Support Vector Machine

Dans ce travail pratique, On se réfère aux scripts de référence fournis pour explorer l'impact des paramètres de régularisation et des variables de nuisance dans un modèle SVM. Cependant, des modifications ont été apportées aux scripts originaux afin d'améliorer l'analyse et de restituer un code coherent Les détails de ces modifications, ainsi que les justifications des choix effectués, sont discutés dans un miscellanées en annexe pour une meilleure compréhension de leur influence sur les résultats.

Table of contents

1. Introduction aux SVM de Sklearn **3**
1.1. L'implementation sklearn.svm.SVC 3
1.2. Test des classifieur à noyaux lineaire, et polynomial, sur le dataset Iris 4

2. Le choix du paramètre `C` **5**
2.1. Illustration et explications dans le cas d'un jeu de données très déséquilibré . . . 5
2.2. Solutions envisageables proposer par la classe SVC 6

3. Classification de visages **7**
3.1. Exemple quantitatif de l'Influence du paramètre C et des variables de nuisance sur la performance du SVM 8
3.2 PCA 10

0. annexe **11**
0.1 code_snippet 11
0.1 Miscélénaire 11

1. Introduction aux SVM de Sklearn

Dans cette section, on s'attache à répondre aux question 1 et 2 du TP :

```
En vous basant sur la documentation à l'adresse suivante :
http://scikit-learn.org/stable/modules/svm.html
Ecrivez un code qui va classifier la classe 1 contre la classe
2 du dataset iris.
En utilisant les deuxpremières variables et un noyau linéaire.
En laissant la moitié des données de côté,
évaluez la performance en généralisation du modèle.
Comparez le résultat avec un SVM basé sur noyau polynomial
```

1.1. L'implementation sklearn.svm.SVC

L'implémentation de la classe sklearn.svm.SVC dans Scikit-learn permet de créer un C-classificateur à vecteurs de support (SVM), une méthode d'apprentissage supervisé utilisée pour les tâches de classification. Ce modèle vise à trouver l'hyperplan optimal qui sépare les données en deux classes tout en maximisant la marge entre celles-ci. L'algorithme supporte à la fois les données linéairement séparables (avec un noyau linéaire) et non-linéaires grâce à des noyaux comme RBF, polynomial, etc.

Cette classe va occuper une place centrale dans la suite de ce TP. On reprends alors la documentation de sklearn pour illustrer rapidement les différents paramètres qui serons amenés à manipuler par la suite.

Listing 1 la classe SVC

```
class sklearn.svm.SVC(*, C=1.0, kernel='rbf', degree=3,
gamma='scale', coef0=0.0, shrinking=True, probability=False,
tol=0.001, cache_size=200, class_weight=None, verbose=False,
max_iter=-1, decision_function_shape='ovr', break_ties=False,
random_state=None)
```

Pour fixer le role de certain de ces paremêtre, reprenons les notations des différent noyaux disponible :

- linéaire : $\langle x, x' \rangle$
- polynomial : $(\gamma \langle x, x' \rangle + r)^d$, où d est spécifié par le paramètre `degree`, et r par `coef0`.
- RBF : $\exp(-\gamma \|x - x'\|^2)$, où γ est spécifié par le paramètre `gamma`, qui doit être supérieur à 0.
- sigmoïde : $\tanh(\gamma \langle x, x' \rangle + r)$, où r est spécifié par `coef0`.

1.2. Test des classifieur à noyaux lineaire, et polynomial, sur le dataset Iris

On souhaite comparer les performances des noyaux linéaire et polynomial pour la classification avec un SVM. Pour cela, on utilise une GridSearch afin de trouver les meilleurs paramètres pour chacun de ces noyaux.

La recherche par grille permet d'optimiser les hyperparamètres clés, comme `C` pour le noyau linéaire, et `C`, `degree` et `coef0` pour le noyau polynomial.

En sélectionnant les paramètres optimaux pour chaque noyau, on peux comparer les résultats de manière équitable, en s'assurant que chaque modèle fonctionne à son meilleur potentiel.

A noté que les parametre optimales trouver sont fondementalement dépendant de l'espace des parametre candidat, il faut d'assurer que chaque modèle dispose d'une plage suffisement large pour que l'optimisation par GriSearch soit pertinente (voir Listing 2 pour plus de détaille)

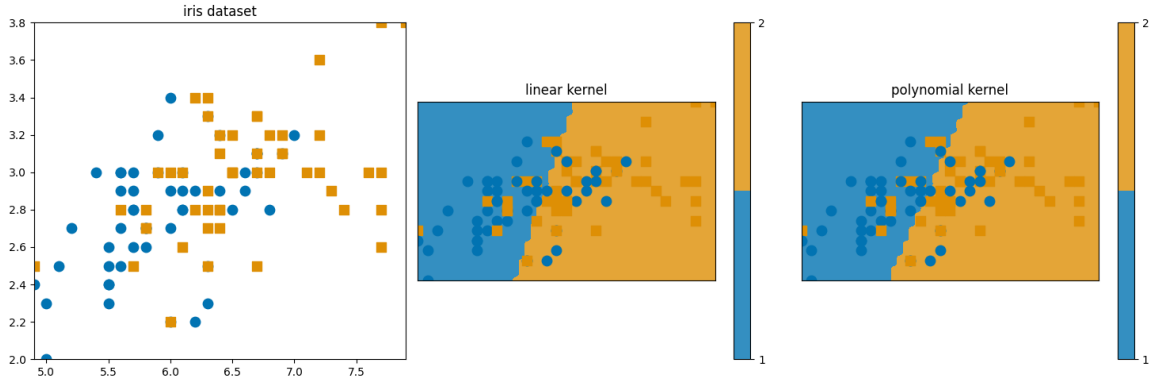


Figure 1: Exemples de frontières

```
polynomial : train : 0.724 test : 0.674
linear : train : 0.75 test : 0.682
```

Ici, les performance des deux modèle optimiser sont d'ordre simillaire. L'impacte du choix de `C` sur la qualité de la prédiction fait l'objet de la partie suivante.

2. Le choix du paramètre `C`

Dans cette section, on s'attache à répondre à la question 3 du TP :

```
Lancez le script svm_gui.py disponible à l'adresse
: https://scikit-learn.org/1.2/auto_examples/applications/svm_gui.html
Cette application permet en temps réel d'évaluer l'impact
du choix du noyau et du paramètre de régularisation C.

Générez un jeu de données très déséquilibré
avec beaucoup plus de points dans une classe que dans l'autre
(au moins 90% vs 10%).

À l'aide d'un noyau linéaire et en diminuant le paramètre C qu'observez vous ?
```

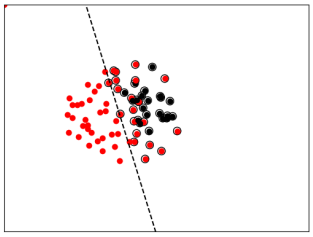
2.1. Illustration et explications dans le cas d'un jeu de données très déséquilibré

Le choix du paramètre `C` influence fortement les performances d'un SVM linéaire, en particulier dans le cas d'un jeu de données très déséquilibré où une classe est largement dominante.

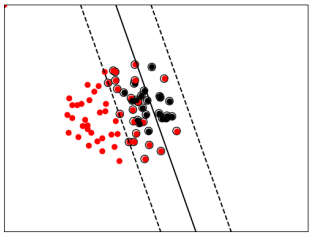
Intuitivement, `C` contrôle un compromis entre minimiser les erreurs de classification, et maximiser la distance entre l'hyperplan séparateur et les points les plus proches de chaque classe (les vecteurs de support).

Ainsi, avec un `C` élevé, le modèle pénalise fortement les erreurs, ce qui peut conduire rapidement à un surapprentissage de l'échantillons.

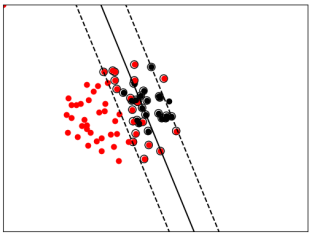
Inversement un `C` trop faible peut rendre le modèle trop permissif, réduisant sa capacité à capturer les nuances des points minoritaires.



C=0.1 , Accuracy: 72.0



C=10 , Accuracy: 81.33



C=20 , Accuracy: 84.0

Ainsi, ajuster `C` de manière appropriée est crucial pour équilibrer correctement les performances sur un jeu de données déséquilibré.

On conclu que, même si réduire `C` permet theoriquement une meilleure généralisation (puisque le modèle tolère plus d'erreurs et ne cherche pas à classer parfaitement chaque point.) dans un jeu de données déséquilibré, cela peut accentuer le biais vers la classe majoritaire (puisque les erreurs sur la classe minoritaire sont alors davantage tolérées) et baissé ainsi la qualité global de la prédicion.

2.2. Solutions envisageables proposer par la classe SVC

Le paramètre `class_weight` dans Scikit-learn permet de répondre au problème des jeux de données avec une classe très majoritaire en rééquilibrant l'importance des classes dans le modèle, via l'attribution de poids différent pour chaques classes sur la fonction de coût.

On peut soit spécifier délibérément dans un dictionnaire le poid de chaques classe, ou demendé une pondération automatique via la fréquence respective de chaques classes.

```
source : https://github.com/scikit-learn/scikit-learn/blob/main/
sklearn/utils/class_weight.py

class_weight : dict, "balanced" or None
  If "balanced", class weights will be given by
  `n_samples / (n_classes * np.bincount(y))`.
  If a dictionary is given, keys are classes and values are
  corresponding class
  weights.
  If `None` is given, the class weights will be uniform.
```

3. Classification de visages

Dans cette dernière parties, on s'attache a répondre aux question 4,5 et 6 du TP

Figure 2: Exemple de visages

```
L'exemple suivant est un problème de classification de visages.

Montrez l'influence du paramètre de régularisation.
On pourra par exemple afficher l'erreur de
prédiction en fonction de C sur une échelle logarithmique entre 1e5 et 1e-5.
En ajoutant des variables de nuisances,
augmentant ainsi le nombre de variables à nombre de points
d'apprentissage fixé, montrez que la performance chute.
Améliorez la prédiction à l'aide d'une réduction de
dimension basée sur l'objet
sklearn.decomposition.PCA(svd_solver='randomized').
```

3.1. Exemple quantitatif de l'Influence du paramètre C et des variables de nuisance sur la performance du SVM

Pour illustrer concrètement l'impact du paramètre de régularisation `C` sur les performances d'un SVM, on s'attache maintenant examiner un exemple quantitatif.

Sur cette classification de visage,en faisant varier `C` sur une large échelle logarithmique, il vas étre vue comment la capacité du modèle à faire des prédictions évolue en fonction de ce paramètre.

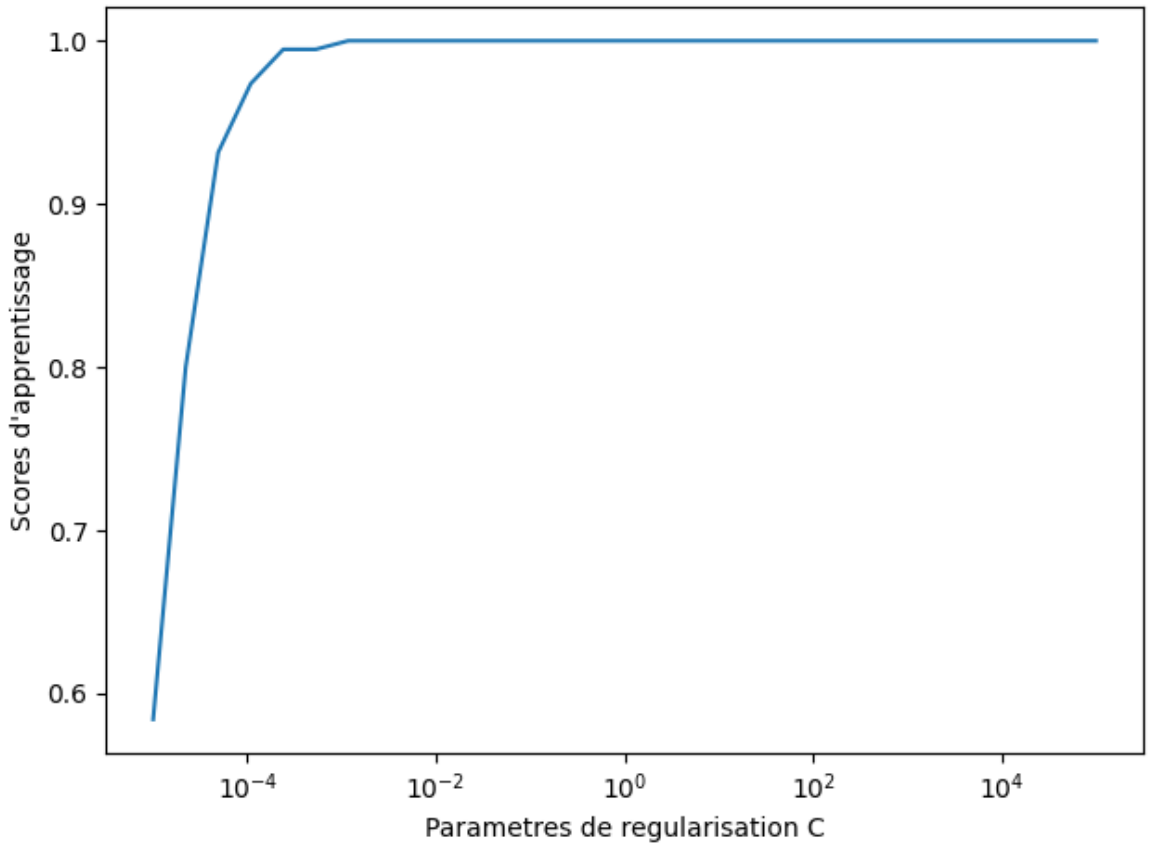


Figure 3: Optimisation du parametre C

```
Best C: 0.0011721022975334804
Best score: 1.0
Chance level : 0.6210526315789474
Accuracy : 0.9263157894736842
```

Ainsi sur ce tirage, à partir de 0.0011, le compromis entre maximisation de la marge et minimisation des erreurs est optimal : augmenter `C` au-delà de cette valeur n'améliore plus les performances, car le modèle a déjà trouvé une marge suffisamment large tout en pénalisant efficacement les erreurs. Prendre une valeur de `C` plus grande comporterais le risque de surapprentissage , comme discuter dans la partie 2.

Poursuivons notre analyse avec cet exemple quantitatif en explorant un autre facteur influençant les performances du modèle : l'ajout de variables de nuisance, non informatives ou bruitées.

On montre comment l'augmentation du nombre de variables parasites entraîne une baisse des performances de prédiction, en diluant les signaux pertinents et en compliquant la tâche de classification du SVM.

Table 1: Variable de nuisances centrée

Nombre de variables de nuisance	Variance des variables de nuisance	Signal dilué	Train Score	Test Score
0	N/A	1	1.0	0.8842
10 000	1	0.5	1.0	0.8684
20 000	1	0.33	1.0	0.8368
30 000	1	0.25	1.0	0.8368
40 000	1	0.2	1.0	0.7632

Les données de Table 1 on était généré par Listing 3 On observe que le nombre de variables de nuisance a un impact les performance du model, bien que tant que leur variance reste faible, le SVM parvient à maintenir un score de prédiction relativement stable. le train score constant de 1 suggère un réel impact du bruit sur la qualité d'aprentissage. Pour mettre en perspectives, rappelons que le model constant (`chance level`) score 0.62 en test.

On peut de plus s'interresser a un cas un peut moins subtile, lorsque ces variables présentent une forte variance. Dans ce cas, elles perturbent davantage l'apprentissage en dominant les dimensions informatives, ce qui dégrade significativement les performances du modèle.

Table 2: Variable de nuisances non centrée

Nombre de variables de nuisance	Variance des variables de nuisance	Signal dilué	Train Score	Test Score
0	N/A	N/A	1.0	0.8842
10 000	1	0.5	1.0	0.8684
10 000	2	0.5	1.0	0.7842
10 000	3	0.5	1.0	0.7263
10 000	4	0.5	1.0	0.6895

Ce faisant, il est important de noter que ce scénario est moins réaliste, puisqu'on à initiallement centré et réduit nos donnée d'origine. Cela pourrait affecter la comparaison avec les résultats initiaux, car un bruit non centré introduit un biais d'échelle qui n'est plus seulement lié au nombre de variables, mais à leur valeur numériques intrinsèque. Voir Listing 4 pour le code qui a généré Table 2

Un solution envisageable à l'existence de donnée parasite est La réduction de dimension, visant à projeter les données dans un espace de plus faible dimension tout en conservant au maximum l'information pertinente. c'est l'objet de la PCA qui est le propos de la partie suivante.

3.2 PCA

Dans cette dernière partie du travail pratique, on se concentre sur l'utilisation de l'Analyse en Composantes Principales (PCA) pour réduire la dimensionnalité des données, cette fois sans ajouter de bruit. Cette technique permet non seulement d'améliorer l'efficacité des algorithmes d'apprentissage, mais aussi de minimiser le risque de surapprentissage en éliminant les redondances et les corrélations superflue entre les variables. On applique la PCA sur 20 composente sur le jeu de données avec 40 000 variable de nuisance (signial initial dilué a 0.2) pour observer comment la réduction de dimension affecte les performances du SVM.

2 solveur on étaient tester, voici leur résultat :

```
# solver='randomized'
score sans PCA :
Cross validate Train scores:  1.0
Cross validate Test scores:  0.7736842105263158
score avec PCA :
Cross validate Train scores:  0.7736842105263158
Cross validate Test scores:  0.7473684210526316
```

```
#solver='auto'
score sans PCA :
Cross validate Train scores:  1.0
Cross validate Test scores:  0.7473684210526316

score avec PCA :
Cross validate Train scores:  0.8789473684210526
Cross validate Test scores:  0.8210526315789474
```

C'est bien évidemment attendu, le solveur randomized étant très mal adapté au protocole proposé ici, où le choix a été fait de noyer le signal dans le bruit. En utilisant le solveur classique (qui est bien moins rapide que le randomisé), on un gain significatif à l'emploi de la PCA sur les données bruitées.

0. annexe

0.1 code_snippet

0.1 Miscélénaire

- J'ai modifié le script d'origine en raison d'une erreur fondamentale dans l'instruction suivante : `X_noisy = X_noisy[np.random.permutation(X.shape[0])]`. Cette ligne de code mélange aléatoirement les lignes de X_noisy avant l'ajout de bruit, ce qui altère la structure et la relation entre les données. Dans le cadre de cette analyse, on souhaite juger de l'impact de l'ajout de variables de nuisance sur le modèle SVM. En permutant les lignes, on fausse grandement les résultats et la comparaison est impossible.
- j'ai remplacer `astype(np.int)` par `astype(int)`, `np.int` est déprécier. (voir https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations)
- j'ai retiré la section `Toyset` du code, elle contien des exemples bien utiles mais superflue pour le rendu final.

Listing 2 Linear and Polynomial SVC

```
Cs=list(np.logspace(-3, 3, 200))                                                    ①

parameters_linear = {'kernel': ['lineaseed=1

print("Score sans variable de nuisance\n")
run_svm_cv(X, y, random_seed=seed)

for i in range(1000, 5000,1000):

    X_noisy = []
    noise = sigma * np.random.randn(n_samples, i)
    X_noisy = np.concatenate((X, noise), axis=1)
    print("-----------------------------------------------------")
    print("Score avec ",i," variable de nuisance\n")
    run_svm_cv(X_noisy, y, random_seed=seed)eters_linear)                           ③
clf_l_grid.fit(X_train, y_train)
clf_l = clf_l_grid.best_estimator_
```

```
Cs = list(np.logspace(-3, 3, 5))                                                    ①
gammas = 10. ** np.arange(1, 2)
degrees = np.r_[1, 2, 3]

parameters_polynomial = {'kernel': ['poly'], 'C': Cs, 'gamma':                      ②
gammas, 'degree': degrees}

clf_p_grid=GridSearchCV(SVC(), parameters_polynomial)                               ③
clf_p_grid.fit(X_train, y_train)
clf_p=clf_p_grid.best_estimator_
```

① Définir l'espaces des candidats pour chaques paramètres.
② Formater ces espaces aux standard du classifieur.
③ Effectuer une GridSearch sur ces paramètres pour le classifieur a optimiser.

Listing 3 Les Variables de nuisances centrée

```
sigma = 1.
seed=1

print("Score sans variable de nuisance\n")
run_svm_cv(X, y, random_seed=seed)

for i in range(1, 5,1):
    np.random.seed(seed)
    X_noisy = []
    noise = sigma * np.random.randn(n_samples, i*10000)
    X_noisy = np.concatenate((X, noise), axis=1)

    print("-----------------------------------------------------")
    print("Score avec ",i*10000," variable de nuisance de variance :",sigma,", signal dilué a
    run_svm_cv(X_noisy, y, random_seed=seed)
```

Listing 4 Les Variables de nuisances non centrée

```
for i in range(1, 5,1):
    sigma = i
    np.random.seed(seed)
    X_noisy = []
    noise = sigma * np.random.randn(n_samples, 10000)
    X_noisy = np.concatenate((X, noise), axis=1)

    print("-----------------------------------------------------")
    print("Score avec ",10000," variable de nuisance de variance ",sigma,", signal initial di
    run_svm_cv(X_noisy, y, random_seed=seed)
```