哈尔滨工业大学

模式识别与深度学习

实验一：利用 sklearn 实现数字手写体识别

姓　　名：刘天瑞

院　（系）：未来技术学院

专　　业：视听觉信息处理

学　　号：7203610121

指导教师：金野

提交日期：2023.5.1

目 录

一、 课题来源及研究的目的和意义

1.1 课题来源

阿拉伯数字的手写体识别是一种人工智能领域的应用，它的来源可以追溯到数字识别技术的发展历程。数字识别技术的出现可以追溯到上世纪 60 年代末期，当时研究人员开始探索使用计算机来识别手写数字的方法。

这项技术最初是应用于邮政编码的自动识别上，后来逐渐被应用于其他领域，比如银行支票处理、身份证号码识别、手写数字签名认证等。

随着科技的不断进步和人工智能的发展，手写体的数字识别技术也在不断提升和完善。如今，阿拉伯数字的手写体识别已经广泛应用于各种场景，比如智能手机的解锁、数码笔记本的手写输入、自动识别银行支票、快递单号等等。

1.2 研究目的

为了进一步地掌握建立机器学习中模式识别模型的流程,理解模式识别的基本概念,此次实验我选择针对 sklearn 自带的 digits 数据集（手写字体识别数据集）进行分类预测。通过实现数字手写体识别系统，并采用 sklearn 中的机器学习算法和工具，可以提高识别的准确性和精度。这有助于应用场景中提高数据处理的自动化程度和准确性。

数字手写体的特征和规律是数字手写体识别的重要基础。通过实现数字手写体识别系统，可以深入分析数字手写体的特征和规律，进而研究手写体识别的相关问题，如数据增强、特征提取等。数字手写体识别是机器学习技术在实际应用中的一个典型案例。通过实现数字手写体识别系统，可以探索机器学习技术在实际应用中的应用场景，如图像识别、语音识别、自然语言处理等，为实际应用提供参考和支持。

在此次实验中,我主要选用了 KNN（K 邻近算法）、MLP（多层感知机）、Random Forest（随机森林）、Adaboost（自适应增强算法）、SVM（支持向量机）、Decision Tree（决策树）、Logistic Regression（逻辑回归）这七种机器学习算法模型进行数字手写字体的模式识别。通过实践和实验，可以比较各种算法和模型的优缺点，研究它们的特性和适用场景，从而对机器学习的理论和实践有更深入的认识。

本次模式识别实验的实验环境为 Windows 11；Visual Studio 2022；编程语言为 Python3.9.7，并且使用其中的 Scikit-learn 库。经测试，各种算法模型在安装了 python3.9 的 VS 软件中均可以正常运行

1.3 研究意义

实现数字的手写体体识别是一项经典的机器学习模式识别任务，通过实现它的功能可以帮助我深入了解机器学习的基本原理和算法，并掌握模式识别的实践技能。

sklearn 是 Python 中最流行的机器学习库之一，它提供了丰富的机器学习算法和工具，可以帮助我快速地实现数字手写体识别系统，并进行数据预处理、特征提取、模型训练和评估等操作。

数字手写体识别实验是一项非常具有实际应用价值的任务，它可以帮助实现自动化数据处理、自动识别文本信息、自动化图像处理等多种应用场景，从而提高工作效率、节省人力资源。通过实现数字手写体识别，我还逐渐地掌握了机器学习的调参技巧，例如该怎样选择合适的模型、优化参数、调整超参数等等，这些技巧在实际的模式识别应用中非常重要。

总之，基于 sklearn 库来实现数字（digits）手写体识别不仅可以帮助我提高模式识别的实践技能和理论知识，还可以为实际应用场景提供有力的支持。

二、 主要研究内容以及拟解决的关键问题

2.1 digits 手写体数据集的查看

首先我将 sklearn 库中的 digits 数据集导入到实验项目中，其中 X 为 1797*64 大小的二维数组，代表 digits 数据集中 1797 个样本，其中每一个样本手写体图片均为一维向量，其特征维度为 64，每个像素点代表 1 维的特征，其取值范围为 0-16；y 为样本所对应的标签值，其共有 10 个手写体类别：取值范围为 0-9 中的任意一个数字。将数据集导入到实验项目中的代码，如下图 1 所示：

```
13    from sklearn.datasets import load_digits
14
15
16    digits = load_digits()
17    X = digits.data
18    y = digits.target
```

图 1

再者如果要实现可视化查看数据集，则需要将样本手写体图片转换为二维向量，然后再借助 matplotlib 包里的 pyplot 来实现升维，以数据集中的第 1791 个

样本 4 为例，具体的实现方式如下图 3 所示：

```
def show(image):
    test = image.reshape((8, 8))  # 从一维升至二维，这样才能被显示出来，此处以数字4为例
    print(test.shape)  # 查看是否为二维数组
    # print(test)
    plt.imshow(test)  # 显示灰色图像
    plt.show()
```

```
if __name__ == '__main__':
    # 查看数据集，以图片显示
    print(X.shape, y.shape)
    print(X, y)
    show(X[1791]*15)
    print(y[1791])
```

图 2

数字 4 的样本图片如下图 3 所示：

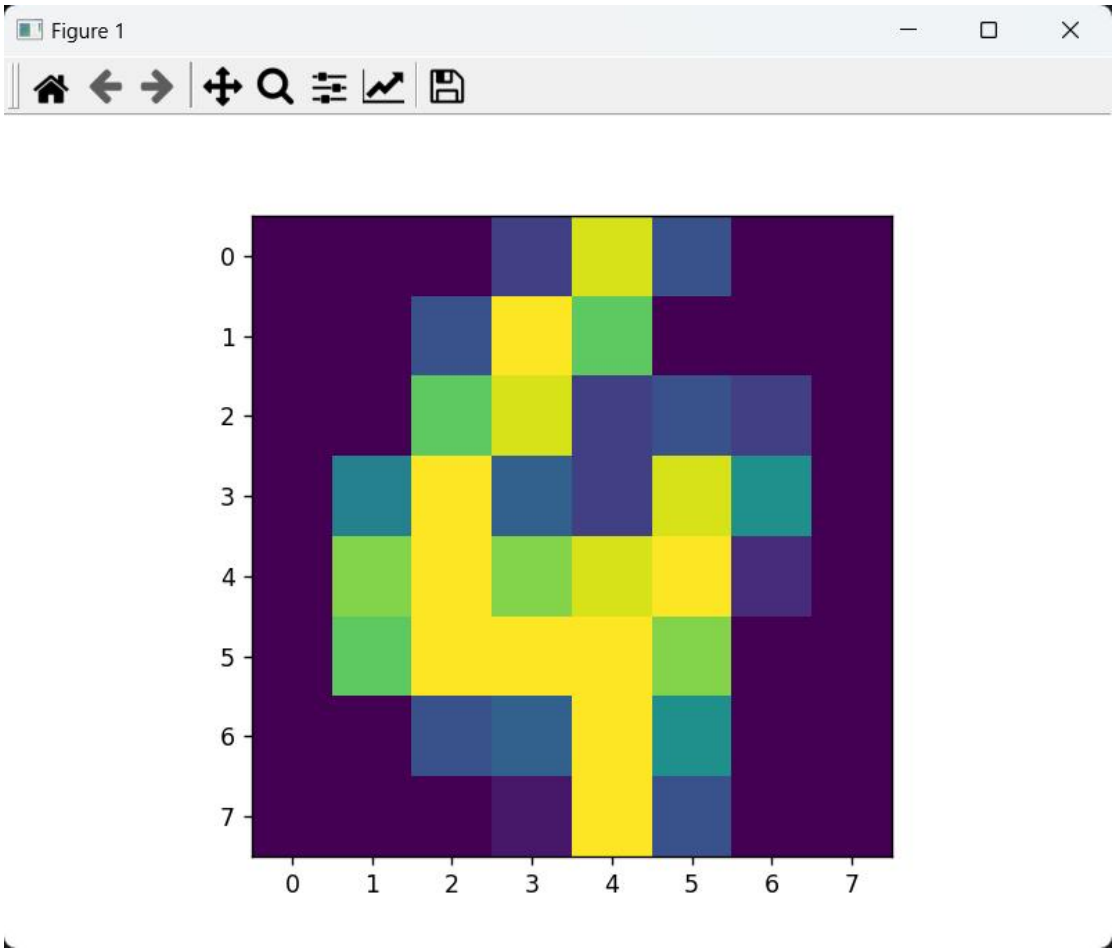


图 3

考虑到在后续算法模型的建立过程中，需要用到的样本是一维向量，故 show()

函数只用来对部分数据集进行查看和检验，并不将数据集全部进行转换。

2.2 digits 手写体数据集的预处理

2.2.1 特征标准化

调用 sklearn 库中对数据集进行标准化处理的函数 StandardScaler()来处理样本手写体图片 X，其特征标准化的过程如下图 4 所示：

```
from sklearn.preprocessing import StandardScaler
```

```
def standard_demo(data):
    transfer = StandardScaler()
    data_new = transfer.fit_transform(data)
    print(data_new)
    return data_new
```

图 4

2.2.2 特征降维

考虑到样本手写体图片中的特征维度为 64 维，其中有可能包含一些相邻近的维度或者噪音等等，这些维度或者噪音会影响算法模型的性能和准确度，容易导致产生过拟合等现象，因此在数据清理时，我需要对样本进行特征降维。实现降维的具体代码如下图 5 所示：

```
from sklearn.decomposition import PCA
```

```
def pca_demo(data):
    transfer = PCA(n_components=0.92)
    data_new = transfer.fit_transform(data)
    print(data_new)
    return data_new
```

```
# 数据集的预处理（包括特征标准化、特征降维）
X_new = standard_demo(X)
X_new = pca_demo(X_new)
```

图 5

其中在进行降维的过程中，保留原本数据集大小的百分比对与算法的准确率具有较大的影响，此次实验中我以 KNN 算法为例，百分比从 84%一直调高到 96%，算得对应 PCA 降维之后算法的准确率在稳步升高，由于这是在调参之后测出的

结果，所以在四舍五入之后，它们之间并没有很明显的差距，但是在保留原数据集至少 92%的情况下，准确率依然很高并且不明显提高，故此次实验的降维参数设置为 n_components=0.92，特征维度降为 34 维，并且由此可见，数据集中的确存在大量的噪音或比较接近的特征维度。

三、详细的建模研究方案

数据集已经准备完毕，下一步就是建立机器学习算法模型，对模型进行训练以及测试。

3.1 KNN 算法

我第一个所选择的算法模型为 KNN 算法，其原理比较简单，对于我选择的这种小数据集来说，理论上应该会有不错的准确度表现，KNN 模型构建如下图 6 所示：

```
from sklearn.model_selection import train_test_split
```

```
from sklearn.neighbors import KNeighborsClassifier
```

```
def knn_demo(data, label):
    # 划分数据集
    X_train, X_test, y_train, y_test = train_test_split(data, label, random_state = 6)
    # 训练模型
    estimate = KNeighborsClassifier(n_neighbors = 10)
    estimate.fit(X_train, y_train)  # 模型构建完毕
```

```
knn_demo(X_new, y)  # 数据已经准备完毕
```

图 6

在模型构建之前，首先需要调用 train_test_split()函数对数据集进行划分，这里采用了数据集默认的划分方式，即训练集：测试集=3 ： 1，其次再调用 KNeighborsClassifier()函数构建模型，参数 n_neighbors=10 代表对每一个样本点进行分类时，都要找到离他最近的 10 个“邻居”点。

```
准确率为：
 0.9755555555555555
```

3.2 MLP 算法

我第二个选择的算法模型为人工神经网络中的全连接网络，即多层感知机算

法(MLP)，MLP 模型构建如下图 7 所示：

```
from sklearn.neural_network import MLPClassifier
```

```
def code_demo(data, label):
    numFiles = len(data)
    hwLabels = np.zeros([numFiles, 10])  # 用于存放对应的独热标签
    for i in range(numFiles):  # 遍历所有的样本手写体图片
        digit = label[i]
        hwLabels[i][digit] = 1.0  # 将对应的独热标签设置为1
    return hwLabels
```

```
def MLP_demo(data, label):
    # 对标签进行独热编码
    label = code_demo(data, label)
    # print(label)
    # # 独热编码逆转回数字标签   argmax返回列表最大值索引
    # label = label.argmax(axis = 1)
    # print(label)
    # 划分数据集
    X_train, X_test, y_train, y_test = train_test_split(data, label, random_state = 6)
    # 训练模型
    estimate = MLPClassifier(hidden_layer_sizes = (100,),
                             activation = 'relu', solver = 'lbfgs',
                             learning_rate_init = 0.001, max_iter = 2000)
    estimate.fit(X_train, y_train)  # 模型构建完毕
```

```
MLP_demo(X_new, y)  # 数据已经准备完毕
```

图 7

此次实验中由于 MLP 算法的参数比较多，并且各种参数对算法的性能影响较大，因此需要对模型的各种参数进行调节从而寻求最优越的分类模型，分别对 MLP 模型中的隐藏层、激活函数、权重优化器、学习率以及迭代次数进行调节。

隐藏层的变化对于模型性能的影响很小，随着隐藏层及神经元的增加，对于硬件设施的要求也高，而且运行效率低，因此，为了一点性能的提升，加大硬件设施的负担，性价比并不高，因此我最终选择了一层隐藏层，神经元数量为 100，也可以达到较为不错的效果。

relu 激活函数在此次实验中具有较为不错的表现，可以提升模型的准确率，因此本算法模型最终选用 relu 作为激活函数；lbfgs 权重优化器更适合此次数字手写字体数据集的识别，因此本算法模型选择 lbfgs 作为权重函数优化器；学习率为 0.001 时不仅算法模型的准确率高，对硬件设施的负担也较小，故此次 MLP 模型选择 0.001 为学习率参数；而从性价比方面考虑，选择 2000 作为最大迭代次数更为优越。

在对比 MLP 的一系列参数之后，构建模型的算法准确率约为 94%，但是如果不对标签数据进行独热编码，使用 MLP 模型可以得到约 96%的准确率，这也是我此次实验疑惑的一处，后续会在总结时进一步分析原因。

```
准确率为：
 0.9377777777777778
```

3.3 RandomForest 算法

随机森林是集成学习算法 bagging 系列的代表算法之一，它使得几个弱监督分类器集中起来，最后拥有强监督器的分类能力。除此之外，随机森林属于并行式的分类算法，在多分类问题上采用各个分类器投票表决的模式，因此理论上来看，它在此次数据集的识别上应该会有比较不错的表现。RandomForest 模型构建如下图 8 所示：

```
from sklearn.ensemble import RandomForestClassifier
```

```
def randomforest_demo(data, label):
    # 划分数据集
    X_train, X_test, y_train, y_test = train_test_split(data, label, random_state = 6)
    # 训练模型
    estimate = RandomForestClassifier(n_estimators = 100, criterion = 'gini', max_depth = 20000)
    estimate.fit(X_train, y_train)  # 模型构建完毕
```

```
randomforest_demo(X_new, y)  # 数据已经准备完毕
```

图 8

该模型的性能主要受到随机森林中树木的最大深度、树木的数量以及节点分裂函数的影响，对这两个参数进行反复调整，最终确定树最大深度设置为 20000，树木数量为 100，节点分裂函数选择 gini。

```
准确率为：
 0.9577777777777777
```

3.4 Adaboost 算法

Adaboost 是集成学习算法 boost 系列的代表算法之一，与随机森林算法不同的是，它属于串行式的分类算法，在二分类问题上具有极大的优越性，但在多分类问题上，还需要进一步的测试，Adaboost 模型构建如下图 9 所示：

```
from sklearn.ensemble import AdaBoostClassifier
```

```
def Adaboost_demo(data, label):
    # 划分数据集
    X_train, X_test, y_train, y_test = train_test_split(data, label, random_state = 6)
    # 训练模型
    estimate = AdaBoostClassifier(n_estimators = 100, learning_rate = 0.01)
    estimate.fit(X_train, y_train)  # 模型构建完毕
```

```
Adaboost_demo(X_new, y)  # 数据已经准备完毕
```

图 9

该模型的性能主要受到森林中树木数量以及学习率的影响，对这两个参数进行反复调整，Adaboost 算法模型在此次实验中的表现比较差，不过也可以理解，根据没有免费的午餐定理，Adaboost 在二分类问题上的优越性也决定了它在多分类问题上的局限性，故该算法不太适合处理多分类问题。最终确定树木数量设置为 100，学习率为 0.01。

```
准确率为：
 0.6288888888888889
```

3.5 SVM 算法

SVM 算法在处理二分类问题上具有比较优越的性能，其在多分类问题上的表现还需进一步的测试，SVM 模型构建如下图 10 所示：

```
from sklearn import svm
```

```
def svm_demo(data, label):
    # 划分数据集
    X_train, X_test, y_train, y_test = train_test_split(data, label, random_state = 2)
    # 训练模型
    estimate = svm.SVC()
    estimate.fit(X_train, y_train)  # 模型构建完毕
```

```
svm_demo(X_new, y)  # 数据已经准备完毕
```

图 10

SVM 算法的默认多分类方法为 ovr，即多类聚为一类对另一类，把多分类问题转换为二分类问题，最后再通过投票否决进行分类。虽然这样做可能会使得数据集分布不均衡进而影响到分类模型的性能，并且由于 SVM 在二分类上的优越性，我认为它在多分类上应该不会有很好的表现。但是出乎意料，SVM 模型在多分类问题上依然有着很高的性能表现。

3.6 DecisionTree 算法

决策树模型的构建如下图 11 所示：

```
from sklearn.tree import DecisionTreeClassifier
```

```
def DecisionTree_demo(data, label):
    # 划分数据集
    X_train, X_test, y_train, y_test = train_test_split(data, label, random_state = 6)
    # 训练模型
    estimate = DecisionTreeClassifier(criterion = 'entropy', splitter = 'best', max_depth = 200)
    estimate.fit(X_train, y_train)  # 模型构建完毕
```

```
DecisionTree_demo(X_new, y)  # 数据已经准备完毕
```

图 11

决策树模型的性能主要受到森林中的节点分裂函数、树木的最大深度以及节点拆分策略的影响，对这几个参数进行调整，最终确定节点分裂函数选择 entropy，树木的最大深度为 200 以及节点拆分策略为 best。

3.7 Logistic Regression 算法

逻辑回归算法是处理分类问题的算法，尤其在处理二分类问题方面表现不错，在处理多分类问题方面主要有两种方式，一种是 SVM 算法默认的“ovr”方法，另一种是“multinomial”方法。Logistic Regression 模型构建如下图 12 所示：

```
from sklearn.linear_model import LogisticRegression
```

```
def LogisticRegression_demo(data, label):
    # 划分数据集
    X_train, X_test, y_train, y_test = train_test_split(data, label, random_state = 6)
    # 训练模型
    estimate = LogisticRegression(multi_class='multinomial', solver = 'lbfgs', max_iter = 10000)  # 可以选择多分类方法 ovr 和 multinomial
    estimate.fit(X_train, y_train)  # 模型构建完毕
```

```
LogisticRegression_demo(X_new, y)  # 数据已经准备完毕
```

图 12

Logistic Regression 模型主要受到采用何种多分类方法以及权重优化器的影响，从运行时长以及算法准确率两个方面综合来考虑，该模型应当选择“multinational”方法来处理数字手写字体识别问题，权重优化器则应选择’lbfgs’。

```
准确率为：
 0.9422222222222222
```

四、实验分析与结论

4.1 绘制各类模型的学习曲线

经过对上述各类机器学习算法的讨论以及各种参数的调整，最终确定模式识别算法参数以及在测试集上的准确率，这其中 SVM 和 KNN 算法模型在处理该手写数字识别问题数据集时，性能比较优越；而 Adaboost 算法则不能很好地处理这种多分类问题，在测试集上地准确率较低；决策树算法由于本身特点在训练集上可能出现了过拟合的现象，导致其测试集上的准确率并不突出，调整参数后准确率依然达不到 90%以上；而其他各类算法的准确率均能保持在 90%以上。

其次，学习曲线指的是随着训练样本的增加，模型在训练集上的准确率与交叉验证得到的准确率之间的变化，画出各类算法模型各自的学习曲线，实现具体代码如下图 13 所示：

```
# 绘制学习曲线
cv = ShuffleSplit(n_splits = 10, test_size = 0.2, random_state = 0)  # 10折交叉验证
fig, ax = plt.subplots(1, 1, figsize = (10, 6), dpi = 144)
plot_learning_curve(ax, estimate, "Learn Curve",
                    X_train, y_train, ylim = (0.0, 1.01), cv = cv)
plt.show()
```

图 13

此次绘制曲线中，采用的是 10 折交叉验证的方法，训练集与验证集的比例被划分为 4：1，最终绘制出的曲线图像如下图 14 所示：

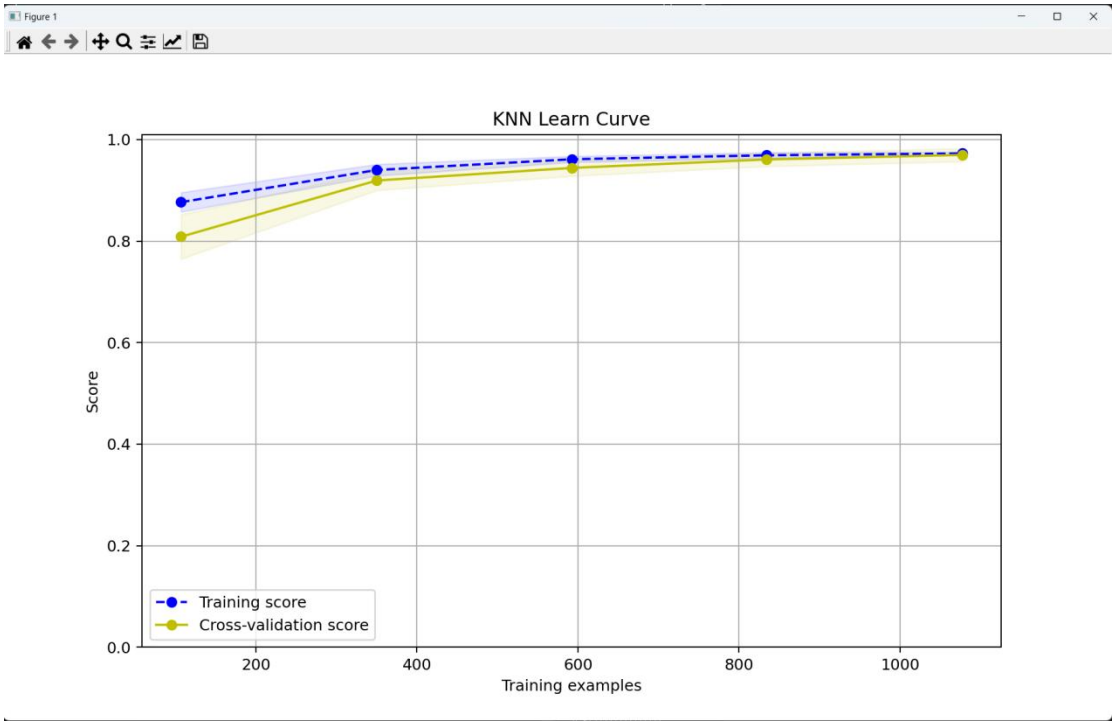


Figure 1
KNN Learn Curve
Score
Training examples
Training score
Cross-validation score
1.0
0.8
0.6
0.4
0.2
0.0
200
400
600
800
1000

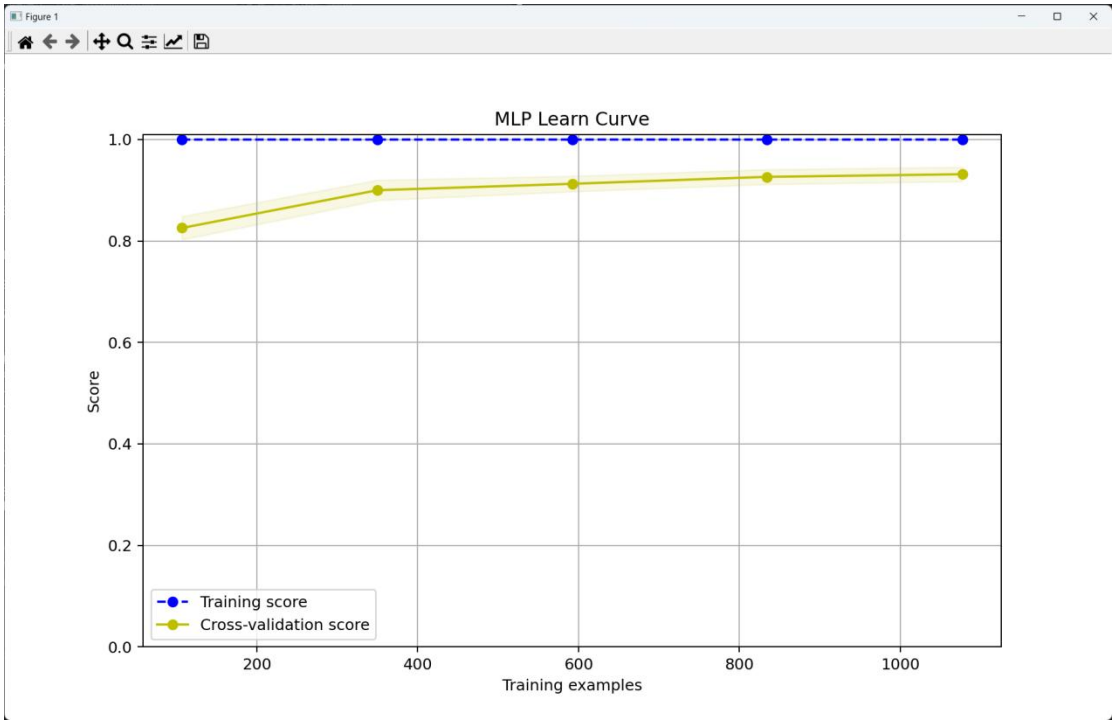


Figure 1
MLP Learn Curve
Score
Training examples
Training score
Cross-validation score
1.0
0.8
0.6
0.4
0.2
0.0
200
400
600
800
1000

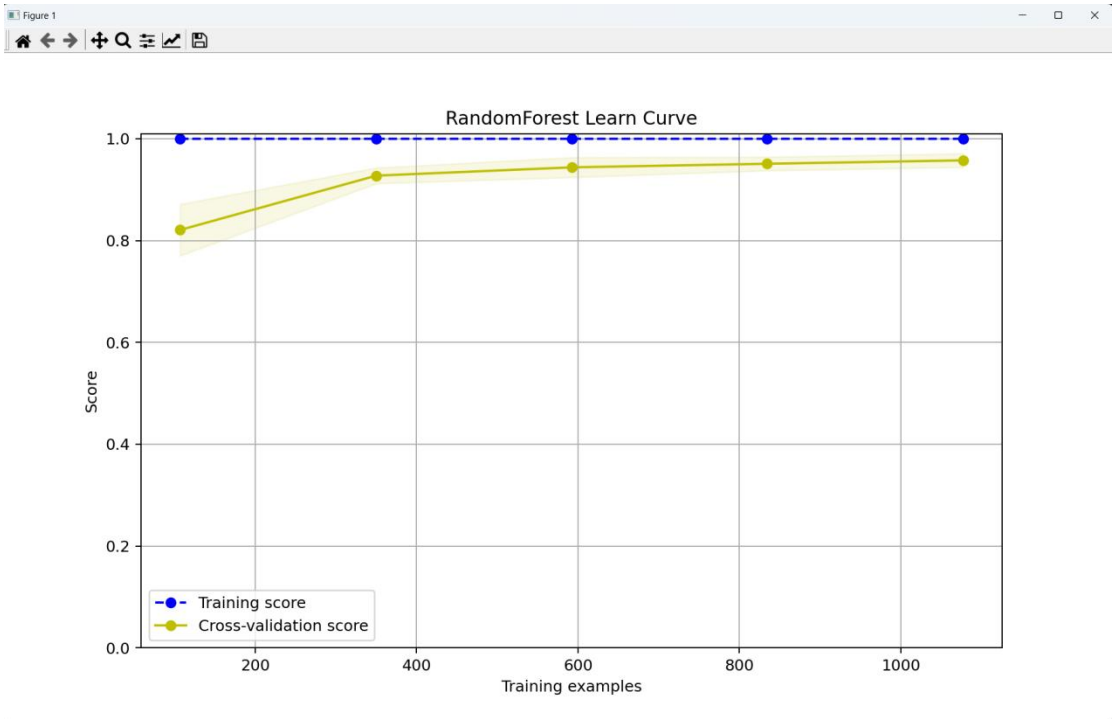


Figure 1
RandomForest Learn Curve
Score
Training examples
Training score
Cross-validation score
1.0
0.8
0.6
0.4
0.2
0.0
200
400
600
800
1000

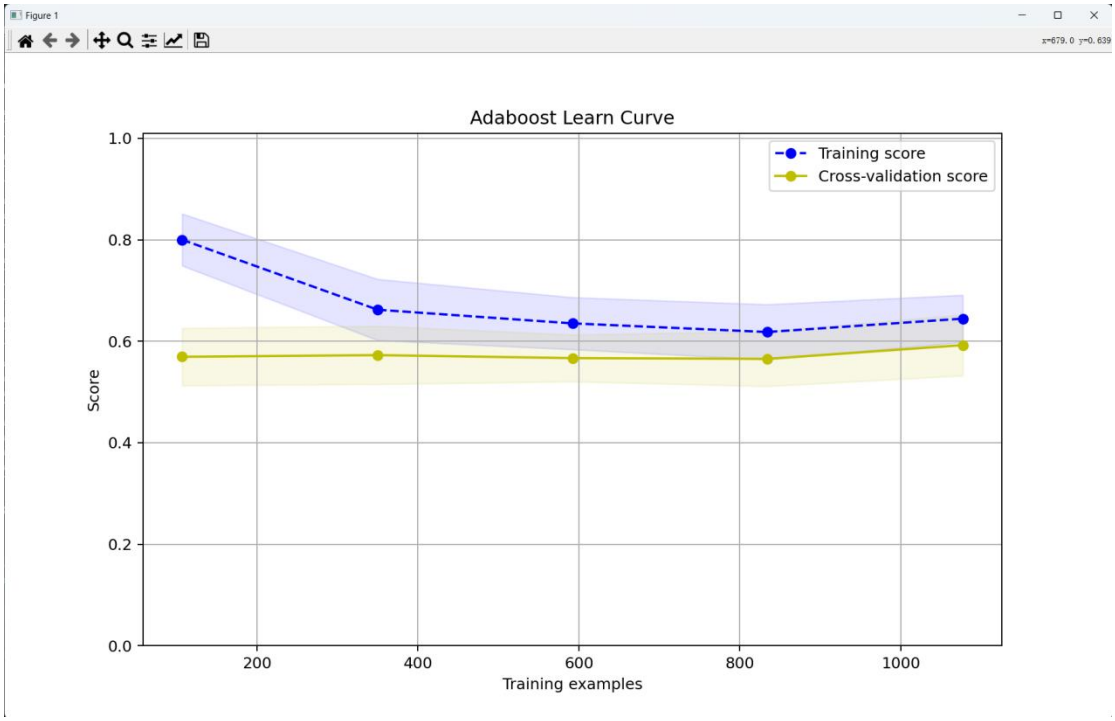


Figure 1
Adaboost Learn Curve
Score
Training examples
Training score
Cross-validation score
1.0
0.8
0.6
0.4
0.2
0.0
200
400
600
800
1000

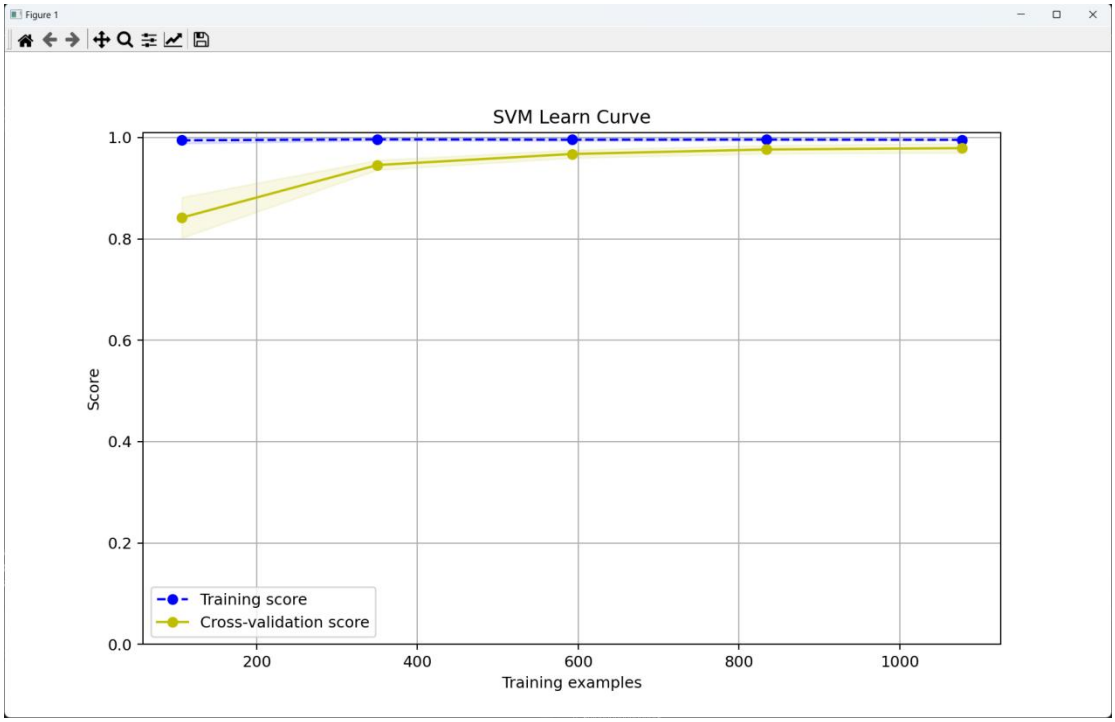


Figure 1
SVM Learn Curve
Score
Training examples
Training score
Cross-validation score
1.0
0.8
0.6
0.4
0.2
0.0
200
400
600
800
1000

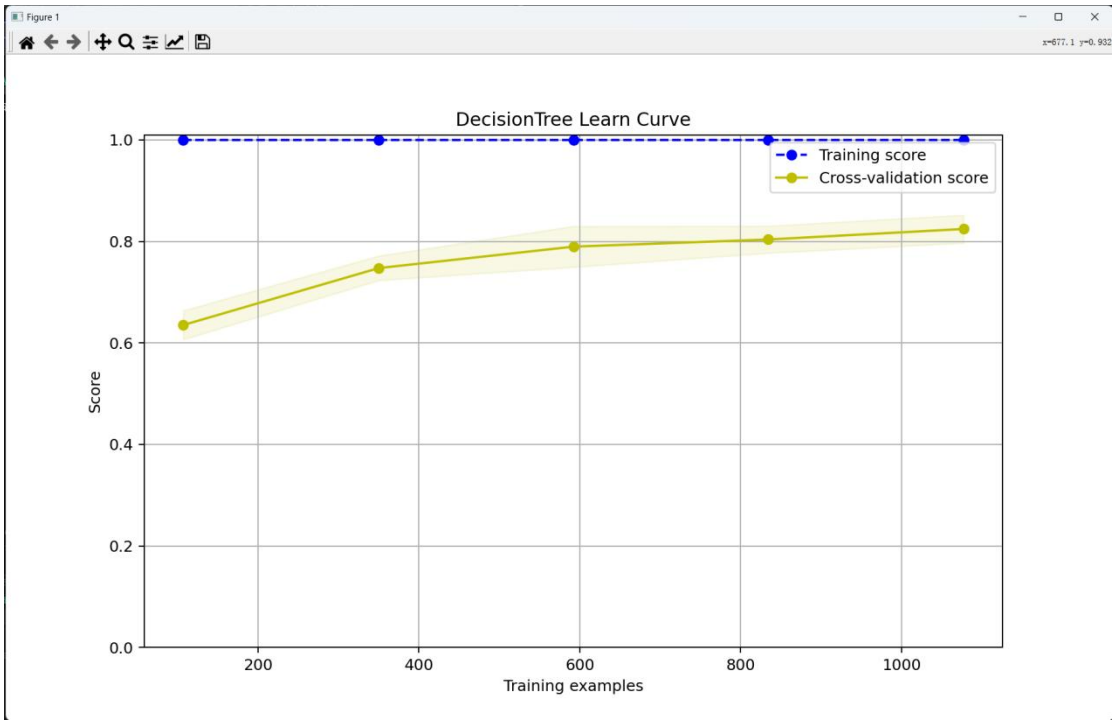


Figure 1
DecisionTree Learn Curve
Score
Training examples
Training score
Cross-validation score
1.0
0.8
0.6
0.4
0.2
0.0
200
400
600
800
1000

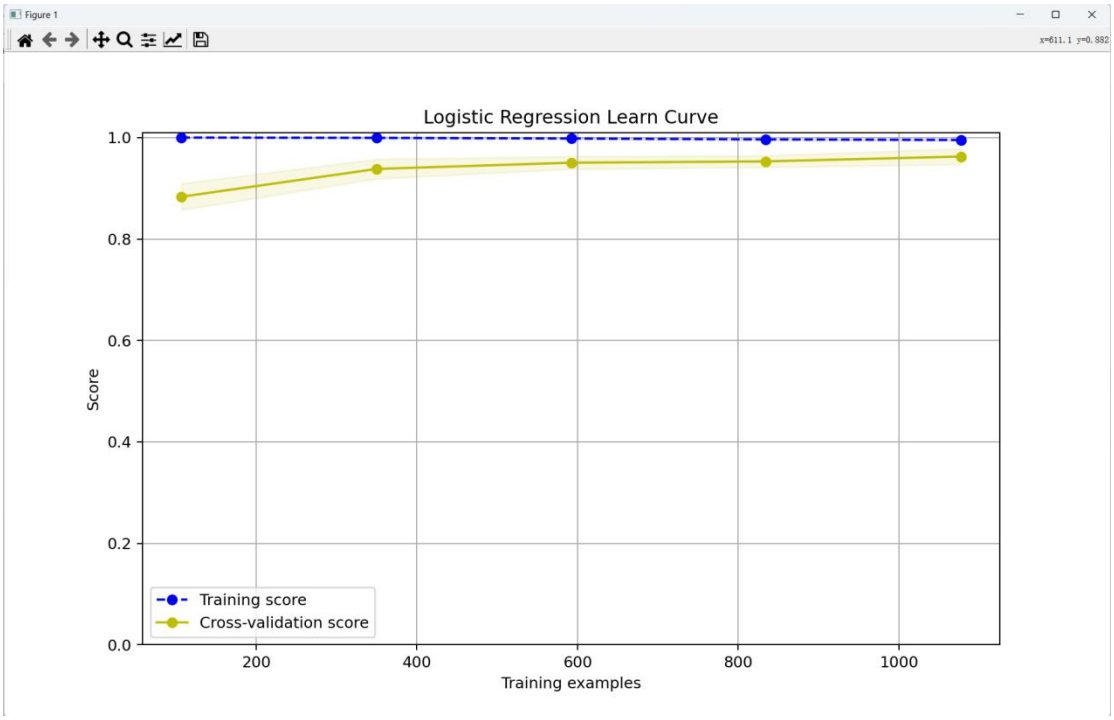


图 14

从学习曲线也可以直观地看出 KNN 与 SVM 算法的性能更为优越，决策树存在过拟合现象，Adaboost 算法存在欠拟合现象，其他各类算法表现均较为良好。

4.2 衡量模型的性能

根据上述结果已经可以确定哪些模型在此实验中具有更好的表现，哪些模型不太适合此次手写体多分类实验，但是为了更进一步地表示出算法在样本上的估计率以及针对每一类别它们的预测效果如何，本次实验我还采用了可视化的混淆矩阵和每一类别的精确率以及召回率来计算结果，从而继续对实验结果进行分析。可视化混淆矩阵以及计算精确率和召回率的代码实现方式如下图 15 所示：

```
from sklearn.metrics import confusion_matrix

import seaborn as sn
```

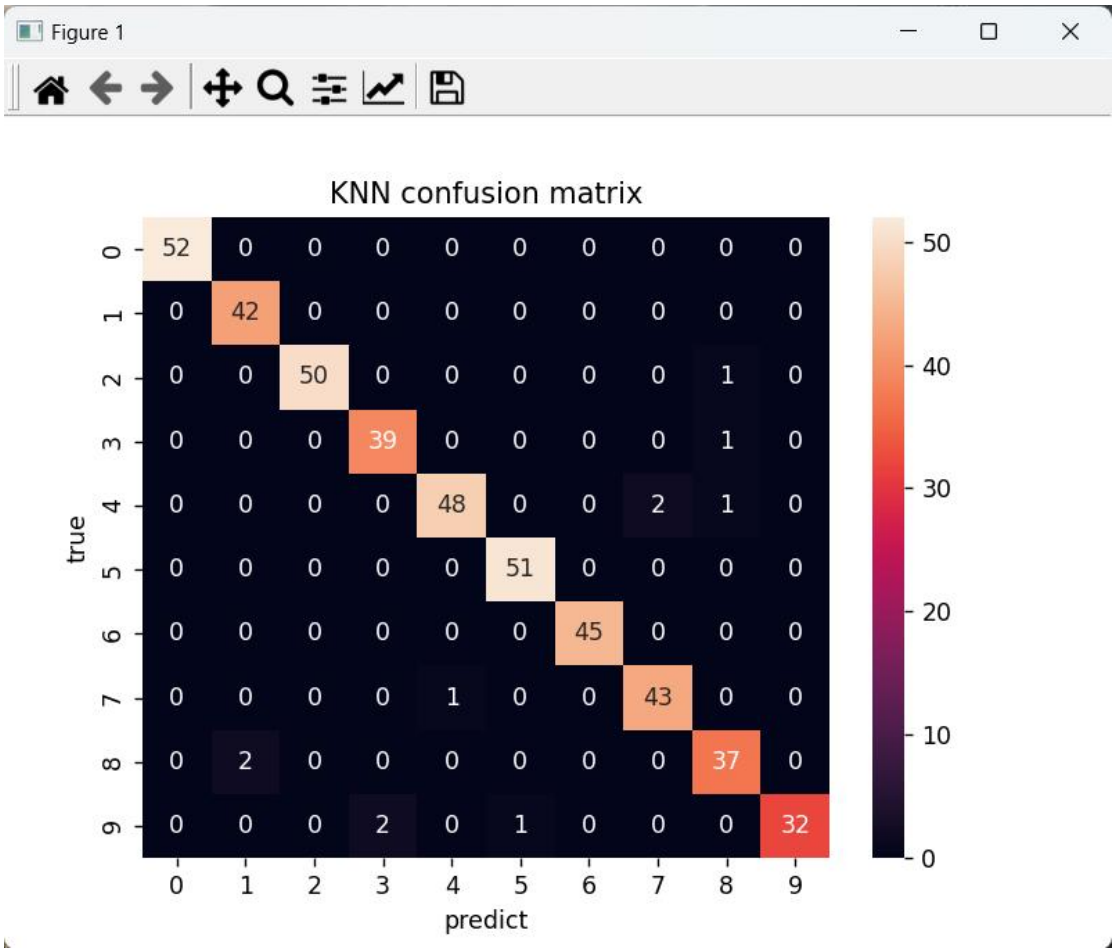
```
# 混淆矩阵
cm = confusion_matrix(y_test, y_predict)
print(cm)
# 可视化显示混淆矩阵
# annot = True 显示数字 ，fmt参数不使用科学计数法进行显示
ax = sn.heatmap(cm, annot = True, fmt = '.20g')
ax.set_title('confusion matrix')  # 标题
ax.set_xlabel('predict')  # x轴
ax.set_ylabel('true')  # y轴
plt.show()
```

```
# 计算精确率与召回率
report = classification_report(y_test, y_predict, labels = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
                               target_names = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'])
print(report)
```

图 15

在绘制各类算法的混淆矩阵时，我并没有借助 matplotlib 来实现，而是采用了 seaborn 库来实现，它是对 matplotlib 进行二次封装实现的，故操作更为简单。在计算精确率与召回率的过程中，我将数字范围 0-9 分为了 10 个类别，从而更为形象直观。各类算法的混淆矩阵以及精确率和召回率的计算结果如下图 16 所示：

混淆矩阵：



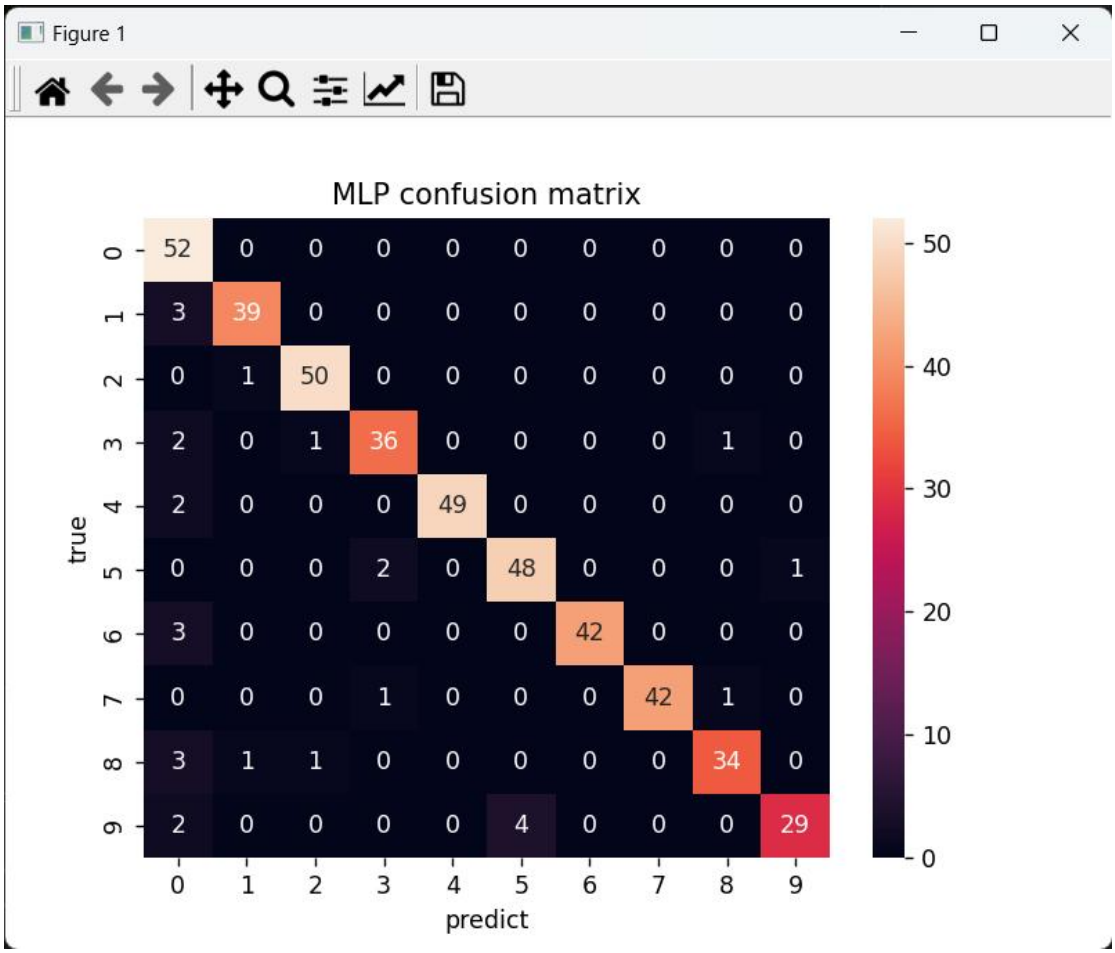


Figure 1
MLP confusion matrix
true
predict
0 1 2 3 4 5 6 7 8 9
52 0 0 0 0 0 0 0 0 0
3 39 0 0 0 0 0 0 0 0
0 1 50 0 0 0 0 0 0 0
2 0 1 36 0 0 0 0 1 0
2 0 0 0 49 0 0 0 0 0
0 0 0 2 0 48 0 0 0 1
3 0 0 0 0 0 42 0 0 0
0 0 0 1 0 0 0 42 1 0
3 1 1 0 0 0 0 0 34 0
2 0 0 0 0 4 0 0 0 29
50 40 30 20 10 0

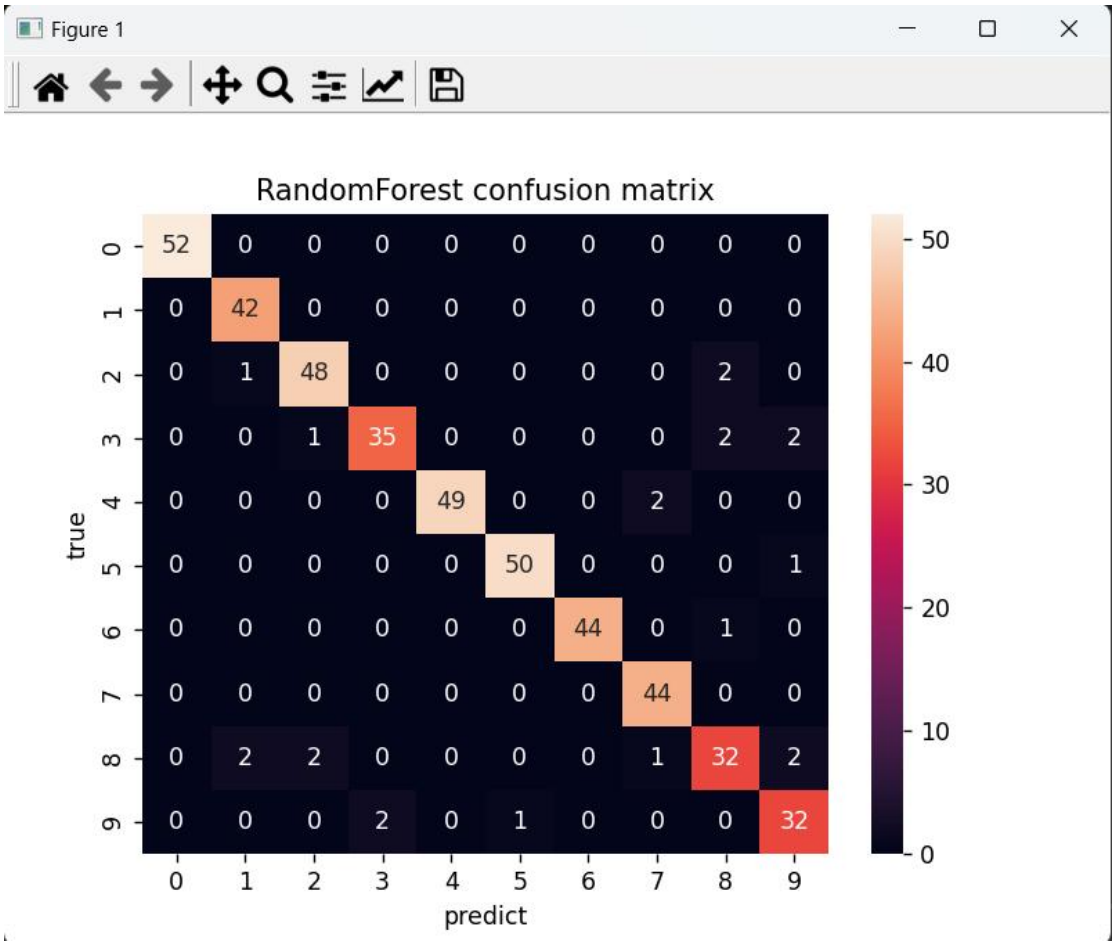


Figure 1
RandomForest confusion matrix
true
predict
0 1 2 3 4 5 6 7 8 9
52 0 0 0 0 0 0 0 0 0
0 42 0 0 0 0 0 0 0 0
0 1 48 0 0 0 0 0 2 0
0 0 1 35 0 0 0 0 2 2
0 0 0 0 49 0 0 2 0 0
0 0 0 0 0 50 0 0 0 1
0 0 0 0 0 0 44 0 1 0
0 0 0 0 0 0 0 44 0 0
0 2 2 0 0 0 0 1 32 2
0 0 0 2 0 1 0 0 0 32
50 40 30 20 10 0

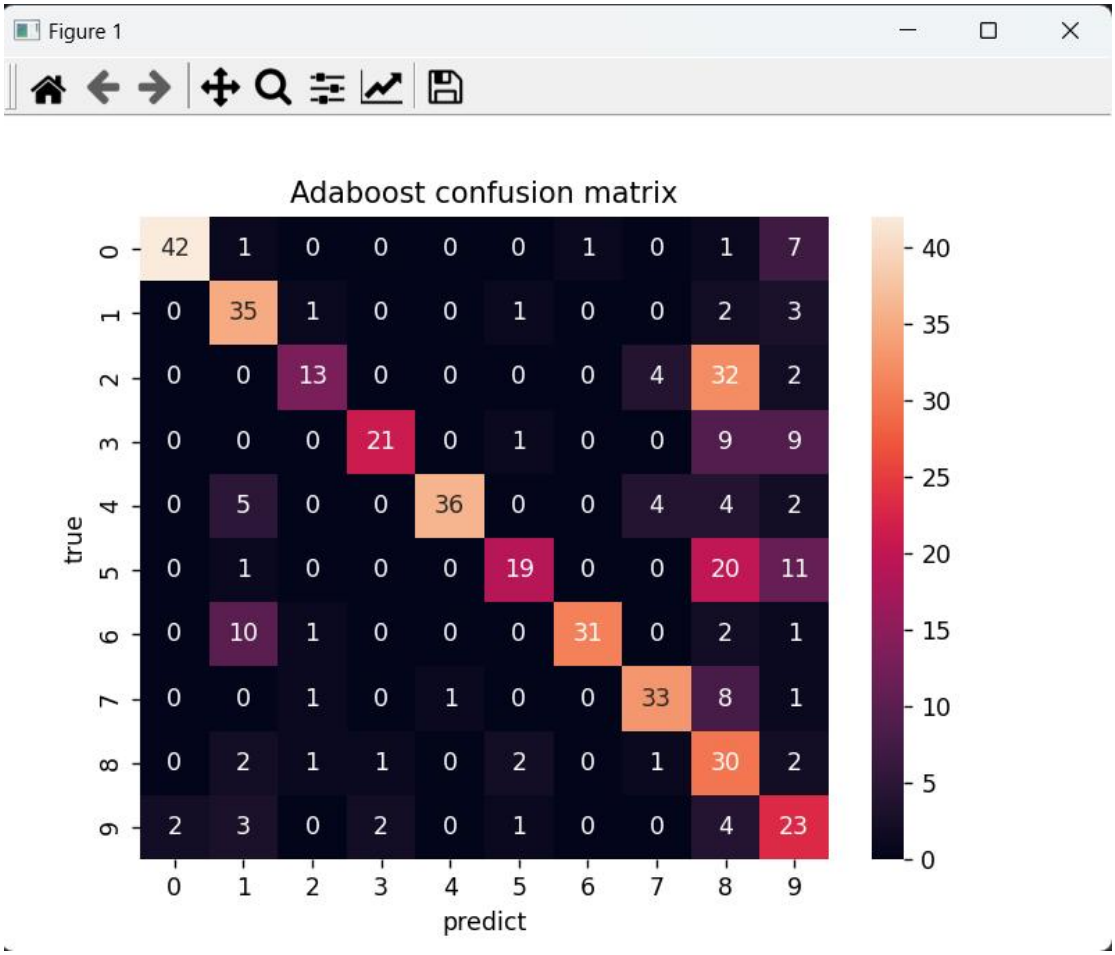


Figure 1
Adaboost confusion matrix
true
predict
0 1 2 3 4 5 6 7 8 9
42 1 0 0 0 0 1 0 1 7
0 35 1 0 0 1 0 0 2 3
0 0 13 0 0 0 0 4 32 2
0 0 0 21 0 1 0 0 9 9
0 5 0 0 36 0 0 4 4 2
0 1 0 0 0 19 0 0 20 11
0 10 1 0 0 0 31 0 2 1
0 0 1 0 1 0 0 33 8 1
0 2 1 1 0 2 0 1 30 2
2 3 0 2 0 1 0 0 4 23
40 35 30 25 20 15 10 5 0

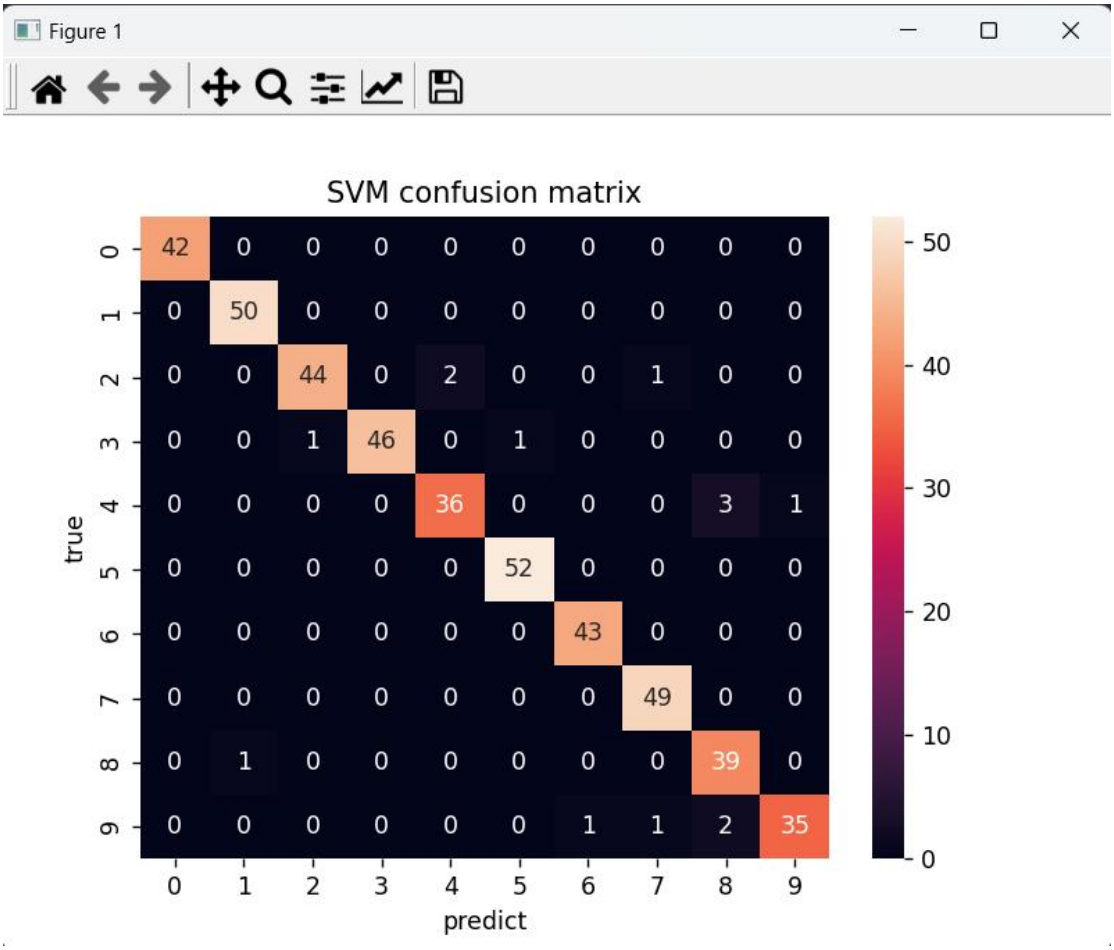


Figure 1
SVM confusion matrix
true
predict
0 1 2 3 4 5 6 7 8 9
42 0 0 0 0 0 0 0 0 0
0 50 0 0 0 0 0 0 0 0
0 0 44 0 2 0 0 1 0 0
0 0 1 46 0 1 0 0 0 0
0 0 0 0 36 0 0 0 3 1
0 0 0 0 0 52 0 0 0 0
0 0 0 0 0 0 43 0 0 0
0 0 0 0 0 0 0 49 0 0
0 1 0 0 0 0 0 0 39 0
0 0 0 0 0 0 1 1 2 35
50 40 30 20 10 0

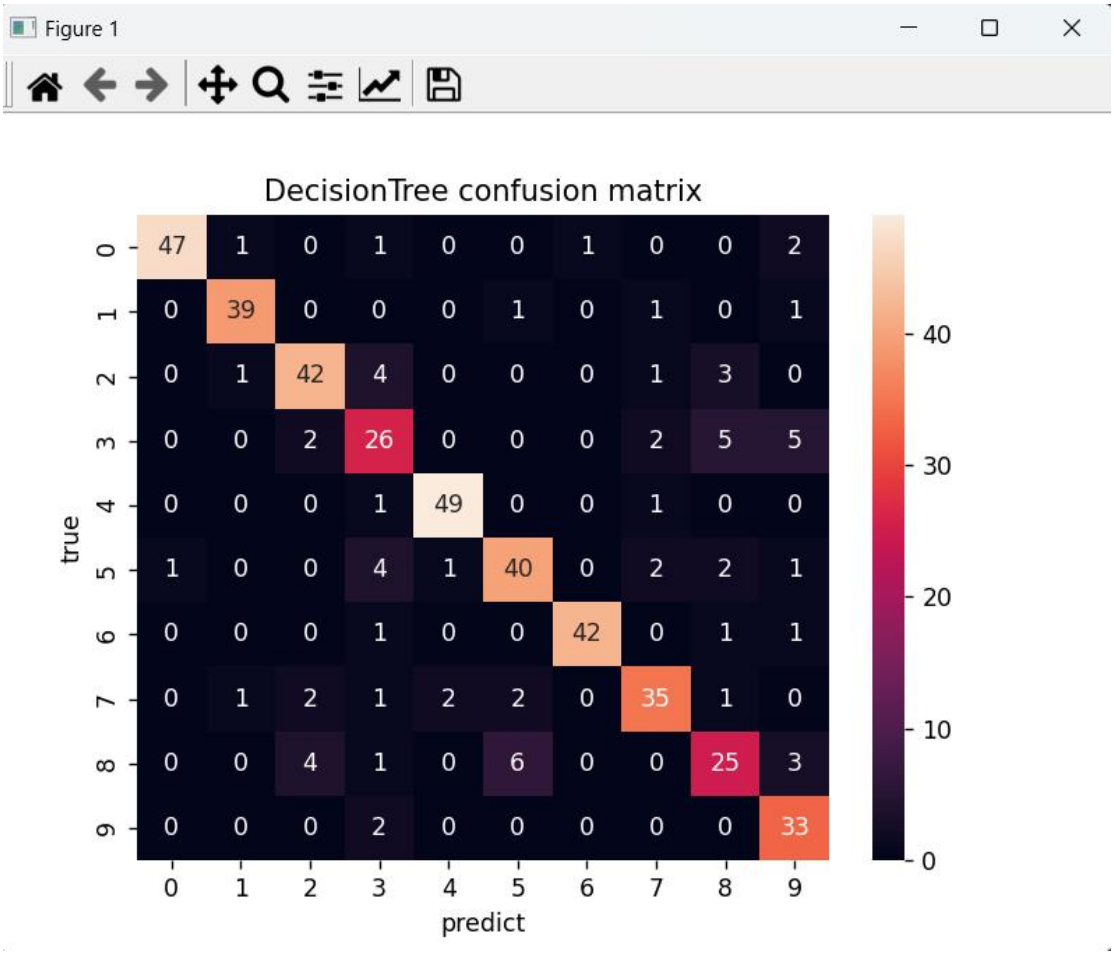
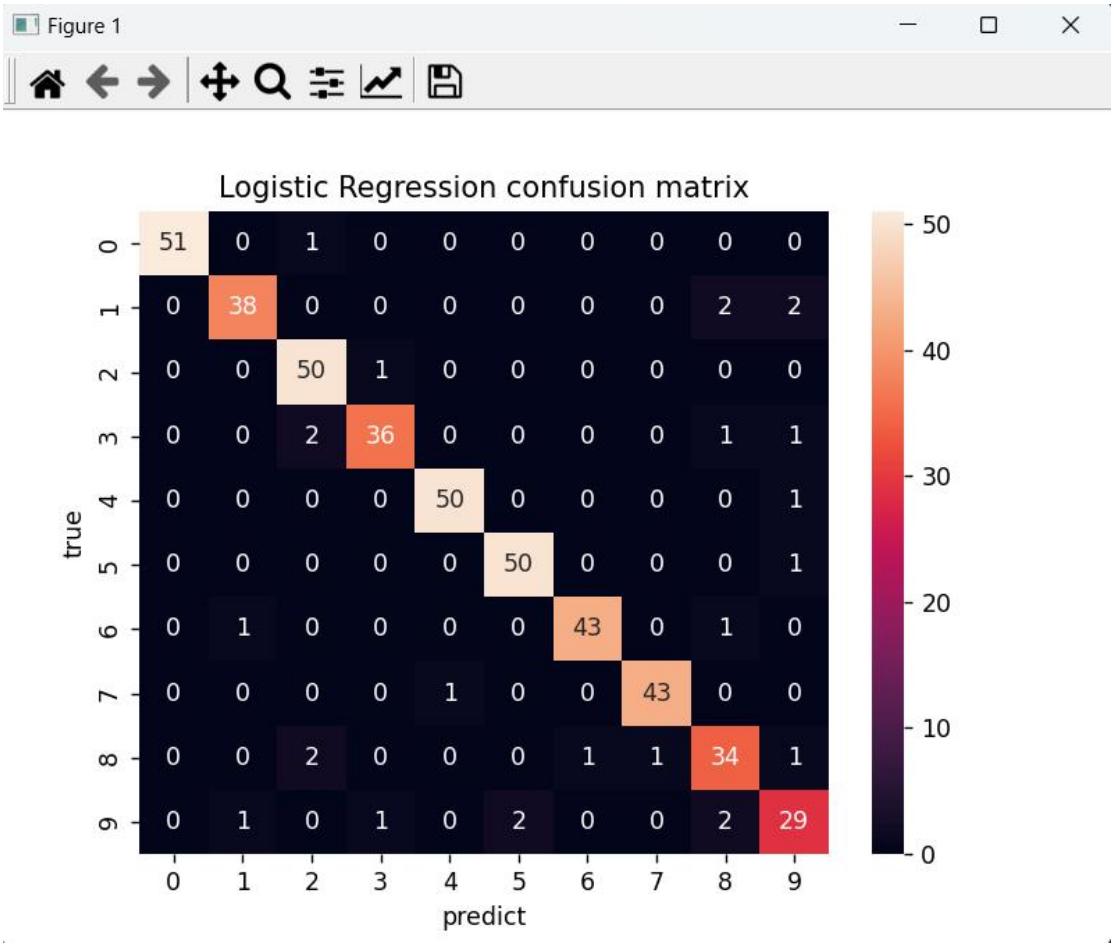


Figure 1
DecisionTree confusion matrix
true
predict
0 1 2 3 4 5 6 7 8 9
47 1 0 1 0 0 1 0 0 2
0 39 0 0 0 1 0 1 0 1
0 1 42 4 0 0 0 1 3 0
0 0 2 26 0 0 0 2 5 5
0 0 0 1 49 0 0 1 0 0
1 0 0 4 1 40 0 2 2 1
0 0 0 1 0 0 42 0 1 1
0 1 2 1 2 2 0 35 1 0
0 0 4 1 0 6 0 0 25 3
0 0 0 2 0 0 0 0 0 33
40
30
20
10
0



精确率与召回率计算：

	precision	recall	f1-score	support
0	1.00	1.00	1.00	52
1	0.95	1.00	0.98	42
2	1.00	0.98	0.99	51
3	0.95	0.97	0.96	40
4	0.98	0.94	0.96	51
5	0.98	1.00	0.99	51
6	1.00	1.00	1.00	45
7	0.96	0.98	0.97	44
8	0.93	0.95	0.94	39
9	1.00	0.91	0.96	35
accuracy			0.98	450
macro avg	0.97	0.97	0.97	450
weighted avg	0.98	0.98	0.98	450

KNN

	precision	recall	f1-score	support
0	0.78	1.00	0.87	52
1	0.95	0.93	0.94	42
2	0.96	0.98	0.97	51
3	0.92	0.90	0.91	40
4	1.00	0.96	0.98	51
5	0.92	0.94	0.93	51
6	1.00	0.93	0.97	45
7	1.00	0.95	0.98	44
8	0.94	0.87	0.91	39
9	0.97	0.83	0.89	35
accuracy			0.94	450
macro avg	0.94	0.93	0.93	450
weighted avg	0.94	0.94	0.94	450

MLP

	precision	recall	f1-score	support
0	1.00	1.00	1.00	52
1	0.93	1.00	0.97	42
2	0.94	0.94	0.94	51
3	0.95	0.88	0.91	40
4	1.00	0.96	0.98	51
5	0.98	0.98	0.98	51
6	1.00	0.98	0.99	45
7	0.94	1.00	0.97	44
8	0.86	0.82	0.84	39
9	0.86	0.91	0.89	35
accuracy			0.95	450
macro avg	0.95	0.95	0.95	450
weighted avg	0.95	0.95	0.95	450

RandomForest

	precision	recall	f1-score	support
0	0.95	0.81	0.88	52
1	0.61	0.83	0.71	42
2	0.76	0.25	0.38	51
3	0.88	0.53	0.66	40
4	0.97	0.71	0.82	51
5	0.79	0.37	0.51	51
6	0.97	0.69	0.81	45
7	0.79	0.75	0.77	44
8	0.27	0.77	0.40	39
9	0.38	0.66	0.48	35
accuracy			0.63	450
macro avg	0.74	0.64	0.64	450
weighted avg	0.76	0.63	0.65	450

Adaboost

	precision	recall	f1-score	support
0	1.00	1.00	1.00	42
1	0.98	1.00	0.99	50
2	0.98	0.94	0.96	47
3	1.00	0.96	0.98	48
4	0.95	0.90	0.92	40
5	0.98	1.00	0.99	52
6	0.98	1.00	0.99	43
7	0.96	1.00	0.98	49
8	0.89	0.97	0.93	40
9	0.97	0.90	0.93	39
accuracy			0.97	450
macro avg	0.97	0.97	0.97	450
weighted avg	0.97	0.97	0.97	450

SVM

	precision	recall	f1-score	support
0	0.98	0.90	0.94	52
1	0.93	0.93	0.93	42
2	0.84	0.82	0.83	51
3	0.63	0.65	0.64	40
4	0.94	0.96	0.95	51
5	0.82	0.78	0.80	51
6	0.98	0.93	0.95	45
7	0.83	0.80	0.81	44
8	0.68	0.64	0.66	39
9	0.72	0.94	0.81	35
accuracy			0.84	450
macro avg	0.83	0.84	0.83	450
weighted avg	0.84	0.84	0.84	450

DecisionTree

	precision	recall	f1-score	support
0	1.00	0.98	0.99	52
1	0.95	0.90	0.93	42
2	0.91	0.98	0.94	51
3	0.95	0.90	0.92	40
4	0.98	0.98	0.98	51
5	0.96	0.98	0.97	51
6	0.98	0.96	0.97	45
7	0.98	0.98	0.98	44
8	0.85	0.87	0.86	39
9	0.83	0.83	0.83	35
accuracy			0.94	450
macro avg	0.94	0.94	0.94	450
weighted avg	0.94	0.94	0.94	450

Logistic Regression

从上述结果中不仅可以看出算法整体的准确率，还可以观察到算法对各个类别的识别率，上述结果均是调整好参数后得到的数据，由以上得到的信息我可以依据这七种算法模型在数据集上的表现进行排序，排序结果如下所示：

KNN>SVM>MLP>RandomForest>LogisticRegression>DecisionTree>Adaboost

再者，根据天下没有免费的午餐定理，上述排序仅限于该实验手写体数字识别问题上，如果数据集发生变化上述结论可能也会发生变化。甚至让不同实验参与者进行调参得到的结果也可能不尽相同，毕竟参数对算法模型的影响是巨大的。

总之，此次实验构建的七种模型中，KNN 与 SVM 是最令人满意的。

4.3 实验问题分析

1. 此次实验中我遇到的第一个问题是如何让手写体样本以图片的形式显示出来，在数据集里拿到的数据是 1797*64 的二维数据，每个手写体样本均为一维向量，在网上查阅相关问题之后我发现 imshow()函数的输入必须是二维的，然后又找到了 reshape()函数，该函数可以对向量进行转换，我利用它将一维向量升维成了二维向量，最终可以成功地显示数据集中的图片；

2. 在 MLP 算法模型的建立过程中，最开始我没有想到对标签进行独热编码变换，而是直接地利用数据集分割函数来分割数据集，然后再用模型训练它们，最终得到 96%的准确率。构建完 MLP 模型之后我想再用独热编码会不会提升模型的性能，于是将每一个标签都利用独热编码的规则转换成 10 维的行向量重新构建 MLP 模型，结果却发现准确率居然不升反降，原因是我分析是固然独热编码的确有很大的优越性，在计算方面可能会更加简单，但是 MLP 是一个全连接网络，这个模型本身空间和时间复杂度已经很高了：我构建的是一个含有 100 层神经元隐藏层的 MLP 模型，而我的数据集总共却只有 1797 个手写体样本，在划分数据集之后，训练集可能就只有大约 1348 个手写体样本了。所以我推测不是独热编码的原因使得准确率下降，而是因为算法模型比较复杂，数据集相对比较少，该算法在训练集上出现了过拟合导致的。于是我又重新对比了一下对标签进行独热编码和不对标签进行独热编码的结果，发现可以直观地看出在对标签数据进行独热编码的情况下，模型确实出现了一定程度的过拟合，由此可以验证我上述的猜想是正确的；

3. 在进行实验之前，对于多分类问题，我能想到的解决办法只有 KNN 与 MLP 算法，因为在我狭隘的眼光中 SVM，Logistic Regerssion 等等算法都是用来处理二分类问题的，虽然它们也可以处理多分类问题，但是需要提前构建多个分类器或者对数据集划分，操作起来还是相当困难的。但是当我真正去尝试的时候发现 SVM、Random Forest、Logistic Regression 等算法对于多分类问题依然可以得到准确度较高的结果。原因可能是 sklearn 库的代码编程内部工作人员也想到了这些问题，SVM 和 Logistic Regression 等算法确实需要通过一对多或一类对另一类的方法实现多分类，但这些都不需要我自己去动手重新搭建，在这些算法模型的类中早就设定好了这些参数，如果想要实现一个多分类问题，我只需要去认为改变这些参数就可以了，于是，我就把这七种算法都尝试了一遍，发现大多算法都可以取得不错的性能；

4. 在本次模式识别实验的几种机器学习算法中，准确率大都可以保持在 90%或 80%以上，唯有 Adaboost 算法的准确率只在 63%左右，在我调整了学习率和分类器的个数之后还是没有明显提高，其余六种算法模型在默认情况下，准

确率都在 90%和 80%以上，而 Adaboost 算法在默认情况下更是只有大约 30%多一点。在经过仔细缜密地调参之后，准确率才上升到 63%左右，但这显然要远远低于其他算法。我查阅相关资料后发现，Adaboost 作为一种并行式的算法，是处理二分类问题的利器，但在多分类问题上不甚很理想，除此之外还有一个原因，可能由于数据集的样本数不多，导致该算法出现欠拟合的现象。

4.4 实验心得体会

本次模式识别实验之前，我在机器学习算法方面的学习，理论知识是要远多于动手实践的。所以很多细节我并没有仔细地思考过。在代码编程环节中，我通过不断地实践各类算法，同时对它们的各种参数进行仔细调整，以往被我经常性忽略的一些问题也能重新主动思考它们了，这是理论知识学习所无法带给我的。

而且我认为代码编程环节的练习比理论的教学更有趣一点，它并不像推导各类算法公式那般枯燥无味，有时候公式推导一遍后可能很快就会又忘却了，但是当我看到自己构建的模型可以最终得到一个较为不错的准确率时，心里还是会有一点成就感，当然理论知识与编程实践绝对是相辅相成的，都很重要。

之前我自学过一段时间的模式识别，但是理论知识有点缺乏，所以没有坚持下来，而这学期跟着老师系统的学习一遍，在理论上，我还是有所收获的，几个著名的模式识别算法也不是完全不懂了，在编程上也可以更好的入手与学习了。这门课的学习以及实验令我受益匪浅。

源代码以及注释见附件。