程序报告

姓名：秦嘉俊　学号：3210106182

1　问题介绍

手写数字识别（MNIST）是图像分类领域的一个经典问题，它在许多实际应用中具有广泛的应用，如自动邮件分拣和识别手写数字验证码等。在本实验中，我们旨在使用支持向量机（SVM）算法实现邮政手写数字的识别，并通过选择不同的核函数，如多项式核、径向基函数（RBF）核和线性核函数，来比较分类的效果。我们还将通过调整不同的参数，如多项式核的阶数和 RBF 核的参数，来观察它们对分类结果的影响。

此外，我们将尝试基于 Adaboost 的思想，将多个 SVM 分类器组合成一个更好的分类器。通过与单个 SVM 分类器的对比，我们评估了 Adaboost 在提升分类器性能方面的有效性。

2　数据集介绍

我们这里使用的数据集是 MNIST。**MNIST（Modified National Institute of Standards and Technology）**是一个广泛使用的手写数字识别数据集。MNIST 数据集包含了大量的手写数字图像样本，用于训练和评估机器学习算法在图像分类任务上的性能。

MNIST 数据集的主要特点如下：

- 图像样本：MNIST 数据集中的每个样本都是一个 28×28 像素的灰度图像。这意味着每个图像由 784 个像素值组成，每个像素值表示图像对应位置的灰度级别（从 0 到 255 的整数）。
- 标签信息：每个图像样本都有一个与之对应的标签，表示图像中显示的手写数字。标签是一个 0 到 9 之间的整数，对应于数字 0 到 9。
- 数据集划分：MNIST 数据集包括训练集和测试集两部分。训练集通常用于训练机器学习模型，测试集用于评估模型的性能。训练集包含了 60000 个样本，测试集包含了 10000 个样本。
- 数据可用性：MNIST 数据集在机器学习和图像处理领域被广泛应用，它已成为一个公认的基准数据集。该数据集可以轻松地在互联网上获得，并且在许多机器学习框架和库中都有内置的加载函数，方便进行实验和研究。

3 算法介绍

3.1 SVM 算法

支持向量机（Support Vector Machine，简称 SVM）是一种常用的监督学习算法，广泛应用于模式识别、分类和回归等任务。SVM 的目标是在特征空间中找到一个最优超平面，以将不同类别的样本分隔开来，并在保持最大间隔的同时，对新样本进行准确的分类。

3.1.1 基本原理

支持向量机学习的基本思想是求解能正确划分训练数据集并且几何间隔最大的分离超平面。

对于线性可分支持向量机，我们的算法（硬间隔最大化）如下

输入:线性可分训练数据集 $T = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$ 其中 $x_i \in \mathcal{X} = \mathcal{R}^n, y_i \in \mathcal{Y} = \{+1, -1\}, i = 1, 2, \cdots, N.$

输出：最大间隔分离超平面和分类决策函数

1. 构造并求解约束最优化问题，得到最优解 w^*, b^*

$$\begin{aligned} &\max_{w,b} \frac{\hat{\gamma}}{||w||} \\ &s.t. \quad y_i(w \cdot x_i + b) \geq \hat{\gamma}, i = 1, 2, \ldots, N \end{aligned}$$

2. 由此得到分离超平面 $w^* \cdot x + b^* = 0$ 以及分类决策函数 $f(x) = sign(w^* \cdot x + b^*)$

硬间隔最大化适合于线性可分的数据集。而对于线性不可分的数据集，我们可以引入松弛变量 $\xi_i \geq 0$, 修改约束条件，得到软间隔最大化的学习方法，这里不再重复。

3.1.2 核技巧

原始的线性分类支持向量机对求解线性分类问题是非常有效的，但当分类是非线性的时，我们就需要核技巧。

核技巧（Kernel Trick）是一种重要的技术，用于处理非线性分类问题。它通过将数据从原始空间映射到高维特征空间，从而使得在原始空间中线性不可分的问题在高维特征空间中成为线性可分的问题。核技巧的核心思想是通过计算在特征空间中的内积，而不必显式地计算特征空间的映射。

在 SVM 中，当原始空间中的样本点无法用线性超平面进行完美分类时，就出现了线性不可分问题。例如，当样本点呈现环状或螺旋形状分布时，无法用一条直线将不同类别的样本点分开。这时，传统的线性分类方法无法解决这种问题。为了解决线性不可分问题，核技

巧引入了非线性映射。它将原始空间中的样本点映射到一个更高维度的特征空间，使得在该特征空间中，原本线性不可分的问题变得线性可分。

常用的核函数包括多项式核、径向基函数（RBF）核和线性核函数等，选择不同的核函数取决于数据的特征和问题的性质。

3.1.3 多类别分类

SVM 最初是用来解决二分类问题，但在本次编程大作业以及平时实际应用中，我们仍然可以利用 SVM 实现多分类。常见的策略是将多分类问题转化为二分类问题解决的，以下为两种常用策略：

- **一对多（One-vs-Rest）策略**
  具体步骤如下：
  1. 对于每个类别 i，将它作为正例，将其他类别作为负例，构建一个二分类器。
  2. 对于每个二分类器，使用训练数据对其进行训练。
  3. 在预测时，通过计算新样本到每个二分类器的距离或得分，选择具有最高距离或得分的类别作为预测结果。

  一对多策略的优点是简单直观，且训练速度较快。然而，它可能存在类别不平衡的问题，即某些类别的样本数量远多于其他类别，这可能导致分类器对少数类别的识别能力较弱。

- **一对一（One-vs-One）策略**
  具体步骤如下：
  1. 对于每两个类别 i 和 j，将它们分别作为正例和负例，构建一个二分类器。
  2. 对于每个二分类器，使用训练数据对其进行训练。
  3. 在预测时，通过投票或多数表决的方式，选择获得最高票数的类别作为预测结果。

  一对一策略的优点是能够充分利用每个二分类器之间的差异性，避免了类别不平衡问题。然而，它的缺点是需要构建更多的二分类器，增加了训练时间和空间复杂度。

3.2 Adaboost 算法

Adaboosting（Adaptive Boosting）是一种集成学习算法，用于提升弱分类器的性能。它是由 Freund 和 Schapire 在 1996 年提出的。Adaboosting 算法通过迭代训练一系列弱分类器，将它们加权组合成一个强分类器，从而提高整体分类的准确性。其基本流程如下：

1. 初始化样本权重
   对于包含 N 个样本的训练集，初始化每个样本的权重为相等的值，即 $\frac{1}{N}$。

2. 迭代训练弱分类器

   (a) 在当前样本权重下，训练一个弱分类器。弱分类器通常是一个性能稍好于随机猜测的分类器，例如决策树桩（只有一个分支的决策树）。

   (b) 计算该弱分类器的误差率（错误分类的样本在总样本中的比例）。

   (c) 根据误差率计算该弱分类器的权重。误差率越小的弱分类器获得的权重越大，表示其在整体分类中的重要性。

   (d) 更新样本权重，增加被错误分类的样本的权重，减小被正确分类的样本的权重。

3. 更新样本权重
   通过对错误分类的样本增加权重，减少正确分类的样本权重，来聚焦于被错误分类的样本。

4. 组合弱分类器
   将所有训练得到的弱分类器按其权重进行加权组合，形成一个强分类器。

5. 重复迭代
   重复步骤 2 至步骤 4，直到达到指定的迭代次数或分类准确率满足要求。

最终的 Adaboosting 分类器是由多个弱分类器组成的集合，每个弱分类器的权重取决于其分类性能。在预测时，通过对弱分类器进行加权投票或加权求和来得到最终的分类结果。

Adaboosting 算法的优点包括：

- 能够处理复杂的分类问题，具有较高的准确性。
- 不容易发生过拟合，具有较好的泛化性能。
- 对于弱分类器的选择没有严格要求，可以使用简单的分类器作为基分类器。

然而，Adaboosting 算法也存在一些限制：

- 对噪声和异常值敏感，容易受到噪声样本的干扰。
- 训练过程中每个弱分类器都要依赖前面的分类器，因此训练速度较慢。
- 如果弱分类器性能较差，Adaboosting 的整体性能也可能较差。

在这里，我们尝试将 SVM 作为基分类器，并利用 Adaboosting 的思想将多个 SVM 基分类器组合起来，以提高分类效果。

4 代码实现

我们使用 Python 进行代码实现。注意到 `scikit-learn`（`sklearn`）库中的 SVM 分类器支持多分类问题。`sklearn` 中的 SVM 分类器使用了一对一的策略来实现多分类。因此我们这里直接调用了 `sklearn` 库中的 SVM 分类器。同理，我们可以从 `sklearn.ensemble` 中引入 `AdaBoostClassifier` 实现 Adaboosting 的分类器。

4.1 导入数据集

我们提前将 MNIST 的数据集下载，并解压后放在了 `data` 文件夹下。我们利用 Python 的 `struct` 模块。注意到所有的数字都是按照 MSB（大端）的方式存储的，我们可以在 `struct` 中的 `fmt` 格式符中加上大于号以解决这种情况。随后我们按照图像和标签的格式依次处理，并保存到数据中即可。

4.2 建立模型并测试

`sklearn` 中的 SVM 分类器有诸多参数供我们选择：

```
1   sklearn.svm.SVC(C=1.0, kernel='rbf', degree=3, gamma='auto', coef0=0.0,
        shrinking=True, probability=False, tol=0.001, cache_size=200,
        class_weight=None, verbose=False, max_iter=-1, decision_function_shape
        =None,random_state=None)
```

这里是部分重要参数的介绍：

- `C`（默认值为 `1.0`）
  `C` 是 SVM 分类器的正则化参数。它控制了错误分类样本的惩罚程度，`C` 值越小，容忍更多的错误分类，`C` 值越大，对错误分类的惩罚越严厉。较小的 `C` 值可以使决策边界更加平滑，但可能导致训练误差较大。

- `kernel`（默认值为 `"rbf"`）
  `kernel` 参数指定 SVM 分类器使用的核函数。常用的核函数包括线性核函数（`"linear"`）、多项式核函数（`"poly"`）、径向基函数核（`"rbf"`）等。不同的核函数适用于不同类型的数据和分类问题，需要根据具体情况进行选择。

- `gamma`（默认值为 `"scale"`）
  `gamma` 参数是径向基函数核（RBF kernel）和多项式核函数（polynomial kernel）的一个参数。它控制了数据点对决策边界的影响力。较小的 `gamma` 值表示影响范围较大，较大的 `gamma` 值表示影响范围较小。

- `degree`（默认值为 `3`）
  `degree` 参数是多项式核函数的阶数。它控制多项式核函数的复杂度。较低的阶数通常适用于简单的非线性问题，而较高的阶数可以处理更复杂的非线性问题。

- `coef0`（默认值为 `0.0`）
  `coef0` 参数是多项式核函数和 Sigmoid 核函数的一个参数。它控制了多项式核函数和 Sigmoid 核函数的影响力。较大的 `coef0` 值会增加非线性映射的影响，可以使决策边界更加灵活。

- `probability`（默认值为 `False`）
  `probability` 参数指定是否启用概率估计。当设置为 `True` 时，SVM 分类器将计算每个样本属于每个类别的概率。这对一些后续处理（如概率阈值的设定）可能是有用的。

为了便于调参，我们使用了 `grid_search`, 通过穷举搜索给定参数范围内的所有可能组合，并通过交叉验证来评估每种参数组合的性能，从而选择最佳的参数组合。

下面是我们 SVM 选择的参数组合

```
svm_module_1 = svm.SVC(kernel="rbf", probability=True) # RBF 核函数
param_grid_rbf = {'C': [1e-2, 1e-1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1.0]}
svm_module_2 = svm.SVC(kernel="poly", probability=True) # 多项式核函数
param_grid_poly = {'C': [1e-2, 1e-1, 1, 10, 100], 'gamma': [0.1, 0.01, 1.0], 'degree': [1, 3, 5, 7]}
svm_module_3 = svm.SVC(kernel="linear", probability=True) # 线性核函数
param_grid_linear = {'C': [1e-2, 1e-1, 1, 10, 100]}
svm_module_4 = svm.SVC(kernel="sigmoid", probability=True) # 线性核函数
param_grid_sigmoid = {'C': [1e-2, 1e-1, 1, 10, 100], 'gamma': [1, 2, 3, 4], 'coef0': [0.2, 0.4, 0.6, 0.8, 1 ]}
```

不同核函数选出最佳参数后，我们再将 SVM 作为基分类器去实现 Adaboosting. 这里 AdaBoostClassifier 也有下面这些可以调节的参数：

- `n_estimators`
  弱分类器的数量，默认为 50. 增加弱分类器的数量可以提高模型的复杂度和性能，但也可能导致过拟合。

- `learning_rate`
  学习率，默认为 1.0. 学习率控制每个弱分类器的权重，较小的学习率可以降低过拟合的风险，但可能需要更多的弱分类器来达到相同的性能。

AdaBoostClassifier 相关代码如下

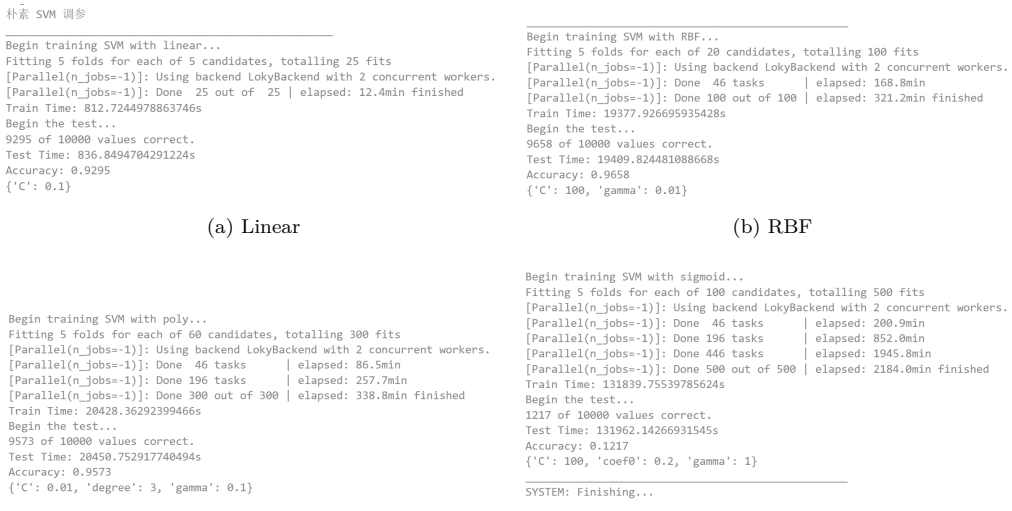
```
base_rbf = svm.SVC(kernel="rbf", probability=True, C=100, gamma=0.01)

param_grid = {
   'n_estimators': [50, 100],
   'learning_rate': [1.0, 0.5],
}
model_rbf = AdaBoostClassifier(base_estimator=base_rbf)
grid_search_rbf_ada = GridSearchCV(model_rbf, param_grid, n_jobs = -1,
    verbose = 1)
Test(grid_search_rbf_ada)
print(grid_search_rbf_ada.best_params_)
```

5 实验结果及分析

5.1 SVM

这里为了时间考虑，我们没有使用完整的 MNIST 数据集，而是在训练中选取了前 10000 个数据用于训练。测试集与 MNIST 测试集相同，共 10000 个测试数据。经过几十个小时的计算，我们得到了如下的输出：



(a) Linear　　(b) RBF

(c) Poly　　(d) Sigmoid

图 1: 朴素 SVM 算法

可以看到除了 Sigmoid 核外，我们朴素的 SVM 算法均有相当高的分类正确率，其中 RBF 核函数的效果最好。

5.2 Adaboosting

基于上面得到的 RBF 核函数的最佳参数，我们将其作为 Adaboosting 的基分类器进行 Adaboosting 的分类。

```
SVM + Adaboosting
Begin training Adaboosting with RBF SVM...
Fitting 5 folds for each of 4 candidates, totalling 20 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
/home/jovyan/.virtualenvs/basenv/lib/python3.7/site-packages/joblib/externals/
  "timeout or by a memory leak.", UserWarning
[Parallel(n_jobs=-1)]: Done  20 out of  20 | elapsed: 709.9min finished
Train Time: 46796.8696204945455s
Begin the test...
8860 of 10000 values correct.
Test Time: 48560.585398448863s
Accuracy: 0.886
{'learning_rate': 0.5, 'n_estimators': 50}
SYSTEM: Finishing...
SYSTEM: Done!
```

图 2: 基于 SVM 的 Adaboosting 算法

可以看到使用了 Adaboosting 后我们的模型并没有提高正确率，相反效果还不如朴素的 SVM 模型。

5.3 分析

可以看到在 MNIST 手写数字识别任务中，对于不同的核函数类型，SVM 的效果有所不同，我认为原因主要在于以下几点

- 其中 Linear 核函数用于线性 SVM 分类器，它在处理线性可分问题时表现良好。在 MNIST 数据集这样高维且复杂的图像分类任务中，线性模型可能无法充分捕捉到手写数字的非线性特征，因此其效果较差。

- Polynomial 核函数在处理非线性问题时具有一定的优势。通过调节多项式的阶数，可以引入更高阶的特征交互，从而增强模型的非线性拟合能力。然而，在 MNIST 数据集上使用多项式核函数时，过高的多项式阶数可能导致过拟合，而过低的阶数可能无法充分捕捉到手写数字的特征。

- Sigmoid 核函数是一种常用的非线性核函数，但在 MNIST 数据集上使用 Sigmoid 核函数分类性能十分差，我认为是因为 Sigmoid 核函数的特性不适合处理复杂图像分类任务。MNIST 数据集包含大量的手写数字图像，每个数字具有复杂的纹理和形态特征，需要一个具有较强非线性拟合能力的分类器来准确识别。

  Sigmoid 核函数的主要问题在于其输出在零点两侧不对称，可能导致决策边界的偏离和分类器的偏差。此外，Sigmoid 函数具有饱和性和难以调节的形状，这也限制了它在复杂分类任务中的表现。

- RBF 核函数具有较好的非线性建模能力，可以更好地适应手写数字的复杂特征，从而提高分类性能。因此在我们的实验中可以看到 RBF 核的效果是最好的。

而对于使用 Adaboosting 后的模型正确率不增反降，我认为原因主要在于以下几点

- 过拟合。Adaboosting 算法容易在训练集上过拟合，尤其是当基分类器的复杂性较高时。基于 RBF 核的 SVM 模型作为基分类器可能具有更高的复杂性，因此容易在 Adaboosting 的迭代过程中过拟合训练数据，导致性能下降。
- 样本噪声。Adaboosting 对于噪声样本比较敏感。如果训练数据中存在噪声样本，这些噪声样本可能会对 Adaboosting 的训练过程产生不利影响，从而降低整体的分类性能。
- 样本权重。Adaboosting 通过调整样本的权重来训练基分类器，使得错误分类的样本权重增加。然而，基于 RBF 核的 SVM 模型对于一些困难样本可能并不表现出较好的分类性能，这可能导致 Adaboosting 过于关注这些困难样本，而忽略了其他更容易分类的样本，从而影响了整体的分类性能。
- 训练数据量。因为本身 MNIST 数据集的规模非常大，我们的算力不足以支撑完整的计算，因此实际只选择了较低比例的训练数据集用于训练，可能因此训练效果不佳，无法得到很好的模型。

6 改进与讨论

我认为将 SVM 作为基分类器，并通过 Adaboosting 的思想将基分类器组合是一个非常好的思路，可以获得较好的手写数字识别性能。通过适当的参数调优、特征选择和数据处理技术，可以进一步提高算法的性能。

在实现中，这样的做法也存在一些问题：

- 计算复杂度高：SVM 算法在处理大规模数据集时可能需要较长的训练时间，特别是当使用复杂的核函数和参数组合时。而基于 SVM 为基分类器的 Adaboosting 算法的时间同样需要非常长的时间。

  正如我们的测试，持续运行了几十个小时才得到答案，在实践中这样大的时间开销几乎是不能接受的。

- 参数调节的困难：SVM 算法中的核函数和正则化参数需要进行调优，以获得最佳的分类性能。这涉及到在大量参数组合中进行搜索，增加了调参的复杂性。

  在调节参数时，我们往往只能选取几个常见的值进行尝试，这样可能遗漏中间的参数。

可能的改进：

- 特征选择和提取：对于 MNIST 数据集，我们可以尝试使用适当的特征选择和特征提取技术，以减少特征维度并提高分类性能。例如，可以使用主成分分析（PCA）或其他降维方法来提取最具信息量的特征。

- 模型压缩和加速：对于 SVM 模型，可以考虑使用模型压缩技术来减少模型的存储空间和计算复杂度，例如使用稀疏表示或低秩近似方法。这可以在保持较好性能的同时减少模型的资源消耗。
- 集成策略：除了传统的 Adaboosting 算法，我们还可以尝试其他集成学习方法，如随机森林、梯度提升决策树等。这些方法能够通过结合多个基分类器的预测结果来提高分类性能。
- 硬件方面：选用更好的 GPU, 以承担更多的计算，通过更多的训练进一步提高我们模型的预测和泛化能力。

Source Code

```
import numpy as np
import timeit
from sklearn import svm
import struct
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import GridSearchCV

TRAIN_ITMES = 60000 # 实际我们只选取了前 10000 个数据用于训练
TEST_ITEMS = 10000

def LoadMNIST():
   mnist_data = []
   for img_file,label_file,items in zip(
      ['data/train-images-idx3-ubyte','data/t10k-images-idx3-ubyte'],
      ['data/train-labels-idx1-ubyte','data/t10k-labels-idx1-ubyte'],
      [TRAIN_ITMES, TEST_ITEMS]):
      data_img = open(img_file, 'rb').read()
      data_label = open(label_file, 'rb').read()
      fmt = '>iiii'
      offset = 0
      magic_number, img_number, height, width = struct.unpack_from(fmt,
          data_img, offset)
      print('magic number is {}, image number is {}, height is {} and width
          is {}'.format(magic_number, img_number, height, width))
      offset += struct.calcsize(fmt)
      image_size = height * width
      fmt = '>{}B'.format(image_size)
      if items > img_number:
```

```
27          items = img_number
28       images = np.empty((items, image_size))
29       for i in range(items):
30          images[i] = np.array(struct.unpack_from(fmt, data_img, offset))
31          images[i] = images[i]/256
32          offset += struct.calcsize(fmt)
33
34       fmt = '>ii'
35       offset = 0
36       magic_number, label_number = struct.unpack_from(fmt, data_label, offset
             )
37       print('magic number is {} and label number is {}'.format(magic_number,
              label_number))
38       offset += struct.calcsize(fmt)
39       fmt = '>B'
40       if items > label_number:
41          items = label_number
42       labels = np.empty(items)
43       for i in range(items):
44          labels[i] = struct.unpack_from(fmt, data_label, offset)[0]
45          offset += struct.calcsize(fmt)
46
47       mnist_data.append((images, labels.astype(int)))
48
49    return mnist_data
50
51 def Test(module):
52    start_time = timeit.default_timer()
53    # train
54    module.fit(training_data[0], training_data[1])
55    train_time = timeit.default_timer()
56    print('Train Time: {}s'.format(str(train_time - start_time) ) )
57    # test
58    print("Begin the test...")
59    predictions = [int(a) for a in module.predict(test_data[0])]
60    correct_nums = sum(int(a == y) for a, y in zip(predictions, test_data[1]))
61
62    print("%s of %s values correct." % (correct_nums, len(test_data[1])))
63    end_time = timeit.default_timer()
64    print("Test Time: {}s".format(str(end_time - start_time) ) )
65    print("Accuracy: {}".format(correct_nums/TEST_ITEMS))
66
```

```
training_data, test_data = LoadMNIST()          # 加载 Mnist 数据
print("朴素 SVM 调参")
print("_____________________________________________________")
print("Begin training SVM with linear...")
svm_module_3 = svm.SVC(kernel="linear", probability=True) # 线性核函数
param_grid_linear = {'C': [1e-2, 1e-1, 1, 10, 100]}
grid_search_linear = GridSearchCV(svm_module_3, param_grid_linear, n_jobs =
    -1, verbose = 1)
Test(grid_search_linear)
print(grid_search_linear.best_params_)
print("_____________________________________________________")
print("Begin training SVM with RBF...")
svm_module_1 = svm.SVC(kernel="rbf", probability=True) # RBF 核函数
param_grid_rbf = {'C': [1e-2, 1e-1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1,
    1.0]}
grid_search_rbf = GridSearchCV(svm_module_1, param_grid_rbf, n_jobs = -1,
    verbose = 1)
Test(grid_search_rbf)
print(grid_search_rbf.best_params_)
print("_____________________________________________________")
print("Begin training SVM with poly...")
svm_module_2 = svm.SVC(kernel="poly", probability=True) # 多项式核函数
param_grid_poly = {'C': [1e-2, 1e-1, 1, 10, 100], 'gamma': [0.1, 0.01, 1.0],
    'degree': [1, 3, 5, 7]}
grid_search_poly = GridSearchCV(svm_module_2, param_grid_poly, n_jobs = -1,
    verbose = 1)
Test(grid_search_poly)
print(grid_search_poly.best_params_)
print("_____________________________________________________")
print("Begin training SVM with sigmoid...")
svm_module_4 = svm.SVC(kernel="sigmoid", probability=True) # 线性核函数
param_grid_sigmoid = {'C': [1e-2, 1e-1, 1, 10, 100], 'gamma': [1, 2, 3, 4], '
    coef0': [0.2, 0.4, 0.6, 0.8, 1 ]}
grid_search_sigmoid = GridSearchCV(svm_module_4, param_grid_sigmoid, n_jobs =
     -1, verbose = 1)
Test(grid_search_sigmoid)
print(grid_search_sigmoid.best_params_)
print("_____________________________________________________")
print("SVM + Adaboosting")
base_rbf = svm.SVC(kernel="rbf", probability=True, C=100, gamma=0.01)

param_grid = {
```

```
    'n_estimators': [50, 100],
    'learning_rate': [1.0, 0.5],
}
print("Begin training Adaboosting with RBF SVM...")
model_rbf = AdaBoostClassifier(base_estimator=base_rbf)
grid_search_rbf_ada = GridSearchCV(model_rbf, param_grid, n_jobs = -1,
    verbose = 1)
Test(grid_search_rbf_ada)
print(grid_search_rbf_ada.best_params_)
```