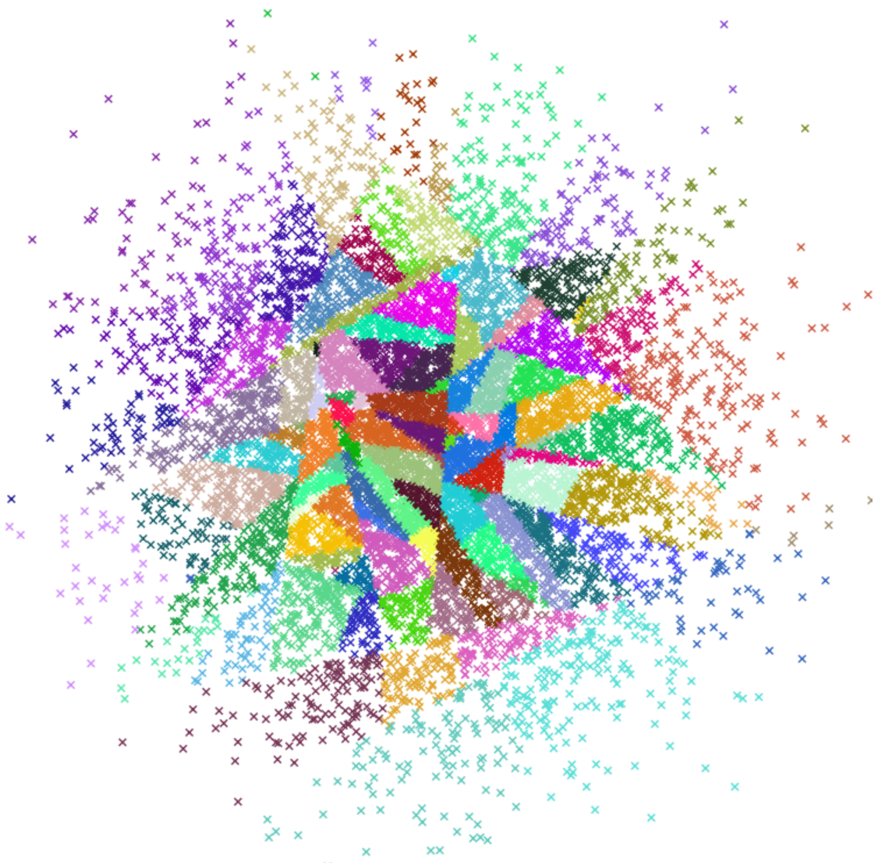
Simple Approximate Nearest Neighbors in Python with Annoy and lmdb

Kevin Yang Follow
Jan 4, 2018 · 4 min read



Colorful points in space

Recently, I've been playing around with adding and subtracting word embeddings learned from GloVe. For example, we can take the embedding for the word "king" subtract the embedding for "man" and then add the embedding for "woman" and end up with a resulting embedding (vector). Next, if we have a corpus of these word-embedding pairs we can search through it to find the most similar embedding and retrieve the corresponding word. If we did this for the query above we would get:

```
King + (Woman - Man) = Queen
```

There are many ways to search through this corpus of word-embedding pairs for the nearest neighbors to a query. One way to guarantee that you find the optimal vector is to iterate through your corpus and compare how similar each pair is to the query. This however is incredibly time consuming and not often used. A better exact technique would be to use a vectorized cosine distance shown below:

```
vectors = np.array(embeddingmodel.embeddings)
ranks = np.dot(query,vectors.T)/np.sqrt(np.sum(vectors*
mostSimilar = []
[mostSimilar.append(idx) for idx in ranks.argsort()[::-
```

For more info on cosine distance check out this great resource: Cosine Similarity

Vectorized cosine distance is notably faster than the iterative method but still may be too slow. This is where approximate nearest neighbors shines: returning approximate results but blazingly quickly. Many times you don't need exact optimal results, for example: what even is the exact similar word for "Queen?" In these cases where you need good enough results quickly, you should use approximate nearest neighbors.

In this article we will be writing a simple python script to quickly find approximate nearest neighbors. We will use the Python library Annoy and lmdb. For my corpus, I will be using word-embedding pairs, but the instructions below work with any type of embeddings: such as embeddings of songs to create a recommendation engine for similar songs or even photo embeddings to enable reverse image search.

Making an Annoy Index

Lets create a python script called: "make_annoy_index." To start off lets include the dependencies that we will use:

```
'''
Usage: python2 make_annoy_index.py \
    --embeddings=<embedding path> \
    --num_trees=<int> \
    --verbose

Generate an Annoy index and lmdb map given an embeddin

Embedding file can be
  1. A .bin file that is compatible with word2vec bina
     There are pre-trained vectors to download at http
  2. A .gz file with the GloVe format (item then a lis
  3. A plain text file with the same format as above

'''
```

The last line we have is an import from “vector_utils”. We will be writing “vector_utils” later on so don’t worry now!

Next, on to the meat of our script: the “create_index” function. Here we will be generating our lmdb map and our Annoy index.

1. First we find the length of our embedding which is used to instantiate an Annoy index.
2. Next we instantiate an lmdb map with the line: “env = lmdb.open(fn_lmdb, map_size=int(1e9)).”
3. Make sure we don’t have an Annoy index or lmdb map in the current path
4. Add every key and vector from our embeddings file to both our lmdb map and our Annoy index
5. Build and save our Annoy index

```
'''
function create_index(fn, num_trees=30, verbose=False)
-------------------------------
Creates an Annoy index and lmdb map given an embedding

Input:
    fn              - filename of the embedding file
    num_trees       - number of trees to build Annoy i
    verbose         - log status

Return:
    Void
'''
def create_index(fn, num_trees=30, verbose=False):
    fn_annoy = fn + '.annoy'
    fn_lmdb = fn + '.lmdb' # stores word <-> id mappin

    word, vec = get_vectors(fn).next()
    size = len(vec)
    if verbose:
        print("Vector size: {}".format(size))

    env = lmdb.open(fn_lmdb, map_size=int(1e9))
    if not os.path.exists(fn_annoy) or not os.path.exi
        i = 0
        a = annoy.AnnoyIndex(size)
        with env.begin(write=True) as txn:
            for word, vec in get_vectors(fn):
                a.add_item(i, vec)
                id = 'i%d' % i
                word = 'w' + word
```

I've included "argparse" so we can call our script from the command line:

```
'''
private function _create_args()
-------------------------------
Creates an argeparse object for CLI for create_index()

Input:
    Void

Return:
    args object with required arguments for threshold_

'''
def _create_args():
    parser = argparse.ArgumentParser()
```

Add a main function to call our script and we are done with "make_annoy_index.py":

```
if __name__ == '__main__':
    args = _create_args()
    create_index(args.embeddings, num_trees=args.num_tr
```

Now we can just call our new script from the command line to generate an Annoy index and a corresponding lmdb map!

```
python2 make_annoy_index.py \
    --embeddings=<embedding path> \
    --num_trees=<int> \
    --verbose
```

Writing Vector Utils

In "make_annoy_index.py" we included a python script "vector_utils." We will write this script now. "Vector_utils" is used to help read in vectors from .txt, .bin, and .pkl files.

Writing this script is not that relevant to what we are doing so I've just included the whole script below:

```
'''
Vector Utils

Utils to read in vectors from txt, .bin, or .pkl.

Taken from Erik Bernhardsson
Source: https://github.com/erikbern/ann-presentation/b
'''
import gzip
import struct
import cPickle

def _get_vectors(fn):
    if fn.endswith('.gz'):
        f = gzip.open(fn)
        fn = fn[:-3]

    else:
        f = open(fn)

    if fn.endswith('.bin'): # word2vec format
        words, size = (int(x) for x in f.readline().st

        t = 'f' * size

        while True:
            pos = f.tell()
            buf = f.read(1024)
            if buf == '' or buf == '\n': return
            i = buf.index(' ')
            word = buf[:i]
            f.seek(pos + i + 1)

            vec = struct.unpack(t, f.read(4 * size))

            yield word.lower(), vec

```

Testing our Index and lmdb Map

Now that we have generated an Annoy index and lmdb map lets write a script to use them to do inference.

Name our file "annoy_inference.py" and include the dependencies:

```
'''
Usage: python2 annoy_inference.py \
    --token='hello' \
    --num_results=<int> \
    --verbose

Query an Annoy index to find approximate nearest neigh

'''
```

Now we need to load in our Annoy index and our lmdb map; we will load them globally for easy access. Note that for me "VEC_LENGTH" is 50. Make sure that your "VEC_LENGTH" matches the length of your embedding, otherwise Annoy will not be happy.

```
VEC_LENGTH = 50
FN_ANNOY = 'glove.6B.50d.txt.annoy'
FN_LMDB = 'glove.6B.50d.txt.lmdb'

a = annoy.AnnoyIndex(VEC_LENGTH)
a.load(FN_ANNOY)
```

The fun part, the "calculate" function.

1. Get the index for our query from our lmdb map

2. Get the corresponding vector from Annoy with "get_item_vector(id)"

3. Get the nearest neighbors from Annoy with "a.get_nns_by_vector(v, num_results)"

```
'''
private function calculate(query, num_results)
-------------------------------
Queries a given Annoy index and lmdb map for num_resul

Input:
    query           - query to be searched
    num_results     - the number of results

Return:
    ret_keys        - list of num_results nearest neig

'''
def calculate(query, num_results, verbose=False):
    ret_keys = []
    with env.begin() as txn:
        id = int(txn.get('w' + query)[1:])
```

Again, here is an "argparse" object to make reading command line arguments easier

```
'''
private function _create_args()
-------------------------------
Creates an argeparse object for CLI for calculate() fu

Input:
    Void

Return:
    args object with required arguments for threshold_

'''
def _create_args():
    parser = argparse.ArgumentParser()
```

And, a main function to call "annoy_inference.py" from the command line:

```
if __name__ == '__main__':
    args = _create_args()
    print(calculate(args.token, args.num_results, args.
```

Great! Now we can use our Annoy index and lmdb map to get the nearest neighbors to a query!

```
python2 annoy_inference.py --token="test" --num_results=30

['test', 'tests', 'determine', 'for', 'crucial', 'only',
'preparation', 'needed', 'positive', 'guided', 'time',
'performance', 'one', 'fitness', 'replacement', 'stages',
'made', 'both', 'accuracy', 'deliver', 'put',
'standardized', 'best', 'discovery', '.', 'a', 'diagnostic',
'delayed', 'while', 'side']
```

Code

You can clone the corresponding repo to get all the code in this tutorial: https://github.com/kyang6/annoy_tutorial

Conclusion

Boy, that was a lot of code. But, now we have scripts to take any embedding data and generate an Annoy index and lmdb map to find approximate nearest neighbors!

If you have questions or concerns please email me! kyang6@cs.stanford.edu