Annotation Summary

10 annotations on 6 pages by jmichaelbrunner

x10

The function I will return 1 whenever the data label y_i does not agree with the classification prediction $G(x_i)$ and 0 when they do agree.

A better way to compute the error is to take advantage of the importance weight:

$$err = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G(x_i))}{\sum_{i=1}^{N} w_i}$$

Here we multiply each misclassification by the importance weight w_i. In this way, our error metric is more sensitive to misclassified examples that have a greater importance weight. As Charles says in this video from the class, even if we get many examples wrong, we may still get a low error rate, because in a sense some examples are more important than others.

The factor $\sum_{i=1}^{N} w_i$ in the denominator is simply a normalization factor to

#1 p.3

---

1. Initialize the importance weights $w_i = 1/N$ for all training examples *i*.
2. For m = 1 to M:
   a) Fit a classifier $G_m(x)$ to the training data **using the weights** w_i.
   b) Compute the error:

$$err_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$$

#2 p.3

c) Compute $\alpha_m = log((1 - err_m)/err_m)$
d) Update weights: $w_i \leftarrow w_i \cdot exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$ for $i = 1, 2, \ldots N$

3. Return $G(x) = sign[\sum_{m=1}^{M} \alpha_m G_m(x)]$.

#3 p.4

error

We can see that for error < 0.5, the α_m parameter is positive. The smaller the weighted training error is, the greater the alpha parameter

#4 p.4

becomes. The parameter α_m is therefore telling us how well the classifier G_m performs on training data - large α_m implies low error and therefore, accurate performance.

The last step inside the loop is to update the importance weights w_i. The old weight is multiplied by an exponential term that depends on both α_m and whether the classifier was correct at predicting the training example *i* corresponding to the importance weight w_i.

Suppose that a training example *i* is difficult to classify, i.e. a classifier

#5 p.5

additive model.

The AdaBoost algorithm is a special case of an additive model: we fit a model on elementary *basis* functions. The additive model takes the form:

$$f(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma_m).$$

Here β_m are the expansion coefficients, and b are the basis functions, parametrized by γ_m. We fit the basis functions by picking β_m and γ_m that satisfy certain optimization procedures. The resultant classifier $f(x)$ is formed by combining the base classifiers $b(x)$.

Additive models like this are quite common in machine learning. For

#6 p.6

the importance weights w_p and w_q.

Generally, the way an additive model is fitted is by minimizing some sort of a error (or loss function *L*):

$$min_{\{\beta_m, \gamma_m\}_1^M} [\, L(y_i, f(x_i)) \,] = min_{\{\beta_m, \gamma_m\}_1^M} [\, L(y_i, \sum_{m=1}^{M} \beta_m b(x_i; \gamma_m)) \,]$$

The goal is to determine the parameters $\{\beta_m, \gamma_m\}_1^M$ of the expansion which minimize the loss function *L* - remember that *L* is a function of the target label y_i and the predicted label $f(x_i)$. Once we find those parameters, we have successfully trained a model $f(x_i)$. Note however, that this is a very computationally intensive task, because we would need to find all β parameters as well as γ parameters (one for each basis function *b* for a total of 2 M parameters) all at once.

#7 p.6

Forward stagewise additive modeling approximates this minimization process by adding a new basis function b at each iteration of the algorithm. The outline of the algorithm is below:

---

1. Initialize $f_0(x) = 0$
2. For $m = 1$ *to* M :
   (a) Compute:

$$(\beta_m, \gamma_m) = arg\ min_{\beta, \gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta\, b(x_i; \gamma_i)).$$

   (b) Set $f_m(x) = f_{m-1}(x) + \beta\, b(x_i; \gamma_i)$.
3. Return $f_M(x)$ as the additive expansion $f(x)$

Let's examine the parallel between the more general *forward stagewise*

#8 p.7

additive modeling and the specific AdaBoost algorithm.

For each iteration of the forward modeling, we find parameters β_m, γ_m that minimize the loss function between the actual target y_i and the best guess of the target we have currently, at the iteration step m: $f_{m-1}(x_i) + \beta\, b(x_i; \gamma_i)$. In the case of boosting, the parameter β_m corresponds to α_m while γ_m determines the weights w_i. At each iteration, there will be a new parameter γ_m, and therefore the importance weights will be updated accordingly.

At each stage, we are fitting a classifier G_m that is parametrized by the importance weights. We are adding G_m (our basis function) to the weighted additive model we have so far $f_{m-1}(x_i)$ in a way that minimizes the error between actual and predicted target label.

We call this forward stagewise modeling because once we have

#9 p.7

base learners.

Schapire also outlines the advantages and caveats of AdaBoost. Some of the major ones are summarized here:

Advantages:

- Computationally efficient.
- No difficult parameters to set.
- Versatile - a wide range of base learners can be used with AdaBoost.

Caveats:

- Algorithm seems susceptible to uniform noise.
- Weak learner should not be too complex - to avoid overfitting.
- There needs to be enough data so that the weak learning requirement is satisfied - the base learner should perform consistently better than random guessing, with generalization error < 0.5 for binary classification problems.

This listing shows that selecting an appropriate weak learner is one of

#10 p.9

Introduction to Boosting

This document will introduce boosting as a general approach to supervised learning and it will focus on the AdaBoost algorithm as a solution to the boosting problem.

Overview and Motivation:

In class Charles and Michael introduced ensemble learning through bagging and boosting. In this article we will focus on boosting as it is more challenging and perhaps more interesting.

Boosting is one of the most popular and successful general approaches to supervised learning. The original *boosting problem* asks whether a set of *weak learners* can be combined to produce a learner with an arbitrary high accuracy. A weak learner is a learner whose performance (at classification or regression) is only slightly better than random guessing.

The rest of this article will cover the topics outlined below. The topics follow sections 10.1 - 10.3 and 10.5 from *Chapter 10: Boosting and Additive Trees* from the wonderful book *The Elements of Statistical Learning* [1], while adding some additional discussion regarding the AdaBoost algorithm.

Outline of the AdaBoost algorithm. This is the most popular boosting algorithm.
Deeper insights: the AdaBoost as an example of additive expansion (optional). This section covers material outside of class but

[1] Hastie, Tibshirani, Friedman. *The Elements of Statistical Learning,* 2nd Edition. Springer (2008).

it might be useful for you to put boosting in perspective. We will discuss the AdaBoost algorithm in more detail as part of the more general *additive model* and the *basis expansion*.
Practical considerations We will discuss how the algorithm is used in practice.
Advantages and caveats. Summary of the main advantages and caveats of AdaBoost, as presented by Schapire (one of the inventors of the algorithm).

Outline of the AdaBoost algorithm:

The AdaBoost algorithm was developed by Freund and Schapire in 1995. Originally it was designed to perform classification tasks and this is what we will focus on here. The algorithm has been later generalized to tackle regression as well.

The AdaBoost algorithm trains multiple weak classifiers on training data, and then combines those weak classifiers into a single *boosted* classifier. The combination is done through a weighted sum of the weak classifiers with weights dependent on the weak classifier accuracy.

How does every classic supervised learning task begin? It begins with a training set of data.

So let's start with a set containing *N* training examples: every example *i* contains features x_i (a vector that in general may contain multiple features) and a target classification label $y_i \in \{-1, +1\}$. Every example *i* also has an associated observation weight w_i. You can think of w_i as an importance weight that tells us how important the example *i* is for our current learning task.

The observation weight might come in as domain knowledge - for example, we may know, based on past experience, that we are more likely to encounter a training example *j* compared to example *k*. Therefore we should assign a larger weight to example *j* - it is the one that will have a greater influence on our function approximation process.

One advantage of using importance weights is that we can generalize the notion of error to take into account those weights. Let's say we have a classifier G that takes input features x_i and produces a prediction

$G(x_i) \in \{-1, 1\}$. We can measure its training error by simply counting all the misclassified training examples:

$$err_s = \sum_{i=1}^{N} I(y_i \neq G(x_i))$$

The function *I* will return 1 whenever the true label y_i does not agree with the classification prediction $G(x_i)$ and 0 when they do agree.

A better way to compute the error is to take advantage of the importance weight:

$$err = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G(x_i))}{\sum_{i=1}^{N} w_i}$$

Here we multiply each misclassification by the importance weight w_i. In this way, our error metric is more sensitive to misclassified examples that have a greater importance weight. As Charles says in this video from the class, even if we get many examples wrong, we may still get a low error rate, because in a sense some examples are more important than others.

The factor $\sum_{i=1}^{N} w_i$ in the denominator is simply a normalization factor, to ensure that the error is normalized (between *0* and *N*) in case some of the weights become very large.

In the boosting algorithm, the importance weights w_i are sequentially updated by the algorithm itself. Here is an outline of the algorithm - we are going to go through it step by step:

---

1. Initialize the importance weights $w_i = 1/N$ for all training examples *i*.
2. For m = 1 to M:
    a) Fit a classifier $G_m(x)$ to the training data **using the weights** w_i.
    b) Compute the error:

$$err_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$$

c) Compute $\alpha_m = log((1 - err_m)/err_m)$

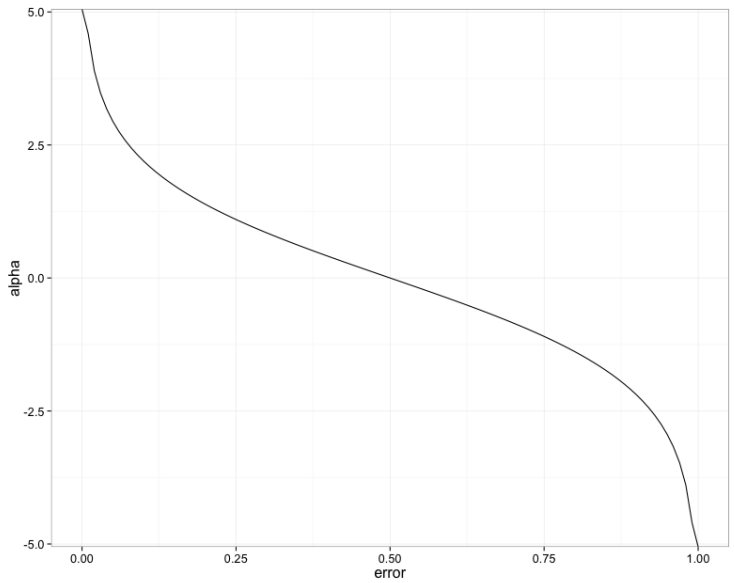
d) Update weights: $w_i \leftarrow w_i \cdot exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$ for $i = 1, 2, \ldots N$

3. Return $G(x) = sign[\sum_{m=1}^{M} \alpha_m G_m(x)]$.

Initially we set all weights to be equal to $1/N$. Then iterate through the loop (step 2) M times and fit a different classifier at each iteration. The classifier at step m is generated based on the weighted dataset using the current weights w_i.

We then compute the weighted error err_m to determine how well G_m has done on the training set. Note that the error is sensitive to the current weights w_i.

In step 2c of the algorithm the α_m parameter is computed based on the error metric. Let's take a look at a plot of the α_m parameter as a function of the error err_m to better understand their relationship. Here is a plot of the $log((1 - err_m)/err_m)$ function:



We can see that for error < 0.5, the α_m parameter is positive. The smaller the weighted training error is, the greater the alpha parameter

becomes. The parameter α_m is therefore telling us how well the classifier G_m performs on training data - large α_m implies low error and therefore, accurate performance.

The last step inside the loop is to update the importance weights w_i. The old weight is multiplied by an exponential term that depends on both α_m and whether the classifier was correct at predicting the training example *i* corresponding to the importance weight w_i.

Suppose that a training example *j* is difficult to classify - i.e. a classifier G_m fails to classify *j* correctly [$I(y_i \neq G_m(x_i)) = 1$] . As a result, the importance weight of *j* will increase: $w_j \leftarrow w_j \,.\, exp[\alpha_m]$. Then the next classifier G_{m+1} will "pay more attention" to example *j* during classification training, since j now has a greater weight. The opposite holds for examples that were correctly classified in the previous iteration - future classifiers will have a lower priority of correctly classifying such examples.

Finally we combine all classifiers $G_m \, for \, m = 1 \ldots M$ into a single boosted classifier G by doing a weighted sum on the weights α_m .

$$G(x) = sign[\sum_{m=1}^{M} \alpha_m G_m(x)].$$

In this way, classifiers that have a poor accuracy (high error rate, low α_m) are penalized in the final sum.

In the lecture, Charles presents a visual example that is very helpful in providing intuition on how Boosting can be applied. It is particularly helpful in visualising how the weights change during the boosting run and how that change affects classification.

Note that so far we have not specified what the base learners should be. We say that the AdaBoost method is *agnostic* to the learner - you can use many different learning algorithms as base learners. The only formal requirement is that the base learners are **consistently** (with a high probability) achieving performance greater than random guessing. To ensure that you understand weak learner requirement, I encourage you to go through this quiz from lecture. Furthermore, there is a great discussion on StackOverflow that discusses desirable properties for base learners when used in practice.

Deeper insights - the AdaBoost as an example of additive expansion:

In this section we will put boosting in the context of the more general additive model.

The AdaBoost algorithm is a special case of an additive model: we fit a model on elementary *basis* functions. The additive model takes the form:

$$f(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma_m).$$

Here β_m are the expansion coefficients, and b are the basis functions, parametrized by γ_m. We fit the basis functions by picking β_m and γ_m that satisfy certain optimization procedures. The resultant classifier $f(x)$ is formed by combining the base classifiers $b(x)$.

Additive models like this are quite common in machine learning. For example, in a single-layer neural networks the basis functions are sigmoid functions: $b(x; \gamma) = \sigma(\gamma_0 + \gamma_1^T x)$. In this case, the γ parameter determines the linear combinations of the input variables, and β_m multiplies the output of each sigmoidal unit to produce the final output.

In the case of boosting, the basis functions $b(x; \gamma_m)$ are the weak classifiers $G_m(x)$, which are parametrized by the importance weights w_m. We say, "parametrized by w_m" because what makes a weak classifier $G_p(x)$ different from another weak classifier $G_q(x)$ are exactly the importance weights w_p and w_q.

Generally, the way an additive model is fitted is by minimizing some sort of a error (or loss function *L*):

$$min_{\{\beta_m, \gamma_m\}_1^M} [\, L(y_i, f(x_i)) \,] = min_{\{\beta_m, \gamma_m\}_1^M} [\, L(y_i, \sum_{m=1}^{M} \beta_m b(x_i; \gamma_m)) \,]$$

The goal is to determine the parameters $\{\beta_m, \gamma_m\}_1^M$ of the expansion which minimize the loss function *L* - remember that *L* is a function of the target label y_i and the predicted label $f(x_i)$. Once we find those parameters, we have successfully trained a model $f(x_i)$. Note however, that this is a very computationally intensive task, because we would need to find all β parameters as well as γ parameters (one for each basis function *b* for a total of 2 M parameters) all at once.

Forward stagewise additive modeling approximates this minimization process by adding a new basis function b at each iteration of the algorithm. The outline of the algorithm is below:

---

1. Initialize $f_0(x) = 0$
2. For $m = 1 \; to \; M$:
   (a) Compute:

$$(\beta_m, \gamma_m) = arg\, min_{\beta, \gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta\, b(x_i; \gamma_i)).$$

   (b) Set $f_m(x) = f_{m-1}(x_i) + \beta\, b(x_i; \gamma_i)$.
3. Return $f_M(x)$ as the additive expansion $f(x)$

Let's examine the parallel between the more general *forward stagewise additive modeling* and the specific AdaBoost algorithm.

For each iteration of the forward modeling, we find parameters β_m, γ_m that minimize the loss function between the actual target y_i and the best guess of the target we have currently, at the iteration step m: $f_{m-1}(x_i) + \beta\, b(x_i; \gamma_i)$. In the case of boosting, the parameter β_m corresponds to α_m while γ_m determines the weights w_i. At each iteration, there will be a new parameter γ_m, and therefore the importance weights will be updated accordingly.

At each stage, we are fitting a classifier G_m that is parametrized by the importance weights. We are adding G_m (our basis function) to the weighted additive model we have so far $f_{m-1}(x_i)$ in a way that minimizes the error between actual and predicted target label.

We call this forward stagewise modeling because once we have determined the weights β_m, γ_m, we don't go back to basis functions we have already fitted. In other words, once we have computed $f_m(x_i)$, we keep it fixed, and we don't change parameters that we have already computed in the past (for iterations $<= m$). We only have the freedom to compute parameters for future iterations $>= m$. This restriction leads to a computationally efficient algorithm.

This comparison shows that AdaBoost is a special case of the more general forward stagewise modeling and makes it easier for us to find connections between AdaBoost and other supervised learning

algorithms.

Practical considerations:

Let's examine how boosting is applied in data mining.

Classification and regression tasks are important aspect of data mining. Commercial and industrial data often contain features of various types: categorical, binary, free-form (such as natural language). In addition, the number of features is often very large, and it is hard to determine from the start of the analysis which features are important in the classification task.

Consider, for example, the StumbleUpon Evergreen classification challenge on the Kaggle website. The goal is to automate the classification of pages as evergreen or ephemeral. Web pages are classified as *evergreen* if they "maintain a timeless quality" and can be recommended to StumbleUpon users over extended periods of time. Normally this classification is done by humans who review the page contents. For automating this classification task, StumbleUpon provides 26 input features: they include binary, numerical and free-form text types of features!

Decision Trees is one of the popular algorithms for tackling such challenges. Advantages of decision trees include:

- computational scalability
- handling of messy data - missing values, various feature types
- ability to deal with irrelevant features - the algorithm selects "relevant" features first, and generally ignores irrelevant features.
- If the decision tree is short, it is easy for a human to interpret it: decision trees do not produce a black box model.

Decision trees, however, often achieve lower generalization accuracy, compared to other learning methods, such as support vector machines and neural networks. One common way to improve their accuracy is boosting: using decision trees as base classifiers G_m can generate a single boosted learner with high accuracy.

Some of the advantages of decision trees are sacrificed when used in boosting. For example, the AdaBoost algorithm is sensitive to noisy data (mislabeling of the training data). There are several extensions of the

AdaBoost developed to tackle some of these issues. One of these is the gradient boosted model, a generalization of tree boosting. To learn more about gradient boosting, check out the Wikipedia page for an introduction, and Chapter 10, Section 10 from *The Elements of Statistical Learning* for an in-depth discussion.

Advantages and caveats:

If you would like to learn more about boosting from one of the inventors and primary developers, check out this talk from Robert Schapire. The slides are also available on this web page. In particular, in that talk Schapire outlines the *Theory of Margins* that explains why sometimes in practice AdaBoost does not overfit, even with very large numbers of base learners.

Schapire also outlines the advantages and caveats of AdaBoost. Some of the major ones are summarized here:

Advantages:

- Computationally efficient.
- No difficult parameters to set.
- Versatile - a wide range of base learners can be used with AdaBoost.

Caveats:

- Algorithm seems susceptible to uniform noise.
- Weak learner should not be too complex - to avoid overfitting.
- There needs to be enough data so that the weak learning requirement is satisfied - the base learner should perform consistently better than random guessing, with generalization error < 0.5 for binary classification problems.

This listing shows that selecting an appropriate weak learner is one of the key steps towards producing a successful boosting algorithm.