

2^{ème} année cycle supérieur SIT(2CS-SIT1)

TP BDM

SVM non linéaire

Réalisé par :



 ADLA Ilyes chiheb eddine
 IASSAMEN Bilal

Table des matières

• SVM non linéaire	3
• Fonction cout dans ce cas	3
• Algorithme de la descente du gradient.....	4
• Prédire la sortie de la donnée.....	5

SVM non linéaire

SVM : est une techniques d'apprentissage supervisé destinées à résoudre des problèmes de discrimination et de régression.

SVM non lineaire est utilisé lors de l'inexistence d'un hyperplan capable de séparer correctement les données , donc on peut dire que ces données sont non linéairement séparables.

il faut transposer les données dans un espace de plus grande dimension pour pouvoir trouver un hyperplan séparateur

Fonction cout dans ce cas

Voici la formule de la fonction cout

$$h(x) = \omega^* \varphi(x) + b = \sum_{i=1}^n \lambda_i y_i < \varphi(x_i), \varphi(x) > + b = \sum_{i=1}^n \lambda_i y_i k(x_i, x) + b$$

Résoudre :

$$\begin{cases} \max_{\lambda} \sum_{i=1}^n \lambda_i - \frac{1}{2} \sum_{i=1}^n \lambda_i \lambda_j y_i y_j k(x_i, x_j) \\ \sum_{i=1}^n \lambda_i y_i = 0 \\ \lambda_i > 0 \end{cases}$$

Algorithme de la descente du gradient

Dans ce TP nous examinons les machines à vecteurs de support dans leur version non linéaire , dans notre cas on a pris le dataset iris.

```
def calculer_cost(W, X, Y):
    N = X.shape[0]
    distances = 1 - Y * (np.dot(X, W))
    distances[distances < 0] = 0
    hinge_loss = reg_strength * (np.sum(distances) / N)

    cost = 1 / 2 * np.dot(W, W) + hinge_loss
    return cost
```

```
def calculer_loss_gradient(W, X_batch, Y_batch):
    if type(Y_batch) == np.float64:
        Y_batch = np.array([Y_batch])
        X_batch = np.array([X_batch])
    distance = 1 - (Y_batch * np.dot(X_batch, W))
    dw = np.zeros(len(W))
    for ind, d in enumerate(distance):
        if max(0, d) == 0:
            di = W
        else:
            di = W - (reg_strength * Y_batch[ind] * X_batch[ind])
        dw += di
    dw = dw/len(Y_batch)
    return dw
```

```
def sgd(features, outputs):
    max_epochs = 5000
    weights = np.zeros(features.shape[1])
    nth = 0
    prev_cost = float("inf")
    cost_threshold = 0.01
    for epoch in range(1, max_epochs):
        X, Y = shuffle(features, outputs)
        for ind, x in enumerate(X):
            ascent = calculer_loss_gradient(weights, x, Y[ind])
            weights = weights - (learning_rate * ascent)
        if epoch == 2 ** nth or epoch == max_epochs - 1:
            cost = calculer_cost(weights, features, outputs)
            if abs(prev_cost - cost) < cost_threshold * prev_cost:
                return weights
            prev_cost = cost
            nth += 1
    return weights
```

Prédire la sortie de la donnée

- Importer les modules

```
from sklearn import datasets
```

```
from sklearn.svm import SVC
```

- Charger le dataset iris

```
iris=datasets.load_iris()
```

- Effectuer l'apprentissage

```
X = iris.data  
y = iris.target
```

```
clf3 = SVC(gamma=.1, kernel='rbf', probability=True)
```

```
clf3.fit(X,y)
```

- Prédire la sortie de la donnée [5.4,2.3,2.2,7.3]

```
clf3.predict([[5.4,2.3,2.2,7.3]])
```

```
array([2])
```

- La sortie de la donnée est la catégorie 2 : virginica