Networked Life: Project

The project's goal is to introduce new algorithms and techniques to provide custom recommendations to users based on what other users liked. This should remind you of the Netflix chapter, where a linear regression is used to recommend movies to a user based on a large (sparse) matrix of ratings. The chapter tells you about this big dataset that Netflix made available for scientists to play with (with added benefits for Netflix of course!) We will be using parts of this dataset for the project, allowing you to use real data and learn a few tricks that data scientists have developed over the last few years.

The project is divided in three parts:

- First, you will expand on the homework given to you for chapter 4 by reusing the code you have written previously, applying it to the Netflix training dataset. You may need to perform a few additional steps to obtain the data in the right format, but you will be guided along the way.
- Second, you will review a few concepts of Bayesian inference that will come handy when developing another algorithm to predict a user's preferences: the restricted Boltzmann machines. We will give you some theory and it will be your job to fill in the gaps in our code.

As was explained in the Networked Life book, the teams competing in the Netflix prize were given the opportunity to test their algorithms against a hidden test set. We will do the same, hiding from you a part of the dataset, and rewarding teams that perform best against the test set. There is no unique way of implementing the best algorithm, and a lot of the work is "guessing" (smartly) which parameters to use and which combination of techniques works best. Along the way, there will be tests to make sure that you have implemented the base algorithms correctly, but we also encourage you to experiment and see for yourself. Due to the popularity of the Netflix contest, a whole universe of online articles flourished, detailing the techniques used to win the contest and explaining them in a very approachable way. You may find some of these during your research, which you can take inspiration from (and duly cite in your reports!)

The project is built in a way such that you can get an early start, as soon as chapter 4's homework is completed. The three parts will be equally demanding, so we do advise you to start at this point. Both Python and some good skills in machine learning (more broadly: data science) are great to have on your resumes if you are going to pursue any career in technology. We hope this project gives you more tools to convince your dream company that you are ready for the job!

1 Linear regression on the Netflix dataset

In the folder you have downloaded containing the project, you will find in the resources subfolder the training dataset, `training.csv`. The CSV file can be imported in your code using the `getTrainingData` function in the `projectLib` file.

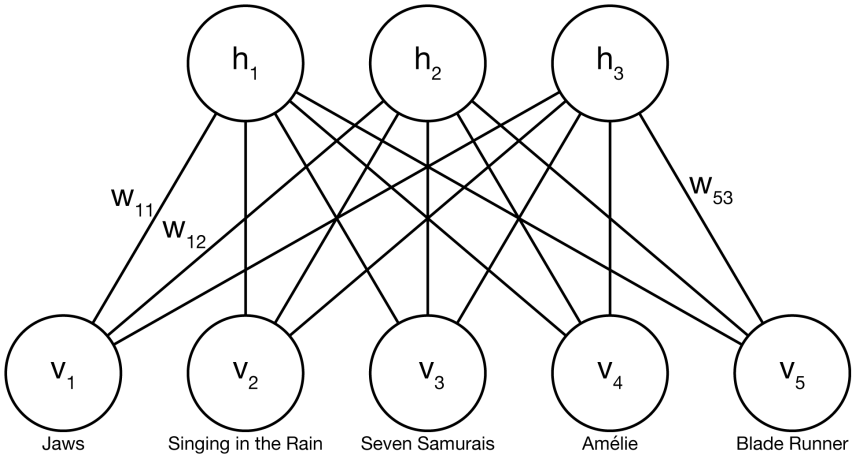

Figure 1: A restricted Boltzmann Machine with 5 visible units and 3 hidden units.

It has three columns: one for the movie ID, one for the user ID and one for the rating ID, *in that order*. For example, a row of 5,3,4 means that user 3 has given to movie 5 a rating of 4. Note that to make the scripts (a lot) simpler, we have reindexed movies and users so that movies are $0, 1, \ldots, M$ and users are $0, 1, \ldots, U$, where M is the total number of movies and U is the total number of users.

The training data set provided can be divided in two, one part, the training set, containing 80% of the data and the other, the validation set, with 20% of the data. In this project it is better to divide the data of each user, so if a user has rated 5 movies, put 4 random ones in the new training set and the remaining one in the validation set.

Question 1.1. *Using the solution you have worked out in the homework for chapter 4 or the one provided, run the parameter estimation* `param` *function to compute the estimator for the training set, without regularisation. Once this is done, use the* `predict` *function to compute the predicted rating of every (movie, user) pair of the training set. You now have the real rating from the dataset $r_{m,u}$ and the one predicted by your model $\hat{r}_{m,u}$. You can compute the RMSE to compare the two.*

Question 1.2. *As you have seen in Chapter 4, you can regularise this model so that it does not overfit, by adding a penalty to your biases. You should now complete the* `param_reg` *function with your previous homework implementation.*

In section 3, you will learn how to tune the regularisation parameter λ to optimise the performance of your model on a validation set, a good proxy for the test set.

2 The hidden patterns: predicting the rating with Restricted Boltzmann Machines

2.1 Presentation of the model

Neural networks are all the rage when it comes to training estimators on very large datasets, due to their flexibility. Our previous model was biased in the sense that we give the shape of the function we use for our estimator (the average rating plus the biases). Neural networks on the other hand are relatively agnostic when it comes to which part of the function space to explore to find the best estimator, which works especially well when the dataset is very big.

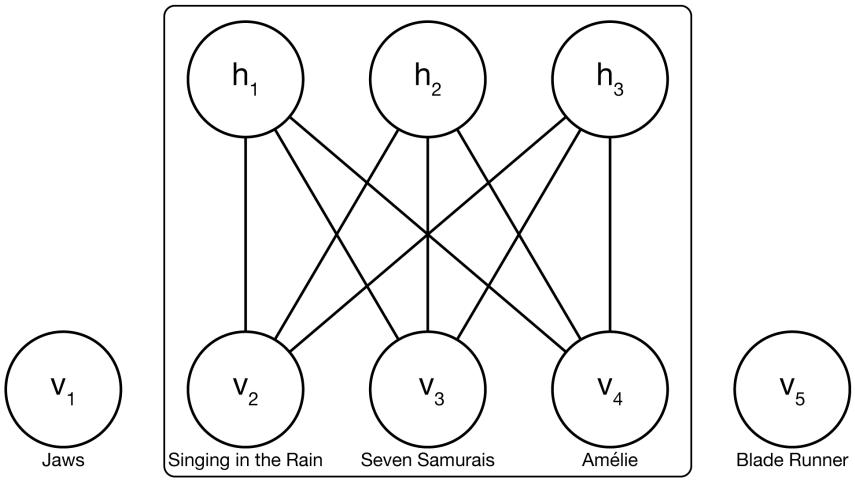

Figure 2: We only look at a part of the main RBM if the user has seen *Singing in the Rain*, *Seven Samurais* and *Amélie*, but not the other two. Only the weights inside the box will be updated when we compute the gradient for that user.

In this project we use a much simpler version of a neural network, the Restricted Boltzmann Machine (RBM). Developed by Salakhutdinov et al. for use on the Netflix dataset, it is a two layer network as shown in figure 1. The bottom layer, which we call *visible layer* V or *input layer*, receives an *input*, typically a piece of data (such as the ratings). The top layer, also called *hidden layer* H, receives a non-linear transformation of the input. The non-linear part matters because this is what allows neural networks to explore a large space of functions to find the estimator. We will present the precise model in the next section.

The edges connecting the visible layer to the hidden one are given *weights* W. It is in fact the end goal of the learning procedure: **to find the best weights W to represent our data.** By representing our data, we mean that we want at the end of our training to have one very good RBM to work with. A good RBM is one that encodes the information contained in our dataset properly, i.e is able to pick out patterns from the ratings of the users. These patterns could be anything from "Users with an odd ID number like movies with an ID ending in a 2" or "Users who enjoyed movie i did not really like movie q", though the latter is definitely what a good RBM will find out. This good RBM corresponds to a particular choice of weights W. After training our model on the data, we get an estimator of W, similar to our previous estimation of the biases in the linear regression method.

We now explain how we will use the RBM model. Each visible unit will correspond to one unique movie. Each hidden unit will correspond to some "abstract" notion. The model is powerful because we do not need to specify what these abstract notions are. It may happen that the model will be able to separate sci-fi movies from action movies, simply by looking at the data and the associated ratings. One hidden unit may *activate* (more on this later) when the ratings fed to the visible layer are those of a sci-fi movie fan, and this may help predict the rating given to an unseen sci-fi movie by the same user.

The dataset we provide contains a smaller set of m movies. We call **the main RBM** one big RBM consisting of m visible units and a number F hidden units (which you can tune). Each user u has rated a subset M_u of movies. This user u will be associated with a part of the main RBM, consisting of all the hidden units and *only* the visible units corresponding to movies rated by this user. You can see some details on figure 2.

2.2 Preliminary work

We will start implementing a few functions to create the model. These functions will help us run operations that are common when using RBMs. Let $v_i, i \in V$ represent the value held by the i-th node of the visible layer. This value is binary, i.e it is equal to 0 or to 1. We denote a vector of inputs by $\mathbf{v} = (v_1, \ldots, v_n)$. Let $h_j, j \in H$ denote the value held by the j-th node of the hidden layer, also binary. The edge connecting v_i to h_j has a weight $W_{ij} \in \mathbb{R}$.

When the visible layer receives an input, this input is transformed and sent to the hidden layer, in the following way: the probability that h_j equals 1, or that h_j is *on*, given the visible input $\mathbf{v}$, is equal to

$$\mathbb{P}(h_j = 1|\mathbf{v}) = \sigma\Big(\sum_{i \in V} v_i W_{ij}\Big), \text{ where } \sigma(x) = \frac{1}{1+e^{-x}} \ . \tag{1}$$

In words, we do a weighted sum of the inputs and apply the σ function, also called *logistic* function, to obtain the probability. Observe that this functions takes values from 0 to 1 and is highly non-linear.

Question 2.1. *In the `rbm.py` file, complete the functions `sig`. It should accept a vector of inputs and return another vector, where the logistic function has been applied to each component.*

Question 2.2. *Implement the function `visibleToHidden`. It takes an input vector $(v_i)_i$ and computes the vector of probabilities for the hidden layer using equation 1. (Hint: you can use `numpy.dot` to implement a matrix multiplication)*

Hold on, our inputs are not binary! The rating from a user can be one of the five options $K = \{1, 2, 3, 4, 5\}$. One way to circumvent this difficulty is to encode each rating as a binary vector of size 5. All components of the vector are zero except the k-th component, if k is the rating given. So if a user gives a rating 3, that piece of data will be encoded by the vector $(0, 0, 1, 0, 0)$. The formula to go from visible states to the hidden ones is very similar:

$$\mathbb{P}(h_j = 1|\mathbf{v}) = \sigma\Big(\sum_{k \in K} \sum_{i \in V} v_i^k W_{ij}^k\Big) \ . \tag{2}$$

Each user will be associated to one part of the main RBM. Each visible unit in this smaller RBM will correspond to one movie. A visible unit receives a binary vector encoding one rating given by the user. So the RBM receives a list of binary vectors $\mathbf{v}$ to work with. The weights w_{ij}^k is divided to five matrices, one for each $k = 1, 2, 3, 4, 5$. You can represent it easily in Numpy, as a 3D array. Our weights array is of dimensions $|V| \times |H| \times |K|$ (respectively the number of visible units, the number of hidden units and the number of possible ratings). To get the matrix of weights for a particular k, you can simply write `W[:,:,k]`. The first and second axes will correspond respectively to the users and the movies.

Question 2.3. *Modify your `visibleToHidden` function to take into account the binary vectors of ratings, in `visibleToHiddenVec`, using equation 2.*

RBMs are part of a family of generative models. Given inputs, we can use the estimated weights to compute the probabilites that hidden units will activate. But we can also go the other direction: if we know the values of the hidden units, then we can *generate* data. Say we have a vector of hidden units that encodes the preferences of a user that likes action movies, but not sci-fi. We could generate data that looks like the ratings of such a user, for example would assign 5 to *The Bourne Identity* but 2 to *Star Trek*.

How do we go from the hidden layer to the visible one then? The equation governing this transformation is known as *softmax*: we are going to be careful when deciding the ratings of a

user with the current activations of the hidden layer, so we will return a probability distribution over the possible ratings, as such:

$$\mathbb{P}(v_i^k = 1|\mathbf{h}) = \frac{\exp\left(\sum_{j\in H} h_j W_{ij}^k\right)}{\sum_{l=1}^{5} \exp\left(\sum_{j\in H} h_j W_{ij}^l\right)} \quad . \tag{3}$$

Notice that summing over k when i is fixed, the probabilities add up to 1. A rating k will also have more probability assigned to it if $\sum_{j\in H} h_j W_{ij}^k$ is high, so you can see here the binding between predicted rating and the weights we are trying to learn.

Question 2.4. *Implement the* `hiddenToVisible` *function, taking as input a binary vector of hidden units and the edge weights. (Hint: Numpy's* `tensordot` *method can come in handy to perform the* $\sum_{j\in H} h_j W_{ij}^k$ *part)*

2.3 Learning the weights (Optional)

This part is here to help you understand how the learning is done. It is not necessary reading but understanding the process may give you hints regarding how to extend your algorithm to make it perform better!

We move on to the next part of the problem: learning the weights. We use what is called a *gradient ascent* method. Finding the best weights to represent our data can be written as an optimisation problem: find the weights that maximise a certain function. We usually pick the log-likelihood, i.e we seek to find the weights that maximise *the probability that our model would generate the data it has been trained on.* Just like the linear regression case, we can *regularise* the model so that it does not fit too closely the training set and is able to generalise well, but we will not do that here (it is, however, an interesting extension to better your performance).

It would be too long to present every last detail of the gradient ascent, but here is the general idea: we want to move our weights in a direction where this log-likelihood function is greater. We first initialise our weights to some random value (done in `getInitialWeights`) and then perform some updates.

Machine learning is full of tricks to make the gradient ascent more accurate and faster. One of them is the *mini-batch* training. In the linear regression, we have used the full dataset at once to get our estimator. On the other hand, in online training, where we may not have the whole data but a stream that keeps growing, we update our estimator with every incoming piece of data. Mini-batch is between the two: we sample from a big dataset a smaller batch (the size can be tuned) and update our current estimator with this new data.

How do we get the gradient then? This formula is derived from the computed $\mathbb{P}(h_j = 1|\mathbf{v})$ and the data v_i^k itself, by simple multiplication:

$$\nabla W_{i,j}^k = \mathbb{P}(h_j = 1|\mathbf{v}) \cdot v_i^k \quad .$$

But what Hinton and peers found out is that if we only update the gradient this way, we do *too much learning.* So we effectively make the model *unlearn* by doing another pass, a procedure called *contrastive divergence.* Here is the full recipe:

- **Learning**
  - We start with the data $\mathbf{v} = v_i^k$ for a particular user.
  - Compute $\mathbb{P}(h_j = 1|\mathbf{v})$ with `visibleToHiddenVec`.
  - Call positive gradient $PG = \mathbb{P}(h_j = 1|\mathbf{v}) \cdot v_i^k$ (computed by `probProduct`).
- **Unlearning**

- Sample the states of the hidden units according to $\mathbb{P}(h_j = 1|\mathbf{v})$.
- Use `hiddenToVisible` to compute *"negative" data* $\overline{\mathbf{v}}$.
- Essentially repeat the steps of the Learning part, this time with "negative" data, i.e compute $\mathbb{P}(h_j = 1|\overline{\mathbf{v}})$ and call "negative" gradient $NG = \mathbb{P}(h_j = 1|\overline{\mathbf{v}}) \cdot \overline{v}_i^k$.

- **Synthesis** We now update the weights W as such

$$W \leftarrow W + \frac{\epsilon}{\#\text{users}} \cdot (PG - NG) \ .$$

where ϵ is a small learning rate (usually set to 0.1) and we divide by the number of users to obtain an average.

2.4 Predicting the ratings

We sum up the work we have done up until now.

- We have implemented some functions to represent the data (ratings) as binary vectors.
- We then worked on how to go from the visible layer to the hidden one and vice-versa.
- We implemented a simple gradient descent algorithm to learn the best estimator of the weights.

We have everything we need to perform the prediction! In fact, we have done most of the hard work required to implement this part. Assume we now would like to predict a movie q for user u. We can get the binary matrix of visible inputs $\mathbf{v}$ for that user and use our `visibleToHiddenVec` function to propagate the values to the hidden units. Once we have these hidden activations, we can go in the reverse direction, just like we do in the unlearning part of the gradient ascent algorithm. We take the vector p of hidden activations and use our `hiddenToVisible` function to get the "negative data". In fact this negative data can be interpreted as a prediction over *all* the movies in the dataset: for each movie i, this negative data gives us a distribution $\overline{v_i}$ over all the ratings, and in particular one distribution for movie q.

Question 2.5. *Implement the function `getPredictedDistribution` to get the predicted distribution over the ratings for movie q.*

We are now given two options to decide the predicted rating we want to return:

- **Predict the rating with the highest score** We could simply return the rating that is the most weighted by the distribution $\overline{v_q}$.
- **Predict the rating with the expectation** We could also return the expected value of the distribution $\overline{v_q}$ as the predicted rating.

As an example, suppose we have $\overline{v_q} = (0.1, 0.2, 0.4, 0.3, 0.0)$, i.e the score of rating 1 is 0.1, the score of rating 2 is 0.2 etc. Our first method would return 3 as predicted rating, since 3 has the highest score. Our second method would return

$$0.1 \times 1 + 0.2 \times 2 + 0.4 \times 3 + 0.3 \times 4 + 0.0 \times 5 = 2.9$$

Question 2.6. *Implement the two functions `predictRatingMax` and `predictRatingExp`, described above.*

You can test which of the two functions give you the better RMSE.

2.5 Some extensions

To make your algorithm better, we encourage you to tweak it using the tips we give below. They are not explicitly detailed but we add a number of stars next to indicate the difficulty of implementing them, (*) being easier and (***) harder. You may find additional resources online to help you with the implementation. This implementation can only add some extra points to your grade,but it is not required.

- **Momentum (*)** An easy addition to the code, momentum will give some inertia to the gradient updates, limiting the risk that your gradient starts oscillating.
- **Adaptive learning rates (**)** This method progressively reduces the learning rate, as your algorithm zeroes in on the best estimator of the weights.
- **Early stopping (*)** With the momentum and other tricks, your RMSE is susceptible to increase from one iteration to the next. Early stopping simply means keeping in memory the best weights found so far, e.g if your RMSE at epoch 950 is better than the one at the end of your training, you would use the weights obtained at epoch 950.
- **Mini batch (**)** In the linear regression, we have used the full dataset at once to get our estimator. On the other hand, in online training, where we may not have the whole data but a stream that keeps growing, we update our estimator with every incoming piece of data. Mini-batch is between the two: we sample from a big dataset a smaller batch (the size can be tuned) and update our current estimator with this new data. In practice, you will compute the gradient for a few users, add these up and *then* update your weights with this computed gradient.
- **Biases (***)** The method described above is not exactly complete. We would usually add biases to the visible and hidden units, and estimate their value just like we would for the weights, using learning. The model with biases is described in the Salakhutdinov et al. paper, and parts of it are implemented in the associated code[1], though you will need to adapt it for the case of visible units receiving binary vectors as inputs. This is the hardest extension but an essential one to give your algorithm a performance boost.

3 Testing out your algorithms

After finishing up the two previous sections, you should have two algorithms to predict the rating that a user would give to unrated movies. One makes use of linear regression to compute biases per user and per movie, while the other is adapted from neural networks and uses latent factors to predict the rating. Both algorithms can be tuned to work better by using *cross-validation*.

The implementation of the cross-validation is done in the `splitDataset` function of the `projectLib` file. You then train your model on the training set and predict the ratings of the validation set and compare them using the RMSE.

Why would you want to do this? In both methods, you have *hyperparameters* that you can change to make the algorithm more performant. Let us recapitulate them here:

- **Linear regression**
  - **The regularisation parameter** λ This parameter controls the penalty given to weights in the objective function. A higher λ will penalise large biases, making your algorithm less likely to overfit (but with the cost of higher variance).

[1] Paper and code can be found in the project folder, as `hintonrbm.py` and `hintonrbm.py`

- **Restricted Boltzmann Machines**
  - **The number of hidden units** F This parameter controls how many hidden units appear in your main RBM. The higher it is, the more likely it is that your model with overfit.
  - **The learning rate** ϵ Gradient ascent needs a learning rate controlling how much the gradient is allowed to change from one iteration to the next. A higher learning rate will make your algorithm converge faster to a better value, but a value that is too high can also derail it and provoke oscillations. Here is an idea on how to tune it: at the end of every batch, when you update your weights, compute the RMSE on the training set. If it is steadily decreasing, you are on the right track and maybe you can increase your learning rate a bit. If it seems to oscillate, or increases, you have to decrease your learning rate.

The problem is that if you train your model to be extremely performant (i.e give very good predictions) on the training set itself, you run the risk of *overfitting*. Overfitting happens when your model estimates very precisely ratings from your training set, but is incapable of generalising to a different dataset. By computing the RMSE on the validation set, you make predictions over data that your model *has not seen*, and thus you are able to control whether your model overfits. Overfitting typically happens when at first, both the training RMSE and the validation RMSE decrease, and after a while your validation RMSE starts increasing again, while training RMSE keeps going down. We are logging both values every 100 iterations of the gradient ascent for you to control, but you could go beyond and plot these curves to have a visual idea of what is going on.

Coming back to the hyperparameters, you should then be tuning their values such that your model avoids both underfitting and overfitting. Underfitting is harder to control for, but happens when your model is not trained enough. If at the end of your training, both validation and training RMSEs keep decreasing consistently, you may want to increase your number of epochs or, e.g, increase your number of hidden units.

3.1 Apply your model on the test data

Perhaps the most important part of the project: we are keeping from you a test set where real ratings appears. These ratings are taken from the same set of users and movies you have trained your model on, so you do not need to worry about how to treat these new cases. Once you believe that your hyperparameters have been optimised correctly (i.e, after repeated testing against a validation set), you will be able to try out your model on the hidden test.

References

[1] Introduction to restricted boltzmann machines. `http://blog.echen.me/2011/07/18/introduction-to-restricted-boltzmann-machines/`. Accessed: 2017-01-17.

[2] G. Louppe. *Collaborative filtering: Scalable approaches using restricted Boltzmann machines*. PhD thesis, Université de Liège, Belgique, 2010.

[3] R. Salakhutdinov, A. Mnih, and G. Hinton. Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*, pages 791–798. ACM, 2007.