Melba Nuzen **Problem Set 3** CS 155 February 22, 2020

1 SVD and PCA

Problem A:
The singular values of X are square roots of the eigenvalues of XX^T.

Problem B:
The eigenvalues are non-negative because looking at some eigenvector v, we calculate the length to be some non-negative number, meaning the eigenvalue must also be non-negative.

Problem C:
We can prove that trace is invariant under cyclic permutations for $\text{Tr}(AB)$ with the following proof:

$$\begin{aligned}
tr(AB) &= \sum_{i=1}^{n}(AB)_{ii} && (1)\\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} A_{ij}B_{ji} && (2)\\
&= \sum_{j=1}^{m}\sum_{i=1}^{n} B_{ji}A_{ij} && (3)\\
&= \sum_{j=1}^{m}(BA)_{jj} && (4)\\
&= tr(BA) && (5)
\end{aligned}$$

More generally, we can see that cyclic invariance holds regardless of the matrices' dimensions because the trace of a product in a certain order is just the sum of all the products in corresponding entries, thus making the order of the products irrelevant to the actual trace value.

Problem D:

Problem E:
Since only the first N number of rows in Σ contains non-zero entries and since the last $D-N$ rows are all known to be zero, when multiplying together U and Σ we know that the values multiplied by 0 end up as 0, and therefore can be omitted. So the last $D-N$ columns of U can be omitted, and U only needs the first N columns. Thus, U must have a shape of D by N and Σ should have a shape of N by N.

Problem F:
We want to show that since U' is not square, it cannot be orthogonal. We will show this

knowing that our matrix A is orthogonal if $AA^T = A^TA = I$. Comparing the shapes of $U'U'^T$ and U'^TU' gives us shapes D by D and N by N respectively. Since these shapes are not equal, the matrices cannot be equal, and thus, by the known information above, U' can't be orthogonal, as desired.

Problem G:
We want to show that $U'^TU' = I_{N\times N}$. We note that U is orthogonal (stated), which means the dot product of its columns and rows must be 1 since $UU^T = U^TU = I$ and $\Sigma_{i=1}^N u_i^2 = 1$ for all the u rows and columns in U. U'^TU' must also be equivalent to the first N columns of U (taken the dot product against itself) because U' has the same first N columns of U. U'^TU' must equal $I_{N\times N}$ because when U' is transposed, its rows become the columns of U and, by the rules of matrix multiplication, each value becomes equivalent to 1.

We see that $U'U'^T$ is not equivalent to $I_{D\times D}$ because there are D rows of N columns and so U' can't be independent and thus not orthonormal. Additionally, the dot product taken against itself isn't guaranteed to be 1 since we have a different dimension for rows.

Problem H:
If we have some SVD $X = U\Sigma V^T$, then we know some solution X^+ must be a pseudoinverse if $X^+X = XX^+ = I$, which holds true. We also know that $X^+ = (X^TX)^{-1}X^T = V\Sigma^+U^T$, which is also equal to $V\Sigma^{-1}U^T$, which we will prove in the following problem. So then if Σ is invertible, then $X^+ = V\Sigma^{-1}U^T$.

Problem I:
We want to show that $X^{+\prime} = (X^TX)^{-1}X^T$ is equivalent to $V\Sigma^{-1}U^T$, which we can do in the following way:

$$\begin{aligned}
X^{+\prime} &= (X^TX)^{-1}X^T && (6)\\
&= (V\Sigma U^TU\Sigma V^T)^{-1}V\Sigma U^T && (7)\\
&= (V\Sigma^2V^T)^{-1}V\Sigma U^T && (8)\\
&= (V^T)^{-1}\Sigma^{-2}V^{-1}V\Sigma U^T && (9)\\
&= V\Sigma^{-1}U^T && (10)
\end{aligned}$$

2 Matrix Factorization

Problem A:

$$\partial_{u_i} = \lambda u_i - \sum_j v_j(y_{ij} - u_i^T v_j) \tag{11}$$

$$\partial_{v_j} = \lambda v_j - \sum_i u_i(y_{ij} - u_i^T v_j) \tag{12}$$

Problem B: We want to find the expression for the u_i that minimizes the above regularized square error, and for the v_j that minimizes its given U. To do that, we will use the expressions we derived from the previous questions, ∂_{u_i} and ∂_{v_j} by setting them each to zero, and solving for u_i and v_j, respectively. This gives us the following expressions:

$$0 = \partial_{u_i} = \lambda u_i - \sum_j v_j(y_{ij} - u_i^T v_j) \tag{13}$$

$$0 = \lambda u_i - \sum_j v_j(y_{ij} - u_i^T v_j) \tag{14}$$

$$0 = \lambda u_i - \sum_j v_j(y_{ij} - u_i^T v_j) \tag{15}$$

Then, we can multiply the summation out and begin solving for u_i, knowing that $v_j = u_i$:

$$0 = \lambda u_i - \sum_j v_j y_{ij} - \sum_j u_i^T v_j v_j \tag{16}$$

$$0 = \lambda u_i - \sum_j v_j y_{ij} - \sum_j v_j^T v_j u_i \tag{17}$$

$$\sum_j v_j y_{ij} = \lambda u_i - (\sum_j v_j^T v_j) u_i \tag{18}$$

$$\sum_j v_j y_{ij} = u_i(\lambda I - \sum_j v_j^T v_j) \tag{19}$$

$$u_i = \frac{\sum_j v_j y_{ij}}{\lambda I - \sum_j v_j^T v_j} \tag{20}$$

We do the same for ∂_{v_j}:

$$0 = \partial_{v_j} = \lambda v_j - \sum_i u_i(y_{ij} - u_i^T v_j) \tag{21}$$

$$0 = \lambda v_j - \sum_i u_i(y_{ij} - u_i^T v_j) \tag{22}$$

$$0 = \lambda v_j - \sum_i u_i y_{ij} - \sum_i u_i^T v_j u_i \tag{23}$$

$$0 = \lambda v_j - \sum_i u_i y_{ij} - \sum_i u_i^T v_j v_j \tag{24}$$

$$\sum_i u_i y_{ij} = \lambda v_j - \sum_i (u_i^T v_j) v_j \tag{25}$$

$$\sum_i u_i y_{ij} = v_j(\lambda I - \sum_i u_i^T u_i) \tag{26}$$

$$v_j = \frac{\sum_i u_i y_{ij}}{\lambda I - \sum_i u_i^T u_i} \tag{27}$$

Problem C:
See attached code.

Problem D:
As the value of k increases we see that E_{in} goes down and E_{out} goes up. This makes sense: k represents the number of latent factors, meaning smaller k's represent smaller dimensions for our two matrices U and V and smaller representations of our original input data. Smaller k's represent more reduction in our original input space (as opposed to larger k's, which have matrices of similar size to our original spaces). So these smaller k's correlate with less overfitting since there is less data to work with in general, corresponding with the lower level of test error that we see. Increasing the value of k, however, increases that test error as we increase likelihood of overfitting; so E_{in} decreases because of generalization.

Problem E:
We see similar patterns from Problem D: larger k values corresponding with lower E_{in} and higher E_{out}. When we look at the varying values of λ, we see smaller λ's correlate with previous results. Conversely, λ's that are larger have higher E_{in} as well as higher E_{out}, which we could possibly attribute to underfitting.

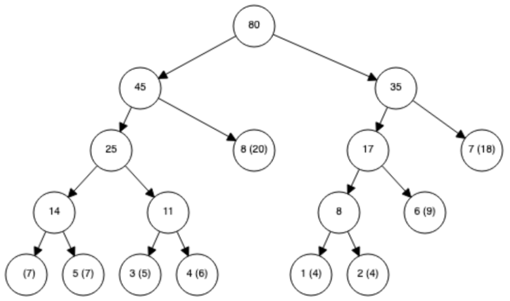
3 Word2Vec Principles

Problem A:
Calculating ∇ *log* $p(w_O|w_I)$ would scale linearly with the number of words, W, because we see that computing the numerator's derivative would take constant time, and increasing W would increase the number of terms in summation in the denominator. Increasing W increases the number of terms needed to differentiate, so the computation of ∇ *log* $p(w_O|w_I)$ would have a run-time of $O(W)$.

Problem B:
To make it easier to analyze occurences, we will name the words 1 through 8 in order from least frequent to most frequent, so:

1. you: 4
2. of: 4
3. devil: 5
4. queen: 6
5. know: 7
6. way: 9
7. do: 18
8. the: 20

We can generate a Huffman tree that resembles the following:



And then determine the Huffman code based on left being 0 and right being 1. The words, their codes, and their code lengths are as follows:

1. you: 0010—4

2. of: 0011—4
3. devil: 1010—4
4. queen: 1011—4
5. know: 000—3
6. way: 100—3
7. do: 01—2
8. the: 11—2

Now we determine expected representation length by taking the sum of the frequencies times code length all over the sum of the actual frequencies. That gives us a value of about 2.739.

We do the same for a binary tree and get the following words and their code lengths:

1. you—2
2. of—1
3. devil—2
4. queen—0
5. know—2
6. way—1
7. do—2
8. the—3

And find expected representation length the same way as before. This gives us a value of about 1.931, which is shorter as expected, since the Huffman tree inserted nodes that weren't originally in the dataset.

Problem C:
Increasing D increases the complexity needed to compute the training objective and thus increases runtime. Having a larger D could make it very expensive to calculate corresponding computations and increases D could also increases probability of overfitting to the training set, which is why we might not want to use a very large D.

Problem D:
See attached code.

Problem E:
Hidden layer dimensions: 308 by 10.

Problem F:
Output layer dimensions: 10 by 308.

Problem G:

```
Pair(drink, thing), Similarity: 0.9914622
Pair(thing, drink), Similarity: 0.9914622
Pair(shoe, foot), Similarity: 0.984953
Pair(foot, shoe), Similarity: 0.984953
Pair(green, eggs), Similarity: 0.9821921
Pair(eggs, green), Similarity: 0.9821921
Pair(bump, jump), Similarity: 0.97820556
Pair(jump, bump), Similarity: 0.97820556
Pair(off, shoe), Similarity: 0.977806
Pair(ham, eggs), Similarity: 0.97737646
Pair(fly, kite), Similarity: 0.9735595
Pair(kite, fly), Similarity: 0.9735595
Pair(did, ever), Similarity: 0.97241956
Pair(ever, did), Similarity: 0.97241956
Pair(likes, drink), Similarity: 0.972407
Pair(eight, nine), Similarity: 0.972128
Pair(nine, eight), Similarity: 0.972128
Pair(pink, drink), Similarity: 0.96961546
Pair(zeep, today), Similarity: 0.96937186
Pair(today, zeep), Similarity: 0.96937186
Pair(finger, top), Similarity: 0.9672727
Pair(top, finger), Similarity: 0.9672727
Pair(sad, glad), Similarity: 0.96681505
Pair(glad, sad), Similarity: 0.96681505
Pair(their, heads), Similarity: 0.9643549
Pair(heads, their), Similarity: 0.9643549
Pair(wink, likes), Similarity: 0.9632417
Pair(gone, today), Similarity: 0.962011
Pair(brush, comb), Similarity: 0.96060026
```

Problem H:
Many words seen together can be flipped, i.e. (a,b) is repeated and (b,a) is repeated, which makes sense since if one word is seen with another, then they are often seen together. Additionally, there are words together that rhyme, which makes sense since Dr Seuss is known for rhyming in many of his works.

Problem 2

Authors: Fabian Boemer, Sid Murching, Suraj Nair, Alex Cui

In [85]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
```

2C:

Fill in these functions to train your SVD

In [149]:
```
def grad_U(Ui, Yij, Vj, reg, eta):
    """
    Takes as input Ui (the ith row of U), a training point Yij, the column
    vector Vj (jth column of V^T), reg (the regularization parameter lambda),
    and eta (the learning rate).

    Returns the gradient of the regularized loss function with
    respect to Ui multiplied by eta.
    """
    t1 = reg * Ui

    Vj = np.squeeze(np.asarray(Vj))
    Ui = np.squeeze(np.asarray(Ui))

    UT = np.dot(Vj,Ui)
    t2 = Vj * (Yij - UT)
    return eta*(t1 - t2)

def grad_V(Ui, Yij, Vj, reg, eta):
    """
    Takes as input the column vector Vj (jth column of V^T), a training point Yij,
    Ui (the ith row of U), reg (the regularization parameter lambda),
    and eta (the learning rate).

    Returns the gradient of the regularized loss function with
    respect to Vj multiplied by eta.
    """
    t1 = reg * Vj

    Vj = np.squeeze(np.asarray(Vj))
    Ui = np.squeeze(np.asarray(Ui))

    UT = np.dot(Vj,Ui)
    t2 = Ui * (Yij - UT)
    return eta*(t1 - t2)

def get_err(U, V, Y, reg=0.0):
    """
    Takes as input a matrix Y of triples (i, j, Y_ij) where i is the index of a us
er,
    j is the index of a movie, and Y_ij is user i's rating of movie j and
    user/movie matrices U and V.

    Returns the mean regularized squared-error of predictions made by
    estimating Y_{ij} as the dot product of the ith row of U and the jth column of
V^T.
    """
    totErr = 0
    U_transpose = np.matrix(np.transpose(U))
    rest = np.asarray(U_transpose * np.matrix(V))
    #assert(rest.shape == Y.shape)

    for row in range(len(Y)):
        for col in range(len(Y[0])):
            if Y[row][col] > 0.1:
                totErr += pow(Y[row][col] - rest[row][col],2)
    return sqrt(totErr)

def train_model(M, N, K, eta, reg, Y, eps=0.0001, max_epochs=300):
    """
    Given a training data matrix Y containing rows (i, j, Y_ij)
    where Y_ij is user i's rating on movie j, learns an
    M x K matrix U and N x K matrix V such that rating Y_ij is approximated
    by (UV^T)_ij.
```

2D:

Run the cell below to get your graphs

In [ ]:
```
Y_train = np.loadtxt('./data/train.txt').astype(int)
Y_test = np.loadtxt('./data/test.txt').astype(int)

M = max(max(Y_train[:,0]), max(Y_test[:,0])).astype(int) # users
N = max(max(Y_train[:,1]), max(Y_test[:,1])).astype(int) # movies
print("Factorizing with ", M, " users, ", N, " movies.")
Ks = [10,20,30,50,100]

reg = 0.0
eta = 0.03 # learning rate
E_in = []
E_out = []

# Use to compute Ein and Eout
for K in Ks:
    U,V, err = train_model(M, N, K, eta, reg, Y_train)
    E_in.append(err)
    E_out.append(get_err(U, V, Y_test))

plt.plot(Ks, E_in, label='$E_{in}$')
plt.plot(Ks, E_out, label='$E_{out}$')
plt.title('Error vs. K')
plt.xlabel('K')
plt.ylabel('Error')
plt.legend()
plt.savefig('2d.png')
```

2E:

Run the cell below to get your graphs. This might take a long time to run, but it should take less than 2 hours. I would encourage you to validate your 2C is correct.

In [ ]:

```
Y_train = np.loadtxt('./data/train.txt').astype(int)
Y_test = np.loadtxt('./data/test.txt').astype(int)

M = max(max(Y_train[:,0]), max(Y_test[:,0])).astype(int) # users
N = max(max(Y_train[:,1]), max(Y_test[:,1])).astype(int) # movies
Ks = [10,20,30,50,100]

regs = [10**-4, 10**-3, 10**-2, 10**-1, 1]
eta = 0.03 # learning rate
E_ins = []
E_outs = []

# Use to compute Ein and Eout
for reg in regs:
    E_ins_for_lambda = []
    E_outs_for_lambda = []

    for k in Ks:
        print("Training model with M = %s, N = %s, k = %s, eta = %s, reg = %s"%(M,
N, k, eta, reg))
        U,V, e_in = train_model(M, N, k, eta, reg, Y_train)
        E_ins_for_lambda.append(e_in)
        eout = get_err(U, V, Y_test)
        E_outs_for_lambda.append(eout)

    E_ins.append(E_ins_for_lambda)
    E_outs.append(E_outs_for_lambda)


# Plot values of E_in across k for each value of lambda
for i in range(len(regs)):
    plt.plot(Ks, E_ins[i], label='$E_{in}, \lambda=$'+str(regs[i]))
plt.title('$E_{in}$ vs. K')
plt.xlabel('K')
plt.ylabel('Error')
plt.legend()
plt.savefig('2e_ein.png')
plt.clf()

# Plot values of E_out across k for each value of lambda
for i in range(len(regs)):
    plt.plot(Ks, E_outs[i], label='$E_{out}, \lambda=$'+str(regs[i]))
plt.title('$E_{out}$ vs. K')
plt.xlabel('K')
plt.ylabel('Error')
plt.legend()
plt.savefig('2e_eout.png')
```

In [ ]:

In [ ]:

Problem 3

Authors: Sid Murching, Suraj Nair, Alex Cui

In [9]:
```
import numpy as np
from P3CHelpers import *
from keras.models import Sequential
import sys
```

3D:

Fill in the generate_traindata and find_most_similar_pairs functions

In [10]:
```
def get_word_repr(word_to_index, word):
    """
    Returns one-hot-encoded feature representation of the specified word given
    a dictionary mapping words to their one-hot-encoded index.

    Arguments:
        word_to_index: Dictionary mapping words to their corresponding index
                       in a one-hot-encoded representation of our corpus.

        word:          String containing word whose feature representation we wish t

    Returns:
        feature_representation:     Feature representation of the passed-in word.
    """
    unique_words = word_to_index.keys()
    # Return a vector that's zero everywhere besides the index corresponding to <wor
    feat_rep = np.zeros(len(unique_words))
    feat_rep[word_to_index[word]] = 1
    return feat_rep

def generate_traindata(word_list, word_to_index, window_size=4):
    """
    Generates training data for Skipgram model.

    Arguments:
        word_list:     Sequential list of words (strings).
        word_to_index: Dictionary mapping words to their corresponding index
                       in a one-hot-encoded representation of our corpus.

        window_size:   Size of Skipgram window.
                       (use the default value when running your code).

    Returns:
        (trainX, trainY):     A pair of matrices (trainX, trainY) containing trainin
                              points (one-hot-encoded vectors representing individua
                              their corresponding labels (also one-hot-encoded vecto

                              For each index i, trainX[i] should correspond to a wor
                              <word_list>, and trainY[i] should correspond to one of
                              a window of size <window_size> of trainX[i].
    """
    trainX = []
    trainY = []

    numWords = len(word_to_index)
    allZeroes = [0 for i in range(numWords)]

    for i in range(len(word_list)):
        for j in range(-window_size, window_size + 1):
            if i + j >= 0 and i + j < len(word_list) and j != 0:
                trainXV = get_word_repr(word_to_index, word_list[i])
                trainX.append(trainXV)
                trainYV = get_word_repr(word_to_index, word_list[i+j])
                trainY.append(trainYV)

    return (np.array(trainX), np.array(trainY))
```

In [12]:
```python
def find_most_similar_pairs(filename, num_latent_factors):
    """
    Find the most similar pairs from the word embeddings computed from
    a body of text

    Arguments:
        filename:           Text file to read and train embeddings from
        num_latent_factors: The number of latent factors / the size of the embedding
    """
    # Load in a list of words from the specified file; remove non-alphanumeric chara
    # and make all chars lowercase.
    sample_text = load_word_list(filename)

    # Create dictionary mapping unique words to their one-hot-encoded index
    word_to_index = generate_onehot_dict(sample_text)
    # Create training data using default window size
    trainX, trainY = generate_traindata(sample_text, word_to_index)

    # TODO: 1) Create and train model in Keras.

    # vocab_size = number of unique words in our text file. Will be useful when addi
    # to your neural network
    vocab_size = len(word_to_index)
    model = Sequential()
    model.add(Dense(num_latent_factors, input_dim=(vocab_size)))
    model.add(Dense(vocab_size))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
                  metrics=['accuracy'])
    fit = model.fit(trainX, trainY)

    print("Hidden layer shape" + str(model.layers[0].get_weights()[0].shape))
    print("Output layer shape" + str(model.layers[1].get_weights()[0].shape))
    # TODO: 2) Extract weights for hidden layer, set <weights> variable below

    weights = model.layers[0].get_weights()[0]

    # Find and print most similar pairs
    similar_pairs = most_similar_pairs(weights, word_to_index)
    for pair in similar_pairs[:30]:
        print(pair)
```

3G:

Run the function below and report your results for dr_seuss.txt.

In [ ]:
```python
find_most_similar_pairs('data/dr_seuss.txt', 10)
```