CS165B: Intro. to Machine Learning, Fall 2020
Programming assignment # 4

Due: Sunday 11:59pm, Dec. 13th

IMPORTANT NOTE You can discuss the problems with your TAs and instructor, but all answers and codes written down and turned in must be your own work.

Implement both **Bagging** and **AdaBoost** algorithm.
The base classifiers are the decision trees, but these trees are of two kinds: (1) a sophisticated one with depth no more than 20 and (2) a simple one with depth no more than 2 (4 leaf nodes maximum, often called a decision stump). As for the sophisticated trees, you are allowed to aggregate 50 of them and they will be used as a bagging ensemble. You will use the majority vote to determine the class label. The simple trees will be used with a boosting ensemble under the Adaboost framework and you will use 1,000 of them.

Data Set

We will retry the same Red Wine Quality Data Sets in the previous assignment to see if you can improve on the single decision tree results. For simplicity, we will have 2 classes of wine instead of 3 classes by merging the "good/2" and the "normal/1" class as one. The samples are labeled as: "good/1" (QS>=6) and "bad/0" (QS<6). Then, we partition the data set into a training set and a testing set including 1000 samples and 599 samples respectively. You are provided with both the training set and the testing set.

Skeleton code

A Bagging/Boosting instance is able to take in a feature array `X` (a numpy array of size $(n, 11)$) and a label array `y` (a numpy array of size $(n,)$), train and store the many classifiers by calling the `train` method. Then, it should be able to output label prediction of shape $(k,)$ given a feature array `X_test` of size $(k, 11)$ by calling the `test` method. Please refer to the skeleton code file for the meaning of each input arguments.

```
class Bagging(object):
    def __init__(self, n_classifiers, max_depth):
    def train(self, X, y):
    def test(self, X):
class Boosting(object):
    def __init__(self, n_classifiers, max_depth):
    def train(self, X, y):
    def test(self, X):
```

Requirements

Bagging

- `n_classifiers` is fixed to 50 and `max_depth` is fixed to 20. This combination should be able to get you a good baseline performance. You do not need to do grid search this time.
- You can use `sklearn.tree.DecisionTreeClassifier` to build each tree. This is the only library function allowed for this assignment.

Boosting

- `n_classifiers` is fixed to 1000 and `max_depth` is fixed to 2. You do not need to do grid search this time.
- You can use `sklearn.tree.DecisionTreeClassifier` to build each tree. This is the only library function allowed for this assignment. **You must set the input argument `splitter='random'` if you use this function.**
- Grading will be based on testing accuracy given that all requirements above are fulfilled.
- (Optional) Implement more functionality or variants of Bagging and Boosting algorithm to improve the testing accuracy. You can also try preprocessing the data.

Hint

Bagging

To create component classifiers for bagging, the standard practice is to perform data resampling for training different component classifiers. If you would like to better "balance" the training data sets in your training process is also up to you.
To implement boostrap, you could consider using `np.random.choice`. Read through the numpy documentation and set the correct input arguments.

Boosting

Recall that boosting is all about proper weighing: in the training process, samples will receive different weights through the iterative process so that "troublesome" samples will receive progressively heavier weights for later classifiers to focus on. During the testing, classifiers will be weighed properly based on their training performance.
One caveat is that Adaboost's good performance is guaranteed only if the component classifiers achieve $> 50\%$ accuracy. For a binary classifier, this means that a component classifier need only be better than a coin flip. However, for a multi-class classifier with k classes, math dictates that the component classifiers still must achieve $> 50\%$ accuracy (not $> \frac{1}{k}$ chance accuracy), which can be hard to do when k is large. While multi-class Adaboost algorithm does exist, in this assignment, you will implement the standard Adaboost scheme for a binary classifier to be used.
You can use `scipy.stats.mode(prediction, axis=0).mode.reshape(-1)` to get the majority vote from the array of ensemble predictions. Read through the scipy documentation to understand how it works.