Projet Fouille de Données

Michael Hajage, Lucas Rodrigues Pereira et Mathieu Pont

Mai 2019

Abstract

1 Introduction

Dans le cadre de la première année du master informatique de l'université de Paris (anciennement Paris Descartes) il nous a été demandé dans le cours "Data Science" de réaliser un projet utilisant le jeu de données Fashion-MNIST.

Nous devons analyser ce jeu de données à l'aide des différents algorithmes suivants:

- Principal Component Analysis (PCA)
- t-distributed Stochastic Neighbor Embedding (t-SNE)
- Autoencoder
- K-Means
- Support Vector Machine (SVM)
- Linear Discriminant Analysis (LDA)

Le code du projet est disponible sur github[1].

2 Overview

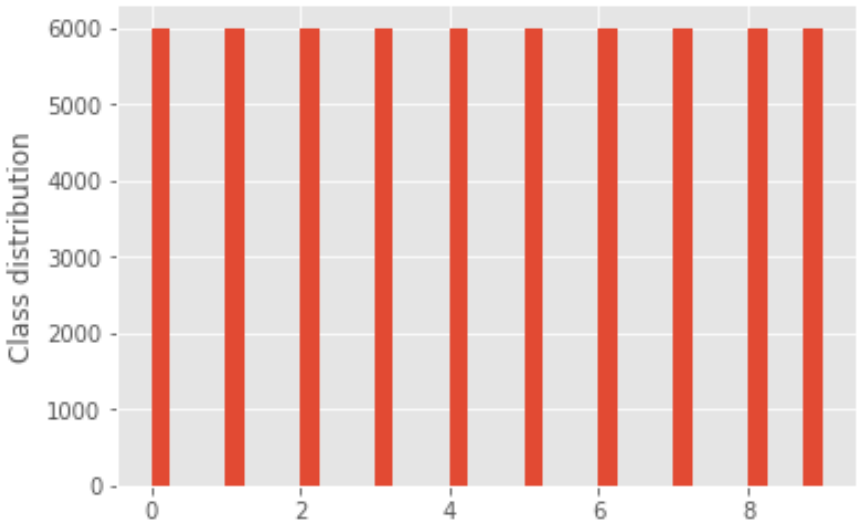


Figure 1: Distribution des observations entre les classes.

[1] https://github.com/MatPont/Fashion-MNIST

Nous voyons sur la figure 1 que la répartition des observations entre les classes est uniforme.

Ce jeu de données contient des images de différents vêtements ou accessoires. Il y a 10 classes, avec des pantalons, des chaussures, des sacs, etc.

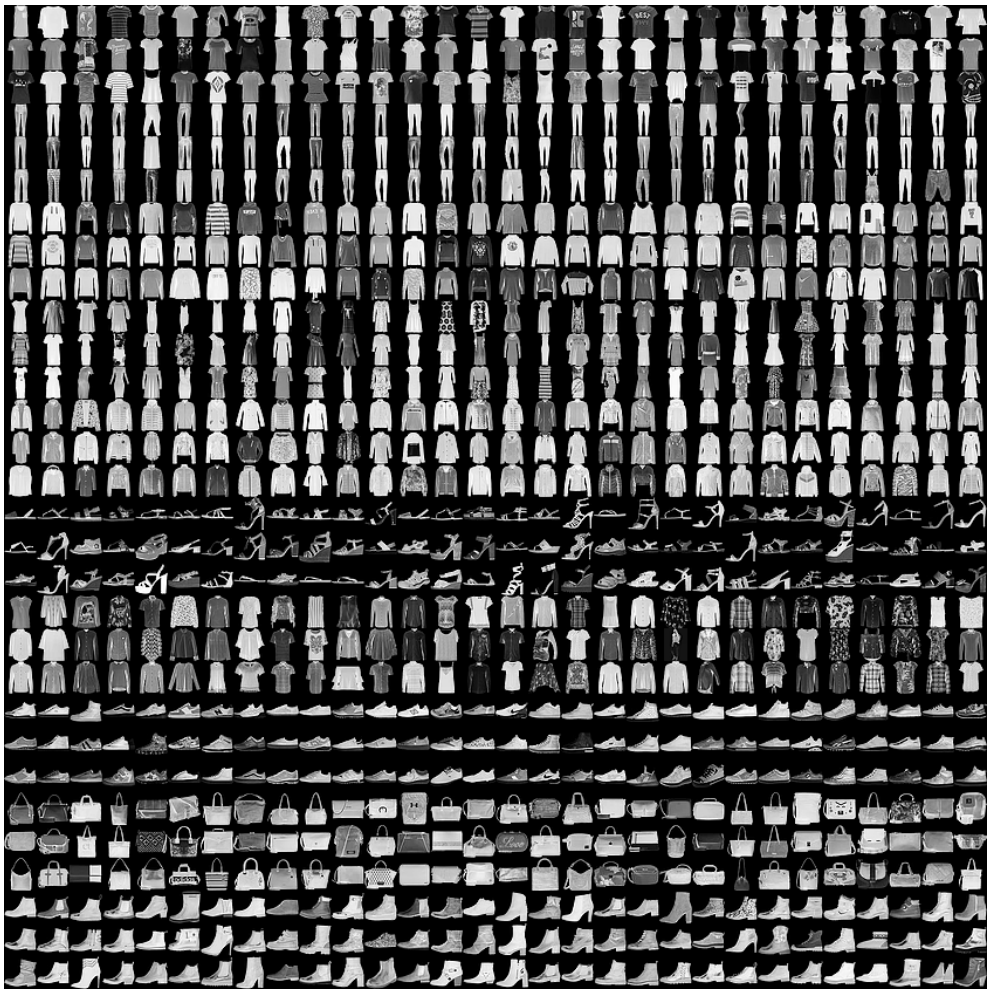


Figure 2: Exemples d'images de Fashion MNIST.

3 Principal Component Analysis (PCA)

Nous avons commencé par l'analyse en composantes principales (en utilisant R et le package *FactoMineR*) dont les deux premières composantes n'expliquent que 46.8% de la variance totale des données. La troisième composante (non représentée sur les différentes figures) n'apporte que 6% d'inertie.

Etant donné que toutes les variables sont dans un intervalle identique nous avons décidé de faire une ACP non normée, avec une ACP normée nous n'avons que 36.49% d'inertie pour les deux premières composantes.

Le plan factoriel des variables (figure 3) est assez complexe à analyser étant donné le nombre de dimensions de départ. Ce que l'on peut tout de même voir c'est qu'aucune des dimensions d'origine ne sont corrélées négativement avec les deux principales composantes du PCA à la fois.

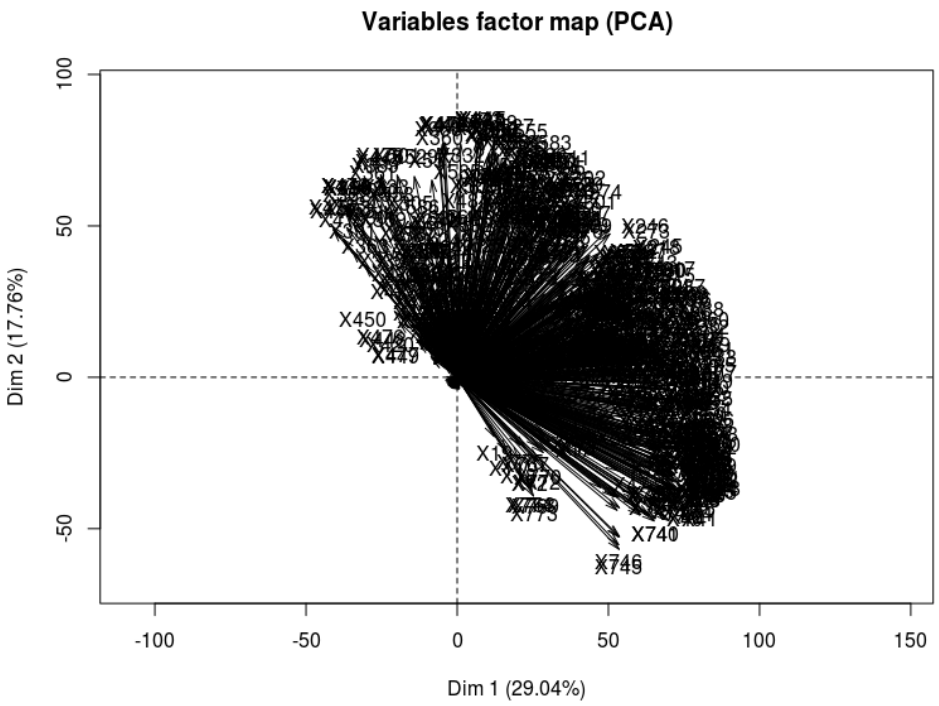


Figure 3: Plan factoriel des variables avec PCA.

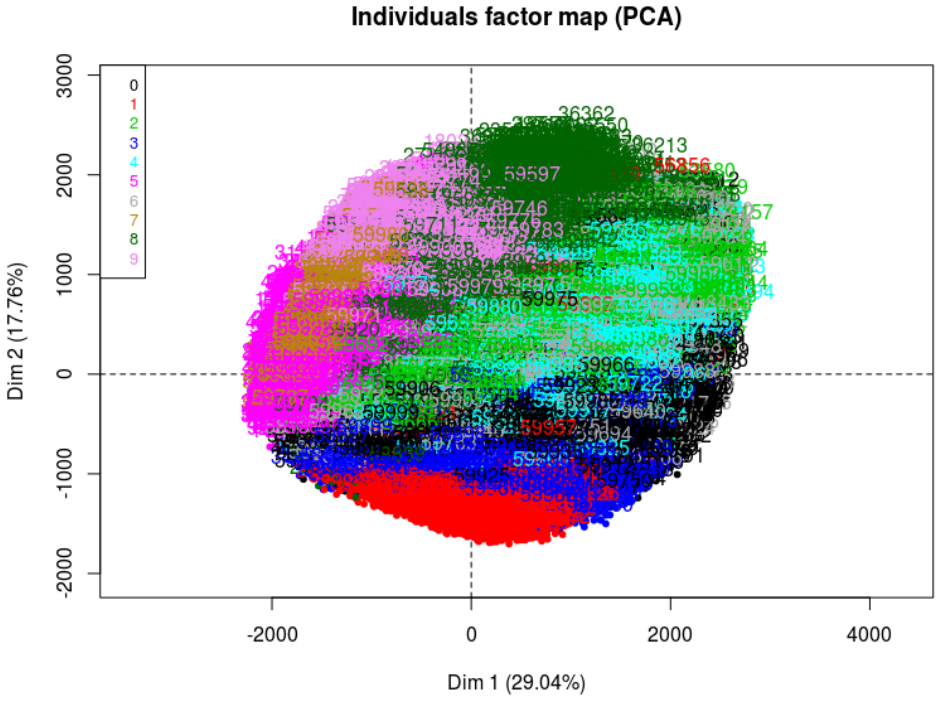
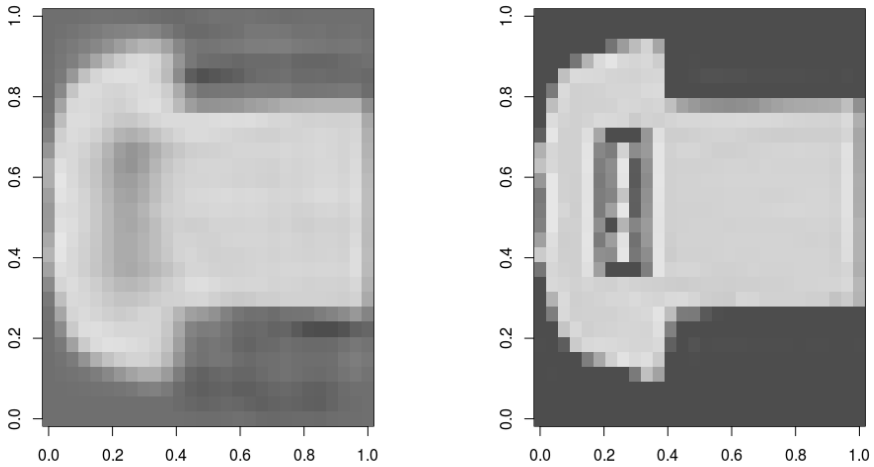


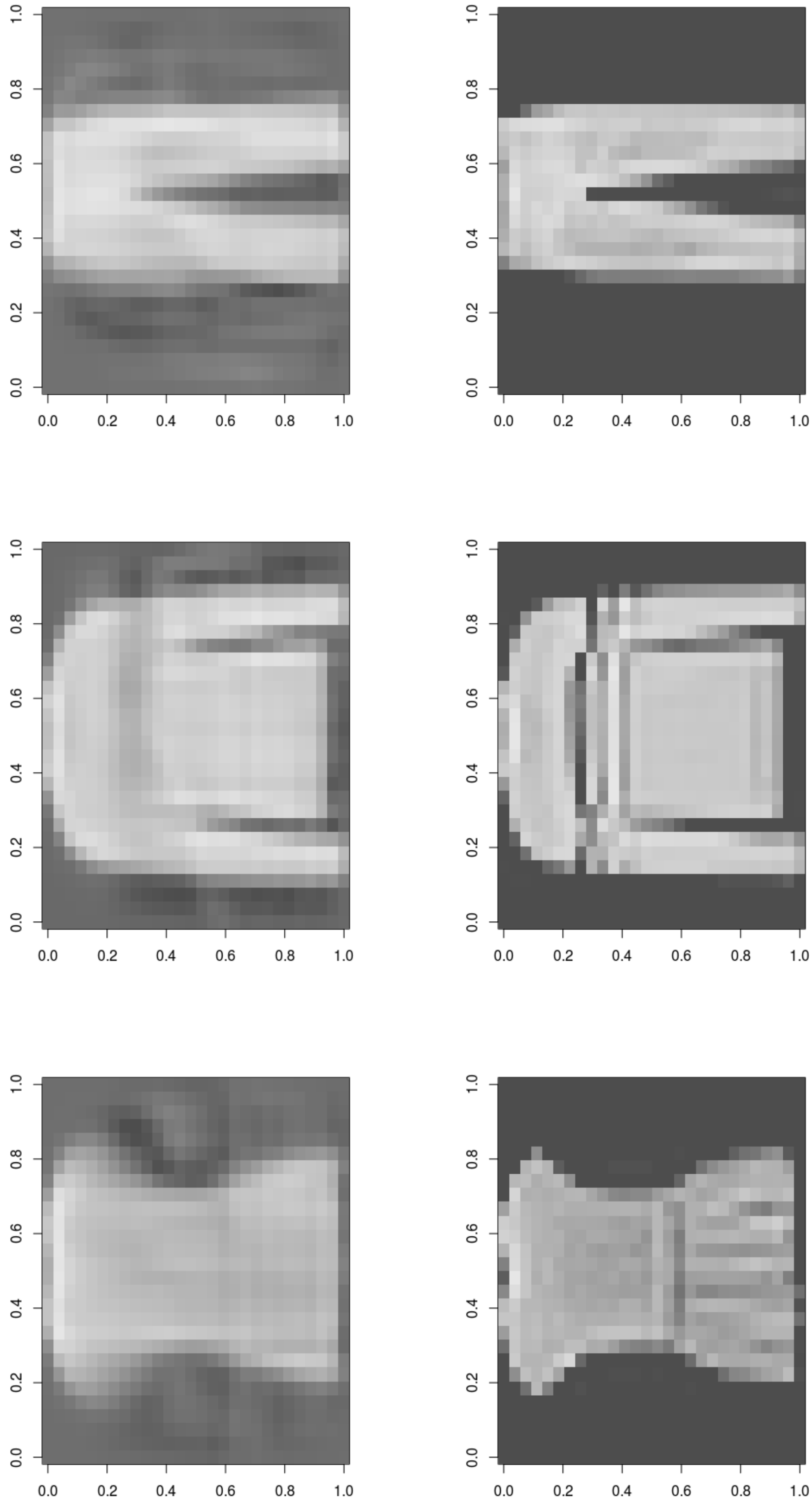
Figure 4: Plan factoriel des individus avec PCA.

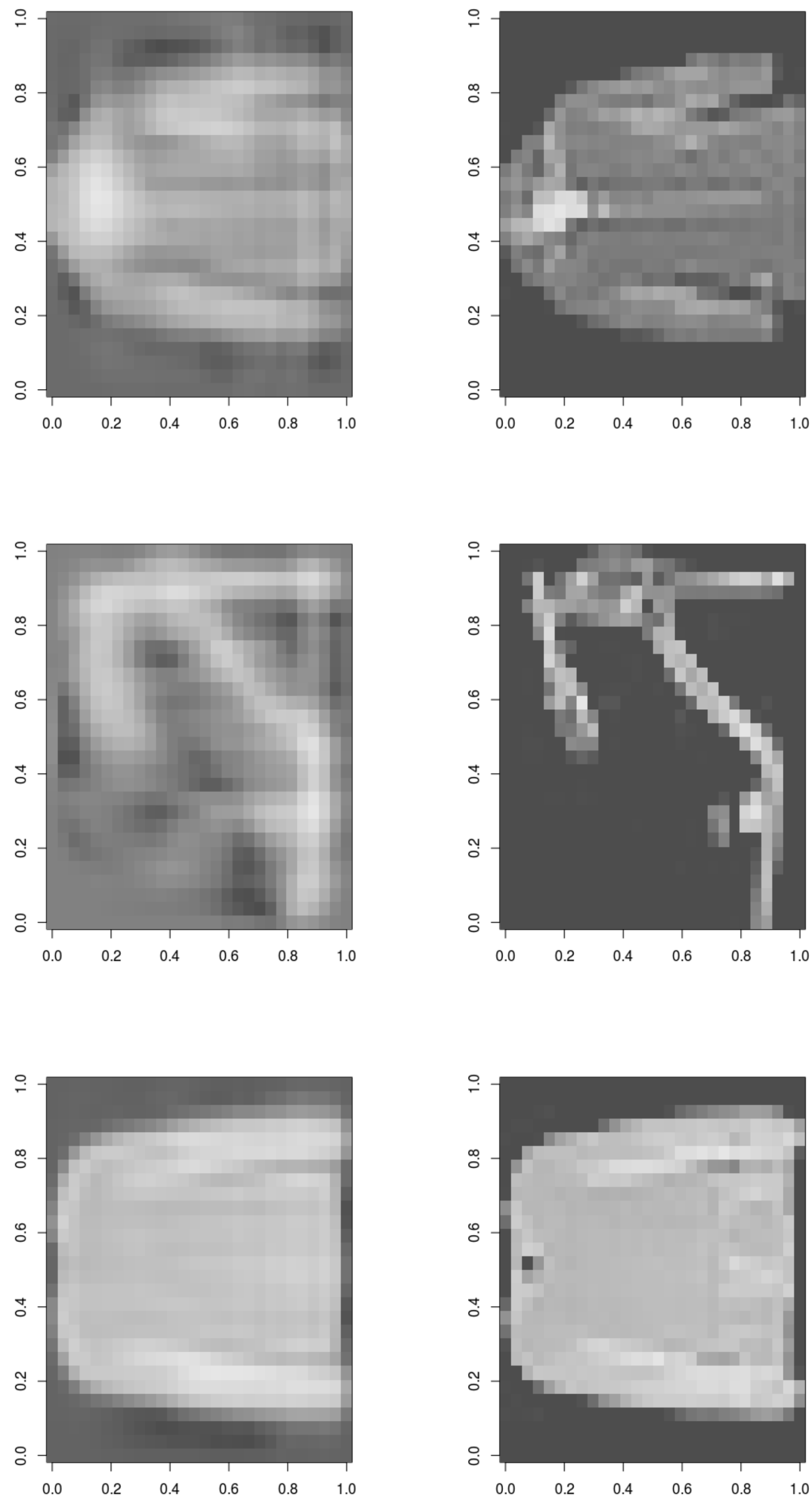
Figure 5: Ellipse des classes avec PCA.

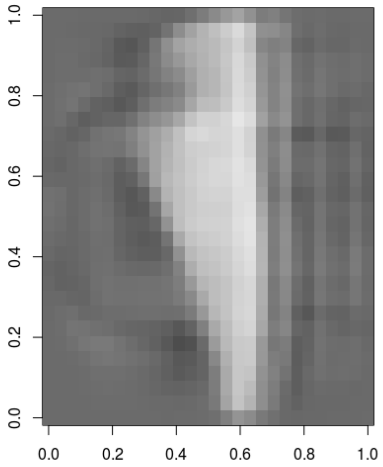
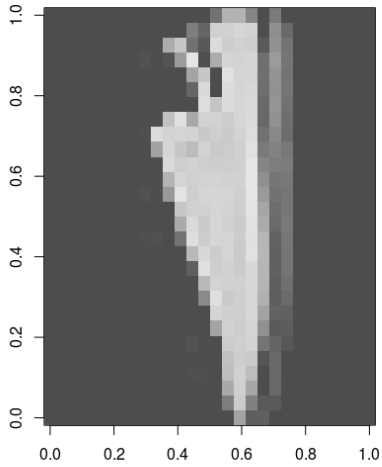
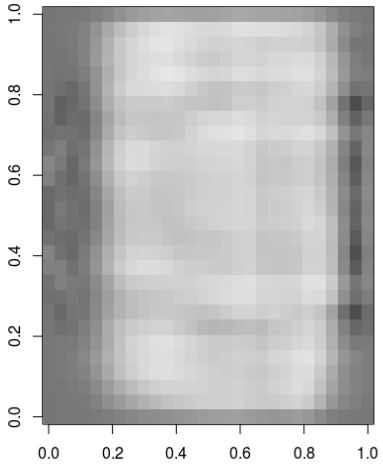
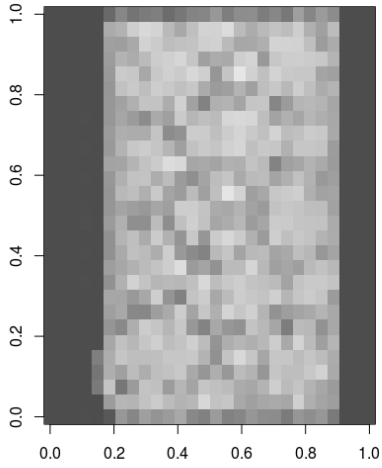
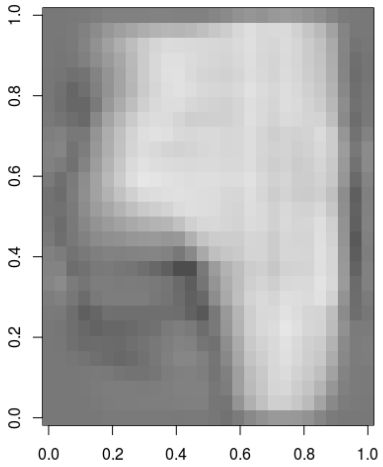
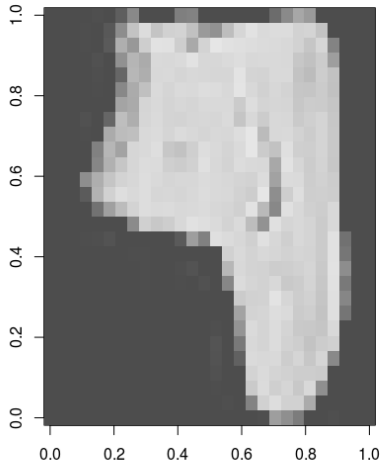
On remarque sur la figure 5 que les clusters se confondent et semblent n'être pas sufisamment séparés sur les deux composantes principales du PCA. Cela fait sens avec le pourcentage de la variance expliquée par ces deux composantes (46.8%).

Sur les figures suivantes nous voyons à gauche l'image reconstruite et à droite l'image originale pour chaque classe. Nous avons choisi les 50 premières composantes qui expliquent 86.27% de la variance totale. Le choix du nombre de 50 s'est fait en affichant la variance totale expliquée pour chaque composante, c'est environ à ce chiffre que la variance augmente très lentement.




0.0
0.2
0.4
0.6
0.8
1.0


0.0 0.2 0.4 0.6 0.8 1.0

4 t-distributed Stochastic Neighbor Embedding (t-SNE)

Après avoir étudié la documentation de la fonction *Rtsne* de la librairie *Rtsne* pour R nous observons 2 principaux hyper-paramètres à ajuster: *perplexity* et *learning rate*.

La méthode nous donne en sortie la valeur de la divergence de Kullback-Leibler (qui sera noté D_{KL} dans la suite) qui est une mesure de dissimilarité entre deux distributions, nous devons donc chercher les hyper-paramètres minimisant cette valeur le plus possible.

Selon [Maaten et al. 2009] le paramètre *perplexity* doit varier entre 5 et 50.

Dans un premier temps nous avons donc fait tourner l'algorithme pour des valeurs de 5 à 50 en augmentant de 5 à chaque fois et en gardant le *learning rate* par défaut, à savoir 200. La valeur de la divergence de Kullback-Lieber ne faisait que diminuer au fur et à mesure que la *perplexity* augmentait. Nous avons refait cette expérience mais cette fois pour des valeurs allant jusqu'à 100 et le même résultat s'est produit. Nous sommes donc allé jusqu'à une valeur de 350 (avec un écart de 20 entre chaque) et de même, D_{KL} ne faisait que diminuer. Les résultats sont sur la figure 6.

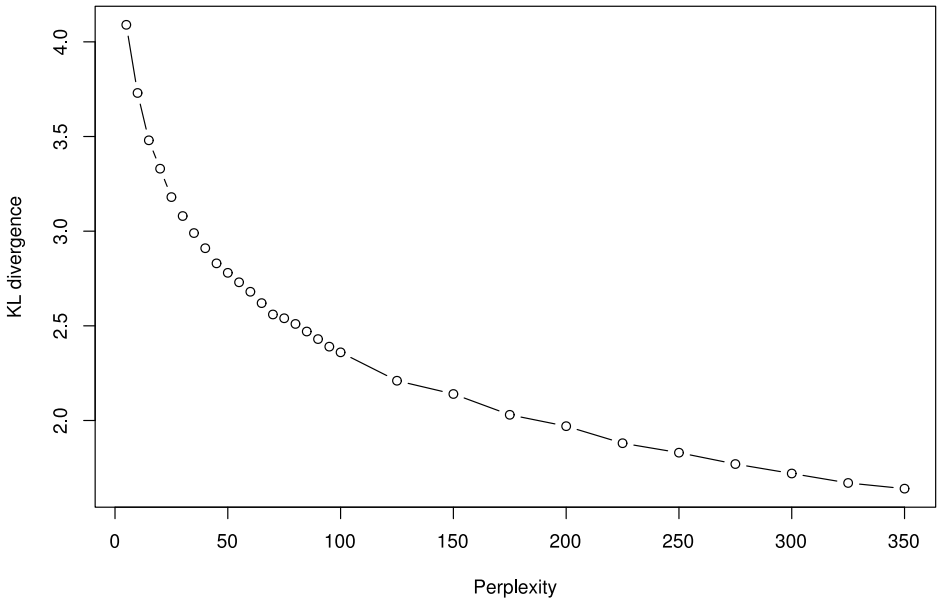


Figure 6: Valeur de la divergence de KL en fonction de l'hyperparamètre *perplexity*, pour un *learning rate* = 200.

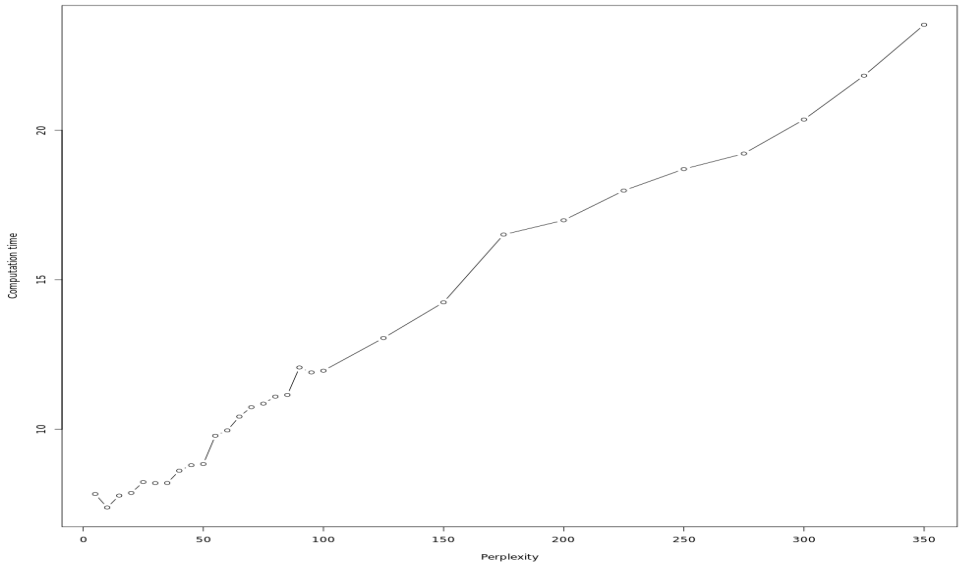


Figure 7: Temps de calcul en minutes en fonction de l'hyperparamètre *perplexity*, pour une valeur de *learning rate* = 200.

Remarquons que le temps de calcul (figure 7) augmente de façon presque linéaire en fonction du paramètre *perplexity*.

Cela fait sens car selon la documentation de *Rtsne* la descente de gradient se fait en utilisant l'implantation de Barnes-Hut [Maaten et al., 2013] afin d'éviter la complexité computationnelle pour les gros jeux de données (qui est $O(n^2)$ contre $O(nlog(n))$ pour Barnes-Hut).

Cette méthode utilise pour calculer les similarités entre les échantillons une méthode cherchant les $3 \times perplexity$ voisins de l'échantillon. C'est d'ailleurs pour cela que la documentation dit que l'inégalité suivante doit être respectée: $3 \times perplexity < nrow(X) - 1$ où $\mathbf{X}$ est le jeu de données et *nrow* une fonction donnant le nombre de lignes (d'échantillons).

En effet, il ne faut pas que le $3 \times perplexity$ (nombre de voisins à analyser) soit plus grand que le nombre d'échantillons dans le jeu de données.

On peut donc s'attendre qu'en augmentant l'hyperparamètre *perplexity* le temps de calcul augmente (plus de voisins à prendre en compte) mais que la "précision" de l'algorithme augmente (prend en compte plus de données pour chaque échantillon).

Nous avons ensuite fait varier la valeur de *learning rate* pour voir son influence sur D_{KL}.

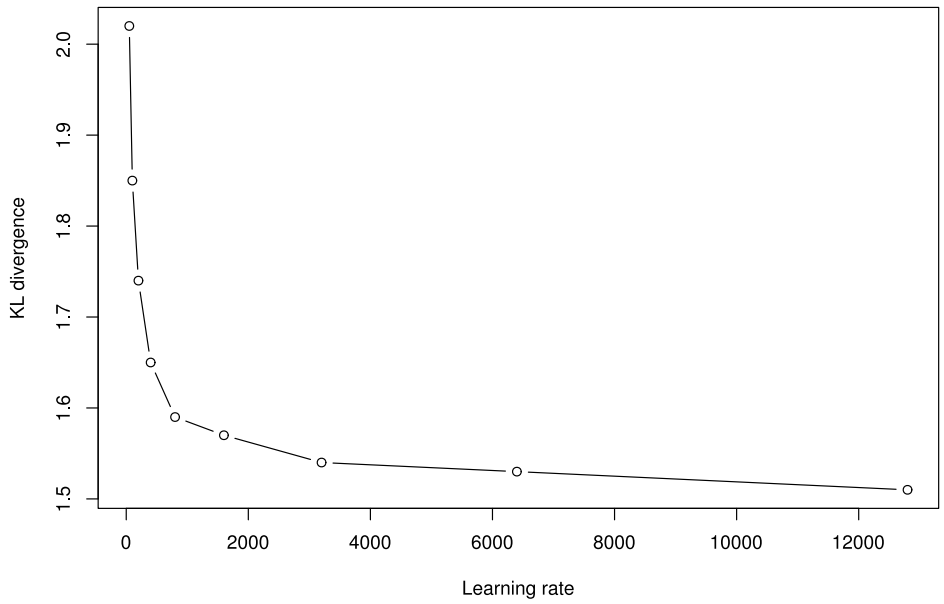


Figure 8: Valeur de la divergence de KL en fonction de l'hyperparamètre *learning rate*, pour une valeur de *perplexity* = 300.

On voit sur la figure 8 qu'au bout d'une certaine valeur augmenter le learning rate n'augmente plus significativement le résultat. Ici on peut voir un coude à la valeur $learning\_rate = 800$.

Ici nous avons dans un premier temps fait varier la valeur de *perplexity* (pour un *learning rate* fixe) puis celle de *learning rate* (pour un *perplexity* fixe). Cependant il aurait été intéressant de faire un *grid search* qui consisterait à faire varier les deux paramètres en même temps.

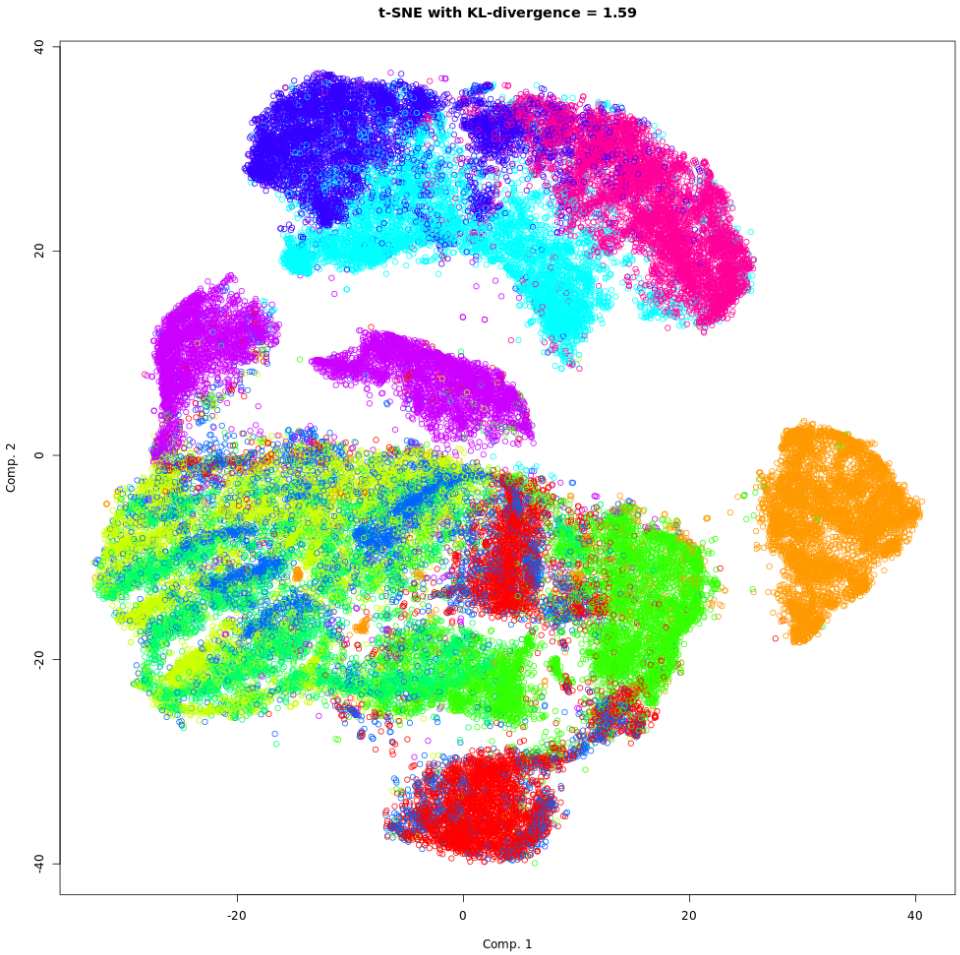


Figure 9: Clusters formés par t-SNE pour $perplexity = 300$ et $learning\_rate = 800$

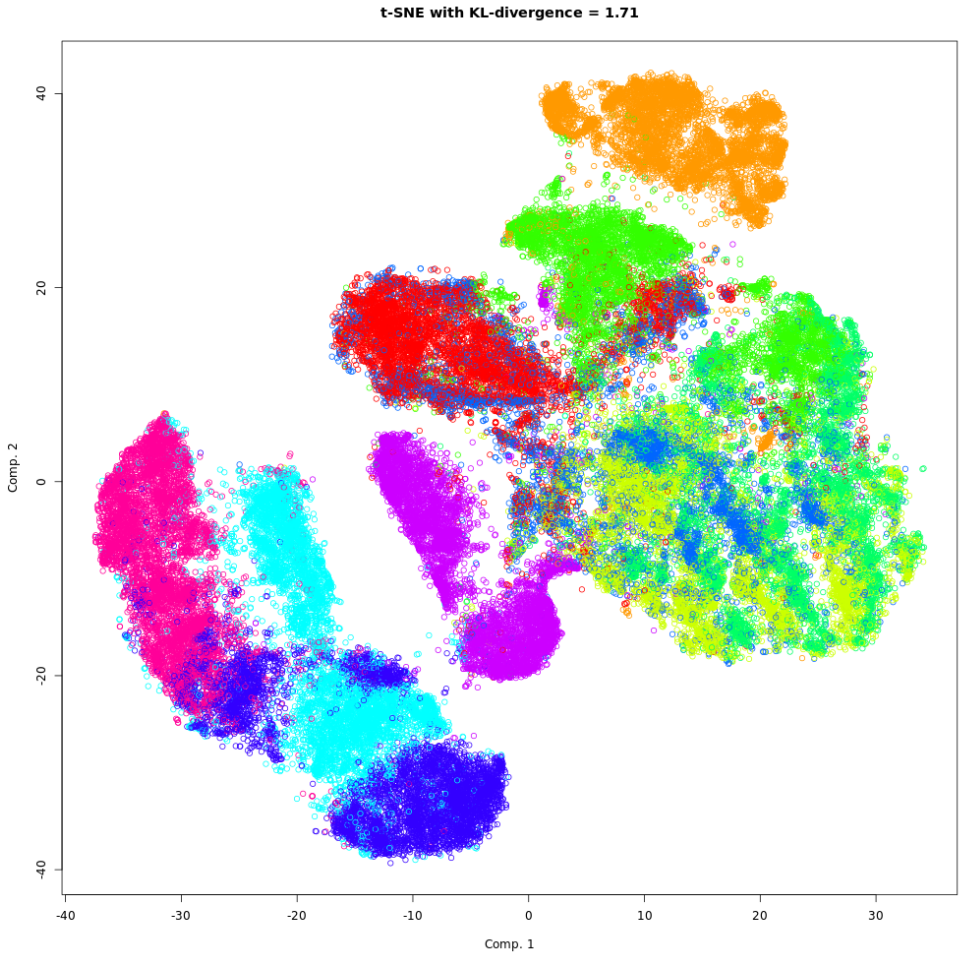


Figure 10: Clusters formés par t-SNE avec les images encodées par l'Autoencoder pour $perplexity = 300$ et $learning\_rate = 800$

Pour les mêmes hyper-paramètres ($perplexity = 300$ et $learning\_rate = 800$) nous avons $D_{KL} = 1.71$ à partir des images encodées par l'Autoencoder et $D_{KL} = 1.59$ pour les images originales. Un meilleur résultat pour les images originales pouvait être attendu étant donné qu'elles contiennent plus d'informations (784 dimensions contre 32 pour les images encodées). Cependant l'écart entre les deux D_{KL} est très faible ce qui montre que l'encodage de l'Autoencoder semble bon, les informations importantes ont été gardées et compressées dans les 32 dimensions.

5 Autoencoder

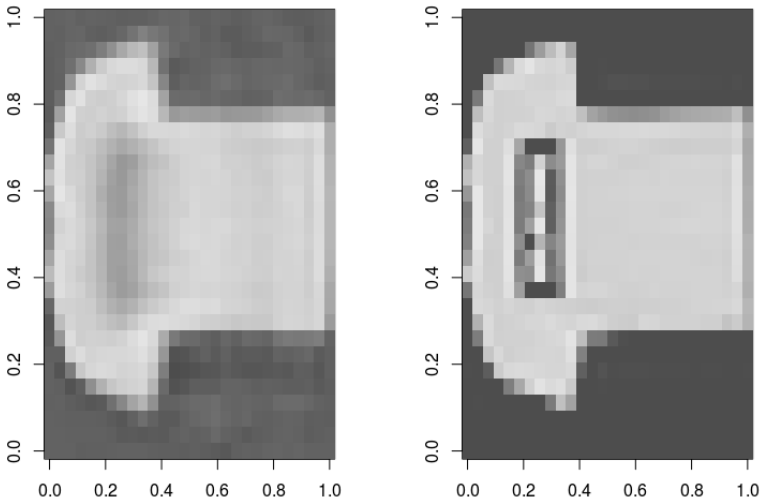
On utilise un auto-encodeur pour réduire la dimension de nos images et ainsi permettre à nos algorithmes d'être plus rapides. Nous avons testé plusieurs modèles et nous avons eu les résultats suivants:

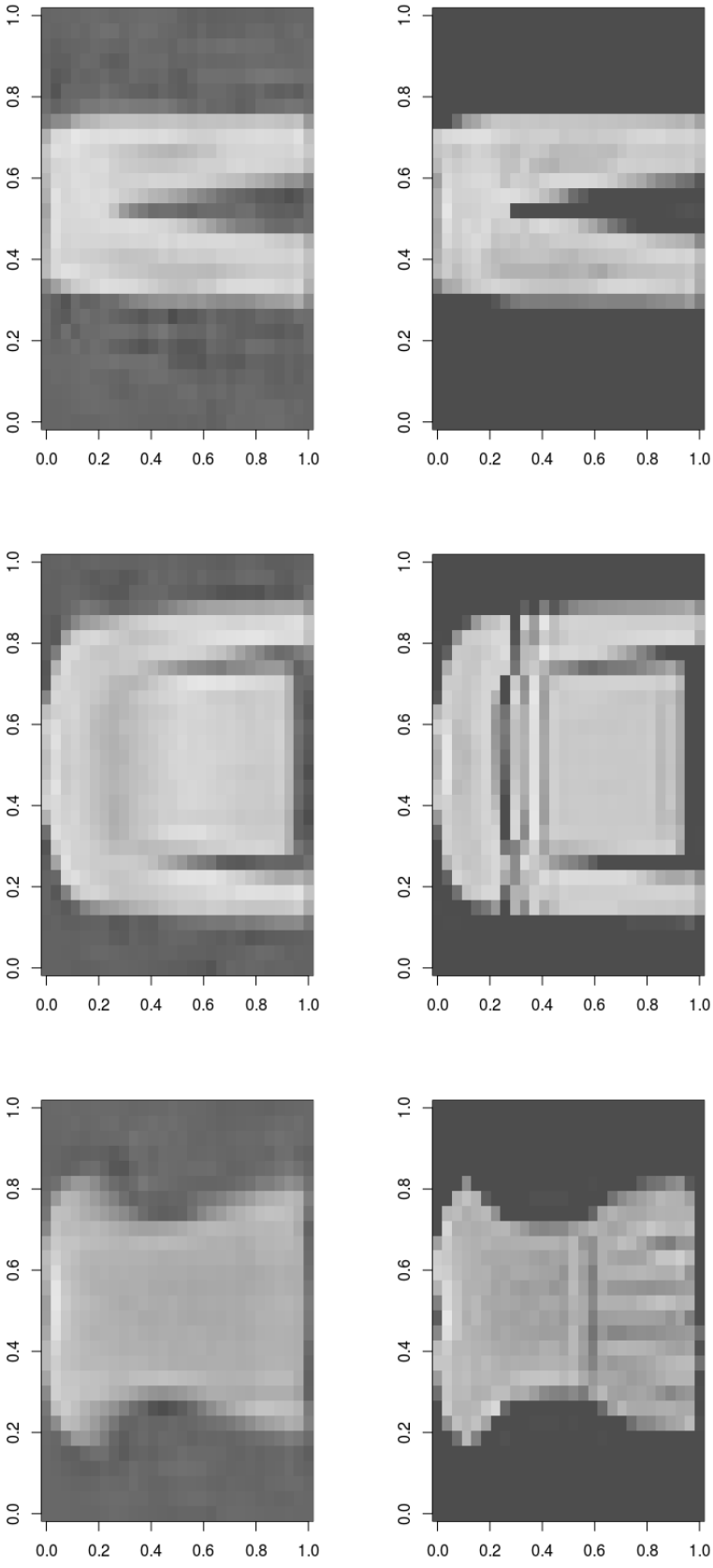
Couches	epochs	train loss	val loss
784-256-128- 64 -128-256-784	30	903.60	899.10
784-256-128-64- 32 -64-128-256-784	30	1229.18	1226.16
784-256- 128 -256-784	30	501.44	508.49
784-256- 128 -256-784	100	493.56	502.74
784-256-128- 32 -128-256-784	100	676.56	696.71
784-256- 64 -256-784	100	550.91	562.83
784-256- 32 -256-784	200	582.22	597.44

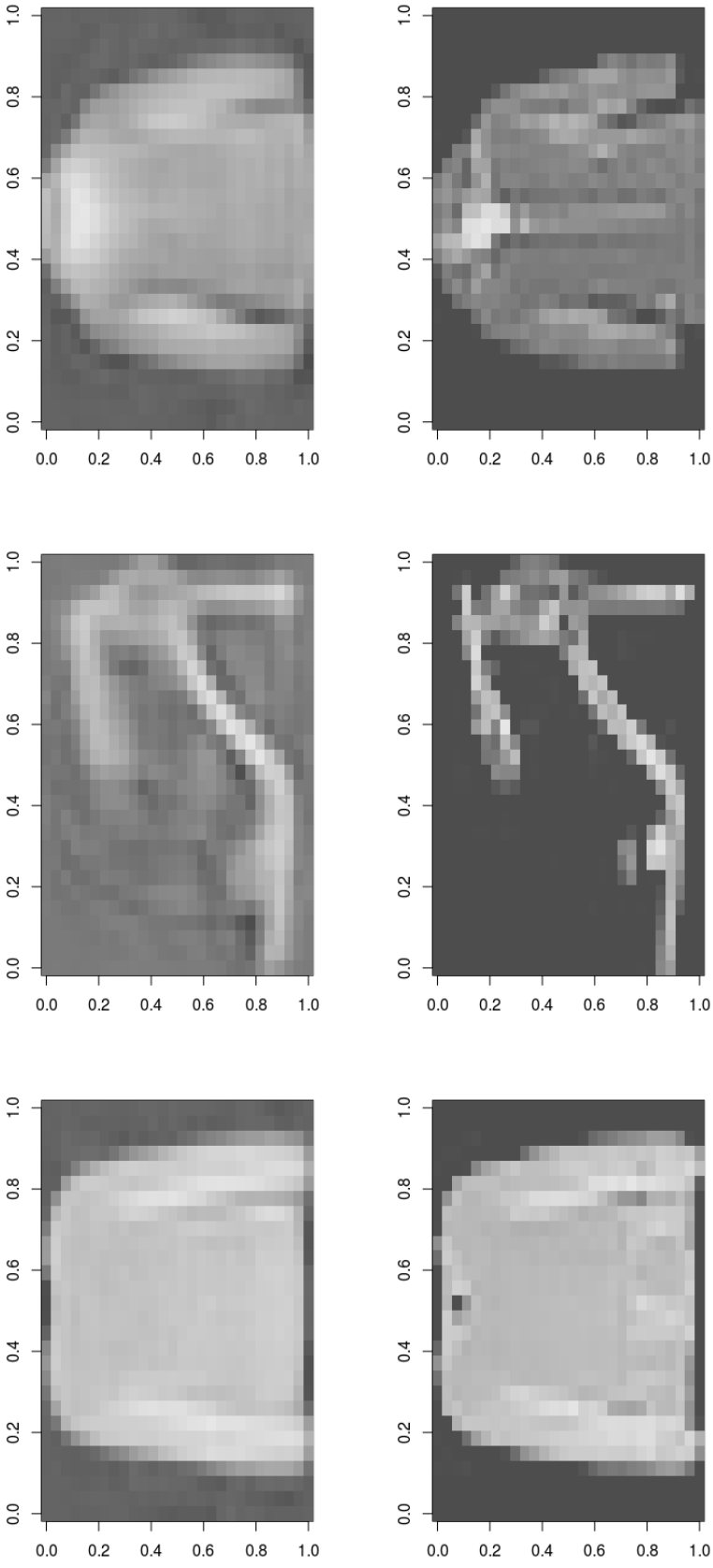
Table 1: Différents hyper-paramètres pour l'Autoencoder.

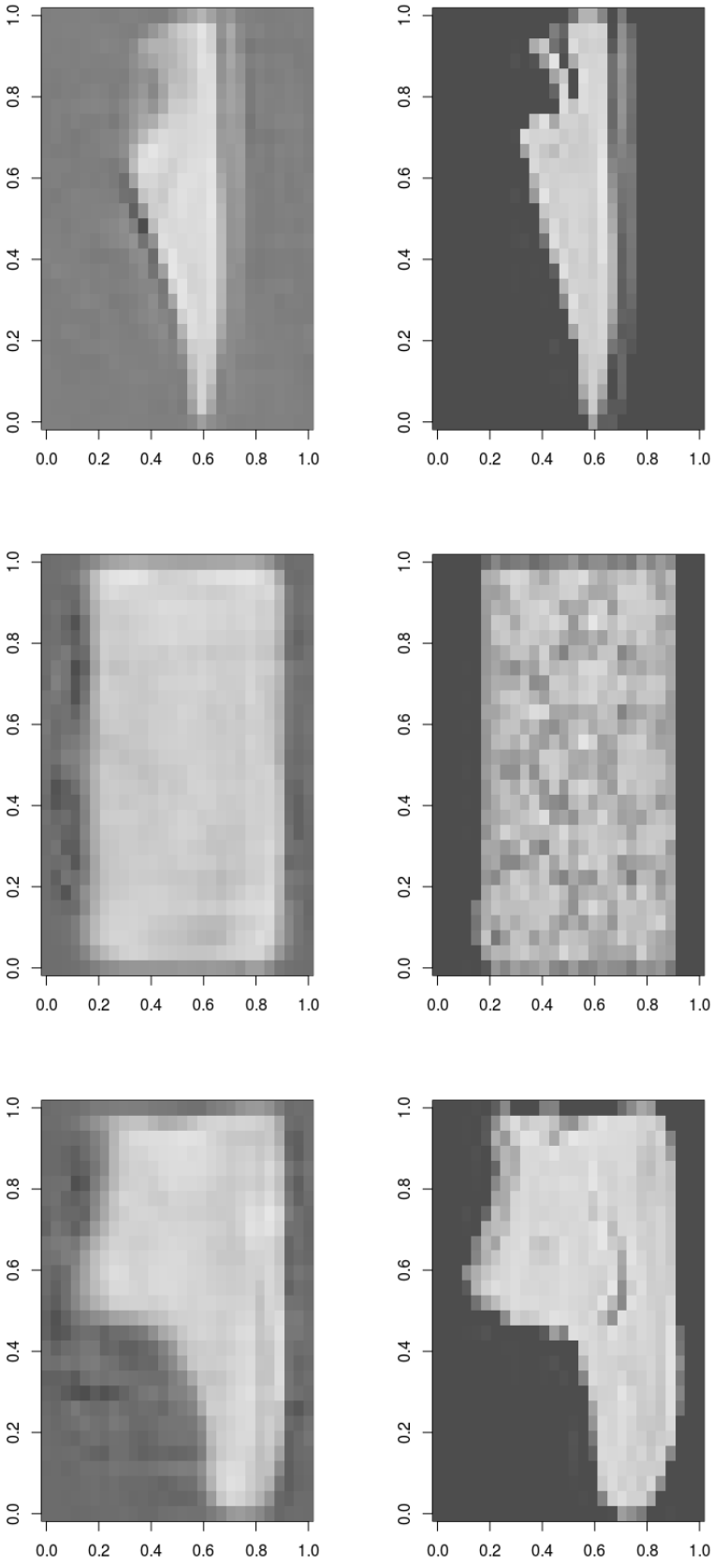
Nous cherchons à prendre un modèle qui minimise l'erreur entre l'image initiale et l'image décodée, tout en limitant le surapprentissage. Nous voulons également un modèle qui compresse au maximum l'image. Nous avons choisi de prendre le dernier modèle à 5 couches: 786 - 256 - 32 - 256 - 784 qui est un juste milieu entre nos deux contraintes (minimisation de la loss, maximisation de l'encodage).

Nous allons vous présenter les différences entre les images initiales et les images encodées.




0.0
0.2
0.4
0.6
0.8
1.0


0.0
0.2
0.4
0.6
0.8
1.0


0.0
0.2
0.4
0.6
0.8
1.0

6 K-Means

Nous avons effectué un KMeans avec 10 clusters sur les images encodées par l'auto encodeur. Nous avons utilisé plusieurs métriques pour mesurer les résultats. Toutes les mesures utilisées ont des résultats entre 0 et 1 (sauf Adjusted Rand Index).

- Normalized Mutual Information (NMI) : C'est une mesure de similarité entre deux distributions.
- Ajusted Rand Index (ARI): l'ARI calcule des similarités entre deux clusterings en considerant toutes les paires du jeu de données et en comptant les paires qui sont assignées au même ou à un différent cluster dans le vrai clustering.
- Completeness score: Les résultats d'un clustering sont dit complets si toutes données d'une même classe sont dans le même cluster.
- Homogeneity score: Un clustering est dit homogène si tous les clusters contiennent uniquement des données qui sont membre d'une seule classe.
- The V-mesure: La V-mesure est la moyenne harmonique entre l'homogénéité et la complétude:

$$v = \frac{homogeneity * completeness}{homogeneity + completeness} \tag{1}$$

NMI	0.58
ARI	0.41
Completeness Score	0.58
Homogeneity Score	0.62
V Measure Score	0.60

Table 2: Valeur des différentes métriques pour le clustering de K-Means.

Le clustering est presque autant homogène (0.62) que complet (0.58). La V-Mesure, prenant la moyenne harmonique de la complétude et de l'homogénéité a un score cohérent (0.60). Malgré le faible score de ARI (0.41), on peut déduire de ces résultats qu'environ 60% des données ont été bien classées.

7 Support Vector Machine (SVM)

Le Support Vector Machine a été dévéloppé pour classifier 2 classes à la fois. Comme le Fashion-MNIST a 10 classes, nous avons décidé d'utiliser le *One-vs-the-rest* (OvR), une stratégie multi-classes/multi-labels. Ainsi, le SVM est lancé 1 fois par chaque classe, pour la differencier de tous les autres.

Pour decider les hyper-paramètres, nous nous sommes basé sur le *benchmark* de Zalando[2]. D'après le *benchmark*, le meilleur resultát est obtenu avec: $c = 10$ et en utilisant le *kernel* polynomial. Nous avons utilisé *Sklearn*:

```
from sklearn.svm import SVC
from sklearn.multiclass import OneVsRestClassifier
clf = OneVsRestClassifier(SVC(gamma=0.1, kernel='poly', C=10))
clf.fit(X_train, y_train)
```

[2]http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/

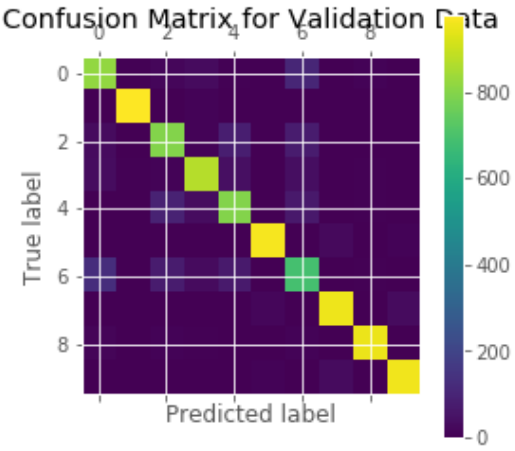


Figure 11: Matrice de confusion pour SVM.

	1	2	3	4	5	6	7	8	9	10
1	822	6	18	28	7	2	107	0	10	0
2	3	981	2	10	2	0	2	0	0	0
3	30	1	799	10	84	0	75	0	1	0
4	31	7	11	872	40	0	35	0	4	0
5	2	1	93	34	799	0	67	0	4	0
6	0	0	0	1	0	967	0	24	0	8
7	130	0	75	27	69	0	693	0	6	0
8	0	0	0	0	0	18	0	955	0	27
9	13	0	9	5	3	3	11	3	953	0
10	1	1	0	0	0	8	1	28	0	961

Table 3: Matrice de confusion pour SVM. (Validation data)

Nous avons réussi une précision dans le jeu de données de test de 88.02%.

Après, on a essayé de refaire le SVM en utilisant les données encodées avec l'autoencodeur. La dimension du jeu de données a été réduit de 728 à 32. Le temps d'éxecution a été optimisé et les résultats obtenus identiques.

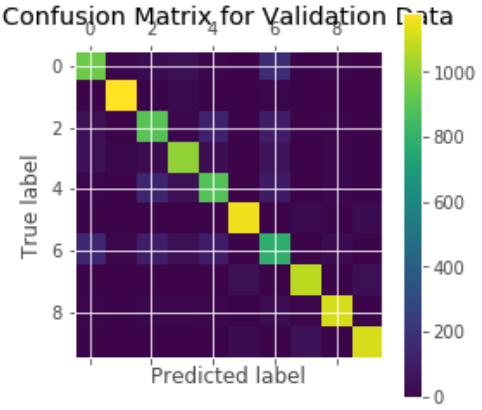


Figure 12: Matrice de confusion pour SVM. (Données encodés)

	1	2	3	4	5	6	7	8	9	10
1	942	6	25	38	8	0	174	1	8	0
2	3	1185	3	16	2	0	8	0	1	1
3	39	4	902	11	134	0	106	0	9	0
4	46	12	15	1005	36	0	60	0	10	0
5	7	3	142	48	892	0	97	0	12	1
6	1	1	0	1	0	1160	2	18	4	24
7	151	6	108	44	99	0	787	0	23	0
8	0	0	0	0	1	38	0	1074	2	44
9	9	2	13	10	8	3	28	6	1117	1
10	0	0	0	4	1	23	0	52	1	1122

Table 4: Matrice de confusion pour SVM. (Données encodés)

Avec les données encodées nous avons une précision de 84.33%. (Note: en guise de jeu de test nous avons ici utilisé 20% du jeu d'entrainement encodé, il aurait été préférable d'encoder l'ensemble de test original)

Encore une fois, la faible différence entre la précision sur le jeu de données original et celui encodé nous montre que l'encodage de l'Autoencoder semble être bon.

8 Linear Discriminant Analysis (LDA)

Le Linear Discriminant Analysis ne demande pas d'hyper-paramètres. Nous avons aussi utilisé la librairie Python *Sklearn*:

```
from sklearn.discriminant_analysis
    import LinearDiscriminantAnalysis as LDA
lda = LDA(n_components=10)
X_train = lda.fit_transform(X_train, y_train)
```

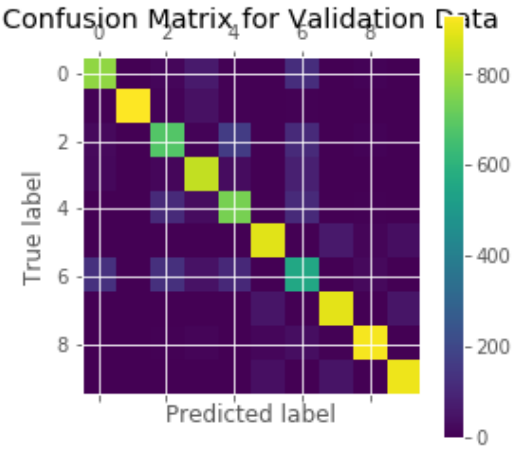


Figure 13: Matrice de confusion pour SVM.

	1	2	3	4	5	6	7	8	9	10
1	777	0	15	66	7	3	123	0	9	0
2	5	933	9	43	4	0	4	0	2	0
3	24	0	682	10	165	0	110	0	9	0
4	19	1	17	846	32	2	81	0	2	0
5	0	2	111	30	743	0	110	0	4	0
6	0	0	0	1	0	890	0	64	12	33
7	134	0	131	46	106	1	558	0	24	0
8	0	0	0	0	0	54	0	893	1	52
9	2	0	5	13	3	15	36	5	920	1
10	0	0	0	1	0	39	2	49	0	909

Table 5: Matrice de confusion pour LDA.

Nous avons une précision dans le jeu de données de test de 81.51%.

References

Maaten et al., 2009 - L. van der Maaten et G. Hinton. Visualizing Data using t-SNE. 2009.

Maaten et al., 2013 - L. van der Maaten. Barnes-Hut-SNE. 2013.