[*林远钊/1700010672*]{}

机器学习第4次上机作业

集成学习Review

基本介绍

集成学习（ensemble learning）通过构建并结合读个学习器来完成学习的任务。集成学习主要有两类：

一类是通过平均方法（即Bagging算法），这种方法的思想是通过构建几个独立的估计学习期并且平均他们的预测来得到结果。平均而言，组合后的估计要好于其中任何一个基学习器的估计。

另一类是提升方法，按照一个序列构造基学习器并且尝试在这个过程中减少聚合学习结果的偏误，其背后的思想是通过集成几个弱学习器来形成一个有效的预测器。

值得一提的是，Bagging方法适用于较复杂的、强的模型，因为他能够有效降低过拟合；而Boosting方法则一般对弱学习器更有效。

问题叙述

MultiBoosting 算法将 AdaBoost 作为 Bagging 的基学习器，Iterative Bagging算法则是将Bagging作为AdaBoost的基学习器。实现并比较二者的优缺点。

实现工具

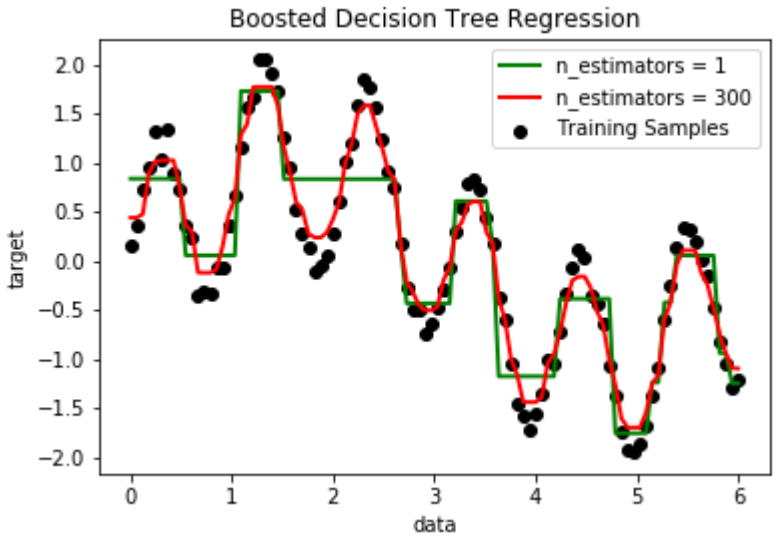
本次上机作业中主要通过scikit-learn这一python包来实现支持向量机。在使用之前需要对其进行一定的了解和学习。

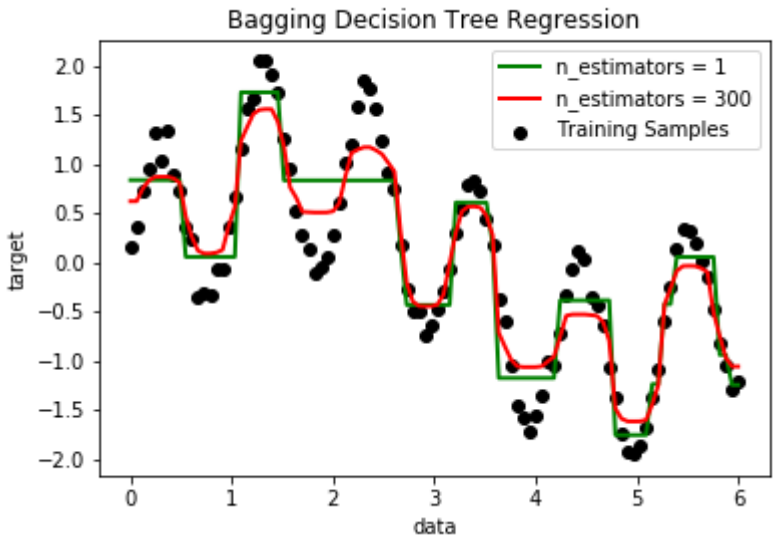
sklearn(scikit-learn包的常用缩写)中，Bagging方法有 BaggingClassifier和BaggingRegressor两个类可以实现，它们接受一个由使用者设定的基学习器参数和一个代表随机选择训练集的方案的参数。以k近邻方法为基学习器的官方代码示例如下

```
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
bagging = BaggingClassifier(KNeighborsClassifier(),
                max_samples=0.5, max_features=0.5)
```

sklear中的sklearn.ensemble 模块中包含了较流行的提升算法如AdaBoost。

在正式使用这两个包实现本次问题前，先利用官方给出的示例对AdaBoost和Bagging算法进行简单的应用，对一个生成的数据给出学习结果如下





---

python实现

由于本次上机中使用了包，大可把其作为黑箱，其中具体实现细节不必深究。以下比较AdaBoost、 Bagging、MultiAdaBoost、Iterative Bagging算法在分类和回归问题中的性能。

分类问题

首先在官方网站的示例代码的基础上进行一个二分类任务的学习和可视化任务，从而对AdaBoost、 Bagging、MultiAdaBoosting和Iterative Bagging四种算法有一些基本的了解

分别运行四种算法得到：

- AdaBoost 算法耗时0.06894421577453613 s
- Bagging 算法耗时0.01584601402282715 s
- MultiAdaBoosting算法耗时0.6348793506622314 s
- Iter Bagging算法耗时3.324193239212036 s

其得到的分类边界图分别为

综合以上信息能够得到一些简单的结果——四种算法中 AdaBoost算法得到的模型预测效果稍差（在(a)图中可以看出，在A、B两类混合的区域中分类效果较差），并且IterBagging 算法的运行时间显著长于其他三种。

Breast Cancer数据集

接下来，对于一个实际的数据集——Breast_Cancer数据集，进行分类问题的训练，并且通过10折验证来得到习得模型的性能的度量。

首先给出算法的核心代码：

```
X = load_breast_cancer().data
y = load_breast_cancer().target
bdt = AdaBoostClassifier(BaggingClassifier(), n_estimators = 200)
#bdt = AdaBoostClassifier()
#bdt = BaggingClassifier()
#bdt = BaggingClassifier(AdaBoostClassifier(),
#        max_samples = 0.63, max_features = 0.63)
bdt.fit(X, y)
```

分别运行四种算法得到运行结果

AdaBoost

```
the consequence of AdaBoost:
time cost 0.2986288070678711 s
10-folds validation' score: 0.9655172413793104 0.9482758620689655 0.9473684210526315 0.9649122807017544 0.9824561403508771 0.94736
84210526315 0.9473684210526315 0.9464285714285714 1.0 0.9642857142857143
average score is  0.9613981073373088
```

Bagging

```
the consequence of Bagging:
time cost 0.15372514724731445 s
10-folds validation' score: 0.9482758620689655 0.9137931034482759 0.8947368421052632 0.9122807017543859 0.9824561403508771 0.98245
61403508771 0.9122807017543859 1.0 0.9464285714285714 1.0
average score is  0.9492708063261602
```

MultiAdaBoost

```
the consequence of MultiAdaBoosting:
time cost 1.3461744785308838 s
10-folds validation' score: 0.9827586206896551 0.9482758620689655 0.9473684210526315 0.9649122807017544 1.0 0.9824561403508771 0.9
649122807017544 0.9642857142857143 0.9642857142857143 0.9642857142857143
average score is  0.9683540748422779
```

Iterative Bagging

```
the consequence of Iterative Bagging:
time cost 9.171002388000488 s
10-folds validation' score: 0.9827586206896551 0.896551724137931 0.9122807017543859 0.9824561403508771 1.0 1.0 0.9649122807017544
0.9821428571428571 0.9821428571428571 0.9642857142857143
average score is  0.9667530896206031
```

从这些数据可以得到一些基本的结果：

- MultiAdaBoost和Iterative Bagging算法的运行时间显著长于AdaBoost、Bagging算法（这是显然符合事实的
- Iterative Bagging 和 MultiAdaBoost算法相比，在10折验证的过程中，波动（方差）显著高于MultiAdaBoosting算法，最终的预测正确率则相差差不多，和两个基算法比起来，学习率略有上升

---

wine数据集

类似的，我们使用wine数据集，再对四种算法进行测试。

Iterative Bagging

```
the consequence of Iterative Bagging:
time cost 0.044667959213256836 s
10-folds validation' score: 0.8421052631578947 0.8888888888888888 0.8333333333333334 0.9444444444444444 0.8888888888
888888 1.0 1.0 0.9444444444444444 0.9411764705882353 1.0
average score is  0.9283281733746132
```

MultiAdaBoosting

```
the consequence of MultiBoosting:
time cost 0.9141650199890137 s
10-folds validation' score: 0.7894736842105263 0.8333333333333334 1.0 1.0 0.9444444444444444 0.9444444444444444 0.83
33333333333334 0.9444444444444444 1.0 1.0
average score is  0.9289473684210527
```

AdaBoosting

```
the consequence of AdaBoosting:
time cost 0.1215214729309082 s
10-folds validation' score: 0.5263157894736842 0.9444444444444444 1.0 0.9444444444444444 0.8888888888888888 0.888888
8888888888 0.7222222222222222 0.8333333333333334 1.0 1.0
average score is  0.8748538011695907
```

Bagging

```
the consequence of Bagging:
time cost 0.07787775993347168 s
10-folds validation' score: 0.9473684210526315 0.8333333333333334 0.9444444444444444 0.9444444444444444 1.0 1.0 1.0
0.9444444444444444 1.0 1.0
average score is  0.9614035087719298
```

- 在这个数据集中Bagging算法获得了出人意料的高学习率
- 在运行所需时间的方面，此时MultiAdaBoosting算法所需时间显著多于其他三个算法。

回归问题

在以下回归问题中，评分（score）的标准暂定为 R^2

Boston数据集

我们首先使用Boston房价数据集对四种算法在回归问题中的表现进行一些测试。 依次展示 AdaBoost、 Bagging、MultiAdaBoosting、Iterative Bagging的运行结果

AdaBoost

```
fit_time : [0.6918509  0.55091929 0.52968645 0.5247035  0.55005836 0.5317111
 0.54560018 0.55853605 0.55993462 0.558496  ]
average fit_time : 0.5601496458053589
score_time : [0.019876   0.01678014 0.01887846 0.01682663 0.01837063 0.01788497
 0.02087426 0.015872   0.01896024 0.01885033]
average score_time : 0.018317365646362306
test_score : [ 0.70951069  0.82466169  0.49115745  0.80094416  0.81106039  0.47938887
  0.22279379  0.41625281 -0.11889723  0.1870999 ]
average test_score : 0.4823972514180549
train_score : [0.95075861 0.94999475 0.94836452 0.94305198 0.95172287 0.93909908
 0.94726556 0.95150693 0.94715049 0.95403973]
average train_score : 0.9482954530725651
```

Bagging

```
fit_time : [0.51085782 0.49747539 0.47284746 0.46772981 0.48905754 0.44892073
 0.4771111  0.41018319 0.50869155 0.5074091 ]
average fit_time : 0.47902836799621584
score_time : [0.02128768 0.02130413 0.02265763 0.0178113  0.02278233 0.01884842
 0.01789737 0.01735687 0.01785636 0.02233005]
average score_time : 0.02001321315765381
test_score : [ 0.63391754  0.82338805  0.38536969  0.76971804  0.80469648  0.57624369
  0.40749232  0.35310365 -0.5996746   0.08043664]
average test_score : 0.42346915051473094
train_score : [0.91971812 0.91529256 0.9148581  0.91614759 0.91739887 0.90491042
 0.91972405 0.91912813 0.91392016 0.92426704]
average train_score : 0.9165365049576559
```

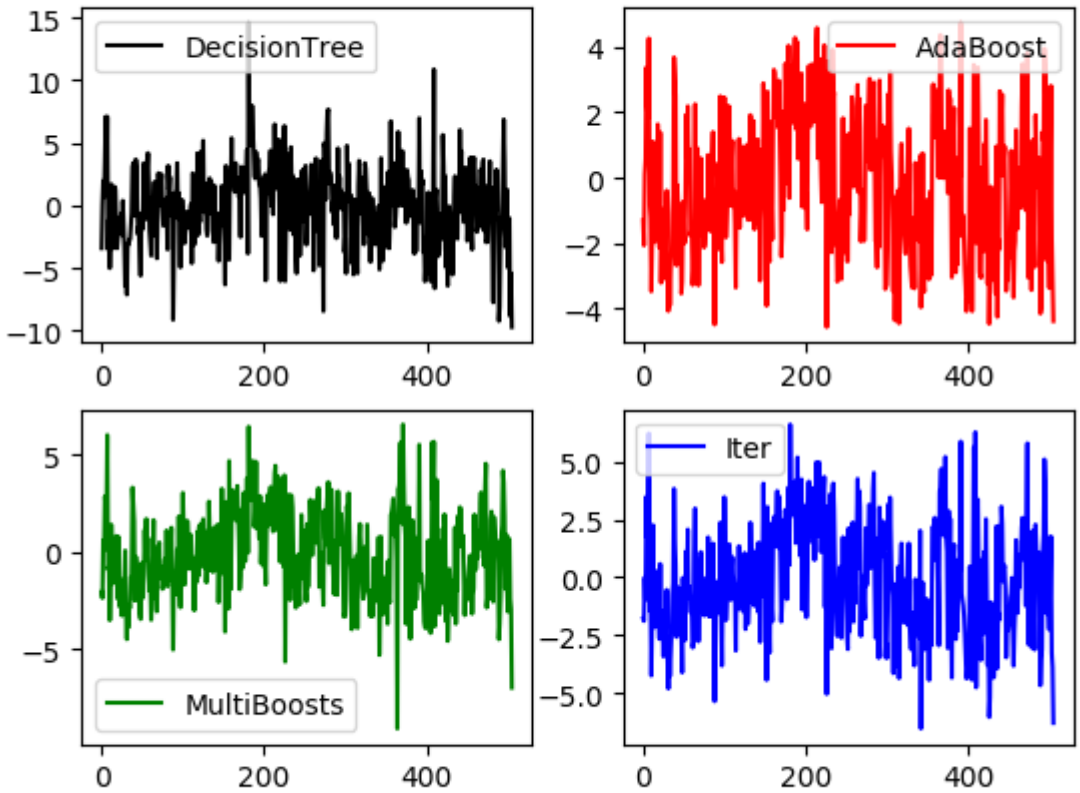
MultiAdaBoost

```
fit_time : [18.7924242  19.21855021 19.43080544 18.45269537 19.31626987 18.40692472
 17.02565479 20.36919665 18.9938271  14.96531892]
average fit_time : 18.49716672897339
score_time : [0.89031935 0.7831862  0.72763371 0.8377502  0.87406492 0.83614731
 0.60615396 0.73808742 0.73512292 0.63343406]
average score_time : 0.7661900043487548
test_score : [ 0.66903668  0.55444155  0.43121529  0.77065496  0.80624756  0.51616462
 -0.89008292  0.27430404 -0.82547217 -0.21329277]
average test_score : 0.20932168220578787
train_score : [0.8289316  0.82229203 0.81356211 0.8021659  0.80573506 0.80167925
 0.82244009 0.87883161 0.83294287 0.81306939]
average train_score : 0.8221649905723198
```

Iterative Bagging

```
fit_time : [1.83421111 1.83024001 3.3897047  2.68832517 2.79044008 2.22353411
 1.6605382  1.93093133 2.7686801  2.56522894]
average fit_time : 2.3681833744049072
score_time : [0.0962224  0.09225631 0.17157888 0.135396   0.16661406 0.10168314
 0.11457467 0.12400031 0.17604494 0.15525484]
average score_time : 0.1333625555038452
test_score : [ 0.63371825  0.56435185  0.64733854  0.74943999  0.72699648  0.43815368
 -0.30904525  0.27980875 -0.67868801 -0.15351768]
average test_score : 0.28985565943251623
train_score : [0.83143585 0.82484402 0.82460493 0.81485623 0.81106456 0.81189092
 0.82846624 0.87272354 0.84344332 0.81724546]
average train_score : 0.8280575073209612
```

同时，还比较了决策树桩、AdaBoost算法、MultiAdaBoosting和Iterative Bagging算法 的预测值和真实值之差



- 在回归问题的第一次尝试中，发现MultiAdaBoost算法所耗时间平均比其余三个高一个数量级
- 本次回归问题中，新的两个算法的学习效果似乎并不好

Diabetes数据集

再尝试一下Diabetes数据集的算法运行结果

展示的运行结果分别是MultiBoosting、Iterative Bagging、Bagging、AdaBoosting

MultiAdaBoost

```
fit_time : [15.36595249 14.94350958 12.03502274  8.1339066  14.91319847 13.09734058
 11.717031   10.98193812 12.36230898 12.78493214]
average fit_time : 12.633514070510865
score_time : [0.66765976 0.60519648 0.45485282 0.34328008 0.45988393 0.44540882
 0.61596584 0.3931222  1.09269142 0.59020472]
average score_time : 0.566826605796814
test_score : [0.43255391 0.15515197 0.4089785  0.53916372 0.33174718 0.523541
 0.40618006 0.38337089 0.38831341 0.55782345]
average test_score : 0.41268240808257717
train_score : [0.52956996 0.54376458 0.53171763 0.51230751 0.53790675 0.52062666
 0.53426982 0.54089379 0.54485083 0.51572833]
average train_score : 0.5311635863681713
```

Iterative Bagging

```
fit_time : [2.97504568 1.5720315  2.27635193 1.91006589 1.64523053 1.96713018
 1.29897976 1.96171355 1.76966    1.85158205]
average fit_time : 1.922779107093811
score_time : [0.1577251  0.1006093  0.16352153 0.07588649 0.22168541 0.08633804
 0.10366297 0.10168076 0.090276   0.13195896]
average score_time : 0.12333445549011231
test_score : [0.44079219 0.18025233 0.43556312 0.52950156 0.35153254 0.49918192
 0.4032663  0.33793954 0.42683982 0.56014106]
average test_score : 0.4165010385447312
train_score : [0.52627179 0.53182089 0.53328555 0.49484762 0.52923798 0.51540674
 0.51464832 0.53364109 0.5302532  0.49683148]
average train_score : 0.5206244661849642
```

Bagging

```
fit_time : [0.14731073 0.11817694 0.13793015 0.12300515 0.14780641 0.13190269
 0.18107319 0.15719628 0.13093591 0.1715734 ]
average fit_time : 0.1446910858154297
score_time : [0.00808263 0.00731301 0.00693226 0.01091242 0.00793505 0.01537991
 0.01389527 0.007936   0.00849128 0.01339173]
average score_time : 0.010026955604553222
test_score : [0.42274824 0.21522383 0.40118116 0.57327322 0.3587236  0.49933201
 0.38847872 0.19864249 0.32913958 0.57985876]
average test_score : 0.39666016127489645
train_score : [0.49707312 0.50402332 0.49936046 0.48440777 0.50559165 0.48351752
 0.50048532 0.50909625 0.51116214 0.48655505]
average train_score : 0.49812726079570674
```

AdaBoost

```
fit_time : [0.03076959 0.03620553 0.03219223 0.0208323  0.02526021 0.04013062
 0.06045413 0.03372765 0.03819013 0.01937962]
average fit_time : 0.03371419906616211
score_time : [0.001441   0.00198174 0.00152111 0.0008533  0.00096393 0.00149035
 0.00248146 0.00308657 0.00198412 0.00099182]
average score_time : 0.001679539680480957
test_score : [0.48760114 0.15562626 0.43280132 0.50409688 0.37282089 0.48707822
 0.41499599 0.31021942 0.412193   0.58527148]
average test_score : 0.4162704598672959
train_score : [0.49126073 0.50833145 0.5209734  0.49304025 0.53007408 0.49091843
 0.52915781 0.52392823 0.52901713 0.48726137]
average train_score : 0.5103962882144442
```

- 在此次回归问题中，新的 两个算法的解释力比起原本的有一定提升，但并不十分显著
- MultiAdaBoosting 算法的运行时间问题仍然突出

总结和思考

在所有实验开始之前，依照笔者原本的考虑：

MultiAdaBoost是以提升算法AdaBoost作为Bagging的基学习器，这个算法理当有更好的效果。因为一般情况下，Bagging方法适用于较复杂的、强的模型，因为他能够有效降低过拟合；而Boosting方法则一般对弱学习器更有效，因为它的长处在于将弱学习器提升增强。

因而，若以AdaBoost为基础，能为Bagging算法提供一个较强的已有学习器，再由Bagging算法降低过拟合风险，可以得到较好的预测效果；而反之，若以Bagging算法为基学习器，而Bagging算法又以默认的不太强的学习算法训练，这妨碍了Bagging算法发挥其本应有的功效。而通过Bagging算法得到的较强的基学习器，又不是提升算法所需要的，获得性能提升并不明显，因而MultiAdaBoost算法当显著好于Iterative Bagging算法。

但是在对几个问题进行实验之后，发现并没有证据可以支持笔者的这个想法，出现了一些数据上的其他的共同点

- Iterative Bagging算法学习率的方差普遍偏大
- MultiAdaBoosting 算法的运行时间普遍比其他算法长一个数量级
- Iterative Bagging 和MultiAdaBoosting算法和原本的AdaBoost算法和Bagging算法比起来，并没有显著的性能提升（这一点倒可以理解，本身AdaBoost和Bagging算法的学习效果并不差，进一步提升比较困难）

虽然最开始做了一些看起来似乎有道理的考虑，但是似乎并未能从实验数据中找到支持。最后基于几组实验结果，观察数据中表现出的一些共同点，给出了一些可能的方向。

同时，也发现了一些需要努力的方向——比如对实验数据的处理还比较迷茫，并不知道有哪些量是应当研究的，只选取了一些简单的数据。再比如对于自己总结出的共同点还不能给出一个合理的解释。

（PS：由于本次作业中图片较多，使用Latex 排版非常乱，因而转用markdown，比起Latex 可能不是那么正式，向助教学长表示抱歉qwq）