Projet Intelligence Artificielle

Semestre 1

Badr Ouarafana / Vincent Joulain

Année 2022 - 2023

Sommaire

Travaux Pratiques 1 - Kppv & Perceptron

K plus proches voisins

L'algorithme Kppv est le plus simple à implémenter. Il consiste à récupérer les données de tous les points disposés sur le repère *(chacun ayant une classe définie)* et d'utiliser ces derniers dans l'objectif de prédire les nouveaux points posés.

Supposons le point A venant d'être posé, nous cherchons les K points les plus proches parmi tous les points *(ici avec le théorème de pythagore)*. Admettons K = 3 et les points les plus proches X, Y, et Z respectivement de classe 0, 1 et 1.

Nous obtenons la prédiction suivante : **le point A est de classe 1.**

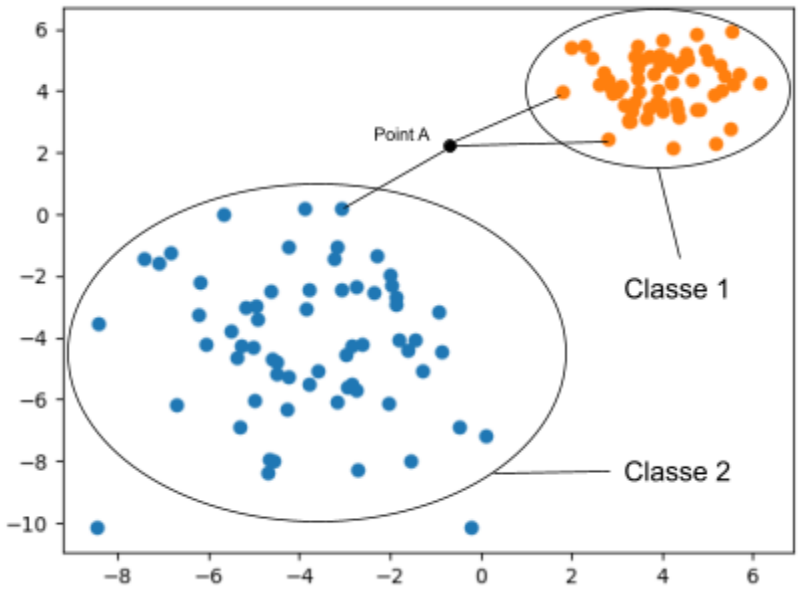


Figure 1.1.1 *- Séparation entre K plus proches voisins*

Afin de déterminer la classe d'un point, on va stocker la distance entre ce point et tous les autres sous forme d'un objet JSON :

```
tabDistanceEntrePoint.append({
            'indice point test' : i,
            'indice point appren' : j,
            'distance' : dist(pointA=pointTest, pointB=pointAppr)
        })
```

Il ne nous reste alors plus qu'à itérer sur cet objet et récupérer la classe majoritaire parmi les **k** plus proches voisins.

```
for k in range (0 , K):
        if oracle[tabDistanceEntrePoint[k]['indice point appren']] == 1 :
            nombreDeClassA +=1
        else:
            nombreDeClassB += 1
    clas.append(0 if nombreDeClassA > nombreDeClassB else 1 )
```

Perceptron

Le perceptron est un neurone *(pouvant faire partie d'un réseau)* qui prend en entrée des valeurs, chacune associée à un poids, sur lesquelles il va effectuer une prédiction. Si la prédiction est mauvaise, les poids de chacune des valeurs *(ainsi que le biais choisi)* sont corrigés et le perceptron repart à nouveau avec les valeurs suivantes.

Le processus est répété ainsi en fonction du nombre d'itérations choisies sur toute la base de données jusqu'à obtenir une précision satisfaisante. La finalité étant d'obtenir une droite linéaire scindant les deux classes déterminées :

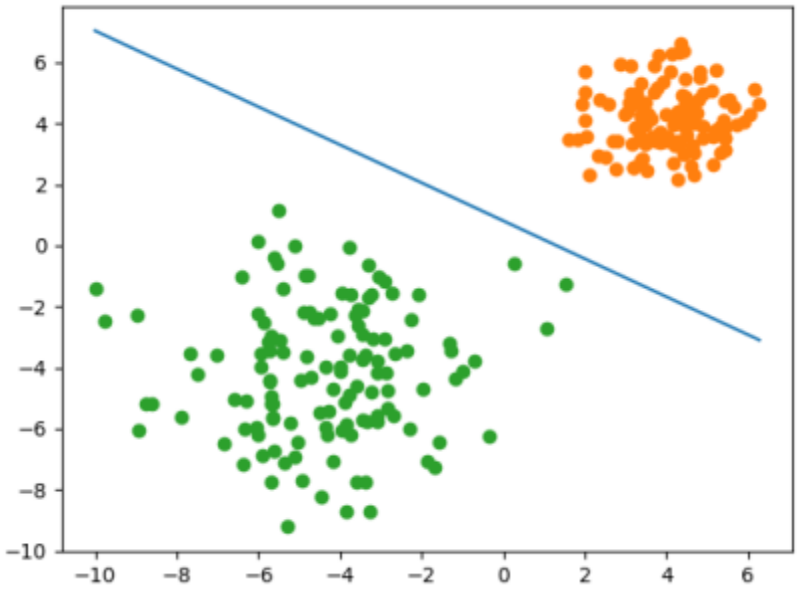


Figure 1.2.1 *- Séparation des classes 1 et 2 par le perceptron*

Les poids sont choisis aléatoirement et l'apprentissage est effectué 100 fois. Le biais est stocké dans le vecteur contenant les coordonnées x et y des différents points.

```
for it in range (0, nbApprentissages):
    for i in range (0 , len(x[0])):
        point = [1 , x[0][i] , x[1][i] ]
        prediction = perceptron(point, w, active)
```

Si la prédiction s'avère mauvaise, les poids, le biais ainsi que le vecteur d'erreurs cumulée sont mis à jour :

```
if prediction != yd[i] :
    for j in range (0 , len(w)):
        w[j] = w[j] + (yd[i] - prediction) * learning_rate * point[j]
    erreur += math.pow(yd[i]-classe, 2)
mdiff.append(erreur)
```

Le problème n'étant pas très difficile, les erreurs sont rapidement corrigées et le perceptron se trompe peu une fois un certain seuil dépassé. Nous pouvons voir ici qu'il s'agit de la quinzième itération :

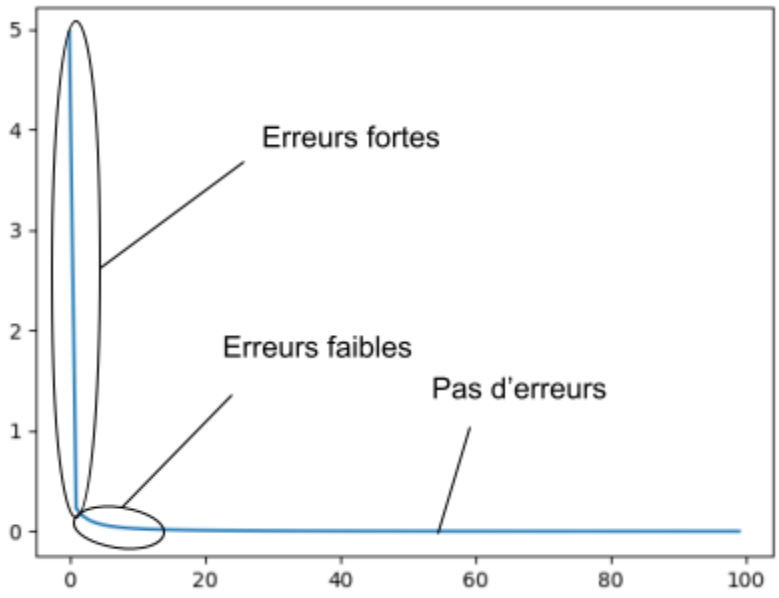


Figure 1.2.2 *- Evolution du nombre d'erreurs en fonction du nombre d'itérations*

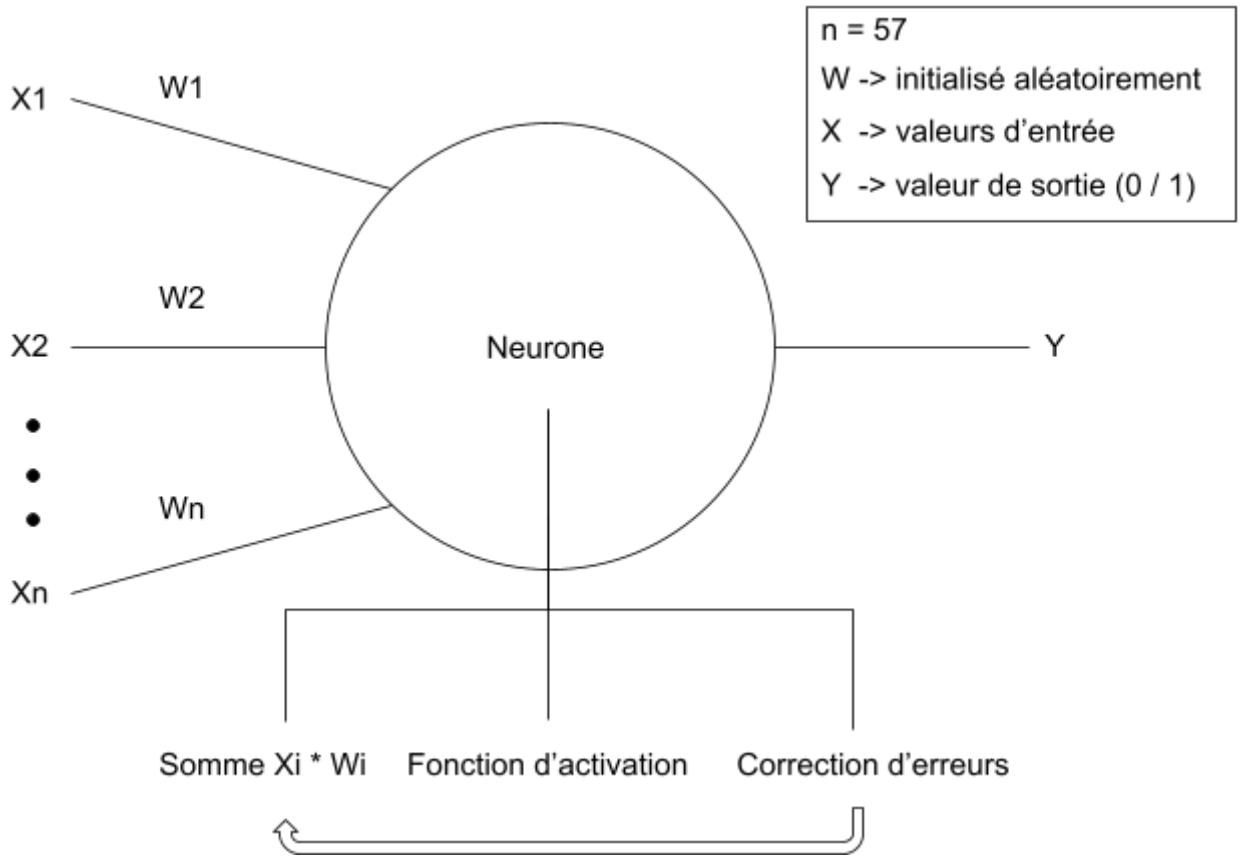
Travaux Pratiques 2 - Anti-Spam

Modèle 1 - sans couches cachées

Le principe d'un modèle sans couches cachées est très similaire au neurone unique sur lequel nous avons travaillé précédemment *(perceptron)*. Toutefois, l'utilisation de la librairie Keras va nous simplifier grandement la tâche.

Notre neurone va prendre en entrée les 57 variables trouvées dans le csv fourni et appliquer la fonction d'activation sigmoid afin d'obtenir une valeur de sortie binaire *(0 ou 1).*

Notre neurone peut être modélisé de la manière suivante :



Une fois toutes les valeurs extraites des fichiers csv nous obtenons les variables :

- Creation Input Values
- Creation Output values
- Prediction Input Values
- Prediction Output Values

Il ne nous reste plus qu'à construire notre modèle Keras, pour ce faire nous commençons par lui dire que nous allons suivre la construction du réseau de manière séquentielle *(ajout couche par couche à notre modèle, ici une couche suffira)*

```
model = Sequential()
model.add(Dense(1, input_shape=(57,), activation="sigmoid"))
```

Sur la deuxième ligne nous définissons une couche composée **d'un neurone**, prenant **57 variables** d'entrée, utilisant la fonction d'activation **sigmoid**.

Il nous faut maintenant compiler notre modèle en spécifiant la fonction de perte ainsi que les variables que nous souhaitons étudier *(dans notre cas la précision de notre neurone)*.

```
model.compile(loss="binary_crossentropy", optimizer="adam",
metrics=["accuracy"])
```

Le modèle défini et compilé, nous pouvons lancer l'apprentissage :

```
model.fit(creationInputValues, creationOutputValues, epochs=10,
batch_size=10)
```

- **epochs** : nombre d'itérations sur l'ensemble des données
- **batch_size** : taille du lot avant lequel le neurone ne mettra pas à jour les poids des variables

Notre neurone est terminé et prêt à lancer des prédictions, nous pouvons évaluer la précision de celui-ci en utilisant les valeurs de prédictions fournies :

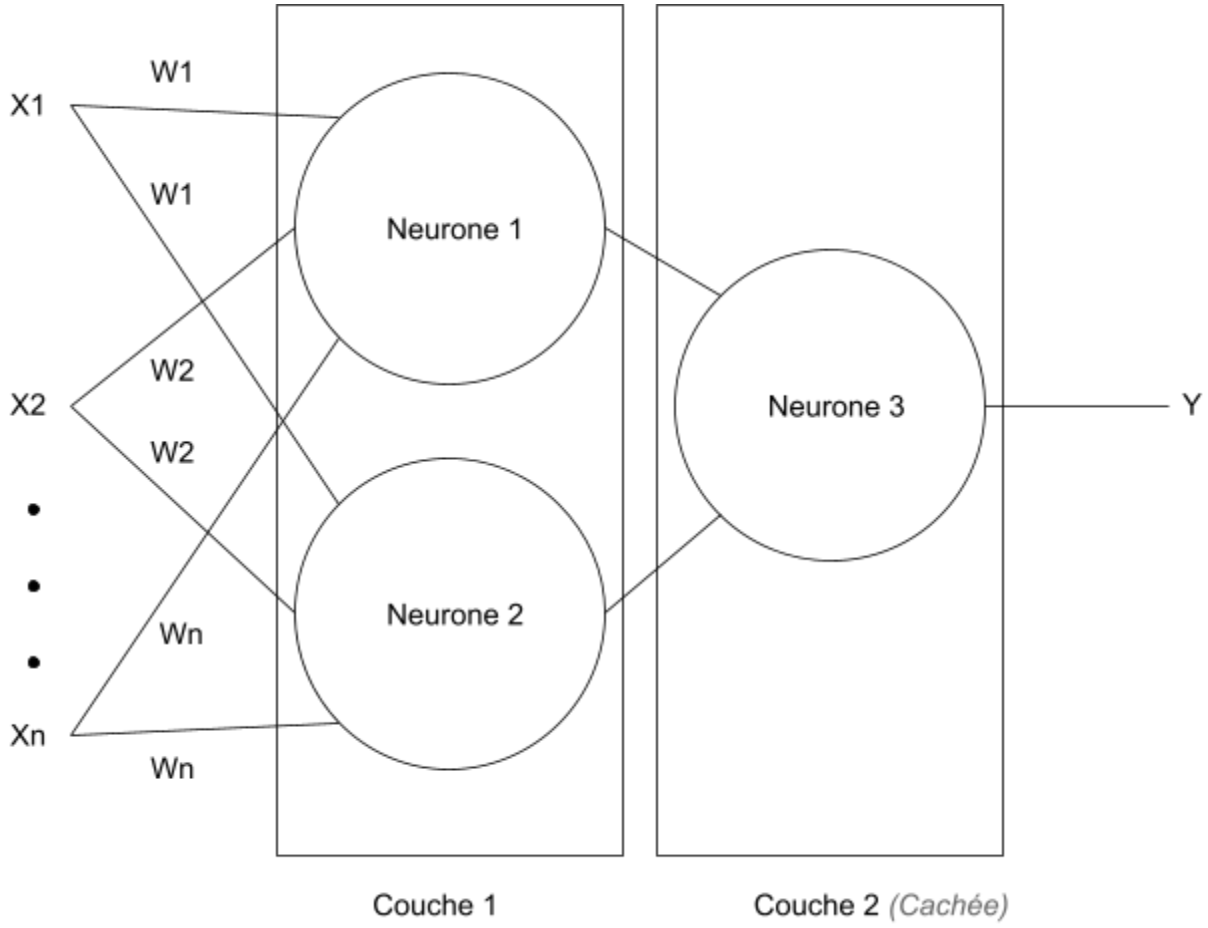
```
_, accuracy = model.evaluate(predictionInputValues, predictionOutputValues)
print("Accuracy: %.2f" % (accuracy*100))
```

Avec les valeurs données ci-dessus, nous obtenons une précision de **91% en moyenne**.

Modèle 2 - avec couches cachées

Un modèle avec couches cachées est très similaire à un modèle sans couches cachées à la différence près que nous avons ajouté des neurones, ceux-ci disposés en couches.

Une des manières les plus simple de construire un modèle avec couches cachées est la suivante :



La seule différence à apporter sur notre code va être au niveau de la construction du modèle Keras, étant donné que nous avons défini un modèle séquentiel, il nous suffira d'ajouter :

```
model = Sequential()
model.add(Dense(2, input_shape=(57,), activation="sigmoid"))
model.add(Dense(1, activation="sigmoid"))
```

La fonction d'activation **relu** semble être une meilleure option dans la majorité des cas, toutefois nous obtenons une précision de 53% en utilisant cette fonction sur les couches intermédiaires *(neurones 1 et 2)* contrairement à une précision de **93% en moyenne** en utilisant uniquement la fonction **sigmoid** sur tous les neurones.

Modèle 3 - classifieur bayesien naïf

Comme pour les modèles précédents, nous récupérons les données fournies dans les fichiers csv et les stockons sous forme de dataframe.

probaDataFrameNoSpam, probaDataFrameSpam contenant respectivement les données dans le cas d'un message non spam et d'un spam.

```
ipdb> probaDataFrameNoSpam
      word_freq_make  word_freq_address  word_freq_all  word_freq_3d  word_freq_our  ...
0       1.544488e-01           0.233502       0.469382      8.789252       0.643041  ...
1       1.261239e+00           0.233502       0.769270      8.789252       0.643041  ...
2       1.261239e+00           0.233502       0.769270      8.789252       0.643041  ...
3       2.489299e-08           0.233502       0.769270      8.789252       0.643041  ...
4       1.300372e+00           0.234767       0.807697      8.789252       0.673742  ...
...              ...                ...            ...           ...            ...  ...
1269    1.261239e+00           0.233502       0.769270      8.789252       0.643041  ...
1270    1.261239e+00           0.233502       0.533375      8.789252       0.499779  ...
1271    1.261239e+00           0.228050       0.161825      8.789252       0.649331  ...
1272    1.261239e+00           0.228751       0.794644      8.789252       0.652368  ...
1273    1.261239e+00           0.233502       0.769270      8.789252       0.146487  ...

[1274 rows x 59 columns]
```

```
ipdb> probaDataFrameSpam
      word_freq_make  word_freq_address  word_freq_all  word_freq_3d  word_freq_our  ...
0       2.833085e-01           0.962580       0.671850      0.168407       0.444029  ...
1       1.101634e+00           0.962580       0.586266      0.168407       0.444029  ...
2       1.101634e+00           0.962580       0.586266      0.168407       0.444029  ...
3       5.307229e-07           0.962580       0.586266      0.168407       0.444029  ...
4       1.194405e+00           1.034245       0.653364      0.168407       0.502056  ...
...              ...                ...            ...           ...            ...  ...
1269    1.101634e+00           0.962580       0.586266      0.168407       0.444029  ...
1270    1.101634e+00           0.962580       0.721149      0.168407       0.558620  ...
1271    1.101634e+00           0.383378       0.326591      0.168407       0.559315  ...
1272    1.101634e+00           0.413997       0.814761      0.168407       0.557601  ...
1273    1.101634e+00           0.962580       0.586266      0.168407       0.335162  ...

[1274 rows x 59 columns]
```

On commence par calculer la variance et l'espérance de chacun des attributs afin d'obtenir :

```
ipdb> tableauEsperance
   word_freq_make  word_freq_address  word_freq_all  word_freq_3d  word_freq_our  ...
0        0.075012           0.254149       0.191444      0.001784       0.184566  ...
1        0.152555           0.173036       0.395308      0.180986       0.492934  ...

[2 rows x 57 columns]
ipdb> tableauVariance
   word_freq_make  word_freq_address  word_freq_all  word_freq_3d  word_freq_our  ...
0        0.091663           2.839923       0.224664      0.001710       0.343903  ...
1        0.101775           0.134559       0.232755      5.560006       0.482088  ...

[2 rows x 57 columns]
```

Une fois ces données obtenues, il nous faut définir une fonction de calcul de probabilités *(attention, dans le cas ou la variance serait nulle il nous faut éviter la division par zéro, nous ajoutons donc 0.1 par précaution)* :

```
def calculerProba(x, esperance, variance) :
        return
(1/(math.sqrt(2*math.pi*variance)+0.1))*(math.exp((-1/(2*variance))*(pow(x-esperance,2))))
```

Il ne nous reste alors plus qu'à appliquer cette fonction sur chacun des éléments de nos données de prédiction afin de calculer les probabilités d'appartenir à la classe C_1 ou C_2 :

$$P(x_1 = attribut_1, \ldots, x_n = attribut_n \mid classe = C_1, C_2)$$

La classe possédant la probabilité la plus élevée sera celle choisie par le classifieur Bayésien naif, on obtient **finalClassDataFrame** :

Calculons maintenant la précision en comparant **modelCreation** et **finalClassDataFrame** contenant respectivement les classes à obtenir *(objectifs)* ainsi que les classes obtenues par le modèle, estimation basée sur la probabilité la plus haute (prédictions) :

```
ProjectX@MSI MINGW64 ~/Desktop/AI/ProjetIA (master)
$ python reseauBayes.py
Model de Prédiction
Nombre Total :  1274
Nombre de Spam :  719
Nombre de NoSpam :  555

Model de Création
Nombre Total :  1274
Nombre de Spam :  545
Nombre de NoSpam :  729

Nombre de similarités :  1050
Précision  82.42 %
```

Nous obtenons avec le classifieur Bayesien naif une précision de **83% en moyenne**.

Conclusion

Les modèles des **Kppv** et du **Perceptron** fonctionnent particulièrement bien avec une précision très bonne et un taux d'erreur faible. La courbe d'erreur du perceptron se rapproche grandement de zéro au bout de cinq itérations pour sembler l'atteindre à la quinzième.

Toutefois les problèmes sur lesquels nous avons travaillées sont très simples et adaptés à ce genre d'algorithme, les classes 1 et 2 :

- peuvent être efficacement séparées par une droite dans le cas du perceptron
- sont regroupées entre elles dans le cas des Kppv

Ce qui nous à facilité grandement la tâche.

Dans le cas des **Modèles 1, 2 et 3**, les données dépendent de beaucoup plus de paramètres et déterminer si le message X est un spam semble quelque peu plus compliqué.

Néanmoins nous obtenons des précisions correctes :

- Modèle 1 ⇒ 91 %
- Modèle 2 ⇒ 93 % *(avec sigmoid)* ⇒ 50 % *(avec relu et sigmoid)*
- Modèle 3 ⇒ 83 %

Le modèle le plus intéressant est le modèle 2 utilisant uniquement la fonction d'activation sigmoid sur tous les neurones.