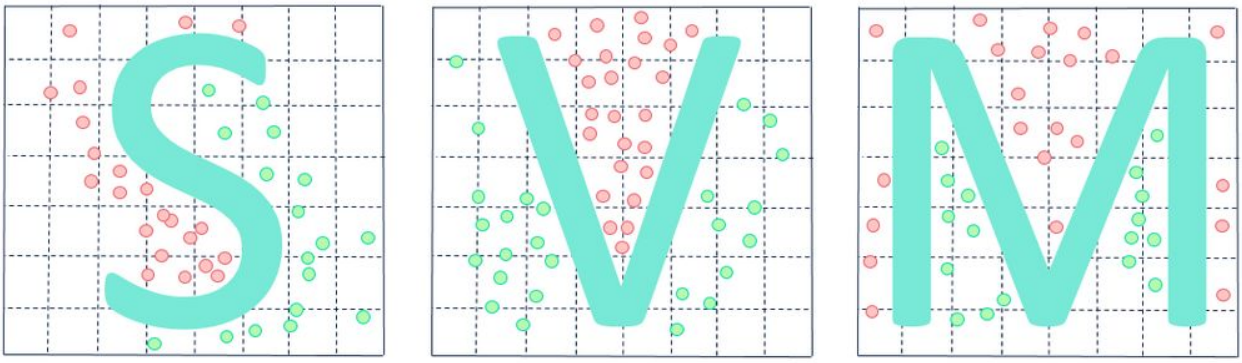
RAPPORT PROJET

Optimisation pour les sciences des données



AZOULAY Mikael

Janvier 2020

M2 MIAGE ID Classique

INTRODUCTION

Dans ce rapport nous allons tenter de retranscrire le travail effectué sur les différents SVM et datasets mis à disposition. Dans une première partie nous explorerons les datasets et les préparons au mieux pour l'apprentissage automatique. Dans la seconde partie nous ajusterons au mieux les différents paramètres de nos modèles afin d'optimiser la classification. Enfin dans la troisième et dernière partie nous reviendrons sur certains problèmes rencontrés et autres. Ce rapport expose et analyse les résultats de mon code python.

Ce rapport et code sont réalisés dans le cadre du projet d'optimisation pour les sciences des données en Master 2 Miage ID à l'université Paris-Dauphine.
contact : **mikazoulay101@gmail.com**

Sommaire

I- Data preprocessing

Dans cette partie nous allons d'abord explorer nos datasets puis effectuer quelques traitements afin d'améliorer la qualité de l'apprentissage de nos classifiers SVM. On compte 5 datasets : *liver, diabetes, heart-bin, orig-bc, sonar.* Ils présentent tous de la même façon **n examples** :

- **k features** (numerical)
- 1 binary **'target' class** (= { 1 , 2 })

Dataset	**n** (number of examples)	**k** (number of features)
heart-bin	270	20
diabetes	768	8
bc-orig	683	9
liver	345	6
sonar	208	60

1- Data Exploring

Avant d'appliquer nos SVM sur les différents datasets, il est utile d'en avoir un premier aperçu à travers de la visualisation de données. On implémente la méthode *visualize* qui va faire appel aux librairies seaborn et matplotlib de python afin d'avoir des graphiques reprenant les aspects de notre dataset. Nous allons nous pencher sur les résultats affichés lors de son exécution sur le fichier '*diabetes.tx*t' reprenant les informations de santé de patients (les features) et si oui ou non ils sont diabétiques (le label). Par la suite notre classifier svm entraîné sur ces données pourra ensuite prédire d'un patient en disposant de ses informations de santé si oui ou non il a le diabète. Mais avant visualisons ces données à l'aide la méthode *visualize* que nous avons implémenté :

```
visualize('diabetes.txt')
```

1- Statistics

```
768  examples, 8  features  and 1 target class
```

	timespreg	gluctol	diaspb	triceps	...	massindex	pedigree	age	target
0	6.0	148.0	72.0	35.0	...	33.6	0.627	50.0	b'2'
1	1.0	85.0	66.0	29.0	...	26.6	0.351	31.0	b'1'
2	8.0	183.0	64.0	0.0	...	23.3	0.672	32.0	b'2'
3	1.0	89.0	66.0	23.0	...	28.1	0.167	21.0	b'1'
4	0.0	137.0	40.0	35.0	...	43.1	2.288	33.0	b'2'

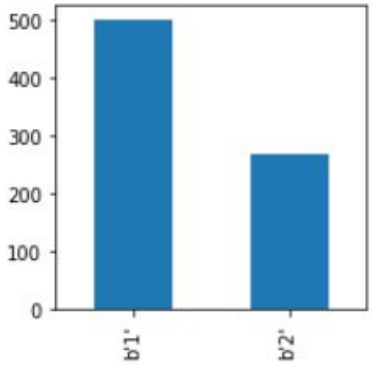
```
[5 rows x 9 columns]
```

	count	mean	std	...	50%	75%	max
timespreg	768.0	3.845052	3.369578	...	3.0000	6.00000	17.00
gluctol	768.0	120.894531	31.972618	...	117.0000	140.25000	199.00
diaspb	768.0	69.105469	19.355807	...	72.0000	80.00000	122.00
triceps	768.0	20.536458	15.952218	...	23.0000	32.00000	99.00
insulin	768.0	79.799479	115.244002	...	30.5000	127.25000	846.00
massindex	768.0	31.992578	7.884160	...	32.0000	36.60000	67.10
pedigree	768.0	0.471876	0.331329	...	0.3725	0.62625	2.42
age	768.0	33.240885	11.760232	...	29.0000	41.00000	81.00

Les statistiques ci-dessus dévoilent le fait qu'il existe des valeurs extrêmes (outliers) dans notre dataset. Par exemple, l'insuline moyenne est à 79 mais un individu en possède une égale à 846. il apparaît clair que ce genre de valeurs devront par la suite être corrigées afin qu'elle n'influencent pas à tort l'apprentissage de nos classifiers.

2- Repartition

```
Balance :
 b'1'    500
b'2'    268
Name: target, dtype: int64
```

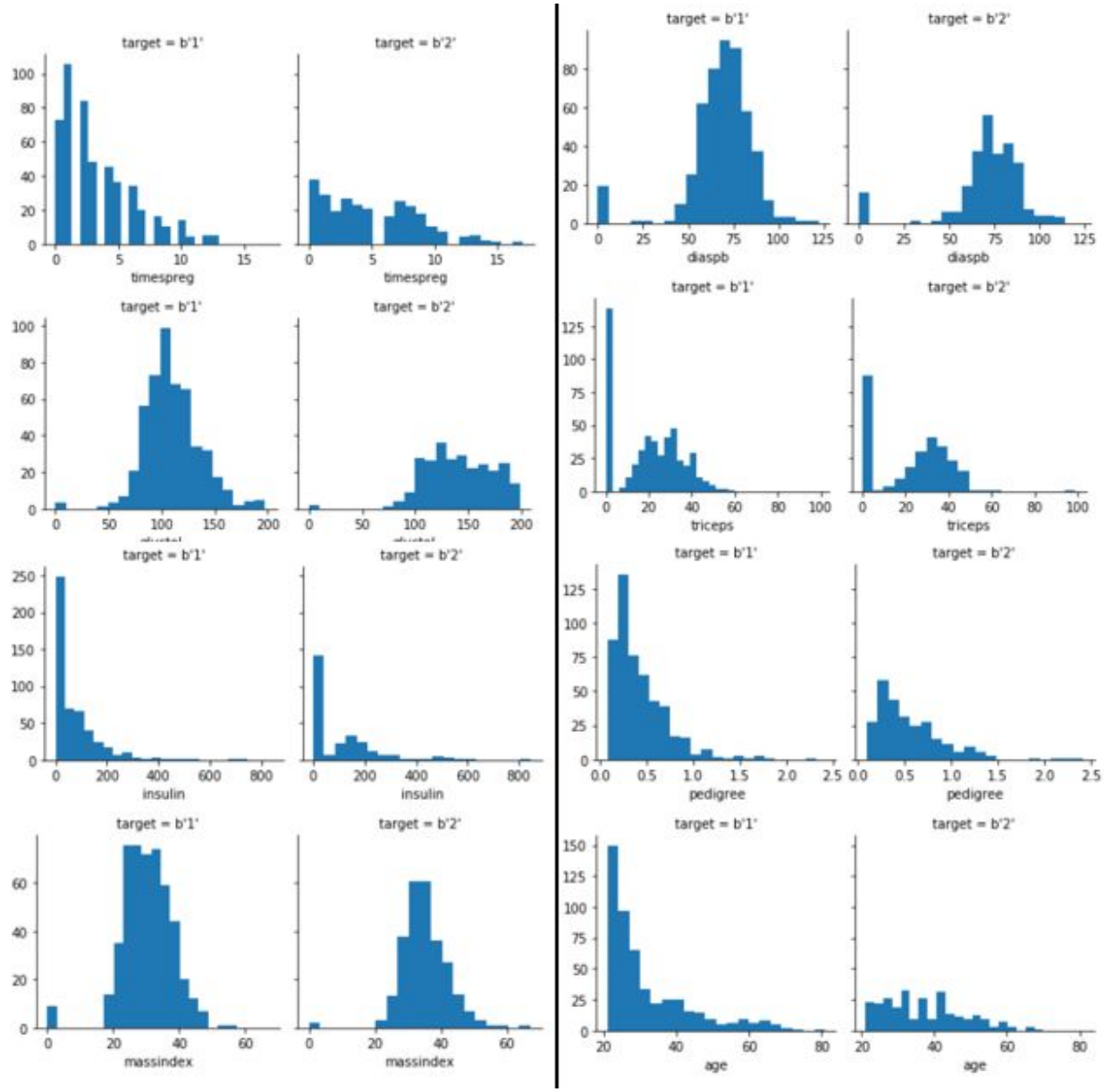


On en apprend un peu plus sur les features et sur la class target : 2 signifie que la personne est diabétique 1 signifie qu'elle ne l'est pas. Regardons la répartition de cette classe au sein de notre dataset :

Le graphe ci contre montre que notre dataset est biaisé. Il y'a environ 2 fois plus de non-diabétiques que de diabétiques dans nos données.

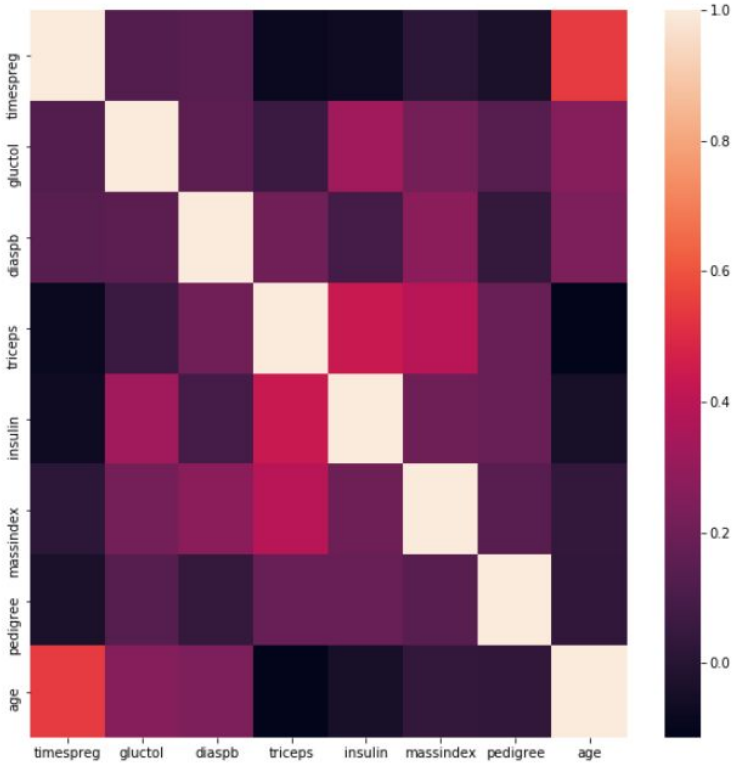
Il faudra prendre en compte cette information quand on entraînera le modèle et lorsqu'on lira son score.

Observons les différences de répartition entre nos classes selon chacun des features.
En x les valeurs des features et en y le nombre d'individus.



Grace a ces graphiques on peut déjà faire de nombreuses interprétations. Par exemple, on réalise l'importance du critère *massindex* sur la différenciation entre diabétique ou sain. En effet les personnes diabétiques forme une gaussienne au niveau de 35 de *massindex* tandis que les personnes saines sont vers 25. Cela s'explique par le surpoids présent chez certaines personnes diabétiques.

3- Correlation



Cette matrice de corrélation confronte deux à deux les features du dataset à la recherche de corrélations linéaires entre eux. Cette visualisation nous permet de constater des dépendances entre notre features. Par exemple les critères *age* et *timespreg* sont assez corrélés cela s'explique simplement que plus une femme est âgée plus elle a de chance d'avoir été enceinte.

2- Data transformation

Avant d'entraîner nos SVM avec les données issues des différents data sets on va appliquer en amont différents traitements.

1- Data loading

On charge les données des fichiers texte dans dataframe python.

2- Data type

On fait passer le type des labels de *object* à *int.*

3- Data splitting

D'abord on en garde que les n premières instances du data set . Ensuite on le divise en un training set et un validation set de façon à avoir 15% des données dans le validation set. On

utilise pour cela la méthode *train_test_split* de la librairie python scikit-learn qui mélange nos données et effectue la division de façon aléatoire en fonction d'un random seed.

4- Correcting outliers

On corrige les outliers (valeurs aberrantes) de la façon suivante : sont considérées comme outliers les exemples possédant un feature (attribut) qui s'éloigne de la moyenne de plus de deux fois l'écart type.
Autrement dit soient **Z** les valeurs de features aberrantes, soit **μ** la valeur moyenne de l'attribut et **σ** l'écart-type de l'attribut alors :

- **Z > μ + 2 σ OU Z < μ - 2 σ**

On corrige ces valeurs en les ramenant à la moyenne + ou - deux fois l'écart-type, autrement dit :

- **IF Z > μ + 2 σ THEN Z := μ + 2 σ**
- **IF Z < μ - 2 σ THEN Z := μ - 2 σ**

A noter que l'on corrige les outliers uniquement dans le training set !

5- Splittings labels and features

On sépare les features des labels. On met les features dans un tableau X et les labels dans un tableau Y.

6- Scaling the data

On normalise (standardize) nos valeurs x afin d'avoir une distribution telle que **μ = 0 et σ = 1.** On procède de la façon suivante :

Standardization:

$$z = \frac{x-\mu}{\sigma}$$

with mean:

$$\mu = \frac{1}{N}\sum_{i=1}^{N}(x_i)$$

and standard deviation:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)^2}$$

Par la suite nous mesurerons l'impact de cette standardization sur l'apprentissage de nos SVM afin de justifier ce choix.

On implémente ces différents traitements sont dans une méthode que l'on appelle *preprocess :*

```
preprocess(datafile, n_max = 500, standardization = True,
random_seed = None)
```

Arguments :

datafile (str) : nom du fichier du dataset
n_max (int) : nombre d'exemples maximum du dataset
standardization (bool) : True on normalise, false on ne le fait pas
random_seed (int) : random seed utilisé pour effectuer la division training/test set

La méthode retourne 4 tableaux de valeurs : X_train, y_train, X_test, y_test.
Le training séparé en features (X_train) et labels (y_train) et pareil pour le test set.

II- SVM tuning

Dans cette partie nous allons appliquer nos 3 classifiers svm telle que définis par la consigne aux différents datasets. Nous allons tenter de découvrir la configuration optimale de ces classifiers en tâtonnant différents paramètres. Nous allons mesurer les performances en termes d'**accuracy** c'est à dire nombre d'éléments bien classifiés sur nombre total . On rappelle les 3 svm avec : **Hinge** loss, **Hard Margin** loss et **Ramp** loss. Nous ferons dans un premier temps varier l'**hyperparameter C** énoncé dans la consigne. Puis nous testerons différentes **dimensions** pour nos datasets.

1- Implementation

Pour implémenter les SVM on a choisit d'utiliser la CPLEX 12.6.3 Python API ainsi que la libraire de machine learning python Scikit-learn. Ainsi notre classe *CustomSVM* permet de résoudre des problèmes de classification à l'aide de SVM et en donnant le choix à l'utilisateur entre les 3 fonctions de loss possibles et une valeur du paramètre C.
On définit 4 méthodes :

```
__init__(self, loss="ramp", C=1.0, time_limit = False)
```

Cette première méthode instancie le problème avec les paramètres voulus : Loss, C, Time_limit.

```
fit(self, X_train, y_train)
```

Une fois instancié notre problème doit être résolu. Pour se faire on appelle cette méthode qui fit un training set et résout le problème.

```
predict(self, X_test)
```
Utilise la solution issue de la méthode fit pour prédire la classification de données dont on ne connaît pas forcément la classification.

```
score(self, X_test, y_test, measure="accuracy")
```
Donne le score de notre résolution sur un test set (ou un training set). Par défaut le score est l'accuracy c'est à dire la proportion d'éléments bien classifiés que donne notre modèle. On peut choisir d'autre type de mesure comme la "precision", la "sensitivity" ou la "confusion_matrix".

On a ainsi implémenté de la même façon que les modèles sur scikit-learn.

2- A First gridsearch

Afin de déterminer le meilleur paramétrage de nos SVM pour résoudre nos problèmes de classification nous allons tester différentes valeur de C pour chacune des 3 Loss Function de nos SVM et pour chacun de nos 3 datasets et nous mesurons l'efficacité en terme d'accuracy de la classification. On implémente la méthode *gridsearch* qui effectue les tests en croisant les paramètres suivant : datasets, C et les 3 loss possibles. Elle retourne le score dans chacun des cas sur le training set et sur le test set. Nous sommes dans le cas où les résultats dépendent de la conception aléatoire du test set. C'est pour cette raison pour plus d'exactitude nous allons lancer plusieurs exécutions pour chacun des paramètres et le score sera le score moyen de ces différentes exécutions souhaitées.

```
gridsearch(datafiles, paramC, time_limit = False, n_max = 500, RUNS = 1)
```

Arguments :

datafiles (str array[]) : nom des fichiers des data sets à tester
n_max (int) : nombre maximum d'exemples dans pour chacun des data sets
paramC (int arry[]) : tableau contenant les valeurs de C à tester
time_limit(int) : le temps de calcul maximum d'un apprentissage en seconde (par défaut il n'y en a pas)
random (bool) : Si random = true alors à chaque exécution la division train/test set est effectuée de façon complètement aléatoire sinon elle suit pour chaque exécution le random seed égal au numéro de l'exécution
RUNS (int) : le nombre d'exécution voulues

Voici les résultats de la commande suivante :

```
paramC = [0.001,0.01,0.1,1,10,100,1000]
datafiles = ["diabetes.txt", "liver.txt", "heart-bin.txt"]
gs1 = gridsearch(datafiles, paramC, time_limit = 10, n_max = 50, RUNS = 10)
```

On teste 8 valeurs de C x 3 datasets x 3 loss = 63 résultats, issus de la moyenne de **630 exécutions**.

			score_train	score_test	time
dataset	loss	C			
diabetes.txt	hard_margin	0.001	0.5075	0.47	0.044417
		0.010	0.5075	0.47	0.054093
		0.100	0.5075	0.47	0.098231
		1.000	0.5100	0.45	0.508903
		10.000	0.5600	0.36	4.149016
		100.000	0.5825	0.38	4.185235
		1000.000	0.5875	0.39	5.648729
	hinge	0.001	0.6875	0.63	0.001312
		0.010	0.6900	0.63	0.001445
		0.100	0.7525	0.61	0.001562
		1.000	0.7650	0.66	0.001777
		10.000	0.7725	0.65	0.002105
		100.000	0.7775	0.64	0.003195
		1000.000	0.6425	0.63	0.003715
	ramp	0.001	0.5225	0.41	20.014833
		0.010	0.6375	0.49	20.061890
		0.100	0.7550	0.63	19.881913
		1.000	0.8375	0.69	4.131626
		10.000	0.8750	0.71	3.379724
		100.000	0.8875	0.67	1.519298
		1000.000	0.8975	0.61	1.673117
heart-bin.txt	hard_margin	0.001	0.5500	0.50	0.036811
		0.010	0.5500	0.50	0.071462
		0.100	0.7325	0.69	6.075432
		1.000	0.8125	0.65	9.995570
		10.000	0.9300	0.75	0.253533
		100.000	0.9500	0.76	0.106061
		1000.000	0.9625	0.69	0.070988
	hinge	0.001	0.8250	0.75	0.001353

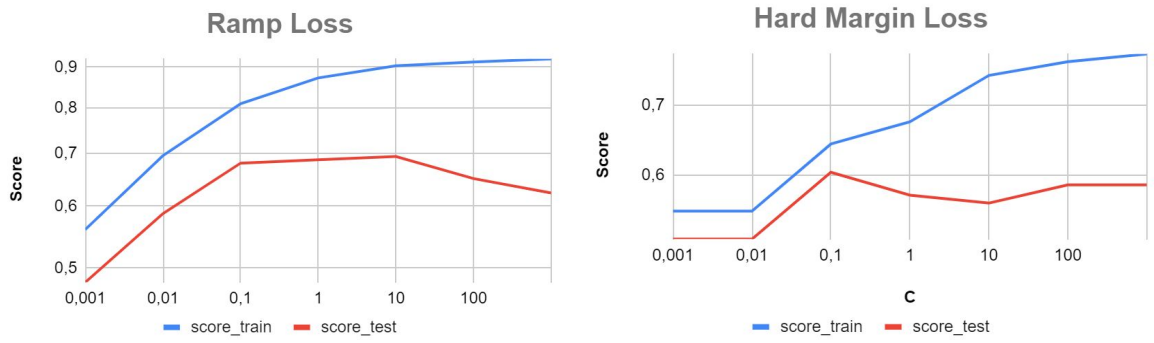
		0.010	0.8475	0.75	0.001650
		0.100	0.8950	0.77	0.001707
		1.000	0.9675	0.72	0.001839
		10.000	1.0000	0.75	0.002054
		100.000	1.0000	0.75	0.002051
		1000.000	1.0000	0.75	0.002259
	ramp	0.001	0.5550	0.44	20.020147
		0.010	0.8450	0.68	20.058769
		0.100	0.9050	0.76	0.714775
		1.000	0.9750	0.73	0.080245
		10.000	0.9950	0.75	0.025404
		100.000	1.0000	0.73	0.018524
		1000.000	1.0000	0.73	0.017885
liver.txt	hard_margin	0.001	0.6025	0.59	0.036704
		0.010	0.6025	0.59	0.040808
		0.100	0.6875	0.65	0.060770
		1.000	0.7025	0.62	0.132616
		10.000	0.7550	0.58	0.211209
		100.000	0.7825	0.62	0.193441
		1000.000	0.8050	0.68	0.151333
	hinge	0.001	0.7525	0.72	0.001296
		0.010	0.7525	0.72	0.001583
		0.100	0.7725	0.63	0.001751
		1.000	0.7650	0.62	0.001721
		10.000	0.7575	0.64	0.002152
		100.000	0.7425	0.65	0.003016
		1000.000	0.6625	0.53	0.003480
	ramp	0.001	0.6025	0.59	20.015170
		0.010	0.6025	0.59	20.056778
		0.100	0.7675	0.65	15.093650
		1.000	0.8050	0.64	4.198538
		10.000	0.8425	0.62	1.986876
		100.000	0.8550	0.55	1.458030
		1000.000	0.8700	0.53	3.324149

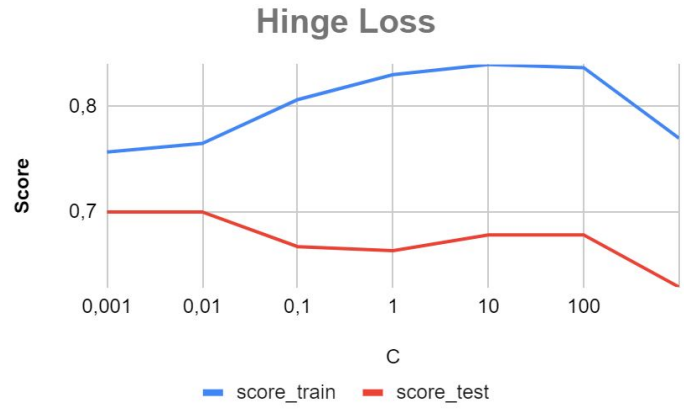
Résulats : Tout d'abord on remarque de nettes différences de comportements d'un dataset à un autre. A paramètre égale un modèle peut se comporter de manière complètement différente selon le dataset.
Pour le dataset *diabetes* les meilleurs résultats sont obtenus en utilisant une ramp loss et C autour de 10.
Pour le dataset *liver c'est la hinge qui obtient les meilleurs résultats* avec des valeurs de C très faibles. Pour le dataset heart-bin c'est encore la hinge même si la ramp a également de très bons résultats. On rappelle que certains datasets comme diabetes sont biaisés le score sur un test set équi réparti en termes de classe sera donc plus grand.

3- SVM parameter C

Le paramètre C nous permet de réguler l'apprentissage de notre modèle. Plus il est grand plus notre modèle 'fit' le training set la contrepartie est qu'il 'apprend par coeur' c'est à dire qu'il aura tendance à moins généraliser et donc son score sur le test set sera plus faible.
En diminuant la valeur de C on peut éviter cet 'overfitting' et avoir un modèle plus robuste sur des test sets étrangers. Le tout est de trouver le bon compromis afin de ni underfit ni overfit le training set. En reprenant les résultats précédents on trace ces courbes qui nous aideront à déterminer ce compromis pour le dataset *diabetes.*

```
gs2=gs1.groupby(['loss','C'])['score_train', 'score_test','time'].mean()
gs2.to_excel("output.xlsx")
```

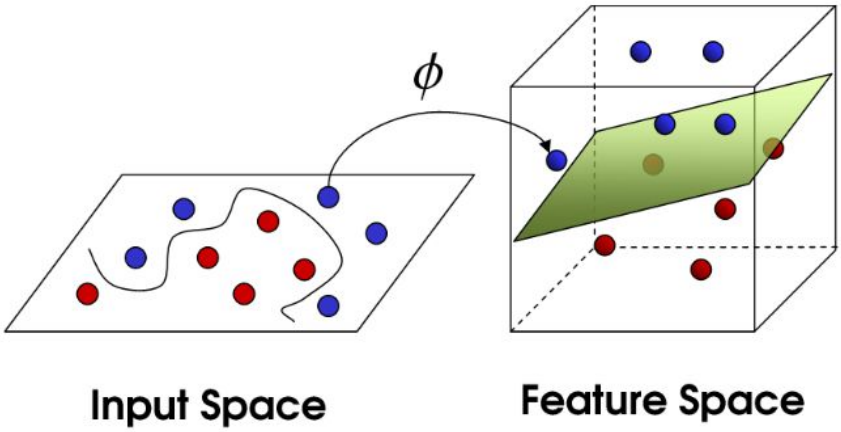




On choisirait donc C = 10 pour la Ramp et la Hinge loss et C = 0.1 pour la hard_margin.

4- SVM polynomial parameter

Nous avons nos valeurs optimales de C cependant les scores de nos classifiers peuvent être encore améliorés. En effet nous sommes dans un cas ou nos datasets ne sont pas linearly separable et une approche qui permettrait d'augmenter leur précision serait d'ajouter des features polynomiaux autrement dit d'augmenter de dimensions nos datasets afin de mieux pouvoir séparer nos classes.



Sur l'image ci dessus on observe que le passage de notre dataset de 2D à 3D a permis de pouvoir mieux séparer linéairement nos classes bleus et rouges.

Dans notre cas pour chacun de nos datasets nous allons tester le svm avec hinge loss pour plusieurs dimensions de la 1 (dataset inchangé) à la 10 et nous allons à chaque fois lancer 100 exécutions et récolter les scores moyens obtenus. Pour ce faire nous implémentons la méthode suivante:

```
def polysearch(datafiles, dim, RUNS = 100)
```

Arguments :

datafiles (str array[]) : nom des fichiers des data sets à tester
n_max (int) : nombre maximal d'exemples dans pour chacun des data sets
dim(int arry[]) : tableau contenant les valeurs de dimensions à tester
RUNS (int) : le nombre d'exécution voulues

Résultats :

```
polysearch(["diabetes.txt",                                    "liver.txt",
"heart-bin.txt","bc-orig.txt","sonar.txt"],[1,2,3,5,7,10],100)
```

dataset	dim	score_train	score_test
bc-orig.txt	1	0.972491	0.966788
	2	0.977179	0.964745
	3	0.981429	0.957226
	5	0.994707	0.945766
	7	0.999689	0.935912
	10	1.000000	0.930146
diabetes.txt	1	0.778029	0.765909
	2	0.795717	0.755584
	3	0.838664	0.738766
	5	0.931319	0.701494
	7	0.983795	0.680130
	10	0.999430	0.669416
heart-bin.txt	1	0.866898	0.835926
	2	0.916944	0.815556
	3	0.962731	0.785185
	5	0.999907	0.747222
	7	1.000000	0.745741
	10	1.000000	0.752407
liver.txt	1	0.715181	0.683478
	2	0.784348	0.728841
	3	0.820688	0.702029
	5	0.909674	0.686522
	7	0.948659	0.652029
	10	0.990362	0.628116
sonar.txt	1	0.865361	0.775714
	2	0.986446	0.825476
	3	1.000000	0.867857
	5	1.000000	0.876429
	7	1.000000	0.876667
	10	1.000000	0.815952

Première remarque qui s'applique pour tous les datasets plus on augmente le degrée plus le score sur le training set augmente. Ce paramètre est donc tout comme le paramètre C un paramètre qui permet une regularization du modèle, le piège étant de ne pas tomber dans l'overfitting encore une fois.

Deuxième observation le passage de nos datasets à des dimensions supérieures n'augmente pas forcément le score sur le test set au contraire. Pour les datasets *diabetes, heart-bin* et *bc-orig* la dimension 1 c'est à dire inchangée offre les meilleurs performances. Pour *liver* le passage à la dimension 2 augmente légèrement le score.
Troisième observation, certains dataset comme *sonar* profitent véritablement du passage à des dimensions supérieures. En effet on peut imaginer que les données de ce dataset sont très peu séparables linéairement car circulaires et donc il est nécessaire de le passer à des dimensions au dessus. Ainsi en passant de la dimensions 1 à 7 on gagne 10% d'accuracy.

III- Computation time, standardization

1- Computation time limit

Dans le cas de SVM avec Ramp Loss ou Hard Margin Loss la présence de variables binaires rend les temps de calculs longs en fonction du nombre d'instances de notre problème. Nous allons déterminer le temps que mets notre solver à trouver la solution optimale à mesure que le nombre d'instances augmente. On fixe comme limite de temps d'une exécution 10 minutes.
Pour cela nous implémentons la méthode *timesolving* qui résout pour différentes taille d'un dataset la classification et donne le temps de calcul dans chacun des cas (ce temps est la moyenne obtenu lors de k exécutions.

```
timesolving(datafile, loss, n_max, time_limit = 600, RUNS = 1)
```

Arguments :

Datafile (str) : nom du fichier du dataset
Loss (str) : nom de la fonction de loss, 'hard_margin' / 'ramp'
n_max (int array[]) : tableau contenant les différentes tailles n à tester du dataset
time_limit(int) : le temps de calcul maximum d'une exécution en seconde
RUNS (int) : le nombre d'exécutions pour chaque valeur de n

Nous allons utiliser pour nos tests successivement différentes taille du dataset ‘diabetes.txt’. Soit n le nombre d’instances sélectionnées au hasard dans l’ensemble du dataset et sur lequel notre modèle va s'entraîner, voici ci-dessous le tableau reprenant les temps de calcul selon la taille du dataset n :

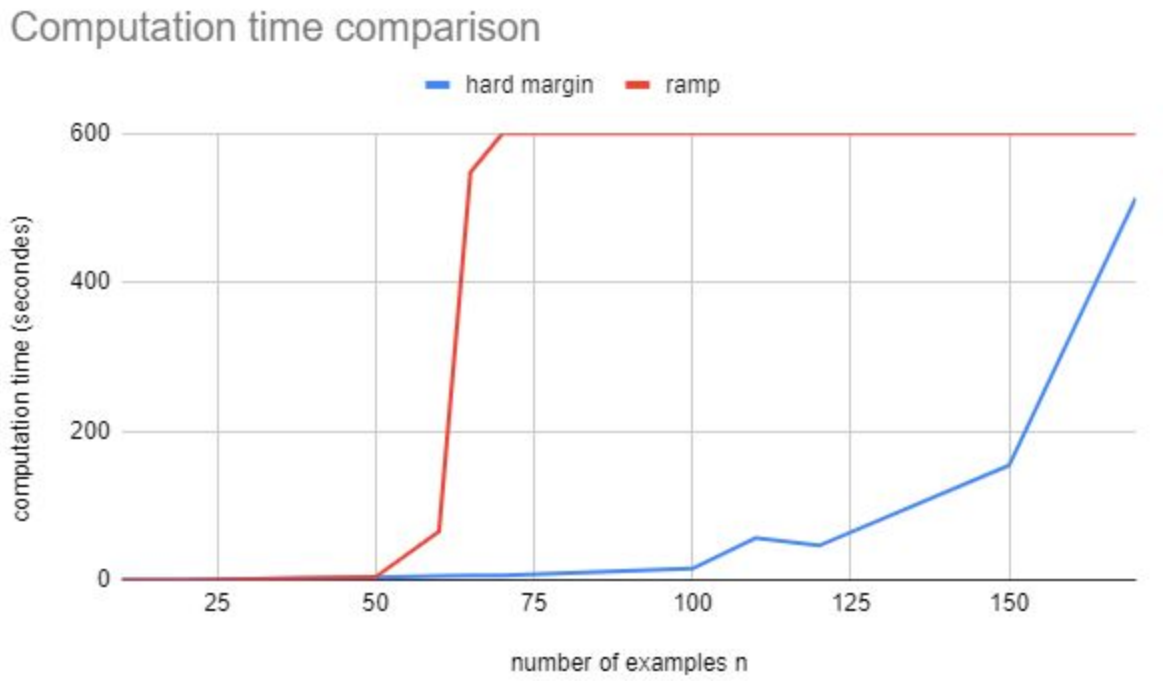
```
times = timesolving('diabetes.txt', 'hard_margin', n_maxHM, time_limit = 600, RUNS = 10)
```

Loss = ‘hard_margin’

	dataset	loss	n	time
0	diabetes.txt	hard_margin	10	0.013632
1	diabetes.txt	hard_margin	20	0.041913
2	diabetes.txt	hard_margin	50	2.857822
3	diabetes.txt	hard_margin	70	6.139929
4	diabetes.txt	hard_margin	100	15.431139
5	diabetes.txt	hard_margin	110	56.447264
6	diabetes.txt	hard_margin	120	46.045151
7	diabetes.txt	hard_margin	150	154.268882
8	diabetes.txt	hard_margin	170	513.062432
9	diabetes.txt	hard_margin	200	600.027250

Loss = ‘ramp’

	dataset	loss	n	time
0	diabetes.txt	ramp	50	3.771310
1	diabetes.txt	ramp	60	65.117663
2	diabetes.txt	ramp	65	548.642234
3	diabetes.txt	ramp	70	600.024410

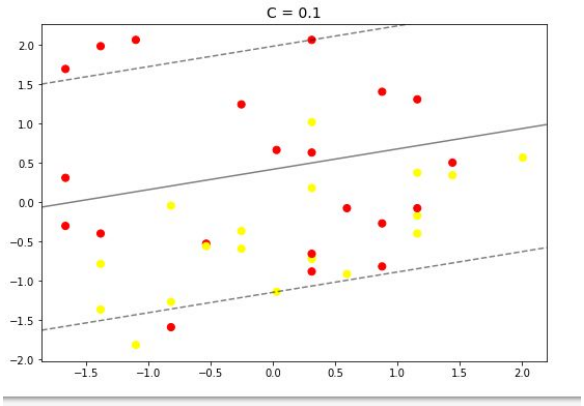


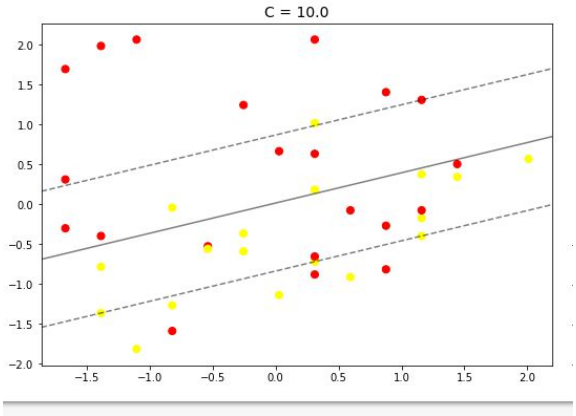
Résultats : Pour la Hard Margin Loss si le dataset possède plus de 170 instances alors le temps de calcul explose et dépasse les 10 minutes.
Pour la Ramp Loss les temps de calcul explosent plus vite puisqu'il suffit que le dataset dépasse 70 instances pour que le temps de calcul soit supérieur à 10 minutes afin de trouver la solution optimale.

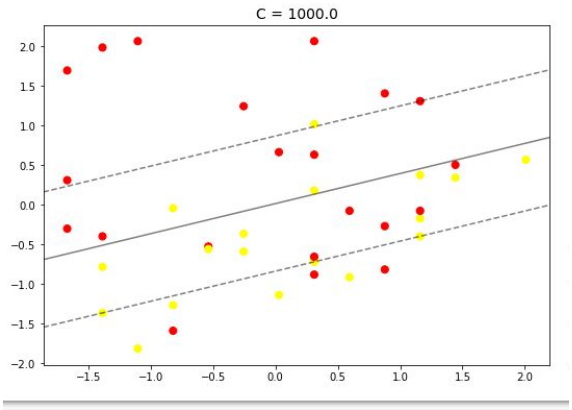
2- Standardization impact

Ci dessous on visualise la decision boundary ainsi que les marges de notre SVM hinge loss.
On représente seulement les 2 premières colonnes du dataset diabetes. En jaune les personnes diabétiques en rouge celles qui ne le sont pas. On compare pour 3 valeurs de et pour des données normalisées (à gauche) et non normalisées (à droite).

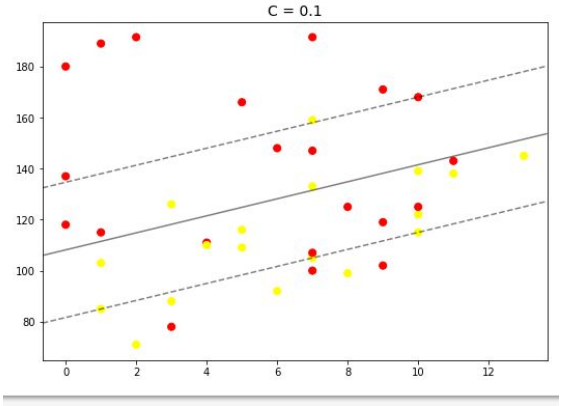
Standardized data

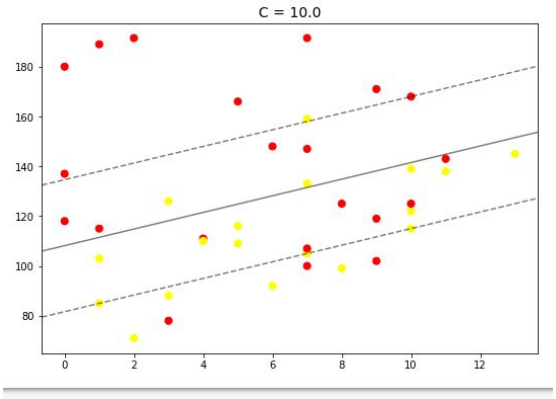


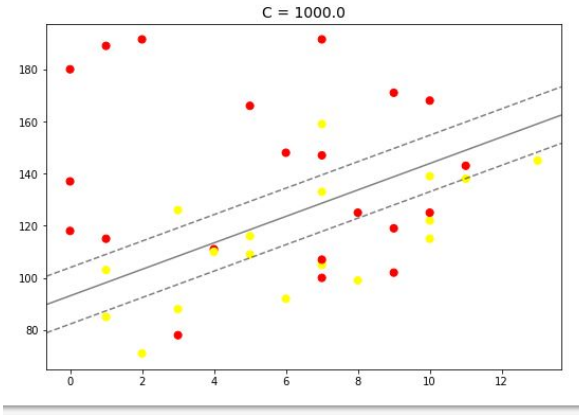




Non-Standardized data







Dans les deux cas plus C est grand plus la distance entre les marges est étroite. A l'inverse les marges sont espacées lorsque C est petit. Cependant, on remarque que les données normalisées produisent des marges plus cohérentes avec la valeur de C.
Aussi on remarque que dans le cas des données non normalisées la valeur du coefficient C a un impact beaucoup plus significatif sur l'hyperplan. La visualisation est faite à l'aide de note méthode *visualize_svm.*

IV- Annexe

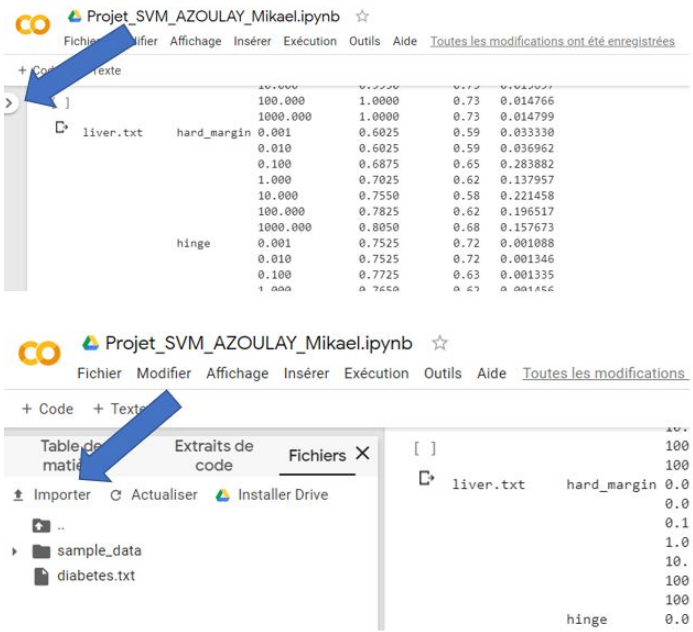
1- Source code

Mon code python est accessible au lien ci dessous (lien google colaboratory) :
https://colab.research.google.com/drive/1UGwDTm5eF4KAECzTcLEr7aFkFzPwyUd1

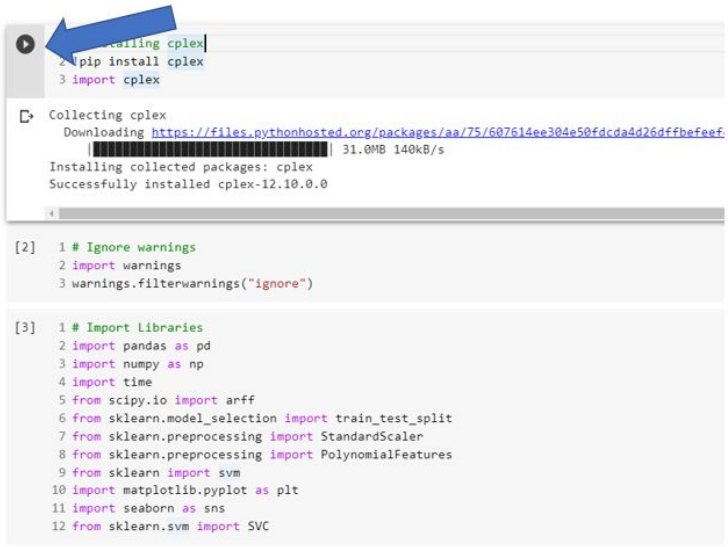
Il est possible de le faire tourner directement sur navigateur internet en créant une copie du google colab. Les svm ramp loss et hard_margin loss ont été implémenté à l'aide de CPLEX Python API. Le svm hinge loss à l'aide la librairie python Scikit-Learn. La visualisation à l'aide des librairies python Searborn et Matplotlib. Les différents gridsearch et data processing via Numpy, Pandas ainsi que Scikit-learn.

2- Utilisation

On fait une copie du google colab puis on importe les datasets comme ci dessous.



Puis on peut exécuter une à une les cellules de codes et observer les résultats. On peut également ajouter de nouvelles cellules de codes pour réaliser d'autres exécutions.



3- Pour aller plus loin…

Dual problem & kernel trick

Augmenter les dimensions du datasets revient à augmenter le nombre de features et donc le temps de calcul. Une façon de de réduire ces temps de calcul à rallonge est d'exprimer le "dual" de notre problème. Une fois que l'on travail sur le dual on peut appliquer le "kernel trick" qui nous permet de simuler le fait d'avoir ajouter des features sans avoir à les ajouter réellement.
kernel trick : http://www.cs.cmu.edu/~guestrin/Class/10701-S07/Slides/kernels.pdf

Gaussian rbf kernel and other kernels

Les svm vus sont linéaires (ont un noyau/kernel linéaire) ou polynomiaux lorsqu'on les passe à des dimensions supérieurs mais il existe d'autres types de svm plus adaptés à d'autres formes de problèmes. Par exemple les string kernel sont utilisés pour classifier des séquences d'ADN ou des documents textes.
rbf kernel : https://en.wikipedia.org/wiki/Radial_basis_function_kernel