Machine Learning & Data Mining
Caltech CS/CNS/EE 155
Homework 5

Changhao Xu
UID: 2103530
February 17th, 2020

---

1 SVD and PCA

Question A:

$$XX^T = U\Sigma V^T(U\Sigma V^T)^T = U\Sigma V^T V\Sigma^T U^T = U\Sigma^2 U^T$$

Since $XX^T = U\Lambda U^T$, let $\Lambda = \Sigma^2$, then columns of U are the principal components of X. Eigenvalues of XX^T are squared singular values of X.

Question B: Intuitive explanation: eigenvalues of the PCA of X are the variance of X along the direction of eigenvector, and the variance are always non-negative.
Mathematical justification: from Question A, eigenvalues of the PCA of X have the property of $\Lambda = \Sigma^2$, so they must be non-negative.

Question C:

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

Let B = BC, then Tr(ABC) = Tr(BCA). Let A = CA, then Tr(CAB) = Tr(BCA).
In general, for any number of square matrices $A_1 \cdots A_N$, we have

$$Tr(A_1 \cdots A_N) = Tr(A_2 \cdots A_N A_1) = \cdots = Tr(A_N A_1 \cdots A_{N-1}).$$

Question D: To store a truncated SVD with $U\Sigma V^T$, for U we need N*k values, for Σ we need k values since it's a diagonal matrix and all other coefficients are 0, for V we need N*k values. Therefore, in total we need (2N+1)k values. When $(2N+1)k < N^2$, that is $k < \frac{N^2}{2N+1}$, storing the truncated SVD is more efficient than storing the whole matrix.

Question E: Since Σ only has non-zero values on entries Σ_{ii}, where $i \in \{1, \cdots N\}$, when multiply

$$(U\Sigma)_{ij} = \sum_{k=1}^{D} U_{ik}\Sigma_{kj} = \sum_{k=1}^{N} U_{ik}\Sigma_{kj} + \sum_{k=N+1}^{D} U_{ik}\Sigma_{kj} = \sum_{k=1}^{N} U_{ik}\Sigma_{kj} + \sum_{k=N+1}^{D} U_{ik}0 = \sum_{k=1}^{N} U_{ik}\Sigma_{kj}$$

where $i \in \{1, \cdots D\}, j \in \{1, \cdots N\}$. Therefore, $U\Sigma = U'\Sigma'$, where U' is the D $\times$ N matrix consisting of the first N columns of U, and Σ' is the N $\times$ N matrix consisting of the first N rows of Σ.

Question F: U' is a D $\times$ N matrix, and U'^T is a N $\times$ D matrix. Therefore, $U'U'^T$ is a D $\times$ D matrix and $U'^T U'$ is a N $\times$ N matrix. Since they are not equal, U' is not orthogonal.

Question G: Since columns of U' are still orthonormal,

$$(U'^T U')_{ij} = \sum_{k=1}^{D} U'^T_{ik} U'_{kj} = \sum_{k=1}^{D} U'_{ki} U'_{kj} = 1$$

if and only if i=j, where $i \in \{1, \cdots N\}, j \in \{1, \cdots N\}$. So $(U'^T U') = I_{N \times N}$.
On the other hand, $(U' U'^T) \neq I_{D \times D}$. This is because rows of U′ cannot be orthonormal since rank(U′) ≤ N while D >N.

Question H: On lecture 10 slide 53, $X^+ = V \Sigma^+ U^T$, where Σ^+ is a diagonal matrix.

$$\Sigma^+ = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \sigma_D \end{bmatrix} \qquad \sigma^+ = \begin{cases} 1/\sigma & \text{if } \sigma > 0 \\ 0 & \text{otherwise} \end{cases}$$

When Σ^+ is also invertible, that is $\sigma_i \neq 0$, $\Sigma^{-1} = \Sigma^+$ where all diagonal units are $\frac{1}{\sigma_i}$. So $X^+ = V \Sigma^{-1} U^T$.

Question I: $X^{+\prime} = (X^T X)^{-1} X^T \Leftrightarrow X^T X X^{+\prime} = X^T$.
On the other hand, since $X X^+ = I$, $X^T X X^+ = X^T$.
Therefore, $X^T X X^{+\prime} = X^T X X^+$. That is $X^{+\prime} = X^+$.

Question J: The least squares solution of pseudoinverse $X^{+\prime} = (X^T X)^{-1} X^T$ is prone to numerical errors. From Question A, $X^T X = V \Sigma^2 V^T$, eigenvalues of $X^T X$ are squared singular values of X. Compared with $X^+ = V \Sigma^+ U^T$, condition number $\kappa(X^T X)$ is higher than that of $\kappa(\Sigma)$.

2 Matrix Factorization

Question A:

$$\partial_{u_i} = \lambda u_i - \sum_{j=1}^{N} v_j (y_{ij} - u_i^T v_j)$$

$$\partial_{v_j} = \lambda v_j - \sum_{i=1}^{N} u_i (y_{ij} - u_i^T v_j)$$

Question B: Let $\partial_{u_i} = 0$, and $\partial_{v_j} = 0$.
That is,

$$\begin{cases} \lambda u_i - \sum_{j=1}^{N} v_j (y_{ij} - u_i^T v_j) = 0 \\ \lambda v_j - \sum_{i=1}^{N} u_i (y_{ij} - u_i^T v_j) = 0 \end{cases}$$

From 1st equation,

$$\lambda u_i = \sum_{j=1}^{N} v_j y_{ij} - \sum_{j=1}^{N} v_j u_i^T v_j = \sum_{j=1}^{N} v_j y_{ij} - \sum_{j=1}^{N} v_j v_j^T u_i$$

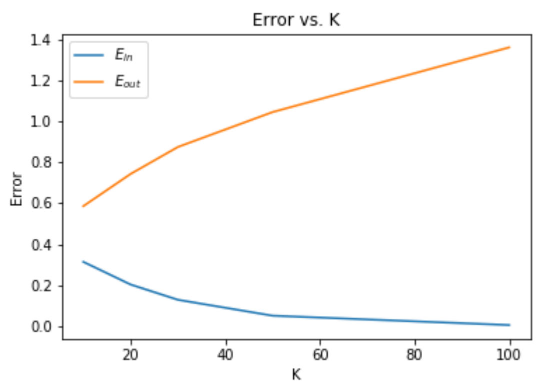
$$u_i = (\lambda I + \sum_{j=1}^{N} v_j v_j^T)^{-1} \sum_{j=1}^{N} v_j y_{ij}$$

Similarly, from 2nd equation,

$$v_j = (\lambda I + \sum_{i=1}^{N} u_i u_i^T)^{-1} \sum_{i=1}^{N} u_i y_{ij}$$

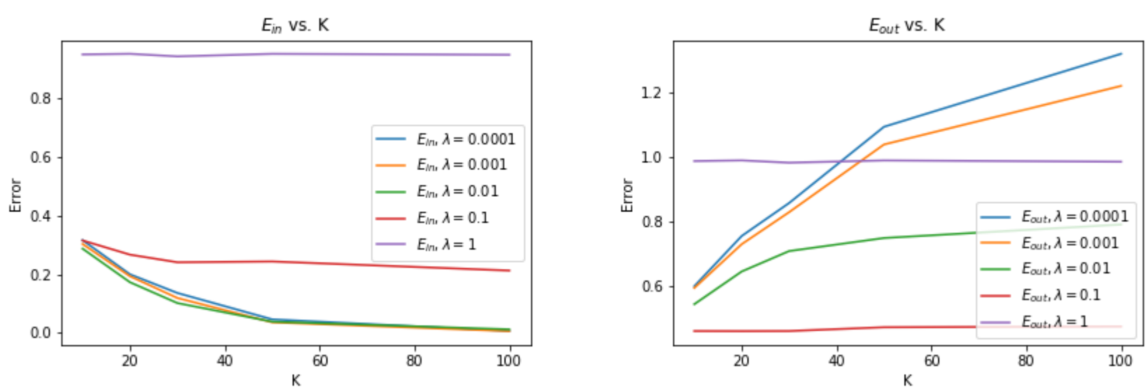
Question C: See jupyter notebook.

Question D:



As k increases, E_{in} decreases while E_{out} increases. The increase of k means more latent factors, and as more parameters our model have, overfitting occurs which has a low training error but a high out-of-sample error.

Question E:



As k increases for small regularization λ, E_{in} decreases while E_{out} increases. The increase of k means more latent factors, and as more parameters our model have while regularization term λ is small, overfitting occurs which has a low training error but a high out-of-sample error.
As λ increases, E_{in} increases while E_{out} first decrease then increase due to the penalty on overfitting. When

Changhao Xu
UID: 2103530
February 17th, 2020

λ = 0.1, testing error is the lowest while training error is low as well, indicating good performance with different k. When λ is too large and reachs 1, underfitting occurs which has a high training and testing error.

3 Word2Vec Principles

Question A: $\log p(w_O|w_I) = {v'}_{w_O}^T v_{w_I} - \log \sum_{w=1}^{W} \exp({v'_w}^T v_{w_I})$.
Therefore,

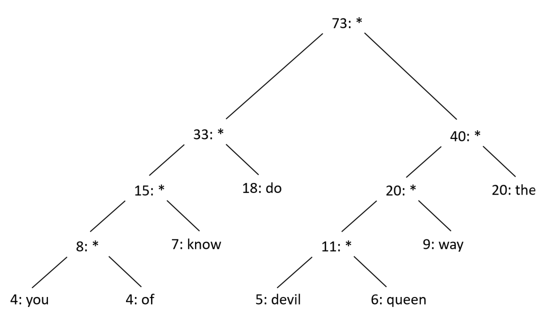
$$\nabla_{{v'}_{w_O}} \log p(w_O|w_I) = v_{w_I} - v_{w_I} \times \frac{\exp({v'}_{w_O}^T v_{w_I})}{\sum_{w=1}^{W} \exp({v'_w}^T v_{w_I})}$$

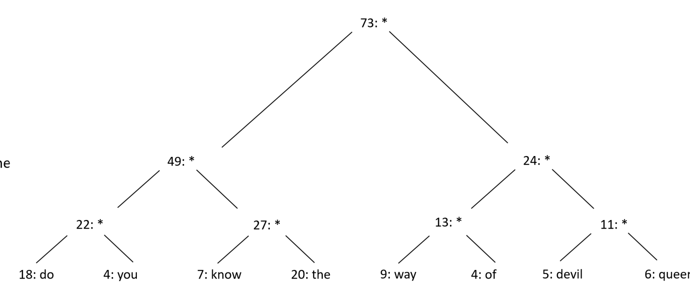
, and

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

.
That is, computing these gradients scale with O(W). Similarly, these gradients scale with O(D). So time complexity is O(WD).

Question B:





Expected representation length of Huffman tree = $\frac{(4+4+5+6)\times 4+(7+9)\times 3+(18+20)\times 2}{73} = 2.73973$, while expected representation length of balanced binary tree is 3.

Question C: The training objective will increase as D increases. A larger D means more features in the embedding space, which will lead to overfitting for very large D.

Question D: See Jupyter notebook.

Question E: (308, 10)

Question F: (10, 308)

Question G:
Pair(them, would), Similarity: 0.9628089

Pair(would, them), Similarity: 0.9628089
Pair(car, them), Similarity: 0.95937026
Pair(like, or), Similarity: 0.957788
Pair(or, like), Similarity: 0.957788
Pair(not, them), Similarity: 0.95743936
Pair(eat, would), Similarity: 0.9547398
Pair(a, eat), Similarity: 0.95205164
Pair(in, not), Similarity: 0.9470372
Pair(i, or), Similarity: 0.9458327
Pair(ned, dear), Similarity: 0.9441935
Pair(dear, ned), Similarity: 0.9441935
Pair(do, a), Similarity: 0.9412332
Pair(eleven, boat), Similarity: 0.93666583
Pair(boat, eleven), Similarity: 0.93666583
Pair(red, oh), Similarity: 0.93665606
Pair(oh, red), Similarity: 0.93665606
Pair(could, in), Similarity: 0.9363972
Pair(things, sing), Similarity: 0.9356929
Pair(sing, things), Similarity: 0.9356929
Pair(open, cans), Similarity: 0.9347308
Pair(cans, open), Similarity: 0.9347308
Pair(and, i), Similarity: 0.9308523
Pair(samiam, car), Similarity: 0.92888826
Pair(with, box), Similarity: 0.9280398
Pair(box, with), Similarity: 0.9280398
Pair(from, red), Similarity: 0.9266325
Pair(low, goodbye), Similarity: 0.92613554
Pair(goodbye, low), Similarity: 0.92613554
Pair(here, samiam), Similarity: 0.92585975

Question H: Many pairs appear at the same time, such as (them, would), (would, them). This is because they has the same similarity. Also words are more similar when they often appear closely in sentences, such as (dear, ned), (open, cans), (and, i).

2_notebook

February 18, 2020

1 Problem 2

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

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1.1 2C:

Fill in these functions to train your SVD

[2]:
```
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.
    """
    grad_U = eta * (reg * Ui - Vj * (Yij - np.dot(Ui, Vj) ) )
    return grad_U
    pass

def grad_V(Vj, Yij, Ui, 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.
    """
    grad_V = eta * (reg * Vj - Ui * (Yij - np.dot(Ui, Vj) ) )
    return grad_V
    pass
```

```python
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␣
↪user,
    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.
    """
    err = 0
    for k in range(len(Y)):
        (i, j, Y_ij) = Y[k]
        err += 0.5 * (Y_ij - np.dot(U[i-1], V[j-1]) )**2

    err += 0.5 * reg * (np.linalg.norm(U) ** 2 + np.linalg.norm(V) ** 2)

    return err/float(len(Y))
    pass


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.

    Uses a learning rate of <eta> and regularization of <reg>. Stops after
    <max_epochs> epochs, or once the magnitude of the decrease in regularized
    MSE between epochs is smaller than a fraction <eps> of the decrease in
    MSE after the first epoch.

    Returns a tuple (U, V, err) consisting of U, V, and the unregularized MSE
    of the model.
    """
    U = np.random.uniform(-0.5, 0.5, (M, K))
    V = np.random.uniform(-0.5, 0.5, (N, K))

    index = np.arange(Y.shape[0])
    np.random.shuffle(index)

    err_last = get_err(U, V, Y, reg)

    for idx in index:
        (i, j, Y_ij) = Y[idx]
```

```
        U[i-1] -= grad_U(U[i-1], Y_ij, V[j-1], reg, eta)
        V[j-1] -= grad_V(V[j-1], Y_ij, U[i-1], reg, eta)

    err_now = get_err(U, V, Y, reg)
    Delta_0 = err_last - err_now

    err_last = err_now

    epoch = 1
    delta = Delta_0

    while epoch < max_epochs and delta > eps * Delta_0:
        index = np.arange(Y.shape[0])
        np.random.shuffle(index)

        for idx in index:
            (i, j, Y_ij) = Y[idx]
            U[i-1] -= grad_U(U[i-1], Y_ij, V[j-1], reg, eta)
            V[j-1] -= grad_V(V[j-1], Y_ij, U[i-1], reg, eta)

        err_now = get_err(U, V, Y, reg)
        delta = err_last - err_now

        err_last = err_now

        epoch += 1

    err_unreg = get_err(U, V, Y, reg = 0)

    return (U, V, err_unreg)

    pass
```

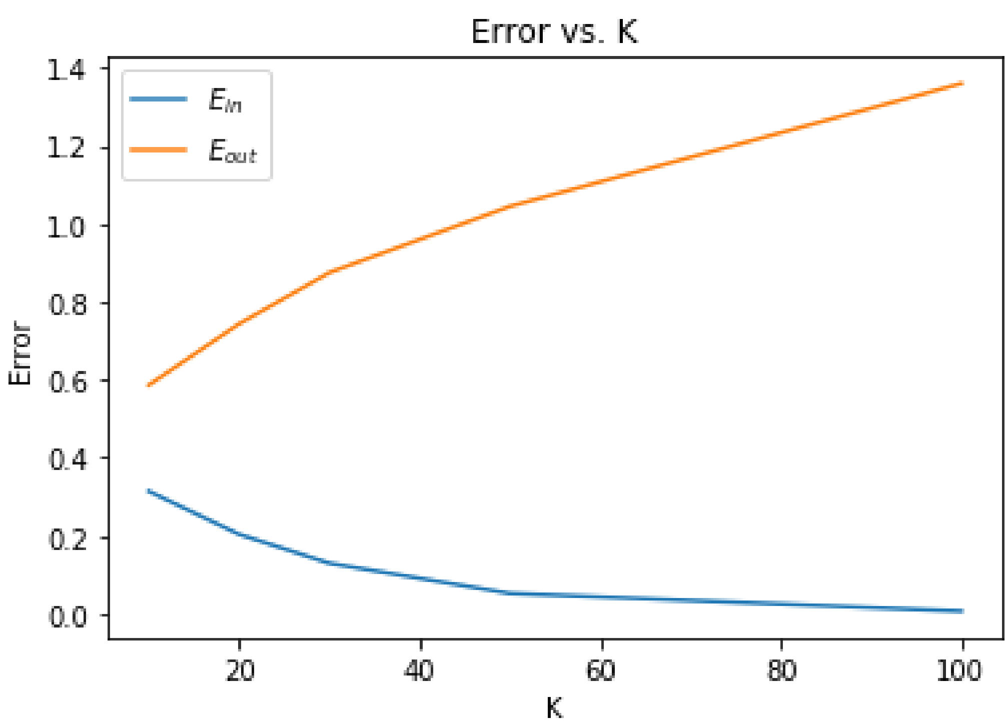
1.2 2D:

Run the cell below to get your graphs

[3]:
```
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
```

```python
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')
```

```
Factorizing with  943  users,  1682  movies.
```

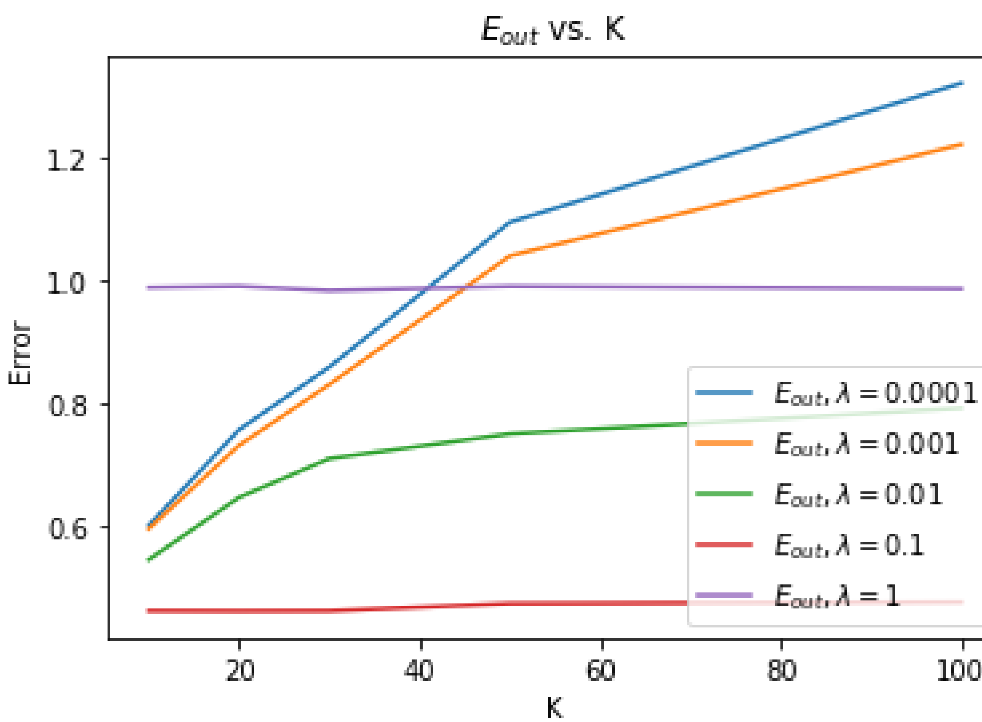


1.3 2E:

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.

[4]:
```
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')
```

```
Training model with M = 943, N = 1682, k = 10, eta = 0.03, reg = 0.0001
Training model with M = 943, N = 1682, k = 20, eta = 0.03, reg = 0.0001
Training model with M = 943, N = 1682, k = 30, eta = 0.03, reg = 0.0001
Training model with M = 943, N = 1682, k = 50, eta = 0.03, reg = 0.0001
Training model with M = 943, N = 1682, k = 100, eta = 0.03, reg = 0.0001
Training model with M = 943, N = 1682, k = 10, eta = 0.03, reg = 0.001
Training model with M = 943, N = 1682, k = 20, eta = 0.03, reg = 0.001
Training model with M = 943, N = 1682, k = 30, eta = 0.03, reg = 0.001
Training model with M = 943, N = 1682, k = 50, eta = 0.03, reg = 0.001
Training model with M = 943, N = 1682, k = 100, eta = 0.03, reg = 0.001
Training model with M = 943, N = 1682, k = 10, eta = 0.03, reg = 0.01
Training model with M = 943, N = 1682, k = 20, eta = 0.03, reg = 0.01
Training model with M = 943, N = 1682, k = 30, eta = 0.03, reg = 0.01
Training model with M = 943, N = 1682, k = 50, eta = 0.03, reg = 0.01
Training model with M = 943, N = 1682, k = 100, eta = 0.03, reg = 0.01
Training model with M = 943, N = 1682, k = 10, eta = 0.03, reg = 0.1
Training model with M = 943, N = 1682, k = 20, eta = 0.03, reg = 0.1
Training model with M = 943, N = 1682, k = 30, eta = 0.03, reg = 0.1
Training model with M = 943, N = 1682, k = 50, eta = 0.03, reg = 0.1
Training model with M = 943, N = 1682, k = 100, eta = 0.03, reg = 0.1
Training model with M = 943, N = 1682, k = 10, eta = 0.03, reg = 1
Training model with M = 943, N = 1682, k = 20, eta = 0.03, reg = 1
Training model with M = 943, N = 1682, k = 30, eta = 0.03, reg = 1
Training model with M = 943, N = 1682, k = 50, eta = 0.03, reg = 1
Training model with M = 943, N = 1682, k = 100, eta = 0.03, reg = 1
```



3_notebook

February 21, 2020

1 Problem 3

Authors: Sid Murching, Suraj Nair, Alex Cui

[1]:
```
import numpy as np
from P3CHelpers import *
from keras.models import Sequential
from keras.layers.core import Dense, Activation

import sys
```

```
Using TensorFlow backend.
```

1.1 3D:

Fill in the generate_traindata and find_most_similar_pairs functions

[2]:
```
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 to compute.

    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␣
↪<word>
    feature_representation = np.zeros(len(unique_words))
    feature_representation[word_to_index[word]] = 1
```

```
    return feature_representation

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␣
 ↪training
                              points (one-hot-encoded vectors representing␣
 ↪individual words) and
                              their corresponding labels (also one-hot-encoded␣
 ↪vectors representing words).

                              For each index i, trainX[i] should correspond to␣
 ↪a word in
                              <word_list>, and trainY[i] should correspond to␣
 ↪one of the words within
                              a window of size <window_size> of trainX[i].
    """
    trainX = []
    trainY = []
    # TODO: Implement this function, populating trainX and trainY
    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:
                point_X = get_word_repr(word_to_index, word_list[i]) # vector␣
 ↪of the word in word_list
                trainX.append(point_X) #
                point_Y = get_word_repr(word_to_index, word_list[i+j]) # vector␣
 ↪of other words in the window
                trainY.append(point_Y) #
    return (np.array(trainX), np.array(trainY))
```

[3]:

```
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␣
↪characters
    # and make all chars lowercase.
    sample_text = load_word_list(filename)
    print('sample_text length', len(sample_text))

    # 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)
    print('trainX.shape = ', trainX.shape, 'trainY.shape = ', trainY.shape)

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

    # vocab_size = number of unique words in our text file. Will be useful when␣
↪adding layers
    # to your neural network
    vocab_size = len(word_to_index) # input dim
    model = Sequential()
    model.add(Dense(num_latent_factors, input_dim=(vocab_size))) # a single␣
↪hidden layer of num_latent_factors/10 units
    model.add(Dense(vocab_size)) # output: vocab_size  vector
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='rmsprop',␣
↪metrics=['accuracy'])    # multi-class classification
    fit = model.fit(trainX, trainY)



    # TODO: 2) Extract weights for hidden layer, set <weights> variable below

    weights = None

    print('layer_0 dim = ', model.layers[0].get_weights()[0].shape) # get␣
↪layers[0] weight, get_weights()[1] gets the bias term

    print('layer_1 dim = ', model.layers[1].get_weights()[0].shape) # get␣
↪layers[0] weight, get_weights()[1] gets the bias term

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

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

1.2 3G:

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

[5]:
```python
find_most_similar_pairs('data/dr_seuss.txt', 10)
```

```
sample_text length 2071
trainX.shape =  (16548, 308) trainY.shape =  (16548, 308)
Epoch 1/1
16548/16548 [==============================] - 1s 38us/step - loss: 5.2212 -
accuracy: 0.0517
layer_0 dim =  (308, 10)
layer_1 dim =  (10, 308)
Pair(them, would), Similarity: 0.9628089
Pair(would, them), Similarity: 0.9628089
Pair(car, them), Similarity: 0.95937026
Pair(like, or), Similarity: 0.957788
Pair(or, like), Similarity: 0.957788
Pair(not, them), Similarity: 0.95743936
Pair(eat, would), Similarity: 0.9547398
Pair(a, eat), Similarity: 0.95205164
Pair(in, not), Similarity: 0.9470372
Pair(i, or), Similarity: 0.9458327
Pair(ned, dear), Similarity: 0.9441935
Pair(dear, ned), Similarity: 0.9441935
Pair(do, a), Similarity: 0.9412332
Pair(eleven, boat), Similarity: 0.93666583
Pair(boat, eleven), Similarity: 0.93666583
Pair(red, oh), Similarity: 0.93665606
Pair(oh, red), Similarity: 0.93665606
Pair(could, in), Similarity: 0.9363972
Pair(things, sing), Similarity: 0.9356929
Pair(sing, things), Similarity: 0.9356929
Pair(open, cans), Similarity: 0.9347308
Pair(cans, open), Similarity: 0.9347308
Pair(and, i), Similarity: 0.9308523
Pair(samiam, car), Similarity: 0.92888826
Pair(with, box), Similarity: 0.9280398
Pair(box, with), Similarity: 0.9280398
Pair(from, red), Similarity: 0.9266325
Pair(low, goodbye), Similarity: 0.92613554
Pair(goodbye, low), Similarity: 0.92613554
Pair(here, samiam), Similarity: 0.92585975
```