《机器学习》课程学习报告（一）

学号：16281012 姓名：聂小禹 日期：20181031

1 学习目的

在后半部分我们要进行对于支持向量机、聚类、贝叶斯分类器，如 GaussianNB, BernouliNB, MultinomialNB 等类进行原理上的学习，利用 sklearn 中的自带库完成对不同数据集中的处理，模型的建立，并进行数据的预测。 一个支持向量机的构造一个超平面，或在高或无限维空间，其可以用于分类，回归，或其它任务中设定的超平面的。直观地，一个良好的分离通过具有到任何类的最接近的训练数据点的最大距离的超平面的一般实现中，由于较大的裕度下分类器的泛化误差。可能在一个有限维空间中所述，经常发生以鉴别集是不是在该空间线性可分。因此， 在原始有限维空间映射到一个高得多的立体空间，推测使分离在空间比较容易。保持计算负荷合理，使用支持向量机计划的映射被设计成确保在点积可在原空间中的变量而言容易地计算，通过定义它们中选择的核函数 k（x，y）的计算以适应的问题。贝叶斯分类器是各种分类器中分类错误概率最小或者在预先给定代价的情况下平均风险最小的分类器。它的设计方法是一种最基本的统计分类方法。其分类原理是通过某对象的先验概率，利用贝叶斯公式计算出其后验概率，即该对象属于某一类的概率，选择具有最大后验概率的类作为该对象所属的类。聚类在生活中使用更加常见，它可以将数据主动地分类，我们将学习它选择类中心和不断调整的类中心和数据划分的原理和方法。

2 学习内容

一、 支持向量机

实验内容:

1. 了解核函数对 SVM 的影响
2. 绘制不同核函数的决策函数图像
3. 简述引入核函数的目的，其中核函数包括高斯核，sigmoid 核，多项式核。
4. 了解分离超平面、间隔超平面与支持向量的绘制
5. 调整 C 的值，绘制分离超平面、间隔超平面和支持向量
6. 简述引入软间隔的原因，以及 C 值对 SVM 的影响
7. 使用支持向量机完成 spambase 垃圾邮件分类任务
8. 使用训练集训练模型，计算测试集的精度，查准率，查全率，F1 值
9. 使用支持向量机完成 kaggle 房价预测问题
10. 使用训练集训练模型，计算测试集的 MAE 和 RMSE

二、 贝叶斯分类器

实验内容：

1. 使用 GaussianNB 完成 spambase 邮件分类
2. 计算十折交叉验证的精度、查准率、查全率、F1 值
3. 使用 BernoulliNB 完成 spambase 邮件分类
4. 计算十折交叉验证的精度、查准率、查全率、F1 值
5. 使用 MultinomialNB 完成 spambase 邮件分类
6. 计算十折交叉验证的精度、查准率、查全率、F1 值
7. 实现高斯朴素贝叶斯分类器
8. 计算模型的查准率，查全率，F1 值

三、 聚类

实验内容：

1. 使用 sklearn 的 DBSCAN 和 GaussianMixture 在两个数据集上完成聚类任务
2. 对聚类结果可视化
3. 对比外部指标 FMI 和 NMI
4. 选做：调整密度聚类的 eps 参数，绘制聚类结果
5. 使用 sklearn 的 Kmeans 完成两个数据集的聚类任务
6. 计算外部指标 FMI 和 NMI
7. 对聚类结果可视化
8. 使用 sklearn 的 AgglomerativeClustering 完成两个数据集的层次聚类
9. 计算外部指标 FMI 和 NMI
10. 对 6 种形式的聚类效果可视化
11. 实现一个 K-means 聚类算法
12. 计算外部指标 FMI 和 NMI
13. 对聚类结果可视化
14. 完成第二个数据集上 myKmeans 与层次聚类(single)算法的对比

3 学习过程

一、支持向量机

第一题

产生双半月形数据，随机筛选一部分数据作为测试集。

使用不同的核函数寻找支持向量并超出超平面。

线性核函数

plot_model(SVC(kernel = "linear", probability = True), 'Linear SVM')

高斯核函数

plot_model(SVC(kernel = "rbf", probability = True), 'Radial Basis Function')

sigmoid 核函数

plot_model(SVC(kernel = "sigmoid", probability = True), 'Sigmoid')

多项式核函数

plot_model(SVC(kernel = "poly", probability = True), 'Poly')

可视化

4. 样本到分离超平面距离的可视化

我们使用SVM模型里面的decision_function方法，可以获得样本到分离超平面的距离。

下面的函数将图的背景变成数据点，计算每个数据点到分离超平面的距离，映射到不同深浅的颜色上，绘制出了不同颜色的背景。

```
def plot_model(model, title):

    # 训练模型，计算精度
    model.fit(X_train, y_train)
    score_train = model.score(X_train, y_train)
    score_test = model.score(X_test, y_test)

    # 将背景网格化
    h = 0.02
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # 计算每个点到分离超平面的距离
    Z = model.decision_function(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    # 设置图的大小
    plt.figure(figsize = (14, 6))

    # 绘制训练集的子图
    plt.subplot(121)

    # 绘制决策边界
    plt.contourf(xx, yy, Z, cmap = cm, alpha=.8)

    # 绘制训练集的样本
    plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright, edgecolors='k')
```

```
# 设置图的上下左右界
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())

# 设置子图标题
plt.title("training set")

# 图的右下角写出在当前数据集中的精度
plt.text(xx.max() - .3, yy.min() + .3, ('acc: %.3f' % score_train).lstrip('0'), size=15, hor

plt.subplot(122)
plt.contourf(xx, yy, Z, cmap = cm, alpha=.8)

plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, edgecolors='k', alpha=0.6)

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())

plt.title("testing set")

plt.text(xx.max() - .3, yy.min() + .3, ('acc: %.3f' % score_test).lstrip('0'), size=15, hori

plt.suptitle(title)
```

为什么要引入核函数？

当我们在解决线性不可分的问题时，我们需要通过一个映射函数，把样本值映射到更高维的空间或者无穷维。在特征空间中，我们对线性可分的新样本使用前面提到过的求解线性可分的情况下的分类问题的方法时，需要计算样本内积，但是因为样本维数很高，容易造成“维数灾难”，所以这里我们就引入了核函数，把高维向量的内积转变成了求低维向量的内积问题。

第二题

4. 样本到分离超平面距离的可视化

我们使用SVM模型里面的decision_function方法，可以获得样本到分离超平面的距离。

下面的函数将图的背景变成数据点，计算每个数据点到分离超平面的距离，映射到不同深浅的颜色上，绘制出了不同颜色的背景。

```
def plot_model(model, title):

    # 训练模型，计算精度
    model.fit(X_train, y_train)
    score_train = model.score(X_train, y_train)
    score_test = model.score(X_test, y_test)

    # 将背景网格化
    h = 0.02
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # 计算每个点到分离超平面的距离
    Z = model.decision_function(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    # 设置图的大小
    plt.figure(figsize = (14, 6))

    # 绘制训练集的子图
    plt.subplot(121)

    # 绘制决策边界
    plt.contourf(xx, yy, Z, cmap = cm, alpha=.8)

    # 绘制训练集的样本
    plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright, edgecolors='k')
```

```
  # 设置图的上下左右界
  plt.xlim(xx.min(), xx.max())
  plt.ylim(yy.min(), yy.max())

  # 设置子图标题
  plt.title("training set")

  # 图的右下角写出在当前数据集中的精度
  plt.text(xx.max() - .3, yy.min() + .3, ('acc: %.3f' % score_train).lstrip('0'), size=15, hor

  plt.subplot(122)
  plt.contourf(xx, yy, Z, cmap = cm, alpha=.8)

  plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, edgecolors='k', alpha=0.6)

  plt.xlim(xx.min(), xx.max())
  plt.ylim(yy.min(), yy.max())

  plt.title("testing set")

  plt.text(xx.max() - .3, yy.min() + .3, ('acc: %.3f' % score_test).lstrip('0'), size=15, hori

  plt.suptitle(title)
```

5. 绘制间隔

根据SVM的原理，间隔超平面的方程是：

$$w^{\mathrm{T}}x + b = \pm 1$$

我们先讨论右侧为1的情况：

$$w^{\mathrm{T}}x + b = 1$$

写成分量形式：

$$w_0 x_1 + w_1 x_2 + b = 1$$

根据上面第四节的变换，可以得到：

$$x2 = \frac{1}{w_1} - \frac{w_0}{w_1}x_1 - \frac{b}{w_1}$$

同理，当右侧为-1时，可得：

$$x2 = -\frac{1}{w_1} - \frac{w_0}{w_1}x_1 - \frac{b}{w_1}$$

可以发现，间隔超平面的方程就是在分离超平面上增加或减去$\frac{1}{w_1}$

```
def compute_margin(model, x1):
    '''
    计算二维平面上的间隔超平面，
    我们通过w0，w1，b以及x1计算出x2，只不过这里的x1是一个ndarray，x2也是一个ndarray

    Parameters
    ----------
    model: sklearn中svm的模型

    x1: numpy.ndarray，如[-5, 5]，表示超平面上这些点的横坐标

    Returns
    ----------
    x2_up: numpy.ndarray, 一条间隔超平面上对应的纵坐标

    x2_down: numpy.ndarray, 另一条间隔超平面上对应的纵坐标

    '''

    # 先调用compute_hyperplane计算超平面的纵坐标
    x2 = compute_hyperplane(model, x1)
    w0 = model.coef_[0][0]
    w1 = model.coef_[0][1]
    b = model.intercept_[0]
    # YOUR CODE HERE
    x2_up = 1/w1-(w0/w1)*x1-b/w1

    # YOUR CODE HERE
    x2_down = -1/w1-(w0/w1)*x1-b/w1

    return x2_up, x2_down
```

6. 标出支持向量

模型的support_vectors_属性包含了支持向量

```
# 绘制数据
plot_data(X, Y)

# 在横坐标上选两个点
x1 = np.array([-5, 5])

# 计算超平面上这两个点的对应纵坐标
x2 = compute_hyperplane(clf, x1)

# 计算间隔超平面上这两个点的对应纵坐标
x2_up, x2_down = compute_margin(clf, x1)

# 绘制分离超平面和间隔超平面
plt.plot(x1, x2, '-', color = 'red')
plt.plot(x1, x2_up, 'k--')
plt.plot(x1, x2_down, 'k--')

# 标出支持向量
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s = 200,
            facecolors = 'none', edgecolors = 'red')
```

以上给出了本道题所需要的基本代码，我们接下来要更改 C 值，并修改利用不同的核函数来进行 svm 的实现数据训练和预测。
我们分别观察 C＝10，C=1,C=0.1 的情况。

C＝10

```
# 创建模型
clf1 = SVC(kernel = 'linear', C = 10, random_state = 32)

# 训练模型
clf1.fit(X, Y)
```

C＝1

```
# 创建模型
clf2 = SVC(kernel = 'linear', C = 1, random_state = 32)

# 训练模型
clf2.fit(X, Y)
# 创建模型

clf3 = SVC(kernel = 'linear', C = 0.1, random_state = 32)

# 训练模型
clf3.fit(X, Y)
```

第三题

使用 rbf，linear，sigmoid 三种核函数，看 svc 对于 Spambase 数据集的分类情况。
并计算四种指标。

第四题
使用 svr 完成 kaggle 房价预测，使用 lotarea，bsmtunfsf，garagearea 作为特征值。
当使用线性核函数的时候，使用 linearsvr 类，不需要设置参数
引入对应的包，处理分割数据，用 model 保存 svr 模型，prediction 保存相应 svr 模型所得出来的预测值。

二 贝叶斯分类器

第一题

导入模型

```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
```

十折验证下的 Gaussian 分类

```
model = GaussianNB()
prediction=cross_val_predict(model,spamx,spamy,cv=10)
prediction.shape

print(accuracy_score(spamy,prediction))
print(precision_score(spamy,prediction))
print(recall_score(spamy,prediction))
print(f1_score(spamy,prediction))
```

第二题
引入模型

```
from sklearn.naive_bayes import BernoulliNB
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
```

bernouli 的贝叶斯朴素分类器需要所有数据都符合 bernouli 分布
将所有的数据都转换成二值型
spamx_binary = (spamx != 0).astype('float64')
如果不为 0，就为 1

```
spamx_binary
```

```
array([[0., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 0., 1., ..., 1., 1., 1.],
       ...,
       [1., 0., 1., ..., 1., 1., 1.],
       [1., 0., 0., ..., 1., 1., 1.],
       [0., 0., 1., ..., 1., 1., 1.]])
```

```
model = BernoulliNB()
prediction=cross_val_predict(model,spamx_binary,spamy,cv=10)
prediction.shape

print(accuracy_score(spamy,prediction))
print(precision_score(spamy,prediction))
print(recall_score(spamy,prediction))
print(f1_score(spamy,prediction))
```

第三题
引入模型

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
```

数据预处理
这里仍然要将数据换成 0，1

```
model = MultinomialNB()
prediction=cross_val_predict(model,spamx_binary,spamy,cv=10)
prediction.shape

print(accuracy_score(spamy,prediction))
print(precision_score(spamy,prediction))
print(recall_score(spamy,prediction))
print(f1_score(spamy,prediction))
```

第四题

我们要实现一个可以处理连续特征的，服从高斯分布的朴素贝叶斯分类器。

符号

给定训练集 T

$$T = \{(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)\}$$

其中，x 为样本的特征，y 是该样本对应的标记，下标表示对应的是第几个样本，上标表示第几个特征。训练集 T 内一共 $|T| = N$ 个样本。

假设我们的任务是处理 K 类分类任务，记类标记分别为 $c_1, c_2, \ldots, c_k$ 。

目标

我们的目标是对样本进行分类，这里我们用概率的方法，求 $P(Y = c_k \mid X = x),\ k = 1, 2, \ldots, K$ 中最大的那个概率对应的 k 是哪个，也就是，给定样本 x ，模型认为它是哪个类别的概率最大。

原理

由贝叶斯公式：

$$\begin{aligned} P(Y = c_k \mid X = x) &= \frac{P(Y = c_k, X = x)}{P(X = x)} \\ &= \frac{P(X = x \mid Y = c_k)P(Y = c_k)}{\sum_k P(X = x \mid Y = c_k)P(Y = c_k)} \end{aligned}$$

这里，我们要求 K 个概率中最大的那个，而这 K 个概率的分母都相同，我们可以忽略分母部分，比较分子部分的大小，也就是比较 **先验概率** $P(Y = c_k)$ 和 **似然** $P(X = x \mid Y = c_k)$ 的乘积。

通过先验概率分布

$$P(Y = c_k),\ k = 1, 2, \ldots, K$$

和条件概率分布

$$P(X = x \mid Y = c_k) = P(X^{(1)} = x^{(1)}, \cdots, X^{(n)} = x^{(n)} \mid Y = c_k),\ k = 1, 2, \ldots, K$$

我们就可以得到联合概率分布 $P(X = x, Y = c_k)$ 。

那么，问题就转化为了，如何求先验概率和似然？

1. 先验概率 $P(Y = c_k)$ ：

先验概率的求解很简单，只要统计训练集中类别 k 出现的概率即可。

$$P(Y = c_k) = \frac{\text{number of } c_k}{N}$$

2. 似然 $P(X = x \mid Y = c_k)$ ：

求解这个条件概率比较复杂，**这里我们要假设特征之间相互独立**，可得

$$P(X = x \mid Y = c_k) = \prod_{j=1}^{n} P(X^{(j)} = x^{(j)} \mid Y = c_k)$$

其中， $x^{(j)}$ 表示样本 x 的第 j 个特征。

这样，复杂的条件概率就转换为了多个特征条件概率的乘积。

3. 特征 j 的条件概率 $P(X^{(j)} = x^{(j)} \mid Y = c_k)$ ：

因为我们处理的特征都是连续型特征，一般我们假设这些特征服从正态分布。

当 $Y = c_k$ 时，$X^{(j)} = a_{jl}$ 的概率可由下面的公式计算得到：

$$P(X^{(j)} = a_{jl} \mid Y = c_k) = \frac{1}{\sqrt{2\pi\sigma_{c_k j}^2}} \exp\left(- \frac{(a_{jl} - \mu_{c_k j})^2}{2\sigma_{c_k j}^2} \right)$$

这里 $\mu_{c_k j}$ 和 $\sigma_{c_k j}^2$ 分别表示当 $Y = c_k$ 时，第 j 个特征的均值和方差，**这个均值和方差都是通过训练集的样本计算出来的**。

因为正态分布只需要两个参数（均值和方差）就可以确定，对于特征 j 我们要估计 K 个类别的均值和方差，所以特征 j 的参数共有 $2K$个。

综上

朴素贝叶斯分类器可以表示为：

$$y = \arg\max_{c_k} P(Y = c_k) \prod_j P(X^{(j)} = x^{(j)} \mid Y = c_k)$$

实现 ¶

实现的时候会遇到数值问题，在上面的条件概率连乘中，如果有几个概率值很小，它们的连乘就会导致下溢，解决方案就是将其改写为连加的形式。

首先，我们的目标是：

$$y = \arg\max_{c_k} P(Y = c_k) \prod_j P(X^{(j)} = x^{(j)} \mid Y = c_k)$$

比较这 K 个数值的大小，然后取最大的那个数对应的 k。

为了解决可能出现的下溢问题，我们对上面的式子取对数，因为是对 K 项都取对数，不会改变单调性，所以取对数是不影响它们之间的大小关系的。

那目标就变成了：

$$\begin{aligned} y &= \arg\max_{c_k} \left[\log^{P(Y=c_k) \prod_j P(X^{(j)}=x^{(j)} \mid Y=c_k)} \right] \\ &= \arg\max_{c_k} \left[\log^{P(y=c_k)} + \sum_j \log^{P(X^{(j)}=x^{(j)} \mid Y=c_k)} \right] \end{aligned}$$

在求条件概率的时候，也进行变换：

$$\begin{aligned} \log^{P(X^{(j)}=x^{(j)} \mid Y=c_k)} &= \log^{\left[\frac{1}{\sqrt{2\pi\sigma_{c_k j}^2}} \exp\left(- \frac{(a_{jl} - \mu_{c_k j})^2}{2\sigma_{c_k j}^2} \right) \right]} \\ &= \log^{\frac{1}{\sqrt{2\pi\sigma_{c_k j}^2}}} + \log^{\exp\left(- \frac{(a_{jl} - \mu_{c_k j})^2}{2\sigma_{c_k j}^2} \right)} \\ &= -\frac{1}{2} \log^{2\pi\sigma_{c_k j}^2} - \frac{1}{2} \frac{(a_{jl} - \mu_{c_k j})^2}{\sigma_{c_k j}^2} \end{aligned}$$

所以，高斯朴素贝叶斯就可以变形为：

$$y = \arg\max_{c_k} \left[\log^{P(y=c_k)} + \sum_j \left(- \frac{1}{2} \log^{2\pi\sigma_{c_k j}^2} - \frac{1}{2} \frac{(a_{jl} - \mu_{c_k j})^2}{\sigma_{c_k j}^2} \right) \right]$$

上式就是我们需要求的，我们要求出 K 个值，然后求最大的那个对应的 k。

导入数据集，划分数据集后开始实现朴素贝叶斯分类器

```
class myGaussianNB:
    '''
    处理连续特征的高斯朴素贝叶斯
    '''
    def __init__(self):
        '''
        初始化四个字典
        self.label_mapping    类标记 与 下标(int)
        self.probability_of_y 类标记 与 先验概率(float)
        self.mean             类标记 与 均值(np.ndarray)
        self.var              类标记 与 方差(np.ndarray)
        '''
        self.label_mapping = dict()
        self.probability_of_y = dict()
        self.mean = dict()
        self.var = dict()

    def _clear(self):
        '''
        为了防止一个实例反复的调用 fit 方法，我们需要每次调用 fit 前，将之前学习到的参数删除掉
        '''
        self.label_mapping.clear()
        self.probability_of_y.clear()
        self.mean.clear()
        self.var.clear()

    def fit(self, trainX, trainY):
        '''
        这里，我们要根据 trainY 内的类标记，针对每类，计算这类的先验概率，以及这类训练样本每个特征的均值和方差

        Parameters
        ----------
            trainX: np.ndarray, 训练样本的特征, 维度：(样本数, 特征数)

            trainY: np.ndarray, 训练样本的标记, 维度：(样本数, )
        '''

        # 先调用_clear
        self._clear()

        # 获取类标记
        labels = np.unique(trainY)

        # 添加类标记与下标的映射关系
        self.label_mapping = {label: index for index, label in enumerate(labels)}

        # 遍历每个类
        for label in labels:

            # 取出为 label 这类的所有训练样本，存为 x
```

```
        x = trainX[trainY == label, :]

        # 计算先验概率，用 x 的样本个数除以训练样本总个数，存储到 self.probability_of_y 中，键为 label，值为先验概率
        # YOUR CODE HERE
        self.probability_of_y[label] = len(x)/len(trainX)

        # 对 x 的每列求均值，使用 keepdims = True 保持维度，存储到 self.mean 中，键为 label，值为每列的均值组成的一个二维 np.ndarray
        # YOUR CODE HERE
        self.mean[label] = np.mean(x,axis=0,keepdims=True)

        # 这句话是 debug 用的，如果不满足下面的条件，会直接跳出
        assert self.mean[label].shape == (1, trainX.shape[1])

        # 对 x 的每列求方差，使用 keepdims = True 保持维度，存储到 self.var 中，键为 label，值为每列的方差组成的一个二维 np.ndarray
        # YOUR CODE HERE
        self.var[label] = np.var(x,axis=0,keepdims=True)

        # debug
        assert self.var[label].shape == (1, trainX.shape[1])

        # 平滑，因为方差在公式的分母部分，我们要加一个很小的数，防止除以 0
        self.var[label] += 1e-9 * np.var(trainX, axis = 0).max()

    def predict(self, testX):
        '''
        给定测试样本，预测测试样本的类标记，这里我们要实现化简后的公式

        Parameters
        ----------
            testX: np.ndarray, 测试的特征, 维度：(测试样本数, 特征数)

        Returns
        ----------
            prediction: np.ndarray, 预测结果, 维度：(测试样本数, )
        '''

        # 初始化一个空矩阵 results，存储每个样本属于每个类的概率，维度是 (测试样本数，类别数)，每行表示一个样本，每列表示一个特征
        results = np.empty((testX.shape[0], len(self.probability_of_y)))

        # 初始化一个列表 labels，按 self.label_mapping 的映射关系存储所有的标记，一会儿会在下面的循环内部完成存储
        labels = [0] * len(self.probability_of_y)

        # 遍历当前的类，label 为类标记，index 为下标，我们将每个样本预测出来的这个 label 的概率，存到 results 中的第 index 列
        for label, index in self.label_mapping.items():

            # 先验概率存为 py
            py = self.probability_of_y[label]
```

```
        # 使用变换后的公式，计算所有特征的条件概率之和，存为 sum_of_conditional_probability
        # YOUR CODE HERE
        conditional_probability = -np.log(2*math.pi*self.var[label])/2-(testX-self.mean[label])*(testX-
self.mean[label])/self.var[label]/2
        sum_of_conditional_probability = np.sum(conditional_probability,axis = 1)

        # debug
        assert sum_of_conditional_probability.shape == (len(testX), )

        # 使用变换后的公式，将 条件概率 与 log 先验概率 相加，存为 result，维度应该是 (测试样本数, )
        # YOUR CODE HERE
        result = np.log(py)+ sum_of_conditional_probability

        # debug
        assert result.shape == (len(testX), )


        # 将所有测试样本属于当前这类的概率，存入到 results 中
        results[:, index] = result

        # 将当前的 label，按 index 顺序放入到 labels 中
        labels[index] = label

    # 将 labels 转换为 np.ndarray
    np_labels = np.array(labels)

    # 循环结束后，就计算出了给定测试样本，当前样本属于这类的概率的近似值，存放在了 results 中，
每行对应一个样本，每列对应一个特征
    # 我们要求每行的最大值对应的下标，也就是求每个样本，概率值最大的那个下标是什么，结果存入
max_prob_index 中
    # YOUR CODE HERE
    max_prob_index = np.argmax(results,axis = 1)

    # debug
    assert max_prob_index.shape == (len(testX), )

    # 现在得到了每个样本最大概率对应的下标，我们需要把这个下标变成 np_labels 中的标记
    # 使用上面小技巧中的第五点求解
    # YOUR CODE HERE
    prediction = np_labels[max_prob_index]

    # debug
    assert prediction.shape == (len(testX), )

    # 返回预测结果
    return prediction
```

三 聚类

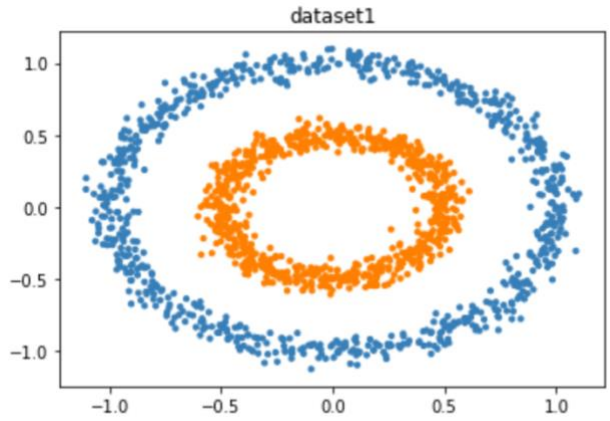
第一题

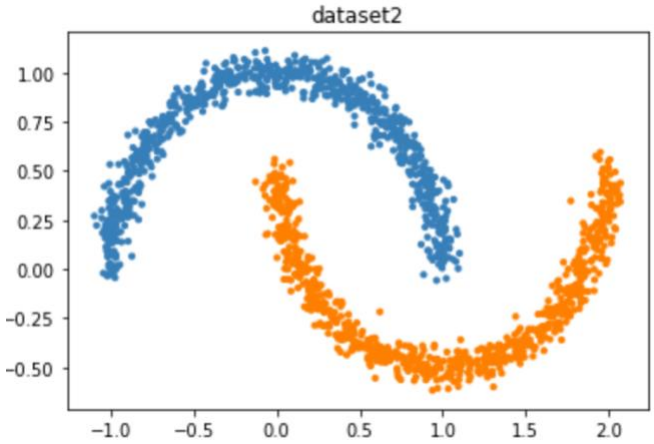
实现 DBSCAN

生成圆形和月形的数据集

```
from sklearn import datasets
X1, y1 = datasets.make_circles(n_samples = 1500,
factor = 0.5, noise = 0.05, random_state = 32)
X2, y2 = datasets.make_moons(n_samples = 1500,
noise = 0.05, random_state = 32)
```

如图



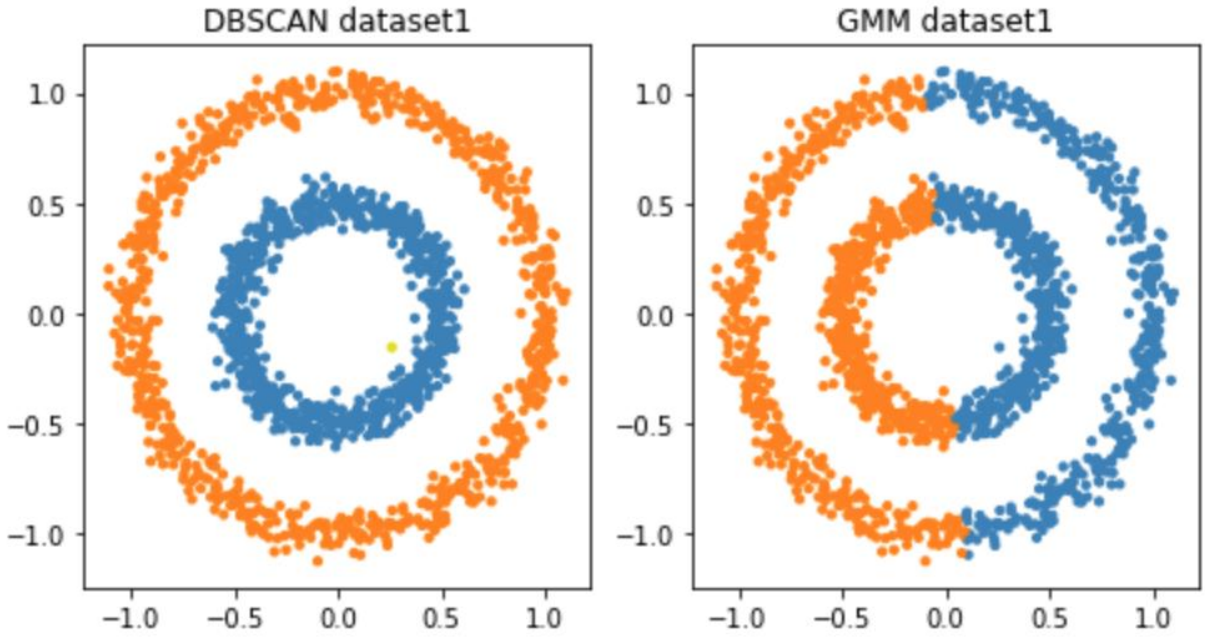


引入模型

```
from sklearn.cluster import DBSCAN
from sklearn.mixture import GaussianMixture
```

结果可视化

```
plt.figure(figsize = (8, 4))
plt.subplot(121)
plt.title('DBSCAN dataset1')
plt.scatter(X1[:, 0], X1[:, 1], s = 10, c =
colors[dbscan.labels_])

plt.subplot(122)
plt.title('GMM dataset1')
plt.scatter(X1[:, 0], X1[:, 1], s = 10, c =
colors[gmm.predict(X1)])
```



指标计算

我们这里选用两个外部指标，FMI 和 NMI。

互信息(mutual information)表示了两个分布的一致程度。归一化的互信息(NMI)将互信息值映射到 0 到 1 的空间内。值越高，说明两个分布的一致性越高。

FMI 是 Fowlkes-Mallows index，使用 precision 和 recall 计算得到，其值域也是 0 到 1，越大说明聚类效果越和参考模型相近。

```
from sklearn.metrics import normalized_mutual_info_score
from sklearn.metrics import fowlkes_mallows_score
```

第二题

实现 kmeans

分成 k 类 cluster

引入模型

```
from sklearn.cluster import KMeans
```

训练模型

```
model = KMeans(n_clusters=2)
model.fit(X1)
```

进行预测

```
model.predict(X1)
```

聚类结果可视化

```
plt.title('dataset1')
plt.scatter(X1[:, 0], X1[:, 1], s = 10,
color=colors[model.predict(X1)])
```

第三题

实现层次聚类

导入模型

```
from sklearn.cluster import AgglomerativeClustering
```

训练模型

```
model = AgglomerativeClustering(n_clusters = 2, linkage="single")
model.fit(X1)
```

此处 linkage 出了有 single 值，还有 complete 和 average。

层次聚类的 linkage 参数表示我们要用哪种距离（最小、最大、平均）进行聚类。这里我们选择 single，表示最小距离。complete 表示最大距离，average 表示平均距离。

对于层次聚类，使用 labels_来获得聚类活动
数据预测
model.labels_
聚类效果可视化
```
pred = model.labels_
plt.title('dataset1')
plt.scatter(X1[:, 0], X1[:, 1], s = 10, color = colors[pred])
```

之后使用另外两个 linkage 参数，并计算 dataset2 的结果
与上述雷同。

第四题

自己实现 kmeans

1）初始化

K-means 在实现的时候，首先需要选取类簇中心。类簇中心的选取方法有很多，我们这里使用最简单的方法，随机选取。也就是，从给定的待聚类的样本中，随机选取 K 个样本，作为 K 个类簇的中心。

2）优化

选取类中心后，就需要不断的调成类中心的位置，开始优化过程，优化主要分为两步：

第一步

计算所有样本到 K 个类中心的距离。每个样本，选择距自己最近的类中心作为自己属于的类簇。（这里的距离我们选择欧式距离）

第二步

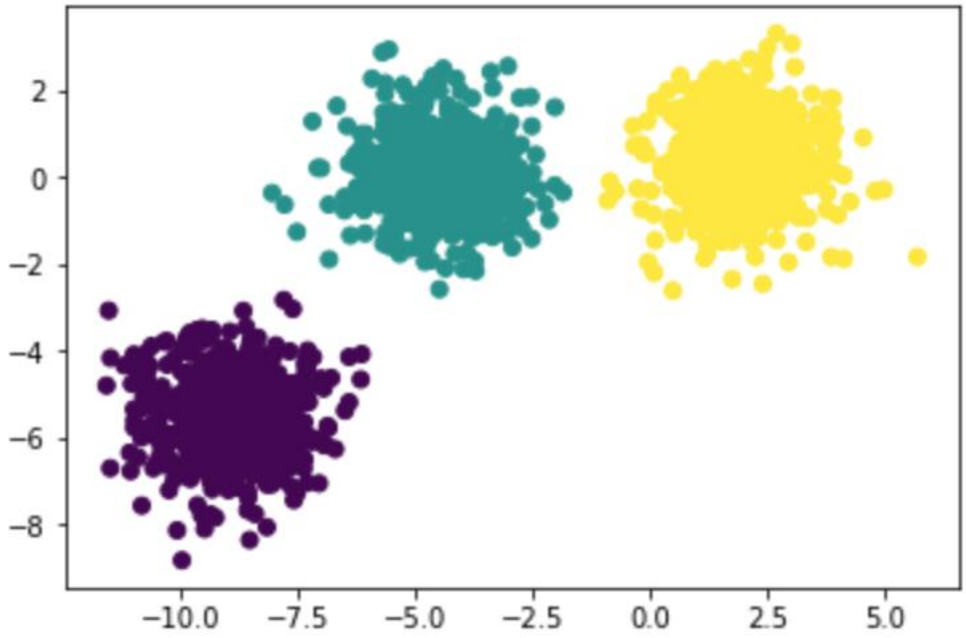
针对第一步分出来的 K 个类簇，计算每个类簇内样本的均值，将计算得到的 K 个均值向量，作为这 K 个类簇新的中心。

然后循环第一步和第二步，直至一定的迭代次数，或类中心无显著的位置改变为止。

3）导入模块生成数据集

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import warnings
warnings.filterwarnings('ignore')
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples = 1500, random_state = 170)
```

plt.scatter(X[:, 0], X[:, 1], c = y)



3. 欧式距离的实现

给定向量 $x \in \mathbb{R}^m$，$y \in \mathbb{R}^m$，两个向量的欧式距离定义为：

$$E(x,y) = \sqrt{\sum_{i=1}^{m}(x_i - y_i)^2}$$

其中，i 表示向量的第 i 个分量。

我们要实现一个可以计算多个样本组成的矩阵 X，与某一个类中心 y 之间欧氏距离的函数。

给定输入矩阵 $X \in \mathbb{R}^{n \times m}$，其中 n 是样本数，m 是特征数，给定输入的类簇中心 $y \in \mathbb{R}^m$。

我们要计算 n 个样本到某一类簇中心 y 的欧式距离，最后的结果是 $E \in \mathbb{R}^n$，每个元素表示矩阵 X 中的每个样本到类中心 y 的欧式距离。

下面要计算数据与数据之间的距离，就要求它们之间的欧式距离。

根据上述 E（x，y）的表达式，编写求解距离的函数

```
def compute_distance(X, y):
    '''
    计算样本矩阵 X 与类中心 y 之间的欧氏距离

    Parameters
    ----------
    X, np.ndarray, 样本矩阵 X, 维度：(n, m)
    y, np.ndarray, 类中心 y，维度：(m, )
    Returns
    ----------
    distance, np.ndarray, 样本矩阵 X 每个样本到类中心 y 之间的欧式距离，维度：(n, )
```

```
    '''

    # YOUR CODE HERE
    distance = (np.sum((X-y)**2,axis=1))**0.5
    return distance
```

下面实现 mykmeans 的算法

```
class myKmeans:
    def __init__(self, n_clusters, max_iter = 100):
        '''
        初始化，三个成员变量

        Parameters
        ----------
        n_clusters: int, 类簇的个数

        max_iter, int, default 100, 最大迭代轮数，默认为 100

        '''
        # 表示类簇的个数
        self.n_clusters = n_clusters

        # 表示最大迭代次数
        self.max_iter = int(max_iter)

        # 类簇中心
        self.centroids = None

    def choose_centroid(self, X):
        '''
        选取类簇中心

        Parameters
        ----------
        X: np.ndarray, 样本矩阵 X，维度：(n, m)

        Returns
        ----------
        centroids: np.ndarray, 维度：(n_clusters, m)

        '''
        centroids = X[np.random.choice(np.arange(len(X)), self.n_clusters, replace = False), :]
        return centroids
    ——
```

上述语句
np.arange(lenx)会生成一个 0 到 lenx-1 的 ndarray
也就是 0 一 lenx-1
然后用 np.random.choice,replace=false 做无放回的抽样，
抽取的是 self.n_clusters 个

也就是抽取要分成簇的个数，先自动随机选取 k 个类簇中心

——

```
    def compute_label(self, X):
        '''
        给定样本矩阵 X，结合类中心矩阵 self.centroids，计算样本矩阵 X 内每个样本属于哪个类簇

        Parameters
        ----------
        X: np.ndarray, 样本矩阵 X，维度：(n, m)

        Returns
        ----------
        labels: np.ndarray, 维度：(n, )

        '''
        # 将每个样本到每个类簇中心的距离存储在 distances 中，每行表示当前样本对于不同的类中心的距离
        distances = np.empty((len(X), self.n_clusters))

        # 遍历类中心，对每个类中心，计算所有的样本到这个类中心的距离
        for index in range(len(self.centroids)):

            # 计算样本矩阵 X 所有样本到当前类中心的距离，存储在 distances 中的第 index 列中
            # YOUR CODE HERE
            distances[:, index] = compute_distance(X, self.centroids[index,:])
```

上述语句将距离中的一行，也就是一个 label 的 distance 存入 distances 的列表中成为一个 ndarray

```
        # 取 distances 每行最小值的下标，这个下标就是这个样本属于的类簇的标记
        # YOUR CODE HERE
        labels = np.argmin(distances, axis = 1)

        # 返回每个样本属于的类簇的标记
        return labels

    def fit(self, X):
        '''
        聚类，包含类中心初始化，类中心优化两个部分

        Parameters
        ----------
        X: np.ndarray, 样本矩阵 X，维度：(n, m)

        '''
        # 类中心随机初始化
```

```
        self.centroids = self.choose_centroid(X)

        # 优化 self.max_iter 轮
        for epoch in range(self.max_iter):

            # 计算当前所有样本的属于哪个类簇
            labels = self.compute_label(X)

            # 重新计算每个类簇的类中心
            for index in range(self.n_clusters):

                # 重新计算第 index 个类中心，对属于这个类簇的样本取均值
                # YOUR CODE HERE
                self.centroids[index, :] = np.mean(X[labels==index,:])
```

选出每个 label 为 index 的中心

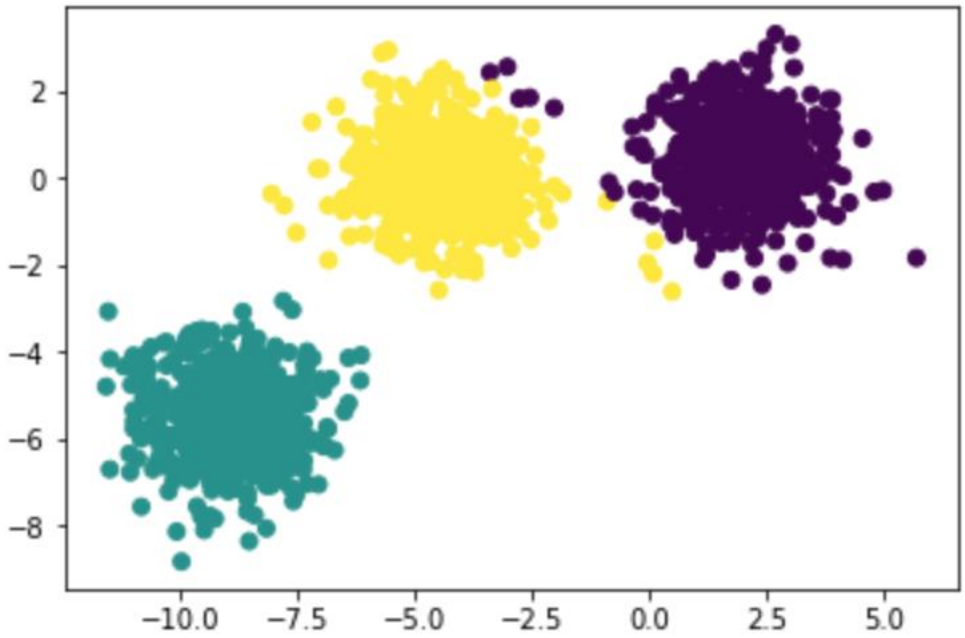
开始进行聚类

```
# 初始化一个 3 类簇的模型
model = myKmeans(3)

# 对 X 进行聚类，计算类中心
model.fit(X)

# 计算 X 的类标记
prediction = model.compute_label(X)
```

聚类结果可视化

```
# 使用我们的预测结果上色
```

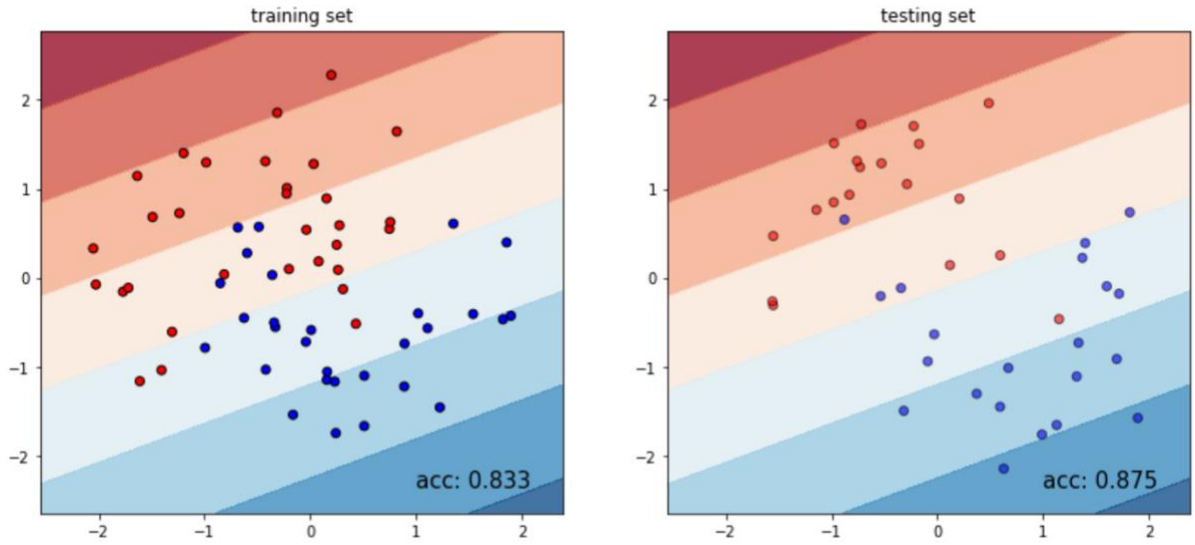
plt.scatter(X[:, 0], X[:, 1], c = prediction)



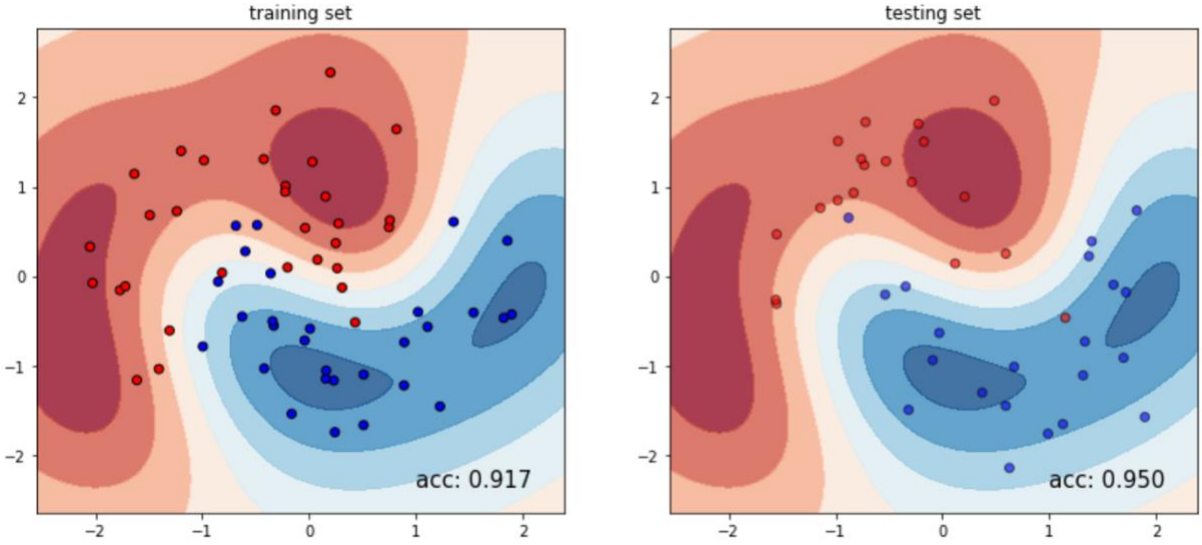
3 实验结果

线性模型

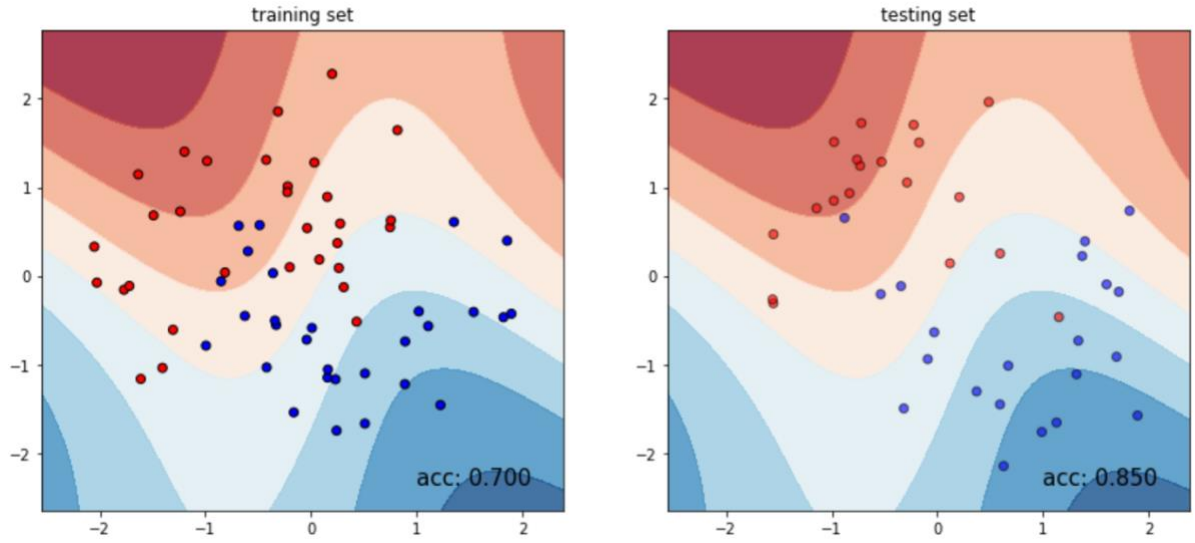
Linear SVM



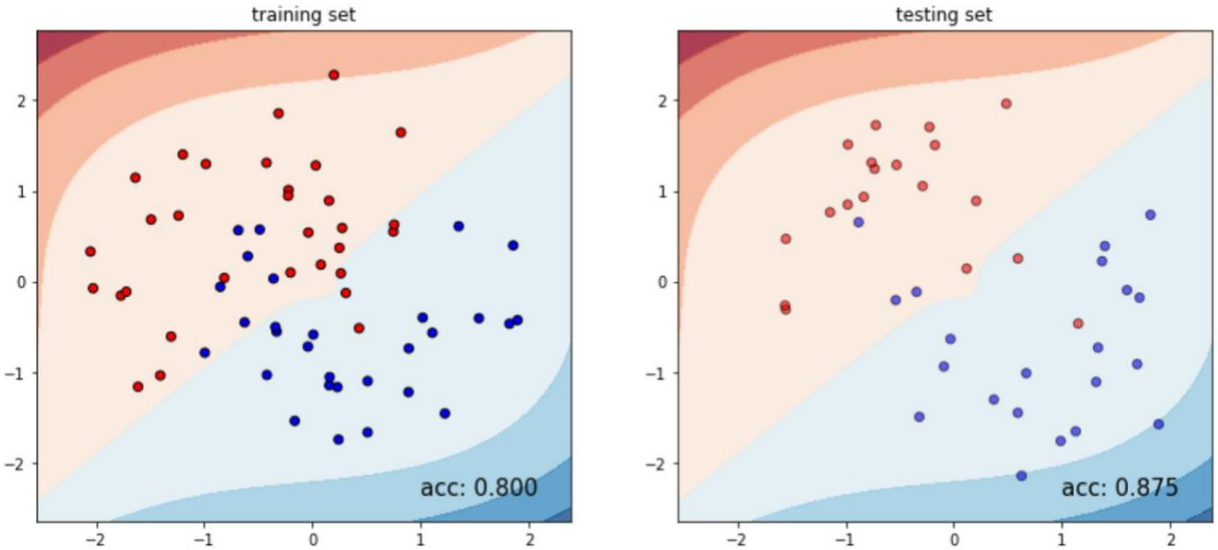
Radial Basis Function

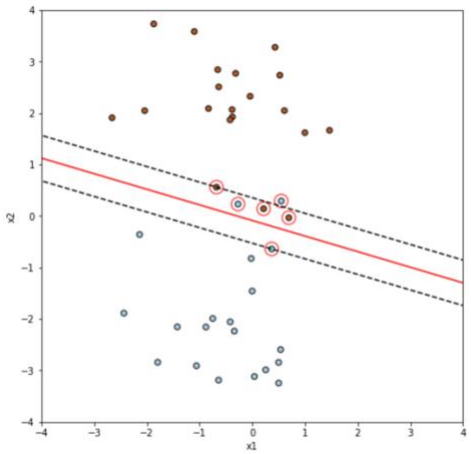


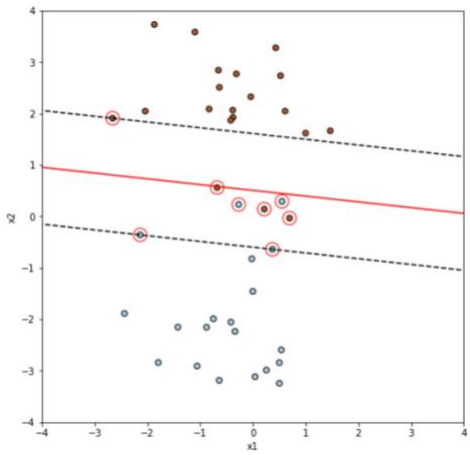
Sigmoid

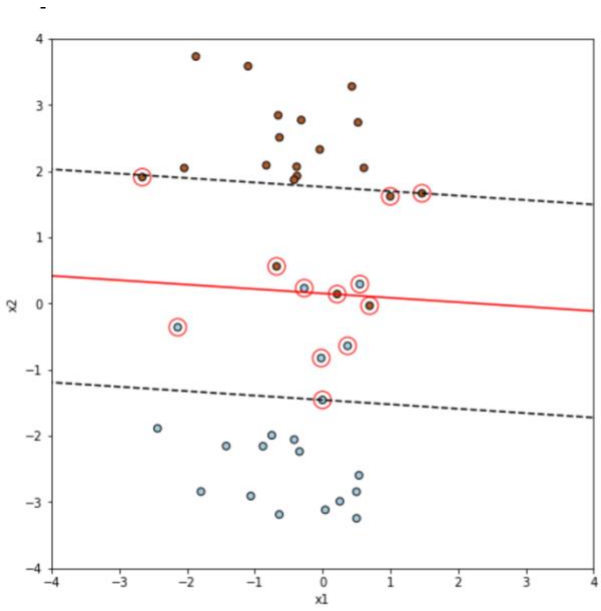


Poly









C 依次递减

10，1，0.1

较 同核函数,以及相同核 同 c 值的区别。结果 分析:线性核函数的 MAE 和 RMse 斯核函数和 sigmoid ,精度和 F1 这两个要 ,因此线性核函数要 斯核函数和 sigmoid 精确。C 越小的时候我们想要越大的间隔

核函数	C	精度	查准率	查全率	F1
rbf	0.1	0.7414916727009413	0.6429780033840947	0.7224334600760456	0.6803939122649955
rbf	1	0.8298334540188269	0.7650273224043715	0.7984790874524715	0.7813953488372094
linear	0.1	0.9000724112961622	0.9024896265560166	0.8269961977186312	0.863095238095238
linear	1	0.8747284576393918	0.8262476894639557	0.8498098859315589	0.837863167760075
sigmoid	0.1	0.45981173062997827	0.06349206349206349	0.030418250950570342	0.04113110539845758
sigmoid	1	0.37074583635047065	0.12958963282937366	0.11406844106463879	0.12133468149646108

核函数	C	MAE	RMSE
rbf	0.1	56514.17784132763	79839.01037512351
rbf	1	56514.17156396131	79838.90862178346
linear	0.1	44970.89199251632	71316.00062993738
linear	1	45284.06319738523	78535.41963544753
sigmoid	0.1	56514.17853881279	79839.02168120441
sigmoid	1	56514.17853881279	79839.02168120441

贝叶斯分类器

高斯

精度	查准率	查全率	F1值
0.8217778743751358	0.7004440855874041	0.9569773855488142	0.8088578088578089

伯努力

精度	查准率	查全率	F1值
0.8845903064551185	0.8824582338902148	0.8157749586321015	0.8478073946689596

多项

精度	查准率	查全率	F1值
0.9069767441860465	0.8654353562005277	0.9045780474351903	0.8845738942826322

自己实现的高斯朴素贝叶斯分类器

查准率	查全率	F1
0.6829511465603191	0.9620786516853933	0.7988338192419826

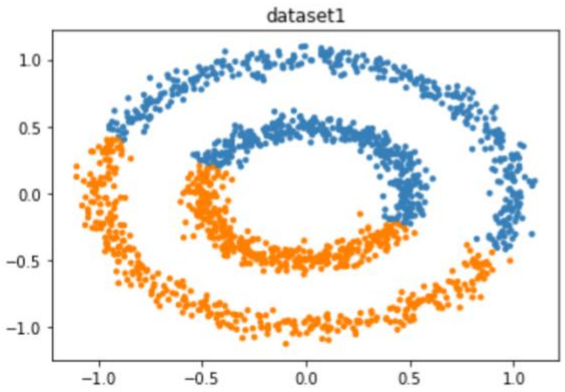
聚类

算法	FMI	NMI
密度聚类	0.9993331109628394	0.9963558570293466
高斯混合模型	0.4993373439981344	1.2824238916756398e-06

5. 聚类结果可视化

```
plt.title('dataset1')
plt.scatter(X1[:, 0], X1[:, 1], s = 10, color=colors[model.predict(X1)])
```

```
<matplotlib.collections.PathCollection at 0x1a1ba08c18>
```



dataset2

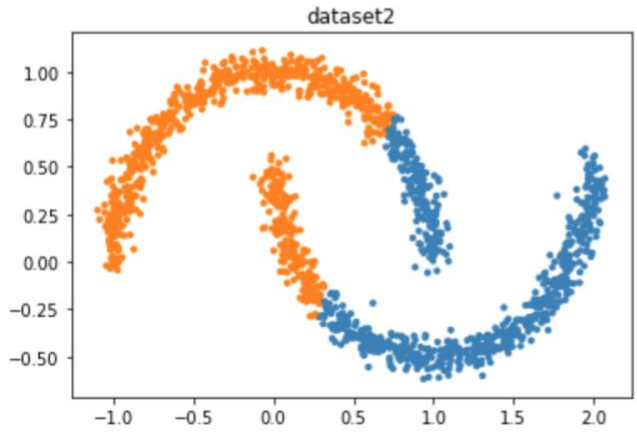
```
model2 = KMeans(n_clusters=2)
model2.fit(X2)
model2.predict(X2)
```

```
array([0, 0, 0, ..., 1, 0, 1], dtype=int32)
```

数据集绘制

```
plt.title('dataset2')
plt.scatter(X2[:, 0], X2[:, 1], s = 10, color=colors[model2.predict(X2)])
```

```
<matplotlib.collections.PathCollection at 0x1a1ba64518>
```



算法	数据集	FMI	NMI
KMeans	数据集1	0.4995087715346489	0.0001283041935915151
KMeans	数据集2	0.5825194545661486	0.1887663125222284

1. 数据集1

距离度量方式	FMI	NMI
single	1	1
complete	0.5092291475803046	0.003349157958282007
average	0.5039593145853657	0.0031237686343950815

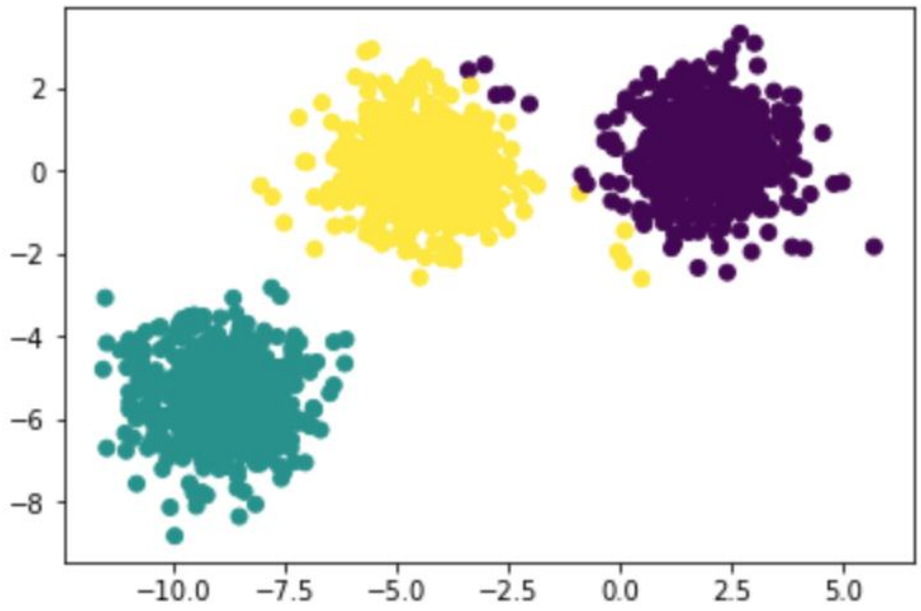
2. 数据集2

距离度量方式	FMI	NMI
single	1	1
complete	0.6755844521626881	0.2768579244983244
average	0.7660505593160595	0.5263489914669924

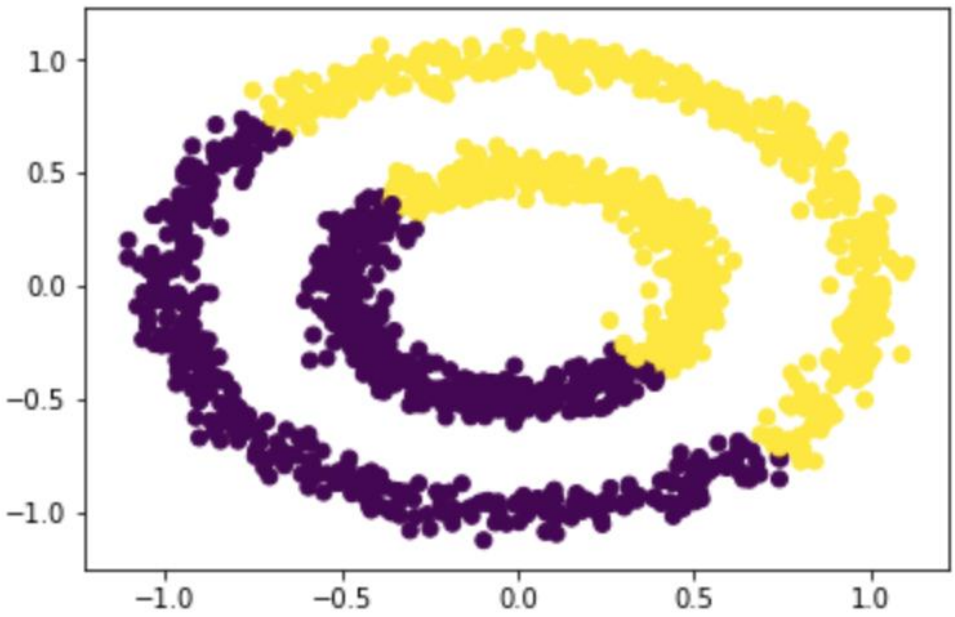
5. 聚类结果可视化

```
# 使用我们的预测结果上色
plt.scatter(X[:, 0], X[:, 1], c = prediction)
```

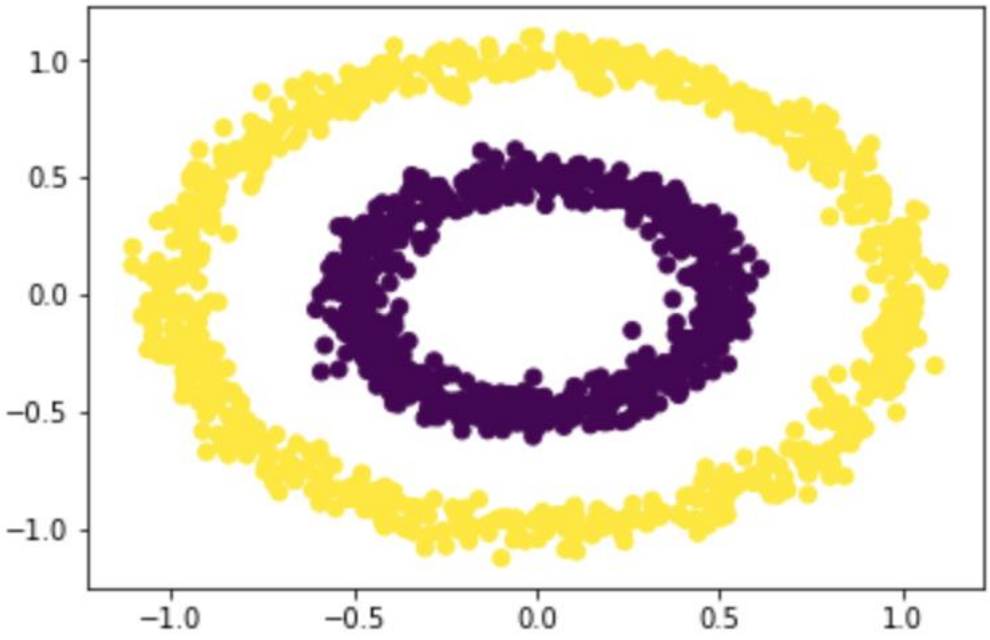
```
<matplotlib.collections.PathCollection at 0x1a1ff2d2b0>
```



算法	FMI	NMI
myKmeans	0.4993907884327544	3.206641029427453e-05
AgglomerativeClustering	1.0	1.0



mykmeans 结果可视化



层次聚类结果可视化

4 参考指标

在聚类时引出的两个外部指标 FMI 和 NMI

6. 指标计算

我们这里选用两个外部指标，FMI和NMI。

互信息(mutual information)表示了两个分布的一致程度。归一化的互信息(NMI)将互信息值映射到0到1的空间内。值越高，说明两个分布的一致性越高。

FMI是Fowlkes-Mallows index，使用precision和recall计算得到，其值域也是0到1，越大说明聚类效果越和参考模型相近。

```
: from sklearn.metrics import normalized_mutual_info_score
  from sklearn.metrics import fowlkes_mallows_score
```

5 学习总结

在这三章中，我们学习了支持向量机，贝叶斯分类器和聚类三种模型。支持向量机找到的是最大的支持向量，在两类（1，-1）中选择中间的超平面，我们还在实验里学习了软间隔和 C 参数，我们简单了解了贝叶斯分类器中三种高斯，伯努力和多项贝叶斯朴素分类器。聚类则是先随机抽取 k 个类中心，通过计算数据与它们的距离，重新判断数据应该是属于什么标记（label）不断递归调用，每次更新 labels 这个列表，如何判断是属于那个标记，即属于哪个簇呢，离的最近的则应分到哪个簇中，之后类中心再取更新后的数据平均值，再次计算，直到类中心不再有显著的移动之后，分类成功。

机器学习是一门比较综合且有实际应用价值的学科。它的核心思想是通过一个模型或者一个结构的建立和训练从数据中挖掘出规律来，从而我们可以从这个模型中继续处理分类回归判断等预测功能。这个机器模型的构想已经是比较具有挑战性了，因为我们要把自然情形下的分类情况抽象成一个模型，而这个模型要简单易行，要有一定的泛化能力；再有在数学层次上，机器学习的严谨性也是非常具有难度，由于机器中的运算和生活中的运算的差别尚存在，我们面临很多问题，由于计算机具有反复重复计算、操作的优势，我们在连乘的时候很容易出现数据爆炸或者消失。我们需要针对这些情形进行预想并专门解决在不断递归中可能会出现的问题。

好在由于机器学习被人们高度关注，现在已经有了很多第三方的工具，如 sci-kit learn，是非常有用的第三方库，里面的算法都进行了优化，在一般情形的使用上，我们可以很方便的调用这些工具。但是如果我们还想让机器学习更加发展，我们需要从原理层面对机器学习的模型和算法进行创新，结合集成学习，我相信有朝一日，机器学习会比现在更被人们所依赖和利用，届时，我们的机器时代也将变得更智能。