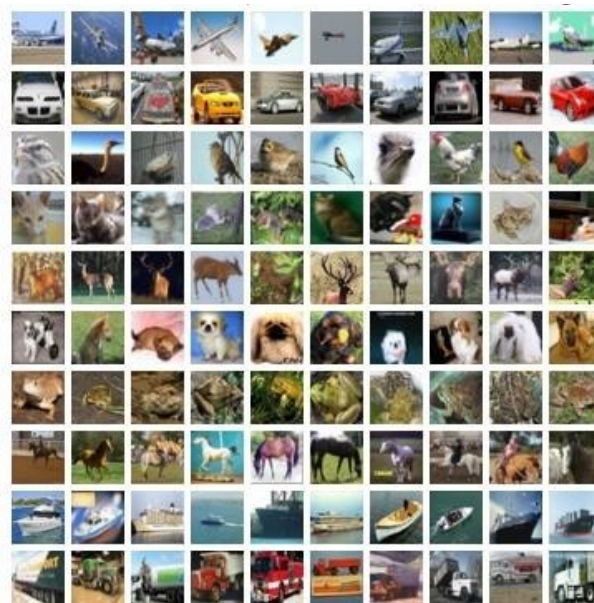


# Rapport de soutenance

# la reconnaissance d'images



Réalisé par :

-Nafaa SI SAID

-Youcef LASLA

# **SOMMAIRE :**

## **I. Introduction**

## **II. Base de données CIFAR-10**

- 1) Explication et manipulation des données
- 2) chargement des données

## **III. Approche Naïve (Apprentissage supervisé)**

- 1) K Plus Proche Voisins (K Nearest Neighbors)
- 2) Régression logistique
- 3) Bayésien naïf

## **IV.Apprentissage non supervisé**

- 1) K\_Means (K\_moyenne)
  - 1-a) Explication
  - 1-b) Tests

## **V.comparaison entre les algorithmes**

## **VI.Conclusion**

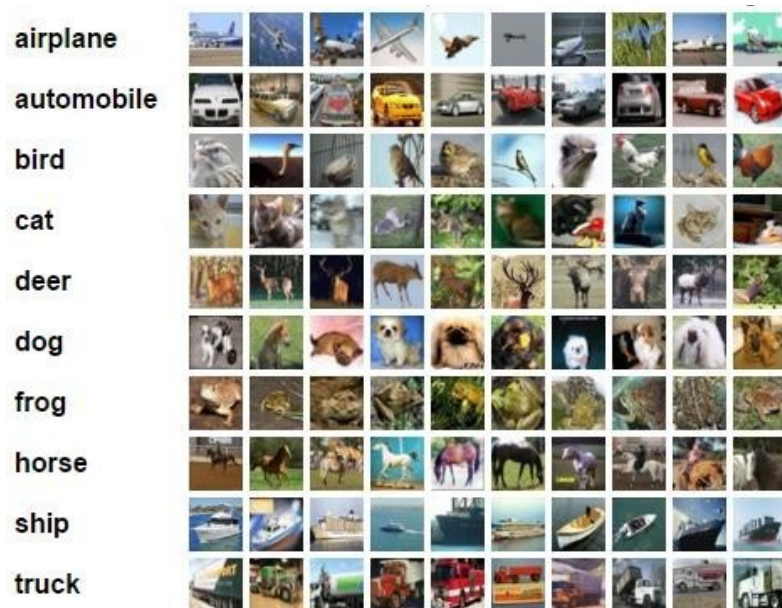
## **vii. annexes**

## I) Introduction :

L'objectif du projet consiste à développer un programme permettant de reconnaître des images selon les classes définies par notre base de données (CIFAR-10) en utilisant différents algorithmes (K-Means, KNN, ...). Dans ce rapport, nous allons commencer par analyser notre base de données et savoir comment elle fonctionne. Puis nous analyserons ses données en mettant en place deux types d'approches : l'approche naïve issue de l'apprentissage supervisé et l'autre qui est traitée par des algorithmes d'apprentissage non supervisé. Bien entendu, ce rapport expliquera également les termes importants dans l'apprentissage .

## II) Base de données CIFAR-10

La base de données CIFAR-10 contient des images réparties en classes :



Dans l'image ci-dessus , les classes sont représentées par les appellations définissant l'image (avions, voitures, chat, chien.....).

### 1) Explication et manipulation des données :

Les données sont structurées via un dictionnaire qui contient deux champs:

- 1) Data : qui est ordonnée 10000x3072 dans une liste numpy , chaque ligne de cette liste contient 32\*32 couleurs d'image, les couleurs primaires (Rouge, vert , bleu) sont arrangées dans cet ordre :
- les 1024 premières entrées sont pour la couleur rouge
  - les 1024 suivant sont pour la couleur verte
  - les 1024 derniers sont pour la couleur bleu

Dans notre code, pour faciliter le traitement d'un seul pixel, nous avons parcouru pour le i ème pixel, nous prendrons les i+1, i+1024 et i+2048 entrées pour i allant de 0 à 31.

2)Labels : une liste de 10000 nombres décimaux, ça représente les labels des datas (chat, ....)

Nous avons également à notre disposition un dictionnaire qui possède les entrées :

Label\_names : une liste de 10 éléments donnant le nom de chaque classes.

Exemple :

```
label_names[0] == "airplane",  
label_names[1] == "automobile"  
label_names[2] == "bird"
```

ainsi de suite suivant l'ordre donné dans l'image ci-dessus.

Afin de ne pas tomber sur des temps d'attente trop long à cause des données volumineuses , nous allons seulement travailler sur les données du « batch1 ».

Après l'extraction de toutes les données (images) et labels du batch1 on essaiera de les spliter en deux sous ensembles ( 80 % des données pour l'apprentissage, et les 20 % restantes pour tester nos modèles avec afin de pouvoir trouver les taux d'erreur.

## 2) chargement des données :

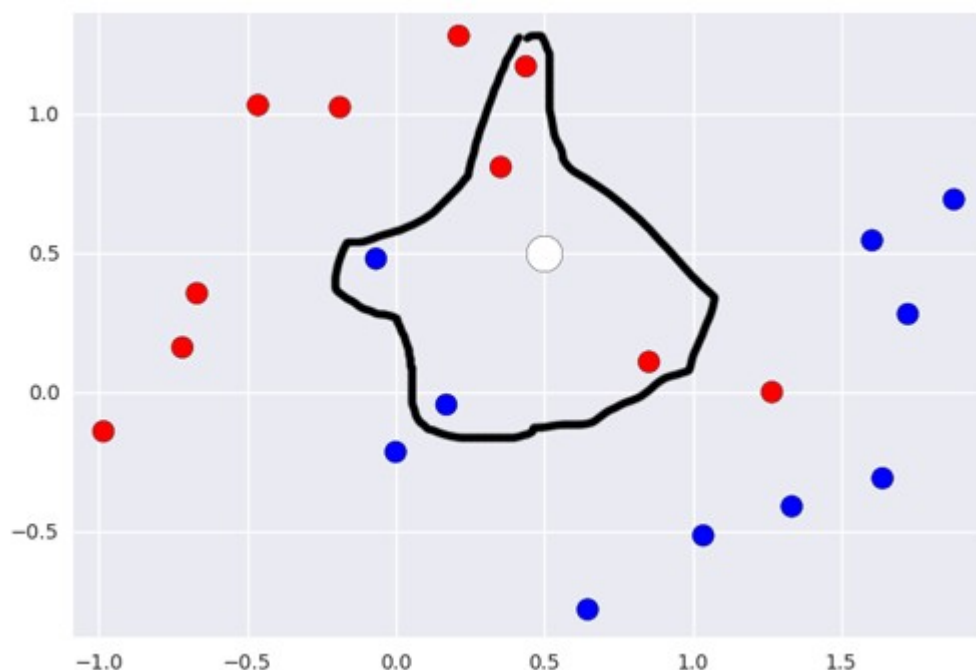
on avait chargé toutes les données CIRFA10 , mais nous avons utilisé que celles du patch1 pour tester les algorithmes, on peut bien-sur utiliser toutes les données du fichier CIRFA10, mais cela va nous prendre beaucoup de temps.

Le tableau (Images) contient toutes les images de tous les fichiers  
le tableau (Labels) contient tous les labels de toutes les images de CIRFA10

(voir code dans l'annexe tout en dernier)

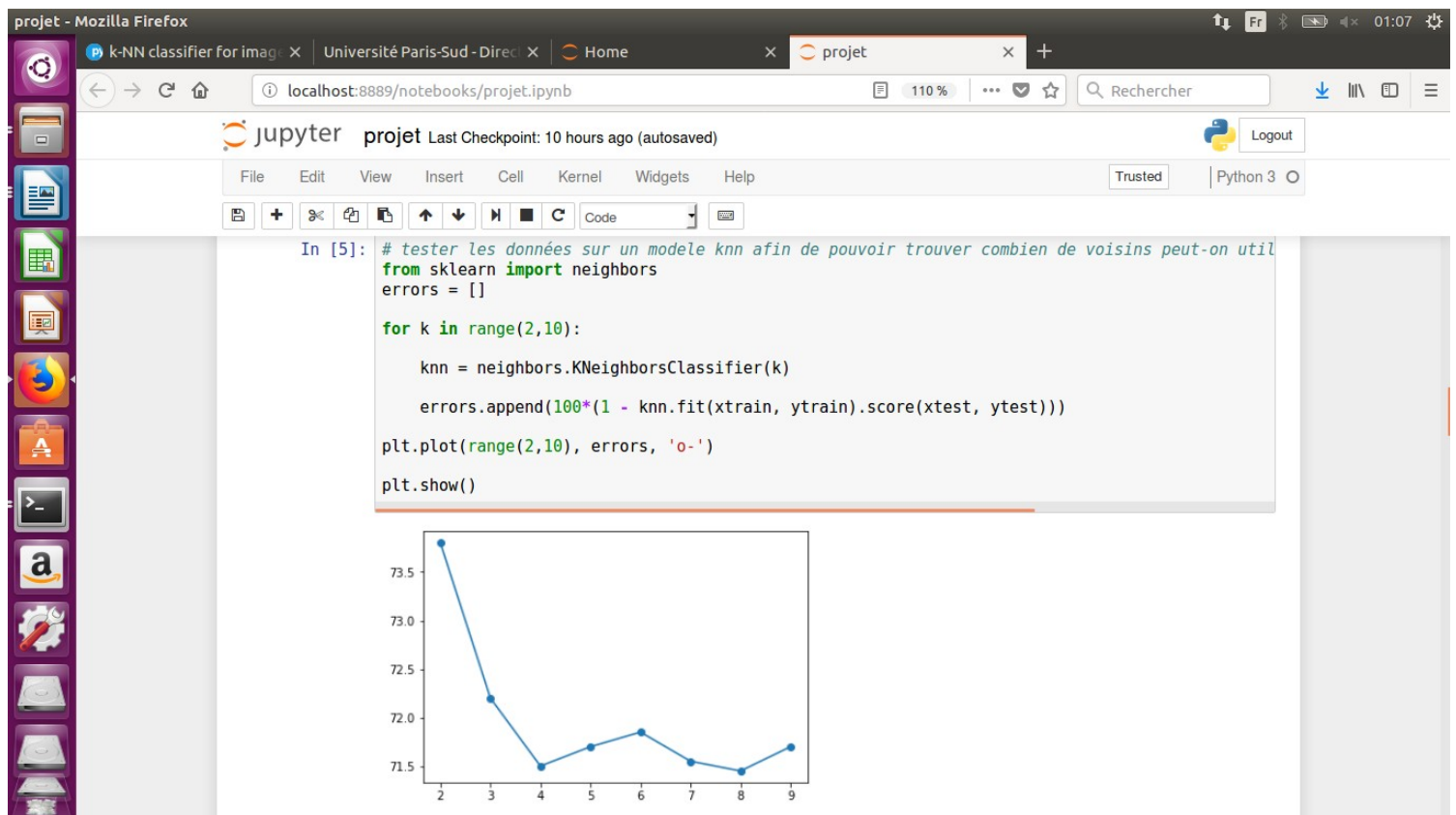
## III) Approche Naïve :

### 1) K Plus Proches Voisins :



Nous avons mis en place l'algorithme des K plus proches voisins qui est un algorithme d'apprentissage supervisé. Cet algorithme permet de classer les images en calculant la distance entre cette dernière et ses k plus proches voisins.

Le problème est dans le choix de nombres de voisins à prendre, pour cela nous avons essayé un petit algorithme d'affichage, on lui passant l'intervalle des nombres de voisins désirés, il va nous renvoyer un graphe qui nous guidera dans le choix de nombres de voisins et ça en fonction des scores obtenus. Notre faible taux d'erreur est avec  $k = 8$  avec un score de 71,5 %.



## 2) Régression logistique :

La régression logistique et la régression linéaire appartiennent à la même famille des modèles GLM (Generalized Linear Models) : dans les deux cas on relie un événement à une combinaison linéaire de variables explicatives. Pour la régression linéaire, la variable dépendante suit pas une loi normale  $N(\mu, s)$  où  $\mu$  est une fonction linéaire des variables explicatives. Pour la régression logistique, la variable dépendante, aussi appelée variable réponse, suit une loi de Bernoulli de paramètre  $p$  ( $p$  la probabilité moyenne pour que l'événement se produise), lorsque l'expérience est répétée une fois, ou une loi Binomiale( $n, p$ ) si l'expérience est répétée  $n$  fois (par exemple la même dose est essayée sur  $n$  insectes). Le paramètre de probabilité  $p$  est ici une fonction d'une combinaison linéaire des variables explicatives.

## 3) Bayésien naïf:

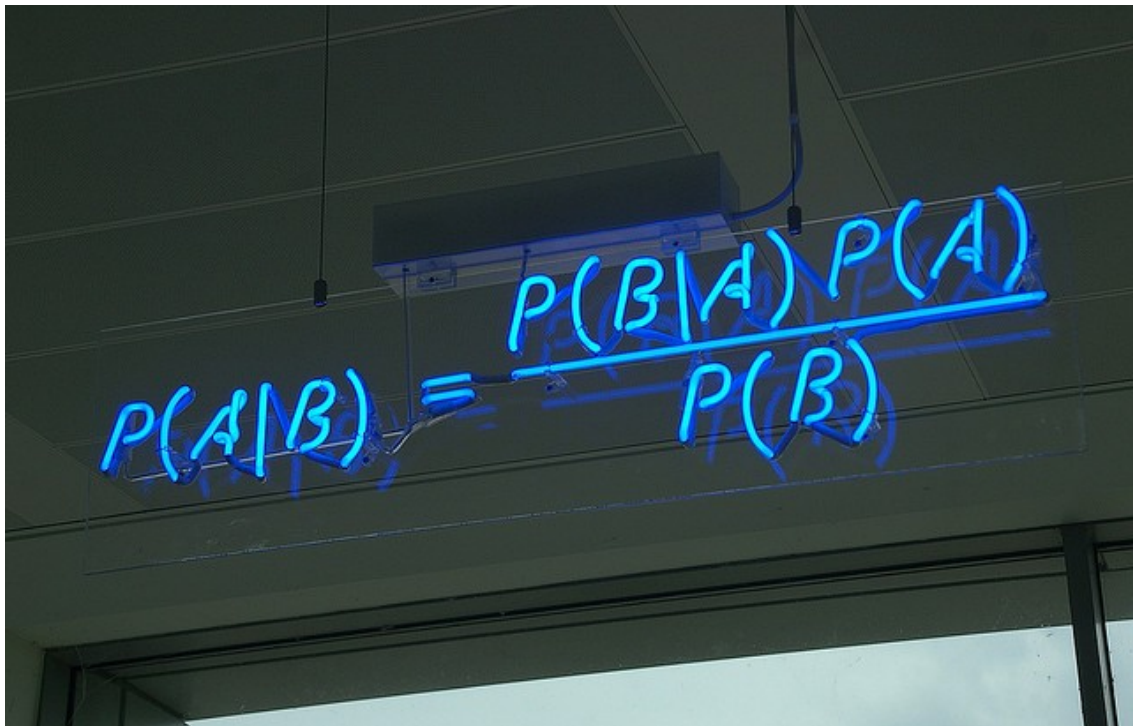
La classification naïve bayésienne est un type de classification Bayésienne probabiliste simple basée sur le théorème de Bayes (regardez la figure ci-dessous) avec une forte indépendance (dite naïve) des hypothèses. Elle met en œuvre un classifieur bayésien naïf, ou classifieur naïf de Bayes, appartenant à la famille des classifieurs linéaires.

Un terme plus approprié pour le modèle probabiliste sous-jacent pourrait être «modèle à caractéristiques statistiquement indépendantes».

En termes simples, un classifieur bayésien naïf suppose que l'existence d'une caractéristique pour une classe, est indépendante de l'existence



d'autres caractéristiques. Prenons l'exemple de (CIRFA 10) un chat peut être considéré comme un chien s'il est marron, d'une taille moyenne ,.... Même si ces caractéristiques sont liées dans la réalité, un classifieur bayésien naïf déterminera que l'animal en question est un chat en considérant indépendamment ces caractéristiques de couleur, de forme et de taille.


$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$



## IV) Apprentissage non supervisé :

### 1) K-Means :

#### 1-a) Explication de l'algorithme :

L'algorithme des K-Means sur les images est une autre méthode de reconnaissance d'images. Elle est différente de K plus Proches Voisins car pour créer un cluster (et donc regrouper les images de même « type ») on utilise ce que l'on appelle un centroïde.

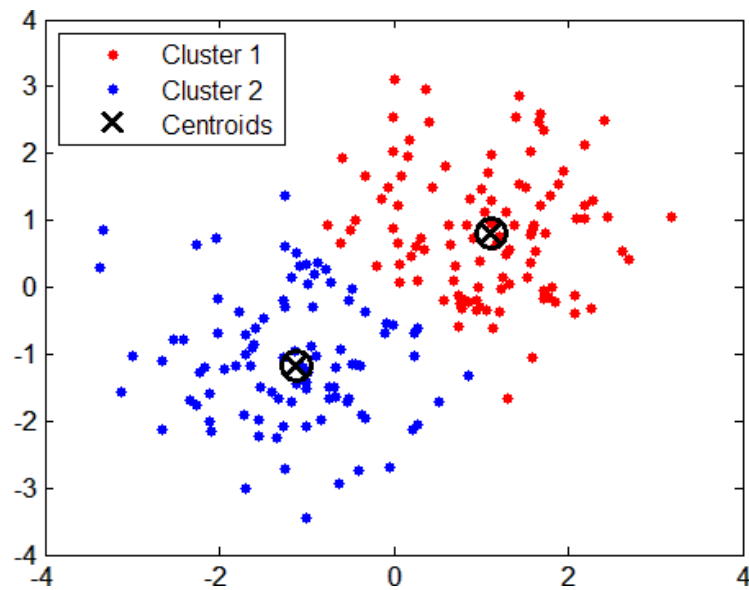
Au départ, on associe chaque cluster à chaque image, et le but sera d'avoir des clusters qui regroupe les images avec une distance (la norme) la ou les plus petites selon la classe d'images.

Un centroïde est un point qui se situe au centre d'un cluster. Pour nous, le centroïde est une image donc elle contient  $1024 * 3$  pixels.

Dans notre algorithme, nous allons dans un premier temps créer un centroïde placé de manière aléatoire, puis au fur et à mesure que nous avons une similarité (distance la plus proche entre les points) au niveau des cluster le centroïde va converger vers le centre du cluster, au milieu des points que l'on jugera similaire.

Lorsque le point du centroïde ne bougera plus, cela voudra dire qu'il est précisément au centre d'un cluster.

comme l'indique dans le graphe ci- dessous :



Une fois l'algorithme mis en place, il faut fixer le nombre d'itération maximum, en effet il peut y avoir un nombre important d'itérations qui permettront aux centroïdes d'avoir la meilleure précision possible.

L'algorithme s'arrête également si le centroïde actuelle est le même que le précédent, cela veut dire que le centroïde a atteint le milieu du cluster et qu'il est le plus précis possible.

Quand on teste les K-Means, on ne sait pas vraiment quel cluster représente quelle classe. Ce que nous faisons pour calculer le taux d'erreur, c'est que dans chaque cluster on récupère la classe dominante et on compare toutes les images de ce même cluster pour voir si leur classe est bien la classe dominante.

Ce qui veut dire qu'on peut avoir sur les 10 clusters, 2 qui représentent des chiens et 0 des avions.

Il faut donc faire attention à ne pas confondre classes et clusters, ce sont deux ensembles différents.

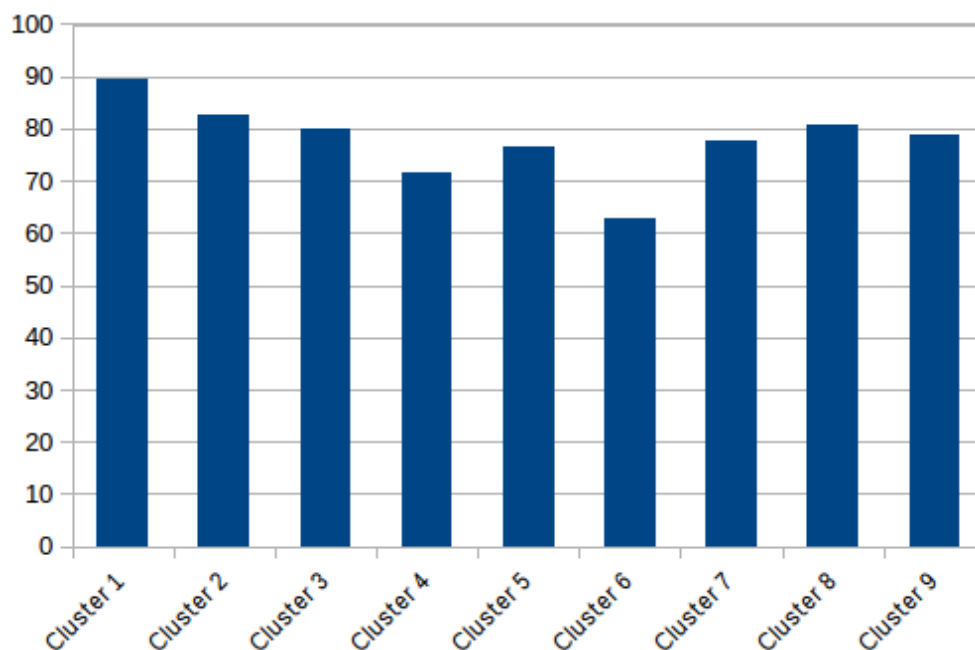
Puis nous avons mis en place un taux qui vérifie le pourcentage d'erreur de la classe dominante en fonction de la classe dans cluster. Nous avons ici deux paramètres à faire varier, le nombre de clusters et le nombre d'itérations max.

### 1-b) Tests :

pour obtenir le meilleur taux d'erreur, tout en gardant une cohérence dans nos cluster , nous avons utiliser le même nombre pour les clusters ainsi que pour le nombre de groupes que nous souhaitons classer.

Pour les K-Means, on obtient un taux d'erreur moyen à 69,9% (pour le 1<sup>er</sup> fichier batch). A noter qu'il s'agit du taux d'erreur par rapport à la classe dominante du cluster et non pas par rapport au label (l'identifiant du cluster est différent de l'identifiant de la classe).

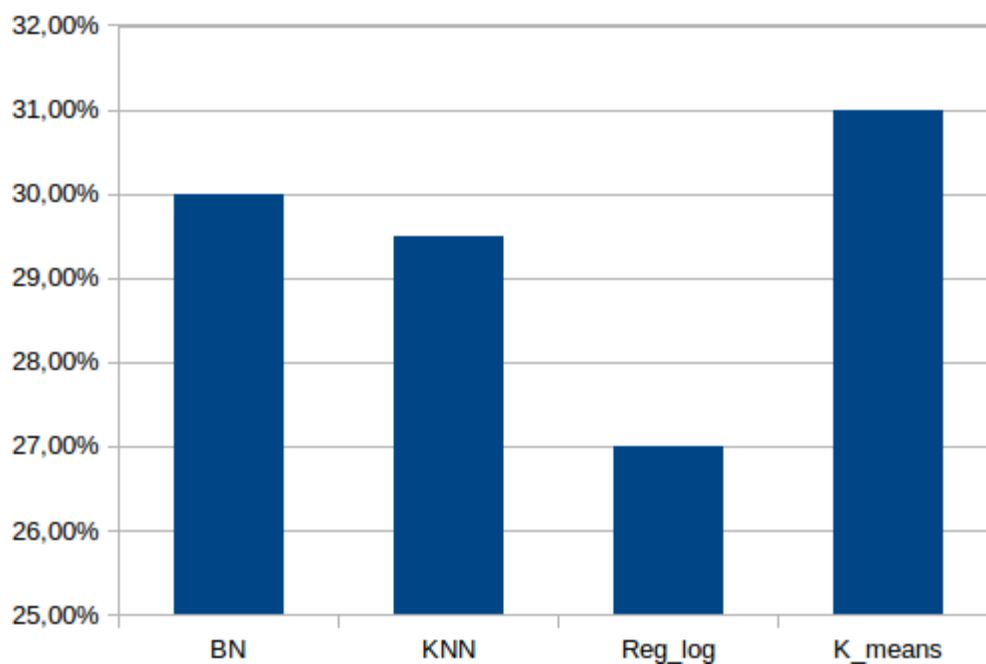
Le taux d'erreur sur les différents clusters : (cluster 0 est vide)



## V) Comparaison entre les algorithmes :

Après avoir tester nos données sur les algorithmes que nous avons utilisé, nous avons constaté qu'ils donnent pas un bon score d'apprentissage, le score le plus élevé qu'on avait pu avoir était le k\_means ( somme de tous les scores des clusters / par le nombre de clusters) avec (31% de réussite ce qui n'est pas bon pour faire de l'apprentissage)

on peut trouver les taux de réussite des différents algorithmes dans le diagramme ci-dessous :



## VI) Conclusion

Ce projet nous a appris à nous adapter par rapport aux contraintes (images similaires, quantité de la base, paramètres multiples), et à travers plusieurs méthodes, nous avons pu voir comment fonctionnait les mécanismes d'apprentissage, les différentes évolutions de notre propre travail (k\_means , KNN, .....). Ainsi le meilleur taux que nous avons pu obtenir est de 69% d'erreurs en parcourant le fichier batch1.

## VII) Annexes

### 1) K-means

```
In [1]: import pickle as cPickle
import numpy as np
import random
random.seed(1) # set a seed so that the results are consistent

#chargement du fichier cifar-10-batches-py qui contient 5 fichiers patches
path = '/home/nafaa/Documents/projet_IA/cifar-10-batches-py/'

# recuperer le dictionnaire du premier patch
f1 = open(path+'data_batch_1', 'rb')
dict1 = cPickle.load(f1, encoding='latin1')

# récupérer le dictionnaire du 2ème patch
f2 = open(path+'data_batch_2', 'rb')
dict2 = cPickle.load(f2, encoding='latin1')

f3 = open(path+'data_batch_3', 'rb')
dict3 = cPickle.load(f3, encoding='latin1')

f4 = open(path+'data_batch_4', 'rb')
dict4 = cPickle.load(f4, encoding='latin1')

f5 = open(path+'data_batch_5', 'rb')
dict5 = cPickle.load(f5, encoding='latin1')

images = [] # mettre les images de tous les fichiers dans un seul
```

```

#recuperer les images de tous les patch
images1 = dict1['data'] # les images du patch 1
for i in images1 :
    images.append(i)

    images2 = dict2['data']
for i in images2 :
    images.append(i)

images3 = dict3['data'] # les images du patch 3
for i in images3 :
    images.append(i)

images4 = dict4['data']
for i in images4 :
    images.append(i)

images5 = dict5['data']
for i in images5 :
    images.append(i)

# print(images) affichage du tableau de toutes les images

#images = np.reshape(images, (10000, 3, 32, 32))
labels = [] #mettre les labels de tous les fichiers dans un seul

# récupérer les labels des images
labels1 = dict1['labels'] # labels des images du patch1
for i in labels1 :
    labels.append(i)

labels2 = dict2['labels']

```

```

# récupérer les labels des images
labels1 = dict1['labels'] # labels des images du patch1
for i in labels1 :
    labels.append(i)

labels2 = dict2['labels']
for i in labels2 :
    labels.append(i)

labels3 = dict3['labels'] # labels des images du patch 3
for i in labels3 :
    labels.append(i)

labels4 = dict4['labels']
for i in labels4 :
    labels.append(i)

labels5 = dict5['labels']
for i in labels5 :
    labels.append(i)

# print(labels) affichage des labels de toutes les images

```

```

In [2]: %matplotlib inline
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans

```

```

In [5]: # creation des clusters
def K_means(k, images, centroids, taille, max_iter):
    # clusters initialisés à vide
    clusters = None
    labels = None
    meme = False
    # nombre d'itérations initialisées à 0
    it = 0

    while (not(meme) and it < max_iter):
        # on copie les centroids de l'itération it-1
        last_centroids = list(centroids)

        # On reinitialise les clusters
        clusters = [[] for x in range(k)]
        labels = [[] for x in range(k)]

        # On associe un cluster à chaque image
        for j in range(taille): # Pour chaque image
            # calculer la distance entre l'image en question et le centroid du premier cluster
            d = np.linalg.norm(images[j] - centroids[0])
            c = 0 # prenons le centroid 0 comme étant le plus proche
            for i in range(1, k):
                # mettre à jour le centroid le plus proche
                # en calculant la distance entre l'image et les autres clusters
                dist = np.linalg.norm(images[j] - centroids[i])
                if (d > dist):
                    c = i
                    d = dist
            clusters[c].append(images[j]) # affecter une image à un cluster
            labels[c].append(j) # enregistre les labels des images

```

```

        labels[c].append(j) # enregistre les labels des images

    # Mise à jour de chaque centroid en faisant la moyenne des points
    for i in range(k):
        # calculer la moyenne des clusters
        if (len(clusters[i]) != 0):
            moy = np.mean(clusters[i], axis=0) # calculer la moyenne
        else:
            moy = np.mean([0], axis=0)
        centroids[i] = np.copy(moy)

    # Tester si les centroids de l'itération précédente == centroids de l'itération actuelle
    meme = True
    for i in range(k):
        # np.array_equal(x,y) renvoie true si x == y
        if (not(np.array_equal(centroids[i], last_centroids[i]))):
            meme = False

    it += 1
    # retourner les clusters, centroids et labels
    return clusters, centroids, labels

```

```

In [6]: nb_clusters = 10

# création de 10 centroids (10 tableaux de 3072 valeurs comprises entre 0 et 256)
centroids = [ np.random.randint(256, size=3072) for x in range(nb_clusters)]

# application du K_means sur les images de patch1
cl, centr, lab = K_means(nb_clusters, images1, centroids, len(images1), 10)

```



```

In [7]: # calculer le taux d'erreur pour chaque cluster
def eval_k_means_per_classes (clusters, centroids, labels, real_labels, erreur):
    for i in range(len(clusters)):
        classes = [0 for x in range(10)]
        # Calcul de la classe dominante
        for j in range(len(clusters[i])):
            classes[ real_labels[labels[i][j]] ] += 1

        # On détermine quelle classe est la plus représentée
        ind = 0
        value = classes[0]
        for j in range(1, len(classes)):
            if (classes[j] > value):
                ind = j
                value = classes[j]

        # On détermine le taux d'erreurs par clusters
        errors = 0
        for j in range(len(clusters[i])):
            if(ind != real_labels[labels[i][j]]):
                errors+=1
        if(len(clusters[i]) != 0):
            print("le nombre d'elements du cluster ",i," est :",len(clusters[i]))
            print("le taux d'erreurs est :", (float(errors)/ len(clusters[i])))
            print("-----")
            erreur.append(float(errors)/ len(clusters[i]))
        else:
            print("le cluster ",i,"est vide")
        print("le taux d'erreur moyen est",np.sum(erreur)/10)
    return erreur

```

```

In [ ]: # affichage des taux d'erreur des clusters
import matplotlib.pyplot as plt
#diagramme en batons
fig = plt.figure()
#les clusters sachant que le cluster 0 est vide
x = [1,2,3,4,5,6,7,8,9]
width = 1.0
plt.bar(x, erreur, width)
plt.title("le taux d'erreurs des clusteurs")
plt.savefig('erreurs.png')
# fonction d'affichage du diagramme en batons
plt.show()

```

## 2) Approche naïve :

```
In [2]: # récupérer les noms des labels (label_names)

def load_classes():
    file = 'batches.meta'

    f = open(file, 'rb')
    dict = cPickle.load(f)
    return dict['label_names']

def print_classes(label_names):
    for i in range(0, 10):
        print (str(i) + " : " + label_names[i] + " ")

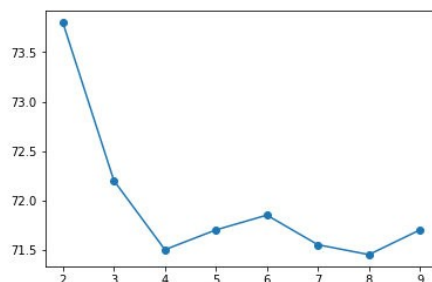
label_names = load_classes()
print_classes(label_names)
```

```
In [5]: # tester les données sur un modele knn afin de pouvoir trouver combien de voisins peut-on utiliser pour avoir un bon score
from sklearn import neighbors
errors = []

for k in range(2,10):
    knn = neighbors.KNeighborsClassifier(k)

    errors.append(100*(1 - knn.fit(xtrain, ytrain).score(xtest, ytest)))

plt.plot(range(2,10), errors, 'o-')
plt.show()
```



```
In [4]: # plus proches voisins
from sklearn import neighbors
def knn_mod (data_set, labels_set, data_test):
    knn = neighbors.KNeighborsClassifier(n_neighbors=6)
    knn.fit(xtrain, ytrain)
    return (knn.predict(data_test))
```

```
In [80]: # bayésien naïf
from sklearn.naive_bayes import GaussianNB
def BNaif (data_set, labels_set, data_test):
    clf = GaussianNB()
    clf.fit(xtrain, ytrain)
    return (clf.predict(data_test))
```

```
In [8]: #regression logistique
from sklearn import linear_model
def Reg_Log (data_set, labels_set, data_test):
    cf = linear_model.LogisticRegression()
    cf.fit(xtrain, ytrain)
    return cf.predict(xtest)
```

```
In [5]: def taux_reussite (X, labels_test):
    j = 0
    s = 0
    for i in labels_test :
        if i != X[j]:
            s = s + 1
        j = j + 1

    return "le taux de réussite est : ", 100 - ((s * 100)/len(X))
```

```
In [5]: def taux_reussite (X, labels_test):  
        j = 0  
        s = 0  
        for i in labels_test :  
            if i != X[j]:  
                s = s + 1  
            j = j + 1  
  
        return "le taux de réussite est : ", 100 - ((s * 100)/len(X))
```

```
In [83]: z = BNaif (xtrain, ytrain, xtest)  
print("le taux de réussite de l'algorithme bayésien_naif est : ", taux_reussite(z, ytest))  
le taux de réussite de l'algorithme bayésien naif est : ('le taux de réussite est : ', 30.0)
```

```
In [84]: z = knn_mod(xtrain, ytrain, xtest)  
print("le taux de réussite de l'algorithme knn est:",taux_reussite(z, ytest))  
le taux de réussite de l'algorithme knn est: ('le taux de réussite est : ', 29.5)
```

```
In [9]: z = Reg_Log(xtrain, ytrain, xtest)  
print("le taux de réussite de l'algorithme Reg_Log est :",taux_reussite(z, ytest))  
le taux de réussite de l'algorithme Reg_Log est : ('le taux de réussite est : ', 24.0)
```

# FIN