

UNIVERSITE  
DE LIMOGES



FACULTE DES SCIENCES  
ET TECHNIQUES

Faculté des Sciences  
& Techniques



INTELLIGENCE  
ARTIFICIELLE



**Filière** : Master Mathématiques et Applications, Parcours ACSYON

---

## Projet d'Intelligence Artificielle

---

Rédigé par :

Rami MECHI

Cleque Marlain MBOULOU

Professeur de la matière :

Karim TAMINE

# TABLE DES MATIÈRES

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>TP1 : Algorithme des K plus proches voisins et Perceptron</b> | <b>3</b>  |
| 1.1      | Algorithme des K plus proches voisins . . . . .                  | 3         |
| 1.1.1    | Description du problème . . . . .                                | 3         |
| 1.1.2    | Analyse . . . . .  | 4         |
| 1.2      | Perceptron . . . . .   | 5         |
| 1.2.1    | Description du problème . . . . .                                | 5         |
| 1.2.2    | Analyse . . . . .  | 6         |
| <b>2</b> | <b>TP2 : Construction d'un Anti-Spam</b>                         | <b>8</b>  |
| 2.1      | Description . . . . .  | 8         |
| 2.2      | Analyse . . . . .  | 9         |
| 2.2.1    | Model M1 : Réseau de neurones sans couche cachée . . . . .       | 9         |
| 2.2.2    | Model M2 : Réseau de neurones avec une couche cachée . . . . .   | 9         |
| 2.2.3    | Model M3 : Classifieur Naïf Bayésien . . . . .                   | 10        |
| <b>3</b> | <b>Conclusion</b>  | <b>12</b> |
|          | <b>Bibliographie</b>   | <b>13</b> |

# CHAPITRE

## 1

# TP1: ALGORITHME DES K PLUS PROCHES VOISINS ET PERCEPTRON

## 1.1 Algorithme des K plus proches voisins

En intelligence artificielle, plus précisément en apprentissage automatique, la méthode des k plus proches voisins (KPPV) est une méthode d'apprentissage supervisé, c'est-à-dire qu'on va considérer un training set (ensemble de données) qui va surveiller notre apprentissage.

### 1.1.1 Description du problème

Notre problème est assez simple : on relève sur des individus de différentes classes 0 ou 1 des paramètres dans  $\mathbf{R}^2$ . On sait donc que pour tel individu de telle classe, on a tels paramètres. L'objectif est de pouvoir prévoir à quelle classe appartient un nouvel objet uniquement à l'aide de ses paramètres.

Le dataset est généré de la manière suivante :

La première classe est composée de 128 individus et est associée à la fonction de densité de la loi normale  $\mathcal{N}_2((4, 4)^T, I_2)$  :

$$f(x) = \frac{1}{2\pi} \exp \left\{ -\frac{1}{2} \left( x - \begin{pmatrix} 4 \\ 4 \end{pmatrix} \right)^T \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^{-1} \left( x - \begin{pmatrix} 4 \\ 4 \end{pmatrix} \right) \right\}$$

La seconde classe est composée de 128 individus et est associée à la fonction de densité de la loi normale  $\mathcal{N}_2((-4, -4)^T, 4I_2)$  :

$$f(x) = \frac{1}{8\pi} \exp \left\{ -\frac{1}{2} \left( x - \begin{pmatrix} -4 \\ -4 \end{pmatrix} \right)^T \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}^{-1} \left( x - \begin{pmatrix} -4 \\ -4 \end{pmatrix} \right) \right\}$$

La création sous python donne :

```
# Données de test
mean1 = [4, 4] # moyenne de la distribution
cov1 = [[1, 0], [0, 1]] # Matrice de covariance de la distribution
data1 = np.transpose(np.random.multivariate_normal(mean1, cov1, 128))
mean2 = [-4, -4]
cov2 = [[4, 0], [0, 4]] # Matrice de covariance de la distribution.
# Elle doit être symétrique et semi-définie positive pour un échantillonnage correct.
data2 = np.transpose(np.random.multivariate_normal(mean2, cov2, 128))
data = np.concatenate((data1, data2), axis=1)
oracle = np.concatenate((np.zeros(128), np.ones(128))) # Liste des classe
```

### 1.1.2 Analyse

Afin, de déterminer les k plus proches voisins à un point x, il nous faut évaluer la distance entre ce point et tous les autres points données d'apprentissage. Pour se faire nous avons créée une fonction qui correspond à la norme 2 :

```
def distance_euclidienne(point1, point2):
    d_carre = (point1[0] - point2[0])**2 + (point1[1] - point2[1])**2
    return sqrt(d_carre)
```

Nous allons ensuite déterminé grâce à notre algorithme KPPV, la classe à laquelle appartient l'individu x :

#### Algorithme KPPV :

1. Déterminer les distance entre x et tous les points de l'ensemble d'apprentissage
2. Ranger ces distances dans une liste dist
3. Récupérer les indices des distance triées par ordre croissant
4. Récupérer les K premiers et les ranger dans une liste
5. Déterminer la classe de chaque indice voisin à x
6. Compter les indices de même classe
7. La classe de x est la classe majoritaire

### Implémentation sous Python

```
def kppv_i(x, appren, oracle, K):
    dist = []
    indice = []
    l = []
    t = 0

    classe = 0 # On initialise la classe à 0 dans le cas contraire à la suite
    for j in range(len(appren[0])):
        d = distance_euclidienne(appren[:,j], x) # calcul la distance euclidienne entre le point x[:,i] et appren[:,j]
        dist.append((d)) # La fonction __.append() remplit la liste_distance_a_xi entre le point x[:,i] et tout les points
        # de appren

    # l = sorted(dist) # trie la liste par ordre croissant des distances
    l = np.argsort(dist) # renvoie les indices des distances triées par ordre croissant

    for k in range(K):
        indice.append(l[k]) # On construit une liste contenant les indices des k plus proche voisins de x

    for j in range(K):
        if(oracle[indice[j]] == 1):
            t = t + 1 # On compte ceux qui sont de classe 1

    if(t > int(K/2) + 1): # On vérifie si ceux qui sont de classe 1 sont en majorité
        classe = 1 # On ce cas la classe prend la valeur 1
    return classe
```

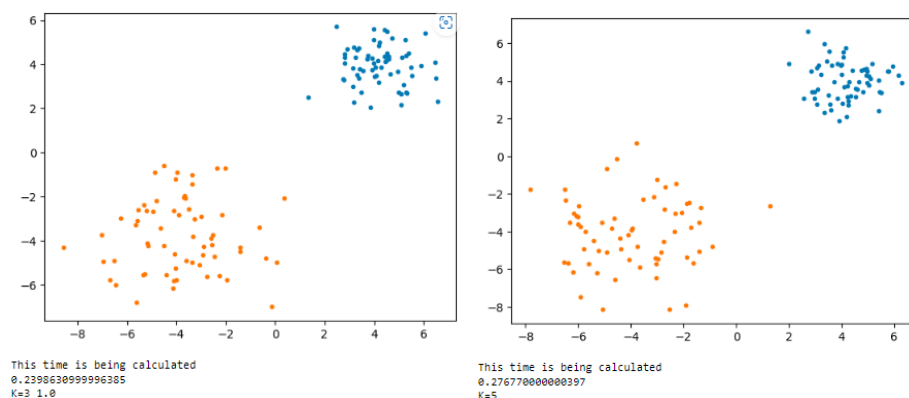
Nous appliquons alors notre algorithme avec les données test :

```
test1=np.transpose(np.random.multivariate_normal(mean1, cov1, 64))
test2=np.transpose(np.random.multivariate_normal(mean2, cov2,64))
test=np.concatenate((test1,test2), axis=1)
```

Il suffit donc de faire une boucle qui parcourt les données test :

```
def kppv(test,appren,oracle,K):
    clas=[]
    for i in range(len(test[0])):
        classe=kppv_i(test[:,i],appren,oracle,K)# On détermine La classe de chaque éléments de notre ensemble de test
        clas.append((classe)) # Puis on ajoute cette classe dans une liste
    clas=np.array(clas) # Puis que les données renvoyées doivent être dans un tableau, on converti la liste en tableau numpy
    #print(len(clas))
    return (clas)
```

On a les résultats suivants :



Ils ont identique car les points de nos données test tels que les 68 premiers sont de classe 0 et les 68 derniers de classe 1. Ainsi on crée un tableau que l'on nomme *orace* dont les 68 premières entrées sont des 0 et les 68 dernières des 1, puis on calcule l'accuracy.

```
from sklearn import metrics
accuracy = metrics.accuracy_score(clas, orace)
print(accuracy)
```

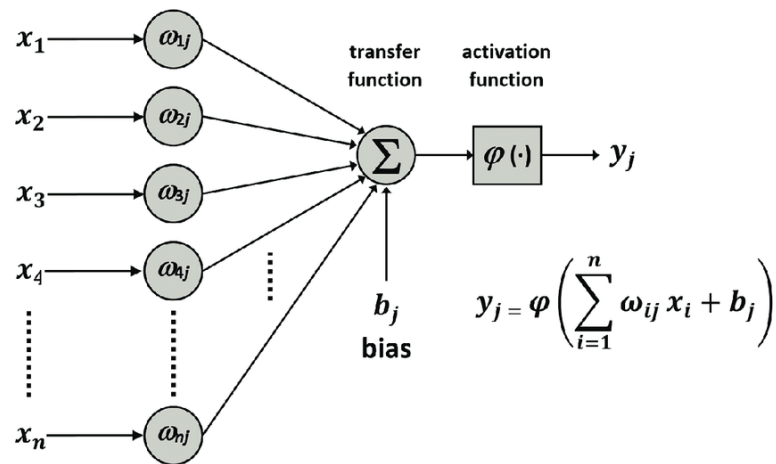
1.0

## 1.2 Perceptron

Le perceptron est un algorithme d'apprentissage supervisé de classifieurs binaires (c'est-à-dire séparant deux classes). Il a été inventé en 1957 par Frank Rosenblatt<sup>1</sup> au laboratoire d'aéronautique de l'université Cornell. Il s'agit d'un neurone formel muni d'une règle d'apprentissage qui permet de déterminer automatiquement les poids synaptiques de manière à séparer un problème d'apprentissage supervisé. Si le problème est linéairement séparable, un théorème assure que la règle du perceptron permet de trouver une séparatrice entre les deux classes.

### 1.2.1 Description du problème

Notre objectif est de trouver une droite qui sépare au mieux les données de classe 1 et celles de classe 2. La règle d'apprentissage d'un perceptron est la descente du gradient.



### 1.2.2 Analyse

Voici les données :

```
#Données de test
N=10 #Nombre d'époques

mean1 = [4, 4]
cov1 = [[1, 0], [0, 1]] #
data1 = np.transpose(np.random.multivariate_normal(mean1, cov1, 128))
mean2 = [-4, -4]
cov2 = [[4, 0], [0, 4]] #
data2 = np.transpose(np.random.multivariate_normal(mean2, cov2, 128))
data=np.concatenate((data1, data2), axis=1)
oracle=np.concatenate((np.zeros(128)-1,np.ones(128)))
```

Dans notre cas la fonction d'activation la sigmoïde et nous utiliserons aussi le produit scalaire :

```
def sigmoïde(x):#Fonction sigmoïde
    z=1/(1+exp(-x))
    return z

def produit_sc(x,y):
    s=x[0]*y[1]+x[1]*y[2]
    return s
```

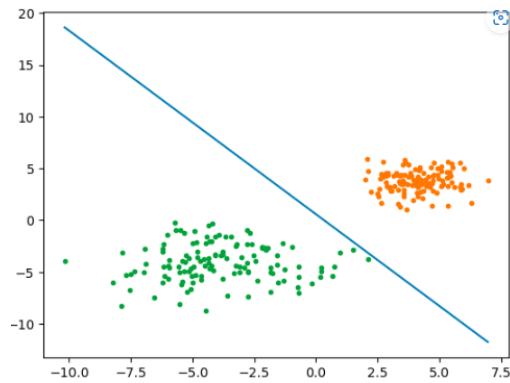
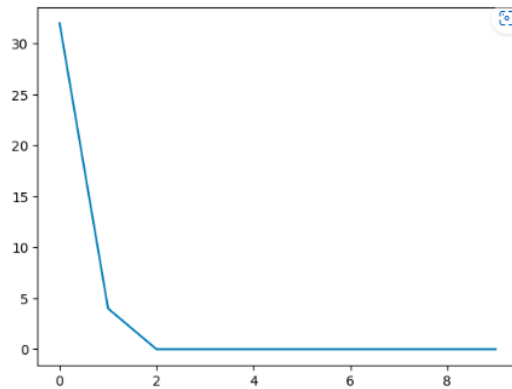
### Perceptron

```
def perceptron(x,w,active,b):
    y=0
    if(active==0):
        y=sign(b+produit_sc(x,w))#y prend le signe de b+produit_sc(x,w)
    if(active==1):
        if(sigmoïde(b+produit_sc(x,w))>=0.5):#
            y=1
        if(sigmoïde(b+produit_sc(x,w))<0.5):
            y=-1
    return y
```

### Apprentissage

```
def apprentissage(x,yd,active):
    w=[1,-2,4] #Initialisation de w
    mdiff=[]
    b=0.5 #initilisation
    for j in range(N):# On crée Les époque
        r=0 #Après chaque époque L'erreur est initialisé à 0 pour éviter Le cumul d'erreur
        for i in range(len(yd)):
            if(perceptron(x[:,i],w,active,b)!=yd[i]):# test si classe prédite par Le perceptron est la même que celle désirée
                r=r+4 #erreur (yd_i-y_i)^2=(1-(-1))^2=(-1-1)^2=4
                d=yd[i]-sigmoide(b+produit_sc(x[:,i],w))# Puis on détermine L'erreur d'apprentissage
                [w[1],w[2]]=w[1],w[2]+0.1*d*x[:,i] # on fait apprendre Les paramètres variant de w
                b=b+0.1*d # puis on fait apprendre Le biais
            mdiff.append((r))# Après chaque époque, on range L'erreur cumulée dans une Liste mdiff
    print(mdiff)
    return w,mdiff
```

### Résultats erreur et données séparées



# CHAPITRE

## 2

# TP2: CONSTRUCTION D'UN ANTI-SPAM

Avec les machines qui commencent à imiter très bien l'intelligence humaine, nous voudrions que celles-ci soient en mesure de détecter si un courrier est indésirable ou pas. La construction d'un Anti-Spam est alors un moyen de donner à la machine cette capacité.

## 2.1 Description

Dans cette partie nous allons mettre en œuvre un Anti-Spam en réalisant des modèles de classifieurs binaires construits à l'aide d'un DataSet (fourni) basé sur les activités normales et anormales d'une messagerie électronique.

### Données d'apprentissage et de prédiction

```
#import de la dataset
data=pd.read_csv("Spam detection - For model creation.csv",sep=";")
data
class_le= LabelEncoder()#
data['GOAL-Spam']=class_le.fit_transform(data['GOAL-Spam'].values)# Transforme les données de la colonne GOAL-SPAM qui contient
# des No et Yes en 0 pour No et 1 pour Yes
print("Shape is:",data.shape)
data
```

Shape is: (2972, 58)

```
#Données de prédictions
data_prediction=pd.read_csv("Spam detection - For prediction.csv")
data_prediction
```

### Préparations des données



```
x_train=data_prediction.drop('Spam', axis=1) #Dans le fichier data_prédiction on supprime la colonne Spam qui contient
#Les valeur de prédiction
y_train=data_prediction['Spam']

x_test=data.drop('GOAL-Spam', axis=1)
y_test=data['GOAL-Spam']

print("x_test:", x_test.shape,"y_test",y_test.shape)
print("x_train:", x_train.shape,"y_train",y_test.shape)
```

```
x_test: (2972, 57) y_test (2972,)
x_train: (1274, 57) y_train (2972,)
```

On va ensuite convertir nos données en tableau numpy

```
x_test, y_test=np.array(x_test), np.array(y_test)
x_train, y_train=np.array(x_train), np.array(y_train)
```

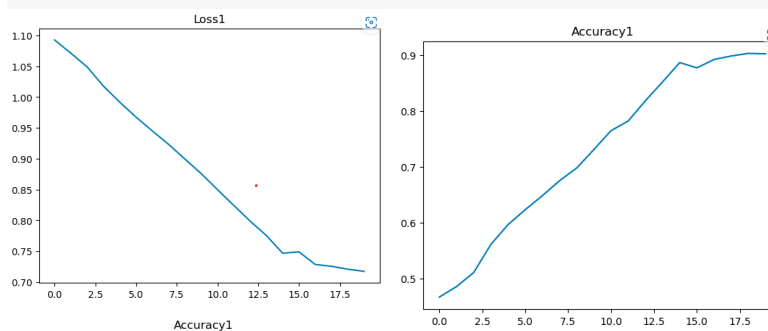
## 2.2 Analyse

### 2.2.1 Model M1 : Réseau de neurones sans couche cachée

```
model = keras.Sequential()
model.add(keras.layers.Input(57, name="InputLayer"))#Couche d'entrée
model.add(keras.layers.Dense(1, activation="sigmoid"))#Couche de sortie

model.compile(loss=keras.losses.categorical_hinge,
              optimizer="adam",# Nous permet de minimiser L'erreur
              metrics=['accuracy'])

history = model.fit(x_train,
                    y_train,
                    epochs = 25,#nombre d'époques
                    batch_size = 10)#correction après chaque dizaine
```

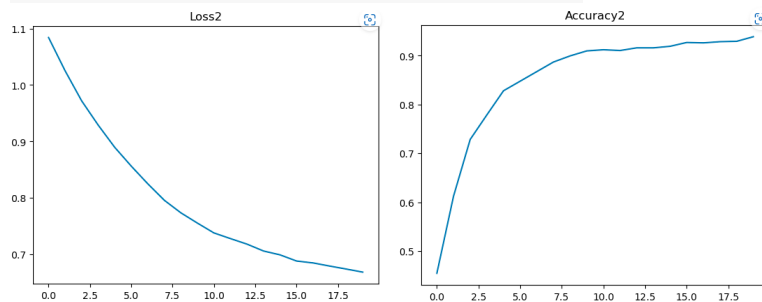


### 2.2.2 Model M2 : Réseau de neurones avec une couche cachée

```
model = keras.Sequential()
model.add(keras.layers.Input(57, name="InputLayer"))#Couche d'entrée
model.add(keras.layers.Dense(10, activation="sigmoid"))#couche cachée
model.add(keras.layers.Dense(1, activation="sigmoid"))# couche de sortie

model.compile(loss=keras.losses.hinge,# L'erreur loss=keras.losses.hinge
              optimizer="adam",# Nous permet de minimiser L'erreur
              metrics=['accuracy'])
```

```
history = model.fit(x_train,
                    y_train,
                    epochs = 20,
                    batch_size = 10)
```



### 2.2.3 Model M3 : Classifieur Naïf Bayésien

La classification naïve bayésienne est un type de classification bayésienne probabiliste simple basée sur le Théorème de Bayes avec une forte indépendance (dite naïve) des hypothèses.

#### Construction du classifieur

. Le classifieur bayésien naïf se construit en appliquant une règle de décision couramment employée : la règle du *maximum a posteriori* définie par :

Soit une donnée  $X = (X_1=x_1, X_2=x_2, \dots, X_n=x_n)$  à classer.

Alors

$$\text{Classifieur}(x_1, x_2, \dots, x_n) = \text{ArgMax } P(C=c_j) \prod P(X_i = x_i / C=c_j)$$

```
import scipy.stats
def Clas_Bayes(data):
    esp0 = []
    esp1 = []
    ecar_t0 = []
    ecar_t1 = []
    liste0 = []
    liste1 = []

    for i in range(len(T)): #Dans cette boucle on va créer les listes des indices des éléments de classe 0 et 1
        if(T[i,0]==0):
            liste0.append(i) # On ajoute à liste0 tous les éléments de classe 0
        if(T[i,0]==1):
            liste1.append(i) # On ajoute à liste1 tous les éléments de classe 1
    n0=len(liste0) # Cardinal de liste0
    n1=len(liste1) # Cardinal de liste1
    for j in range(len(T.columns)):

        esp0.append((sum(T[i,j] for i in liste0))/n0) # on calcul l'espérance de chaque critère de détection de spam sachant que
        esp1.append((sum(T[i,j] for i in liste1))/n1) # on calcul l'espérance de chaque critère de détection de spam sachant que
        ecar_t0.append(sqrt(sum((T[i,j]-esp0[j])**2 for i in liste0)/(n0-1))) # on calcul la variance de chaque critère de détec
        ecar_t1.append(sqrt(sum((T[i,j]-esp1[j])**2 for i in liste1)/(n1-1))) # on calcul la variance de chaque critère de détec

    del esp0[0] # Suppression de l'espérance de classe 0
    del esp1[0] # Suppression de l'espérance de classe 1
    del ecar_t0[0] # Suppression de l'espérance de classe 0
    del ecar_t1[0] # Suppression de l'espérance de classe 1
```

```

# Liste des produit des fonctions densite classe 0
oracle0 = []

for i in range(len(data)):
    p0=n0/len(T) # Probabilité de la classe 0
    for j in range(len(T.columns)-1):
        mu = esp0[j] # Espérance
        sigma = ecar_t0[j] # Ecart-type
        #f=scipy.stats.norm.pdf(data[i,j],sigma,mu)
        p0 = p0*1/(sigma*sqrt(2*pi))*exp(-0.5*((data[i,j]-mu)/sigma)**2) # produit des fonctions de densité par p0

    oracle0.append((p0))

# Liste des produit des fonctions densite classe 1
oracle1 = []

for i in range(len(data)):
    p1 = n1/len(T) # Probabilité de la classe 1
    for j in range(len(T.columns)-1):
        mu = esp1[j] # Espérance
        sigma = ecar_t1[j] # Ecart-type
        if(mu!=0): #
            #f=scipy.stats.norm.pdf(data[i,j],sigma,mu)
            p1 = p1*1/(sigma*sqrt(2*pi))*exp(-0.5*((data[i,j]-mu)/sigma)**2) # produit des fonctions de densité par p1

    oracle1.append((p1))

#classifieur de bayes
oracle = []
for i in range(len(data)):#l'individu i de data aura la de
    t = 0
    if(oracle1[i]>oracle0[i]):
        t=1
    oracle.append((t))

return oracle

```

### Comparaison

```

# Comparaison des Accuracy (Performance de chaque model)

from sklearn import metrics
accuracy3 = metrics.accuracy_score(data[:,57], Clas_Bayes(data))

print("<<<Accuracy M1:",accuracy_curve1[19],">>> <<<Accuracy M2: ",accuracy_curve2[19],">>> <<<Accuracy M3: ",accuracy3,">>>")

<<<Accuracy M1: 0.9018838405609131 >>> <<<Accuracy M2: 0.9387755393981934 >>> <<<Accuracy M3: 0.8163265306122449 >>>

```

Après comparaison, le model constuit avec Kéras est le meilleur et l'ajout des couches cachées rend le model plus performant.

## CHAPITRE

3

## CONCLUSION

Au cours de ces travaux pratiques, qui étaient pour nous une sorte d'initiation à python, nous avons appris le langage en le mettant en pratique sur les méthodes d'apprentissage supervisée telles que celle des K plus proche voisin, Perceptron, Classifieur Naïf Bayésien. Nous pouvons conclure que l'intelligence artificielle est une grande opportunité à condition que celle-ci soit maîtrisée car nous avons vu travers nos recherche quelques application de la méthode des KPPV ( dans le transport par exemple), et des réseaux de neurones dans la détection des de SPAM.

# BIBLIOGRAPHIE

- [1] [Classifieur Naïf Bayésien](#) [en ligne]
- [2] [Cours Apprentissage 2021 - partie1](#)Fichier
- [3] [Réseau de neurones avec Kéras](#)