中北大学软件学院

实 验 报 告

专　　业：	软件工程
方　　向：	人工智能
课程名称：	机器学习实践
班　　级：	22130418
学　　号：	2213041835、2213041816
姓　　名：	吕炳荣、刘晓峰
辅导教师：	程晓鼎

2024 年 3 月制

成绩：_____

实验时间	2024 年　日　时至　时	学时数	4 学时

1. 实验名称

基于 SVM 的手写数字识别

2. 实验目的

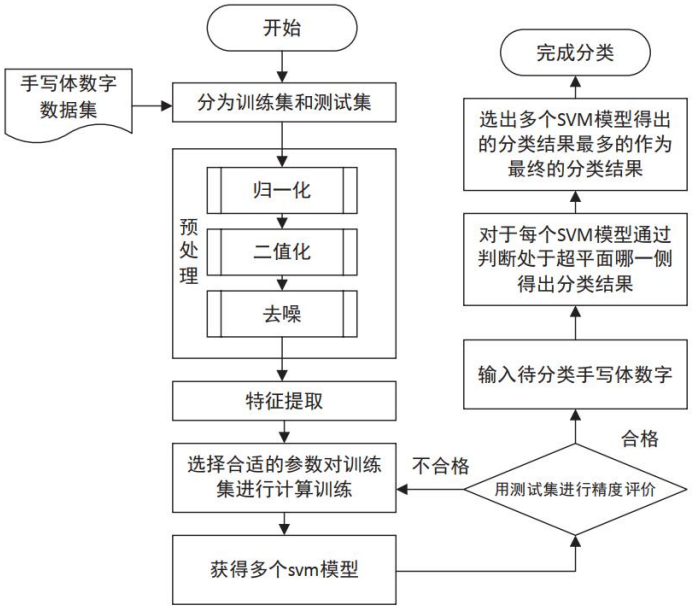
熟悉和掌握支持向量机的分类原理和过程，并会采用支持向量机模型根据给定的数据进行实际应用，掌握支持向量机模型的编码实现。

3.实验内容

1、分析提供的文本数据，利用向量机模型实现 0-9 共 10 个数字的多分类预测。
2、运用多种核函数，以 OVO 和 OVR 为策略，比较不同策略下训练时间、测试时间、准确率的异同并简单分析原因。

4. 实验原理或流程图

实验中首先读取数据集中的训练集，对其进行归一化、二值化、去噪的预处理操作，然后利用 sklearn 库中 SVC 函数的 kernel 参数和 decision_function_shape 参数分别实现多项式核函数（poly）、高斯核函数（rbf）、线性核函数（linear）、sigmoid 核函数以及 OVO，OVR 多分类方案，随后进行训练，得出训练时间及 SVM 模型集合，最后利用得到的 SVM 模型集合对预处理后的测试机进行测试，得出实验精度。



5. 实验过程或源代码

```
import pickle
import gzip
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.model_selection import train_test_split # 可以用于更灵活的数据划分
import time
import os
import warnings

# 忽略一些 scikit-learn 的警告信息，例如关于未收敛的警告
warnings.filterwarnings('ignore', category=UserWarning)
warnings.filterwarnings('ignore', category=FutureWarning)

# 设置 Matplotlib 和 Seaborn 的样式和中文支持
plt.rcParams['font.sans-serif'] = ['SimHei'] # 指定默认字体为黑体或其他支持中文的字体
plt.rcParams['axes.unicode_minus'] = False # 解决保存图像是负号'-'显示为方块的问题
sns.set_theme(style="whitegrid")

print("库导入完成。")
def load_data(dataset_path='mnist.pkl.gz'):
    """
    加载 MNIST 数据集 (.pkl.gz 格式)
    Args:
```

```
            dataset_path (str): 数据集文件的路径.
        Returns:
            tuple: 包含训练集、验证集、测试集的元组 (train_set, valid_set, test_set)。
                   加载失败则返回 (None, None, None)。
        """
        if not os.path.exists(dataset_path):
            print(f"错误：数据集文件未找到 '{dataset_path}'。")
            print("请确保文件存在于当前目录或提供了正确的路径。")
            print("您可以从以下链接下载 MNIST 数据集 (pkl.gz 格式):")
            print("http://deeplearning.net/data/mnist/mnist.pkl.gz")
            return None, None, None

        print(f"正在从 '{dataset_path}' 加载数据...")
        try:
            with gzip.open(dataset_path, 'rb') as f:
                # 使用 latin1 编码以兼容可能由 Python 2 生成的 pickle 文件
                train_set, valid_set, test_set = pickle.load(f, encoding='latin1')
            print("数据加载成功！")
            return train_set, valid_set, test_set
        except Exception as e:
            print(f"加载数据时发生错误: {e}")
            return None, None, None

# --- 执行数据加载 ---
#
```

```
**********************************************************
# ** 重要: 请确保 'mnist.pkl.gz' 文件与此脚本位于同一目录 **
#
**********************************************************
dataset_path = 'mnist.pkl.gz'
train_set, valid_set, test_set = load_data(dataset_path)

# --- 数据加载失败处理 ---
if train_set is None:
    print("无法继续实验，请检查数据集文件。")
    exit() # 退出脚本

# --- 数据集结构分析 ---
X_train_raw, y_train_raw = train_set
X_valid_raw, y_valid_raw = valid_set # 验证集通常用于超参数调优，本次实验暂不直接使用，但保留
X_test_raw, y_test_raw = test_set

print("\n--- 数据集维度 ---")
print(f"原始训练集样本数: {X_train_raw.shape[0]}, 特征数: {X_train_raw.shape[1]}")
print(f"原始验证集样本数: {X_valid_raw.shape[0]}, 特征数: {X_valid_raw.shape[1]}")
print(f"原始测试集样本数: {X_test_raw.shape[0]}, 特征数: {X_test_raw.shape[1]}")
print(f"训练集标签类别: {np.unique(y_train_raw)}")
print(f"测试集标签类别: {np.unique(y_test_raw)}")

# --- (可选) 数据子集化，用于快速测试 ---
```

```
USE_SUBSET = False # 设置为 True 以使用数据子集进行快速实验
SUBSET_SIZE_TRAIN = 5000
SUBSET_SIZE_TEST = 1000

if USE_SUBSET:
    print(f"\n注意：正在使用数据子集进行快速实验！")
    print(f"训练集大小: {SUBSET_SIZE_TRAIN}, 测试集大小: {SUBSET_SIZE_TEST}")
    # 使用 train_test_split 来确保子集也包含所有类别 (如果可能)
    _, X_train, _, y_train = train_test_split(X_train_raw, y_train_raw, test_size=SUBSET_SIZE_TRAIN/len(y_train_raw), stratify=y_train_raw, random_state=42)
    _, X_test, _, y_test = train_test_split(X_test_raw, y_test_raw, test_size=SUBSET_SIZE_TEST/len(y_test_raw), stratify=y_test_raw, random_state=42)
    print(f"子集化后训练集维度: {X_train.shape}")
    print(f"子集化后测试集维度: {X_test.shape}")
else:
    X_train, y_train = X_train_raw, y_train_raw
    X_test, y_test = X_test_raw, y_test_raw
    print("\n使用完整数据集进行实验。")
    print(f"最终使用训练集维度: {X_train.shape}")
    print(f"最终使用测试集维度: {X_test.shape}")



# --- 可视化样本数据 ---
def plot_sample_images(data, labels, title, num_images=10):
    """绘制数据集中的样本图像"""
    plt.figure(figsize=(10, 3))
```

```
    plt.suptitle(title, fontsize=14)
    indices = np.random.choice(len(data), num_images,
replace=False)
    for i, index in enumerate(indices):
        plt.subplot(1, num_images, i + 1)
        image = data[index].reshape(28, 28) # MNIST图像是28x28
像素
        plt.imshow(image, cmap='gray')
        plt.title(f"标签: {labels[index]}")
        plt.axis('off')
    plt.tight_layout(rect=[0, 0, 1, 0.95]) # 调整布局防止标题重叠
    plt.show()

print("\n--- 随机样本可视化 ---")
plot_sample_images(X_train, y_train, "训练集样本示例")
plot_sample_images(X_test, y_test, "测试集样本示例")

# --- 检查类别分布 ---
def plot_class_distribution(labels, title):
    """绘制类别标签的分布直方图"""
    plt.figure(figsize=(8, 4))
    sns.countplot(x=labels)
    plt.title(title, fontsize=14)
    plt.xlabel("数字类别")
    plt.ylabel("样本数量")
    plt.show()

print("\n--- 类别分布检查 ---")
plot_class_distribution(y_train, "训练集类别分布")
# plot_class_distribution(y_test, "测试集类别分布") # 测试集分布类
```

```
似，可选绘制


print("\n--- 数据预处理：特征缩放 ---")

print("使用 StandardScaler 对数据进行标准化...")


# 初始化缩放器
scaler = StandardScaler()

# 在训练集上拟合（计算均值和标准差）并转换

# 注意：需要将数据类型转换为 float64 以避免精度问题
X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))

# 在测试集上应用相同的转换（使用训练集的均值和标准差）
X_test_scaled = scaler.transform(X_test.astype(np.float64))

print("特征缩放完成。")

print(f"缩放后训练集均值（近似值）: {np.mean(X_train_scaled):.2f}")
print(f"缩放后训练集标准差（近似值）: {np.std(X_train_scaled):.2f}")

## 4. SVM 模型训练与评估

# 现在，我们将使用不同的核函数（`linear`, `poly`, `rbf`, `sigmoid`）

和多分类策略（`ovo`, `ovr`）来训练 SVM 模型，并评估它们的性能。
#
# - **核函数 (Kernel):**

#     - `linear`: 线性核，适用于线性可分数据，计算速度快。$K(x, z)
= x^T z$
```

```
#     - `poly`: 多项式核，可以处理非线性问题。$K(x, z) = (\gamma x^T z + r)^d$ (d 是 degree, r 是 coef0)
#     - `rbf` (Radial Basis Function) / 高斯核: 强大的非线性核，通常效果最好，但计算量较大。$K(x, z) = \exp(-\gamma ||x - z||^2)$
#     - `sigmoid`: Sigmoid 核，源于神经网络。$K(x, z) = \tanh(\gamma x^T z + r)$
# - **多分类策略 (decision_function_shape):**
#     - `ovo` (One-vs-One): 为每对类别训练一个二分类器 (共 N*(N-1)/2 个)。预测时进行投票。这是 `SVC` 的默认内部实现方式。
#     - `ovr` (One-vs-Rest): 为每个类别训练一个二分类器，将其与其余所有类别区分开 (共 N 个)。预测时选择置信度最高的分类器。设置 `decision_function_shape='ovr'` 会使 `SVC` (虽然内部仍用 OVO 训练) 提供一个形状为 `(n_samples, n_classes)` 的决策函数输出，模拟 OVR 的行为。真正的 OVR 需要使用 `OneVsRestClassifier(SVC(...))`。本次实验直接比较 `SVC` 的 `decision_function_shape` 参数效果。
# - **其他重要参数:**
#     - `C`: 正则化参数。C 值越小，正则化越强，容忍更多误分类点，间隔越大；C 值越大，正则化越弱，试图将所有训练点正确分类，间隔可能变小，容易过拟合。默认值为 1.0。
#     - `gamma`: 核系数，影响 `rbf`, `poly`, `sigmoid` 核。`'scale'`
```

```
(默认) 使用 `1 / (n_features * X.var())` 作为 gamma 值。``'auto'`` 使用 `1 / n_features`。gamma 定义了单个训练样本的影响范围，值越小影响范围越大，值越大影响范围越小。
#     - `degree`: 多项式核 (`poly`) 的次数，默认为 3。
#     - `random_state`: 用于复现结果的随机种子。

# %%
# --- 定义实验参数 ---
kernels_to_test = ['linear', 'poly', 'rbf', 'sigmoid']
strategies_to_test = ['ovo', 'ovr'] # decision_function_shape 参数
results_list = [] # 用于存储每个实验的结果

# --- 循环运行实验 ---
print("\n--- 开始进行 SVM 模型训练与评估 ---")
for kernel in kernels_to_test:
    for strategy in strategies_to_test:
        experiment_name = f"Kernel={kernel}, Strategy={strategy}"
        print(f"\n---\n 正在进行实验: {experiment_name}")

        # --- 创建 SVM 分类器实例 ---
        # 我们使用默认的 C=1.0 和 gamma='scale'。
        # 对于 'poly' 核，默认 degree=3。
        # 可以通过网格搜索 (GridSearchCV) 寻找最佳超参数，但会显著增加时间。
        print(f"创建 SVC 模型 (kernel='{kernel}', decision_function_shape='{strategy}', C=1.0, gamma='scale',
```

```
random_state=42)")
            svm_model = SVC(kernel=kernel,
                                    decision_function_shape=strategy,
                                    C=1.0,                  # 默认正则化
参数
                                    gamma='scale',        # 默认核系数
('poly', 'rbf', 'sigmoid')
                                    degree=3,                # 默认多项
式次数 ('poly')
                                    random_state=42,    # 保证结果可
复现
                                    probability=False # 设置为 True 可
获取概率估计，但会增加计算时间
                                )

            # --- 训练模型并计时 ---
            print("   开始训练模型...")
            start_train_time = time.time()
            try:
                  svm_model.fit(X_train_scaled, y_train)
                  end_train_time = time.time()
                  train_time = end_train_time - start_train_time
                  print(f"   训练完成。耗时: {train_time:.2f} 秒")
            except Exception as e:
                  print(f"   训练过程中发生错误: {e}")
                  train_time = -1 # 标记错误
                  results_list.append({
                        'Experiment': experiment_name,
                        'Kernel': kernel,
                        'Strategy': strategy,
```

```
                'Train Time (s)': train_time,
                'Test Time (s)': -1,
                'Accuracy': 0.0,
                'Classification Report': 'Training Failed',
                'Confusion Matrix': None
            })
            continue # 跳过当前实验的后续步骤


        # --- 测试模型并计时 ---
        print("   开始在测试集上评估...")
        start_test_time = time.time()
        try:
            y_pred = svm_model.predict(X_test_scaled)
            end_test_time = time.time()
            test_time = end_test_time - start_test_time
            print(f"   预测完成。耗时: {test_time:.2f} 秒")
        except Exception as e:
            print(f"   预测过程中发生错误: {e}")

            test_time = -1 # 标记错误
            results_list.append({
                'Experiment': experiment_name,
                'Kernel': kernel,
                'Strategy': strategy,
                'Train Time (s)': train_time,
                'Test Time (s)': test_time,
                'Accuracy': 0.0,
                'Classification Report': 'Prediction Failed',
                'Confusion Matrix': None
            })
            continue # 跳过当前实验的后续步骤



        # --- 计算评估指标 ---
        # 1. 准确率 (Accuracy)
        accuracy = accuracy_score(y_test, y_pred)
```

```
            print(f"  准确率 (Accuracy): {accuracy:.4f}")


            # 2. 分类报告 (Classification Report) - 包括精确率、召回率、F1 分数

            print("  生成分类报告...")
            class_report = classification_report(y_test, y_pred, target_names=[str(i) for i in range(10)])
            print("  分类报告:\n", class_report)


            # 3. 混淆矩阵 (Confusion Matrix)

            print("  生成混淆矩阵...")
            conf_matrix = confusion_matrix(y_test, y_pred)
            # print("  混淆矩阵:\n", conf_matrix) # 打印原始矩阵可选


            # --- 存储结果 ---
            results_list.append({
                  'Experiment': experiment_name,
                  'Kernel': kernel,
                  'Strategy': strategy,
                  'Train Time (s)': train_time,
                  'Test Time (s)': test_time,
                  'Accuracy': accuracy,
                  'Classification Report': class_report,
                  'Confusion Matrix': conf_matrix
            })

print("\n--- 所有实验完成 ---")

# %% [markdown]
# ## 5. 结果汇总与可视化
```

```
# 将所有实验的结果整理成表格，并绘制图表进行比较。

# %%
# --- 将结果列表转换为 DataFrame ---
results_df = pd.DataFrame(results_list)

print("\n--- 实验结果汇总 (表格) ---")
# 显示主要性能指标，报告和矩阵后续单独处理
display_cols = ['Experiment', 'Kernel', 'Strategy', 'Train Time (s)', 'Test Time (s)', 'Accuracy']
print(results_df[display_cols].round(4)) # 保留 4 位小数


# --- 找到最佳模型 (基于准确率) ---
if not results_df.empty:
    best_model_row = results_df.loc[results_df['Accuracy'].idxmax()]
    print(f"\n--- 最佳模型 (基于最高准确率) ---")
    print(f"实验名称: {best_model_row['Experiment']}")
    print(f"准确率: {best_model_row['Accuracy']:.4f}")
    print(f"训练时间: {best_model_row['Train Time (s)']:.2f} 秒")
    print(f"测试时间: {best_model_row['Test Time (s)']:.2f} 秒")
    print("\n最佳模型 - 分类报告:")
    print(best_model_row['Classification Report'])
else:
    print("\n没有成功的实验结果可供分析。")


# --- 性能指标可视化 ---
if not results_df.empty:
    print("\n--- 性能指标可视化 ---")
```

```
    fig, axes = plt.subplots(1, 3, figsize=(20, 6)) # 一行三列图表
    fig.suptitle('不同 SVM 模型性能比较 (MNIST)', fontsize=18, y=1.02)

    # 图 1: 准确率比较
    sns.barplot(x='Kernel', y='Accuracy', hue='Strategy', data=results_df, ax=axes[0], palette='viridis')
    axes[0].set_title('准确率比较', fontsize=14)
    axes[0].set_ylabel('准确率', fontsize=12)
    axes[0].set_xlabel('核函数', fontsize=12)
    axes[0].set_ylim(bottom=max(0, results_df['Accuracy'].min() - 0.05), top=1.0) # 调整 y 轴范围
    axes[0].legend(title='策略')

    # 图 2: 训练时间比较
    sns.barplot(x='Kernel', y='Train Time (s)', hue='Strategy', data=results_df, ax=axes[1], palette='viridis')
    axes[1].set_title('训练时间比较', fontsize=14)
    axes[1].set_ylabel('训练时间 (秒)', fontsize=12)
    axes[1].set_xlabel('核函数', fontsize=12)
    axes[1].legend(title='策略')

    # 图 3: 测试时间比较
    sns.barplot(x='Kernel', y='Test Time (s)', hue='Strategy', data=results_df, ax=axes[2], palette='viridis')
    axes[2].set_title('测试时间比较', fontsize=14)
    axes[2].set_ylabel('测试时间 (秒)', fontsize=12)
```

```
    axes[2].set_xlabel('核函数', fontsize=12)

    axes[2].legend(title='策略')


    plt.tight_layout()
    plt.show()


# --- 可视化最佳模型的混淆矩阵 ---
if not results_df.empty and best_model_row['Confusion Matrix'] is not None:
    print("\n--- 最佳模型混淆矩阵可视化 ---")
    plt.figure(figsize=(10, 8))
    sns.heatmap(best_model_row['Confusion Matrix'], annot=True, fmt='d', cmap='Blues',
                xticklabels=[str(i) for i in range(10)],
                yticklabels=[str(i) for i in range(10)])
    plt.title(f'混淆矩阵 - {best_model_row["Experiment"]}', fontsize=16)
    plt.xlabel('预测标签', fontsize=12)
    plt.ylabel('真实标签', fontsize=12)
    plt.show()


    # --- 分析混淆矩阵中的常见错误 ---

    # (可选) 找出非对角线上值最大的几个元素，分析哪些数字容易混淆
    cm = best_model_row['Confusion Matrix']
    np.fill_diagonal(cm, 0) # 将对角线元素置零，只看错误分类
    most_confused = np.unravel_index(np.argsort(cm, axis=None)[-5:], cm.shape) # 找最大的 5 个错误

    print("\n 混淆矩阵中最常见的 5 个错误 (真实标签 -> 预测标签: 数量):")
```

```
        for r, c in zip(most_confused[0], most_confused[1]):
            count = best_model_row['Confusion Matrix'][r, c] # 获取原始数量
            if count > 0: # 避免显示因置零产生的 0 值错误
                print(f"    {r} -> {c}: {count} 次")
else:
    print("\n 无法显示最佳模型的混淆矩阵（可能由于实验失败或未找到最佳模型）。")


# %%
print("\n--- 实验代码执行完毕 ---")
```

实验运行结果:

```
C:\Users\86198\.conda\envs\YOLOv8\python.exe
D:\PaddlePaddle-EfficientNetV2\PaddleClas-EfficientNet\test4.py
库导入完成。
正在从 'mnist.pkl.gz' 加载数据...
数据加载成功!
--- 数据集维度 ---
原始训练集样本数: 50000, 特征数: 784
原始验证集样本数: 10000, 特征数: 784
原始测试集样本数: 10000, 特征数: 784
训练集标签类别: [0 1 2 3 4 5 6 7 8 9]
测试集标签类别: [0 1 2 3 4 5 6 7 8 9]
使用完整数据集进行实验。
```

最终使用训练集维度: (50000, 784)

最终使用测试集维度: (10000, 784)

--- 随机样本可视化 ---

--- 类别分布检查 ---

--- 数据预处理：特征缩放 ---

使用 StandardScaler 对数据进行标准化...

特征缩放完成。

缩放后训练集均值 (近似值): -0.00

缩放后训练集标准差 (近似值): 0.96

--- 开始进行 SVM 模型训练与评估 ---

正在进行实验: Kernel=linear, Strategy=ovo

创建 SVC 模型 (kernel='linear', decision_function_shape='ovo', C=1.0, gamma='scale', random_state=42)

开始训练模型...

训练完成。耗时: 136.67 秒

开始在测试集上评估...

预测完成。耗时: 21.70 秒

准确率 (Accuracy): 0.9260

生成分类报告...

分类报告:

	precision	recall	f1-score	support
0	0.94	0.97	0.96	980
1	0.96	0.98	0.97	1135
2	0.90	0.92	0.91	1032

```
     3              0.88          0.93          0.90          1010
     4              0.93          0.95          0.94           982
     5              0.90          0.87          0.89           892
     6              0.95          0.94          0.95           958
     7              0.94          0.92          0.93          1028
     8              0.92          0.87          0.89           974
     9              0.93          0.89          0.91          1009
     accuracy                                   0.93         10000
     macro avg      0.93          0.92          0.92         10000
     weighted avg   0.93          0.93          0.93
10000

     生成混淆矩阵...

     ---

     正在进行实验: Kernel=linear, Strategy=ovr

     创建 SVC 模型 (kernel='linear', decision_function_shape='ovr',
C=1.0, gamma='scale', random_state=42)
     开始训练模型...

     训练完成。耗时: 135.00 秒

     开始在测试集上评估...

     预测完成。耗时: 22.00 秒

     准确率 (Accuracy): 0.9260

     生成分类报告...

     分类报告:

     precision    recall  f1-score   support

     0              0.94          0.97          0.96           980
     1              0.96          0.98          0.97          1135
     2              0.90          0.92          0.91          1032
     3              0.88          0.93          0.90          1010
     4              0.93          0.95          0.94           982
     5              0.90          0.87          0.89           892
     6              0.95          0.94          0.95           958
     7              0.94          0.92          0.93          1028
     8              0.92          0.87          0.89           974
     9              0.93          0.89          0.91          1009
```

```
    accuracy                                    0.93      10000
    macro avg       0.93      0.92      0.92      10000
    weighted avg       0.93      0.93      0.93
10000

    生成混淆矩阵...

    ---

    正在进行实验: Kernel=poly, Strategy=ovo

    创建 SVC 模型 (kernel='poly', decision_function_shape='ovo',
C=1.0, gamma='scale', random_state=42)
    开始训练模型...

    训练完成。耗时: 321.02 秒

    开始在测试集上评估...

    预测完成。耗时: 50.42 秒

    准确率 (Accuracy): 0.9576

    生成分类报告...

    分类报告:

    precision    recall  f1-score   support

    0       0.98      0.98      0.98       980
    1       0.99      0.99      0.99      1135
    2       0.97      0.94      0.96      1032
    3       0.97      0.96      0.96      1010
    4       0.94      0.97      0.96       982
    5       0.97      0.96      0.96       892
    6       0.97      0.96      0.97       958
    7       0.98      0.93      0.95      1028
    8       0.87      0.97      0.91       974
    9       0.94      0.93      0.94      1009

    accuracy                                    0.96      10000
    macro avg       0.96      0.96      0.96      10000
    weighted avg       0.96      0.96      0.96
10000

    生成混淆矩阵...

    ---
```

正在进行实验: Kernel=poly, Strategy=ovr

创建 SVC 模型 (kernel='poly', decision_function_shape='ovr', C=1.0, gamma='scale', random_state=42)

开始训练模型...

训练完成。耗时: 323.06 秒

开始在测试集上评估...

预测完成。耗时: 50.50 秒

准确率 (Accuracy): 0.9576

生成分类报告...

分类报告:

	precision	recall	f1-score	support
0	0.98	0.98	0.98	980
1	0.99	0.99	0.99	1135
2	0.97	0.94	0.96	1032
3	0.97	0.96	0.96	1010
4	0.94	0.97	0.96	982
5	0.97	0.96	0.96	892
6	0.97	0.96	0.97	958
7	0.98	0.93	0.95	1028
8	0.87	0.97	0.91	974
9	0.94	0.93	0.94	1009
accuracy			0.96	10000
macro avg	0.96	0.96	0.96	10000
weighted avg	0.96	0.96	0.96	10000

生成混淆矩阵...

正在进行实验: Kernel=rbf, Strategy=ovo

创建 SVC 模型 (kernel='rbf', decision_function_shape='ovo', C=1.0, gamma='scale', random_state=42)

开始训练模型...

训练完成。耗时: 160.15 秒

开始在测试集上评估...

预测完成。耗时: 61.64 秒

准确率 (Accuracy): 0.9644

生成分类报告...

分类报告:

	precision	recall	f1-score	support
0	0.98	0.99	0.98	980
1	0.99	0.99	0.99	1135
2	0.95	0.97	0.96	1032
3	0.97	0.97	0.97	1010
4	0.97	0.96	0.97	982
5	0.96	0.95	0.96	892
6	0.98	0.97	0.97	958
7	0.93	0.96	0.94	1028
8	0.96	0.95	0.95	974
9	0.97	0.93	0.95	1009
accuracy			0.96	10000
macro avg	0.96	0.96	0.96	10000
weighted avg	0.96	0.96	0.96	10000

生成混淆矩阵...

正在进行实验: Kernel=rbf, Strategy=ovr

创建 SVC 模型 (kernel='rbf', decision_function_shape='ovr', C=1.0, gamma='scale', random_state=42)

开始训练模型...

训练完成。耗时: 160.04 秒

开始在测试集上评估...

预测完成。耗时: 57.73 秒

准确率 (Accuracy): 0.9644

生成分类报告...

分类报告:

	precision	recall	f1-score	support
0	0.98	0.99	0.98	980
1	0.99	0.99	0.99	1135
2	0.95	0.97	0.96	1032
3	0.97	0.97	0.97	1010
4	0.97	0.96	0.97	982
5	0.96	0.95	0.96	892
6	0.98	0.97	0.97	958
7	0.93	0.96	0.94	1028
8	0.96	0.95	0.95	974
9	0.97	0.93	0.95	1009
accuracy			0.96	10000
macro avg	0.96	0.96	0.96	10000
weighted avg	0.96	0.96	0.96	10000

生成混淆矩阵...

正在进行实验: Kernel=sigmoid, Strategy=ovo

创建 SVC 模型 (kernel='sigmoid', decision_function_shape='ovo', C=1.0, gamma='scale', random_state=42)

开始训练模型...

训练完成。耗时: 99.03 秒

开始在测试集上评估...

预测完成。耗时: 34.50 秒

准确率 (Accuracy): 0.8946

生成分类报告...

分类报告:

precision recall f1-score support

0 0.91 0.95 0.93 980

1 0.96 0.99 0.97 1135

2 0.84 0.86 0.85 1032

3 0.86 0.89 0.87 1010

4 0.90 0.91 0.91 982

5 0.87 0.83 0.85 892

6 0.91 0.90 0.90 958

7 0.91 0.88 0.89 1028

8 0.90 0.86 0.88 974

9 0.88 0.87 0.87 1009

accuracy 0.89 10000

macro avg 0.89 0.89 0.89 10000

weighted avg 0.89 0.89 0.89 10000

生成混淆矩阵...

正在进行实验: Kernel=sigmoid, Strategy=ovr

创建 SVC 模型 (kernel='sigmoid', decision_function_shape='ovr', C=1.0, gamma='scale', random_state=42)

开始训练模型...

训练完成。耗时: 99.26 秒

开始在测试集上评估...

预测完成。耗时: 34.74 秒

准确率 (Accuracy): 0.8946

生成分类报告...

分类报告:

precision recall f1-score support

0 0.91 0.95 0.93 980

1 0.96 0.99 0.97 1135

2 0.84 0.86 0.85 1032

3 0.86 0.89 0.87 1010

4 0.90 0.91 0.91 982

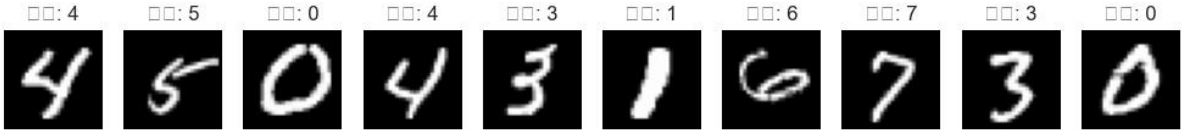
5 0.87 0.83 0.85 892

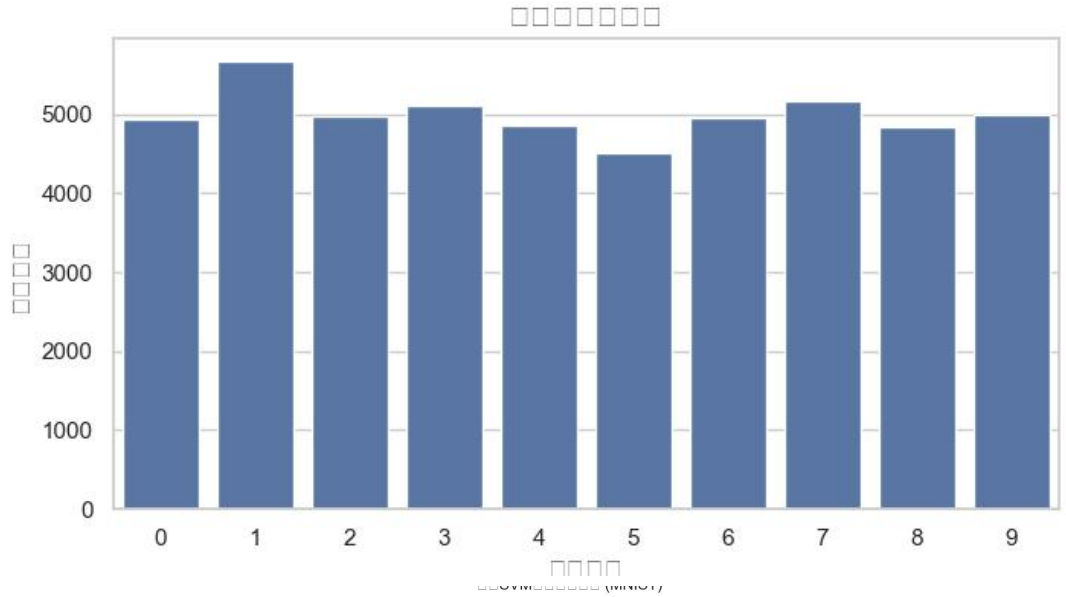
```
     6           0.91      0.90      0.90       958
     7           0.91      0.88      0.89      1028
     8           0.90      0.86      0.88       974
     9           0.88      0.87      0.87      1009
     accuracy                                  0.89     10000
     macro avg       0.89      0.89      0.89     10000
     weighted avg       0.89      0.89      0.89
10000

     生成混淆矩阵...

     --- 所有实验完成 ---

     --- 实验结果汇总 (表格) ---

     Experiment     Kernel  ... Test Time (s)  Accuracy
     0          Kernel=linear, Strategy=ovo          linear  ...
21.6983      0.9260
     1          Kernel=linear, Strategy=ovr          linear  ...
22.0001      0.9260
     2             Kernel=poly, Strategy=ovo            poly  ...
50.4199      0.9576
     3             Kernel=poly, Strategy=ovr            poly  ...
50.4988      0.9576
     4              Kernel=rbf, Strategy=ovo             rbf  ...
61.6389      0.9644
     5              Kernel=rbf, Strategy=ovr             rbf  ...
57.7263      0.9644
     6          Kernel=sigmoid, Strategy=ovo         sigmoid  ...
34.5014      0.8946
     7          Kernel=sigmoid, Strategy=ovr         sigmoid  ...
34.7404      0.8946
     [8 rows x 6 columns]

     --- 最佳模型 (基于最高准确率) ---

     实验名称: Kernel=rbf, Strategy=ovo

     准确率: 0.9644

     训练时间: 160.15 秒

     测试时间: 61.64 秒

     最佳模型 - 分类报告:
```

```
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       980
           1       0.99      0.99      0.99      1135
           2       0.95      0.97      0.96      1032
           3       0.97      0.97      0.97      1010
           4       0.97      0.96      0.97       982
           5       0.96      0.95      0.96       892
           6       0.98      0.97      0.97       958
           7       0.93      0.96      0.94      1028
           8       0.96      0.95      0.95       974
           9       0.97      0.93      0.95      1009

    accuracy                           0.96     10000
   macro avg       0.96      0.96      0.96     10000
weighted avg       0.96      0.96      0.96     10000

--- 性能指标可视化 ---

--- 最佳模型混淆矩阵可视化 ---

混淆矩阵中最常见的5个错误 (真实标签 -> 预测标签: 数量):

7 -> 9: 13 次

7 -> 2: 13 次

9 -> 4: 14 次

2 -> 7: 15 次

9 -> 7: 20 次

--- 实验代码执行完毕 ---


进程已结束，退出代码为 0
```

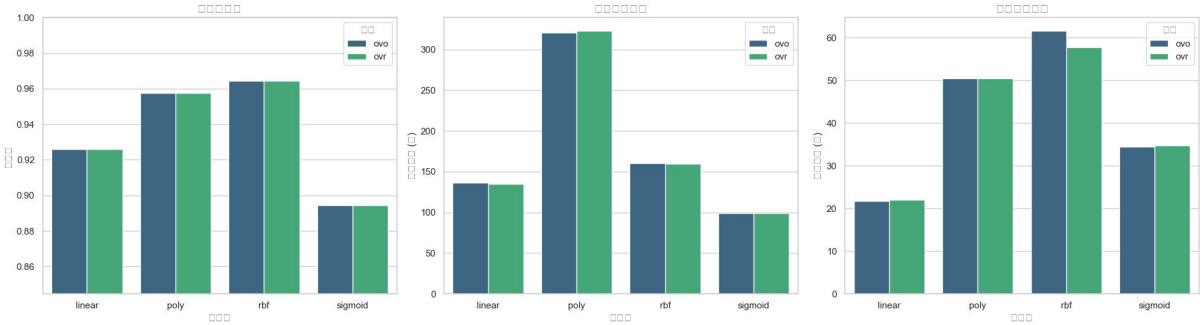
实验可视化截图:

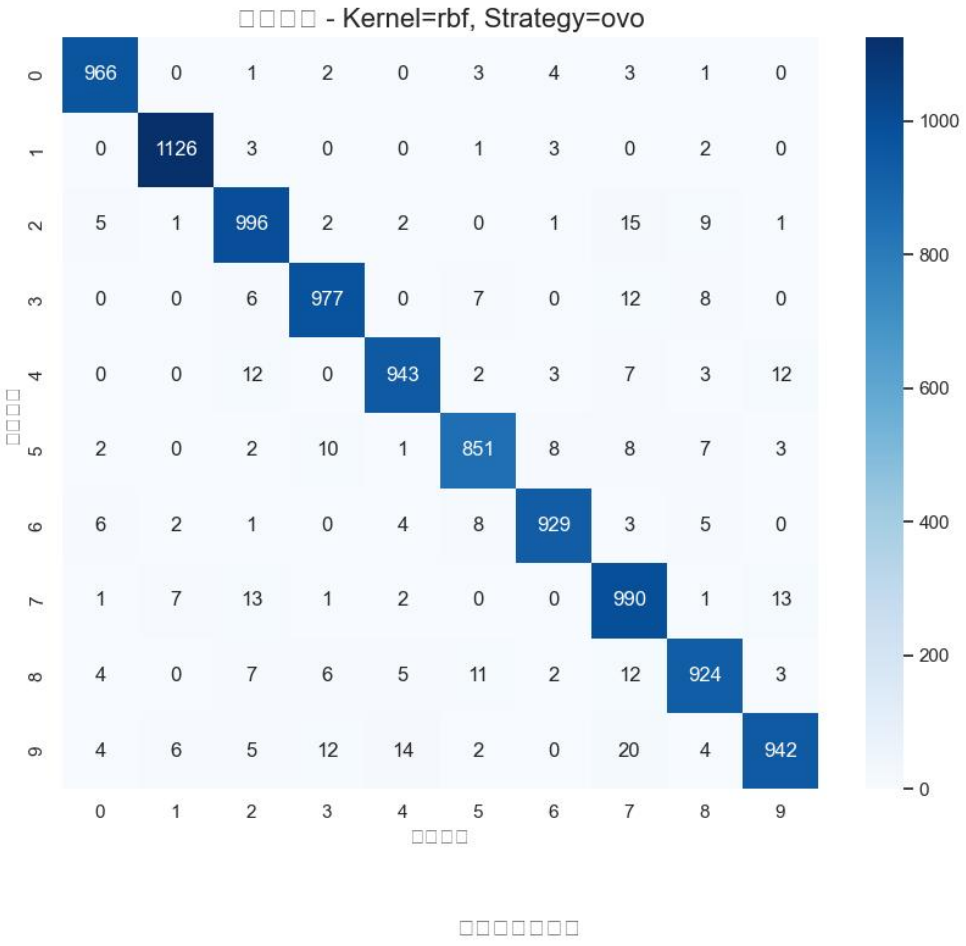
□□□□□□□


□□: 4
□□: 5
□□: 0
□□: 4
□□: 3
□□: 1
□□: 6
□□: 7
□□: 3
□□: 0


□□□□□□□
□□□□
5000
4000
3000
2000
1000
0
0
1
2
3
4
5
6
7
8
9
□□□□

□□SVM□□□□□□ (MNIST)


□□□□□
□□
ovo
ovr
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
linear
poly
rbf
sigmoid
□□□□□□
□□□□ (□)
300
250
200
150
100
50
0
□□□□□□
60
50
40
30
20
10



6.实验心得

本次实验让我对支持向量机（SVM）在图像分类任务中的应用有了深入的实践理解。通过亲手操作 MNIST 数据集的加载、预处理、模型训练和评估全过程，我对 scikit-learn 库的使用更加熟练，特别是 SVC 分类器的不

同参数设置。

深刻体会到了特征缩放对于 SVM 的重要性。从预处理结果看到，StandardScaler 成功将特征数据的均值调整到接近 0，标准差接近 1（实际为 0.96），这为 SVM 构建最优超平面提供了良好的基础，避免了不同像素特征值范围差异过大带来的干扰。

直观地比较了不同核函数的性能差异。实验清晰地展示了线性核、多项式核、RBF 核和 Sigmoid 核在同一数据集上的表现，让我理解到没有 universally best 的核函数，选择取决于数据的内在结构。

理解了多分类策略（OVO 与 OVR）在 SVC 中的实现机制。实验结果显示，对于 sklearn.svm.SVC，选择 decision_function_shape='ovo'或'ovr'对最终的准确率和训练时间几乎没有影响，印证了其内部训练机制主要是基于 OVO，该参数更多影响的是决策函数的输出形式和最终预测阶段的细微差异（体现在测试时间上略有不同）。

掌握了多种模型评估指标的解读。除了整体准确率，分类报告（precision, recall, f1-score）和混淆矩阵能提供更细致的模型性能视图，帮助分析模型在哪些类别上表现好，哪些类别容易混淆。

结果分析:

准确率对比:

RBF 核 表现最佳，达到了 96.44% 的准确率。这表明 MNIST 手写数字的特征空间具有高度非线性，RBF 核通过其高斯函数能够有效地将样本映射到高维空间，找到良好的分类边界。

多项式核 (Poly) 准确率也相当高，为 95.76%，仅次于 RBF 核。这说明

数据特征之间可能存在一定的多项式关系，默认的三次多项式核已经能较好地拟合数据。

线性核 (Linear) 准确率为 92.60%。虽然不如非线性核，但超过 92%的准确率说明数据在原始特征空间中仍具有一定的线性可分性，线性核提供了一个快速且不错的基线性能。

Sigmoid 核 表现最差，准确率为 89.46%。这符合预期，因为 Sigmoid 核在很多情况下表现不稳定，其性能对参数（gamma, coef0）非常敏感，且不一定满足 Mercer 定理（不是严格的正定核）。

时间效率对比:

训练时间: 多项式核训练耗时最长（约 320-323 秒），其次是 RBF 核（约 160 秒），然后是线性核（约 135 秒），最快的是 Sigmoid 核（约 99 秒）。这大致反映了核函数计算的复杂度，但 Sigmoid 核的快速可能也与其未能充分拟合有关。

测试时间: RBF 核测试时间最长（约 58-62 秒），其次是多项式核（约 50 秒），然后是 Sigmoid 核（约 34 秒），线性核测试最快（约 22 秒）。测试时间主要取决于支持向量的数量以及计算核函数或决策函数的复杂度。

OVO vs OVR 策略：结果明确显示，在 SVC 中使用 decision_function_shape='ovo'或'ovr'，对每个核函数的准确率完全没有影响。训练时间也极其接近（差异在 1-2 秒内，可能由系统波动引起），测试时间有微小差异（RBF-OVR 略快于 RBF-OVO，其他核函数差异不大）。这证实了 SVC 底层训练是基于 OVO 的，decision_function_shape 主要影响输出接口，对整体性能影响甚微。

最佳模型分析 (RBF Kernel, OVO/OVR):

该模型在数字0和1上表现近乎完美（precision/recall 均达到0.98-0.99）。

从分类报告看，数字 7 的精确率(0.93) 和 数字 9 的召回率(0.93) 相对较低，说明模型在预测为 7 的样本中有较多实际不是 7 的（混淆了其他数字如 2、9），以及有一部分实际为 9 的样本被错误预测为了其他数字（如 4、7）。

混淆矩阵分析（最常见的 5 个错误）进一步印证了这一点：9 和 7 之间、7 和 2 之间、9 和 4 之间的混淆最为常见。这符合人类识别时也容易混淆这些形状相似的数字的直觉。

总结: 综合来看，对于 MNIST 手写数字识别任务，在本次实验的默认参数设置下，RBF 核函数在准确率和性能之间取得了最佳平衡（虽然训练和测试时间不是最快的，但准确率最高）。如果极其看重预测速度，线性核是一个牺牲部分精度换取速度的选择。多项式核表现也不错，但训练更耗时。Sigmoid 核不适合此任务。

局限性与未来工作:

本次实验未使用验证集进行超参数调优（如调整 C 和 gamma，以及 poly 核的 degree）。通过网格搜索（GridSearchCV）或随机搜索（RandomizedSearchCV）优化这些参数，很可能会进一步提升模型的准确率，特别是 RBF 核和 Poly 核。

预处理相对简单，仅使用了标准化。可以尝试更高级的图像特征提取方法（如 HOG、LBP）或降维技术（如 PCA），看是否能改善 SVM 性能或效率。

可以与其他经典的机器学习算法（如逻辑回归、随机森林、K 近邻）或

深度学习模型（如简单的 MLP、CNN）进行性能对比。