DAT470/DIT065 Assignment 3

Xinyan Liu
liuxinya@chalmers.se

Leah Wanja Ndirangu
leahw@chalmers.se

2025-05-11

Problem 1

As a preprocessing step, you should normalize all your data to have unit length before doing anything else. When you are using NumPy operations correctly (array operations), you should get a degree of parallelization for free because the underlying libraries can use multicore processing; this requires that you only apply array operations and try not to access individual elements yourself. Normalize all of the data vectors to have unit-length. Make sure to use array operations. Use lsh normalize.py as your starting point for the interface. In your report, record how many seconds it took to normalize the larger dataset (glove.840B.300d.txt).

Below is the implementation of the `map` function in Python:

```
#!/usr/bin/env python3

import numpy as np
import pandas as pd
import csv
import argparse
import time

def load_glove(filename):
    """
    Loads the glove dataset. Returns three things:
    A dictionary that contains a map from words to rows in
        the dataset.
    A reverse dictionary that maps rows to words.
    The embeddings dataset as a NumPy array.
    """
    df = pd.read_table(filename, sep=' ', index_col=0,
        header=None,
                              quoting=csv.QUOTE_NONE)
    word_to_idx = dict()
    idx_to_word = dict()
    for (i,word) in enumerate(df.index):
```

```python
        word_to_idx[word] = i
        idx_to_word[i] = word
    return (word_to_idx, idx_to_word, df.to_numpy())

def normalize(X):
    """
    Reads an n*d matrix and normalizes all rows to have unit
        -length (L2 norm)

    Implement this function using array operations! No loops
         allowed.
    """
    data = np.array(X)
    data_normalized = (data-data.min())/(data.max()-data.min
        ())
    return data_normalized


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('dataset', help='Glove dataset
        filename',
                        type=str)
    args = parser.parse_args()
    (word_to_idx, idx_to_word, X) = load_glove(args.dataset)

    start = time.time()

    X = normalize(X)

    end = time.time()
    normalize_time =end-start

    print(f"Time to Normalized time {normalize_time}")
```

Time to Normalized time 46.39153838157654s

Problem 2

The file queries.txt contains a list of 10 words. For each word, determine the 3 closest words (not including the word itself), in terms of cosine similarity. Remembering that the matrix product AB has the interpretation that (AB)ij is the same as the dot product between the ith row vector of A, and the jth column vector of B, construct a 10×d matrix Q that contains the query vectors, and compute QXT where X is the n × d data matrix. Use lsh matmul.py as your starting point for the interface. In your report, report the 3 closest words for each word as a nicely formatted table, and also report the time it took to compute the matrix product, the time it took to sort the results, and the combined time, for the larger dataset (glove.840B.300d.txt).

Here are the code

```
#!/usr/bin/env python3

import numpy as np
import pandas as pd
import csv
import argparse
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split

import time

def load_glove(filename):
    """
    Loads the glove dataset. Returns three things:
    A dictionary that contains a map from words to rows in
        the dataset.
    A reverse dictionary that maps rows to words.
    The embeddings dataset as a NumPy array.
    """
    df = pd.read_table(filename, sep=' ', index_col=0,
        header=None,
                            quoting=csv.QUOTE_NONE)
    word_to_idx = dict()
    idx_to_word = dict()
    for (i,word) in enumerate(df.index):
        word_to_idx[word] = i
        idx_to_word[i] = word
    return (word_to_idx, idx_to_word, df.to_numpy())

def normalize(X):
    """
    Reads an n*d matrix and normalizes all rows to have unit
        -length (L2 norm)
```

```
    Implement this function using array operations! No loops
        allowed.
    """
    data = np.array(X)
    data_normalized = (data-data.min())/(data.max()-data.min
        ())
    column_size = len(data_normalized[0])
    row_size = len(data_normalized)
    # print(f"The rows are {row_size} and The column value
        for X is {column_size}")
    print(f"data_normalized is {data_normalized}")
    return data_normalized

def construct_queries(queries_fn, word_to_idx, X):
    """
    Reads queries (one string per line) and returns:
    - The query vectors as a matrix Q (one query per row)
    - Query labels as a list of strings
    """
    with open(queries_fn, 'r') as f:
        queries = f.read().splitlines()
    Q = np.zeros((len(queries), X.shape[1]))
    print(f"This is Q :{Q} before assigning X values")
    for i in range(len(queries)):
        Q[i,:] = X[word_to_idx[queries[i]],:]
    print(f"Q is of length:{len(Q)} and column size is {len(
        Q[0])}")
    return (Q,queries)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('dataset', help='Glove dataset
        filename',
                            type=str)
    parser.add_argument('queries', help='Queries filename',
        type=str)
    args = parser.parse_args()

    (word_to_idx, idx_to_word, X) = load_glove(args.dataset)


    X = normalize(X)

    (Q,queries) = construct_queries(args.queries,
        word_to_idx, X)

    t1 = time.time()
    dot_product = np.dot(Q,X.transpose())
    t2 = time.time()
    magnitude_X = np.linalg.norm(X)
    magnitude_Q = np.linalg.norm(Q)
    cosine_similarity = dot_product / (magnitude_Q *
        magnitude_X)
```

```
    print(f"cosine_similarity is {cosine_similarity}")
    print('matrix multiplication took', t2-t1)


    # Compute here I such that I[i,:] contains the indices
        of the nearest
    # neighbors of the word i in ascending order.
    # Naturally, I[i,-1] should then be the index of the
        word itself.
    # raise NotImplementedError()
    I = np.argsort(cosine_similarity,axis=1)[:,::-1]
    t3 = time.time()

    for i in range(I.shape[0]):
        neighbors = [idx_to_word[i] for i in I[i,-2:-5:-1]]
        print(f'{queries[i]}: {" ".join(neighbors)}')

    print('matrix multiplication took', t2-t1)
    print('sorting took', t3-t2)
    print('total time', t3-t1)
```

The table 1 below reports the 3 closest words for each word as a nicely formatted table

Word	3 Closest Words
priest	Wooffer, workforce, plan
fork	1999/468/EC, challenges, debris
horse	challenges, workforce, 1999/468/EC
beef	debris, 1999/468/EC, handMore
daoist	debris, Wooffer, handMore
polish	debris, 1999/468/EC, handMore
vehicle	debris, Wooffer, jacuzzi/whirlpool
crepe	plan, debris, challenges
daytime	challenges, plan, Wooffer
scotland	debris, handMore, 1999/468/EC

Table 1: Three closest words for each term based on co-occurrence in the text

The time it took to compute the matrix product: 0.16595125198364258
The time it took to sort the results: 1.178757905960083
The combined time, for the larger dataset (glove.840B.300d.txt): 1.3447091579437256

Problem 3

In your report, include the amount of time it took to transform the bigger dataset (860B) using D = 50 hyperplanes

Here is our code

```
#!/usr/bin/env python3

import numpy as np
```

```python
import pandas as pd
import csv
import argparse
import time

def load_glove(filename):
    """
    Loads the glove dataset. Returns three things:
    A dictionary that contains a map from words to rows in
        the dataset.
    A reverse dictionary that maps rows to words.
    The embeddings dataset as a NumPy array.
    """
    df = pd.read_table(filename, sep=' ', index_col=0,
        header=None,
                                quoting=csv.QUOTE_NONE)
    word_to_idx = dict()
    idx_to_word = dict()
    for (i,word) in enumerate(df.index):
        word_to_idx[word] = i
        idx_to_word[i] = word
    return (word_to_idx, idx_to_word, df.to_numpy())

def normalize(X):
    """
    Reads an n*d matrix and normalizes all rows to have unit
        -length (L2 norm)

    Implement this function using array operations! No loops
        allowed.
    """
    l2norms = np.linalg.norm(X,axis=1)
    l2norm= l2norms[:,np.newaxis]

    return X / l2norm

def construct_queries(queries_fn, word_to_idx, X):
    """
    Reads queries (one string per line) and returns:
    - The query vectors as a matrix Q (one query per row)
    - Query labels as a list of strings
    """
    with open(queries_fn, 'r') as f:
        queries = f.read().splitlines()
    Q = np.zeros((len(queries), X.shape[1]))
    for i in range(len(queries)):
        Q[i,:] = X[word_to_idx[queries[i]],:]
    return (Q,queries)

class RandomHyperplanes:
    """
    This class mimics the interface of sklearn:
    - the constructor sets the number of hyperplanes
    - the random hyperplanes are drawn when fit() is called
```

```
      (input dimension is set)
    - transform actually transforms the vectors
    - fit_transform does fit first, followed by transform
    """
    def __init__(self, D, seed = None)->None:
        """
        Sets the number of hyperplanes (D) and the optional
            random number seed
        """
        self._D = D
        self._seed = seed

    def fit(self, X):
        """
        Draws _D random hyperplanes, that is, by drawing _D
            Gaussian unit
        vectors of length determined by the second dimension
             (number of
        columns) of X
        """
        rng = np.random.default_rng(self._seed)
        hyperplanes = []
        for _ in range(self._D):
            vector = rng.normal(size=X.shape[1])
            hyperplanes.append(vector)

        self.R = normalize(np.array(hyperplanes))

    def transform(self, X):
        """
        Project the rows of X into binary vectors
        """
        projections = np.matmul(X, self.R.transpose())
        binary_result = np.zeros_like(projections, dtype=int
            )
        binary_result[projections > 0] = 1
        return binary_result

    def fit_transform(self, X):
        """
        Calls fit() followed by transform()
        """
        self.fit(X)
        return self.transform(X)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-D', help='Random hyperplanes
        dimension', type=int,
                              required = True)
    parser.add_argument('dataset', help='Glove dataset
        filename',
                              type=str)
    parser.add_argument('queries', help='Queries filename',
```

```
        type=str)
    args = parser.parse_args()

    (word_to_idx, idx_to_word, X) = load_glove(args.dataset)


    X = normalize(X)

    (Q,queries) = construct_queries(args.queries,
        word_to_idx, X)

    start = time.time()

    rh = RandomHyperplanes(args.D, 1234)
    X2 = rh.fit_transform(X)
    Q2 = rh.transform(Q)

    end = time.time()

    print(end-start)
```

The amount of time it took to transform the bigger dataset (860B) using D = 50 hyperplanes: 1.9656956195831299

Problem 4

Try your implementation on the larger dataset (glove.840B.300d.txt) with the following parameters: D = 50, k = 20, L = 10. In your report, report the time it took to fit the data and the time it took to perform the 10 queries.

Time to fit took: 39.16793608665466
Time to perform query took: 31.500043869018555
The following table 2 shows queries: Here is our code for this problem

Word	3 Closest Words
priest	rememberest, mediumAdd, toysrusinc.com
fork	mediumAdd, toysrusinc.com, rememberest
horse	rememberest, toysrusinc.com, copyright.If
beef	toysrusinc.com, rememberest, mediumAdd
daoist	correctly.Not, rememberest, mediumAdd
polish	toysrusinc.com, mediumAdd, rememberest
vehicle	mediumAdd, rememberest, toysrusinc.com
crepe	copyright.If, correctly.Not, rememberest
daytime	mediumAdd, copyright.If, rememberest
scotland	copyright.If, rememberest, mediumAdd

Table 2: Three closest words for each term based on co-occurrence in the text

```
#!/usr/bin/env python3
```

```python
import numpy as np
import numpy.typing as npt
import pandas as pd
import csv
import argparse
import time
from operator import itemgetter

def load_glove(filename):
    """
    Loads the glove dataset. Returns three things:
    A dictionary that contains a map from words to rows in
        the dataset.
    A reverse dictionary that maps rows to words.
    The embeddings dataset as a NumPy array.
    """
    df = pd.read_table(filename, sep=' ', index_col=0,
        header=None,
                                 quoting=csv.QUOTE_NONE)
    word_to_idx = dict()
    idx_to_word = dict()
    for (i,word) in enumerate(df.index):
        word_to_idx[word] = i
        idx_to_word[i] = word
    return (word_to_idx, idx_to_word, df.to_numpy())

def normalize(X):
    """
    Reads an n*d matrix and normalizes all rows to have unit
        -length (L2 norm)

    Implement this function using array operations! No loops
        allowed.
    """
    data = np.array(X)
    data_normalized = (data-data.min())/(data.max()-data.min
        ())
    return data_normalized

def construct_queries(queries_fn, word_to_idx, X):
    """
    Reads queries (one string per line) and returns:
    - The query vectors as a matrix Q (one query per row)
    - Query labels as a list of strings
    """
    with open(queries_fn, 'r') as f:
        queries = f.read().splitlines()
    Q = np.zeros((len(queries), X.shape[1]))
    for i in range(len(queries)):
        Q[i,:] = X[word_to_idx[queries[i]],:]
    return (Q,queries)

class RandomHyperplanes:
    """
```

```
    This class mimics the interface of sklearn:
    - the constructor sets the number of hyperplanes
    - the random hyperplanes are drawn when fit() is called
      (input dimension is set)
    - transform actually transforms the vectors
    - fit_transform does fit first, followed by transform
    """
    def __init__(self, D, seed = None):
        """
        Sets the number of hyperplanes (D) and the optional
            random number seed
        """
        self._D = D
        self._seed = seed

    def fit(self, X):
        """
        Draws _D random hyperplanes, that is, by drawing _D
            Gaussian unit
        vectors of length determined by the second dimension
             (number of
        columns) of X
        """
        rng = np.random.default_rng(self._seed)
        hyperplanes = []
        for _ in range(self._D):
            vector = rng.normal(size=X.shape[1])
            hyperplanes.append(vector)

        self.R = normalize(np.array(hyperplanes))

    def transform(self, X):
        """
        Project the rows of X into binary vectors
        """
        projections = np.matmul(X, self.R.transpose())
        binary_result = np.zeros_like(projections, dtype=int
            )
        binary_result[projections > 0] = 1
        return binary_result

    def fit_transform(self, X):
        """
        Calls fit() followed by transform()
        """
        self.fit(X)
        return self.transform(X)

class LocalitySensitiveHashing:
    """
    Performs locality-sensitive hashing by projecting unit
        vectors to binary vectors
    """
```

```
    # intended members
    # _D: int number of random hyperplanes
    # _k: int hash function length
    # _L: int number of hash functions (tables)
    # _hash_functions numpy integer array, the actual hash
        functions
    # _random_hyperplanes: RandomHyperplanes random
        hyperplanes object
    # _H: list of dicts from binary vectors to sets of
        integers, hash tables
    # _X: numpy array, the original data

    def __init__(self, D, k, L, seed = None):
        """
        Sets the parameters
        - D internal dimensionality (used with random
            hyperplanes)
        - k length of hash functions (how many elementary
            hash functions
          to concatenate)
        - L number of hash tables
        - seed random number generator seed (used for
            intializing random
          hyperplanes; also used to seed the random number
              generator
          for drawing the hash functions)
        """
        self._D = D
        self._k = k
        self._L = L
        rng = np.random.default_rng(seed)

        # draw the hash functions here
        # (essentially, draw a random matrix of shape L*k
            with values in
        # 0,1,...,D-1)
        self._hash_function = rng.integers(0,D, size=(L,k),
            endpoint=False)
        # also initialize the random hyperplanes
        self._random_hyperplanes= RandomHyperplanes(D,seed=
            seed)
        #initializing hash tables of L
        self._H = [{}for _ in range(L)]
        #original dataset
        self._X = None

    def fit(self, X: npt.NDArray[np.float64])->None:
        """
        Fit random hyperplanes
        Then project the dataset into binary vectors
        Then hash the dataset L times into the L hash tables
        """
        self._X = X
        binary_vector = self._random_hyperplanes.
```

```
            fit_transform(X)
        for l in range(self._L):
            hash_table = self._H[l]
            hash_function = self._hash_function[l]
            for idx,bvec in enumerate(binary_vector):
                key_ = tuple(bvec[hash_function])
                if key_ not in hash_table:
                    hash_table[key_] =set()
                hash_table[key_].add(idx)



    def query(self, q: npt.NDArray[np.float64])->npt.NDArray
        [np.int64]:
        """
        Queries one vector
        Returns the *indices* of the nearest neighbors in
            descending order
        That is, if the returned array is I, then X[I[0]] is
             the nearest
        neighbor (if the vector was member of the dataset,
            then typically
        this would be itself), X[I[1]] the second nearest
            etc.
        """

        # Project the query into a binary vector
        # Then hash it L times
        q_binary = self._random_hyperplanes.transform(q.
            reshape(1,-1))[0]
        # Collect all indices from the hash buckets
        candidates = set()
        for l in range(self._L):
            hash_function = self._hash_function[l]
            key_ = tuple(q_binary[hash_function])
            bucket = self._H[l].get(key_,set())
            candidates.update(bucket)
        if not candidates:
            return np.array([],dtype=np.int64)
        # Then compute the dot products with those vectors
        candidate_list = list(candidates)
        candidate_vectors = self._X[candidate_list]
        similarities = candidate_vectors @ q
        # Finally sort results in *descending* order and
            return the indices
        sorted_indices = np.argsort(-similarities)
        sorted_candidate_indices = np.array(candidate_list)[
            sorted_indices]

        return sorted_candidate_indices


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
```

```
parser.add_argument('-D', help='Random hyperplanes
    dimension', type=int,
                        required = True)
parser.add_argument('-k', help='Hash function length',
    type=int,
                        required = True)
parser.add_argument('-L', help='Number of hash tables (
    functions)', type=int,
                        required = True)
parser.add_argument('dataset', help='Glove dataset
    filename',
                        type=str)
parser.add_argument('queries', help='Queries filename',
    type=str)
args = parser.parse_args()

(word_to_idx, idx_to_word, X) = load_glove(args.dataset)


X = normalize(X)

(Q,queries) = construct_queries(args.queries,
    word_to_idx, X)

t1 = time.time()
lsh = LocalitySensitiveHashing(args.D, args.k, args.L,
    1234)

t2 = time.time()
lsh.fit(X)

t3 = time.time()
neighbors = list()
for i in range(Q.shape[0]):
    q = Q[i,:]
    I = lsh.query(q)
    neighbors.append([idx_to_word[i] for i in I][1:4])
t4 = time.time()

print('init took',t2-t1)
print('fit took', t3-t2)
print('query took', t4-t3)
print('total',t4-t1)

for i in range(Q.shape[0]):
    print(f'{queries[i]}: {" ".join(neighbors[i])}')
```

Hyperparameter search

Distance	init time	fit	query	total time
50	0.00035	39.6540	32.6222	72.2767
60	0.00035	40.3697	33.366	73.727
70	0.000874	40.722	33.7105	74.4334
80	0.000304	40.9717	33.8151	74.7871
100	0.000316	41.8206	34.0751	75.896

Table 3: Comparing time taken among different distances

Word	3 Closest Words
priest:	rememberest mediumAdd toysrusinc.com
fork:	mediumAdd toysrusinc.com rememberest
horse:	rememberest toysrusinc.com copyright.If
beef:	toysrusinc.com rememberest mediumAdd
daoist:	correctly.Not rememberest mediumAdd
polish:	toysrusinc.com mediumAdd rememberest
vehicle:	mediumAdd rememberest toysrusinc.com
crepe:	copyright.If correctly.Not rememberest
daytime:	mediumAdd copyright.If rememberest
scotland:	copyright.If rememberest mediumAdd

Table 4: Data out from hashing

Distance	init time	fit	query	total time
50	0.00026	121.895	79.49	201.3871
60	0.0002792	115.87	39.70	155.57
70	0.000388	116.193	39.522	155.715
80	0.000316	117.399	38.756	156.155
100	0.000308	118.8735	39.800	158.67

Table 5: Comparing time taken among different distances l is 30

Distance	init time	fit	query	total time
50	0.0003872	63.074	33.56	96.63
60	0.0003312	63.49	33.565	97.05
70	0.000388	63.69	33.31	97.01
80	0.000354	64.35	33.91	98.26
100	0.005873	65.4791	33.37	98.856

Table 6: Comparing time taken among different distances k is 40

explanation

The data generated meets the expectations that as the distance increases high computation time increases meaning theres better embedding into Hamming space. Another observation is that the data generated differs from that in Problem 2 but remains similar wherever the distance, the hash tables L and concatenating value k parameters are changed.