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——01调库实现SVM手写数字识别

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01篇——调库实现SVM手写数字识别。

上文我们已经了解了数据的加载与预处理，并且理解了理论上的一些问题。

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  
(70000, 784) float32 (70000,) int64 [0 1 2 3 4 5 6 7 8 9]  
50400 5600 14000

## 1 实验内容

### 1.1 调库实现SVM

为了给我们后面的实验一个参照，我们调用现有代码库的SVM，关注其精度与速度的情况。 当然如果我们Project在此收尾，只能酌情被扣除分数。 在本节之后，我们将使用 PyTorch 和 numpy 这样的基础科学计算库，来在GPU和CPU上实现SVM及其优化。

!!! important

本次Project首先展示了几个常用的SVM库的精度与速度，并且对其进行调参；随后本次Project基于基础科学计算库手写实现了SVM及其优化，和前面的库的精度与速度进行了对比。

#### 1.1.1 sklearn实现的 SMO 形式的 RBF kernel SVM

from sklearn.svm import SVC

svc = SVC(kernel='rbf', probability=True)  
svc.fit(X\_train, y\_train)

SVC(probability=True)

y\_pred = svc.predict(X\_test)  
y\_pred\_proba = svc.predict\_proba(X\_test)  
svc\_res = compute\_classification\_metrics(y\_test, y\_pred\_proba, logits\_to\_prob=False, y\_pred=y\_pred, labels=list(range(10)))  
svc\_res

{  
 'acc1': 0.9722222222222222,  
 'acc2': 0.9833333333333333,  
 'acc3': 0.9916666666666667,  
 'acc5': 0.9916666666666667,  
 'acc10': 1.0,  
 'acc20': 1.0,  
 'matthews\_corrcoef': 0.9692503039897137,  
 'f1': 0.972029880729665,  
 'precision': 0.973182392919235,  
 'recall': 0.971984126984127,  
 'balanced\_accuracy': 0.971984126984127,  
 'cohen\_kappa': 0.9691339500827382,  
 'roc\_auc': 0.9993403631348077,  
 'hinge\_loss': 0.11966705184959518,  
 'log\_loss': 0.1287164515187897,  
 'acc1\_pred': 0.9722222222222222  
}

需要注意的是，SVM的概率计算的机制较为特殊，所以用SVM的概率预测值进行决策的话，与SVM本身的预测是有所不同的。 因此我们上面看到的结果中，根据y\_pred\_proba计算的acc1(top k 准确率) 与 根据 y\_pred计算的acc1并不相同，SVM本身的预测比SVM的概率预测更加准确。

我写的代码库 namable\_classify 就充分考虑了这种情况的发生，允许用户传入 y\_pred。

为了更好理解，我们代码可以看到具体的区别在哪里：

import numpy as np

interested = np.where(np.argmax(y\_pred\_proba, axis=1)!=y\_pred)  
y\_pred\_proba[interested], np.argmax(y\_pred\_proba, axis=1)[interested], y\_pred[interested]

(  
 array([[0.04655995, 0.06806033, 0.01481866, 0.01379496, 0.04681734,  
 0.08531872, 0.31274431, 0.02019028, 0.37692727, 0.01476818],  
 [0.00642917, 0.04689452, 0.01313312, 0.04944728, 0.01523242,  
 0.03770956, 0.00456987, 0.3768156 , 0.01336737, 0.43640109],  
 [0.13989772, 0.01654543, 0.16119182, 0.17785276, 0.05271123,  
 0.13162163, 0.13944642, 0.02354677, 0.03741341, 0.1197728 ]]),  
 array([8, 9, 3]),  
 array([6, 7, 2])  
)

#### 1.1.2 sklearn 实现的 SMO形式 的 linear SVM

svc\_linear = SVC(kernel='linear', probability=True)  
svc\_linear.fit(X\_train, y\_train)

SVC(kernel='linear', probability=True)

y\_pred = svc\_linear.predict(X\_test)  
y\_pred\_proba = svc\_linear.predict\_proba(X\_test)  
compute\_classification\_metrics(y\_test, y\_pred\_proba, logits\_to\_prob=False, y\_pred=y\_pred, labels=list(range(10)))

{  
 'acc1': 0.9777777777777777,  
 'acc2': 1.0,  
 'acc3': 1.0,  
 'acc5': 1.0,  
 'acc10': 1.0,  
 'acc20': 1.0,  
 'matthews\_corrcoef': 0.9723041620958817,  
 'f1': 0.9748756231876998,  
 'precision': 0.9756801209103841,  
 'recall': 0.9748326898326898,  
 'balanced\_accuracy': 0.9748326898326898,  
 'cohen\_kappa': 0.9722207932506817,  
 'roc\_auc': 0.9995142149923252,  
 'hinge\_loss': 0.13536783910745212,  
 'log\_loss': 0.1218441979874856,  
 'acc1\_pred': 0.975  
}

#### 1.1.3 sklearn 实现的 基于 liblinear 库 的 Linear SVM

The main differences between LinearSVC and SVC lie in the loss function used by default, and in the handling of intercept regularization between those two implementations.

n\_samples, n\_features = X\_train.shape  
sklearn\_will\_use\_dual\_problem = n\_samples < n\_features  
n\_samples, n\_features, sklearn\_will\_use\_dual\_problem

(1293, 64, False)

from sklearn.svm import LinearSVC

linear\_svc = LinearSVC(penalty='l2', loss='squared\_hinge', dual=sklearn\_will\_use\_dual\_problem,   
 tol=0.0001, C=1.0, multi\_class='ovr', fit\_intercept=True, intercept\_scaling=1)  
linear\_svc.fit(X\_train, y\_train)

LinearSVC(dual=False)

y\_pred = linear\_svc.predict(X\_test)  
compute\_classification\_metrics(y\_test, y\_pred=y\_pred, labels=list(range(10)), y\_pred\_metrics\_only=True)

{  
 'matthews\_corrcoef': 0.9507782171913305,  
 'f1': 0.9551493507947526,  
 'precision': 0.9564648995608748,  
 'recall': 0.9553131703131704,  
 'balanced\_accuracy': 0.9553131703131704,  
 'cohen\_kappa': 0.9506151669738929,  
 'acc1\_pred': 0.9555555555555556  
}

#### 1.1.4 sklearn 实现的 SGD优化Hinge Loss形式 的 linear SVM

https://scikit-learn.org/1.5/modules/generated/sklearn.linear\_model.SGDClassifier.html

sklearn实现的 soft-margin linear Support Vector Machine 在 from sklearn.linear\_model import SGDClassifier类中给出。 sklearn使用的是 one versus all (OVA) scheme，文档摘录如下

For each of the classes, a binary classifier is learned that discriminates between that and all other classes. At testing time, we compute the confidence score (i.e. the signed distances to the hyperplane) for each classifier and choose the class with the highest confidence.

sklearn文档指出这种形式的SVM的潜在优化，包括以下几点

* modified\_huber，这是一个平滑版本的hinge loss，有研究表明效果更好。
* ASGD，这个似乎类似于深度学习中的 [Stochastic Weight Averaging (SWA)](https://pytorch.org/blog/stochastic-weight-averaging-in-pytorch/)、[Model Exponential moving average (Model EMA)](https://timm.fast.ai/training_modelEMA)和[Sharpness-Aware Minimization (SAM)](https://github.com/davda54/sam)，我们有空深入研究一下这里面的区别与联系。
* Elastic Net, 结合了L1和L2的正则项，来控制模型复杂度。
* optimal learning rate, 具体来说是 Leon Bottou 提出的一种方法，公式在[此处](https://stackoverflow.com/questions/52577305/the-formula-for-the-optimal-learning-rate-in-the-sgdclassifier-in-scikit-learn)可见。

from sklearn.linear\_model import SGDClassifier

sgd\_svm = SGDClassifier(  
 # loss="hinge",   
# penalty="l2",   
 loss="modified\_huber",   
 penalty="elasticnet",   
 max\_iter=1000, fit\_intercept=True,   
 learning\_rate='optimal',   
 early\_stopping=True, validation\_fraction=0.1, n\_iter\_no\_change=5, warm\_start=True, average=True)  
sgd\_svm.fit(X\_train, y\_train)

SGDClassifier(average=True, early\_stopping=True, loss='modified\_huber',  
 penalty='elasticnet', warm\_start=True)

y\_pred = sgd\_svm.predict(X\_test)  
compute\_classification\_metrics(y\_test, y\_pred=y\_pred, labels=list(range(10)), y\_pred\_metrics\_only=True)

{  
 'matthews\_corrcoef': 0.9538068512149889,  
 'f1': 0.9581518233244193,  
 'precision': 0.9591395565079776,  
 'recall': 0.958086658086658,  
 'balanced\_accuracy': 0.958086658086658,  
 'cohen\_kappa': 0.9537005281569381,  
 'acc1\_pred': 0.9583333333333334  
}

#### 1.1.5 使用更加高性能的SVM库来加速训练

##### 1.1.5.1 CPU上对SVM训练的加速

我们首先尝试CPU上的加速。针对英特尔CPU，Intel对sklearn库进行了加速，我们通过下面的方式启用

pip install scikit-learn-intelex

参考 [博客](https://stackoverflow.com/questions/17310752/can-you-capture-the-output-of-ipythons-magic-methods-timeit) 进行测速。

# 加速之前  
original\_sklearn = [0]\*4  
original\_sklearn[0] = %timeit -o svc.fit(X\_train, y\_train)  
original\_sklearn[1] = %timeit -o svc\_linear.fit(X\_train, y\_train)  
original\_sklearn[2] = %timeit -o linear\_svc.fit(X\_train, y\_train)  
original\_sklearn[3] = %timeit -o sgd\_svm.fit(X\_train, y\_train)

235 ms ± 39.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)  
94.6 ms ± 6.33 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)  
232 ms ± 16.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)  
79.5 ms ± 7.33 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

from sklearnex import patch\_sklearn  
import logging

patch\_sklearn(verbose=False)  
logging.disable(logging.INFO)  
# 因为存在JIT影响，先跑一遍  
svc.fit(X\_train, y\_train)  
svc\_linear.fit(X\_train, y\_train)  
linear\_svc.fit(X\_train, y\_train)  
sgd\_svm.fit(X\_train, y\_train)  
pass

# 加速之后  
intel\_sklearn = [0]\*4  
intel\_sklearn[0] = %timeit -o svc.fit(X\_train, y\_train)  
intel\_sklearn[1] = %timeit -o svc\_linear.fit(X\_train, y\_train)  
intel\_sklearn[2] = %timeit -o linear\_svc.fit(X\_train, y\_train)  
intel\_sklearn[3] = %timeit -o sgd\_svm.fit(X\_train, y\_train)

284 ms ± 38.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)  
99.1 ms ± 7.91 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)  
274 ms ± 34.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)  
58 ms ± 4.13 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

[英特尔宣称](https://raw.githubusercontent.com/intel/scikit-learn-intelex/master/doc/sources/_static/scikit-learn-acceleration.PNG) 对SVC的加速可以达到203.7%，然而我们并没有看到这样的结果。 我们使用假设检验来看看英特尔的sklearn是不是显著地块？这里由于结果是相关的，而且不能假设是正态分布，所以使用 wilcoxon signed rank 检验。

from scipy.stats import wilcoxon

original\_sklearn\_time = [t.best for t in original\_sklearn]  
intel\_sklearn\_time = [t.best for t in intel\_sklearn]  
res\_less = wilcoxon(intel\_sklearn\_time, original\_sklearn\_time, zero\_method='zsplit',alternative='less' # 实验备则假设， intel\_sklearn 用时更短  
 )  
if res\_less.pvalue > 0.05:  
 print("Null hypothesis cannot be rejected, so I have to accept that intel\_sklearn is not faster than original\_sklearn. ")  
res\_less  
res\_greater = wilcoxon(intel\_sklearn\_time, original\_sklearn\_time, zero\_method='zsplit', alternative='greater' # 实验备则假设，intel\_sklearn 用时更长  
 )  
res\_less, res\_greater

Sun 2024-11-17 01:36:45.594639

INFO Null hypothesis cannot be rejected, so I have to accept that intel\_sklearn is not faster d=111460;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=111460;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\  
 than original\_sklearn.

(WilcoxonResult(statistic=7.0, pvalue=0.8125), WilcoxonResult(statistic=7.0, pvalue=0.3125))

可以看出，不能显著地说哪个更快，都是在随机误差范围之内。可以认为这两个方法一样快。

##### 1.1.5.2 GPU上对SVM的加速

pip install thundersvm

from thundersvm import SVC

# thunder\_svc = SVC(kernel='rbf', probability=True)  
thunder\_svc = SVC(gpu\_id=2)  
thunder\_svc.fit(X\_train, y\_train)

:   
[1;31m在当前单元格或上一个单元格中执行代码时 Kernel 崩溃。  
  
[1;31m请查看单元格中的代码，以确定故障的可能原因。  
  
[1;31m单击<a href='https://aka.ms/vscodeJupyterKernelCrash'>此处</a>了解详细信息。  
  
[1;31m有关更多详细信息，请查看 Jupyter <a href='command:jupyter.viewOutput'>log</a>。

import pandas as pd

y\_pred = thunder\_svc.predict(X\_test)  
y\_pred\_proba = thunder\_svc.predict\_proba(X\_test)  
thunder\_svc\_res = compute\_classification\_metrics(y\_test, y\_pred\_proba, logits\_to\_prob=False, y\_pred=y\_pred, labels=list(range(10)))  
# thunder\_svc\_res, svc\_res  
res = []  
for k, v in thunder\_svc\_res.items():  
 delta = v - svc\_res[k]  
 # print(f"{k}: {v:.4f} ({delta:+.4f})")  
 res.append(dict(metric=k, value=v, delta=delta))  
pd.DataFrame(res)

|  | metric | value | delta |
| --- | --- | --- | --- |
| 0 | acc1 | 0.972222 | 0.000000 |
| 1 | acc2 | 0.988889 | 0.005556 |
| 2 | acc3 | 0.991667 | 0.002778 |
| 3 | acc5 | 0.994444 | 0.002778 |
| 4 | acc10 | 1.000000 | 0.000000 |
| 5 | acc20 | 1.000000 | 0.000000 |
| 6 | matthews\_corrcoef | 0.969250 | 0.000000 |
| 7 | f1 | 0.972030 | 0.000000 |
| 8 | precision | 0.973182 | 0.000000 |
| 9 | recall | 0.971984 | 0.000000 |
| 10 | balanced\_accuracy | 0.971984 | 0.000000 |
| 11 | cohen\_kappa | 0.969134 | 0.000000 |
| 12 | roc\_auc | 0.992732 | -0.006617 |
| 13 | hinge\_loss | 0.124893 | 0.005310 |
| 14 | log\_loss | 0.370853 | 0.242670 |
| 15 | acc1\_pred | 0.972222 | 0.000000 |

看来精度上和sklearn的SVC基本一致，基本没有差别，除了log\_loss多一点。

thunder\_svc\_time = %timeit -o thunder\_svc.fit(X\_train, y\_train)

164 ms ± 59.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

svc\_time = %timeit -o svc.fit(X\_train, y\_train)

186 ms ± 4.74 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

这里时间是不相关的，所以使用 mannwhitneyu 检验。

from scipy.stats import mannwhitneyu

res\_less = mannwhitneyu(thunder\_svc\_time.all\_runs, svc\_time.all\_runs, alternative='less' # 实验备则假设， intel\_sklearn mannwhitneyu  
 )  
if res\_less.pvalue > 0.05:  
 print("We cannot reject the null hypothesis! ")  
else:  
 print("Null hypothesis rejected! ThunderSVC is significantly faster than SVC.")  
res\_less

Fri 2024-11-15 23:22:46.455835

INFO Null hypothesis rejected! ThunderSVC is significantly faster than SVC. d=545742;file:///home/ye\_canming/repos/novelties/cv/cls/NamableClassify/namable\_classify/utils.py\utils.py;;\:d=458913;file:///home/ye\_canming/repos/novelties/cv/cls/NamableClassify/namable\_classify/utils.py#77\77;;\

MannwhitneyuResult(statistic=0.0, pvalue=0.0002913752913752914)

看来thundersvm在GPU上加速还是显著地更快一些。

### 1.2 实现 Hinge Loss+SGD 版本的 Soft Margin Linear SVM 并对其进行调参优化

现在我们已经调研并且运行了已有SVM库的代码，了解了他们的用法以及在MNIST数据集上的表现。接下来我们将手写实现SVM，以及对其进行优化。

接下来的内容请见文件 [02svm\_handy\_crafted\_linear](./02svm_handy_crafted_linear.html)。

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