White Paper

Failure prediction

Abstract

Throughout the whole industry many systems relies on clouds, network, Datacenter and software components stabilities. Thus, each application has to be available most of the time. In addition, the IT governance model pressure IT teams to take strong commitments (KPI) of accountability to address the outages as fast as possible. This work is about a learning algorithm whose goal is to give a hint on when an unavailability is going to occur. In cooperation with monitoring systems used to collect the data on applications availability, like a nagios based softwares, the algorithm's aims to process data by gathering few different classical techniques, such as neural network genetic algorithm. This method is however somehow based on empirical thoughts and target at guessing a failure before it happens.

1 Introduction

Contrated applications runs within a maximum amount of time of unavailability per day. Thus when an incident occurs a race against time begins. The issue has to be diagnosed and fixed as fast as possible because this time is taken on the whole unavailability time. This amount is one of the most important key performance indicator (KPI) of an service leval aggreement (SLA) contracted by indrustry class IT services. We try to find a solution that could help anticipating these problems so the resolution time is the shorter possible.
When in 2017 we tried to find a solution to the problem we decided to use the log files of Nagios based monitoring system we had access to. Our thougth was that we could establish a learning process using only the nagios disponibility data (which is described in the first part) to predict any failure of the application. The whole initial solution is explained in details throughout the first part but is build upon the assumption that the changes in the structure of the application are scarce which was a coherent assumption by this time.
However technologies have evolved and applications may have small changes in their structure on a regular basis which make any form of learning pretty difficult since the training is made with a specific structure of the application. This change in architecture are typically those expected with new kind of infrastructure like, as an example, the orchestrated containerized applications. This is why in a second part we'll focus on how to target specific hosts, those with the most influence on the application's state, so we can apply our previous learning method on a subset of all the hosts in the application that won't change too much. Finally we'll also try to regroup some of the hosts together in such a way that some may be added or remove only the group while be taken into account and the learning won't be affected by the structure changes.

2 Availability based method

This first method uses nagios disponibility data, which means we have access, for each host and service of the application, only to the disponibility score between 0 and 3 (0: OK, 1: Warning, 2: Critical, 3: Unknown) and not to the exact value of each parameter. We are going to use historical data from old logs to train an algorithm of machine learning combining two learning techniques which are neural networs and the principle of genetic algorithms that we'll then use on the live data to predict the outage of the application that will eventually occur. However as explain in the introduction this method has serious rigidity issues as the training is made on a specific application's structure we'll discuss this limit as a conclusion to this method.

2.1 Application model

The model for the application we chose is a tree of dependencies. The application is the root of the tree, the internal nodes are the hosts and the leaves are the services. To each node is associated a relationship to its children that indicates wether all of them are required for the availability of this node or some can be down without making this node unavailable. These relationships are only the known ones, which means that they are explicit, they are obvious looking at the design of the application. We may have relationships that are implied by the interaction of the internal nodes that occurs in a real life system without us being aware of it, but they are not obvious during the creation process of the application. (see Figure 1 below)

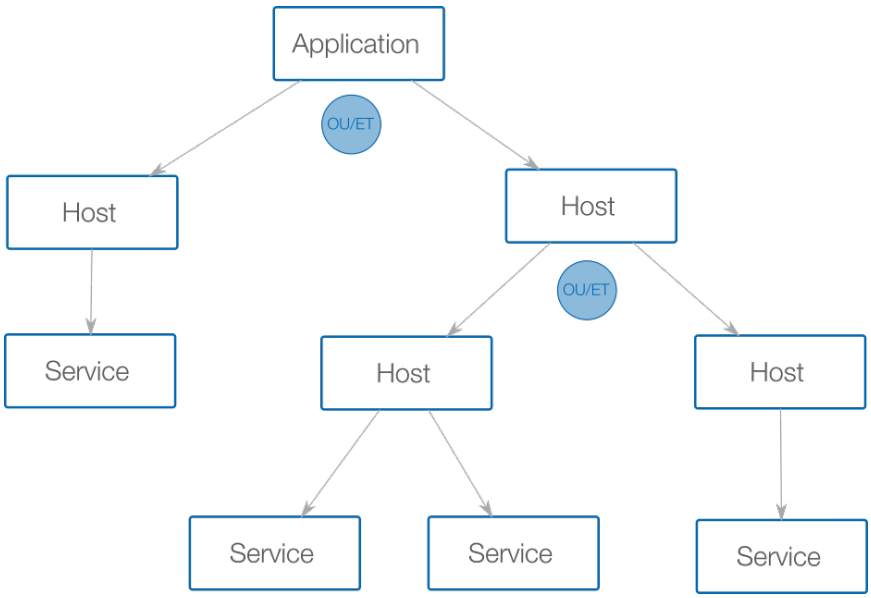

Figure 1: Tree example

We also define an image as the state of the whole application i.e. the state of all of its components (hosts and services) at a point in time. This is represented as an array where there are 4 rows, one for each state (0-3) and as many column as hosts and services. The state of an host or a service is given by a boolean in the array (true in the row that is the corresponding state and false everywhere else) illustrated on Figure 2

	host	host	serv	host	serv	serv	serv
0	1	0	1	1	0	1	1
1	0	0	0	0	0	0	0
2	0	0	0	0	1	0	0
3	0	1	0	0	0	0	0

Figure 2: General image

2.2 Processing data

We are going to use supervised machine learning therefore we need to label the historical data we'll use to train our neural network. The neural network will have to guess the amount of epoch remaining until the next outage hence we have to extract from the log all the epochs where the application was unaivalable. This can be achieved with a request to the database in which are stored all the dates of beginning and ending of an outage and then by taging the images whose epoch are in between a start and an end. We also need to associate with each image the time remaining before the next outage. This can be done easily using what we calculated before.

We now compute a mean of all the image immediately preceding an outage. Here we made an assumption, we assumed that enough information to feed the neural network and have a raisonnably accurate guess would be stored in the comparison between an image and the mean we calculated. We are going to call this mean image the reference.

The comparison between an image and the reference is made by using a simple coordinate by coordinate distance i.e. noting a_i the coordinates of the image and

r_i the ones of the reference we use, for a given $k \in \mathbb{N}$, the following formula:

$$d(a,r) = \sqrt[k]{\sum \mid a_i - r_i \mid^k}$$

2.3 Neural networks

The first learning of the algorithm is via neural networks. We are going to train a neural network to guess how much time is remaining until the next outage. A network with a given structure is the set of its weights and biases which are the parameters we'll try to evolve during the learning process and they are going to be randomly initialized, this is why, before the training, our network can be summarize to its structure. So we'll represent a network by two list of integers. One that will specify the input of the neural network and another for its hidden layers (we only need one output : the guess). The first list length is the total of inputs of the network and each integer in this list will be used as the k in the distance formula thus each input will take a different value of a different comparison between the image and the reference. The second list length is the total number of hidden layer and the $i-th$ integer in the list is the amount of neurones in the layer i.

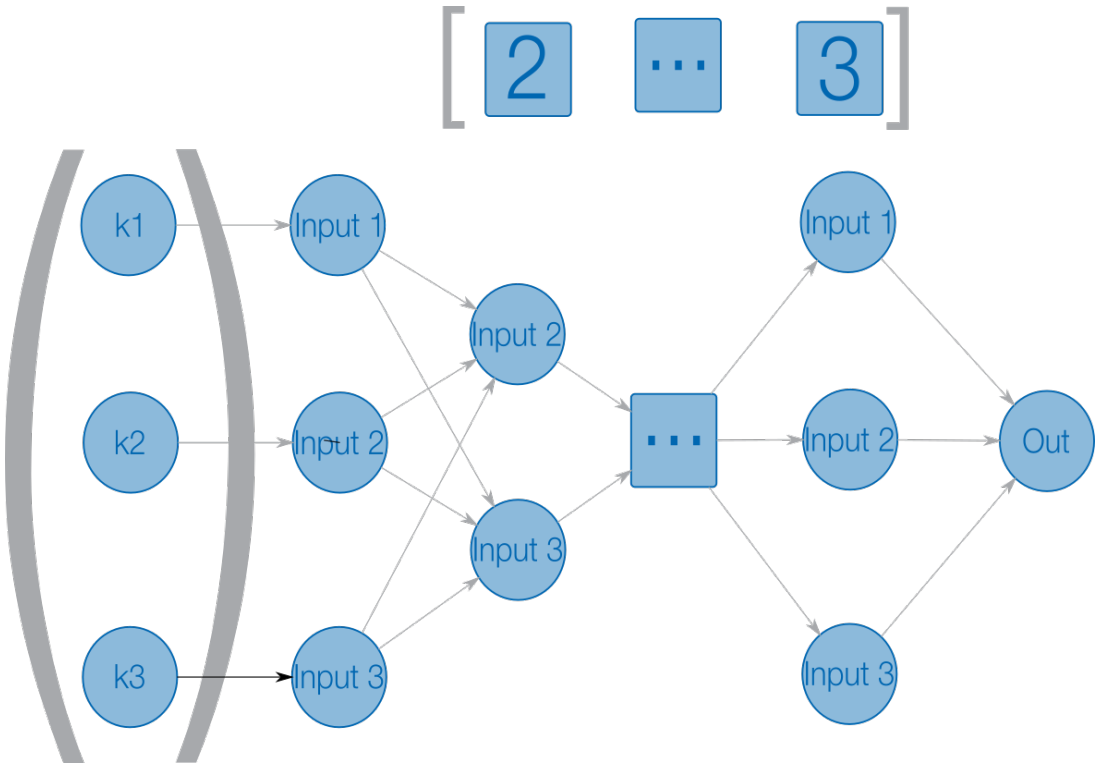

Figure 3: Network structure

Once the network is defined we start the learning phase. We use our labelled data to train the network to guess the time until the next outage with the back-propagation algorithm. This classical algorithm do the computation with the labelled data, evaluate the cost i.e. the error made by the network between the result just obtained and the label associated with the data, this cost is then used to modify the weights and biases of the network. The formula to reevaluate the weights w^l and biases b^l is (for the layer l):

$$w^l \leftarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l}\, {}^t(a^{x,l-1})$$

$$b^l \leftarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$$

Where η is the learning rate, which means a coeffi-

cient to control the speed of the changes made upon the weights and biases, m is the number of element used to evaluate the changes that have to be made, δ is the error and x is a piece of training data. However several problems appears here. First when we train train the network we are minimizing a function of its inputs which is not convex so it has not necessarily only one global minimum (we can for instance switch two neurones on the same layer) thus it could converge only to a local minimum. The second problem is that we have to determine a good set of hyperparameters i.e. the structure lists of the network, the learning rate and the size of the training data.

2.4 Evolving hyperparameters

The solution we proposed is a solution for both problem. It consists in applying the principle of genetic algorithm to our neural networks. After every application outage we'll evaluate the performance of each network with a fitness function that takes into account the difference between the guess and the reality and its score on an evaluation set of historical data randomly selectionned. So with the distance $x \in \mathbb{R}^{+*}$ and the score $y \in [0, 1]$ we have the function fitness f arbitrarily defined by :

$$f(x, y) = y^{\frac{1}{50} + \frac{1}{1+e^{\frac{1}{1+x}}}}$$

This function has only been chosen for its shape illustrated on Figure 4

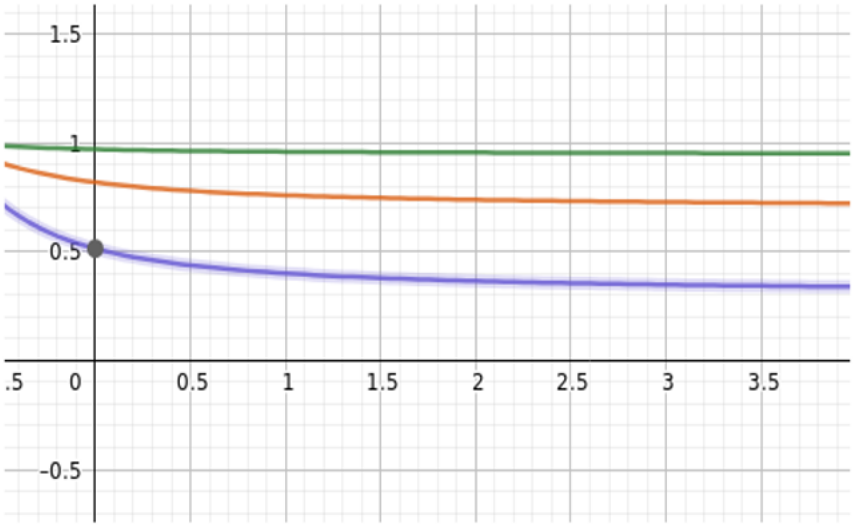

Figure 4: Fitness evaluation for several values of y

At the beginning we create a population of multiple neural networks that we train separatly from different starting parameters (structure, weights and biases), during the analysis we can use the computation of all these networks and because they started their training from different configurations they propably didn't converge to the same minimum of the function, thus we increase the probability of getting close to a global minimum.
We use all the guesses from all of the networks in the population to get the amount of time we are trying to predict. Then after the application outage we apply the fitness function we introduced above to evaluate individualy the networks (Figure 6) and we generate the next generation using the ones with best results. To do the selction we normalize all the score so that the total sum is 1 and we randomly select some networks, the higher their score the higher the probability of being selected.

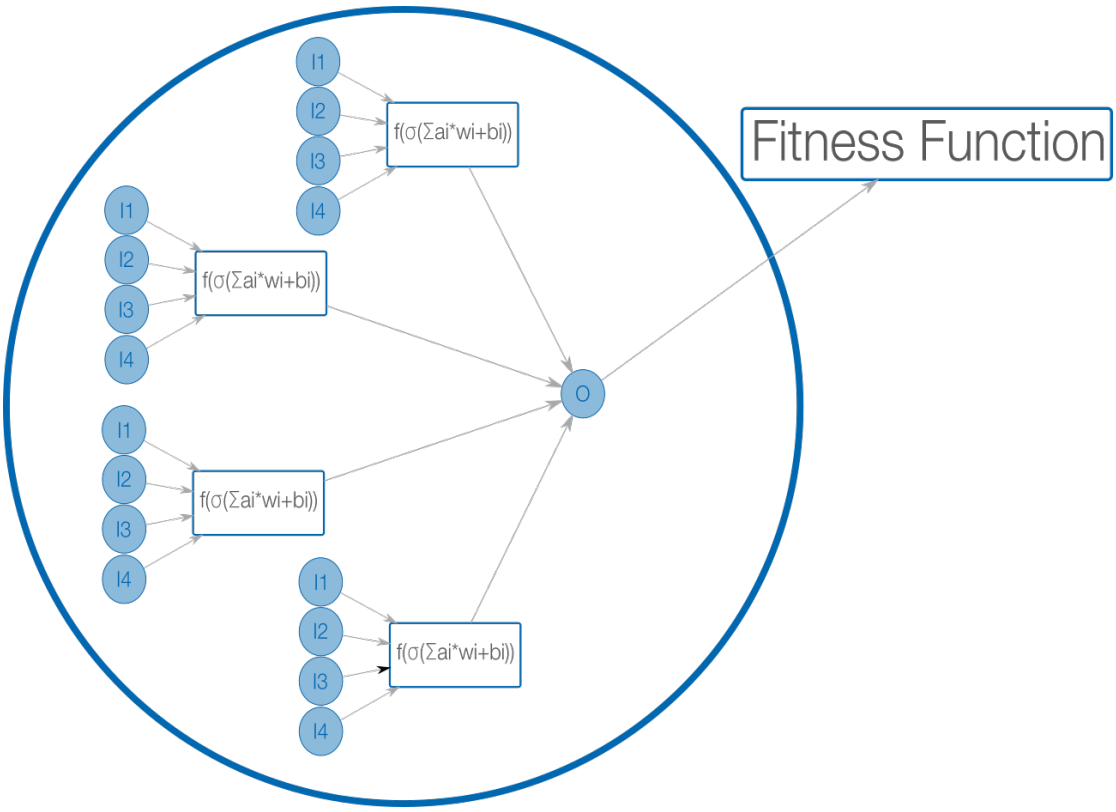

Figure 5: A population of network

In the process we also add little mutations to the hyperparameters so that we have less chance of getting stuck in a non-optimal configuration. This method also solved the problem of choosing the set of hyperparameters because they are evolving throughout the execution of the algortihm. With two given network another is created by generating an empty list we fill with elements of a list or the other with probability $\frac{1}{2}$. Then with a little probability called *mutation rate* we introduce mutations which can be either a little variation of an integer in the lists or of the size of each list.

2.5 Conclusion and limits

This method can be pretty efficient because we can also collect data from past runs to determine the best structure to start the genetic principle with, which could be seen as another form of learning. However eventhough some part of it could be useful the full method is not feasible because it's unadapted to the new standard architectures of application. Indeed the whole process lies on the training of neural networks that was made using a specific architecture of the application and it shall not evolve during the execution of the algorithm for two big reasons. The first is that the learning would be, in the best case, far less efficient and the second problem is that the comparison is made coordinate by coordinate with an image calculated from historical data. Thus we couldn't compare new images with the old mean and the whole algorithm falls apart.
Still this indicates a path to follow. We should find a way to identify which variables of the application are the most impactful. This is an interesting question because identifying these variable would not only permit to restrain the number of data used to feed the neural network, making possible to add new ones without directly breaking the whole training because if the newly introduced variable is important the training could be done later without preventing the comparison. Also this would be a relevant information to help the diagnosis of an application. This is why the following part will focus on the identification of these variables. With this in mind analazing raw data will be more relevant hence we wont use the nagios data but directly values obtained from sensors. Therefore some new method of learning will eventually be explored.

3 Raw data enhancements

We extend here our scope of possibility to all the types of data because we need to solve the rigidity problem of the previous method. First we'll try to find a way of selecting the most influential hosts in the application i.e. the hosts for whom a variation of behavior has an important impact on the whole application. We'll then find a way to organize hosts into groups with common behaviors so that we can analyze the mean of the total group and then we'll be able to add and remove hosts from these group without losing the learning, it's also allowing more data to be used during the training of the algorithm.

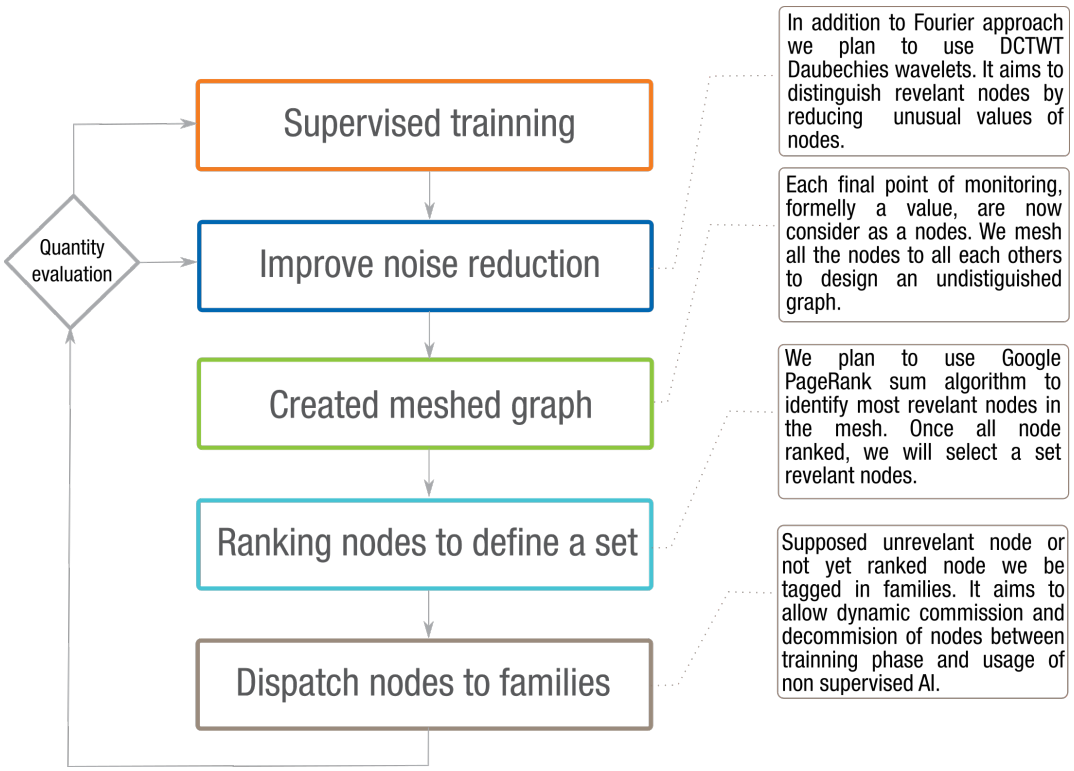

Figure 6: Training phasing

3.1 Feature selection

3.1.1 Signal Processing

The signal we get from the historical data is full of small variation that may be due to measure errors or changes that are periodical parasite which are irrelevant to treat. Therefore we will try to eliminate as much noise as possible to get a better accuracy in the computation of the influence. Indeed we are going to use a correlation between signals to guess the influence of a variable over another so we have to take into account only the significant event.Furthermore we have to find a proper way of evaluating the influence a node has over another.
In order to reduce the noise in our signal we'll use some wavelets theory. The choosen method is described in described in detail in [?] Basically we transform the data in wavelet domain with DTCWT - which can be done with the dtcwt library in python for instance (see example code from the documentation below) - we then do the processing, it consists in shrinking the noisy coefficient and finally we recover the de-noised signal by doing the inverse transformation.

```
from matplotlib.pylab import *
import dtcwt


# 1D transform, 5 levels
transform = dtcwt.Transform1d()
vecs_t = transform.forward(vecs, nlevels=5)

# Inverse
vecs_recon = transform.inverse(vecs_t)
```

Listing 1: Python dtcwt library example

The transformation is a decomposition of the function into a sum of some coefficient multiplied by each vector of a family of wavelets who have specific properties (the theory is explained in detail in the paper). in this work we'll use the Daubechies' wavelets (db11) which was chosen using [?]

3.1.2 Influence evaluation

In this part it is necessary to remember that what we are doing is done before the execution of the application during a learning phase, therefore we don't have any constraints in time and the algorithm can be slow. Our goal is to implement a way to evaluate the influ-

ence of each host and service on each other, we can visualize this idea as a graph where the nodes are the hosts and services and the edges are oriented. The edge (s_i, s_j) is linked to the value of the influence s_j has over s_i. As on Figure 7

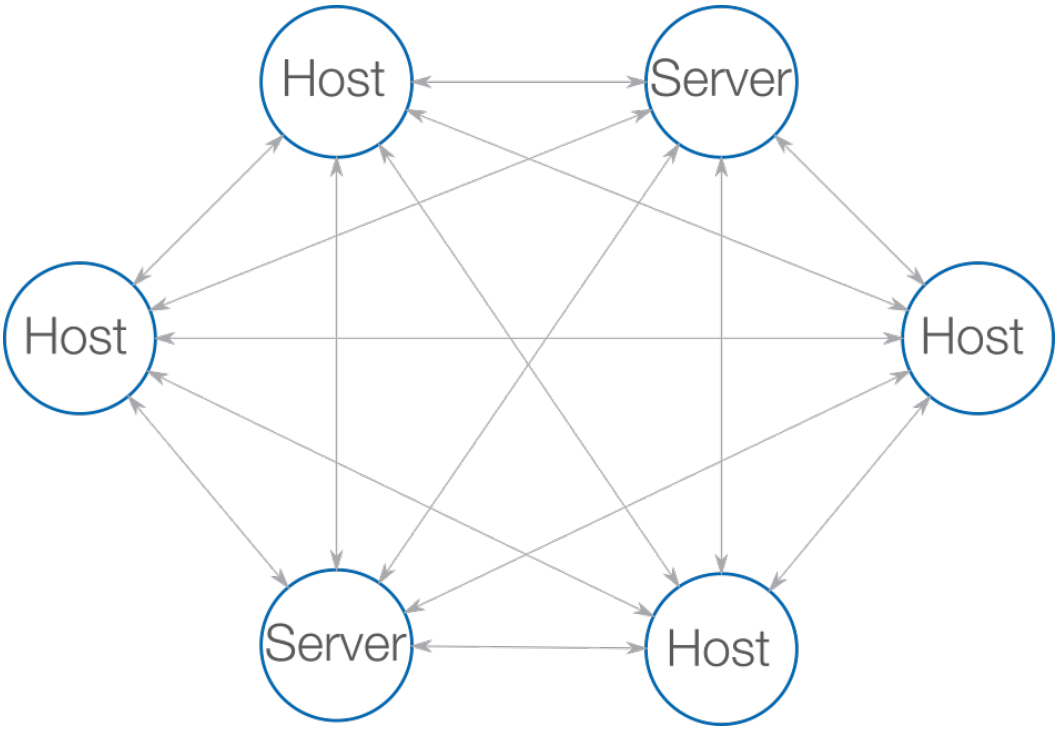

Figure 7: Graph example

The influence is assumed to be in our case the information flow between s_i and s_j so using the Liang-Kleeman information flow [?] on some data we have, we can compute the value of the edges of the graph.

3.1.3 Targeting influential hosts

We now want to sort the nodes by the influence they have in the graph, we'll use the principle of the *page rank* algorithm [?] on the graph where the probability to follow an edge is given by it's value of influence. this is however slightly different than the classical PageRank algorithm because edges have weights. Intuitively it can be seen as a random walk throughout the graph with the probability to follow an edge proportionnal to its weight. To compute it however, we'll use the power method witch is a simple and fast way to compute it. Letting M the matrix of transition between the nodes, we obtain it by normalizing the scores of the edges coming out of a node and using them as probability. We compute the rank vector X by starting with an arbitrary X_0 and then using :

$$X_{t+1} = (dM + \frac{1-d}{N} J_N) X_t$$

until we reach :

$$| X_t + 1 - X_t | < \epsilon$$

where ϵ is an arbitrarly small parameter, N is the total number of nodes in the graph i.e. the number of hosts and services in the application, J_N is the square matrix of size N containing only ones and finally d is a parameter which represent the probability of stoping the random walk set at 0.85 because it as been shown by experimentation done by Google that it is the optimal value to have both a decent converging rate and chance of a reaching the X vector we are computing.

```
import numpy as np

def pagerank(M, eps, d=0.85):
    N = M.shape[1]
    X = np.random.rand(N, 1)
    X = X / np.linalg.norm(X, 1)
    X_{t-1} = np.ones((N, 1), dtype=np.float32) * np.inf
    T = (d * M) + (((1 - d) / N) * np.ones((N, N), dtype=np.float32))

    while np.linalg.norm(X - X_{t-1}, 2) > eps:
        X_{t-1} = X
```

```
12          X = np.matmul(T, X)
13      return X
```

Listing 2: Python Power Method

3.1.4 Conclusion

Using these different techniques we manage to get a subset of all the hosts and services which are the one with most influence on the whole application. We can focus our learning on these particular variables. Which means that we'll have to deal with architectural changes of the application less often, beside it should also lighten the computations during the training phase. However we'll try to get some more information by grouping the other variables together insuring the flexibility of the application and light computations.

-(1806.10474) éventuellement...

3.2 Grouping features

Finally when we selected our principal features we cluster all the others in order to be able to adapt to changes in the application architecture. We assumed that the variables that will be removed and added the most often also have limited impact on the global application. We'll use the nearest neighbors process clustering described in this article [**?**] (which also expose a K mean algorithm but this one is said to be less effective with small amount of data, which may be the case in our situation) and then we'll use the method described in the first part on the selcted features and a mean of the ones in each cluster.

References