Machine Learning & Data Mining
Set 5

Caltech CS/CNS/EE 155
February 2022

Homework submitted using 3 late hours. I have 45 late hours remaining.

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: *The solution to PCA is U such that $XX^T = U\Lambda U^T$.*

$$\begin{aligned} XX^T &= U\Sigma V^T(U\Sigma V^T)^T \\ &= U\Sigma V^T V\Sigma^T U^T \\ &= U\Sigma\Sigma^T U^T && \textit{since } V \textit{ is orthogonal} \\ &= U\Sigma^2 U^T && \textit{since } \Sigma \textit{ is a diagonal matrix} \end{aligned}$$

Since Σ is a diagonal matrix with the singular values of X, Σ^2 is a diagonal matrix that satisifies $XX^T = U\Lambda U^T$ with $\Lambda = \Sigma^2$, since U is orthogonal in both the singular value decomposition and the solution to PCA. Since Λ is a diagonal matrix that contains the eigenvalues of X, the singular values of X are the square roots of the eigenvalues of XX^T.

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: *We know that the eigenvalue*

$$\lambda_j = \sum_{i=1}^{N} (u_j^T x_i)^2$$

which can also be interpreted as the variation along the axis u_j. The variation of a variable must be non-negative, so the eigenvalues in the PCA of X are non-negative. Mathematically, since the eigenvalues of XX^T are the squares of the singular values of X (σ), they must be ≥ 0, since the square of any real number is 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: *We can produce any cyclic permutation of the product* $A_1A_2...A_n$ *by swapping 2 subproducts. Let the desired permutation be* $A_iA_{i+1}...A_nA_1...A_{i-1}$. *Then, let* $A' = A_iA_{i+1}...A_n$ *and* $B' = A_1...A_{i-1}$. *If we can show that* $\text{Tr}(A'B') = \text{Tr}\,(B'A')$, *then we have shown that* $\text{Tr}(A_1A_2...A_n) = \text{Tr}(A_1...A_{i-1})$ *for any arbitrary cyclic permutation of an arbitrary length product of matrices.*

To show that $\text{Tr}(AB) = \text{Tr}(BA)$, *we use the given definition.*

$$\begin{aligned}\text{Tr}(AB) &= \sum_{i=1}^{N}(AB)_{ii}\\ &= \sum_{i=1}^{N}\sum_{j=1}^{N}A_{ij}B_{ji}\\ &= \sum_{j=1}^{N}\sum_{i=1}^{N}B_{ji}A_{ij}\\ &= \sum_{j=1}^{N}(BA)_{jj}\\ &= \text{Tr}(BA)\end{aligned}$$

Thus, the trace is invariant under cyclic permutation.

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: *In the matrix U, we must store kN entries from the first k columns, since these are multiplied by the first k largest singular values. In the matrix V, we must store kN entries from the first k columns as well. We must also store k values from Σ, which are the largest singular values. Since Σ is a diagonal matrix, to store k singular values, we only need to store k values along the diagonal, since all other values are 0. Thus, in total we must store $2kN + k$ values.*

For $N^2 > 2kN + k$ (less values stored using SVD method), we need $k < \frac{N^2}{2N+1}$.

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: *Let $A = U\Sigma$. Then, by the definition of matrix multiplication,*

$$A_{ij} = \sum_{d=1}^{D} U_{id}\Sigma_{dj}$$

If X has rank N, then Σ only has non-zero values for Σ_{ii} with $1 \leq i \leq N$. Since $D > N$, the matrix multiplication becomes

$$A_{ij} = \sum_{d=1}^{N} U_{id}\Sigma_{dj}$$

Thus, only the first N columns of U, Σ have values referenced in the matrix product, so the other columns can be eliminated to produce U', Σ' which has the same matrix product.

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: *U'^TU' is an $N \times N$ matrix, which cannot be equal to $U'U'^T$, which is a $D \times D$ matrix.*

Problem G [4 points]: Even though U' is not orthogonal, it still has similar properties. Show that $U^{'T}U' = I_{N\times N}$. 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: *We know that the original U is orthogonal, so $U^TU = I$. In other words, for each column u_i in U, $u_i \cdot u_i = 1$ and for 2 different columns u_i, u_j (which are linearly independent, given by orthonormality), $u_i \cdot u_j = 0$. In the product of U'^TU', we are computing the dot product of columns of U, so the diagonal still has values of 1 and other entries have value 0, so the result is $I_{N\times N}$. In the product $U'U'^T$, we are computing the dot product of rows of U', and since there are more rows than columns, the rows of U' cannot be linearly independent, and thus not orthonormal. Since the rows are not orthonormal, either one of the values along the diagonal is not 1 or one of the other values is not 0, so the result is not $I_{D\times D}$.*

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 11 for the definition of pseudoinverse.

Solution H: *The matrix Σ has only σ_i values along the diagonal. Since the inverse of a matrix is a matrix such that $\Sigma^{-1}\Sigma = I$, each value along the diagonal of Σ^{-1} is $1/\sigma_i$, and since we assume the matrix is invertible, all $1/\sigma_i$ are real. Thus, Σ^{-1} is exactly the definition of the pseudo-inverse, where each value along the diagonal is $1/\sigma$ in the pseudo-inverse.*

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: *We will use $X = U\Sigma V^T$. We also know that the inverse of an orthogonal matrix is its transpose, and U, V are orthogonal. Additionally, for diagonal matrices, the inverse is simply the diagonal matrix of reciprocals.*

$$\begin{aligned}
X^{+'} &= (X^TX)^{-1}X^T \\
&= ((U\Sigma V^T)^T(U\Sigma V^T))^{-1}(U\Sigma V^T)^T \\
&= (V\Sigma^TU^TU\Sigma V^T)^{-1}V\Sigma^TU^T \\
&= (V\Sigma^2V^T)^{-1}V\Sigma U^T \\
&= (V^T)^{-1}(\Sigma^2)^{-1}V^{-1}V\Sigma U^T \\
&= V\Sigma^{-2}\Sigma U^T \\
&= V\Sigma^{-1}U^T
\end{aligned}$$

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^TX. 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: *The higher the condition number, the more prone to numerical error, since a smaller change in the input can lead to a larger variation in the output. The condition number for Σ, which is used in the method from part H, is simply $\kappa\left(\frac{\sigma_{max}(\Sigma)}{\sigma_{min}(\Sigma)}\right)$. The condition number for $X^TX = V^T\Sigma^2V$ used in method I is $\kappa\left(\left(\frac{\sigma_{max}(\Sigma)}{\sigma_{min}(\Sigma)}\right)^2\right)$. Since $\sigma_{max} \geq \sigma_{min}$, the condition number of X^TX is larger than the condition number of Σ, so computing the inverse of X^TX is more prone to errors than computing the inverse of Σ. Thus, the method presented in I is more prone to numerical 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: *We know that* $||U||_F^2 = \text{Tr}(U^T U) = \sum u_i \cdot u_i$. *Since we take the partial derivative with respect to* u_i, *we consider the other terms in the norm and the summation as constants. Then,*

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

Similarly,

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

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: *To do this, we set the gradient equal to 0 and solve.*

$$\begin{aligned}
\partial_{u_i} &= \lambda u_i - \sum_j (y_{ij} - u_i^T v_j) v_j \\
0 &= \lambda u_i - \sum_j y_{ij} v_j + \sum_j (u_i^T v_j) v_j \\
\lambda u_i &= \sum_j y_{ij} v_j - \sum_j v_j v_j^T u_i \\
\lambda u_i + u_i \sum_j v_j v_j^T &= \sum_j y_{ij} v_j \\
\left(\lambda I + \sum_j v_j v_j^T \right) u_i &= \sum_j y_{ij} v_j \\
u_i &= \left(\lambda I + \sum_j v_j v_j^T \right)^{-1} \left(\sum_j y_{ij} v_j \right)
\end{aligned}$$

Similarly,

$$\begin{aligned}
\partial_{v_j} &= \lambda v_j - \sum_i (y_{ij} - u_i^T v_j) u_i \\
0 &= \lambda v_j - \sum_i y_{ij} u_i + \sum_i (u_i^T v_j) u_i \\
\lambda v_j &= \sum_i y_{ij} u_i - v_j \sum_i u_i u_i^T \\
\left(\lambda I + \sum_i u_i u_i^T \right) v_j &= \sum_i y_{ij} u_i \\
v_j &= \left(\lambda I + \sum_i u_i u_i^T \right)^{-1} \left(\sum_i y_{ij} u_i \right)
\end{aligned}$$

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: *Code link*

https://colab.research.google.com/drive/1-UvQS4XmeSaR7l-eZAWVMhJoxWI-fMPv?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:

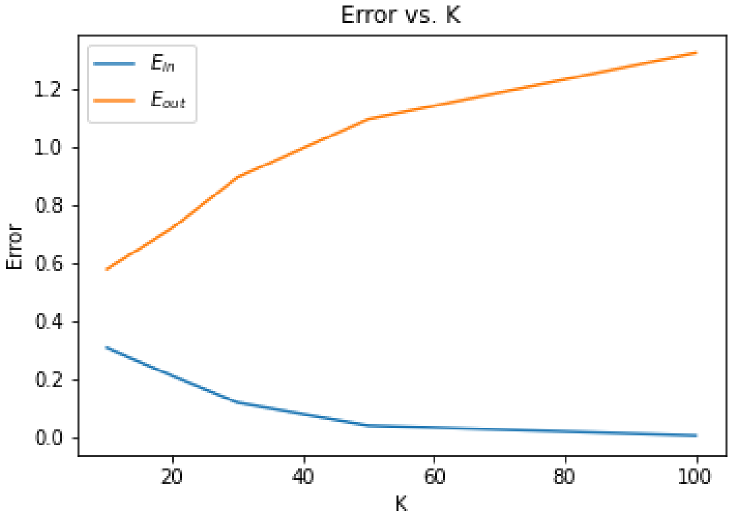

Figure 1: In-sample and out-of-sample error as a function of representation size.

As k increases, the in-sample error decreases while the out-of-sample error increases (the model overfits more). This is because the model has more latent factors to better fit the training (in-sample) data, but in doing so overfits the data and causes the out-of-sample error to increase.

Problem E [5 points]: Now, repeat problem D, but this time with the regularization term. Use the following regularization values: $\lambda \in \{10^{-4}, 10^{-3}, 0.01, 0.1, 1\}$. 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:

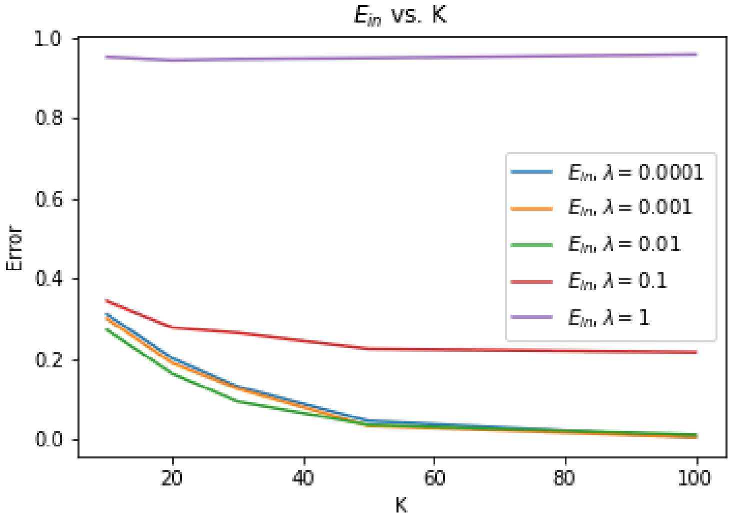

Figure 2: In-sample error for different regularization and k

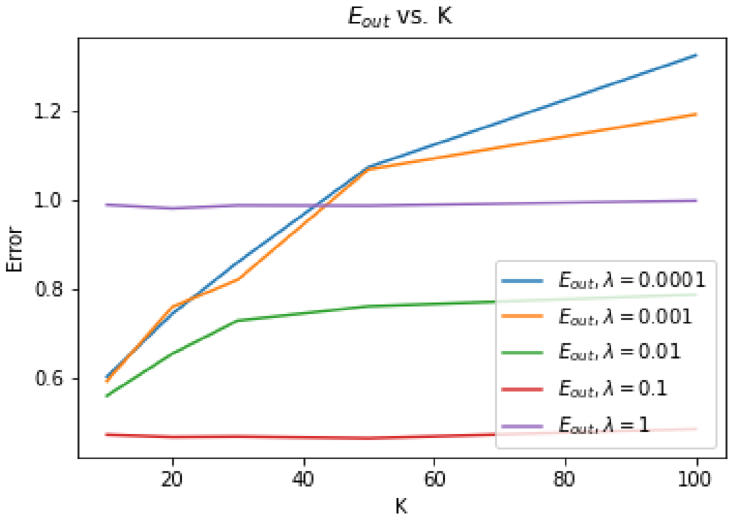

Figure 3: Out-of-sample error for different regularization and k

Generally, as k increases, as with the last problem, in-sample error decreases and out-of-sample error increases. A higher regularization strength slows the decrease of in-sample-error and decreases the growth of out-of-sample error as k increases, but a too high regularization value results in no additional learning as k increases, with a high in-sample and out-of-sample error ($\lambda = 1$). Thus, the regularization counteracts the overfitting caused by larger k, so that the error does not change even as k changes.

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:

$$\begin{aligned}
\nabla \log p(w_O|w_I) &= \nabla \log \frac{\exp(v'^T_O v_I)}{\sum_{w=1}^{W} \exp(v'^T_w v_{w_I})} \\
&= \nabla \left(v'^T_{w_O} v_{w_I} - \log \sum_{w=1}^{W} \exp(v'^T_w v_{w_I}) \right) \\
&= v_{w_I} - \frac{v_{w_I}}{\sum_{w=1}^{W} \exp(v'^T_w v_{w_I})} \exp v'^T_w v_{w_I}
\end{aligned}$$

Thus, for each word, the time complexity scales linearly by the dimension of the embedding space, since the dot product (scales linearly with D) must be computed for each word (summation in denominator). Thus, the overall complexity for calculating the gradients for a single pair of words is $O(WD)$.

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

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 $D = 100$ 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:

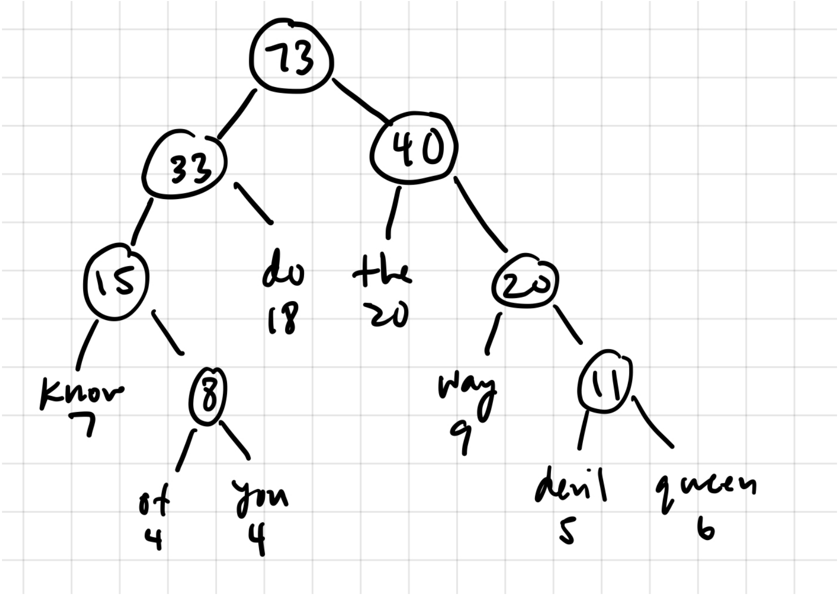

Figure 4: Huffman tree created using word frequencies.

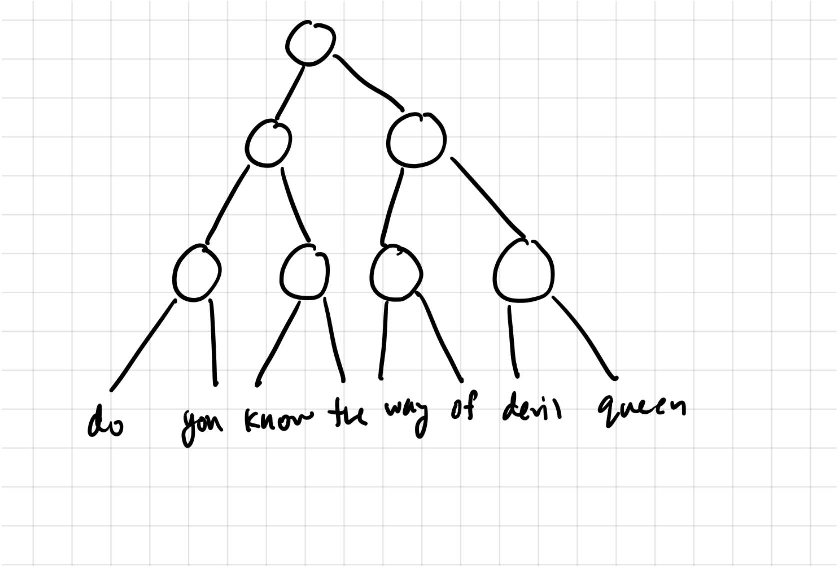

Figure 5: Balanced binary tree created using words.

The expected representation length of the Huffman tree is

$$\frac{1}{73}(2 \cdot 18 + 2 \cdot 20 + 3 \cdot 7 + 3 \cdot 9 + 4 \cdot 4 + 4 \cdot 4 + 4 \cdot 5 + 4 \cdot 6) = \boxed{2.7397}$$

The expected representation length of the balanced binary tree is 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: *The training objective value increases as D increases (less error), since there are more dimensions in the representation for the model to fit, thus lowering error. However, this may not be a good idea since the model could be prone to overfitting as a result.*

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 Pytorch. 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 $s = 2$:

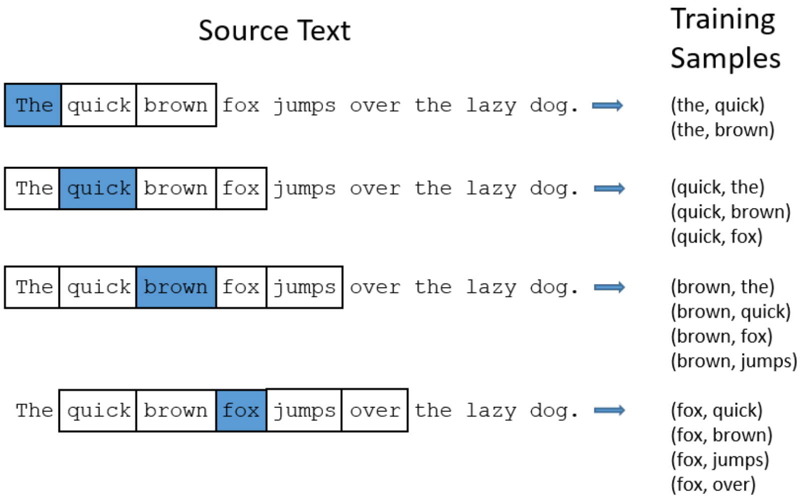

Figure 6: Generating Word2Vec Training Points

(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 set5_prob3.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: *Code link*

https://colab.research.google.com/drive/1_NBkNvw3P_O1XJZFMxLRgxDk9YdC8KkH?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 PyTorch tensor is* 10×308 *(however, PyTorch transposes the matrices, so the model really performs multiplication with a* 308×10 *matrix).*

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

Solution F: *The PyTorch tensor is* 308×10. *(Again, PyTorch transposes the matrix, so the actual matrix is* 10×308, *which matches the hidden layer dimension/output dimension)*

Problem G [1 points]: List the top 30 pairs of most similar words that your model generates.

Solution G:

- *Pair(or, anywhere), Similarity: 0.99958247*
- *Pair(anywhere, or), Similarity: 0.99958247*
- *Pair(eggs, ham), Similarity: 0.9990665*
- *Pair(ham, eggs), Similarity: 0.9990665*
- *Pair(do, them), Similarity: 0.9983842*
- *Pair(them, do), Similarity: 0.9983842*
- *Pair(star, lot), Similarity: 0.9981338*
- *Pair(lot, star), Similarity: 0.9981338*

- *Pair(car, train), Similarity: 0.99767697*
- *Pair(train, car), Similarity: 0.99767697*
- *Pair(am, sam), Similarity: 0.9976195*
- *Pair(sam, am), Similarity: 0.9976195*
- *Pair(dish, sam), Similarity: 0.99745214*
- *Pair(hop, am), Similarity: 0.9972182*
- *Pair(wish, hop), Similarity: 0.9971074*
- *Pair(go, gox), Similarity: 0.99676085*
- *Pair(gox, go), Similarity: 0.99676085*
- *Pair(long, way), Similarity: 0.996485*
- *Pair(way, long), Similarity: 0.996485*
- *Pair(book, read), Similarity: 0.9962874*
- *Pair(read, book), Similarity: 0.9962874*
- *Pair(dog, cat), Similarity: 0.9962772*
- *Pair(cat, dog), Similarity: 0.9962772*
- *Pair(another, today), Similarity: 0.99608654*
- *Pair(today, another), Similarity: 0.99608654*
- *Pair(two, home), Similarity: 0.99598694*
- *Pair(home, two), Similarity: 0.99598694*
- *Pair(no, shoe), Similarity: 0.99592084*
- *Pair(shoe, no), Similarity: 0.99592084*
- *Pair(be, gox), Similarity: 0.9954331*

Problem H [2 points]: What patterns do you notice across the resulting pairs of words?

Solution H: *Most words that are frequently used near each other ("green eggs and ham") have a high similarity score. Additionally, due to the symmetry in the generation of x,y pairs, the two orderings of a pair of words are similar to each other in similarity score.*