更高效的支持向量机算法实现及其在手写数字识别中的应用——03不同Kernel的SVM超参数优化

大数据机器学习课程第二次实验项目

叶璨铭 (2024214500) ycm24@mails.tsinghua.edu.cn

2024年11月11日星期一

目录

本文是[更高效的支持向量机算法实现及其在手写数字识别中的应用](./00svm.html)系列文章第03篇——不同Kernel的SVM超参数优化。

dataset\_dict\_uci\_digits = load\_digits(as\_frame=False)  
X\_train, X\_val, X\_test, y\_train, y\_val, y\_test, train\_set, val\_set, test\_set, data\_module, categories = process\_sklearn\_dataset\_dict(dataset\_dict\_uci\_digits, 'all')  
# dataset\_dict\_full\_mnist = fetch\_openml("mnist\_784", as\_frame=True)  
# X\_train\_full, X\_val\_full, X\_test\_full, y\_train\_full, y\_val\_full, y\_test\_full, train\_set\_full, val\_set\_full, test\_set\_full, data\_module\_full, categories\_full = process\_sklearn\_dataset\_dict(dataset\_dict\_full\_mnist, 'all')

(1797, 64) float32 (1797,) int64 [0 1 2 3 4 5 6 7 8 9]  
1293 144 360

### 0.1 附加题: 对比不同 kernel 方法下的 SVM 分类器 （对完整SVM进行调参）

这一题本质上是让我们以 kernel 的选择（也包括选择线性Kernel）作为目标元参数，其他参数作为冗余或固定元参数，进行调参实验，发现不同 kernel 方法下的 SVM 分类器的分类效果数值上的区别及其显著性，并且从可视化分析上也作出进一步解释。

为了让问题简单清晰，我们再这一节不用我们自己实现的SVM（我们刚才实现的Linear Soft Margin SGD SVM），而是直接使用成熟的经过检验的，刚才我们测试出来最快的 Thunder SVM 库（与sklearn接口兼容，速度更快，使用GPU）。这样我们就可以专注于调参优化问题本身，而不是关注具体实现细节。

#### 0.1.1 科学调参的原则与方法

我在上次Project作业KD树中详细描述了谷歌AI团队《深度学习调优指南》的思想，涉及到的概念包括目标元参数、冗余元参数和固定元参数，贝叶斯优化、演化计算、近似随机搜索，科学实验的控制变量法与调参实验设计中的探索与利用、调参结果的假设检验分析等。这里我们不再赘述，需要的话可以阅读[上次作业的文档](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/notebooks/coding_projects/P1_KNN/kd_tree.ipynb)。

#### 0.1.2 搜索空间定义

我们使用dataclass，要求传入函数的参数是强类型，而且有一个随机概率分布，这样方便定义调参。这里用到我自己写的scholarly\_infrastructure库的一个核心功能，对Python标准的dataclass进行了改进。

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/svm/kernel_hpo.py#L19)

### 0.2 SupportVectorClassifierConfig

SupportVectorClassifierConfig (C:float=1.0, kernel:str='rbf',  
 degree:int=3,  
 gamma:Union[str,float]='scale',  
 coef0:float=0.0, shrinking:bool=True,  
 probability:bool=False, tol:float=0.001,  
 cache\_size:float=200, class\_weight:Union[d  
 ict,str,NoneType]=None,  
 verbose:bool=False, max\_iter:int=-1,  
 decision\_function\_shape:str='ovr',  
 break\_ties:bool=False,  
 random\_state:Optional[int]=None)

from scholarly\_infrastructure.rv\_args.nucleus import RandomVariable, experiment\_setting  
from optuna.distributions import IntDistribution, FloatDistribution, CategoricalDistribution  
from typing import Optional, Union

@experiment\_setting  
class SupportVectorClassifierConfig:  
 # 惩罚系数 C  
 C: float = ~RandomVariable(  
 default=1.0,  
 description="Regularization parameter. The strength of the regularization is inversely proportional to C.",  
 distribution=FloatDistribution(1e-5, 1e2, log=True)  
 )  
   
 # 核函数类型  
 kernel: str = ~RandomVariable(  
 default="rbf",  
 description="Kernel type to be used in the algorithm.",  
 distribution=CategoricalDistribution(choices=["linear", "poly", "rbf", "sigmoid",  
 "precomputed"  
 ])  
 )  
   
 # 多项式核函数的度数  
 degree: int = ~RandomVariable(  
 default=3,  
 description="Degree of the polynomial kernel function ('poly').",  
 distribution=IntDistribution(1, 10, log=False)  
 )  
   
 # 核函数系数 gamma  
 gamma: Union[str, float] = ~RandomVariable(  
 default="scale",  
 description="Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.",  
 distribution=CategoricalDistribution(choices=["scale", "auto"]) # 可以添加浮点数分布视需求  
 )  
   
 # 核函数独立项 coef0  
 coef0: float = ~RandomVariable(  
 default=0.0,  
 description="Independent term in kernel function. It is significant in 'poly' and 'sigmoid'.",  
 distribution=FloatDistribution(0, 1)  
 )  
   
 # 收缩启发式算法  
 shrinking: bool = ~RandomVariable(  
 default=True,  
 description="Whether to use the shrinking heuristic.",  
 distribution=CategoricalDistribution(choices=[True, False])  
 )  
   
 # 是否启用概率估计  
 probability: bool = ~RandomVariable(  
 default=False,  
 description="Whether to enable probability estimates. Slows down fit when enabled.",  
 distribution=CategoricalDistribution(choices=[True, False])  
 )  
   
 # 停止准则的容差 tol  
 tol: float = ~RandomVariable(  
 default=1e-3,  
 description="Tolerance for stopping criterion.",  
 distribution=FloatDistribution(1e-6, 1e-1, log=True)  
 )  
   
 # 内核缓存的大小（MB）  
 cache\_size: float = ~RandomVariable(  
 default=200,  
 description="Specify the size of the kernel cache (in MB).",  
 distribution=FloatDistribution(50, 500, log=False)  
 )  
   
 # 类别权重 class\_weight  
 class\_weight: Optional[Union[dict, str]] = ~RandomVariable(  
 default=None,  
 description="Set C of class i to class\_weight[i]\*C or use 'balanced' to adjust weights inversely to class frequencies.",  
 distribution=CategoricalDistribution(choices=[None, "balanced"])  
 )  
   
 # 是否启用详细输出  
 verbose: bool = ~RandomVariable(  
 default=False,  
 description="Enable verbose output (may not work properly in a multithreaded context).",  
 distribution=CategoricalDistribution(choices=[True, False])  
 )  
   
 # 最大迭代次数  
 max\_iter: int = ~RandomVariable(  
 default=-1,  
 description="Hard limit on iterations within solver, or -1 for no limit.",  
 distribution=IntDistribution(-1, 1000, log=False)  
 )  
   
 # 决策函数形状  
 decision\_function\_shape: str = ~RandomVariable(  
 default="ovr",  
 description="Whether to return a one-vs-rest ('ovr') decision function or original one-vs-one ('ovo').",  
 distribution=CategoricalDistribution(choices=["ovo", "ovr"])  
 )  
   
 # 是否打破决策函数平局  
 break\_ties: bool = ~RandomVariable(  
 default=False,  
 description="If True, break ties according to the confidence values of decision\_function when decision\_function\_shape='ovr'.",  
 distribution=CategoricalDistribution(choices=[True, False])  
 )  
   
 # 随机种子 random\_state  
 random\_state: Optional[int] = ~RandomVariable(  
 default=None,  
 description="Controls random number generation for probability estimates. Ignored when probability=False.",  
 distribution=IntDistribution(0, 100) # 根据需求设置范围  
 )

# show\_dataframe\_doc(SupportVectorClassifierConfig)[:1]  
SupportVectorClassifierConfig.show\_dataframe\_doc()[:1]  
# SupportVectorClassifierConfig.get\_optuna\_search\_space(frozen\_rvs={"verbose", "cache\_size", "random\_state"})

|  | name | type | default | default\_factory | init | repr | hash | compare | metadata | kw\_only | description | distribution |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | C | <class 'float'> | 1.0 | <dataclasses.\_MISSING\_TYPE object at 0x7faee47... | True | True | None | True | None | <dataclasses.\_MISSING\_TYPE object at 0x7faee47... | Regularization parameter. The strength of the ... | FloatDistribution(high=100.0, log=True, low=1e... |

from dataclasses import asdict

#### 0.2.1 定义目标函数

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/svm/kernel_hpo.py#L135)

### 0.3 evaluate\_svm

evaluate\_svm (config:\_\_main\_\_.SupportVectorClassifierConfig, X\_train,  
 y\_train, trial:optuna.trial.\_trial.Trial=None,  
 critical\_metric='acc1\_pred', num\_of\_repeated=5)

import optuna  
from sklearn.svm import SVC  
from sklearn.model\_selection import KFold

def evaluate\_svm(config:SupportVectorClassifierConfig, X\_train, y\_train,  
 trial:optuna.Trial = None,   
 critical\_metric="acc1\_pred", num\_of\_repeated=5):  
 # 使用k fold交叉验证，相当于做了5次独立实验。  
 kf = KFold(n\_splits=num\_of\_repeated, shuffle=True, random\_state=config.random\_state)  
   
   
 result\_dict = dict()  
  
 metric\_names = set()  
   
 # 进行5折交叉验证  
 for experiment\_index, (train\_index, test\_index) in enumerate(kf.split(X\_train)):  
 # 分割训练集和测试集  
 X\_train\_fold, X\_test\_fold = X\_train[train\_index], X\_train[test\_index]  
 y\_train\_fold, y\_test\_fold = y\_train[train\_index], y\_train[test\_index]  
   
 # 创建分类器实例  
 model = SVC(\*\*asdict(config))  
   
 # 训练模型  
 model.fit(X\_train\_fold, y\_train\_fold)  
   
 # 预测测试集  
 y\_pred = model.predict(X\_test\_fold)  
   
 # 计算准确率  
 single\_run\_result = compute\_classification\_metrics(y\_test\_fold, y\_pred=y\_pred, labels=list(range(10)), y\_pred\_metrics\_only=True)  
   
 metric\_names.update(single\_run\_result.keys())  
 single\_run\_result = {f"{k}-run{experiment\_index}":v for k, v in single\_run\_result.items()}  
 result\_dict|=single\_run\_result  
   
 if trial is not None:  
 for k, v in single\_run\_result.items():  
 trial.set\_user\_attr(k, v)  
 trial.report(single\_run\_result[f"{critical\_metric}-run{experiment\_index}"], experiment\_index)  
 for metric\_name in metric\_names:  
 all\_runs\_results = [result\_dict[f"{metric\_name}-run{i}"] for i in range(num\_of\_repeated)]  
 result\_dict[f"{metric\_name}-mean"] = sum(all\_runs\_results) / len(all\_runs\_results)  
 if trial is not None:  
 trial.set\_user\_attr(f"{metric\_name}-mean", result\_dict[f"{metric\_name}-mean"])  
 if trial is not None:  
 trial.set\_user\_attr(f"num\_of\_repeated", num\_of\_repeated)  
 return result\_dict

import numpy as np

# 把 X\_train 和 X\_val 重新合并   
X\_train\_val = np.vstack((X\_train, X\_val))  
y\_train\_val = np.hstack((y\_train, y\_val))

evaluate\_svm(SupportVectorClassifierConfig(kernel='linear'), X\_train\_val, y\_train\_val).keys()

dict\_keys(['matthews\_corrcoef-run0', 'f1-run0', 'precision-run0', 'recall-run0', 'balanced\_accuracy-run0', 'cohen\_kappa-run0', 'acc1\_pred-run0', 'matthews\_corrcoef-run1', 'f1-run1', 'precision-run1', 'recall-run1', 'balanced\_accuracy-run1', 'cohen\_kappa-run1', 'acc1\_pred-run1', 'matthews\_corrcoef-run2', 'f1-run2', 'precision-run2', 'recall-run2', 'balanced\_accuracy-run2', 'cohen\_kappa-run2', 'acc1\_pred-run2', 'matthews\_corrcoef-run3', 'f1-run3', 'precision-run3', 'recall-run3', 'balanced\_accuracy-run3', 'cohen\_kappa-run3', 'acc1\_pred-run3', 'matthews\_corrcoef-run4', 'f1-run4', 'precision-run4', 'recall-run4', 'balanced\_accuracy-run4', 'cohen\_kappa-run4', 'acc1\_pred-run4', 'balanced\_accuracy-mean', 'f1-mean', 'acc1\_pred-mean', 'recall-mean', 'precision-mean', 'matthews\_corrcoef-mean', 'cohen\_kappa-mean'])

固定元参数定义

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/svm/kernel_hpo.py#L192)

### 0.4 objective\_svm

objective\_svm (trial:optuna.trial.\_trial.Trial, X\_train\_val, y\_train\_val,  
 num\_of\_repeated=5, critical\_metric='acc1\_pred',  
 critical\_reduction='mean')

fixed\_meta\_params = SupportVectorClassifierConfig(  
 probability = False, # 暂时不研究，只关注 acc1\_pred  
 # 与性能无关  
 cache\_size = 200,   
 verbose = False,  
 random\_state = 42, # 今天我们根据 K Fold 来做重复实验，不根据随机种子来做重复实验  
)  
frozen\_rvs = {"probability", "cache\_size", "verbose", "random\_state"}

def objective\_svm(trial:optuna.Trial, X\_train\_val, y\_train\_val, num\_of\_repeated=5, critical\_metric="acc1\_pred", critical\_reduction="mean"):  
 config:SupportVectorClassifierConfig = SupportVectorClassifierConfig.optuna\_suggest(  
 trial, fixed\_meta\_params, frozen\_rvs=frozen\_rvs)  
 try:  
 cross\_val\_results = evaluate\_svm(config, X\_train\_val, y\_train\_val,   
 trial=trial,  
 critical\_metric=critical\_metric, num\_of\_repeated=num\_of\_repeated)  
 critical\_metric\_name = f"{critical\_metric}-{critical\_reduction}"  
 critical\_result = cross\_val\_results[critical\_metric\_name]  
 except ValueError as e:  
 # logger.exception(e)  
 logger.warning(f"Trial {trial.number} failed with error: {e}, we consider this as a pruned trial since we believe such failure is due to the implicit constraints of the problem. ")  
 raise optuna.exceptions.TrialPruned()  
 return critical\_result

#### 0.4.1 执行调参搜索

from thu\_big\_data\_ml.help import runs\_path

study\_path = runs\_path / "optuna\_studies.db"  
sqlite\_url = f"sqlite:///{study\_path}"

from optuna.samplers import \*  
from optuna.pruners import \*  
import json

study = optuna.create\_study(  
 study\_name="svm kernel hpo 11.17 3.0",   
 storage=sqlite\_url,   
 load\_if\_exists=True,   
 sampler=QMCSampler(seed=42), # 谷歌建议  
 pruner=WilcoxonPruner(), # 对重复实验进行假设检验剪枝  
 direction="maximize")  
study.set\_user\_attr("contributors", ["Ye Canming"])  
study.set\_user\_attr("fixed\_meta\_parameters", json.dumps(asdict(fixed\_meta\_params)))

ExperimentalWarning:  
  
QMCSampler is experimental (supported from v3.0.0). The interface can change in the future.  
  
<ipython-input-1-ad5a12a62694>:6: ExperimentalWarning:  
  
WilcoxonPruner is experimental (supported from v3.6.0). The interface can change in the future.

接下来运行

# %%capture cap  
study.optimize(objective\_svm, n\_trials=100)

study.optimize(objective\_svm, n\_trials=100)

#### 0.4.2 分析实验结果

首先加载调参实验的结果，实验分析的时候，不用重新再跑一次实验，这两个是分开的

study = optuna.load\_study(  
 study\_name="svm kernel hpo 11.17 3.0",   
 storage=sqlite\_url)

##### 0.4.2.1 最好的模型是哪个？

study.best\_params

{  
 'C': 1.7782794100389216,  
 'kernel': 'poly',  
 'degree': 3,  
 'gamma': 'auto',  
 'coef0': 0.25,  
 'shrinking': False,  
 'tol': 1.7782794100389246e-05,  
 'class\_weight': None,  
 'max\_iter': 750,  
 'decision\_function\_shape': 'ovo',  
 'break\_ties': False  
}

study.best\_value

0.9902584204413472

##### 0.4.2.2 Optuna 可视化分析结果

可以通过下面的命令对我们刚才保存调参结果的数据库文件进行可视化展示：

optuna-dashboard sqlite:///optuna\_studies.db --port 18081

我们也可以把这些图用代码画在本文档中

from optuna.visualization import plot\_contour  
from optuna.visualization import plot\_edf  
from optuna.visualization import plot\_intermediate\_values  
from optuna.visualization import plot\_optimization\_history  
from optuna.visualization import plot\_parallel\_coordinate  
from optuna.visualization import plot\_param\_importances  
from optuna.visualization import plot\_rank  
from optuna.visualization import plot\_slice  
from optuna.visualization import plot\_timeline

plot\_param\_importances(study)

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

这个图非常有用，提示了我们元参数的重要性，这其实也是元参数的敏感性。如果目标值对某个参数过于敏感，那么它必须是冗余元参数，在其他变量变动的时候需要重新调参，而如果不是很敏感，我们就可以先固定它的值缩小搜索范围。

从图中可以看出，kernel是最重要的选择。

max\_iter 很重要的话就意味着我们给的搜索空间max\_iter可能还不够，有些方法潜在地需要更多的迭代次数才能收敛。在深度学习调参项目中，我们下一轮的实验迭代可能就要考虑修改搜索空间。这里我们只是做作业，所以暂时忽略。tol也很重要意味着其实我们tol可以给得更小，说明更小对优化精度是有用的。

网上的资料指出，C和gamma的选择对SVM的性能影响很大，然而在我们的200次实验观测中，可以看到gamma的影响并不是很大，至少在sklearn的两个自动选择gamma的方法来说，这两个自动选择都是合理的，可能网上的建议是说手动指定gamma的float会有很大影响。

plot\_intermediate\_values(study)

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

0-4是不同的K Fold下的结果。可以看出有些参数下不是那么稳定，特别是整体值比较差的的时候，随机波动也比较大。

plot\_parallel\_coordinate(study, params=["kernel", "C", "gamma"])

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

这个图其实我个人觉得意义不大，虽然很多调参软件都会有，看起来很炫酷，但是我觉得无法从图中读到什么有用的insight。因为好的值和坏的值图中交叉来交叉去，没法直接看出相关关系。

plot\_contour(study, params=["C", 'tol', 'coef0'])

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

等高线图，看起来C比较大的时候颜色更深、

plot\_slice(study)

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

这个图比较有用，可以看到每个随机变量不同值的影响。

plot\_rank(study, params=['C', 'tol', 'kernel'])

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

##### 0.4.2.3 以 kernel 为目标元参数进行假设检验分析——期望改进分析

df = study.trials\_dataframe()  
df.head()

|  | number | value | datetime\_start | datetime\_complete | duration | params\_C | params\_break\_ties | params\_class\_weight | params\_coef0 | params\_decision\_function\_shape | ... | user\_attrs\_precision-run2 | user\_attrs\_precision-run3 | user\_attrs\_precision-run4 | user\_attrs\_recall-mean | user\_attrs\_recall-run0 | user\_attrs\_recall-run1 | user\_attrs\_recall-run2 | user\_attrs\_recall-run3 | user\_attrs\_recall-run4 | state |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | NaN | 2024-11-17 02:54:45.626373 | 2024-11-17 02:54:45.788622 | 0 days 00:00:00.162249 | NaN | NaN | NaN | NaN | NaN | ... | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | FAIL |
| 1 | 1 | NaN | 2024-11-17 02:55:18.722161 | 2024-11-17 02:55:18.797816 | 0 days 00:00:00.075655 | NaN | NaN | NaN | NaN | NaN | ... | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | FAIL |
| 2 | 2 | 0.956165 | 2024-11-17 02:56:10.992341 | 2024-11-17 02:56:11.578482 | 0 days 00:00:00.586141 | 0.004186 | True | None | 0.708073 | ovr | ... | 0.961807 | 0.957685 | 0.964761 | 0.956196 | 0.951926 | 0.954759 | 0.954058 | 0.958502 | 0.961732 | COMPLETE |
| 3 | 3 | NaN | 2024-11-17 02:56:11.607921 | 2024-11-17 02:56:11.738309 | 0 days 00:00:00.130388 | 0.191854 | True | None | 0.000000 | ovr | ... | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | PRUNED |
| 4 | 4 | 0.749492 | 2024-11-17 02:56:11.754529 | 2024-11-17 02:56:12.693953 | 0 days 00:00:00.939424 | 0.031623 | False | None | 0.500000 | ovr | ... | 0.839718 | 0.859636 | 0.849986 | 0.761556 | 0.733830 | 0.775699 | 0.789870 | 0.766079 | 0.742302 | COMPLETE |

target = "value"  
treatment = "params\_kernel"

intersted\_cols = [c for c in df.columns if c.startswith("params") or "user\_attrs" in c or c=="value"]  
# intersted\_cols = [c for c in df.columns if c.startswith("params") or c=="value"]  
dfi = df[intersted\_cols].dropna(subset=[target])  
dfi.head()

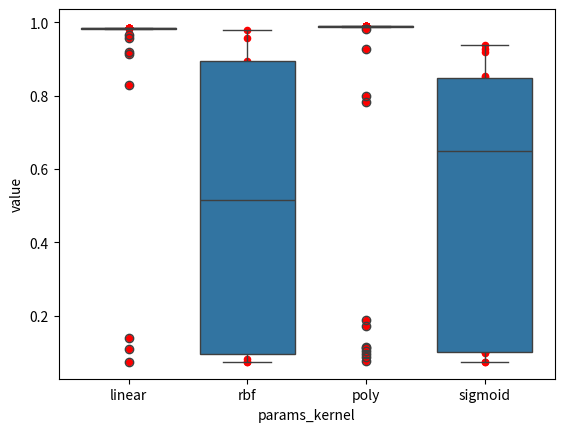
|  | value | params\_C | params\_break\_ties | params\_class\_weight | params\_coef0 | params\_decision\_function\_shape | params\_degree | params\_gamma | params\_kernel | params\_max\_iter | ... | user\_attrs\_precision-run1 | user\_attrs\_precision-run2 | user\_attrs\_precision-run3 | user\_attrs\_precision-run4 | user\_attrs\_recall-mean | user\_attrs\_recall-run0 | user\_attrs\_recall-run1 | user\_attrs\_recall-run2 | user\_attrs\_recall-run3 | user\_attrs\_recall-run4 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 0.956165 | 0.004186 | True | None | 0.708073 | ovr | 1.0 | scale | linear | 182.0 | ... | 0.947915 | 0.961807 | 0.957685 | 0.964761 | 0.956196 | 0.951926 | 0.954759 | 0.954058 | 0.958502 | 0.961732 |
| 4 | 0.749492 | 0.031623 | False | None | 0.500000 | ovr | 5.0 | scale | rbf | 499.0 | ... | 0.801343 | 0.839718 | 0.859636 | 0.849986 | 0.761556 | 0.733830 | 0.775699 | 0.789870 | 0.766079 | 0.742302 |
| 5 | 0.990258 | 1.778279 | False | None | 0.250000 | ovo | 3.0 | auto | poly | 750.0 | ... | 0.985863 | 0.988889 | 0.981481 | 1.000000 | 0.989942 | 0.991154 | 0.986591 | 0.989747 | 0.982219 | 1.000000 |
| 6 | 0.130139 | 0.000562 | False | balanced | 0.750000 | ovr | 8.0 | scale | sigmoid | 249.0 | ... | 0.013542 | 0.010105 | 0.010453 | 0.038927 | 0.120000 | 0.100000 | 0.100000 | 0.100000 | 0.100000 | 0.200000 |
| 8 | 0.984688 | 13.335214 | False | balanced | 0.125000 | ovo | 9.0 | scale | linear | 875.0 | ... | 0.990013 | 0.982012 | 0.975091 | 0.993521 | 0.984161 | 0.982063 | 0.988435 | 0.981054 | 0.975855 | 0.993398 |

import matplotlib.pyplot as plt  
import seaborn as sns

fig, ax = plt.subplots()  
sns.boxplot(data=dfi, x=treatment, y=target, ax=ax)  
dfi.plot(x=treatment, y=target, ax=ax, kind='scatter', c='red')

uniform与distance方法平均准确率箱线图对比

<Axes: xlabel='params\_kernel', ylabel='value'>

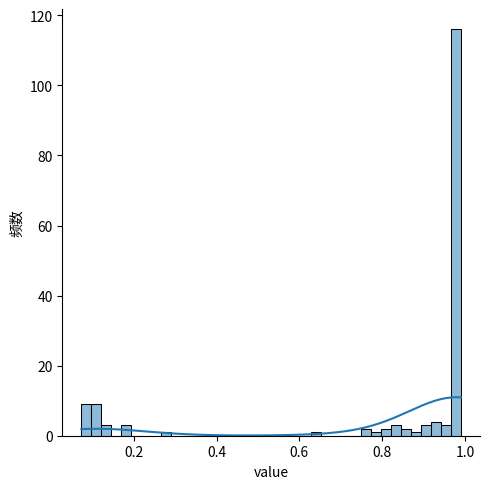


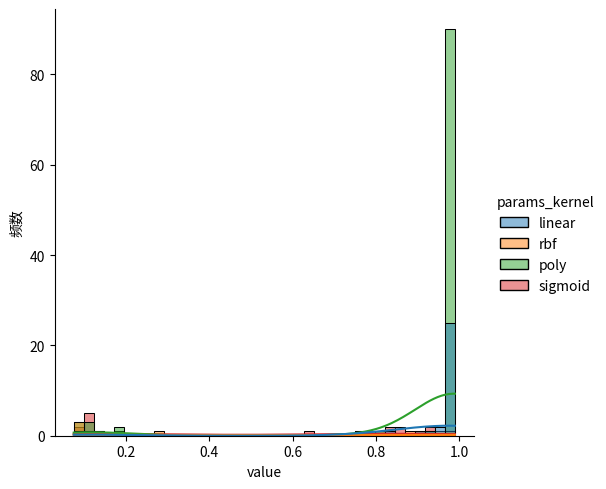
从中可以看出, linear和 poly 平均值比较高，而且太集中了画不出图。 而 rbf 和 sigmoid 则是有很多结果比较低，平均值在中间。

除了箱线图，还可以画分布图，来展示不同kernel对精度的影响。

sns.displot(data=dfi, x=target, kde=True, kind="hist")  
plt.ylabel("频数")  
sns.displot(data=dfi, x=target, hue=treatment, kde=True, kind="hist")  
plt.ylabel("频数")

Text(22.89317438271604, 0.5, '频数')





我们在同时比较多个方法，可以用ANOVA检验吗？ 可以使用我之前写的代码，看看ANOVA的条件是否满足。

from thu\_big\_data\_ml.big\_data\_analytics.anova import test\_normality\_group, homogeneity\_of\_variance

test\_normality\_group(dfi, interesting\_col=target, hue\_col=treatment)

|  | linear | poly | rbf | sigmoid |
| --- | --- | --- | --- | --- |
| Kolmogorov-Smirnov | Not Normal | Not Normal | Not Normal | Not Normal |
| Anderson-Darling | Not Normal | Not Normal | Not Normal | Not Normal |
| D'Agostino-Pearson | Not Normal | Not Normal | Not Normal | Not Normal |

log\_transform = lambda data\_column: np.log(data\_column+np.min(data\_column)+1) # 保证 大于0   
test\_normality\_group(dfi, interesting\_col=target, hue\_col=treatment, transform=log\_transform)

|  | linear | poly | rbf | sigmoid |
| --- | --- | --- | --- | --- |
| Kolmogorov-Smirnov | Not Normal | Not Normal | Not Normal | Not Normal |
| Anderson-Darling | Not Normal | Not Normal | Not Normal | Not Normal |
| D'Agostino-Pearson | Not Normal | Not Normal | Not Normal | Not Normal |

homogeneity\_of\_variance(dfi, interesting\_col=target, hue\_col=treatment)

The variances may be homogeneous, according to rule of thumb.  
Reject the null hypothesis of equal variances!

{  
 'ratio\_largest\_to\_smallest': 1.7386168726213924,  
 'bartlett\_result': BartlettResult(statistic=12.750075189817064, pvalue=0.005209469719008144)  
}

看来我们的数据并不服从正态分布，就算经过了log变换也不服从。 而且bartlett假设检验显著地发现方差齐性不满足，虽然经验法则说勉强可以接受<2的方差差异。

from thu\_big\_data\_ml.big\_data\_analytics.anova import auto\_anova\_for\_df, auto\_kruskal\_for\_df

auto\_anova\_for\_df(dfi, interesting\_col=target, hue\_col=treatment)

|  | Source | Sum of Squares (SS) | Degrees of Freedom (df) | Mean Square (MS) | F | p | p\_excel |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | Between | 4.040434 | 3 | 1.346811 | 18.119793 | 3.648672e-10 | 4.007048e-10 |
| 1 | Within | 11.818181 | 159 | 0.074328 | NaN | NaN | NaN |
| 2 | Total | 15.858614 | 162 | NaN | NaN | NaN | NaN |

尽管ANOVA的使用条件不满足，但是我们勉强使用ANOVA，可以看到ANOVA方法的p值很小，很显著地不同kernel之间存在显著差异！

接下来我们使用 Kruskal-Wallis 检验，这个检验不需要方差齐性和正态性满足，就可以对多列样本进行比较

res = auto\_kruskal\_for\_df(dfi, interesting\_col=target, hue\_col=treatment)  
if res.pvalue < 0.05:  
 print("There is a significant difference between the kernel functions.")  
res

Mon 2024-11-18 21:43:36.212606

INFO There is a significant difference between the kernel functions. d=143532;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=188497;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

KruskalResult(statistic=89.11831011539, pvalue=3.3878339514086004e-19)

Kruskal-Wallis H-test tests的零假设是 不同组的中位数之间没有显著差异。看来我们可以拒绝原假设，认为不同kernel的SVM的中位数有显著差异。

刚才我们检验出来存在显著差异，但是不知道具体是谁比谁大有显著性，所以我们还需要使用post-hoc类型的假设检验来进行进一步的分析。

首先刚才检验的是中位数有显著差异，我们来求一下分组的中位数。

import pandas as pd

medians = df.groupby(treatment)[target].median().sort\_values(ascending=False)  
medians

params\_kernel  
poly 0.989559  
linear 0.985383  
sigmoid 0.650010  
rbf 0.516373  
precomputed NaN  
Name: value, dtype: float64

order = medians.index[:-1]

Dunn’s test就是针对 Kruskal-Wallis one-way analysis 对应的 post-hoc 检验方法（之一，还有其他的）。

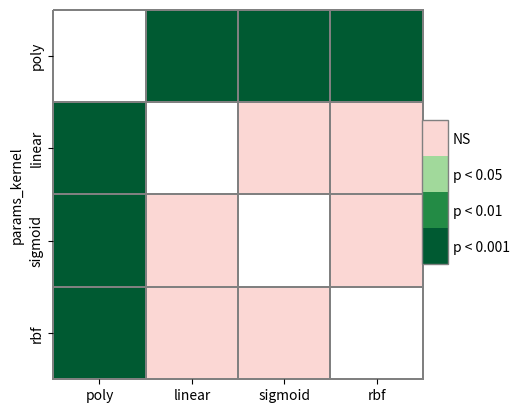
import scikit\_posthocs as sp  
from scikit\_posthocs import posthoc\_dunn

res = posthoc\_dunn(dfi, val\_col=target, group\_col=treatment,   
 sort=True, p\_adjust='holm')  
  
# 按照中位数大小排序  
res = res.reindex(order)  
res = res[order]  
res

|  | poly | linear | sigmoid | rbf |
| --- | --- | --- | --- | --- |
| params\_kernel |  |  |  |  |
| poly | 1.000000e+00 | 3.388027e-08 | 1.939419e-12 | 2.481610e-07 |
| linear | 3.388027e-08 | 1.000000e+00 | 6.615530e-02 | 1.597734e-01 |
| sigmoid | 1.939419e-12 | 6.615530e-02 | 1.000000e+00 | 9.446169e-01 |
| rbf | 2.481610e-07 | 1.597734e-01 | 9.446169e-01 | 1.000000e+00 |

# res < 0.05  
heatmap\_args = {'linewidths': 0.25, 'linecolor': '0.5', 'clip\_on': False, 'square': True, 'cbar\_ax\_bbox': [0.80, 0.35, 0.04, 0.3]}  
sp.sign\_plot(res, \*\*heatmap\_args)

(<Axes: ylabel='params\_kernel'>, <matplotlib.colorbar.Colorbar object at 0x7f3af82283a0>)



从结果表可以看出，poly非常显著地优于其他三个方法，是最好的方法。而linear虽然看起来中位数比剩下两个方法高，但是不显著。

以上的结果似乎不符合读者的直觉，你可能会说“kernel method不是很厉害吗，SVM重要的方法之一哎，怎么会不行呢？”

需要注意的是，我们刚才的实验随机采样了很多参数，kernel method也有厉害的参数，也有不厉害的参数。那我们刚才严格的假设检验如何解读呢？

正确的解读应该是这样的。假如有一个小白，不懂SVM，让他随机选择一个参数来调用sklearn库。我们是调过参的，我们告诉他，你是小白，如果你用rbf和sigmoid，你随便选别的参数，你很容易犯错误，让指标特别低。相反地，如果你用linear或者poly，你随便调其他的参数都无所谓，你大概率是能得到较好的结果的。

这就是我们上面的研究的意义，一个新的随机用户在随机场景下使用你的方法来替代其他方法时，他得到的期望改进。我们的结果就是 linear 和 poly的期望更高。但是这不意味着专家不能用rbf 和 sigmoid经过仔细调参得到更好的结果。

##### 0.4.2.4 以 kernel 为目标元参数进行假设检验分析——最优对比分析

max\_rows = dfi.loc[dfi.groupby(treatment)[target].idxmax()]  
max\_rows

|  | value | params\_C | params\_break\_ties | params\_class\_weight | params\_coef0 | params\_decision\_function\_shape | params\_degree | params\_gamma | params\_kernel | params\_max\_iter | ... | user\_attrs\_precision-run1 | user\_attrs\_precision-run2 | user\_attrs\_precision-run3 | user\_attrs\_precision-run4 | user\_attrs\_recall-mean | user\_attrs\_recall-run0 | user\_attrs\_recall-run1 | user\_attrs\_recall-run2 | user\_attrs\_recall-run3 | user\_attrs\_recall-run4 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 24 | 0.985383 | 60.429639 | True | balanced | 0.343750 | ovr | 6.0 | scale | linear | 656.0 | ... | 0.990013 | 0.985716 | 0.974978 | 0.990491 | 0.984965 | 0.985093 | 0.988435 | 0.984900 | 0.975855 | 0.990541 |
| 5 | 0.990258 | 1.778279 | False | None | 0.250000 | ovo | 3.0 | auto | poly | 750.0 | ... | 0.985863 | 0.988889 | 0.981481 | 1.000000 | 0.989942 | 0.991154 | 0.986591 | 0.989747 | 0.982219 | 1.000000 |
| 45 | 0.979820 | 77.736503 | False | balanced | 0.015625 | ovo | 4.0 | auto | rbf | 453.0 | ... | 0.981515 | 0.985716 | 0.971346 | 0.983036 | 0.979199 | 0.972401 | 0.983471 | 0.984758 | 0.971617 | 0.983750 |
| 57 | 0.937369 | 0.835363 | False | balanced | 0.171875 | ovr | 2.0 | scale | sigmoid | 797.0 | ... | 0.934272 | 0.942155 | 0.938151 | 0.939654 | 0.936771 | 0.942251 | 0.931733 | 0.938035 | 0.936699 | 0.935135 |

critical\_metric = "acc1\_pred"  
num\_repeated = 5  
kfold\_metrics = [f"user\_attrs\_{critical\_metric}-run{i}" for i in range(num\_repeated)]

intereting\_df = max\_rows[[treatment]+kfold\_metrics]  
intereting\_df = intereting\_df.set\_index(treatment)  
intereting\_df

|  | user\_attrs\_acc1\_pred-run0 | user\_attrs\_acc1\_pred-run1 | user\_attrs\_acc1\_pred-run2 | user\_attrs\_acc1\_pred-run3 | user\_attrs\_acc1\_pred-run4 |
| --- | --- | --- | --- | --- | --- |
| params\_kernel |  |  |  |  |  |
| linear | 0.986111 | 0.989583 | 0.986063 | 0.975610 | 0.989547 |
| poly | 0.993056 | 0.986111 | 0.989547 | 0.982578 | 1.000000 |
| rbf | 0.975694 | 0.982639 | 0.986063 | 0.972125 | 0.982578 |
| sigmoid | 0.940972 | 0.934028 | 0.940767 | 0.937282 | 0.933798 |

analysis\_df = intereting\_df.transpose()  
analysis\_df = analysis\_df.reset\_index(drop=False)  
analysis\_df = analysis\_df.rename(columns={analysis\_df.columns[0]: 'runs'})  
analysis\_df = analysis\_df.rename\_axis('index')  
analysis\_df

| params\_kernel | runs | linear | poly | rbf | sigmoid |
| --- | --- | --- | --- | --- | --- |
| index |  |  |  |  |  |
| 0 | user\_attrs\_acc1\_pred-run0 | 0.986111 | 0.993056 | 0.975694 | 0.940972 |
| 1 | user\_attrs\_acc1\_pred-run1 | 0.989583 | 0.986111 | 0.982639 | 0.934028 |
| 2 | user\_attrs\_acc1\_pred-run2 | 0.986063 | 0.989547 | 0.986063 | 0.940767 |
| 3 | user\_attrs\_acc1\_pred-run3 | 0.975610 | 0.982578 | 0.972125 | 0.937282 |
| 4 | user\_attrs\_acc1\_pred-run4 | 0.989547 | 1.000000 | 0.982578 | 0.933798 |

array\_data = analysis\_df.drop(analysis\_df.columns[0], axis=1).values  
array\_data

array([[0.98611111, 0.99305556, 0.97569444, 0.94097222],  
 [0.98958333, 0.98611111, 0.98263889, 0.93402778],  
 [0.98606272, 0.98954704, 0.98606272, 0.94076655],  
 [0.97560976, 0.9825784 , 0.97212544, 0.93728223],  
 [0.98954704, 1. , 0.9825784 , 0.93379791]])

analysis\_df.columns[1]

'linear'

from scikit\_posthocs import \_\_convert\_to\_block\_df

analysis\_df\_regular, \_, \_, \_ = \_\_convert\_to\_block\_df(array\_data)  
analysis\_df\_regular = analysis\_df\_regular.rename(columns={'blocks': 'runs',   
 'groups': treatment, 'y':target})  
analysis\_df\_regular[treatment] = analysis\_df\_regular[treatment].apply(lambda x: analysis\_df.columns[1+x])  
analysis\_df\_regular

|  | runs | params\_kernel | value |
| --- | --- | --- | --- |
| 0 | 0 | linear | 0.986111 |
| 1 | 1 | linear | 0.989583 |
| 2 | 2 | linear | 0.986063 |
| 3 | 3 | linear | 0.975610 |
| 4 | 4 | linear | 0.989547 |
| 5 | 0 | poly | 0.993056 |
| 6 | 1 | poly | 0.986111 |
| 7 | 2 | poly | 0.989547 |
| 8 | 3 | poly | 0.982578 |
| 9 | 4 | poly | 1.000000 |
| 10 | 0 | rbf | 0.975694 |
| 11 | 1 | rbf | 0.982639 |
| 12 | 2 | rbf | 0.986063 |
| 13 | 3 | rbf | 0.972125 |
| 14 | 4 | rbf | 0.982578 |
| 15 | 0 | sigmoid | 0.940972 |
| 16 | 1 | sigmoid | 0.934028 |
| 17 | 2 | sigmoid | 0.940767 |
| 18 | 3 | sigmoid | 0.937282 |
| 19 | 4 | sigmoid | 0.933798 |

现在我们对五折交叉验证的结果来对四个方法进行假设检验。 这一次我们该使用 Friedman 检验, 这也是一个不需要正态性和方差齐性的检验方法，某种程度上[论文](https://www.jmlr.org/papers/volume7/demsar06a/demsar06a.pdf)会选择这个而非 Kruskal-Wallis。 刚才我们不用是因为刚才每一组的样本量不一致，而 Friedman 检验要求样本量一致

import scipy.stats as ss

res = ss.friedmanchisquare(\*array\_data.T)  
if res.pvalue < 0.05:  
 print("Reject null hypothesis. Therefore, the data is significantly different.")  
res

Mon 2024-11-18 21:44:18.996180

INFO Reject null hypothesis. Therefore, the data is significantly different. d=544274;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=59060;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

FriedmanchisquareResult(statistic=13.653061224489791, pvalue=0.003417529443793913)

Nemenyi 是其中的一个 对应于 Friedman 的测试。

首先我们需要计算4个方法的平均排名（在同一个cv fold上，相互之间的排名），这个是 判断哪个方法更好的核心依据。

avg\_rank = analysis\_df\_regular.groupby('runs').value.rank(pct=True).groupby(analysis\_df\_regular.params\_kernel).mean().sort\_values(ascending=False)  
avg\_rank

params\_kernel  
poly 0.950  
linear 0.775  
rbf 0.525  
sigmoid 0.250  
Name: value, dtype: float64

按照平均排名重新排序

analysis\_df = analysis\_df[avg\_rank.index]  
analysis\_df

| params\_kernel | poly | linear | rbf | sigmoid |
| --- | --- | --- | --- | --- |
| index |  |  |  |  |
| 0 | 0.993056 | 0.986111 | 0.975694 | 0.940972 |
| 1 | 0.986111 | 0.989583 | 0.982639 | 0.934028 |
| 2 | 0.989547 | 0.986063 | 0.986063 | 0.940767 |
| 3 | 0.982578 | 0.975610 | 0.972125 | 0.937282 |
| 4 | 1.000000 | 0.989547 | 0.982578 | 0.933798 |

array\_data = analysis\_df.values  
array\_data

array([[0.99305556, 0.98611111, 0.97569444, 0.94097222],  
 [0.98611111, 0.98958333, 0.98263889, 0.93402778],  
 [0.98954704, 0.98606272, 0.98606272, 0.94076655],  
 [0.9825784 , 0.97560976, 0.97212544, 0.93728223],  
 [1. , 0.98954704, 0.9825784 , 0.93379791]])

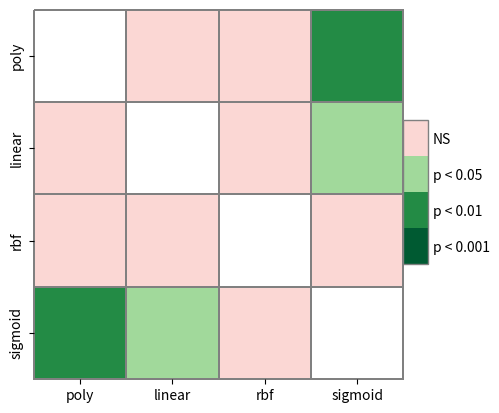
import scikit\_posthocs as sp

# res = sp.posthoc\_nemenyi\_friedman(analysis\_df\_regular,   
# block\_col="runs",   
# group\_col=treatment, y\_col=target, sort=True)  
res = sp.posthoc\_nemenyi\_friedman(array\_data)   
res

|  | 0 | 1 | 2 | 3 |
| --- | --- | --- | --- | --- |
| 0 | 1.000000 | 0.826788 | 0.158925 | 0.003389 |
| 1 | 0.826788 | 1.000000 | 0.611061 | 0.049612 |
| 2 | 0.158925 | 0.611061 | 1.000000 | 0.532733 |
| 3 | 0.003389 | 0.049612 | 0.532733 | 1.000000 |

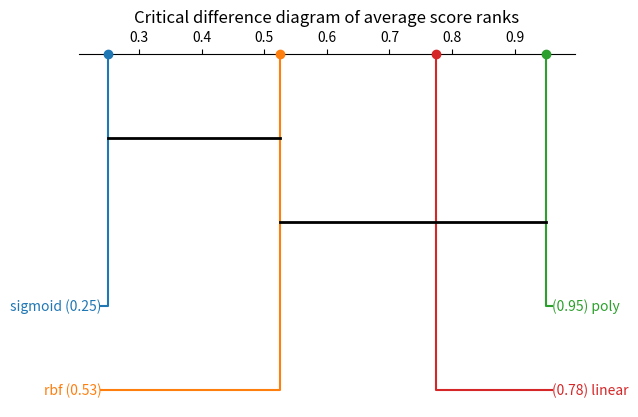
sp.sign\_plot(res, xticklabels=avg\_rank.index, yticklabels=avg\_rank.index, \*\*heatmap\_args)

(<Axes: >, <matplotlib.colorbar.Colorbar object at 0x7f3b1845e0e0>)



从检验的结果图可以看出，poly和linear方法明显优于sigmoid，而其他方法之间就没有足够的实验证据来进行比较。 这说明sigmoid方法即使调到了较优的超参数，还是显著比前两个方法要弱一些，不建议新手使用。

plt.title('Critical difference diagram of average score ranks')  
sp.critical\_difference\_diagram(avg\_rank, res)  
pass



我们还画出了 Critical difference diagram。 图中靠右边的方法平均排名高，左边的方法平均排名低。 排名的差距是否具有统计上的显著性，关键看差距是否大于图中的黑色实线的长度。

我们可以看到poly几乎就要显著优于rbf了，而rbf几乎就要显著优于sigmoid了，但是都还差一点不太显著。

##### 0.4.2.5 如何解释 poly kernel 为什么显著优于 sigmoid kernel? SVM决策边界可视化分析

如果是2维或者3维的数据，我们可以画图来解释SVM工作的原理，绘制决策边界，但是我们现在是 64维 (UCI digits) 或者 728维 (MNIST) 的输入数据。这一方面可以参考这一个工具[mlxtend](https://rasbt.github.io/mlxtend/user_guide/plotting/plot_decision_regions/)，对SVM的可视化非常全面。

对于深度学习模型来说，深度学习最后一层的特征往往被认为是表示学习的结果，用来进行降维可视化，但是我们这里 SVM 是单层的模型，没有对特征进行变换，输入的维度就已经是特征。直接对输入的数据特征进行降维可视化，完全无法展示SVM的能力，因为SVM并不是一个特征提取器。

综合[stackoverflow讨论](https://stackoverflow.com/questions/23168285/the-visualization-of-high-dimensional-input-for-two-class-classification-in-svm?rq=4) 和 [quora讨论](https://www.quora.com/What-is-the-best-way-to-visualize-an-n-dimensional-svm)，我认为有两个主要方法。

* 对降维后的数据重新使用不同kernel的SVM来学习，进一步绘制决策边界。使用刚才每一个kernel下调参最优的SVM参数来进行训练。
* SVM的概率输出，概率本身就是一个重要的信息，体现了不同SVM的决策有何不同。类似与Fisher提出的LDA分类方法自己也是一种降维方法，带概率输出的SVM把输入维度映射到类别维度上的概率值，本身是一种降维手法，而且体现了不同参数SVM的学习有何不同。

如果不是为了看SVM的能力，而是为了解释kernel method的能力，我们还可以使用 [KernelPCA](https://scikit-learn.org/dev/modules/generated/sklearn.decomposition.KernelPCA.html) 的可视化进行分析。

首先我们调取最优的训练参数

from sklearn.svm import SVC

# 提取以 'params\_' 开头的列  
param\_cols = [col for col in max\_rows.columns if col.startswith('params\_')]  
  
# 去掉 'params\_' 前缀并重命名列  
max\_rows\_params = max\_rows[param\_cols].rename(columns=lambda x: x[len('params\_'):])  
  
# 将每一行转换为字典的列表  
dict\_list = max\_rows\_params.to\_dict('records')  
dict\_list[0]

{  
 'C': 60.42963902381333,  
 'break\_ties': True,  
 'class\_weight': 'balanced',  
 'coef0': 0.34375,  
 'decision\_function\_shape': 'ovr',  
 'degree': 6.0,  
 'gamma': 'scale',  
 'kernel': 'linear',  
 'max\_iter': 656.0,  
 'shrinking': True,  
 'tol': 0.008058421877614822  
}

optuna优化时，把数据类型有些变成了float，需要更正。

from dataclasses import fields  
from typing import get\_args, Union

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/svm/kernel_hpo.py#L235)

### 0.5 dict\_to\_dataclass

dict\_to\_dataclass (dataclass\_type, data\_dict)

def dict\_to\_dataclass(dataclass\_type, data\_dict):  
 converted\_dict = {}  
 for field in fields(dataclass\_type):  
 field\_name = field.name  
 field\_type = field.type  
 if field\_name in data\_dict:  
 value = data\_dict[field\_name]  
 # 处理 Union 类型  
 if getattr(field\_type, '\_\_origin\_\_', None) is Union:  
 # 获取 Union 中的所有类型  
 types = get\_args(field\_type)  
 # 选择第一个非 NoneType 的类型  
 non\_none\_types = [t for t in types if t is not type(None)]  
 target\_type = non\_none\_types[0] if non\_none\_types else None  
 # print(target\_type)  
 if target\_type not in [dict, list, tuple]: # TODO 递归处理  
 converted\_dict[field\_name] = target\_type(value)  
 else:  
 converted\_dict[field\_name] = value  
 else:  
 converted\_dict[field\_name] = field\_type(value)  
 return dataclass\_type(\*\*converted\_dict)

# 将 dict\_list 中的参数转换为 SupportVectorClassifierConfig 的实例列表  
configs = [dict\_to\_dataclass(SupportVectorClassifierConfig, params) for params in dict\_list]  
# 根据 dict\_list 初始化 SVC 列表  
svc\_list = [SVC(\*\*asdict(config)) for config in configs]  
svc\_list[0]

SVC(C=60.42963902381333, break\_ties=True, class\_weight='balanced',  
 coef0=0.34375, degree=6, kernel='linear', max\_iter=656,  
 tol=0.008058421877614822)

然后我们先使用第一种分析方法，做降维。

from sklearn.decomposition import PCA

# 定义 PCA 模型，设定降维后的维度，例如 n\_components=10  
pca = PCA(n\_components=2)  
  
# 对 X\_train 进行降维  
X\_train\_reduced = pca.fit\_transform(X\_train)  
  
# 对 X\_test 进行降维  
X\_test\_reduced = pca.transform(X\_test)  
  
# 查看 pca 的一些指标  
pca.explained\_variance\_ratio\_

array([0.12330114, 0.10126638], dtype=float32)

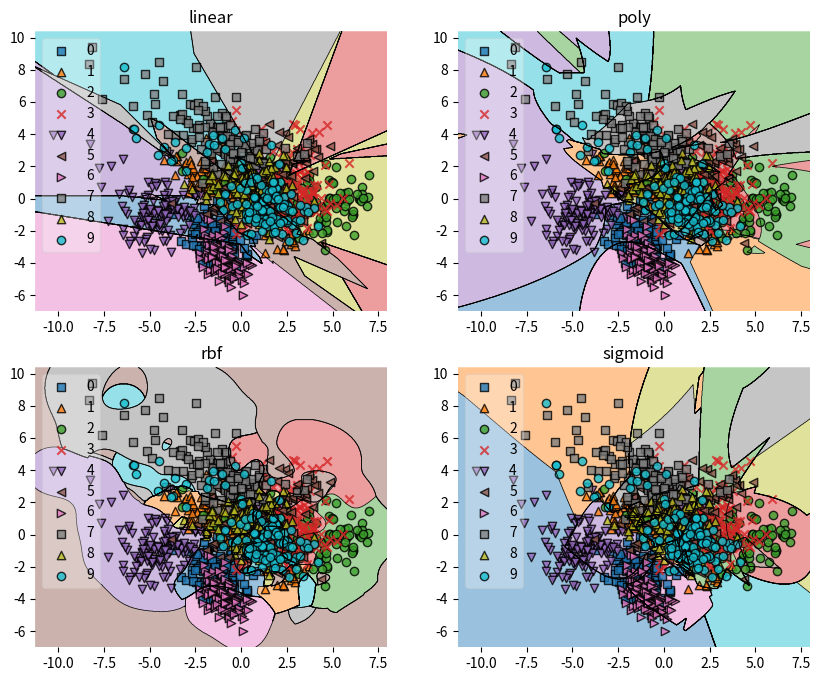
可视化

labels = [svc.kernel for svc in svc\_list]  
labels[0]

'linear'

import matplotlib.pyplot as plt  
from mlxtend.plotting import plot\_decision\_regions  
import matplotlib.gridspec as gridspec  
import itertools

gs = gridspec.GridSpec(2, 2)  
  
fig = plt.figure(figsize=(10,8))  
  
for clf, lab, grd in zip(svc\_list,  
 labels,  
 itertools.product([0, 1], repeat=2)):  
  
 clf.fit(X\_train\_reduced, y\_train)  
 ax = plt.subplot(gs[grd[0], grd[1]])  
 fig = plot\_decision\_regions(X=X\_train\_reduced, y=y\_train, clf=clf, legend=2)  
 plt.title(lab)  
  
plt.show()



##### 0.5.0.1 如何解释 poly kernel 为什么显著优于 sigmoid kernel? Kernel SVM概率输出的画图分析

首先在没有降维的数据集上按照最优参数来训练SVC。注意为了速度，刚才调参的时候没有probality=True，现在为了分析需要有。

svc\_list = [SVC(\*\*(asdict(config)| dict(probability=True)) ) for config in configs]  
svc\_list = [svc.fit(X\_train, y\_train) for svc in svc\_list]

得到 y\_pred\_prob

y\_pred\_train\_probs = [svc.predict\_proba(X\_train) for svc in svc\_list]  
y\_pred\_test\_probs = [svc.predict\_proba(X\_test) for svc in svc\_list]

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/svm/kernel_hpo.py#L262)

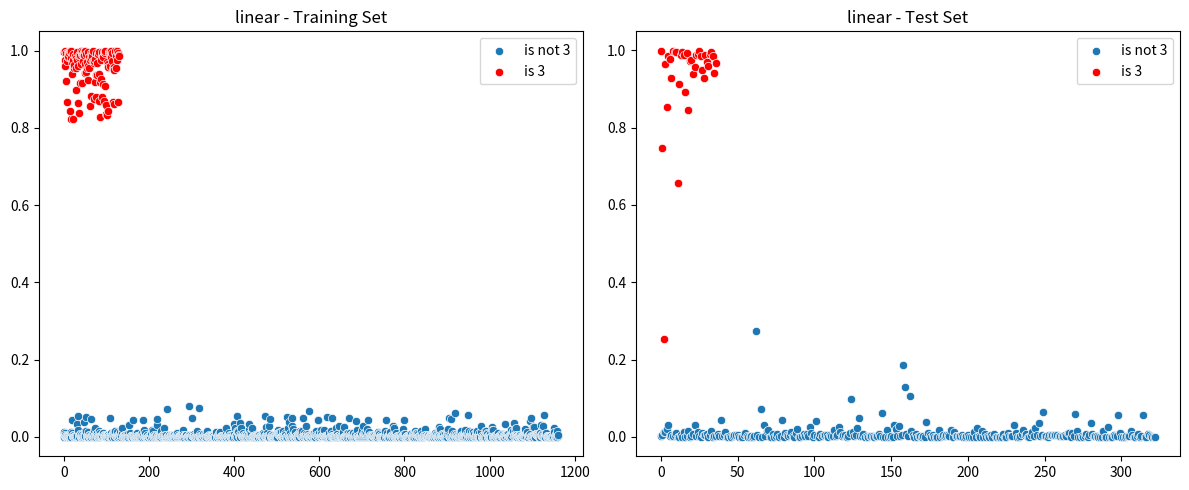
### 0.6 draw\_probs

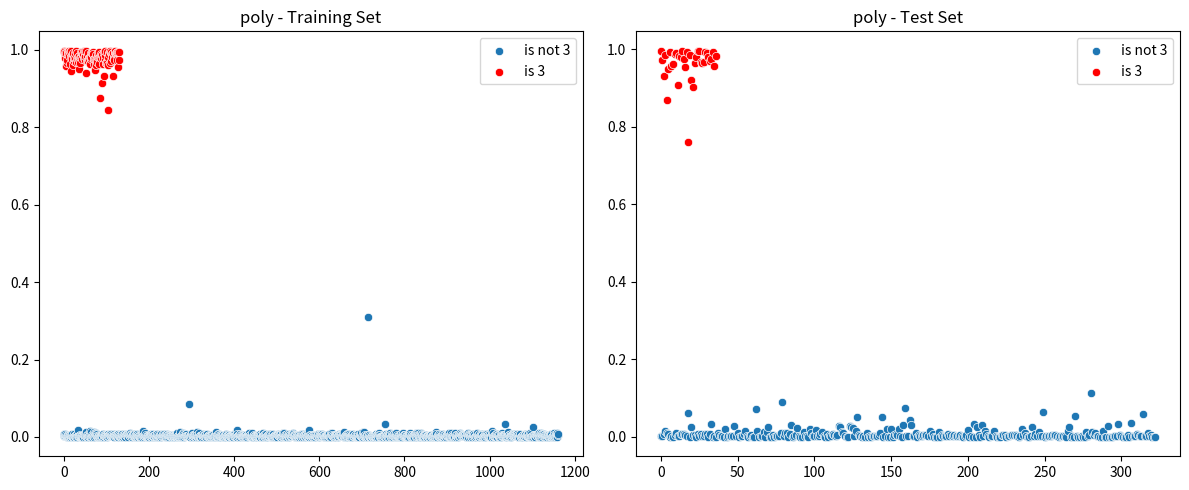
draw\_probs (y\_pred\_prob, y\_test, interested\_class:int)

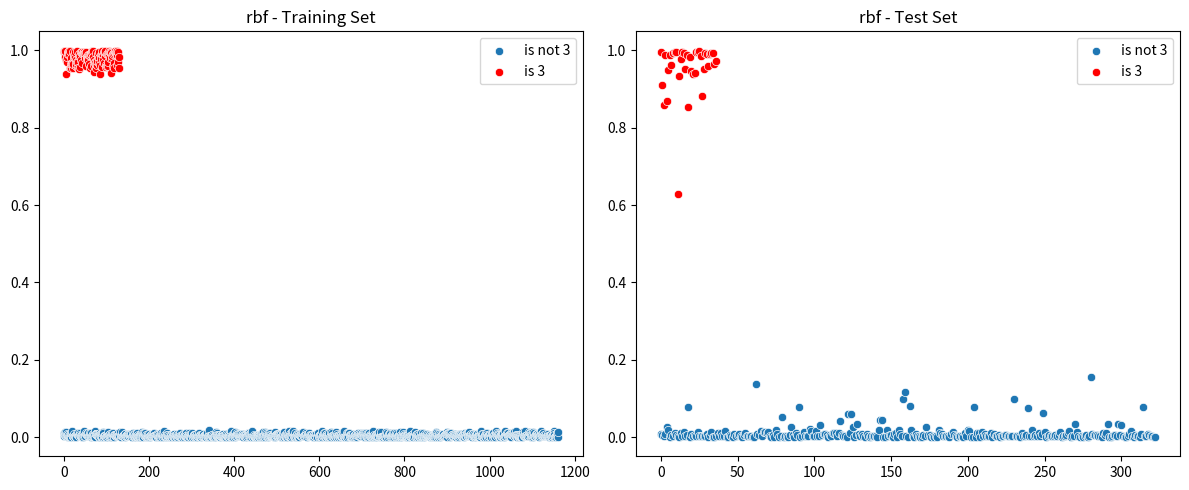
import seaborn as sns

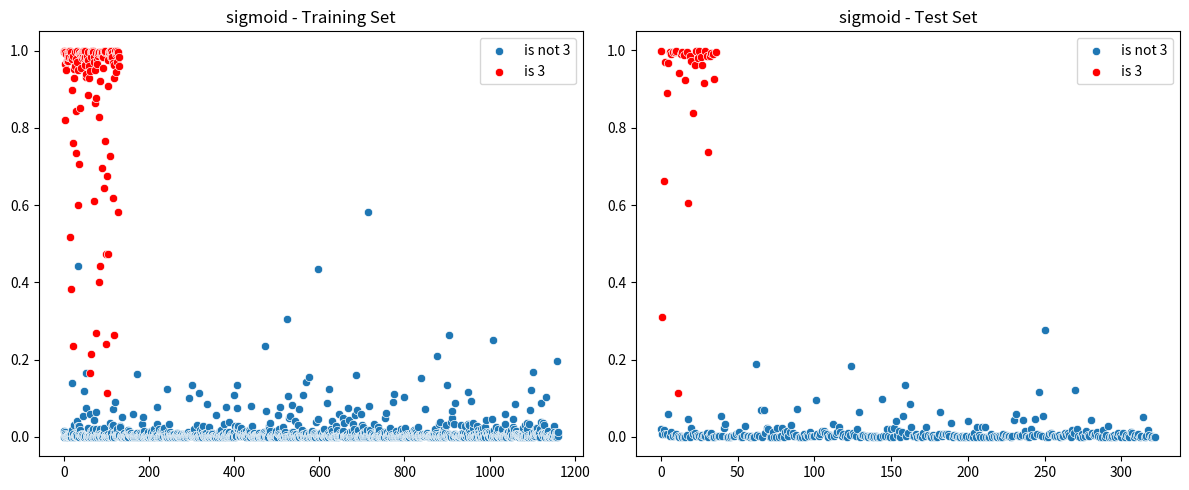
def draw\_probs(y\_pred\_prob, y\_test, interested\_class:int):  
 sns.scatterplot(y\_pred\_prob[y\_test!=interested\_class][:, interested\_class], label=f"is not {interested\_class}")  
 sns.scatterplot(y\_pred\_prob[y\_test==interested\_class][:, interested\_class], label=f"is {interested\_class}", color="red")

# 对所有的 y\_pred\_train\_probs 和 y\_pred\_test\_probs，调用 draw\_probs 函数并绘制图形  
# intersted\_class = 0  
intersted\_class = 3  
for i, model\_name in enumerate(labels):  
 train\_probs = y\_pred\_train\_probs[i]  
 test\_probs = y\_pred\_test\_probs[i]  
   
 # 创建左右并排的图形  
 plt.figure(figsize=(12, 5))  
   
 # 绘制训练集的概率分布图  
 plt.subplot(1, 2, 1)  
 draw\_probs(train\_probs, y\_train, interested\_class=intersted\_class)  
 plt.title(f'{model\_name} - Training Set')  
   
 # 绘制测试集的概率分布图  
 plt.subplot(1, 2, 2)  
 draw\_probs(test\_probs, y\_test, interested\_class=intersted\_class)  
 plt.title(f'{model\_name} - Test Set')  
   
 plt.tight\_layout()  
 plt.show()







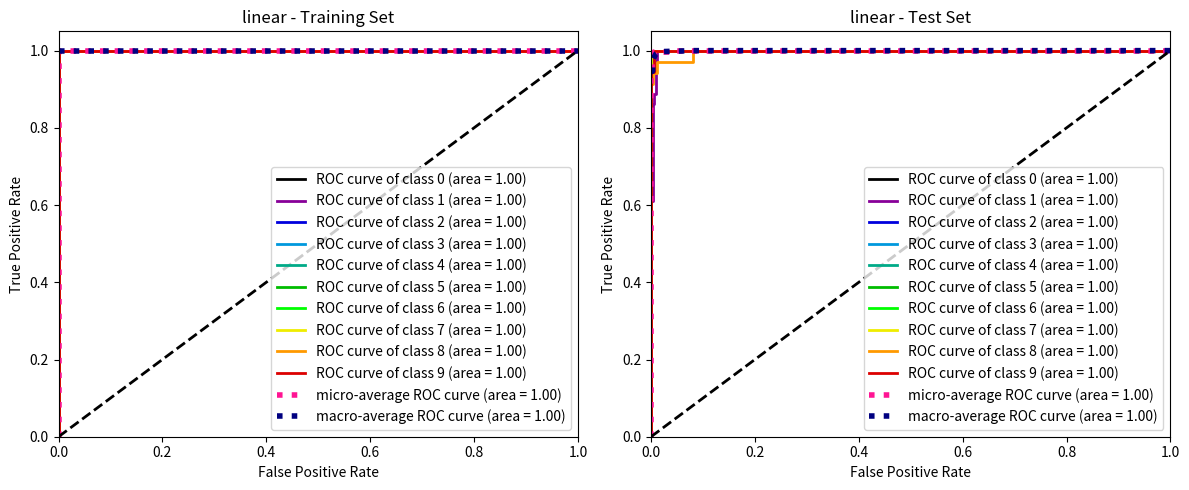


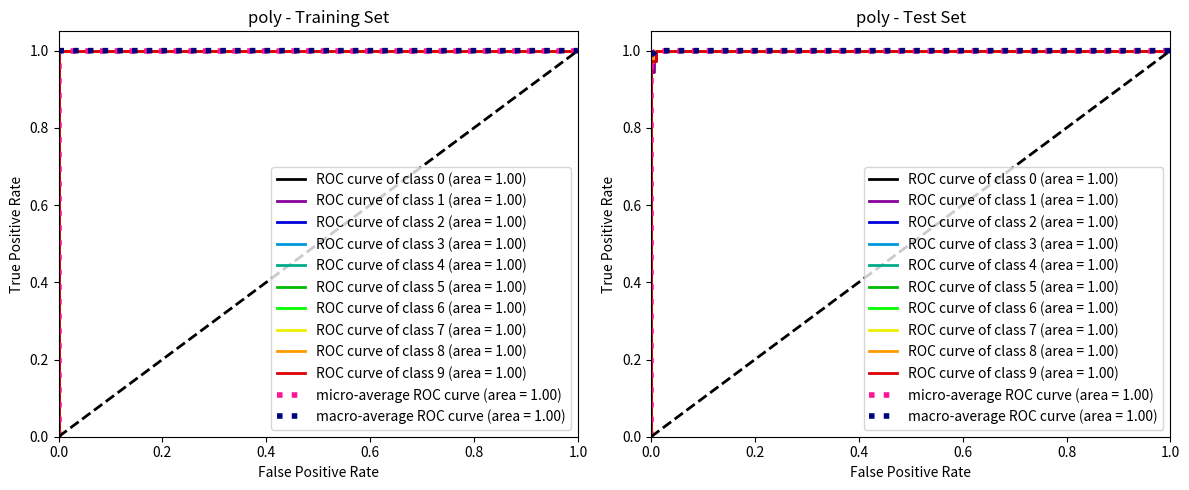
我们在图上可以比较明显地看出，sigmoid相比其他方法，probabilities 会往上和往下飘，是否为这个类别的分隔的不是很开，所以最后的效果不如poly。而其他方法可能测试集会稍微没有训练集分隔得开。

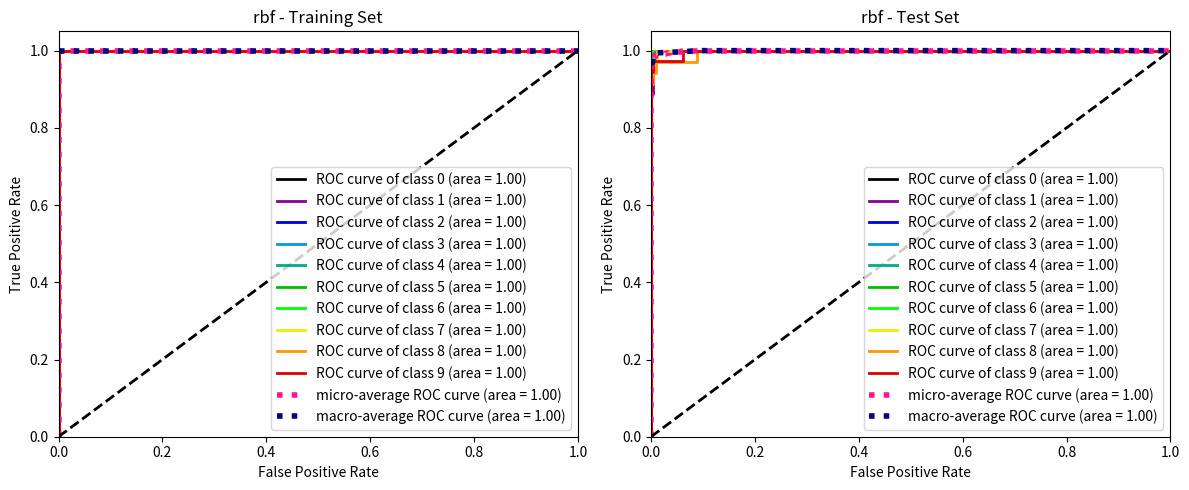
除了直接画出来概率的散点图，还有一些更加成熟的机器学习绘图分析，可以更加直观地看到模型的差距在哪。

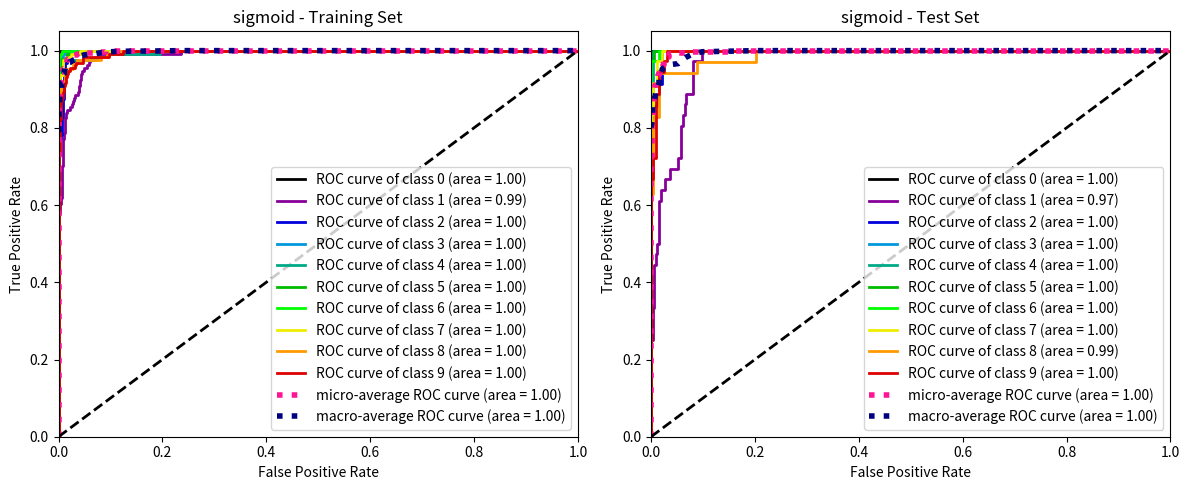
import scikitplot as skplt  
import scikitplot as skplt  
import matplotlib.pyplot as plt

# 假设 labels 是模型名称的列表，与预测概率列表对应  
# labels = ['linear', 'poly', 'rbf', 'sigmoid']  
  
for i, model\_name in enumerate(labels):  
 train\_probs = y\_pred\_train\_probs[i]  
 test\_probs = y\_pred\_test\_probs[i]  
   
 # 创建左右并排的图形  
 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))  
   
 # 绘制训练集的 ROC 曲线  
 skplt.metrics.plot\_roc(y\_train, train\_probs, ax=ax1)  
 ax1.set\_title(f'{model\_name} - Training Set')  
   
 # 绘制测试集的 ROC 曲线  
 skplt.metrics.plot\_roc(y\_test, test\_probs, ax=ax2)  
 ax2.set\_title(f'{model\_name} - Test Set')  
   
 plt.tight\_layout()  
 plt.show()







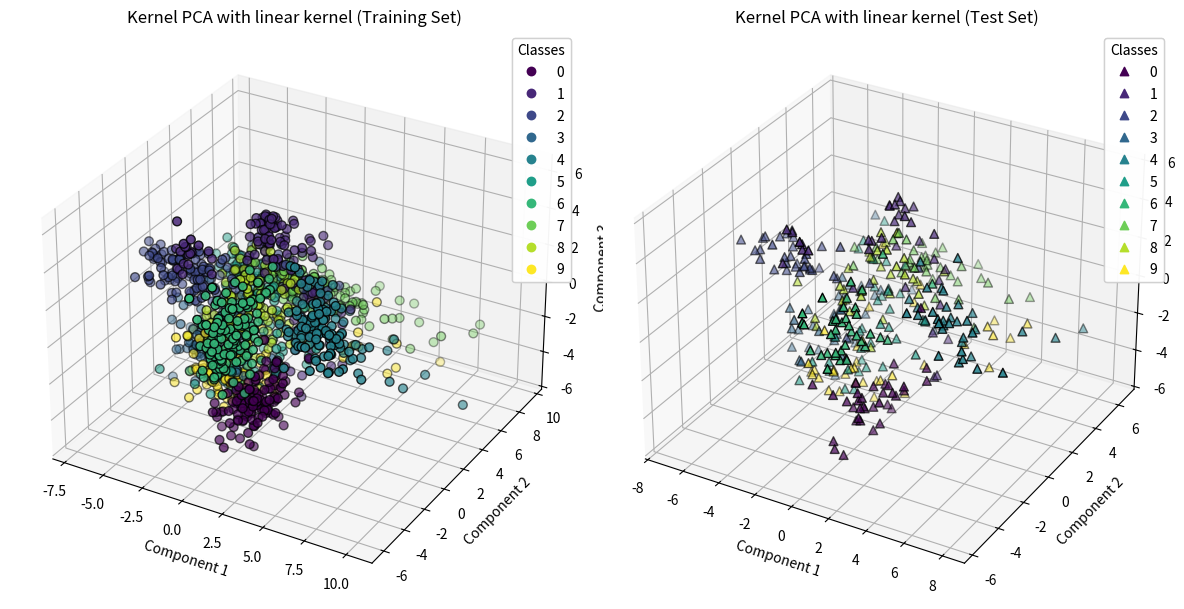


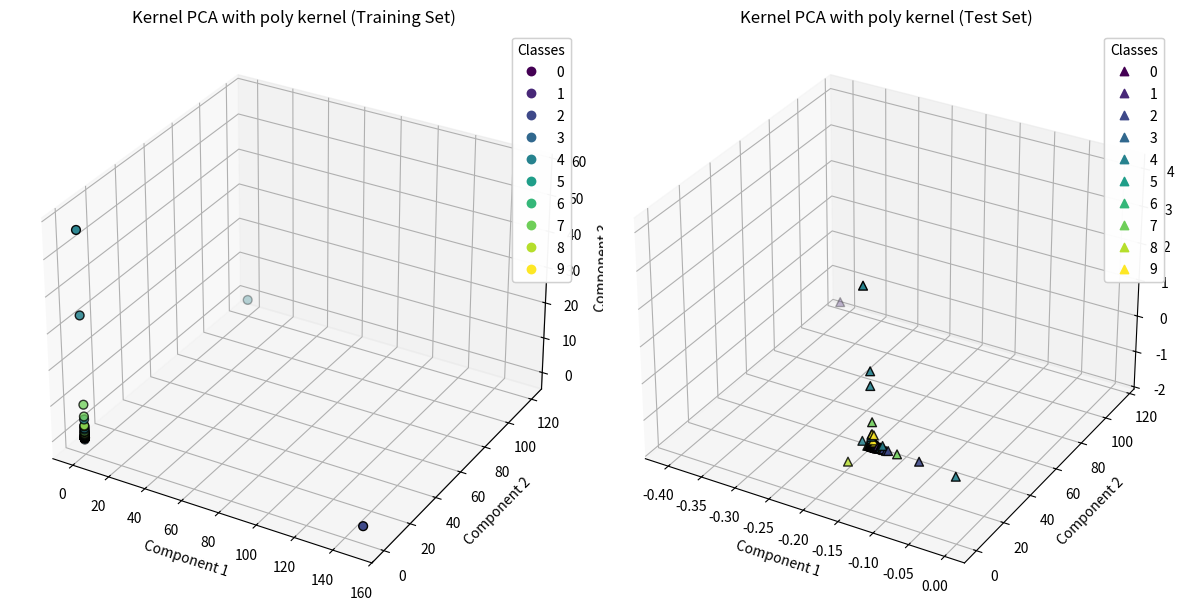
图中可以明显看到 sigmoid kernel方法在 类别 1和8上面表现不佳。而poly的曲线接近完美。

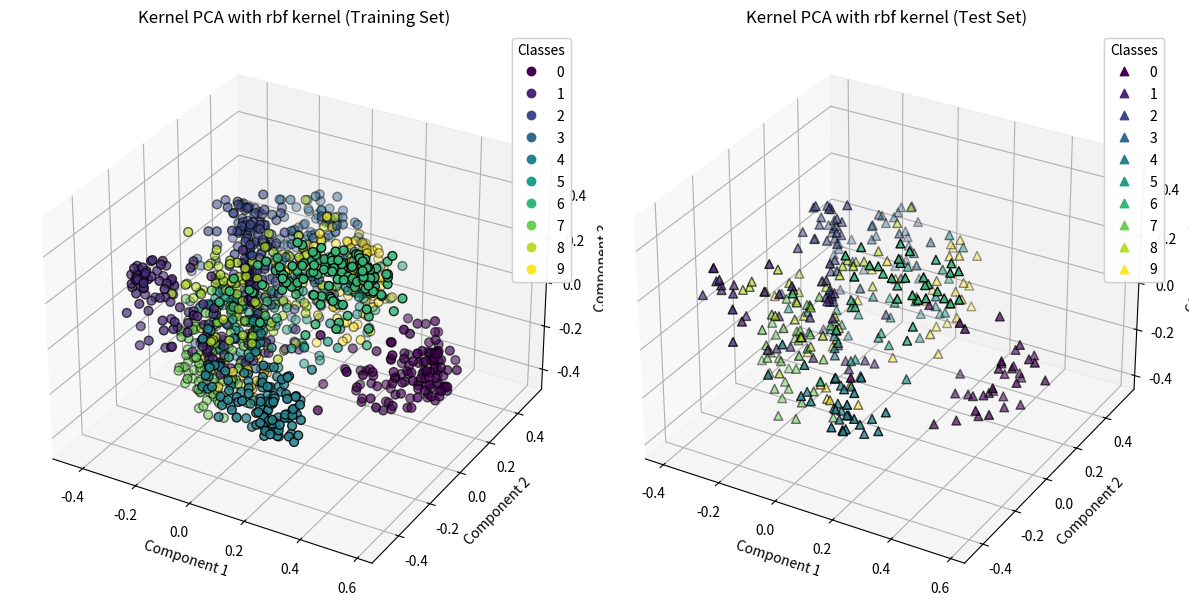
##### 0.6.0.1 如何解释 poly kernel 为什么显著优于 sigmoid kernel? Kernel PCA 可视化

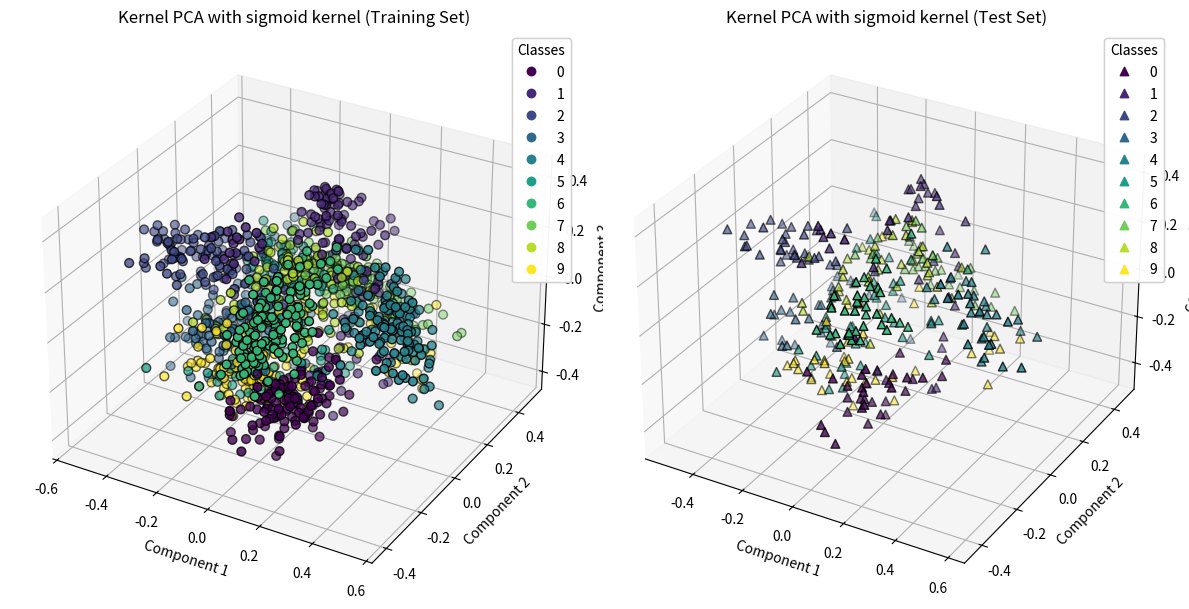
from sklearn.decomposition import KernelPCA  
import matplotlib.pyplot as plt  
from mpl\_toolkits.mplot3d import Axes3D

# labels = ['linear', 'poly', 'rbf', 'sigmoid']  
  
for kernel in labels:  
 # 使用 Kernel PCA 将数据降维到 3 维  
 kpca = KernelPCA(n\_components=3, kernel=kernel)  
 X\_train\_reduced = kpca.fit\_transform(X\_train)  
 X\_test\_reduced = kpca.transform(X\_test)  
   
 # 创建一个包含两个子图的图形  
 fig = plt.figure(figsize=(12, 6))  
   
 # 绘制训练集的 3D 散点图  
 ax1 = fig.add\_subplot(1, 2, 1, projection='3d')  
 scatter1 = ax1.scatter(  
 X\_train\_reduced[:, 0], X\_train\_reduced[:, 1], X\_train\_reduced[:, 2],  
 c=y\_train, cmap='viridis', marker='o', edgecolor='k', s=40  
 )  
 ax1.set\_title(f'Kernel PCA with {kernel} kernel (Training Set)')  
 ax1.set\_xlabel('Component 1')  
 ax1.set\_ylabel('Component 2')  
 ax1.set\_zlabel('Component 3')  
 legend1 = ax1.legend(\*scatter1.legend\_elements(), title="Classes")  
 ax1.add\_artist(legend1)  
   
 # 绘制测试集的 3D 散点图  
 ax2 = fig.add\_subplot(1, 2, 2, projection='3d')  
 scatter2 = ax2.scatter(  
 X\_test\_reduced[:, 0], X\_test\_reduced[:, 1], X\_test\_reduced[:, 2],  
 c=y\_test, cmap='viridis', marker='^', edgecolor='k', s=40  
 )  
 ax2.set\_title(f'Kernel PCA with {kernel} kernel (Test Set)')  
 ax2.set\_xlabel('Component 1')  
 ax2.set\_ylabel('Component 2')  
 ax2.set\_zlabel('Component 3')  
 legend2 = ax2.legend(\*scatter2.legend\_elements(), title="Classes")  
 ax2.add\_artist(legend2)  
   
 plt.tight\_layout()  
 plt.show()









### 0.7 附加题: 构建使用 kernel 方法的 SVM 分类器 （手动实现SMO）

在 [02svm\_handy\_crafted\_linear](./02svm_handy_crafted_linear.html) 中，我们已经手动实现了 基于梯度下降方法的 Linear SVM 分类器。在这一节，我们将手动实现 基于 SMO 方法的 SVM 分类器，以便于我们可以更好地理解 SVM 的原理。

接下来的内容请见文件 [04svm\_handy\_crafted\_kernel](./04svm_handy_crafted_kernel.html)

本次Project的目录请见绪论 [00svm](./00svm.html)。