Machine Learning & Data Mining — Caltech CS/CNS/EE 155
Set 5 — February 2024

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:

Given the singular decomposition $X = U\Sigma V^T$, where X is an $N \times N$ matrix:

- *U is an $N \times N$ orthogonal matrix whose columns are the left singular vectors of X.*
- *Σ is an $N \times N$ orthogonal matrix with non-negative real numbers on the diagonal, which are the singular values of X.*
- *V is an $N \times N$ orthogonal matrix whose columns are the right singular vectors of X.*

$$\begin{aligned} XX^T &= U\Sigma V^T(U\Sigma V^T)^T \\ &= U\Sigma V^T V\Sigma U^T \\ &= U\Sigma^2 U^T && \textit{(Since } V^TV = I\textit{)} \end{aligned}$$

Multiplying XX^T by U gives $XX^TU = U\Sigma^2$, which shows that the columns of U are scaled by the square of the singular values of X, which are the eigenvalues of XX^T. The relationship between the singular values of X and the eigenvalues of X^TX is that the non-zero eigenvalues of X^TX (and XX^T) are the squares of the singular values of X. Thus, for SVD, each column of U is a principal component of X, and the singular values of X are the positive square roots of the eigenvalues of X^TX.

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, the eigenvalues represent the variance captured by each principal component. Since variance is a measure of the spread of data points around the mean and cannot be negative, the eigenvalues, which quantify this spread, must also be non-negative. Mathematically, the eigenvalues of X^TX are non-negative because we have shown in question 1 that they are the squares of the singular values which are always positive.

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 Tr$(AB) = \sum_{i=1}^{N}(AB)_{ii}$. Can you find a way to expand $(AB)_{ii}$ in terms of another sum?

Solution C:

To show that the trace is invariant under cyclic permutations, we start with two matrices and then generalize to any number of square matrices. For two matrices A and B, the trace of their product is given by:

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

where $(AB)_{ii}$ represents the i-th diagonal element of the matrix product AB. Now, expanding $(AB)_{ii}$ we have:

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

Hence, the trace can be written as:

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

Due to the commutative property of scalar addition, we can write:

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

This shows that the trace is invariant under the exchange of A and B.

To generalize for a product of any number of square matrices, we use basic induction. Assume the property holds for the product of k matrices. Now consider the product of $k+1$ matrices:

$$Tr(A_1A_2\ldots A_kA_{k+1}) = Tr((A_1A_2\ldots A_k)A_{k+1})$$

By the inductive hypothesis, we can cyclically permute the first k matrices without changing the trace:

$$Tr((A_1A_2\ldots A_k)A_{k+1}) = Tr(A_{k+1}(A_1A_2\ldots A_k)) = Tr(A_{k+1}A_1A_2\ldots A_k)$$

Therefore, the trace is invariant under cyclic permutations for any number of square matrices.

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 the full $N \times N$ matrix X, we require N^2 values. In the case of a truncated SVD, we store only the k largest singular values and the corresponding columns of U and V. For the k largest singular values in Σ_k, we store k values since Σ_k is a diagonal matrix. For the corresponding columns of U and V, which are $N \times k$ matrices, we store Nk values for U_k and Nk values for V_k. Therefore, the total number of values stored for the truncated SVD is $k + 2Nk = k(1 + 2N)$.

Truncated SVD is more efficient than storing the whole matrix when $k(1 + 2N) < N^2$, which simplifies to $k < \frac{N^2}{1+2N}$. We do not need to store every entry of the diagonal matrix Σ; we only store the non-zero singular values, which are k in number for the truncated SVD.

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 $(\Sigma)_{ii}$, $i \in \{1, \dots, K\}$, 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 $D > N$ and that X has a rank N. We also know that Σ has non-zero values on entries $(\Sigma)_{ii}, i \in \{1, ...N\}$. If we let Σ' be the $N \times N$ matrix with the first N rows of Σ and U' be the $D \times N$ matrix with the first columns of U, we can see that Σ' is Σ with the rows of zeros after the last diagonal value is removed. Therefore, we can see that $U\Sigma = U'\Sigma'$ because the rows lost from Σ multiplied by the columns lost from U become 0.

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:

Since U' is not square, $U'U'^T \neq U'^T U' \neq I$. This is because U' has dimensions of $D \times N$. As a result, $U'U'^T$ has dimensions $D \times D$ and $U'^T U'$ has dimensions $N \times N$.

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:

Since the columns of U' are still orthonormal, we know that they are linearly independent. Thus, when they are multiplied by the corresponding row in U'^T, it will result in 1's along the diagonal and 0's everywhere else. Thus, the resulting matrix is an $N \times N$ identity matrix. It is not true that $U'U'^T = I_{D \times D}$ because this would only hold if U' was orthogonal.

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: *We are given that Σ is invertible, X is a matrix of size $D \times N$ where $D > N$ with "thin" SVD $X = U\Sigma V^T$, and that X has rank N. To show that $X^+ = V\Sigma^+ U^T$ is equivalent to $V\Sigma^{-1}U^T$, we need to show that $\Sigma^+ = \Sigma^{-1}$. From the lecture, we know that Σ^+ is the Σ diagonal matrix with every positive element being an inverse of itself. If Σ is a diagonal matrix with non-zero entries, then its pseudoinverse, Σ^+, is simply a diagonal matrix with the reciprocals of Σ's entries (i.e., Σ^{-1}). Thus, $\Sigma^+ = \Sigma^{-1}$ and the pseudoinverse of X can be written as $V\Sigma^{-1}U^T$.*

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

Solution I:

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

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 condition number of a matrix A, $\kappa(A)$, is given by the ratio of the maximum singular value of A to the minimum singular value, $\kappa(A) = \frac{\sigma_{max}(A)}{\sigma_{min}(A)}$.

For the matrix Σ, which is diagonal and contains the singular values of X, the condition number is simply the ratio of the largest to the smallest singular value of X.

For the matrix X^TX, the singular values are the squares of the singular values of X. Therefore, the condition number of X^TX is the square of the condition number of X. This is because if σ_i are the singular values of X, then the singular values of X^TX are σ_i^2, and thus $\kappa(X^TX) = \frac{\sigma_{max}^2}{\sigma_{min}^2} = \left(\frac{\sigma_{max}}{\sigma_{min}}\right)^2 = \kappa(X)^2$.

Since squaring the condition number greatly increases it if it is larger than 1, the expression $X^+ = (X^TX)^{-1}X^T$ will be more prone to numerical errors, especially when $\kappa(X)$ is large, due to the squaring of the condition number in the computation of $(X^TX)^{-1}$.

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:

$$\begin{aligned} u_i &= u_i - \eta\partial_{u_i} \\ v_j &= v_j - \eta\partial_{v_j} \end{aligned}$$

where η is the learning rate.

Solution A:

To find the gradient of $J(U,V)$ with respect to u_i, we take the derivative of the regularized squared error function:

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

For v_j:

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

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:

For the optimal u_i given fixed V, we set the derivative of J with respect to u_i to zero:

$$\begin{aligned}
\frac{\partial J}{\partial u_i} &= \lambda u_i - \sum_j (y_{ij} - u_i^T v_j) v_j = 0 \\
\lambda u_i - \sum_j (v_j)(y_{ij}) + \sum_j (v_j)(u_i^T v_j) &= 0 \\
\lambda u_i - \sum_j (v_j)(y_{ij}) + (u_i) \sum_j (v_j)(v_j^T) &= 0 \\
(u_i)(\lambda I + \sum_j (v_j)(v_j^T)) &= \sum_j (v_j)(y_{ij}) \\
u_i &= \frac{\sum_j (v_j)(y_{ij})}{\lambda I + \sum_j (v_j)(v_j^T)}
\end{aligned}$$

By similar procedure,

$$v_i = \frac{\sum_i (u_i)(y_{ij})}{\lambda I + \sum_i (u_i)(u_i^T)}$$

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:

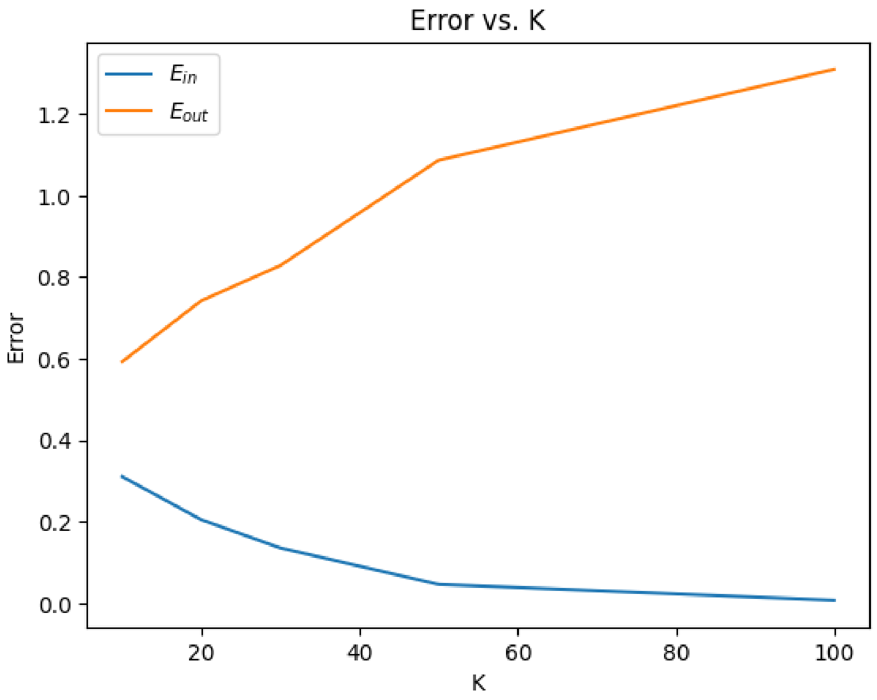
- Keep track of the loss reduction on the training set from epoch to epoch, and stop when the relative loss reduction compared to the first epoch is less than $\epsilon = 0.0001$. That is, if $\Delta_{0,1}$ denotes the loss reduction from the initial model to end of the first epoch, and $\Delta_{i,i-1}$ is defined analogously, then stop after epoch t if $\Delta_{t-1,t}/\Delta_{0,1} \leq \epsilon$.

Solution C:

mantripragada_ishaan_set5_prob2.ipynb

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?

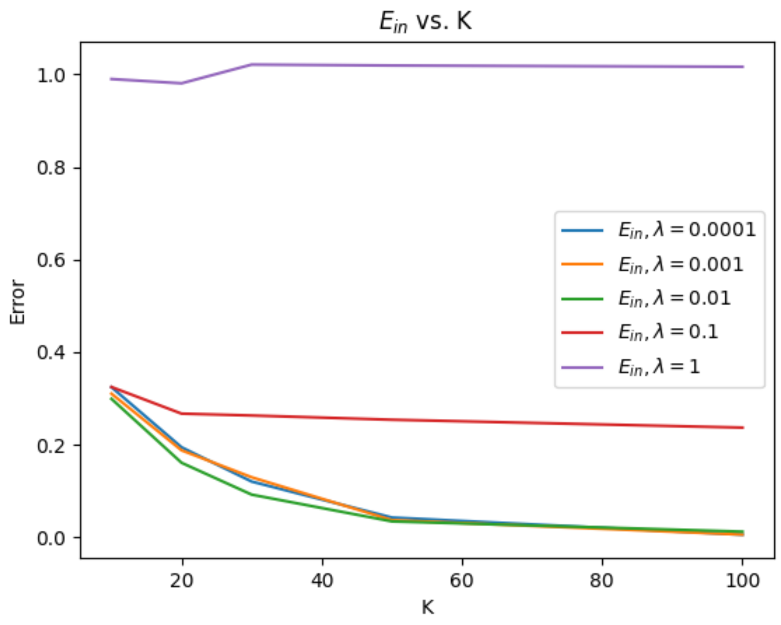
Solution D:

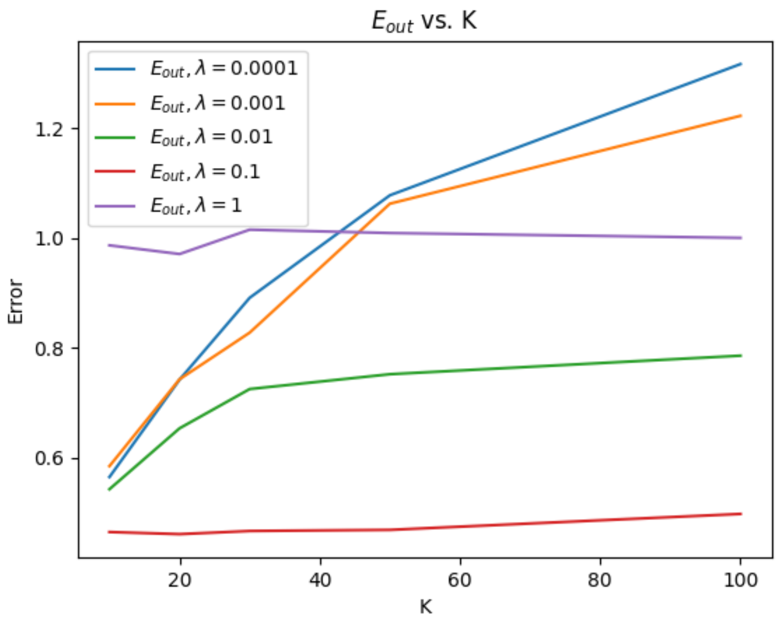


The plot shows that as k increases, E_{in} decreases while E_{out} increases. A possible explanation for this is that there might be overfitting due to the large number of latent factors.

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:





For E_{in} vs k, we see that as k increases, the training error decreases. On the other hand, for E_{out} vs k, we see that as k increases, the testing error increases. Again, this makes sense because the large number of latent factors likely causes overfitting. Also, for both graphs, when $\lambda = 1$, the error roughly remains the same around an error value of 1. This makes sense because the model does not learn in this case. In general, we can see that when we increase the regularization by too much, it results in overfitting.

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 $O(WD)$ because as we iterate over the number of words W, we compute dot products with D dimensions each.*

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.

Table 1: Words and frequencies for Problem B

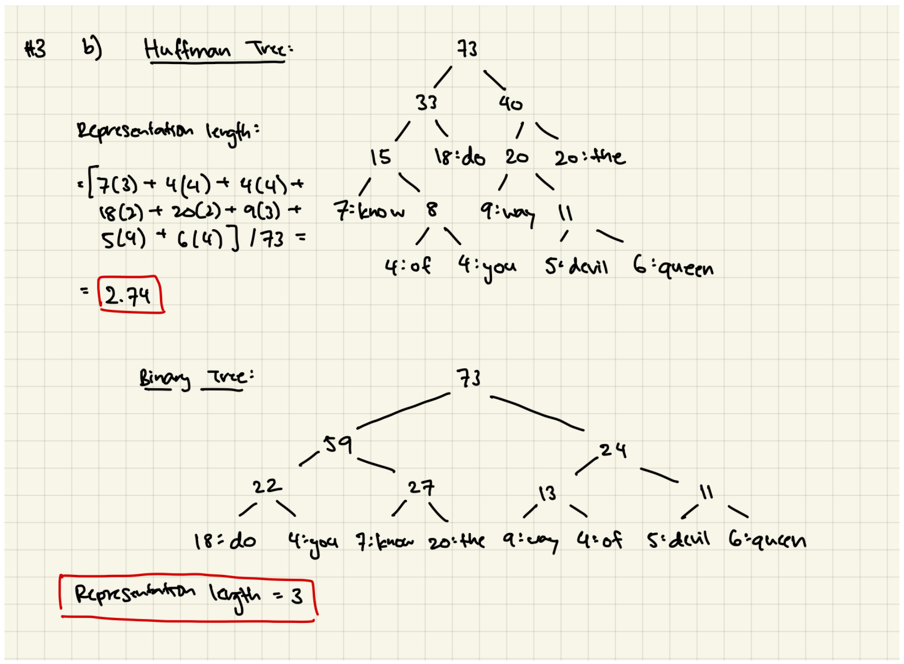
Word	Occurrences
do	18
you	4
know	7
the	20
way	9
of	4
devil	5
queen	6

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:



#3 b) Huffman Tree:

Representation length:

$= [7(3) + 4(4) + 4(4) + 18(2) + 20(2) + 9(3) + 5(4) + 6(4)] / 73 =$

$= \boxed{2.74}$

Binary Tree:

$\boxed{\text{Representation length} = 3}$

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 D increases, I expect the value of the training objective to also increase since the training accuracy is improved. However, a large number of dimensions could easily cause the model to overfit.*

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 $s = 2$:

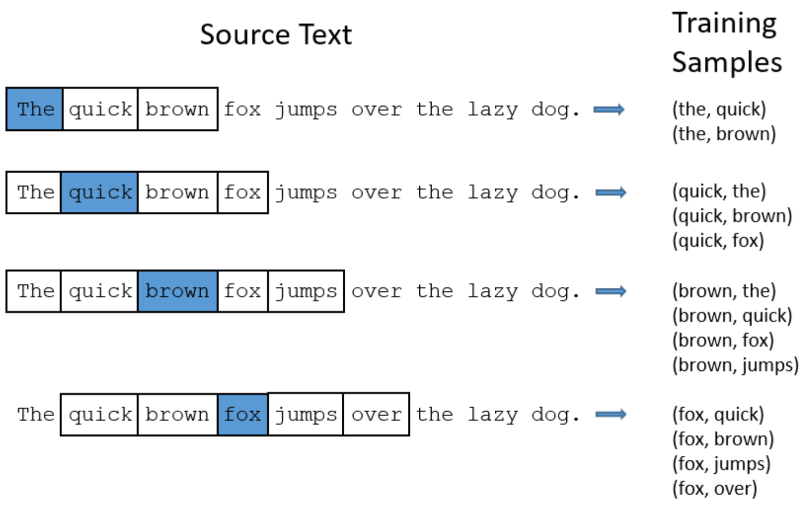


Figure 1: Generating Word2Vec Training Points

(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: *mantripragada_ishaan_set5_prob3.ipynb*

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 dimensions of the weight matrix of the hidden layer are* 308×10. *This is because there are 308 unique words and 10 hidden units.*

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

Solution F: *The dimensions of the weight matrix of the output layer is* 10×308. *This is because we still have 10 hidden unites and 308 words unique to output.*

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

Solution G:

```
Pair(when, girls)      Similarity: 0.9780765
Pair(girls, when)      Similarity: 0.9780765
Pair(hello, hold)      Similarity: 0.97372353
Pair(hold, hello)      Similarity: 0.97372353
Pair(six, hook)        Similarity: 0.97061664
Pair(hook, six)        Similarity: 0.97061664
Pair(be, gox)          Similarity: 0.96206355
Pair(gox, be)          Similarity: 0.96206355
Pair(can, down)        Similarity: 0.9586113
Pair(down, can)        Similarity: 0.9586113
Pair(thin, four)       Similarity: 0.9574201
Pair(four, thin)       Similarity: 0.9574201
Pair(sheep, be)        Similarity: 0.95560324
Pair(bet, hair)        Similarity: 0.9554841
Pair(hair, bet)        Similarity: 0.9554841
Pair(black, just)      Similarity: 0.9546966
Pair(just, black)      Similarity: 0.9546966
Pair(fly, be)          Similarity: 0.95065325
Pair(fast, six)        Similarity: 0.95047075
Pair(eleven, be)       Similarity: 0.9499002
Pair(us, may)          Similarity: 0.9466558
Pair(may, us)          Similarity: 0.9466558
Pair(that, sheep)      Similarity: 0.94457984
Pair(jump, quiet)      Similarity: 0.9434635
Pair(quiet, jump)      Similarity: 0.9434635
Pair(story, should)    Similarity: 0.9390091
Pair(should, story)    Similarity: 0.9390091
Pair(no, ink)          Similarity: 0.938219
Pair(ink, no)          Similarity: 0.938219
Pair(now, down)        Similarity: 0.9378871
```

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

Solution H: *We can see that each pair shows up twice because comparing the first word to the second word is the same as comparing the second word to the first word.*