DEGREE PROJECT IN MACHINE LEARNING 120 CREDITS, SECOND CYCLE

STOCKHOLM, SWEDEN 2015

Machine Learning for Unsupervised Fraud Detection

RÉMI DOMINGUES

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

Machine Learning for Unsupervised Fraud Detection

R e m i D o m i n g u e s

Master's Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2015
Supervisor at CSC was Erik Fransen
Supervisor at INSA was Mehdi Kaytoue
Examiner was Anders Lansner

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

Abstract

Fraud is a threat that most online service providers must address in the development of their systems to ensure an efficient security policy and the integrity of their revenue. Amadeus, a Global Distribution System providing a transaction platform for flight booking by travel agents, is targeted by fraud attempts that could lead to revenue losses and indemnifications.

The objective of this thesis is to detect fraud attempts by applying machine learning algorithms to bookings represented by Passenger Name Record history. Due to the lack of labelled data, the current study presents a benchmark of unsupervised algorithms and aggregation methods. It also describes anomaly detection techniques which can be applied to self-organizing maps and hierarchical clustering.

Considering the important amount of transactions per second processed by Amadeus back-ends, we eventually highlight potential bottlenecks and alternatives.

Acknowledgements

This thesis was performed at Amadeus under the supervision of Jennifer Arnello and Yves Grealou. It was made in collaboration with KTH, Sweden and INSA de Lyon, France.

I would like to thank my supervisor at KTH Erik Fransén and my supervisor at INSA de Lyon Mehdi Kaytoue for their advices on anomaly detection.

A special thank to Francesco Buonora and Romain Senesi for their strong interest and help in understanding the functional concepts lying behind the data managed by Amadeus.

Thank you Anders Lansner for the examination of this thesis.

Contents

Chapter 1

Introduction

1 Context

1.1 Amadeus

Amadeus is the leading Global Distribution System (GDS) competing with Sabre and Galileo. This company has built a platform connecting together the travel industry agents in order to facilitate the distribution of travel products and services through IT solutions. Their customers include travel agencies, airlines, airports, hotels, railway companies, cruise lines and car rental companies.

As an IT provider, Amadeus can host the data of travel companies and distribute their content to buyers such as travel agencies. Thanks to this intermediary, companies can extend their market share and benefit from an efficient platform supporting search, pricing, booking and ticketing.

Therefore, Amadeus customers are divided in the following classes:

- **Travel providers:** providing the content to sell on the GDS.
- **Travel sellers:** buying the content distributed by Amadeus and selling it to end users. Sellers can be travel agents, e.g. agencies selling trip packages to customers, or ATO/CTO (Airport Travel Operators / City Travel Operators) who are employees of travel providers selling their own content, e.g. employees of an Air France office.
- **Global Distribution Systems:** such as Sabre, performing transactions with the Amadeus systems when a reservation is made through another GDS and targets at least one company hosted by Amadeus.

1.2 Passenger Name Record (PNR)

The booking information are stored in a PNR. The reservation stored can belong to multiple passengers, as long as they all have the same itinerary. It is not limited to one segment (e.g. we can store several flights). This data structure is divided in

multiple envelopes. An envelope describes a list of changes (addition of a passenger, ticketing...) applied to a PNR, and is created when a user commits the changes he has made. The information contained by a PNR include but are not limited to:

- Record Locator (RLOC): PNR identifier
- Passengers information: name, contact...
- Segments: origin, destination, flight ID, departure and arrival date...
- Services: additional luggage, seat, special meal...
- Frequent traveler cards
- Tickets: segment ID, fare
- Payments: type, credit card number...

Due to the complexity of a GDS, different kinds of fraud attempts may occur during one of the many processes provided.

2 Problem

Thanks to the feedback provided by its content provider customers, Amadeus has identified a few misuses and fraud profiles. Those fraud attempts are carried out by some users in order to **gain access to undeserved advantages**, **perform prohibited actions** or **receive unmerited incentives**[1].

Examples of fraud attempts are:

- **Time limit churning:** by taking advantage of various functionalities, agents are able to lock the booking of a seat for an unlimited time without issuing and paying a ticket. This gives them the possibility to offer an unlimited reflection period to their customers without the usual price increase.
- **Frequent flyer abuse:** abusive use of frequent flyer cards to be granted higher privileges.
- **Flooding**

Those fraud attempts threaten the image and revenue integrity of Amadeus and travel providers.

Despite the knowledge of some fraud types and since fraud attempts can evolve and appear with software updates, Amadeus wants to avoid the use of hard coded rules.

Eventually, the company suspects the existence of unknown fraud attempts targeting its systems and no labelled data is available for the prototype.

[1] Incentive: fee paid by Amadeus to travel agents in order to incite them to use Amadeus instead of another GDS.

3 Objective

Due to the absence of a labelled dataset and the important amount of time needed to build such a dataset, the aim of this Master thesis is to **study and develop a fraud detection prototype** based on unsupervised machine learning algorithms.

Using unsupervised techniques, the prototype should **detect new types of fraud attempts** that were unknown to Amadeus if such attempts exist.

Therefore and since the company wants to avoid the use of hard coded rules, the system built should automatically compute boundaries between regular PNRs and fraudulent ones. To do so, the prototype will detect outlying PNRs by comparing each PNR to a sufficient number of PNRs.

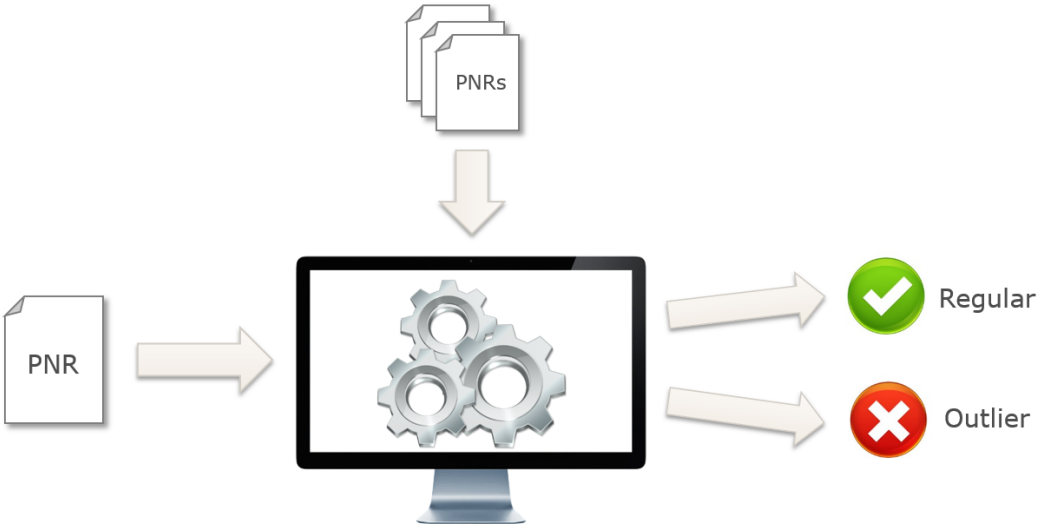

Figure 1: Prototype concept

This study will finally highlight the possible bottlenecks of its approach and **benchmark and recommend** the best algorithms to use in terms of quality and computational efficiency.

4 Constraints

As previously mentioned, a strong constraint lies in the absence of labelled data.

One must also consider a few figures in the development of this prototype. On average of 3.7 million bookings are performed each day on the targeted systems, but 20 to 30 millions of PNR envelopes are created each day with a usual peak of 600 PNR transactions per second.

If the quality of the results of the fraud detection engine is convincing, a product could be developed and made available to travel providers. No study has been made yet regarding the expected number of providers subscribing to this product

though performances will be a key decision factor. Therefore, the performances of the prototype must be carefully benchmarked.

One way to reduce the computational cost and to improve the stability of the results would be to use models which can be stored and for which streaming predictions could be applied. Doing so would also free us from the heavy computation and storage of a distance matrix.

If no constraints were specified regarding the languages used for the implementation of the prototype, care must be taken that those do not impact too much the computation time.

Chapter 2

Theory

1 Machine learning

Machine learning is a field of artificial intelligence describing algorithms which are able to learn from data and therefore adapt their behaviour. This field is divided into three categories.

In **supervised learning** algorithms build a model during a training phase in which they receive input data and the corresponding output data. Datasets for which each input data is mapped to an expected class label or value are called *labelled*. Once they have been trained, those algorithms should subsequently be able to predict accurate outputs using unseen input data only. The aim of those algorithms is thus to learn an accurate way to match input data to output data.

Reinforcement learning targets the learning of a decision process by presenting to the algorithm an environment in which it can perform a set of actions leading to a final reward.

Unsupervised learning makes use of unlabelled data by trying to achieve various goals. One may look for hidden patterns, try to cluster similar data points together or even seek outliers in a dataset.

Since no labelled data were provided for the current study, we are interested in the last category. Considering that a majority of the data should not be fraudulent, we aim at finding anomalies in our dataset, i.e. data points which are significantly different from the others, also called outliers. If the dataset is big enough and the frauds in minority, we can expect those outliers to be either frauds, misuses or very rare use cases.

Outliers may also be generated by system malfunctions and therefore contain invalid or extreme values. Based on the critical nature of the information stored in PNRs and the number of users of the Amadeus GDS, we will assume that the data retrieved is reliable. If not, the outliers detected will still highlight potential system malfunctions and thus provide a valuable feedback to Amadeus.

Figure 1 shows an example of outlier which lies far from the average behaviour of the dataset computed by linear regression.

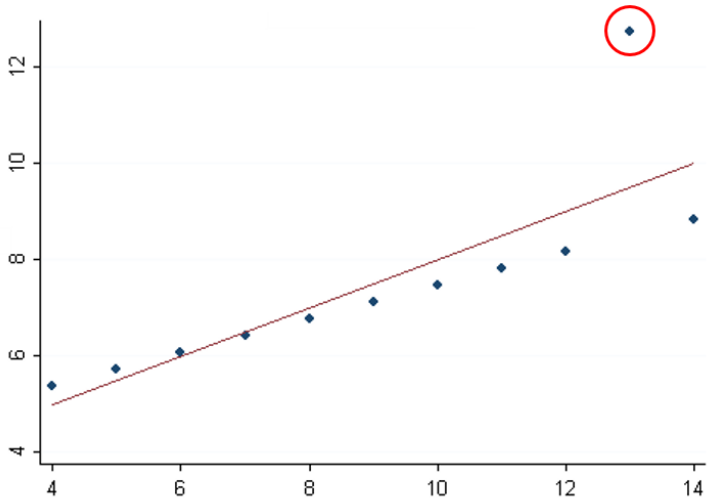

Figure 1: Outlier

1.1 DBSCAN

The Density-Based Spatial Clustering of Applications with Noise (DBSCAN)[4] algorithm finds clusters of arbitrary shape in large datasets.

This algorithm is quite interesting for the current use case since it claims to provide relevant clusters even if the dataset contains *noise*. The noise is a set of data points that are very different from the other data points, and thus outliers. This guarantee is valuable to the extent that outliers should not impact the construction of the clusters which will keep their integrity. However, the best performances are achieved for clusters of similar density and we cannot assume it for the current data.

As opposed to K-Means[7] which assigns a cluster to each data point, DBSCAN is able to recognize outliers and do not cluster them. Instead of requiring a number of clusters, DBSCAN is eventually able to automatically compute any number of clusters based on the parameters given.

Algorithm

Two parameters are required for this algorithm:

- Eps: maximum distance between two samples for them to be considered in the same neighborhood
- MinPts: minimum number of samples in the neighborhood of a point in order to flag the point as a core point, i.e. belonging to a cluster. This number includes the point itself

For a point to belong to a cluster, it has to contain at least *MinPts* data points in its neighborhood, including itself, the neighborhood radius being the *Eps* parameter. If a point in the neighborhood of the data point already belongs to a cluster, the current data point is assigned to the existing cluster. Otherwise a new cluster is created.

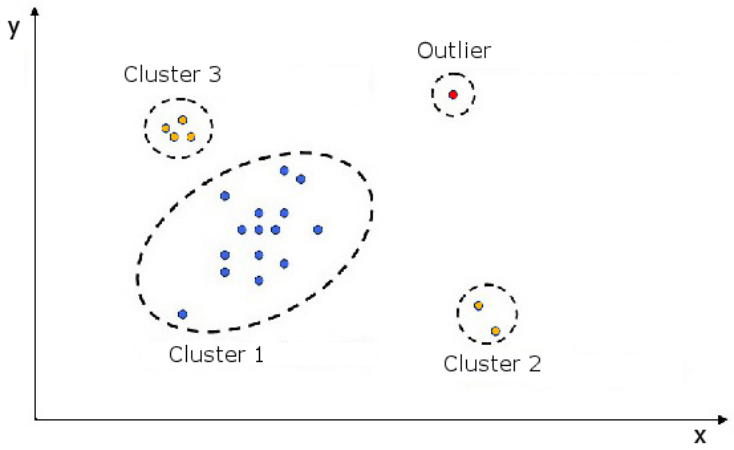

Figure 2: Clustering

Figure 2 shows a possible clustering using the DBSCAN algorithm. Depending on *MinPts*, *Cluster 2* and *Cluster 3* can also be marked as outliers. With a smaller *Eps*, *Cluster 1* could be splitted into multiple clusters and outliers, or could be merged with *Cluster 3* if we used a higher *Eps*.

This example highlights the difficulty of detecting outliers in a dataset and shows that a thorough analysis and a good understanding of the functional aspect of the data are required to efficiently detect frauds. The use of manually defined thresholds is also a need that must be investigated.

A final parameter required by this algorithm and many others is the distance used to compute the neighborhood. This study will benchmark the efficiency of various metrics applied to the DBSCAN algorithm.

Euclidean distance

The Euclidean is the most common metric, also called L^2 *norm*. It is defined as follows:.

$$d(u, v) = \sqrt{\sum_{i=1}^{n}(v_i - u_i)^2} \tag{2.1}$$

Mahalanobis distance

The Mahalanobis distance between a vectors $\vec{v} = (v_1, v_2, ..., v_n)^T$ and a dataset having an average vector $\vec{\mu} = (\mu_1, \mu_2, ..., \mu_n)^T$ and a covariance matrix $\sum$ is defined in equation 2.2.

$$D_M(\vec{v}) = \sqrt{(\vec{v} - \vec{\mu})^T \sum^{-1} (\vec{v} - \vec{\mu})} \tag{2.2}$$

The Mahalanobis distance between two vectors $\vec{u}$ and $\vec{v}$ is defined in equation 2.3, with $\sum$ the covariance matrix.

$$d(\vec{u}, \vec{v}) = \sqrt{(\vec{u} - \vec{v})^T \sum^{-1} (\vec{u} - \vec{v})} \tag{2.3}$$

This metric can be quite useful in outlier detection since it computes the distance of each N-dimensional vector (the data points) from the center of the dataset normalized by the standard deviation of each dimension (the features) and adjusted for the covariance of those dimensions. By doing so, data points containing extreme values will be given a high Mahalanobis distance which will allow us to mark them as outliers.

Yet, since this metric uses means and standard deviations, the quality of the results may be affected by extreme values impacting those measures.

1.2 MeanShift

MeanShift[3] is a non-parametric clustering algorithm which aims at discovering groups in datasets of smooth density by finding the maximum of a density function.

Using a kernel function $K(x_i - x)$ with x the initial estimate of the maximum of the density function, MeanShift computes the weight of the data points surrounding x in order to re-estimate this value. This centroid-based approach will thus compute the mean of the data points of various regions in the dataset in order to obtain a few core data points which will be the centroids of our final clusters. A post-processing step is eventually applied to remove the near-duplicates centroids.

As in DBSCAN, data points far from the centroids can be ignored by the clustering process and thus flagged as outliers.

A bandwidth h is required by the algorithm in order to estimate the density function (equation 2.4 with x_i a data point and $k()$ the kernel) on the dataset using kernel density estimation (KDE). Note that the bandwidth h can be automatically selected by an estimation method described in [13], or computed using two manually defined parameters which are a quantile Q and a number of seeds N. Once the density function is computed, optimization methods (such as gradient descent) are used to find the local maxima.

$$f(x) = \sum_i K(x - x_i) = \sum_i k\left(\frac{||x - x_i||^2}{h^2}\right) \tag{2.4}$$

Based on the description of this algorithm and the proof by MA Carreira-Perpinan in [2], MeanShift is actually an expectation-maximization algorithm (EM) when a Gaussian kernel is used and a generalized EM algorithm when a non-Gaussian kernel is used. Since our experiment will use a Gaussian kernel, the clusters found are likely to have a globular shape. The EM algorithm is described in more details in section 1.3.

1.3 Gaussian Mixture Model (GMM)

GMM is a clustering algorithm which computes clusters by fitting a given number of Gaussians to the dataset and iteratively estimating their parameters. A mixture of Gaussians is a probability distribution obtained by the weighted sum of K normal distributions $P(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x; \mu_k, \sigma_k^2)$ where $\sum_{k=1}^{K} \pi_k = 1$ and $\pi_k > 0$. Parameters are computed by optimizing the maximum-likelihood of by the data $P(x, h|\theta)$ with h_{ik} the probability of assigning each data point x_i to each Gaussian component of the mixture and θ the Gaussian parameters.

By doing so, each data point has a probability to belong to each Gaussian and outliers are points having a very low probability to be generated by the K Gaussians.

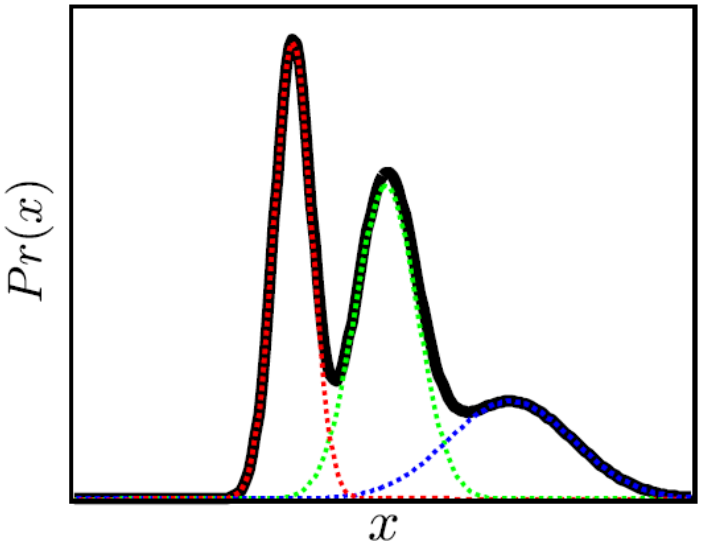

Figure 3: Mixture of Gaussians

It is similar to K-means since it requires a given number of clusters and guarantees to find a local maximum. However, it gets better to the extent that it provides a clustering probability for each data point and cluster.

One way to estimate the Gaussian parameters is to use the EM algorithm.

Expectation-Maximization (EM)

This iterative algorithm starts by initializing the parameters of the K Gaussians with random values. Those parameters are then updated by repeating the two following steps:

- E-step: for each Gaussian component, compute the posterior probability $P(h_i = k|x_i, \theta^{(t)})$ that x_i was generated by this component according to the current parameters.

- M-step: fit each Gaussian according to the posterior probabilities previously computed. This is achieved by computing the maximum likelihood of the parameters.

1.4 One-class SVM

One-class SVM is an extension of Support Vector Machines algorithms which has been introduced by Schölkopf et al[11] and makes use of unlabelled data in order to perform unsupervised novelty detection in high-dimensional data. As for SVM, this algorithm is based on the use of a kernel (linear, polynomial, sigmoid or RBF) and uses the kernel trick in order to compute the dot product between data points represented in a high-dimensional space to find a separating hyperplane.

The one-class SVM fits a decision boundary on the entire dataset in order to have an accurate representation of the data distribution. As for many algorithms, the decision boundary should fit the data as much as possible without implying overfitting, hence using a margin and a slack. To do so, a parameter ν is given, representing the maximum fraction of training errors and minimum fraction of support vectors. A parameter γ manually defines a kernel coefficient for the polynomial, sigmoid and RBF kernels.

The hyperplane is computed by trying to separate all the data points from the origin of the feature space and maximizing the distance between this hyperplane and the origin. The result of this computation is a binary function which indicates whether a data point is inside or outside the boundary containing the training points.

An example of decision boundary is shown in figure 4 with training and testing data points.

A possible issue with the application of this algorithm to our use case is that the data used for the training should not be contaminated by outliers as the decision boundary may fit them. Since we cannot define a standard pattern for our PNRs because of the many use cases supported by Amadeus, since many frauds cannot be detected using a simple filter and since we are also looking for unknown frauds, this constraint is likely to impact the quality of the results output by the one-class SVM algorithm.

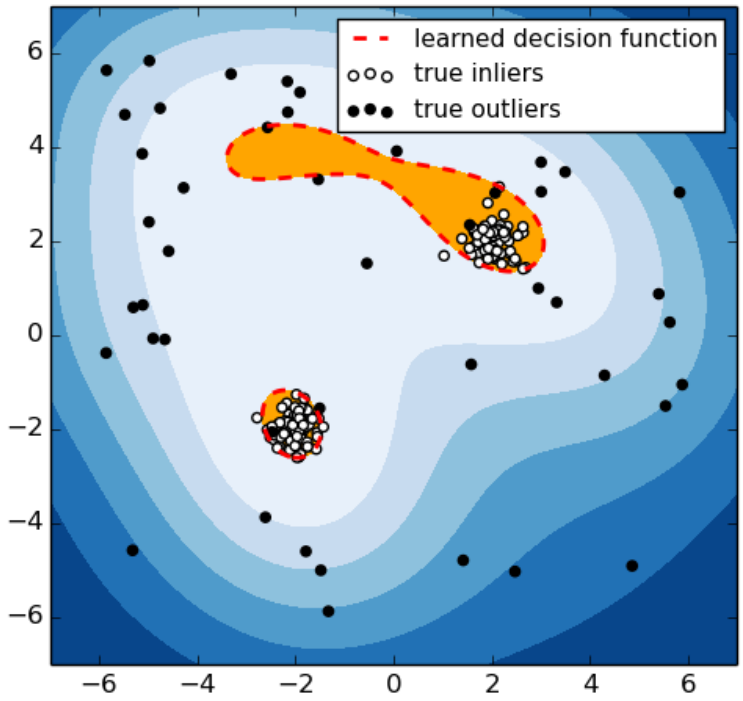

Figure 4: Outlier detection using one-class SVM (`scikit-learn.org`)

1.5 Z-Score and Median absolute deviation (MAD)

Another way to detect outliers is to work on each feature separately instead of computing distances between high-dimensional vectors. This approach cannot find correlations between features but has the advantage of being resistant to the curse of dimensionality.

We can achieve this by assigning an outlying score to each dimension of a data point $\vec{x} = (x_1, ..., x_m)$, and then aggregate those scores to obtain the outlyingness of the feature vector. As we are looking for extreme values even for a single feature, the aggregation method chosen here is the maximum of the scores: $O(\vec{s}) = \max_{i=1}^{m} \vec{s}_i$

Z-Score

The Z-score, also called standard score, of a one-dimensional dataset V is defined in equation 2.5. It is the number of standard deviations between a value and the mean of the dataset. For the current study, we will use the absolute value of this score as described in the equation with μ and $sigma$ the average and standard deviation of V.

$$zscore(v) = \frac{|v - \mu|}{\sigma} \tag{2.5}$$

Such a score, as illustrated in figure 5, is a good estimation of the outlyingness

of a value. However, this outlier detection method is designed for features having a normal distribution and could return poor results if the number of outliers or their value is high enough to significantly impact the mean and standard deviation.

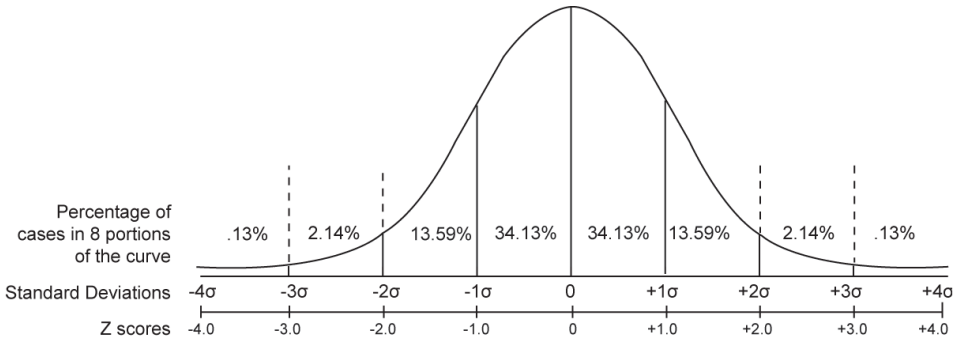

Figure 5: Z-Scores

Median absolute deviation (MAD)

Using the MAD described in equation 2.6 could help us with this issue.

$$MAD(V) = median(|V - median(V)|) \tag{2.6}$$

This measure describes the median of the distance between each data point and the median of the dataset. It could thus be used to compute a score S showing how different a value is from the others. This is done in equation 2.7 where the result is the number of MAD between the median of the dataset and a given value. Note the similarity between equations 2.5 and 2.7.

The score computed here has the advantage of using the median and the MAD and thus won't be affected by the value of the outliers.

$$S(v) = \frac{|v - median(V)|}{MAD(V)} \tag{2.7}$$

Yet and since this score is based on the absolute number of MAD from the median, this measure is efficient only for datasets having a symmetric distribution. Indeed, computing the distance from the median on a distribution having for example one tail longer than the other would not make much sense. To solve this issue, we can compute the score using the same previous formula but using two different MADs.

The first MAD is then computed using only the values less than or equal to the median. This first MAD will be used when computing the score of a value lower than the median. To the opposite, the second MAD uses only values higher than or equal to the median and is used to compute the score for values higher than the median.

1.6 Hierarchical clustering

This class of algorithms builds a hierarchy of clusters, the smallest containing a single data point and the biggest the entire dataset. Depending on the approach, the algorithm either merges clusters starting with one cluster per data point until one final cluster remains (agglomerative method), or starts with a single big cluster and splits it in two clusters at each step until all clusters contain one data point (divisive method).

For this purpose, the algorithm require a metric to compute the dissimilarity between the data points, e.g. Euclidean, and a function which can be applied to the pairwise distances of observations in the different clusters, e.g. Ward's minimum variance method[18]. This method states that the two clusters to merge at a given step are the ones minimizing the increase of total within-cluster variance after the merging step.

Once the hierarchical clustering has been applied to the data, one must find outliers in the resulting tree of clusters, also called dendrogram. We mention here a method detailed in [15]. This method ranks all the data points according to their outlyingness, which allows us to mark as outliers the data points having a ranking higher than a given threshold.

The outlyingness O of a data point x is here the maximum score obtained by x at a merging step i of the algorithm (eq. 2.8 where N is the size of the dataset and thus $N-1$ the number of merging steps of the algorithm). The score s of x at a step i can be computed according to three different methods detailed below.

$$O(x) = \max_{i=1}^{N} s_i(x) \tag{2.8}$$

Linear

In equation 2.9, $|g|$ is the number of data points in the cluster where x belongs at step i and $p()$ is a function penalizing large groups defined in equation 2.10.

$$s_i(x) = \frac{i}{N-1} * p(|g|) \tag{2.9}$$

The penalization function is detailed here, with n the cluster size and $1 \leq t \leq N$.

$$p(n) = (1 - \frac{n-1}{N-2})\mathbb{1}_{n<t} \tag{2.10}$$

Sigmoid

The sigmoid score is computed as follows, with $p()$ defined in equation 2.12.

$$s_i(x) = e^{-2\frac{(i-(N-1))^2}{(N-1)^2}} * p(|g|) \tag{2.11}$$

$$p(n) = (1 - e^{-4\frac{(n-2t)^2}{(2t)^2}})\mathbb{1}_{n<2t} \tag{2.12}$$

Size difference

Below is the size difference equation, where $g_{y,i}$ and $g_{x,i}$ are the two groups merged at step i and $g_{x,i}$ is the group which x belongs to.

$$s_i(x) = max\left(\frac{|g_{y,i}| - |g_{x,i}|}{|g_{y,i}| + |g_{x,i}|}, 0\right) \tag{2.13}$$

1.7 Hidden Markov Model (HMM)

HMMs are a statistical model introduced by Baum et al.[9]. To explain this concept, we must first detail what a Markov chain is.

Markov chains

A Markov chain is a directed graph with transition probabilities (fig. 6). For a system in a given state S, with here $S \in \{r, c, s\}$, the sum of the transition probabilities leaving the state is 1.

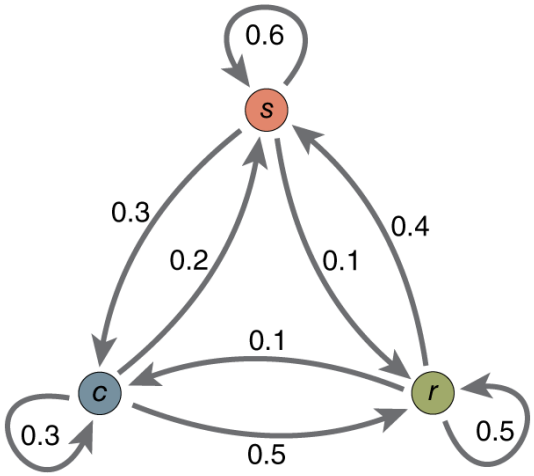

Figure 6: Markov Chain

Hidden Markov Model

In a HMM, the sequence of vertices of the Markov chain taken by the model over time is usually unknown, which is why the states are called *hidden states*. In such models, each state has its own probability distribution to generate what is called

an emission. If the states are unknown, we usually know the sequence of emissions generated by the states, also called *observations*.

The HMM, described in figure 7 where x_i are the hidden states and y_i the observations over time, is thus a Markov process since the sequence of hidden states over time is represented by a Markov chain. A HMM is hence described by three matrices:

- Initial distribution: $\pi = P(x_0)$, the vector containing the probability to be in each hidden state at the step s_0
- Transition matrix: $A = P(x_{t+1}|x_t)$, transition probabilities between hidden states
- Emission matrix $B = P(y_t|x_t)$: probability of generating the observations for each hidden state

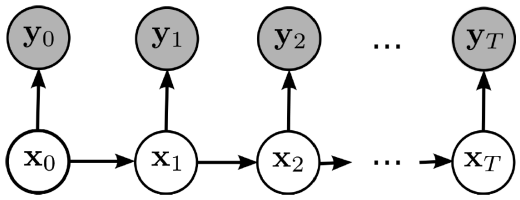

Figure 7: Hidden Markov Model

The hidden Markov models allow us to solve various problems:

1. Compute the probability of a sequence of T observations
2. Given a sequence of observations, find the most probable sequence of hidden states
3. Train the parameters A, B and π to maximize the probability of a sequence of observations $P(y_{1:T})$. This training requires to specify the number of hidden states, also called components

HMM training

The training of a HMM is done using the Baum-Welch algorithm[1]. At each step of this algorithm, we compute

- $\alpha_t(i) = P(y_{1:T}, x_t = i) = b_i(y_t) \sum_{j=1}^{N} a_{ji}\alpha_{t-1}(j)$ with $\alpha_1(i) = b_i(y_1)\pi_i$ and $2 \leq t \leq T$ (forward algorithm)
- $\beta_t(i) = P(y_{t+1:T}|x_t = i) = \sum_{j=1}^{N} a_{ij}b_j(y_{t+1}\beta_{t+1}(j))$ with $\beta_T(i) = 1$ and $1 \leq t \leq T-1$ (backward algorithm)

- The gamma function $\gamma_t(i) = P(x_t = i|y_{1:T}) \propto P(x_t = i, y_{1:T}) = \frac{\alpha_t(i)\beta_t(i)}{\sum_i \alpha_T(i)}$ according to the forward/backward algorithm
- The digamma function $\gamma_t(i,j) = P(x_t = i, x_{t+1} = j|y_{1:T}) = \frac{a_{ij}b_j(y_{t+1})\alpha_t(i)\beta_{t+1}(j)}{\sum_i \alpha_T(i)}$, with a_{ij} and $b_j(y_{t+1})$ probabilities from A and B

A, B and π are eventually estimated using the previous results:

$$\begin{aligned} \pi_i &= \gamma_1(i) \forall i = 1, ..., N \\ a_{ij} &= \frac{\sum_{t=1}^{T-1} \gamma_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \forall i, j = 1, ..., N \\ b_j(k) &= \frac{\sum_{t=1, y_t=k}^{T} \gamma_t(i)}{\sum_{t=1}^{T} \gamma_t(i)} \forall i, j = 1, ..., N \end{aligned} \tag{2.14}$$

Thanks to the previous algorithm, the HMM of a dataset can be trained. For this, we need our dataset to contain sequences of observations, possibly of different length, instead of data points belonging to a specific space. This is an entirely different approach.

Outlier detection

Let's assume that we have a HMM trained according to the distribution of the sequences observed in a dataset. Outlier detection corresponds to the first problem previously mentioned for HMMs.

Therefore, we compute the probability of each action sequence to be generated by the model under the parameters. This is done in equation 2.15 and requires a preliminary run of the forward algorithm. Once this is done, we simply flag as outliers the sequences for which the probability to be generated is lower than a given threshold.

$$P(y_{1:T}) = \sum_{i=1}^{N} \alpha_T(i) \tag{2.15}$$

1.8 Self-Organizing Maps (SOM)

A Self-Organizing Map, also called Kohonen Map[5], is a type of neural network which maps points from an input space to points in an output space. This transformation keeps the topology of the data by using a set of neurons in the same feature space fitted to the dataset so that the final topology of the neural network is a good representation of the data. By doing so, points that were close in the input space will also be close in the output space.

For an input data point in the high-dimensional feature space, the corresponding output will be the neuron in the feature space which is the closest to this point.

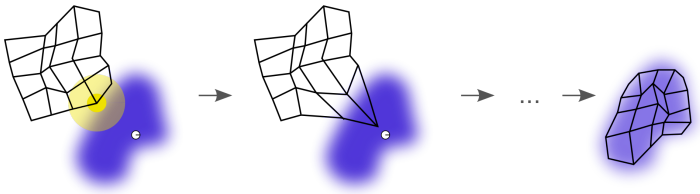

Figure 8: Training of a SOM

Using for example a 1-dimensional or 2-dimensional grid topology of the network, we are able to reduce the dimensionality of the data to one or two dimensions.

On-line algorithm

The conventional on-line algorithm updates the neurons of the network every time an input vector is given in input. We initialize the neural network with random values for the weights of the neurons, then a step in the training is done as follows:

1. Select x, a random data point in the dataset
2. Compute the similarity, e.g. the opposite of the Euclidean distance, between x and the neurons of the network
3. Find the winning node, i.e. the most similar neuron to x
4. Update the weights of the winning node and its neighbors in the output grid so that they are moved closer to the input pattern

Since there is only one neuron at each step which is the closest to the given point and since only this neuron and its neighborhood are updated, we call this training a competitive learning. The neighborhood of a node depends on the network topology used (see figures 9 and 10) and is not related to the distance between the neurons in the feature space.

During the training, the size of the neighborhood is progressively reduced, so that many nodes are updated at the beginning of the algorithm and only a few when the training reaches its end. The weights of the neighbors h_{ck} can be computed using a Gaussian for which the mean is the winning node, and σ the radius of the neighborhood in the grid.

To update the weights, we move the winning neuron and its neighbors closer to the data point. In equation 2.16, w_k is an updated neuron in the feature space, x_t the data point randomly selected at step t, η the learning rate and h_{ck} a weight based on the winning node w_c, the updated node w_k and the neighborhood function.

$$w_k(t+1) = w_k(t) + \eta h_{ck}(x_t - w_k(t)) \tag{2.16}$$

In order to improve the convergence of the training algorithm, we apply a decay function (eq. 2.17, with t the current iteration and T the total number of iterations) to the neighborhood size σ and to the learning rate η.

$$decay_t(v) = \frac{v}{1 + \frac{t}{T}} \tag{2.17}$$

Figures 9 and 10 show examples of SOM structures with an input layer in a 6-dimensional feature space (on the left) and two possible network topologies (on the right). The only connections shown are between the input vector and one output neuron, but each dimension of the input layer is actually connected to every node.

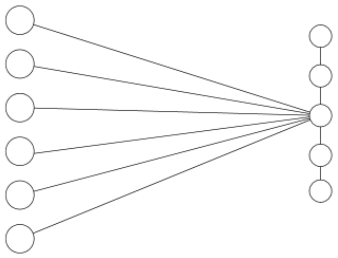

Figure 9: 1-dimensional topology

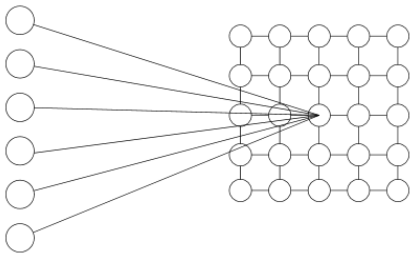

Figure 10: 2-dimensional topology

Batch algorithm

In the previous algorithm, the neurons were updated after the presentation of each input vector. The batch algorithm[6] updates the network at the end of each epoch during which N input vectors are presented.

The algorithm still uses a Gaussian neighborhood and a decay function applied to its standard deviation, but drops the learning rate η. Also, the weights of a given neuron are now replaced at the end of each epoch by the weighted sum of the input vectors having their winning node in the neighborhood of the given neuron. The network is now updated according to equation 2.18 where t_0 and t_f are the start and finish of the present epoch, $w_k(t_f)$ the weights of the neuron k computed at the end of the epoch and h_{ck} the neighborhood weight.

$$w_k(t_f) = \frac{\sum_{t'=t_0}^{t'=t_f} h_{ck} x_{t'}}{\sum_{t'=t_0}^{t'=t_f} h_{ck}} \tag{2.18}$$

Once the network has been randomly initialized and t set to 0, the iteration of each epoch is done according to the folllowing steps:

1. Initialize the numerator and denominator of equation 2.18 to 0
2. For each input vector x_t
    a) Compute the Euclidean distance between x_t and all the neurons $w_k(t_0)$

b) Compute the winning node, which is the closest neuron to x_t

c) Update the numerator and denominator of all neurons according to equation 2.18

d) $t = t + 1$

3. Update the weights of all neurons using equation 2.18

This algorithm has a few advantages. Among those, the training is no longer impacted by the presentation order of the input vectors, which could previously lead to a stronger influence for the last input vectors presented. Dropping the learning rate also simplifies the parametrization of the algorithm and avoid a poor convergence that was obtained when using an inadequate parameter.

Median interneuron distance (MID) matrix

One of the advantages of SOM is to perform dimensionality reduction. However, if our data can be mapped to a 1 or 2-dimensional network as previously described, we now have to visualize this network. This can be done by computing the median interneuron distance matrix.

For a 2-dimensional grid network of MxN neurons, each value of the MxN matrix is the median of the Euclidean distance between a neuron $w_{i,j}$ and the neurons in its neighborhood, with $1 \leq i \leq M$ and $1 \leq j \leq N$. As before, the size of the neighborhood must be manually defined.

Note that the mean or the maximum of the distance could be used instead of the median, and that another metric could also be used.

After normalization, we obtain a weight matrix that can be plotted in 2D space. Each value of this matrix corresponds to a neuron in the network, and values close to 1 show neurons far from their neighborhood.

Examples of plots can be found in the following section and in the experiment (section 5.4).

Outlier detection

Once the SOM is trained and the MID matrix computed, outliers can be detected as detailed in [8].

This detection starts by identifying **outlying neurons**, which are neurons lying far from the other neurons and could have been attracted by dense sets of outliers such as in figure 11. If such neurons exist, they can be easily identified using the MID matrix. As you can see in figure 12, outlying neurons have a value in the MID matrix much higher than the one of the other neurons. We can thus use a simple threshold or compare those distances to do our selection. The plot of the MID matrix is a good tool to check the existence of outlying neurons.

When this is done, outliers are the data points having for winning node an outlying neuron.

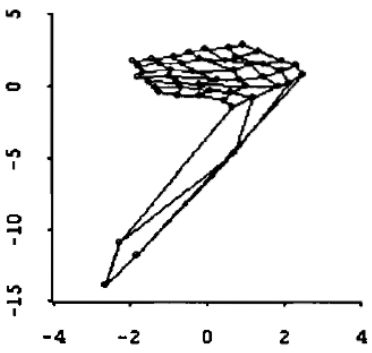

Figure 11: Outlying neurons

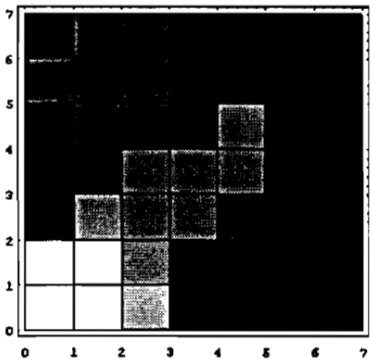

Figure 12: Median interneuron distance matrix

Eventually, we consider the case where very few outliers are present in the dataset or are not dense enough to attract a neuron. In this case, we don't detect any outlying neuron but use the **quantization error** to find the outliers. This measure is simply the dissimilarity (e.g. distance) between a data point and its winning node.

Those outliers are detected using a threshold, which can be found using a box plot showing the QEs of the dataset (section 5.4).

2 Quality assessment

One of the difficulties of unsupervised learning is to measure the quality of the results output by a model. Here are various ways to interpret those results.

2.1 Silhouette Coefficient

This scoring method can be applied to clustering algorithms. It gives (eq. 2.19) a score to each sample based on a mean intra-cluster distance a (average dissimilarity between a sample and the other samples in the cluster) and a mean nearest-cluster distance b (lowest average dissimilarity between the sample and a cluster which it does not belong to).

$$s(x) = \frac{b(x) - a(x)}{max(a(x), b(x))} \tag{2.19}$$

The score of all samples is then aggregated, e.g. by taking the average, in a single score for which 1 indicates very dense clusters far from the others and -1 clustering where samples are likely to be assigned to the wrong clusters.

2.2 Quantization error

As previously mentioned in section 1.8, the quantization error is usually the distance between a sample and the closest centroid of a cluster. Hence, we can compute the mean squared quantization error (MSQE) in equation 2.20 where c_i is the centroid of the closest cluster. This criterion is actually the one used by K-means in its objective function.

$$MSQE = E[(x_i - c_i)^2] \tag{2.20}$$

2.3 Precision, recall and F1 score

Precision and recall are measures widely used in supervised learning where the true label of each sample is known. **Precision** allows us to measure the number of samples that have been correctly classified for a given class divided by the number of samples predicted in this class. For a binary classification (positive and negative), samples can be labelled with the right label (true prediction) or the wrong one (false prediction).

$$precision = \frac{|true\ positives|}{|true\ positives + false\ positives|} \tag{2.21}$$

The **recall** is the number of samples that have been correctly classified in a class divided by the number of samples actually belonging to the class. This shows us the proportion of samples belonging to a given class which have been labelled as such.

$$recall = \frac{|true\ positives|}{|true\ positives + false\ negatives|} \tag{2.22}$$

The **F1 score** is the harmonic mean of the precision and recall and can be used to measure the efficiency of a binary classification. It is therefore

$$F1score = 2 * \frac{precision * recall}{precision + recall} \tag{2.23}$$

3 Ensemble learning

Ensemble learning is a class of machine learning algorithms which combine multiple learning algorithms (e.g. decision tree, SVM...) in order obtain better predictions (e.g. boosting, bagging...).

We will restrict this study to a few aggregation operators in order to combine the results output by the models detailed in the previous section. For each data point given to an unsupervised algorithm, the operators below aim at aggregating the corresponding outputs in a single score.

3.1 Weighted sum

This operator, also called weighted averaging is a simple multi-criteria decision analysis (MCDA) method consisting in the weighted sum of a_i values, with $\sum_i w_i = 1$.

$$WA(a_1, .., a_n) = \sum_{i=1}^{n} w_i a_i \tag{2.24}$$

3.2 Ordered Weighted Averaging (OWA)

OWA[19] is an non-linear operator based on fuzzy logic. It also relies on a weighted sum but has the advantage of automatically assigning a weight to each score depending on the rank of the score σ_i in the sorted list of scores. In equation 2.25, the weights w_i are defined according to a distribution (e.g. Gaussian) and $\sum_i w_i = 1$.

$$OWA(a_1, .., a_n) = \sum_{i=1}^{n} w_i a_{\sigma_i} \tag{2.25}$$

3.3 Weighted Ordered Weighted Averaging (WOWA)

An alternative to the OWA operator has been introduced in 1997[17]. It uses a weight vector W defined as previously according to a distribution and assigns those weights depending on the score ordering. However, it takes an additional weight vector $p = (p_1, ..., p_n)$in parameter, with $\sum_i p_i = 1$.

$$WOWA(a_1, .., a_n) = \sum_{i=1}^{n} v_i b_i \tag{2.26}$$

v_i is defined in equation 2.27 where f is a non-decreasing function that interpolates the points $(\frac{i}{n}, \sum_{j \leq i} w_j)$ together with the point $(0, 0)$. We can observe that if $p = (\frac{1}{n}, ..., \frac{1}{n})$ then the WOWA operator returns the same result than the OWA operator.

$$v_i = f(\sum_{j \leq i} p_{\sigma_j}) - f(\sum_{j \leq i-1} p_{\sigma_j}) \tag{2.27}$$

Chapter 3

Experiment

The Unsupervised Fraud Detection prototype has been implemented according to the architecture detailed in figures 1 and 3.

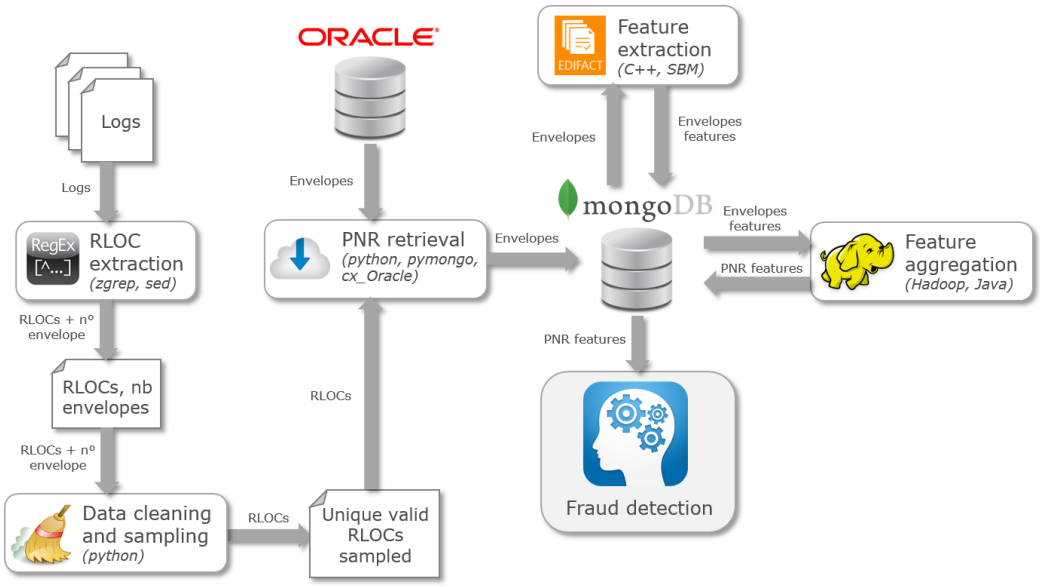

Figure 1: Architecture

1 Data collection

1.1 RLOC extraction

The first step to build our dataset is to retrieve a representative list of PNRs. This is achieved by getting the record locators (RLOCs, PNR identifiers) of all the PNRs updated on the 02/07/2015.

This retrieval is done by parsing the logs of an Amadeus system using the UNIX commands *zgrep* to decompress and filter 107GB of logs and *sed* to keep only the RLOC from the logs filtered.

Since some PNRs can be corrupted because of an invalid state at the end of a system transaction, we applied a similar process to these logs in order to retrieve only the corrupted ones.

1.2 Data cleaning and sampling

Using Python scripts, we removed the duplicates from both lists and filtered the corrupted PNRs, obtaining 6 164 304 unique valid RLOCs.

Due to a limited storage capacity, we uniformly sampled 60 000 RLOCs from this shuffled list in order to retrieve their content.

1.3 PNR retrieval

PNRs are composed of envelopes, one per committed transaction, describing a history of actions which is stored in a distributed Oracle database. However, this history is deleted when the purge date of a PNR is reached, usually a few weeks after the last flight. Because of this, only 40 183 PNRs were retrieved by our Python script. This is equivalent to 0.65% of the total number of PNRs updated in one day.

An envelope textually describes the changes applied to a PNR during a transaction. It follows the EDIFACT format (see appendix A). In order to maintain the representativeness of our dataset and since those PNRs were retrieved some days after parsing the RLOCs, we removed all the envelopes created after July, 2^{nd} (0.42% of envelopes). The remaining 852 590 envelopes were stored in a MongoDB instance. This collection has an average of 21 envelopes per PNR and weights 19.11GB after decompression. It is equivalent to 3.31% of the envelopes created in one day in the Amadeus GDS.

Querying the database, decompressing the envelopes and storing them in MongoDB allows us to process 1.23 envelope per second (purged PNRs were excluded from this benchmark). The decompression and insertion are negligible comparing to the query. The total process lasted about 12 days, which is a very strong limitation to our use case.

2 Feature extraction

2.1 Feature extraction (Envelope)

Once we have the raw data, those EDIFACT messages must be parsed to extract relevant information that could allow us to detect suspicious behaviors. We have built a list of relevant feature with functional experts having a fraud knowledge. This allowed us to extract 58 features per envelope. 57 of them are mostly counters applied to the most important aspects of a PNR (passengers, points of sale, travel

segments (including marriages[1], special service requests (SSR), frequent traveler cards (FQTV) and forms of payment), but can also be timestamps (e.g. creation date of the envelope).

Those features give a very good grasp of the state of a PNR, but we had to ignore some information. To overcome this limitation, we also extracted the list of every action performed in each envelope. This list is a collection of action codes, given in the same order for each envelope (i.e. they are not given in the same order they were performed).

Extracting the features has been done using a C++ parser developed by Amadeus. Extraction and insertion in MongoDB required 12 hours for a collection size of 872MB. It has been computed on a SUSE Linux Enterprise Server 11, using one core of an Intel(R) Xeon(R) CPU X5690 @ 3.47GHz with a remote MongoDB instance. Network communications through optical fibers is negligible.

The final implementation of this prototype should use a multi-threaded architecture, possibly distributed. The algorithm is embarrassingly parallel since we can assign a list of PNRs to each process.

2.2 Feature aggregation (PNR)

According to experts from Amadeus, an envelope often does not contain enough data to identify a fraud. The entire PNR history is usually required for this purpose, which is why we must aggregate the features per envelope into a single feature vector having the same dimension for all PNRs.

During this process counters are often aggregated by taking the value in the last envelope, the sum, average and standard deviation (some aggregations ignore envelopes where the value is equal to 0). We have also defined ratios (e.g. $\frac{final\ number\ of\ segment}{sum\ of\ added\ segments}$), computed the PNR age or the total number of envelopes.

The aggregation is implemented using a Java MapReduce job running on Hadoop. We also used an existing MongoDB Connector for Hadoop[2] which allows us to use MongoDB as input and output for our MapReduce job, instead of exporting and importing our data to/from the Hadoop File System (HDFS).

The MapReduce job starts by splitting the dataset of envelopes into independent chunks.

The **mappers** receive a chunk of envelopes as input. They map each envelope received to a key/value pair, using the RLOC as key and the envelope as value. The set of key/value pairs built is then sorted and output.

While the map tasks are processed, the framework shuffles the outputs received and groups by key the values, i.e. the envelopes are grouped by PNR ID.

[1] Marriage: Two segments can be married if they are sold together, i.e. one of them cannot be sold at the specified fare if bought alone.

[2] `https://github.com/mongodb/mongo-hadoop`

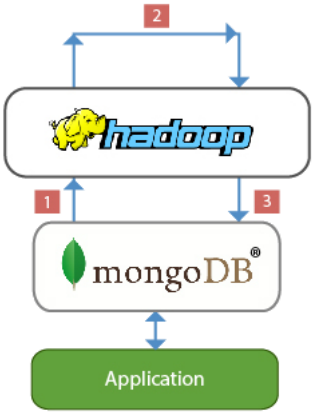

Figure 2: Batch aggregation

Each **reducer** receives lists of envelopes related to the same RLOCs. For each RLOC, they sort the corresponding envelopes by envelope number and then apply aggregation operators (sum, avg,...) to the values in order to build the PNR features.

Using MapReduce jobs allows a very efficient parallelization of our algorithm, which can be distributed on a cluster for better performances. The MongoDB Connector for Hadoop did not allow us to use *secondary sort*. This functionality uses composite keys which in order to sort the values after grouping them by key. By doing so, we could have sorted the envelopes by creation date before sending them to the reducers while waiting for the completion of the mappers, this would have spared us the manual sort processed at the beginning of the reducers.

Feature aggregation is much more efficient than extraction and required only 6 minutes and 32 seconds. The output is a collection of JSON documents weighting 244MB. Each JSON document contains 121 features. One of those features is the concatenation of the sorted action sequences. The MapReduce job has been executed on a single node, using a virtual machine running on LinuxMint 17.4, with 4 hyper-threaded cores of a Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz and a remote MongoDB instance.

3 Data cleaning

Cleaning the data is the first step of the fraud detection module detailed in figure 3.

When one must implement a data mining workflow, Python is a handy language which comes with a lot of powerful libraries to manipulate data (pandas[3]), ap-

[3] `http://pandas.pydata.org`

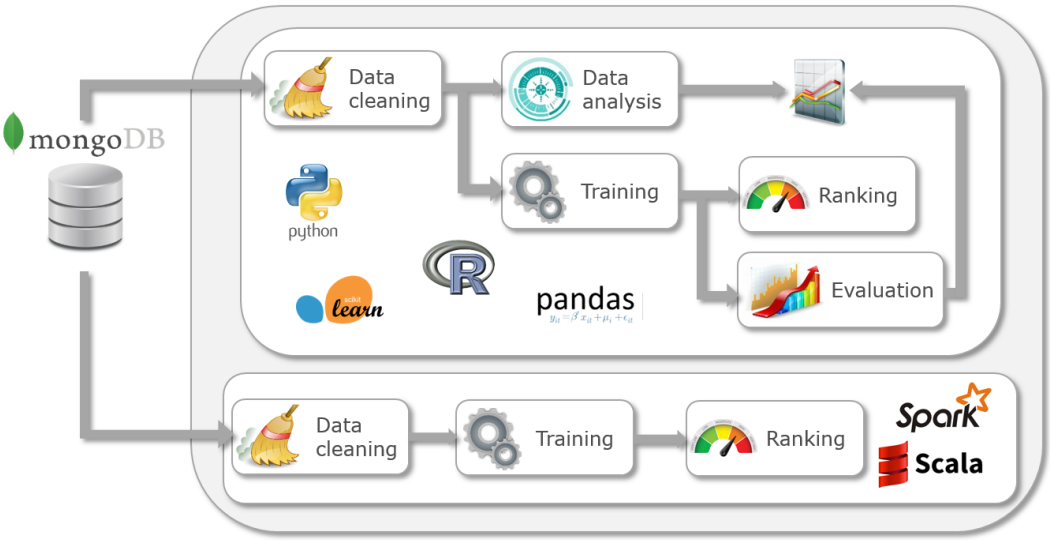

Figure 3: Fraud detection module architecture

ply machine learning algorithms (scikit-learn[4]) and visualize statistics (matplotlib[5], seaborn[6]).

Keeping in mind the need for high performances, we also developed a fraud detection engine using Spark[7], a very efficient and distributed framework providing APIs in Java, Scala, Python and R to process large-scale data. In Spark, data is stored in a Resilient Distributed Dataset (RDD) on which parallel methods can be called.

Since both approaches are interesting, we started with a detailed fraud detection module including data analysis and benchmarks in python. Once the best algorithms were selected, we built a scalable and distributed proof of concept using Spark and Scala.

As previously mentioned, the data retrieved is already very clean and we only need to apply some adjustments. This cleaning was implemented in Spark and Python.

3.1 Unknown values

For some PNRs, a specific feature cannot be computed. When this happen, we replace the unknown value by the feature average.

[4] http://scikit-learn.org
[5] http://matplotlib.org/
[6] http://stanford.edu/~mwaskom/software/seaborn
[7] http://spark.apache.org

3.2 Linearly correlated features

Data analysis shows that some features are always 0, which occurs for very rare actions absent from our dataset. This prevents us from computing the Mahalanobis provided by scikit-learn.

Those values are an issue since we need to invert the correlation matrix of our dataset, and such values make our matrix not invertible. We must thus solve the matrix singularity by removing the linear correlations between the features. Those correlations could be detected by looking at singular value decomposition (SVD) and finding values close to 0.

In this implementation, we only match features always equal to 0, which is enough to remove the linear correlations of our dataset. 9 features match this filter and are filtered out, leaving 112 remaining features including the action sequence.

3.3 Scaling

To make the time features readable by humans, we convert them from milliseconds to floating hours.

Ratios usually expressed between 0 and 1 (can be higher than one due to splits for example) are scaled to percentages usually between 0 and 100.

Eventually, we normalized the data so that the values of all features remain between 0 and 1.

3.4 Feature redefinition

After some discussions with fraud experts, we removed 33 features related to SSRs, including 4 linearly correlated. 83 features remain, including the action sequence.

4 Data analysis

This section describes a dataset of 20 000 PNRs uniformly sampled from the initial dataset of 40 183 PNRs. The size of the dataset had to be reduced due to limited computational resources, the virtual machine used to run the machine learning algorithms containing 20GB of RAM (swap included).

Data analysis has been processed in Python using pandas, matplotlib and seaborn. Statistics and histograms are shown for 111 features (all, except linear correlations), the other plots are applied to the final list of 82 features (action sequences are not analyzed here).

4.1 Statistics

A sample of the statistics computed is given below. Average, standard deviation, minimum, maximum and 5 quantiles are shown. *nb_env* is the number of envelopes, the PNR age *creation.age* is given in hours, *seg.add.sum* is the number of added travel segments, *fp.add* is the number of added forms of payment.

Feature	nb_env	creation.age	seg.add.sum	split	fp.add
mean	21.076100	789.528140	4.780950	0.09740	0.863000
std	30.367974	1326.969628	22.846721	0.59811	1.400332
min	1.000000	0.000000	0.000000	0.00000	0.000000
5%	3.000000	0.109139	0.000000	0.00000	0.000000
25%	6.000000	45.538056	2.000000	0.00000	0.000000
50%	13.000000	273.224028	3.000000	0.00000	1.000000
75%	26.000000	856.738056	5.000000	0.00000	1.000000
95%	61.000000	3547.940917	12.000000	1.00000	3.000000
max	1471.000000	50258.785278	2342.000000	27.00000	69.000000

We can definitely observe some strong outliers. While 95% of the PNRs have 61 envelopes or less, one PNR has 1471 envelopes. Similarly, 95% of the PNRs have been created less than 5 months ago, but one has been created almost 6 years ago. There is definitely something strange about the purging date of this PNR.

Usually, a few travel segments are added in a PNR, each segment corresponding for example to a flight. Yet, somebody added 2342 travel segments to a PNR, which is definitely suspicious.

Eventually, we can observe a surprisingly high number of 69 forms of payment added.

Those few numbers show that we certainly have frauds and misuses in our dataset, even if we only took a small sample of the data generated by Amadeus each day.

4.2 Feature distributions

Figure 4 shows the distribution of some features. Left Y-axes represent the number of PNRs in the histogram while the right Y-axes show the density of the probability density functions obtained by kernel density estimations (KDE) using Gaussian kernels and Scott's rule[12].

We can also see some extreme values here, e.g. 35 different points of sale updating a PNR (*pos.updater*), the PNR created 6 years ago, the one with 1471 envelopes or a PNR where the final number of tickets is four times higher than the number of added tickets (*seg.tkt.add_ratio*), which means that most of the tickets were cancelled.

4.3 Box-and-whisker plot

The box plot is a useful visualization showing the quartiles (25%, 50%, 75%) of a set of values using a box. This diagram is completed by lines extending the box and called **whiskers**. The length of those lines is $1.5 * IQR$, with IQR the interquartile range defined as $IQR = Q3 - Q1$. Those concepts are shown in picture 5.

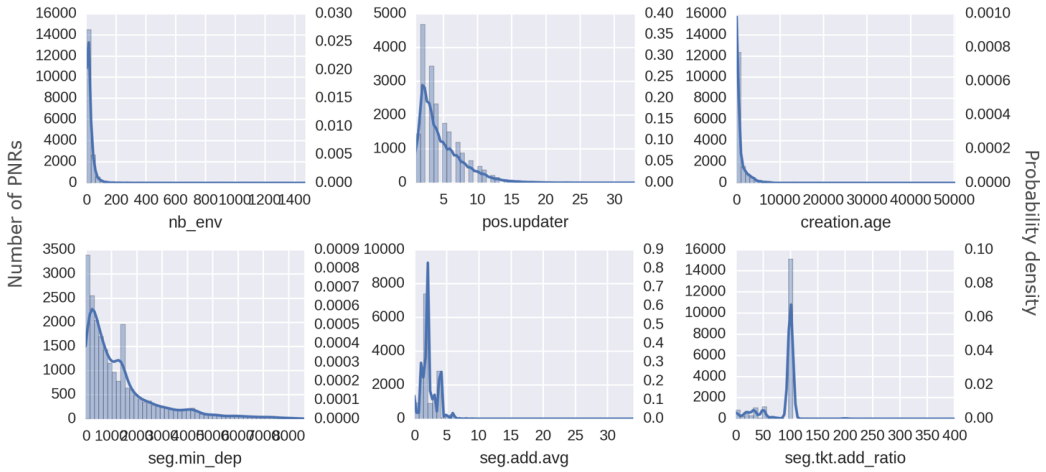

Figure 4: Histogram and probability density function per feature

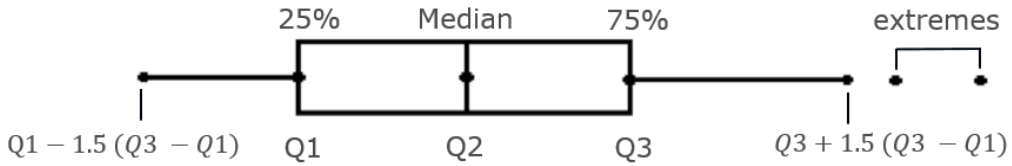

Figure 5: Box-and-whisker plot - Explanation

Figure 6 shows the box-and-whisker plot of each feature. Values higher than the extremity of the right whisker are represented by gray diamonds. The 20 000 values of each feature are represented by light blue dots. A log scale is used.

Ratios represent a percentage (can be higher than 100) and some features such as *Feature 1* are only represented by a whisker. This feature has a zero-IQR, since $Q1 = Q3 = 0$, thus we should have only extreme values plotted instead of the whisker. This is a bug in the matplotlib library for which the issue was still opened[8] when generating the diagram.

[8] `https://github.com/matplotlib/matplotlib/issues/5331`

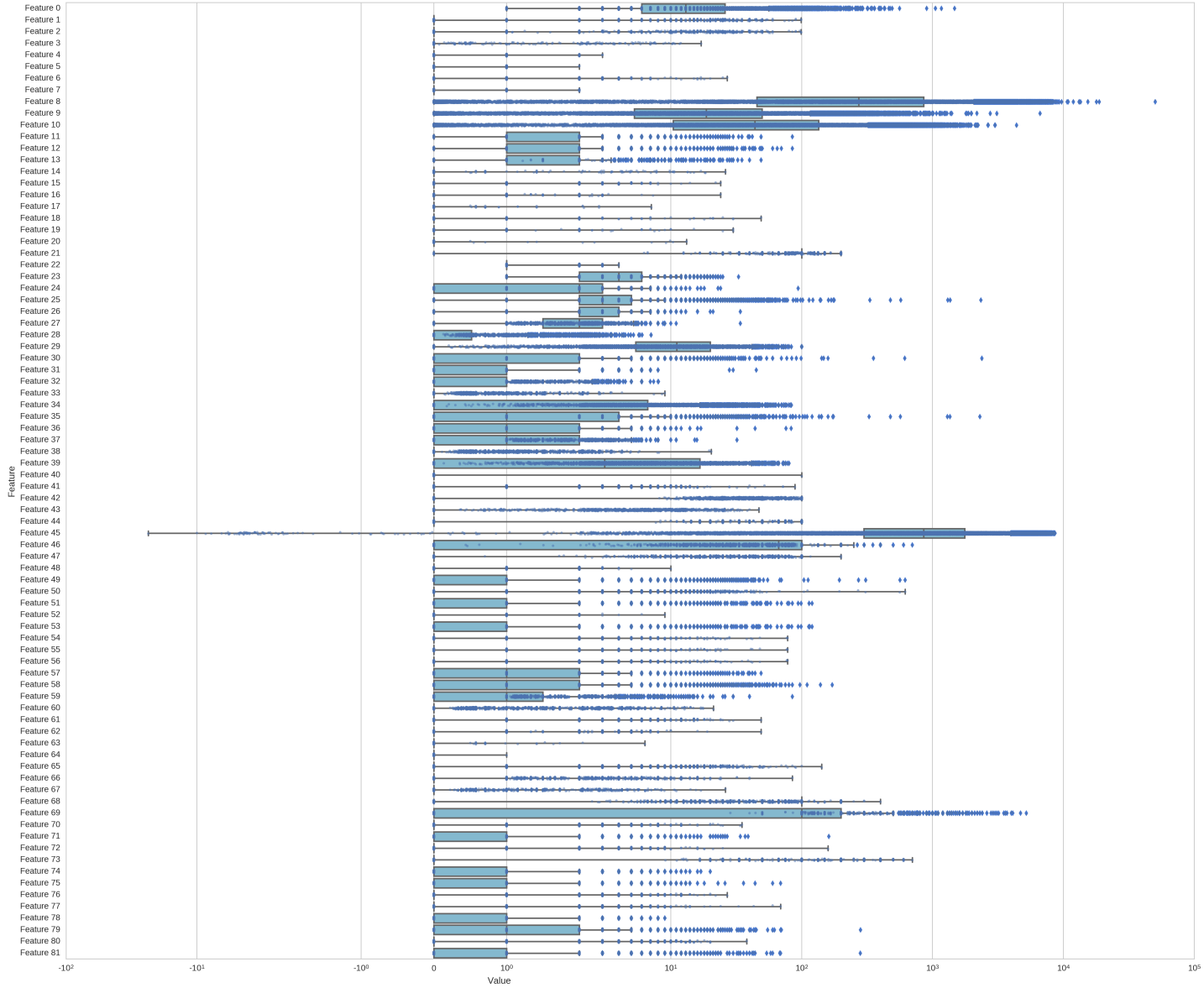

Figure 6: Box-and-whisker plot (20 000 PNRs)

4.4 Correlation heatmap

Figure 7 shows the pairwise correlation between the features. The coefficients are computed according to the Pearson product-moment correlation coefficient defined in equation 3.1 where R_{ij} is the correlation coefficient between features i and j, and C_{ij} is the covariance between those features.

$$R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}} \tag{3.1}$$

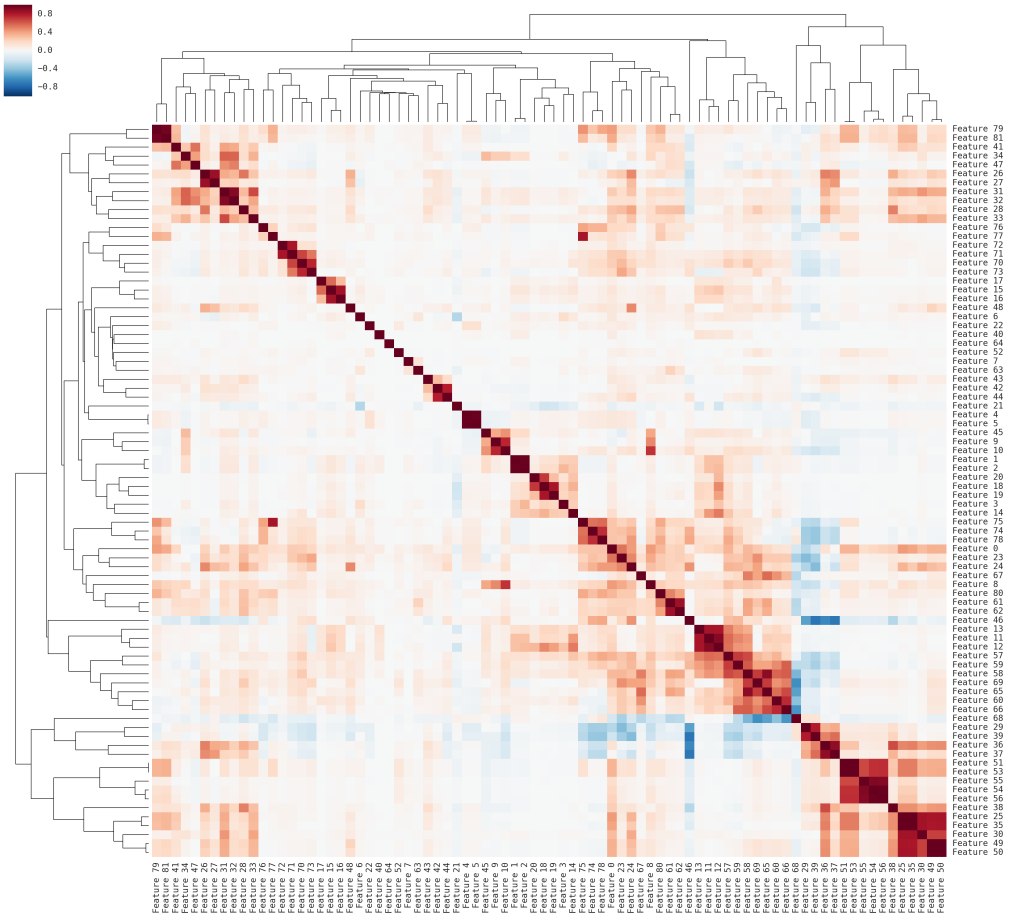

Figure 7: Hierarchical clustering applied to feature correlation heatmap

A hierarchical clustering has been applied on the resulting symmetric correlation matrix. This clustering helps us understand the processes behind the PNRs. We can for example observe that the number of marriages (segments sold together) is strongly correlated to the number of segments added.

If needed, this visualization could help us reducing the dimensionality of our dataset by applying correlation feature selection (CFS).

4.5 Principal component analysis

PCA[14] is an interesting tool used to reduce the dimensionality of a dataset. When dealing with high dimensional feature vectors, the relevant data can be expressed using less dimensions. This is achieved by building the covariance matrix of our dataset then extracting the principal components defining the output space by computing the N eigenvectors corresponding the highest eigenvalues.

This transformation could be useful to visualize the data and work on a dataset of lower dimension, thus less impacted by the curse of dimensionality. A 2D representation of our dataset is given in figure 8.

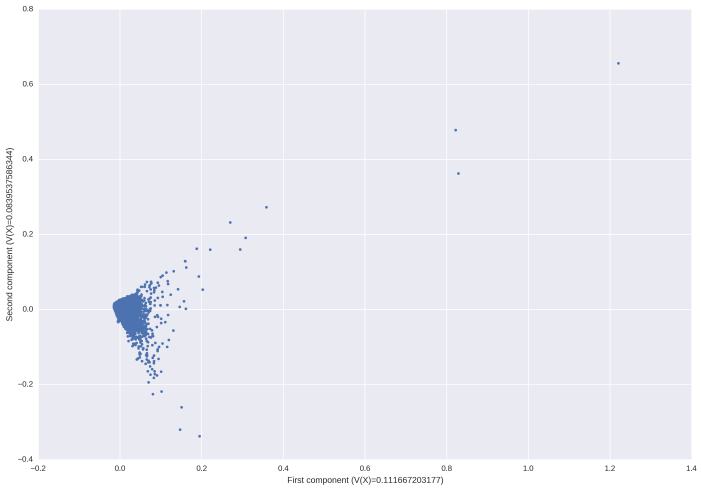

Figure 8: PCA - 2 components

As previously explained, the eigenvectors are computed in order to preserve the variance in the dataset. As seen in figure 6, some feature with high variance, e.g. features 8, 9 and 10, will have a strong weight in the transformation from the input to output space but do not highlight many interesting outliers. To the opposite, features such as feature 64 having almost always the same values except for some outliers will be almost ignored by the space transformation.

Because of this, applying PCA would result in an important loss of information and would prevent us from efficiently detecting outliers. This has been confirmed by a manual check of some outliers from the PCA representation, where the PNRs studied were all regular ones.

4.6 Manual fraud detection

In order to evaluate the performances of the unsupervised machine learning algorithms, a set of fraudulent PNRs from our dataset would be interesting.

Working with fraud experts, we manually investigated some PNRs for which extreme values were noticed, we defined some inclusion and exclusion rules and were able to identify some of the most evident known fraudulent samples. For other frauds, no simple rule could be defined and we had to create lists of RLOCs.

The number of frauds found in 20 000 PNRs is described below.

Fraud	**Count**	**Percentage**
Fraud 2	1	0.005%
Fraud 5	8	0.04%
Fraud 1	3	0.015%
Fraud 3	13	0.065%
Fraud 4	96	0.48%

By identifying frauds, we were also able to study their profile. This can be achieved by highlighting fraudulent PNRs on a box-and-whisker plot and selecting features for which those PNRs have only extreme values.

The profile of Fraud 4 is given in figure 9. Blue dots are regular PNRs and green dots are fraudulent samples. We can see here that features 51, 53, 54 and 56 are very interesting to identify this fraud.

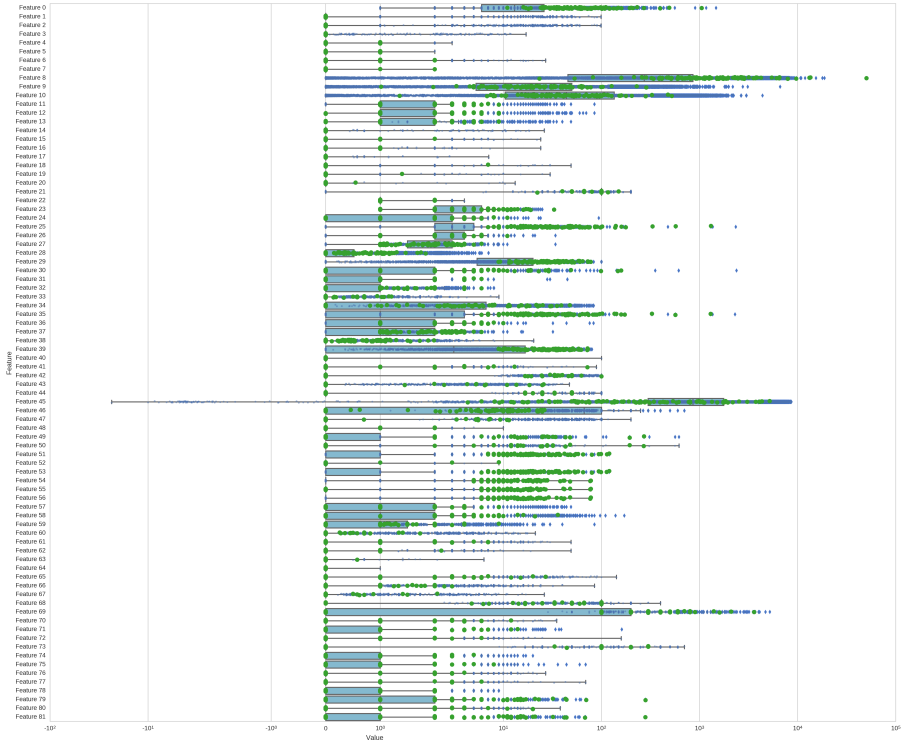

Figure 9: Fraud profile - Fraud 4

5 Fraud detection

The algorithms used for fraud detection are described in section 1. All algorithms are fed with the 82 normalized features of 20 000 PNRs, except HMM which receives sequences of actions.

The following algorithms were already implemented in scikit-learn for the python module: DBSCAN, MeanShift, Gaussian Mixture Model, One-class SVM. All of them return a predicted class ID or -1 if the PNR is an outlier, except the GMM which returns the log probability under the model.

5.1 Hierarchical clustering

Hierarchical clustering and the corresponding formulas for outlier detection were called in python using a R package called Data Mining with R[16] (DMWR).

The python module rpy2[9] has been used to interface R with python.

5.2 Z-Score and Median absolute deviation

Z-Score and MAD were both implemented in python. We explained in section 1.5 the advantages of MAD over Z-Score when scoring a dataset containing outliers. However, it has been seen in the previous box plots that some features have more than 50% of identical values. This is an issue since the MAD computed in equation 2.6 and used as denominator in equation 2.7 will be equal to 0.

From here, we could either decide not to apply the MAD, or to use a denominator of 1 when the MAD is equal to 0, which would flag as outliers all the values not equal to the median.

We choose to use the Z-Score, which is less resistant to outliers but is not subject to the previous issue. It allows us to give a score to each feature of each PNR. In order to obtain a single score per PNR, we use the maximum score obtained for any feature. The score are then scaled between 0 and 1 to represent a probability. Log operator is used to spread the values, and we defined that values which are 40 standard deviations are more away from the mean will have an outlier probability of 100%. Final score is computed according to equation 3.2, with $zscore()$ defined in equation 2.5.

$$score(v) = max(min(\frac{log(zscore(v))}{log(40)}, 1), 0) \tag{3.2}$$

The distribution of the scores in the dataset is given in figure 10.

Note that a score of 10 is already quite suspicious. Without the log operator, $\frac{10}{40} = 0.25$, which is quite a low probability. We now obtain $\frac{log(10)}{log(40)} \approx 0.62$. A probability of one corresponds thus to a Z-Score of 40 or more.

[9] `http://rpy.sourceforge.net`

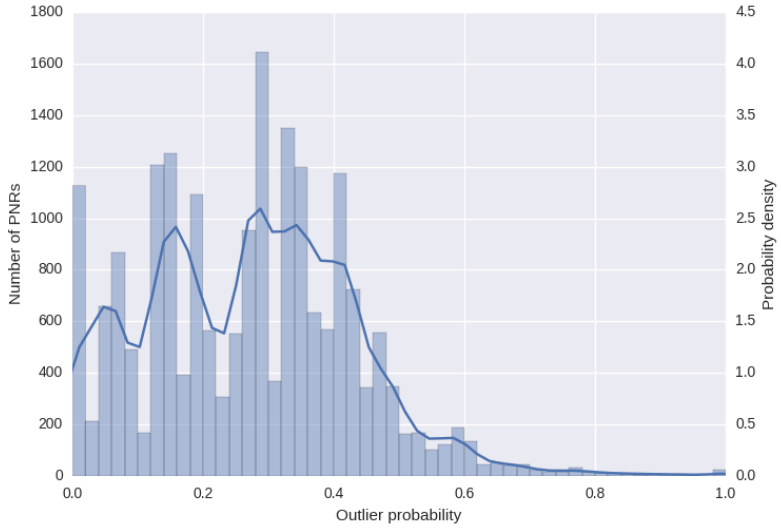

Figure 10: Z-Score distribution

In our first attempts with Z-Score, we obtained quite a lot of group PNRs. To refine a bit our results, we ignored 18 features which were systematically returning regular group PNRs, which leaves us with 64 features for this algorithm.

5.3 Hidden Markov Model

HMMs were trained using the *hmmlearn*[10] Python API. We fed this algorithm with the concatenation of the action sequences, updating the initial distribution vector accordingly. The total number of actions used to train the algorithm is 4 675 603, which gives an average of 233 actions per PNR or 11.1 actions per envelope.

The score returned is the log probability of the action sequence of a PNR under the trained model. The log operator is required to avoid a loss of information when multiplying low probabilities between each other, hence quickly approaching 0.

The probability for a sequence to be generated given a model strongly depends on the sequence length, a long sequence having a small probability to be generated. The sequence length could be an interesting feature, but it still has a high weight in the final probability and is already in the feature vectors used by the previous algorithms. To remove this factor, we can divide the log probability by the length of the sequence T.

Our experiment benchmarked the HMMs using the log likelihood returned by the model, and after removing the length factor. After having scored all the PNRs, we normalize the scores to obtain probabilities between 0 and 1. Normalized log likelihood is shown in figure 11. Figure 12 shows normalized log likelihood including length normalization.

[10] `https://github.com/hmmlearn/hmmlearn`

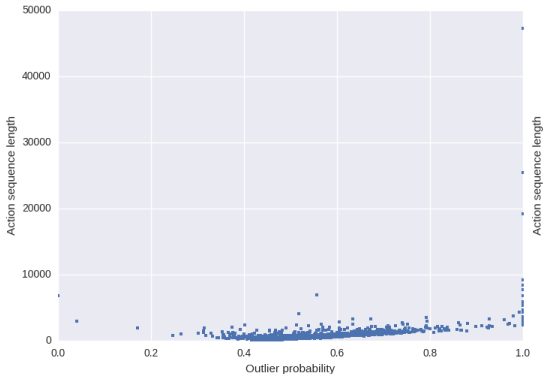

Figure 11: Normalized log likelihoods

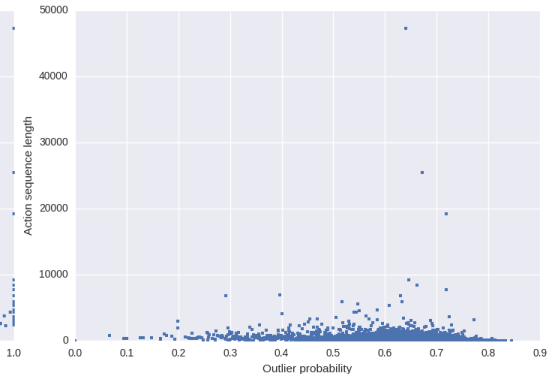

Figure 12: Normalized log likelihoods previously divided by sequence lengths

5.4 Self-Organizing Maps

On-line algorithm

On-line SOM were entirely implemented in python. We also implemented the computation of the MID matrix and the outlier detection algorithm. The code responsible for the training of the SOM was later replaced by the MiniSom[11] library, also providing an on-line training.

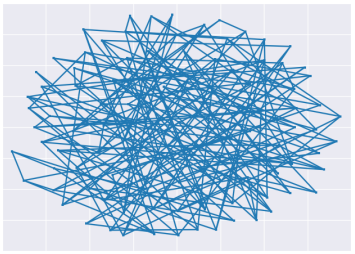

Figure 13: Initialization

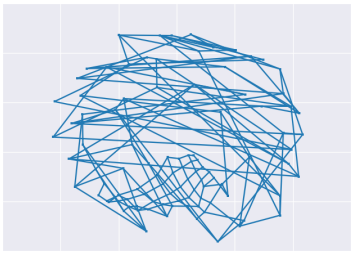

Figure 14: 100 000 steps

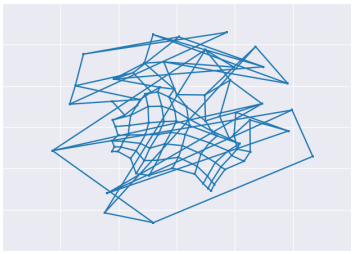

Figure 15: 200 000 steps

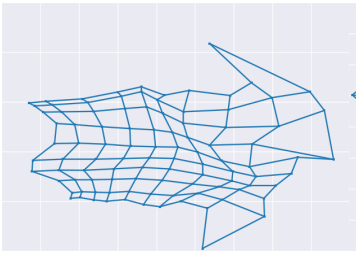

Figure 16: 300 000 steps

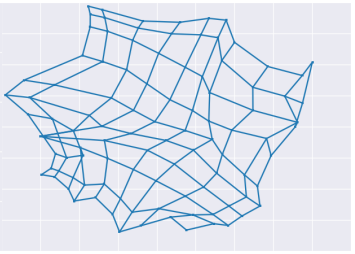

Figure 17: 400 000 steps

[11] `https://github.com/JustGlowing/minisom`

Figures 13 to 17 show the training of the SOM. We have used a 2-dimensional grid of size 10x10, each step updates the value of the winning node and its neighborhood. 400 000 iterations (equivalent to 20 times the size of the dataset) were used with an initial learning rate of 0.01 and an initial standard deviation for the neighborhood of $\sigma = 1$. The 2D representation of the neural network has been computed by the multidimensional scaling (MDS) algorithm implemented in scikit-learn.

An example of weights used to update the winning node and its neighbors is plotted in figure 18 (Gaussian distribution, $\sigma = 1$, $winning = (4, 4)$).

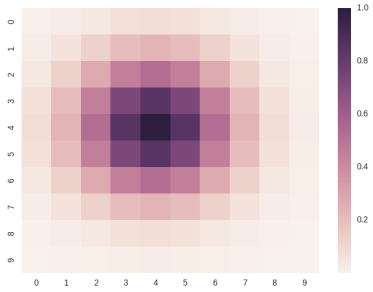

Figure 18: Gaussian neighborhood, $\sigma = 1$

The MID matrix of the trained network is plotted in figure 19. It was computed using the median of the distance with the 8 neighbors of each neuron. Neurons represented by values close to 1 are far from their neighborhood and will be flagged as outlying neurons. We have plotted the frauds manually detected during the data analysis phase on top of their winning node: Fraud 4 (cyan cross), Fraud 5 (purple triangle), Fraud 2 (red circle), Fraud 3 (green square) and Fraud 1 (blue diamond).

Figure 20 shows a box plot of the quantization error in the dataset, i.e. the distance between each data point and its winning node. This plot shows some PNRs lying far from their winning node, thus outliers which are flagged as such.

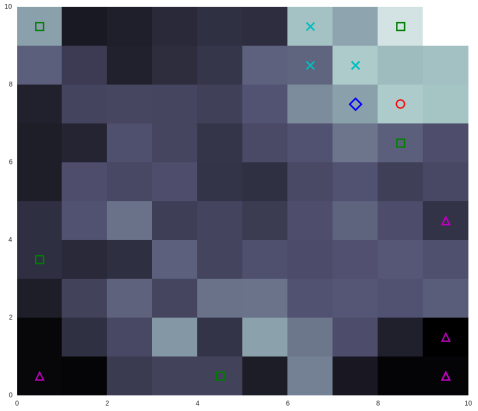

Figure 19: MID matrix

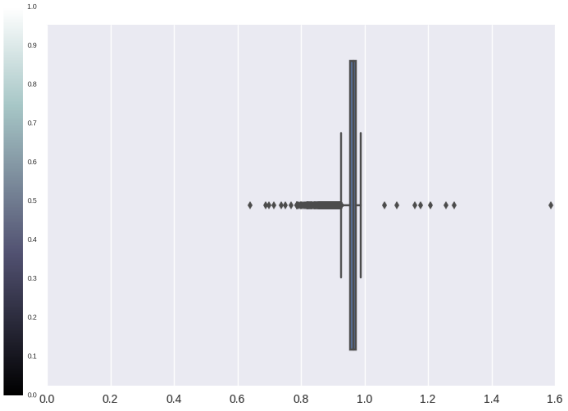

Figure 20: Box plot - Quantization errors

Distributed batch algorithm

Based on [10], we have implemented a distributed batch training in Scala using Spark.

Developing a parallel SOM training could have been achieved by either partitioning the neural network or splitting the data among the processes. A good speedup for a network partitioning algorithm would require to assign a sufficient number of neurons to each process in order to reduce the amount of communication. Since we use a neural network of 100 neurons, this approach may not be the most efficient.

Partitioning our dataset into equal chunks is thus likely to give better results. The on-line training implies a communication step after presenting each input vector. To the opposite, data partitioning with batch training would only require a synchronization at the end of each epoch, thus reducing the communication cost and increasing the speedup.

A distributed batch training has been implemented in Scala using Spark based on [10]. It follows the MapReduce programming model and each process is assigned a partition d_p of the data.

1. Randomly initialize the weights of the neural network
2. For each epoch
   a) Initialize the numerator and denominator of equation 2.18 to 0
   b) **Map:** For each process p
      - For each input vector x in d_p
        - Compute the Euclidean distance between x and all the neurons $w_k(t_0)$
        - Compute the winning node, which is the closest neuron to x
        - Output the numerator and denominator of all neurons according to equation 2.18 for x
   c) **Reduce:** Update the weights of all neurons according to equation 2.18 by summing up the output of the Map tasks

The output of the Map tasks is a tuple containing the numerator and denominator of each neuron. The numerator is a 3-dimensional matrix containing for each neuron the input vector x multiplied by the neighborhood weight. The denominator is a 2-dimensional matrix containing the neighborhood weight of each neuron

Figure 21 shows the execution time in seconds required by the training of a 10x10 SOM for 100 epochs and a dataset split into several partitions. This benchmark has been performed using local threads on a hyper-threaded quad-core. The computation time of the outlier detection is negligible.

A speedup of 1.7 can be observed when using an even number of partitions and 2 threads: $S_{2,2} = \frac{T_{1,1}}{T_{2,2}} = \frac{550}{323} = 1.70$ with $T_{t,p}$ the computation time for t threads and p partitions. With 3 threads and 3 partitions, we reach $S_{3,3} = \frac{T_{1,1}}{T_{3,3}} = \frac{550}{266} = 2.07$. However, using more threads does not significantly improve the performances. This may be caused by the increasing amount of communications required or the presence of other threads (scheduler, memory manager...) managed by Spark consuming CPU time.

The same amount of computations for the on-line algorithm (100 times the size of the dataset, hence 2 000 000 on-line iterations) took 1680 seconds, which is 3 times more than the single threaded implementation on Spark. This difference is probably due to the fact that Scala can be faster than Python and because of additional threading optimization that may be provided by the framework.

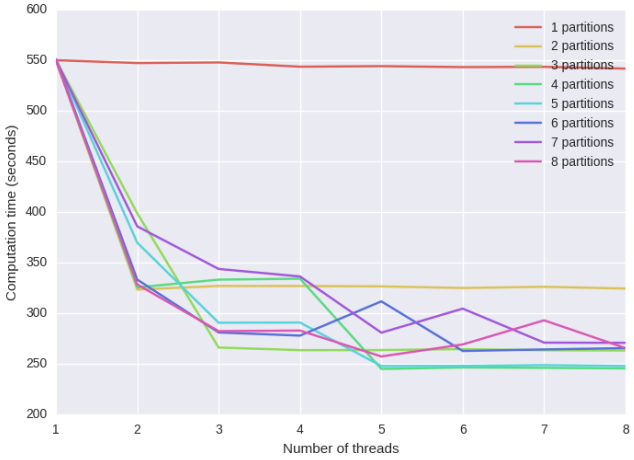

Figure 21: Computation time of the distributed SOM batch training

Figures 22, 23 and 24 show the result of the distributed training.

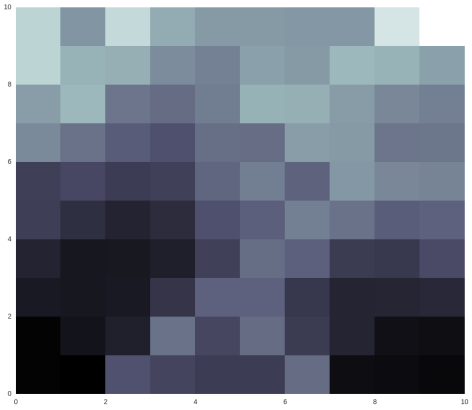

Figure 22: Spark MID matrix

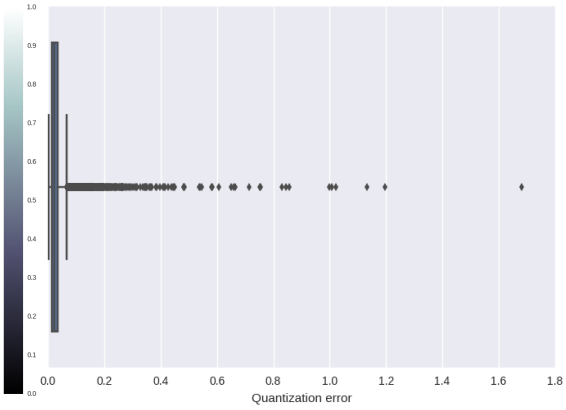

Figure 23: Spark quantization errors

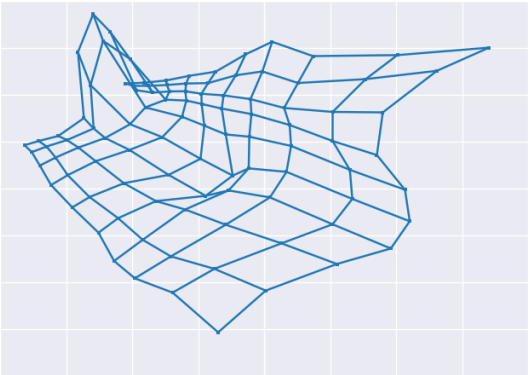

Figure 24: 40 iterations, $\sigma = 1.8$

5.5 Model aggregation

The weighted averaging is simple and has been implemented. The OWA and WOWA are computed using an internal library of Amadeus.

The weights used for WA and WOWA are the F1-score of each algorithm computed thanks to the known frauds manually detected. All algorithms return either a binary output (0 if regular or clustered PNR, 1 if outlier) or a probability between 0 and 1. Those scores are aggregated according to the different aggregation operators to obtain the final scores. By doing so, OWA remains the most unsupervised approach.

Chapter 4

Results

This section details the precision and recall of each algorithm and for each known fraud. This benchmark is processed by tuning the parameters required by the algorithms in Python. The Spark module has not been benchmarked yet.

Please consider that the precision and recall are measured only by comparing the manually detected known frauds to the outliers found by the algorithm.

$$precision = \frac{|known\ frauds\ correctly\ flagged\ as\ outliers|}{|outliers|} \tag{4.1}$$

$$recall = \frac{|known\ frauds\ correctly\ flagged\ as\ outliers|}{|known\ frauds|} \tag{4.2}$$

Since those measures are based on an incomplete manual labelling and since we are also interested in discovering new frauds, it is important to note that those benchmarks only show estimations and that we do not target a precision of 100%. Such a figure is the objective of supervised learning. Yet, we can use precision and recall to select the most efficient models and their parameters. Note however that the proportion of Fraud 4 in the known frauds has an important impact on those metrics.

1 Unsupervised algorithms

1.1 DBSCAN

DBSCAN has been benchmarked using the Euclidean and Mahalanobis distances. Parameters are the neighborhood radius (eps) and the minimum number of points in a cluster (minClusterSize)

Euclidean distance

Figure 1 shows one precision recall curve per minClusterSize. The measurements for each curve are obtained by using different eps.

Figure 2 shows the detail of the precision recall curve having the highest F1 score. This diagram contains the recall of each known fraud and the percentage of outliers.

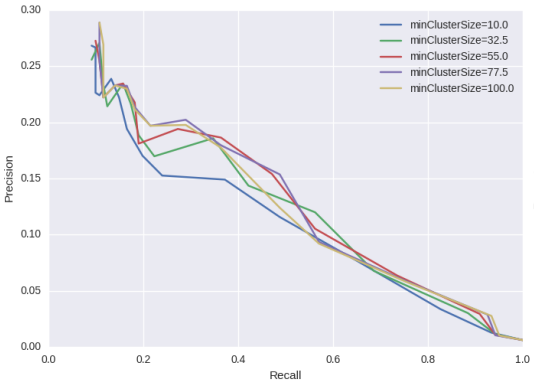

Figure 1: DBSCAN (Euclidean)

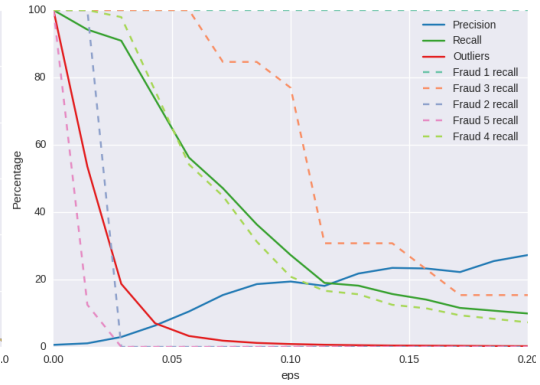

Figure 2: DBSCAN (Euclidean) - minClusterSize=55, best F1 score=0.25

We can hardly find a trend in the precision recall plot, except that allowing clusters containing too few data points will group clouds of outliers together and thus ignore some known frauds. With a minimum cluster size of 55 and an eps of 0.086, we reach a F1 score of 0.25. This is obtained for a precision of 18.64% and a recall of 36.36%, where 236 outliers are detected and all the other data points are clustered together. All the 3 Fraud 1 are found, and we have a good recall on the Fraud 3. One third of the Fraud 4 are found, but the two other frauds are lost.

Those results could be better, and the part of unknown data outliers may be too important to be analyzed.

Mahalanobis distance

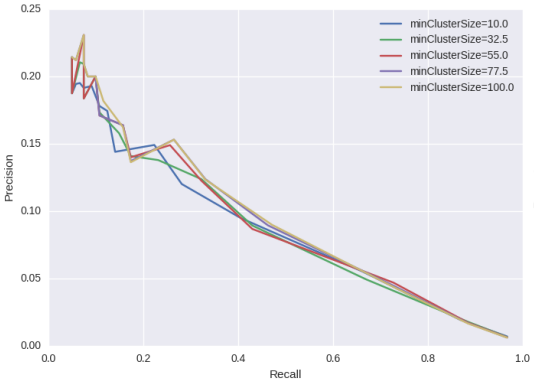

Figure 3: DBSCAN (Mahalanobis)

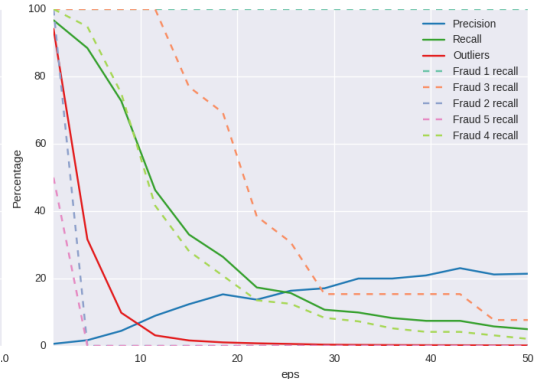

Figure 4: DBSCAN (Mahalanobis) - minClusterSize=77.5, best F1 score=0.19

We though that the Mahalanobis distance would allow a more efficient outlier detection, but its application in the DBSCAN algorithm gives the results depicted in figures 3 and 4 for which the maximum F1 score reached is only 0.19 (15.31% precision and 26.15% recall).

This metric is known to be useful for identifying outliers in data following multivariate normal distributions. However, most of our features have a different kind of distribution, e.g. exponential, which may be the reason of those worse results.

1.2 MeanShift

MeanShift has been benchmarked with a Gaussian kernel. Figure 5 shows that the number of seeds used to initialize the algorithm have little influence on the results.

With a maximum F1 score of 0.21 (13.03% precision, 58.68% recall), MeanShift does not perform better than DBSCAN. Both algorithms must deal with high-dimensional data which is likely to affect the detection of dense clouds of data points.

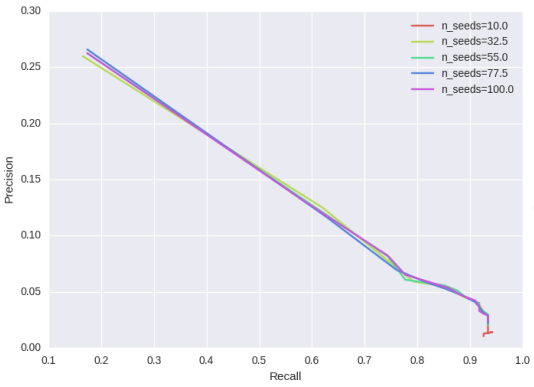

Figure 5: MeanShift

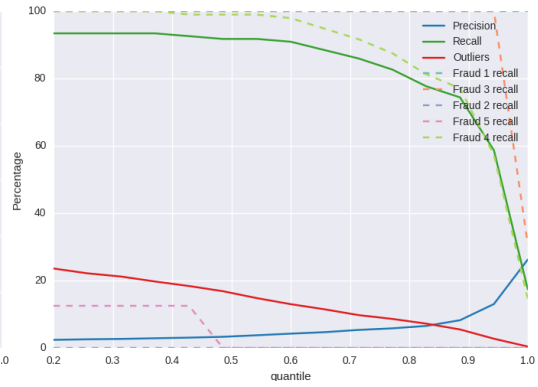

Figure 6: MeanShift - n_seeds=100, best F1 score=0.21

1.3 Gaussian Mixture Model (GMM)

Like MeanShift, GMM targets globular clusters by fitting a mixture of Gaussians to the dataset. According to the precision recall curves, two Gaussian components are enough to cluster most of the data points since we reach a F1 score of 0.28 (23.24% precision, 35.45% recall) with this configuration and a threshold of 193 (outliers are PNRs having a log probability under the model lower than this threshold).

The precision recall curves are very similar to DBSCAN, but we chose to use two components where DBSCAN only found one cluster. This representation is probably more accurate and explains the better performances.

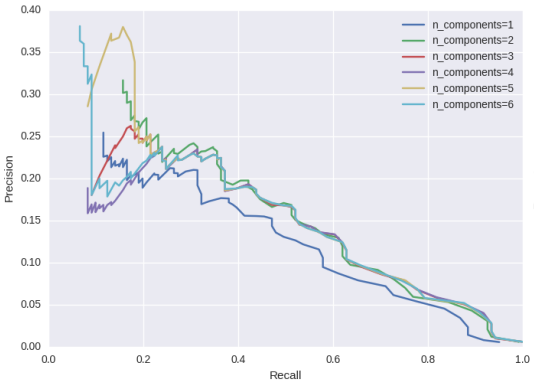

Figure 7: GMM

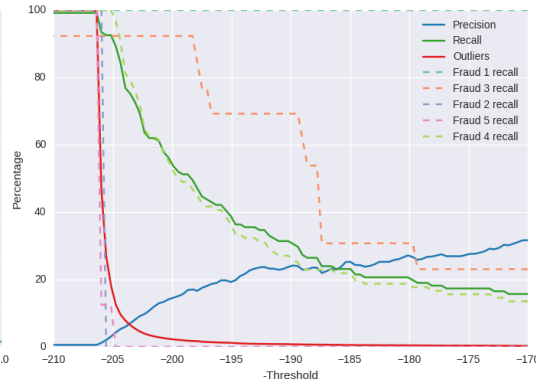

Figure 8: GMM - n_components=2, best F1 score=0.28

1.4 One-class SVM

As detailed in the theory section, the one-class SVM is supposed to be trained with a dataset exempt of unusual behaviors. One of the goal of our final model is to be able to evolve with new use cases. In addition, even if we wanted to remove manually labelled frauds from our dataset, unknown frauds could not be discarded.

This is why we did not filter out PNRs from the dataset, making use of the slack of the one-class SVM to ignore most of the outliers. This experiment showed results similar to the ones previously obtained, with little influence for the parameter γ.

A F1 score of 0.26 has been reached for a precision of 23.49% and a recall of 28.93%.

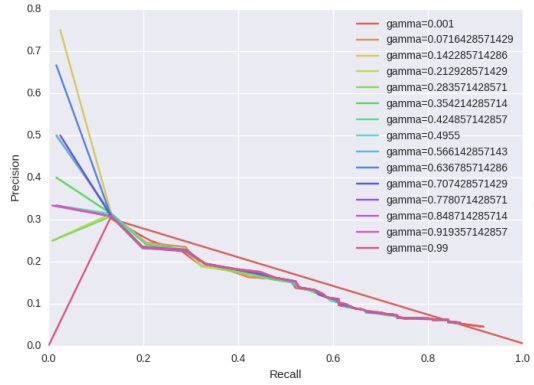

Figure 9: One-class SVM

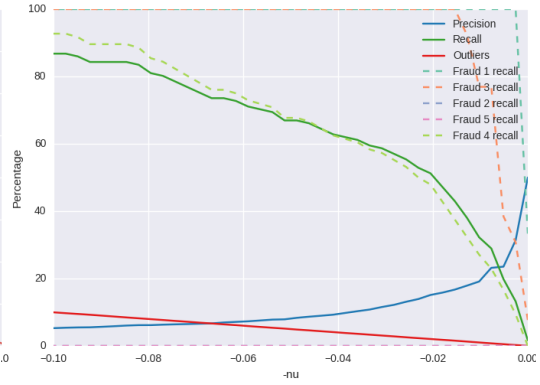

Figure 10: One-class SVM - γ=0.07, best F1 score=0.26

1.5 Z-Score

Z-Score is a special model since it focuses on the most outlying feature of each data point and thus ignore all the others. It is also supposed to be applied to Gaussian distributions.

Nevertheless, a F1 score of 0.29 (33.71% precision, 24.76% recall) was reached for a threshold of 19.32 (minimum number of standard deviations between the average and the value for the outliers). Results are still similar to the ones of the previous algorithms, but using the Z-Score allowed us to easily recognize Fraud 4 and Fraud 3 which have extreme values on some of their features.

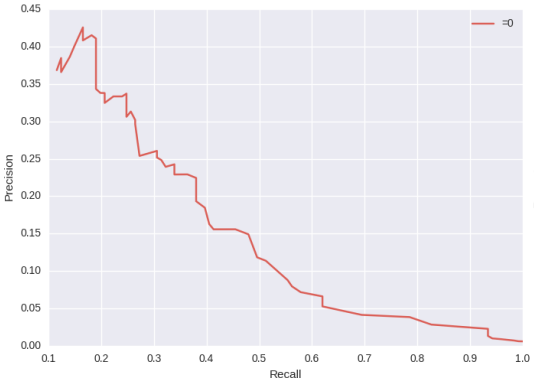

Figure 11: Z-Score

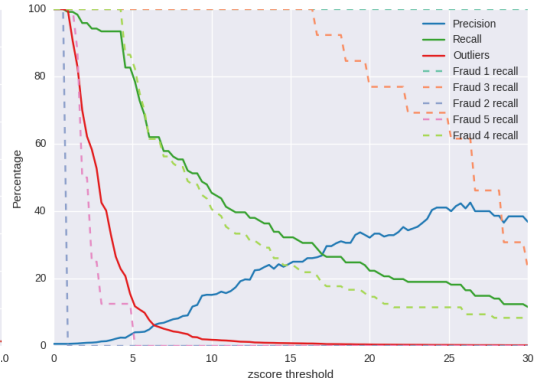

Figure 12: Z-Score - Best F1 score=0.29

1.6 Hierarchical clustering

The sizeDiff method for hierarchical clustering definitely returns the best results. We reach a F1 score of 0.35 (27.11% precision, 50.41% recall) for a probability threshold of 0.91.

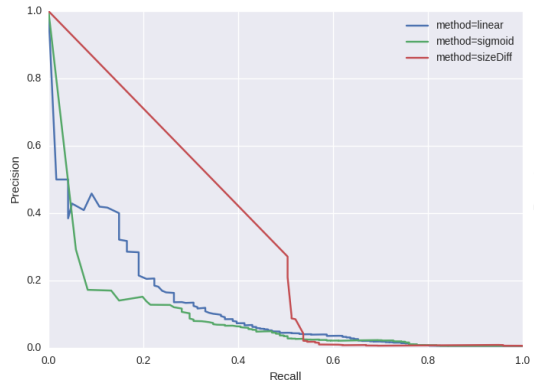

Figure 13: Hierarchical clustering

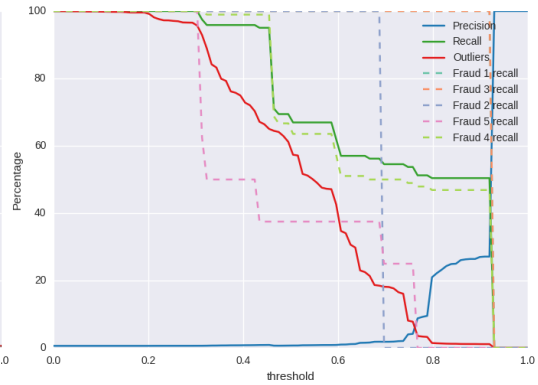

Figure 14: Hierarchical clustering - method=sizeDiff, best F1 score=0.35

1.7 Hidden Markov Model (HMM)

HMMs returns average results, with a F1 score of 0.26 (24.44% precision, 27.27% recall) when long sequences are disadvantaged (figures 15 and 16). However, normalizing the probabilities by the length return very poor results (0.06 F1 score, figures 19 and 18). In this case, Fraud 5 is the most easily fraud found, probably because of the absence of an action usually performed in most of the PNRs. Yet, a manual check on the outliers triggered when normalizing by the sequence lengths showed very rare use cases (car, hotel and cruise bookings) and identified some bots.

Length is not the only factor here, we tried to apply z-score considering only the number of actions and obtained half-good results. However, we noticed that Fraud 1 and Fraud 4 are indeed PNRs with a usually high number of actions.

Note that results may be improved if the actions were not reordered for each envelope by the parser.

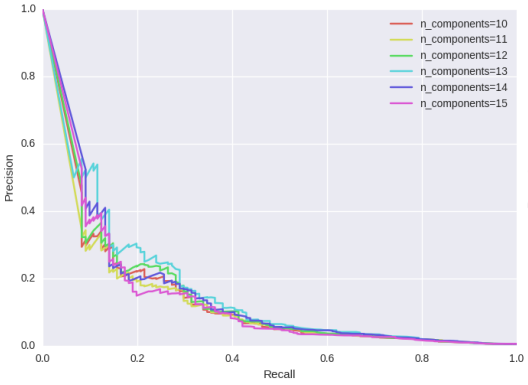

Figure 15: HMM (log probability)

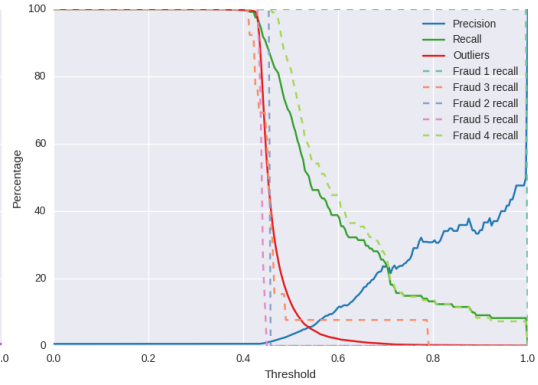

Figure 16: HMM (log probability) - n_components=13, best F1 score=0.26

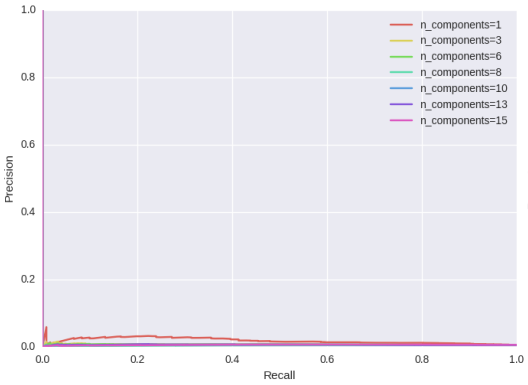

Figure 17: HMM (length normalization)

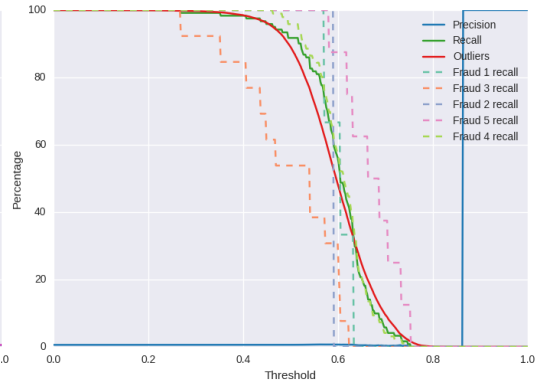

Figure 18: HMM (length normalization) - n_components=15, best F1 score=0.06

1.8 Self-Organizing Map (SOM)

The two parameters used by SOM to detect outliers are the thresholds applied to neurons (MID threshold) and PNRs (QE Z-Score threshold). To detect PNRs lying far from their closest neuron, we applied the threshold to the Z-Score of the quantization error (Euclidean distance between the PNR and the winning neuron). The figures below show that a high value for both thresholds is required to have good results.

Those results are quite good. SOM has a F1 score of 0.42 (27.64% precision, 84.30% recall), detecting 370 outliers with MID threshold = 0.75 and QE Z-Score threshold = 5.0. This efficient detection can be partly explained by the plot of the MID matrix which showed that most of the known frauds were projected on outlying neurons. SOM is also the only algorithm that found Fraud 2.

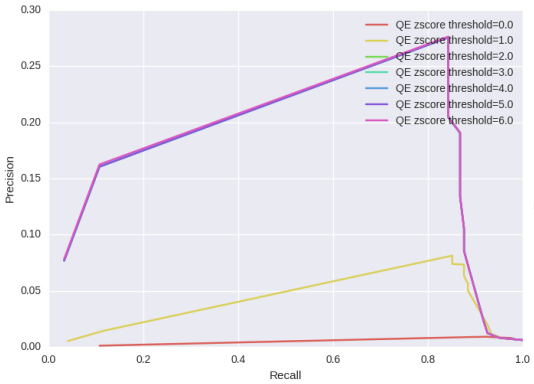

Figure 19: SOM

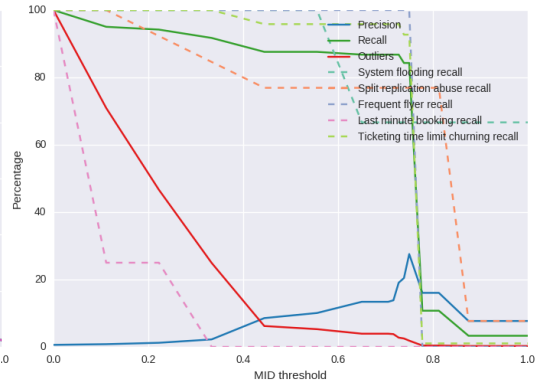

Figure 20: SOM - QE Z-Score threshold=5.0, best F1 score=0.42

During the training of the model, we have noticed that the quality of the results could vary, depending on the random initialization of the weights. Increasing the number of iterations did not solve this local minima issue, so we fixed the seed of the random generator in order to perform stable benchmarks.

2 Model aggregation

Ensemble learning often gives better results than single weak learners. This is beyond doubt for any aggregation operator used below. The weights used for weighted averaging (WA) and WOWA are the F1 score of the algorithms. Aggregation has been performed using the three best algorithms we had, namely SOM, hierarchical clustering and Z-Score. The configuration for which the highest F1 score is reached has been chosen for SOM and hierarchical clustering. Z-Score does not need any parameter.

These aggregations get quickly rid of the PNRs which were not considered as outliers by at least two of the tree algorithms. By increasing the threshold, i.e.

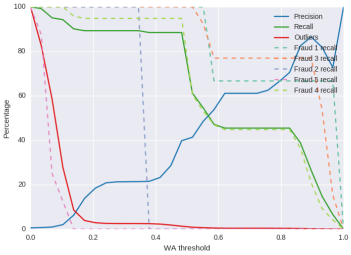

Figure 21: WA - F1s=0.55

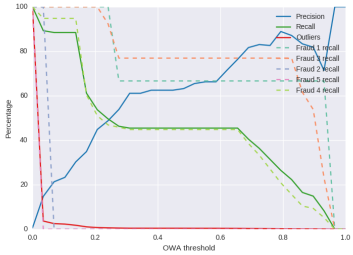

Figure 22: OWA - F1s=0.57

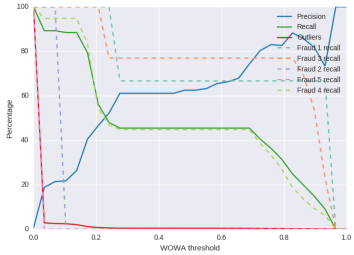

Figure 23: WOWA - F1s=0.56

Aggregation operator	Best F1 score	Precision	Recall	Outliers	Threshold
WA	0.55	39.78	88.43	269	0.482
OWA	0.57	76.39	45.45	72	0.655
WOWA	0.56	74.32	45.45	74	0.689

flagging only the data points having a high outlyingness, we are able to reach a very good precision for an acceptable recall (more than 45%) and a F1 score of 0.57. If completeness matters, we can also select a lower threshold and still have about 40% precision for 88% recall.

If those three aggregation methods give better results than any single algorithm, the similarity of their results added to the uncertainty of our measures do not allow us to rank them with confidence.

3 Self-Organizing Map and Z-Score

Excellent results were obtained using SOM, Z-Score and hierarchical clustering. However, the last algorithm cannot predict the outlyingness of a PNR which was not presented during the training since it requires to compute the pairwise distance between all PNRs. To the opposite, the model trained by SOM and Z-Score can be stored and thus allows us to make predictions without the need of training. The representation of a SOM is the two thresholds and the weight matrix describing the neurons of the neural network, and the Z-Score model is simply a dictionary containing the average and standard deviation of each feature.

Since one of the goal of our fraud detection prototype is to handle streaming data, we choose to benchmark the weighted average of the SOM and Z-Score. The results are plotted in figures 24 and 25.

Hierarchical clustering was excluding many PNRs. By removing this algorithm from the aggregation, we reach the best and final F1 score of 0.63 (50.75% precision, 83.47% recall). This score is obtained with an aggregation threshold of 0.746, using an outlying neuron threshold of 0.81 and a QE Z-Score threshold (used by SOM) of 3.75.

The 20 000 feature vectors are loaded from the database in 1min 15s. If a cache

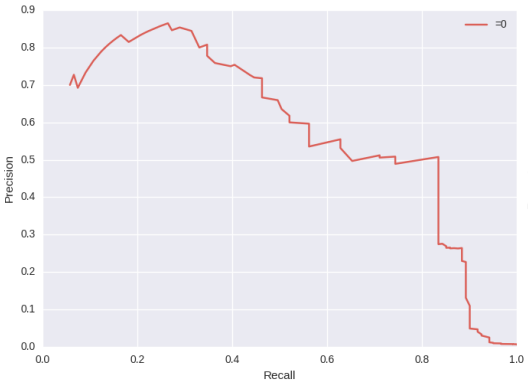

Figure 24: SOM & Z-Score

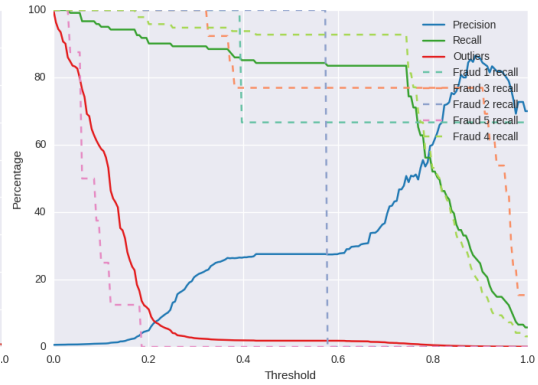

Figure 25: SOM & Z-Score - Best F1 score=0.63

is used, only 3s 500ms are needed. The algorithm was trained in 5min 20s 530ms: 30ms for Z-Score and 5 min 20s 500ms for SOM.

The recall per fraud for this configuration is detailed in the table below.

Fraud	Proportion	Recall
Fraud 1	$^2/_3$	66.67
Fraud 3	$^{10}/_{13}$	76.92
Fraud 2	$^0/_1$	0.00
Fraud 5	$^0/_8$	0.00
Fraud 4	$^{89}/_{96}$	92.71

We had a look at the first 16 outliers returned by the algorithm, it is a small sample but a thorough investigation of each PNR requires some time. 10 of them were frauds belonging to classes of frauds known by Amadeus. 3 of them were new types of frauds and one was a rebooking which cannot be differentiated from one of the known frauds according to the current features. The two others were regular PNRs corresponding to very unusual use cases including actions never performed on other PNRs.

To confirm the quality of our result, we have applied the same model to the remaining 20 183 PNRs excluded from the initial dataset. Those PNRs contain 2 Fraud 1, 8 Fraud 5, 90 Fraud 4, 4 Fraud 2 and 9 Fraud 3. Thanks to the shuffling and random selection applied to build the training dataset, we can observe a similar distribution for this **testing dataset**.

The predictions with the same configuration as previously described showed a F1 score of 0.49, for a precision of 41.25% and a recall of 60.00% (0% for Fraud 5, 0% for Fraud 1, 25.00% for Fraud 3, 25% for Fraud 2 and 71.59% for Fraud 4). If the performances are slightly lower than for the training set, they still show that

the models trained are not overfitted and could be applied to new PNRs to detect frauds.

7.32 seconds were required to score the testing dataset, resulting in a prediction capacity of 2 730 PNRs per second.

Depending on whether the priority is a high precision or the completeness of the detection, different configurations could also be used based on the benchmarks plotted.

Distributed implementation

SOM and Z-Score were implemented on Spark. If Z-Score is deterministic and returns the same results, the distributed SOM reached a F1 score of 0.56 (43.66% precision, 76.86% recall (100% Fraud 1, 93.75% Fraud 4)). The aggregation of both algorithms returned a F1 score of 0.66 (55.43% precision, 80.17% recall (100% Fraud 1, 23.08% Fraud 3, 94.76% Fraud 4)).

The distributed implementation is hence faster and at least as efficient.

4 Summary

The best precision recall of each algorithm on the training set, including the final aggregation, is plotted in figure 26. Triangles show the highest F1 score of each algorithm.

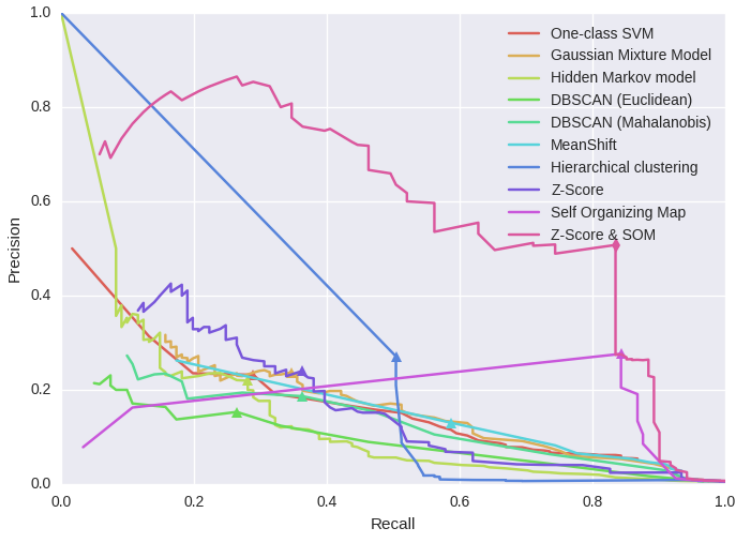

Figure 26: Outlier

Algorithm	Best F1 score	Precision	Recall
DBSCAN (Euclidean)	0.25	18.64	36.36
DBSCAN (Mahalanobis)	0.19	15.31	26.45
MeanShift	0.21	13.03	58.68
GMM	0.28	23.24	35.54
One-class SVM	0.26	23.49	28.93
Z-Score	0.29	33.71	24.79
Hierarchical	0.35	27.11	50.41
HMM	0.26	24.44	27.27
SOM	0.42	27.64	84.30
Distributed Z-Score	0.29	33.71	24.79
Distributed SOM	0.56	43.66	76.86
Z-Score & SOM	0.63	50.75	83.47
Distributed Z-Score & SOM	0.66	55.43	80.17

Chapter 5

Conclusion

This study has benchmarked 8 unsupervised machine learning algorithms using different metrics and outlier detection techniques. Those algorithms have been aggregated using 3 different operators, which produced a scalable model able to handle streaming data while performing an efficient fraud detection.

The outliers detected by the final model are sorted according to their outlyingness which allows a fast insight of the most critical situations.

The computation time of the overall process has an important bottleneck caused by the queries made to the database containing the PNR envelopes. These queries take about 96% of the total time and limit the computational capacity of the prototype.

Most of the remaining time is consumed by the feature extraction, for which a parallel implementation could be easily provided. The training of the algorithms requires little time and can be done only once or repeated once a month if needed (e.g. if new major use cases change the distribution of the features), while the predictions are very fast.

To facilitate the production launch of the prototype and reduce the computation time required by the training and predictions, a distributed implementation of the most efficient algorithm has been provided.

This study performed an estimation of the percentage of fraudulent PNRs for each known fraud in the Amadeus GDS.

New frauds were discovered, such as the delay of purging date of a PNR, and new samples of known frauds which were not labelled as such were found by the prototype.

Eventually, we have seen that Fraud 2 and Fraud 5 have a feature distribution similar to regular PNRs and are thus hard to detect.

Chapter 6

Future work

A few points may be improved in the current prototype to build a production ready product.

First, a faster access to the PNR history must be provided in order to reach a reasonable number of transactions per second. A subscription to a live feed coupled with a PNR database dedicated to the targeted markets may by a solution.

The computation time required by the feature extraction could be reduced by making it parallel and even distributed if needed. With a few processes, this time will quickly become negligible.

The distributed implementation should be completed with the weighted average of Z-Score in order to increase the performances on Spark.

Some features should be refined, for example the ones based on timestamps for which time zones should be retrieved using the IATA code of the corresponding airports or points of sale. Rebookings are sometimes confused with an existing fraud. Adding a feature in order to differentiate actions made by the Amadeus systems and external users or systems would improve the results.

This prototype could be redesigned into a product and made available for travel providers. Applied to PNRs approved by a preliminary supervised prediction, it would provide data samples to improve the supervised algorithms and raise alerts for unusual PNRs.

To efficiently recognize fraudulent PNRs based on those alerts, an ergonomic Web user interface possibly based on a box plot should be developed.

If the prototype detects new types of frauds, the previous plot could help in selecting features in order to retrieve similar samples.

The figure below shows how reducing the number of features can increase the detection of Fraud 4 with DBSCAN. The 42 features used are the ones related to travel segments while the 3 selected features are the ones containing only extreme

values for Fraud 4. By studying the profile of a fraud, one may thus be able to build a dataset which could be fed to a supervised algorithm.

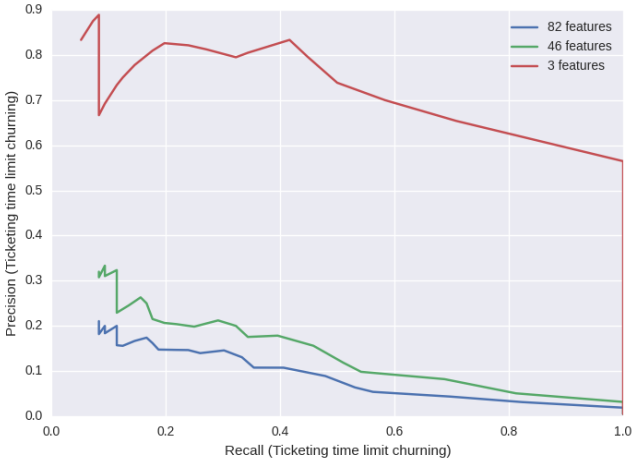

Figure 1: Detection of Fraud 4

References

[1] Leonard E Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The annals of mathematical statistics*, pages 164–171, 1970.

[2] Miguel A Carreira-Perpinan. Gaussian mean-shift is an em algorithm. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(5):767–776, 2007.

[3] Dorin Comaniciu and Peter Meer. Mean shift: A robust approach toward feature space analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(5):603–619, 2002.

[4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[5] Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological cybernetics*, 43(1):59–69, 1982.

[6] Richard D. Lawrence, George S. Almasi, and Holly E. Rushmeier. A scalable parallel algorithm for self-organizing maps with applications to sparse data mining problems. *Data Mining and Knowledge Discovery*, 3(2):171–195, 1999.

[7] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[8] Alberto Muñoz and Jorge Muruzábal. Self-organizing maps for outlier detection. *Neurocomputing*, 18(1):33–60, 1998.

[9] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[10] Tugdual Sarazin, Hanane Azzag, and Mustapha Lebbah. Som clustering using spark-mapreduce. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1727–1734. IEEE, 2014.

[11] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. Estimating the support of a high-dimensional distribution. *Neural computation*, 13(7):1443–1471, 2001.

[12] David W Scott. *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons, 2015.

[13] Simon J Sheather and Michael C Jones. A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 683–690, 1991.

[14] Lindsay I Smith. A tutorial on principal components analysis. *Cornell University, USA*, 51:52, 2002.

[15] Luis Torgo. Resource-bounded fraud detection. In *Progress in Artificial Intelligence*, pages 449–460. Springer, 2007.

[16] Luis Torgo and Maintainer Luis Torgo. Package 'dmwr'. 2013.

[17] Vicenç Torra. The weighted owa operator. *International Journal of Intelligent Systems*, 12(2):153–166, 1997.

[18] Joe H Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

[19] Ronald R Yager. On ordered weighted averaging aggregation operators in multicriteria decisionmaking. *Systems, Man and Cybernetics, IEEE Transactions on*, 18(1):183–190, 1988.

Appendix A

Envelope (EDIFACT)

Listing A.1: Envelope (EDIFACT)

```
UNH+1+PASBCQ:08:1:1A'&
STX+ACT'&
IMD+NM'&
EMS++ :1++0'&
IRV++++:::2301C6D200000C55'&
BLB+29+E+TIF+AIL INTERRUPT+CANCEL SSR'&
'&
IMD+RF'&
EMS++ :0++0'&
BLB+80+E+LFT++A'&
LFT++OFFICEID'&
SDI+++2015:7:22:8:33'&
TID+ATID+1AOFFICEID:4'&
UID+DUTYCODE'&
'&
IMD+AP'&
EMS++ :1++0'&
BLB+11+E+LFT+3:AP+1'&
'&
IMD+AIR'&
EMS++ :1++0'&
BLB+353+E+STX+NGI:1*CAB:T'&
TVL+300715:1930:300715:2250+FRA+HEL+6X+3613:Y+ET'&
SDI+DTB+GMT+2015:7:22:8:33'&
SDI+1LG+LOC+2015:7:30'&
SDI+KDT+GMT+2015:7:22:8:33'&
RPI++NN*NN'&
STX+CTJ:1'&
MSG+1'&
```

```
RCI+6X:  '&
RPI+1+HK'&
RPI++LK'&
APD+733:0:::4'&
SDT+P2:LDS'&
CTD+C:6X:FR+Y:0:I+0'&
UID+91496716:OFFICEID+A'&
SYS++1A:NCE:NCE'&
PRE+FR'&
ABI++:OFFICEID'&
SYS+++DCD  1  0  0'&
UID+:OFFICEID++AGENTSIGN'&
PTD+0'&
'&
 [...]
'&
UNT+76+1'
```