DAT470/DIT066
Computational techniques for large-scale data

Assignment 6
Deadline: 2025-05-25 23:59

In this assignment, we are going to apply Locality-Sensitive Hashing (LSH) on a pre-trained GloVe dataset [1]. We are going to make use of two particular datasets: the smaller 50-dimensional Wikipedia dataset that is found on Minerva at `/data/courses/2025_dat470_dit066/glove/glove.6B.50d.txt` and the very large 300-dimensional Common Crawl dataset `/data/courses/2025_dat470_dit066/glove/glove.840B.300d.txt`.

The datasets contain *embeddings* of words into a high-dimensional vector space $\mathbb{R}^d$, constructed using an unsupervised machine learning method. This means that we can compute, for example, distances between the words that should reveal some kind of strength of association between them. To this end, we are going to use *cosine similarity* $s(\mathbf{x}, \mathbf{y}) = \cos\theta = \frac{\mathbf{x}\cdot\mathbf{y}}{||\mathbf{x}||\,||\mathbf{y}||}$ as our metric of similarity. That is, we only look at the *angle* between the vectors, not their magnitude.

When you are experimenting, start by using the smaller dataset first. Only use the larger dataset when you are happy with your results. Give all your responses in the report with respect to the larger dataset.

Problem 1: Preprocessing (2 pts)

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 (840B).

Problem 2: Matrix multiplication (4 pts)

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 $10 \times d$ matrix Q that contains the query vectors, and compute $QX^\top$ where X is the $n \times d$ 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 (840B).

Problem 3: Hyperplanes (6 pts)

Implement the reduction to binary vectors using random hyperplanes. Use `lsh_hyperplanes.py` as your starting point. Suppose the input dimensionality of data is d, and the parameter (number of hyperplanes) is D. Complete the implementation of the class `RandomHyperplanes` such that, in `fit()`, you draw D random Gaussian unit vectors of length (that is, draw D vectors $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ such that $x_1, x_2, \ldots, x_d \sim \mathcal{N}(0, 1)$ independently at random, and then normalize them $\hat{\mathbf{x}} = \frac{\mathbf{x}}{||\mathbf{x}||} = \frac{\mathbf{x}}{\sqrt{\sum_{i=1}^{d} x_i^2}}$). Let us call the output matrix R of shape $D \times d$. Then, in `transform()` project the $n \times d$ dataset X into D dimensions by computing $X' \leftarrow XR^\top$, and then compute matrix X'' such that $X''_{ij} = 1$ if $X'_{ij} > 0$ and $X''_{ij} = 0$ otherwise.

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

Problem 4: LSH (10 pts)

Implement the LSH, using `lsh_lsh.py` as your starting point. We will use the following family of elementary hash functions: $\mathcal{H} = \{f_i : \{0,1\}^D \to \{0,1\} \mid f_i(\mathbf{x}) = x_i, i \in [D]\}$. That is, each hash function simply selects one coordinate of the binary vector and projects it to that.

We will choose parameters L and k. We will construct L hash tables by *concatenating* k such randomly chosen elemenatary hash functions, and use that hash function for the hash table. That is, for hash table ℓ, we will use a randomly chosen hash function $f^{(\ell)} : \{0,1\}^D \to \{0,1\}^k$ such that $f^{(\ell)} = (f_{i_1}, f_{i_2}, \ldots, f_{i_k})$ where $i_1, \ldots, i_k \in [D]$ have been chosen uniformly at random, and we define $f^{(\ell)}(\mathbf{x}) = (x_{i_1}, x_{i_2}, \ldots, x_{i_k})$.

In practice, hash functions may simply be drawn by drawing a $L \times k$ matrix of random integers in the range $\{0, 1, \ldots, D-1\}$. As NumPy can index with arbitrary lists of indices, we can then just project the dataset efficiently by selecting columns by a row of the matrix.

Once we've drawn the hash functions, we will need to be able to fit the data. We do this as follows: we use the random hyperplanes from previous problem to project the data into binary vectors. Then we hash the data L times by using the L hash functions into hash tables. The hash tables will map binary k-vectors into a set of indices that record which elements in the original data were hashed into that particular tuple.

When we want to *query* a vector $\mathbf{q} \in \mathbb{R}^d$, we do as follows. We use the same random hyperplanes to transform $\mathbf{q}$ into a binary vector. Then we hash it L times using the same hash functions. We collect the union of all elements that were hashed in the same buckets with the vector. Finally, we compute the true distance to the vectors by taking the usual inner product with the orignal vectors (which we can access using the indices).

Try your implementation on the larger dataset (840B) 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.

Hyperparameter search (2 pts)

Although there are principled ways to select the parameters, we will just be happy to explore the effects a bit by trying them out. The intuition is as follows: The larger the D, the better the embedding into the Hamming space is, however, at computational cost because we need longer vectors. The higher k, the fewer collisions we have, so the number of vectors in the hash buckets is smaller, which makes queries more efficient but increases the probability that we miss the nearest neighbor. The higher L, the more buckets we look at, so this increases the probability of success, but uses more space and incurs a significant computational cost as we need to hash the data more times.

Try out how the values react. Discuss how the results look like in comparison to the ground truth (computed in Problem 2) when you adjust the parameters.

Hints

- The L_2 norm can be computed using `np.linalg.norm`.
- Use the `axis` parameter to only apply an operation along a certain axis (e.g., `axis=1` applies the operation by the rows).
- You can *broadcast*[1] the results of an operation that, e.g., provides a column- or rowwise summary, so that the values are implicitly copied to the other dimensions; this makes it efficient to, e.g., divide all rows by the norm of the vector along the row. No looping occurs on Python side which is the important bit!
- While the underlying linear algebra libraries provide parallelization, it doesn't seem to be very reliable and controlling the amount of cores used can be somewhat difficult. We'll just tacitly ignore this for now.
- `np.argsort` allows one to find indices that contain the greatest values for an array.
- The function `np.sign` is useful for determining whether a value is positive or negative.
- Selecting columns by another vector that indexes them (think: the hash function!) is easy, just say `X[:,I]`.
- Python dictionaries require that the keys are *hashable*; hashable types are typically immutable. As such, lists and NumPy arrays are not hashable. Use `tuple`s.
- You cannot avoid loops, for example, in the `fit` function of the LSH because you will need to interact with fundamental Python types.

[1] See `https://numpy.org/doc/stable/user/basics.broadcasting.html`

Returning your assignment

Return your assignment on Canvas. Your submission should consist of a report that answers all questions as PDF file (preferably typeset in LaTeX) called `assignment6.pdf`. In addition, you should provide the code you used in `assignment6_problem1.py`, `assignment6_problem2.py`, `assignment6_problem3.py`, and `assignment6_problem4.py`. That is, you should produce the code for your implementation; you needn't produce your experiments. The code must match the interfaces of `lsh_normalize.py`, `lsh_matmul.py`, `lsh_hyperplanes.py`, and `lsh_lsh.py`. Do *not* deviate from the requested filenames and do *not* produce the plots in these files; these files will be used for evaluating the quality of your implementations automatically.

References

[1] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "Glove: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*. 2014, pp. 1532–1543.