Ryan Brady 912045807
I worked on this homework with Noa Shadmon in office hours and in class; thus some of our code and homework may be similar due to us discussing the techniques of deriving functions and how to program them.

Hw 2:

Problem 1:

By multiplying ATA we can use ATA as our sparse matrix in our power method function to find the right singular vector. The time complexity of matrix multiplication is O(n^3) because we have to do dot products of rows and columns. We can see the powermethod here:

```
def power_method(mat, maxit):
    A2T = mat.transpose()
    result = np.random.rand(mat.shape[1])
    for i in range(maxit):

        result = A2T.dot(result)
        result = mat.dot(result)
        result = result/np.linalg.norm(result)
        result = result.reshape(mat.shape[0],1)
    #Is this how you comput eigenvector and value,or top singlur value and right vector??

    result2 = A2T.dot(result)
    tresult = result2.transpose()
    matts = np.matrix(tresult)
    thismat = csr_matrix(matts)
    matt = mat.transpose()
    first = thismat.dot(matt)
    second = first.dot(mat)
    tmat = thismat.transpose()
    getval = second.dot(tmat)
    A_frob = scipy.sparse.linalg.norm(mat,ord='fro')
    vals = math.sqrt(math.sqrt(a[2][0]))
    print vals/A_frob
    print 'computed eigenvalue :'  ,vals
    np.savetxt('test1.txt', result, delimiter=',')   # result is an array
    return result,vals
ressy = power_method(A2,10)
```

Part B:
From the code above we see the eigenvalue computed after the result/singular vector is computed.
ATA = $V\sigma^2VT$
So $VTATAvV= VTV\sigma^2VTV$
Thus $VTATAV= \sigma^2$
Which means that VTATAV is the squared eigenvalue.

The range of all eigenvalues has a max of the eigenvalue of the right singular matrix when v is the right singular vector. We see that the output of the power_method shows and eigenvalue of 787. In the code below we see the eigenvalue estimated for iterations 1,2,5,10,20 in descending order.

```
(phall)-~$ vi timer.py
(phall)-~$ time python2.7 timer.py
 computed eigenvalue : 753.166707535
 computed eigenvalue : 775.930513618
 computed eigenvalue : 787.372158601
 computed eigenvalue : 787.891622183
 computed eigenvalue : 787.89514152
```

Part C:
The power method has a lower time complexity than svds and is simply an approximation. Thus the power method is more efficient than svds at obtaining an approximate top eigenvalue. For accuracy svds is better but takes much longer. The SVD Calculator gives us an eigenvalue of 787.89514171 and took 3 minutes and 37 seconds to calculate. The power method for 20 iters is as follows:

```
def thesort(newlinks):
        newlinks = newlinks.values[:,0]
        print newlinks[:4]
        labs = np.absolute(newlinks)
        oo = np.argsort(labs)
        finfive = oo[-5:]
    print(finfive)
        return finfive
```

Above we see the sorting maneuever and algorithm.

```
(phall)-~$ time python2.7 timer.py
computed eigenvalue : 787.89514152

real    3m55.424s
user    3m14.254s
sys     0m12.580s
```

This took longer and thus would be less efficient than just using svds function from scipy.
The Power method for 10 iterations was slightly shorter than svds and was still very accurate.

```
(phall)-~$ time python2.7 timer.py
computed eigenvalue : 787.891618344

real    3m45.487s
user    2m58.359s
sys     0m8.678s
```

PartD:
The top 5 authority scores are found by finding the indices of the top 5 values in the right singular vector.
Similar to Part C just that we manipulated the matrices to solve for U rather than V

```
186873    186874,"List of United States tornadoes from M...
2774619                          2774620,"Manter, Kansas"
186871                  186872,"2010 United States Census"
3165770                       3165771,"Khezri Dasht Beyaz"
2038044       2038045,"List of people from Tulsa, Oklahoma"
```

Part E:

To find the top hub scores we can simply multiply our right singular vector by A.
Av will give us our hub scores.

```
4030617  4030618,"Narrative of Some Things of New Spain...
1186296                          1186297,"Bonaventure"
192859            192860,"Jeopardy! College Championship"
195374                           195375,"Crosswordese"
210477                      210478,"Scorpions Tour 2002"
```

Part F:

To find the page rank we must use the formula $x^{t+1} = x^t((\alpha PT) + (1-\alpha)/n(eeT))$ where e = [1,…,1] and PT is our transition matrix where the rows sum up to one. We also can infer eeT = n thus our equation is $x^{t+1} = x^t((\alpha PT) + (1-\alpha))$. Because our matrix is so large though, we must distribute x^t. s.t. $x^{t+1} = x^t\alpha PT + x^{t(}1-\alpha)$. To calculate page rank we set our x^0 = to [1/n,…1/n] of length n. This is similar to the power method. Our equation thus acts similar to the power method in that it iteratively solves for the right singular vector, the max values of which will give the highest page ranks.

```
def page_rank(A2,maxit):
    w_normalized = normalize(A2, norm='l1', axis=1)
    init = np.ones(w_normalized.shape[0])*(1./w_normalized.shape[0])
    e = np.ones(w_normalized.shape[0])
    e.shape = (len(e),1)
    et = e.transpose()
    result = init
    for _ in range(maxit):
        result = .9*w_normalized.dot(result) + (.1/w_normalized.shape[0])*result
    print np.argsort(result)[:5]
page_rank(A2,10)
```

```
3026971                    3026972,"Prostitution in Brunei"
3313808                         3313809,"Betalingsservice"
2239654  2239655,"Wyoming Workforce Development Council"
4204774                               4204775,"Edmund Key"
1467555       1467556,"Borris Great, Borris, County Laois"
```

Problem 2:

For this problem we use the dataset from problem one. We use the columns with both questions through preprocess functions from hw1. I then combined the lists, and use a counting function, wordcount in order to obtain a dictionary that returns the number of repetions fo each word. From this we get an n length dictionary, where the key is the count and value is the unique word.

In order to calculate the PMI we use two dicts as inputs. The function will take the pairs of words and the word count dictionary. We then use the equation from the homework, the PPMI function. In the function d is number of word pairs throughout the corpus. We proceed to multiply by two because each pair is a unique pairing both ways. WC is the word count of word w in the corpus, and c is the unique occurences of the word w. After obtaining PPMI, we update the each key in pairs dict with PPMI. Now we move on to constructing a CSR matrix. For the CSR matrix we must go through the first dictionary and obtain each ppmi value. We now have a ppmi value for each pair of words. With this new matrix we can then obtain the index of each word pair. Once we obtain all of the pairs we proceed to concat the rows and columns, and our matrix is symmetric since each pair has two values (x,y). Using the CSR matrix we find that the time complexity is only O(n) since we only have to loop once. If we need all the values in csr matrix we must do a double loop thus the time complexity is O(n^2), combining these functions we have a total time complexity of O(2n + n^2) which converge to O(n^2) when n is large.

```
def cppmi(pairsDict, count):
    d = len(pairsDict)
    for (key1, key2) in pairsDict:
        key = tuple((key1,key2))
        w = count.get(key1)
        c = count.get(key2)
        wc = pairsDict.get(key)
        num = float(np.log(float(wc*2*d)/float(w*c)))
        if num < 0:
            num = 0
        pairsDict[key] = num
    return pairsDict
```

```
def CSRmatrix(pairsDict, words):
    words2 = {y:x for x,y in words.iteritems()}
    row = []
    column = []
    data = []
    i = 0
    for key1,key2 in pairsDict:
        num = pairsDict.get(tuple(sorted((key1,key2))))
        part1 = words2.get(key1)
        part2 = words2.get(key2)
        row.append(part1)
        column.append(part2)
        data.append(num)
        i += 1
    data = np.array(data)
    row = np.array(row)
    column = np.array(column)
    data2 = np.concatenate((data,data))
    row2 = np.concatenate((row,column))
    column2 = np.concatenate((column, row))
    if (max(row2) > max(column2)):
        n = max(row2) + 1
    else:
        n = max(column2) + 1
    sparse = csr_matrix((data2, (row2, column2)), shape=(n, n))
    return sparse
```

```
(phall)--$ python2.7 problem2.py
('CSR matrix shape is: ', (102901, 102901))
('Number of non zero elements is: ', 4798083)
('Frobenius norm is: ', 10918.230499414403)
```

Problem 3:

From problem 3 we find the best accuracy was .6298 which occurs when our threshold is one. The ppmi values also provide a similar best value when the threshold is one of .6283. Most of problem three used the same code of problem2. Some results differed though. Here we calculate cosine matrix for word2vec model.

```
def COSINES(f1,f2, thresh,data):
    real = data[5]
    correct = 0
    for i in range(0, len(f1)):
        try:
            similar = (np.dot(np.transpose(f1[i]), f2[i]))/(scipy.linalg.norm(f1[i]) * scipy.linalg.norm(f2[i]))
            if (similar - thresh) < 0:
                val = 0
            else:
                val = 1
            if val == real[i]:
                correct += 1
        except:
            continue
    res = float(correct)/float(len(f1))
    return res
```

```
(phall)--$ python2.7 problem3.py
training.csv results
0.8 threshold results in 0.572884732581 accuracy value
0.82 threshold results in 0.583152184981 accuracy value
0.84 threshold results in 0.593441131979 accuracy value
0.86 threshold results in 0.601227586288 accuracy value
0.88 threshold results in 0.608064171851 accuracy value
0.9 threshold results in 0.613641777286 accuracy value
0.92 threshold results in 0.617165289537 accuracy value
0.94 threshold results in 0.617292123684 accuracy value
0.96 threshold results in 0.614795657944 accuracy value
0.98 threshold results in 0.609684741738 accuracy value
1 threshold results in 0.629786825962 accuracy value
('Max accuracy occurs when the threshold is ', 1, 'accuracy value is ', 0.62978682596280854)
Now testing on validation.csv
('Accuracy when threshold is 1 for validation.csv is ', 0.628257972159147)
```