Dictionary Learning Algorithms

Objective

Our goal is to implement simple and fast dictionary learning algorithms. These algorithms are used to compress the embedding tables in deep learning models, which is often the largest component, and hence, very costly to store on mobile devices.

Background

The basic dictionary learning is stated as follows. Given a matrix **A** of size **m** x **n** (in the context of machine learning **n** is the number of features and **m** is the number of observations), we want to approximate the original matrix **A** by a product of a "sparse" representation **C** and a dictionary **D**. In this work, we seek to minimize the Frobenius norm of **E = A - C*D**. See https://en.wikipedia.org/wiki/Sparse_dictionary_learning for more details.

In the factorization, the representation **C** is a sparse matrix of size **m** x $\mathbf{m_D}$ and the dictionary **D** has size $\mathbf{m_D}$ x **n**, where $\mathbf{m_D} \ll \mathbf{m}$ and each row of **C** has at most **t** non-zero entries. Intuitively, each row of **A** is approximated by a linear combination of **t** rows of the dictionary **D**. Note that this problem is known to be NP-hard.

We define the compression ratio $\mathbf{r} := (\mathbf{mt} + \mathbf{m_D n}) / (\mathbf{mn})$. Normally, we will pick **t** and $\mathbf{m_D}$ such that **r** is in the range [0.1, 0.25].

Our algorithm

In this work, we study a simple iterative method. First, we initialize either **C** by kNN (k nearest neighbors) or **D** by k-means clustering. Next, we alternatively optimize **C** and **D** in each iteration:

- compute the best-possible representation **C** based on the current dictionary **D** -- this problem is significantly easier as **D** is fixed,
- compute the best-possible dictionary **D** based on the current representation **C** -- this is simply a least-squares problem.

We refer to the basic dictionary learning described above as the sum-approach (i.e., each row of **A** is approximated by a *weighted sum* of **k** rows of **D**).

For initialization of **C** and **D**, we can just start with random matrices. However, the algorithm converges much more quickly if we guess a better initial value of **C** or **D**. We use one of the following:

- Initializing **C** by kNN: Randomly choose a sample of some $\mathbf{m_D}$ rows of **A**. For each row $\mathbf{A_i}$, find **t** rows in the sample that are "closest" to $\mathbf{A_i}$. Next, row $\mathbf{C_i}$ will be initialized by setting the entries corresponding to these **t** rows to 1.
- Initializing **D** by k-means: Simply run the k-means clustering algorithm on the rows of **A** with **k** = $\mathbf{m_D}$. The centers will then be used as words of **D**.

As mentioned above, we will alternatively solve for **C** (or **D**) given a fixed value of **D** (or **C**) in each iteration:

- Optimizing the representation **C**: This problem is known as the Orthogonal Matching Pursuit (OMP) problem:
    argmin_**C** || **A** - **C*****D** ||^2
    subject to a constraint that each row of **C** has at most **t** non-zero entries.
  (This problem can be solved in polynomial time.)
- Optimizing the dictionary **D**: Again, we want to find **D** such that ||**A**-**C*****D**||^2 is minimized. Observe that this is just a least-squares problem.

Compared to scikit-learn's dictionary learning function, our approach has several advantages (see the next section for detailed comparisons):

- Scikit-learn's implementation does not consider sparisity of **C** as a hard constraint. Instead, it tries to control the sparsity via **alpha** -- a multiplier of the L1 norm of C which is an additional term in the objective function.
- Experiments with a few different datasets show that our algorithm generally gives better approximation of **A**.

Experimental results

Our datasets include an embedding table created from the PTB dataset and a randomly generated matrix.

PTB

We will use the popular Penn Tree Bank (PTB) dataset to compare our approach with the standard implementation of dictionary learning in scikit-learn. We run a Tensorflow model to obtain an embedding table of size 10,000 x 650. (The matrix can be found at /cns/iz-d/home/discrete-algorithms-eng/PTB/ptb.2.weights.txt). Our results are as follows.

1. Running scikit's implementation with **alpha** (the sparsity controlling parameter) in {0.025,

0.03, 0.035, 0.04, 0.045, 0.05}, **n_components** = 1000 (i.e., extract 1000 words in the dictionary), and **max_iter** = 32 gives

alpha	Compression ratio	Normalized Frobenius error (after 32 iterations)	Running time
0.025	37.53%	41.10%	17h 13m
0.03	30.60%	46.34%	20h 8m
0.035	25.40%	50.58%	6h 29m
0.04	21.63%	54.06%	9h 52m
0.045	18.92%	56.83%	6h 9m
0.05	16.98%	59.12%	5h 26m

Table 1: The Frobenius errors by scikit-learn's implementation of dictionary learning.

2. Results of our implementation (for the sum approach) with **n_iterations** = 32 is shown in the following table.

row_percentage	col_percentage	Compression ratio	Normalized Frobenius error (after 32 iterations)	Running time
0.2	0.1	30%	31.13%	9h 15m
0.2	0.15	35%	24.89%	1d 6h 19m
0.2	0.2	40%	19.85%	1d 13h 15m
0.15	0.1	25%	34.62%	21h 32m
0.15	0.15	30%	28.27%	17h 13m

0.15	0.2	35%	23.05%	1d 1h 58m
0.1	0.1	20%	39.43%	10h 25m
0.1	0.15	25%	33.18%	12h 43m
0.1	0.2	30%	27.87%	1d 4h 33m

Table 2: The Frobenius errors of the sum approach on the PTB embedding table.

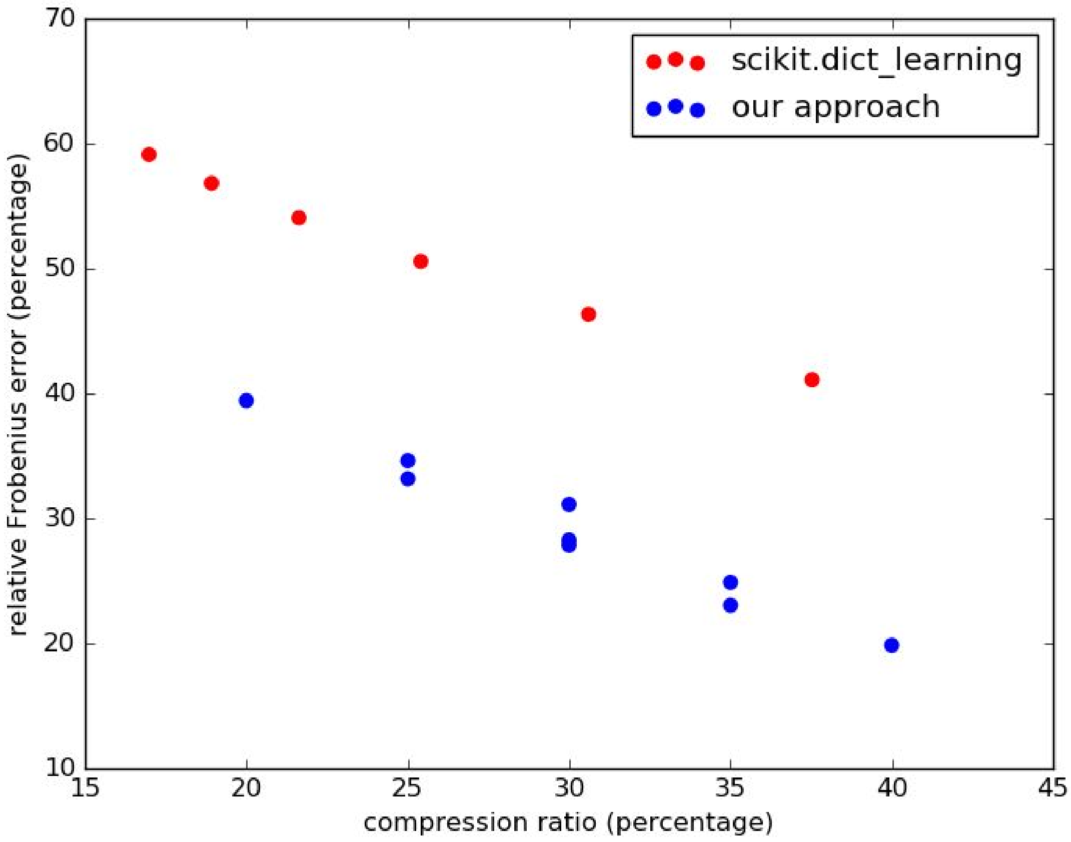

Figure 1: Scikit-learn’s dictionary learning function vs. our approach (PTB).

Random matrix

Here we randomly generated a matrix of size 10,000 x 1,000 where each entry is uniformly distributed in the range [-1, 1]. We compare the performance of scikit-learn’s function dict_learning() with our implementation using this random matrix. Our results are as follows.

1. Running scikit's implementation with **alpha** (the sparsity controlling parameter) in {}, **n_components** = 1000 (i.e., extract 1000 words in the dictionary), and **max_iter** = 32 gives

alpha	Compression ratio	Normalized Frobenius error (after 32 iterations)	Running time
0.03	83.27%	15.30%	3h 45m
0.035	79.66%	17.69%	4h 20m
0.04	76.42%	20.05%	6h 59m
0.045	73.36%	22.37%	4h 8m
0.05	70.29%	24.65%	5h 39m
0.1	46.38%	45.31%	1h 52m
0.15	30.05%	62.09%	2h 2m
0.2	19.66%	74.93%	42m
0.25	13.90%	83.75%	35m
0.3	11.45%	89.09%	32m
0.35	10.49%	92.65%	28m
0.4	10.14%	95.23%	20m

Table 3: The Frobenius errors by scikit-learn's implementation of dictionary learning.

2. Results of our implementation (for the sum approach) with **n_iterations** = 32 is shown in the following table.

row_percentage	col_percentage	Compression ratio	Normalized Frobenius error (after 32 iterations)	Running time

0.2	0.1	30%	36.66%	53m
0.2	0.15	35%	26.94%	59m
0.2	0.2	40%	20.14%	1h 53m
0.15	0.1	25%	40.43%	1h 21m
0.15	0.15	30%	29.15%	39m
0.15	0.2	35%	22.87%	1h 38m
0.1	0.1	20%	45.79%	34m
0.1	0.15	25%	34.01%	49m
0.1	0.2	30%	25.58%	51m

Table 4: The Frobenius errors of the sum approach on the above random matrix.

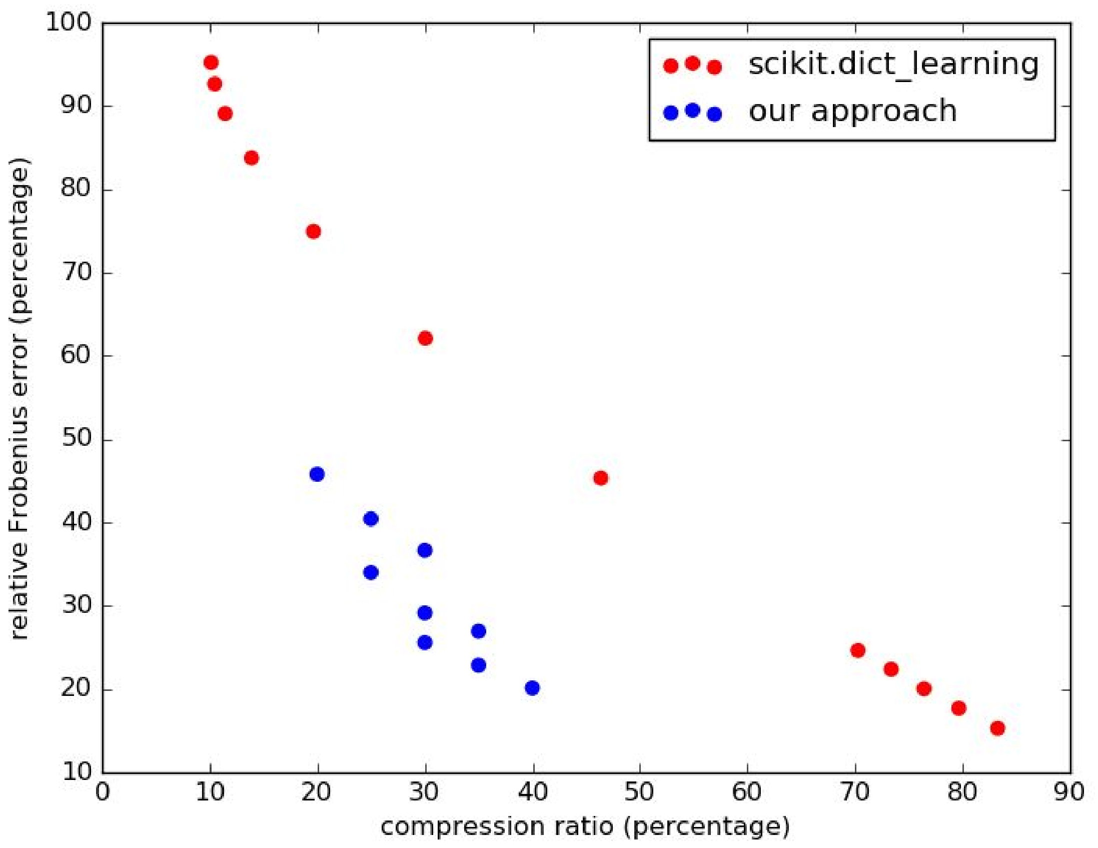

Figure 2: Scikit-learn's dictionary learning function vs. our approach (random matrix).