Rapport du Challenge - IMA205

Pap smear cells classification

Introduction

Le but de ce challenge est de classifier des images d'abord de manière binaire (cellules cancéreuses ou non), puis en multi-classes selon différentes catégories auxquelles peut appartenir une cellule.
Pour pouvoir classifier ces images, il y a deux grandes étapes. D'abord, trouver des *features* qui caractérisent bien les images, et qui permettent de les discriminer selon leurs classes d'appartenance. Ensuite, une fois ces features trouvées, il faut trouver un moyen de séparer l'espace des features : on détermine un classificateur à partir de l'ensemble d'entraînement. Cela permettra ensuite de réaliser des prédictions de classes sur les images de test.

Extraction manuelle de features

J'ai commencé mes essais en recherchant manuellement des features descriptives des cellules. Pour cela, je suis parti de l'article donné, qui donne un ensemble de features utilisées dans les recherches de l'auteur de l'article :

Feature
Nucleus area
Cytoplasm area
N/C ratio
Nucleus brightness
Cytoplasm brightness
Nucleus shortest diameter
Nucleus longest diameter
Nucleus elongation
Nucleus roundness
Cytoplasm shortest diameter
Cytoplasm longest diameter
Cytoplasm elongation
Cytoplasm roundness
Nucleus perimeter
Cytoplasm perimeter
Nucleus position
Maxima in nucleus
Minima in nucleus
Maxima in cytoplasm
Minima in cytoplasm

En python, la plupart de ces features peuvent être facilement calculées en utilisant *measure.regionprops* de *skimage*.

Normalisation

En partant de ces features, j'ai d'abord remarqué un problème. Les images données n'ont aucune échelle de taille, il est donc impossible d'utiliser comme features des valeurs de distance, telles que le plus long diamètre du noyau. En effet, on peut imaginer mesurer un noyau de même taille sur deux images prises à des échelles différentes, alors que le noyau serait en réalité de taille différente.

Pour pallier à ce problème, j'ai choisi d'utiliser uniquement des rapports de distances (sans unité) comme features de distance. Par exemple, j'ai remplacé les quatre features *petit diamètre du noyau*, *grand diamètre du noyau*, *petit diamètre du cytoplasme*, *grand diamètre du cytoplasme* par quatre features normalisées ne dépendant plus de l'échelle : *rapport du petit diamètre du noyau sur petit diamètre du cytoplasme*, *rapport du grand diamètre du noyau sur grand diamètre du cytoplasme*, *rapport du petit diamètre du noyau sur grand diamètre du noyau*, *rapport du petit diamètre du cytoplasme sur grand diamètre du cytoplasme*.

Pour les features de couleur, telles que la luminosité moyenne du noyau, je n'ai pas eu besoin de faire cette normalisation car la luminosité est donnée dans la même échelle (entre 0 et 255) sur toutes les images.

Traitement des données manquantes

Une fois après avoir créé mon script de calcul des features normalisées pour les images, je me suis rendu compte d'un problème. Le calcul de certaines features échouent sur certaines images, cela étant dû par exemple à l'absence de noyau sur certaines images. Il est alors impossible de calculer la luminosité moyenne du noyau, et d'autres features. Pour résoudre ce problème qui concerne 1 à 2% des images, j'ai d'abord pensé à donner une valeur arbitraire aux features qui ne peuvent pas être calculées. Mais cela risque de fausser la classification. En effet, si les features choisies arbitrairement sont trop éloignées de la moyenne de ces features sur les autres images, elles risquent d'être très discriminantes et vont probablement être le facteur principal de décision de classification de ces cellules. Or, si on choisit arbitrairement la valeur de ces features, cela revient à classer au hasard les cellules avec des valeurs manquantes.
Pour résoudre ce problème, il faut comprendre que comme il nous manque des features sur certaines images, notre but va être de classer ces images en utilisant l'information donnée par les features non-manquantes. Pour cela, il faut faire en sorte que le features qui n'ont pas pu être calculées ne soient pas du tout discriminantes pour le classificateur.

J'ai donc choisi de remplacer les valeurs de features manquantes par la moyenne des valeurs de ces features sur toutes les autres images. Pour cela, je met les valeurs manquantes à *None* lors du calcul des features, puis je traite le tableau de toutes les features en remplaçant les *None* par les moyennes des features des autres images.
On peut voir cela sur l'image suivante, qui affiche la valeur moyenne des 25 features sur toutes les images. Dans *INITIALLY*, on a des *nan* qui correspondent aux features qui n'ont pas pu être calculées pour certaines images (on ne peut pas calculer la moyenne sur lorsqu'il y a des valeurs *None*). Dans *MEAN*, on calcule cette fois les valeurs moyennes des features en enlevant les

None. Enfin, on remplace tous les *None* par ces valeurs moyennes de features, ce qui permet d'observer dans *FINALLY* les valeurs moyennes des features de toutes les images, en constatant qu'il n'y a plus de *nan*, donc plus de *None*.

```
INITIALLY
[139.72334326 125.92519197 149.72425656 187.42009132 177.63607306
 196.12328767  84.18310502  74.04931507 102.16164384          nan
          nan          nan          nan          nan          nan
          nan          nan          nan          nan          nan
          nan          nan          nan          nan          nan]

MEAN
All features: (2190, 25) | Features without None: (2178, 25)
[139.70399089 125.76454624 149.53948069 187.30211203 177.32644628
 195.88475666  84.31037649  74.09412305 102.17171717  93.1258945
  77.89491062 112.27955031 132.72451791 116.76033058 146.70982553
  70.89118457  58.20110193  89.96280992   0.63279204   0.53290066
   0.30925686   0.94155236   0.96165206   1.02349186   1.02509222]

FINALLY
Cleaned features: (2190, 25)
[139.72334326 125.92519197 149.72425656 187.42009132 177.63607306
 196.12328767  84.18310502  74.04931507 102.16164384  93.1258945
  77.89491062 112.27955031 132.72451791 116.76033058 146.70982553
  70.89118457  58.20110193  89.96280992   0.63279204   0.53290066
   0.30925686   0.94155236   0.96165206   1.02349186   1.02509222]
```

Standardisation

Maintenant, on a calculé toutes nos features normalisées, et on a complété les valeurs manquantes. Un problème est que ces features ne varient pas toutes dans les mêmes intervalles. Typiquement, les features de couleur vont varier dans [0,255], là où les rapports de distance du type *petit axe* sur *grand axe* vont varier dans [0,1]. Cela cause comme problème que toutes les features ne vont pas avoir la même importance au moment de séparer les données avec le classificateur.
On peut résoudre facilement ce problème en standardisant, c'est-à-dire en ramenant toutes les valeurs de features dans le même intervalle (en enlevant la moyenne et en divisant par la variance). Cela se fait rapidement en utilisant *StandardScaler* de *sklearn*.

Proportions des classes

Un biais peut être introduit lorsque l'ensemble d'entraînement ne contient pas exactement les mêmes proportions de chaque classe. Ici, c'est un peu le cas sur la classification normale (avec 55% de non cancéreux et 45% de cancéreux), mais c'est encore plus le cas sur la classification multiple :

```
Group 0: 682 occurences
Group 1: 551 occurences
Group 2: 69 occurences
Group 3: 105 occurences
Group 4: 127 occurences
Group 5: 102 occurences
Group 6: 138 occurences
Group 7: 578 occurences
Group 8: 569 occurences
```

On voit par exemple que le groupe 2 est presque dix fois moins représenté que le groupe 0, ce qui peut introduire des biais lors de l'entraînement. La méthode la plus simple de pallier à ce problème est d'utiliser comme méthode de score *balanced_accuracy* plutôt que *accuracy*.

Premières classifications

Avant de donner mes premiers résultats, je vais expliquer comment j'ai travaillé. Comme je n'avais pas accès aux vrais résultats de l'ensemble de test, il était impossible de vérifier la qualité du classificateur (sans passer par Kaggle). Pour cela, j'ai séparé mon ensemble d'entrainement en un sous-ensemble d'entraînement (70%) et un sous-ensemble de test (30%). Je pouvais alors évaluer les scores en mesurant l'accuracy sur le sous-ensemble de test. Lorsque les scores étaient suffisamment bons, je relançais l'entraînement cette fois sur tout l'ensemble d'entraînement, et je testais sur le vrai ensemble de test pour exporter les résultats vers Kaggle. Par la suite, je vais donner tous mes résultats en parlant de l'*accuracy* que j'ai obtenue avec cette division de l'ensemble de test. J'ai parfois regardé les résultats avec *Matthews Correlation Coefficient*, mais pas systématiquement, donc j'exprimerai tous les résultats avec l'*accuracy*.

J'ai commencé mes tests en utilisant une simple régression linéaire, des classificateurs linéaires paramétriques (LDA, QDA, Bayes), et non paramétriques (KNN).

```
Linear Regression accuracy: 0.8974008207934336 | Training accuracy: 0.9050228310502283
LDA accuracy: 0.8974008207934336 | Training accuracy: 0.9045662100456621
QDA accuracy: 0.8221614227086184 | Training accuracy: 0.8406392694063927
Bayes accuracy: 0.7346101231190151 | Training accuracy: 0.7301369863013699
KNN accuracy: 0.9028727770177839 | Training accuracy: 0.9420091324200913
```

Ces premiers scores étaient assez bons pour des classificateurs linéaires. On constate de plus qu'il n'y a pas d'overfitting, les scores sur l'ensemble de test sont très proches des scores sur l'ensemble d'entraînement. Le fait qu'il n'y a pas d'overfitting signifie qu'il n'y a pas trop de features, qu'on peut encore ajouter de l'information en trouvant de nouvelles features, sans pour autant rendre le classificateur trop spécifique aux données d'entraînement.
J'ai donc tenté d'ajouter d'autres features (en plus des features de l'article), en cherchant lesquelles pourraient ajouter de nouvelles informations discriminantes, et faire augmenter l'*accuracy*. En réalisant plusieurs tests, j'ai trouvé les features *Euler Number* (nombre relatif à la forme du noyau et du cytoplasme, qui varie selon leurs déformations) et *Solidity* (ratio du

nombre pixels dans le masque - du noyau ou du cytoplasme - sur les pixels composant leurs enveloppe convexe).
Ces nouvelles features m'ont permis d'atteindre de meilleures scores avec les même classificateurs que précédemment :

```
Linear Regression accuracy: 0.9233926128590971 | Training accuracy: 0.910958904109589
LDA accuracy: 0.9233926128590971 | Training accuracy: 0.910958904109589
QDA accuracy: 0.8057455540355677 | Training accuracy: 0.8141552511415525
Bayes accuracy: 0.7551299589603283 | Training accuracy: 0.7406392694063927
KNN accuracy: 0.9192886456908345 | Training accuracy: 0.9552511415525115
```

À nouveau, on constate qu'il n'y a toujours pas d'overfitting (hormis pour KNN). On obtient exactement les mêmes scores avec LDA et Linear Regression, ce qui laisse supposer qu'on arrive à séparer l'ensemble des features des images linéairement de la même façon pour les deux méthodes. Je pense que c'est donc un plafond de ce qu'on est capables d'obtenir avec les classificateurs linéaires. Cela m'a donc poussé à essayer des classificateurs non linéaires plus complexes.

Non-linear SVM

Comme je souhaitais obtenir de meilleurs résultats qu'avec les estimateurs linéaires précédents, je me suis orienté d'abord vers Non-linear SMV. Ce classificateur envoie les données dans une espace de dimension supérieure où elles pourraient être plus facilement séparables. Théoriquement, il devrait me permettre d'obtenir de meilleurs résultats.
Cependant, SVM requiert plusieurs paramètres, à savoir C, Gamma, et le noyau utilisé. Pour trouver les meilleurs paramètres, j'ai procédé en faisant une recherche exhaustive sur toutes les combinaisons de paramètres en utilisant *GridSearchCV*, qui fait de l'apprentissage par validation croisée. Au cours de tous mes essais sur différentes features, C et Gamma ont un peu varié au cours des différentes exécutions, mais le noyau choisi par la recherche exhaustive était toujours *rbf*. Je l'ai donc choisi définitivement, et enlevé de la recherche exhaustive pour accélérer les calculs.

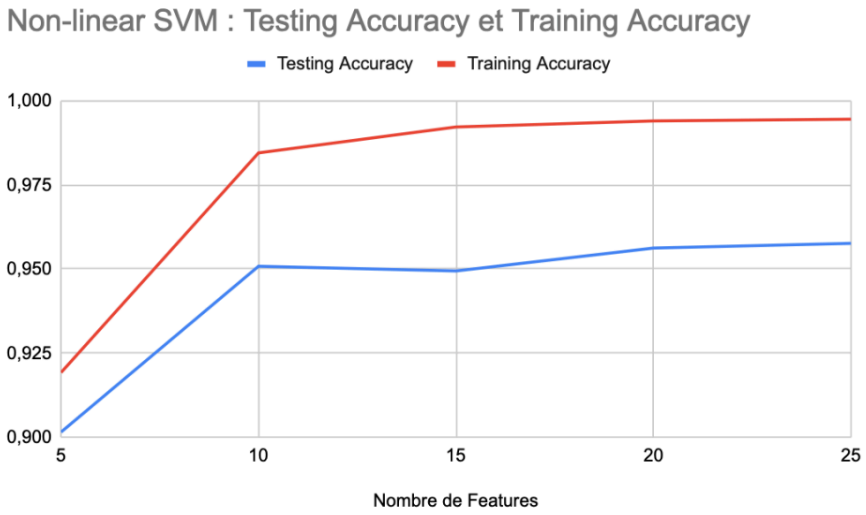
Avec le meilleur estimateur donné par la recherche exhaustive, j'ai en pratique bien observé de meilleurs résultats qu'avec les estimateurs linéaires précédents, comme attendu :

```
NL SVM accuracy: 0.9548563611491108 | Training accuracy: 0.9926940639269406
```

Cependant, on peut également remarquer qu'il semble y avoir un phénomène d'overfitting. J'ai donc essayé d'enlever certaines features pour réduire la complexité du modèle, mais sans succès.
Mais plutôt qu'enlever manuellement certaines features au hasard, je me suis dit qu'il avait mieux réaliser un PCA pour supprimer les features les moins discriminantes vis à vis de la classification à faire.

Dimension Reduction

J'ai donc essayé d'inclure une étape de PCA entre le calcul des features et l'entraînement du modèle. J'avais actuellement 25 features, et j'ai décidé de faire des tests en les réduisant à 20,15,10 et 5 :



Mais surprise, il n'y avait pas vraiment d'overfitting, le résultat sur l'ensemble d'entraînement n'a fait que diminuer en réduisant le nombre de features. À partir de là, je me suis dit que j'ai dû atteindre un certain plafond lié à mon choix de features. Mais l'avantage d'utiliser PCA est qu'on peut mettre autant de features que possible en entrée, et on choisit le nombre de features en sortie, qui sont celles qui favorisent le plus la dispersion des données.
En utilisant les features de *measure.regionprops* de *skimage*, j'ai utilisé 45 features. J'ai réduit ce nombre en utilisant PCA, mais j'ai obtenu des résultats légèrement inférieurs aux précédents. Je pense que cela s'explique par le fait que PCA donne en sortie des features qui maximisent la dispersion des données, mais ne prend pas en compte quelles features sont importantes pour discriminer les données vis à vis de la classification à faire.

De ce fait, le meilleur résultat que j'ai obtenu à ce moment était le résultat initial de Non-linear SVM, c'est-à-dire avec 25 features et sans appliquer PCA.

Boosting

Une idée pour améliorer le résultat que j'ai obtenu avec Non-linear SVM était de faire du boosting. L'idée est de réaliser une séquence d'estimateurs, et de combiner leurs résultats en les faisant voter, par exemple avec l'algorithme AdaBoost.
J'ai donc lancé AdaBoost, en choisissant comme estimateur de base le meilleur estimateur issu de la recherche exhaustive de Non-linear SVM. J'ai obtenu un résultat très légèrement meilleur, ce qui est peu significatif. Surtout le temps d'exécution pour ici un boosting avec 500 estimateurs est de presque 20 minutes, ce qui est vraiment très long.

✓ 1134.1s

```
AdaBoost accuracy: 0.9562243502051984 | Training accuracy: 0.9995433789954338
```

Ce temps très long m'a donc amené à utiliser des estimateurs beaucoup plus simples pour le boosting (à savoir les estimateurs par défaut des méthodes de *sklearn*, pour revenir au principe même du boosting. C'est-à-dire combiner un très grand nombre (ici 500) d'estimateurs basiques pour obtenir un bon résultat.

Avec AdaBoost, les résultats sont moins bons que précédemment avec Non-linear SVM comme estimateur de base, mais les résultats sont beaucoup beaucoup plus rapides. En augmentant le nombre d'estimateurs, l'*accuracy* augmente sur l'ensemble d'entraînement mais pas sur l'ensemble de test : on a de l'overfitting.

✓ 12.7s

```
AdaBoost accuracy: 0.948016415868673 | Training accuracy: 0.9684931506849315
```

Également, j'ai testé d'autres méthodes de boosting, notamment Gradient Boosting et Histogram Gradient Boosting, qui m'ont donné de meilleurs résultats :

```
Gradient boosting accuracy: 0.9521203830369357 | Training accuracy: 0.9648401826484019
```

```
Histogram-Gradient boosting accuracy: 0.9658002735978112 | Training accuracy: 1.0
```

Ces résultats sont mêmes meilleurs que ceux obtenus avec Non-linear SVM, cependant on remarque qu'on a beaucoup d'overfitting avec Histogram Gradient Boosting (l'*accuracy* sur l'ensemble d'entraînement est à 1 !). Je ne suis pas parvenu à réduire cet overfitting en limitant le nombre de features, que ce soit manuellement ou avec PCA.

MLP

Enfin, je me suis orienté vers le Multi-Layer Perceptron pour conclure mes essais. Les réseaux neuronaux permettent aujourd'hui d'obtenir les meilleurs résultats de classification, et je voulais donc les comparer aux algorithmes que j'ai utilisés précédemment.
J'ai fait de nombreux essais, notamment en variant le nombre de couches cachées, avec ou sans *batch normalization*, avec différents nombres d'*epoch* et tailles de *batch*. J'ai obtenu les meilleurs résultats avec *batch normalization*.
Je suis parvenu à des résultats similaires, parfois légèrement meilleurs que ceux obtenus précédemment. Mais j'ai constamment remarqué que j'avais de l'overfitting, ce qui se constate avec l'*accuracy* sur l'ensemble d'entraînement très proche de 1.

```
MLP accuracy: 0.957592339261286 | Training accuracy: 0.9995433789954338
```

Pour pallier à cet overfitting, j'ai essayé de régulariser en utilisant la méthode de *weight decay* (en utilisant l'optimiseur AdamW plutôt que Adam sur Tensorflow). Mais je n'ai pas trouvé de bon compromis ; en régularisant peu, le résultat ne changeait pas, et en régularisant plus, on avait moins d'overfitting mais l'*accuracy* sur l'ensemble d'entraînement diminuait aussi.

En y réfléchissant, j'ai obtenu mes meilleurs résultats de trois manières différentes : avec Non-linear SVM, avec Histogram Gradient Boosting sur beaucoup de petits estimateurs, et avec MLP. À mon avis, je pense que cela signifie que j'ai atteint les meilleurs résultats possibles (ou que j'en

suis très proche) avec mes features choisies initialement manuellement. J'ai essayé de modifier un peu mes features, d'en ajouter ou d'en enlever, mais dans le meilleur des cas, je n'ai réussi qu'à gagner quelques décimales non significatives. Finalement, pour palier à cette limite due à ce choix de features, il me reste à essayer de trouver des features optimales en utilisant les Réseaux Convolutionnels Neuronaux.

CNN

En réalisant plusieurs essais avec les CNN, je n'ai jamais réussi à approcher les résultats que j'ai obtenus avec mes précédentes features. Une des raisons est peut-être le fait de m'être limité à des modèles sans trop de paramètres, qui s'exécutaient sur ma machine dans un temps acceptable. Je n'ai pas eu le temps d'essayer des modèles plus compliqués sur Google Colab.

Conclusion

Finalement, j'ai obtenu de bons résultats avec Non-linear SVM, Histogram Gradient Boosting, et MLP, qui me permettent de réaliser des prédictions justes dans 96% des cas.
Je pense que ces résultats étaient surtout plafonnés par mon choix manuel de features, et que ces trois méthodes auraient pu donner des résultats encore meilleurs avec des features encore plus discriminantes. C'est peut-être ce que j'aurais pu observer si j'avais réussi à déterminer de meilleures features avec CNN.

Pour conclure, j'ai trouvé ce projet très intéressant, et il m'a permis de mieux comprendre, mettre en perspective et comparer différentes méthodes d'apprentissage. J'ai pu également comprendre tout le travail à effectuer pour trouver des features adaptées et pour les traiter de manière à les utiliser en entrée des algorithmes d'apprentissage.