Université Paris-Dauphine

Mathématiques et Informatique de la Décision et des Organisations

Map-Reduce implementation of Expectation Maximisation algorithm on a Hidden Markov Model

Authors:
Hosseinkhan Rémy
Deschamps Théo
Hattabi Mahmoud

Advisor:
Colazzo Dario

M2 Intelligence Artificielle Systèmes Données

Febuary 2020

Acknowledgements

It was a real pleasure to learn the Hidden Markov Model theory and implement it using the distributed computing framework Spark. Thanks to Dario Colazzo for proposing this research topic that has opened our scientific minds. A lot of knowledge has been gained through this work.

Contents

1 Introduction

The goal of this project was to applied a map-reduce algorithm on large data sets to find "rules" (find best parameters) that illustrate the most our examples. Our work support was the sixth chapter of [1]. The method implemented is well known to induce rules on textual data. For example, it works well for finding annotations given as input sentences. We use a probabilistic approach based on two topics: Expectation-Maximisation algorithm and Hidden Markov Model.

1.1 Definition

In this framework, we have observable input sequences $(X_t)_{t\in\mathbb{N}}$ from a finite set X and latent variables sequences $(Y_t)_{t\in\mathbb{N}}$ from an countable space Y unobserved. We assume that every sequence from Y is a Markov chain, and so it is describe by an initial law and a transition matrix. We also supposed that every emitted word from a state are independent. It means that for every $K \geq 0$, $x_1, \ldots, x_K \in X$ and $y_1, \ldots, y_K \in Y$, we have:

$$P(X_1 = x_1, \ldots, X_K = x_K | Y_1 = y_1, \ldots, Y_K = y_K) = \prod_{i=1}^{K} P(X_i = x_i | Y_i = y_i)$$

This implies that the model, (the joint probability between X and Y) is a hidden Markov model (HMM). It is describes by a tuple $\langle S, O, \Theta \rangle$ where S is the set of states, O refers to the observation vocabulary and Θ is the set of parameters. It is composed by a transition matrix A, an emission matrix B and an initialisation vector probability π. The main advantage of those assumptions is for computing the likelihood of the observable sequence over data. Indeed, we could compute it this way:

$$\begin{aligned}
P(X_1 = x_1, \ldots, X_K = x_K) &= \sum_{y_1,\ldots,y_K \in Y^K} P(X_1 = x_1, \ldots, X_K = x_K, Y_1 = y_1, \ldots, Y_K = y_K) \\
&= \sum_{y_1,\ldots,y_K} P(X_1 = x_1, .., X_K = x_K | Y_1 = y_1, .., Y_K = y_K) P(Y_1 = y_1, ., Y_K = y_K) \\
&= \sum_{y_1,\ldots,y_K} P(Y_1 = y_1) \prod_{i=2}^{K} P(Y_i = y_i | Y_{i-1} = y_{i-1}) P(X_i = X_i | Y_i = y_i)
\end{aligned}$$

This is the intuition over the forward algorithm which give us the opportunity of computing the likelihood over a sequence at each time. It is computed recursively.

After assuming that we had an HMM, the question that we wanted to answer was to find the maximum likelihood estimation (MLE) given training data. The problem here is that the sequences Y are not observable. So we will computed the MLE over the marginal likelihood of the training data and summing over all latent variables. We then find a formulation that can't be computed analytically but we can use the EM algorithm.

The EM algorithm creates a series of parameters that improves the marginal likelihood. It is decomposed in two steps. The first one, E-step, assign a distribution over the latent variables, which is the posterior probability of each hidden variable for every input sentences and with the current parameter estimation. Thanks to this distribution we can now compute the MLE over the expected log likelihood of the joint distribution with respect to the distribution computed in the E-step. This is the M-step.

If we want to bring clother EM and HMM, we can see that computed the forward and backward algorithm (they are respectively two ways of computing the probability of being in a state given a input sentence and the probability of seeing the rest of the sentence for a known state) can be used to compute the number of times each state transitions into an other state, or the the number of times each word is generated from a state. This is the E-step:

$$P(y_i = q|x;\theta) = \alpha_i(q).\beta_i(q)$$
$$P(y_i = q, y_{i+1} = r|x;\theta) = \alpha_i(q).A_q(r)B_r(x_{i+1})\beta_{i+1}(r)$$

Those probabilities are used in order to compute the M-step:

$$\mathbb{E}[O_{qo}] = \sum_{i=1}^{|x|} P(y_i = q|x;\theta).\delta(x_i, o)$$
$$\mathbb{E}[T_{qr}] = \sum_{i=1}^{|x|-1} P(y_i = q, y_{i+1} = r|x;\theta)$$

where T_{qr} is the number of times state transits from q to r, and O_{qo} counts the number of times word o is emitted from state q.
The M-step is thus normalising those quantities in order to have probabilities matrices.

Now that we have introduced those two concepts, we can present how to implemented EM algorithm over an hidden Markov model with the map-reduce paradigm.

1.2 Map-Reduce design

Each map-reduce job correspond to one EM iteration. Each job works as follows: first, we compute the forward-backward algorithm for every instance of the dataset (for example for every sentences). By doing this, we get actually the probabilities necessary for computing the number of time we switch from a state to another, also the one for computing the emission frequency of a word given each state and the one to compute the number of time we start in every state. This is done for every observations in the map phase and correspond to the E-step of EM algorithm. After the shuffle and sort process, the reducer is applied. Here it consists only of aggregating (by summing) all together the previous probabilities computed in the mapper in order to compute the expected values of starting in a state, of transiting to a state to another, and of emit a word given a particular state. Then, still in the reduce phase, we normalise all parameters in order to have probabilistic matrices (all rows summing to one). It corresponds to the M-step of EM.

In the paper, we have implemented two versions of this map-reduce algorithm:

- A first one fully coded with python (without requirement of any python library), based on associative array for representing parameters.
- A second one based on the numpy library. We represent parameters as numpy array and we use pre-implemented numpy operations to boost computation efficiency.

Below is presented the Baum-Welch algorithm. It is a special case of the EM algorithm where both states and observations spaces are discrete. For Further theoretical explanations see [2][3].

2 First implementation description

The first algorithm designed only requires pySpark with Python 3.7. Intentionally, this implementation does only depend on built-in Python packages. It was intended to faithfully replicate the book's implementation. Hence the inefficient loops were kept and the associative arrays are represented by built-in Python dictionaries.

Basically, both implementations first process the input text data to clean it and create a text RDD. Then a unique count is performed to create a *vocabulary* of all the unique words occurring in the corpus resulting in a set object that represents the support $\mathcal{O}$ of the non-hidden variables defined by the process X.
Then, the initial, transition and emission matrix are randomly initialised so that the expectation step can be done by running the forward and backward steps in parallel. The result of these two computations is returned in two separate RDD to finally derive the new probabilities for each partition. Note that since this latter part involves summing, hence aggregating intermediary results, it may be considered as an in-mapper combiner. More on this topic will be discussed in the last section of this report.
This process is repeated until the step counter reaches a given treshold.

Moreover, all model parameters and inputs such as the three parameter matrix are *broadcasted* to enhance performances:

> Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost. [4]

At each iterations, the estimated matrix are uncached then the newly computed initial, emission, transition parameters are broadcasted again. This should enhance algorithm performance.

Besides, the whole text file is processed from the very beginning using Spark Context methods and regex mappings in order to create a RDD only containing short sentences. It shall be important to consider short sentences because of the computational behaviour that long sequences involve: during the expectation step, the forward and backward variables computing results in large products that may not be well managed by the computer due to computation limitations. More details about this issue will be given later.

Also, one critical RDD transformation for both implementation is the *zip* method:

> Zips this RDD with another one, returning key-value pairs with the first element in each RDD, second element in each RDD, etc. Assumes that the two RDDs have the *same number of partitions* and the *same number of elements in each partition* (e.g. one was made through a map on the other).

It allows to barely merge our sequences with the others RDD containing the results of the Baum-Welch algorithm. These operations are time consuming hence the zipped RDD are persisted in memory.

Since this implementation only depends on built-in Python and try to keep as much as possible the book's design, a vectorized NumPy-based implementation as been done to

compare performances. Two component of this second implementation may affect the performances of the Hidden Markov Model training process, both in memory and speed :

- NumPy allows to vectorize matrix multiplications and avoid loops thanks to its well-designed C/C++ backend.
- The second implementation requires a non-dynamic memory allocation when building matrix, while Python dictionnaries have appealing memory properties.

3 Second implementation

As we said before, we have tried to implement the map-reduce pseudo-code algorithm with the help of the scientific numpy library. Numpy is an array object oriented library, and thus is optimised for arrays multiplication.

In this implementation, it is true that using an array representation and numpy built-in functions can save our self some time and some lines of code.
As an example, the forward algorithm apply on a given sentence, x compute an $|x| \times |\mathcal{O}|$ alpha matrix where each row t correspond to the joint probability of having these observations $x_1, \dots x_t$ (here observations are words) and being in a particular state (so a row is a vector of probability where each component is associated to a state). In the book, the forward algorithm formulation is given as follows:

$$\begin{cases} \alpha_1(q) = \Pi_q . B_q(x_1) \\ \alpha_t(r) = B_r(x_t) . \sum_{q \in S} \alpha_{t-1} . A_q(r) \end{cases}$$

for some states q and r in S. We can see that this algorithm can be re-write as matrix multiplication, and so it will avoid us looping over state for computing the alpha matrix of a particular sentence x.

$$\begin{cases} \alpha_1 = \Pi W(x_1) \\ \alpha_t = \alpha_{t-1} A W(x_{t-1}) \end{cases}$$

where $W(x)$ =diag$[B(x,1), \dots, B(x,|\mathcal{S}|)]$ and α_t is a vector of length $|\mathcal{S}|$. Thus, using this recursive formulation in a function, and "np.matmul" methods from numpy (compute matrix product of two arrays) will be an efficient way of computing the forward algorithm. In a similar way, we have implemented the same "trick" for the backward algorithm.

The rest of the algorithm is quite trivial when we have computed the forward and backward algorithm. We can used a point wise multiplication method from numpy ("np.multiply") to avoid us looping over states again.

However this approach has two main disadvantages compare with the associative array method:

- The first one is that we need one more operation with this approach. Indeed, we have to get all words indices for each sentence in order to update the emission probabilities matrix which (give us the probability of generating a particular word knowing a state). Each column index of this matrix refer to the same index word of the vocabulary list (which is composed of all unique words of the corpus). This process is quite expensive and is computed with a loop operation over all words for each sentences. We think we can optimise the way we have gotten theses indices.

- The second one is a memory issue and is related with the previous point. For each sentence, we need to allocate an $|\mathcal{S}| \times |\mathcal{O}|$ for updating the full observation matrix. It is not optimised because for each particular sentence, we only need to upload the probability values of seeing those words considering each state (and not for all unseen words). In the associative array method, we don't need to allow this big matrix for each sentence. Since dictionary's keys are words, it will only load the values associated to those words. So for a sentence with a few number of words, it will be a small memory allocation compare to the numpy approach where we upload the full matrix.

One can think of some optimisation improvements from our code implementation:

- find a matrix multiplication formulation for the "loop_t" function. It is the one which the compute the probabilities to get the expectation of the transition matrix between every couple of states.
- find a way of getting words indexing without looping over words for every sentences (a an optimised look_up function will be great).

4 Results comparisons

To compare efficiency of both algorithms, ten iterations of the forward-backward algorithm are performed. The associated Spark application is divided in 31 jobs for this 10-steps training task. It is worth noting that both implementations almost follow the same map and reduce structure: the main differences lie in the way mappings are constructed and the in the initialization since the NumPy algorithm needs to perform a mapping between word tokens and integers.

The underlying tested model is a three states Hidden Markov Model. The data used for the experiment is an excerpt of the *Wesbury Lab Wikipedia corpus (2010)* [5], it contains 298876 words and 27536 unique words. Formally, $|\mathcal{S}| = 3$ and $|\mathcal{O}| = 27536$.

4.1 Execution time comparison

First, an execution time comparison is proposed. Jobs duration time series of the whole training are presented in the figure 1 below.

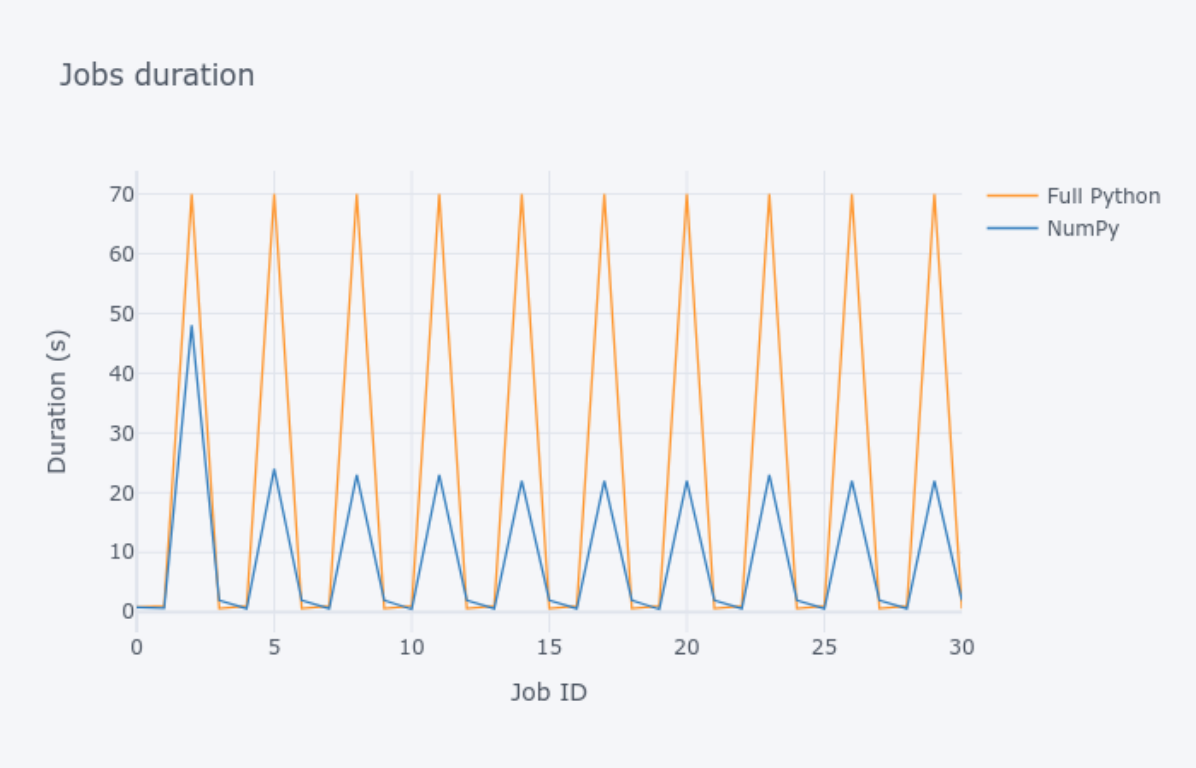

Figure 1: Jobs duration for each implementation of the HMM training

Ten periodic spikes can be observed, they describe the huge amount of computation required by the expectation step to derive the new estimation of the large emission matrix O. One can notice the first NumPy implementation spike is higher than the next ones because of the preprocessing related to the specific implementation. Anyway, the Python-only design is globally slower than the vectorized one as it is shown in the figure 2.

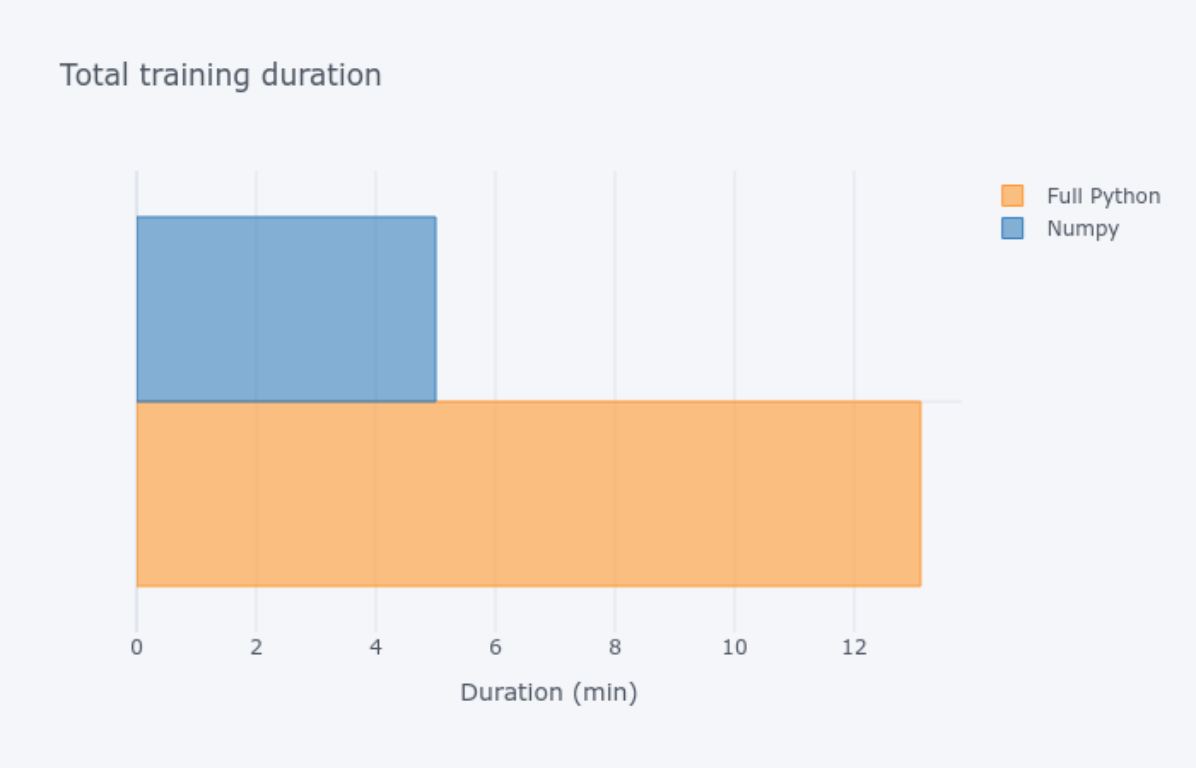

Figure 2: Total duration of the HMM training Spark application.

Indeed, the NumPy program runtime is 2,6 times faster than the full built-in Python program. The recursivity schema and the vectorization really improved the performances of the Baum-Welsch algorithm for the Markov Model parameter estimation.

4.2 Internal memory usage

In addition of the execution time improvment, the runtime memory consumption has also decrease thanks to Numpy arrays. The figure 3 shows the memory size of jobs input during the program execution.

One can note that before job 7, jobs input memory sizes are barely 1.5 time greater for the naive implementation and then become 3 to 4 time larger than the jobs input sizes of the NumPy model.

Hence the underlying low-level structure of NumPy arrays has a bit improved the memory efficiency of the distributed algorithm despite repetitive in-mapper large matrix initialisation.

Finally, even if the proposed algorithms should be improved, the use of associative array in the most classical Pythonic way - dictionaries - seems to be less-optimal than using high-performance multidimensional array objects.

Figure 3: Memory used by internal data structures

5 A brief book review

In this part, a brief discussion of some parts of *Data-Intensive Text Processing with MapReduce* [1] on which the work was based is proposed. It focuses mainly on the sixth chapter.

First of all, the purpose of the *combiner* suggested page 137 is not clear. Indeed, the combiner are supposed to perform *local aggregation* before the shuffle and sort phase but the only possible local optimisation seems to be already done when looping over observations and states to compute the new values of the transition matrix T and O.

Hence the combiner seems to be introduced inside the mapper as it is discussed in the section 3.1.1 of the book. This technique is called *in-mapper combining*. Moreover, the mapper emits *unique* keys, one for each state defined in the Hidden Markov Model and each parameter matrix I, O and T. Thus $3|\mathcal{S}|$ distinct keys are emitted by the mapper for each matrix, therefore it is not possible to *locally* reduce anything by key.

Another important consideration is the vanishing statistics problem that is common for Hidden Markov Models parameters estimation. Even more for text processing, when the size $|\mathcal{O}|$ of the set of possible observations $\mathcal{O}$ is large, the transition and emission matrix are initialised with very small values. The magnitude of the coefficient of the emission matrix decreases with the size of the set $\mathcal{O}$ since each row of the matrix sums to 1. For instance, it is common in NLP to deal with tens of thousands words which implies a very low magnitude for probabilities values.

On top of that, some observations, *i.e.* some words are very unlikely : thus a huge amount of data is required to accuratly estimates their associated emission probabilities. Consequently, due to computation limitation, the forward-backward algorithms returns sparse

matrix. To address this issue, it is possible to compute the *log-likelihood* of the observed model that is more *well-scaled* than the classic one.

It is worth noting that besides converging toward local minima, the estimated parameters found using Baum-Welsh algorithm have a large variance. A consistent preprocessing should be done before fitting such a model, as many unsupervised learning problems.

For this reason, the Map Reduce implementation presented in the section 6.3.1 may be considered as a demonstrative implementation. Optimisations might be performed within the mappers (loop removal) but the purpose of this chapter is certainly to explain how Markov Model training and by extension Expectation Maximisation algorithm can be parallelized. Also, by reading this chapter one should understand that many *expectation* problems may be parallelized since models with batches of observations that are processed iteratively are well designed for distributing computation. Namely, loss function minimisation using batch-gradient descent method is an other example of technique that can be paralellized.

6 Conclusion

Hidden Markov Model are frequently used for pattern recognition, natural language processing or for modelling biological sequences. The map reduce programming model allow to perform *batch processing* for the local optimum approximation.
The NumPy library and in-mappers optimizations such as recursion improve both in *space* and *memory* the algorithmic efficiency of the distributed Hidden Markov Model training. Aside from these improvements, there exist several optimizations that should allow to the algorithm and the implementation to work better.

References

[1] Chris Dyer Jimmy Lin. *Data-Intensive Text Processing with MapReduce*. 2010.

[2] Tobias Rydén Olivier Cappé Eric Moulines. *Inference in Hidden Markov Models*. 2009.

[3] Fabien Campillo. *Modèles de Markov cachés et filtrage particulaire*. 2006. URL: `https://cel.archives-ouvertes.fr/inria-00103752`.

[4] *Class Broadcast*. 2020. URL: `https://spark.apache.org/`.

[5] *The Wesbury Lab Wikipedia corpus*. 2010. URL: `https://www.psych.ualberta.ca/~westburylab/downloads/westburylab.wikicorp.download.html`.