Contents

1 Combining Different Models for Ensemble Learning **1**
- 1.1 Learning with ensembles . . . 1
- 1.2 Bagging – building an ensemble of classifiers from bootstrap samples 1
  - 1.2.1 Bagging in a nutshell . . . 1
- 1.3 Leveraging weak learners via adaptive boosting . . . 2
  - 1.3.1 How adaptive boosting works . . . 2
- 1.4 Gradient boosting – training an ensemble based on loss gradients . . . 3
  - 1.4.1 Outlining the general gradient boosting algorithm . . . 3

2 Applying Machine Learning to Sentiment Analysis **6**
- 2.1 Introducing the bag-of-words model . . . 6
  - 2.1.1 Assessing word relevancy via term frequency-inverse document frequency . . . 6
  - 2.1.2 Processing documents into tokens . . . 7
- 2.2 Working with bigger data – online algorithms and out-of-core learning 7

3 Predicting Continuous Target Variables with Regression Analysis **8**
- 3.1 Fitting a robust regression model using RANSAC . . . 8

Chapter 1

Combining Different Models for Ensemble Learning

1.1 Learning with ensembles

The goal of ensemble methods is to combine different classifiers into a meta-classifier that has better generalization performance than each individual classifier alone.

1.2 Bagging – building an ensemble of classifiers from bootstrap samples

Instead of using the same training dataset to fit the individual classifiers in the ensemble, we draw bootstrap samples (random samples with replacement) from the initial training dataset, which is why bagging is also known as *bootstrap aggregating*.

The concept of bagging is summarized in Figure 1.1:

1.2.1 Bagging in a nutshell

In fact, random forests are a special case of bagging where we also use random feature subsets when fitting the individual decision trees.

In practice, more complex classification tasks and a dataset's high dimensionality can easily lead to overfitting in single decision trees, and this is where the bagging algorithm can really play to its strengths. Finally, we must note that the bagging algorithm can be an effective approach to reducing the variance of a model. However, bagging is ineffective in reducing model bias, that is, models that are too simple to capture the trends in the data well. This is why we want to perform bagging on an ensemble of classifiers with low bias, for example, unpruned decision trees.

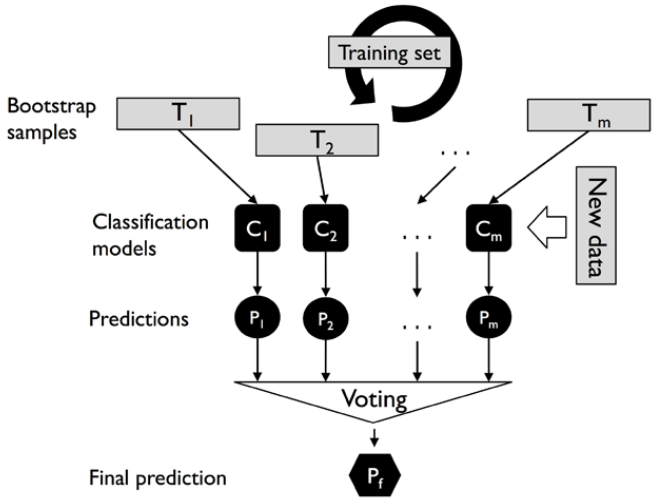

Figure 1.1: The concept of bagging

1.3 Leveraging weak learners via adaptive boosting

We will discuss **boosting**, with a special focus on its most common implementation: **Adaptive Boosting** (AdaBoost).

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 examples that are hard to classify, that is, to let the weak learners subsequently learn from misclassified training examples to improve the performance of the ensemble.

1.3.1 How adaptive boosting works

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

1. Draw a random subset (sample) of training examples, d_1, without replacement from the training dataset, D, to train a weak learner, C_1.

2. Draw a second random training subset, d_2, without replacement from the training dataset and add 50 percent of the examples that were previously misclassified to train a weak learner, C_2.

3. Find the training examples, d_3, in the training dataset, 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.

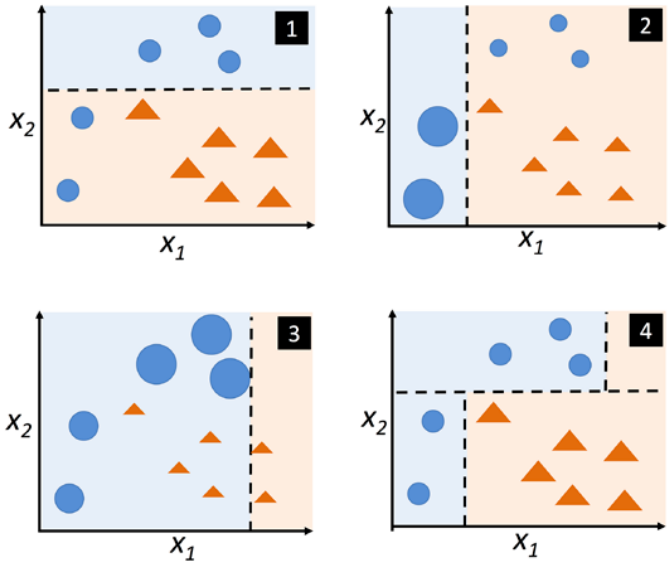

Figure 1.2: The concept of AdaBoost to improve weak learners

As discussed by Leo Breiman, 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.

1.4 Gradient boosting – training an ensemble based on loss gradients

1.4.1 Outlining the general gradient boosting algorithm

Algorithm 1: AdaBoost algorithm

begin
  initialization, Set the weight vector, $\mathbf{w}$, to uniform weights, where $\sum_i w_i = 1$;
  for $j \rightarrow 1$**to** m **do**
    Train a weighted weak learner: $C_j = train(\mathbf{X}, \mathbf{y}, \mathbf{w})$;
    Predict class labels: $\hat{y} = predict(C_j, \mathbf{X})$;
    Compute the weighted error rate: $\varepsilon = \mathbf{W} \cdot (\hat{\mathbf{y}} \neq \mathbf{y})$;
    Compute the coefficient:$\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$;
    Update the weights: $\mathbf{w} \leftarrow \mathbf{w} \times \exp(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y})$;
    Normalize the weights to sum to 1:$\mathbf{w} \leftarrow \mathbf{w} / \sum_i w_i$;
  end
  Compute the final prediction: $\hat{\mathbf{y}} = \left(\sum_{j=1}^{m} (\alpha_j \times predict(C_j, \mathbf{X})) \right)$;
end

Algorithm 2: The general outline of the gradient boosting algorithm.

begin

Initialize a model to return a constant prediction value. For this, we use a decision tree root node; that is, a decision tree with a single leaf node. We denote the value returned by the tree as $\hat{y}$, and we find this value by minimizing a differentiable loss function L that we will define later:

$$F_0(x) = \underset{\hat{y}}{\operatorname{argmin}} \sum_{i=1}^{n} L(y_i, \hat{y})$$

Here, n refers to the n training examples in our dataset;

foreach *m=1,. . . , M* **do**

Compute the difference between a predicted value $F(x_i) = \hat{y}_i$and the class label y_i. This value is sometimes called the pseudo-response or pseudo-residual. More formally, we can write this pseudo-residual as the negative gradient of the loss function with respect to the predicted values:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}, \; i = 1, \ldots, n$$

Note that in the notation above $F(x)$ is the prediction of the previous tree, $F_{m-1}(x)$;

Fit a tree to the pseudo-residuals rim. We use the notation $R_{jm}$to denote the $j = 1...Jm$ leaf nodes of the resulting tree in iteration m. For each leaf node Rjm, we compute the following output value:

$$\gamma_{jm} = \underset{\gamma}{\operatorname{argmin}} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$

At this point, we can already note that leaf nodes R_{jm} may contain more than one training example, hence the summation;

Update the model by adding the output values γ_m to the previous tree:

$$F_m(x) = F_{m-1}(x) + \eta \gamma_m$$

However, instead of adding the full predicted values of the current tree γ_m to the previous tree F_{m-1} , we scale γ_m by a learning rate η , which is typically a small value between 0.01 and 1. In other words, we update the model incrementally by taking small steps, which helps avoid overfitting.

end

end

Chapter 2

Applying Machine Learning to Sentiment Analysis

2.1 Introducing the bag-of-words model

The idea behind bag-of-words is quite simple and can be summarized as follows:

1. We create a vocabulary of unique tokens—for example, words—from the entire set of documents.
2. We construct a feature vector from each document that contains the counts of how often each word occurs in the particular document.

2.1.1 Assessing word relevancy via term frequency-inverse document frequency

you will learn about a useful technique called the term frequency-inverse document frequency (tf-idf), which can be used to downweight these frequently occurring words in the feature vectors. The tf-idf can be defined as the product of the term frequency and the inverse document frequency:

$$tf-idf(t,d) = tf(t,d) \times idf(t,d)$$

Here, $tf(t,d)$ is the term frequency, and $idf(t,d)$ is the inverse document frequency, which can be calculated as follows:

$$idf(t,d) = \log \frac{n_d}{1+df(d,t)} \tag{2.1}$$

Here, n_d is the total number of documents, and $df(d,t)$ is the number of documents, d, that contain the term t.

Equation 2.1 for the inverse document frequency implemented in scikit-learn is computed as follows:

$$idf(t,d) = \log \frac{1+n_d}{1+df(d,t)} \tag{2.2}$$

Similarly, the tf-idf computed in scikit-learn deviates slightly from the default equation we defined earlier:

$$tf - idf(t,d) = tf(t,d) \times (idf(t,d) + 1)$$

Note that the "+1" in the idf equation is due to setting `smooth_idf=True`, which is helpful for assigning zero weight (that is, $idf(t,d) = log(1) = 0$) to terms that occur in all documents.

2.1.2 Processing documents into tokens

After successfully preparing the movie review dataset, we now need to think about how to split the text corpora into individual elements. One way to tokenize documents is to split them into individual words by splitting the cleaned documents at their whitespace characters.

In the context of tokenization, another useful technique is word stemming, which is the process of transforming a word into its root form.

2.2 Working with bigger data – online algorithms and out-of-core learning

Since not everyone has access to supercomputer facilities, we will now apply a technique called out-of-core learning, which allows us to work with such large datasets by fitting the classifier incrementally on smaller batches of a dataset.

The word2vec model

A more modern alternative to the bag-of-words model is word2vec, an algorithm that Google released in 2013 (*Efficient Estimation of Word Representations in Vector Space by T.Mikolov, K. Chen, G. Corrado, and J. Dean*,).
The word2vec algorithm is an unsupervised learning algorithm based on neural networks that attempts to automatically learn the relationship between words. The idea behind word2vec is to put words that have similar meanings into similar clusters, and via clever vector spacing, the model can reproduce certain words using simple vector math, for example, *king–man + woman = queen*.

Chapter 3

Predicting Continuous Target Variables with Regression Analysis

3.1 Fitting a robust regression model using RANSAC

Algorithm 3: RANdom SAmple Consensus (RANSAC) algorithm

```
1 begin
2     repeat
3         Select a random number of examples to be inliers and fit the model;
4         Test all other data points against the fitted model and add those points
            that fall within a user-given tolerance to the inliers;
5         Refit the model using all inliers;
6         Estimate the error of the fitted model versus the inliers;
7     until the performance meets a certain user-defined threshold or if a fixed
        number of iterations was reached;
8 end
```