Chapter 7
Naïve Bayes and Decision Trees

In the previous chapter we dove into the workings of the Logistic Regression Model. As you might remember, the Logistic Regression model utilizes the sigmoid function to map out arbitrary input values to output values between 0 and 1. How close an output was to either 0 or 1 gave us a certainty about classifying it, while an output closer to 0.5 meant higher ambiguity. In this chapter we will introduce a another way to explore these probabilities.

7.1 Naïve Bayes

Naïve Bayes is an alternative way to compute a class probability for a given input. The formula for a given input x and its output class y is as follows:

Definition 7.1: Bayes Rule

Bayes Theorem computes the probability of a class using the formula below. If we want to compute the probability that y is a positive review given review x, we multiply the probability of x given that y is positive with the probability that y is positive, and divide it with the probability of x alone.
$P(y = +1|x) = \frac{P(x|y=+1)P(y=+1)}{P(x)}$

We use this equation to compute how likely a review is positive or negative:

$$\frac{P(\text{“The sushi \&everything else was awesome!”} \mid y = +1)\, P(y = +1)}{P(\text{“The sushi \& everything else was awesome!”})}$$

We may discard the divisor, as we are only comparing which class's probability is greater. Now, think about how you would approach this problem. We would need to plug in a probability for each part of the Bayes Theorem. Do we know P("The sushi everything else was awesome!" | y=+1), the probability of this specific review, given that we are told the review is positive? Most likely not, as the sentence "The sushi everything else was awesome!" is unique, and it only appears in one review.

Thus, we make the naïve assumption that every word's probability is independent from each other: Instead of computing this entire sentence's probability, we compute the probability of every word occuring alongside each other. So we compute the probability of "The" given the review is positive AND the probability of "sushi" given the review is positive, and so forth:

P("The sushi everything else was awesome!" | y=+1) = P(The | y=+1) * P(sushi | y=+1) * P(| y=+1) *
P(everything | y=+1) * P(else | y=+1) * P(was | y=+1) * P(awesome | y=+1)

Our final model is thus:

$$P(y|x_1, x_2, x_3, ..., x_d) = \prod_{j=1}^{d} P(x_j|y)P(y)$$

There are a number of issues with this approach. First, since we are multiplying so many probabilities, our final product will be a very long decimal number. Since this decimal number is so long, we might end up with floating point overflow. We can overcome this by taking the log of each probabilities, such that we compute a sum instead of a product. Another issue that we encounter with using products is that if we encounter a word we haven't seen before, its probability will be 0 and so the entire product will be 0. Laplacian Smoothing, adding a constant to each term to avoid multiplying by 0, can be used in this case.

Let us now compare the two models we have learned for computing classification probabilities thus far:

Logistic Regression:
$P(y = +1|x, w) = \frac{1}{1+e^{-w^T h(x_i)}}$
Naïve Bayes:
$P(y = +1|x_1, x_2, x_3, ..., x_d) = \prod_{j=1}^{d} P(x_j|y = +1)P(y = +1)$

While the Logistic Regression model is discriminative, the Naïve Bayes model is generative.

Definition 7.2: Discriminative Model

A model is discriminative when it only cares about finding and optimizing a decision boundary.

Definition 7.3: Generative Model

A model is generative when it defines a distribution for generating x.

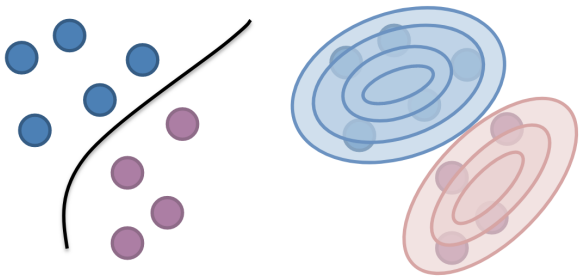

Figure 7.31: On the left visualize a discriminative model, where we only care about optimizing a decision boundary. On the right, we have a generative mode, where the likelihood of each class is also captured.

The Naïve Bayes model is considered a generative model because it computes each input's probability given the output $P(x_j|y = +1)$, so it can be used to map out a probability distribution for each x. This is unlike the Logistic Regression model, that does not compute the probability of each input, but rather solely the probability of the output given the input. Usually a discriminative model like Logistic Regression will do better, but generative models are useful for modeling distributions.

7.2 Decision Trees

The next application of machine learning we will explore will be decision trees. Take a look at the decision tree below:

Definition 7.4: Decision Tree

A decision tree is a series of questions that lead to multiple outcomes such that we can explore each question further.

Decision trees are most applicable for nonlinear decision boundaries that occur along a set of distinct criteria. Let's consider a Loan Application for buying a house as an example. To make an informed decision about a candidate, a loan application may ask about credit history, income, the lease term, and personal information. So, an example dataset might look like this:

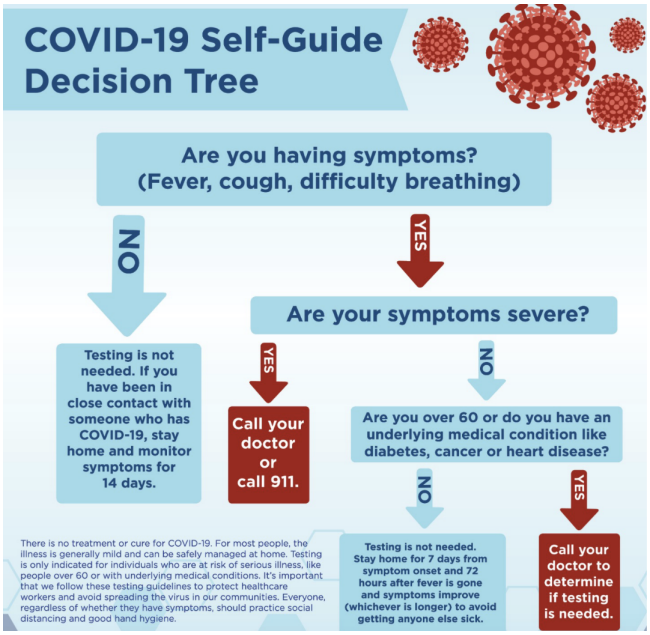

Figure 7.32: A decision tree on COVID-19 safety guidelines based on a series of questions. Source: Holzer

Credit	Term	Income	y
excellent	3 yrs	high	safe
fair	5 yrs	low	risky
fair	3 yrs	high	safe
poor	5 yrs	high	risky
excellent	3 yrs	low	safe
fair	5 yrs	low	safe
poor	3 yrs	high	risky
poor	5 yrs	low	safe
fair	3 yrs	high	safe

Figure 7.33: Let our training dataset be n number of inputs and output pairs. Each input has 3 features: credit, term and income and one output, y.

After training on the train dataset, we aim to reach an accurate prediction $\hat{y}$ of whether a new candidate is safe or risky based on their loan application:

A decision tree for this example can be set up like so:

Definition 7.5: Tree Terminology

A **Tree** in computer science is a datastructure characterized by an interconnected set of nodes, where no connection between nodes creates a cycle.
Node: any datapoint in the tree.

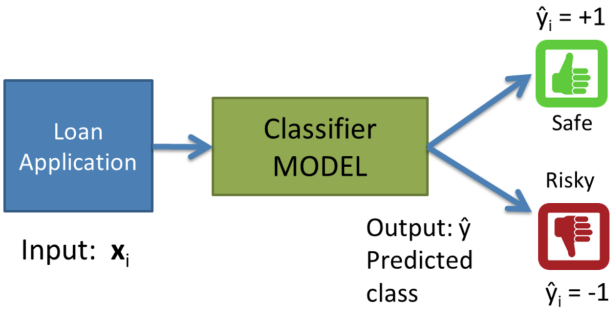

Figure 7.34: A loan application is fed into the model as input x_i. The output will be either $\hat{y}_i = +1$ (safe) or $\hat{y}_i = -1$ (risky).

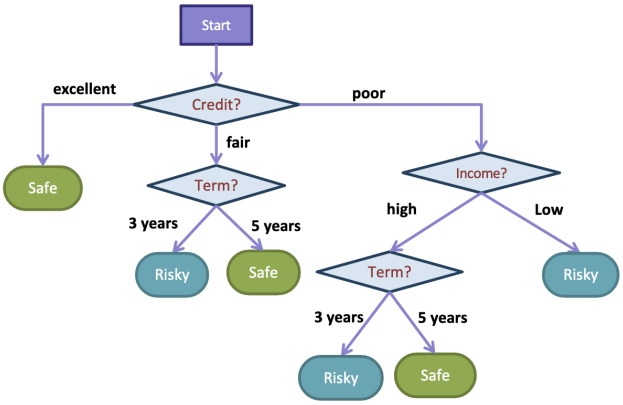

Figure 7.35: A decision tree for determining whether a candidate is safe or risky to lend a loan to. The branch/internal nodes split into possible values of a feature, while the leaf nodes are the final decision, the output class.

Root: the first datapoint in the tree.
Leaf: one of the last datapoints in the tree.

In this example, the model first splits up loan applications based on credit history. If the credit history is excellent, a candidate's application is considered safe. However, if their credit history is poor, the tree splits into a more complex structure as more questions are required to determine if the candidate should receive a loan.

At this point, you may be wondering why we use credit history as the first criteria to judge an applicant, why we split into further questions on certain nodes, and how a lease term can determine an application's safety. These are all choices our model made to optimize its tree such that the tree would output the most probably prediction $\hat{y}$ based on its training on the train data. We will now examine how these choices are made in the building of a decision tree.

7.2.1 Building the Decision Tree

How does a model decide on the optimal decision tree structure? Recall that our training data is what our machine learning model uses to learn on. Given a set of x and y training pairs, we will build a decision tree that aims to lead as many train inputs x to their correct train outputs y as would perform the best in the real world. Since we want to avoid overfitting on the train set, this doesn't necessarily mean that we will build our tree such that every training input x will always lead to its exact correct training output y, so we will also have to decide a good stopping point for growing our tree. The end goal is that when new, unseen test data x is fed into the tree, we can achieve a decently accurate prediction $\hat{y}$.

Let's first start with the very first node in our tree, the root node:

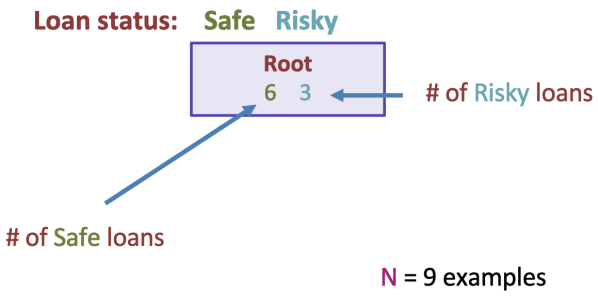

Figure 7.36: The root node of the decision tree stores information about the safe loans and the risky loans. In total, there are 9 loan applications in our dataset.

Since we have just one node, this one node is the root node and it is a leaf node at the same time. In a decision tree, the decisions are made at the leaf nodes. The outputted decision is based on the majority class of the training set inside that node. So if our tree is just a root, then our tree would always output "Safe" since there are 6 safe loans in the training set and 3 risky loans. So, on future unseen test data, we will always output "Safe".

On the train data, this gives us a $6/9 = 66.66\%$ accuracy. If we decide this isn't good enough, we will add a second layer to the tree. This second layer is a **decision stump**

Definition 7.6: Decision Stump

In a Decision Tree, a **stump** is a point at which a node splits into further decision categories.

Suppose then, that we decide to split on a candidate's credit history. Credit history can be either "excellent", "fair" or "poor".

Since we split on credit history, we add a node to our tree for each type of credit history. For each of these nodes, we now store how many of the loans are safe and how many are risky. Recall in our original dataset in Figure 16.82 there are 2 safe loans and 0 risky loans that have excellent credit history, 3 safe loans and 1 risky loan with fair credit history, and 1 safe loans and 2 risky loans with poor credit history.

If we end our tree here, we must finish it off with leaf nodes that have final output classes in them as the final decisions. Remember that these are the majorities of each node.

Now, our training accuracy is $7/9 = 77.77\%$ since 7 out of the 9 loan applications are correctly classified, and 2 are not. What if we were to split on Term first instead of Credit? Let's compare the two choices:

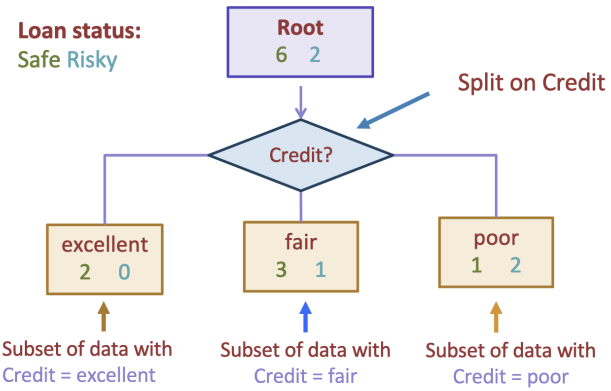

Figure 7.37: The tree splits the training data into different levels of credit history.

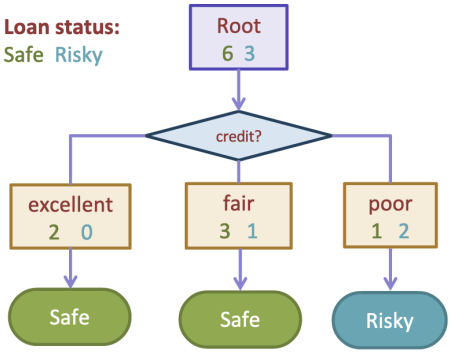

Figure 7.38: The majority class of each node determines the final output.

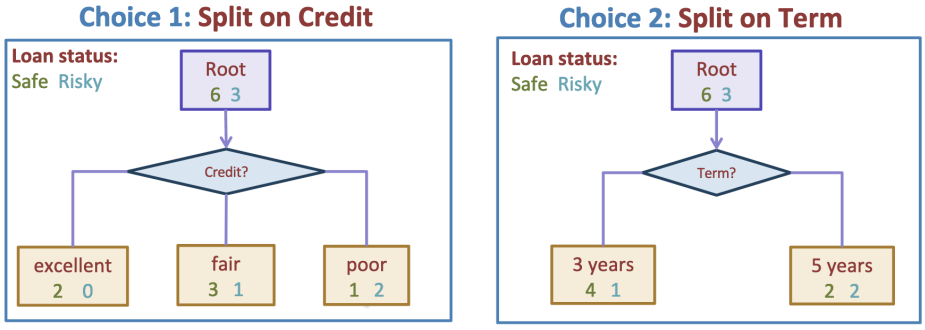

Figure 7.39: A split on Credit versus a split on Term in the decision tree.

In Choice 1 when we split on credit, we have a $7/9 = 77.77\%$ accuracy. Let's calculate the accuracy for Choice 2. For a term of 3 years, we have 4 safe loans and 1 risky loan. Safe loans make up the majority here, so anything with a term of 3 years will be classified as "Safe". For a term of 5 years, there are 2 safe loans and 2 risky loans. Since there is a tie, we can choose either class. Let's designate the output of a 5 year term as a "Risky" loan. Since 4 loan applications for 3 year terms are classified as "Safe" when they actually are

and 2 loan applications for 5 year terms are classified as "Risky" when they actually are, $4 + 2 = 6$ loan applications are correctly classified. $6/9 = 66.66\%$ of loan applications are classified correctly. Likewise, we can also use an error metric here: 33.33% of loan applications are classified incorrectly if we split on the Term feature, and 22.22% of loan applications are classified incorrectly if we split on the Credit feature. Since the Credit feature split yields a lower error rate, our tree will split on Credit first. This decision making is how the model splits a node in a tree and it can be summarized by the following pseudocode:

Split(node)

- Given a subset of data M in node
- For each feature h_i:
  - Compute classification error for a split of M according to feature h_i:
- Chose feature $h^*(x)$ with lowest classification error and expand the tree to include the children of current node after the split

Figure 7.40: The algorithm for selecting the best feature a node should be split on.

Our model will choose between features to split on for every node in the tree, until the number of datapoints in each node or the error rate reaches a certain threshold to stop beuilding the tree. This algorithm can be summarized as:

BuildTree(node)

- If the number of datapoints at the current node or the classification error is within a certain threshold:
  - Stop
- Else:
  - Split(node)
  - For child in node:
    - BuildTree(child)

Figure 7.41: The algorithm for building the tree.

Notice that the Decision Tree algorithm is greedy: It aims to optimize the classification error at each node. As a result, the final result won;t be globally optimal, but it guarantees computational efficiency. Also take note that the Decision Tree algorithm is recursive: From the current node, if we decide to further expand the tree, we repeat the same operations in the child node.

How do we decide when to stop? Going back to Figure 12.71, we observe that for applicants with excellent credit history, all two of them would be safe loans and none of them would be risky. So, the majority class classifier would suffice here, because it would yield no error. But when the Credit is fair or poor, we had some safe loans and some risky loans. As there is no absolute majority for either of these nodes, we could recursively treat them as roots of their own tree which we continue to build below. Thus by further splitting these nodes, we grow a more complex and precise path in our tree with the hopes of lowering the error rate.

We can set a certain error threshold and stop growing our tree until every leaf yields that error or less.

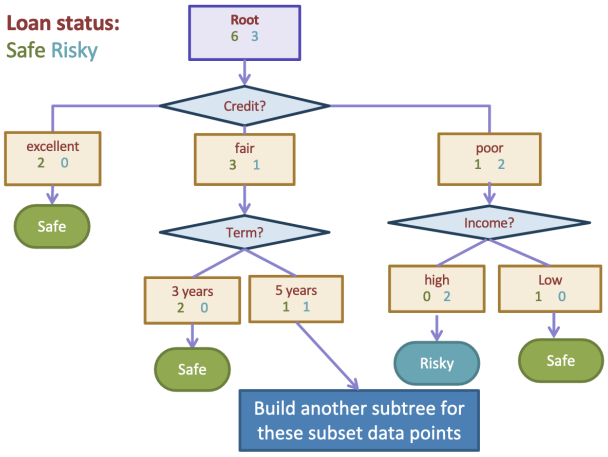

Figure 7.42: Every subtree build from each node will be a subset of the original trainset where the condition of the node is true for the remainder of the tree.

7.2.2 Numeric v.s. Categorical Features

In our Loan Application example, we have been looking at data without numeric value. For example, a candidate's credit history can be of the values "good", "fair", or "poor". There are 3 main datatypes a machine learning algorithm can learn from.

Definition 7.7: Numeric

Numeric data is data that has numerical value, such as square footage of a house.

Definition 7.8: Ordinal

Ordinal data is data that has ordered categories, even if it is not numerical. Credit score is ordinal, because you can rank "good", "fair", and "bad".

Definition 7.9: Nominal

Nominal data is data that has no numeric value and cannot be ordered. Colors such as "red", "blue", "green" cannot be ranked.

Datatypes that are not numeric like ordinal and nominal data are **categorical**.

Some decision trees may or may not require all numerical inputs depending on their implementation. However, in models that use differentiable loss functions (like Linear Regression / Logistic Regression, some forms of decision trees), you need to transform categorical data.

Transforming Ordinal Data: Rank the values (bad = 0, fair = 1, good = 2).
Transforming Nominal Data: Use one-hot encoding.

Definition 7.10: One-Hot Encoding

One-hot encoding is a transformation technique of nominal data. For a feature, it assigns a digit for every possible category that feature can be. For example, for a Color feature, we can assign "red" to the first digit, "blue" to the second digit, and "green" to the third. So, red would be 100, blue would be 010, and green would be 001.

It is important to note that many nominal feature do come in the form of a number, yet they cannot be ordered. An example of this is zip code when estimating house prices. 10018 is a zip code in Manhattan, NY while 98105 is a zip code in Seattle, WA. If we use zip code as a feature of a linear regression model to predict house prices, this does not mean that houses in Seattle are more expensive than those in Manhattan.

7.2.3 Threshold Split for Numeric Features

For numeric features, it doesn't make sense to have individual branches in our tree for every single numerical value that feature could hold. So, we use an inequality instead. One branch will be less than a certain number v, and the other branch can be greater than or equal to. Given a range of numerical values from a single feature, we can choose the best threshold to make this split by calculating the classification error for every possible split that occurs exactly in between each of the possible numerical values $v_1, v_2, ..., v_n$ and select the threshold yielding the lowest error. For more precision, we can split these inequalities further. There is a limitation to using classification error: Two different splits can give us the same error, so continuous loss functions like entropy loss or Gini impurity loss (which we will not talk about in this text) are used.

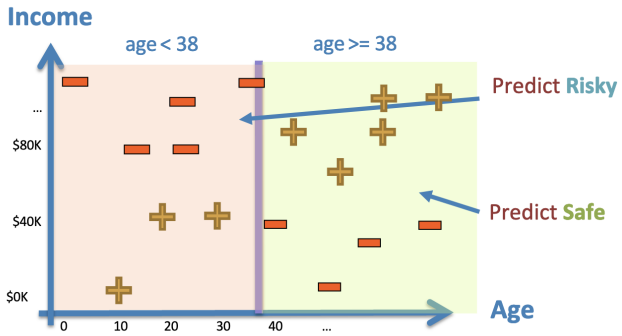

Figure 7.43: For a decision tree, a splitting of a numeric feature will always be along the axis.

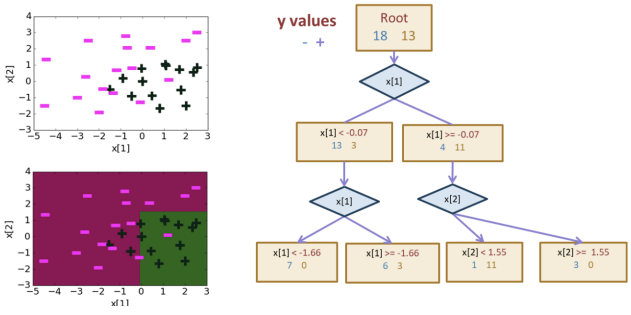

Figure 7.44: If we decide to split further, we continue to split along the axes.

7.2.4 Advantages of Decision Trees

1. Decision Trees are easy to interpret
2. They can handle both continuous and categorical variables without preprocessing
3. They do not require normalization
4. They can create non-linear decision boundaries
5. They can handle missing values

7.2.5 Disadvantages of Decision Trees

1. Deep Decision trees are prone to overfitting. They are not suitable for large datasets for this reason.
2. The decision boundaries are unstable because adding a new datapoint to our trainset can cause the entire tree to regenerate.
3. The decision boundaries must be axis-parallel.

To overcome overfitting, we can implement conditions for early stopping such as establishing a fixed depth length, setting a maximum number of nodes, or halting growth if error does not considerably decrease. We can also **prune** our trees by cutting off some nodes.

Chapter 8
Ensemble Methods

So far in our exploration of Classification, we have covered Logistic Regression, Bayes Theorem, and Decision Trees as candidates for modeling data that is intended to be separated into classes. With the Decision Tree approach, we can model our data in a much more interpretable way, but Decision Trees are not known for reaching stellar accuracy. In this chapter we will discuss the modifications done unto Decision Trees to improve their performance. Specifically, we will discuss the Ensemble Method.

8.1 Ensemble Method

The basic idea behind the Ensemble Method is to combine a collection of models in hopes that their unity will achieve better performance rather than designing a new model from scratch. This collection of models is referred to as a **model ensemble**.

Definition 8.1: Model Ensemble

A **model ensemble** is a collection of (generally weak) models that are combined in such a way to create a more powerful model.

With trees, this is done in via either the **Random Forest (Bagging)** method, or **AdaBoost (Boosting)**.

8.1.1 Random Forest

The Random Forest method, also called Bagging, utilizes a collection of Decision Trees. Each decision tree casts a "vote" for a prediction based on an input and the ensemble predicts the majority vote of all of its trees. Each tree is built from a subset of the training dataset by random sampling the training dataset with replacement. This technique is also called **bootstrapping**.

When training the decision trees on the bootstrapped samples, the goal is to build very deep overfitting trees. Each tree thus has very high variance. As multiple models are used for the final prediction in the ensemble, the overall outcome is a model with low bias. This is because if many overfitting models are averaged out, the result is a model that does not overfit to any one sample so the ensemble has low bias and lower variance. So, while each tree is a robust modeling of a data sample, the united model ensemble is more powerful.

Example(s)

Random Forest has many applications in image processing and object detection. Microsoft used the Random Forest modeling technique in their Kinect system to identify the "pose" of a person from the depth camera.

The advantage of using a Random Forest is that it is versatile, and has many applications in classification, regression, and clustering. It is also low-maintenance and hence easily interpretable: There are few to none hyper-parameters in most cases, and generally more trees gives us better performance. Random Forest is also efficient, because the trees, as separate components, may be trained in parallel. So instead of waiting for a single complex model to train one step at a time as we may do with other models, we can assign each tree of the ensemble to a different machine so that multiple trees are trained simultaneously.

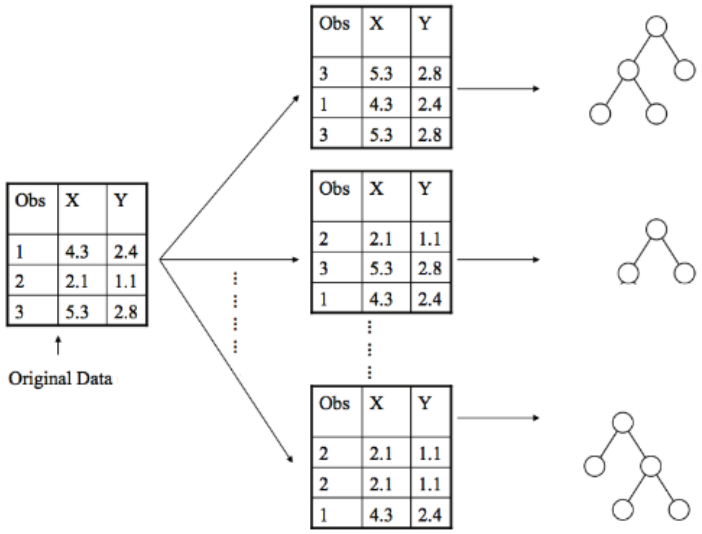

Figure 8.45: A subset of the original dataset is randomly selected with replacement for every tree we intend to build.

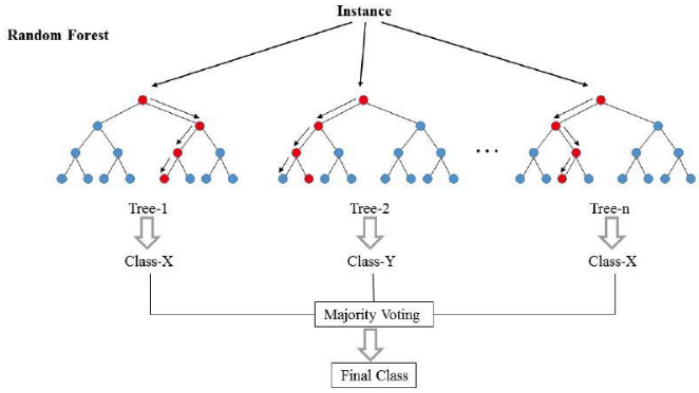

Figure 8.46: Each tree predicts its own final decision based on the input and the majority prediction is selected by the ensemble.

8.1.2 AdaBoost

Recall that a decision stump in a tree is a node that splits into multiple decisions. When we say "stump" we will refer to a tree of one-level. As opposed to growing very deep and overfitting trees in Random Forest, Adaboost utilizes stumps. So, the model ensemble for Adaboost is a collection of one-level trees. Each of these trees are assigned a weight. A weighted majority of votes from these decision stumps is what determines the ensemble's final prediction.

A key difference between Random Forest and AdaBoost is that AdaBoost is sequential, while Random Forest can train each of its trees in parallel. This is because in AdaBoost, each stump belongs to a certain order, and the first stump's decision impacts the second, which impacts the third, and so forth.

Training AdaBoost

When training AdaBoost, the error of the previous stump affects the learning of the next stump. To do this, two types of weights are stored:

1. $\hat{w}_t$: the weight of each stump t.
2. α_i: the weight of each datapoint i.

The training process of AdaBoost is as follows:

1. Decide on our stumps: the more features our data has, the more stumps we will need in AdaBoost since we will have to make at least one decision from each feature. We will call the total set of stumps T, and each stump in T is t.
2. Initialize the weights for all the datapoints in our training set α_i to be equal to each other.
3. For every stump t in T:
   (a) Learn a final prediction $\hat{f}_t(x)$ based on α, the weights of the datapoints used in t.
   (b) Compute the error of t's prediction $\hat{f}_t(x)$. As AdaBoost introduces weights to datapoints, we must compute a **weighted** error:
   $$WeightedError(f_t) = \frac{\sum_{i=1}^{n} \alpha_i || \{\hat{f}_t(x_i) \neq y_i\}}{\sum_{i=1}^{n} \alpha_i} \tag{8.23}$$
   This equation is simply taking the weighted sum of all the misclassified datapoints and then dividing it by the total weight of all the datapoints. In the weighted error, every datapoint's unique weight is used to compute the total error.
   (c) Based on the error rate of prediction $\hat{f}_t(x)$, update model t's weight $\hat{w}_t$. If t yields a low error, then we want $\hat{w}_t$ to be greater, but if the error is high, then the weight should be low. This is because we want more better performing models to have a higher influence in the final decision of the ensemble. The formula for computing the model weight is:
   $$\hat{w}_t = \frac{1}{2} ln(\frac{1 - WeightedError(\hat{f}_t)}{WeightedError(\hat{f}_t)}) \tag{8.24}$$
   (d) Recompute the datapoint weights α. AdaBoost will *increase* the weight of those datapoints it misclassified, and *decrease* the weights of those it classified correctly. This way, misclassified datapoints are more sensitively accounted for in the training of the next stump. This is the notation for updating the weights:
   $$\alpha_i \leftarrow \begin{cases} \alpha_i e^{-\hat{w}_t} & if \hat{f}_t(x_i) = y_i \\ \alpha_i e^{\hat{w}_t} & if \hat{f}_t(x_i) \neq y_i \end{cases}$$
4. After we have computed $\hat{f}_t(x)$ for every stump t, we take the sign of the model outputs' weighted sum like so:
   $$\hat{y} = \hat{F}(x) = sign(\sum_{t=1}^{T} \hat{w}_t \hat{f}_t(x)) \tag{8.25}$$

Real-Valued Features

If we are splitting real-valued numeric features in our AdaBoost method, the algorithm is more or less the same but the splits must happen dependent on the weights of each numerical value.

Normalizing Weights

Generally, the weights for some datapoints can get very large or very small in magnitude due to how the data is laid out. If our numbers are on wildly different scales, we may run in to problems when running AdaBoost on our machine due to the finite precision of computers when it comes to real numbers. To resolve this, we normalize all the weights to sum to 1:

$$\alpha_i \leftarrow \frac{\alpha_i}{\sum_{j=1}^{n} \alpha_j} \tag{8.26}$$

Visualizing AdaBoost

Recall that all the weights for the datapoints are initialized to be equal to each other. After training on one stump, we compute new weights α_i based on the errors of $\hat{f}_1$.

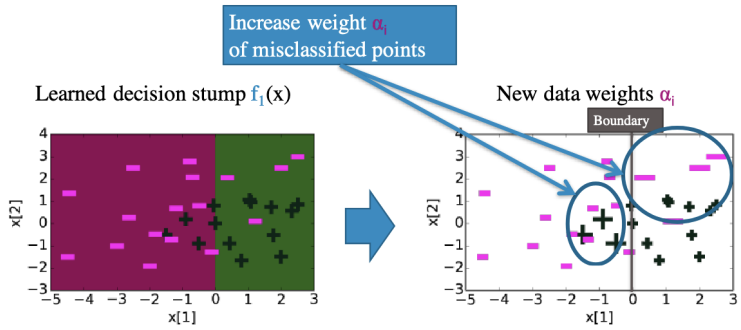

Figure 8.47: After training on the first stump, misclassified datapoints are given a greater weight (drawn larger).

The new weights α_i are used to learn the best stump that minimizes the weighted classification error. Then, update the weights again based on the error from $\hat{f}_2$.

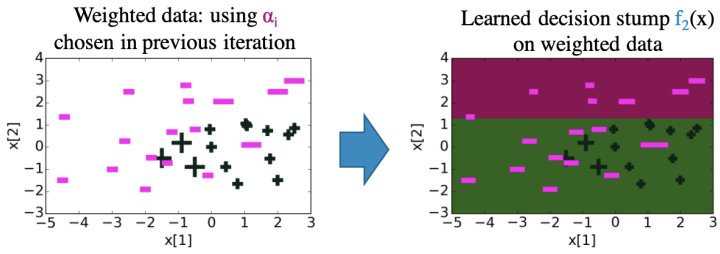

Figure 8.48: $\hat{f}_2$ is predicted based on the weights computed in the previous iteration.

If we plot what the predictions would be for each point, we get something that looks like this:

Eventually, we can achieve a training error of 0 with a large enough set of weak learners.

The example above illustrates that AdaBoost is still capable of overfitting!

AdaBoost Theorem

The core idea behind the Ensemble Method is that a collection of weaker models combined will yield a more powerful model. This is the case with AdaBoost, where a series of weak single-stump trees can gradually

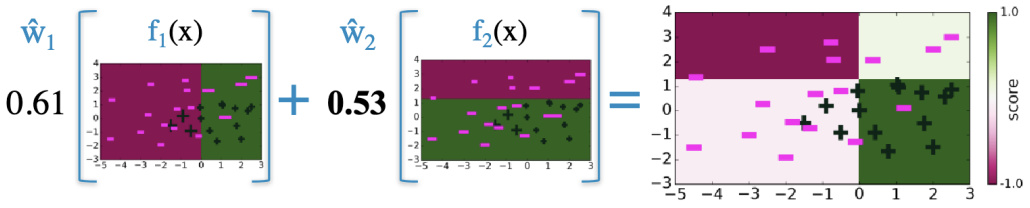

Figure 8.49: The weighted sum of the models $t = 1$ and $t = 2$ yields the weighted decision boundary on the right.

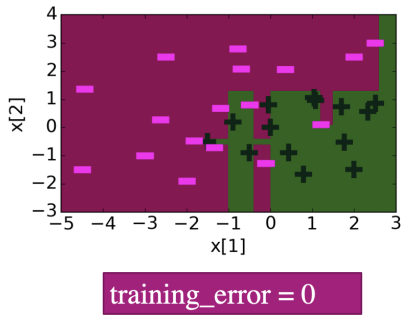

Figure 8.50: Decision boundary for $t = 30$.

reduce the training error of the overall model. Let's compare AdaBoost to a standard Decision Tree.

While a Decision Tree can significantly reduce the error of its training set, we see that this is at the expense of severe overfitting. So a Decision Tree lacks the ability to generalize to unseen test data. However, due to the internal mechanisms of AdaBoost, we avoid this severe overfitting and are more likely to achieve a lower test error. AdaBoost is also capable of overfitting as we have seen in the previous example, but in general the test error will stabilize. To determine the best T to avoid overfitting and underfitting, we must treat T as a hyper-parameter and compare different T's with the intent of minimizing the validation error.

Applications of AdaBoost

Boosting, AdaBoost, and other variants like Gradient Boost are some of the most successful models to date. They are extrmeley useful in computer vision, as they are the standard for face detection. Most industry Machine Learning systems use model ensembles.

To conclude, AdaBoost is a powerful model ensemble technique and it typically does better than Random Forest with the same number of trees. However, while you do not have to tune parameters for AdaBoost (besides selecting T), boosting is sequential, so trees cannot be trained in parallel.

Figure 8.51: The train and test error curves for a standard Decision Tree and AdaBoost.