Unsupervised learning

Olga Lyudchik

Non-member state summer student

Supervisor: Dr. Jean-Roch Valery VLIMANT (EP-UCM)

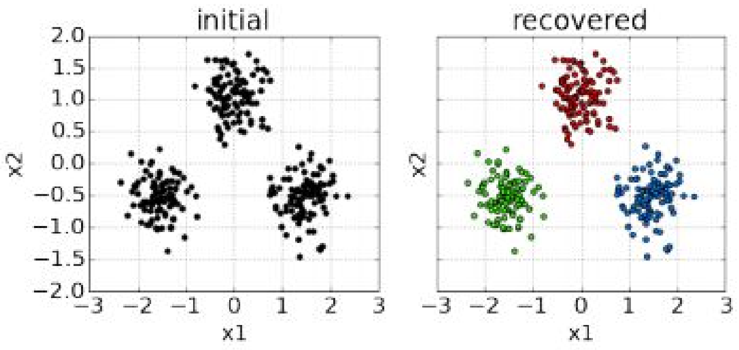
Contents list

General idea

Find functional relationship between input variables x and output variables y based on expert knowledge and only x observations:

$$X_1, X_2, , X_N$$

Unsupervised learning is also known as clustering (for discrete output).



Unsupervised learning methods

K-means

K-means is one of the simplest unsupervised learning algorithms that solve the well known clustering problem. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed a priori. The main idea is to define k centroids, one for each cluster. These centroids should be placed in a cunning way because different location causes different result. So, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is pending, the first step is completed and an early groupage is done. At this point we need to re-calculate k new centroids as barycenters of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new centroid. A loop has been generated and because of this loop, we may notice that the k centroids change their location step by step until no more changes are done. In other words, centroids do not move any more.
Finally, this algorithm aims at minimizing an *objective function*, in this case a squared error function. The objective function

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \|x_i^{(j)} - c_j\|^2,$$

where $\|x_i^{(j)} - c_j\|^2$ is a chosen distance measure between a data point $x_i^{(j)}$ and the cluster center c_j, is an indicator of the distance of the n data points from their respective cluster centers.

The algorithm is composed of the following steps:

- Input: K, set of points $x_1, \dots x_N$
- Place centroids $c_1, \dots c_K$ at random locations

- Repeat until convergence:

  For each point x_I:

  - Find nearest centroid c_j *arg min* $D(x_i, c_j)$ - distance (e.g. Euclidean) between instance x_i and cluster center c_j
  - Assign the point x_i to cluster j

  For each cluster $j = 1 \dots K$:

  New centroid c_j = mean of all points x_i assigned to cluster j in previous step

$$c_j(a) = \frac{1}{n_j} \sum_{x_i \to Cj} x_i(a) \ \ for\ a = 1, \dots d$$

- Stop when no points change cluster memberships

Advantages

1) If variables are huge, then K-Means most of the times computationally faster than hierarchical clustering, if we keep k smalls.
2) K-Means produce tighter clusters than hierarchical clustering, especially if the clusters are globular.

Disadvantages

1) Difficult to predict K-Value.
2) With global cluster, it doesn't work well.
3) Different initial partitions can result in different final clusters.
4) It does not work well with clusters (in the original data) of different size and different density.

Some examples of implementation K-Means can be found on that link.

K-means kernel

This algorithm applies the same trick as k-means but with one difference that here in the calculation of distance, kernel method is used instead of the Euclidean distance. The main idea is that map data points in the input space onto a high-dimensional feature space using the kernel function. Then perform K-means on the mapped feature space.

For performing kernel k-means we can use the following kernel functions:

- Polynomial kernel of degree h: $K(x_i, x_j) = (x_i*x_j + 1)^h$
- Gaussian kernel: $K(x_i, x_j) = e^{-||xi - xj||2/2\sigma 2}$
- Sigmoid kernel: $K(x_i, x_j) = \tanh(\kappa x_i*x_j - \delta)$

The formula for kernel matrix K for any two points x_i, $x_j \epsilon C_k$: $K_{xi,\, xj} = \varphi(x_i) * \varphi(x_j)$

The SSE criterion: $\sum_{k=1}^{K} \sum_{x_i \in C_k} ||\varphi(x_i) - c_k||2$

The formula for cluster centroid: $ck = \frac{\sum_{xi\, \epsilon\, Ck} \varphi(xi)}{|C_k|}$

Advantages

1) It is able to identify the non-linear structures.
2) Algorithm is best suited for real life data set.

Disadvantages

1) Number of cluster centers need to be predefined.
2) Algorithm is complex in nature and time complexity is large.

The opensource code is available on that link.

Self-organizing map (Kohonen)

(SOMs) are a data visualization technique, which reduce the dimensions of data using self-organizing neural networks. The way SOMs go about reducing dimensions is by producing a map of usually 1 or 2 dimensions which plot the similarities of the data by grouping similar data items together. So SOMs accomplish two things, they reduce dimensions and display similarities. The way that SOMs go about organizing themselves is by competing for representation of the samples. Neurons are also allowed to change themselves by learning to become more like samples in hopes of winning the next competition. This selection and learning process makes the weights organize themselves into a map representing similarities. So with these two components (the sample and weight vectors), how can one order the weight vectors in such a way that they will represent the similarities of the sample vectors? This is accomplished by using the very simple algorithm shown below:

- Initialize the weight vector map
  Note: unfortunately calculating SOMs is very computationally expensive, so there are some variants of initializing the weights so that samples that you know for a fact are not similar start off far away. This way you need less iterations to produce a good map and can save yourself some time.

- For t from 0 to number of cycles
  - Randomly select a sample.
  - Get best matching unit.

    Just go through all the weight vectors and calculate the distance from each weight to the chosen sample vector. The weight with the shortest distance is the winner.
  - Scale neighbors.

There are two parts to scaling the neighboring weights: determining which weights are considered as neighbors and how much each weight can become more like the sample vector.

$W_v(t + 1) = W_v(t) + \Theta(t)\alpha(t)(D(t) - W_v(t))$.

- Increase t a small amount

- End for loop

Advantages

1) Very easy to understand, you can quickly pick up on how to use it in an effective manner.
2) SOMs classify data well and then are easily evaluate for their own quality so you can actually calculated how good a map is and how strong the similarities between objects are.

Disadvantages

1) Difficult to get the right data as you need a value for each dimension of each member of samples in order to generate a map.
2) Every SOM is different and finds different similarities among the sample vectors. SOMs organize sample data so that in the final product, the samples are usually surrounded by similar samples, however similar samples are not always near each other. That means that many maps need to be constructed in order to get one final good map.
3) Very computationally expensive which is a major drawback since as the dimensions of the data increases, dimension reduction visualization techniques become more important, but unfortunately then time to compute them also increases.

Libraries for implementation of SOM are available on the link.

Basic autoencoder

The traditional autoencoder is an artificial neural network that attempts to reproduce its input, i.e., the target output is the input. More formally an autoencoder takes an input vector $x \in [0, 1]^d$ and maps it to a hidden representation $y \in [0, 1]^d$ through a deterministic mapping $y = h_\theta(x) = s(Wx + b)$, parameterized by $\theta = \{W, b\}$.

W is a $d' \times d$ weight matrix, b is a bias vector and s is the sigmoid[3] activation function, $s(x) = 1/(1+e^{-x})$. The hidden representation y, sometimes called the *latent* representation, is then mapped back to a reconstructed vector $z \in [0, 1]^d$, where $z = g_{\theta'}(y) = s(W'y + b')$, with $\theta' = \{W', b'\}$.

The basic idea here is that the autoencoder is constructed in such a way that the mapping $x^{(i)} \rightarrow y^{(i)}$ reveals essential structure in the input vector $x^{(i)}$ that is not otherwise obvious.
The parameters θ and θ' of the model are optimized to minimize the average reconstruction error as shown in the following equation:

$$\theta, \theta' = arg\ min\ \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, z^{(i)}) = arg\ min \sum_{i=1}^{n} L(x^{(i)}, g_{\theta'}(f_\theta(x^{(i)}))$$

Here L is a loss function such as the traditional squared error $L(x, z) = ||x - z|^2|$.

Advantages

Defines a simple, tractable optimization objective that can be used to monitor progress.

Disadvantages

Can easily memorize the training data - i.e. find the model parameters that map every input seen to a perfect reconstruction with zero error - given enough hidden units h.

COREX

Correlation Explanation (CorEx) is an information-theoretic method for discovering a hierarchy of abstract representations for complex data. This representation is optimized to be maximally informative about the data.

The algorithm can be seen below:

Input : A matrix of size n_s x n representing n_s samples of n discrete random variables

Set : Set m, the number of latent variables, Y_j , and k, so that $|Y_j| = k$

Output: Parameters $a_{i,j}$, $p(y_j|x_i)$, $p(y_j)$, $p(y|x^{(l)})$ *for* $i \in N_n$, $j \in N_m$, $l \in N_{ns}$, $y \in N_k$, $x_i \in X_i$

Randomly initialize $a_{i,j}$, $p(y|x^{(l)})$;

Repeat

- Estimate marginals, $p(y_j)$, $p(y_j\ |x_i)$ using the following eq.:

$$p(y_j|x_i) = \frac{\sum_{\bar{x}} p(y_j|\bar{x})\, p(\bar{x}) \delta_{\bar{x}_i, x_i}}{p(x_i)}\ and\ p(x_i) = \sum_{\bar{x}} p(y_j|\bar{x})\, p(\bar{x})$$

- Calculate $I(X_i : Y_j)$ from marginals;
- Update a using the following eq.:

$$\alpha_{i,j}^{t+1} = (1 - \lambda)\alpha_{i,j}^{t} + \lambda \alpha_{i,j}^{**}$$

- Calculate $p(y|x^{(l)})$, $l = 1,...,n_s$ using the following eq.:

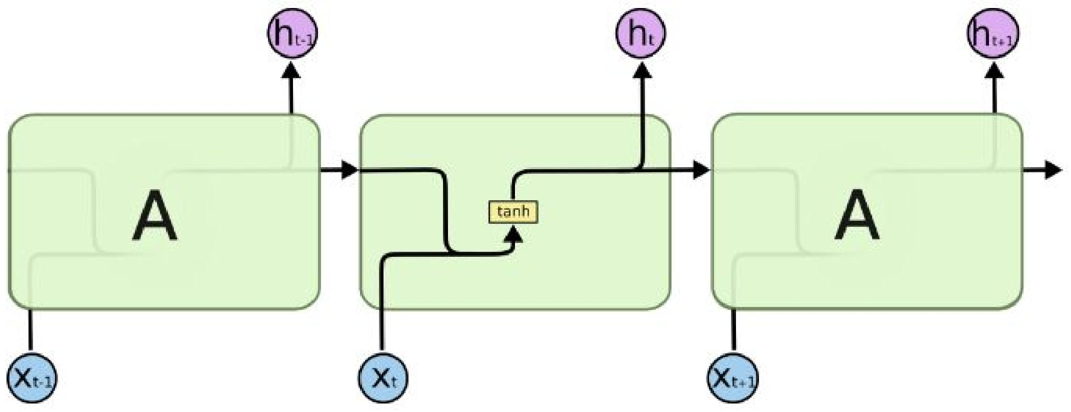
$$p(y_j|x_i) = \frac{1}{Z_j(x)} p(y_j) \prod_{i=1}^{n} (\frac{p(y_j|x_i)}{p(y_j)})^{\alpha_{i,j}}$$

until convergence;

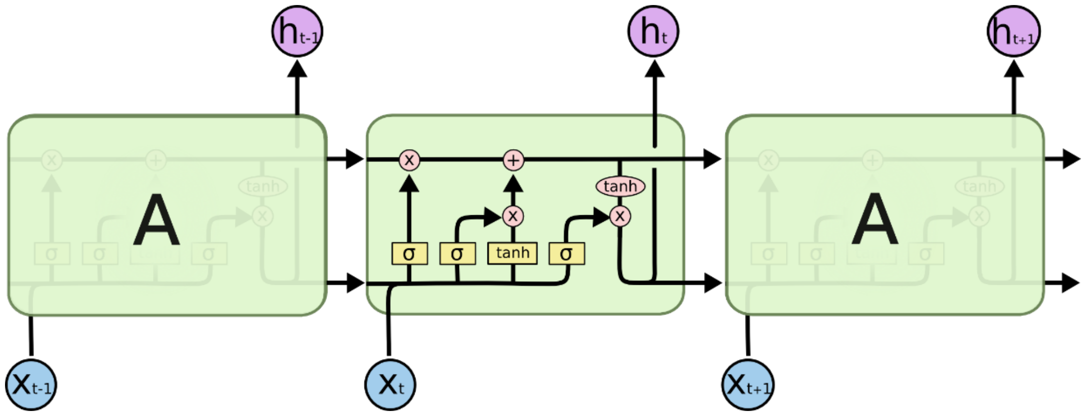
The opensource code for COREX can be founв on the link.

LSTM Networks

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They work tremendously well on a large variety of problems, and are now widely used. LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods is practically their default behavior, not something they struggle to learn! All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer:



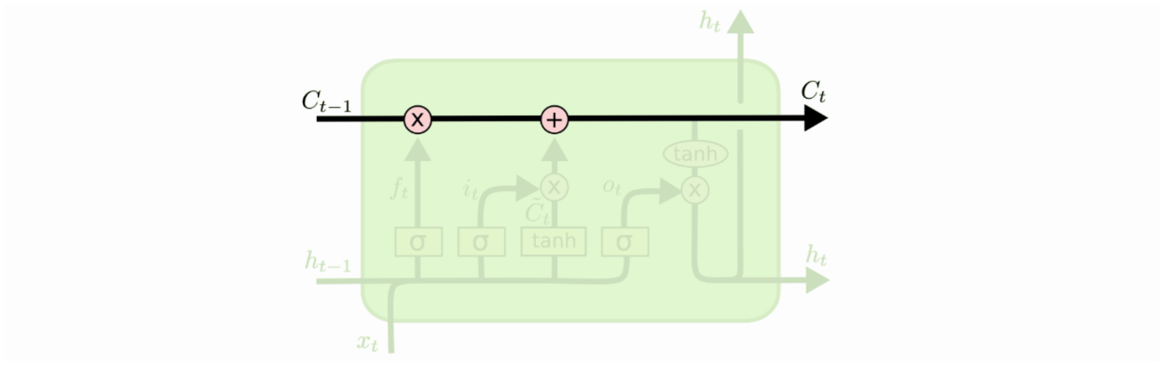
LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.

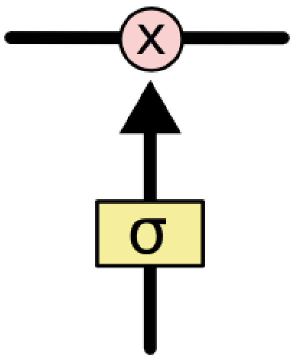
The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It’s very easy for information to just flow along it unchanged.



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

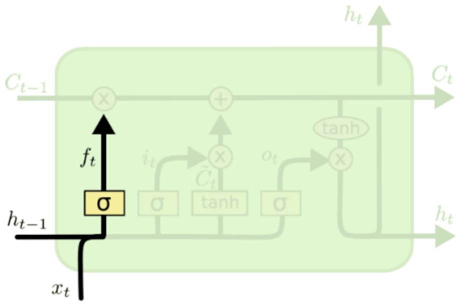
Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”

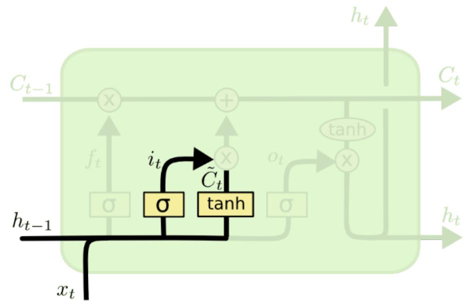
An LSTM has three of these gates, to protect and control the cell state.

The first step in our LSTM is to decide what information we’re going to throw away from the cell state. This decision is made by a sigmoid layer called the “forget gate layer.” It looks at ht−1ht−1and xtxt, and outputs a number between 00 and 11 for each number in the cell state Ct−1Ct−1. A 11 represents “completely keep this” while a 00 represents “completely get rid of this.”



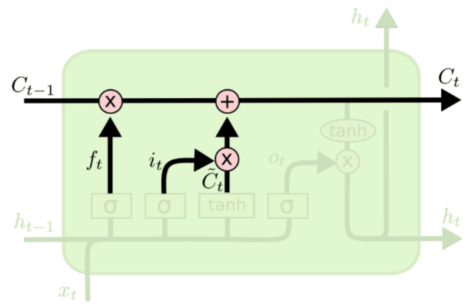
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

The next step is to decide what new information we’re going to store in the cell state. This has two parts. First, a sigmoid layer called the “input gate layer” decides which values we’ll update. Next, a tanh layer creates a vector of new candidate values, C~tC~t, that could be added to the state. In the next step, we’ll combine these two to create an update to the state.



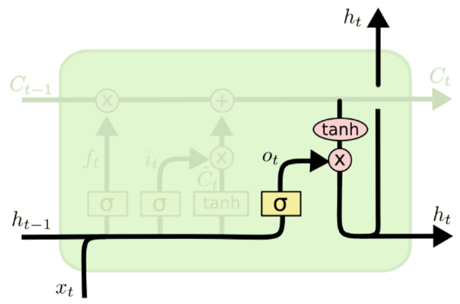
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

It’s now time to update the old cell state, Ct−1Ct−1, into the new cell state CtCt. The previous steps already decided what to do, we just need to actually do it.
We multiply the old state by ftft, forgetting the things we decided to forget earlier. Then we add it*C~tit*C~t. This is the new candidate values, scaled by how much we decided to update each state value.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Finally, we need to decide what we’re going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we’re going to output. Then, we put the cell state through tanh (to push the values to be between −1−1 and 11) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.



$$o_t = \sigma\left(W_o \ [h_{t-1}, x_t] \ + \ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Advantages

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods is practically their default behavior, not something they struggle to learn.

Disadvantages

It is slower than other normal activation functions, such as sigmoid, tanh or rectified linear unit.

Example of implementation LSTM-network in python is available via the link.