Leveraging weak learners via adaptive boosting

In this last section about ensemble methods, we will discuss **boosting** with a special focus on its most common implementation, **AdaBoost** (**Adaptive Boosting**).

The original idea behind AdaBoost was formulated by Robert E. Schapire in 1990. *The Strength of Weak Learnability*, *R. E. Schapire*, *Machine Learning*, 5(2): 197-227, *1990*. After Robert Schapire and Yoav Freund presented the AdaBoost algorithm in the *Proceedings of the Thirteenth International Conference* (ICML 1996), AdaBoost became one of the most widely used ensemble methods in the years that followed (*Experiments with a New Boosting Algorithm* by *Y. Freund*, *R. E. Schapire*, and others, *ICML*, volume 96, 148-156, *1996*). In 2003, Freund and Schapire received the Goedel Prize for their groundbreaking work, which is a prestigious prize for the most outstanding publications in the field of computer science.

In boosting, the ensemble consists of very simple base classifiers, also often referred to as **weak learners**, which often only have a slight performance advantage over random guessing—a typical example of a weak learner is a decision tree stump. The key concept behind boosting is to focus on training samples that are hard to classify, that is, to let the weak learners subsequently learn from misclassified training samples to improve the performance of the ensemble.

The following subsections will introduce the algorithmic procedure behind the general concept boosting and a popular variant called **AdaBoost**. Lastly, we will use scikit-learn for a practical classification example.

How boosting works

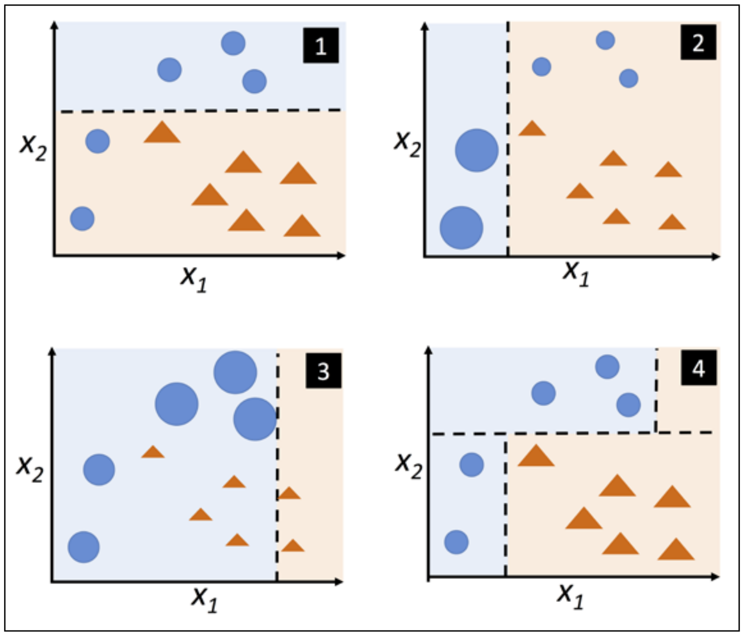
In contrast to bagging, the initial formulation of boosting, the algorithm uses random subsets of training samples drawn from the training dataset without replacement; the original boosting procedure is summarized in the following four key steps:

1. Draw a random subset of training samples d_1 without replacement from training set D to train a weak learner C_1.
2. Draw a second random training subset d_2 without replacement from the training set and add 50 percent of the samples that were previously misclassified to train a weak learner C_2.

3. Find the training samples d_3 in training set D, which C_1 and C_2 disagree upon, to train a third weak learner C_3.
4. Combine the weak learners C_1, C_2, and C_3 via majority voting.

As discussed by Leo Breiman (*Bias, variance, and arcing classifiers*, *L. Breiman*, *1996*), boosting can lead to a decrease in bias as well as variance compared to bagging models. In practice, however, boosting algorithms such as AdaBoost are also known for their high variance, that is, the tendency to overfit the training data (*An improvement of AdaBoost to avoid overfitting*, *G. Raetsch*, *T. Onoda*, and *K. R. Mueller*. Proceedings of the International Conference on Neural Information Processing, CiteSeer, 1998).

In contrast to the original boosting procedure as described here, AdaBoost uses the complete training set to train the weak learners where the training samples are reweighted in each iteration to build a strong classifier that learns from the mistakes of the previous weak learners in the ensemble. Before we dive deeper into the specific details of the AdaBoost algorithm, let's take a look at the following figure to get a better grasp of the basic concept behind AdaBoost:



To walk through the AdaBoost illustration step by step, we start with subfigure **1**, which represents a training set for binary classification where all training samples are assigned equal weights. Based on this training set, we train a decision stump (shown as a dashed line) that tries to classify the samples of the two classes (triangles and circles), as well as possibly by minimizing the cost function (or the impurity score in the special case of decision tree ensembles).

For the next round (subfigure **2**), we assign a larger weight to the two previously misclassified samples (circles). Furthermore, we lower the weight of the correctly classified samples. The next decision stump will now be more focused on the training samples that have the largest weights—the training samples that are supposedly hard to classify. The weak learner shown in subfigure **2** misclassifies three different samples from the circle class, which are then assigned a larger weight, as shown in subfigure **3**.

Assuming that our AdaBoost ensemble only consists of three rounds of boosting, we would then combine the three weak learners trained on different reweighted training subsets by a weighted majority vote, as shown in subfigure **4**.

Now that have a better understanding behind the basic concept of AdaBoost, let's take a more detailed look at the algorithm using pseudo code. For clarity, we will denote element-wise multiplication by the cross symbol $(\times)$ and the dot-product between two vectors by a dot symbol $(\cdot)$:

```
1. Set the weight vector w to uniform weights, where $\sum_i w_i = 1$.
2. For j in m boosting rounds, do the following:
    a. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
    b. Predict class labels: $\hat{y} = \text{predict}(C_j, \boldsymbol{X})$.
    c. Compute weighted error rate: $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} \neq \boldsymbol{y})$.
    d. Compute coefficient: $\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$.
    e. Update weights: $\boldsymbol{w} := \boldsymbol{w} \times \exp(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$.
    f. Normalize weights to sum to 1: $\boldsymbol{w} := \boldsymbol{w} / \sum_i w_i$.
3. Compute the final prediction: $\hat{\boldsymbol{y}} = \left( \sum_{j=1}^{m} \left( \boldsymbol{\alpha}_j \times \text{predict}(C_j, \boldsymbol{X}) \right) > 0 \right)$.
```

Note that the expression $(\hat{\boldsymbol{y}} \neq \boldsymbol{y})$ in step 2c refers to a binary vector consisting of 1s and 0s, where a 1 is assigned if the prediction is incorrect and 0 is assigned otherwise.

Although the AdaBoost algorithm seems to be pretty straightforward, let's walk through a more concrete example using a training set consisting of 10 training samples, as illustrated in the following table:

Sample indices	x	y	Weights	$\hat{y}$(x <= 3.0)?	Correct?	Updated weights
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	No	0.167
8	8.0	1	0.1	-1	No	0.167
9	9.0	1	0.1	-1	No	0.167
10	10.0	-1	0.1	-1	Yes	0.072

The first column of the table depicts the sample indices of training samples 1 to 10. In the second column, we see the feature values of the individual samples, assuming this is a one-dimensional dataset. The third column shows the true class label, y_i, for each training sample x_i, where $y_i \in \{1, -1\}$. The initial weights are shown in the fourth column; we initialize the weights uniformly (assigning the same constant value) and normalize them to sum to one. In the case of the 10-sample training set, we therefore assign 0.1 to each weight w_i in the weight vector **w**. The predicted class labels $\hat{\boldsymbol{y}}$ are shown in the fifth column, assuming that our splitting criterion is $x \leq 3.0$. The last column of the table then shows the updated weights based on the update rules that we defined in the pseudo code.

Since the computation of the weight updates may look a little bit complicated at first, we will now follow the calculation step by step. We start by computing the weighted error rate ε as described in step 2c:

$$\varepsilon = 0.1\times0 + 0.1\times0 + 0.1\times0 + 0.1\times0 + 0.1\times0 + 0.1\times0 + 0.1\times1 + 0.1\times1$$
$$+ 0.1\times1 + 0.1\times0 = \frac{3}{10} = 0.3$$

Next, we compute the coefficient α_j —shown in step 2d—which is later used in step 2e to update the weights, as well as for the weights in the majority vote prediction (step 4):

$$\alpha_j = 0.5 \log\left(\frac{1-\varepsilon}{\varepsilon}\right) \approx 0.424$$

After we have computed the coefficient α_j, we can now update the weight vector using the following equation:

$$\boldsymbol{w} := \boldsymbol{w} \times \exp\left(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y}\right)$$

Here, $\hat{\boldsymbol{y}} \times \boldsymbol{y}$ is an element-wise multiplication between the vectors of the predicted and true class labels, respectively. Thus, if a prediction $\hat{y}_i$ is correct, $\hat{y}_i \times y_i$ will have a positive sign so that we decrease the *i*th weight, since α_j is a positive number as well:

$$0.1 \times \exp\left(-0.424 \times 1 \times 1\right) \approx 0.065$$

Similarly, we will increase the *i*th weight if $\hat{y}_i$ predicted the label incorrectly, like this:

$$0.1 \times \exp\left(-0.424 \times 1 \times (-1)\right) \approx 0.153$$

Alternatively, it's like this:

$$0.1 \times \exp\left(-0.424 \times (-1) \times (1)\right) \approx 0.153$$

After we have updated each weight in the weight vector, we normalize the weights so that they sum up to one (step 2f):

$$\boldsymbol{w} := \frac{\boldsymbol{w}}{\sum_i w_i}$$

Here, $\sum_i w_i = 7 \times 0.065 + 3 \times 0.153 = 0.914$.

Thus, each weight that corresponds to a correctly classified sample will be reduced from the initial value of 0.1 to $0.065/0.914 \approx 0.071$ for the next round of boosting. Similarly, the weights of the incorrectly classified samples will increase from 0.1 to $0.153/0.914 \approx 0.167$.

Applying AdaBoost using scikit-learn

The previous subsection introduced AdaBoost in a nutshell. Skipping to the more practical part, let's now train an AdaBoost ensemble classifier via scikit-learn. We will use the same Wine subset that we used in the previous section to train the bagging meta-classifier. Via the `base_estimator` attribute, we will train the `AdaBoostClassifier` on 500 decision tree stumps:

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
                                  random_state=1,
...                               max_depth=1)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                          n_estimators=500,
...                          learning_rate=0.1,
...                          random_state=1)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 0.916/0.875
```

As we can see, the decision tree stump seems to underfit the training data in contrast to the unpruned decision tree that we saw in the previous section:

```
>>> ada = ada.fit(X_train, y_train)
>>> y_train_pred = ada.predict(X_train)
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print('AdaBoost train/test accuracies %.3f/%.3f'
...       % (ada_train, ada_test))
AdaBoost train/test accuracies 1.000/0.917
```

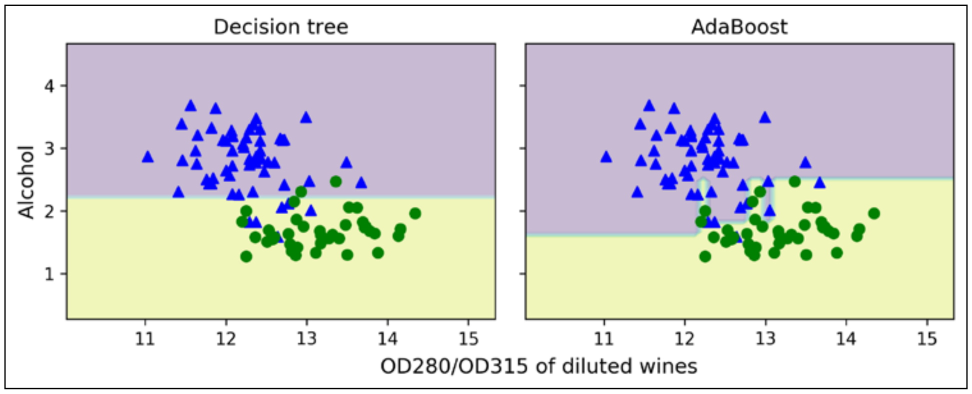
As we can see, the AdaBoost model predicts all class labels of the training set correctly and also shows a slightly improved test set performance compared to the decision tree stump. However, we also see that we introduced additional variance by our attempt to reduce the model bias—a higher gap between training and test performance.

Although we used another simple example for demonstration purposes, we can see that the performance of the AdaBoost classifier is slightly improved compared to the decision stump and achieved the very similar accuracy scores as the bagging classifier that we trained in the previous section. However, we shall note that it is considered bad practice to select a model based on the repeated usage of the test set. The estimate of the generalization performance may be over-optimistic, which we discussed in more detail in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

Lastly, let us check what the decision regions look like:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, ada],
...                         ['Decision Tree', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='red',
...                        marker='o')
...     axarr[idx].set_title(tt)
...     axarr[0].set_ylabel('Alcohol', fontsize=12)
>>> plt.text(10.2, -0.5,
...          s='OD280/OD315 of diluted wines',
...          ha='center',
...          va='center',
...          fontsize=12)
>>> plt.show()
```

By looking at the decision regions, we can see that the decision boundary of the AdaBoost model is substantially more complex than the decision boundary of the decision stump. In addition, we note that the AdaBoost model separates the feature space very similarly to the bagging classifier that we trained in the previous section:



As concluding remarks about ensemble techniques, it is worth noting that ensemble learning increases the computational complexity compared to individual classifiers. In practice, we need to think carefully about whether we want to pay the price of increased computational costs for an often relatively modest improvement in predictive performance.

An often-cited example of this trade-off is the famous $1 million *Netflix Prize*, which was won using ensemble techniques. The details about the algorithm were published in *The BigChaos Solution to the Netflix Grand Prize* by *A. Toescher*, *M. Jahrer*, and *R. M. Bell*, Netflix prize documentation, *2009*, which is available at `http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf`. The winning team received the $1 million grand prize money; however, Netflix never implemented their model due to its complexity, which made it infeasible for a real-world application:

"*We evaluated some of the new methods offline but the additional accuracy gains that we measured did not seem to justify the engineering effort needed to bring them into a production environment.*" (`http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html`).