GSoC 2017:
Online Non-negative Matrix Factorization for Gensim

A project proposal submitted by: Markus Beuckelmann

Online Non-Negative Matrix Factorization (NNMF) for Gensim

Abstract

Non–negative matrix factorization (NNMF) is a powerful *unsupervised* learning algorithm that is used to discover and unravel the *latent structure* of high–dimensional data, by decomposing a potentially huge but low–rank matrix V into a product two much smaller matrices. One of the big advantages of NNMF is its simplicity and the fact that it easily allows for interpretability, which is in contrast to many other unsupervised learning techniques. This projects involves contributing an *online*, *out–of–core* implementation of NNMF to Gensim that is able to make use of multiple physical cores on one machine at once. The implementation will be benchmarked in terms of accuracy and speed on different standard corpora in NLP. As of right now, different implementations of online NNMF exist, primarily in C++, but they do not necessarily support all features mentioned above. However, existing code bases could be used as a starting point for an efficient *Python/Cython*–based online, multi–core implementation for the Python ecosystem.

Technical Details

Non–negative matrix factorization (NNMF) is a very popular *unsupervised* matrix decomposition algorithm used to discover *latent structure* in high–dimensional data. It has been successfully applied in *various* fields from signal processing to computer vision, processing text corpora, audio spectrograms, recommending movies or clustering of gene expressions. Generally speaking, given a matrix $V \in \mathbb{R}^{n \times m}$ it tries to find a suitable decomposition of V into two low–rank *non–negative* matrices $W \in \mathbb{R}^{n \times p}$ and $H \in \mathbb{R}^{p \times m}$ in latent space, such that $V \approx W \cdot H$, where p is the number of latent variables expected. W is often called *basis* or *feature matrix* whereas H contains *coefficients* such that, in the end, each column in the product $W \cdot H$ is a linear combination of the column vectors in W. In other words: We rewrite V in a new basis that essentially corresponds to *hidden features* in the data, which ultimately results in a model that is easy to interpret. It turns out that NNMF has a *clustering property* and is very similar to the popular unsupervised k–means clustering.

An *online* algorithm does not require the whole data set to be kept in memory at the same time, but reads and processes the data in a streaming manner, where only one data point or a chunk of data points are held in memory at once. This is necessary because modern data sets can easily exceed multiple gigabytes in size and consequently the whole data sets do not fit into memory. Another reasons might be that the data actually arrives in a time–series stream, i.e. the whole data is simply not available at any point in time.

Furthermore, modern machines typically provide multiple physical CPU cores that are able to perform different tasks in parallel. However, not all algorithms are suitable for parallelization, but many algorithms can be re–phrased into individual sub–problems that can be (approximately) dealt with in parallel. If this is the case, this provides significant speed–ups. Fortunately this is possible for the NNMF formulation described below, and for this reasons, this project aims for an implementation that can distribute the workload over multiple CPUs on the same machine, achieving a run time that scales linearly with the number of available physical cores.

Existing Implementations

Other Python packages do include NNMF implementations, e.g. `scikit-learn` [4], but as of today I am not aware of any Python package that implements both an *online* version of NNMF that can deal with *mini–batches* to update the model in a sequential way during training *and* deal with multiple cores. There is a Python package on GitHub by `rserizel` called `beta_nmf` [6], which implements a `Theano` based mini–batch multiplicative update (MU) scheme for NNMF with β-divergence. Furthermore, there is a very fast and reliable *C++* implementation of online NNMF by the Machine Learning Group at *National Taiwan University* called `libmf`.

All three packages `beta_nmf`, `libmf` and `smallk` could be a possible starting point for a Gensim port in pure *Python/Cython*, where `beta_nmf` seems most promising. `beta_nmf` and `smallk` are well documented, whereas the `libmf` code base unfortunately comes without explanatory comments. Additionally, `smallk` comes with a Python wrapper called `pysmallk`. A first step could be to implement a Python *wrapper* (see below) that wraps

the `libmf` library, which will certainly be helpful in terms of comparing and benchmarking accuracy and speed of different implementations in later stages of development.

O–NNMF Implementation Details

The details of implementing an online version of NNMF are explained in Wang et al. (2011) and Serizel et al.. The following is a short synopsis of the algorithm, where $X \in \mathbb{R}^{d \times n}$ is the data matrix to be decomposed into the latent matrices $F \in \mathbb{R}^{d \times r}$ and $G \in \mathbb{R}^{n \times r}$ s.t. $X \approx F \cdot G^{\mathrm{T}}$.

Traditional NNMF algorithms compute F and G by using some kind of multiplicative update rule (mostly Lee and Seung's method) on matrices $(F^{(t+1)}, G^{(t+1)})$, where $F^{(t+1)}$ is determined with fixed $G^{(t)}$ followed by determining $G^{(t+1)}$ with fixed $F^{(t+1)}$. However, these update rules given by

$$F_{ij} = F_{ij} \frac{(XG)_{ij}}{(FG^{\mathrm{T}}G)_{ij}},$$

$$G_{ij} = G_{ij} \frac{(X^{\mathrm{T}}F)_{ij}}{(GF^{\mathrm{T}}F)_{ij}},$$

explicitly require to keep the whole data matrix X in memory, which is clearly unfeasible for large d,n. We can work around this using the following insights:

- The *Frobenius loss* $\mathcal{L}(F,G) = ||X - FG^{\mathrm{T}}||_{\mathrm{F}}^2$ can be decomposed as a sum, i.e. $\sum_{i=1}^{n} ||x_i - Fg_i||_{\mathrm{F}}^2$.
- In this case, $g_i \in \mathbb{R}^r$ is the i^{th} column of G^{T} and represents the coefficients to cluster the data points around cluster centers specified by F (the new basis).
- Let the cluster center matrix F be fixed, then $\mathcal{L}$ is minimized by minimizing all individual terms $||x_i - Fg_i||_{\mathrm{F}}^2$ independently. This is a standard problem and involves solving n independent Non–negative Least Square (NLS) problems.
- The solution to the above problems will yield n column vectors g_i representing the cluster coefficients.
- Consequently, the O–NNMF algorithm essentially breaks down in two steps: finding optimal cluster coefficients and adjusting respective the cluster representatives.

1. Finding G: Optimal Cluster Coefficients

In this step, we consider F to be fixed. The we solve n NLS problems, i.e. $\min_{g^{(t)}>=0} ||x^{(t)} - Fg^{(t)}||_{\mathrm{F}}$. This is a standard problem and numerous methods exist, e.g. the *active set method* or iterative procedures such as *Projected Gradient Descent* (PGD). One could also explore more sophisticated methods such as *Fast NNLS* (FNNLS) [8].

2. Finding F: Optimal Cluster Centers

When we obtain a new data point $x^{(t)}$, obtained $g^{(t)}$ from the previous step, we still have to update $F^{(t-1)}$ to obtain $F^{(t)}$. In Wang et al. (2011), different methods are discussed to achieve this. Most of them are a form of *gradient descend* (GD), where the gradient of $\mathcal{L}$ w.r.t. F is given by

$$\nabla_{F^{(t)}} \mathcal{L}^{(t)}\left(F^{(t)}\right) = -2 \sum_{k=1}^{t} \left[x^{(k)} \left(g^{(k)}\right)^{\mathrm{T}} - F^{(t)} g^{(k)} \left(g^{(k)}\right)^{\mathrm{T}} \right].$$

They discuss different forms of GD making using of either first or second–order information about the gradient to speed–up convergence. Their results show that first–order methods performed poorly, and the best performance was given by using second–order information and approximating the Hessian $\mathcal{H}$ using *diagonal approximation* (DA).

Extending to Multiple Cores: Mini–batch O–NNMF implementation with DA

The general idea to achieve parallelization for this problem is to have multiple *workers* work out the update steps for different *mini–batches* of suitable size at the same time. A mini-batch is a chunk of b data points sampled from the training data that is used to come up with a parameter update (in our case for both F and G^{T}) that will improve the model under the complete training data population. In our case, we end up with a procedure that is similar to *stochastic gradient descent*. Afterwards, the individual update steps will be passed on to the master process which will update the model as soon as k results are in the queue, where k is the number of workers. This procedure is inspired by the implementation of Gensim's multi–core LDA described in this article.

From a practical point of view, all the multi–core logic will be implemented using Python's excellent `multiprocessing` module, since it offers both local and remote concurrency, side–stepping CPython's Global Interpreter Lock (GIL) by using sub–processes [7].

I confirm that I have read, understood and actively acknowledge both the *contribution guidelines* and *expectations from students* (CONTRIBUTING-students.md) participating as a GSoC student under the NumFOCUS umbrella organization.

Schedule of Deliverables

May 1st - May 28th, Community Bonding Period

- Wrap up pull request #1227 (if not already done by that time).
- I want to use the time before the official coding periods begins to dive deeper into Gensim's code base. A good way to do this is to work on some open issues, ideally covering multiple different modules. Another way of achieving this sort of familiarity with

the code base, specifically with the *interface*, would be to actively use a good part of the algorithms available in Gensim and see what they offer on different data sets. The tutorial page seems like an excellent starting point for this.

- Become familiar, i.e. read the code and the documentation of the `beta_nmf` project in depth. Use it on example data sets.

May 29th - June 3rd

- To document my progress, I will write *blog posts* on a regular basis about different aspects, as I've found that actively writing down the details of what I am dealing with in a concise manner helps me attaining a deeper understanding of the underlying concepts. During the *Community Bonding Period*, I guess this would mostly focus on Gensim's interface, insights obtained from data sets or *lessons learned* while fixing bugs in Gensim. Then, when the official coding period starts, the focus will probably shift more towards details of *NNMF* and difficulties to overcome with the implementation.
- Find suitable data sets to run the matrix factorization on. A starting point for this could be the `examples` folder in `libmf` which contains data from different fields.
- Optional: Implement a Python wrapper, i.e. a Python interface, that wraps the existing `libmf` online NNMF *C++* implementation.

June 5th - June 9th

- Compare the performance of the different implementations mentioned above, i.e. `libmf`, `beta_nmf` and `smallk`, on data sets that seem suitable for the task.
- Specifically, find out what parameters and which of the various optimization methods used perform well on different data sets and try to find out if the results match the findings presented in the respective papers [2][3][5].

June 12th - June 16th

- Specify the OOP interface needed for the `NNMF` model and get immediate feedback from the community and/or mentor on this. Potential changes to the interface should be incorporated as early as possible.
- Begin drafting the actual `NNMF` model in `gensim/models/nnmf.py` (or maybe `nnmfmodel.py`) and implement most of the *boilerplate* methods as well as easier methods like loading or saving models.

June 19th - June 23rd, End of Phase 1

- Draft the model's `fit()` method (this is the juicy part!) guided by the implementations in `beta_nmf`, `libmf` and `smallk`. This will focus on using only *one* CPU for now.
- The first step is finding optimal cluster coefficients as outlined above. This involves solving multiple NLS problems, e.g. using `scipy.optimize.nnls`.
- At the end of the first phase, the `NNMF` skeleton model with meta–functionality (loading/savings models, etc.) should be committed and reviewed.

June 26th - June 30th, Begin of Phase 2

- Benchmark different algorithms and libraries for solving the NLS problems efficiently.
- To update the cluster centers F we need to implement a form of gradient descend, where a first step could be to implement first–order methods choosing the *learning rate* either manually or using common heuristics such as the Armijo–rule.
- The OOP interface for the `NNMF` model should be more or less fixed by now.

July 3rd - July 7th

- Adopt the learning scheme for finding F to using second–order information about the gradient. The Hessian could be approximated by using only its diagonal (DA).
- Implement unit tests for rudimentary methods of `NNMF` such as saving or loading models.

July 10th - July 14th

- Keep working on second–order methods and incorporating the Hessian into the parameter update process.
- In addition to DA, we could also implement *Conjugate Gradient* (CG) as described in [2].
- Debug the implementation and eliminate or work around possible existing bottlenecks. This step will actually be repeated multiple times during the full period of development. This involves using Cython or the Numba JIT compiler whenever suitable.

July 17th - July 21st, End of Phase 2

- The performance on different (small) data sets should be evaluated as a function of different model parameters.

- At the end of this phase I should have a coarse implementation of NNMF available that is able to perform mini–batch online learning on one core, preferably using second order heuristics to speed–up convergence. This functionality should be committed and reviewed.

July 24th - July 28th, Begin of Phase 3

- Now its time to make the algorithm capable of using multiple cores. To achieve this, multiple workers will work out update rules on different mini–batches. The results will be passed on to the master process which updates the model.
- Implement the necessary logic for the master process and the workers using the `multiprocessing` library. This involves coding up the internals of worker processes and allow for communication between workers and master process.

July 31st - August 4th

- Keep working on the multi–core implementation.
- Find out how the run time scales with number of workers.
- Begin working on documentation of the new model.

August 7th - August 11th

- Sometime early within this week, a working version of multi–core online NNMF should be committed to give the mentors and community enough time for review.
- At this point I should catch up with missing documentation, i.e. adding comments and docstrings when necessary.
- Now I should also start with implementing unit tests testing core functionality including corner cases. This all belongs in `gensim/test/test_nnmf.py` (or maybe `test_nnmfmodel.py`).

August 14th - August 18th

- Experiment with different parameters of the model (specifically different mini-batch sizes), as well as comparing the performance of *one–pass* NNMF to *multi–pass* NNMF.
- Debug the implementation and its bottlenecks. Try to improve the model's performance.
- Ultimately, benchmark the algorithm on standard NLP corpora.

August 21st - August 25th, Final Week

- Write an example IPython/Jupyter notebook containing sample code and comments on how to use the `NNMF` model within Gensim.

August 28th - August 29th, Submit final work

- The final work will consist of one or multiple pull requests against the Gensim GitHub repository containing:

1. An *online*, *out–of-core*, *multi-CPU* implementation of NNMF in `gensim/models` that implements the Gensim interface.
2. The implementation must contain *comments* for non–trivial code and provide *docstrings* for methods and functions.
3. Unit tests that cover ideally the complete interface and assure its functionality on small test cases including corner cases.
4. A benchmark testing accuracy and speed on one or two standard corpora. This includes an analysis of how the run time scales with number of workers/physical cores.
5. An *IPython/Jupyter* Notebook containing example code and explanatory comments on how to use the NNMF model, preferably on text data such as the English Wikipedia or the Lee corpus.

Future works

- So far the implementation only supports multiple cores on one machine. One step further would be to implement a *distributed* version of NNMF that is able to work in a *cluster environment* consisting of multiple machines (and multiple cores). This is something that could be done based on Tensorflow.
- The algorithm detailed in [13] is also something that could be implemented in the context of a distributed *high–performance* algorithm as part of future work.

Development Experience

I have a strong background in *Python* (> 10 years experience) including the *scientific stack* (e.g. *Numpy*, *SciPy*, *Pandas*, *scikit-learn*, *Theano*, *Matplotlib*, ...) around it (>6 years experience) that I have used extensively to work on a variety of projects over the last years. I feel confident in benchmarking and identifying bottlenecks using various profilers and implementing performance critical functions in Cython. I am a big fan of Git and GitHub and have made minor contributions to other open–source projects.

Other than Python, I have experience in *C* and *C++* and have worked as a web developer using technologies such as *PHP* (in the old days), *Django* and *Flask* (more recently) for backend development and *JavaScript*, *HTML5*, *Bootstrap* and the *Jinja2* template engine for frontend development.

I edit files using (a shockingly complex and ever growing configuration of) Emacs on ArchLinux. I occasionally write blog posts and enjoy solving algorithmic and mathematical puzzles on ProjectEuler or LeetCode.

Other Experiences

I am currently a second year Master's student pursuing a *M.Sc.* degree in physics at University of Heidelberg in Germany. I have specialized in *computational physics* with a focus on *statistical modeling* and *Machine Learning*, and have worked on a variety of *ML* related projects in the past. Over the course of my studies I have obtained a rigorous mathematical foundation. Additionally, I have taken a whole lot of computer science classes where I've learned a lot about *algorithms*, *data structures*, *theoretical CS* and *scientific computing* to efficiently develop software with a focus on performance critical applications.

You will find more information on both Machine Learning and software engineering projects on my personal website and on my GitHub profile.

Why this Project?

- I feel very excited to contribute to a code base that has a large number of users in both academia and industry. Working on a project like this helps me to stay motivated over the full period.
- This algorithm has many applications in diverse fields, not only restricted to NLP, which makes it particularly interesting for the scientific Python ecosystem. I feel that I have both the necessary mathematical foundation and programming capabilities, but that there is a lot (!) that I could learn this summer.
- Natural language processing is one of the fields of ML that I haven't had a lot of experience, yet. However, it has been on my list for a very long time, and contributing to Gensim will obviously provide many opportunities to work with text data.
- After all, this is also a thrilling opportunity to learn and expand my knowledge on various technologies used throughout the project, on improving both my software development and communication skills to work effectively as part of a team.
- I have always wanted to get more involved in contributing to open–source projects, specifically to the *scientific stack* around Python. The participation in GSoC could be a starting point for a more continuous habit of actively contributing and giving back to the community.

Appendix

[1]: Gensim's GSoC 2017 ideas page, [2]: Efficient Document Clustering via Online Nonnegative Matrix Factorizations, [3]: LIBMF: A Library for Parallel Matrix Factorization in Shared-memory Systems, [4]: NMF in scikit-learn, [5]: Mini-batch Stochastic Approaches For Accelerated Multiplicative Updates in Non–Negative Matrix Factorization With β-divergence, [6]: `beta_nmf` on GitHub, [7]: multiprocessing — Process-based "threading" interface, [8]: Fast NNLS [9]: Scalable SGD, [10]: Parallelized Online Matrix Factorization for Collaborative Filtering using Stochastic Gradient Descent, [11]: High-performance Non-negative Matrix Factorizations (NMF) - Python/C++, [12]: Recommender System Using Parallel Matrix Factorization (libMF wrapper), [13]: A High-Performance Parallel Algorithm for Nonnegative Matrix Factorization