Reduction Dimension and outlier detection applied to financial data

Arthur DI BETTA - Mathis GUENET - Alexandre HULLARD - Hadrien DOUIN

Mars 2021

Contents

1 Introduction

We are the group 76 of PI² of the 2022 promotion of ESILV. 4 members of the IF major make up the team : Arthur DI BETTA, Alexandre HULLARD, Mathis GUENET, Hadrien DOUIN. We would like to thank Mr. Jiang PU who accompanied us throughout the project through our many meetings.

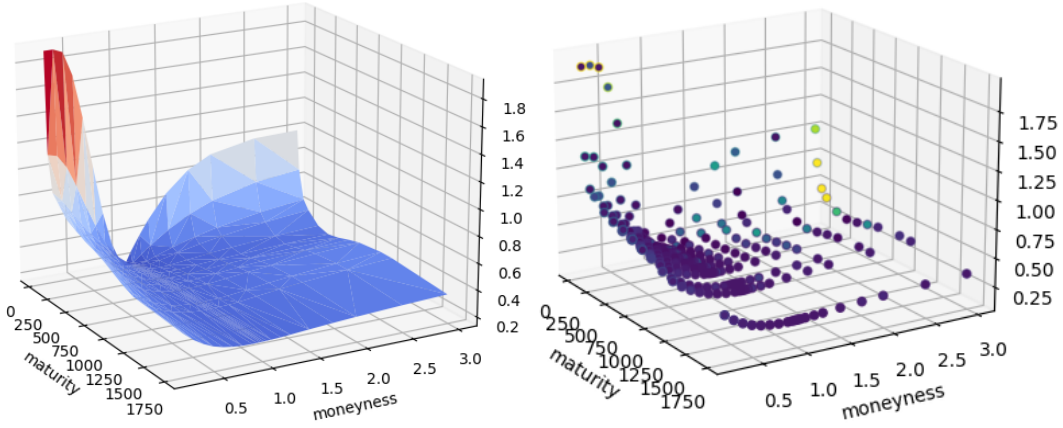
The choice of this subject (Dimension reduction and outlier detection applied to financial data) adhered perfectly with our school orientations. It allowed us to learn more about the different anomalies that finance can generate and resolve them through machine learning methods. The goal is to be able to reuse this knowledge in our future internships.

1.1 Basic Concepts

First of all, we would like to introduce basic concepts definitions that may be useful to understand our project :

- Machine learning: It is an artificial intelligence technology that allows computers to learn without being explicitly programmed to this purpose.
- Outlier: There exists many definitions of an outlier. We chose the Hawkin's one because it fitted well the concept in our case "An outlier is an observation which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism".
- Volatility surface and implied volatility: The volatility surface is a three-dimensional plot of the implied volatility of a stock option. Implied volatility exists due to discrepancies with how the market prices stock options and what stock option pricing models (Black-Scholes) say the correct prices should be.
- Outlier for volatility surface: A volatility surface is said normal if there is no arbitrage opportunity within this surface and it is said abnormal if there exists an arbitrage opportunity within this surface

Figure 1: Surface plot and scatter plot of normal volatility surface



1.2 Problematic

Nowadays, outlier detection is used in many sectors as fraud detection, medicine or public health. In our project, we focused on the volatility surface case. The purpose of our project is to try to give an answer to the following problematic :

How can we detect outliers in a volatility surface dataset of high dimension ?

1.3 Etat de l'art

In Machine learning, we can distinguish two general scenarios, the supervised and the unsupervised learning :

- Supervised learning : The input and output of the model are known before developing the model. We can train the model on the provided data.
- Unsupervised learning : Only the input is known, so we can't train the model. This is the most common scenario.

We use unsupervised learning method since we don't know if our volatility surfaces are normal or abnormal yet.

We decided to explore the existing methods for outlier detection and clustering. We distinguished different approaches among those algorithms :

- Labelling: We have a binary output (normal or abnormal observation).
- Scoring: We compute the probability for being an outlier for each object and we sort the data according to their scores (the probability).
- Model-based: We apply a model to represent normal data points and if some points don't fit the model, we consider them as outliers. Probabilistic and statistical tests and models are used.
- Proximity-based: We consider the spatial proximity of objects to determine if they are outliers or not. We can use distance and density-based approaches.

After considering these different approaches, we implemented four algorithms on simulated 2D datasets to test the efficiency, advantages and disadvantages of these algorithms. You can find below some graphs and tables that summarize the principal clustering methods.

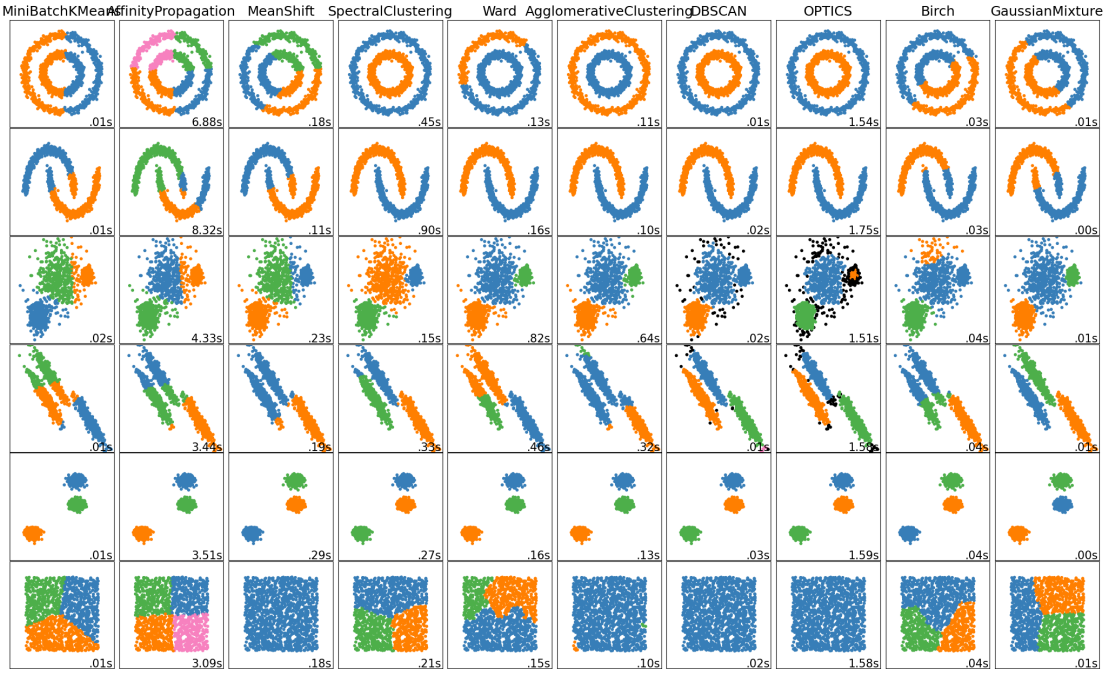
In particular, we implemented K-Means, Hierarchical clustering, DBSCAN and Isolation Forest. The two first quoted were finally not adapted for outlier detection and were not working on every data shape (convex data for example). K-Means and Hierarchical clustering are more clustering algorithms than outlier detection ones. On the contrary, DBSCAN and Isolation Forest are more robust and adapted for outlier detection. We decided to keep these two methods and reject the two other ones.

In the next part, we will focus on DBSCAN and Isolation Forest methods, give theorical analysis and explanation and expose practical tests.

Figure 2: Summary principal clustering methods

Method name	Parameters	Scalability	Usecase	Geometry (metric used)
K-Means	number of clusters	Very large `n_samples`, medium `n_clusters` with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with `n_samples`	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium `n_samples`, small `n_clusters`	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters or distance threshold	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters or distance threshold, linkage type, distance	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large `n_samples`, medium `n_clusters`	Non-flat geometry, uneven cluster sizes	Distances between nearest points
OPTICS	minimum cluster membership	Very large `n_samples`, large `n_clusters`	Non-flat geometry, uneven cluster sizes, variable cluster density	Distances between points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers
Birch	branching factor, threshold, optional global clusterer.	Large `n_clusters` and `n_samples`	Large dataset, outlier removal, data reduction.	Euclidean distance between points

Figure 3: Summary principal clustering methods



2 Outlier detection

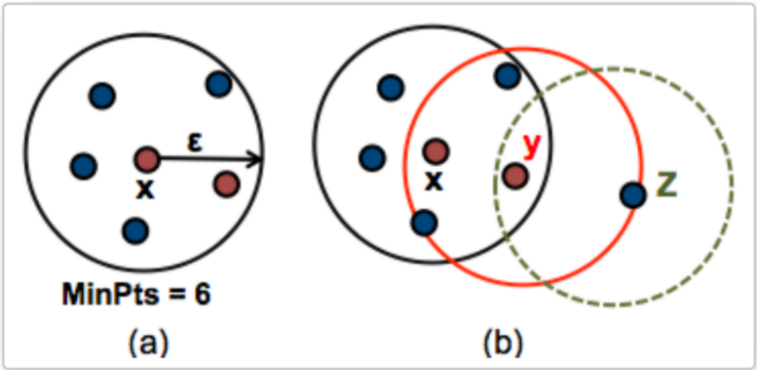
2.1 DBSCAN

DBSCAN is a clustering algorithm based on the density approach (spatial proximity approach). It provides clusters of different size and isolated points that don't belong to any cluster and can be considered as outliers.The algorithm takes 2 arguments as input :

- ξ (epsilon) : the minimum neighborhood radius
- MinPts : the minimum number of points to form a cluster

A point which has at least MinPts points in its neighborhood of ξ radius is called a core point. We call X a border point if the number of points in its neighborhood is less than MinPts. You can see below a little drawing representing a simple iteration where x is a core point :

Figure 4:



To have a better understanding of the algorithm, we need to define 3 concepts :

- Direct density reachable : A point X is directly density reachable to another point Y if X belongs to the circle of center Y where Y is a core point.
- Density reachable : A point X is density reachable to Y if there exists a core points group joining Y to X.
- Density connected : X and Y are density connected when there exists a core point Z from where points X and Y are density reachable.

For example, X and Y are density reachable and X and Z are density connected on the drawing above.

The clusters provided by the algorithm are density-based clusters, which means that every cluster is a set of density connected points.

The algorithm works as follow :

- For every point x, compute the distance to other points, find among these points those that are to a lower distance to x than ξ (the neighbors of x).
- Every point with at least MinPts neighbors is classified as a core point.

- For every core point that doesn't belong to a cluster, create a new cluster.
- Find all the density connected points to the core point and put them in the same cluster.
- Iterate over the points left.

In summary, the main advantages of this method is that it is a very flexible method so it adapts well to new data and different data shapes. However, it is costly in term of complexity, memory and it is not well adapted for high dimensions datasets.

Now, we will expose the use of DBSCAN in practical situation. We implemented the algorithm and made few simulations with simulated datasets. After several tests, we decided to fix arbitrarily the parameter MinPts to 20. For the ξ parameter (radius), we used the knee of the kNN plot which is in many cases the best value.

You can find below different plots resulting from our DBSCAN algorithm applied on different dataset shapes (Blue points are the outliers) :

Figure 5: Dispatch Gaussian distribution and Anisotropicly distribution

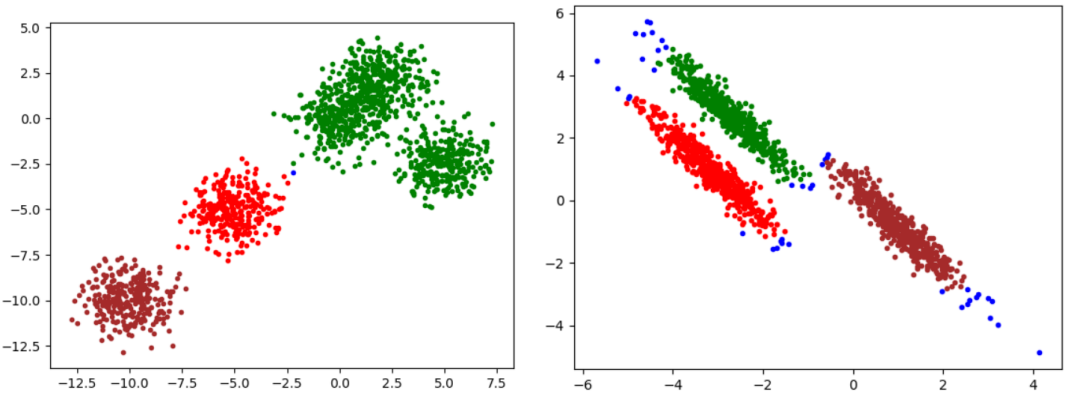
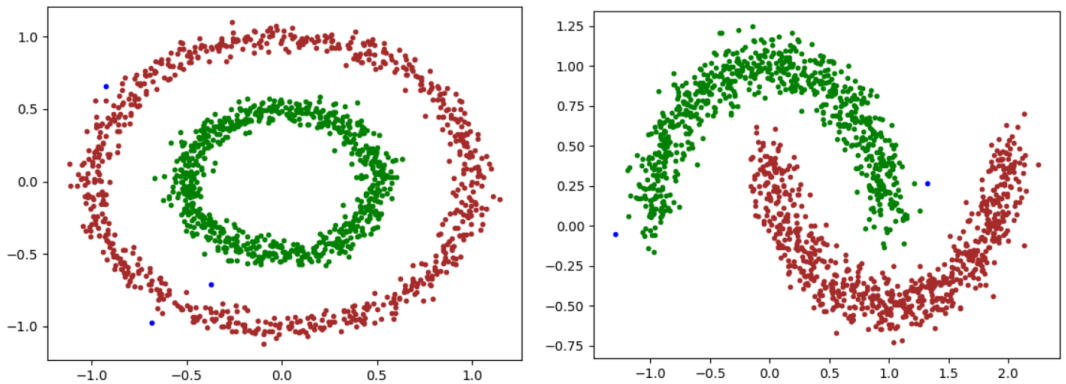


Figure 6: Noisy circle distribution and noisy moon distribution



This is the code for the DBSCAN's main function :

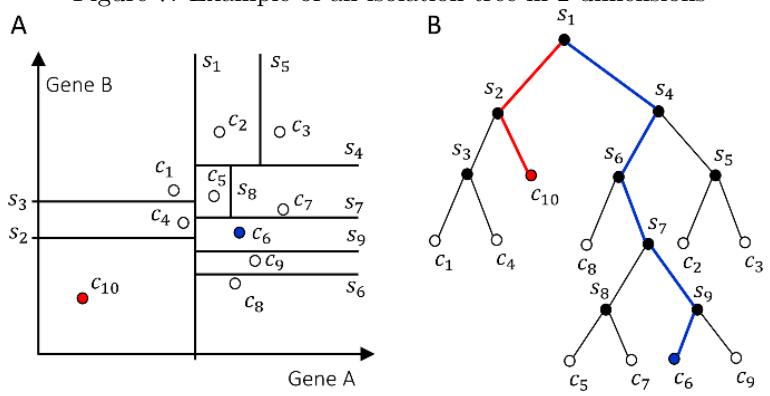
```
def predict_DBSCAN(data):
    k = 10
    NN = NearestNeighbors(n_neighbors = k).fit(data)
    distances, indices = NN.kneighbors(data)
    distanceSorted = sorted(distances[:,k-1], reverse = True)
    radiusArray = np.percentile(distanceSorted,99)
    clustering = DBSCAN(data, radiusArray, 20)
    print('Set radius = ' +str(clustering.radius)+ ', Minpoints = '+str(clustering.MinPt))
    pointlabel, cluster = clustering.dbscan()
    clustering.plotRes(pointlabel, cluster)
    plt.show()
    print('Number of clusters found: ' + str(cluster - 1))
    counter=collections.Counter(pointlabel)
    print(counter)
    outliers  = pointlabel.count(0)
    print('Numbrer of outliers found: '+str(outliers) +'\n')
```

The code for the DBSCAN's class is in the annex.

2.2 Isolation Forest

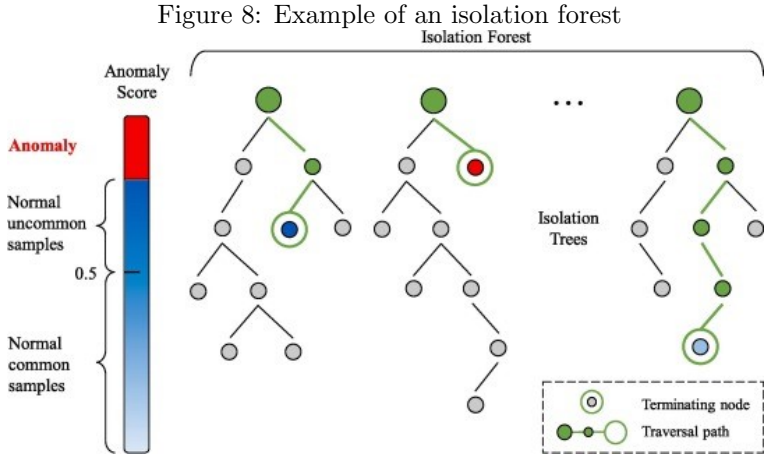
The goal of the Isolation forest algorithm is to isolate observations that are too different from each other by calculating anomaly score for each observation. In order to compute this score, the isolation forest uses the bagging method using randomness. The bagging consists to bootstrap the dataset (create k dataset of length n from the original dataset that keeps the same statistics than the original dataset) and apply for each new dataset an decision tree. The classic decision tree is a hierarchically organized structure, which, at each node, splits data (until a specific length has been reached or there are no more than 2 observation) with the best feature (the one who explained the most) and the best threshold that minimize the variance in the both new leaf of the node . In the particular case of Isolation forest, we use isolation tree that, unlike decision tree, splits data with a random feature and a random threshold.

Figure 7: Example of an isolation tree in 2 dimensions



Now that all data are split in the isolation tree, each observation has an anomaly score. More its score is higher, more the observation is at the top of the tree and more it is different from others observation and more it represent an outlier.

Figure 8: Example of an isolation forest



Here if some examples of two dimensional observations with different distribution :

Figure 9: Dispatch Gaussian distribution and Anisotropicly distribution

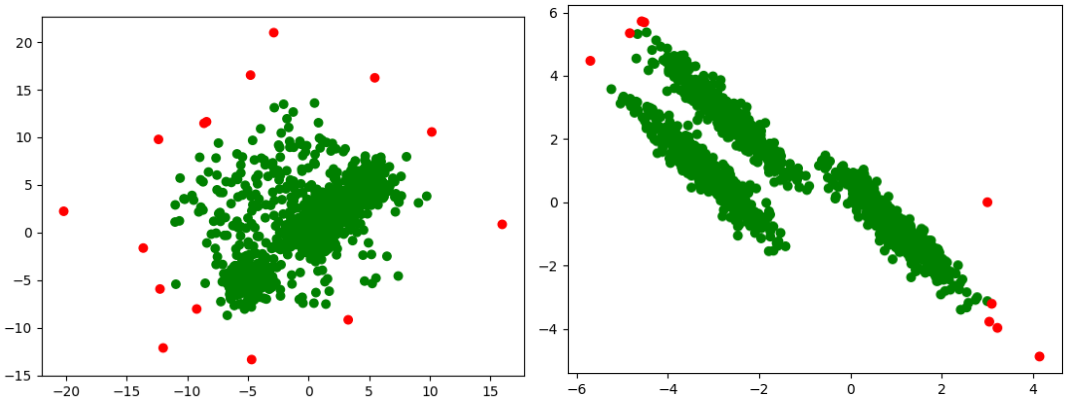
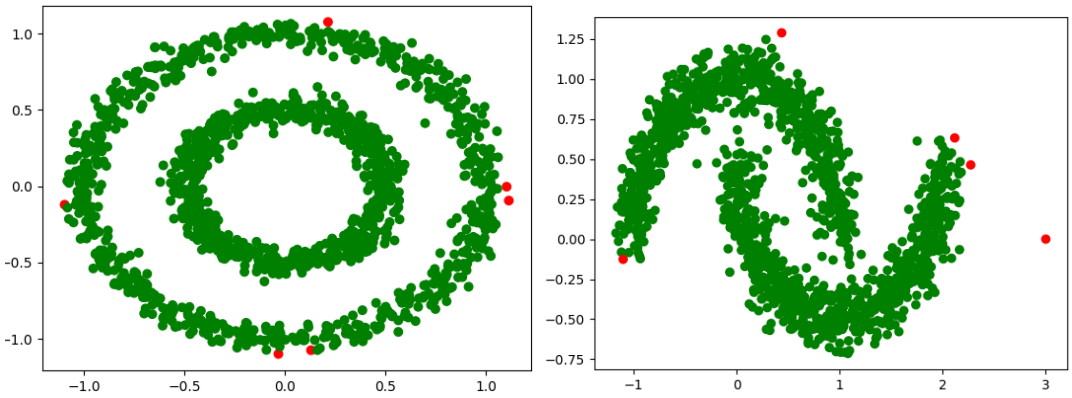


Figure 10: Noisy circle distribution and noisy moon distribution



This is the code for the isolation forest's main function :

```
def predict_IsolationForest(data):
    nbrTree = 100
    forest = IsolationForest(data.shape[0], 100) #Create the isolation forest
    forest.fit(data) #fit data
    forest.path_length(data) #compute mean length in the trees for each observation
    prediction = forest.predict(data, 0.65) #array of prediction's label with respect of the tres
    pltData(data, prediction)
    print(outliersDays(prediction))
```

The code for the Isolation Forest's class is in the annex.

3 Reduction dimension

3.1 Why do we use need to reduce dimension ?

Our dataset is composed of almost 1300 volatility surfaces. Each surfaces represent one business day. So our data represented more than 5 years from 2015 to 2020. The data were stored in into dataFrame (volatility matrix) of around 12 maturities and 21 moneyness. The first problem was that each volatility matrix had different maturities and number of maturities. The second problem was that volatility matrix had a lots of short maturities but not many long maturities.

Figure 11: Original Volatility matrix

nBizDays	Forwards	nCalDays	diff Days	0.2	0.3	0.4	0.5	0.6	0.7	...	1.05	1.1	1.2	1.3	1.5	1.75	2	2.5	3
5	17453.03948	7	0.99999	2.619529	2.247577	1.874444	1.542642	1.257031	1.009347	...	0.339046	0.330423	0.341261	0.37034	0.439454	0.511576	0.567061	0.636824	0.669308
28	17461.68337	42	0.99994	1.845375	1.556973	1.293336	1.067921	0.877184	0.711492	...	0.355313	0.34478	0.336376	0.338593	0.356309	0.377063	0.393646	0.416259	0.428655
48	17461.11965	70	0.99989	1.498521	1.261367	1.055559	0.883417	0.739108	0.612024	...	0.357494	0.350708	0.346266	0.349599	0.367559	0.389636	0.407642	0.433318	0.448658
68	17351.45	98	0.99985	1.323719	1.109531	0.930703	0.783583	0.661026	0.551321	...	0.354536	0.350202	0.34809	0.351689	0.368294	0.389397	0.406823	0.432392	0.44853
109	17365.27423	161	0.99976	1.156712	0.964377	0.811353	0.68793	0.585398	0.492323	...	0.348921	0.347474	0.349056	0.353807	0.37033	0.392995	0.41214	0.441448	0.461456
173	17353.25379	252	0.99963	1.017814	0.846386	0.715504	0.611648	0.524261	0.445081	...	0.341604	0.342778	0.348574	0.355617	0.373491	0.398844	0.421039	0.45622	0.48184
232	17259.44011	343	0.99949	0.924582	0.770077	0.654645	0.563678	0.485154	0.416051	...	0.336825	0.339239	0.346874	0.354953	0.372849	0.398109	0.420954	0.457766	0.485433
352	17132.21254	525	0.99925	0.795002	0.668004	0.575256	0.501409	0.434699	0.381462	...	0.330391	0.333588	0.34211	0.350552	0.366868	0.389116	0.410613	0.446033	0.473586
475	17015.96247	707	0.99896	0.713028	0.605251	0.527341	0.462797	0.404284	0.362021	...	0.326147	0.329413	0.337668	0.345777	0.360492	0.379589	0.398845	0.431464	0.45728
597	16873.38883	889	0.99859	0.659156	0.564603	0.49616	0.436649	0.384619	0.349959	...	0.3234	0.326593	0.334385	0.342029	0.355518	0.372287	0.389474	0.4195	0.443589
721	16744.93914	1071	0.99816	0.620699	0.536019	0.473827	0.417717	0.371274	0.34209	...	0.321586	0.324643	0.331945	0.339125	0.351631	0.36664	0.38207	0.409824	0.432222
842	16601.9412	1253	0.99748	0.59241	0.515203	0.457002	0.403567	0.361865	0.336759	...	0.320707	0.323652	0.330546	0.337316	0.349045	0.362777	0.376839	0.402697	0.423752
972	16482.93095	1442	0.99664	0.569164	0.498383	0.443125	0.392411	0.354998	0.333149	...	0.320428	0.323239	0.329733	0.336122	0.34716	0.359806	0.37266	0.39676	0.416594
1091	16335.26645	1624	0.99561	0.551791	0.485777	0.432365	0.384057	0.35016	0.330816	...	0.320904	0.323641	0.329853	0.335936	0.346435	0.358312	0.370295	0.393019	0.411888
1215	16210.71607	1806	0.99443	0.537252	0.475323	0.423455	0.377633	0.346835	0.329543	...	0.321827	0.324479	0.330422	0.336227	0.34624	0.357439	0.368649	0.390106	0.4081
1456	15969.53217	2170	0.99127	0.515954	0.459976	0.410483	0.369257	0.343372	0.329154	...	0.324807	0.327329	0.332849	0.338203	0.347432	0.357597	0.36763	0.387043	0.403613
1699	15755.05854	2534	0.98708	0.501312	0.449444	0.402071	0.364971	0.342811	0.330814	...	0.328839	0.331246	0.336421	0.341409	0.349999	0.359357	0.368484	0.386244	0.401638
2200	15386.47976	3262	0.97569	0.484255	0.43733	0.394055	0.363685	0.346665	0.337655	...	0.338639	0.340844	0.345464	0.34987	0.357443	0.365572	0.373354	0.388552	0.40285
2975	15143.26389	4361	0.94989	0.475308	0.432527	0.39482	0.370779	0.357731	0.350939	...	0.353123	0.355026	0.358983	0.362765	0.369268	0.376108	0.382486	0.394937	0.406322

[19 rows x 24 columns]

To resolve this probleme, we decided to use the linear interpolation for specific maturities and moneyness.

Figure 12: Volatility matrix after interpolation

nBizDays	Forwards	nCalDays	diff Days	0.2	0.3	0.4	0.5	0.6	0.7	...	1.05	1.1	1.2	1.3	1.5	1.75	2	2.5	3
5	17453.039408	7.000000	0.999990	2.619529	2.247577	1.874444	1.542642	1.257031	1.009347	...	0.339046	0.330423	0.341261	0.370340	0.439454	0.511576	0.567061	0.636824	0.669308
13	17456.046050	19.173913	0.999973	2.350258	2.007367	1.672319	1.377522	1.124910	0.905745	...	0.344704	0.335417	0.339562	0.359298	0.410534	0.464789	0.506743	0.560106	0.585603
28	17461.683370	42.000000	0.999940	1.845375	1.556973	1.293336	1.067921	0.877184	0.711492	...	0.355313	0.344780	0.336376	0.338593	0.356309	0.377063	0.393646	0.416259	0.428655
36	17461.457882	53.200000	0.999920	1.706633	1.438731	1.198225	0.994119	0.821954	0.671705	...	0.356186	0.347151	0.340332	0.342995	0.360809	0.382093	0.399244	0.423083	0.436656
48	17461.119650	70.000000	0.999890	1.498521	1.261367	1.055559	0.883417	0.739108	0.612024	...	0.357494	0.350708	0.346266	0.349599	0.367559	0.389636	0.407642	0.433318	0.448658
58	17406.284825	84.000000	0.999870	1.411120	1.185449	0.993131	0.833500	0.700067	0.581673	...	0.356015	0.350455	0.347178	0.350644	0.367927	0.389517	0.407232	0.432855	0.448594
68	17351.450000	98.000000	0.999850	1.323719	1.109531	0.930703	0.783583	0.661026	0.551321	...	0.354536	0.350202	0.348090	0.351689	0.368294	0.389397	0.406823	0.432392	0.448530
80	17355.496116	116.439024	0.999824	1.274839	1.067047	0.895771	0.755587	0.638891	0.534053	...	0.352892	0.349404	0.348373	0.352309	0.368890	0.390450	0.408379	0.435042	0.452313
109	17365.274230	161.000000	0.999760	1.156712	0.964377	0.811353	0.687930	0.585398	0.492323	...	0.348921	0.347474	0.349056	0.353807	0.370330	0.392995	0.412140	0.441448	0.461456
110	17365.086411	162.421875	0.999758	1.154542	0.962533	0.809856	0.686738	0.584443	0.491585	...	0.348807	0.347401	0.349049	0.353835	0.370379	0.393087	0.412279	0.441679	0.461774
140	17359.451829	205.078125	0.999697	1.089433	0.907225	0.764926	0.650981	0.555785	0.469440	...	0.345377	0.345200	0.348822	0.354683	0.371861	0.395828	0.416450	0.448603	0.471329
173	17353.253790	252.000000	0.999630	1.017814	0.846386	0.715504	0.611648	0.524261	0.445081	...	0.341604	0.342778	0.348574	0.355617	0.373491	0.398844	0.421039	0.456220	0.481840
200	17310.322106	293.644068	0.999566	0.975149	0.811465	0.687653	0.589695	0.506365	0.431796	...	0.339417	0.341158	0.347796	0.355313	0.373197	0.398508	0.421000	0.456927	0.483484
232	17259.440110	343.000000	0.999490	0.924582	0.770077	0.654645	0.563678	0.485154	0.416051	...	0.336825	0.339239	0.346874	0.354953	0.372849	0.398109	0.420954	0.457766	0.485433
300	17187.344487	446.133333	0.999354	0.851154	0.712236	0.609658	0.528392	0.456563	0.396451	...	0.333179	0.336037	0.344174	0.352459	0.369460	0.393013	0.415094	0.451117	0.478720
352	17132.212540	525.000000	0.999250	0.795002	0.668004	0.575256	0.501409	0.434699	0.381462	...	0.330391	0.333588	0.342110	0.350552	0.366868	0.389116	0.410613	0.446033	0.473586
400	17086.846659	596.024390	0.999137	0.763012	0.643515	0.556557	0.486341	0.422830	0.373876	...	0.328735	0.331959	0.340376	0.348689	0.364380	0.385398	0.406021	0.440348	0.467222
475	17015.962470	707.000000	0.998960	0.713028	0.605251	0.527341	0.462797	0.404284	0.362021	...	0.326147	0.329413	0.337668	0.345777	0.360492	0.379589	0.398845	0.431464	0.457280
500	16986.730167	744.295082	0.998884	0.701988	0.596922	0.520952	0.457439	0.400255	0.359550	...	0.325584	0.328835	0.336995	0.345009	0.359472	0.378092	0.396925	0.429013	0.454458
597	16873.388830	889.000000	0.998590	0.659156	0.564603	0.496160	0.436649	0.384619	0.349959	...	0.323400	0.326593	0.334385	0.342029	0.355518	0.372287	0.389474	0.419500	0.443589
721	16744.939140	1071.000000	0.998160	0.620699	0.536019	0.473827	0.417717	0.371274	0.342090	...	0.321586	0.324643	0.331945	0.339125	0.351631	0.366640	0.382070	0.409824	0.432222
800	16651.576849	1189.826446	0.997716	0.602229	0.522428	0.462842	0.408479	0.365131	0.338610	...	0.321012	0.323996	0.331031	0.337944	0.349943	0.364118	0.378655	0.405170	0.426692
842	16601.941200	1253.000000	0.997480	0.592410	0.515203	0.457002	0.403567	0.361865	0.336759	...	0.320707	0.323652	0.330546	0.337316	0.349045	0.362777	0.376839	0.402697	0.423752
900	16548.844319	1337.323877	0.997105	0.582038	0.507699	0.450811	0.398590	0.358801	0.335148	...	0.320582	0.323468	0.330183	0.336783	0.348204	0.361452	0.374974	0.400048	0.420558
972	16482.930950	1442.000000	0.996640	0.569164	0.498383	0.443125	0.392411	0.354998	0.333149	...	0.320428	0.323239	0.329733	0.336122	0.347160	0.359806	0.372660	0.396760	0.416594
1091	16335.266450	1624.000000	0.995610	0.551791	0.485777	0.432365	0.384057	0.350160	0.330816	...	0.320904	0.323641	0.329853	0.335936	0.346435	0.358312	0.370295	0.393019	0.411888
1150	16276.004576	1710.596774	0.995049	0.544873	0.480803	0.428126	0.381001	0.348578	0.330210	...	0.321343	0.324040	0.330124	0.336074	0.346342	0.357897	0.369512	0.391633	0.410086
1215	16210.716070	1806.000000	0.994430	0.537252	0.475323	0.423455	0.377633	0.346835	0.329543	...	0.321827	0.324479	0.330422	0.336227	0.346240	0.357439	0.368649	0.390106	0.408100
1400	16025.574902	2085.419087	0.992004	0.520903	0.463542	0.413497	0.371203	0.344176	0.329244	...	0.324114	0.326667	0.332285	0.337744	0.347155	0.357560	0.367867	0.387755	0.404656
1456	15969.532170	2170.000000	0.991270	0.515954	0.459976	0.410483	0.369257	0.343372	0.329154	...	0.324807	0.327329	0.332849	0.338203	0.347432	0.357597	0.367630	0.387043	0.403613
1550	15886.567062	2310.886584	0.989649	0.510290	0.455982	0.407229	0.367599	0.343155	0.329796	...	0.326367	0.328844	0.334231	0.339443	0.348425	0.358278	0.367960	0.386734	0.402849
1699	15755.058540	2534.000000	0.987080	0.501312	0.449444	0.402071	0.364971	0.342811	0.330814	...	0.328839	0.331246	0.336421	0.341409	0.349999	0.359357	0.368484	0.386244	0.401638
2000	15533.616998	2971.381238	0.986237	0.491064	0.442166	0.397255	0.364199	0.345127	0.334924	...	0.334727	0.337812	0.341854	0.346492	0.354471	0.363091	0.371410	0.387631	0.401886
2200	15386.479760	3262.000000	0.975690	0.484255	0.437330	0.394055	0.363685	0.346665	0.337655	...	0.338639	0.340844	0.345464	0.349870	0.357443	0.365572	0.373354	0.388552	0.402850
2450	15308.022770	3616.516129	0.967109	0.481369	0.435780	0.394301	0.365974	0.350235	0.341940	...	0.343311	0.345419	0.349825	0.354030	0.361258	0.368971	0.376300	0.390612	0.403428

Figure 13: Final volatility matrix

nBizDays	Forwards	nCalDays	diff Days	0.2	0.3	0.4	0.5	0.6	0.7	...	1.05	1.1	1.2	1.3	1.5	1.75	2	2.5	3
5	17453.039480	7.000000	0.999990	2.619529	2.247577	1.874444	1.542642	1.257031	1.009347	...	0.339046	0.330423	0.341261	0.370340	0.439454	0.511576	0.567061	0.636824	0.669308
13	17456.046050	19.173913	0.999973	2.350258	2.007367	1.672319	1.377522	1.124910	0.905745	...	0.344704	0.335417	0.339562	0.359298	0.410534	0.464789	0.506743	0.560106	0.585603
36	17461.457882	53.200000	0.999920	1.706633	1.438731	1.198225	0.994119	0.821954	0.671705	...	0.356186	0.347151	0.340332	0.342995	0.368089	0.382093	0.399244	0.423083	0.436656
58	17406.284825	84.000000	0.999870	1.411120	1.185449	0.993131	0.833500	0.700067	0.581673	...	0.356015	0.350455	0.347178	0.350644	0.367927	0.389517	0.407232	0.432855	0.448594
80	17355.496116	116.439024	0.999824	1.274839	1.067047	0.895771	0.755587	0.638891	0.534053	...	0.352892	0.349404	0.348373	0.352309	0.368890	0.390450	0.408379	0.435042	0.452313
110	17365.086411	162.421875	0.999758	1.154542	0.962533	0.809856	0.686738	0.584443	0.491585	...	0.348807	0.347401	0.349049	0.353835	0.370379	0.393087	0.412279	0.441679	0.461774
140	17359.451829	205.078125	0.999697	1.089433	0.907225	0.764926	0.650981	0.555785	0.469440	...	0.345377	0.345200	0.348822	0.354683	0.371861	0.395828	0.416450	0.448603	0.471329
200	17310.322106	293.644068	0.999566	0.975149	0.811465	0.687653	0.589695	0.506365	0.431796	...	0.339417	0.341158	0.347796	0.355313	0.373197	0.398508	0.421000	0.456927	0.483484
300	17187.344487	446.133333	0.999354	0.851154	0.712236	0.609658	0.528392	0.456563	0.396451	...	0.333179	0.336037	0.344174	0.352459	0.369460	0.393013	0.415094	0.451117	0.478720
400	17086.846659	596.024390	0.999137	0.763012	0.643515	0.556557	0.486341	0.422830	0.373876	...	0.328735	0.331959	0.340376	0.348689	0.364380	0.385398	0.406021	0.440348	0.467222
500	16986.730167	744.295082	0.998884	0.701988	0.596922	0.520952	0.457439	0.400255	0.359550	...	0.325584	0.328835	0.336995	0.345009	0.359472	0.378092	0.396925	0.429013	0.454458
800	16651.576849	1189.826446	0.997716	0.602229	0.522428	0.462842	0.408479	0.365131	0.338610	...	0.321012	0.323996	0.331031	0.337944	0.349943	0.364118	0.378655	0.405170	0.426692
900	16548.844319	1337.323077	0.997105	0.582038	0.507699	0.450811	0.398590	0.358801	0.335148	...	0.320582	0.323468	0.330183	0.336783	0.348204	0.361452	0.374974	0.400048	0.420558
1150	16276.004576	1710.596774	0.995849	0.544873	0.480883	0.428126	0.381001	0.348578	0.330210	...	0.321343	0.324048	0.330124	0.336074	0.346342	0.357897	0.369512	0.391633	0.410086
1400	16025.574902	2085.419087	0.992004	0.520903	0.463542	0.413497	0.371203	0.344176	0.329244	...	0.324114	0.326667	0.332285	0.337744	0.347155	0.357560	0.367867	0.387755	0.404656
1550	15886.567062	2310.886584	0.989649	0.510290	0.455902	0.407229	0.367599	0.343155	0.329796	...	0.326367	0.328844	0.334231	0.339443	0.348425	0.358278	0.367960	0.386734	0.402849
2000	15533.616998	2971.381238	0.980237	0.491064	0.442166	0.397255	0.364199	0.345127	0.334924	...	0.334727	0.337012	0.341854	0.346492	0.354471	0.363891	0.371410	0.387631	0.401886
2450	15308.022770	3616.516129	0.967100	0.481369	0.435780	0.394301	0.365974	0.350235	0.341940	...	0.343311	0.345419	0.349825	0.354030	0.361258	0.368971	0.376300	0.390612	0.403428
2700	15229.565779	3971.032258	0.958529	0.478482	0.434231	0.394548	0.368262	0.353805	0.346225	...	0.347984	0.349994	0.354186	0.358189	0.365072	0.372369	0.379245	0.392671	0.404806
2900	15166.808187	4254.645161	0.951664	0.476173	0.432992	0.394745	0.370093	0.356660	0.349653	...	0.351722	0.353654	0.357675	0.361517	0.368124	0.375088	0.381602	0.394319	0.405988
3200	15143.263090	4361.000000	0.949090	0.475388	0.432527	0.394820	0.370779	0.357731	0.350939	...	0.353123	0.355026	0.358983	0.362765	0.369268	0.376108	0.382486	0.394937	0.406322

[21 rows x 24 columns]

After interpolating and keeping only the maturities and moneyness we wanted, we realised that we had exactly 441 dimensions (21 maturities * 21 moneyness). Our algorithms such as DBSCAN and Isolation forest are not very efficient with very high dimension. So, in order to use them, we needed to reduce dimension.

3.2 Reduction dimension with PCA

3.2.1 Concept

PCA is defined as an orthogonal linear transformation that transforms the data to a new coordinate system such that the greatest variance by some scalar projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on. It finally perform a change of basis of the data and project each data point onto only the first few principal components to obtain lower-dimensional data while preserving as much of the data's variation as possible.

3.2.2 Algo

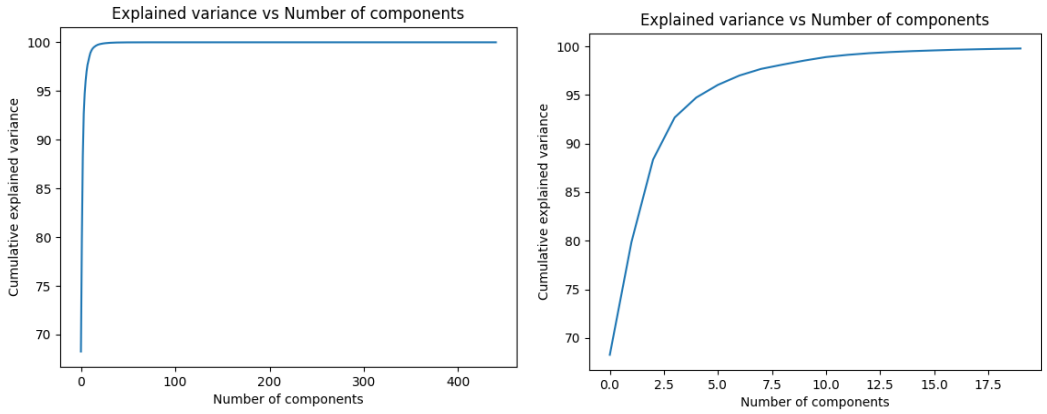
- Step 1: Calculate the Correlation matrix data consisting of n dimensions. The Correlation matrix will be of shape n*n.
- Step 2: Calculate the Eigenvectors and Eigenvalues of this matrix.
- Step 3: Take the first k-eigenvectors with the highest eigenvalues.
- Step 4: Project the original dataset into these k eigenvectors resulting in k dimensions where $k \leq n$.

The code below is the code for the PCA, we used the numpy library to help us :

```
def PCA(X:np.ndarray):
    #Standardize data
    X_scaled = (X - np.mean(X , axis = 0))/np.std(X)

    # calculating the covariance matrix of the mean-centered data.
    cov_mat = np.cov(X_scaled , rowvar = False)

    #Calculating Eigenvalues and Eigenvectors of the covariance matrix
    #Eigen Values is 1D array and Eigen Vectors is ndarray
    eigen_values , eigen_vectors = np.linalg.eigh(cov_mat)
    eigen_values = np.real(eigen_values)

    #sort the eigen values and eigen vectors in descending order
    sorted_index = np.argsort(eigen_values)[::-1]
    sorted_eigenvalue = eigen_values[sorted_index]
    sorted_eigenvectors = eigen_vectors[:,sorted_index]

    # Calculating the explained variance on each of components
    variance_explained = []
    for i in sorted_eigenvalue:
        variance_explained.append((i/sum(sorted_eigenvalue))*100)
    # Cumulative explained variance
    cumulative_variance_explained = np.cumsum(variance_explained)

    #Projection Matrix
    n_components = 8
    projection_matrix = (eigen_vectors.T[:][:n_components]).T
    X_pca = np.dot(X_scaled, projection_matrix)
    return X_pca
```

The cumulative explained variance is :

Figure 14:



We observe that the cumulative explained variance converges very fast to 100%. Indeed the first component already explains 78% of the variance and the first two components explain 90% of the variance. We need 5 dimensions to reach 95% of the explained variance.

We decided to only use the first five components because we consider that 95% of explained variance is enough.

3.2.3 Application of Isolation Forest after PCA

We finally have our 5 dimensions for each observation and we can now apply the isolation forest to those observations. We will plot the result with the first two components to have better visualisation of the data and we will compare several threshold:

Figure 15: Threshold 0.55 vs Threshold 0.58

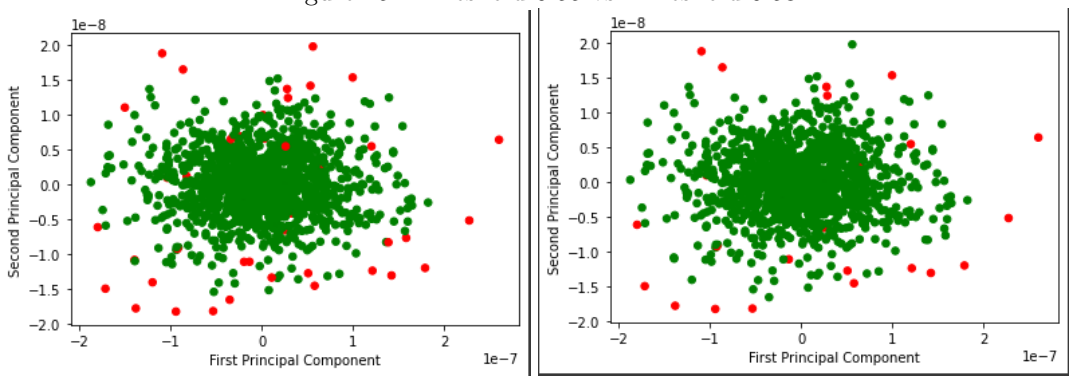
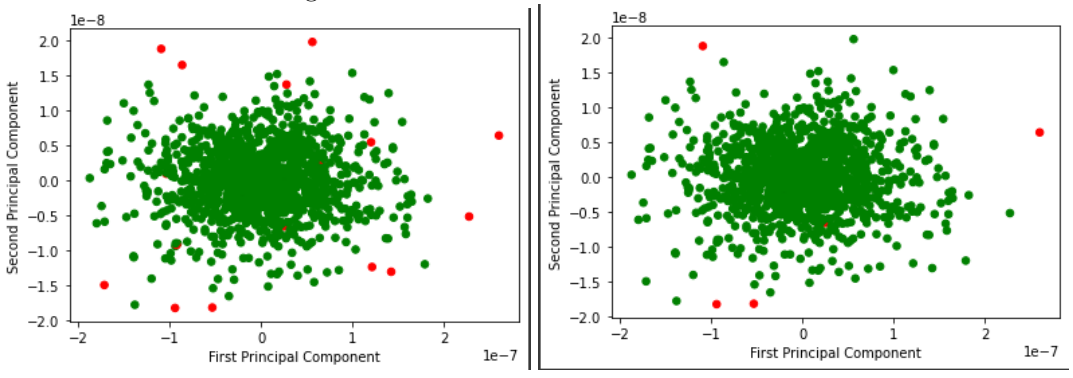


Figure 16: Threshold 0.60 vs Threshold 0.65



The threshold represent the limit where the observation is or isn't an outlier with the respect of its anomaly score. For example if an observation has 0.65 for its outlier score and the threshold is 0.60, the observation will be consider as an outlier.

More the threshold is high less the Isolation Forest will accept observations as outlier. That's why we have less outlier with 0.65 as threshold. At the end the Isolation Forest return a list of indexes that represent which observations are an outlier.

The limit of the Isolation Forest here is that we have to decide arbitrary what threshold we should use and the result will depend on it. But even if there is a part of randomness in the

Isolation Forest algorithm, we can observe that they give the same outliers every we run the algorithm. The outliers depends only of the threshold.

3.2.4 Application of DBSCAN after PCA

For the DBSCAN, we will compare a PCA that reduces dimensions to 5 and a PCA that reduces dimensions to 8. We will plot the result with the first two components to have better visualisation of the data and we will compare with different minimum points :

Figure 17: DBSCAN 5 dimensions

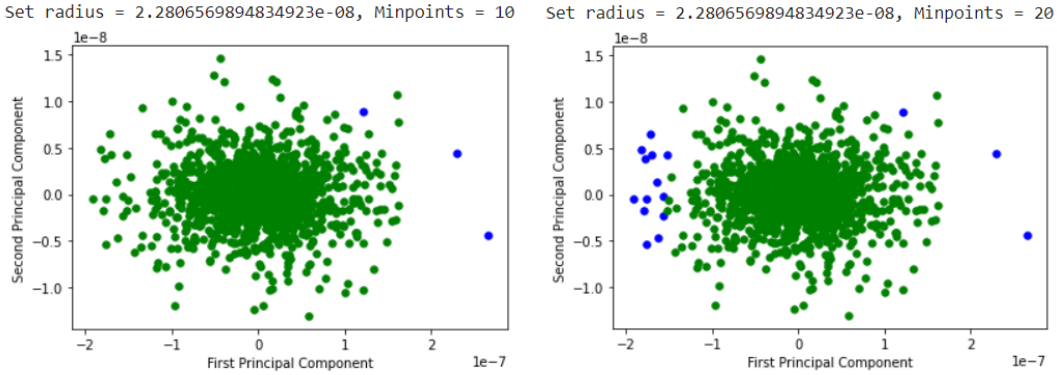
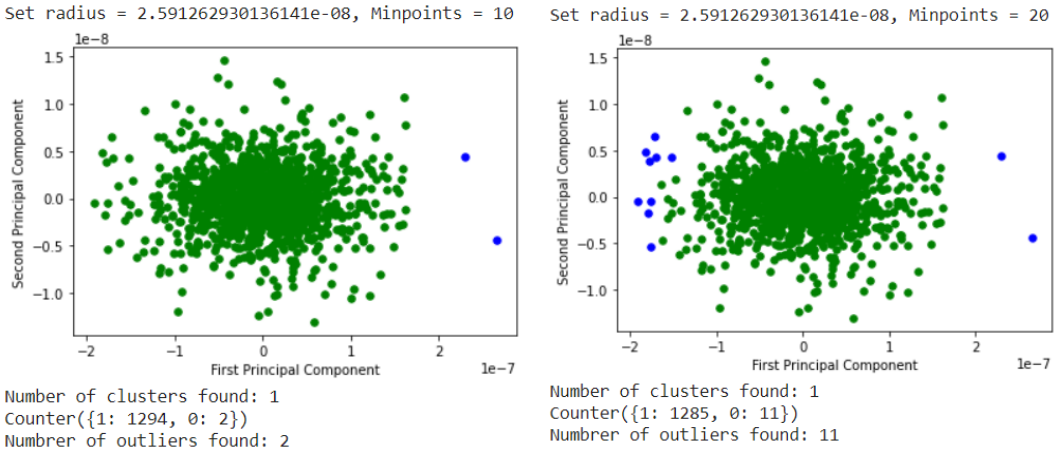


Figure 18: DBSCAN 8 dimensions



The number of outliers is positively correlated with the parameter minimum point. Indeed, more minimum point is high, more the DBSCAN return many outliers. We also compare DBSCAN with different dimensions but we can observe that the number of outlier doesn't change much.

The limit is that we have to set arbitrarily the minium point and the result will depend on it.

3.3 Dimension Reduction with autoencoder

3.3.1 Concept

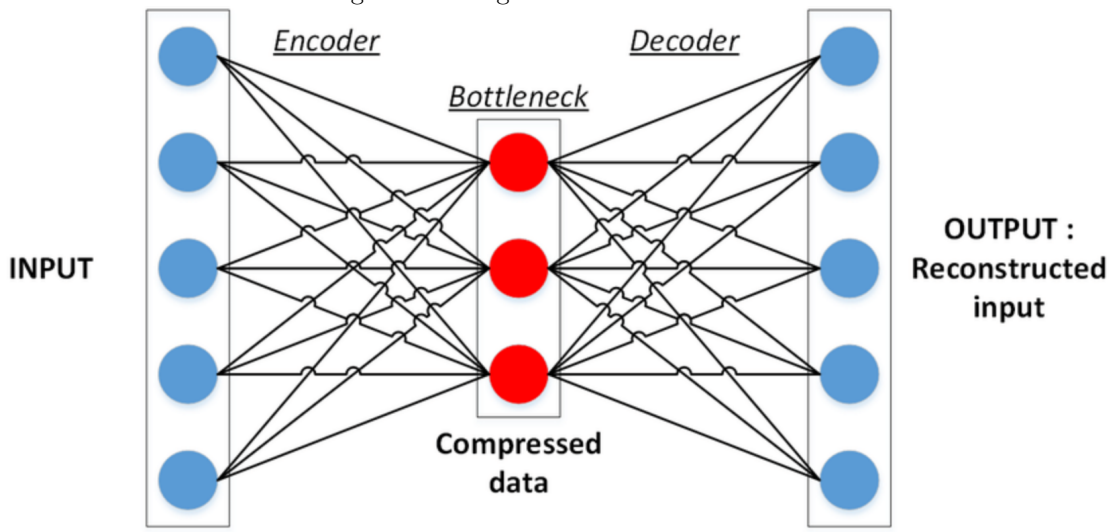
A neural network is a system that try to recognise complex pattern. It can be supervised (feed-forward, CNN, RNN) or unsupervised (ANN). It resolves problem such as classification, regression, dimension reduction and outlier detection etc...

It is made up of several "layers" of neurons, each receiving and interpreting information from the previous layer. In order to do so, each neuron of each layer is connected by weights coming from the neurons of the previous layer. It is a self-learning method which, through an algorithm, allows to learn from errors (cost function) which are returned in all layers. Its goal is to minimize the cost between the calculated output and the real output by changing the weights in the network.

Since our problem is unsupervised we are going to use a specific network called the autoencoder. The autoencoder is used to reduce dimension. Indeed, the autoencoder is build in 2 parts:

- Encoder: Take the dataset as the input layer and an hidden layer of lower dimension as its output layer, its goal is to compress the data in lower dimension
- Decoder: Take the output of the encoder as input and the output of the neural network as its output, its goal is to replicate the compress data as the original data

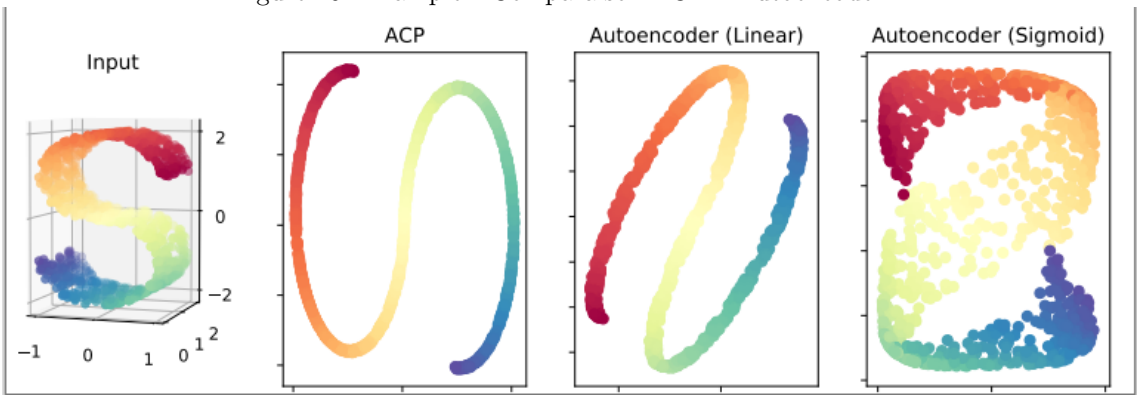
Figure 19: Diagram of an autoencoder



Its concept is to learn how to replicate the inputs into the outputs. During the training it will minimize the cost between the true output and the computed output by changing weights. After the training, we just have to feed forward the inputs to the outputs and return the compressed data that are located in the output of the encoder. We repeat it for all observations.

3.3.2 Autoencoder vs PCA

Figure 20: Example : Comparaison PCA - Autoencoder



1. PCA is a linear transformation of data while the autoencoder can be linear or non-linear depending on the choice of the activation function.
2. PCA projects data into dimensions that are orthogonal to each other resulting in very low or close to zero correlation in the projected data. However the autoencoder transformed data doesn't guarantee that because the way it's trained is merely to minimize the reconstruction loss.
3. PCA is pretty fast as there exist algorithms that can fast calculate it while the autoencoder trains through Gradient descent and is slower comparatively

3.3.3 Algo

Since we don't want a linear transformation, we'll be using the sigmoid function to make a non-linear transformation. We will be using the sigmoid activation in the encoder the decoder. However we use the sigmoid in the ouput layer that gives us a result between 0 and 1. So we first have to scaled the data before training our network. The code for the autoencoder reduction dimensions is below:

```
def autoencoder_dimensionReduction(data):
    mms = preprocessing.MinMaxScaler()
    trainDataScaled = mms.fit_transform(trainData)
    # 2 floats -> compression of factor 0.1, assuming the input is 441 floats
    encoding_dim = 2
    input = Input(shape=(441,))
    encoded = Dense(encoding_dim, activation='sigmoid')(input)
    decoded = Dense(441, activation='sigmoid')(encoded)
    autoencoder = Model(inputs=input, outputs=decoded)
    encoder = Model(inputs=input, outputs=encoded)
    encoded_input = Input(shape=(encoding_dim,))
    decoder_layer = autoencoder.layers[-1]
    decoder = Model(inputs=encoded_input, outputs=decoder_layer(encoded_input))
```

```
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
model_autoencoder = autoencoder.fit(trainDataScaled, trainDataScaled,
                                    epochs=200, batch_size=100,
                                    shuffle=True,verbose=verbose)
encoded = encoder.predict(trainDataScaled)
decoded = decoder.predict(encoded)
return encoded
```

3.3.4 Application with our dataset

We train our autoencoder with different number of neurons in the output layer of the encoder. Here are the results of the lost function in function of the number of epoch.

Figure 21: Example : Dim2 vs Dim4

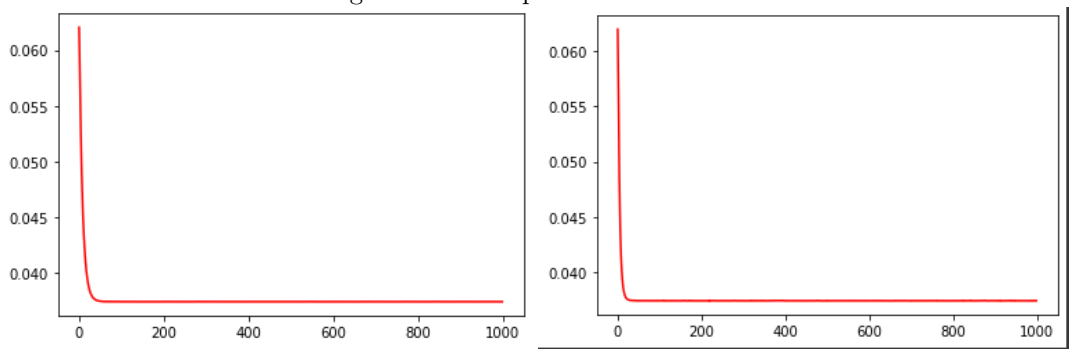


Figure 22: Example : Dim6 vs Dim8

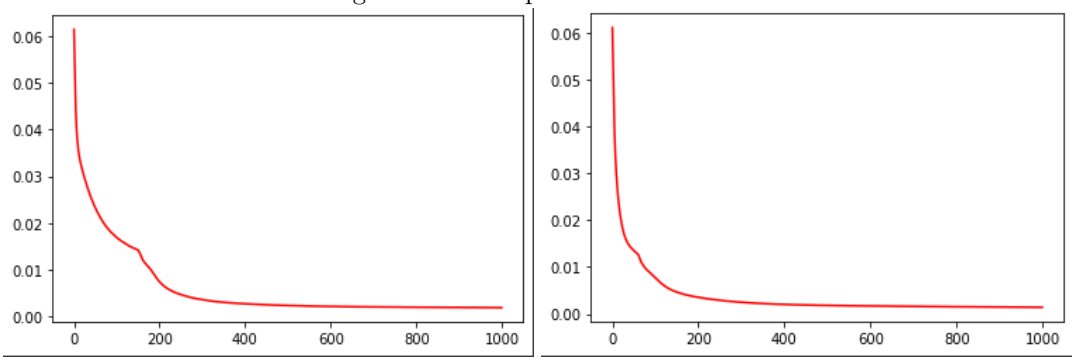
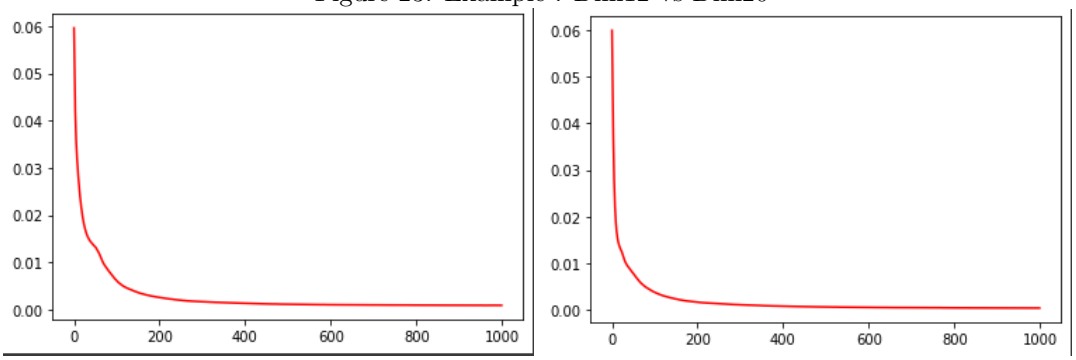


Figure 23: Example : Dim12 vs Dim20



We will choose 8 dimensions since it converges with a lower cost function than dimension 2 or 4. After 8 dimensions it converges faster but we will use only 8 dimensions that are more adapted to our algorithms such as DBSCAN and Isolation Forest

3.3.5 Application of Isolation Forest after autoencoder

We will plot the result in 3 dimensions to have a better visualisation, but since the autoencoder is non linear you will see that only ploting in 3 dimensions doesn't show the reality.

Figure 24: Example : Threshold 0.55 (40 outliers) vs Threshold 0.58 (14 outliers)

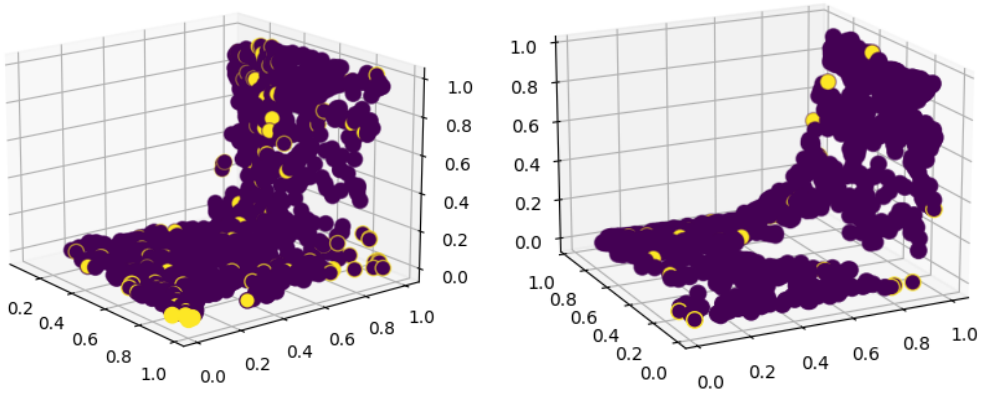
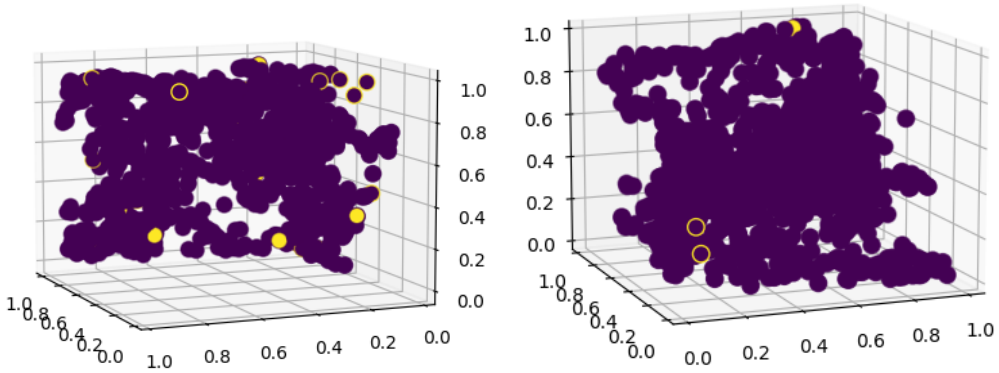


Figure 25: Example : Threshold 0.60 (13 outliers) vs Threshold 0.63 (3 outliers)



It's hard to say if the isolation forest is efficient with this plot. Surprisingly, it gives almost the same number of outliers as the PCA but unfortunately it doesn't give the same outliers. Like we said for the PCA application, the isolation forest algortihm returns the same outliers every time we run the code except when we recalculate our autoencoder it returns different outliers. The number of outliers depends on the threshold and our autoencoder that is not stable.

To compare the Isolation Forest after PCA and after AE, we compare the list of outliers they return for the specific threshold 0.58 :

Figure 26: Isolation Forest after PCA threshold 0.58

```
[3, 6, 22, 24, 32, 65, 69, 83, 87, 91, 105, 116, 126, 273, 300, 394, 422, 995, 1143, 1171, 1213, 1220, 1252, 1273, 1276]
```

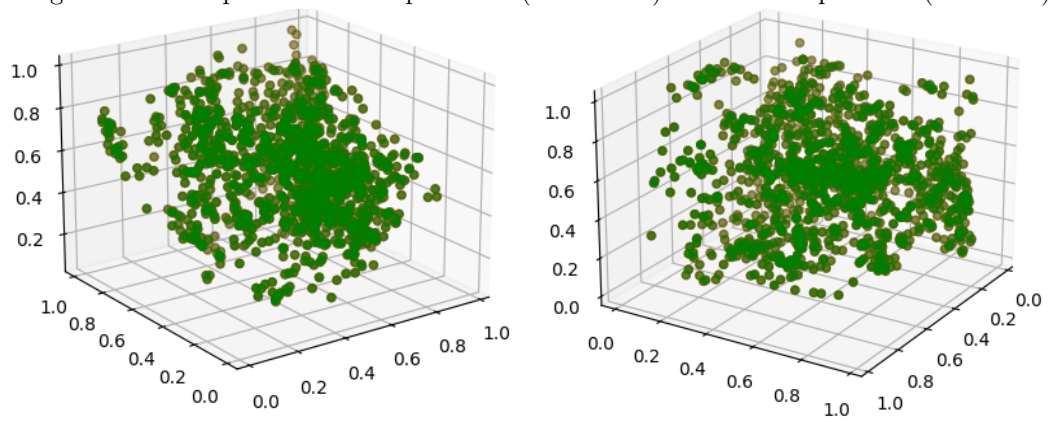
Figure 27: Isolation Forest after AE threshold 0.58

```
[84, 167, 168, 288, 416, 980, 981, 982, 983, 1033]
```

Unfortunately we see that there is no common outliers.

3.3.6 Application of DBSCAN after autoencoder

Figure 28: Example : Minimum points 20 (10 outliers) vs Minmum point 10 (5 outliers)



As the Isolation Forest, it's hard to say if the DBSCAN is effecient just by looking the plot. However they both return a list of outlier which led us to see directly their surfaces and see if they are normal or abnormal.

In the next part, we will use the autoencoder to directly return the list of outliers without using Isolation Forest or DBSCAN.

4 Autoencoder : Outlier Detection

The objetive of this autoencoder is to find outlier points in high dimension. It is used to learn how to replicate inputs data in the output layer. We compute for each observations its cost (distance) between the true outputs and the output layer. If the distance of the observation is too high, we can consider it as an outlier because the autoencoder has difficulties to replicate it (it is abnormal compared to others). In general, the autoencoder has difficulties to replicate it in specific dimension that greatly increase the cost.

So after training the autoencoder with all our data, we try to predict all observations (same than during the training), we obtain the cost (distances) for all observation depending the number of neurons in the output of the decoder.

Figure 29: Example : 8 neurons

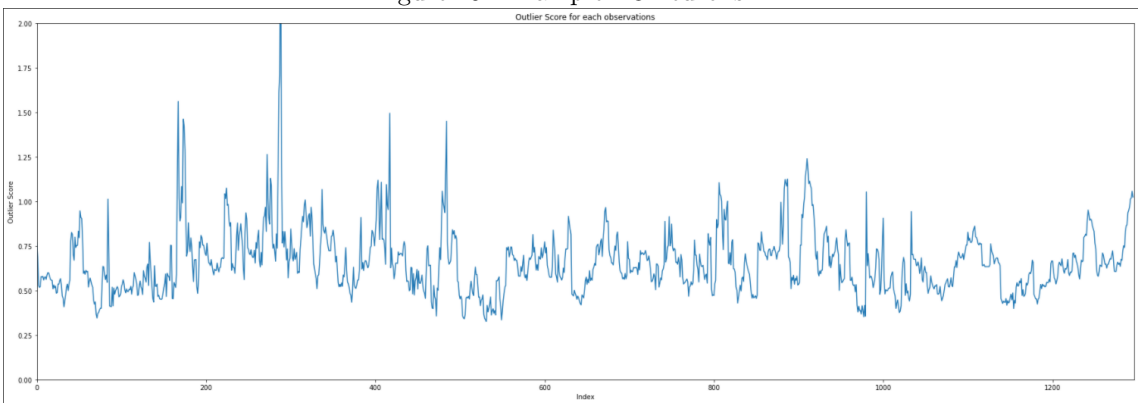


Figure 30: Example : 12 neurons

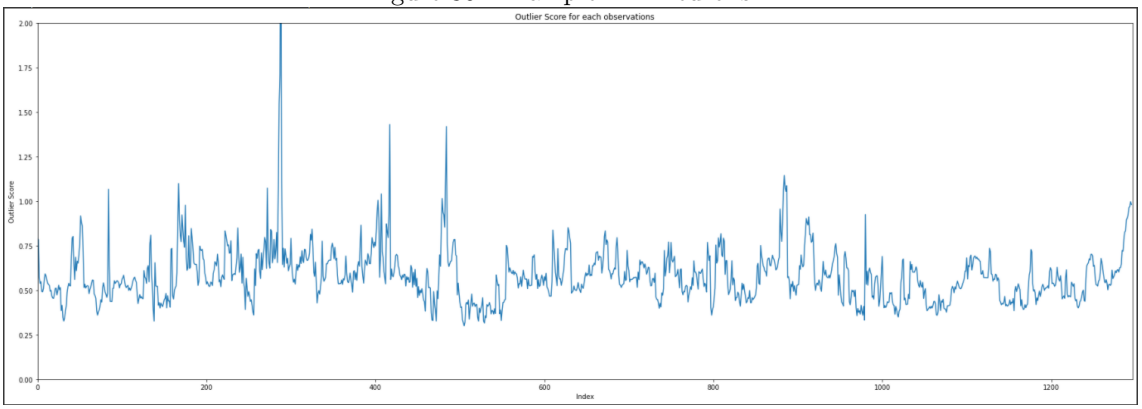


Figure 31: Example : 20 neurons

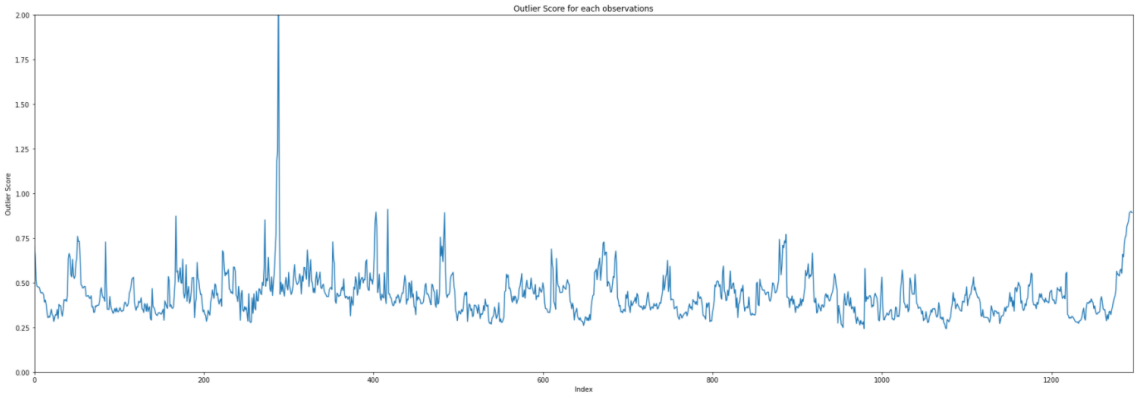
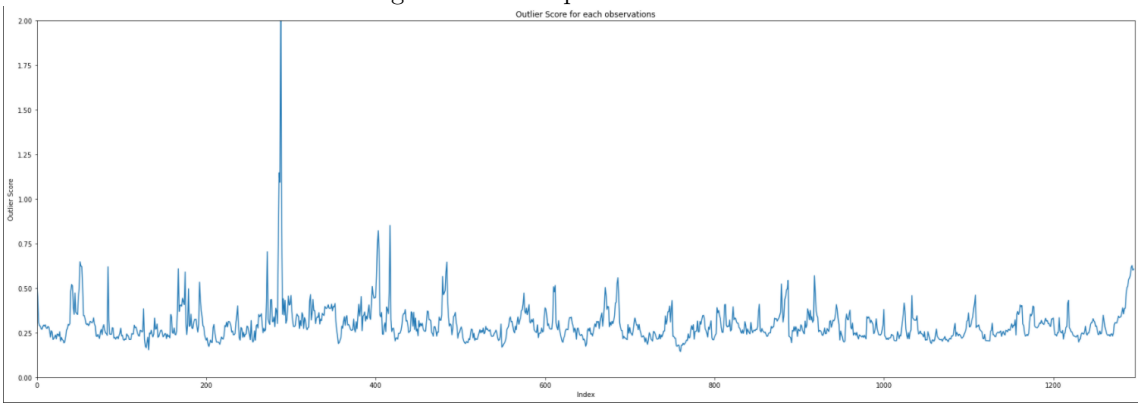


Figure 32: Example : 40 neurons



We can observe that more neurons there are in our decoder output, more the cost look stable. However, it's always the same observations which our autoencoder has difficulties to replicate (the cost is higher)

So again, we have to choose arbitrarily a threshold that says if the observation's cost (distances) is higher than the threshold, it considerate as an outlier. We will take for example 12 neurons and we obtain 3 anomalies for threhold 1.5, 11 anomalies for threshold 1.1, 24 anomalies for threshold 1 and 36 anomalies for 0.9.

Unlike the isolation forest or the dbscan after autoencoder, it gives always the same result even if we rerun the autoencoder code. After looking the surfaces that the autoencoder returns to us, we observe that some of them had abnormal surfaces.

As mentioned above, the autoencoder has difficulties to replicate it in specific dimension that greatly increase the cost. So we ploted the error of an outlier depending its dimensions

Figure 33: Example : Reconstruction error of observation 1033

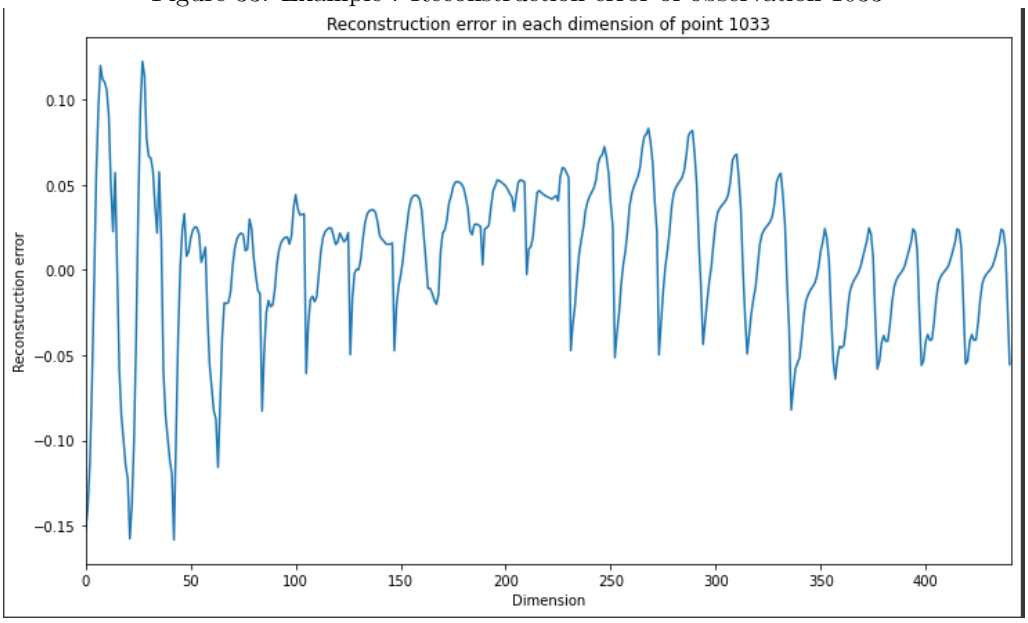
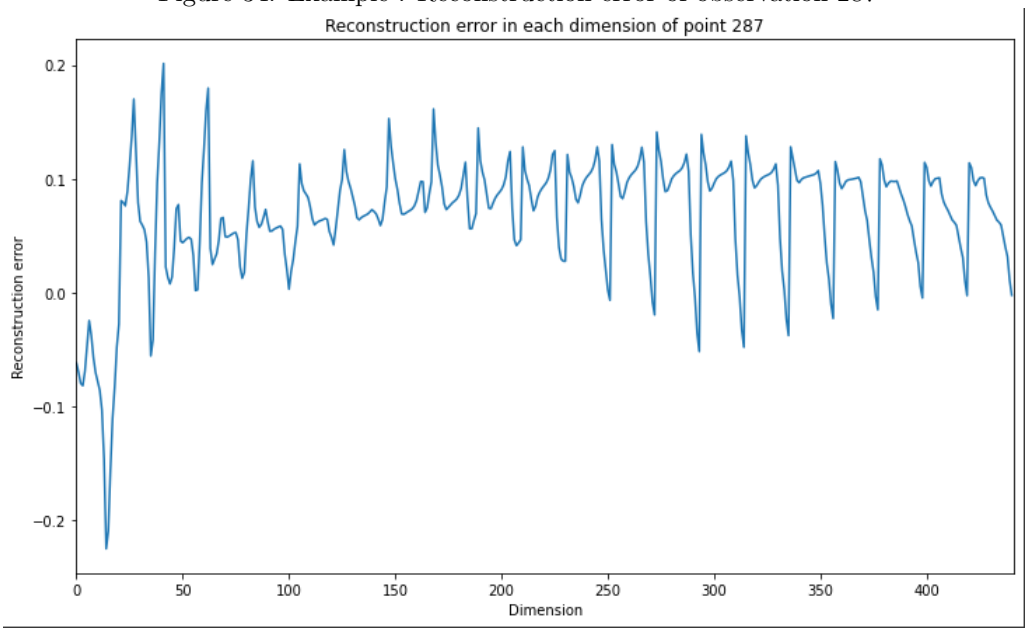


Figure 34: Example : Reconstruction error of observation 287



To conclude, we think that the autoencoder is more efficient, we have better result, it looks much more stable than the isolation forest and DBSCAN after the dimension with the autoencoder and unlike the PCA, the autoencoder is non-linear like our data.

You can see the code of the autoencoder in the annex

5 Conclusion

We can observe some surfaces that our algorithms detected as outliers surface. Some of these surface shapes look abnormal compared to others. There exists more surfaces that we didn't plot even though their shapes look abnormal.

Figure 35: Example

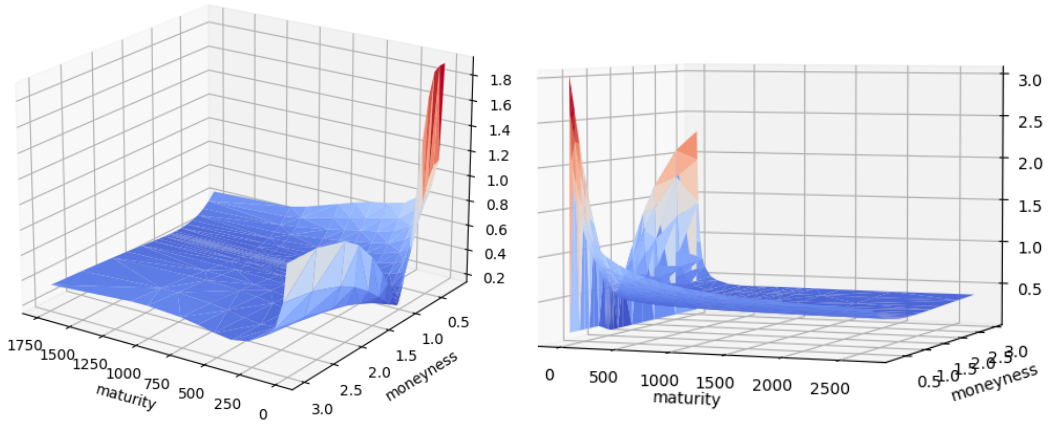
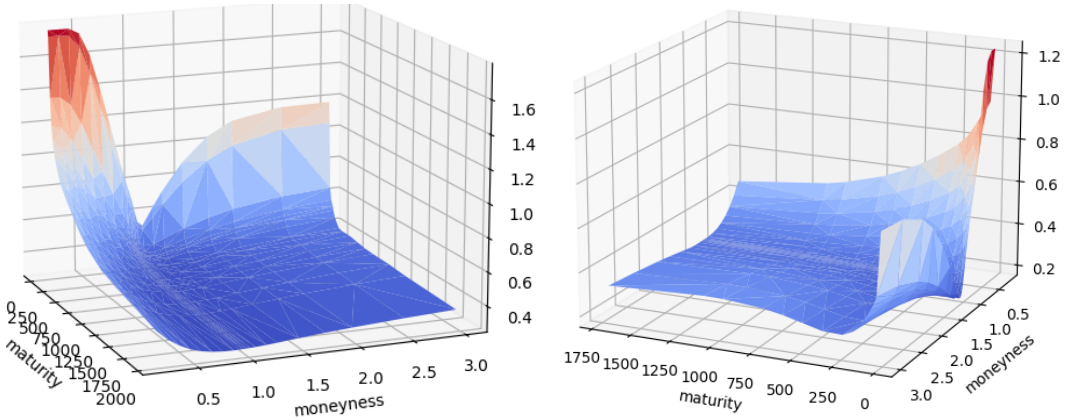


Figure 36: Example



However our algorithms detected many outlier surfaces whose shapes looked normal to us. The outliers were not obvious by just comparing their surfaces to others. To go further we could try to find outlier patterns and use them as a criteria for next volatility surfaces to make it easier and more efficient. We face some difficulties during the project :

- we had to interpolate a lot of data which reduced the accuracy of the volatility surface and increased the bias
- the high dimension of the problem led us to reduce dimension with PCA and autoencoder
- even though we reduced dimension (from 441 to 8), it became harder to interpret the result especially when using autoencoder-dimension reduction

- despite the stability of the detection algorithms after PCA, the problem was non-linear unlike the PCA and so we can't conclude on its quality
- on the contrary, the autoencoder solves non-linear problem but we obtained less stable results than with the PCA

After consideration of different methods, we conclude that the autoencoder-outlier detection is the most stable and efficient model between the ones studied during this project.

6 Annex

6.1 Isolation Forest Class

```
class LeafNode:
    def __init__(self, size, data):
        self.size = size
        self.data = data


class DecisionNode:
    def __init__(self, left, right, splitFeature, splitValue):
        self.left = left
        self.right = right
        self.splitFeature = splitFeature
        self.splitValue = splitValue


class IsolationTree:
    def __init__(self, height, maxDepth):
        self.height = height
        self.maxDepth = maxDepth

    def fit(self, X):
        """
        Given a 2D matrix of observations, create an isolation tree. Set field
        self.root to the root of that tree and return it.
        """
        if self.height >= self.maxDepth or X.shape[0] <= 2: #X.shapes[0] number of points
            self.root = LeafNode(X.shape[0], X)
            return self.root

        # Choose Random Split features and Value
        num_features = X.shape[1] #X.shapes[1] number of features
        splitFeature = np.random.randint(0, num_features) #take radomly a feature
        splitValue = np.random.uniform(min(X[:, splitFeature]), max(X[:, splitFeature])) #take ra

        X_left = X[X[:, splitFeature] < splitValue]
        X_right = X[X[:, splitFeature] >= splitValue]

        leftTree = IsolationTree(self.height + 1, self.maxDepth)
        rightTree = IsolationTree(self.height + 1, self.maxDepth)
        leftTree.fit(X_left)
        rightTree.fit(X_right)
        self.root = DecisionNode(leftTree.root, rightTree.root, splitFeature, splitValue)
        self.n_nodes = self.count_nodes(self.root)
        return self.root

    def count_nodes(self, root):
```

```
        count = 0
        stack = [root]
        while stack:
            node = stack.pop()
            count += 1
            if isinstance(node, DecisionNode):
                stack.append(node.right)
                stack.append(node.left)
        return count

class IsolationForest:
    def __init__(self, sample_size, n_trees=10):
        self.sample_size = sample_size
        self.n_trees = n_trees

    def fit(self, X): #X must be ndarray
        """
        Given a 2D matrix of observations, create an ensemble of IsolationTree
        objects and store them in a list: self.trees.  Convert DataFrames to
        ndarray objects.
        """
        self.trees = [] #array of n treess
        if isinstance(X, pd.DataFrame):
            X = X.values
        n_rows = X.shape[0]
        height_limit = np.ceil(np.log2(self.sample_size))
        for i in range(self.n_trees):
            #data_index = np.random.choice(range(n_rows), size=self.sample_size, replace=False)
            #We are using the bootstrap in order to create new Sub_data
            #choose randomly the sample (size = sample_size) from the dataSet wich we are going t
            data_index = np.random.randint(0, n_rows, self.sample_size)
            X_sub = X[data_index]
            tree = IsolationTree(0, height_limit)
            tree.fit(X_sub)
            self.trees.append(tree)
        return self

    def path_length(self, X:np.ndarray) -> np.ndarray:
        """
        Given a 2D matrix of observations, X, compute the average path length
        for each observation in X, we compute the path length for x_i using every
        tree in self.trees then compute the average for each x_i.
        Return an ndarray of shape (len(X),1).
        """
        paths = []
        for row in X:
            path = []
            for tree in self.trees:
                node = tree.root
```

```
                length = 0
                while isinstance(node, DecisionNode):
                    if row[node.splitFeature] < node.splitValue:
                        node = node.left
                    else:
                        node = node.right
                    length += 1
                leaf_size = node.size
                pathLength = length + c(leaf_size)
                path.append(pathLength)
            paths.append(path)
        paths = np.array(paths)
        return np.mean(paths, axis=1)

    def anomaly_score(self, X:pd.DataFrame) -> np.ndarray:
        """
        Given a 2D matrix of observations, X, compute the anomaly score
        for each x_i observation, returning an ndarray of them.
        """
        if isinstance(X, pd.DataFrame):
            X = X.values
        avg_length = self.path_length(X)
        scores = np.array([np.power(2, -l/c(self.sample_size))for l in avg_length])
        return scores

    def predict_from_anomaly_scores(self, scores:np.ndarray, threshold:float) -> np.ndarray:
        """
        Given an array of scores and a score threshold, return an array of
        the predictions: 1 for any score >= the threshold and 0 otherwise.
        """
        return np.array([1 if s >= threshold else 0 for s in scores])

    def predict(self, X:np.ndarray, threshold:float) -> np.ndarray:
        "A shorthand for calling anomaly_score() and predict_from_anomaly_scores()."
        scores = self.anomaly_score(X)
        prediction = self.predict_from_anomaly_scores(scores, threshold)
        return prediction
```

6.2 DBSCAN Class

```
class DBSCAN:

    def __init__(self, data, radius, MinPt):
        self.data = data
        self.radius = radius
        self.MinPt = MinPt
        self.noise = 0
        self.unassigned = 0
        self.core=-1
        self.edge=-2

    #function to find all neigbor points in radius
    def neighbor_points(self, pointId):
        points = []
        for i in range(len(self.data)):
            #Euclidian distance using L2 Norm
            if sum((self.data[i] - self.data[pointId])**2) <= self.radius**2:
                points.append(i)
        return points


    def dbscan(self):
        #initilize all pointlabel to unassign
        pointlabel  = [self.unassigned] * len(self.data)
        pointcount = []
        #initialize list for core/noncore point
        corepoint=[]
        noncore=[]

        #Find all neigbor for all point
        for i in range(len(self.data)):
            pointcount.append(DBSCAN.neighbor_points(self, i))

        #Find all core point, edgepoint and noise
        for i in range(len(pointcount)):
            if (len(pointcount[i])>=self.MinPt):
                pointlabel[i]=self.core
                corepoint.append(i)
            else:
                noncore.append(i)

        for i in noncore:
            for j in pointcount[i]:
                if j in corepoint:
                    pointlabel[i]=self.edge
                    break
```

```
        #start assigning point to luster
        cl = 1
        #Using a Queue to put all neigbor core point in queue and find neigboir's neigbor
        for i in range(len(pointlabel)):
            q = queue.Queue()
            if (pointlabel[i] == self.core):
                pointlabel[i] = cl
                for x in pointcount[i]:
                    if(pointlabel[x]==self.core):
                        q.put(x)
                        pointlabel[x]=cl
                    elif(pointlabel[x]==self.edge):
                        pointlabel[x]=cl
                #Stop when all point in Queue has been checked
                while not q.empty():
                    neighbors = pointcount[q.get()]
                    for y in neighbors:
                        if (pointlabel[y]==self.core):
                            pointlabel[y]=cl
                            q.put(y)
                        if (pointlabel[y]==self.edge):
                            pointlabel[y]=cl
                cl=cl+1 #move to next cluster
        return pointlabel,cl

    #Function to plot final result with different clusters and anomalies
    def plotRes(self, clusterRes, clusterNum):
        nPoints = len(self.data)
        scatterColors = ['black', 'green', 'brown', 'red', 'purple', 'orange', 'yellow']
        for i in range(clusterNum):
            if (i==0):
                #Plot all noise point as blue
                color='blue'
            else:
                color = scatterColors[i % len(scatterColors)]
            abscissa = []
            ordinate = []
            for j in range(nPoints):
                if clusterRes[j] == i:
                    abscissa.append(self.data[j, 0])
                    ordinate.append(self.data[j, 1])
            plt.scatter(abscissa, ordinate, c=color, alpha=1, marker='.')

def days_outliers(data):
        res = []
        for i in range(len(data)):
            if data[i]==0:
                res.append(i)
        return res
```

```
def Distances(data):
    distance=[]
    for i in range(len(data)):
        distance1=0
        for j in range(len(data)):
            if i!= j:
                distance1 += np.linalg.norm(data[i] - data[j])
        distance.append(distance)
    return distance
```

6.3 Autoencoder function : outlier detection

```
def autoencoder_outlierDetection(data, verbose = 0, threshold = 1.1):
    # No test data needed, trainData is all our dataset
    n_train = int(len(data)*100/100)
    trainData = data[:n_train]
    #Preprocessing : we need our input data between 0 and 1
    # Since we'll be using sigmoid for the output layer
    mms = preprocessing.MinMaxScaler()
    trainDataScaled = mms.fit_transform(trainData)

    # this is the size of our encoded representations
    encoding_dim = 40  # 40 floats -> compression of factor 0.1, assuming the input is 441 floats
    # this is our input placeholder
    input = Input(shape=(441,))
    # "encoded" is the encoded representation of the input
    encoded = Dense(encoding_dim, activation='relu')(input)
    # "decoded" is the lossy reconstruction of the input
    decoded = Dense(441, activation='sigmoid')(encoded)
    # Since we create an autoencoder this model must map an input to its reconstruction
    autoencoder = Model(inputs=input, outputs=decoded)
    # create the encoder model that maps an input to its encoded representation
    encoder = Model(inputs=input, outputs=encoded)
    # create a placeholder for an encoded (40-dimensional) input
    encoded_input = Input(shape=(encoding_dim,))
    # retrieve the last layer of the autoencoder model
    decoder_layer = autoencoder.layers[-1]
    # create the decoder model that maps the encoded inputs to its reconstruction
    decoder = Model(inputs=encoded_input, outputs=decoder_layer(encoded_input))

    # We train over 200 epochs, with a batch size of 100
    # Finally, we get the prediction of the network for our data
    autoencoder.compile(optimizer='Nadam', loss='binary_crossentropy')
    model_autoencoder = autoencoder.fit(trainDataScaled,
                                        trainDataScaled,
                                        epochs=200,
                                        batch_size=100,
                                        shuffle=True,
                                        verbose=verbose)
    if verbose == 1 :
        plt.plot(model_autoencoder.history["loss"], color = "r")
        autoencoder.summary()
    encoded = encoder.predict(trainDataScaled)
    decoded = decoder.predict(encoded)
    if verbose == 1 :
        print("TrainDataScaled[10, 10:30] \n" + trainDataScaled[10, 10:30])
        print("decoded[10,10:30] \n" + decoded[10,10:30])

    # We compute the euclidean distance from each point to its reconstruction.
```

```
    # We use it as an outlier score:
    dist = np.zeros(len(trainDataScaled))
    for i, x in enumerate(trainDataScaled):
        dist[i] = np.linalg.norm(x-decoded[i])
    if verbose == 1 :
        plt.figure(figsize=(30,10))
        plt.plot(dist)
        plt.xlim((0,1296))
        plt.ylim((0,2))
        plt.xlabel('Index')
        plt.ylabel('Outlier Score')
        plt.title("Outlier Score for each observations")

    outliers = [i for (i, x) in enumerate(dist) if x > threshold]
    return outliers
```