REPORT

APPLIED MATHEMATICS PROJECT

Principal Component Analysis

Supervised by :
LOUNICI Karim

Authors :
EL HAJOUJI Oualid
SAADANI HASSANI Jad

Abstract

The aim of this work is to study the Principal Component Analysis algorithm, an unsupervised learning technique of dimension reduction with various applications in data science. We begin by presenting the algorithm in a specific context of data with a centered Gaussian distribution, and justify the theoretical link between the algorithm and spectral theory. In order to evaluate the consistency of the algorithm, we analyse the theory of this method, and find upper bounds to the errors between the eigenvectors and eigenvalues of the sample covariance and the real covariance matrix Σ. We validate and complete the theoretical results found through numerical simulation of Gaussian distributions in the Spiked Covariance Model. Finally, we study an application of Principal Component Analysis, which is anomaly detection, using real satellite image data.

Contents

1 Theoretical frame

The aim of this section is to introduce PCA, and the basic theoretical and practical concepts that this method involves.

1.1 Introduction to PCA

Principal component analysis (PCA) is a family of techniques that take high-dimensional data, and use the dependencies between the variables to represent it in a more tractable, lower-dimensional form, without losing too much information. We start with p-dimensional vectors (each one representing an observation), and want to transform them by projecting down into a k-dimensional subspace. The result will be the projection of the original vectors on to k directions, the principal components, which span the sub-space.

There are several equivalent ways of deriving the principal components mathematically. The simplest one is by finding the projections which maximize the variance. The first principal component is the direction in space along which projections have the largest variance. The second principal component is the direction which maximizes variance among all directions orthogonal to the first. The i^{th} component is the variance-maximizing direction orthogonal to the previous $i-1$ components.

There are k principal components in total. By choosing the first k projections that maximize the variance, and by making a k-linear combination of the projections of the initial p features, we create a new set of k relevant features that describe observations almost as well. This reduction of dimensionality while keeping an almost exhaustive description of the data allows a more relevant visualization and analysis of the data.

1.2 Link with spectral theory

1.2.1 Mathematical frame

In the rest of this report, we will mathematically represent all the observations that constitute the data set by a set of n p-dimensional vectors $(X_1, X_2, ..., X_n)$. We will assume that these vectors are independent random variables and identically distributed according to a centered Normal distribution.

For all $i \in [\![1, n]\!]$: $X_i \sim N_p(0, \Sigma)$ where Σ is the covariance matrix of the variables X_i.

We construct an unbiased estimator of the covariance matrix (which happens to be the maximum likelihood estimator): $\hat{\Sigma} = \frac{1}{n} \sum_{i=1}^{n} X_i X_i^T$

1.2.2 Minimizing mean squared error

Rather than maximizing variance, it might sound more plausible to look for the projection with the smallest average (mean-squared) distance between the original vectors and their projections on to the principal components; this turns out to be equivalent to maximizing the variance.
We'll start by looking for a one-dimensional projection. That is, we have p-dimensional vectors, and we want to project them on to a line. Let's note β a unit vector carried by this line: the projection of a data vector X_i on to the line is $\nu_i = (\beta^T X_i).\beta$.

The distance : $\|X_i - \nu_i\|$ tells "how far" is the new vector $(\beta^T X_i).\beta$ from the actual data vector X_i and we will call this distance the error, or the residual of the projection.

Since $\beta^T \beta = 1$, we find $\nu_i = X_i^T X_i - (\beta^T X_i)^2$

We sum all the residuals of the projections, and therefore calculate the Mean Square Error (MSE). We find: $MSE(\beta) = \frac{1}{n}\sum_{i=1}^n \nu_i = \frac{1}{n}\sum_{i=1}^n X_i^T X_i - (\beta^T X_i)^2)$

Minimizing the MSE is equivalent to maximizing the second term of the sum $(\sum_{i=1}^n (\beta^T X_i)^2)$ because the first term is constant $(\sum_{i=1}^n X_i^T X_i)$.

We find that : $\frac{1}{n}\sum_{i=1}^n (\beta^T X_i)^2 = (\frac{1}{n}\sum_{i=1}^n (\beta^T X_i))^2 + Var(\beta^T X_i)$

Since we assumed that the random variables were centered, we notice that $\frac{1}{n}\sum_{i=1}^n (\beta^T X_i) = \beta^T (\frac{1}{n}\sum_{i=1}^n X_i) = 0$ (the mean of each variable is 0).

Then $\frac{1}{n}\sum_{i=1}^n (\beta^T X_i)^2 = Var(\beta^T X_i)$ and minimizing the MSE turns out to be equivalent to maximizing the variance of the projections.

1.2.3 Maximizing variance using spectral theory

Given X a normal centered random vector in $\mathbb{R}^p$ with covariance matrix Σ, we are looking for normalized linear combinations of its components with maximum variance. That is to say, what are the vectors β in $\mathbb{R}^p$ such that $Var(\beta^T X)$ is maximal ?

Proposition 1. *The normalized vectors β that maximize $Var(\beta^T X)$ are the eigenvectors associated with the highest eigenvalue of Σ.*

Proof. As X is centered, we have:

$$\begin{aligned} Var(\beta^T X) &= \mathbb{E}((\beta^T X)^2) && (1)\\ &= \mathbb{E}(\beta^T X X^T \beta) && (2)\\ &= \beta^T \mathbb{E}(X X^T)\beta \qquad (\textit{linearity}) && (3)\\ &= \beta^T \Sigma \beta \qquad (\Sigma = \mathbb{E}(X X^T)) && (4) \end{aligned}$$

We define $f(\beta) = \beta^T \Sigma \beta$, and maximize f with the constraint $\beta^T \beta = 1$. Let $\mathcal{L}$ be the Lagrangian associated, and λ a Lagrange multiplier: $\mathcal{L} = \beta^T \Sigma \beta - \lambda(\beta^T \beta - 1)$

We have $\frac{\partial \mathcal{L}}{\partial \beta} = 2\Sigma\beta - 2\lambda\beta$

Let β be a maximizer of f (which exists because f is continuous and defined on a compact set). It is a critical point of the Lagrangian for a certain λ. Therefore, for a certain λ, we have $\Sigma\beta = \lambda\beta$, and λ is, consequently, an eigenvalue of Σ and β is a corresponding normalized eigenvector.

Moreover, $f(\beta) = \lambda$: λ being a maximum of f, and the other eigenvalues being part of the image of f, λ is the highest eigenvalue of Σ.

Note that Σ being symmetric, it is diagonalizable.

□

We can obtain a much more general result, which enables us to find a collection of maximizers of variance, and justifies the fact that PCA considers all eigenvalues, and not only the highest one.

Theorem 1. *There exists an orthonormal basis $\beta_1, \beta_2, ..., \beta_p$ of eigenvectors of Σ such that, for all $i \in [\![1, p]\!]$, β_i maximizes $Var(\beta^T X)$ over all normalized vectors uncorrelated with $\beta_1, ..., \beta_{i-1}$*

Proof. This is a simple consequence of the spectral theorem applied to Σ, which is symmetric.

□

Assuming that the goal of PCA is obtaining new features with maximal variance (The assumption being validated in 1.2.2), it appears that, due to Theorem 1, the right way to do that is through the search of eigenvectors of the covariance matrix.

1.3 The algorithm

The covariance matrix Σ is generally unknown, as the distribution of the observations is not known in advance. Therefore, one cannot directly compute its eigenvectors, and can only rely on an estimation of the latter.

1.3.1 Basic computation

The precedent subsection allows us to derive the following algorithm, which is the PCA algorithm in its simplest formulation:

Instead of computing the eigenvectors of Σ (which is impossible), we compute those of $\hat{\Sigma}$: we will see in Section 2 that, depending on certain conditions, they are consistent estimators of the real eigenvectors.

Algorithm 1 Principal Component Analysis (general form)

```
***Input*** : Observations (in $\mathbb{R}^p$) $X_1, X_2, ..., X_n$, columns of the matrix $\mathcal{X}$
Step 1 : Compute the mean vector $m = \frac{1}{n}\sum_{i=1}^{n} X_i$ (We don't assume the data is centered)
Step 2 : Compute $\hat{\Sigma} = \frac{1}{n}\sum_{i=1}^{n}(X_i - m)(X_i - m)^T$, which is an unbiased estimator of $\Sigma$
Step 3 : Compute $\beta_1, \beta_2, ..., \beta_p$ the eigenvectors of $\hat{\Sigma}$ and $\lambda_1, \lambda_2, ..., \lambda_p$ the corresponding eigenvalues in decreasing order
Step 4 : Compute the transformed data $\tilde{\mathcal{X}} = (\beta_1|\beta_2|...|\beta_p)^T \times X$
***Output*** : Transformed data $\tilde{\mathcal{X}}$ and variances of the new features $\lambda_1, \lambda_2, ..., \lambda_p$.
```

1.3.2 Dimension reduction

The output data of the algorithm, which was described in its simplest form, has the same shape as the input data. While PCA is often depicted as a dimension reduction method, one can notice that dimensions were not reduced at all in this case.
Actually, we can select a certain number of lines k (the principal components !) of the output matrix and exclude all other features with low variance. This implies that a decision has to be made as for the choice of the final number of features k. For that purpose, one can plot an Explained Variance plot: it consists in the points $(i, g(i))$, which allow to know at which point the components become less significant, where

$$g(i) = \frac{\sum_{j=1}^{i} \lambda_j}{\sum_{j=1}^{p} \lambda_j} \tag{5}$$

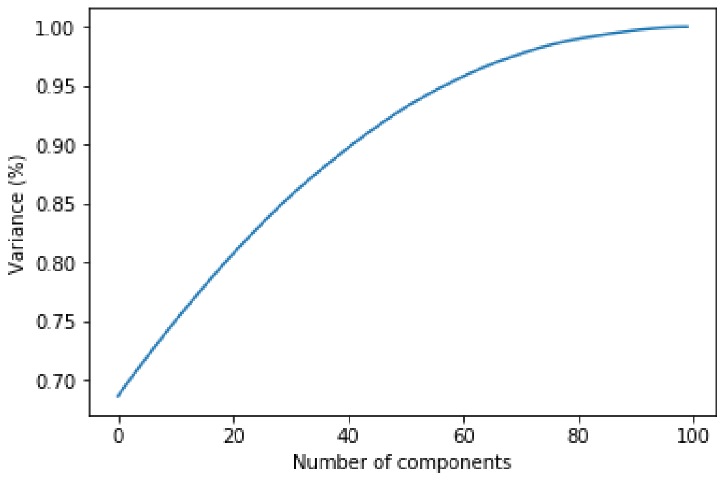


Figure 1: Explained variance plot for simulated normal observations ($p = 100, n = 100000$)

Here, it seems reasonable to choose $k = 60$

2 Consistency of the algorithm

Under certain conditions on n and the covariance matrix Σ, we can show that the PCA algorithm is consistent, meaning that it successfully provides the components with the highest variance. Here, we will quantify the error between the covariance estimator and the real covariance matrix by finding an upper-bound of it. Afterwards, this will allow us to find upper-bounds of the errors between eigenvalues/eigenvectors estimators and real eigenvalues/eigenvectors, thus validating the PCA algorithm for sufficiently low upper-bounds.

2.1 Convergence of the covariance estimator and error

Thanks to the strong law of large numbers, knowing that $\mathbb{E}(XX^T) = \Sigma$, we deduce that $\hat{\Sigma} = \frac{1}{n}\sum_{i=1}^{n} X_i X_i^T$ converges almost surely to Σ when $n \to \infty$.

Remark. *We denote as $\|.\|$ the euclidean norm in the case of a vector space, or the operator norm associated to the euclidean norm in the case of a matrix space.*

This convergence can be precised with an upper-bound (2) of the error between Σ and $\hat{\Sigma}$.

Theorem 2. *For all $t \geq 1$, with some constant $C > 0$, and with probability at least $1 - e^{-t}$:*

$$\left\|\hat{\Sigma} - \Sigma\right\| \leq C \left\|\Sigma\right\| \left[\sqrt{\frac{r_e(\Sigma)}{n}} \bigvee \frac{r_e(\Sigma)}{n} \bigvee \sqrt{\frac{t}{n}} \bigvee \frac{t}{n}\right]$$

where $r_e(\Sigma) = \frac{Tr(\Sigma)}{\|\Sigma\|}$

From now on, we will admit this theorem.

2.2 Eigenvalue error

The aim, here, is to bound the error between the estimators of the eigenvalues/eigenvectors and the eigenvalues/eigenvectors. To do that, we will bound the error with the covariance estimation error, of which we already obtained an upper-bound.

Theorem 3. *(Weyl) Let $\|.\|$ be the operator norm in the matrix space. Given S and T two symmetric matrices of size p, with $(\lambda_i(S))_i$, $(\lambda_i(T))_i$ the non-increasing sequences of their eigenvalues, we have, for all $i \in [\![1, p]\!]$:*

$$|\lambda_i(S) - \lambda_i(T)| \leq \|S - T\| \tag{6}$$

Proof. We will be using the Courant-Fischer's *min-max theorem*, according to which, for a symmetric matrix A:

$$\lambda_i(A) = max_{dimE=i}\ min_{x \in S(E)} < Ax, x > \tag{7}$$

$S(E)$, here, is the unit sphere of E. With no loss of generality, let's assume that $\lambda_i(S) \geq \lambda_i(T)$.

Let E_S be a subspace of dimension i such that $\lambda_i(S) = min_{x \in S(E_S)} < Sx, x >$.

We have $\lambda_i(T) \geq min_{x \in S(E_S)} < Tx, x >$. Let x^* be an element of $S(E_S)$ such that

$$min_{x \in S(E_S)} < Tx, x > = < Tx^*, x^* > \tag{8}$$

Therefore, $\lambda_i(S) - \lambda_i(T) \leq min_{x \in S(E_S)} < Sx, x > - < Tx^*, x^* >$

Then, using the min definition:

$$\begin{aligned} \lambda_i(S) - \lambda_i(T) &\leq < Sx^*, x^* > - < Tx^*, x^* > & (9) \\ &\leq < (S - T)x^*, x^* > & (10) \\ &\leq \|(S - T)x^*\| \qquad (Cauchy - Schwarz) & (11) \\ &\leq \|S - T\| & (12) \end{aligned}$$

This achieves the proof.

$\square$

We easily deduce from Weyl's inequality an upper-bound for the error between the estimator of the eigenvalue and the real eigenvalue, which proves the convergence:

Theorem 4. *Let $(\lambda_j)_j$ (resp. $(\hat{\lambda_j})_j$) be the decreasing sequence of eigenvalues of Σ (resp. $\hat{\Sigma}$). Then, we have:*

$$|\hat{\lambda_j} - \lambda_j| \leq \left\| \hat{\Sigma} - \Sigma \right\| \tag{13}$$

2.3 Eigenvector error

Having now proven a convergence result for the eigenvalues, the precision of the eigenvector estimator still has to be evaluated. For that purpose, the Davis-Kahan (1) theorem is useful.

Lemma 1. *(Davis-Kahan $sin(\theta)$ theorem)*
Given S and T two symmetric matrices of size p, with $(\lambda_i(S))_i$, $(\lambda_i(T))_i$ the non-increasing sequences of their eigenvalues, we define:

$$\delta_i(S, T) = min_{j \neq i} |\lambda_j(T) - \lambda_i(S)| \tag{14}$$

We assume that this quantity is positive, and define θ ($\in [0, \frac{\pi}{2}]$) as the angle between the unitary eigenvectors $v_i(S)$ (associated to $\lambda_i(S)$) and $v_i(T)$ (associated to $\lambda_i(T)$). Then:

$$\sin\theta \leq \frac{\|S - T\|}{\delta_i(S, T)} \tag{15}$$

Proof. Let $H = T - S$. Let's note $x = v_i(S)$, $y = v_i(S)$, $\lambda = \lambda_i(S)$, $\tilde{\lambda} = \lambda_i(T)$ and $\delta = \delta_i(S, T)$. Using the spectral theorem, we find E (*resp.* F) a $p \times (p-1)$ matrix of eigenvectors of S (*resp.* T) that form an orthonormal basis with x (*resp.* y), such that:

$$S = \lambda x x^T + E \Lambda E^T \tag{16}$$
$$T = \tilde{\lambda} y y^T + F \tilde{\Lambda} F^T \tag{17}$$

where Λ and $\tilde{\Lambda}$ are diagonal matrices.

We have:

$$Hx = Tx - \lambda x \tag{18}$$

Given the orthogonality of the eigenvector family of T:

$$F^T H x = F^T T x - \lambda F^T x \tag{19}$$
$$= \tilde{\Lambda} F^T x - \lambda F^T x \tag{20}$$

Considering the norm:

$$\left\| F^T H x \right\| = \left\| \tilde{\Lambda} F^T x - \lambda F^T x \right\| \tag{21}$$
$$= \left\| (\tilde{\Lambda} - \lambda I) F^T x \right\| \tag{22}$$

The absolute value of the eigenvalues of $\tilde{\Lambda} - \lambda I$ is superior or equal to δ, given the definition of the latter. Therefore, $\tilde{\Lambda} - \lambda I$ is invertible, and:

$$\left\| (\tilde{\Lambda} - \lambda I)^{-1} \right\| \leq \frac{1}{\delta} \tag{23}$$

Which implies:

$$\left\| (\tilde{\Lambda} - \lambda I) F^T x \right\| \geq \frac{\left\| F^T x \right\|}{\left\| (\tilde{\Lambda} - \lambda I)^{-1} \right\|} \tag{24}$$
$$\geq \delta \left\| F^T x \right\| \tag{25}$$

On the one hand, we have $\left\| F^T x \right\| = |\sin(\theta)|$. Indeed, knowing that the columns of F and y form an orthonormal basis, we have:

$$\left\| F^T x \right\|^2 = x^T F F^T x \tag{26}$$
$$= x^T (I - y y^T) x \tag{27}$$
$$= 1 - (x|y)^2 \tag{28}$$

On the other hand, we have $\left\|F^T Hx\right\| \leq \|H\|$. Indeed, thanks to Cauchy-Schwarz and the fact that $F^T F = I$:

$$\begin{aligned}
\left\|F^T Hx\right\|^2 &= (x|HFF^T Hx) && (29)\\
&\leq \|H\| \left\|FF^T Hx\right\| && (30)\\
&= \|H\| \left\|F^T Hx\right\| && (31)
\end{aligned}$$

Combining the obtained inequalities, we finally get the result:

$$\sin\theta \leq \frac{\|H\|}{\delta} \tag{32}$$

i.e

$$\sin\theta \leq \frac{\|S-T\|}{\delta_i(S,T)} \tag{33}$$

□

Theorem 5. *(Davis-Kahan) With the same previous notations, and defining the quantity $gap_i(S)$ that we assume positive:*

$$gap_i(S) = min_{j\neq i}|\lambda_j(S) - \lambda_i(S)| \tag{34}$$

Then:

$$\sin\theta \leq \frac{2\|S-T\|}{gap_i(S)} \tag{35}$$

Proof. Let's assume that $gap_i(S) > 2\|S-T\|$, otherwise the result is trivial.
Let $j \in [\![1,p]\!]$, $j \neq i$. Then, using Weyl's theorem:

$$\begin{aligned}
gap_i(S) &\leq |\lambda_i(S) - \lambda_j(T) + \lambda_j(T) - \lambda_j(S)| && (36)\\
&\leq |\lambda_i(S) - \lambda_j(T)| + \|S-T\| && (37)
\end{aligned}$$

Which is equivalent to:

$$\begin{aligned}
|\lambda_i(S) - \lambda_j(T)| &\geq gap_i(S) - \|S-T\| && (38)\\
&\geq \frac{gap_i(S)}{2} > 0 && (39)
\end{aligned}$$

By choosing the right j in the above equality, we obtain that:

$$\delta_i(S,T) \geq \frac{gap_i(S)}{2} \tag{40}$$

Using the lemma, we obtain the intended result.

□

Theorem 6. *Let P_i (resp. $\hat{P}_i$) be the spectral projector associated with the eigenvalue λ_i (resp. $\hat{\lambda}_i$) of Σ (resp. $\hat{\Sigma}$). Then:*

$$\left\| \hat{P}_i - P_i \right\|_{op} \leq \frac{2^{\frac{3}{2}} \left\| \hat{\Sigma} - \Sigma \right\|}{gap_i(\Sigma)} \tag{41}$$

Proof. Let x be a normalized p-dimensional vector. Then $P_i(x)$ and $\hat{P}_i(x)$ are eigenvectors of Σ and $\hat{\Sigma}$. Let θ be the angle between the two vectors. According to the Davis-Kahan theorem:

$$\sin\theta \leq \frac{2 \left\| \hat{\Sigma} - \Sigma \right\|}{gap_i(\Sigma)} \tag{42}$$

Knowing that $\cos(\theta) =< P_i(x), \hat{P}_i(x) >$, we have:

$$\sin(\theta) = \sqrt{1 - cos^2(\theta)} \tag{43}$$
$$\geq \sqrt{1 - cos(\theta)} \tag{44}$$
$$\geq \sqrt{\frac{1}{2} \left\| \hat{P}_i(x) - P_i(x) \right\|} \tag{45}$$

Using the Davis-Kahan result, we deduce:

$$\left\| \hat{P}_i(x) - P_i(x) \right\| \leq \frac{2^{\frac{3}{2}} \left\| \hat{\Sigma} - \Sigma \right\|}{gap_i(\Sigma)} \tag{46}$$

That being true for all x, we deduce the final result. □

2.4 Estimation error and consistency

Combining the covariance estimator inequality with the two eigenvector/eigenvalue inequalities, we obtain an upper-bound of the eigenvector/eigenvalue estimation error, with high probability.

That is to say, choosing $t = 5$ in theorem 5, and supposing $n \geq 5$, we obtain:

Theorem 7. *With some constant $C > 0$, and with probability at least 0.99:*

$$\left\|\hat{\Sigma} - \Sigma\right\| \leq C\left\|\Sigma\right\| \left[\sqrt{\frac{r_e(\Sigma)}{n}} \bigvee \frac{r_e(\Sigma)}{n} \bigvee \sqrt{\frac{5}{n}}\right]$$

With the same probability, for $j \in [\![1, p]\!]$:

$$|\hat{\lambda_j} - \lambda_j| \leq C\left\|\Sigma\right\| \left[\sqrt{\frac{r_e(\Sigma)}{n}} \bigvee \frac{r_e(\Sigma)}{n} \bigvee \sqrt{\frac{5}{n}}\right]$$

and

$$\left\|\hat{P}_j - P_j\right\|_{op} \leq \frac{2^{\frac{3}{2}} C\|\Sigma\| \left[\sqrt{\frac{r_e(\Sigma)}{n}} \bigvee \frac{r_e(\Sigma)}{n} \bigvee \sqrt{\frac{5}{n}}\right]}{gap_j(\Sigma)}$$

Depending on n, p and Σ, this upper-bound will be sufficient (or not) to ensure the consistency of PCA.

Remark. *When p and Σ are fixed, and $n \to \infty$, theorem 7 proves the consistency of PCA. While the convergence of the sample covariance was obvious because of the law of large numbers, the convergence of the eigenvectors/eigenvalues was not as obvious.*

Remark. *In most cases, $5 \leq r_e(\Sigma) \leq n$, which gives a much simpler upper bound $C\left\|\Sigma\right\| \sqrt{\frac{r_e(\Sigma)}{n}}$ In the case where $r_e(\Sigma) \geq n$, which can become true when p and Σ are not fixed as n varies, the upper bound obtained no longer guarantees the consistency of PCA.*

Remark. *We have $r_e(\Sigma) \leq p$, so $r_e(\Sigma)$ can be replaced by p in the upper bound. This less precise upper bound shows that, as long as p is fixed, no matter how Σ varies, PCA will still be consistent.*

3 Numerical simulation and consistency evaluation

This section aims at confirming the results (theorem 7) of the previous section through numerical simulation. More precisely, in different contexts, we simulate n observations of a p-dimensional vector that has a centered Gaussian distribution, with covariance matrix Σ, to which we apply the algorithm of PCA. By varying n, p and Σ, we find numerical results that complete the theoretical ones found in section 2, about the covariance estimation error as well as the eigenvalues/eigenvectors estimation errors. The simulation code (in Python) can be found in the appendix.

3.1 Spiked covariance model

All our simulations are done with a specific class of covariance matrices Σ: the covariance model chosen for simulation is the spiked covariance model.

Definition 1. *A matrix Σ is said to follow the spiked covariance model if we can find $k \geq 1$, $(s_i)_{1\leq i\leq k}$ a family of positive scalars, $(v_i)_{1\leq i\leq k}$ a family of orthonormal vectors such that:*

$$\Sigma = I + \sum_{i=1}^{k} s_i v_i v_i^T \tag{47}$$

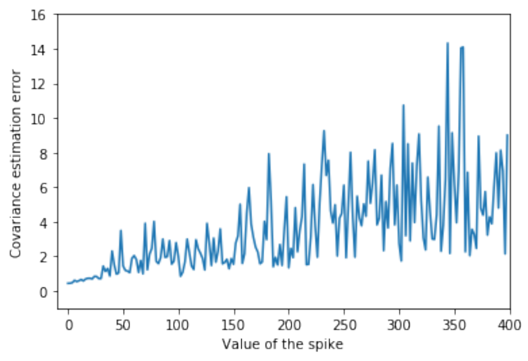
In that model, the eigenvalues of Σ are $(1+s_i)_{1\leq i\leq k}$, with associated eigenvectors $(v_i)_{1\leq i\leq k}$, as well as 1, the associated eigenvector space being the orthogonal space of $\{v_1, ..., v_k\}$.

For s a sequence of positive scalars, we denote $\Sigma \sim Spike(s,p)$ when Σ is of size p, and can be written $\Sigma = I + \sum_{i=1}^{k} s_i v_i v_i^T$.

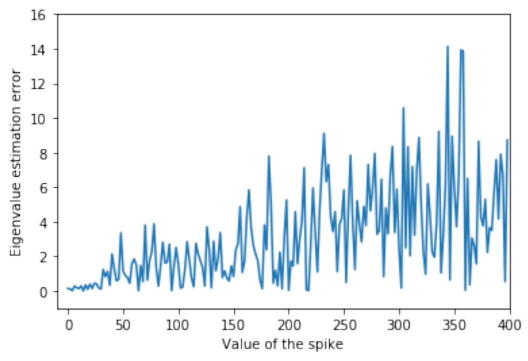
3.2 Varying the spike in the simple spiked model

Here, we simulate several "simple" spiked models, meaning that the sequence s contains only one positive scalar. For values of s in the set $\{0, 2, 4, ..., 398, 400\}$, we simulate $n = 10000$ observations of a centered Gaussian distribution with a matrix $\Sigma \sim Spike(s,p)$, with $p = 50$. The idea here is to study the influence of the value of s on the covariance estimation error, as well as the eigenvector/eigenvalue estimation error for the largest eigenvalue $(1+s)$.

Here are the plots obtained for the covariance estimation error as well as the first eigenvalue estimation error:



(a) Covariance estimation error



(b) Eigenvalue estimation error

Figure 2: Covariance and eigenvalue estimation errors depending on the spike in simple spiked model

First, there is an increasing trend of the covariance estimation error, which is coherent with the upper bound for $\left\|\hat{\Sigma} - \Sigma\right\|$ found in theorem 7. Indeed, as s increases, $\|\Sigma\| \sqrt{\frac{r_e(\Sigma)}{n}}$ increases as well. We can only observe a trend and the estimation error does not have the same behaviour as $\|\Sigma\| \sqrt{\frac{r_e(\Sigma)}{n}}$, which is not a contradiction as there was no reason for $\left\|\hat{\Sigma} - \Sigma\right\|$ to vary like its upper bound.

Second, it is interesting to note that $\left\|\hat{\Sigma} - \Sigma\right\|$ and $|\hat{\lambda_1} - \lambda_1|$ seem to have very similar behaviors, which is, once again, coherent with Weyl's theorem. We can then conjecture the existence of a lower bound for $|\hat{\lambda_1} - \lambda_1|$ that behaves like $\left\|\hat{\Sigma} - \Sigma\right\|$.

As for the eigenvector (associated with λ_1) estimation error , we obtain the following plot:

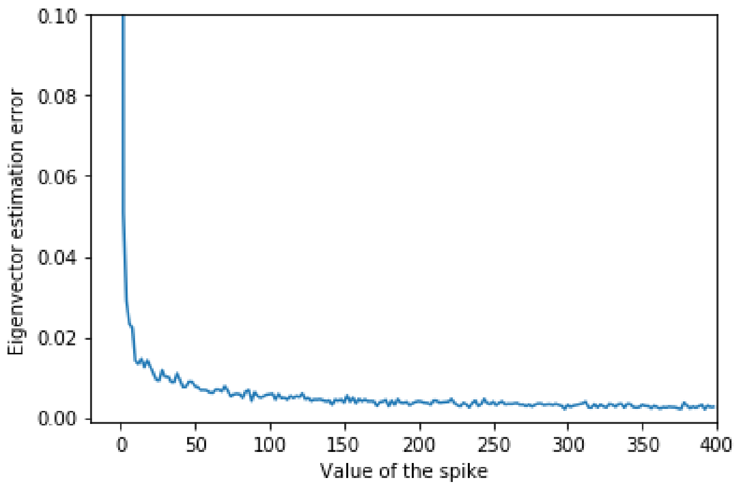
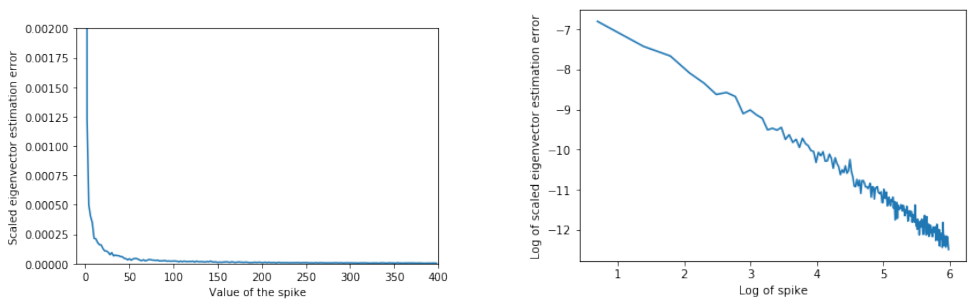


Figure 3: Eigenvector estimation errors depending on the spike in simple spiked model.

The main difference between the upper bound for the eigenvector estimation error and the other upper bounds found in theorem 7 is the presence of a factor $\frac{1}{gap_i(\Sigma)}$. In this simulation, we have $gap_1(\Sigma) = s$, and the plot obtained seems to show a $\frac{1}{s}$ behavior of the eigenvector estimation error. However, the upper bound found in theorem 7 for the eigenvector estimation error also contains the factor $\|\Sigma\| \sqrt{r_e(\Sigma)}$, which prevents us from truly observing the effect of the gap on the eigenvector estimation error. Therefore, we plot the "scaled" eigenvector estimation error, which corresponds to the original error divided by $\|\Sigma\| \sqrt{r_e(\Sigma)}$. The result obtained, which can be seen below, shows a decreasing in $\frac{1}{s}$ of the eigenvector estimation error.



(a) Normal scale (b) Logarithmic scale

Figure 4: Scaled eigenvector estimation error

3.3 Influence of the effective rank

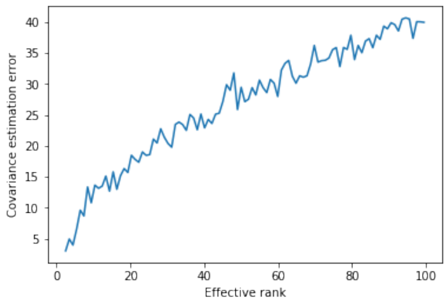
Our goal, in this part, is to observe the influence of the effective rank on the different errors (covariance, eigenvalue, eigenvector). For the simulation, we choose a class of matrices Σ following spiked covariance models, with identical sizes, norms and gap_1, but with varying effective ranks. In this subsection, $p = 100$ and $n = 10000$.

For $j \in [\![1, p-1]\!]$, we simulate n observations with covariance matrix $\Sigma_j \sim Spike(s^j, p)$, where s^j has a size of $j + 1$ and x is a positive number (we choose $x = 200$):

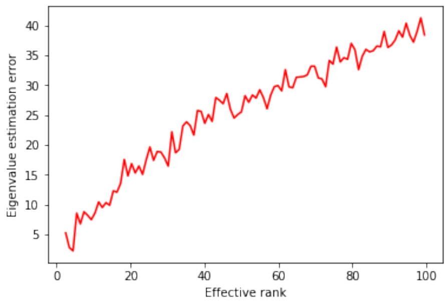
$$s_i^j = \begin{Bmatrix} \text{x} & \text{if} \quad \text{i} = 1 \\ \text{x-1} & \text{else} \end{Bmatrix} \tag{48}$$

The matrices Σ_j have the same norm $(x + 1)$ and the gap corresponding to the first eigenvalue is always equal to 1. Only the effective rank varies, which allows us to observe its influence

For the covariance and eigenvalue $(1 + s)$ estimation error, the plots are the following ones:



(a) Covariance estimation error



(b) Eigenvalue estimation error

Figure 5: Covariance and eigenvalue estimation error with respect to the effective rank.

Once again, we observe the same behavior for the covariance estimation error and eigenvalue estimation error, which strengthens the hypothesis of a lower bound for $|\hat{\lambda_1} - \lambda_1|$.
Moreover, the covariance estimation error seems to have a square root dependence on the effective rank, which corroborates the presence of the factor $\sqrt{r_e(\Sigma)}$ in the upper bound found in the theorem 7.

For the eigenvector estimation error, we obtain the following plot:

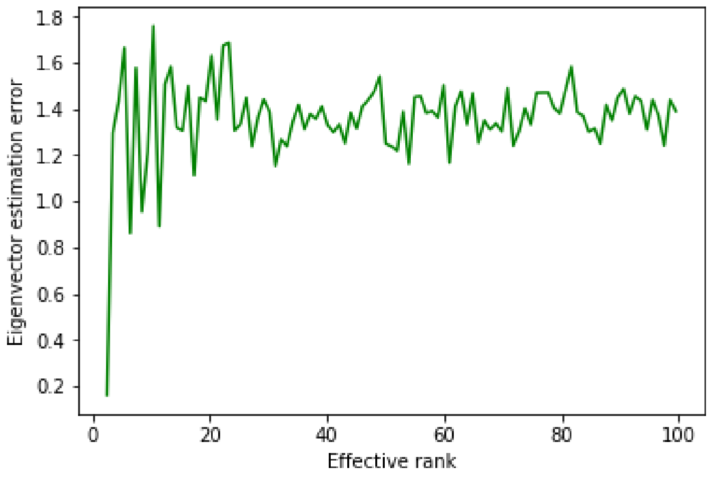


Figure 6: Eigenvector estimation error with respect to the effective rank.

A similar trend (square root) can be observed, but there is no clear indication of a square root behavior.

3.4 Consistency

The aim of this subsection is to numerically show, in the case of a constant p and Σ, the consistency of PCA. For this simulation, we use a simple spiked covariance model, and for several values of p, we simulate $N = 10000$ observations $(X_i)_{1 \leq i \leq N}$ and plot $\left\| \frac{1}{n} \sum_{i=1}^{n} X_i X_i^T - \Sigma \right\|$ with respect to $n \in [\![1, N]\!]$.

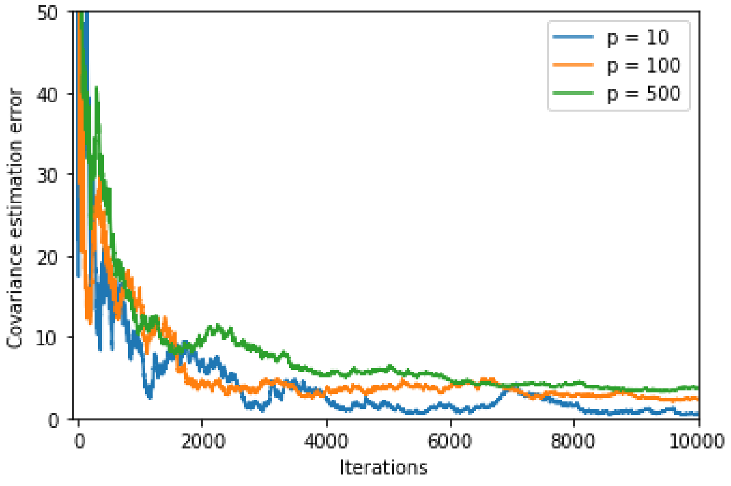


Figure 7: Covariance estimation error with respect to the number of iterations.

This confirms the result of the law of large numbers. The consistency of PCA truly relies on the convergence of the eigenvalues and eigenvector estimators. Here are the plots of the corresponding errors with respect to the number of iterations, which clearly show the consistency of PCA in the case of fixed parameters.

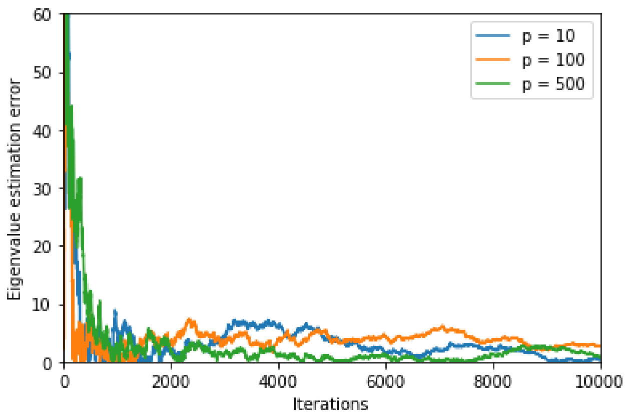


Figure 8: Eigenvalue estimation error with respect to the number of iterations.

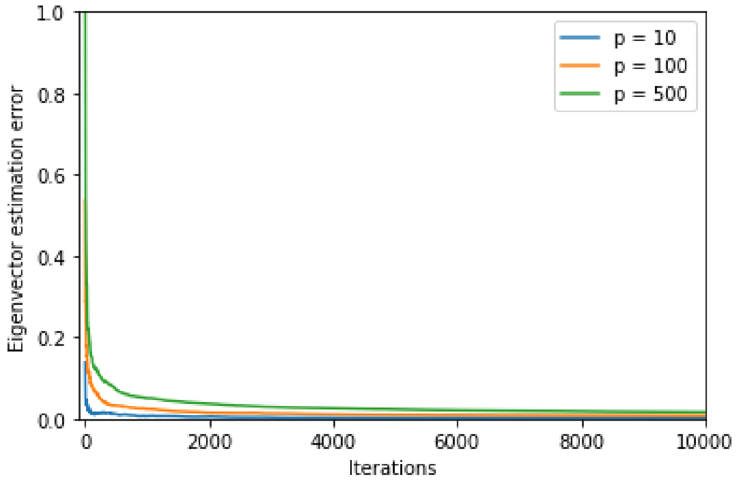


Figure 9: Eigenvector estimation error with respect to the number of iterations.

3.5 An inconsistency case

However, the consistency of PCA is conditional to the parameters of the covariance matrix. In particular, when p and Σ are not fixed when $n \to \infty$. We aim at illustrating this through simulation based on a simple spiked covariance model. For 40 values of n equally spaced in $[\![100, 5000]\!]$, we simulate n observations with a covariance matrix of size p with $p = \lfloor \frac{n}{3} \rfloor$ following a simple spiked covariance model and compute the covariance estimation error. We plot the error for these simulations, and compare this error with the covariance estimation error corresponding to a simulation with fixed p ($p = 50$):

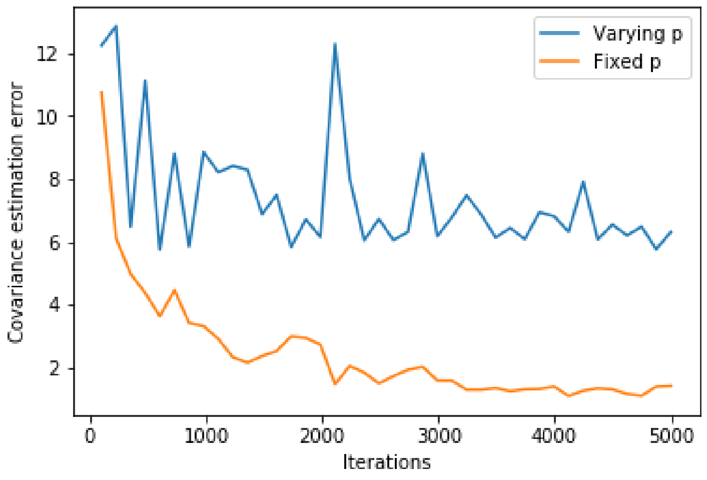


Figure 10: Covariance estimation error for a varying p.

In this case, $\frac{p}{n} \xrightarrow[n\to\infty]{} \frac{1}{3}$: it has been shown (7) in particular cases that, when $\frac{p}{n} \xrightarrow[n\to\infty]{} c$ with $c > 0$, PCA is inconsistent. Through numerical simulation, we have illustrated this result.

4 An application of PCA: anomaly detection

4.1 Context

4.1.1 Dataset and problem

We work here with a dataset from *UCI machine learning repository* composed of 5803 data points of dimension 36 representing satellite images. Each data point corresponds to a 3x3 pixels neighbourhood of a satellite image, so that all the data was extracted from a single satellite image. The features, for a given data point, are the four digital images (numeric representation) for each one of the 9 pixels composing the neighborhood. Some of these points are considered as anomalies, outliers: they are all the points that do not correspond to "soil" in the original image.

The goal of anomaly detection is to identify the outliers: in that specific case, we will be using PCA for a classification purpose.

In order to assess whether our method of outliers detection is accurate, we also have a 5803-vector composed (y) of 0 and 1 that gives the information of whether a data point is an inliner (0) or an outlier (1). In this case, our dataset contains 71 outliers (ie 1.2%)

4.1.2 PCA

To address the problem of anomaly detection, PCA is a tool that is useful, either for visualising the data by projecting the points on a map and physically identifying the outliers, or computing an algorithm that finds the outliers by relying on the principal components.

4.1.3 Visualisation

Projecting the data points on a 2D map would probably give no clue about where the outliers lie. In order to see clear and cut linear boundaries that separate inliers from outliers, we can project the data points on the two principal components (the two eigenvectors corresponding to the two greatest eigenvalues of the covariance matrix).
Before doing this, we can make sure that the two principal components contain the major part of the information. The graphic below confirms that the two principal components are responsible for more than 85% of the variance.

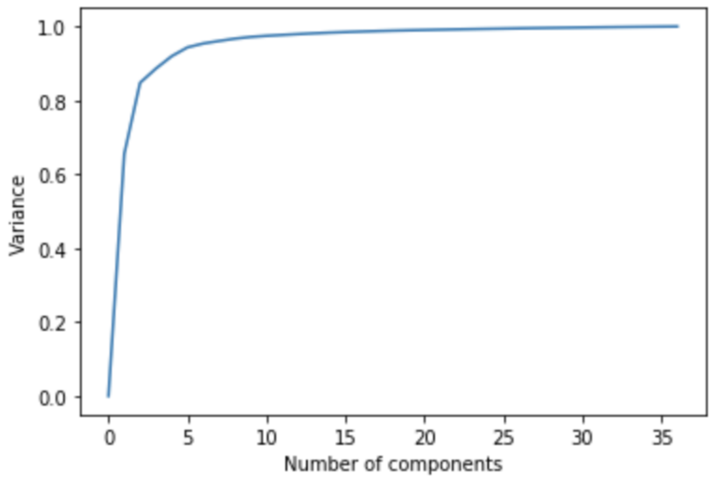


Figure 11: Explained variance.

By drawing the scatterplot of the data points projected on the two principal components, we can indeed identify the points (in yellow) which are far from the main group of points (in purple). These points are the outliers.

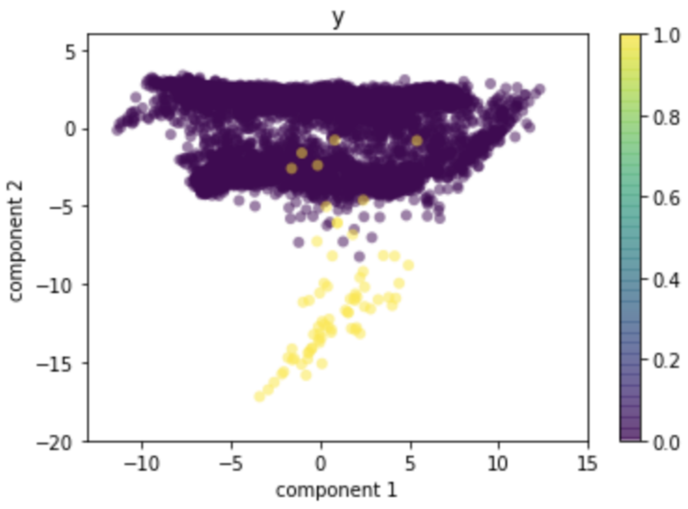


Figure 12: Projection of the data points on the principal components

From this visualisation, we have learned two facts : we can highly reduce the dimension of the space of features (from 36 to 2) and improve considerably the computation time and still keep the major part of the data points information (more than 85%). Plus, finding outliers can be reduced to a distance calculation problem : the more "distant" a point is from the center of the main group of points, the more likely it is an outlier (with a notion of distance that still has to be defined).

4.2 Mahalanobis distance - unsupervised method

Classically, when talking about computing distances, one would think about the most common distance measure which is the Euclidean distance. Nevertheless, this definition is relevant for calculating distance between two points but it gets more complicated when it comes to calculating distance between a point and a distribution. A fitting measure in this case would be the Mahalanobis distance.

Definition 2. *The Mahalanobis distance of a point x to a distribution of mean μ and non-singular covariance Σ is defined as:*

$$D(x) = \sqrt{(x-\mu)^T \Sigma^{-1} (x-\mu)} \tag{49}$$

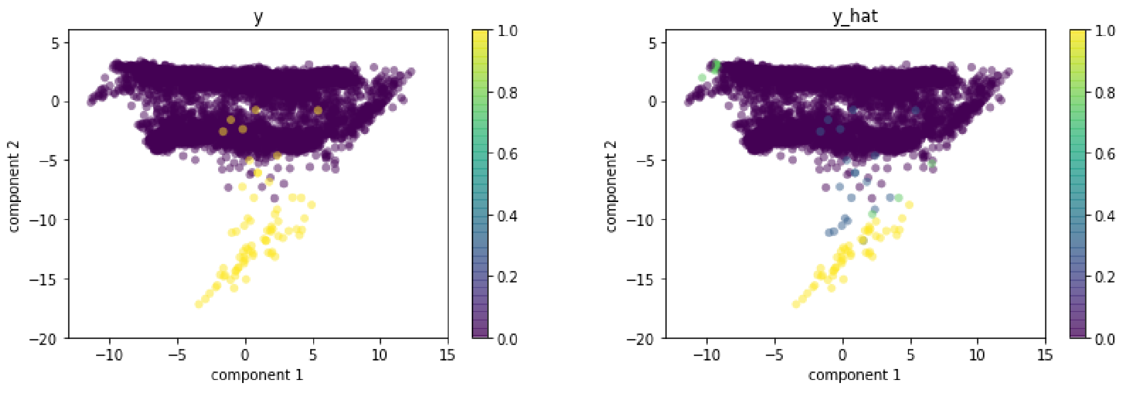
However, computing the Mahalanobis distance can take a lot of time since it inverts a high dimensional matrix (in time $O(p^3)$), especially when it has to be done many times (in a loop for example, see below). Here is another justification of the necessity of applying the PCA algorithm, bringing the space of features to a 2-dimensional space. Computing the inversion of a 2-2 covariance matrix becomes immediate. We propose the following algorithm:

Algorithm 2 Identifying the outliers using Mahalanobis distance

```
Input : X : Data matrix, N : number of points that are outliers in the dataset
Initialization : $A = \emptyset$ (the set of outliers)
First : : Apply the PCA and project the data points on the principal components (2 in our case)
while $Card(A) < N$ do
    Find the point $x$ of $X$ that has the greatest Mahalanobis distance.
    Add the point $x$ to A: $A = A \cup \{x\}$
    Remove the point $x$ from $X$: $X = X \setminus \{x\}$

    Output : The set $A$ of outliers.
```

At each step of the loop, we remove the outlier found from the considered data because it affects significantly the Mahalanobis distance computed afterwards.



(a) Real labels (b) Predicted labels

Figure 13: Outliers detected using Mahalanobis distance based algorithm vs actual outliers .

After implementation, the results found are mixed. Indeed, the first half of outliers in yellow (the more distant from the distribution according to Mahalanobis distance) is well labeled and corresponds to actual outliers. However, there are many false negatives (in blue) and some false positives (in green).

Moreover, this algorithm implies that we have prior knowledge about the number of outliers in the data set, which is usually not the case.

4.3 K-means - semi supervised method

Another way of identifying anomalies is to use a clustering algorithm, such as k-means, on the PCA transformed points. The idea is that, if k is correctly chosen, one of the clusters will correspond to the set of outliers. We split the PCA-transformed data into a training and a testing set. Then, we find the best k, and the whole process of choosing k is what we call the training, which therefore relies on the training set. To do so, we assess the precision and the recall for each value of k by comparing the actual outliers to those labeled as such by the algorithm. It is interesting to note that, by using the k-means algorithm, we will be relying on the Euclidean distance for labelling the points. However, there is a classification problem that comes from the unsupervised nature of the k-means algorithm: the clusters it returns are not labeled as outliers or inliers. We, therefore, label as the outlier cluster the one that contains the furthest point from the distribution in terms of Mahalanobis distance: it seems to be a reasonable heuristic method for identifiying the set of outliers.

Algorithm 3 Finding the outliers using k-means

```
***Input :*** $X$ : Data matrix, $y$ : Vector of 0 and 1
Step 1: Apply the PCA and project the data points on the principal components (2 in our case)
Step 2: Identify the point $x_{max}$ of $X$ with maximum Mahalanobis distance
Step 3: For each value of $k$ between 2 and 25,
    • Apply the k-means algorithm
    • Find the cluster containing $x_{max}$: this cluster corresponds to the predicted outliers
    • Compare the set predicted with $y$ by computing the precision and recall
Step 4: Identify the value $k^*$ of $k$ that has the best compromise between precision and recall
***Output :*** The cluster containing $x_{max}$ computed for $k = k^*$.
    =0
```

In order to find the optimal number of clusters, we can assess the recall and precision using k-means on the training data for each k, and then choose the most suitable one. Here is the plot of recall and precision measures with respect to k, obtained in the training phase of the algorithm:

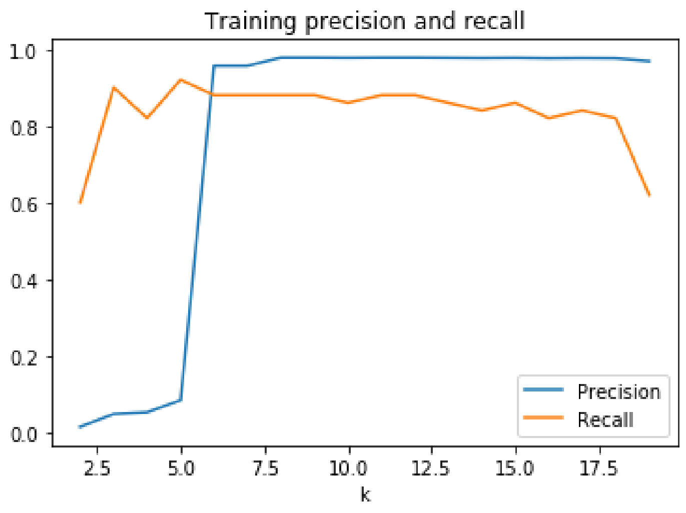


Figure 14: How to choose k

A suitable choice of k here would be $k = 8$ for instance. Then, we perform a prediction of outliers on the testing set using the previous algorithm with $k = 8$: obviously, there is no point in using the real labels in the testing phase. On the testing set, we obtain a precision of 0.95 and a recall of 0.91.

The figure below represents the clusters obtained. One of them (in yellow) corresponds almost exactly to the real outliers, and we successfully flag it as the outlier cluster thanks to the Mahalanobis technique.

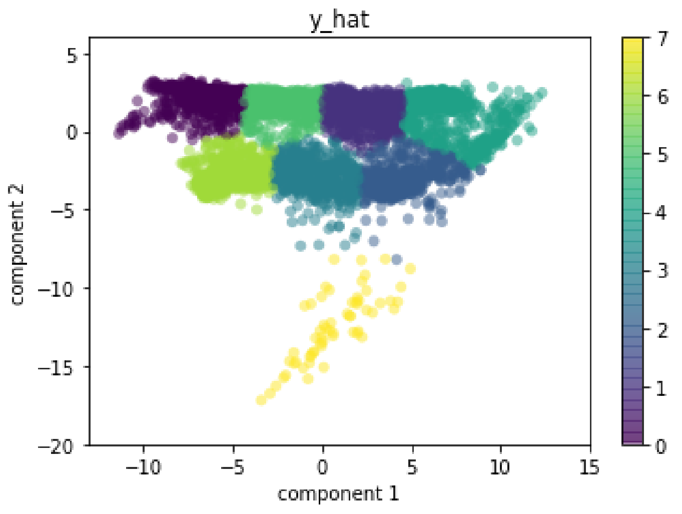


Figure 15: Projection of the clusters with $k = 8$

4.4 Logistic regression - supervised method

Finally, we decided to address the problem by using a supervised algorithm combined to PCA : logistic regression. The idea is similar to the one of the previous algorithm, insofar as we apply the PCA algorithm to the data before any other processing. Then, we simply label the data using a trained model (on a training set) of logistic regression.
As we can see in the scatterplots drawn below, the principal component projection allowed to separate clearly the inliers from the outliers to the point where a line can be drawn between the two groups: we have a linear decision boundary. Thus, the idea of applying a logistic regression seems natural.

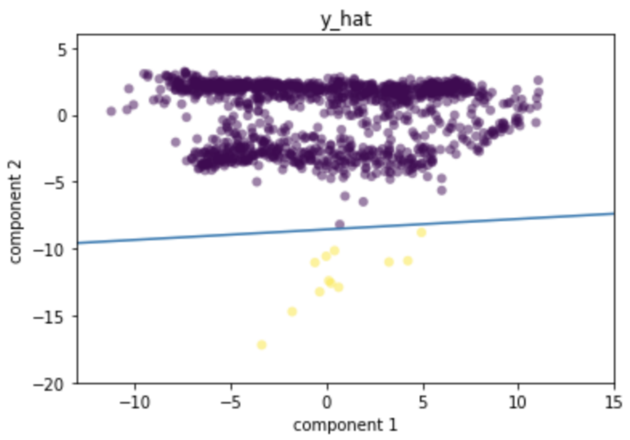


Figure 16: Decision boundary obtained with logistic regression

Using this method, we obtain, on the testing set, a precision of 0.98 and a recall of 0.90. However, these figures were obtained with only one testing set: cross-validation would be an effective method to obtain trustworthy figures.

A Code for numerical simulation (section 3)

```
import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from scipy.stats import ortho_group

def get_sigma_hat(sigma, n=10000):
    """Generate observations and get sample covariance matrix"""
    X = np.random.multivariate_normal([0]*sigma.shape[0], sigma, n).T
    sigma_hat = np.cov(X)

    return sigma_hat

def get_eigen(mat):
    """Get the eigenvectors and eigenvalues in decreasing order of eigenvalues """
    eigenval, eigenvec = np.linalg.eigh(mat)
    eigenvec = eigenvec.T
    zipped = zip(eigenval,eigenvec)
    eigenvec = [x for _,x in sorted(zipped, key = lambda y : y[0], reverse = True)] #
        Sorting vectors/values in decreasing order of values
    eigenval = sorted(eigenval, reverse = True)
    return np.array(eigenval), np.array(eigenvec)

def get_norm_mat(mat): # Works for mat symmetric
    """Get the norm of a symmetric matrix"""
    eigenval, eigenvec = get_eigen(mat)

    return max(np.abs(eigenval))

def get_spike_matrix(s, k=1, p=50):
    """Get a spiked covariance matrix given an input of coefficients """
    v = ortho_group.rvs(dim=p)[:,:k]
    res = np.identity(p)
    for i in range(k):
        res += s[i]*np.dot(v[:,i].reshape(p,1),v[:,i].reshape(p,1).T)

    return res

def get_spike_eigen(s, k=1, p=50):
    """Same as get_spike_matrix but also outputs eigenvalues/eigenvectors"""
    b = ortho_group.rvs(dim=p)
    v = b[:,:k]
    res = np.identity(p)
    for i in range(k):
        res += s[i]*np.dot(v[:,i].reshape(p,1),v[:,i].reshape(p,1).T)

    eigenvec = [list(b[:,j-1:j]) for j in range(1,p+1)]
    eigenval = [j+1 for j in s] + [1]*(p-len(s))
    zipped = zip(eigenval,eigenvec)
    eigenvec = [x for _,x in sorted(zipped, key = lambda y : y[0], reverse = True)]
    eigenval = sorted(eigenval, reverse = True)

    return res, np.array(eigenval), np.array(eigenvec).reshape((p,p))

def get_eigen_error(eigenval, eigenvec, eigenval_hat, eigenvec_hat):
    """Get the error between eigenvalues and eigenvectors"""
    vec_err = []
    val_err = []

    val_err = np.abs(eigenval - eigenval_hat)
    vec_err = np.minimum(np.linalg.norm(eigenvec_hat - eigenvec, axis = 1
        ),np.linalg.norm(eigenvec_hat + eigenvec, axis = 1))
    return vec_err, val_err

#######################################################

def plot_cov_eigenval_error():
    """Plots the covariance and eigenvalue estimation error for increasing spike"""

    spikes = [2*i for i in range(200)]
    mat_error = []
    eigenval_err = []

    for i in range(200):

        s = [spikes[i]]
        sigma, eigenval, eigenvec = get_spike_eigen(s, p=100)

        sigma_hat = get_sigma_hat(sigma, n)
        eigenval_hat, eigenvec_hat = get_eigen(sigma_hat)

        vec_err, val_err = get_eigen_error(eigenval, eigenvec, eigenval_hat,
            eigenvec_hat)

        eigenval_err.append(val_err[0])
        mat_error.append(np.linalg.norm(sigma - sigma_hat))

    plt.figure(0)
    plt.plot(spikes,mat_error)
    plt.xlabel('Value of the spike')
    plt.ylabel('Covariance estimation error')
    plt.axis([-10,400,-1,16])

    plt.figure(1)
    plt.plot(spikes, eigenval_err)
    plt.xlabel('Value of the spike')
    plt.ylabel('Eigenvalue estimation error')
    plt.axis([-10,400,-1,16])



plot_cov_eigenval_error()

##################

def plot_eigenvec_error():
    """Plot the eigenvector error with increasing spikes (for the first eigenvalue)"""
    spikes = [2*i for i in range(200)]
    eigenvec_err = []

    for i in range(200):

        s = [spikes[i]]

        sigma, eigenval, eigenvec = get_spike_eigen(s, p=30)

        sigma_hat = get_sigma_hat(sigma, n)
        eigenval_hat, eigenvec_hat = get_eigen(sigma_hat)

        vec_err, val_err = get_eigen_error(eigenval, eigenvec, eigenval_hat,
            eigenvec_hat)

        eigenvec_err.append(vec_err[0])


    plt.plot(spikes,eigenvec_err)
    plt.xlabel('Value of the spike')
    plt.ylabel('Eigenvector estimation error')
    #plt.axis([-20,400,-0.001,0.2])



plot_eigenvec_error()

########################

def plot_scaled_eigen_error():
    """Plot the scaled eigenvalue/eigenvector errors with increasing spikes (for the
        first eigenvalue)"""
    spikes = [2*i for i in range(200)]
    eigenvec_err = []

    for i in range(200):

        s = [spikes[i]]
        scale = np.sqrt((p+s[0])*(1+s[0]))

        sigma, eigenval, eigenvec = get_spike_eigen(s, p=30)

        sigma_hat = get_sigma_hat(sigma, n)
        eigenval_hat, eigenvec_hat = get_eigen(sigma_hat)

        vec_err, val_err = get_eigen_error(eigenval, eigenvec, eigenval_hat,
            eigenvec_hat)

        eigenvec_err.append(vec_err[0]/scale)

    plt.figure(0)
    plt.plot(spikes,eigenvec_err)
    plt.xlabel('Value of the spike')
    plt.ylabel('Eigenvector estimation error')
    plt.axis([-20,400,-0.001,0.02])

    plt.figure(1)
    plt.plot(np.log(np.array(spikes[1:])),np.log(np.array(eigenvec_err[1:])))
    plt.xlabel('Log of spike')
    plt.ylabel('Log of scaled eigenvector estimation error')
    #plt.axis([-10,400,0,0.002])




plot_scaled_eigen_error()

####################

def plot_wrt_eff_rank(x):
    """Plots covariance estimation error, eigenvalue/eigenvector estimation error with
        respect to the effective rank"""
    p = 100

    eigenvec_err = []
    eigenval_err = []
    mat_err = []

    eff_rank = [(x*(i+1) + p-i)/(x+1) for i in range(1,p)]
    s = [x]

    for i in range(1,p):
        s.append(x - 1)
        sigma, eigenval, eigenvec = get_spike_eigen(s,len(s),p)

        sigma_hat = get_sigma_hat(sigma, n)
        eigenval_hat, eigenvec_hat = get_eigen(sigma_hat)

        vec_err, val_err = get_eigen_error(eigenval, eigenvec, eigenval_hat,
            eigenvec_hat)

        mat_err.append(get_norm_mat(sigma_hat - sigma))
        eigenvec_err.append(vec_err[0])
        eigenval_err.append(val_err[0])


    plt.figure(0)
    plt.xlabel('Effective rank')
    plt.ylabel('Covariance estimation error')
    plt.plot(eff_rank,mat_err)
    #plt.plot(np.log(np.array(eff_rank)), np.log(np.array(vec_err)))

    plt.figure(1)
    plt.xlabel('Effective rank')
    plt.ylabel('Eigenvector estimation error')
    plt.plot(eff_rank,eigenvec_err, color = 'g')
    #plt.plot(np.log(np.array(eff_rank)), np.log(np.array(vec_err)))

    plt.figure(2)
    plt.xlabel('Effective rank')
    plt.ylabel('Eigenvalue estimation error')
    plt.plot(eff_rank,eigenval_err, color = 'r')
    #plt.plot(np.log(np.array(eff_rank)), np.log(np.array(val_err)))



plot_wrt_eff_rank(200)

#####################

def consistency(n,p):
    """Returns an array of covariance estimation errors with increasing sample
        length"""
    sigma = get_spike_matrix([200], 1, p)
    X = np.random.multivariate_normal([0]*p, sigma, n).T

    err = []
    sample_cov = np.zeros((p,p))

    for i in range(n):

        temp = np.dot(X[:,i].reshape(p,1),X[:,i].reshape(p,1).T)
        sample_cov = sample_cov + temp
        err.append(get_norm_mat((sample_cov/(i+1)) - sigma))
    return(err)


# Plotting the estimation error for different covariance matrix sizes
### Case n = 10000

p_space = [10,100,500]

plt.figure()

for p in p_space :

    err = consistency(10000,p)
    plt.plot(err, label = 'p = ' + str(p))


plt.axis([-100,10000,0,50])
plt.xlabel('Iterations')
plt.ylabel('Covariance estimation error')
plt.legend(loc = 'best')
plt.show()

###############

def consistency_eigen(n,p):
    """Returns two arrays of eigenvalue/eigenvector estimation errors with increasing
        sample length"""

    sigma, eigenval, eigenvec = get_spike_eigen([200], 1, p)

    X = np.random.multivariate_normal([0]*p, sigma, n).T

    val_err = []
    vec_err = []
    sample_cov = np.zeros((p,p))

    for i in range(n):

        temp = np.dot(X[:,i].reshape(p,1),X[:,i].reshape(p,1).T)
        sample_cov = sample_cov + temp
        eigenval_hat, eigenvec_hat = get_eigen(sample_cov/(i+1))
        vec = eigenvec_hat[0].reshape((p,1))
        val_err.append(abs(eigenval_hat[0] - eigenval[0]))
        vec_err.append(min(np.linalg.norm(vec - eigenvec[0]),np.linalg.norm(vec +
            eigenvec[0])))


    return(val_err, vec_err)
```

```
p_space = [10,100,500]

val=[]
vec =[]
for p in p_space :
    val_err, vec_err = consistency_eigen(10000,p)
    val.append(val_err)
    vec.append(vec_err)

plt.figure(0)
for i in range(3) :
    plt.plot(val[i], label = 'p = ' + str(p_space[i]))

plt.xlabel('Iterations')
plt.ylabel('Eigenvalue estimation error')
plt.legend(loc = 'best')
plt.axis([-10,10000,0,60])

plt.figure(1)
for i in range(3):
    plt.plot(vec[i], label = 'p = ' + str(p_space[i]))

plt.xlabel('Iterations')
plt.ylabel('Eigenvector estimation error')
plt.legend(loc = 'best')
plt.axis([-100,10000,0,1])

plt.show()

#############

def plot_inconsistency(k):
    """Plots covariance estimation error with varying p"""
    n_space = np.vectorize(int)(np.linspace(100,5000,k))

    p_space = [int(n//3) for n in n_space] # Varying p !
    p_test = 50  #Fixed p

    sigma_test = get_spike_matrix([100], 1, p_test)
    X_test = np.random.multivariate_normal([0]*p_test, sigma_test, n_space[-1]+1).T
    err_test = []

    err = []

    for i in range(k):

        #print(i)
        n = n_space[i]
        p = p_space[i]

        err_test.append(get_norm_mat(np.cov(X_test[:,:n])- sigma_test))

        sigma = get_spike_matrix([100], 1, p)
        X = np.random.multivariate_normal([0]*p, sigma, n).T
        err.append(get_norm_mat(np.cov(X) - sigma))

    return(err,err_test)



err, err_test = plot_inconsistency(40)
plt.plot(n_space, err, label = 'Varying p')
plt.plot(n_space, err_test, label = 'Fixed p')


plt.xlabel('Iterations')
plt.ylabel('Covariance estimation error')
plt.legend(loc = 'best')
plt.show()
```

A Code for anomaly detection (section 4)

```
import scipy.io
import pandas as pd
import numpy as np
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.metrics import precision_recall_curve, average_precision_score, auc,
    roc_curve
import seaborn as sns
import matplotlib.pyplot as plt
import sklearn.cluster
import sklearn.model_selection
import sklearn.linear_model
mat = scipy.io.loadmat('satimage-2.mat')

## Preparing data

X = pd.DataFrame(mat['X'])
y = pd.DataFrame(mat['y'])
y.columns = ['Class']
X = preprocessing.scale(X)
y['Class'] = y['Class'].apply(int)

def get_eigen(mat):
    """Returns eigenvalues/eigenvectors in decreasing order of eigenvalues"""
    eigenval, eigenvec = np.linalg.eigh(mat)
    eigenvec = eigenvec.T
    zipped = zip(eigenval,eigenvec)
    eigenvec = [x for _,x in sorted(zipped, key = lambda y : y[0], reverse = True)]
    eigenval = sorted(eigenval, reverse = True)

    return eigenval, eigenvec

def get_nbr_components(X,threshold, plot = False):
    """Given a variance threshold, returns the number of components necessary to go
        beyond it.
       Also plots the explained variance graph if plot = True"""

    sigma = np.cov(X.T)
    eigenval, eigenvec = get_eigen(sigma)
    explained_var = np.cumsum(eigenval)/np.sum(eigenval)
    nbr_components = 0
    for i in range(len(explained_var)):
        if explained_var[i] > threshold:
            nbr_components = i
            break
    if plot:
        plt.plot([0] + list(explained_var))
        plt.xlabel('Number of components')
        plt.ylabel('Variance')

    return(nbr_components, sigma, eigenvec)

def pca(X,threshold):
    """Given a variance threshold, returns the observations in the principal
        components basis"""
    k, sigma_hat, eigenvec = get_nbr_components(X,threshold, plot = False)
    eigenvec_pca = eigenvec[:k]
    X_pca = np.dot(np.array(eigenvec_pca),X.T)

    return(X_pca)

### Explained variance plot
nbr_components, sigma, eigenvec = get_nbr_components(X,0.85, plot = True)

X_pca = pca(X,0.85)

### Displaying data in 2-dimensional space of the 2principal components
plt.figure(1)
plt.scatter(X_pca.T[:, 0], X_pca.T[:, 1], c= y['Class'], edgecolor='none', alpha=0.5)
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.title('y')
plt.axis([-13,15,-20,6])
plt.colorbar();


### Displaying data in 2-dimensional space of the 2first components
plt.figure(2)
plt.scatter(X[:, 0], X[:, 1], c= y['Class'], edgecolor='none', alpha=0.5)
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.title('y')
plt.axis([-5,5,-5,5])
plt.colorbar();

########################################## MAHALANOBIS ######################

def MahalanobisDist(X_pca, verbose=False):
    """Computes the vector of mahalanobis distances associated to each point"""
    inv_covariance_matrix = np.linalg.inv(np.cov(X_pca))
    vars_mean = X_pca.mean(axis=0)
    diff = X_pca - np.array(vars_mean)
    md = []
    for i in range(diff.shape[1]):
        md.append(np.sqrt(diff[:,i].dot(inv_covariance_matrix).dot(diff[:,i])))

    return md


def anomaly(X_pca, contamination): # rate of contamination between 0and 1
    """Implementation of the classification algorithm based on Mahalanobis
       As an input: the PCA transformed data
       Returns a vector storing the indexes of the outliers, given a previously known
           contamination rate"""
    vec_id = []
    p,n = X_pca.shape
    nbr_outlier = int(contamination*n)
    X_alpha = np.copy(X_pca)
    d = dict()
    for i in range(n):  ### Dictionary that stores the original indexes of the
        observations, as they are dropped during the algorithm
        d.update({X_pca[:,i][0] : i})

    for i in range(nbr_outlier):
        md = MahalanobisDist(X_alpha, verbose=False)
        mean_ = np.array(md).mean(axis=0)
        j = np.argmax(abs(np.array(md)-mean_)) ## Biggest outlier
        vec_id.append(d[X_alpha[:,j][0]])
        X_alpha = np.delete(X_alpha, j, 1) ## Delete the outlier from the data set

    return vec_id


def get_y_hat(vec_id):
    """Transforms output of anomaly to label vector"""
    res = [0]*X.shape[0]
    for i in vec_id:
        res[i] = 1

    return res


def get_metrics_mahalan(X,threshold, contamination):
    """Applies the algorithm and computes various metrics"""

    X_pca = pca(X,threshold)
    vec_id = anomaly(X_pca, contamination)
    y_hat = get_y_hat(vec_id)

    true_pos = 0
    true_neg = 0
    false_pos = 0
    false_neg = 0
    for i in range(len(y_hat)):
        if y['Class'][i] == 0:
            if y_hat[i] == 0:
                true_neg += 1
            else :
                false_pos += 1
        else:
            if y_hat[i] == 1:
                true_pos += 1
            else :
                false_neg += 1

    return true_pos, true_neg, false_pos, false_neg, len(y_hat)


######################### K MEANS ##########################

X_pca = pca(X,0.85) ##Previously apply PCA (2 components --> 0.85 explained variance)

X_train, X_test, y_train, y_test =
    sklearn.model_selection.train_test_split(X_pca.T,y, test_size = 0.3,
                                                                  train_size = 0.7,
                                                                  stratify = y)


def fit(X_train, y_train):
    """Fits the model in order to find the right number k of clusters"""


    y_train = y_train.values
    md = MahalanobisDist(X_train.T, verbose=False)
    mean_ = np.array(md).mean(axis=0)
    j = np.argmax(abs(np.array(md)-mean_))
    score = []
    precision = []
    recall = []
    outliers_length = []
    for k in range(2,20):
        km = sklearn.cluster.KMeans(n_clusters=k, init='k-means++').fit(X_train)
        labels = km.labels_
        outliers = [i for i in range(len(labels)) if labels[i] == labels[j]]
        outliers_length.append(len(outliers))
        y_hat = [0]*len(y_train)
        for i in outliers:
            y_hat[i] = 1

        true_pos = 0
        true_neg = 0
        false_pos = 0
        false_neg = 0
        for i in range(len(y_train)):
            if y_train[i] == 0:
                if y_hat[i] == 0:
                    true_neg += 1
                else :
                    false_pos += 1
            else:
                if y_hat[i] == 1:
                    true_pos += 1
                else :
                    false_neg += 1

        #plt.figure(k)
        #plt.scatter(X_train[:, 0], X_train[:, 1], c= labels, edgecolor='none',
            alpha=0.5)
        #plt.xlabel('component 1')
        #plt.ylabel('component 2')
        #plt.title('y_hat')
        #plt.axis([-13,15,-20,6])
        precision.append(true_pos/(true_pos + false_pos))
        recall.append(true_pos/(true_pos + false_neg))
        score.append(float(true_pos)/(true_pos + false_pos + false_neg))


    plt.figure(0)
    plt.plot(range(2,20), precision, label = 'Precision')
    plt.plot(range(2,20), recall, label = 'Recall')
    plt.legend(loc = 'best')
    plt.xlabel('k')
    plt.title('Training precision and recall')


    list_k = np.argwhere(score == np.amax(score))
    return (list_k[len(list_k)//2][0] + 2)


print(fit(X_train, y_train))

def get_metrics_kmeans():
    """Fits the model, applies the kmeans algorithm to the testing set and gets the
        metrics"""
    X_pca = pca(X,0.85)
    X_train, X_test, y_train, y_test =
        sklearn.model_selection.train_test_split(X_pca.T,y, test_size = 0.3,
                                                                      train_size = 0.7,
                                                                      stratify = y)
    k = fit(X_train, y_train)
    md = MahalanobisDist(X_test.T, verbose=False)
    mean_ = np.array(md).mean(axis=0)
    j = np.argmax(abs(np.array(md)-mean_))
    km = sklearn.cluster.KMeans(n_clusters=k, init='k-means++').fit(X_test)
    labels = km.labels_
    outliers = [i for i in range(len(labels)) if labels[i] == labels[j]]
    y_hat = [0]*len(y_test['Class'])
    for i in outliers:
        y_hat[i] = 1

    true_pos = 0
    true_neg = 0
    false_pos = 0
    false_neg = 0
    for i in range(len(y_test['Class'])):
        if y_test['Class'].iloc[i] == 0:
            if y_hat[i] == 0:
                true_neg += 1
            else :
                false_pos += 1
        else:
            if y_hat[i] == 1:
                true_pos += 1
            else :
                false_neg += 1

    precision = true_pos/(true_pos + false_pos)
    recall = true_pos/(true_pos + false_neg)

    print ('precision = ' + str(precision))
    print ('recall = ' + str(recall))
    return true_pos, true_neg, false_pos, false_neg

get_metrics_kmeans()

########################### LOGISTIC REGRESSION #################

### Perform a logistic regression after PCA

X_pca = pca(X,0.85)
X_train, X_test, y_train, y_test =
    sklearn.model_selection.train_test_split(X_pca.T,y, test_size = 0.2,
                                                                 train_size = 0.8,
                                                                 stratify = y)
### Apply the algorithm (log reg)

log_reg = sklearn.linear_model.LogisticRegression()
log_reg.fit(X_train, y_train)
```

```
y_hat = log_reg.predict(X_test)


### Compute metrics

true_pos = 0
true_neg = 0
false_pos = 0
false_neg = 0
for i in range(len(np.array(y_test))):
    if np.array(y_test)[i] == 0:
        if y_hat[i] == 0:
            true_neg += 1
        else :
            false_pos += 1
    else:
        if y_hat[i] == 1:
            true_pos += 1
        else :
            false_neg += 1

precision = true_pos/(true_pos + false_pos)
recall = true_pos/(true_pos + false_neg)


print ('precision = ' + str(precision))
print ('recall = ' + str(recall))

### Get the coefficients of the regression

coeff = log_reg.coef_[0]
intercept = log_reg.intercept_[0]



plt.figure(0)
plt.scatter(X_test[:, 0], X_test[:, 1], c = y_hat, edgecolor='none', alpha=0.5)
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.title('y_hat')
plt.axis([-13,15,-20,6])

### Plot the decision boundary

x_absc = np.linspace(-13,15,1000)
plt.plot( x_absc, [(-coeff[0]/coeff[1])*x - intercept/coeff[1] for x in x_absc])
plt.figure(1)
plt.scatter(X_test[:, 0], X_test[:, 1], c= y_test['Class'], edgecolor='none',
    alpha=0.5)
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.title('y')
plt.axis([-13,15,-20,6])
```

References

[1] R. Vershynin
High-Dimensional Probability, An Introduction with Applications in Data Science.

[2] V. Koltchinskii and K. Lounici
Normal approximation and concentration of spectral projectors of sample covariance.

[3] T.W Anderson
An introduction to Multivariate Statistical Analysis.

[4] O'Rourke, S., Vu, V. and Wang, K.
Random perturbation of low rank matrices: Improving classical bounds.

[5] F. Bunea, L. Xiao
On the sample covariance matrix estimator of reduced effective rank population matrices with applications to fpca.

[6] Tsybakov, A. B.
Introduction to Nonparametric Estimation.

[7] JOHNSTONE, I. M. and LU, A. Y.
Sparse principal components analysis.