UNIVERSITÉ DE LIMOGES

Faculté de Sciences et Techniques

Master 1

Sécurité de l'Information et Cryptologie

(CRYPTIS)

Parcours Informatique

Projet Intelligence Artificielle - Rapport

Méthodes d'Apprentissage Supervisé

MAKHOUL Vladimir

SALAME Joe

Superviseur :

M. Karim Tamine- Assistant Professor

Novembre 10, 2022

Table de matières

I. Introduction

Nous assistons actuellement à une révolution globale de l'IA dans tous les secteurs. Et l'un des moteurs de cette révolution de l'IA est le deep learning. Grâce à des géants comme Google et Facebook, le deep learning est devenu un terme populaire et les gens pourraient penser qu'il s'agit d'une découverte récente. Mais vous serez peut-être surpris d'apprendre que l'histoire de l'apprentissage profond remonte aux années 1940 avec McCulloch et Pitts, qui ont imité la fonctionnalité d'un neurone biologique.

En effet, le deep learning n'est pas apparu du jour au lendemain, il a plutôt évolué lentement et progressivement pendant sept décennies. Et derrière cette évolution, il y a de nombreux chercheurs en apprentissage automatique qui ont travaillé avec une grande détermination, même lorsque personne ne croyait que les réseaux de neurones avaient un avenir.

Au sein de l'intelligence artificielle et d'apprentissage automatique, il existe deux approches de base : l'apprentissage supervisé et l'apprentissage non supervisé. La principale différence est que l'une utilise des données étiquetées pour aider à prédire les résultats, tandis que l'autre ne le fait pas. Cependant, il existe des nuances entre les deux approches et des domaines clés dans lesquels l'une est plus performante que l'autre.

Dans ce projet, nous allons nous concentrer sur l'apprentissage supervisé et certaines de ces méthodes populaires comme les Kppv, le perceptron, les réseaux de neurones et le classifieur bayésien naïf.

II. Étude théorique

1. Apprentissage supervisé

L'apprentissage automatique supervisé nécessite des données d'entrées et de sortie étiquetées pendant la phase de formation du cycle de vie de l'apprentissage automatique. Ces données de formation sont souvent étiquetées par un scientifique lors de la phase de préparation, avant d'être utilisées pour former et tester le modèle. Une fois que le modèle a appris la relation entre les données d'entrée et de sortie, il peut être utilisé pour classer des ensembles de données nouveaux et non vus et prédire les résultats.

La raison pour laquelle on l'appelle apprentissage supervisé est qu'au moins une partie de cette approche nécessite une supervision humaine. La grande majorité des données disponibles sont des données brutes non étiquetées. Une interaction humaine est généralement nécessaire pour étiqueter avec précision les données prêtes pour l'apprentissage supervisé. Naturellement, ce processus peut être gourmand en ressources, car il nécessite de grandes quantités de données d'entraînement étiquetées avec précision. L'algorithme mesure sa précision à travers la fonction de perte(loss), en s'ajustant jusqu'à ce que l'erreur ait été suffisamment minimisée.

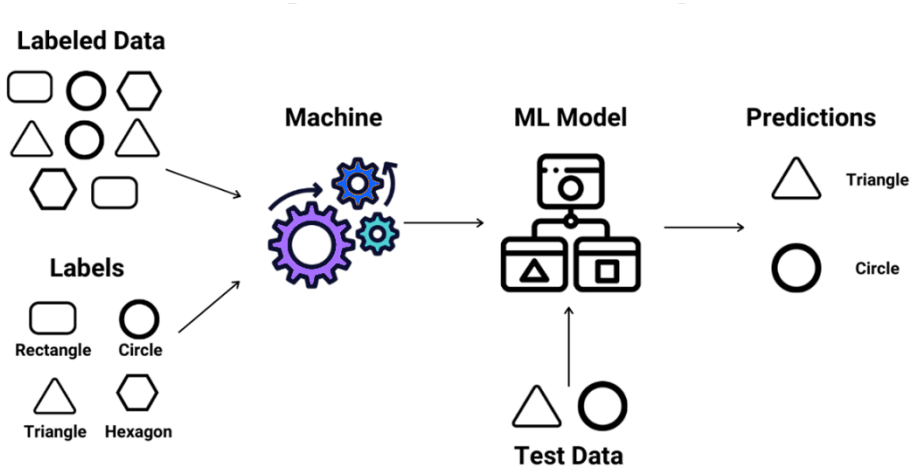


Figure 1- Apprentissage supervisé

2. Méthodes d'apprentissage

L'apprentissage supervisé peut-être diviser en deux types de problèmes lors de l'exploration de données : la classification et la régression.
Dans ce projet nous allons nous concentrer seulement sur la classification en utilisant les méthodes suivantes : les kppv, le perceptron, les réseaux de neurones et le classifieur bayésien naïf.

2.1 Les k-plus proche voisins (Kppv) :

L'algorithme des k plus proches voisins est une méthode de classification des données permettant d'estimer la probabilité qu'un point de données devienne membre d'un groupe ou d'un autre en fonction du groupe auquel appartiennent les points de données les plus proches.
L'algorithme k-plus proche voisin est un type d'algorithmes d'apprentissage automatique supervisé utilisé pour résoudre des problèmes de classification et de régression. Cependant, il est principalement utilisé pour des problèmes de classification.
C'est un apprentissage paresseux et un algorithme non paramétrique.

Il est appelé paresseux, car il n'effectue aucune formation lorsque vous fournissez les données de formation. Au lieu de cela, il stocke simplement les données pendant le temps de formation et n'effectue aucun calcul. Il ne construit pas de modèle tant qu'une requête n'est pas effectuée sur l'ensemble de données. Cela rend Kppv idéal pour l'exploration des données.

Les Étapes :

- Étape 1 : Sélectionnez le nombre K de voisins
- Étape 2 : Calculez la distance Euclidienne :

$$d(x_1, x_2) = \sqrt{\sum_k (x_{1,k} - x_{2,k})^2}$$

- Étape 3 : Prenez les K voisins les plus proches selon la distance calculée.
- Étape 4 : Parmi ces K voisins, comptez le nombre de points appartenant à chaque catégorie.
- Étape 5 : Attribuez le nouveau point à la catégorie la plus présente parmi ces K voisins.
- Étape 6 : Notre modèle est prêt :

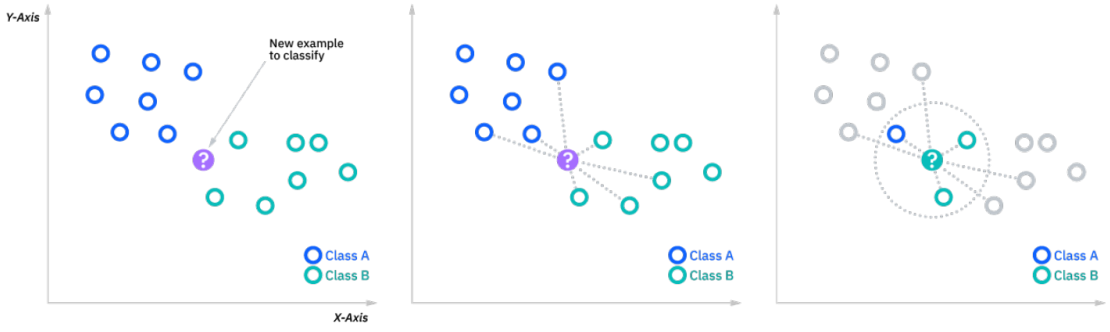


Figure 2 - Example de classification Kppv

2.2 Le Perceptron :

Le perceptron est l'unité de base des réseaux de neurones. Il s'agit d'un modèle de classification binaire, capable de séparer linéairement deux classes de points. Il était inventé en 1957 par l'Américain Frank Rosenblatt qui est un

psychologiste dans le champ de l'intelligence artificiel.

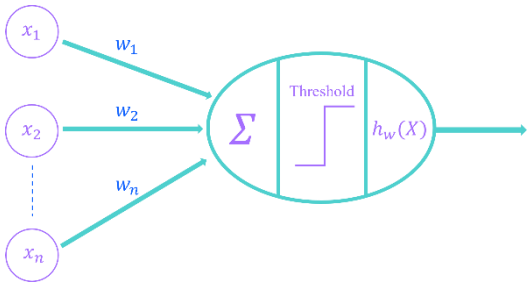


Figure 3 – Architecture du Perceptron

Les étapes de perceptron sont :

- **Somme des vecteurs**

  La première étape est de calculer la somme du vecteur des entrées avec le vecteur des poids en ajoutant le biais b :

$$\boldsymbol{z = \sum_{i=o} x_i \cdot w_i + b}$$

  Où x sont les entrées, w sont les paramètres et b le biais.

- **Threshold**

  Après le calcul de la somme des vecteurs, la fonction d'activation ou (threshold) prend la réponse de cette formule comme un paramètre puis fait la prédiction :

  -Si $z \geq 0$ il renvoie 1.

  -Sinon il renvoie 0.

  Ensuite nous comparons le résultat renvoyer par le threshold avec la vraie classification de cette entrée s'ils sont les mêmes elle prend une autre entrée sinon elle fait la mise à jour des paramètres et du biais.

- **La mise à jour**

  Nous avons utilisé dans la mise à jour la fonction Loss :

$$\boldsymbol{Loss\big(Y_t, hw(X_t)\big) = -\big(Y_t - h_w(X_t)\big)WX_t}$$

  Où $h_w(X_t)$ est la prédiction du threshold.

  Ensuite nous appliquons la dérivée partielle (calcule de la descente du gradient) sur cette formule pour minimiser la perte, la formule devient :

$$\frac{\partial}{\partial w_i} Loss\big(y_t - h_w(X_t)\big) = -(y_t - h_w(X_t))x_{t,i}$$

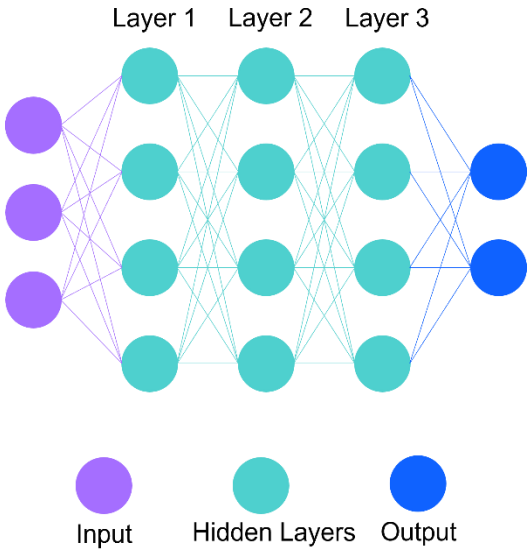
Finalement on met à jour les paramètres en utilisant cette formule :

$$w \leftarrow w + \alpha\big(y_t - h(x_t)\big)x_t$$

$$b \leftarrow b + \alpha\big(y_t - h(x_t)\big)$$

2.3 Réseaux de neurones :

Un réseau de neurones est une méthode d'intelligence artificielle qui apprend aux ordinateurs à traiter les données d'une manière inspirée par le cerveau humain. Il utilise des nœuds ou des neurones interconnectés dans une structure en couches qui ressemble au cerveau humain. Il crée un système adaptatif que les ordinateurs utilisent pour apprendre de leurs erreurs et s'améliorer en permanence. Ainsi, les réseaux de neurones artificiels tentent de résoudre des problèmes compliqués, comme résumer des documents ou reconnaître des visages, avec une plus grande précision.



Comment ça marche ?

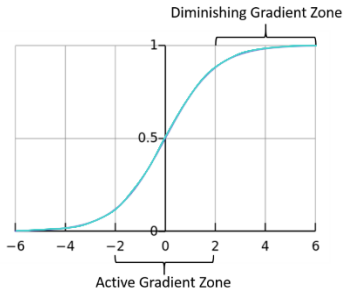
Considérez chaque nœud individuel comme son propre modèle de régression linéaire, composé de données d'entrée, de pondérations, d'un biais (ou seuil) et d'une sortie.

Nous avons deux étapes principales :

a) Forward propagation :

Comme mentionner dans le perceptron nous calculons « z » avec les entrées, les poids et le biais puis nous utilisons la fonction d'activation sigmoïde en général. Sigmoïde est principalement utilisé pour la classification binaire et la classification multi-label. Dans la classification multi-label, il peut y avoir plus d'une réponse correcte.

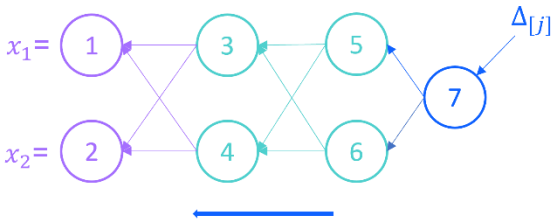
$$Logistic(z) = \frac{1}{1 + e^{-z}}$$



b) Back propagation :

Cette méthode est aussi appelée rétropropagation consistant à mettre à jour les poids de chaque neurone de la dernière couche vers la première. Elle vise à corriger les erreurs selon l'importance de la contribution de chaque élément à celles-ci. Dans le cas des réseaux de neurones, les poids synaptiques qui contribuent plus à une erreur seront modifiés de manière plus importante que les poids qui provoquent une erreur marginale.

$$\Delta_{[j]} = -\frac{\partial}{\partial in_j} Loss$$



$$\Delta_{[j]} = g(in_j)(1 - g(in_j)) \sum_k w_{j,k} \Delta_{[k]}$$

Nous terminons par la mise à jour des poids w en utilisant la formule suivante :

$$w_{i,j} \leftarrow w_{i,j} + \alpha a_i \Delta_{[j]}$$

2.4 Réseau Bayésien Naïf

Le naïf bayésien est une méthode probabiliste basée sur le théorème de bayes qui classifie les données.
Cet algorithme peut gérer les données continues et discrètes.

- **Paramètres discrets**

  Dans le cas des paramètres discrets nous commençons à calculer la probabilité de chaque classe.
  Ensuite nous construisons des tableaux pour chaque colonne contenant les probabilités de ses caractéristiques par rapport à chaque classe.
  Après la construction des tableaux nous prenons les données à prédire et nous appliquons la formule suivante en utilisant les valeurs obtenues dans les tableaux :

$$P(c \mid X) = P(x_1|c) \times P(x_2|c) \times \ldots \times P(x_n|c) \times P(c)$$

  Où « c » est une variable de classe dépendante dont les instances ou classes sont peu nombreuses, conditionnée par plusieurs variables caractéristiques $x_1,\ldots,x_n$.

  La dernière étape est de prendre la valeur maximale de cette formule et nous choisissons la classe qui a donné cette valeur maximale comme prédiction.

- **Paramètres continus**

  Pour les paramètres continus nous appliquons la loi normal donnée par :

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

  Une distribution gaussienne est également appelée distribution normale. Lorsqu'elle est tracée, elle donne une courbe en forme de cloche qui est symétrique par rapport à la moyenne des valeurs des caractéristiques, comme indiqué ci-dessous :

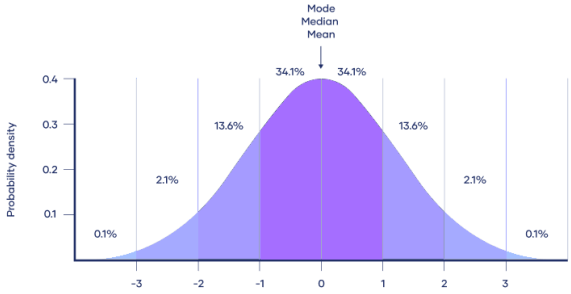


Figure 4 - Courbe de gausse

III. Étude pratique

1. Programmation de la fonction Kppv :

Dans le code de k-plus proches voisins nous avons créé une fonction kppv qui prend les paramètres suivants :

- **Test :** tableau de deux dimensions des données à prédire.
- **Data :** tableau de deux dimensions des données d'apprentissage.
- **Oracle :** tableau d'une seule dimension contenant les classes des données d'apprentissage.
- **K :** le nombre des voisins à prendre pour faire la classification.

Au début, la fonction **Kppv** initialise un tableau **clas** d'une seule dimension pour mémoriser la prédiction des données **test**. La longueur de ce tableau égale à la moitié de longueur **test** car il est composé de deux tableaux test0 et test1 contenant les x1 et x2. Ensuite, nous avons fait une itération dans le tableau **data** contenant les données d'apprentissage.

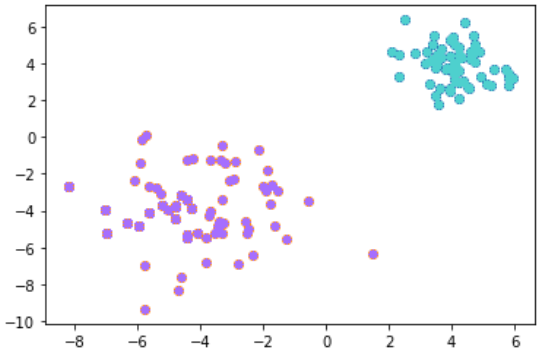
Pour chaque itération, **Kppv** appelle la fonction **euclidean_distance** qui prend comme paramètre les points x1 et x2 du **test** avec ceux du tableau **data** pour calculer la distance entre eux. Ensuite **Kppv** enregistre dans le tableau **result** un tuple composé de la distance calculée avant avec la classe de la donnée d'apprentissage utilisée dans la formule précédente.

Après la fin de la boucle, la fonction **Kppv** trie le tableau result pour prendre le **K** plus petites distances et crée deux compteurs pour les classes qui sont 1 ou 0 puis elle fait une boucle de K itérations pour calculer le nombre de chaque classes de ces k plus petites distances.

Finalement, **Kppv** compare ces deux compteurs, prend celle qui a le plus grand nombre comme la prédiction et l'enregistre dans le tableau **clas**.

```python
def kppv(test,data,oracle,k):
    clas=np.zeros(len(test[0]))

    for i in range(0,len(test[0])):
        result = []
        for j in range(0,len(data[0])):
            d=euclidean_distance(test[0][i],data[0][j],test[1][i],data[1][j])
            result.append((d,oracle[j]))
        result.sort()
        cl0 = 0 #compteur pour la classe 0
        cl1 = 0 #compteur pour la classe 1

        for x in result[0:k]:
            if x[1] == 1:
                cl1+=1
            else:
                cl0+=1

        if cl1>cl0:
            clas[i]=1
        else:
            clas[i]=0

    return clas

def euclidean_distance(x1, x2, y1, y2):
    distance = np.sqrt(((x1-x2)**2)+((y1-y2)**2))
    return distance
```

Output et affichage :



2. Programmation du Perceptron :

Notre code qui réalise la méthode de perceptron est formé de trois fonctions essentielles :

- **Active :** Les paramètres sont deux tableaux, le premier **x** est formé de deux lignes qui sont chacune l'entrées. Le deuxième **w** est formé de trois lignes la première est pour le seuil et les restes pour les poids.
- **Perceptron :** Les paramètres sont les tableaux **x**, **w** avec la fonction **active.**
- **Apprentissage :** Les paramètres sont le tableau des données d'apprentissage **data**, le tableau **oracle** contenant les classes des éléments du tableau data.

Active : Cette fonction fait l'addition de la somme des produits entre les entrées et les poids avec le seuil puis renvoie le résultat.

```
1  def active(x,w):
2      res = 0
3      for i in range(0,2):
4          res += x[0,i]*w[0,i+1] #Calcule la somme des (wi.xi)
5      res += w[0,0] #Addition du bias avec la somme precedente
6      return res
```

Perceptron : Elle appelle la fonction active pour x et w et mémorise le résultat dans **y**, puis elle prend le résultat et applique le threshold pour renvoyer la prédiction **-1** ou **1**.

```
7  def perceptron(x,w,active):
8      y = active(x,w)
9      if y < 0:
10         return -1
11     else:
12         return 1
```

Apprentissage : au début, cette fonction initialise un tableau qui contient : **w** contenant le seuil avec les poids, **alpha** qui est le taux d'apprentissage, un tableau **x** d'une seule dimension qui enregistre les entrées et un tableau **error** vide qui va enregistrer les erreurs**.** Ensuite, dans une boucle de cent itérations la fonction prend chaque donnée et appelle la fonction **perceptron** pour faire la prédiction puis elle calcule les erreurs dans le tableau **erreur**.

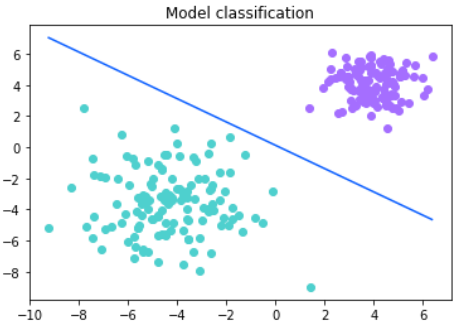
Après cette fonction compare la prédiction obtenue avec la vraie prédiction dans **oracle** :

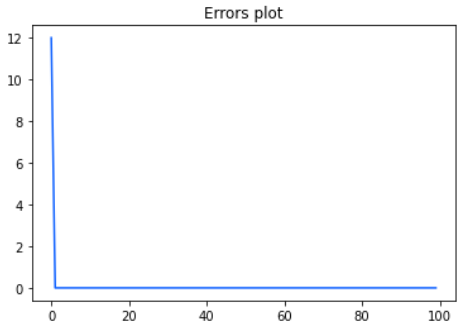
- Si elles sont différentes donc elle applique la mise à jour des paramètres.
- Si non la boucle continue.

À la fin quand la boucle se termine, la fonction renvoie **w** contenant les nouvelles valeurs des poids et du seuil avec le tableau d'erreur.

```
def apprentissage(data,oracle,active,perceptron):
    w = np.array([[0.5, 1.2, 0.8]]) #initialisation des poids avec b
    alpha = 0.1 #taux d'apprentissage
    x = np.array([[0,0]])

    error = [] #tableau qui stocke les erreurs


    for j in range(0,100):
        error += [0] #on ajoute une case au tableau error avec chaque iteration
        for i in range(len(data[0])):
            x[0,0] = data[0][i] #memoriser l'entree dans x
            x[0,1] = data[1][i]
            h = perceptron(x,w,active) #l'appel de la fonction perceptron pour faire la prediction
            error[j]+=(oracle[i]-h)**2 #calculer l'erreurs

            if h != oracle[i]:
                w[0,1] +=  alpha * (oracle[i] - h ) * x[0,0] #mise a jour de w1
                w[0,2] +=  alpha * (oracle[i] - h ) * x[0,1]#mise a jour de w2
                w[0,0] +=  alpha * (oracle[i] - h )  #mise a jour de b(seuil)

    return w , error
```

Output :





3. Programmation du réseau de neurone :

a) Chargement des données :

Tout d'abord, nous importons le fichier csv contenant les données, puis nous stockons les données de prédiction dans la variable y et le reste du tableau dans la variable x.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

data = pd.read_csv('spam.csv')
x = data.drop("Spam", axis=1)
y = data["Spam"]
```

b) Définir le modèle :

Le modèle dans Keras est toujours défini comme une séquence de couches. Cela signifie que nous initialisons le modèle de séquence et nous ajoutons les couches l'une après l'autre. En pratique, nous devons expérimenter avec le processus d'ajout et de suppression des couches jusqu'à ce que nous soyons satisfaits de notre architecture.

Tout d'abord nous utilisons « input_dim » pour spécifier le nombre d'entrées.

Dans cet exemple, nous allons définir un réseau entièrement connecté avec une couche cachée. Pour définir la couche entièrement connectée, nous utilisons la classe Dense de Keras.

- La première couche a 20 neurones et la fonction d'activation Softmax.
- Au niveau de la couche de sortie, nous utilisons 1 unité et la fonction d'activation sigmoïde car il s'agit d'un problème de classification binaire.

```
10  model = Sequential()
11  model.add(Dense(20, input_dim=57, activation="softmax"))
12  model.add(Dense(1, activation="sigmoid"))
```

c) Compilation du modèle:

Lorsque nous compilons le modèle, nous devons spécifier certains paramètres supplémentaires afin de mieux évaluer le modèle :

- **Fonction de perte** (Loss) : il faut spécifier la fonction loss pour évaluer l'ensemble des poids sur lesquels le modèle sera mis en correspondance. Nous utiliserons « binary_crossentropy » comme fonction de perte utilisée pour la classification binaire.
- **Optimizer :** Le second est l'optimiseur pour optimiser la perte. Nous utiliserons « adam » qui est une version populaire de la descente de gradient et donne le meilleur résultat dans la plupart des problèmes.

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

d) Entrainement du modèle:

Nous sommes prêts à adapter les données au modèle et à commencer l'entrainement du réseau de neurones. En plus de fournir des données au modèle, nous devons définir un nombre d'époques et une taille de lots sur lesquels se déroule l'entrainement.

- **Epoch** : un seul passage par toutes les lignes de l'ensemble de données d'entraînement.
- **Batch size :** nombre d'échantillons considérés par le modèle avant la mise à jour des poids.

```
12
13  history = model.fit(x,y,validation_split=0.33, epochs=150, batch_size=10)
14  print(history.history.keys())
15
```

e) Evaluation du modèle :

Après l'entrainement du modèle nous commençons à l'évaluer.

```
15
16  _, accuracy = model.evaluate(x,y)
17  print("Model accuracy: %.2f"% (accuracy*100))
18
```

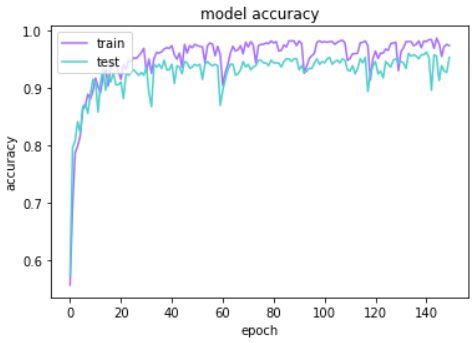
f) Prédictions :

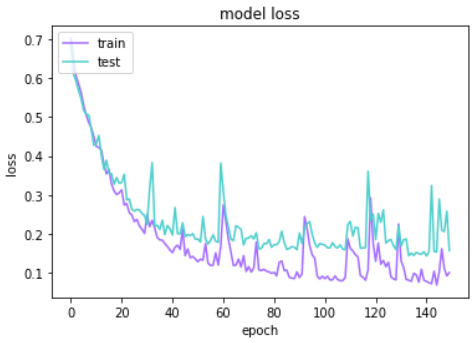
Prédire la sortie de nouvelles données en utilisant simplement la méthode de prédiction. Nous avons un problème de classification binaire, la sortie sera donc simplement 0 ou 1.

```
18
19  predictions = model.predict(x)
20  rounded = [round(x[0]) for x in predictions]
21
```

g) Output et affichage :

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
40/40 [==============================] - 0s 557us/step - loss: 0.0975 - accuracy: 0.9749
Model accuracy: 97.49
```



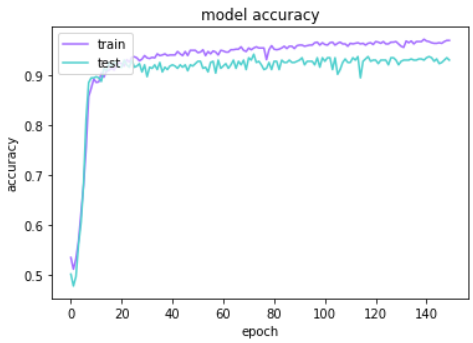


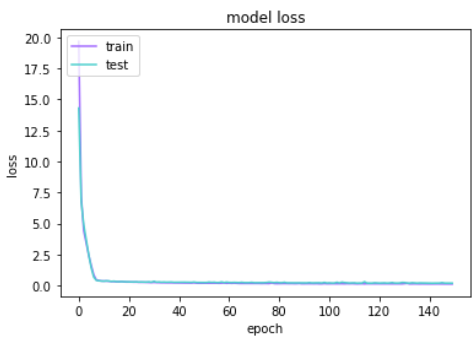
4. Réseaux de neurone (Sans couche cachée)

Techniquement la principale différence dans la mise en place d'un réseau de neurones sans couche cachée est la même qu'avant mais nous n'avons qu'à mettre une seule couche dans le code « model.add(dense()) » qui prend comme paramètre les 57 entrées et une sortie en utilisant la fonction sigmoïde.

```
1
2  model.add(Dense(1, input_dim=57, activation="sigmoid"))
3
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
40/40 [==============================] - 0s 1ms/step - loss: 0.1355 - accuracy: 0.9545
Model accuracy: 95.45
```





5. Programmation du classifieur Naïf bayésien :

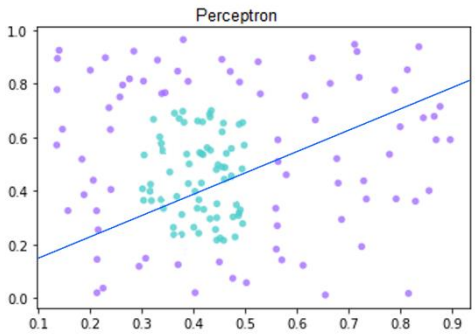
Tout d'abord nous chargeons les données pareilles au réseau de neurones. Puis nous utiliserons le module train test split pour diviser l'ensemble de données en sections d'entrainement et de test. Ensuite nous importons le module « Gaussian », pour ajuster le modèle, nous pouvons passer x_train et y_train. Enfin le accuracy_score reflète le succès avec lequel notre modèle Gaussian a prédit à l'aide des données de test.

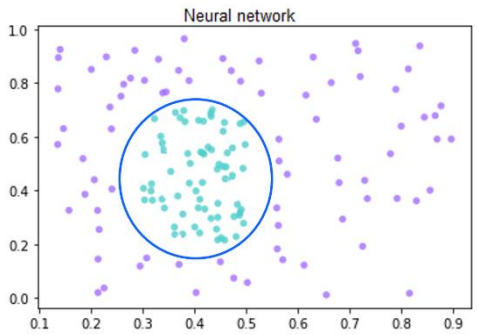
```python
import numpy as np
import pandas as pd
import sklearn
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import accuracy_score

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size = .33, random_state =17)
GausNB = GaussianNB()
y_expect = y_test
GausNB.fit(X_train, y_train)
y_pred = GausNB.predict(X_test)
print (f"Accuracy score: {accuracy_score(y_expect, y_pred)}")

```

```
Accuracy score: 0.8669833729216152
```

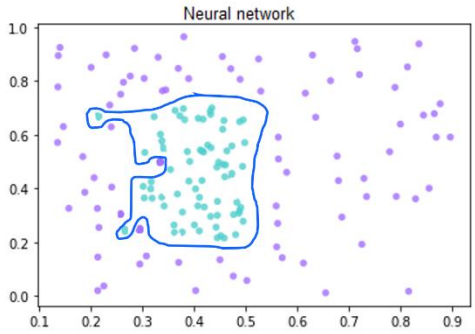
IV. Analyse

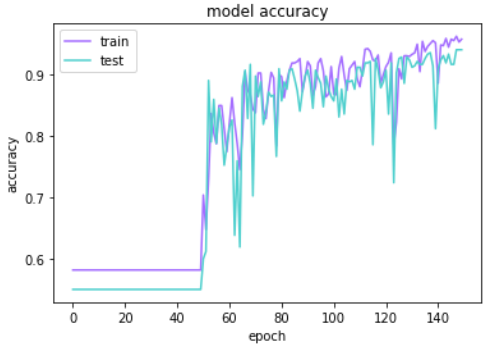
Après observation des deux modèles de réseaux de neurones, sans et avec couche cachée, nous remarquons que le premier modèle est similaire à un modèle de régression linéaire c.-à-d. un perceptron. Comme nous le savons déjà, une régression linéaire tente d'ajuster une équation linéaire aux données observées. Le perceptron n'est pas un approximateur de fonctions universel, ce qui signifie que nous ne pourrons pas l'utiliser pour des tâches générales. Les unités de sortie connectées à nos unités d'entrée par des poids suivis d'une non-linéarité ne peuvent se déclencher que si la corrélation de leurs poids avec les entrées est supérieure à une certaine valeur. Ce qui peut en effet être utile pour quelques tâches relativement simples comme le fait de séparer deux classes de points linéairement séparables, mais pas pour la plupart des data sets modernes. En revanche dès que nous commençons à avoir des problèmes plus sophistiqués ce type de modèle est trop faible pour obtenir de bons résultats. On dit que le modèle est "biaisé" car son caractère linéaire représente un biais qui l'empêche de saisir toute la complexité du problème. Donc pour résoudre ça, nous améliorons notre modèle en ajoutant des couches cachées pour obtenir un réseau de neurones.





En même temps l'ajout des couches cachées supplémentaires peut causer l'overfitting qui affecte négativement le modèle en réduisant l'efficacité et la précision parce qu'il commence à apprendre les bruits et les données inexactes existant dans le tableau d'apprentissage.





Quant au naïf bayésien, c'est un classificateur probabiliste, ce qui signifie qu'il prédit sur la base de la probabilité d'un objet. Il appartient à une catégorie de modèles dits génératifs. Cela signifie que pendant la formation (la procédure où l'algorithme apprend à classer), ce classifieur essaie de savoir comment les données ont été générées en premier lieu. Il essaie essentiellement de comprendre la distribution sous-jacente qui a produit les exemples que nous avons entrés dans le modèle. D'autre part, le réseau de neurones est un modèle discriminant. Il essaie de comprendre quelles sont les différences entre nos exemples positifs et négatifs, afin d'effectuer la classification.

Une raison potentielle de choisir des réseaux de neurones artificiels plutôt que le classifieur naïf bayésien est les corrélations entre les variables d'entrées. Le classifieur naïf bayésien suppose que toutes les variables d'entrées sont indépendantes. Si cette hypothèse n'est pas correcte, cela peut avoir un impact sur la précision du classificateur. Un réseau de neurones avec une structure de réseau appropriée peut gérer la corrélation/dépendance entre les variables d'entrée. En outre si cette hypothèse est correcte le classifieur naïf bayésien peut nous faire gagner beaucoup de temps car il peut être plus performant que les autres modèles et nécessite beaucoup moins de données d'apprentissage.

V. Conclusion

Finalement nous avons conclu que le choix d'un classifieur dépend sur les conditions suivantes :

- Le classifieur naïf bayésien pour la simplicité et la rapidité d'une solution.
- Le réseau de neurones sans couche pour les problèmes simples et linéairement séparables.
- Le réseau neurone avec des couches cachés pour les problèmes complexes.

Veuillez trouver ci-dessous le lien pour le répertoire GitHub qui contient le code :
https://github.com/vlad2000m/Deep-Learning