Rapport de TP 2: Apprentissage supervisé

Université d'Avignon - Décembre 2024
Réalisé par : Mounir Mouterfi, Nadjim Ait Meddour
Proposé par : Juan-Manuel Torres

Introduction

Ce rapport présente les étapes et résultats d'une expérimentation sur l'apprentissage supervisé avec l'algorithme du perceptron. L'objectif principal est d'évaluer les performances de cet algorithme en termes d'erreur d'apprentissage et d'erreur de généralisation sur des ensembles de données spécifiques, tout en analysant la stabilité des exemples testés.

1 PARTIE I

1.1 1. Les données

Nous commençons par importer les bibliothèques nécessaires pour le traitement des données, les calculs mathématiques et la visualisation des résultats. Nous avons utilisé les bibliothèques suivantes : pandas, numpy, et matplotlib.

Les données sont divisées en quatre fichiers CSV distincts, chaque fichier contenant un ensemble spécifique pour l'entraînement et le test. Les ensembles sont constitués de deux classes de données :

- Mines (+1)
- Rocks (-1)

Chaque fichier contient des exemples sous forme de vecteurs de 60 dimensions. Les données sont donc traitées comme suit : chaque ligne est transformée en un tableau de 60 dimensions, puis les exemples des deux classes (Mines et Rocks) sont combinés.

Ainsi, y_{train} est le vecteur des étiquettes : +1 pour Mines, -1 pour Rocks. Le résultat est un tableau contenant les vecteurs d'entraînement et leurs étiquettes.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# Charger les données
data_trainMines = pd.read_csv("train_dataM.csv")
data_testMines = pd.read_csv("test_dataM.csv")
data_trainRocks = pd.read_csv("train_dataR.csv")
data_testRocks = pd.read_csv("test_dataR.csv")
```

```python
# Préparer les données d'entrainement
X_train_mines = data_trainMines['Values'].apply(lambda x: np.array([float(i) for i in x.split()]))
X_train_rocks = data_trainRocks['Values'].apply(lambda x: np.array([float(i) for i in x.split()]))
X_train = pd.concat([X_train_mines, X_train_rocks], ignore_index=True)
y_train = np.concatenate([np.ones(len(X_train_mines)), -1 * np.ones(len(X_train_rocks))]) #les etiquette de l'ense
X_train = pd.DataFrame(X_train.tolist())  # Convertir en DataFrame
X_train['Label'] = y_train
```

```
[Running] python -u "m:\master_1_IA\Apprentissage_supervisé\TP2\TP1.py"
        0       1       2       3  ...      57      58      59  Label
0    0.0491  0.0279  0.0592  0.1270  ...  0.0140  0.0332  0.0439    1.0
1    0.1313  0.2339  0.3059  0.4264  ...  0.0127  0.0178  0.0231    1.0
2    0.0629  0.1065  0.1526  0.1229  ...  0.0057  0.0113  0.0131    1.0
3    0.0587  0.1210  0.1268  0.1498  ...  0.0101  0.0228  0.0124    1.0
4    0.0162  0.0253  0.0262  0.0386  ...  0.0100  0.0048  0.0019    1.0
..      ...     ...     ...     ...  ...     ...     ...     ...    ...
99   0.0274  0.0242  0.0621  0.0560  ...  0.0161  0.0220  0.0173   -1.0
100  0.0126  0.0519  0.0621  0.0518  ...  0.0034  0.0114  0.0077   -1.0
101  0.0260  0.0192  0.0254  0.0061  ...  0.0008  0.0044  0.0077   -1.0
102  0.0459  0.0437  0.0347  0.0456  ...  0.0085  0.0117  0.0056   -1.0
103  0.0291  0.0400  0.0771  0.0809  ...  0.0081  0.0139  0.0111   -1.0

[104 rows x 61 columns]
```

1.2 Implémentation de l'Algorithme du Perceptron

Nous avons choisi une version batch de l'algorithme du perceptron, où les poids sont mis à jour après chaque passage complet sur l'ensemble de données. L'algorithme renvoie les poids, le biais, et une liste des erreurs pour chaque itération.

Poids et biais obtenus après l'exécution de l'algorithme :

Les coefficients des poids w (60 dimensions) sont :

```
**********les N+1 poids de perceptron**********
Poids appris (W):
0     21.58460
1     40.87238
2     24.54852
3     43.18358
4     51.72366
5     31.29936
6     -5.90188
7    -51.98000
8      9.63128
9     28.42616
10    58.18298
11    23.43740
12    15.84420
13    18.23384
14    13.07906
15   -62.14252
```

Le biais obtenu est b = -64,19.

1.3 3. Évaluation des Performances

Deux métriques principales ont été calculées :

- **Erreur d'apprentissage** (E_a) : Le pourcentage d'exemples mal classés dans l'ensemble d'entraînement.
- **Erreur de généralisation** (E_g) : Le pourcentage d'exemples mal classés dans l'ensemble de test.

```
****************************************Apprentissage sur train*************************************
Erreur d'apprentissage (Ea): 19.23%
Erreur de g□n□ralisation (Eg): 26.92%
```

1.4 4. Calcul des Stabilités

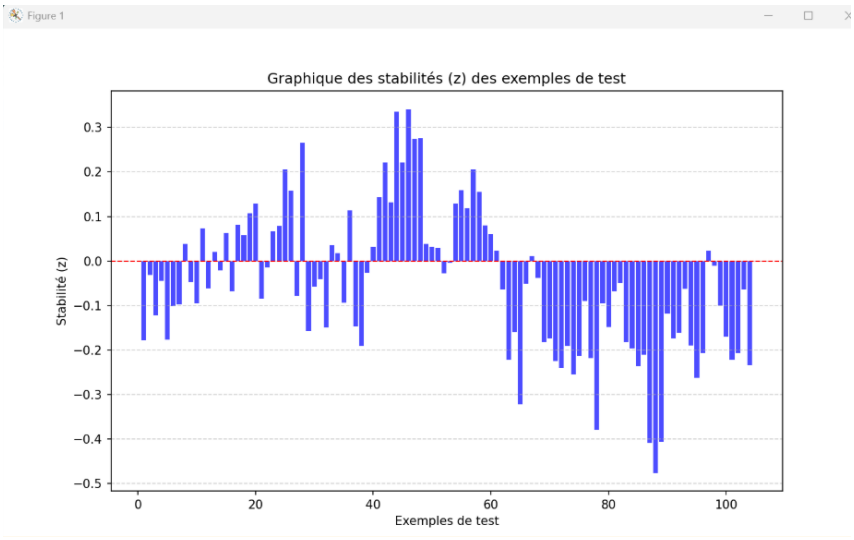
La stabilité mesure la distance signée des exemples par rapport à l'hyperplan séparateur. La formule utilisée pour calculer la stabilité est la suivante :

$$\text{Stabilité} = \frac{y_i \cdot (\mathbf{w}^T \mathbf{x_i} + b)}{\|\mathbf{w}\|}$$

Les résultats des stabilités sont visualisés sous forme de graphique à barres :

Nous avons également effectué un apprentissage en inversant les ensembles d'entraînement et de test. Voici les résultats obtenus :

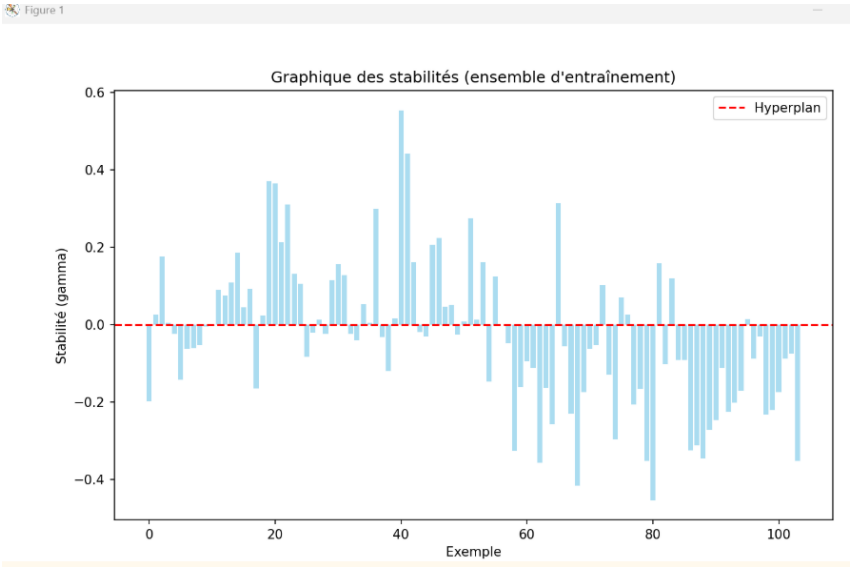
```
Stabilit□s des exemples de test (gamma):
Exemple 1: z = -0.1779
Exemple 2: z = -0.0314
Exemple 3: z = -0.1222
Exemple 4: z = -0.0441
Exemple 5: z = -0.1768
Exemple 6: z = -0.1014
Exemple 7: z = -0.0980
Exemple 8: z = 0.0381
Exemple 9: z = -0.0471
Exemple 10: z = -0.0950
Exemple 11: z = 0.0730
Exemple 12: z = -0.0615
Exemple 13: z = 0.0209
Exemple 14: z = -0.0217
Exemple 15: z = 0.0632
Exemple 16: z = -0.0681
```



Ensuite pour l'apprentissage sur 'test' , on juste inversé les ensembles, et voici les résultats :

```
****************************************Apprentissage sur test***************
Erreur d'apprentissage sur l'ensemble de test (Ea_test) : 16.35%
Erreur de g□n□ralisation sur l'ensemble d'entra□nement (Eg_test) : 29.81%
*********************************les N+1 poids de perceptron*****************
Poids appris (W):
0      6.00596
1    -10.07028
2     -6.65010
3     10.47710
4      4.37626
5    -27.87698
6    -51.72584
7     -1.55444
8     41.34134
9     20.49298
10    68.00620
11    49.18570
12     5.57330
13   -11.98104
14   -33.80626
15    14.66470
16     1.92934
```

Le biais obtenu est b = -31,80. et enfin le graphique :



2 PARTIE II : Mise en œuvre et analyse de l'algorithme Pocket

2.1 Initialisation des poids

Trois types d'initialisation des poids ont été comparés :

- **Initialisation par défaut** : Les poids sont initialisés à 0.
- **Initialisation aléatoire** : Les poids sont tirés aléatoirement à partir d'une distribution normale.
- **Initialisation Hebb** : Une règle d'apprentissage simple qui met à jour les poids en fonction des données et des étiquettes.

Chaque méthode d'initialisation prépare un vecteur de poids initial prêt pour l'apprentissage.

2.2 Implémentation de l'algorithme Pocket

L'algorithme Pocket est une extension du perceptron classique. Il garde les meilleurs poids et biais à chaque itération si une solution avec moins d'erreurs est trouvée. L'apprentissage s'arrête lorsque l'une des deux conditions suivantes est remplie :

- L'erreur totale atteint un seuil maximum (max_errors).
- Le nombre d'itérations atteint un maximum (max_iter).

Voici les résultats obtenus avant l'échange des ensembles de données.

2.2.1 Initialisation par défaut

Les résultats obtenus avec l'initialisation par défaut sont les suivants :

```
[Running] python -u "m:\master_1_IA\Apprentissage_supervisé\TP2\TP2_Partie2.py"
****************Comparaison des initialisations avant echange des ensembles ****************
Erreur d'apprentissage avec initialisation par d♦faut (Ea) : 19.230769230769234
Erreur de generalisation avec initialisation par d♦faut (Eg) : 17.307692307692307
les poids finaux avec l'initialisation par d♦faut :
[ 21.18568  40.15602  24.2678   42.47452  50.9401   30.67786  -5.45008
 -50.91814   9.37402  27.80426  57.04478  23.19116  15.85216  18.328
  12.97202 -60.0886  -20.77832  26.0256   10.59398 -11.24948   7.19182
  19.1113   -5.83048  23.66168   1.63664   4.1862   -4.90164  -7.22384
  20.29576  35.26488 -72.40272  25.58674   4.88168 -13.12702   1.92714
 -30.53858 -24.9685   34.85716  13.85962 -45.76814  11.81414  -6.51736
  -2.11188  19.01674  52.93066  37.62628  33.40512  57.66254  55.30354
  -3.1558    7.49638   7.20662   3.15992   7.56244  -2.7426   -1.03878
  -3.77434   4.45066   2.71112   0.56524]
le biais final avec l'initialisation par d♦faut :   -60.799999999999955
```

2.2.2 Initialisation aléatoire

Les résultats obtenus avec l'initialisation aléatoire sont les suivants :

```
Erreur d'apprentissage avec initialisation al♦atoire (Ea) : 18.269230769230766
Erreur de generalisation avec initialisation al♦atoire (Eg) : 17.307692307692307
les poids finaux avec l'initialisation al♦atoire :
[ 22.25685486  39.08003744  22.6218652   43.63312754  49.04264632
  30.39710114  -4.70033833 -50.37502534  10.17333037  28.21949166
  56.84712256  23.89639492  16.23178337  18.47873149  11.87124366
 -59.38663053 -21.42790939  24.76037238  11.12915397 -11.68600948
   6.74523671  19.16138831  -6.04432246  22.75760666   1.84053072
   4.37019097  -5.58889855  -5.06252297  19.82934501  34.25123253
 -71.51184617  24.9373015    4.81383062 -12.97179915   1.295538
 -30.21400849 -25.64016241  34.52112114  13.53234794 -44.42406033
  10.53683251  -6.26628147  -2.08418277  19.54631047  52.45822021
  36.84833585  33.46401026  55.21389424  54.30272315  -2.58577013
   7.01689571   7.82036632   3.16494758   7.24058341  -1.37381332
  -2.43372763  -5.10481944   4.26971745   3.40338539   1.06110512]
le biais final avec l'initialisation al♦atoire :   -59.80000000000014
```

2.2.3 Initialisation de Hebb

Les résultats obtenus avec l'initialisation de Hebb sont les suivants : et enfin le graphique

```
Erreur d'apprentissage avec initialisation de Hebb (Ea) : 18.269230769230766
Erreur de generalisation avec initialisation de Hebb (Eg) : 18.269230769230766
les poids finaux avec l'initialisation de Hebb :
[ 18.89724  36.24428  22.33452  38.20242  47.15924  28.72016  -3.90624
 -45.07346  11.19234  28.70394  56.53558  24.43156  17.93628  18.66336
  11.26736 -56.8575  -20.93698  21.37166  11.47086 -11.72808   6.87408
  18.89162  -7.32602  19.19586   1.21684   4.13644  -2.70426  -1.2915
  22.59186  32.27616 -67.89218  19.6281    1.73634 -13.85442   0.91566
 -33.29448 -26.34342  32.97892  12.30206 -43.67396   9.86072  -4.86892
  -0.79604  21.61514  52.40446  36.0512   32.61096  53.89798  50.63474
  -2.61818   6.66848   6.2538    2.83658   6.7401   -2.36006  -0.8406
  -3.28434   4.0375    2.53212   0.48494]
le biais final avec l'initialisation de Hebb :   -57.600000000000115
```

:

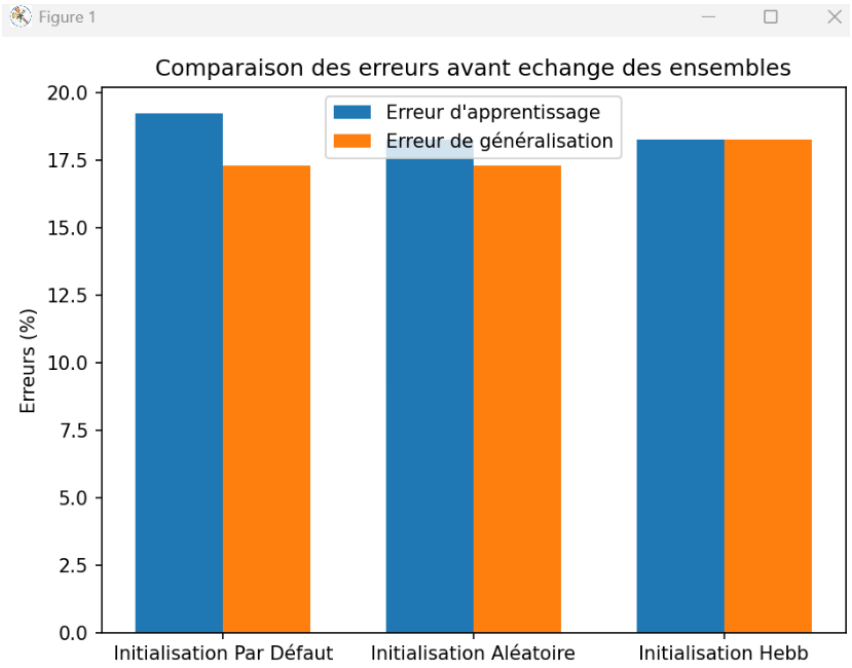


Figure 1: Graphique des erreurs d'apprentissage et de généralisation avant l'échange des ensembles

2.3 Analyse des résultats

L'initialisation aléatoire semble être la meilleure option, offrant un bon équilibre entre l'erreur d'apprentissage et l'erreur de généralisation. L'initialisation de Hebb peut conduire à un surapprentissage, tandis que l'initialisation par défaut montre une moindre performance d'apprentissage, bien que la généralisation soit relativement bonne.

2.4 Après échange des ensembles

Après avoir échangé les ensembles d'entraînement et de test, les mêmes étapes ont été répétées. Voici les résultats obtenus.

2.4.1 Initialisation par défaut

Les résultats obtenus avec l'initialisation par défaut après échange des ensembles sont les suivants :

```
****************Comparaison des initialisations apres echange des ensembles ***************
Erreur d'apprentissage avec initialisation par d�faut (Ea) : 7.6923076923076925
Erreur de generalisation avec initialisation par d�faut (Eg) : 21.153846153846153
les poids finaux avec l'initialisation par d�faut :
[  4.76336  -7.62482  -4.95178   9.2044    4.2896  -20.48122 -39.09358
   0.20226  37.69896  24.31624  62.61272  49.28586   8.48966 -11.33224
 -29.46326   6.95382  -0.78818 -16.576    -1.23052  10.77422  12.6641
   6.0312   19.09998  17.24222 -23.78138 -25.469     2.6266   17.8573
   0.16678  -1.87686 -27.04548   5.9527   -6.96802 -18.78626   1.06252
 -20.44424 -45.51452  -3.38208  28.63974 -15.7954    5.83206  11.14144
  40.33352  54.09126  34.05558   9.71764   6.4904   15.4665    0.79378
  -2.8468    5.71958   4.4321    1.63898   1.19824   0.60788   1.37294
   1.80076   1.50916   4.87484   0.76974]
le biais final avec l'initialisation par d�faut :   -27.600000000000115
```

2.4.2 Initialisation aléatoire

Les résultats obtenus avec l'initialisation aléatoire après échange des ensembles sont les suivants :

```
Erreur d'apprentissage avec initialisation al�atoire (Ea) : 6.730769230769231
Erreur de generalisation avec initialisation al�atoire (Eg) : 21.153846153846153
les poids finaux avec l'initialisation al�atoire :
[  5.2890853   -8.88147984  -5.43897062   9.08695306   4.12924442
 -21.66280617 -40.56793305   0.56168868  39.51865485  23.19234451
  61.29933322  49.59099684   7.76791913 -10.70730064 -30.30138723
   8.44155343   0.9666288  -17.11719154  -1.9650585   10.63709635
  11.49840449   6.00693477  18.05002026  17.64748389 -23.7634221
 -26.20884861   3.82929111  17.17576107   0.13378884  -1.27023347
 -26.49370287   7.92697588  -6.28561191 -19.65117518   2.75787846
 -20.06170376 -46.45279209  -4.31048501  28.75866859 -17.53667045
   7.41558358  10.39017579  40.8190619   54.87907418  35.31912637
   9.88699842   5.93993726  15.75463721   0.8788669   -2.51168844
   7.27186967   4.02316779   1.49036217   0.19584887   0.34278103
   1.63272382   2.60174684   1.11308303   3.71444102   0.53405416]
le biais final avec l'initialisation al�atoire :   -28.000000000000107
```

2.4.3 Initialisation de Hebb

Les résultats obtenus avec l'initialisation de Hebb après échange des ensembles sont les suivants :

```
Erreur d'apprentissage avec initialisation de Hebb (Ea) : 6.730769230769231
Erreur de generalisation avec initialisation de Hebb (Eg) : 21.153846153846153
les poids finaux avec l'initialisation de Hebb :
[  5.0779   -8.18628  -5.52888  10.064     4.30088 -23.76494 -44.48942
  -0.18166  38.61656  22.51624  65.26888  48.6818    7.38868 -10.38394
 -29.50902  10.07338   0.96312 -16.95162  -1.8088    9.82352  12.9142
   5.0385   18.74492  18.76042 -24.93664 -25.25714   4.25992  19.02344
   0.47816  -0.08824 -25.01034   8.58772  -4.06906 -20.09218   4.0204
 -17.86604 -49.58764  -4.49976  31.87876 -16.68742   6.47644  12.94508
  42.68516  53.78112  31.9593    7.5576    6.21948  17.11376   0.88504
  -3.11706   6.61172   5.09192   1.79806   1.36556   0.8271    1.45808
   2.00708   1.7177    5.50308   0.87586]
le biais final avec l'initialisation de Hebb :   -35.000000000000206
```

et enfin le graphique :

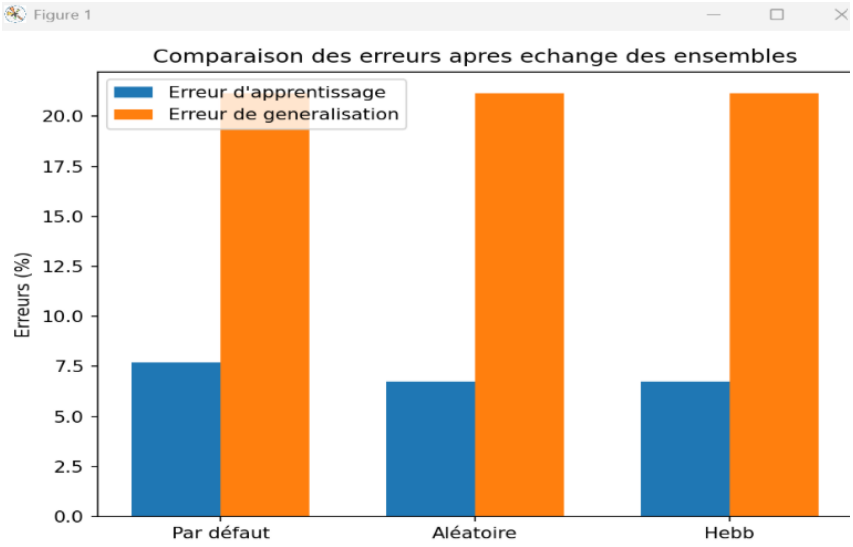


Figure 2: Graphique des erreurs d'apprentissage et de généralisation après l'échange des ensembles

2.5 Observation

L'initialisation de Hebb et l'initialisation aléatoire offrent des avantages sur l'apprentissage par rapport à l'initialisation par défaut. Cependant, toutes les méthodes mènent à des performances similaires en termes de généralisation. Si l'on privilégie un apprentissage plus rapide ou des poids moins extrêmes, l'initialisation de Hebb semble être le choix optimal.

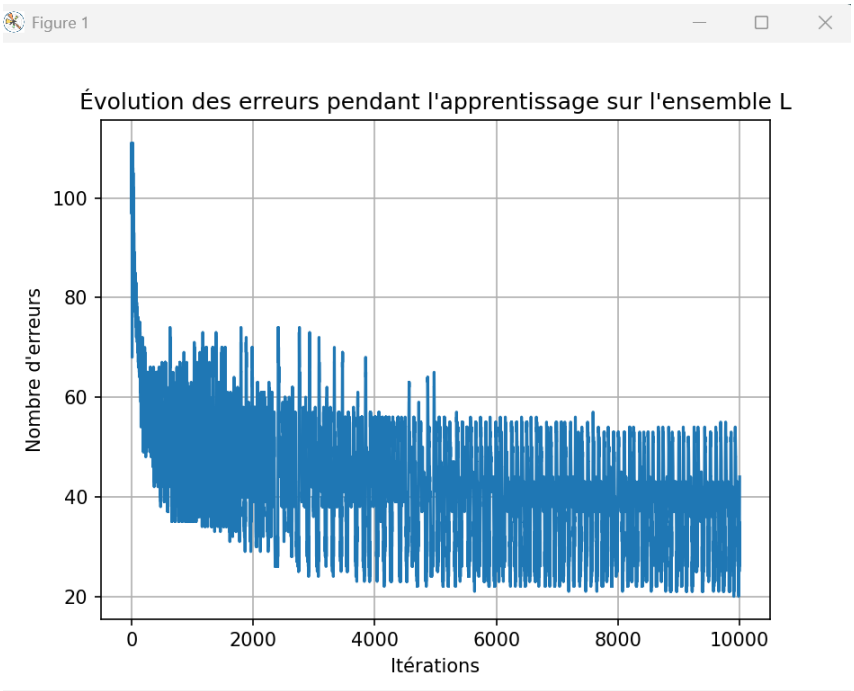
3 PARTIE III

Dans cette partie, nous avons fusionné les ensembles d'entraînement et de test pour créer un nouvel ensemble L=train+test. L'objectif est d'appliquer l'algorithme du perceptron pour déterminer si cet ensemble L est linéairement séparable (LS). Un ensemble est considéré comme LS si un hyperplan peut séparer parfaitement les classes sans erreur.

L'algorithme qu'on a utilisé est une version Pocket du perceptron celui de la partie 2).

Voici les résultat obtenus (pour la question 5):

```
[Running] python -u "m:\master_1_IA\Apprentissage_supervisé\TP2\TP2_Partie3.py"
les poids finaux avec uniquement avec l'initialisation par d♦faut :
[ 169.22542   57.36336 -177.67192  271.94122   64.36856   29.3707
 -187.55024 -156.70024  148.72348  -56.32008  220.47978   65.88606
  -15.01124  -18.13092   40.4963   -40.9572  -109.494     99.958
  -17.80946   13.9163   -20.47494   56.2989   -25.67352  139.54562
  -63.19868  -28.50204   40.67622  -22.9343   -16.18448  177.65066
 -290.73326  147.37658   12.23826  -79.77448  108.7729  -121.95902
  -62.00168   19.09048  108.74046 -177.13324   48.04008    3.14194
   58.99722   55.53328   89.73286   -5.00692   45.811    286.87296
  246.53056 -129.28626  133.5328   116.61164   64.7653    71.39512
   -6.88882   -5.96958  -37.67306   76.11708   78.41154   49.78048]
le biais final avec l'initialisation uniquement aussi par d♦faut :   -142.59999999999522
L'ensemble L n'est pas lin♦airement s♦parable (non LS).
```



L'ensemble L est-il LS ?

D'après les résultats obtenus :

- Le nombre final d'erreurs n'est pas égal à zéro (`final_error_count` $\neq 0$).
- Cela indique que l'ensemble L n'est pas linéairement séparable (**non LS**).

Justification : L'incapacité de l'algorithme à trouver une séparation parfaite après un grand nombre d'itérations (`max_iter` = 10000) et des mises à jour des poids montre que certains exemples ne peuvent pas être correctement classifiés par un hyperplan unique.

Partie IV : Early Stopping

Enoncé :

L'ensemble L (Train + Test) contient 208 patrons. Nous avons divisé cet ensemble en trois parties de manière aléatoire :

- L_A : Ensemble d'apprentissage (50 % des données),
- L_V : Ensemble de validation (25 % des données),
- L_T : Ensemble de test (25 % des données).

L'algorithme a été entraîné sur L_A, validé sur L_V, et testé sur L_T avec une stratégie d'**Early Stopping**. L'expérience a été répétée plusieurs fois pour obtenir les statistiques moyennes sur les erreurs d'apprentissage (E_a), de validation (E_v), et de test (E_t).

Résultats :

Après avoir exécuté 10 répétitions, nous obtenons les résultats suivants :

- **Erreur moyenne d'apprentissage** (E_a) : 10.48 %,
- **Erreur moyenne de validation** (E_v) : 27.88 %,
- **Erreur moyenne de test** (E_t) : 26.35 %.

```
[Running] python -u "m:\master_1_IA\Apprentissage_supervisé\TP2\TP2_Partie3_6.py"

--- R◆sultats Early Stopping ---
Moyenne des erreurs d'apprentissage (Ea) : 10.48%
Moyenne des erreurs de validation (Ev)  : 27.88%
Moyenne des erreurs de test (Et)        : 26.35%
```

Analyse graphique :

La figure ci-dessous illustre les erreurs E_a, E_v, et E_t pour chaque répétition.

- **Ligne bleue** : Évolution de l'erreur d'apprentissage (E_a),
- **Ligne orange** : Évolution de l'erreur de validation (E_v),
- **Ligne verte** : Évolution de l'erreur de test (E_t).

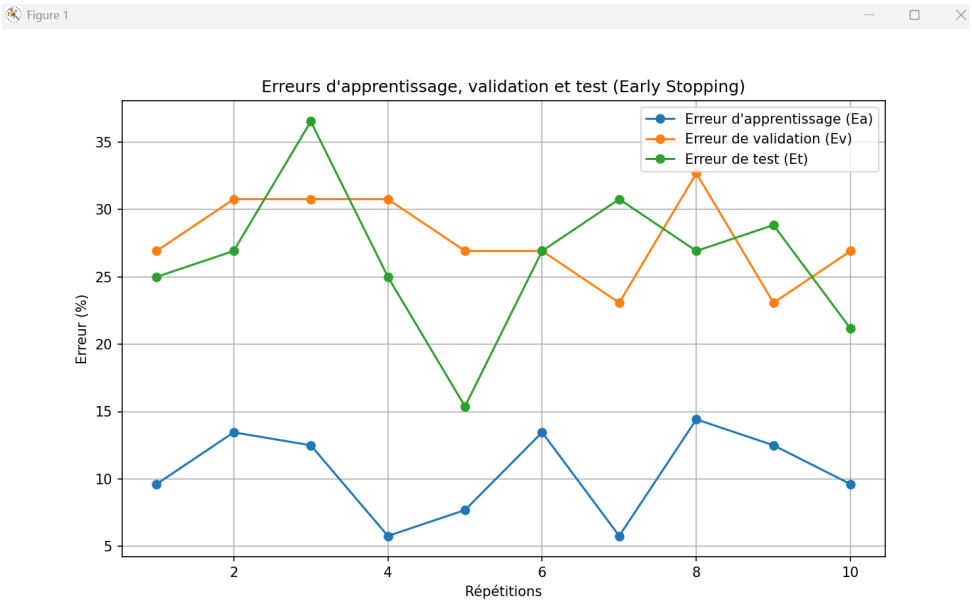


Figure 3: Évolution des erreurs d'apprentissage, validation et test (Early Stopping).

Observations :

1. **Erreur d'apprentissage (E_a)** :
   - Elle reste relativement basse ($\approx 10\,\%$) à travers toutes les répétitions. Cela montre que l'algorithme apprend efficacement sur L_A.
2. **Erreur de validation (E_v)** et **Erreur de test (E_t)** :
   - Ces erreurs sont significativement plus élevées ($\approx 27 - 28\,\%$).
   - L'écart par rapport à E_a pourrait indiquer une certaine **sous-généralisation** ou que L_A ne capture pas bien la complexité des données globales.

Donc l'algorithme semble bien converger sur L_A, mais les performances sur L_V et L_T montrent qu'il y a place à amélioration.

Conclusion

Ce TP a permis de mettre en œuvre et d'analyser les performances des algorithmes du perceptron et de Pocket sur des ensembles de données variés. Les résultats ont montré les limites du perceptron sur des ensembles non linéairement séparables et l'apport de l'algorithme Pocket en termes de robustesse. Enfin, l'utilisation de la stratégie d'Early Stopping a souligné l'importance de l'équilibre entre apprentissage et généralisation.