Implementing Locality Sensitive Hashing from scratch

Locality Sensitive Hashing (LSH) provides for a fast, efficient approximate nearest neighbor search. The algorithm scales well with respect to the number of data points as well as dimensions.

In this assignment, you will

- Implement the LSH algorithm for approximate nearest neighbor search
- Examine the accuracy for different documents by comparing against brute force search, and also contrast runtimes
- Explore the role of the algorithm's tuning parameters in the accuracy of the method

NOTICE: This assignment requires SciPy 0.16.0 or later. To upgrade, run

```
!conda update -y scipy
```

in a blank notebook.

If you are using GraphLab Create

An IPython Notebook has been provided below to you for this assignment. This notebook contains the instructions, quiz questions and partially-completed code for you to use as well as some cells to test your code.

- Download the Wikipedia people dataset in SFrame format: people_wiki.gl.zip
- Download the companion IPython notebook:1_nearest-neighbors-lsh-implementation_blank.ipynb
- Save both of these files in the same directory (where you are calling IPython notebook from) and unzip the data file.

Open the companion IPython notebook and follow the instructions in the notebook. The instructions below do not apply to users of GraphLab Create.

If you are not using GraphLab Create

It is possible to complete this assignment without using GraphLab Create. The instructions below are geared towards Python users, but you are free to adapt them to your specific environment.

Disclaimer. We have tested the assessment using the standard Python installation (with access to scikit-learn). However, the assessment may not be compatible with other tools (e.g. Matlab, R).

Download the dataset

- Download the Wikipedia people dataset in SFrame format: people_wiki.gl.zip. (Those experimenting with other tools, get people_wiki.csv.zip instead.)
- Download the mapping between words and integer indices:people_wiki_map_index_to_word.gl.zip (or alternatively,people_wiki_map_index_to_word.json.zip)
- Download the pre-processed set of TF-IDF scores:people_wiki_tf_idf.npz

Import packages

```
import numpy as np                                                  # dense matrices
import sframe                                                       # see below for install
    instruction
from scipy.sparse import csr_matrix                                 # sparse matrices
from scipy.sparse.linalg import norm                                # norms of sparse matrices
from sklearn.metrics.pairwise import pairwise_distances             # pairwise distances
from copy import copy                                               # deep copies
import matplotlib.pyplot as plt                                     # plotting
%matplotlib inline
```

This assignment requires SciPy 0.16.0 or later. To upgrade, run

```
!conda update -y scipy
```

in a blank cell.

About SFrame. SFrame is a dataframe library Dato has released free-of-charge. Its source code is available here. You may install SFrame via pip:

```
pip install --upgrade sframe
```

Load in the dataset

```
wiki = sframe.SFrame('people_wiki.gl/')
wiki = wiki.add_row_number()             # add row number, starting at 0
```

Extract TF-IDF vectors

As in the previous assignment, we extract the TF-IDF vector of each document.

For your convenience, we extracted the TF-IDF vectors from the dataset. The vectors are packaged in a sparse matrix, where the i-th row gives the TF-IDF vectors for the i-th document. Each column corresponds to a unique word appearing in the dataset.

To load in the TF-IDF vectors, run

```
def load_sparse_csr(filename):
    loader = np.load(filename)
    data = loader['data']
    indices = loader['indices']
    indptr = loader['indptr']
    shape = loader['shape']

    return csr_matrix( (data, indices, indptr), shape)

tf_idf = load_sparse_csr('people_wiki_tf_idf.npz')
```

The word-to-index mapping is given by

```
map_index_to_word = sframe.SFrame('people_wiki_map_index_to_word.gl/')
```

(Optional) Extracting TF-IDF vectors yourself. We provide the pre-computed TF-IDF vectors to minimize potential compatibility issues. You are free to experiment with other tools to compute the TF-iDF vectors yourself. A good place to start issklearn.TfidfVectorizer. Note. Due to variations intokenization and other factors, your TF-IDF vectors may differ from the ones we provide. For the purpose the assessment, we ask you to use the vectors from people_wiki_tf_idf.npz.

Train an LSH model

LSH performs an efficient neighbor search by randomly partitioning all reference data points into different bins. Today we will build a popular variant of LSH known as random binary projection, which approximates cosine distance. There are other variants we could use for other choices of distance metrics.

The first step is to generate a collection of random vectors from the standard Gaussian distribution.

```
def generate_random_vectors(num_vector, dim):
    return np.random.randn(dim, num_vector)
```

To visualize these Gaussian random vectors, let's look at an example in low-dimensions. Below, we generate 3 random vectors each of dimension 5.

```
# Generate 3 random vectors of dimension 5, arranged into a single 5 x 3 matrix.
np.random.seed(0) # set seed=0 for consistent results
print generate_random_vectors(num_vector=3, dim=5)
```

We now generate random vectors of the same dimensionality as our vocubulary size (547979). Each vector can be used to compute one bit in the bin encoding. We generate 16 vectors, leading to a 16-bit encoding of the bin index for each document.

```
# Generate 16 random vectors of dimension 547979
np.random.seed(0)
random_vectors = generate_random_vectors(num_vector=16, dim=547979)
print random_vectors.shape
```

Next, we partition data points into bins. Instead of using explicit loops, we'd like to utilize matrix operations for greater efficiency. Let's walk through the construction step by step.

We'd like to decide which bin document 0 should go. Since 16 random vectors were generated in the previous cell, we have 16 bits to represent the bin index. The first bit is given by the sign of the dot product between the first random vector and the document's TF-IDF vector.

```
doc = corpus[0, :] # vector of tf-idf values for document 0
print doc.dot(random_vectors[:, 0]) >= 0 # True if positive sign; False if negative sign
```

Similarly, the second bit is computed as the sign of the dot product between the second random vector and the document vector.

```
print doc.dot(random_vectors[:, 1]) >= 0 # True if positive sign; False if negative sign
```

We can compute all of the bin index bits at once as follows. Note the absence of the explicit for loop over the 16 vectors. Matrix operations let us batch dot-product computation in a highly efficent manner, unlike the for loop construction. Given the relative inefficiency of loops in Python, the advantage of matrix operations is even greater.

```
print doc.dot(random_vectors) >= 0 # should return an array of 16 True/False bits
print np.array(doc.dot(random_vectors) >= 0, dtype=int) # display index bits in 0/1's
```

All documents that obtain exactly this vector will be assigned to the same bin. We'd like to repeat the identical operation on all documents in the Wikipedia dataset and compute the corresponding bin indices. Again, we use matrix operations so that no explicit loop is needed.

```
print corpus[0:2].dot(random_vectors) >= 0 # compute bit indices of first two documents
print corpus.dot(random_vectors) >= 0 # compute bit indices of ALL documents
```

We're almost done! To make it convenient to refer to individual bins, we convert each binary bin index into a single integer:

```
Bin index                       integer
[0,0,0,0,0,0,0,0,0,0,0,0]   => 0
[0,0,0,0,0,0,0,0,0,0,0,1]   => 1
[0,0,0,0,0,0,0,0,0,0,1,0]   => 2
[0,0,0,0,0,0,0,0,0,0,1,1]   => 3
...
[1,1,1,1,1,1,1,1,1,1,0,0]   => 65532
[1,1,1,1,1,1,1,1,1,1,0,1]   => 65533
[1,1,1,1,1,1,1,1,1,1,1,0]   => 65534
[1,1,1,1,1,1,1,1,1,1,1,1]   => 65535 (= 2^16-1)
```

By the rules of binary number representation, we just need to compute the dot product between the document vector and the vector consisting of powers of 2:

```
doc = corpus[0, :]  # first document
index_bits = (doc.dot(random_vectors) >= 0)
```

```
powers_of_two = (1 << np.arange(15, -1, -1))
print index_bits
print powers_of_two           # [32768, 16384, 8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 1
    8, 4, 2, 1]
```

Since it's the dot product again, we batch it with a matrix operation:

```
index_bits = corpus.dot(random_vectors) >= 0
print index_bits.dot(powers_of_two)
```

This array gives us the integer index of the bins for all documents.

Now we are ready to complete the following function. Given the integer bin indices for the documents, you should compile a list of document IDs that belong to each bin. Since a list is to be maintained for each unique bin index, a dictionary of lists is used.

1. Compute the integer bin indices. This step is already completed.

2. For each document in the dataset, do the following:

- Get the integer bin index for the document.
- Fetch the list of document ids associated with the bin; if no list yet exists for this bin, assign the bin an empty list.
- Add the document id to the end of the list.

```
def train_lsh(data, num_vector=16, seed=None):

    dim = corpus.shape[1]
    if seed is not None:
        np.random.seed(seed)
    random_vectors = generate_random_vectors(num_vector, dim)

    powers_of_two = 1 << np.arange(num_vector-1, -1, -1)

    table = {}

    # Partition data points into bins
    bin_index_bits = (data.dot(random_vectors) >= 0)

    # Encode bin index bits into integers
    bin_indices = bin_index_bits.dot(powers_of_two)

    # Update `table` so that `table[i]` is the list of document ids with bin index equal to i
    for data_index, bin_index in enumerate(bin_indices):
        if bin_index not in table:
            # If no list yet exists for this bin, assign the bin an empty list.
            table[bin_index] = ... # YOUR CODE HERE
        # Fetch the list of document ids associated with the bin and add the document id to th
            end.
        ... # YOUR CODE HERE

    model = {'data': data,
             'bin_index_bits': bin_index_bits,
             'bin_indices': bin_indices,
             'table': table,
```

```
            'random_vectors': random_vectors,
            'num_vector': num_vector}

```

Checkpoint.

```
model = train_lsh(corpus, num_vector=16, seed=143)
table = model['table']
if   0 in table and table[0]   == [39583] and \
   143 in table and table[143] == [19693, 28277, 29776, 30399]:
    print 'Passed!'
else:
    print 'Check your code.'
```

Note. We will be using the model trained here in the following sections, unless otherwise indicated.

Inspect bins

Let us look at some documents and see which bins they fall into.

```
print wiki[wiki['name'] == 'Barack Obama']
```

Quiz Question. What is the document id of Barack Obama's article?

Quiz Question. Which bin contains Barack Obama's article? Enter its integer index.

Recall from the previous assignment that Joe Biden was a close neighbor of Barack Obama.

Quiz Question. Examine the bit representations of the bins containing Barack Obama and Joe Biden. In how many places do they agree?

Compare the result with a former British diplomat, whose bin representation agrees with Obama's in only 8 out of 16 places.

```
print wiki[wiki['name']=='Wynn Normington Hugh-Jones']

print np.array(model['bin_index_bits'][22745], dtype=int) # list of 0/1's
print model['bin_index_bits'][35817] == model['bin_index_bits'][22745]
```

How about the documents in the same bin as Barack Obama? Are they necessarily more similar to Obama than Biden? Let's look at which documents are in the same bin as the Barack Obama article.

```
print model['table'][model['bin_indices'][35817]]
```

There are four other documents that belong to the same bin. Which documents are they?

```
doc_ids = list(model['table'][model['bin_indices'][35817]])
doc_ids.remove(35817) # display documents other than Obama

```

```
docs = wiki.filter_by(values=doc_ids, column_name='id') # filter by id column
print docsIt turns out that Joe Biden is much closer to Barack Obama than any of the four
    documents, even though Biden's bin representation differs from Obama's by 2 bits.
```

It turns out that Joe Biden is much closer to Barack Obama than any of the four documents, even though Biden's bin representation differs from Obama's by 2 bits.

```
def cosine_distance(x, y):
    xy = x.dot(y.T)
    dist = xy/(norm(x)*norm(y))
    return 1-dist[0,0]

obama_tf_idf = corpus[35817,:]
biden_tf_idf = corpus[24478,:]

print '================= Cosine distance from Barack Obama'
print 'Barack Obama - {0:24s}: {1:f}'.format('Joe Biden',
                                             cosine_distance(obama_tf_idf, biden_tf_idf))
for doc_id in doc_ids:
    doc_tf_idf = corpus[doc_id,:]
    print 'Barack Obama - {0:24s}: {1:f}'.format(wiki[doc_id]['name'],
                                                 cosine_distance(obama_tf_idf, doc_tf_idf))
```

Moral of the story. Similar data points will in general *tend* to fall into *nearby* bins, but that's all we can say about LSH. In a high-dimensional space such as text features, we often get unlucky with our selection of only a few random vectors such that dissimilar data points go into the same bin while similar data points fall into different bins. **Given a query document, we must consider all documents in the nearby bins and sort them according to their actual distances from the query.**

Query the LSH model

Let us first implement the logic for searching nearby neighbors, which goes like this:

```
1. Let L be the bit representation of the bin that contains the query documents.
2. Consider all documents in bin L.
3. Consider documents in the bins whose bit representation differs from L by 1 bit.
4. Consider documents in the bins whose bit representation differs from L by 2 bits.
...
```

To obtain candidate bins that differ from the query bin by some number of bits, we use itertools.combinations, which produces all possible subsets of a given list. See this documentation for details.

```
1. Decide on the search radius r. This will determine the number of different bits between the
two vectors.
2. For each subset (n_1, n_2, ..., n_r) of the list [0, 1, 2, ..., num_vector-1], do the
following:
   * Flip the bits (n_1, n_2, ..., n_r) of the query bin to produce a new bit vector.
   * Fetch the list of documents belonging to the bin indexed by the new bit vector.
   * Add those documents to the candidate set.
```

Each line of output from the following cell is a 3-tuple indicating where the candidate bin would differ from the query bin. For instance,

```
(0, 1, 3)
```

indicates that the candidate bin differs from the query bin in first, second, and fourth bits. For illustrations, inspect the output of the following piece of code:

```
from itertools import combinations

num_vector = 16
search_radius = 3

for diff in combinations(range(num_vector), search_radius):
    print diff
```

With this output in mind, implement the logic for nearby bin search:

```
def search_nearby_bins(query_bin_bits, table, search_radius=2, initial_candidates=set()):
    """
    For a given query vector and trained LSH model, return all candidate neighbors for
    the query among all bins within the given search radius.

    Example usage
    -------------
    >>> model = train_lsh(corpus, num_vector=16, seed=143)
    >>> q = model['bin_index_bits'][0]  # vector for the first document

    >>> candidates = search_nearby_bins(q, model['table'])
    """
    num_vector = len(query_bin_bits)
    powers_of_two = 1 << np.arange(num_vector-1, -1, -1)

    # Allow the user to provide an initial set of candidates.
    candidate_set = copy(initial_candidates)

    for different_bits in combinations(range(num_vector), search_radius):
        # Flip the bits (n_1,n_2,...,n_r) of the query bin to produce a new bit vector.
        ## Hint: you can iterate over a tuple like a list
        alternate_bits = copy(query_bin_bits)
        for i in different_bits:
            alternate_bits[i] = ... # YOUR CODE HERE

        # Convert the new bit vector to an integer index
        nearby_bin = alternate_bits.dot(powers_of_two)

        # Fetch the list of documents belonging to the bin indexed by the new bit vector.
        # Then add those documents to candidate_set
        # Make sure that the bin exists in the table!
        # Hint: update() method for sets lets you add an entire list to the set
        if nearby_bin in table:
            ... # YOUR CODE HERE: Update candidate_set with the documents in this bin.

    return candidate_set
```

Checkpoint. Running the function with search_radius=0 should yield the list of documents belonging to the same bin as the query.

```
obama_bin_index = model['bin_index_bits'][35817] # bin index of Barack Obama
candidate_set = search_nearby_bins(obama_bin_index, model['table'], search_radius=0)
if candidate_set == set([35817, 21426, 53937, 39426, 50261]):
    print 'Passed test'
else:
```

```
    print 'Check your code'
print 'List of documents in the same bin as Obama: 35817, 21426, 53937, 39426, 50261'
```

Checkpoint. Running the function with search_radius=1 adds more documents to the fore.

```
candidate_set = search_nearby_bins(obama_bin_index, model['table'], search_radius=1,
    initial_candidates=candidate_set)
if candidate_set == set([39426, 38155, 38412, 28444, 9757, 41631, 39207, 59050, 47773, 53937,
    21426, 34547,
                         23229, 55615, 39877, 27404, 33996, 21715, 50261, 21975, 33243, 58723,
                             35817, 45676,
                         19699, 2804, 20347]):
    print 'Passed test'
else:
    print 'Check your code'
```

Note. Don't be surprised if few of the candidates look similar to Obama. This is why we add as many candidates as our computational budget allows and sort them by their distance to the query.

Now we have a function that can return all the candidates from neighboring bins. Next we write a function to collect all candidates and compute their true distance to the query.

```
def query(vec, model, k, max_search_radius):

    data = model['data']
    table = model['table']
    random_vectors = model['random_vectors']
    num_vector = random_vectors.shape[1]


    # Compute bin index for the query vector, in bit representation.
    bin_index_bits = (vec.dot(random_vectors) >= 0).flatten()

    # Search nearby bins and collect candidates
    candidate_set = set()
    for search_radius in xrange(max_search_radius+1):
        candidate_set = search_nearby_bins(bin_index_bits, table, search_radius,
            initial_candidates=candidate_set)

    # Sort candidates by their true distances from the query
    nearest_neighbors = sframe.SFrame({'id':candidate_set})
    candidates = data[np.array(list(candidate_set)),:]
    nearest_neighbors['distance'] = pairwise_distances(candidates, vec, metric='cosine').flatt
        ()

    return nearest_neighbors.topk('distance', k, reverse=True), len(candidate_set)
```

Let's try it out with Obama:

```
print query(corpus[35817,:], model, k=10, max_search_radius=3)
```

To identify the documents, it's helpful to join this table with the Wikipedia table:

```
result, num_candidates_considered = query(corpus[35817,:], model, k=10, max_search_radius=3)
print result.join(wiki[['id', 'name']], on='id').sort('distance')
```

which produces a table of the form

```
+-------+--------------------+-------------------------+
|   id  |      distance      |           name          |
+-------+--------------------+-------------------------+
| 35817 | -6.66133814775e-16 |       Barack Obama      |
| 24478 |   0.703138676734   |        Joe Biden        |
| 56008 |   0.856848127628   |      Nathan Cullen      |
| 37199 |   0.874668698194   | Barry Sullivan (lawyer) |
| 40353 |   0.890034225981   |      Neil MacBride      |
|  9267 |   0.898377208819   |   Vikramaditya Khanna   |
| 55909 |   0.899340396322   |       Herman Cain       |
|  9165 |   0.900921029925   |   Raymond F. Clevenger  |
| 57958 |   0.903003263483   |    Michael J. Malbin    |
| 49872 |   0.909532800353   |      Lowell Barron      |
+-------+--------------------+-------------------------+
[10 rows x 3 columns]
```

We have shown that we have a working LSH implementation!

Experimenting with your LSH implementation

In the following sections we have implemented a few experiments so that you can gain intuition for how your LSH implementation behaves in different situations. This will help you understand the effect of searching nearby bins and the performance of LSH versus computing nearest neighbors using a brute force search.

Effect of nearby bin search

How does nearby bin search affect the outcome of LSH? There are three variables that are affected by the search radius:

- Number of candidate documents considered
- Query time
- Distance of approximate neighbors from the query

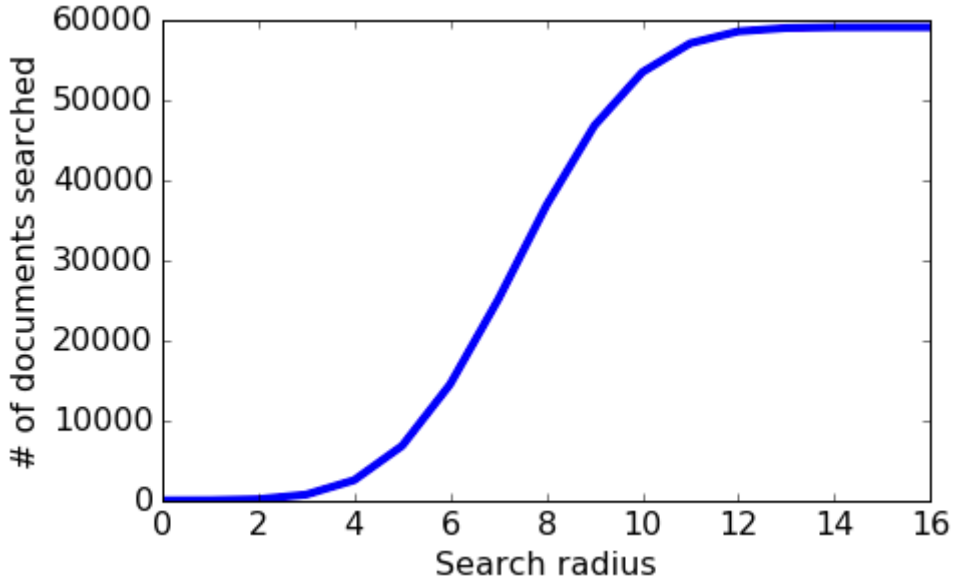
Let us run LSH multiple times, each with different radii for nearby bin search. We will measure the three variables as discussed above.

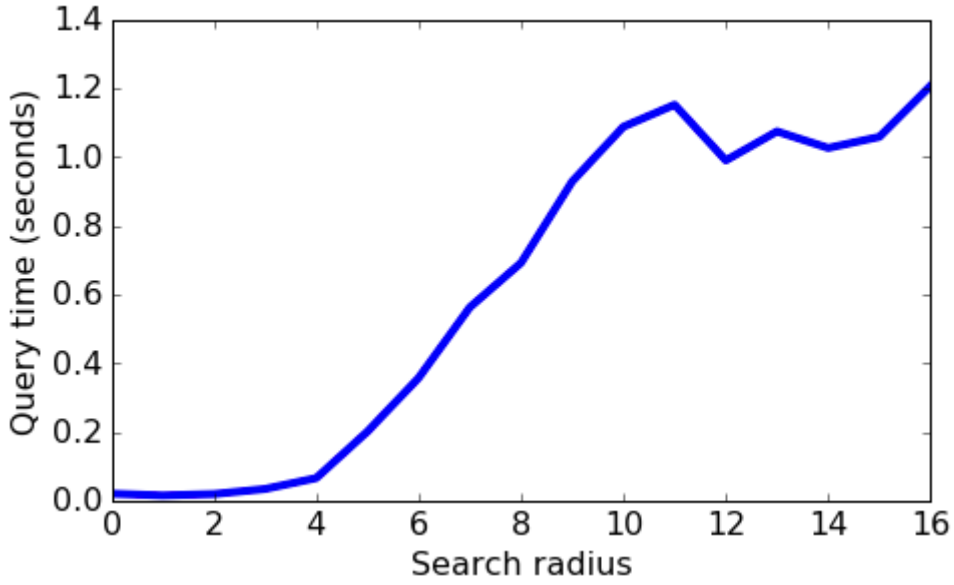
```
num_candidates_history = []
query_time_history = []
max_distance_from_query_history = []
min_distance_from_query_history = []
average_distance_from_query_history = []

for max_search_radius in xrange(17):
    start=time.time()
    # Perform LSH query using Barack Obama, with max_search_radius
    result, num_candidates = query(corpus[35817,:], model, k=10,
                                   max_search_radius=max_search_radius)
```

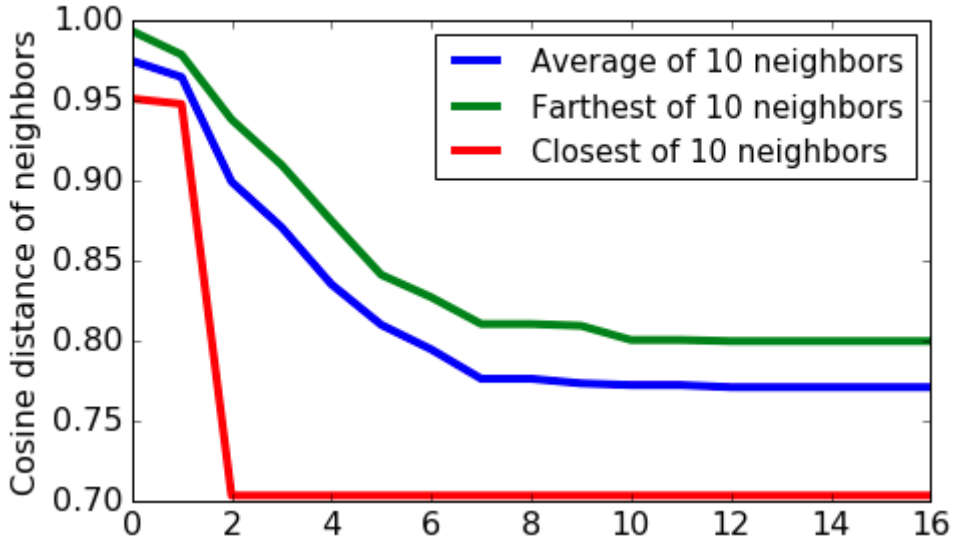
```
    end=time.time()
    query_time = end-start  # Measure time

    print 'Radius:', max_search_radius
    # Display 10 nearest neighbors, along with document ID and name
    print result.join(wiki[['id', 'name']], on='id').sort('distance')

    # Collect statistics on 10 nearest neighbors
    average_distance_from_query = result['distance'][1:].mean()
    max_distance_from_query = result['distance'][1:].max()
    min_distance_from_query = result['distance'][1:].min()

    num_candidates_history.append(num_candidates)
    query_time_history.append(query_time)
    average_distance_from_query_history.append(average_distance_from_query)
    max_distance_from_query_history.append(max_distance_from_query)
```

Notice that the top 10 query results become more relevant as the search radius grows. Let's plot the three variables:

```
plt.figure(figsize=(7,4.5))
plt.plot(num_candidates_history, linewidth=4)
plt.xlabel('Search radius')
plt.ylabel('# of documents searched')
plt.rcParams.update({'font.size':16})
plt.tight_layout()

plt.figure(figsize=(7,4.5))
plt.plot(query_time_history, linewidth=4)
plt.xlabel('Search radius')
plt.ylabel('Query time (seconds)')
plt.rcParams.update({'font.size':16})
plt.tight_layout()

plt.figure(figsize=(7,4.5))
plt.plot(average_distance_from_query_history, linewidth=4, label='Average of 10 neighbors')
plt.plot(max_distance_from_query_history, linewidth=4, label='Farthest of 10 neighbors')
plt.plot(min_distance_from_query_history, linewidth=4, label='Closest of 10 neighbors')
plt.xlabel('Search radius')
plt.ylabel('Cosine distance of neighbors')
plt.legend(loc='best', prop={'size':15})
plt.rcParams.update({'font.size':16})
plt.tight_layout()
```







Some observations:

- As we increase the search radius, we find more neighbors that are a smaller distance away.

- With increased search radius comes a greater number documents that have to be searched. Query time is higher as a consequence.
- With sufficiently high search radius, the results of LSH begin to resemble the results of brute-force search.

Quiz Question. What was the smallest search radius that yielded the correct nearest neighbor, namely Joe Biden?

Quiz Question. Suppose our goal was to produce 10 approximate nearest neighbors whose average distance from the query document is within 0.01 of the average for the true 10 nearest neighbors. For Barack Obama, the true 10 nearest neighbors are on average about 0.77. What was the smallest search radius for Barack Obama that produced an average distance of 0.78 or better?

Quality metrics for neighbors

The above analysis is limited by the fact that it was run with a single query, namely Barack Obama. We should repeat the analysis for the entirety of data. Iterating over all documents would take a long time, so let us randomly choose 10 documents for our analysis.

For each document, we first compute the true 25 nearest neighbors, and then run LSH multiple times. We look at two metrics:

- Precision@10: How many of the 10 neighbors given by LSH are among the true 25 nearest neighbors?
- Average cosine distance of the neighbors from the query

Then we run LSH multiple times with different search radii.

```
def brute_force_query(vec, data, k):
    num_data_points = data.shape[0]

    # Compute distances for ALL data points in training set
    nearest_neighbors = sframe.SFrame({'id':range(num_data_points)})
    nearest_neighbors['distance'] = pairwise_distances(data, vec, metric='cosine').flatten()

    return nearest_neighbors.topk('distance', k, reverse=True)
```

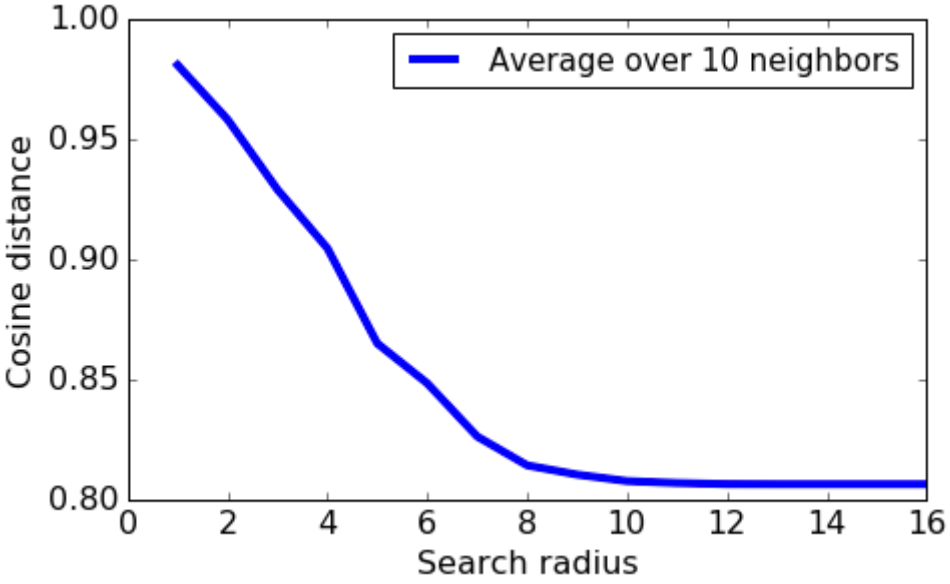
The following cell will run LSH with multiple search radii and compute the quality metrics for each run. Allow a few minutes to complete.

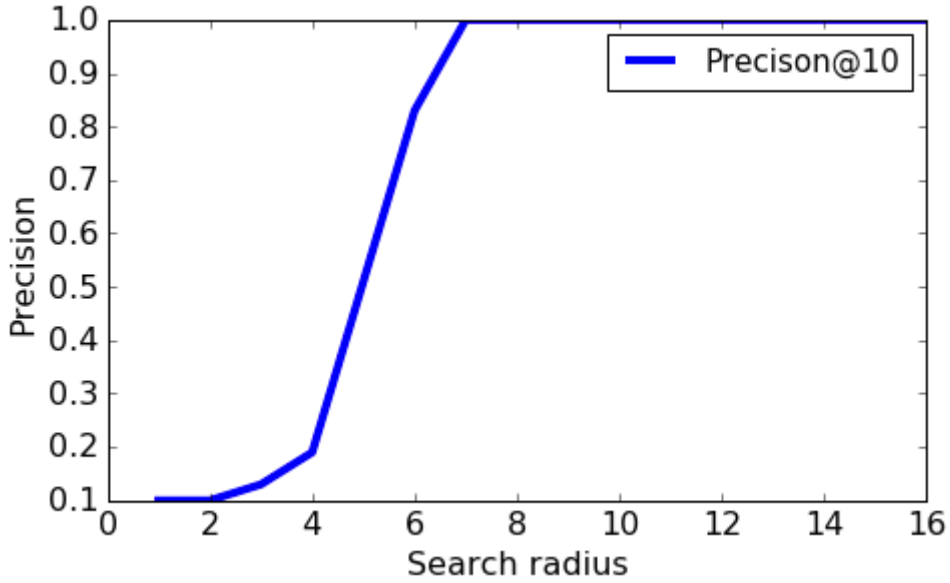
```
max_radius = 17
precision = {i:[] for i in xrange(max_radius)}
average_distance  = {i:[] for i in xrange(max_radius)}
query_time  = {i:[] for i in xrange(max_radius)}

np.random.seed(0)
num_queries = 10
for i, ix in enumerate(np.random.choice(corpus.shape[0], num_queries, replace=False)):
    print('%s / %s' % (i, num_queries))
    ground_truth = set(brute_force_query(corpus[ix,:], corpus, k=25)['id'])
    # Get the set of 25 true nearest neighbors

    for r in xrange(1,max_radius):
```

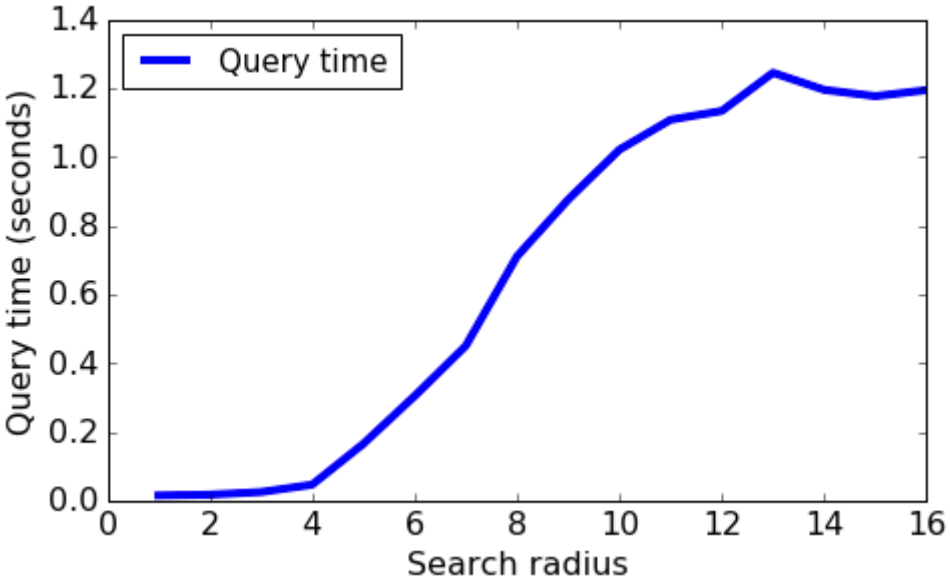
```
        start = time.time()
        result, num_candidates = query(corpus[ix,:], model, k=10, max_search_radius=r)
        end = time.time()

        query_time[r].append(end-start)
        # precision = (# of neighbors both in result and ground_truth)/10.0
        precision[r].append(len(set(result['id']) & ground_truth)/10.0)
```

Let's plot:

```
plt.figure(figsize=(7,4.5))
plt.plot(range(1,17), [np.mean(average_distance[i]) for i in xrange(1,17)], linewidth=4, label='Average over 10 neighbors')
plt.xlabel('Search radius')
plt.ylabel('Cosine distance')
plt.legend(loc='best', prop={'size':15})
plt.rcParams.update({'font.size':16})
plt.tight_layout()

plt.figure(figsize=(7,4.5))
plt.plot(range(1,17), [np.mean(precision[i]) for i in xrange(1,17)], linewidth=4, label='Precison@10')
plt.xlabel('Search radius')
plt.ylabel('Precision')
plt.legend(loc='best', prop={'size':15})
plt.rcParams.update({'font.size':16})
plt.tight_layout()

plt.figure(figsize=(7,4.5))
plt.plot(range(1,17), [np.mean(query_time[i]) for i in xrange(1,17)], linewidth=4, label='Query time')
plt.xlabel('Search radius')
plt.ylabel('Query time (seconds)')
plt.legend(loc='best', prop={'size':15})
plt.rcParams.update({'font.size':16})
plt.tight_layout()
```







The observations for Barack Obama generalize to the entire dataset.

Effect of number of random vectors

Let us now turn our focus to the remaining parameter: the number of random vectors. We run LSH with different number of random vectors, ranging from 5 to 20. We fix the search radius to 3.

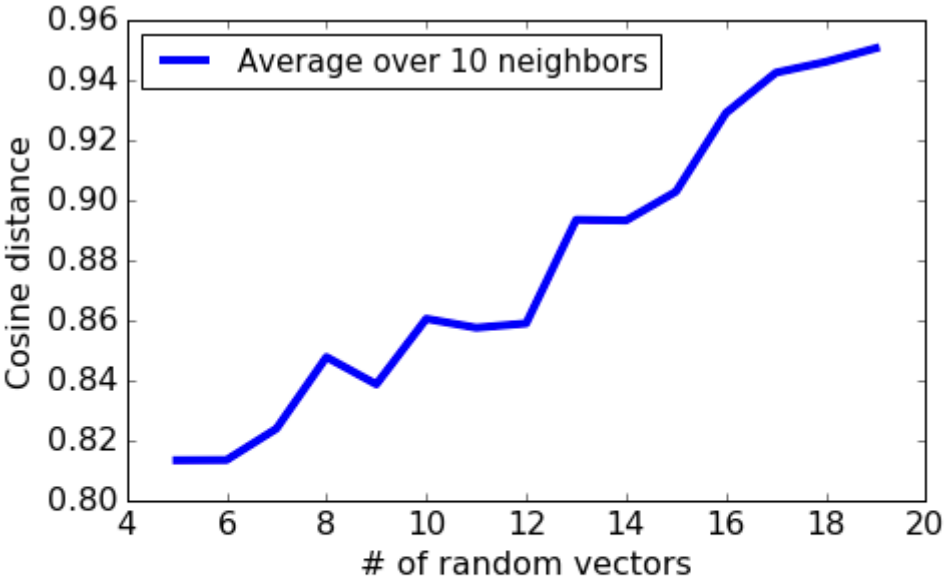
Allow a few minutes for the following cell to complete.

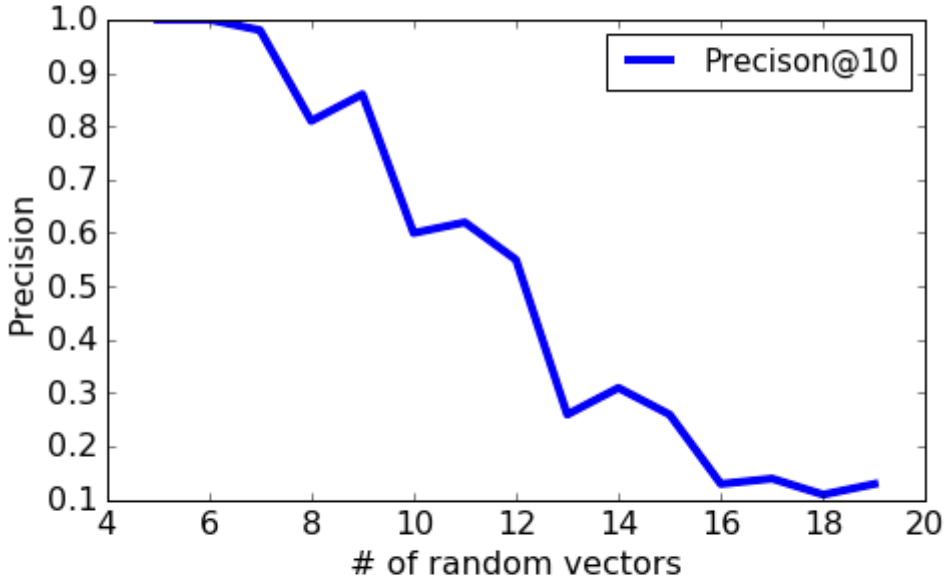
```
precision = {i:[] for i in xrange(5,20)}
average_distance  = {i:[] for i in xrange(5,20)}
query_time = {i:[] for i in xrange(5,20)}
num_candidates_history = {i:[] for i in xrange(5,20)}
ground_truth = {}

np.random.seed(0)
num_queries = 10
docs = np.random.choice(corpus.shape[0], num_queries, replace=False)

for i, ix in enumerate(docs):
    ground_truth[ix] = set(brute_force_query(corpus[ix,:], corpus, k=25)['id'])
    # Get the set of 25 true nearest neighbors

for num_vector in xrange(5,20):
    print('num_vector = %s' % (num_vector))
    model = train_lsh(corpus, num_vector, seed=143)

    for i, ix in enumerate(docs):
        start = time.time()
        result, num_candidates = query(corpus[ix,:], model, k=10, max_search_radius=3)
        end = time.time()

        query_time[num_vector].append(end-start)
        precision[num_vector].append(len(set(result['id']) & ground_truth[ix])/10.0)
        average_distance[num_vector].append(result['distance'][1:].mean())
        num_candidates_history[num_vector].append(num_candidates)
```

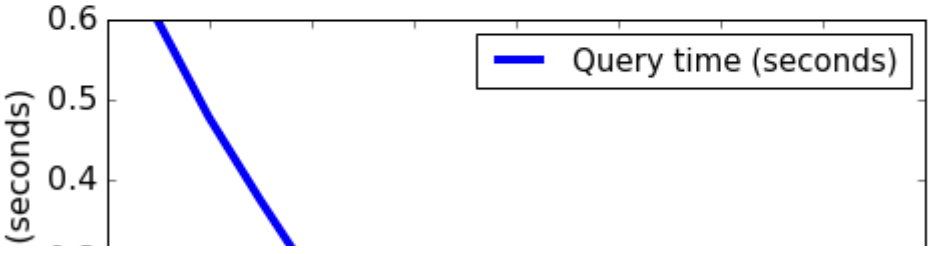
Plot the metrics as a function of the number of random rectors using the following piece of code :

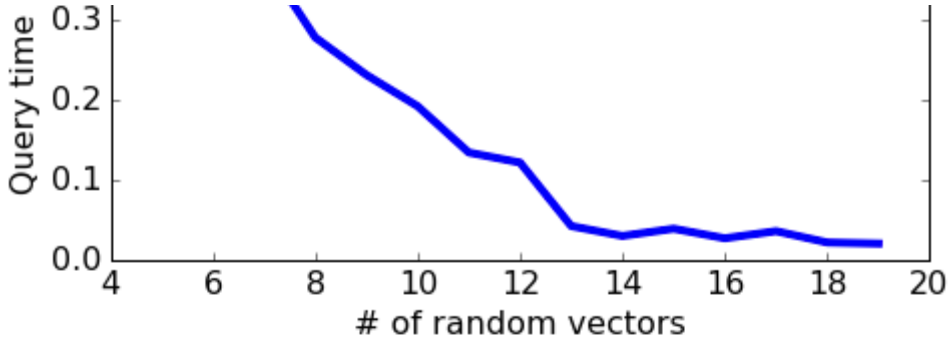
```
plt.figure(figsize=(7,4.5))
plt.plot(range(5,20), [np.mean(average_distance[i]) for i in xrange(5,20)], linewidth=4, label
    ='Average over 10 neighbors')
plt.xlabel('# of random vectors')
plt.ylabel('Cosine distance')
plt.legend(loc='best', prop={'size':15})
plt.rcParams.update({'font.size':16})
plt.tight_layout()

plt.figure(figsize=(7,4.5))
plt.plot(range(5,20), [np.mean(precision[i]) for i in xrange(5,20)], linewidth=4, label
    ='Precison@10')
plt.xlabel('# of random vectors')
plt.ylabel('Precision')
plt.legend(loc='best', prop={'size':15})
plt.rcParams.update({'font.size':16})
plt.tight_layout()

plt.figure(figsize=(7,4.5))
plt.plot(range(5,20), [np.mean(query_time[i]) for i in xrange(5,20)], linewidth=4, label='Quer
    time (seconds)')
plt.xlabel('# of random vectors')
plt.ylabel('Query time (seconds)')
```

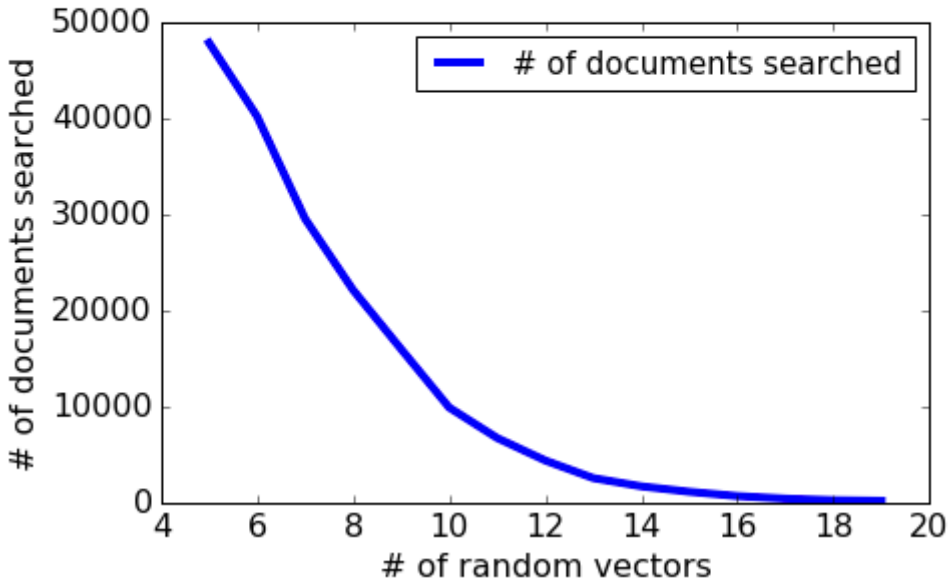
```
plt.legend(loc='best', prop={'size':15})
plt.rcParams.update({'font.size':16})
plt.tight_layout()

plt.figure(figsize=(7,4.5))
plt.plot(range(5,20), [np.mean(num_candidates_history[i]) for i in xrange(5,20)], linewidth=4,
         label='# of documents searched')
plt.xlabel('# of random vectors')
plt.ylabel('# of documents searched')
plt.legend(loc='best', prop={'size':15})
plt.rcParams.update({'font.size':16})
plt.tight_layout()
```











We see a similar trade-off between quality and performance: as the number of random vectors increases, the query time goes down as each bin contains fewer documents on average, but on average the neighbors are likewise placed farther from the query. On the other hand, when using a small enough number of random vectors, LSH becomes very similar brute-force search: Many documents appear in a single bin, so searching the query bin alone covers a lot of the corpus; then, including neighboring bins might result in searching all documents, just as in the brute-force approach.