UNIVERSITÉ DE SHERBROOKE

Dépôt GitHub :
github.com/ColinTr/IFT712_Projet_de_session

IFT 712 - Techniques d'apprentissage, projet de session

Comparaison de 6 méthodes de classification de données

Par Colin Troisemaine et Quentin Levieux

Résumé

Ce rapport de projet de session compare 6 classifieurs différents : AdaBoost, Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), Logistic Regression, Neural Networks, Perceptron, Ridge Regression, Support Vector Machines et Naive Bayes. Les points de comparaison principaux sont le type de données, le type de traitement des données requis, le temps d'exécution, la log loss et le score de test et d'entraînement. Nous passons également en revue quelques bonnes pratiques à appliquer et comparons nos résultats à d'autres personnes ayant utilisé le même ensemble de données provenant du challenge Kaggle "Leaf Classification".

Sommaire

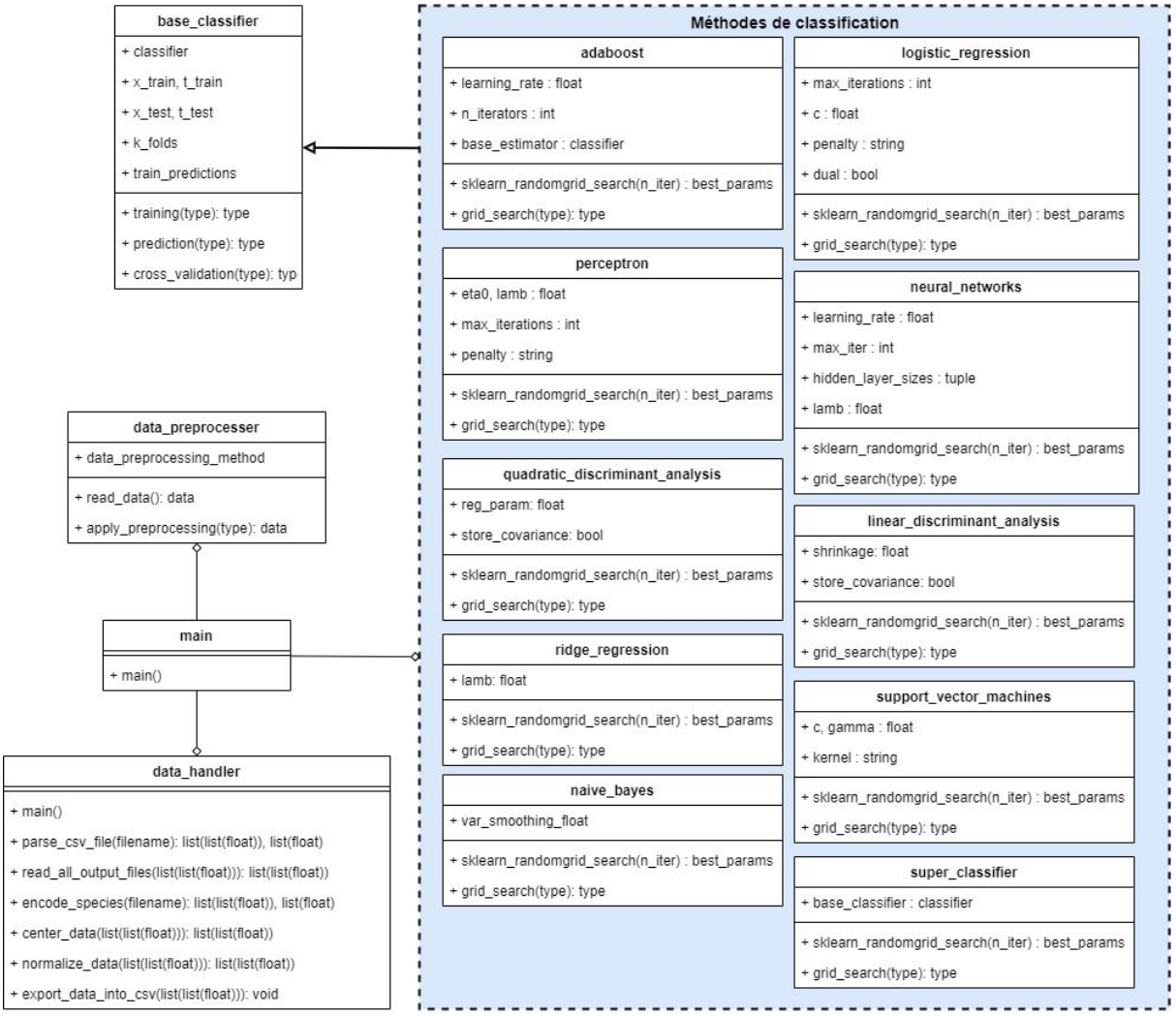
1. Introduction

Ce projet a pour but de tester au moins six méthodes de classification sur une base de données de Kaggle en utilisant la bibliothèque Scikit-learn. Comme proposé, nous avons concentré nos comparaison sur les résultats obtenus à partir de la base de données bien connue "Leaf Classification". Les bonnes pratiques de pré-traitement des données, de cross-validation et d'optimisation d'hyper-paramètres ont une place importante dans nos recherches.

Nous avons orienté notre choix vers des méthodes de classification de natures variées qui font différentes hypothèses / suppositions sur les données afin de pouvoir comparer leurs forces et faiblesses et de pouvoir conclure sur quels types de données elles sont le mieux adaptées. Dans ce rapport, nous allons donc comparer les méthodes suivantes : AdaBoost, Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), Logistic Regression, Neural Networks, Perceptron, Ridge Regression, Support Vector Machines et Naive Bayes (ainsi que notre propre implémentation du regroupement de classes).

2. Présentation du code

Dans l'optique d'écrire du code qui sera réutilisable dans des projets futurs et facilement adaptable, nous avons utilisé une organisation "modulaire" où chaque méthode de classification possède sa propre classe au sein de son propre fichier. Afin d'éviter de répéter du code que l'on retrouverait dans toutes les méthodes de classification, nous avons choisi de faire hériter nos méthodes par une classe "base_classifier" qui comporte par exemple les méthodes de cross-validation ou d'entraînement.



Enfin, le script permettant de lancer l'entraînement des modèles et la génération des résultats de nos méthodes de classification est intitulé *main.py*.
Voici la commande relative à son usage :

```
python main.py train_data_input_filepath output_filepath classifier grid_search data_preprocessing use_pca
```

Il est possible d'y spécifier plusieurs paramètres :

- *train_data_input_filepath* : Le chemin du fichier des données d'entraînement.
- *output_filepath* : Le dossier dans lequel seront stockés les nouveaux fichiers CSV.

- *classifier* : Le numéro du classifier
    - Si 0 est spécifié, tous les classifieurs seront lancés les uns à la suite des autres.
    - Sinon : 1=Neural Networks, 2=Linear Discriminant Analysis, 3=Logistic Regression, 4=Ridge, 5=Perceptron, 6=SVM, 7= AdaBoost, 8=Quadratic Discriminant Analysis, 9=Naive Bayes et 10=Class grouping
- *grid_search* : 0=pas de grid search, 1=grid_search
- *data_preprocessing* : 0=utilisation des données normales, 1=données centrées et normalisées par la déviation standard, 2=données centrées et normalisées par min/max
- *use_pca* : 0=pas de PCA, 1=on réalise une PCA sur toutes les données.

Cet ensemble de paramètres nous a ainsi permis de rapidement générer tous les résultats nécessaires à la comparaison de nos classifieurs.

Il est également possible de se servir uniquement de data_handler afin de traiter les données d'un fichier csv en les séparant en des ensembles de test, d'entraînement et d'entraînement centré et normé.
Cela en utilisant la commande suivante :

```
python data_handler.py train_data_input_filepath output_filepath
```

On peut y préciser le chemin du fichier des données d'entraînement et le dossier dans lequel seront stockés les nouveaux fichiers CSV.

3. Démarche scientifique et analyse des résultats

Note : Nous n'incluons parfois pas tous les classifieurs lors de la présentation des résultats afin de ne pas surcharger les graphiques car nous en avons sélectionné 9. Nous avons éliminé des classifieurs qui ne présentaient pas de résultats intéressants pour chaque point de comparaison.

Note bis : Ne disposant pas de données de test à proprement parler car le challenge de Leaf Classification n'offre que des données d'entraînement étiquetées, nous utiliserons souvent le terme de *données de test* qui désignera en réalité des *données de validation* que l'on aura séparées de nos données d'entraînement. L'idée générale reste la même.

3.1. Cross-validation

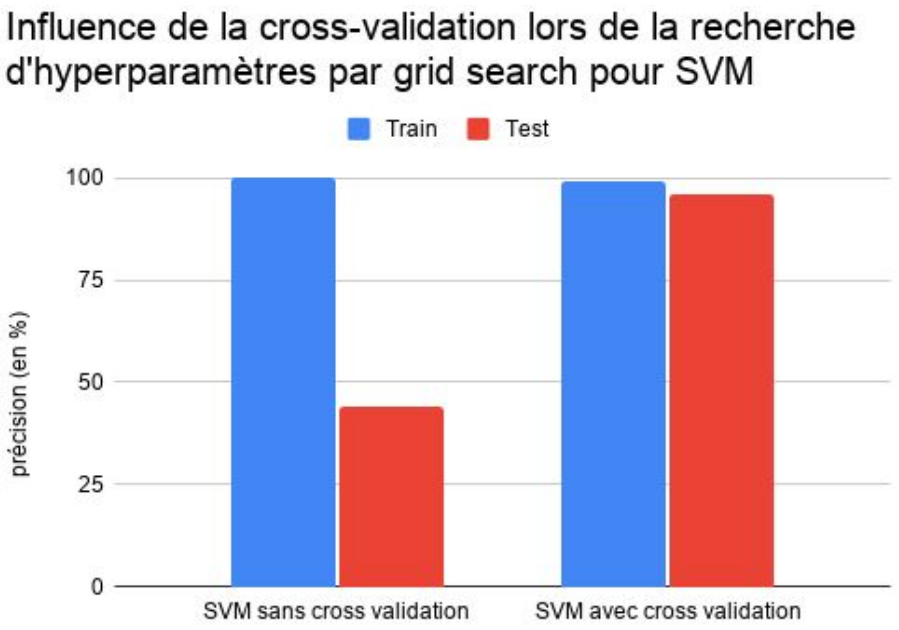
Apprendre les paramètres d'une fonction de prédiction et tester les résultats sur le même ensemble de données est une grave erreur de méthodologie : Cela mènerait rapidement à un sur-apprentissage puisque le modèle répéterait simplement les classes des

données qu'il a déjà vues et obtiendrait un score parfait lors du test. Le modèle ainsi produit serait alors incapable de généraliser et de prédire la classe de données qu'il n'a jamais vues. Une méthode très utilisée pour éviter ce phénomène est la cross-validation. Son fonctionnement est le suivant :

On commence par séparer nos données d'entraînement en *k* "folds". Le modèle est ensuite entraîné en utilisant *k - 1* folds et évalué avec la partie restante des données. La performance du modèle reportée par la *k-fold cross-validation* est alors évaluée en calculant la moyenne des erreurs de test de chacun des folds. Ce type d'évaluation des performances est plus coûteux en temps de calcul qu'une simple évaluation, mais a l'avantage de produire des résultats plus fiables et moins enclins au sur-apprentissage.

Dans notre projet, nous avons choisi d'utiliser une *Stratified k-fold cross-validation*. Son principe et son but sont les mêmes que pour une *cross-validation* classique. La différence réside au moment où l'on sépare les données en nos k - folds :
L'ensemble de données du problème de *leaf-classification* de Kaggle dispose de 990 données étiquetées d'entraînement pour 99 espèces. Cela nous laisse exactement 10 individus dans chaque classe que l'on doit ensuite répartir dans les données que l'on réserve au tout début de notre script pour calculer la précision du test et dans les données dont on se servira pour entraîner notre modèle. Ainsi, si on utilise un simple ShuffleSplit de scikit-learn pour répartir nos données au moment de la séparation test/train et à la répartition des folds de la cross-validation, il ne tiendra pas compte des classes des données et on risquera d'avoir des groupes qui ne possèdent aucun individu de certaines classes. Cela risque d'entraîner un sous-apprentissage et/ou un test final incomplet. Ainsi, pour prévenir ce problème, nous avons utilisé la technique du *Stratified k-fold cross-validation*. Scikit-learn en propose une implémentation via la méthode StratifiedKFold qui va prendre soin de conserver les mêmes ratios de classes au sein de chaque split de la cross-validation.

Afin de démontrer l'influence de la cross-validation, nous avons implémenté notre propre grid-search (nous introduisons ce concept par la suite) afin de déterminer les meilleurs hyperparamètres car scikit-learn propose une méthode de grid search, mais elle utilise forcément une cross-validation. Cela nous a permis de comparer le résultat de cette grid-search lorsque l'on utilise une cross-validation ou un simple entraînement de notre méthode sur nos données d'entraînement suivi d'une évaluation de la performance sur ces mêmes données :



Comparaison de la précision de la prédiction sur les ensembles de test et d'entraînement du classifieur SVM pour les données Leaf Classification standardisées (avec grid search)

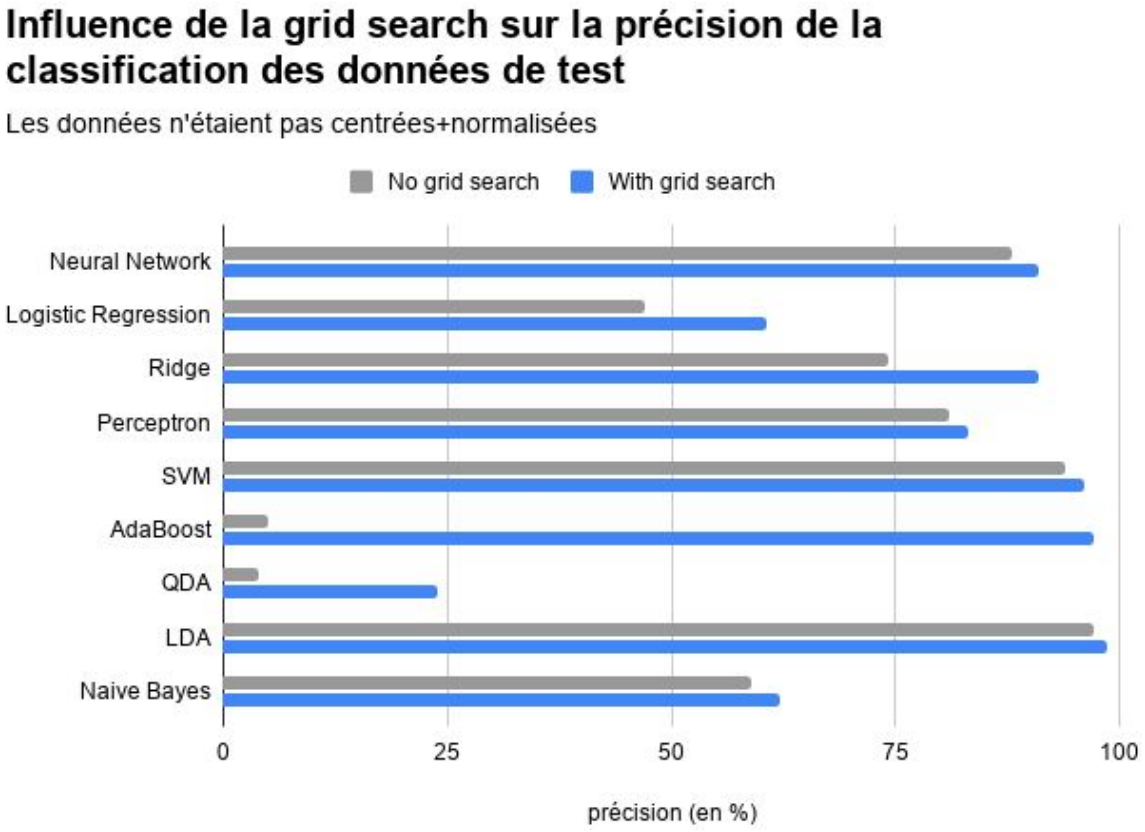
On peut très clairement voir que sans cross-validation, notre modèle obtient après une grid search un score de 100% de réussite à la classification, mais que la classification de ce modèle sur les données de test n'obtient que 44% de réussite. En revanche, si on utilise la cross-validation, le score d'entraînement baisse très légèrement (98 ± 1%), mais le modèle arrive dans ce cas beaucoup mieux à généraliser pour les données de test puisqu'il obtient un score de 96% de réussite !

Même si ce n'est pas une science exacte, la sélection du nombre *k* de folds a une influence sur les résultats. C'est pourquoi nous avons fait attention à ce choix : Nous disposons à la base de 10 individus pour chacune des 99 classes. On réserve 20% des données pour le test en faisant attention aux proportions des classes dans la sélection, ce qui nous laisse 8 individus par classe pour l'entraînement. Il est donc par exemple impossible de réaliser des folds avec un *k* = 10 tout en représentant toutes les classes dans tous les folds. Nous avons ainsi choisi en général des folds de 4 pour conserver 2 individus dans les folds de validation, ce qui est suffisant pour représenter plus ou moins fidèlement les classes.

3.2. Recherche d'hyper-paramètres

Bien que la cross-validation permette d'obtenir une estimation fiable de la performance d'un classifieur, seul il ne peut pas permettre d'optimiser notre classifieur. C'est pourquoi nous nous sommes servis de la technique du *grid search* qui permet d'automatiser la recherche d'hyper-paramètres dans le but d'obtenir la meilleure performance possible. Son fonctionnement général est le suivant : On définit une grille d'hyper-paramètres que l'on souhaite explorer. Par exemple pour le perceptron : 10 valeurs d'alpha entre 0 et 2 et 10 valeurs d'eta0 entre 0 et 1. On constitue alors une grille de 10 * 10 = 100 combinaisons

d'hyper-paramètres à explorer. Pour chacune de ces combinaisons, on va évaluer notre classifieur par cross-validation et à la fin on conserve les hyperparamètres qui nous ont permis d'obtenir le meilleur résultat.

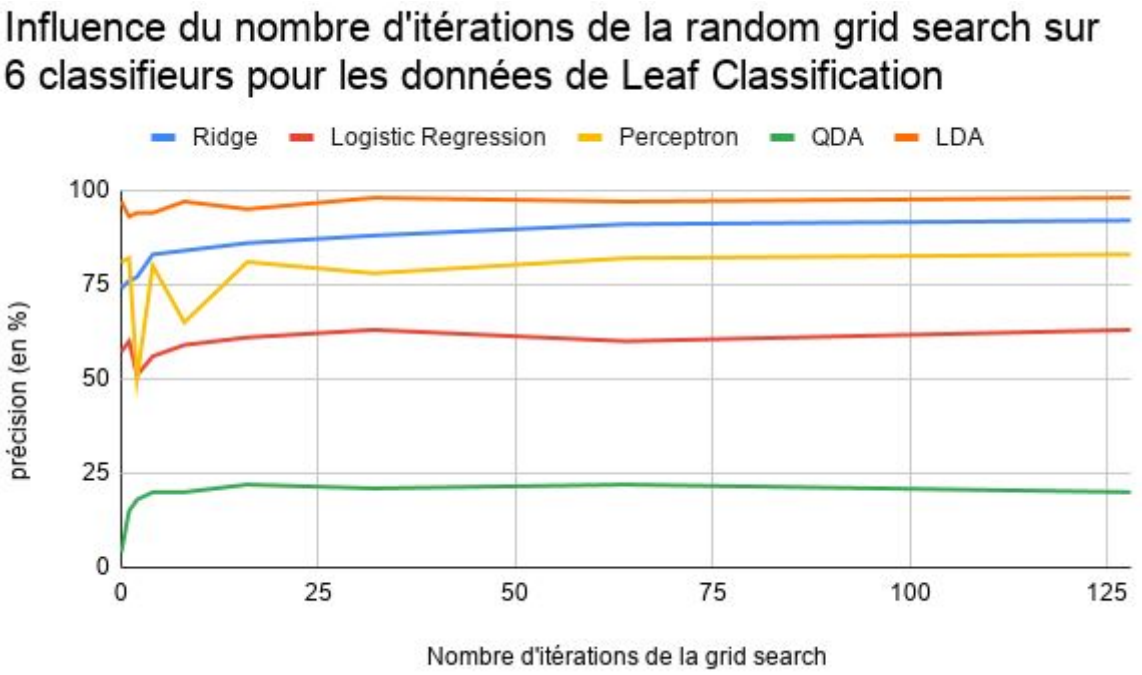


Effet de l'optimisation des hyperparamètres par grid search sur les données Leaf Classification non centrées et normalisées

On peut observer ci-dessus que l'optimisation des hyperparamètres par grid search est variable en fonction des classifieurs, mais a toujours un effet positif. Le cas d'Adaboost est particulièrement remarquable puisqu'il passe de 5% à 97% de précision. Cela est principalement dû au fait qu'Adaboost peut utiliser différents classifieurs de base (ou *weak learners*) qui présentent des performances très différentes.

Lorsque la performance d'un classifieur comme *Naive Bayes* reste relativement mauvaise même après une grid-search, on peut en conclure (et nous en reparlerons) que le problème réside autre part que l'optimisation des hyper-paramètres.

Outre notre implémentation manuelle de grid search, nous avons testé celle de scikit-learn nommée RandomizedSearchCV. Même si nous trouvons des hyperparamètres équivalents avec notre propre grid search, celle de scikit-learn comporte deux avantages majeurs : D'abord elle offre la possibilité d'utiliser tous les cœurs du processeur, ce qui accélère beaucoup l'exécution. Ensuite il est possible de préciser un nombre de combinaisons d'hyper paramètres que l'on souhaite tester (de base RandomizedSearchCV en testera 10). Ce paramètre propose ainsi un compromis entre temps d'exécution et qualité du résultat.



Effet du nombre d'itérations de la random grid search sur la qualité des prédictions des données de Leaf Classification non centrées et normalisées

Ci-dessus, on peut observer le pourcentage de classification correcte de 6 méthodes sur les données de test après avoir été optimisées par un nombre variable d'itérations de la randomized grid search de scikit-learn. Les résultats obtenus ont été réalisés sur une moyenne de 5 exécutions pour chaque point. Cela toujours sur les mêmes données de l'ensemble Leaf Classification (sans les avoir centrées et normées). Ce graphique nous permet de conclure qu'un nombre très élevé d'itérations n'est pas toujours utile pour améliorer les résultats. Un bon compromis entre temps et pourcentage de réussite semble se trouver vers 50 itérations.

Il est également utile de souligner que lorsque le nombre d'itérations est faible, on observe de grandes variations dans la précision du classifieur ainsi entraîné : Pour le perceptron, quand le nombre d'itérations de la grid search était inférieur à 16, on a pu observer des précisions de test qui pouvaient varier entre 35% et 80% pour un même nombre d'epochs sur le même classifieur et les mêmes données. Afin d'obtenir des résultats stables et interprétables, il faut donc un nombre minimal d'environ 30 itérations

Le perceptron varie plus que les autres classifieurs car il est très sensible aux variations de ses hyperparamètres. Dans cette grid search, nous cherchons à optimiser *alpha* et *eta0* pour le perceptron. *eta0* est la constante qui multiplie les mises à jour, elle a donc une importance cruciale sur la précision. On pourrait par exemple imaginer le cas de figure suivant : Si lors de l'entraînement, le perceptron initialise ses paramètres W relativement proches de la solution optimale et un *eta0* faible, il va rapidement converger. Cependant lorsqu'on le ré-entraînera après la grid search pour les meilleurs hyperparamètres trouvés, il pourrait s'initialiser plus loin de la solution et il ne s'en rapprocherait pas assez puisque *eta0* est petit, menant à de mauvais résultats sur les données de test.

3.3. Entraînement et test des méthodes sur les mêmes données

Nous espérons l'avoir fait transparaître dans la rédaction de ce document, mais préférons le préciser ici : lorsque l'on compare nos méthodes de classification sur un même graphique, nous utilisons toujours les mêmes données pour générer nos résultats. Il en va de même pour les autres paramètres que nous avons définis : méthode de normalisation des données, utilisation d'une ACP ou non et utilisation d'une grid search ou non. De cette manière les résultats sont toujours comparables les uns aux autres au sein d'un même graphique.

3.4. Traitements réalisés sur les données

3.4.1. Centrage et normalisation

Une étape de prétraitement incontournable est la normalisation des données. Elle a pour objectif de ramener toutes les dimensions de notre ensemble de données dans un même ordre de grandeur afin d'éviter de voir apparaître des problèmes d'échelles et d'ainsi limiter l'importance de certaines mesures / variables par rapport à d'autres lorsqu'elles sont trop grandes.

Nous avons choisi de comparer l'impact de deux formules de standardisation de données :

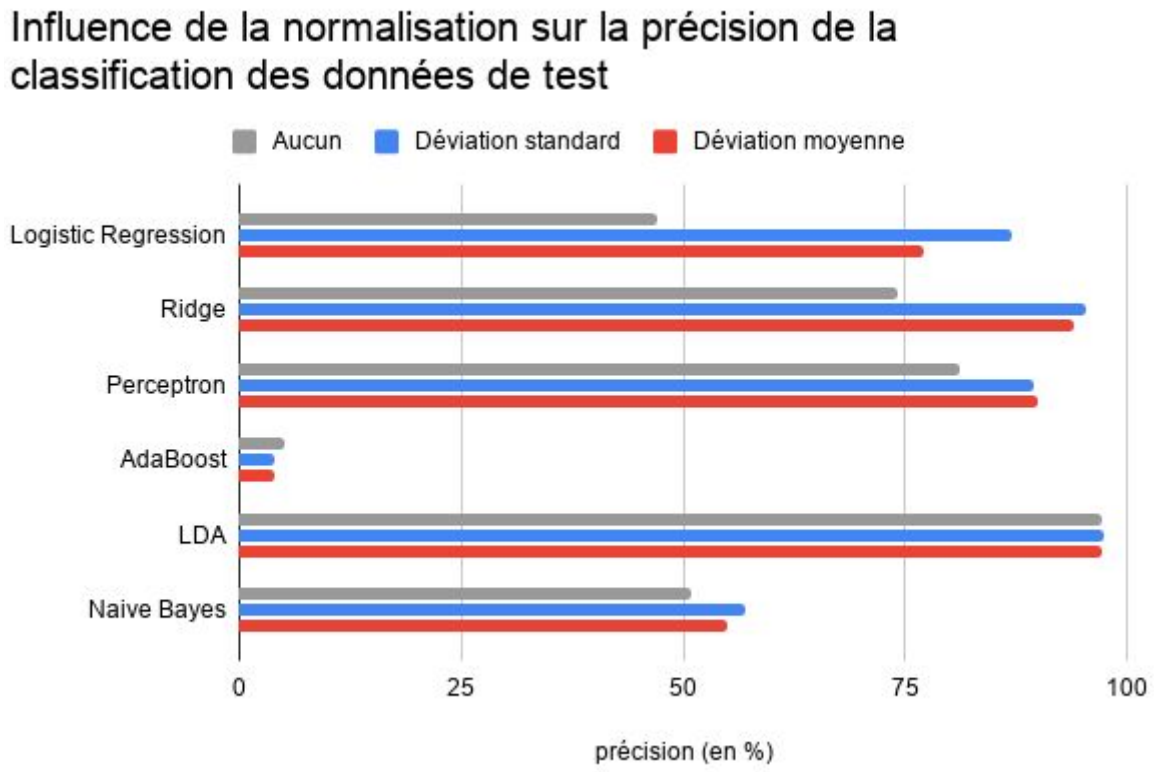
- La déviation standard : $nouvelle\ valeur = \frac{valeur\ actuelle - moyenne}{écart-type}$

Cela a pour conséquence de centrer les données en 0 et d'obtenir un écart-type égal à 1.

- La déviation de moyenne : $nouvelle\ valeur = \frac{valeur\ actuelle - minimum}{maximum - minimum}$

Cette seconde alternative a pour conséquence de centrer les données en 0 également, et d'obtenir une distribution comprise entre 0 et 1.

La moyenne, l'écart-type, le maximum et le minimum sont à calculer pour chaque variable. Dans le cas de l'ensemble de données "Leaf classification", nous en avions donc 192 à calculer, puisque nos données sont de dimension 192.



Comparaison de la précision des prédictions après déviation standard ou déviation moyenne sur les données de Leaf Classification non centrées et normalisées (sans grid search)

On peut observer ci-dessus que l'influence de la normalisation des données a toujours un effet positif sur la précision du classifieur plus ou moins important, quelle que soit la méthode de normalisation. Cependant, la précision de la classification à tendance à ne pas s'améliorer significativement pour AdaBoost qui est très mauvais sans grid search, le fait de standardiser les données n'est pas suffisant pour lui permettre d'améliorer ses résultats. À l'opposé, LDA obtient de très bon résultats sur les données dites "brutes", donc la standardisation n'affecte pas son résultat.

Pour finir, on remarque que le classifieur par régression logistique voit sa précision passer de 47% à 77% avec l'utilisation de la déviation moyenne et jusqu'à 87% avec l'utilisation de la déviation standard. Cela s'explique par le fait que ce classifieur a tendance à mieux fonctionner lorsque les différentes variables des données sont peu corrélées. En effet, si une des variables d'un des individus est une donnée plus grande que la majorité des autres données de cette variable, la déviation moyenne aura un effet moins notable et aura plus de mal à "séparer" ou répartir les données entre 0 et 1 que le déviation standard.

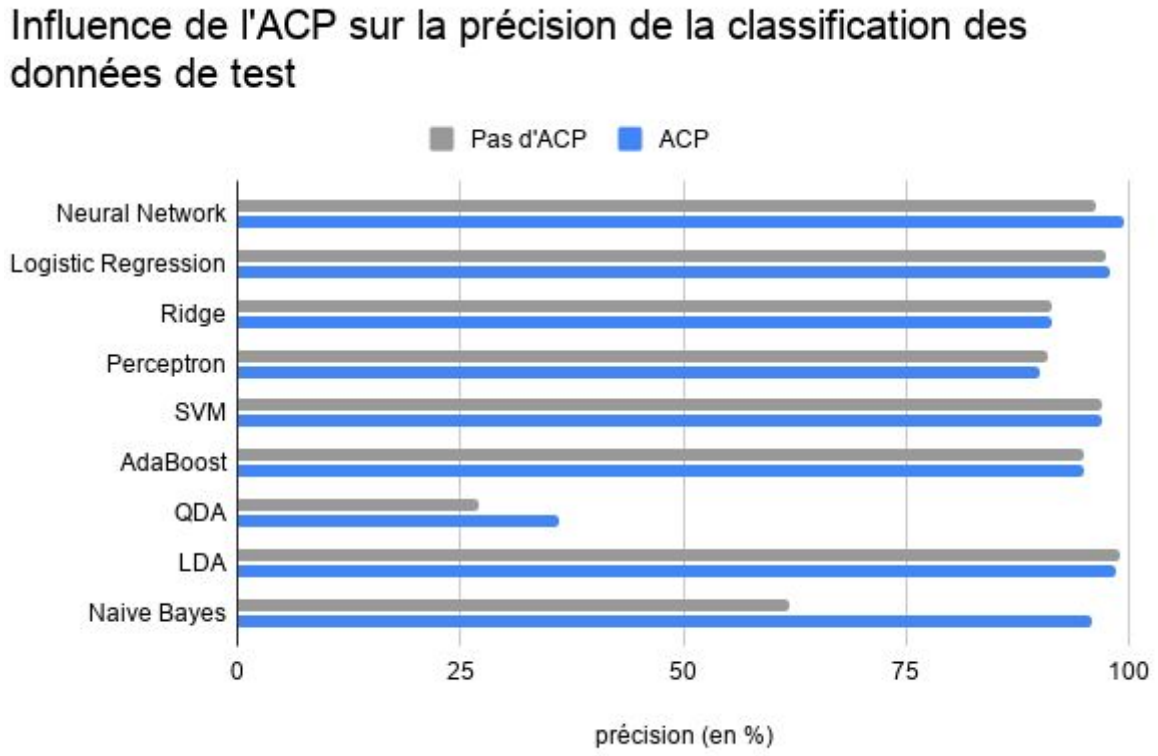
3.4.2. Analyse par Composantes Principales (ACP)

L'ACP est une méthode d'analyse de données. Elle fait partie des statistiques descriptives (à la différence des statistiques inférentielles) et étudie les variables qui sont liées entre elles, c'est-à-dire les variables corrélés. Elle a pour but de créer de nouvelles variables synthétiques appelées composantes principales qui sont une combinaison linéaire de certaines des variables initiales. Elle cherche ainsi à réduire le nombre de variables, ce

qui implique de passer dans un espace de dimension plus petite. En d'autres termes, l'objectif de l'ACP est de trouver un sous espace R^m de R^p avec $m \leq p$. Le sous espace est choisi de manière à perdre le moins d'information possible.

L'utilisation d'une ACP en apprentissage s'avère bénéfique pour deux raisons : Si on réduit suffisamment la dimensionnalité de notre ensemble de données, le temps d'entraînement se verra grandement réduit. Et elle est utile pour filtrer les ensembles de données trop bruitées (par exemple en compression d'image, chaque composant supplémentaire exprimera moins de variance et plus de bruit, donc représenter les données dans un espace plus petit en conservant les dimensions les plus importantes permet de conserver le sens des données en éliminant le bruit).

Nous avons donc utilisé l'implémentation de l'ACP proposée par Sklearn afin d'étudier ses effets sur les classifieurs que nous comparons :



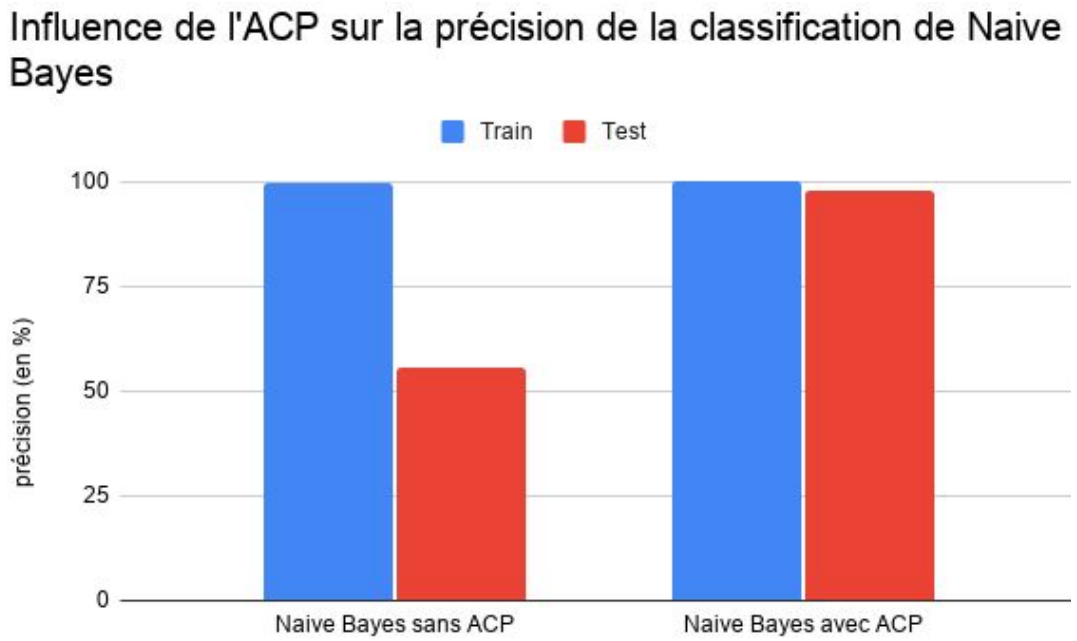
Effet de l'application d'une analyse par composantes principales sur la précision de la prédiction pour les données de Leaf Classification standardisées (avec grid search)

Ces résultats peuvent être obtenus en tapant la commande suivante :

```
python main.py data/raw/train/leaf-classification-train.csv data/processed 0 1 1 1
```

Pour appliquer cette ACP, nous nous sommes encore une fois servis de scikit-learn et de sa méthode PCA très facile d'utilisation. Sans toucher aux paramètres de base, nous sommes passés de 192 dimensions à 191, cela pourrait être associé à un échec au premier abord, mais la réalité est différente : 3 classifieurs présentent une amélioration notable de la précision de leur prédiction sur les données de test.

On remarque que parmis eux, Naive Bayes présente une très grande amélioration, intéressons-nous y donc de plus près :



Comparaison de la précision du classifieur Naive Bayes sur les données de test et d'entraînement sans et avec ACP (avec grid search et standardisation) sur les données de Leaf Classification

Cet histogramme est un clair exemple d'overfitting qui a été correctement corrigé, et c'était effectivement l'une des promesses de l'ACP sur les données en haute dimension ! Ainsi, même si une seule dimension a été supprimée par combinaison linéaire, le classifieur Naive Bayes a vu sa performance grandement augmenter. Pourquoi spécialement lui ? La réponse se trouve dans son nom : "Naive" implique ici qu'il suppose que les variables sont indépendantes. Et de toute évidence, deux variables ne l'étaient pas et ont été fusionnées en une par notre ACP, ce qui a suffit à Naive Bayes pour qu'il améliore grandement sa performance.

Bien que dans la plupart du temps faire une ACP est bénéfique, il s'avère que parfois elle puisse réduire les performances du modèle lorsque l'ensemble de données considéré possède une faible corrélation entre ses variables. De plus, l'ACP s'applique sur des données préalablement centrées-réduites, pour ne pas risquer de fausser le résultat. On dit alors qu'il s'agit d'une ACP normée. Enfin, elle est sensible à la présence de valeurs aberrantes, nous allons aborder ce point ci-dessous.

3.4.3. Données aberrantes

Nous savons que parfois, les ensembles de données peuvent contenir des données dites "aberrantes" qui ne sont pas représentatives des populations dont elles sont issues. Comprendre leur impact sur nos algorithmes ou bien les détecter et les éliminer permet bien souvent d'améliorer les performances des classifieurs utilisés.

C'est pour ces raisons que nous nous sommes penchés vers l'Isolation Forest, un algorithme de détection d'anomalies populaire qui modélise les données de façon à isoler les anomalies qui sont faibles en nombre et possèdent des attributs trop différents des autres points. La librairie scikit-learn en propose sa propre implémentation, IsolationForest. Un paramètre important que l'on retrouve dans la plupart des autres méthodes de détection de données aberrantes est la part de *contamination*. Elle estime le nombre de données aberrantes de notre ensemble de données (elle varie entre 0.0 et 0.5. Ainsi, si on la définit par exemple à 0.1, on choisira d'enlever 10% des données les plus aberrantes de notre ensemble).

Afin de mesurer l'impact de la suppression des données les plus aberrantes, nous avons choisi d'ajouter à la grid search de notre implémentation du classifieur de Ridge la recherche de la contamination qui nous permettrai d'obtenir le meilleur score. Ridge est une formule de type Maximum A Posteriori (MAP), il utilise ainsi une fonction de perte de type moindre carrés qui est donc particulièrement sensible aux données aberrantes.

```
Grid Search final hyper-parameters :
     best_learning_rate= 1e-09
     best_contamination= 0.0
Train accuracy : 99.8737%
Test accuracy : 93.9394%
```

Résultat d'une grid search sur la classification de Ridge avec les données de Leaf Classification standardisées

Comme on peut l'observer sur le résultat ci-dessus, notre grid search a déterminé que la meilleure précision de classification était obtenue lorsque la valeur de contamination renseignée dans IsolationForest était de zéro. Cela signifie que la suppression des données les plus aberrantes de notre ensemble de données ne menait pas à une amélioration de la performance, et cela quelle que soit la proportion éliminée.

On peut donc en conclure que l'ensemble de données de Leaf Classification ne possède pas de données aberrantes et que chercher à en supprimer ne mènera pas à une amélioration de la performance des classifieurs que nous avons testés. Attention, en revanche, cela ne signifie pas qu'il est inutile de chercher à en supprimer dans d'autres ensembles de données, comme nous l'avons dit c'est en général bénéfique !

Une seconde méthode performante pour détecter les valeurs aberrantes est l'utilisation du Z score, qui désigne la valeur d'une donnée après standardisation. La formule est donc la même que celle de standardisation :

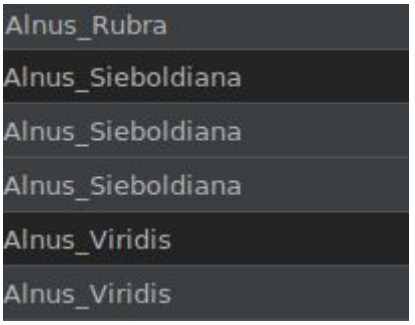
$$Z\ score\ =\ \frac{valeur\ actuelle - moyenne}{écart{-}type}$$

Ce score permet de mettre en avant les données qui sont très grandes et très petites au sein d'une même variable. Ce qui peut être interprété à partir de la formule ci-dessus, comme à combien d'écart-types à la moyenne de la variable une donnée est éloignée. Il est d'usage de dire que si un score un inférieur à -3 ou supérieur à 3, alors notre donnée est assez différentes des autres données de la variable.

Cependant le maniement du Z score dans l'objectif d'identifier les données aberrantes n'est valable que dans le cas de données suivant une distribution Gaussienne. Or nous allons démontrer par la suite que nos données ne suivent pas une distribution Gaussienne.

3.4.4. Regroupement de classes

Dans l'optique de simplifier le problème initial et d'essayer d'améliorer les performances de nos classifieurs, nous avons implémenté la technique de regroupement de classes. L'idée générale est la suivante :
L'ensemble de données de Leaf Classification présente de nombreuses espèces qui font parfois partie d'une plus grande famille.



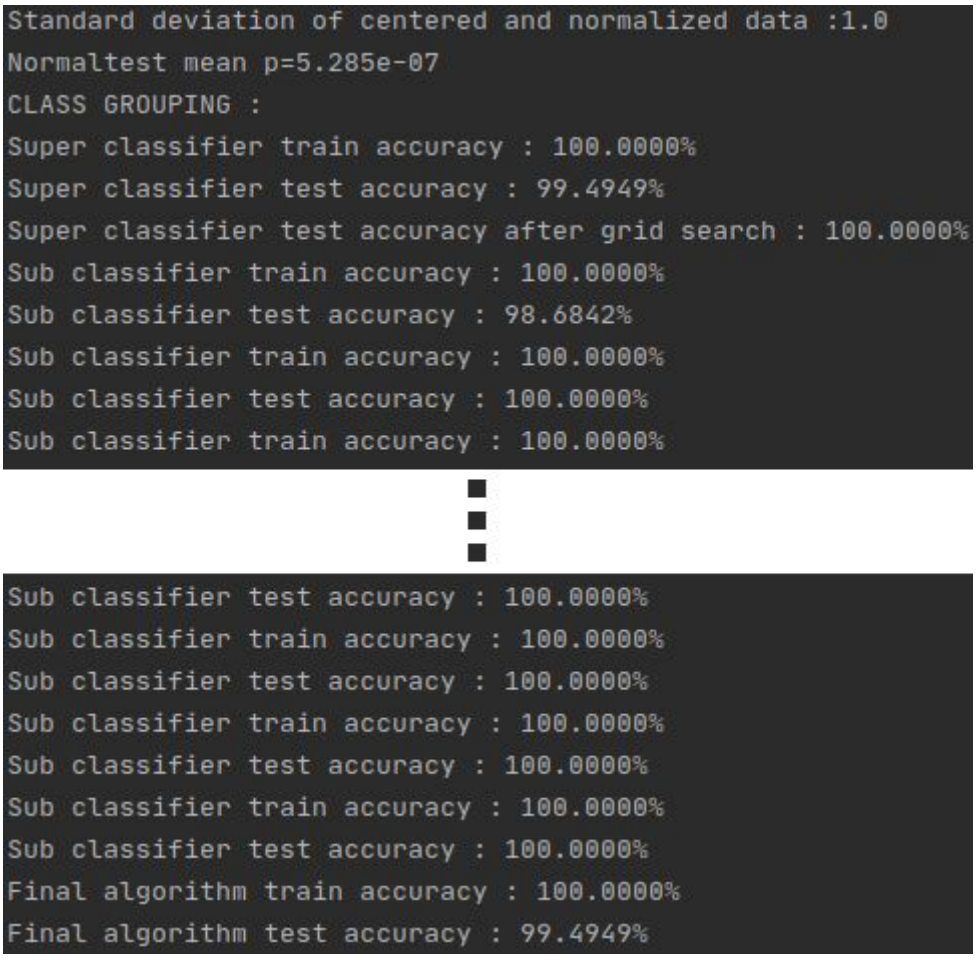
Exemple de sous-espèces d'une famille

Afin de simplifier notre problème, on décide donc de ne garder que la première partie du nom des espèces (par exemple Alnus pour l'image ci-dessus) et on entraîne un classifieur à reconnaître ces familles à partir des données initiales de Leaf Classification. Cela nous permet de manipuler un plus petit nombre de classes et d'avoir un plus grand nombre d'échantillons par classes.

L'étape suivante est donc d'entraîner autant de sous-classifieurs que l'on possède de grandes familles. Ainsi, on sélectionne uniquement les données associées à chaque grande famille et on entraîne pour chacune d'elle un nouveau classifieur qui sera capable d'étiqueter les différentes espèces d'une même famille.

Dans notre cas, on se retrouve avec 34 grandes familles (et donc autant de sous-classifieurs). Le nombre moyen d'échantillons par classe passe donc de 10 à 30 pour le classifieur principal. Avec cette méthode, il est possible d'utiliser n'importe quel classifieur et même d'utiliser différents classifieurs au sein du même algorithme. Nous avons choisi d'utiliser les Réseaux Neuronaux car ils sont flexibles et nous ont toujours donné des résultats très satisfaisants avec peu, voir aucun, traitements sur les données.

Pour lancer notre classifieur, on passe par la même commande que d'habitude puisque nous l'avons implémenté de manière à ce qu'il se comporte comme un classifieur classique.

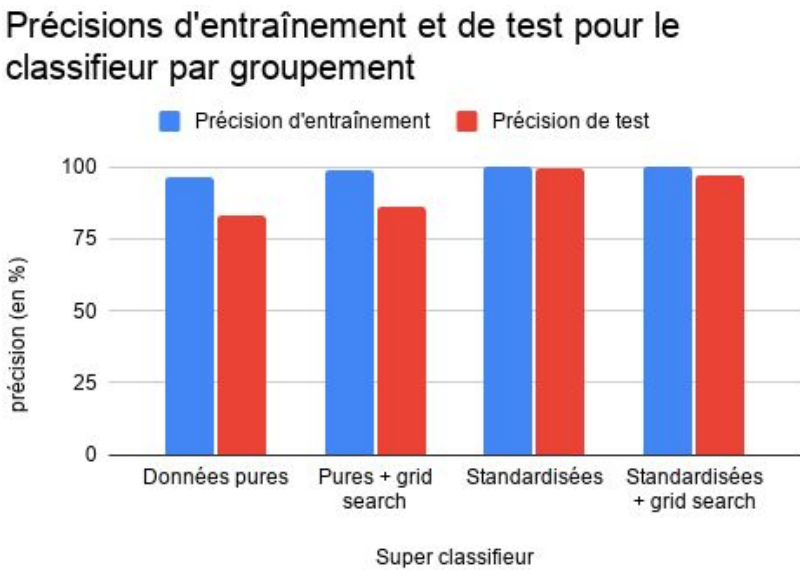


Exemple d'exécution complète du classifieur par regroupement de classes

Afin d'obtenir des résultats optimaux, nous avons appliqué une grid search sur chacun des classifieurs (le principal et les secondaires). Et nous avons normalisé les données par déviation standard.
Vous pouvez obtenir des résultats similaires (avec moins de verbose) en tapant :

```
python main.py data/raw/train/leaf-classification-train.csv data/processed 10 0 1 0
```

On peut noter que l'exécution ne prend que 7,5sec sur notre machine, auxquelles ont peut enlever environ 7 secondes de traitement des données. Il est donc efficace comparé aux autres algorithmes que nous avons testés. En revanche, si on lance une grid search sur cet algorithme, il va également appliquer une grid search sur tous les sous-classifieurs, ce qui se cumule à 35 grid search uniques. Même si nous avons réduit le nombre de combinaisons testées par ces grid search, le temps d'exécution augmente beaucoup et l'amélioration des résultats est faible voir absente comme le montre l'histogramme suivant :



Différentes précisions d'entraînement et de test pour les données Leaf Classification de l'algorithme de groupement de données

3.5. Hypothèses sur les données

Lorsque l'on applique des méthodes de statistiques paramétrique, l'hypothèse doit être faite que les données suivent une distribution connue, souvent Gaussienne. Certaines des méthodes de classification que nous comparons font donc cette hypothèse (QDA, LDA, Naive Bayes et Régression Logistique). Si cette hypothèse n'est pas respectée, on doit s'attendre à ce que ces classifieurs ne fonctionnent pas aussi bien qu'ils ne le devraient.

Une méthode simple de vérification de la "normalité" des données (c'est à dire vérifier si elles suivent une distribution Gaussienne) est tout simplement de réaliser un histogramme de la distribution des données selon une dimension. Malheureusement l'ensemble de Leaf Classification possède 192 dimensions et il n'est réalistiquement pas possible d'appliquer cette méthode. Nous avons donc utilisé la méthode NormalTest de la bibliothèque scipy.stats qui se sert du test K^2 de D'Agostino. Celui-ci calcule la symétrie et le kurtosis des données afin de déterminer si les données proviennent d'une distribution normale.

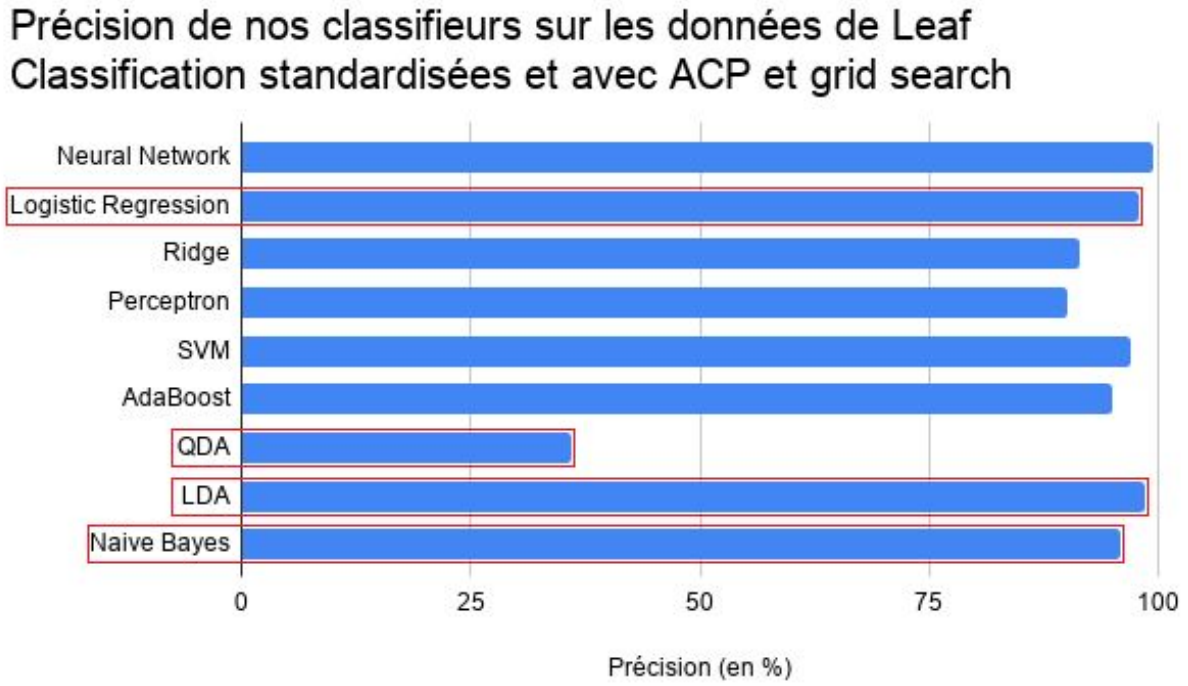
- La symétrie quantifie à quel point les données sont poussées à gauche ou à droite.
- Le kurtosis mesure la concentration des données à proximité ou à distance du centre de probabilité.

Ce test renvoie deux valeurs : statistic et pvalue. Ici seule la pvalue nous intéresse puisque l'on peut directement s'en servir pour interpréter le test : On définit un alpha et on pose notre test comme suit (en général, on pose un alpha de 0,05) :

- Si p <= alpha, on rejette l'hypothèse que nos données sont gaussiennes.
- Si p > alpha, on confirme que nos données sont gaussiennes.

Lorsque l'on applique ce test sur les données de Leaf Classification, on obtient un p moyen de 3e-06 environ (qu'elles soient standardisées ou non), on est donc très loin du seuil alpha

définit plus haut. Cette p value moyenne ne correspond pas directement à la probabilité de données à suivre une distribution gaussienne, mais c'est un bon indicateur pour nous aider à interpréter ce test statistique.



Voici pour rappel, voici les performances de nos classifieurs sur les données de Leaf Classification lorsqu'on applique une ACP et une grid search. Vous pouvez obtenir ces résultats en tapant :

```
python main.py data/raw/train/leaf-classification-train.csv data/processed 0 1 1 1
```

Les classifieurs entourés en rouge sont ceux qui émettent comme hypothèse initiale que les données suivent une distribution gaussienne (ce qui n'est pas le cas comme nous l'avons vu). Ainsi, mise à part l'exception de QDA qui n'a jamais réussi d'obtenir de bons résultats, on remarque que ces classifieurs sont loin d'être les plus mauvais malgré que leur hypothèse initiale ne soit pas vérifiée.

On peut donc en déduire que le fait que leur hypothèse ne soit pas vérifiée ne signifie pas qu'ils ne pourront pas obtenir des résultats satisfaisants, mais simplement qu'ils ne fonctionneront pas au maximum de leur capacité (ils devraient mieux fonctionner sur des données gaussiennes).

3.6. Sanity checks

3.6.1. Une initialisation aléatoire donne une loss maximale

Nous avons choisi d'utiliser la log loss comme métrique pour évaluer nos classifieurs. Nous pouvons également nous en servir pour démontrer que l'initialisation aléatoire des paramètres nous donne un loss maximale. La log loss est également l'entropie croisée, soit :

$$E_D(w) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn}.ln.y_{w,k}(x)$$

Si l'initialisation est aléatoire, alors la probabilité doit être égale pour chaque classe.Nous utilisons la base de données Leaf classification, nous avons donc 99 classes. On peut donc dire :

$$E_D(w) = -\frac{1}{N}\sum_{n=1}^{N} ln(\tfrac{1}{99}) = ln(99) = 4,59$$

Prenons l'exemple avec le réseau de neurones, puisque nous utilisons le MLPClassifier de sklearn il n'est pas possible de connaître la loss à l'initialisation car le modèle initialise ses poids et s'entraîne dans la même fonction. Cependant, en fixant l'attribut maximum d'itération à 1 (on ne peut pas mettre 0) , nous pouvons connaître la log loss après 1 itération qui est très proche de la log loss initiale :

```
Train accuracy : 2.5253%
Train log loss =  4.486123308873246
Test accuracy : 2.5253%
Test log loss =  4.507222407354273
```

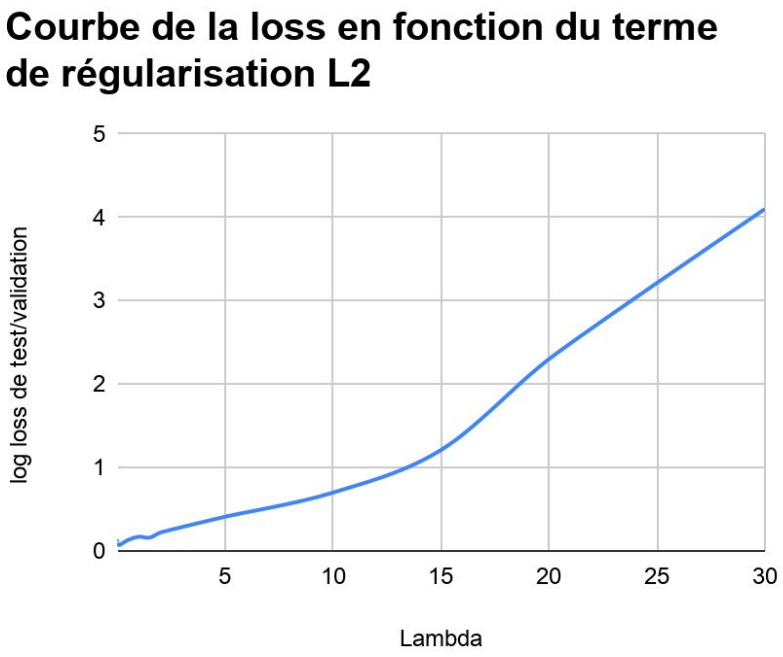
Après avoir effectué une itération, on peut donc calculer notre loss de test et d'entraînement, et comme prévu on obtient des valeurs très proches de la valeur théorique calculée. On peut donc supposer qu'à l'itération 0 avec des poids aléatoires nous avions une log loss légèrement plus haute et proche de 4,59 et ainsi que notre modèle était correctement initialisé de façon aléatoire.

3.6.2. Augmenter la régularisation augmente la perte

Nous savons que la loss s'exprime de cette façon :

$$Loss = E_D(w) + lambda \times R(w) \; avec \; R(w) > 0$$

Donc de façon générale, si le classifieur est bien fait et que l'on augmente lambda alors la loss augmente. Nous avons par exemple effectué ce test sur notre classifieur de réseaux neuronaux sur les données Leaf Classification préalablement standardisées.

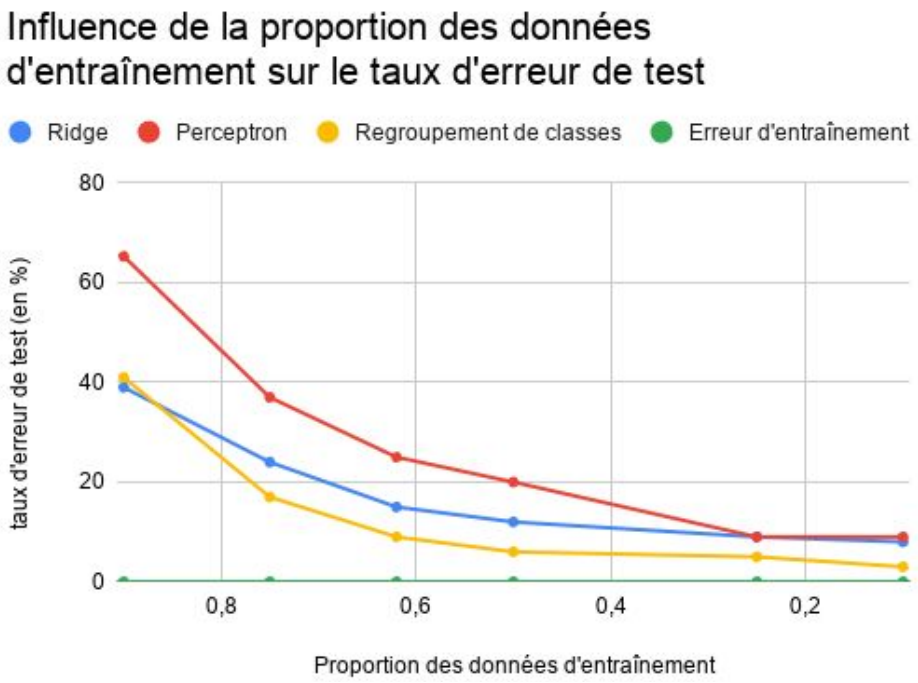


Courbes de la loss en fonction du terme de régularisation L2 pour la classification par Réseaux Neuronaux sur les données Leaf Classification normalisées par déviation standard

On observe que la courbe croît lorsque lambda augmente, on peut alors affirmer que le classifieur semble bien fonctionner.

3.6.3. S'assurer que l'on peut over-fitter sur un petit nombre de données

Si on considère un classifieur correctement paramétré et optimisé, il devrait être capable de sur-apprendre sur un faible nombre de données. La cause est évidente : Ayant un petit échantillon de données sur lesquelles il peut apprendre, il émettra des hypothèses quant à la nature des frontières de décision qui ne seront pas capables de généraliser à un plus grand nombre de données de test. Si notre modèle n'est en revanche pas capable de sur-apprendre, c'est qu'il y a un problème de paramétrage ou bien d'implémentation.

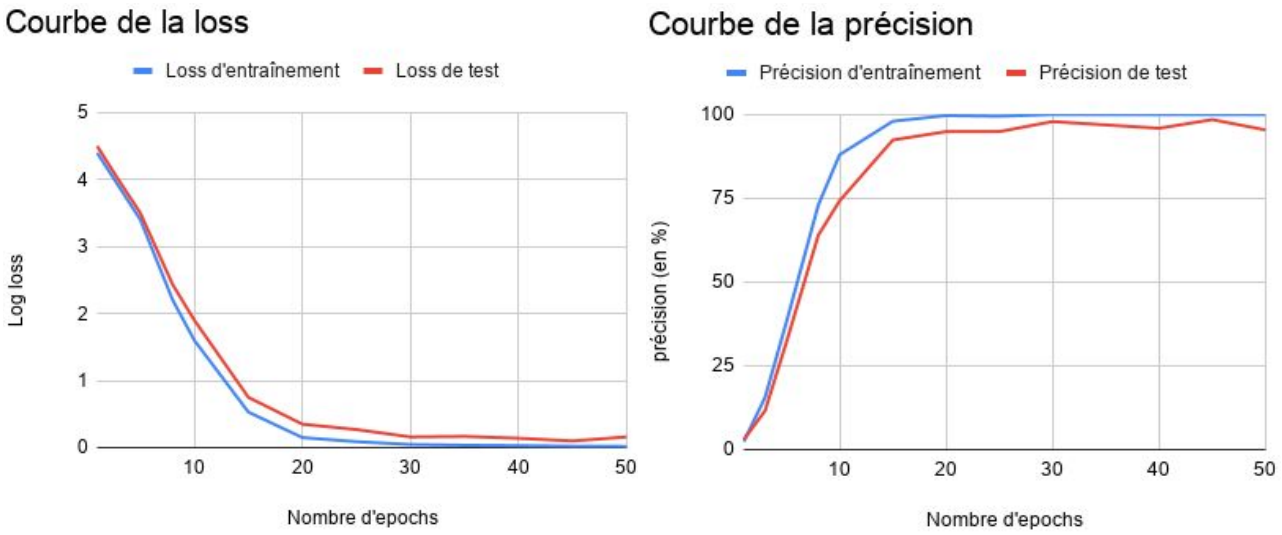


Démonstration de l'influence de la part des données réservées au test sur le taux d'erreur de 3 classifieurs sur les données de Leaf Classification standardisées

Lors du report des résultats des exécutions de notre algorithme pour générer les résultats ci-dessus, nous avons observé qu'après suffisamment d'epochs, nos trois classifieurs obtenaient toujours une loss d'entraînement de 0 et une précision d'entraînement de 100%. Ils ont donc correctement modélisé les données d'entraînement quelle que soit la proportion qui leur était donnée. Mais lorsque celle-ci était trop petite, il leur était impossible de généraliser pour les données de test. Cela prouve encore une fois que nos algorithmes fonctionnent correctement.

3.6.4. Visualiser les courbes d'apprentissage et de validation

Les courbes d'apprentissage sont un moyen de diagnostiquer le biais et la variance d'un modèle dans le cadre de l'apprentissage supervisé. Afin de vérifier que nos modèles étaient correctement implémentés, nous avons parfois été amenés à dessiner ces courbes. Voici un exemple de courbes que nous avons obtenues :



Courbes de la loss et de la précision de la classification par Réseaux Neuronaux sur les données Leaf Classification normalisées par déviation standard

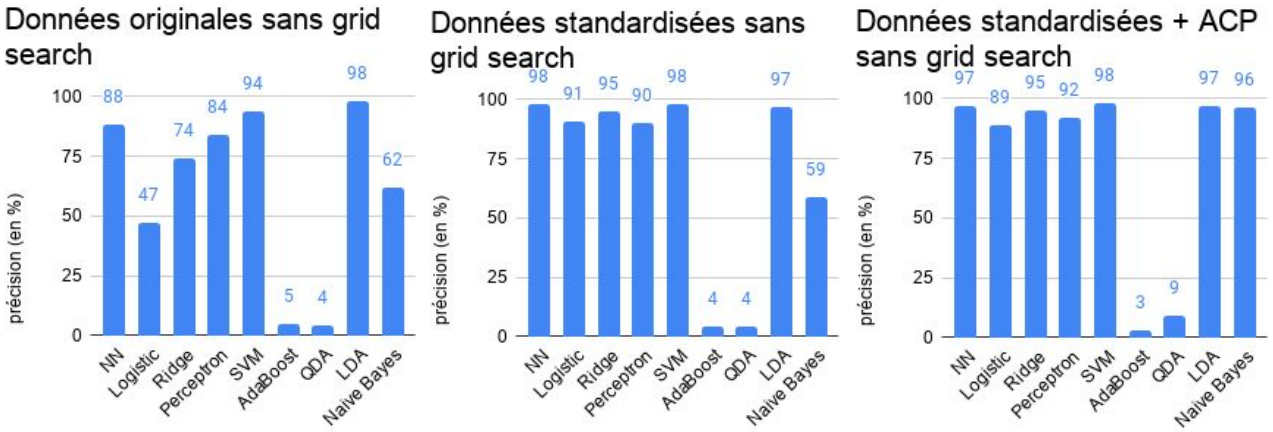
L'interprétation générale que l'on peut en faire est la suivante : la précision des données d'entraînement est toujours légèrement plus haute que celle des données de test, mais pas trop. Ce qui montre que le réseau semble avoir bien appris et aura de bonnes chances de pouvoir généraliser sur des données de test. Si on détaille un peu plus notre interprétation, on peut commencer par remarquer que lorsque le nombre d'itérations (*epochs*) de notre modèle est trop faible, il n'a pas le temps de converger vers une solution correcte, ainsi la loss est haute et la précision est basse pour les ensembles de d'entraînement et de test, ce qui est tout à fait naturel avec l'initialisation aléatoire des poids (*weights*) que l'on trouve dans un réseau neuronal.

Un autre élément important que l'on peut déduire à partir de ces courbes est le nombre nécessaire d'epochs avant que l'algorithme ne s'améliore plus. Ici, vers 30 epochs, la précision de test se stabilise et on ne remarque plus d'amélioration significative. Déterminer rapidement ce nombre d'itérations minimal est très intéressant au cours du développement d'un tel projet, car une fois que l'on utilise une grid search et que l'on exécute notre projet un grand nombre de fois, le temps gagné s'accumule très vite !

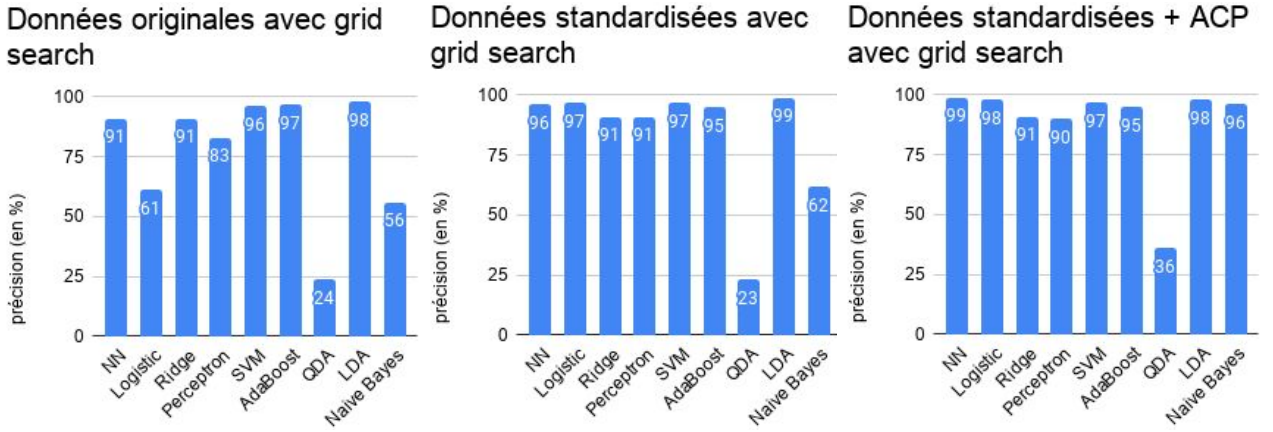
Les courbes ci-dessus ont été obtenues avec les paramètres de base du Réseau Neuronal proposé par scikit-learn et sont un bon exemple d'un classifieur correctement paramétré. Mais différents types de courbes peuvent apparaître et il est important de savoir les interpréter afin de rapidement identifier la source du problème. Le taux d'apprentissage est généralement la première des variables à regarder.

3.7. Comparaison des classifieurs

3.7.1. Comparaison des classifieurs sélectionnés



*Histogrammes des % de précision des classifieurs après différents traitements des données (**sans** grid search et sur les données de Leaf Classification)*



*Histogrammes des % de précision des classifieurs après différents traitements des données (**avec** grid search et sur les données de Leaf Classification)*

On remarque que le classifieur Naive Bayes est le seul à ne pas présenter d'amélioration de sa précision de test lorsque les données sont normalisées ou qu'une grid search est appliquée afin d'optimiser les hyper-paramètres. Cela est principalement dû au fait que sa précision d'entraînement est quasiment toujours de 100% (ce que les histogrammes ne présentent ici) pour une précision de test d'environ 60%. Cela révèle que sans ACP, Naive Bayes sur-apprend quelque soit le scénario pour notre ensemble de données. Il est donc dangereux d'essayer d'appliquer Naive Bayes lorsque les variables ne sont pas toutes indépendantes.

En revanche, si les données de l'ensemble considéré sont bien indépendantes, c'est une méthode intéressante car sans optimisation d'hyper paramètres il arrive à obtenir de très bons résultats rapidement :

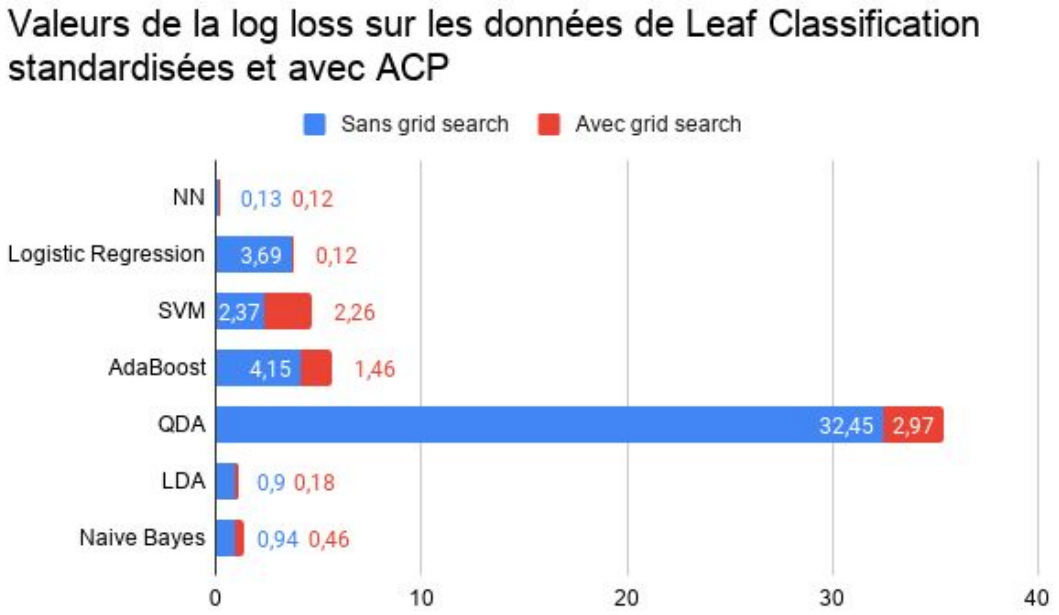
```
NAIVE BAYES :
Train accuracy : 100.0000%
Test accuracy : 95.9596%

real    0m7.505s
user    0m7.745s
sys     0m0.911s
```

Précision et temps d'exécution de Naive Bayes sur les données de Leaf Classification sur lesquelles une ACP a été appliquée

On peut enlever 7,1 sec de temps d'exécution que le traitement des données occupe en moyenne, ce qui nous donne 0,5 sec d'entraînement et de prédiction pour obtenir d'excellents résultats.

Afin de comparer plus précisément les performances des classifieurs que nous considérons, on peut prendre en compte la log loss résultante de la prédiction. Cette dernière n'est pas nécessairement utile / interprétable toute seule, mais couplée au pourcentage de précision elle permet de départager les méthodes de classification. En effet, si on imagine deux algorithmes qui doivent décider de classer une donnée entre 2 classes. Si l'un donne une probabilité pour la classe correcte de 0.51 et l'autre méthode donne 0.99, les loss respectives seront -log(0.51)=0.67 et -log(0.99)=0.01. Les deux méthodes ont toutes deux correctement classé la donnée, cependant la seconde a été meilleure puisqu'elle était bien plus certaine de son choix que la première, ce que l'on peut déduire à partir de sa log loss bien plus proche de zéro.



Log losses des 6 classifieurs capables d'estimer la probabilité de chaque classe de sortie à la prédiction sur les données de Leaf classification standardisées et avec ACP

Sans surprise, les log loss de nos méthodes de classification sont meilleures (c'est à dire plus proches de 0) lorsqu'elles ont été optimisées par grid search. On remarque que les

méthodes qui obtenaient déjà de bons résultats n'ont pas vu d'amélioration significative sur leur log loss, tandis que les méthodes plus impactées voient une différence plus grande.

Les méthodes de Neural Networks, Support Vector Machines et Perceptron sont trois méthodes qui obtiennent également des résultats très satisfaisants sur les données de Leaf Classification sans aucun traitement des données et aucune optimisation de leurs hyperparamètres. Cela s'explique certainement par le fait qu'ils n'émettent pas d'hypothèses sur les données comme la normalité ou l'égalité des matrices de variances-covariances, ce qui les rend tous les trois applicables sur un grand éventail de types d'ensembles de données.
Quelques éléments sont cependant à prendre en compte (le terme Réseau de Neurones désigne aussi les Perceptrons puisqu'ils sont un sous ensemble des réseaux neuronaux) :

- Les SVM ont besoin d'un nombre d'échantillons supérieur au nombre de caractéristiques des individus pour bien performer. Mais comparé aux réseaux neuronaux, la quantité requise par les SVM n'est pas si grande car ils vont utiliser des batchs successifs de données pour s'entraîner.
    - Il est possible d'augmenter artificiellement le nombre de données à notre disposition en ajoutant des modifications à nos données (bruit gaussien, différentes transformations, …).
- Les Réseaux de Neurones ont tendance à être bloqués par des minimums locaux, ce qui impliquerait d'avoir du mal à généraliser. Des méthodes telles que le momentum permettent d'atténuer ce problème.
- Les Réseaux de Neurones peuvent sur-apprendre s'ils sont entraînés pendant trop longtemps, un problème que les SVM n'ont pas.
- Puisque les Réseaux de Neurones utilisent une descente de gradient, ils sont sensibles à l'initialisation aléatoire des poids. De ce point de vue les SVM sont plus robustes et garantissent une convergence à un minimum global quelle que soit leur configuration initiale.

Enfin si on compare la log loss de notre NN et de notre SVM, on peut conclure que notre NN a été finalement bien meilleur que notre SVM dans ses prédictions de classes puisqu'il a obtenu une log loss de 0,12 tandis que SVM obtient 2,26 lorsque les données de Leaf Classification étaient standardisées et avec ACP.

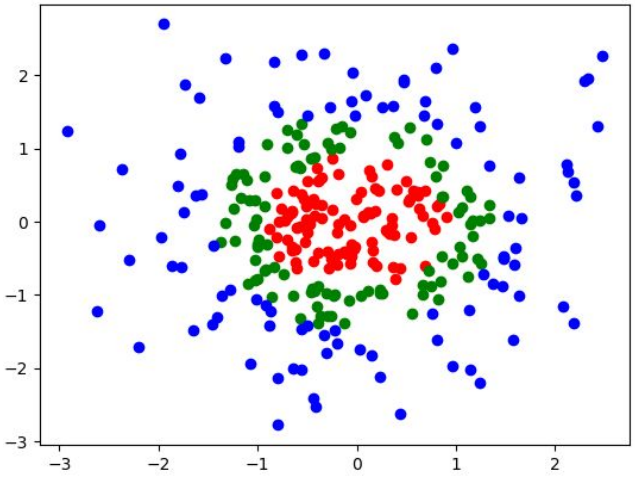
Concernant le classifieur Ridge, c'est un classifieur qui traite le problème comme une régression à plusieurs sorties et se base sur Maximum A Posteriori puisqu'il utilise la régression de Ridge. Étant un modèle linéaire, le classifieur de Ridge a l'avantage d'être moins sujet au sur-apprentissage qui touche les méthodes non-linéaires. Et comme les problèmes de haute dimensionnalité sont généralement linéairement séparables, cela explique pourquoi Ridge obtient de bons résultats sans aucun prétraitement, et de très bons résultats une fois que ses hyper-paramètres ont été optimisés.

Le classifieur par régression logistique fait partie des classifieurs qui ont obtenu des résultats médiocres lorsqu'ils n'étaient pas optimisés et que les données n'étaient pas standardisées. En effet ce classifieur est sensible à l'échelle des variables au moment de son entraînement, c'est pourquoi il est important d'appliquer une normalisation (préférablement par déviation standard qui est moins sensible aux données aberrantes que la déviation moyenne).

AdaBoost est un cas assez spécial, il est difficile de le comparer à d'autres méthodes de classification car ses résultats dépendent directement du *weak learner* ou *weak classifier* qu'il utilise (ou même la combinaison linéaire de weak learners). Parmi les learners que nous avons comparés lors de la grid search d'optimisation de notre AdaBoost, c'est le plus souvent l'ExtraTreesClassifier qui a été choisi. AdaBoost présente l'avantage d'améliorer les performances du *weak learner* utilisé. En revanche, il est très sensible aux données bruitées et aberrantes (ce que Leaf Classification ne possède pas). Pour l'implémenter efficacement, il faut donc avoir une bonne compréhension de ses prérequis et de quel *weak learner* entraîner pour qu'il converge vers un *strong learner* qui sera finalement moins sujet au sur-apprentissage et généralisera très bien sur des données inconnues (source : Understand AdaBoost and Implement it Effectively by Naseem Sadki).
Notre optimisation d'AdaBoost a obtenu un score de 95% de classification sur les données de test, ce qui est tout à fait correct. Mais la log loss est de 1.46, ce qui est relativement élevé comparé à notre log loss de 0.9 sur notre LDA non optimisée par exemple. Cela est certainement dû aux *weak learner* que nous avons comparés qui n'étaient peut être pas les plus adaptés ou optimisés pour notre problème de classification.

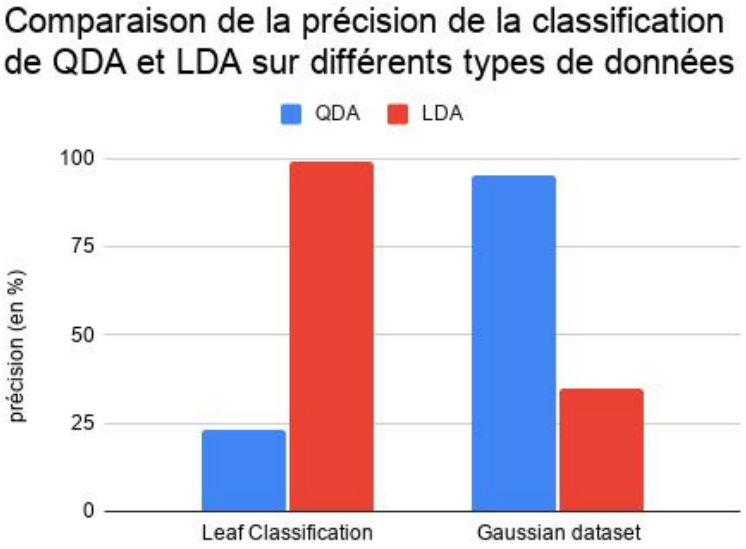
La différence de performance observée entre la Quadratic Discriminant Analysis et la Linear Discriminant Analysis est au premier abord surprenante, car la QDA est considérée comme supérieure à la LDA puisqu'elle ne fait pas de suppositions quand à la forme de la frontière de décision et est capable d'apprendre des frontières quadratiques. Pour cela elle est en général plus flexible. Mais attention, cela ne signifie pas que LDA obtient des résultats inférieurs à ceux de QDA dans tous les cas ! Ceux que nous avons obtenus en sont d'ailleurs la preuve.
James et al. offre plusieurs scénarios différents dans son livre "An Introduction to Statistical Learning" (2013) où QDA est meilleur que LDA et inversement. Dans l'un d'eux, il générait ses données à partir d'une t-distribution de la loi de student qui a une forme similaire à une distribution normale, mais qui a tendance à générer des valeurs très éloignées de la frontière de décision. LDA a donc sous performé car son hypothèse de normalité des données n'était pas respectée. QDA a également sous performé, mais dans une mesure bien plus grande (pour les mêmes raisons de non normalité des données). C'est certainement dans ce cas de figure que nous nous trouvons : Les données de Leaf Classification ne respectent pas l'hypothèse de la normalité, mais la frontière de décision reste toujours linéaire, c'est pour cela que LDA est capable de toujours produire des résultats acceptables tandis que QDA est bien plus affecté et sous performe énormément.
Un dernier élément de réponse concernant la sous performance de QDA : Comme QDA calcule des matrices des variances-covariances pour toutes les classes et pour toutes les variables à partir d'un faible nombre d'individu dans chaque classe (environ 7 au moment de la cross validation), il produit des matrices inexactes et des frontières de décision irréalistes, d'où sa très grande log loss.

Afin de vérifier que notre hypothèse est la bonne, nous avons sélectionné sur Kaggle un nouvel ensemble de données qui a été généré à l'aide de scikit-learn à des fins académiques. Cet ensemble possède 3 classes distinctes en 2 dimensions et 1500 échantillons en tout. Afin de nous assurer qu'il possède bel et bien des données suivant une distribution gaussienne, nous les avons affichées :



Nuage de points d'un échantillon de 20% de l'ensemble de données gaussien colorié par classes

Une fois que les points sont coloriés par classes, il est facile de se persuader que la distribution est gaussienne. À présent calculons les performances de QDA et LDA sur ce nouvel ensemble de données :



Influence du type de données sur la précision de la classification de QDA et LDA (les données ont été standardisées et une grid search a été utilisée)

Comme prévu, les performances de QDA augmentent beaucoup et il performe comme prévu. En revanche, les performances de LDA décroissent beaucoup. On s'attend plutôt à ce que ses performances soient similairement bonnes puisque l'hypothèse des données issues d'une distribution gaussienne est maintenant respectée !

En réalité, la LDA émet deux hypothèses : la normalité des données et l'homoscédasticité qui stipule que les matrices de variances-covariances sont identiques d'un groupe à l'autre. Calculons donc les matrices de variances-covariances de nos trois groupes à l'aide de la fonction corrcoef de numpy :

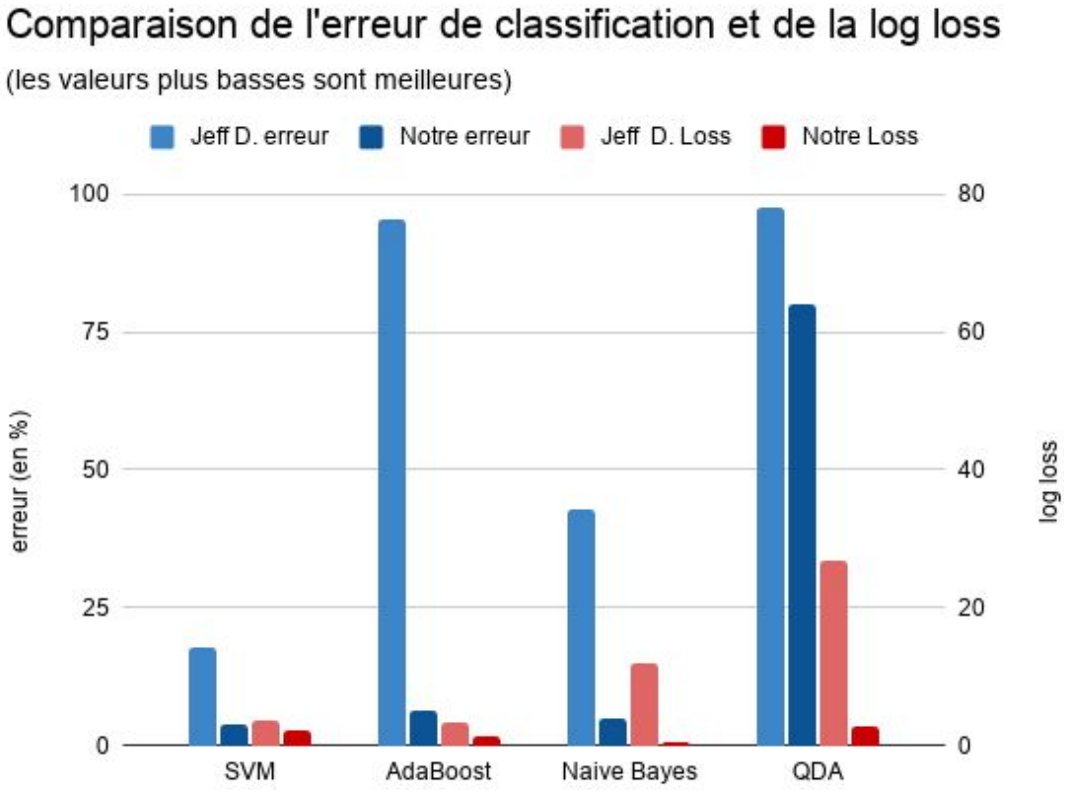
```
[[1.         0.06633432]
 [0.06633432 1.        ]]
[[ 1.         -0.10789333]
 [-0.10789333  1.        ]]
[[ 1.         -0.02987303]
 [-0.02987303  1.        ]]
```

Matrices de variances covariances entre les deux dimensions des trois groupes d'individus de l'ensemble de données gaussiennes

Les variances sont de 0.06 entre les 2 dimensions du premier groupe, elles sont de -0.1 entre les dimensions du second groupe et de -0.03 entre les dimensions du troisième groupe. L'homoscédasticité n'est donc pas respectées, ce qui explique que les performances de LDA diminuent drastiquement tandis que celles de QDA ne sont pas affectées puisque ce classifieur ne fait pas cette hypothèse et assume que chaque classe possède sa propre matrice de variances-covariances.

3.7.2. Comparaison à d'autres résultats du challenge "leaf classification"

Nous avons souhaité comparer la justesse de nos classifieurs avec d'autres personnes ayant posté leurs résultats du challenge Leaf classification sur Kaggle. Nous avons commencé par le Notebook qui a été le plus vu et apprécié. Il s'agit de celui de Jeff Delaney (10-classifier-showdown-in-scikit-learn). Parmi les 10 classifieurs qu'il a implémentés, nous en avons pu en comparer 4 que nous avions également implémentés.



(nos données ont été standardisées et une grid search a été utilisée ainsi qu'une acp)

Nos scores d'erreur sont bien meilleurs que ceux de Jeff D et nos loss sont bien plus faibles que les siennes. Pourtant nous avons utilisé les mêmes classifieurs provenant de la librairie scikit-learn. L'explication de ces différences est simple : Il n'a pas effectué de pré-traitement sur ses données comme la standardisation et/ou l'utilisation d'une ACP. Et le facteur le plus important est la recherche des hyper-paramètres permettant aux classifieurs une précision optimale, avec notamment l'utilisation de la grid-search qu'il n'a pas utilisée.

Si nous observons son classifieur AdaBoost, nous remarquons que c'est avec lui qu'il y a le plus grand écart dans les résultats. Puisqu'il n'a rien paramétré et qu'AdaBoost est très sensible aux hyper-paramètres base_estimator et n_estimators du classifieur de scikit-learn. En effet AdaBoost utilise par défaut l'apprentissage par arbre de décision (DecisionTreeClassifier) ainsi qu'un n_estimator de 50. Le n_estimator représente le nombre de modèles s'entrainant sur les données, et 50 arbres de décisions n'est pas suffisant quant aux types de données : 99 classes et 192 variables.

Notre seconde observation porte sur le classifieur Naive Bayes. Notre erreur est 10 fois inférieure à la sienne, ce qui s'explique par le fait que nous avons utilisé les données sur lesquelles une ACP a été appliquée. Comme expliqué plus haut, l'ACP a permis d'enlever une variable qui était trop corrélé à une autre et donc faussait complètement l'utilisation de ce classifieur.

Nous sommes conscients que Jeff D. ne cherche pas à obtenir les meilleurs résultats possibles. Son Notebook expose une démarche scientifique permettant d'utiliser la librairie de scikit-learn et de manipuler 10 classifieurs de façon basique. Ces résultats démontrent tout de même que nous avons réussi à améliorer les performances des classifieurs considérés.

Le deuxième Notebook que nous avons sélectionné est celui de somaktukai. Dans celui-ci, il cherche à optimiser au mieux ses classifieurs. Ses résultats devraient donc être plus intéressants. Nous allons pouvoir comparer notre précision sur les méthodes de classification de SVM et de la régression logistique. Il obtient un score de précision de 0,98 et 0,97 respectivement pour chacun d'entre eux. Quant à nous, nous avons le même score de 0,98 pour les deux. Nos scores sont quasiment identiques, et cela s'explique par le fait que nous avons réalisé une même démarche scientifique (standardisation + grid search). Une différence mineure provient de l'utilisation du random search grid search de notre côté et d'une grid search de la sienne. Cependant, nous testons plus de valeurs, ce qui revient au même.

4. Conclusion

Ce projet a été pour nous l'occasion de manipuler et de nous familiariser avec des classifieurs dont nous avions uniquement vu la théorie en cours. Nous avons désormais une idée plus précise des bonnes pratiques qu'il faut mettre en place dans un projet de techniques d'apprentissage. Nous avons aussi appris à déterminer les spécificités d'un ensemble de données à l'aide de différents tests statistiques et à choisir le classifieur le plus adapté afin d'obtenir le meilleur score.