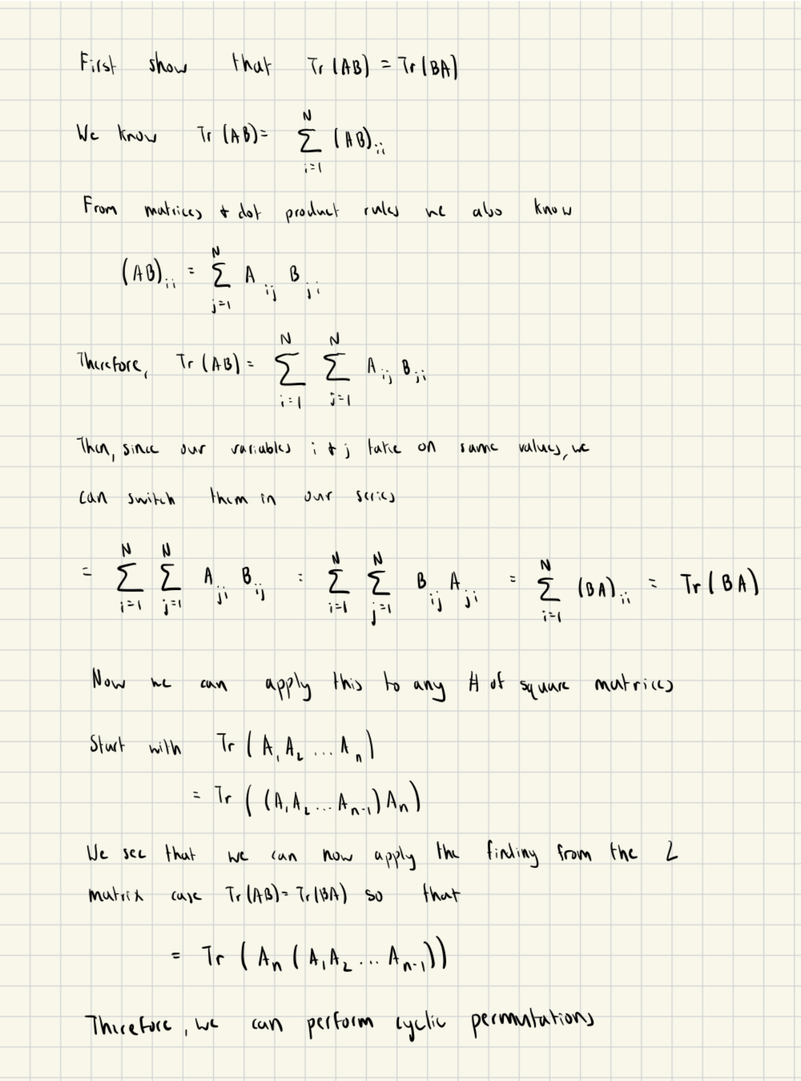
Problem B [4 points]: Provide both an intuitive explanation and a mathematical justification for why the eigenvalues of the PCA of X (or rather XX^T) are non-negative. Such matrices are called positive semi-definite and possess many other useful properties.

Solution B: *Intuitively, we know that the eigenvalues of XX^T are geometrically represented as the variances of the principal components of the data. Variances are always non-negative. Mathematically, we just proved that the eigenvalues of XX^T are the squared singular values of X. Any squared value is necessarily non-negative.*

Problem C [5 points]: In calculating the Frobenius and trace matrix norms, we claimed that the trace is invariant under cyclic permutations (i.e., Tr(ABC) = Tr(BCA) = Tr(CAB)). Prove that this holds for any number of square matrices.

Hint: First prove that the identity holds for two matrices and then generalize. Recall that $\text{Tr}(AB) = \sum_{i=1}^{N}(AB)_{ii}$. Can you find a way to expand $(AB)_{ii}$ in terms of another sum?

Solution C:



First show that $Tr(AB) = Tr(BA)$

We know $Tr(AB) = \sum_{i=1}^{N} (AB)_{ii}$

From matrices + dot product rules we also know

$$(AB)_{ii} = \sum_{j=1}^{N} A_{ij} B_{ji}$$

Therefore, $Tr(AB) = \sum_{i=1}^{N} \sum_{j=1}^{N} A_{ij} B_{ji}$

Then, since our variables i + j take on same values, we can switch them in our series

$$= \sum_{i=1}^{N} \sum_{j=1}^{N} A_{ji} B_{ij} = \sum_{i=1}^{N} \sum_{j=1}^{N} B_{ij} A_{ji} = \sum_{i=1}^{N} (BA)_{ii} = Tr(BA)$$

Now we can apply this to any # of square matrices

Start with $Tr(A_1 A_2 \ldots A_n)$

$$= Tr((A_1 A_2 \ldots A_{n-1}) A_n)$$

We see that we can now apply the finding from the 2 matrix case $Tr(AB) = Tr(BA)$ so that

$$= Tr(A_n (A_1 A_2 \ldots A_{n-1}))$$

Therefore, we can perform cyclic permutations

Problem D [3 points]: Outside of learning, the SVD is commonly used for data compression. Instead of storing a full $N \times N$ matrix X with SVD $X = U\Sigma V^T$, we store a truncated SVD consisting of the k largest singular values of Σ and the corresponding columns of U and V. One can prove that the SVD is the best rank-k approximation of X, though we will not do so here. Thus, this approximation can often re-create the matrix well even for low k. Compared to the N^2 values needed to store X, how many values do we need to store a truncated SVD with k singular values? For what values of k is storing the truncated SVD more efficient than storing the whole matrix?

Hint: For the diagonal matrix Σ, do we have to store every entry?

Solution D: *To store a truncated SVD with k singular values, we need to store $2Nk + k$ values. This is because for the matrices U and V, we simply need to store k columns with N values in each, yielding $2 * (Nk) = 2Nk$. Then, we only need to store k values from Σ. This yields a total of $2Nk + k$.*

We know that we need N^2 values to store X. So for the truncated SVD to be more efficient

$$k(2N+1) < N^2$$

$$k < \frac{N^2}{2N+1}$$

Dimensions & Orthogonality

In class, we claimed that a matrix X of size $D \times N$ can be decomposed into $U\Sigma V^T$, where U and V are orthogonal and Σ is a diagonal matrix. This is a slight simplification of the truth. In fact, the singular value decomposition gives an orthogonal matrix U of size $D \times D$, an orthogonal matrix V of size $N \times N$, and a rectangular diagonal matrix Σ of size $D \times N$, where Σ only has non-zero values on entries $(\Sigma)_{ii}$, $i \in \{1, \ldots, K\}$, where K is the rank of the matrix X.

Problem E [3 points]: Assume that $D > N$ and that X has rank N. Show that $U\Sigma = U'\Sigma'$, where Σ' is the $N \times N$ matrix consisting of the first N rows of Σ, and U' is the $D \times N$ matrix consisting of the first N columns of U. The representation $U'\Sigma'V^T$ is called the "thin" SVD of X.

Solution E: *We are given that $D > N$ and that X has rank N. From this. we know that the last $D - N$ rows of Σ are comprised of entirely zeros. Therefore Σ' is Σ without the $D - N$ rows of zeros. We know that for the original matrix multiplication UΣ, the $D - N$ zero rows in Σ effectively mean that the values after the N^{th} column of each row of U will not provide any purpose. This is because the entries of a matrix product are the dot product of rows of U with columns of Σ and the zeros in the last $D - N$ rows of Σ line up with the $D - N$ columns of U. Thus, we can effectively truncate the $D \times D$ matrix U to $D \times N$ U'. Therefore, we have that $U\Sigma = U'\Sigma'$.*

Problem F [3 points]: Show that since U' is not square, it cannot be orthogonal according to the definition given in class. Recall that a matrix A is orthogonal if $AA^T = A^TA = I$.

Solution F: $AA^T = U'U'^T = (D \times N) \times (N \times D) \in M_{D\times D}$. *On the other hand,* $A^TA = U'^TU' = (N \times D) \times (D \times N) \in M_{N\times N}$. *Therefore, because the dimensions don't match we have that* $AA^T \neq A^TA$

Problem G [4 points]: Even though U' is not orthogonal, it still has similar properties. Show that $U'^TU' = I_{N\times N}$. Is it also true that $U'U'^T = I_{D\times D}$? Why or why not? Note that the columns of U' are still orthonormal. Also note that orthonormality implies linear independence.

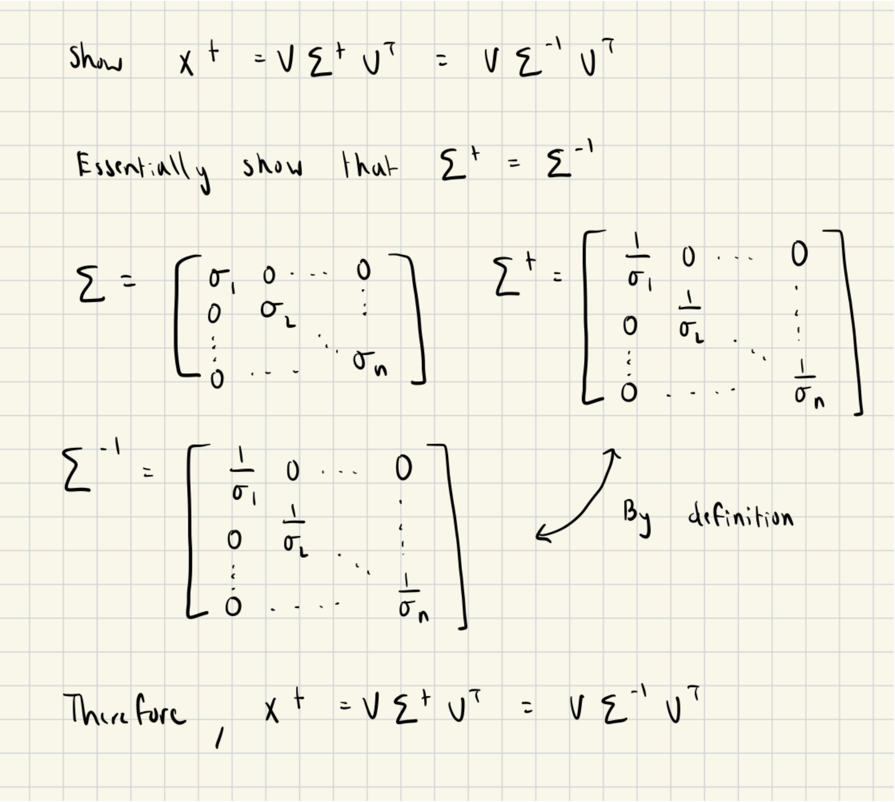
Solution G: *Since the columns of U' are still orthonormal, we know that the dot product of a column in U' (row of U'^T) with its corresponding row in U'^T (column of U') will result in a value of 1, but every other column product will result in 0. Therefore, we find that $U'^TU' = I_{N\times N}$. However, it is not also true that $U'U'^T = I_{D\times D}$? This is because the rows of U' are not orthonormal because $D > N$ and therefore we can only have N pivot rows. Since the rows are not orthonormal, the same dot product properties we applied earlier no longer hold and we are not guaranteed to get the identity matrix.*

Pseudoinverses

Let X be a matrix of size $D \times N$, where $D > N$, with "thin" SVD $X = U\Sigma V^T$. Assume that X has rank N.

Problem H [4 points]: Assuming that Σ is invertible, show that the pseudoinverse $X^+ = V\Sigma^+U^T$ as given in class is equivalent to $V\Sigma^{-1}U^T$. Refer to lecture 10 (slide 53) for the definition of pseudoinverse.

Solution H:



Show $X^+ = V\Sigma^+U^T = V\Sigma^{-1}U^T$

Essentially show that $\Sigma^+ = \Sigma^{-1}$

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & \sigma_n \end{bmatrix} \qquad \Sigma^+ = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2} & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & \frac{1}{\sigma_n} \end{bmatrix}$$

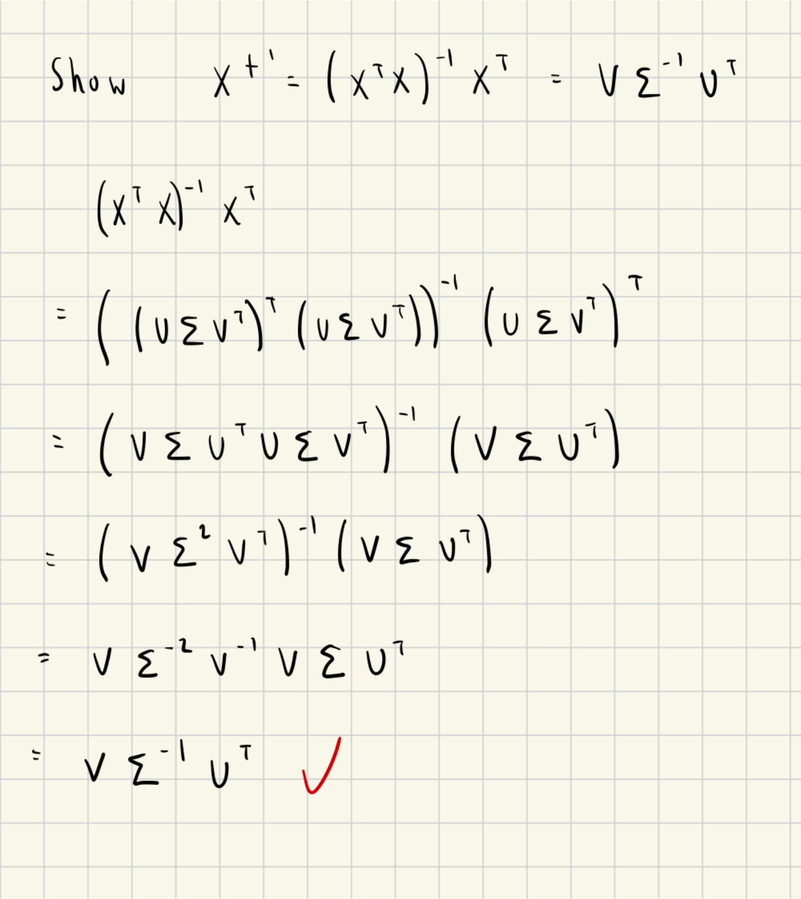
$$\Sigma^{-1} = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2} & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & \frac{1}{\sigma_n} \end{bmatrix}$$

By definition

Therefore, $X^+ = V\Sigma^+U^T = V\Sigma^{-1}U^T$

Problem I [4 points]: Another expression for the pseudoinverse is the least squares solution $X^{+'} = (X^TX)^{-1}X^T$. Show that (again assuming Σ invertible) this is equivalent to $V\Sigma^{-1}U^T$.

Solution I:



Show $X^{+'} = (X^TX)^{-1}X^T = V\Sigma^{-1}U^T$

$$(X^TX)^{-1}X^T$$

$$= \left((U\Sigma V^T)^T(U\Sigma V^T)\right)^{-1}(U\Sigma V^T)^T$$

$$= (V\Sigma U^TU\Sigma V^T)^{-1}(V\Sigma U^T)$$

$$= (V\Sigma^2V^T)^{-1}(V\Sigma U^T)$$

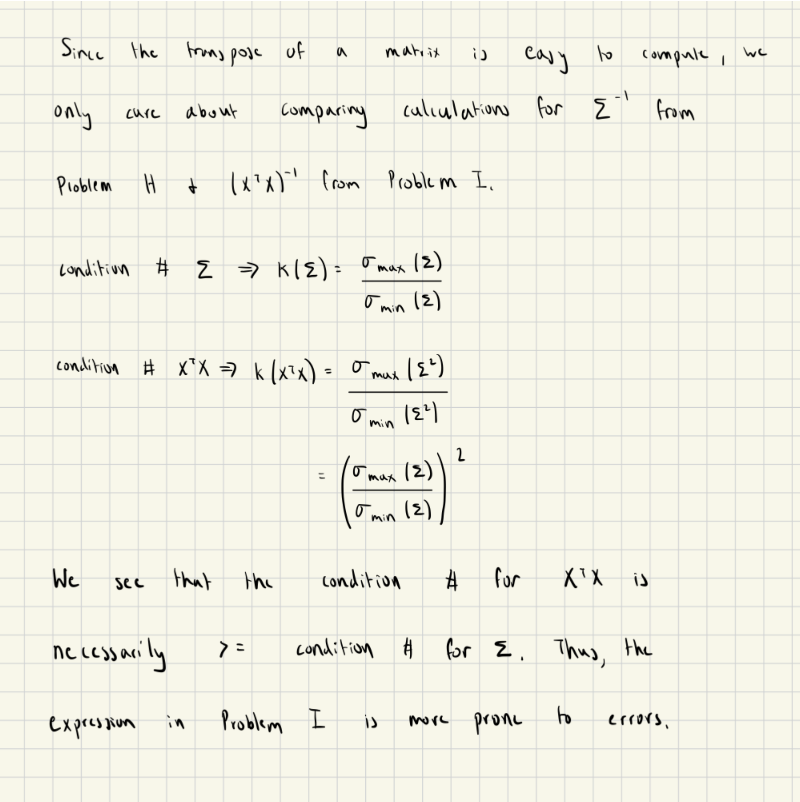
$$= V\Sigma^{-2}V^{-1}V\Sigma U^T$$

$$= V\Sigma^{-1}U^T \quad ✓$$

Problem J [2 points]: One of the two expressions in problems H and I for calculating the pseudoinverse is highly prone to numerical errors. Which one is it, and why? Justify your answer using condition numbers.

Hint: Note that the transpose of a matrix is easy to compute. Compare the condition numbers of Σ and $X^T X$. The condition number of a matrix A is given by $\kappa(A) = \frac{\sigma_{max}(A)}{\sigma_{min}(A)}$, where $\sigma_{max}(A)$ and $\sigma_{min}(A)$ are the maximum and minimum singular values of A, respectively.

Solution J:



Since the transpose of a matrix is easy to compute, we only care about comparing calculations for Σ^{-1} from Problem H & $(X^TX)^{-1}$ from Problem I.

condition # $\Sigma \Rightarrow \kappa(\Sigma) = \dfrac{\sigma_{max}(\Sigma)}{\sigma_{min}(\Sigma)}$

condition # $X^TX \Rightarrow \kappa(X^TX) = \dfrac{\sigma_{max}(\Sigma^2)}{\sigma_{min}(\Sigma^2)}$

$$= \left(\frac{\sigma_{max}(\Sigma)}{\sigma_{min}(\Sigma)}\right)^2$$

We see that the condition # for X^TX is necessarily >= condition # for Σ. Thus, the expression in Problem I is more prone to errors.

2 Matrix Factorization [30 Points]

Relevant materials: Lecture 11

In the setting of collaborative filtering, we derive the coefficients of the matrices $U \in \mathbb{R}^{M\times K}$ and $V \in \mathbb{R}^{N\times K}$ by minimizing the regularized square error:

$$\arg\min_{U,V} \frac{\lambda}{2}\left(\|U\|_F^2 + \|V\|_F^2\right) + \frac{1}{2}\sum_{i,j}\left(y_{ij} - u_i^T v_j\right)^2$$

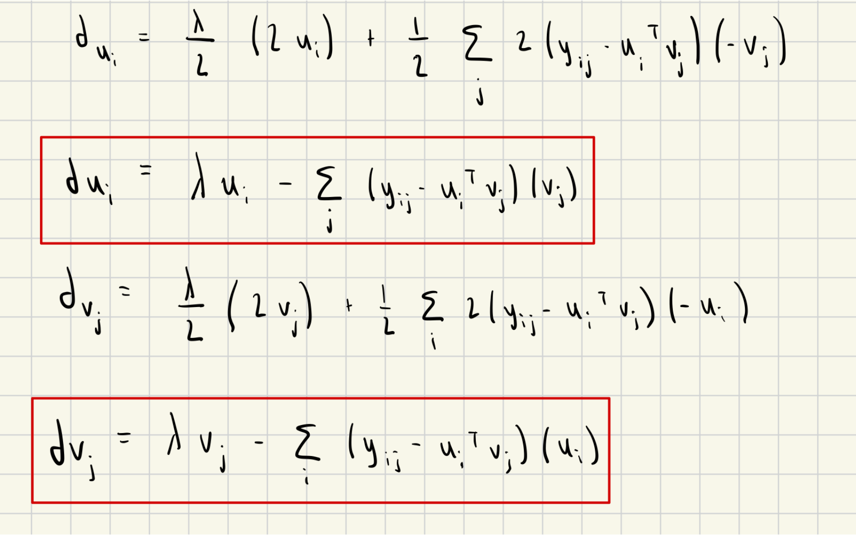
where u_i^T and v_j^T are the i^{th} and j^{th} rows of U and V, respectively, and $\|\cdot\|_F$ represents the Frobenius norm. Then $Y \in \mathbb{R}^{M\times N} \approx UV^T$, and the *ij*-th element of Y is $y_{ij} \approx u_i^T v_j$.

Problem A [5 points]: Derive the gradients of the above regularized squared error with respect to u_i and v_j, denoted ∂_{u_i} and ∂_{v_j} respectively. We can use these to compute U and V by stochastic gradient descent using the usual update rule:

$$u_i = u_i - \eta\partial_{u_i}$$
$$v_j = v_j - \eta\partial_{v_j}$$

where η is the learning rate.

Solution A:



$$\partial_{u_i} = \frac{\lambda}{2}(2u_i) + \frac{1}{2}\sum_j 2(y_{ij} - u_i^T v_j)(-v_j)$$

$$\boxed{\partial u_i = \lambda u_i - \sum_j (y_{ij} - u_i^T v_j)(v_j)}$$

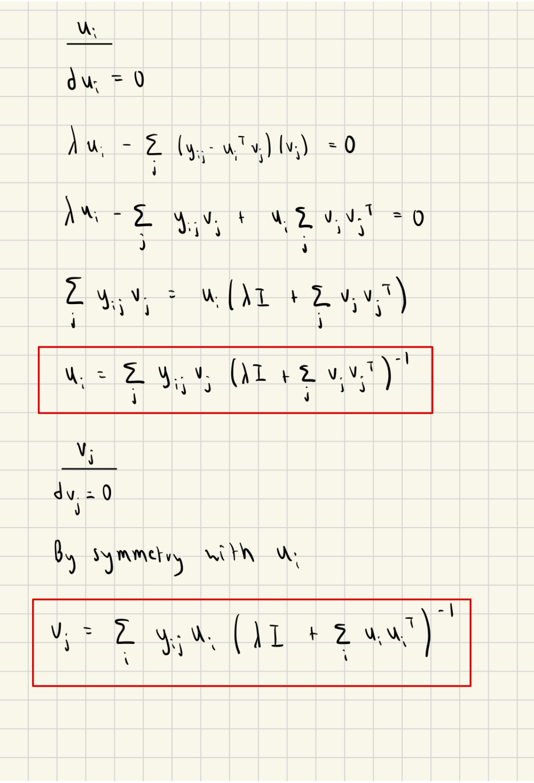
$$\partial_{v_j} = \frac{\lambda}{2}(2v_j) + \frac{1}{2}\sum_i 2(y_{ij} - u_i^T v_j)(-u_i)$$

$$\boxed{\partial v_j = \lambda v_j - \sum_i (y_{ij} - u_i^T v_j)(u_i)}$$

Problem B [5 points]: Another method to minimize the regularized squared error is alternating least squares (ALS). ALS solves the problem by first fixing U and solving for the optimal V, then fixing this new V and solving for the optimal U. This process is repeated until convergence.

Derive closed form expressions for the optimal u_i and v_j. That is, give an expression for the u_i that minimizes the above regularized square error given fixed V, and an expression for the v_j that minimizes it given fixed U.

Solution B:



$$\frac{u_i}{du_i} = 0$$

$$\lambda u_i - \sum_j (y_{ij} - u_i^T v_j)(v_j) = 0$$

$$\lambda u_i - \sum_j y_{ij} v_j + u_i \sum_j v_j v_j^T = 0$$

$$\sum_j y_{ij} v_j = u_i \left(\lambda I + \sum_j v_j v_j^T\right)$$

$$\boxed{u_i = \sum_j y_{ij} v_j \left(\lambda I + \sum_j v_j v_j^T\right)^{-1}}$$

$$\frac{v_j}{dv_j} = 0$$

By symmetry with u_i

$$\boxed{v_j = \sum_i y_{ij} u_i \left(\lambda I + \sum_i u_i u_i^T\right)^{-1}}$$

Problem C [10 points]: Download the provided MovieLens dataset (train.txt and test.txt). The format of the data is (*user, movie, rating*), where each triple encodes the rating that a particular user gave to a particular movie. Make sure you check if the user and movie ids are 0 or 1-indexed, as you should with any real-world dataset.

Implement matrix factorization with stochastic gradient descent for the MovieLens dataset, using your answer from part A. Assume your input data is in the form of three vectors: a vector of is, js, and y_{ij}s. Set $\lambda = 0$ (in other words, do not regularize), and structure your code so that you can vary the number of latent factors (k). You may use the Python code template in 2_notebook.ipynb; to complete this problem, your task is to fill in the four functions in 2_notebook.ipynb marked with TODOs.

In your implementation, you should:

- Initialize the entries of U and V to be small random numbers; set them to uniform random variables in the interval $[-0.5, 0.5]$.

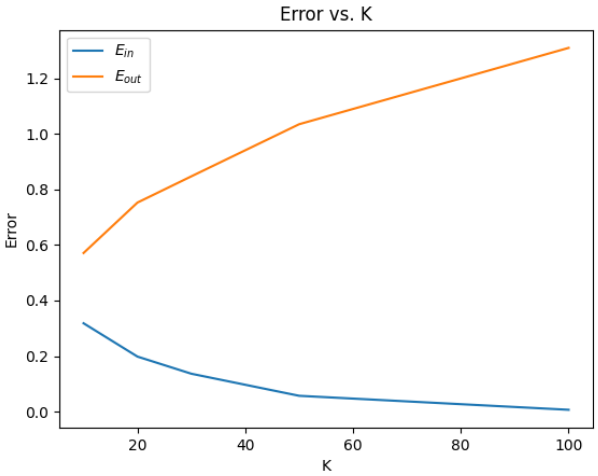
- Use a learning rate of 0.03.
- Randomly shuffle the training data indices before each SGD epoch.
- Set the maximum number of epochs to 300, and terminate the SGD process early via the following early stopping condition:
  - Keep track of the loss reduction on the training set from epoch to epoch, and stop when the relative loss reduction compared to the first epoch is less than $\epsilon = 0.0001$. That is, if $\Delta_{0,1}$ denotes the loss reduction from the initial model to end of the first epoch, and $\Delta_{i,i-1}$ is defined analogously, then stop after epoch t if $\Delta_{t-1,t}/\Delta_{0,1} \leq \epsilon$.

Solution C:

https://colab.research.google.com/drive/1wUEdIEEM7ewOMKKOgCjB3YB-T9T__b3u?usp=sharing

Problem D [5 points]: Use your code from the previous problem to train your model using $k = 10, 20, 30, 50, 100$, and plot your E_{in}, E_{out} against k. Note that E_{in} and E_{out} are calculated via the squared loss, i.e. via $\frac{1}{2}\sum_{i,j}\left(y_{ij} - u_i^T v_j\right)^2$. What trends do you notice in the plot? Can you explain them?

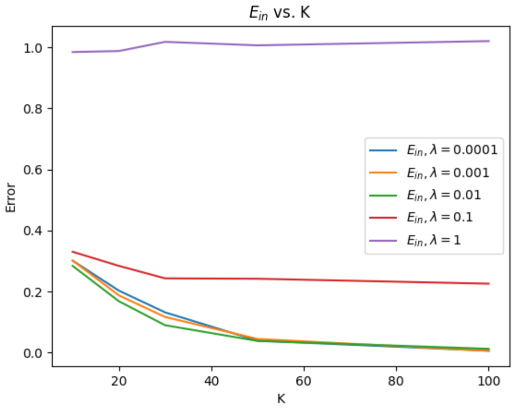
Solution D:

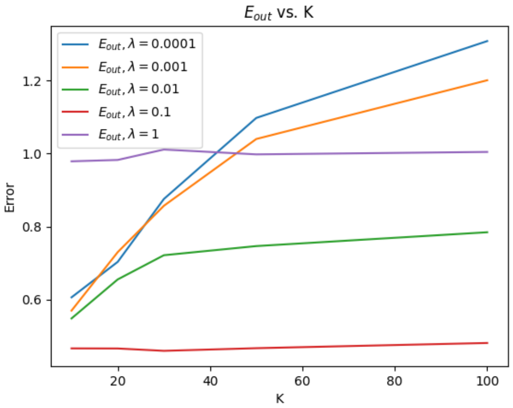


We notice that as K increases, E_{in} decreases while E_{out} increases. This makes sense because as we increase the rank of our latent factor model, we are prone to overfitting. As we allow the model to gain complexity and pick up on increasingly abstract mathematical factors, our ability to generalize decreases.

Problem E [5 points]: Now, repeat problem D, but this time with the regularization term. Use the following regularization values: $\lambda \in \{1e-4, 1e-3, 0.01, 0.1, 1\}$. For each regularization value, use the same range of values for k as you did in the previous part. What trends do you notice in the graph? Can you explain them in the context of your plots for the previous part? You should use your code you wrote for part C in 2_notebook.ipynb.

Solution E:





We generally see the same trends in these graph as we did in Part D. Namely, as K increases, E_{in} decreases while E_{out} increases. There are a few notable exceptions, however. We see that when the regularization parameter is extremely strong ($\lambda = 1$), E_{in} and E_{out} roughly stay constant at Error = 1 when K is increased. In general, we see that when the regularization parameter is increased, the decreasing trend for E_{in} and the increasing trend for E_{out} are weakened. In other words, the curves flatten out. This makes sense because the increased regularization leads to a lower sensitivity to other model factors. We see once again as we have in previous weeks that there is a sweet spot for regularization parameter, where too little causes overfitting and too much causes underfitting. The ideal λ value appears to be 0.1

3 Word2Vec Principles [35 Points]

Relevant materials: Lecture 12

The Skip–gram model is part of a family of techniques that try to understand language by looking at what words tend to appear near what other words. The idea is that semantically similar words occur in similar contexts. This is called "distributional semantics", or "you shall know a word by the company it keeps".

The Skip–gram model does this by defining a conditional probability distribution $p(w_O|w_I)$ that gives the probability that, given that we are looking at some word w_I in a line of text, we will see the word w_O nearby. To encode p, the Skip-gram model represents each word in our vocabulary as two vectors in $\mathbb{R}^D$: one vector for when the word is playing the role of w_I ("input"), and one for when it is playing the role of w_O ("output"). (The reason for the 2 vectors is to help training — in the end, mostly we'll only care about the w_I vectors.) Given these vector representations, p is then computed via the familiar softmax function:

$$p(w_O|w_I) = \frac{\exp\left({v'}_{w_O}^{T} v_{w_I}\right)}{\sum_{w=1}^{W} \exp\left({v'_w}^{T} v_{w_I}\right)} \tag{2}$$

where v_w and v'_w are the "input" and "output" vector representations of word a $w \in \{1, ..., W\}$. (We assume all words are encoded as positive integers.)

Given a sequence of training words $w_1, w_2, \ldots, w_T$, the training objective of the Skip-gram model is to maximize the average log probability

$$\frac{1}{T} \sum_{t=1}^{T} \sum_{-s \leq j \leq s, j \neq 0} \log p(w_{t+j}|w_t) \tag{1}$$

where s is the size of the "training context" or "window" around each word. Larger s results in more training examples and higher accuracy, at the expense of training time.

Problem A [5 points]: If we wanted to train this model with naive gradient descent, we'd need to compute all the gradients $\nabla \log p(w_O|w_I)$ for each w_O, w_I pair. How does computing these gradients scale with W, the number of words in the vocabulary, and D, the dimension of the embedding space? To be specific, what is the time complexity of calculating $\nabla \log p(w_O|w_I)$ for a single w_O, w_I pair?

Solution A: *The time complexity of calculating $\nabla \log p(w_O|w_I)$ for a single w_O, w_I pair is O(WD). This is because the summation term $\sum_{w=1}^{W} \exp\left({v'_w}^{T} v_{w_I}\right)$ involves the computation of W different dot products, each which take D computations themselves (D-dimensional).*

Problem B [10 points]: When the number of words in the vocabulary W is large, computing the regular softmax can be computationally expensive (note the normalization constant on the bottom of Eq. 2). For reference, the standard fastText pre-trained word vectors encode approximately $W \approx 218000$ words in

Table 1: Words and frequencies for Problem B

Word	Occurrences
do	18
you	4
know	7
the	20
way	9
of	4
devil	5
queen	6

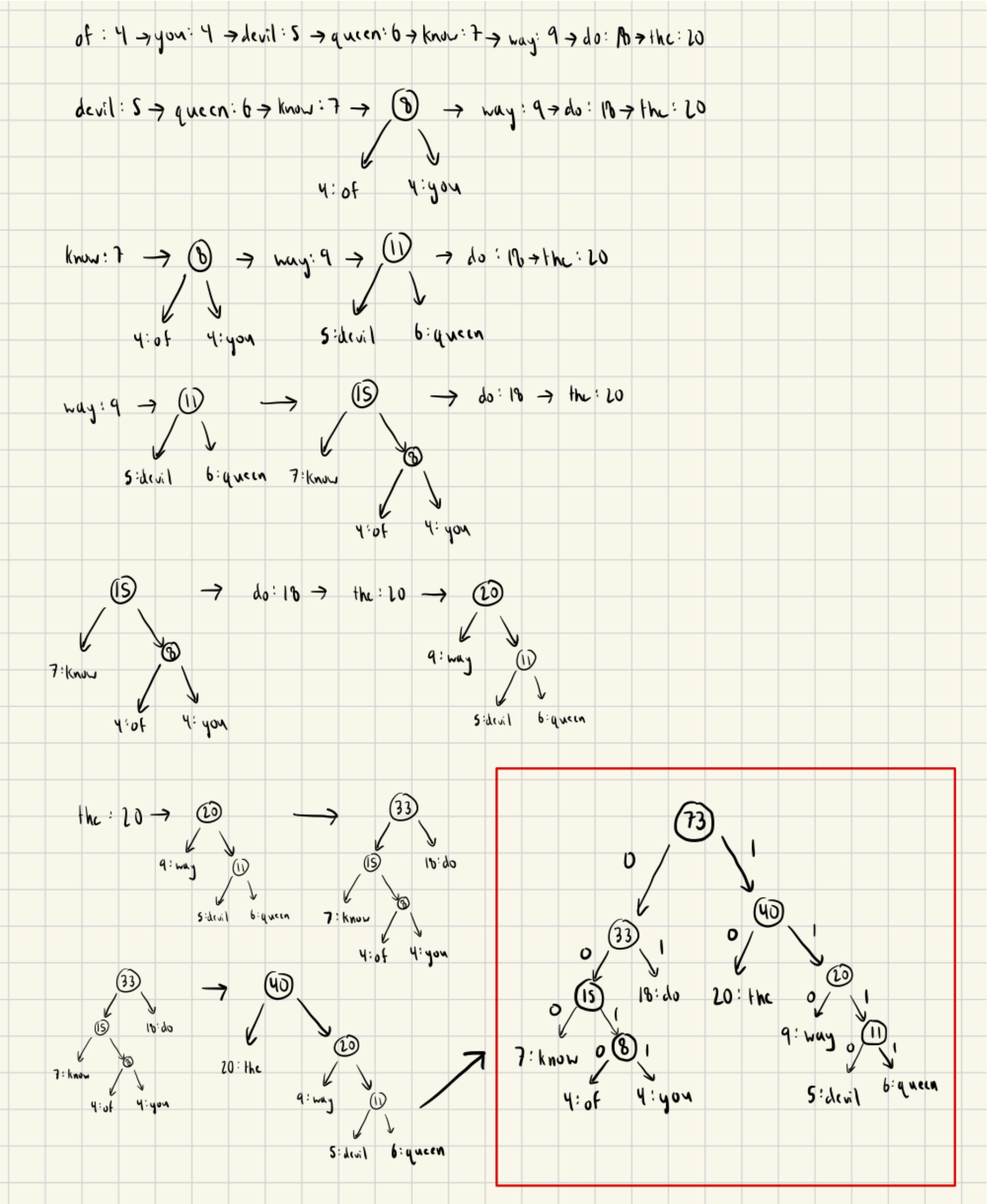
$D = 100$ latent dimensions. One trick to get around this is to instead represent the words in a binary tree format and compute the hierarchical softmax.

When the words have all the same frequency, then any balanced binary tree will minimize the average representation length and maximize computational efficiency of the hierarchical softmax. But in practice, words occur with very different frequencies — words like ”a”, ”the”, and ”in” will occur many more times than words like ”representation” or ”normalization”.

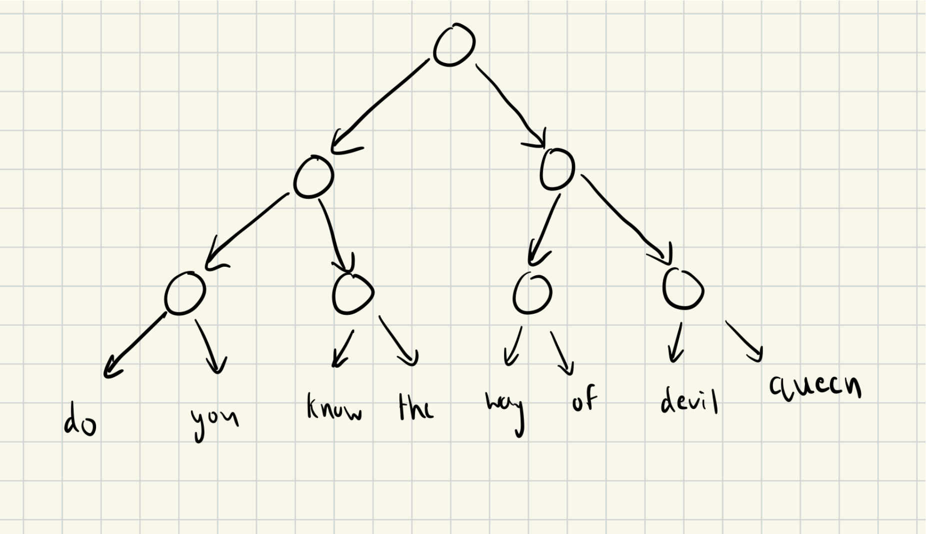
The original paper (Mikolov et al. 2013) uses a Huffman tree instead of a balanced binary tree to leverage this fact. For the 8 words and their frequencies listed in the table below, build a Huffman tree using the algorithm found here. Then, build a balanced binary tree of depth 3 to store these words. Make sure that each word is stored as a *leaf node* in the trees.

The representation length of a word is then the length of the path (the number of edges) from the root to the leaf node corresponding to the word. For each tree you constructed, compute the expected representation length (averaged over the actual frequencies of the words).

Solution B:



*Expected Representation length of Huffman Tree: (3 * 7 + 4 * 4 + 4 * 4 + 2 * 18 + 3 * 9 + 4 * 5 + 4 * 6 + 2 * 20) / 73 = 2.74.*



Expected Representation length of Balanced Tree: 3

Problem C [3 points]: In principle, one could use any D for the dimension of the embedding space. What do you expect to happen to the value of the training objective as D increases? Why do you think one might not want to use very large D?

Solution C: *As we increase D, the training accuracy will increase. Therefore, the training objective also increases. However, we must be cautious using a large D, because it will be more computationally expensive and the model will be prone to overfitting.*

Implementing Word2Vec

Word2Vec is an efficient implementation of the Skip–gram model using neural network–inspired training techniques. We'll now implement Word2Vec on text datasets using Keras. This blog post provides an overview of the particular Word2Vec implementation we'll use.

At a high level, we'll do the following:

(i) Load in a list L of the words in a text file

(ii) Given a window size s, generate up to $2s$ training points for word L_i. The diagram below shows an example of training point generation for $s = 2$:

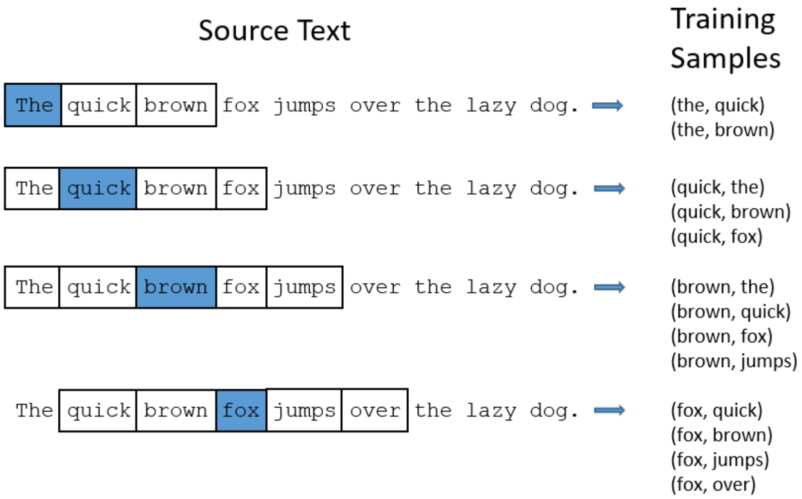


Figure 1: Generating Word2Vec Training Points

(iii) Fit a neural network consisting of a single hidden layer of 10 units on our training data. The hidden layer should have no activation function, the output layer should have a softmax activation, and the loss function should be the cross entropy function.

Notice that this is exactly equivalent to the Skip–gram formulation given above where the embedding dimension is 10: the columns (or rows, depending on your convention) of the input–to–hidden weight matrix in our network are the w_I vectors, and those of the hidden–to–output weight matrix are the w_O vectors.

(iv) Discard our output layer and use the matrix of weights between our input layer and hidden layer as the matrix of feature representations of our words.

(v) Compute the cosine similarity between each pair of distinct words and determine the top 30 pairs of most-similar words.

Implementation

See 3_notebook.ipynb, which implements most of the above.

Problem D [10 points]: Fill out the TODOs in the skeleton code; specifically, add code where indicated to train a neural network as described in (iii) above and extract the weight matrix of its input–to–hidden weight matrix. Also, fill out the generate_traindata() function, which generates our data and label matrices.

Solution D:

https://colab.research.google.com/drive/1vqjseaGMnQ_FILO6H9pUYiyH3hXx8uWE?usp=sharing

Running the code

Run your model on dr_seuss.txt and answer the following questions:

Problem E [2 points]: What is the dimension of the weight matrix of your hidden layer?

Solution E: *The dimension is 308 x 10.*

Problem F [2 points]: What is the dimension of the weight matrix of your output layer?

Solution F:

The dimension is 10 x 308.