RAPPORTS DES TRAVAUX SUR MACHINE

algorithmes de machine learning

(...)
////////////////////////
////////////////////////
////////////////////////
////////////////////////
////////////////////////

Ce rapport concerne les quatre premiers TME commencés en cours et continués en confinement. Le code correspondant est disponible sous forme de Notebooks *et* de scripts .py élagués dans l'archive jointe. Nous nous tenons à votre disposition pour tout complément utile d'ici le 17 juin.

////////////////////////
////////////////////////
////////////////////////
////////////////////////
////////////////////////

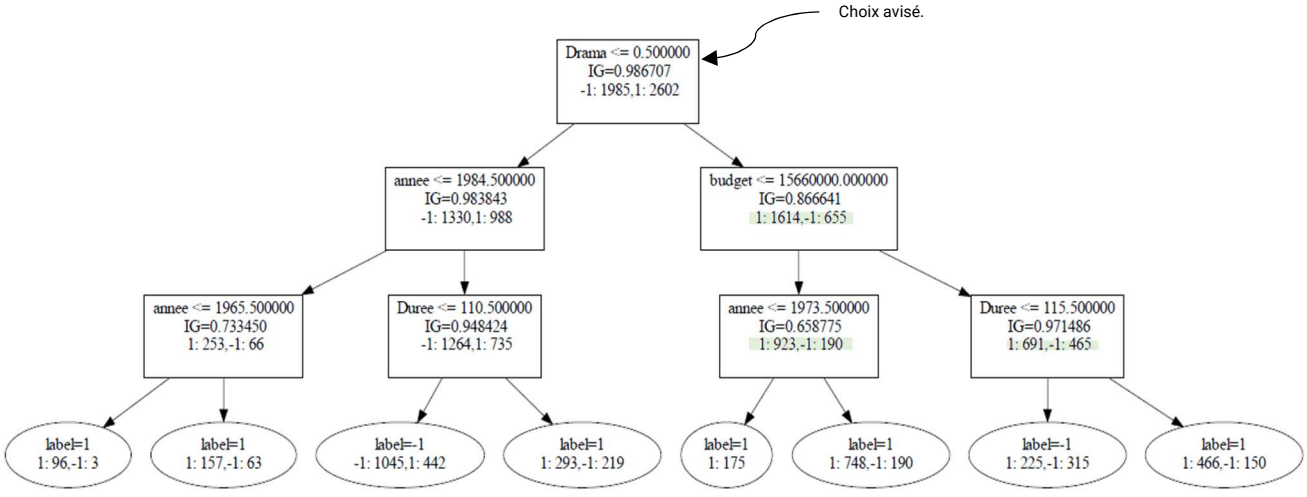
ARBRES DE DECISION

Un arbre de décision est fait de telle sorte que l'on peut suivre un chemin à travers des nœuds binaires jusqu'à atteindre la classe prédite pour chaque point des données. Le chemin à suivre dépend de ses caractéristiques. C'est une méthode assez intuitive lorsque les caractères (*features*) représentés dans la base sont multiples.

Pour générer un arbre, il est bon de savoir où partitionner les données. On utilise les caractères dont l'entropie indique un grand potentiel de différenciation. L'entropie d'un ensemble homogène est nulle, celle d'un ensemble également divisé vaut 1. Le but est de faire en sorte que les feuilles de l'arbre aient une entropie nulle.

Pour chaque attribut, on peut conditionner ce calcul aux valeurs des autres. C'est la différence entre entropie brute et conditionnelle qui est critique. Une valeur proche ou égale à 0 pour un attribut signifie qu'il n'apporte que peu d'information ; l'entropie conditionnelle à la partition considérée est élevée. Au contraire, une différence proche de 1 signifie que l'attribut considéré apporte une quantité importante d'information et donc que son entropie conditionnelle est proche de 0.

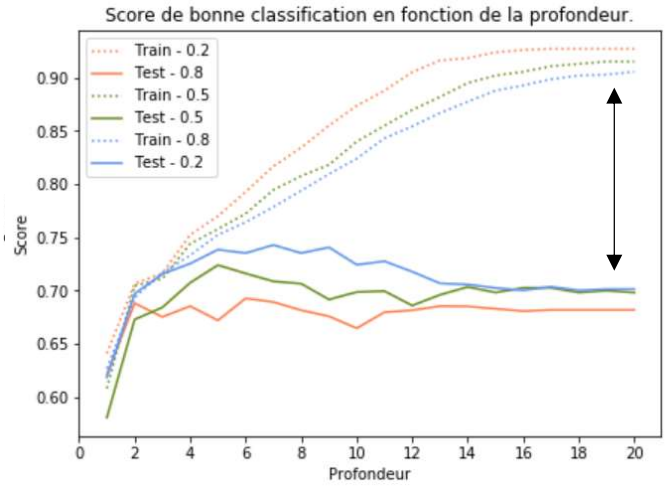
Q3 // Le meilleur attribut pour la première partition est celui qui correspond à la colonne 17, 'Drama'. C'est lui qui présente la plus grande différence et qui est donc le plus informatif pour faire diverger deux chemins. C'est d'ailleurs ce qui est fait pour tous les arbres créés.



Q4 // Un arbre sépare les données en deux sous-groupes à chaque nœud. Alors qu'on avance en profondeur, les échantillons séparés dans chaque case (ils somment à 1 sur deux branches qui viennent d'être séparées) sont de moins en moins nombreux. C'est normal ; on raffine la prédiction à mesure que la répartition se fait.

Q5 // En observant ces arbres, on constate aussi que les scores de bonne classification augmentent avec la profondeur. Comme la base d'entraînement et la base de test sont identiques, augmenter la profondeur revient à mieux coller aux données d'entraînement, ce qui induit nécessairement une amélioration des scores de bonne classification. Aller plus loin, c'est apprendre davantage et augmenter ses certitudes. D'éventuelles performances en test peuvent pâtir de cette confiance excessive, et c'est ce qu'on constatera avec la détection du surapprentissage.

Q6 // Ces scores ne sont pas du tout un indicateur fiable du comportement de l'algorithme, puisqu'ils sont obtenus directement avec les données qui ont permis la construction du modèle. Pour obtenir un modèle plus fiable, il est nécessaire de diviser l'ensemble des données en deux : une partie pour l'entraînement du modèle, une autre pour les tests.

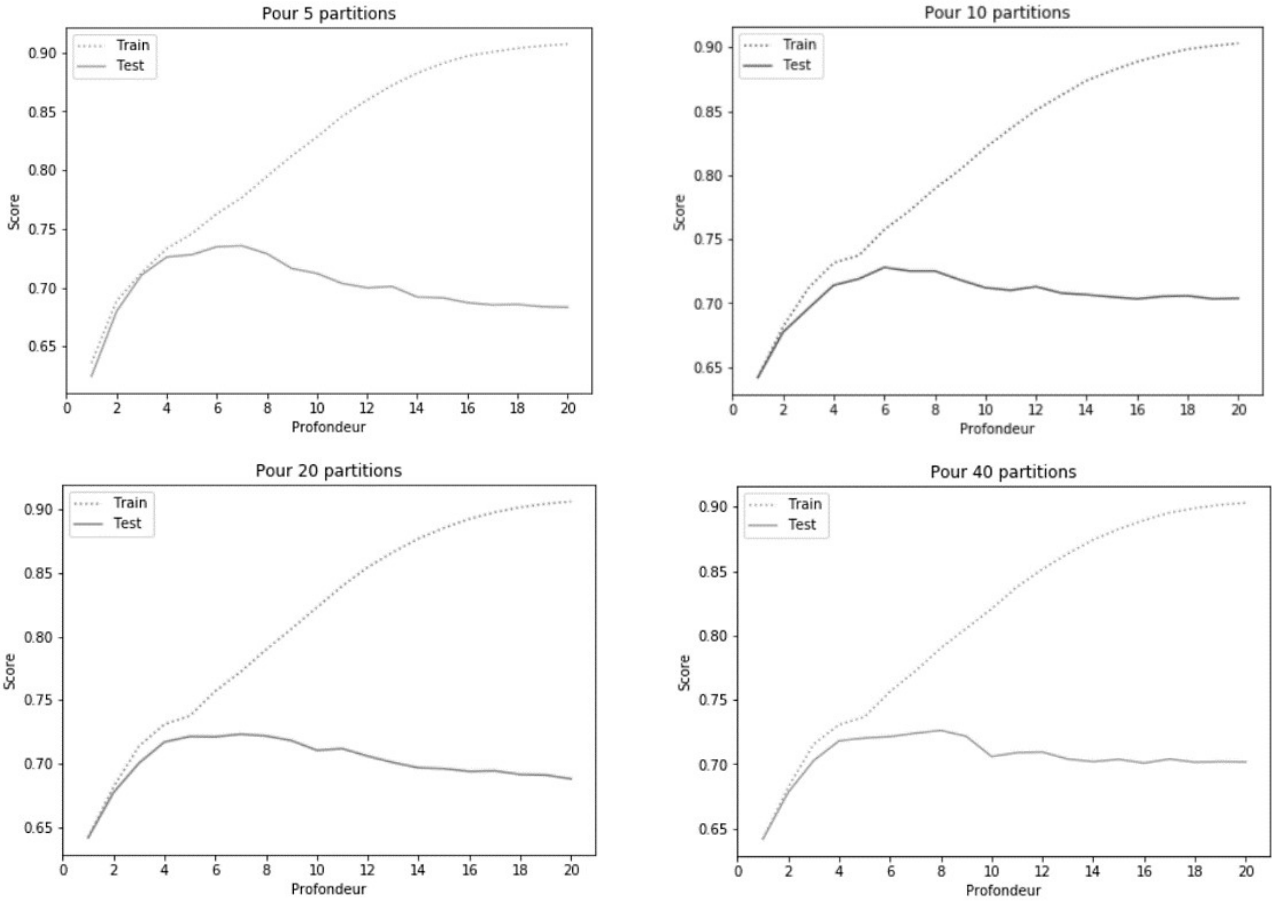


Q8 // On essaie avec plusieurs partitions. Lorsqu'il y a peu d'exemples d'apprentissage (courbe orange), la qualité de la classification sur l'ensemble de test n'est pas très bonne. A l'inverse, lorsqu'il y a beaucoup d'exemples (courbe bleue), la qualité est légèrement meilleure. C'est tout de même difficilement estimable, comme l'ensemble de test contient trop peu de données. Un juste milieu reste donc nécessaire.

En tout cas, quelle que soit la taille de l'ensemble d'entraînement, la dynamique générale des courbes est identique : en fonction de la profondeur, une augmentation constante pour le score en train, et un pic très léger pour le score en test. Lancer le script plusieurs fois n'y change rien.

C'est que la profondeur apprise est un facteur essentiel. On l'a vu, elle n'a que peu d'influence sur les performances en test, qu'elle finit même par gâter, mais améliore exagérément la précision en entraînement. Ces comportements d'erreur divergents en train et en test traduisent un effet de surapprentissage, ce qui confirme les observations effectuées en 5.

Q9 // Avec cette méthode de séparation des données, il a été nécessaire de faire un compromis entre qualité du modèle et fiabilité de la prédiction. Effectuer plutôt une validation croisée permettrait d'avoir des résultats a priori plus stables et plus fiables. C'est ce qu'on vérifie immédiatement.



Il est classique d'utiliser cinq partitions des données pour la validation croisée. En générer plus n'a franchement pas d'impact sur les performances en test, où le modèle est bien moins efficace qu'en train.
Ces validations croisées plus ou moins longues et fastidieuses auront au moins servi à confirmer le surapprentissage, qui est très net dans tous les cas lorsque la profondeur d'arbre est exagérée. Après un plateau, les scores en test commencent systématiquement à décroître pendant que ceux du train explosent.

Pour sélectionner le modèle, on prendra donc soin de rester au niveau du pic de performance, atteint pour une profondeur d'arbre comprise entre 6 et 8, et où l'écart de score train/test est faible. L'allure des courbes montre qu'aller plus loin est contreproductif.

ESTIMATIONS DE DENSITÉ

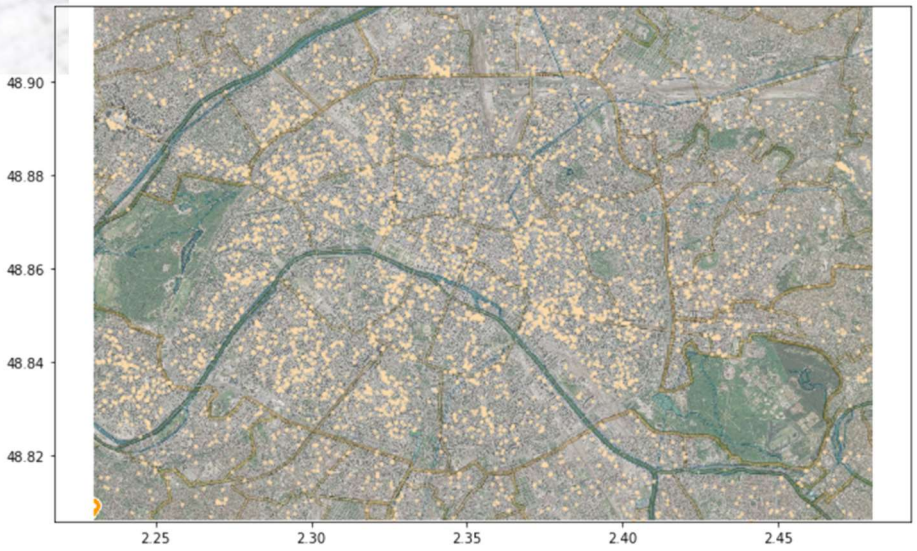
Une estimation de densité permet de modéliser la répartition d'un ensemble de données, de sorte à pouvoir en générer d'autres selon la même distribution. On reconnaît alors les régions de densité élevée, les variations entre diverses classes, etc. La prédiction est elle aussi possible. Deux méthodes d'estimation sont ici mises en œuvre, la méthode des histogrammes et la technique des fenêtres de Parzen : elles ne sont pas paramétriques, c'est-à-dire qu'elles ne supposent rien sur l'appartenance de la fonction de densité recherchée à une famille donnée.

Les données concernées sont des POI (pour Points of Interest) de Paris, pour lesquels on dispose au moins de coordonnées en latitude et longitude, d'une qualification (bar ? restaurant ? nightclub ? ATM ? etc.) et parfois d'une note.

Pour cette première partie, nous avons choisi de travailler sur les `home_good_stores`, nombreux et disséminés sur toute la ville, mais qui présentent ponctuellement des concentrations supérieures dans certains quartiers.

On observera la façon dont chaque algorithme va gérer ces points de concentration.

Vous travaillez actuellement avec les home_goods_stores de Paris.
Vous vous intéresserez à leur répartition sur la ville.



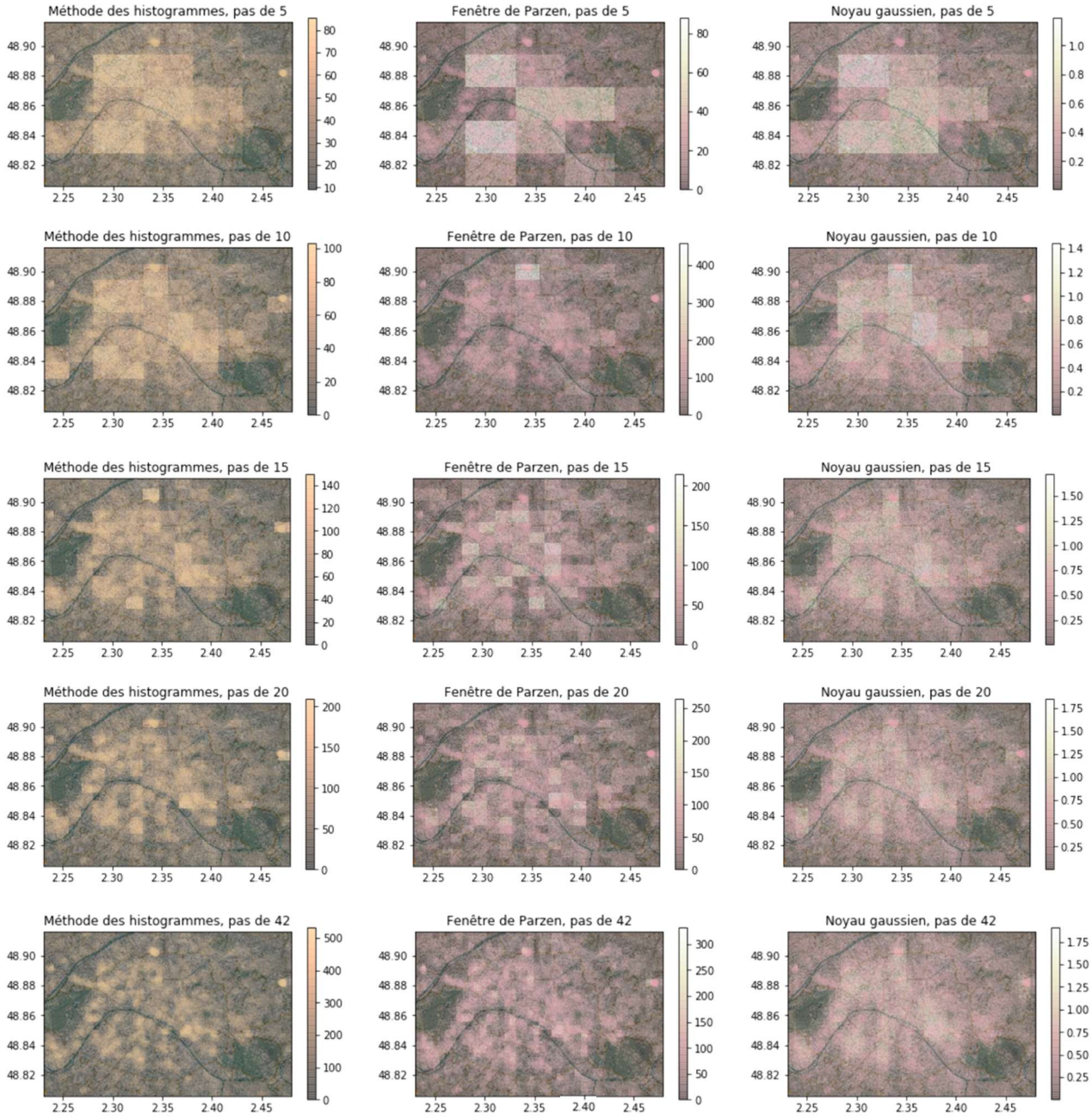
Pour la méthode des histogrammes, la densité des POI peut être estimée en appliquant simplement une grille sur la carte : en choisissant la précision de cette grille, soit le nombre total de cases, on travaille par simple comptage pour donner une image globale de la fonction de densité. Le nombre de subdivisions est décidé par le paramètre `step` $\in$ `[5,10,15,20,42]`.
Mais le problème de la méthode des histogrammes est sa fragilité. Tout décalage de la grille, même minime, modifie le nombre de points par case et change radicalement la fonction de densité. Au lieu de se fonder sur la grille dès le départ, on décide de commencer avec une notion de voisinage (fenêtre de Parzen) ; on évite ainsi le problème des frontières arbitraires. Quelle que soit sa position exacte par rapport à la grille, d'un côté ou de l'autre des frontières, l'impact d'un point en tant que voisin d'un autre sur l'estimation ne dépendra plus que de la distance qui les sépare. Pour savoir quoi retenir dans un voisinage, et avec quel poids, on définit des types de "noyaux", comme les noyaux uniformes (de Rosenblatt) et gaussiens.
Pour la visualisation, la fenêtre de Parzen nécessite quand même une grille, toujours paramétrée par `step`, ce qui nous permet d'observer l'output des algorithmes pour un même `step` côte à côte. Le Notebook contient de nombreux graphiques de comparaison, nous n'en retiendrons que quelques-uns ici (à voir en page suivante).

→ De petites valeurs de `step` induisent une généralisation extrême pour la méthode des histogrammes ; la densité s'uniformise. Pour de grandes valeurs de `step`, la résolution s'améliore, mais l'estimation n'est pas satisfaisante : on retrouve des cases où la densité est très faible, voire nulle, alors que celle adjacente est très élevée. Ce n'est pas représentatif de la réalité : c'est une forme de surapprentissage. Les valeurs collent trop aux données, il ne s'agit plus d'une modélisation, c'est sans intérêt.

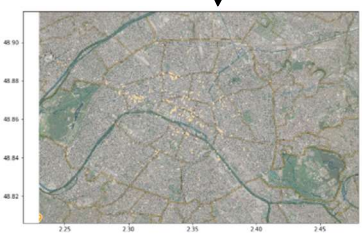
→ On introduit pour la fenêtre de Parzen une constante dite de *lissage* qui paramètre la largeur maximale des voisinages. La variation de `h` $\in$ `[0.042,0.019,0.005,0.001]` est critique pour les méthodes à noyaux. En modulant la taille du noyau uniforme ou l'écart-type de la gaussienne, elle change en fait la zone d'influence de chaque point dans son entourage. On note que des h supérieurs à 0.1 sont inconsistants (flou gaussien sur l'ensemble de la carte, lissage excessif et perte de détails), et que des h trop petits ne permettent plus le calcul : la densité s'annule là où un point n'a pas de voisinage immédiat. C'est aussi à voir en page suivante, où on constate que le noyau uniforme est moins robuste à cet effet que le noyau gaussien.

→ Un modèle est de bonne qualité si les points de test ont une vraisemblance suffisamment élevée selon la fonction de densité calculée sur la zone. Cette qualité peut aussi être évaluée approximativement en générant des données à partir de l'estimation : il est alors possible de juger de leur réalisme en observant leur concordance avec les données initiales.

→ Quant au paramétrage automatique, la validation croisée classique n'est pas permise. Nous ne disposons pas de la « vraie » densité. Il est cependant possible de faire une estimation en posant des frontières d'écart-type : en observant les cases de la grille, elles ne doivent pas toutes avoir des valeurs trop proches entre elles – auquel cas, aucun intérêt, c'est montrer un groupe homogène ; ni trop disparates – auquel cas la modélisation ne sert à rien, c'est montrer la carte telle quelle. Il faudrait juste un a priori sur la distribution générale des données (répartition vaguement uniforme = écarts-types faibles entre les cases).

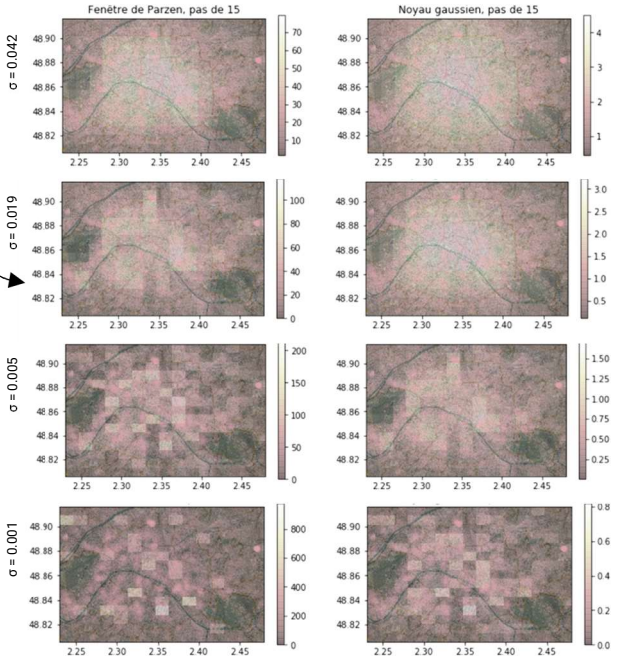


Comparaison entre les données générées pour les night-clubs et les données originales

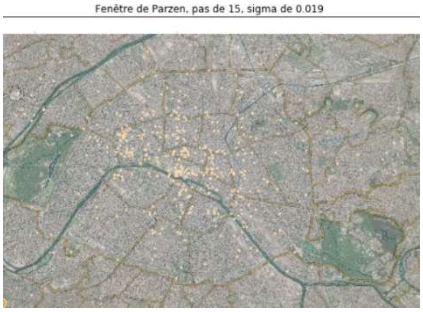


Étude de l'impact du pas de grille pour un même paramètre de lissage

Étude de l'impact du lissage pour un même pas de grille



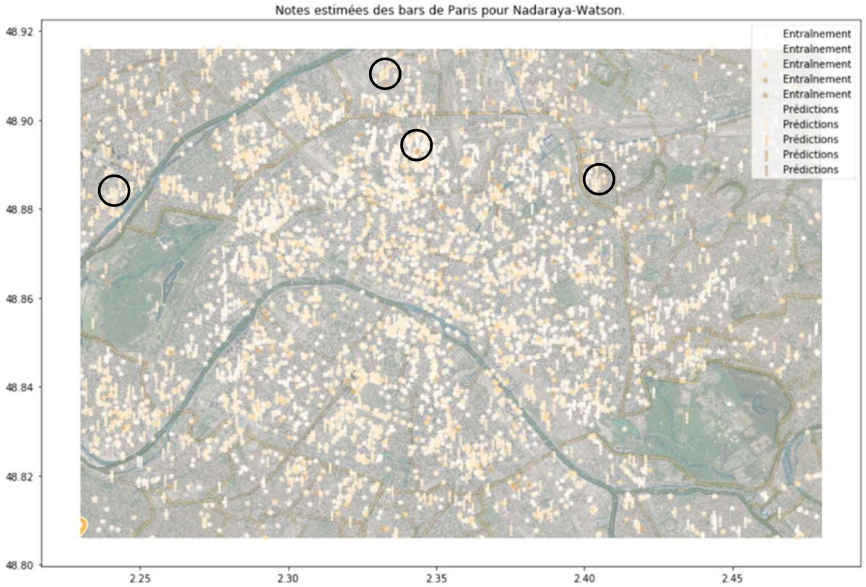
Ci-dessus, les données réelles. Ci-contre, les données générées selon l'estimation en ajoutant un peu de bruit pour le naturel : une répartition très satisfaisante.

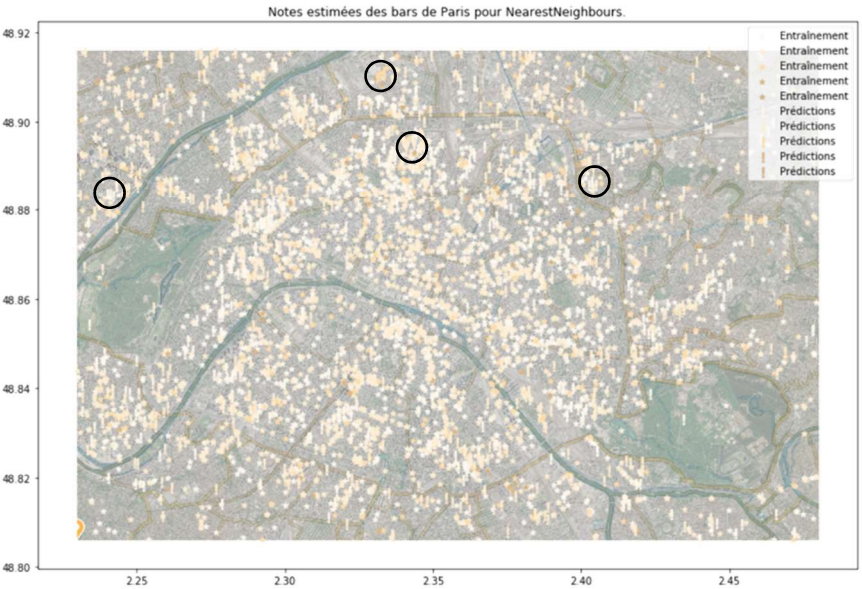


Un extra du TME consistait à mettre en place une stratégie de classification (estimateurs KNN et Nadaraya-Watson) pour estimer les notes manquantes des POI. Nous étudions cette fois les bars de Paris en espérant pouvoir profiter bientôt de nos estimations.
En vert foncé, les notes que l'on recherche, sachant que celles qu'on connaît sont globalement excellentes (au moins supérieures à trois, en clair ; les points plus sombres sont rares).

Même s'il est absurde de juger un lieu en fonction de ses voisins, les résultats n'ont pas l'air incohérents.


```
Notes prédites par Nadaraya-Watson :    [3.96 3.92 3.81 3.63 4.   3.95 3.99 3.96 3.98 3.95 4.05 3.63 3.79 4.02]
Notes prédites par NearestNeighbours :  [4.05 3.76 3.82 3.41 4.3  3.94 3.92 4.06 4.06 4.09 4.   2.94 3.72 3.94]
```





On notera que les notes connues situées dans le voisinage immédiat d'un POI ont plus d'impact sur les notes prédites par K-Nearest Neighbours. Elles « tirent » l'estimation à elles sans qu'il n'y ait de compensation générale sur la zone. Les mauvaises notes, comme les très bonnes, gagnent de l'importance, alors qu'un Nadaraya-Watson a tendance à « lisser » ses prédictions et à toutes les rapprocher de la moyenne.

On constate bien cette différence sur les zones entourées par exemple, où la couleur des notes devinées change entre l'un et l'autre des deux estimateurs – pour le mieux ou pour le pire, en fonction des extrêmes qui les entourent.

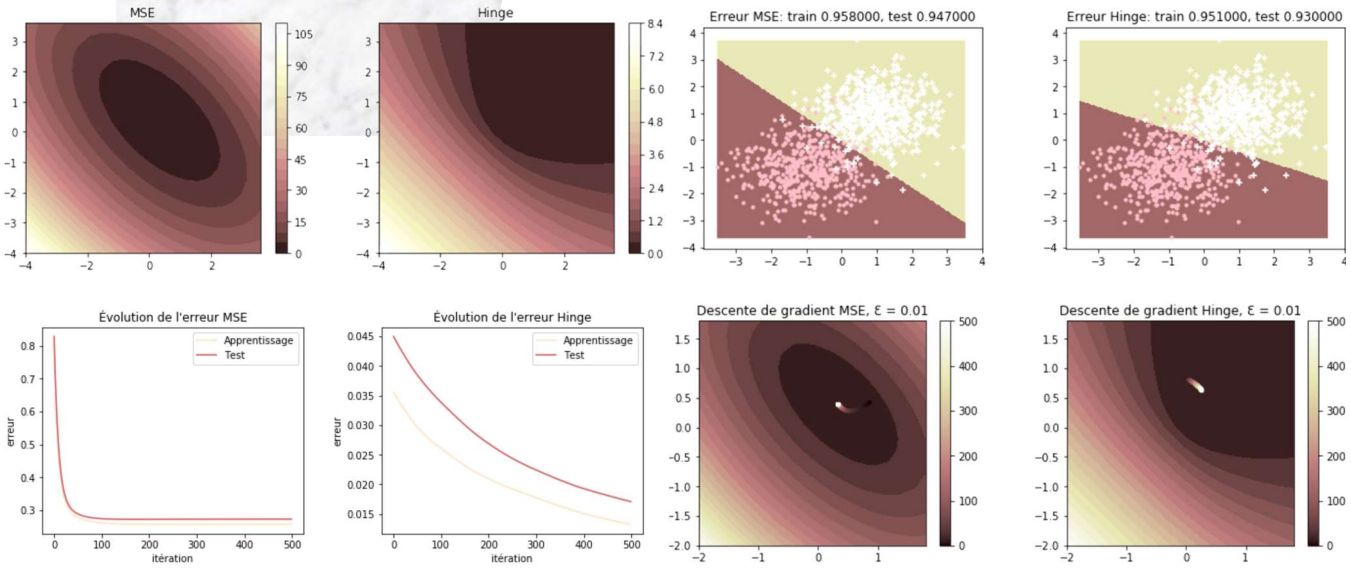
L'intuition visuelle est confirmée par la répartition donnée ci-dessous.

```
Répartition des notes selon Nadaraya-Watson :     0··1 | 0,   1··2 | 0,   2··3 | 2,   3··4 | 583,   4··5 | 252
Répartition des notes selon NearestNeighbours :   0··1 | 0,   1··2 | 0,   2··3 | 4,   3··4 | 500,   4··5 | 333
```

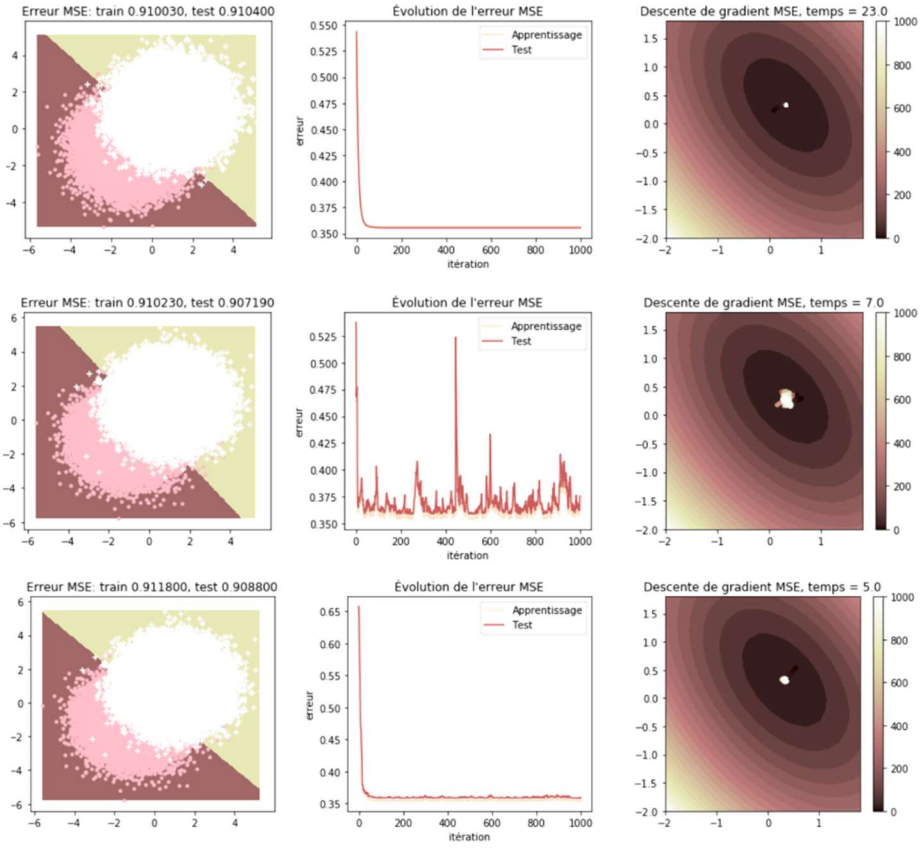
GRADIENTS ET PERCEPTRONS

La régression linéaire classique et le perceptron diffèrent dans la fonction d'erreur utilisée : `MSE` (erreur classique des moindres carrés) ou `Hinge` (coût spécifique au perceptron). En classification supervisée, c'est elle qui permet de recalibrer les poids calculés à chaque étape de la descente de gradient.

Ayant implémenté les deux, on affiche la forme des frontières liées et le résultat sur une première tâche de convergence : on utilise pour cela les données artificielles fournies. On constate que les deux classifieurs s'en sortent bien sur des données bruitées (ε du bruit = 0.8), et qu'ils n'ont pas beaucoup de pas à faire pour obtenir une frontière suffisamment convaincante.



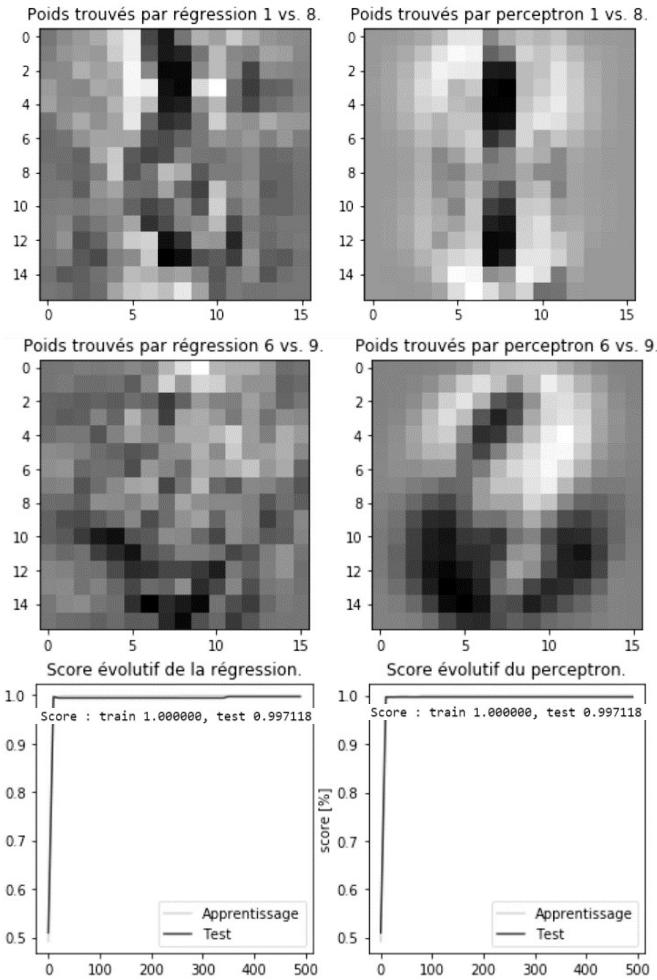
En plus de la prise en compte du biais, dont nous verrons les effets plus bas, nous avons par ailleurs implémenté plusieurs méthodes de descente de gradient que nous avons testées sur des données plus conséquentes (10000 points de données pour bien visualiser la différence de traitement).



La différence entre les types de descente de gradient `batch`, `mini-batch` et `stochastic` réside dans le nombre d'exemples utilisés pour effectuer une seule étape de mise à jour des poids.

L'approche batch réutilise l'ensemble des données. C'est la plus stable. Par contre, l'approche stochastique extrait une seule donnée de la base et recalcule les poids en fonction de ce qu'elle trouve. C'est bien plus rapide, mais sans heuristique particulière, le fait de tirer une donnée d'entraînement plutôt qu'une autre aura un impact conséquent – surtout si le nombre d'itérations total est limité.

Ce choix influence donc, comme on le voit, la justesse du résultat et le temps de calcul. Sur un grand jeu de données, l'approche mini-batch est donc un compromis qui donne une précision intéressante en très peu de temps.

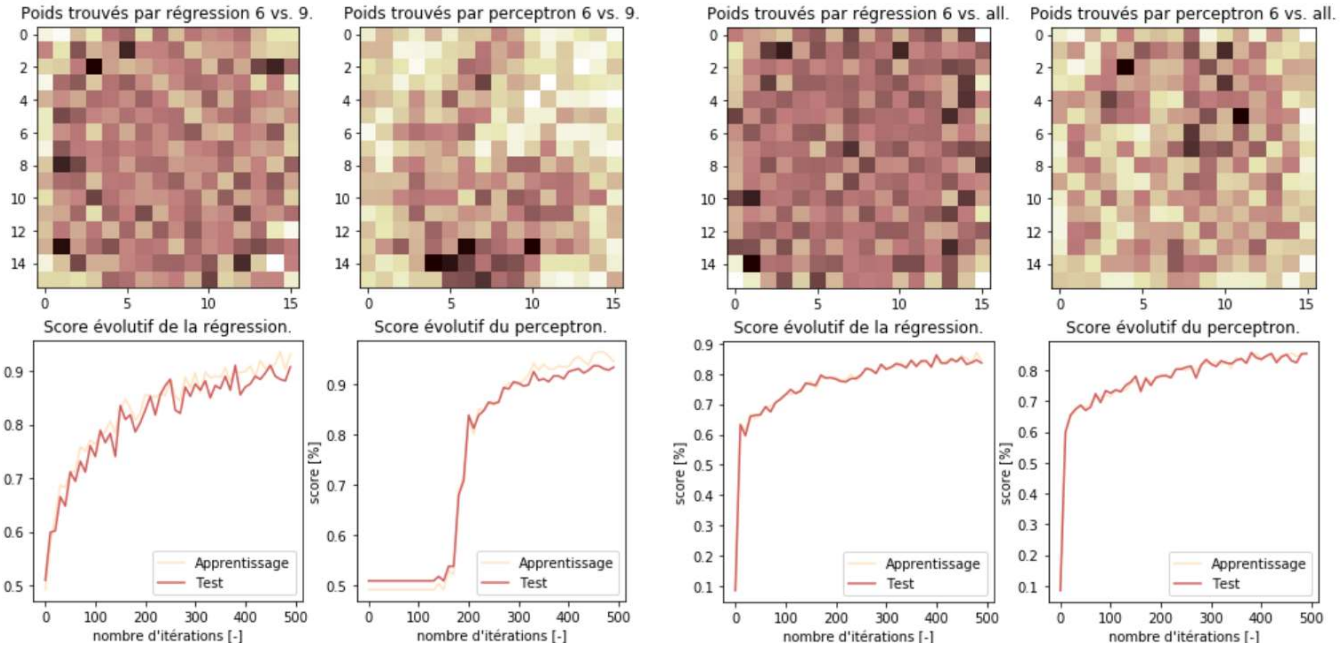


Forts de ces vérifications, on passe aux données réelles avec une classification USPS (chiffres manuscrits). Les comparaisons peuvent se faire en 1 vs. 1, où une classe est comparée à une seule autre, ou en 1 vs. all où l'apprentissage se fait face à tout le reste de la base. En utilisant parallèlement la régression et le perceptron, on réalise un ensemble de tests sur 1 vs. 8, 6 vs. 9 et 6 vs. all, en s'intéressant à chaque fois à l'évolution de l'erreur en fonction du nombre d'itérations.

https://www.youtube.com/watch?v=vZuFq4CfRR8

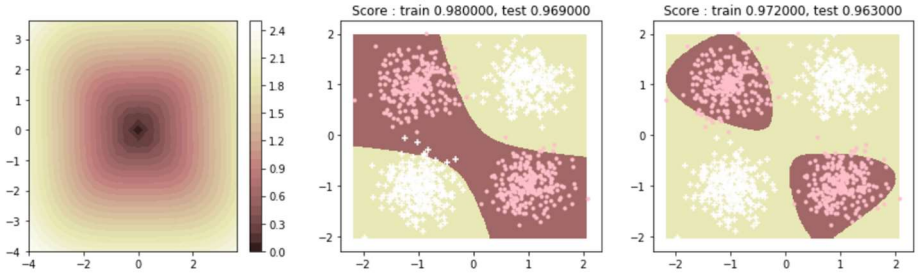
L'entraînement est concluant. Sur les problèmes bi-classes, les performances en train sont à 100% de précision, et autant en test (les courbes ne sont pas discernables sur le graphique). Sur 6 vs. all, le perceptron se maintient à 99% de précision, ce qui est louable, et 1.5% de moins en test. L'erreur de classification diminue très vite dans les premières itérations avant de se stabiliser à des valeurs quasi-nulles. Il suffit de 300 itérations pour obtenir un résultat parfait. Est-ce du surapprentissage ? Pour nous, non ; l'écart est minime. On ne peut pas dénoncer un excès d'entraînement nuisible aux performances en test.

Ces résultats ne sont obtenus que pour une initialisation des poids proche de zéro (pour permettre aux contours des classes antagonistes de se détacher avec des poids positifs et négatifs). C'est un biais inutile sans vraies conséquences sur le score. Une initialisation aléatoire permet un résultat final excellent, dont le visuel est juste moins intéressant du fait du bruit parasite. La convergence se fait très bien malgré ce bruit (la zone d'erreur quasi-nulle est large). Ci-dessous, les résultats sans biais.



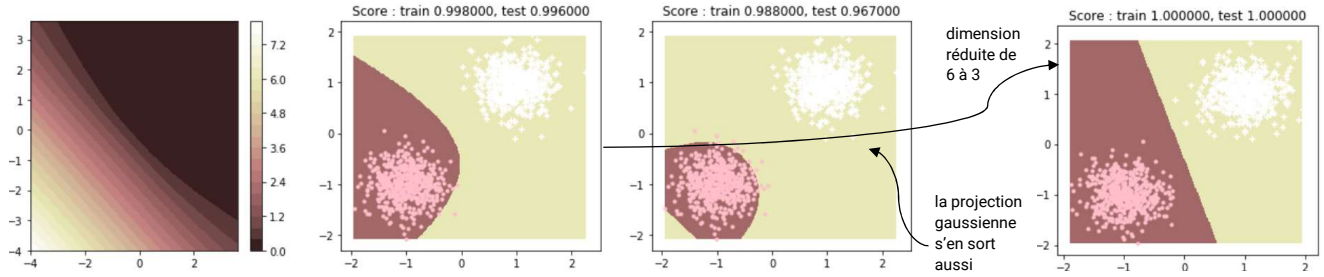
On s'intéresse maintenant aux performances du perceptron basique sur un dataset artificiel pour en comprendre les limites. Quand les données ne sont pas linéairement séparables, le modèle n'est plus efficace. C'est normal ; il n'a aucun moyen de comprendre leur répartition, il s'agit bien d'un classifieur *linéaire*. Une projection permet de l'améliorer en étendant l'espace de représentation initial 2D (en ajoutant des dimensions). Ce « kernel trick » permet de guider le choix des hyperplans séparateurs.

On en code deux. Pour l'un, x=[x_1,x_2] devient [1,x_1,x_2,x_1^2,x_2^2,x_1,x_2], soit une projection quadratique $\varphi(x) = (1, x_1, x_2)'(1, x_1, x_2)$. Le degré de la projection d'arrivée (ici 2) est réglable dans notre implémentation (projection polynomiale générique).
Pour l'autre, on crée d'abord une grille sur avant de calculer la similarité gaussienne des données à chaque point de cette grille : $s(x, p^{i,j}) = Ke^{(||x-p^{i,j}||^2)/\sigma}$. Avec l'espace découpé en 10 sur chaque dimension, le vecteur de poids w est de dimension 100.
Ces deux options de projection peuvent être incluses dans une nouvelle classe de perceptrons sur laquelle on expérimente.

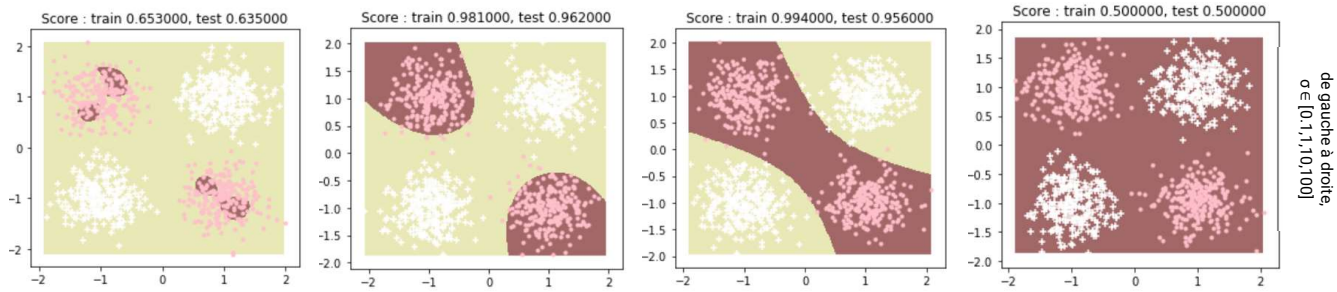


Les frontières trouvées deviennent intéressantes, la notion d'"intérêt" étant fondée sur le pourcentage de bonne classification. C'est parfait pour le type n°1, les doubles gaussiennes (projection polynomiale à gauche, projection gaussienne à droite).

→ Comme d'habitude, diminuer raisonnablement le nombre de dimensions en projection polynomiale améliore la performance en test du perceptron en évitant le surapprentissage. C'est déjà le cas sur des données linéairement séparables.

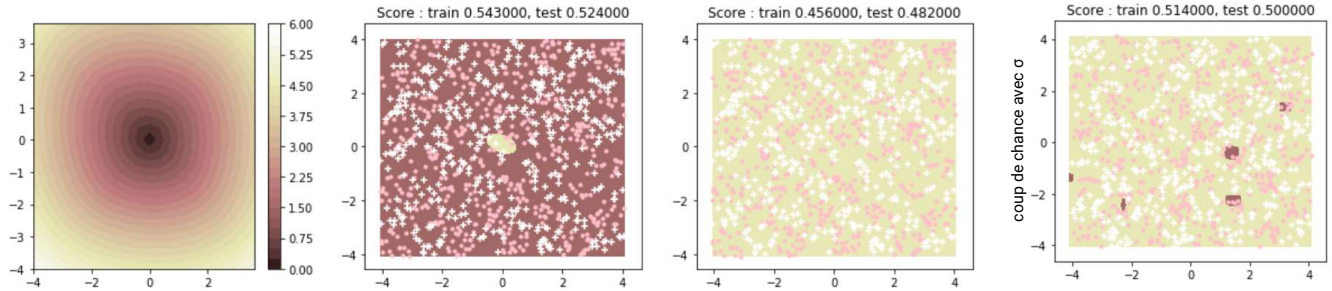


→ On peut aussi étudier le paramètre σ de la projection gaussienne. Un petit σ cantonne la prédiction aux points observés, en ne surlignant que les clusters très concentrés. Quand σ augmente, toutes les similarités tendent vers 1 et la classe majoritaire est prédite pour tout le monde. On peut donc voir σ comme un coefficient de généralisation qui étend la zone d'influence de la grille.



→ Vaut-il mieux considérer beaucoup de points pour la base de projection gaussienne? En augmentant le nombre de points, on augmente quadratiquement le nombre de dimensions et conséquemment la complexité du modèle. Sur les données de type 1, on ne constate pas d'amélioration particulière. Il faut donc se méfier si l'on veut éviter le surapprentissage.

→ Une projection gaussienne (à droite) est le seul moyen de différencier les données de type 2, réparties de façon homogène. Aucune chance du côté de la projection polynomiale (à gauche). Le réglage de σ donne parfois des surprises, mais n'est pas très fiable. Le noyau gaussien des SVM saura mieux faire la distinction, il s'agit probablement de l'effet des marges « douces ».



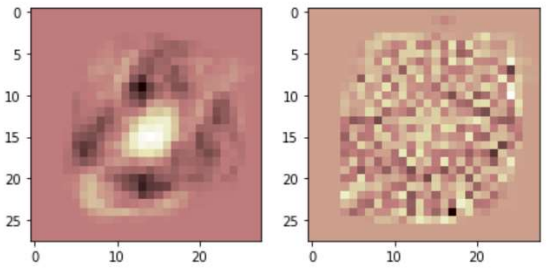
SUPPORT VECTOR MACHINES

Les SVM permettent de contourner les limitations des perceptrons. Pour différencier deux classes, elles maximisent la distance (la *marge*) que la frontière choisie met entre elles – de sorte à augmenter leur capacité de généralisation. Les points qui entourent cette marge sont les « vecteurs support » sur lesquels subsiste une incertitude ; ce sont eux qui servent à son calcul, leur nombre diminue au fil des itérations, et c'est d'eux que les SVM tirent leur nom.

→ On utilisera tout du long les SVM proposées par la bibliothèque `sklearn`. Elle contient d'autres algorithmes désormais familiers. C'est notamment l'occasion pour nous de comparer notre Perceptron à la version built-in, avec un résultat encourageant.

```
Précision maison sur deux classes USPS : train 1.000000, test 1.000000
Précision sklearn en validation croisée : 0.9961832061068702
```

→ Avant de commencer le travail, on s'intéresse à la régularisation de Tikhonov et à son impact sur le score du perceptron. Cette manipulation, qu'on explorera en détail pendant le projet, permet de limiter les dégâts du surapprentissage en réduisant la norme du vecteur de poids. Sur un problème de classification MNIST, 0 vs. 8, ci-contre : à gauche les poids trouvés par notre perceptron régularisé (où les nombres apparaissent) ; à droite ceux d'une SVM (bruit du fait de la génération aléatoire).



```
Précision du perceptron régularisé : train 0.975964, test 0.981576
Précision de la SVM linéaire : train 1.000000, test 0.989253
```

La différence avec la SVM linéaire réside dans l'inversion de l'ordre des scores train/test. La régularisation de Tikhonov permet une meilleure généralisation du modèle, donc un gain de performance en test. Elle est même ponctuellement meilleure que pour la SVM linéaire, dont on avait pourtant dit qu'elle contournait les limites des perceptrons et améliorait systématiquement leurs scores.

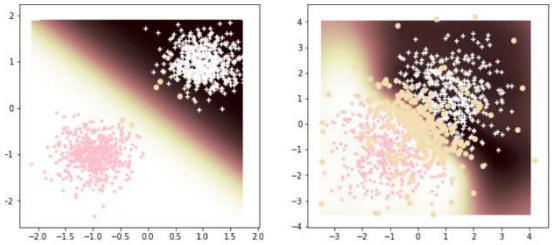
→ Les noyaux des SVM sont prédéfinis : linéaire, polynomial, gaussien et sigmoïde. On essaie de les comparer. Au passage, on se souvient que les SVM sont sensibles à la normalisation des données ; pourtant, les résultats n'étaient pas sensiblement améliorés avec une `Pipeline`, que ce soit avec `MinMaxScaler` ou avec `StandardScaler`. Davantage, le nombre de vecteurs support diminue en la retirant, pour des performances similaires sinon meilleures - il n'y a donc pas d'intérêt. On s'en étonne.
Sur deux classes USPS, comparaison des performances des kernels avec (à gauche) ou sans (à droite) normalisation.

```
// linear // Précision en validation croisée : 0.99771
// linear // Nombre de vecteurs support : [24 24]
// poly // Précision en validation croisée : 0.99771
// poly // Nombre de vecteurs support : [165 166]
// rbf // Précision en validation croisée : 0.98777
// rbf // Nombre de vecteurs support : [96 82]
// sigmoid // Précision en validation croisée : 0.99618
// sigmoid // Nombre de vecteurs support : [28 29]
```

Un kernel linéaire va très vite, consomme peu d'énergie et semble faire au mieux. À part pour le sigmoïde, tout est plus juste sans »» pipeline de scaling.

```
// linear // Précision en validation croisée : 0.99847
// linear // Nombre de vecteurs support : [21 22]
// poly // Précision en validation croisée : 0.99771
// poly // Nombre de vecteurs support : [91 93]
// rbf // Précision en validation croisée : 1.00000
// rbf // Nombre de vecteurs support : [42 45]
// sigmoid // Précision en validation croisée : 0.99771
// sigmoid // Nombre de vecteurs support : [41 38]
```

→ On compare encore les noyaux sur les données artificielles 2D (page suivante), sans `Pipeline`. En général, si la tâche est difficile pour le classifieur, s'il met du temps à converger, le nombre final de support vectors augmente drastiquement. Plus la convergence est lente, moins l'algorithme a le temps de préciser sa classification : l'incertitude demeure.



Sur du 2D, l'affichage des frontières et des support vectors est envisageable. Lorsque la classification est bonne, ils se concentrent près de la frontière. Ils se dispersent et se multiplient en fonction du bruit ou d'autres facteurs : lorsque le classifieur a plus de mal avec les données, il utilise un maximum de points supports car les cas limite sont nombreux. Ici, il a suffi d'augmenter le bruit (à gauche noyau linéaire, peu de bruit, à droite noyau gaussien, bruité) pour constater la différence.

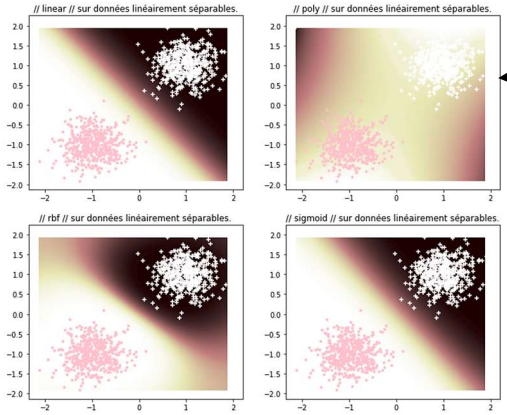
→ Les kernels présentés sont paramétrables. Tous dépendent d'une constante `C` dite de *régularisation*. `C` pondère la « dureté » de la frontière : elle permet d'équilibrer la simplicité de la surface de décision et la qualité générale de la prédiction. Une petite `C` étend la marge, quitte à augmenter le nombre de vecteurs support (points potentiellement mal classés). Une `C` élevée donne une frontière plus alambiquée (du fait d'une marge plus réduite) et laisse donc beaucoup moins de liberté au classifieur, tout en augmentant le risque de surapprentissage. Un noyau linéaire n'a pas d'autres paramètres.
Un noyau polynomial dépend en plus du degré `degree` de la projection (comme dans le TME 3) qui est évidemment le degré maximal du polynôme. Il y a aussi `coef0` qui correspond au terme scalaire.
Un noyau gaussien dépend de `gamma`, similaire à (inverse de) notre σ dans le TME précédent. σ régulait la précision de l'apprentissage ; ici, un `gamma` élevé restreint l'apprentissage aux données, et un `gamma` faible permet de mieux généraliser.
On ne s'intéressera pas davantage au sigmoïde.

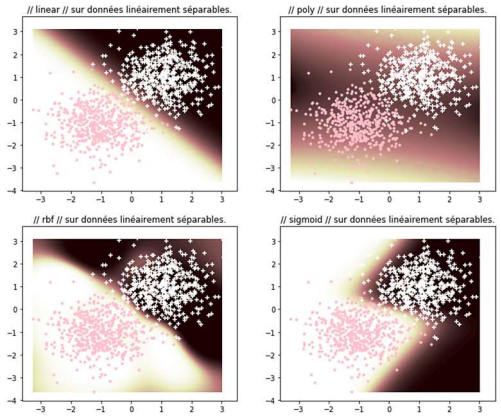
```
// linear // Précision en validation croisée : 1.00000
// linear // Nombre de vecteurs support : [2 2]
// poly // Précision en validation croisée : 0.50600
// poly // Nombre de vecteurs support : [498 498]
// rbf // Précision en validation croisée : 1.00000
// rbf // Nombre de vecteurs support : [4 5]
// sigmoid // Précision en validation croisée : 1.00000
// sigmoid // Nombre de vecteurs support : [3 3]
```

diminution augmentation (bruit !)

```
Précision en validation croisée : 0.96700 // linear //
Nombre de vecteurs support : [45 46] // linear //
Précision en validation croisée : 0.51300 // poly //
Nombre de vecteurs support : [490 491] // poly //
Précision en validation croisée : 0.96100 // rbf //
Nombre de vecteurs support : [43 49] // rbf //
Précision en validation croisée : 0.94500 // sigmoid //
Nombre de vecteurs support : [36 36] // sigmoid //
```

expérience avec degree = 2, permet de mieux classifier les données du type ci-dessous ; par contre, mauvaise performance ici



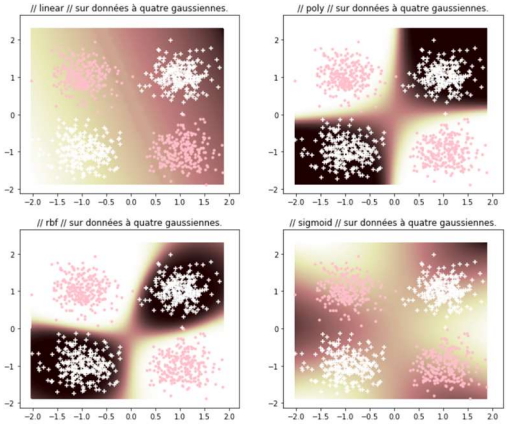


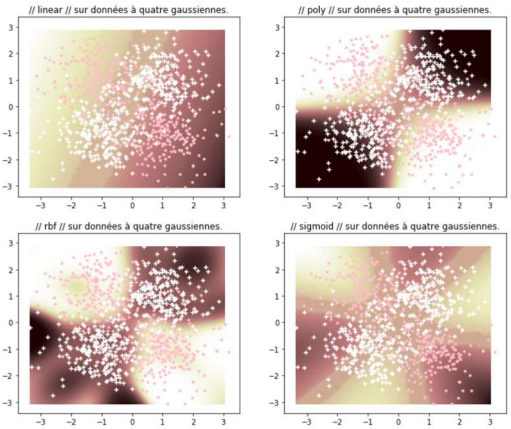
noyau linéaire plus rapide, mais performances comparables partout pour les

DONNEES LINEAIREMENT SEPARABLES

```
// linear // Précision en validation croisée : 0.63800
// linear // Nombre de vecteurs support : [497 497]
// poly // Précision en validation croisée : 0.99600
// poly // Nombre de vecteurs support : [21 22]
// rbf // Précision en validation croisée : 0.99500
// rbf // Nombre de vecteurs support : [5 6]
// sigmoid // Précision en validation croisée : 0.50800
// sigmoid // Nombre de vecteurs support : [271 270]
```

```
Précision en validation croisée : 0.58000 // linear //
Nombre de vecteurs support : [496 496] // linear //
Précision en validation croisée : 0.83500 // poly //
Nombre de vecteurs support : [214 215] // poly //
Précision en validation croisée : 0.83000 // rbf //
Nombre de vecteurs support : [187 185] // rbf //
Précision en validation croisée : 0.51300 // sigmoid //
Nombre de vecteurs support : [256 256] // sigmoid //
```



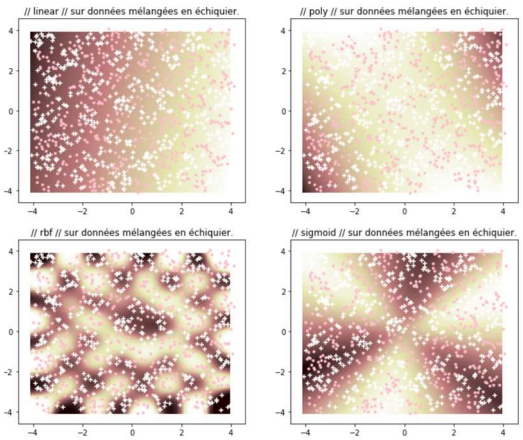


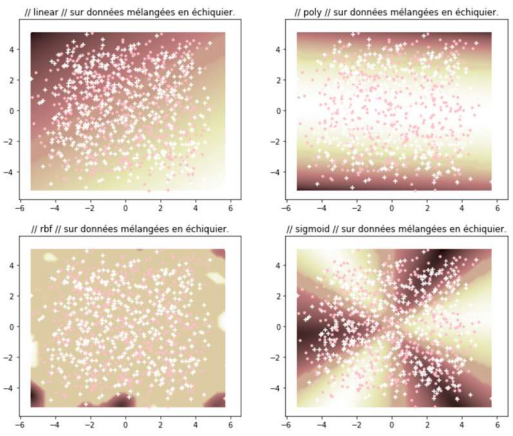
meilleure performance pour ces données, bruitées ou non : noyau gaussien

DONNEES A DOUBLES GAUSSIENNES

```
// linear // Précision en validation croisée : 0.52393
// linear // Nombre de vecteurs support : [479 479]
// poly // Précision en validation croisée : 0.52303
// poly // Nombre de vecteurs support : [478 477]
// rbf // Précision en validation croisée : 0.69701
// rbf // Nombre de vecteurs support : [332 339]
// sigmoid // Précision en validation croisée : 0.52399
// sigmoid // Nombre de vecteurs support : [254 254]
```

```
Précision en validation croisée : 0.49998 // linear //
Nombre de vecteurs support : [505 491] // linear //
Précision en validation croisée : 0.52008 // poly //
Nombre de vecteurs support : [479 476] // poly //
Précision en validation croisée : 0.52708 // rbf //
Nombre de vecteurs support : [418 412] // rbf //
Précision en validation croisée : 0.47690 // sigmoid //
Nombre de vecteurs support : [265 265] // sigmoid //
```



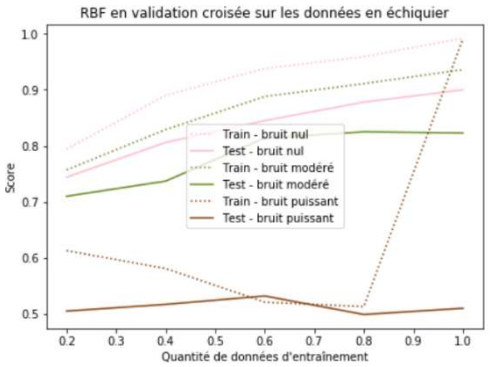


meilleure performance pour ces données, bruitées ou non : encore le noyau gaussien, encore que la tâche à droite semble impossible pour tous

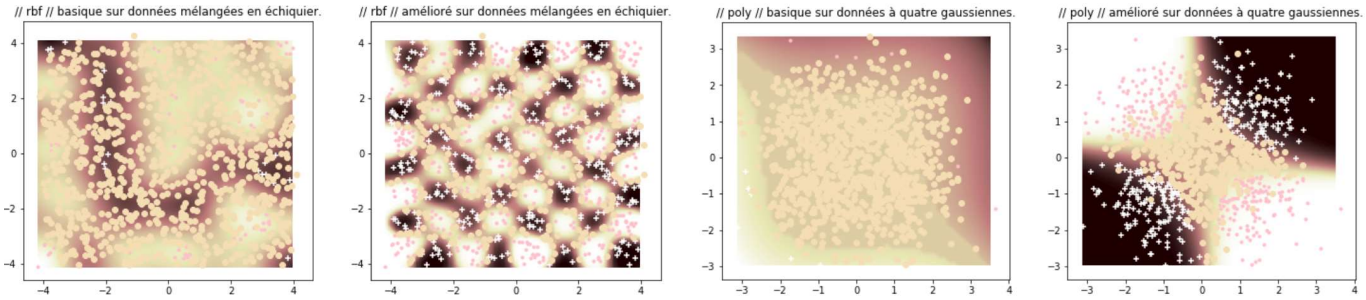
DONNEES MELANGEES EN ECHIQUIER

→ Connaissant ces paramètres, on songe à lancer une procédure automatique pour les choisir de façon adéquate. `sklearn` propose la fonction `GridSearchCV` qui s'en charge. Etant donnée la simplicité d'usage, il était tentant d'effectuer une recherche pour tous les types de données, à travers tous les types de kernels, et pour plusieurs paramétrages de bruit, pour trouver à chaque fois la meilleure option. Nous l'avons fait. Mais la lisibilité des résultats n'était pas optimale, et il y avait trop de variables, donc un temps de post-traitement non négligeable.
Nos expériences ont montré que le kernel linéaire donnait toujours d'excellents résultats sur les données linéairement séparables, bruitées ou non, quoi que vale C – et était inutile ailleurs. De même, les classifieurs « adaptés » sont toujours très performants sur les données non bruitées. On cherche donc des cas plus intéressants : le noyau gaussien est le seul à savoir lire les données en échiquier, autant en tirer un profit maximal.

`GridSearchCV` améliore significativement nos scores pour toutes les partitions train/test possibles. Dans le cas où on utilise toute la base d'entraînement, la page précédente, sans prétentions de paramétrage, donnait 0.69 pour un bruit modéré contre plus de 0.8 ici. Augmenter le nombre de données de train améliore constamment les scores en test ; on peut bien sûr décider d'un équilibre à environ 60% pour passer moins de temps sur l'entraînement et ne pas surapprendre.
On note quand même que même le `GridSearch` devient inutile à partir d'un certain niveau de bruit. Mieux vaut alors s'adresser à un tout autre classifieur. On a ouï dire (et constaté) qu'un `DecisionTree` pouvait être étonnamment performant dans ce contexte. Cette intuition n'est cependant pas de nous.



Utiliser les paramètres conseillés par la `GridSearchCV` diminue le nombre de *support vectors*. On s'y attend, puisque la convergence est meilleure et plus rapide. C'est aussi évident avec une autre `GridSearchCV`, dont l'effet est montrée ci-dessous à droite, lorsqu'on choisit le meilleur degré pour un noyau polynômial (apparemment, il est inutile d'aller chercher plus haut que 2).



Notons que sur ces cas, la `GridSearch` a conseillé des `C` élevés (1000) pour les données peu bruitées, et des `C` plus petits que 1 (0.01) lorsqu'il y a du bruit. De fait, lorsque les données sont pures, on suppose que les données de test aussi ; et il est plus logique d'avoir une frontière bien tranchée.

→ Avant-dernière chose, on s'intéresse maintenant aux SVM en tâche multiclasse. La technique 1 vs. 1 consiste à entraîner autant de classifieurs que de couples d'étiquettes possibles, pour savoir trancher à chaque fois entre deux. On choisit alors la classe pour laquelle votent un maximum de sous-SVMs. La technique 1 vs. Ω consiste à entraîner autant de classifieurs que d'étiquettes possibles. Pour chaque élément de test, la décision est alors prise en faveur du label qu'on sépare le plus facilement du reste.

On sait que la classe `LinearSVC` de `sklearn` permet des calculs rapides, et implémente automatiquement le procédé 1 vs. Ω si elle détecte de multiples classes dans les labels. En lançant une `LinearSVC` en `GridSearch` sur les données USPS brutes, on se rend compte que les performances sont vraiment bonnes pour C = 0.01. C'est donc avec ce paramètre qu'on lancera les tests pour le noyau linéaire (pour 1 vs. Ω comme pour 1 vs. 1, on a pu constater l'amélioration par rapport aux paramètres par défaut). On aurait pu utiliser la même technique pour améliorer les autres kernels, mais ce serait un peu chronophage.

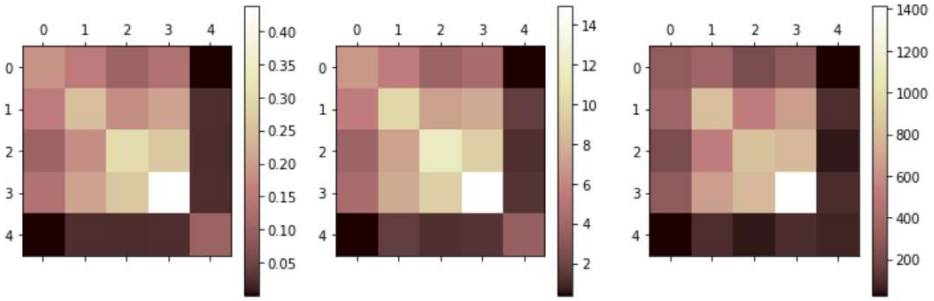
1 vs. 1, résultats après rééquilibrage.	C	1 vs. 1, résultats sans rééquilibrage.
Score pour un noyau linéaire après épissage, 0.92875	**0.01**	**Score pour un noyau linéaire après épissage, 0.92128**
Score pour un noyau polynomial après épissage, 0.92825	1	Score pour un noyau polynomial après épissage, 0.93124
Score pour un noyau gaussien après épissage, 0.94270	1	Score pour un noyau gaussien après épissage, 0.94170

1 vs. Ω, résultats après rééquilibrage.		1 vs. Ω, résultats sans rééquilibrage.
Score cross-validé pour un noyau linéaire, 0.92377	**0.01**	**Score pour un noyau linéaire après épissage, 0.91530**
Score cross-validé pour un noyau polynomial, 0.92546	1	Score pour un noyau polynomial après épissage, 0.93971
Score cross-validé pour un noyau gaussien, 0.93014	1	Score pour un noyau gaussien après épissage, 0.93921

Les remarques sont classiques : 1 vs. 1 donne de meilleurs résultats que 1 vs. Ω, mais consomme tellement de temps pour gagner quelques dixièmes en score que l'amélioration n'est pas rentable. Le noyau gaussien est globalement le meilleur. De plus, les scores en 1 vs. Ω linéaire de notre implémentation ressemblent à ceux du GridSearch (`0.91679`), c'est satisfaisant.
Enfin, le rééquilibrage des données permet de gagner beaucoup de temps, surtout en 1 vs. Ω, mais peut diminuer légèrement la qualité des prédictions (c'est un peu contre-intuitif).

→ Reste l'énigme ultime du String Kernel. On a commencé par coder un noyau capable de reconnaître des bigrammes plus ou moins distants. Plus les chaînes contiennent des bigrammes communs, plus elles sont jugées similaires. Un paramètre λ permet de quantifier la longueur de *gap* autorisée. La matrice de similarité donne, sur des portions de phrases absurdes (λ = 0.1, 0.5, 1) :

Le réglage de λ permet bien de considérer un certain niveau de similarité entre des chaînes où les deux moitiés d'un bigramme sont plus ou moins éloignées. La similarité entre 1, 2 et 3 change (ils contiennent des mots répétés de plus en plus éloignés. On sent que ce n'est quand même pas optimal.

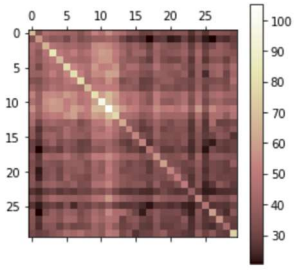


Nous avons décidé de mettre en place un apprentissage sur des poèmes anglais de l'époque de Wilde et de celle de Shakespeare. Avec un entraînement sur ces deux auteurs seuls, on pense pouvoir identifier quelques intertextes. On souhaite observer les scores de similarité et voir si une SVM est capable de classifier correctement leurs contemporains. On commence par afficher la matrice de similarité pour voir s'il est envisageable d'entraîner un modèle :

Le calcul a été fait sur trente fragments de deux vers, « à cheval » entre la partie Shakespeare et la partie Wilde, avant *shuffling* des données de train.

Deux sous-ensembles se détachent comme il faut en haut à gauche et en bas à droite de la matrice ; il est donc possible de distinguer de tout petits fragments des deux auteurs avec ce type de score de similarité. On note même une bande beaucoup plus claire entre 8 et 13, qui représente sans doute un sonnet seul de Shakespeare.

Notons que les sous-ensembles auraient été plus détachés si l'apprentissage s'était fait sur des poèmes entiers (et non plus sur de très courts extraits).



Par contre, cette petite comparaison a mis dix minutes à s'exécuter. C'est ingérable sur une SVM avec 2095 textes. Choqués par la lenteur du code, on décide de simplifier le *string kernel* à une simple comparaison de vocabulaire. En suivant les recommandations de https://stackoverflow.com/questions/26391367/how-to-use-string-kernels-in-scikit-learn, on contourne les limitations syntaxiques de `sklearn` et on adapte ce kernel simplifié à l'apprentissage en SVM.

Trouver les meilleurs paramètres de *preprocessing* (`Vectorizer(max_features...)`) est quelque chose que nous avons déjà fait en TAL, et il ne s'agit pas du coeur du problème. En passant cette étape, nous avons obtenu

```
100.0 % de bonne classification sur les données de train
```

et après `GridSearch`, nous avons pu paramétrer la SVM avec un `C` puissant (100) de sorte à obtenir

```
85.0 % de bonne classification sur un ratio train/test d'environ 70%
```

Les calculs restent terriblement lents.

En comparaison, les fonctions built-in de `sklearn` donnent un résultat beaucoup plus satisfaisant sans aucun effort. Après `GridSearchCV`, les résultats sont très probants sur le linéaire, souvent 100% de précision en train seul, et quelque chose d'assez bon (~87%) en cross-validation. La `GridSearchCV` a aussi permis d'obtenir près de 100% pour le noyau gaussien (C=1000, gamma='auto'), doublant ainsi le score par défaut.

→ On s'est aussi demandé s'il était possible de ranger des contemporains de Wilde et Shakespeare dans la classe idoine en partant de la même base. Mais les données de test sont sans doute beaucoup trop disparates.

Notre string kernel a été plus efficace que la SVM linéaire, et dépassé de peu par le kernel gaussien. Le tout ne fait pourtant pas mieux que le hasard, avec des scores qui plafonnent à 51 % de précision, et ce même après post-traitement (lissage sur la longueur de chaque poème pour unifier l'auteur).

from where thou art why should I haste me thence till I return of posting is no need made snow of all the blossoms at my feet like silver moons the pale narcissi lay and if my own true love you see ah if you see the purple shoon alas tis true I have gone here and there and made myself a motley to the view we hoisted sail the wind was blowing fair for the blue lands that to the eastward lie sometime too hot the eye of heaven shines and often is his gold complexion dimmd what's new to speak what new to register that may express my love or thy dear merit pan is not here and will not come again no love my love that thou mayst true love call all mine was thine before thou hadst this more make but my name thy love and love that still and then thou lovest me for my name is will for when these quicker elements are gone in tender embassy of love to thee when in the chronicle of wasted time I see descriptions of the fairest wights and did you talk with thoth and did you hear the moon-horned io weep and know the painted kings who sleep beneath the wedge-shaped pyramid ay though the gorgd asp of passion feed on my boys heart yet have I burst the bars to me that languishd for her sake but when she saw my woeful state thy bosom is endeared with all hearts which I by lacking have supposed dead how giant grettir fought and sigurd died and what enchantment held the king in thrall betwixt mine eye and heart a league is took and each doth good turns now unto the other and like a withered leaf the moon is blown across the stormy bay o lonely himalayan height grey pillar of the indian sky where is that spirit which living blamelessly yet dared to kiss the smitten mouth of his own century then being askd where all thy beauty lies where all the treasure of thy lusty days spirit of beauty tarry yet awhile although the cheating merchants of the mart why are the lilies flecked with red there is blood on the river sand