Rapport de projet final
Reconnaissance d'images

Enseignant: Guillaume Wisniewski

par

Liang Fei

Gao Jiaxin

UNIVERSITE PARIS-SUD Master Informatique année 2017-2018

1. Introduction

Ce projet est dans le but de développer un système pour reconnaître de différentes images qui permet d'identifier automatiquement des objets existants sur une image et de les mettre dans les classes correspondantes. L'idée principale de ce projet est d'abord d'utiliser différents algorithmes naïfs pour classer les images, et ensuite une autre méthode proposé par A.Coates et al pour déterminer la classe.

La reste de l'article est organisé comme suit: dans la deuxième partie, nous avous expliqué l'analyse de base de données Cifar-10 permettant de répondre des questions proposées sur le sujet, dans la troisième partie, nous vous présentons la réalisation du projet contenant l'algorithme naïve en utilisant les classifieurs. Dans la quartrième partie, nous avons présenté des idées sur l'article. Enfin, nous vous donnons une conclusion.

Le projet est réalisé en binôme. Liang Fei s'est occupé plus la partie de l'analyse de données, Gao Jiaxin s'est occupé la partie de l'algorithme naïfs. Pour la dernière partie, nous avons fait ensemble.

2. Analyse de base de données

Dans ce projet nous utilisons le dataset CIFAR-10.
Il y a 60000 images couleur de taille 32x32 dans la base de donée CIFAR-10,

50 000 sont des images d'entraînement et 10 000 sont des images de test,

répartiés en 10 catégories au total. C'est-à-dire qu'il y a 6 000 images dans chaque catégorie. Les images sont affectées à cinq fichiers d'entraînement et un fichier de test. Chaque fichier contient 10000 images. Pour le fichier de test, il sélectionne au hasard 1000 images pour chaque catégorie.
Pour les fichiers d'entraînement, ils contiennent les 50 000 images restantes. Cependant, chaque fichier peut contenir plus d'images d'une classe que d'une autre. Les fichiers d'entraînement contiennent un total de 5000 images pour une classe.

2.1 Format de la base

Dans le répertoire de la site, il y a 5 "data_batch" fichiers, un "test_batch" fichier et un "batches.meta" fichier.
On utilise la fonction **unpickle()** pour obtenir le dictionnaire qui contient des images de fichier.

```
import cPickle
def unpickle(file):
    with open(file, 'rb') as fx:
        dict = cPickle.load(fx)
    return dict
```

On a utilisé le exemple “data_batch_1” pour analyser la base de donée de les 6 premiere fichiers.

```
train_set = unpickle('./cifar-10-batches-py/data_batch_1')
print(train_set.keys())
```

['data', 'labels', 'batch_label', 'filenames']

Donc, les dictionnaires des 6 premiere fichiers sont avec quartre élements, “data”, “labels”, “batch_label” et “filenames”.
Pour analyser le permier élement data, on utilise la sixième image de le “data _batch_1”.

```
train_set = unpickle('./cifar-10-batches-py/data_batch_1')
img = train_set['data']
single_img = np.array(img[5])
print(type(img))
print(len(img))
print(single_img)
print(len(single_img))
```

Et nous avons obtenu:

<type 'numpy.ndarray'>
10000
[159 150 153 ..., 14 17 19]
3072

Nous avons trouvé que c’est un tableau numpy avec la taille 10000x3072. Chaque ligne du tableau stocke une 32x32 image. C’est à dire, il y a 1024 pixels dans chaque image. Chaque pixel est présenté par trois chiffres qui signifient des valeurs des trois couleurs. Donc, on a 3072 chiffres pour décrire une image, les 1024 premières entrées contiennent les valeurs des canaux rouges, les 1024 suivantes contiennent les vertes et les 1024 dernières contiennent les bleues. L'image est stockée dans l'ordre row-major, les 32 premières entrées du tableau correspondent aux valeurs de canal rouge de la première ligne de l'image.

Pour l’élement “labels”, nous avons affiché la longueur, type et contenu :

```
print(type(train_set['labels']))
print(len(train_set['labels']))
print(train_set['labels'])
```

et nous avons obtenu **<type 'list'>**, **10000**, et une liste
2, 6, 6, 1, 7, 2, 7, 5, 5, 3, 8, 2, 0, 8, 4, 4, 8, 4, 5, 5, 7, 6, 3, 2, 3, 8, 1, 1, 6,.....

Donc, c'est une liste de 10000 chiffres, chaque chiffres sont entre 0 et 9, ils signifient 10 type d'objet pour ces 10000 images.

Pour le élement "filenames"

```
print(type(train_set['filenames']))
f=train_set['filenames']
print(f[5])
```

Nous avons obtenu : **coupe_s_001735.png**. Donc, le "filenames" est une liste de le nom de chaque image, par example le "filenames" de cinquiéme image dans la "data_batch_1" est ''coupe_s_001735.png''.

La derniere fichier "batches.meta", il contient également un objet dictionnaire.

Pour la fichier de "test_batch", nous avons utilisé le meme manière que les data_batch. Et nous avons obtenu le meme chose.

```
test_set = unpickle('./cifar-10-batches-py/test_batch')
print(test_set.keys())
print(len(test_set))
print(type(test_set['data']))
print(len(test_set['data']))
print(type(test_set['labels']))
print(type(test_set['batch_label']))
print(type(test_set['filenames']))
```

et nous avons obtenu:
['data', 'labels', 'batch_label', 'filenames']
<type 'numpy.ndarray'>
10000
<type 'list'>
<type 'str'>
<type 'list'>

Nous avons écrit une partie du code comme suit pour afficher des images stockées dans les batchs, par exemple pour la sixième photo dans la "data_batch_1".

Nous devons d'abord obtenir des valeur des chaque couleur de cet image :

```
new = single_img.reshape(3,32,32)
red   = new[0].reshape(1024,1)
green = new[1].reshape(1024,1)
blue  = new[2].reshape(1024,1)
pic = np.hstack((red,green,blue))
pic_rgb = pic.reshape(32,32,3)
plt.imshow(pic_rgb)
```

et nous avons obtenu:

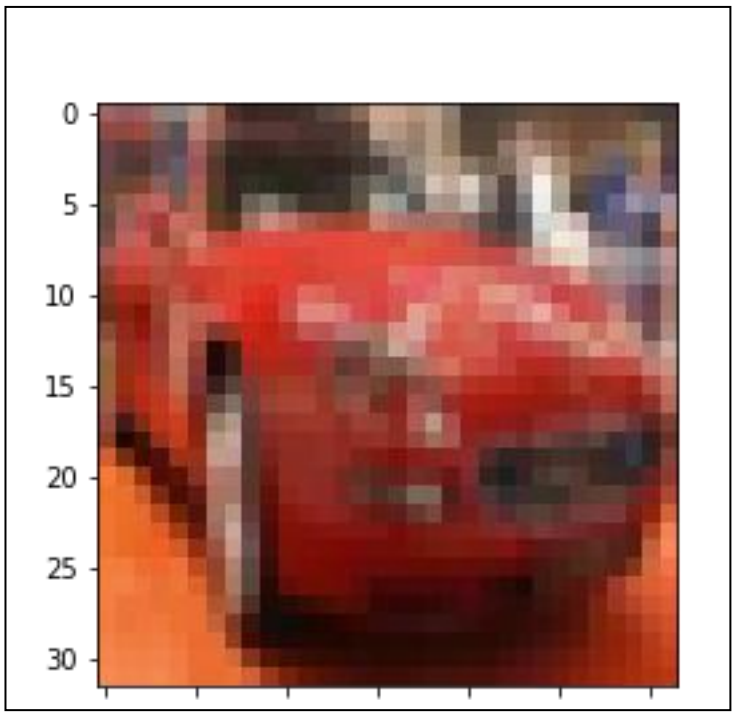

Figure2.1

2.2 Réponses aux question du sujet

- Type des image représentés(est-ce que la tâche est difficile pour un humain? pour un ordinateur? quel taux d'erreur peut-on espérer atteindre?)

C'est plus compliqué pour un ordinateur de identifier le type de une image. Mais cette tâche est facile pour un humain, Même pour un enfant de cinq ans, il peut aussi facilement identifier le type de image comme suit est une voiture. Parce que pour les humains, nous avons déja le structure conceptuelle des objet.

Mais pour un ordinateur, il y a beaucoup des difficultés pour identifier. Le premiere point, il ne sais pas qu'est ce que c'est, il doit d'abord apprendre. Et on doit transformer une image à 3072 chiffres et analyser les chiffres pour bien comprendre les caractéristiques de chaque type de objet.

Le deuxième point, les 10 classes de CIFAR-10 ne sont pas complètement différentes. Ils peuvent avoir les mêmes caractéristiques. Par exemple, les chats et les chiens peuvent avoir des yeux, et les cerfs et les chevaux ont des apparences similaires.

Le troisième point, le meme objet dans CIFAR-10 peut aussi être différent. Par exemple, les deux voitures de la figure ci-dessous peuvent avoir différentes tailles, positions, couleurs, ce qui est difficile pour l'ordinateur de choisir une norme à juger. Nous devons résoudre ces problèmes.

Reconnaître certaines images est facile pour les humains, mais il n'est pas si facile d'identifier des milliers d'images dans une certaine période de temps. Il peut y avoir des erreurs. Donc, il bien sur peut y avoir des erreurs pour l'ordinateur. Le taux d'erreur peut être utilisé pour mesurer la performance de l'algorithme. Selon les données en ligne, en 2010, le taux d'erreur a atteint d'environ 25%. Et en 2015, dans le ImageNet compétition, il atteint 3.5%.

- Statistiques générales du jeu de données (combien y a-t-il de classes? d'exemples par classe?)

Dans un document s'appelant “batches.meta”, nous savons qu’il est un dictionnaire contenant des objets python :

t = unpickle('./batches.meta')

print (t['label_names'])

Nous pouvons obtenir :

['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

Donc labels_names est d’une liste contenant 10 types d’objets différents correspondants aux valeurs de labels qui sont des chiffres entre 0 et 9.

3. Algorithmes naïfs

3.1 Les classifieur

Pour cette partie, nous utilisons différentes méthodes pour classer les images, les données d'entrée étant l'image elle-même(une liste avec 3072 chiffres). Ce sont des algorithmes naïfs. Les différents classifieurs que nous avons utilisé sont:

- L’arbre binaire de décision: C’est est un outil populaire dans l'apprentissage automatique. Il est utilisé dans l'analyse décisionnelle, pour aider à identifier une stratégie plus susceptible d'atteindre un objectif. Dans notre projet nous avons utilisé la fonction **tree.decisiontreeclasssifier( )** de la bibliothèque sklearn.

- Naïve SVC: C’est est un modèle d'apprentissage supervisé et un algorithme d'apprentissage connexe pour analyser des données dans une analyse de classification et de régression. Mais il est trop lent de lancer.

- L’algorithme Naïve Bayes: Dans l'apprentissage automatique, les classificateurs naïfs de Bayes sont une famille de «classificateurs probabilistes» simples basés sur l'application du théorème de Bayes avec des hypothèses d'indépendance fortes (naïves) entre les caractéristiques.

- La méthode des k plus proches voisins (KNN): C’est une méthode d’apprentissage supervisé, dans ce cadre, on dispose d’une base de données d'apprentissage constituée de N couples [entrée, sortie]. Pour estimer la sortie associée à une nouvelle entrée x, la méthode des k plus proches voisins consiste à prendre en compte (de façon identique) les k échantillons d'apprentissage dont l’entrée est la plus proche de la nouvelle entrée x, selon une distance à définir.

- Le perceptron classique: C’est un algorithme d'apprentissage supervisé de classifieurs binaires. 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.

- SGD Claissifier: L’avantage de ce méthod est plus rapide, mais la disavantage est SGD nécessite un certain nombre d'hyperparamètres.

- L’algorithme des k moyennes: On va faire l’introduction détaillée aprés.

Nous avons utilisé la fonctoin **xx.fit(x,y)** pour apprendre la base de donnée des “data_batches”. Et nous avons utilisé la fonction **np.mean()** pour calculer la valeur moyenne de performance. Et nous avons obtenu les résultats sur le test_set comme suivant.

Pour voir plus clairement, nous avons desiné les graphes comme suivant. l‘axe x présente les classes, et l’axe y présente les scores de classificatoin.

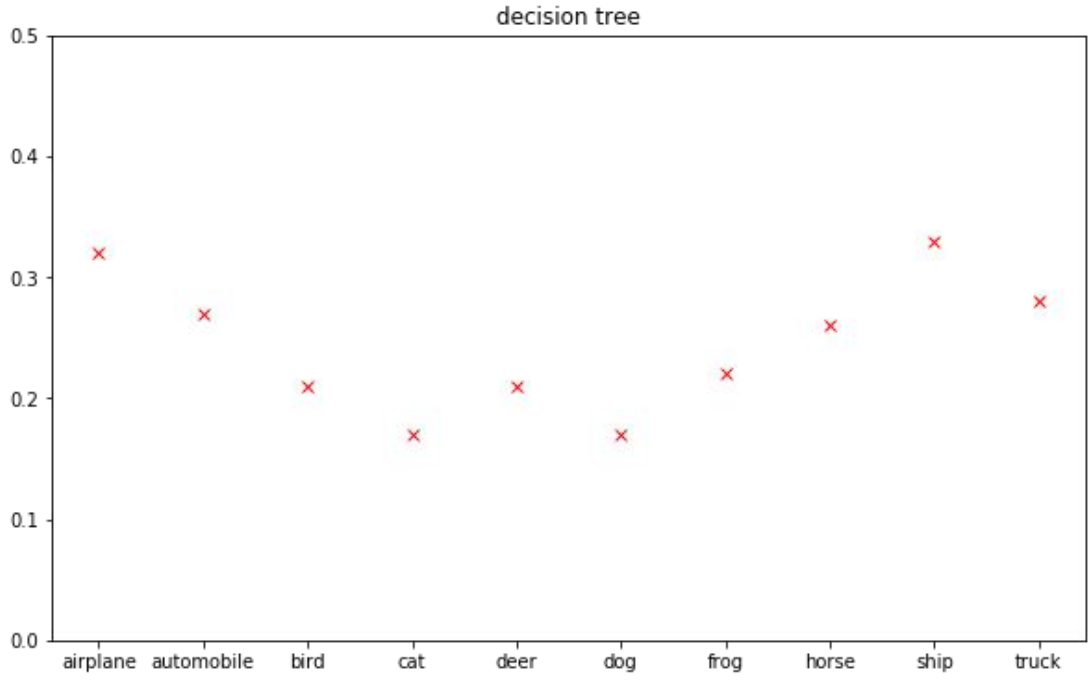

Figure 3.1.1

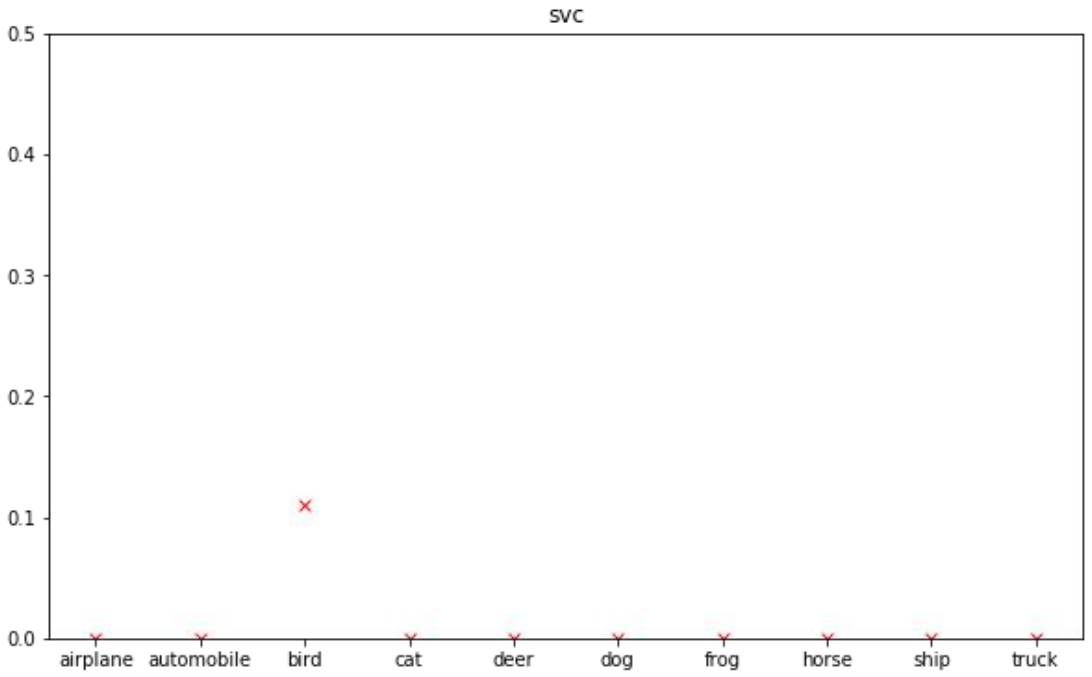

Figure 3.1.2

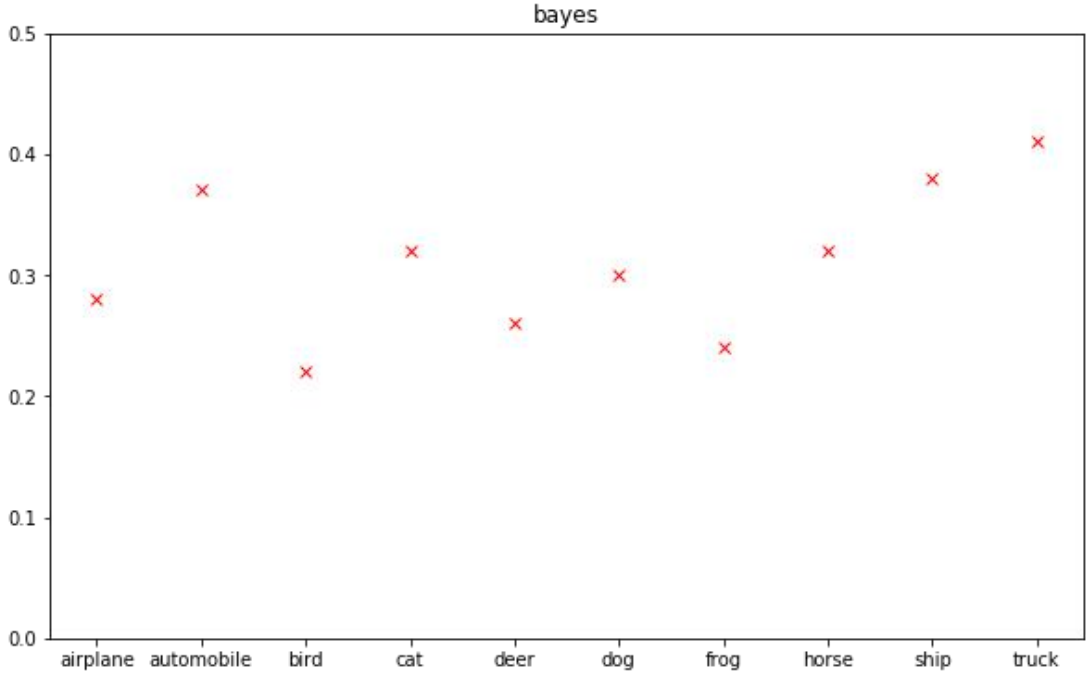

Figure 3.1.3

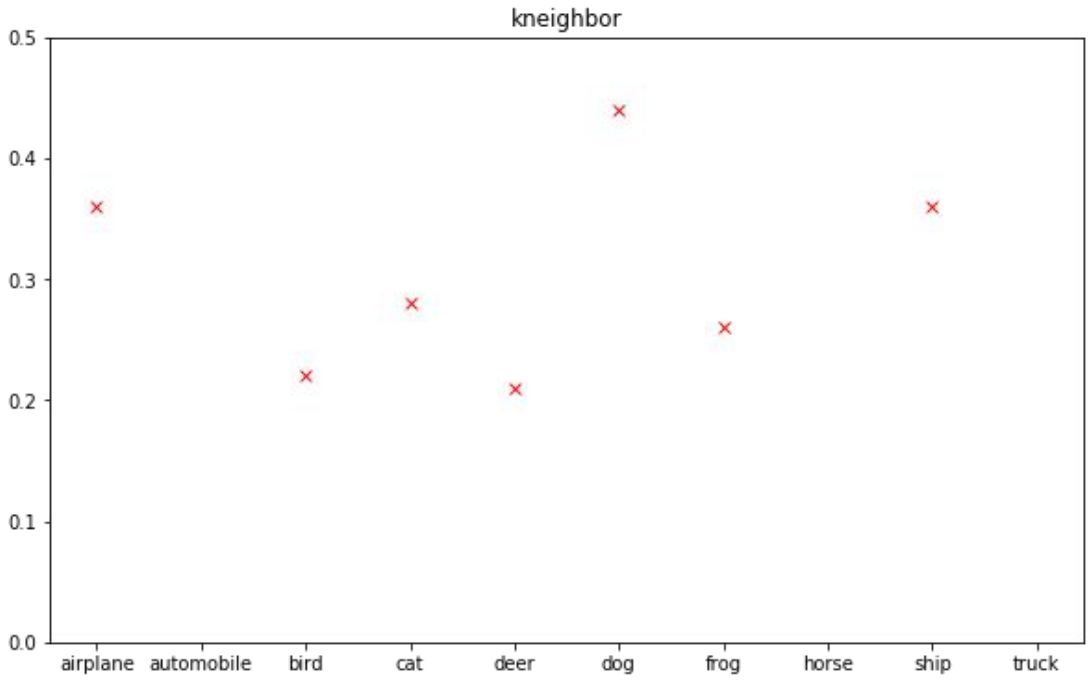

Figure 3.1.4

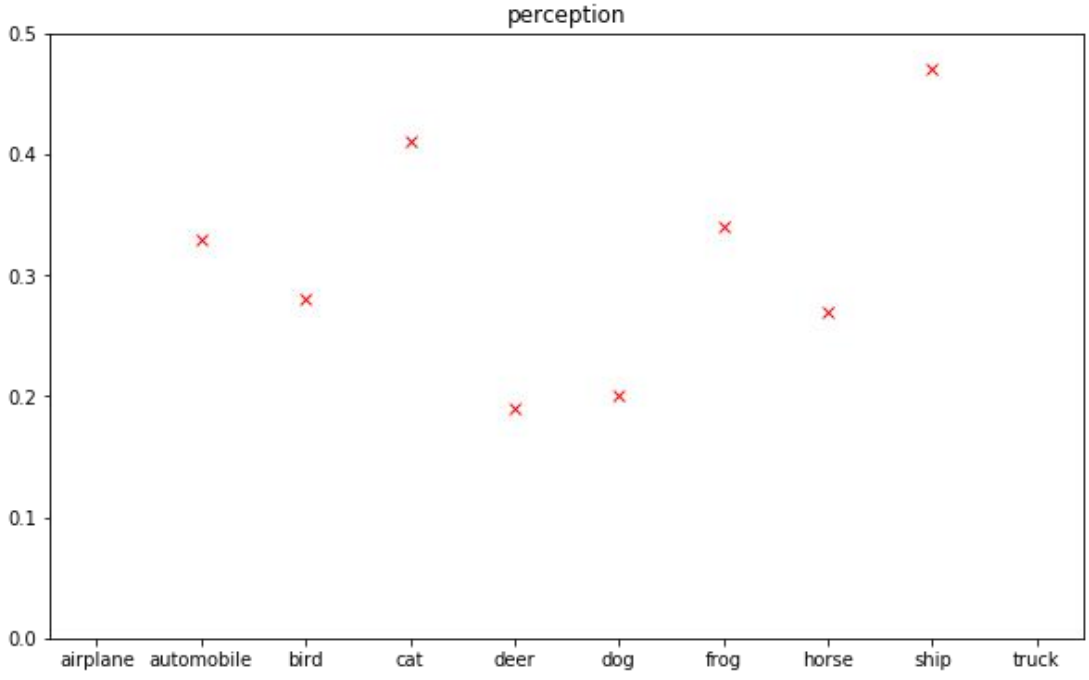

Figure 3.1.5

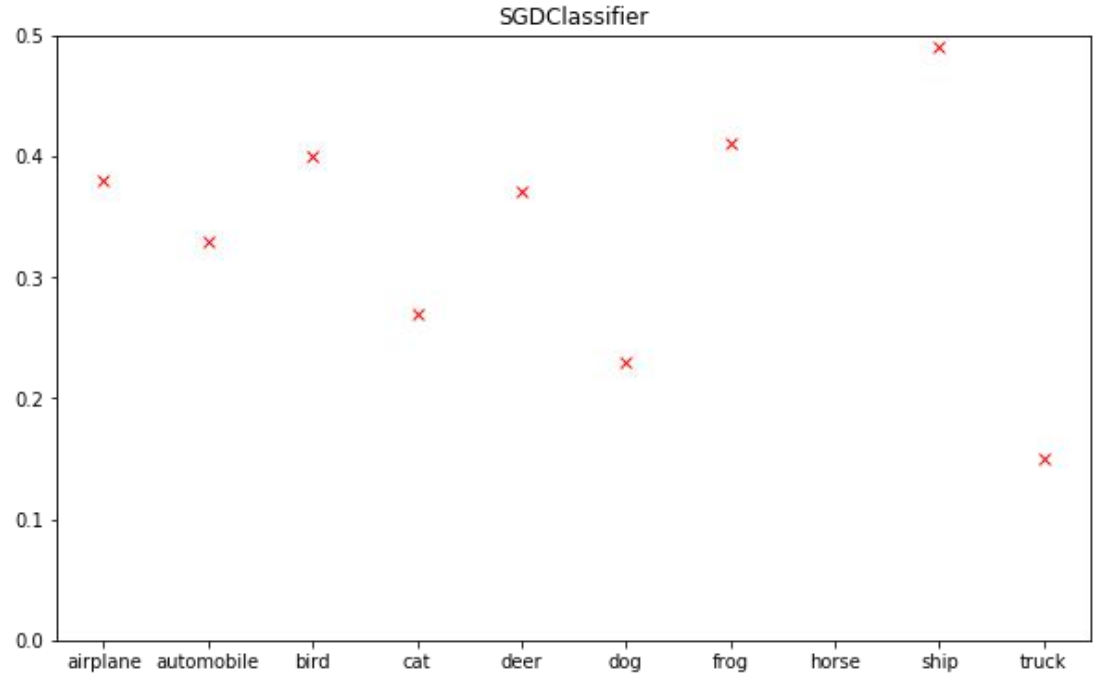

Figure 3.1.6

Dans ces méthodes, l'algorithme Naive Bayes et l'algorithme KNN(avec n_neighbors=12) avons obtenu le meilleur résultat, et le taux correct est d'environ 30%. La performance de l'algorithme SVC n'est pas très bonne, il classe toutes les classes dans une catégorie. La précision des autres algorithmes est d'environ 25%. Du point de vue de la catégorie, le "ship" est relativement faciles à identifier, et le "deer" est plus difficile à identifier.

3.2 Idée du code de méthode k-moyennes

k-moyennes est une méthode de partitionnement de données et un problème d'optimisation combinatoire. Étant donnés des points et un entier k, le problème est de diviser les points en k groupes, souvent appelés clusters, de façon à minimiser une certaine fonction. On considère la distance d'un point à la moyenne des points de son cluster ; la fonction à minimiser est la somme des carrés de ces distances. Étant donné un ensemble de points (x1, x2, …, xn), on cherche à partitionner les n points en k ensembles S = {S1, S2, …, Sk} (k ≤ n) en minimisant la distance entre les points à l'intérieur de chaque partition :

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in S_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2$$

où μi est le barycentre des points dans Si.

Nous avons deux façons d'utiliser algorithme k-moyennes, mais nous avons vu que les résultats sont moyens.

Pour la première façon, kmeans1, nous avons d'abord utilisé l'algorithme k-moyennes pour calculer un nombre de centres. Mais les labels ne sont pas exactement les labels correctes. Donc, nous avons écrit une fonction pour changer les labels(Par exemple, si les données sont étiquetées comme Tag 8, les Tag 5(Étiquette réelle) sont le plus, nous remplacons 8 par 5). A la fin, nous avons obtenu un taux de correction de 21%.

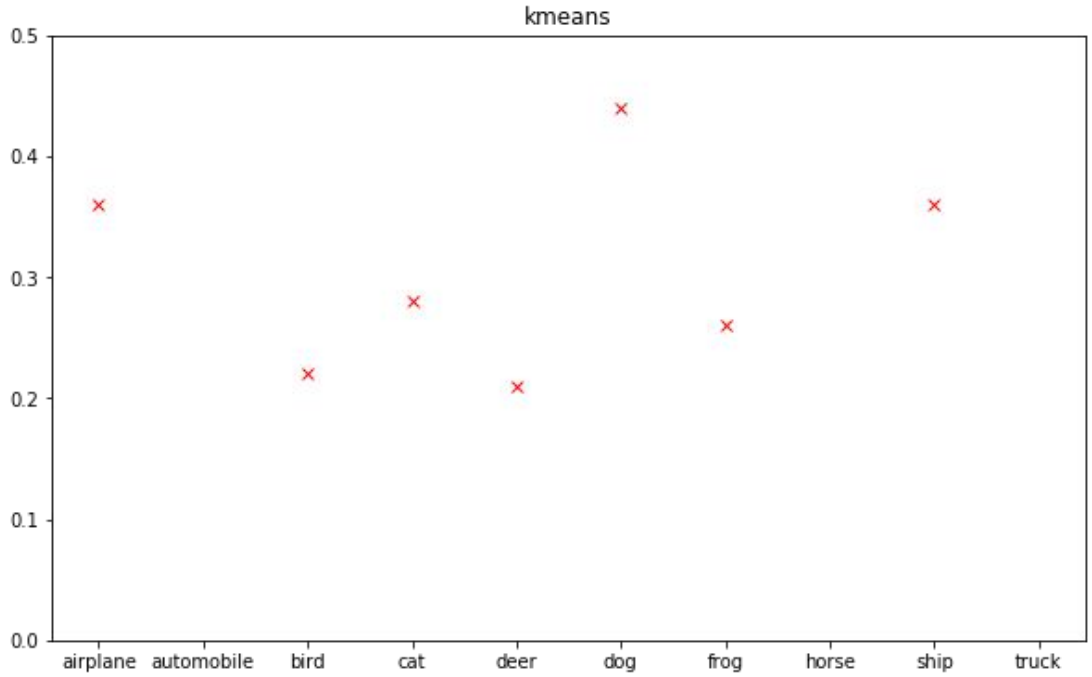

Figure 3.2.1

Pour la deuxième façons, kmeans2, nous avons juste calculé le valeur moyen de chaque classe comme les centres représentant chaque classe, et puis nous avons classifié les nouvelles images par calculer la minimum distance entre les image et les centres nous avons calculé. A la fin, Nous avons obtenu un taux de correction de 27.7%.

4. Algorithme selon l'article “An Analysis of Single-Layer Networks in Unsupervised Feature Learning”

4.1 Le but de l'article:

L'article montre que plusieurs facteurs simples, tels que le nombre de nœuds cachés dans le modèle, peuvent être plus importants pour atteindre des performances élevées que l'algorithme d'apprentissage ou la profondeur du modèle. Plus précisément, ils appliqueront plusieurs algorithmes d'apprentissage de fonctionnalités prêts à l'emploi (clustering k-moyennes...) à CIFAR-10, utilisant uniquement des réseaux à une seule couche. Il présente ensuite une analyse détaillée de l'effet des changements dans la configuration du modèle: la taille du champ réceptif, le nombre de nœuds cachés (caractéristiques), la taille de pas entre les caractéristiques extraites et l'effet de blanchiment. Les résultats montrent qu'un grand nombre de nœuds cachés et l'extraction de caractéristiques denses sont essentiels à la réalisation de hautes performances, ce qui est si important que, lorsque ces paramètres sont poussés à leurs limites, nous atteignons des performances de pointe sur les deux plans CIFAR-10 en utilisant seulement une seule couche de fonctionnalités. Plus étonnant, la meilleure performance est basée sur le clustering k-moyennes, qui est extrêmement rapide, et est très facile à mettre en œuvre.

4.2 Idée principale :

1. Nous utilisons l'algorithme kmeans pour la compression d'image et l'extraction de caractéristiques, l'approche spécifique est: Nous divisons chaque image en quatre patchs, et nous combinons toutes les patchs pour former un nouvel ensemble de données. Après cela, nous utilisons l'algorithme kmeans pour calculer les N centres, et nous utilisons les N centres pour

construire un dictionnaire de longueur N. Ensuite, nous créons les vecteurs de longueur N pour tous les patchs, qui sont initialisé à 0. Après cela, nous calculons les distances entre les patchs et les N centres. Pour un patch, si le k-ème centre est le plus proche, alors on met le k-ème position du vecteur de ce patch à 1. Comme ça, nous pouvons représenter chaque patch avec un vecteur N-dimensionnel, où une seule position est 1. Chaque image est composée de quatre patchs, donc nous pouvons représenter chaque image avec un vecteur 4*N-dimensionnel. De cette façon, la longueur de chaque image est plus petite, et l'algorithme de classification peut être plus vite. Les caractéristiques de chaque image seront également plus apparentes.

2. Nous blanchissons les données selon l'article. Le but du blanchiment(whitening) est de supprimer les informations redondantes des données d'entrée. Les données d'apprentissage sont images, puisqu'il existe une forte corrélation entre les pixels adjacents dans l'image, l'entrée pour l'apprentissage est redondante; le but du blanchiment(whitening) est de réduire la redondance de l'entrée. Après que l'ensemble de données d'entrée X est blanchi, les nouvelles données X rencontrent deux propriétés: Faible corrélation entre les caractéristiques; Toutes les caractéristiques ont la même variance.

3. Nous appliquons des données pré-traitées à différents algorithmes de classification, puis nous comparons les résultats avec les résultats de l'étape précédente(Naive approach).

4.3 Diviser les images

```
#################divisez l'image en 4 patches
def quatre_patchs(image):
    ens = []
    new_ens = []
    for i in image:            #pour chaque images dans batchs
        rouge = i[0:1024]
        vert  = i[1024:2048]
        bleu  = i[2048:3072]
        pixel = np.dstack((rouge,vert))
        pixel = np.dstack((pixel,bleu))
        pixel_n = pixel[0]
        pixel_n = np.reshape(pixel_n,(32,32,3))

        ens.append(pixel_n)
    for e in ens:
            new_ens.append(e[:16,:16])    #en haut à gauche 16 * 16
            new_ens.append(e[:16,16:])    #en haut à droite
            new_ens.append(e[16:,:16])    #en bas à gauche
            new_ens.append(e[16:,16:])    #en bas à droite
    #return (len(new_ens))
    return array(new_ens)
```

Figure 4.3

La fonction **quatre_patchs** ayant un paramètre “image” qui contient des valeurs des pixels des images :

def quatre_patchs(image):

D’abord, nous vous présentons la structure de données :

1 image : 3072 valeurs pour représenter des pixels

3072 pixels : 1024 pixels * 3 couleurs

Donc nous avons pris des valeurs correspondantes aux 3 couleurs de pixels :

rouge = i[0:1024]

vert = i[1024:2048]

bleu = i[2048:3072]

Nous avons utilisé une fonction **numpy.dstack()** qui permet de mettre des valeurs des même colonnes :

pixel = np.dstack((rouge,vert))

pixel = np.dstack((pixel,bleu))

pixel_n = pixel[0]

pixel_n = np.reshape(pixel_n,(32,32,3))

Après on ajoute ‘pixel_n’ dans la liste ens. Dans le prochain ‘for’ cycle, nous divisons chaque donnée en quatre patchs. Nous appliquons cette fonction à toutes les données, nous obtenons 200 000 patchs pour les ensembles d'entraînement ,et 40 000 patchs pour les ensembles de test.

4.4 Blanchir les données

```
#whitening
print "whitening the train patches... "
[D,V]=np.linalg.eig(np.cov(x_train,rowvar=0))

P = V.dot(np.diag(np.sqrt(1/(D + 0.1)))).dot(V.T)
x_train = x_train.dot(P)
[D,V]=np.linalg.eig(np.cov(x_test,rowvar=0))

P = V.dot(np.diag(np.sqrt(1/(D + 0.1)))).dot(V.T)
x_test = x_test.dot(P)
print "end of whitening"
```

Figure 4.4

[D,V]=np.linalg.eig(np.cov(x_train,rowvar=0))

P = V.dot(np.diag(np.sqrt(1/(D + 0.1)))).dot(V.T)

x_train = x_train.dot(P)

numpy.cov() est utilisé pour estimer une matrice de covariance, données et poids donnés. **numpy.linalg.eig(*)** calcule les valeurs propres et les vecteurs propres droits d'un tableau carré. D est la valeur caractéristique, et V est le vecteur de caractéristiques. Après nous calculons la matrice de blanchiment ‘PCA’ P, et blanchissons le x_train avec P. Ensuite, nous faisons la même chose pour x_test.

4.5 Créer un dictionnaire en utilisant la méthode kmeans

```
# Kmeans calcule les centres
from sklearn.cluster import KMeans
K=300
print "Start creating the dictionary"
print "Start using the sklearn Kmeans...."
result = KMeans(n_clusters=K,max_iter=30).fit(x_train[0:120000])
print "K-means has done..."
```

Figure 4.5

Nous utilisons la méthode kmeans de sklearn bibliothèque pour calculer les centres
de x_train, et les construisons comme un dictionnaire. K est la longueur du dictionnaire, nous pouvons la changer pour obtenir les différents dictionnaires. cette étape prend généralement beaucoup de temps.
Ensuite, on peut obtenir les centres en utilisant **"center=result.cluster_centers_"** et la forme du centre est comme (K, 768), 768 est la longueur du chaque patch.

4.6 Créer nouveaux vecteurs pour les patchs

```
# changer les formes des donnes
res=np.zeros((200000,K))
for i in range(len(x_train)):
    s=np.linalg.norm(x_train[i]-center[0])
    index=0
    for j in range(len(center)):
        if np.linalg.norm(x_train[i]-center[j])<s:
            s=np.linalg.norm(x_train[i]-center[j])
            index=j
    res[i][index]=1
res_t=np.zeros((40000,K))
for i in range(len(x_test)):
    s=np.linalg.norm(x_test[i]-center[0])
    index=0
    for j in range(len(center)):
        if np.linalg.norm(x_test[i]-center[j])<s:
            s=np.linalg.norm(x_test[i]-center[j])
            index=j
    res_t[i][index]=1
```

Figure 4.6

Tout d'abord, nous créons 200 000 tableaux de longueur K pour tous les patchs dans x_train, tous remplis de 0. Après cela, nous calculons quel centre est le plus proche de chaque patch. Si ce patch est la plus proche du n-ième centre, alors on met le n-ième position de ce patch à 1. "**numpy.linalg.norm( )**"

est capable de renvoyer le distance entre deux vecteurs(la valeur par défaut est la distance euclidienne). Enfin, nous faisons la même chose pour l'ensemble de test.
Après cela, nous connectons les quatre patchs de chaque image, de cette manière, chaque image peut être représentée par un vecteur de longueur 4*K, où quatre positions sont 1.

4.7 Appliquer des données à différents algorithmes de classification

```
from sklearn.svm import SVC
#train_set=train_final[0:5000]
#train_label=y_train[0:5000]
#test_set=test_final[0:1000]
#test_label=y_test[0:1000]
train=list()
test=list()
clf = SVC()
clf.fit(train_set,train_label)
train.append(np.mean(clf.predict(train_set) == train_label))
test.append(np.mean(clf.predict(test_set) == test_label))

print("svc:" ,train,test)
```

```
('svc:', [0.28010000000000002], [0.28699999999999998])
```

Figure 4.7.1

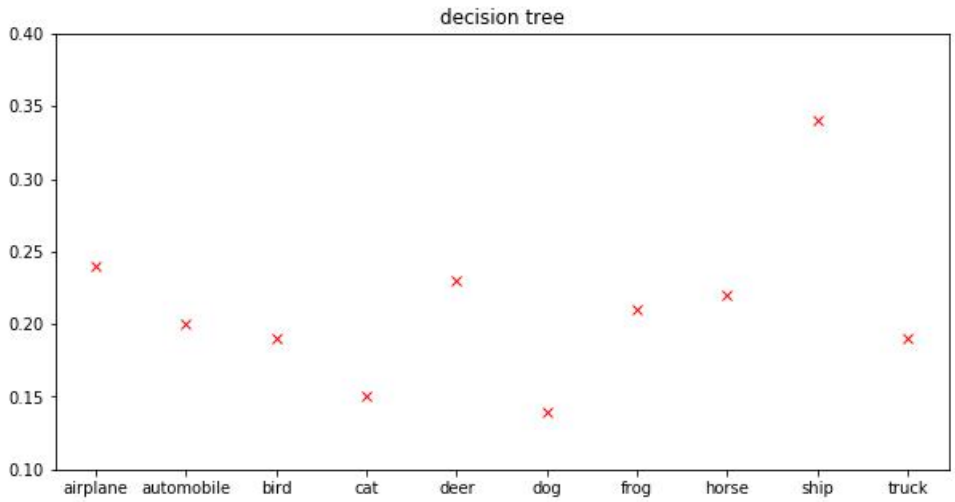

Figure 4.7.2

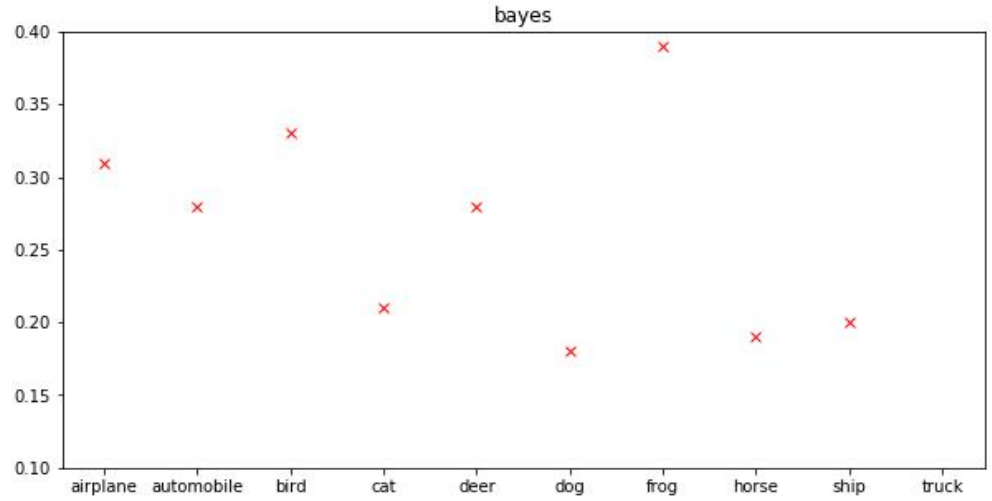

Figure 4.7.3

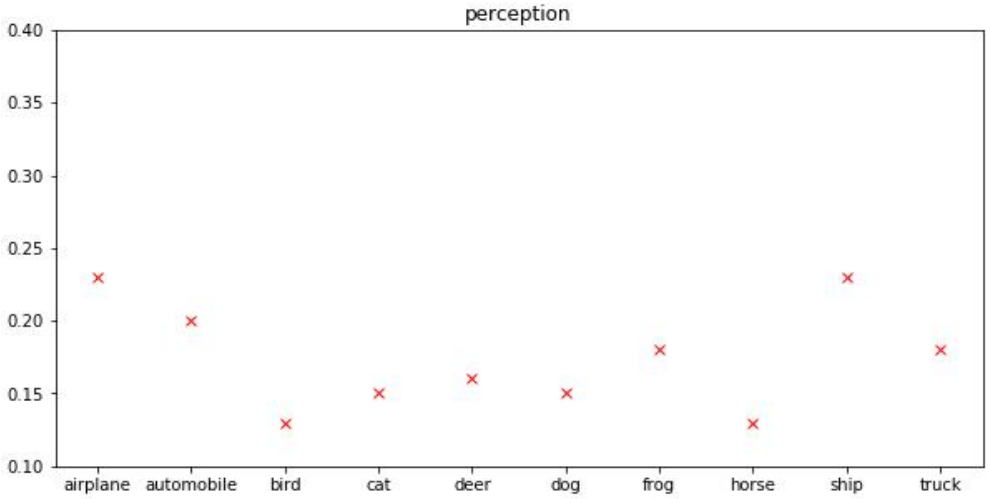

Figure 4.7.4

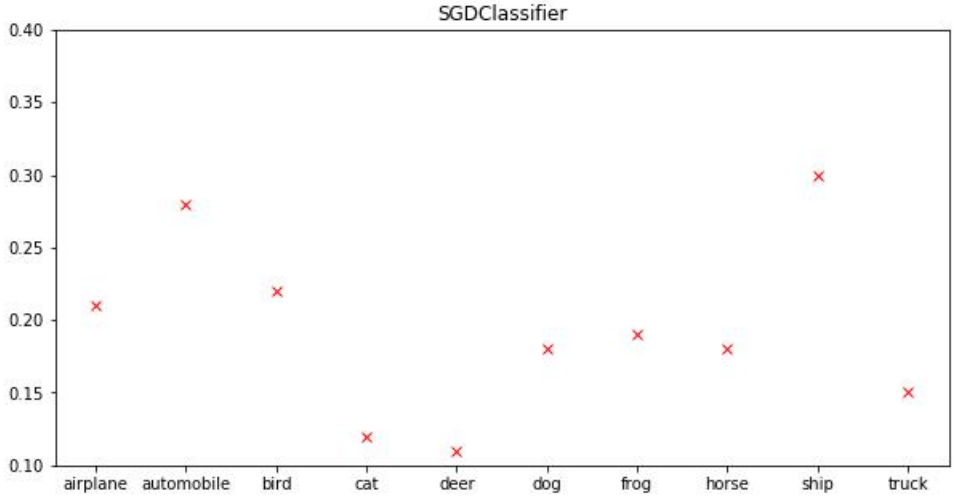

Figure 4.7.5

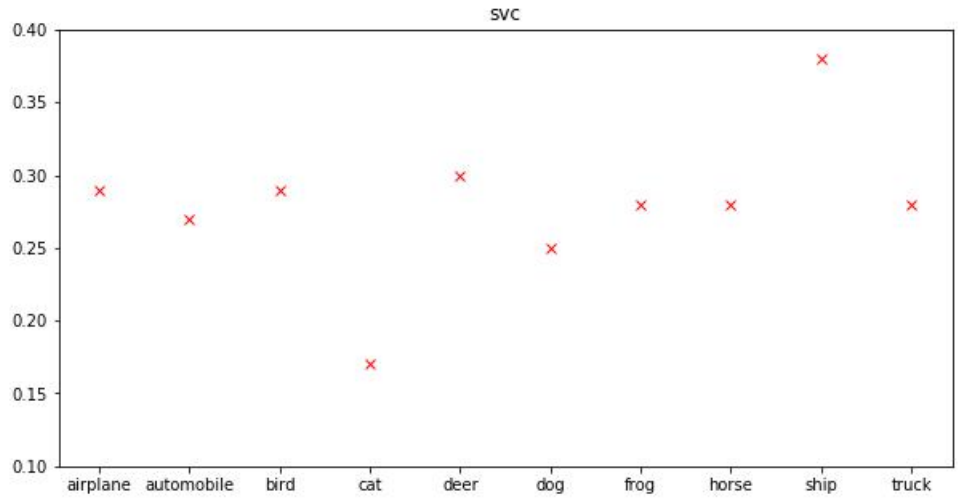

Figure 4.7.6

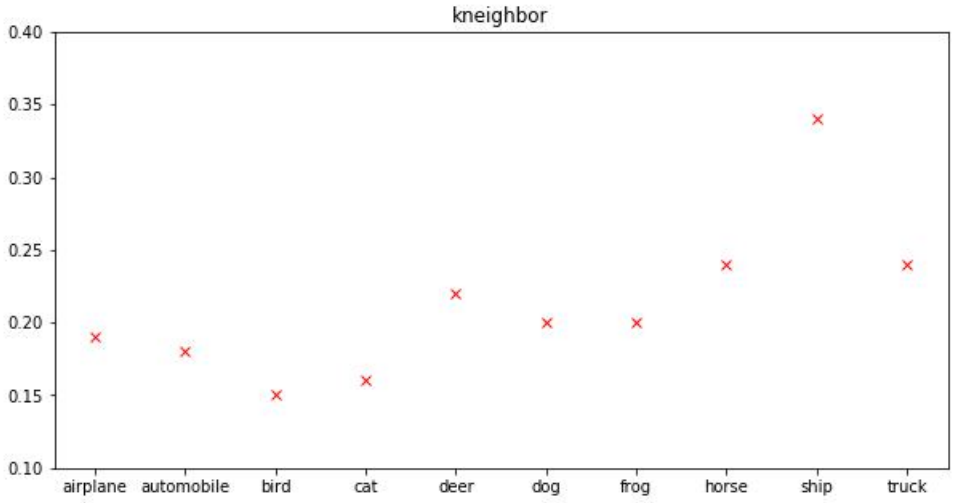

Figure 4.7.7

Nous avons utilisé un total de six algorithmes de classification, comme nous l'avons fait à l'étape précédente. Nous utilisons les fonctions de sklearn bibliothèque aussi. Nous comparons les résultats(ici la longueur du dictionnaire K est 60):

Résultat du test(taux correct)

	train_set (naive)	**test_set (naive)**	**train_set (complexe)**	**test_set (complexe)**
Decision Tree	1.0	0.245	0.598	0.216

Bayes	0.291	0.300	0.215	0.211
KNeighbors	0.380	0.302	0.391	0.202
SVM	1.0	0.106	0.280	0.287
Perceptron	0.340	0.290	0.170	0.171
SGD	0.294	0.251	0.209	0.206

4.8 Analyse des résultats

Grâce au tableau ci-dessus, nous pouvons voir que pour la plupart des algorithmes de classification, le prétraitement des données n'améliore pas seulement les performances de l'algorithme, mais réduit également le taux correct. Cependant, la performance de l'algorithme SVM a été améliorée, dans l'algorithme SVM naive, le taux correct sur test_set est seulement 0.106, mais maintenant il arrive à 0.287 sur test_set. Dans l'article, ils utilisent également le classificateur SVM, nous pouvons donc conclure que le prétraitement des données avec l'algorithme kmeans peut améliorer les performances de l'algorithme SVM.

Le taux final obtenu dans l'article est d'environ 0.7, mais dans notre expérience, le taux final obtenu est d'environ 0.3, nous pensons qu'il y a deux raisons: Dans l'article, ils utilisent des fenêtres coulissantes(slides) pour sélectionner les patchs des images, mais nous avons seulement divisé l'image en quatre parties par position, peut-être il ne suffit pas d'apprendre suffisamment de caractéristiques. Une autre raison possible est que nous n'avons pas choisi de très bons paramètres, par exemple, Il y a beaucoup d'options pour la longueur du dictionnaire K, nous avons essayé quelques chiffres, et parmi lesquels 60 est mieux, mais nous ne pouvons pas essayer beaucoup de chiffres, parce que le programme est lancé trop lentement sur ma machine. De plus, nous pouvons aussi modifier certains des paramètres de l'algorithme SVM, ou augmenter le nombre de train_set, etc. Dans ces cas, nous pouvons obtenir de meilleurs résultats.

5. Conclusion

5.1 Résultats expérimentaux

Nous avons fait les algorithmes “naive” travaillant directement sur l'ensemble des pixels d'une image, des valeurs calculées peuvent représenter des caractéristiques, mais elle ne sont pas assez précises pour représenter des caractéristiques des images,
Et les données sont un peu grandes. Parmi les algorithmes “naive”, l'algorithme bayes obtient les meilleurs résultats, et son taux correct est d'environ 0.3. Nous avons également modifié l'algorithme des kmeans, pour l'appliquer à l'apprentissage supervisé, et nous avons eu un taux correct d'environ 0.25.
Dans la deuxième partie de l'algorithme, nous avons divisé une images en quatre patchs, ensuite, nous traitons les patchs et les utilisons dans différents algorithmes de classification. Et pour l'algorithme SVM, nous avons obtenu un taux de correction mieux que précédente.

5.2 Complexité de l'algorithme

À propos de la complexité des algorithme, quand nous lançons les premier algorithmes, le temps d'exécutions est moyen, mais le second est très lent, nous pouvons bien savoir que le premier algorithme est beaucoup moins complexe que le second. Le taux correct des deux algorithmes est presque la même pour l'instant, donc, globalement, les premiers algorithmes(naive) sont meilleurs actuellement.

5.3 Futurs travaux

Nous espérons utiliser des algorithmes plus avancés pour classer les images dans le futur, par exemple, MLP (Multi Layer Perceptron), CNN(Convolutional Neural Network), etc. Nous espérons apprendre des méthodes plus scientifiques pour choisir les paramètres dans le système d'apprentissage, au lieu d'essayer aveuglément. Nous espérons également appliquer ces algorithmes que nous avons appris à d'autres tâches d'apprentissage dans le futur.

5.4 Récolte expérimentale

Grâce à ce projet, nous avons appris différents algorithmes d'apprentissage, et nous avons su comment utiliser des différentes façons pour classer des images. Nous avons étudié l'article très sérieusement, et nous avons appris beaucoup d'idées intéressantes de l'article. Nous sommes également plus familiers avec le langage de programmation python et certaines bibliothèques d'apprentissage en python, cela sera utile pour notre étude et notre travail futurs. Enfin, merci pour vos conseils et votre aide.