NYU

Big Data and Machine Learning HW1b

Nick Greenquist

October 22, 2017

1) People You Might Know

a) Source Code

Attached in zip folder, folder 1, friends.py

b) Write Up

First I flatmap the input file using a custom mapper function. This function takes every line and split this into a user and the list of their friends. If a user has no friends, return a key of user,user and the value of -1000. If the user has friends, create key value pairs for the user with every friend. Each kvp will be the key of the user,friend and the value of -1000. This negative value ensures that this user will never get their own friends as recommendations. Then, for every friend of this user, create key value pairs of friend,friend and 1. This means each mutual friend will get a value of 1. Do this both ways, so create two kvp's for each group of friends so that each friend will be recommended to each other with similar strength. Return this list of new kvp's from the flat mapper. Next, reduceBykey every kvp of user,user and value by adding all their values. The -1000 values for fiends of each other will overpower the recommendation strength from this pair of users being mutual friends with other users. Finally, map the reduced rdd to a new kvp structure, that of user as the key and value as a tuple of (num, user). This tuple is the friend to recommend and how many times this user is mutual friends of friends of the original user. We then take this new rdd and sort by the num of these tuples. This gives every user key a list of recommendations ordered by that num. Also, sort ties by the user id. The final rdd contains recommendations for every user ranked by strength. I also take(10) for every one of these lists as per the question.

c) Compute and include the recommendations for the users with following user IDs: 8941, 9020, 9021, 9993

8941: 8943, 8944, 8940
9020: 9021, 9016, 9017, 9022, 317, 9023
9021: 9020, 9016, 9017, 9022, 317, 9023
9993: 9991, 13134, 13478, 13877, 34299, 34485, 34642, 37941

d) Output results

For full output (and where I got my answers to part c above), look in the zipped folder, folder 1/reccomendations.txt. This text file contains the top ten recommendations

e) Input

Inside the 1/inputs folder inside the zip

f) How to test this code

spark-submit friends.py /user/<userid>/<inputfile> /user/<userid>/<outputfolder>
To run on my input:
spark-submit friends.py /user/ntg251/friends.txt /user/ntg251/Output/friendsoutput
To gather results:
hadoop fs -getmerge /user/ntg251/Output/friendsoutput $HOME/<outputpath>/output.txt

2) Page Rank

Source Code

Located in the zipped folder in 2/pagerank.py

a) Short description of Map and Reduce tasks

Start by mapping each line using a custom mapper function. This function takes every line that is not a comment and splits this by white space. Then, the src node is the left side and the dest node is the right side. Return a key value pair of src,dest. Next, take the distinct of this rdd and then group all values by key. Now, find the count of number of nodes and set this to a global variable. This will be our n. Next, map each kvp again using another function that returns a new rdd with key value pairs of src,1.0. These are the initial ranks for each node. Next, loop 40 times and do the following. Use a join command to join the firs rdd (the pairs of nodes) and the ranks rdd. Now, we flat map over this joined rdd. Extract all the links that are linked to by the src node and the rank of the src node. Now, the value of the rank passed to each dest link is value/n where n is the number of links the src node has going out. This is the same as how much probability the src node has of picking an out node and passing its rank to it. Yield a new key value pair of (link, value). Reduce the new rdd by key and add up all the ranks given by src links. For simple page rank, this is good enough and you use these new ranks over and over for 40 iterations. However, this does not deal with spider traps and dead ends. So, do escape those, we use a damping function. We use a mapValues functions to take every rank and multiply the rank by 0.8, and add the result of 0.2 divided by the total number of nodes in this web, the gloabl n. After these 40 iterations, the ranks rdd contains the final ranks of every node. Sort this by the rank and print the results.

b) Two output files for each graph, each containing the list of all node IDs and their corresponding PageRank scores, in a simple format as below. <Vertex ID><TAB><PageRank>. Compute both simple PageRank as well as modified PageRank assuming dead ends and spider traps. Use B = 0.8

The 4 files contained in the zip contain the output results for each graph file using damping and no damping (simple page rank). They are named accordingly. The files with 'graphlarge' are the results for page rank on the Notre Dame input. The files with 'graph' are from the smaller input.

c) Input files

Input files are inside the zip file in 2/inputs

d) Source Code

Source code for this problem is in the zip, folder 2, pagerank.py. I used this source code to generate all 4 results. For simple page rank, I just commented out the call to dampen the rank.

e) How to test this code

spark-submit pagerank.py /user/<userid>/<inputfile> output.txt
NOTICE: This will write results to the same LOCAL folder as the pagerank.py file. No need to getmerge from hadoop
To run on my input:
spark-submit pagerank.py /user/ntg251/graphlarge.txt output.txt

f) Assumptions

I chose 40 for the iterations as the textbook mentioned this was optimal enough to converge on the correct answer.

3) Document Similarity and Clustering

The Source Code

PySpark code: inside the zipped folder in 3/kmeans.py
Local Python code: inside the zipped folder in 3/kmeans_local.py

How to run

spark-submit kmeans.py /user/ntg251/Articles.csv <local output file for char matrix> <local output file for signatures> <k>

Input files

Inside the zipped submission in 3/inputs

a1) The characteristic matrix obtained from shingling and hashing

Inside the zip file inside the folder 3/results/charM.txt and 3/output/signatures.txt
The file contains tuples of shingle,document. These can be used to make a matrix where shingle is the row and document is the column and the value is 1. Excluded tuples would be 0 in this matrix.

Here are example tuples for one document that represent the row and column of 20 1 values in this matrix:
abdullah,oil hits new 5.5 year lows as saudis defend

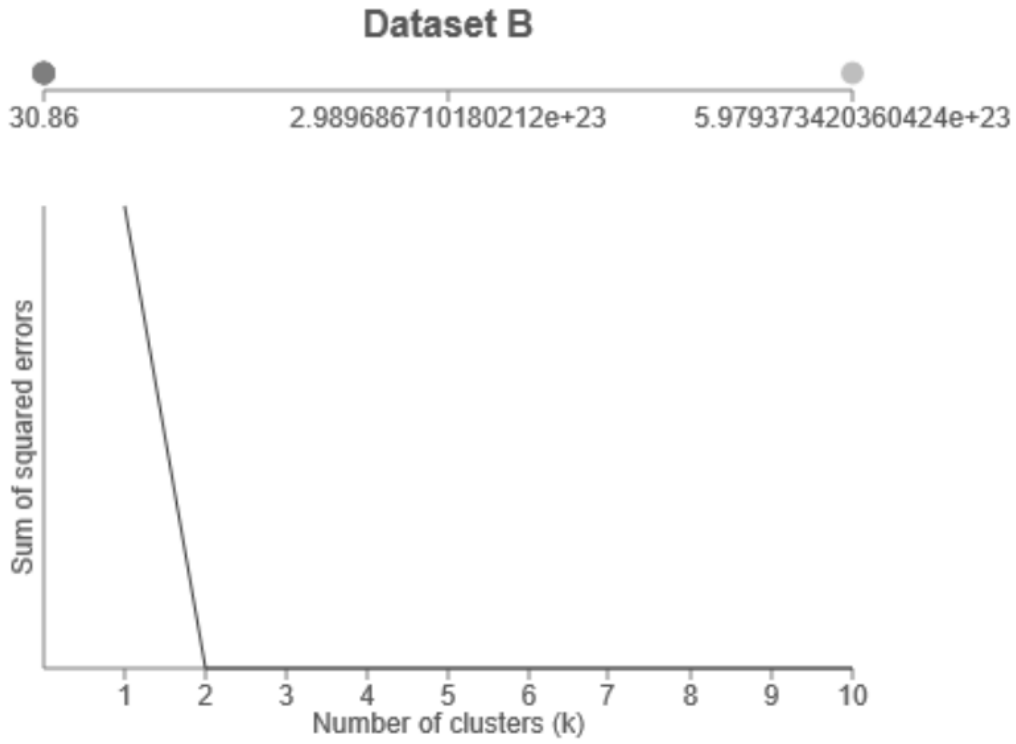
adding,oil hits new 5.5 year lows as saudis defend
afp,oil hits new 5.5 year lows as saudis defend
afpjames,oil hits new 5.5 year lows as saudis defend
ailing,oil hits new 5.5 year lows as saudis defend
analyst,oil hits new 5.5 year lows as saudis defend
appeared,oil hits new 5.5 year lows as saudis defend
approach,oil hits new 5.5 year lows as saudis defend
april,oil hits new 5.5 year lows as saudis defend
arabia,oil hits new 5.5 year lows as saudis defend
arabias,oil hits new 5.5 year lows as saudis defend
back,oil hits new 5.5 year lows as saudis defend
barrel,oil hits new 5.5 year lows as saudis defend
barrelbasically,oil hits new 5.5 year lows as saudis defend
behalf,oil hits new 5.5 year lows as saudis defend
benchmark,oil hits new 5.5 year lows as saudis defend
blame,oil hits new 5.5 year lows as saudis defend
blamed,oil hits new 5.5 year lows as saudis defend
brent,oil hits new 5.5 year lows as saudis defend
concerns,oil hits new 5.5 year lows as saudis defend

Here is an example document with it's 100 numHash signature matrix:
Oil stays below 40 after OPEC decides against output cu,[0.004684683617607165, 0.013695672339425636,
0.007220399540777785, 0.001458158012974921, 0.004514163809895409, 0.00254706758116231 93, 0.007645976702983
0.0056998568853601226, 0.001447776560271939, 0.00062821828540804, 0.0016925099712359604, 0.00155583023481
0.0036704281682482867, 0.007762013907444149, 0.011393059936609143, 0.003957885769342937, 0.00344837362605
0.005627906861617555, 1.2873206242663298e-05, 0.003932823180455633, 0.0004683767894688873, 0.017635351218
0.0003373378410329885, 0.005811071235880706, 0.013178334059735059, 0.0012779331255776349, 0.0006645962107
0.0015767198000916287, 0.003957413588799256, 0.00832946828451426, 0.010620947657312682, 0.003587834058837
0.011295990280471777, 0.004203419419226402, 0.004506429874432169, 0.0038258537982153226, 0.00099297659124
0.014734049276213456, 0.00470856435815141, 0.004760950088637357, 0.013439661776275624, 0.011254808593349
0.00016098655704064333, 0.00836061357394578, 0.004495282408914241, 0.005823340712266 4825, 0.000871645050
0.005683511941402061, 0.005275452491098133, 0.0016152707337799806, 0.004480270653217086, 0.00102436518870
0.00778061358334748, 0.008060652268838652, 0.009797148139458797, 0.00019260728664484124, 0.018318258161943
0.002187219207918204, 0.0010384723973513846, 0.013314459193563813, 0.005026110896051008, 0.00830123151547
0.017618224428902995, 0.0029013717911391106, 0.005491662285669024, 0.0011728324420767634, 0.003017195722
0.004512725615014582, 0.01082740045096469, 0.000572924733023655, 0.005167008825227354, 0.001022160981005
0.0010238876530531992, 0.0006857125530285556, 0.0025378886987296096, 0.002744228336689196, 0.000379237112
0.005604068030589023, 0.002106667023245244, 0.008507448684514562, 0.002268744624678239, 0.002079039339165
0.003507866744739469, 0.0016036278046540875, 0.0016561229655421702, 0.016732342017119487, 0.001901230302
0.0021879996562330062, 0.0005554950311005988, 0.0014367396893093607, 0.0003817595993805 6906, 0.003367860
2.3238360800646382e-05, 0.004986176948344183, 0.003337145305691012, 0.004056317717571565, 0.002768188006
0.009144754117082033, 0.006846854672137923, 0.004190466584903887]

a) The elbow plot that was created to determine optimal number of cluster for k-Means.

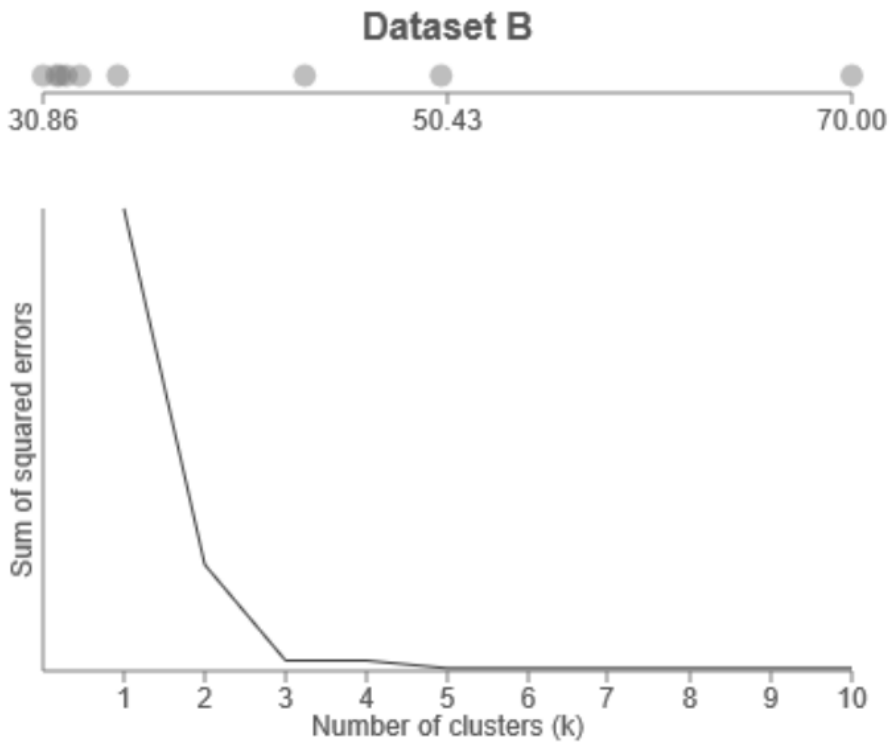
SSE scores that generate these graphs can be found in folder 3/results/ssescores.txt

Elbow graph of SSE scores when numbers are between 0-1

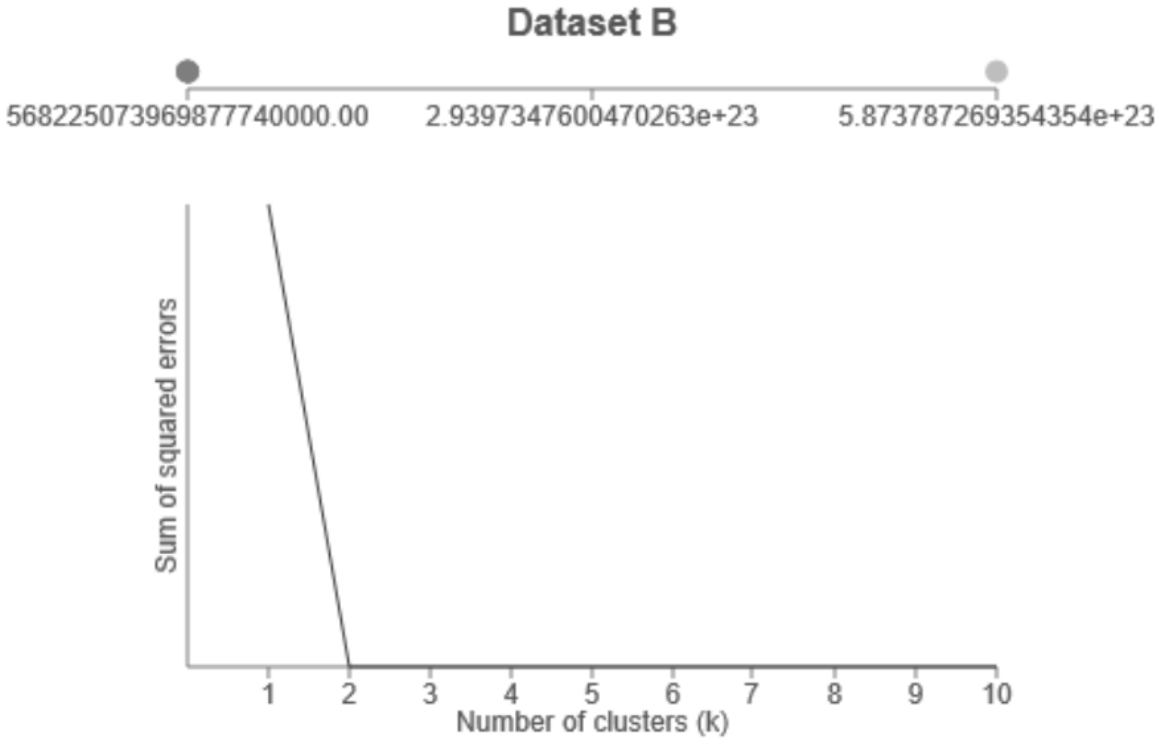


Dataset B:)76, 31.508846151532232, 30.864387055571864

Elbow graph of SSE scores when numbers are between 0-1 and first SSE is reduced to allow a better graph to appear (won't be as accurate)

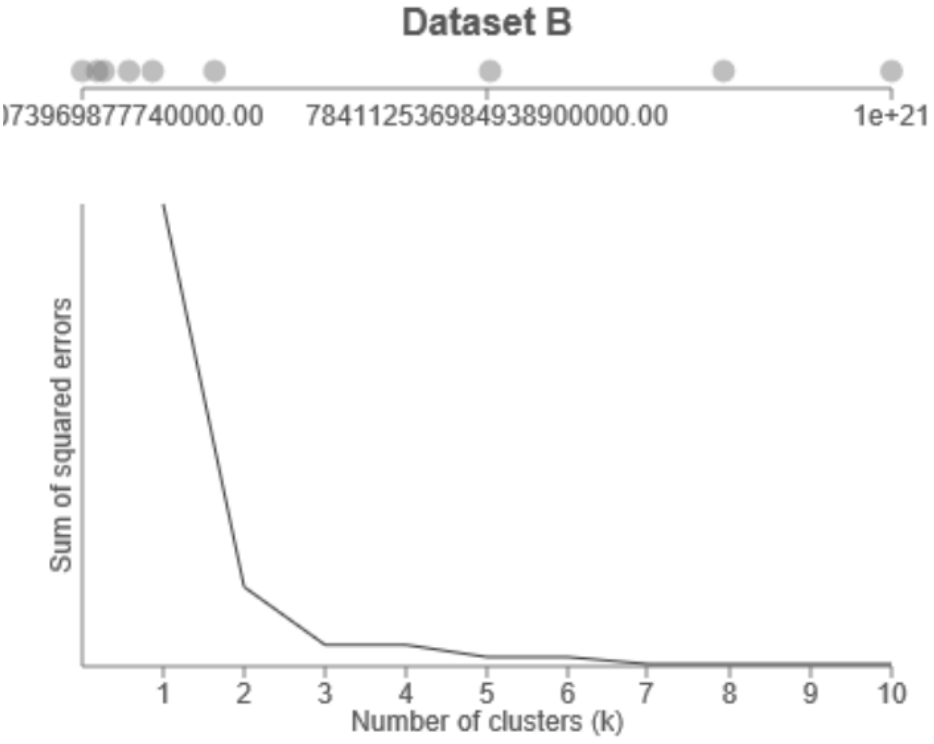


Dataset B: 70, 50.126275748791045, 43.52509560047904, 3



Dataset B: 1655430897655e+20, 5.6822507396987775e+20

Elbow graph of SSE scores when numbers are unbounded and first SSE is reduced to allow a better graph to appear (won't be as accurate)



Dataset B: 10e+20, 9.104420137063755e+20, 7.8585595604

It appears that leaving SSE scores as actual, we get an obvious point to k=2 as optimal. However, I also tested reducing the k=1 SSE to see what the elbows would look like and we get k=3 as the obvious answers. As such, I tested accuracy of k=2 and k=3 clustering using jaccard similarities. As expected, k=2 has better average results vs k=3. Use the jaccard.txt files to verfify this finding.

b) Short write-up detailing the assumptions and flow of computations in your code. Also briefly state, from your manual observation, the performance of your clustering. In particular, highlight any bad examples (most likely as a result of using shortened minhash signatures). Also mention the rationale behind your choices, wherever you have made them, such as (but not limited to) the shingles, k in k-Means and size of signatures.

Start by reading the file as a wholeTextFiles. Why? Because the data is horribly dirty and new lines are present even in articles. Since textFile splits by line, using simple textFile will give erroneous documents. So, parse the entire text file using map reduce wholeTextFiles. Split using regex on 'sports' or 'business'. This gives each article and info like date and heading. Next, extract the heading and article using more regex. If anywhere along the way errors occur, return -1,and the empty list. Next, cleanup the article. Remove punctuation, lowercase it all, remove non characters, encode it to remove non ascii characters, and also remove duplicates. Finally remove stop words and return the key value pair of heading,shingle list. We then print this rdd into the characteristic matrix as seen in the output. Next, generate the random hash functions using the function pickRandomCoeffs and a large prime number. Next, pass the rdd of document,shinglelist into the hashShingles mapper. This hashes every

shingle into a integer. I experimented with trying to hash to 32 bit, hash to 32 bit then mod by some prime, but nothing changed results. The important thing is hash words to integers. Next, pass this new hashed rdd into a signature matrix generater mapper, getSig. This loops 100 times (the num hashes recommended by the text book) and for every shingle in the document, run the shingle int through the random hash functions. Add the minimum hashcode generated by these functions to the signature matrix. Return the new rdd with key as the heading like before the value as a list of 100 hash signatures. We are now ready to perform kmeans using this signature matrix.

I run 30 iterations because after reading some research papers online, using this number converges to within over 10 decimal places of the actual means. So, first we get random centers (k of them) from the signature matrix. In this case, we pick 2 random signature vectors as the initial centroids. Next, we se the initial cluster to signature matrix as we will add which cluster it is assigned to later. Now, repeat the foloowing steps 40 times. First, generate a new cluster using the signature matrix mapped using the distant mapper function. We pass in all the centers to the function as a python list. Inside this function, we loop through every center and compute the euclidean distance of the signature vector to the center. We assign the center to each vector that has the least distance to the vector. Return this new rdd which is the signature matrix with new clusters. Next, generate new centers by using the getTotals function. This function makes the centers have the vector values of all the values in each of the signature vectors assigned to it divided by the number of vectors in this cluster. After 40 iterations, the final centers will be the true cluster centers and each signature vector will have its final cluster assigned to it. We then print out all this information to cluster.txt.

c) List of references (if any)

I studied the following blog posts to understand how kmeans and minhashing could be implemented. Both were for local python so both had to be dissected and the logic implemented in map reduce.
local python code for minhashing: https://github.com/chrisjmccormick/MinHash/blob/master/runMinHashExample.py
local python code for kmeans: https://gist.github.com/larsmans/4952848#file-kmeans-py-L73

I also used the MMDs book, class slides, and Ulman's video series on minhashing.

d) Jaccard Distances from cluster comparisons

The following results file contains thousands of comparisons between documents and also shows their article headings:
Inside the zip file inside the folder 3/results/jaccard-k2.txt
3/results/jaccard-k3.txt contains the jaccard distances of documents using 3 clusters. There are more regions now. At the end of the file explains every region.
This k=2 file is split into 3 regions: the first is random 50 documents all classified to cluster 0 compared to themselves (using jaccard distance of (1 - jaccard similarity)). The second region is cluster 1 vs 1. The third regions is cluster 0 v 1. Finally, the last 3 lines highlight the average jaccard distances of each of these 3 regions. We expect to see the distance of 0v1 to be higher (less similar) than in 0v0 and 1v1.

Below is a table of a more limited selection of Jaccard Distances

Cluster	0 vs 0	1 vs 1	0 vs 1
Average	**0.9838570307**	**0.9890670303**	**0.9931611298**
	0.9795918367	0.9743589744	0.9898989899
	0.9949748744	1	0.9949748744
	0.9949748744	1	0.9898989899
	1	1	1
	0.9743589744	0.9847715736	1
	0.9847715736	1	0.9949748744
	1	1	1
	0.9847715736	1	0.9949748744
	0.9690721649	0.9949748744	0.9637305699
	0.9473684211	0.9949748744	1
	0.9949748744	0.9690721649	1
	0.9949748744	0.9189189189	0.9949748744
	0.9847715736	1	0.9898989899
	0.9898989899	1	0.9847715736
	1	1	0.9949748744
	0.9795918367	0.9847715736	0.9898989899
	0.9795918367	0.9637305699	1
	0.9743589744	1	0.9847715736
	0.9898989899	0.9473684211	0.9949748744
	0.9690721649	1	1
	0.9898989899	1	1
	0.9743589744	0.9847715736	1
	0.9949748744	0.9898989899	1
	1	0.9949748744	0.9898989899
	1	1	1
	0.9795918367	1	0.9795918367
	0.9690721649	1	1
	0.9743589744	0.9898989899	0.9949748744
	0.9898989899	1	0.9898989899
	1	0.9637305699	0.9949748744
	0.9898989899	0.9949748744	0.9898989899
	0.9637305699	1	1
	0.9795918367	0.9847715736	0.9847715736
	0.9898989899	1	0.9949748744
	0.9795918367	0.9743589744	0.9847715736
	0.9690721649	1	1
	0.9898989899	1	0.9949748744
	0.9795918367	1	0.9847715736
	0.9304812834	0.9417989418	1
	0.9847715736	1	0.9949748744
	0.9898989899	1	1
	0.9898989899	1	0.9898989899
	0.9898989899	0.9949748744	0.9949748744
	1	1	1
	0.9898989899	1	1
	0.9795918367	0.9528795812	0.9898989899
	0.9949748744	0.9743589744	0.9743589744
	0.9743589744	0.9949748744	0.9949748744
	0.9847715736	0.9949748744	0.9795918367

e) Jaccard Distances of other documents (articles)

As you can see, the jaccard Distance about two news articles both about Pakistain worker remittances have extremely similar shingles
0.48484848484848486: cluster=1:Pakistan workers remittances rise 6 in 7 vs cluster=1:Overseas Pakistani workers remit US1784163m in 11

Here, we see another highly similar pair of articles both about the Pakistan forex
0.5611510791366907: cluster=1:Pakistans forex reserves drop by 109 milli vs cluster=1:Pakistans forex reserves rise to over 20521 billi

Here is a similar pair of articles both about Pakistan exports
0.7951807228915663: cluster=1:Mobile phone import rises by 677 per vs cluster=1:Pakistan Jewellary exports up 29 FY16

Two similar articles, both about Tokyo stocks
0.717948717948718: cluster=1:Tokyo stocks open higher on BoJ stimulus vs cluster=1:Tokyo stocks open lower after BoJ under

Two articles about CNG
0.7421383647798743: cluster=1:Sale of CNG in litres illegal in Sindh OGRA vs cluster=1:CNG supply resumes Sindh three day long stri

Two articles about the Pakistan economy
0.7878787878787878: cluster=1:Pakistan shares end higher rupee stab vs cluster=1:Pakistan Stock Exchange ends all time hig

Two articles about the Pakistan women's soccer team
0.6486486486486487: cluster=0:Sana Mir to lead Pakistan in Womens World T20 vs cluster=0:Pakistan squad named for Womens T20 World Cu

Two articles about Sri Lankan sports
0.8571428571428572: cluster=0:Rangana Herath spins Sri Lanka out of troub vs cluster=0:Australia skittle out Sri Lanka just 117

f) Reflections

My minhashing works very very well. Manually inspecting documents in the same category (sports and business) and computing their jaccard similarities results in very accurate results. However, I could never get my k-means to cluster in a way that showed the same accuracy. I believe this is because signature matrices just contain too much noise. For example, once stop words are removed, most articles, even in the same category, would only have a small number of common words and a large number of uncommon words. For example, above, there are many articles about the same EXACT topic and they still only share about 20 percent of their words. So, 80 percent of the uncommon words will be hashed and the euclidean distances will be at the mercy at how the minhash decided to hash these. One thing I tried was to leave stop words in so the documents would share more shingles. All that happened was that articles that happened to be written with more simple stop words were shown to be closer. This obviously is not something you want to do: cluster words based on similarities of

words that have no meaning! I tried many many many different ways to improve the performance of my kmeans results and nothing worked. I strongly believe that using minhash signature matrices in kmeans to compute distances is not an effective strategy as majority of signature values are not similar and skew the distances way too much vs the closeness found from similar words. I believe using tf-idf matrix would have been a much better vector representation of documents for kmeans.

Other tidbits

My kmeans was very slow on HDFS. I believe this was because the overhead of running so many iterative jobs was slower on this size of input vs local kmeans. I used my HDFS code to compute the signature matrices and the cluster. Then, I used this output to do comparisons using local python code. My thinking was this: Once the map reduce code generates the signature matrix and the cluster assignments, local code is actually faster to gather meaningful results from this data as the information was been so reduced and compressed thanks to map reduce. What we have after using map reduce is a matrix with only 100 rows each vs the total number of unique shingles in the input. This is very manageable for local python code to run over and computer jaccard similarities.

(i) Any assumptions you may make about formats.

There are only two categories in this Articles.csv (business and sports). So, my regex to split the wholetextfile splits the csv by these categories and THEN splits by commas. I could not use sc.textFile because the input was really dirty and newlines were inside articles and this ruined a line by line textFile map. So, I had to split input using wholeTextFiles. Also, lines didn't split right. So, I used a lot of error checks before adding an article as a valid article. I'd check if the info part of the pair had at least a date, a heading, and a category. I'd only add an article if this was found.

(ii) Any assumptions within your code

I use the prime 4294967311 to hash the shingles to the signature matrix as this number is large enough to contain almost all possible english words (to avoid shingle collisions). I'd hash each shingle using Python's standard hasher and then pass this 32bit value through my custom hashers than mod the result hash by the large prime above. The total values that could be in the signature matrix would be between 1 and 4294967311. I also experimented with modding the intial hash by this prime and the results didn't improve. I also tried dividing each value in the signature matrix by the prime to get values between 0 and 1 and still this didn't really change the performance of k means. Jaccard distances were still accurate no matter what method I used.

(iii) Any assumptions on the type of output

My output would be a tuple of the cluster number, then the heading of the article and the signature matrix for that article. I could then use this output to then compare the jaccard distance of two articles. I could use the cluster it was assigned to to check my accuracy.

(iv) Any assumptions on iterations, parameters etc.

I assumed my SSE scores to heavily point to k=2 as the optimal cluster numbers as the input we were given were only two sets of articles (sports and business). I also used 30 as the number of iterations as this was

mentioned in the book as an optimal number where the answer will converge. However, my kmeans does not perform well at all to based on checking it's result clusters using jaccard distance so there may very well be a better k to use. I tested using k=3 and my jaccard distances were worse so I believe 2 is the correct answer.

(v) Any cases that is handled or not handled in your code.

The input contains MANY dirty unicode characters. No matter how much I cleaned this data, some remained. So, some document headings and shingles contain dirty unicode characters and thus some documents have to removed from the application. I capture these errors using try catch whenever printing the strings. I also remove any non characters from the articles and also try catch every output print. Locally, I don't handle any article that has a heading that isn't between 2 and 12 words as anything less than or greater than this range was usually a piece of an article that accidently got split at. I also removed the first 200 articles in the csv that has newline errors. My hadoop version splits on the original article file using 'business' and 'sports' as the regex splits.