Classification Week 4 - Overfitting in decision trees

Decision trees are prone to overfitting. Overfitting involves training error going down as the model get's more complex, but having true error that goes down and then eventually goes up with model complexity. So we want to avoid overly-complex models that fit the training data very well, but don't generalize.

As the depth of the tree increases, the complexity of the decision boundary increases. With a deep enough tree, we can get a decision boundary that results in zero training error. So for a decision tree;

For Decision Trees, complexity of fit is proportional to tree depth

Why does training error go down as depth increases?
Well, our split algorithm choose a split based on the one that reduces the training error (in the case, the classification error) the most. So at each step, we are reducing the training error. Eventually, with enough splits, we can separate all classes of data into leaf nodes, such that the error at every leaf node is zero, and so the total training error is zero.

Principle of Occam's razor- Learning simpler decision trees

"Among competing hypotheses, the one with the fewest assumptions should be selected." William of Occam, 13th Century

As applied to Machine Learning, the simplest model with the same error should be chosen. In Decision Trees in particular, given the same validation error, choose a simpler Decision Tree.

There are two approaches to creating simpler trees
- **Stopping Early**: stop learning the tree before it becomes too complex
- **Pruning**: Simplify the tree after the learning algorithm finishes

Early stopping in learning decision trees

Approach 1: Limit Tree Depth - max_depth

One approach is to limit the depth of the tree using a max_depth parameter. So then the question becomes, how do we choose max_depth? We can think of max_depth as a tuning parameter, λ. We can choose the tuning parameter the same way we have in the past, we can test a range of values for the tuning parameter and use either a validation set or cross validation to measure the validation error, so that we can see when validation error increases, so we know when to stop.

The problem with using max_depth as the tuning parameter is that it chops off each branch at the same depth. In reality, some branches may already be short and some would naturally be longer. Never the less, this is a commonly used stopping condition.

Approach 2: Classification Error – error_threshold

Another approach to stopping early is to check the classification error each time we split. The node we are has some classification error. If we can't make a split that reduces the classification error, then we would stop recursing on that node. In practice
- We use a threshold for the classification error. If we can't create a split that reduces the classification error by more than the threshold, then we stop.
- There are some pitfalls to this approach (see the pruning section)

This can be risky but it is a very useful approach in practice.

Approach 3: Minimum Node Size - N$_{min}$

Another approach is to stop recursing when the number of data points in a node is less than a threshold. This prevents us from splitting when the amount of data does not support more fine-grained predictions. Remember that models with a small amount of data are more likely to be overfit (model complexity to data ratio easily becomes high), so this is a localized application of that rule.

This rule should always be implemented. A good rule for N_{min} is $10 <= N_{min} <= 100$.

So our Greedy Decision Tree Building algorithm now looks like;

Recursive greedy algorithm with early stopping
- Step 1: Start with an empty tree with a single node that contains all the data. This is the root node. depth = 0
- Step 2:
    a. Select a feature on which we will split the data (the feature whose split results in the lowest classification error)
    b. create new child nodes with data filtered according to the split feature values.
    c. depth = depth + 1
- For each split node
    a. Step 3: if nothing more to do, don't recurse
        i. if not more features to split on
        ii. if no mistakes (classification error == 0)
        iii. if depth >= max_depth
        iv. if node size >= Nmin
        v. if node classification error – split classification error <= error_threshold
    b. Step 4: otherwise, recursively goto step 2 with the split node and continue splitting data.

(OPTIONAL) Pruning decision trees

OPTIONAL Motivating pruning

**Lesson 6 Basic strategies for handling missing data**
We have previously engineered our data, but we have not yet had to deal with missing data.

Here is an example set of data where the term of the loan is missing for several rows;

h1(x) credit	h2(x) term yrs	h3(x) income	y loan status
excellent	3	high	safe
fair	?	low	risky
fair	3	high	safe
poor	5	high	safe
excellent	3	low	safe
fair	5	low	safe
poor	?	high	risky
poor	5	low	risky
fair	?	high	safe

So our training data could have missing values. But we may also have missing data when we go to make a prediction. For instance, what do we do if the data used to predict the output (safe or risky) value is missing the term of the loan?

Given this tree;
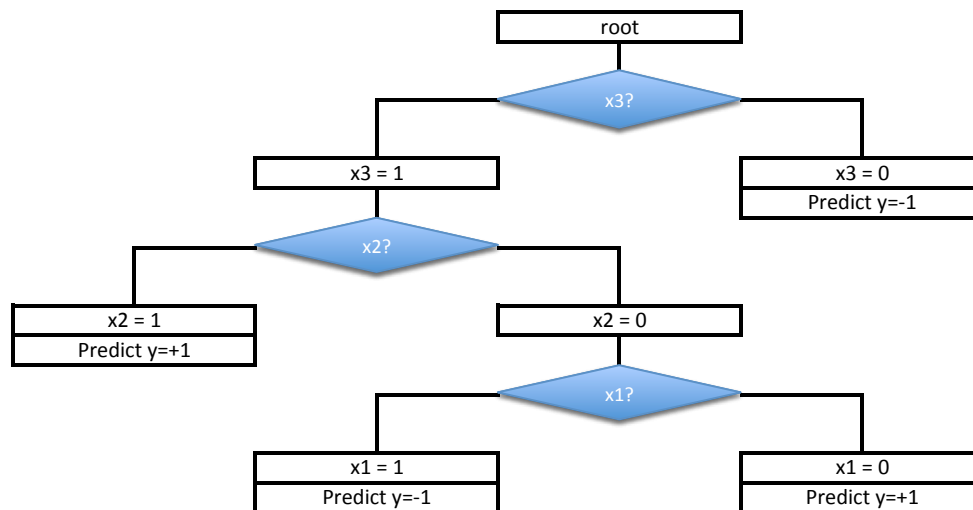
How would we predict y give this data:

x1	x2	x3	y
1	?	1	?

More precisely, what would we do when we reach the x2? decision?

Strategy 1- Purification by skipping missing data
There are two notions for skipping data. First, we can skip any rows that have missing data. For instance, if we skipped the rows that are missing the term values in the table above, we would go from 9 data points to 6 data points;

h1(x) credit	h2(x) term yrs	h3(x) income	y loan status
excellent	3	high	safe
fair	?	low	risky
fair	3	high	safe
poor	5	high	safe
excellent	3	low	safe
fair	5	low	safe
poor	?	high	risky
poor	5	low	risky
fair	?	high	safe

h1(x) credit	h2(x) term yrs	h3(x) income	y loan status
excellent	3	high	safe
fair	3	high	safe
poor	5	high	safe
excellent	3	low	safe
fair	5	low	safe
poor	5	low	risky

Skip input rows with missing data

With enough data, that could be fine. However, if we lose too much data by dropping rows (because a lot of term values are missing, for instance) then we can consider the second skipping option; drop features that have a high percentage of missing data;

h1(x) credit	h2(x) term yrs	h3(x) income	y loan status
excellent	3	high	safe
fair	?	low	risky
fair	3	high	safe
poor	5	high	safe
excellent	3	low	safe
fair	5	low	safe
poor	?	high	risky
poor	5	low	risky
fair	?	high	safe

h1(x) credit	h3(x) income	y loan status
excellent	high	safe
fair	low	risky
fair	high	safe
poor	high	safe
excellent	low	safe
fair	low	safe
poor	high	risky
poor	low	risky
fair	high	safe

Skip features with missing data

Skipping Data has advantages
- It is easy to implement and understandable
- It can be applied to any kind of model, not just decision trees

Skipping Data has disadvantages
- Removing data or features my remove important information
- It is sometimes not clear if data rows or feature columns should be removed.
- Skipping input rows with missing data does not address the issue of data that may be missing at prediction time.
- Skipping input rows may create systematic error – if data is missing because of differing data gathering in different geographical regions, then dropping data may result in losing all data from a given region.

Skipping data is used fairly commonly because of the advantages, but it is not the best way to handle missing data.

Strategy 2- Purification by imputing missing data
In this case, instead of skipping data, we fill in the missing values with some guess or estimate. For example, we might use the majority class in the feature for all the missing values in the feature column.

h1(x)	h2(x)	h3(x)	y		h1(x)	h2(x)	h3(x)	y
credit	term yrs	income	loan status		credit	term yrs	income	loan status
excellent	3	high	safe		excellent	3	high	safe
fair	?	low	risky		fair	**3**	low	risky
fair	3	high	safe		fair	3	high	safe
poor	5	high	safe		poor	5	high	safe
excellent	3	low	safe		excellent	3	low	safe
fair	5	low	safe		fair	5	low	safe
poor	?	high	risky		poor	**3**	high	risky
poor	5	low	risky		poor	5	low	risky
fair	?	high	safe		fair	**3**	high	safe

Impute value based on simple majority class

Simple rules for imputing missing values are;
1. For missing **categorical data, use the mode** (the most common value) in the feature column to fill in the missing data.
2. For missing **numerical features, use the median or the mean** of the values in the feature column to fill in the missing values.

There are other more advanced ways of imputing data, such as the Expectation Maximization (EM) algorithm.

Imputing data has advantages
- It is easy to implement and understandable
- It can be applied to any kind of model, not just decision trees
- It can be used at prediction time; we simply use the same rule that we used to impute missing training data to missing prediction data.

Imputing data has disadvantages
- It can result in severe systematic errors.
  For instance, if we impute a feature value in national data, but we did not realize that feature is always missing for a certain state or region due to collection differences, then we would bias all predictions in the state or region.

Strategy 3 - Modify learning algorithm to explicitly handle missing data
In the case of decision trees, we can modify the algorithm so there is a missing value choice at every decision node; at each decision node we add "or unknown" to one of the possible branch conditions. We do this for each decision node. This means that the final prediction is not always the same for inputs with unknown data because different decision points will handle the missing data differently. This requires us to change the learning algorithm so that we add the "or unknown" conditions to the branches in such a way as to minimize the classification error.

This makes making predictions that include unknown values as easy as predictions with full data.

Explicitly handling missing data in the learning algorithm has advantages
- Addresses missing data at training and prediction time
- Because the algorithm minimized classification error associated with missing values, we get more accurate predictions

Explicitly handling missing data in the learning algorithm has disadvantages
- Required modification of the learning algorithm
  - In the case of decision trees, this is pretty simple.
  - It may be more difficult for other kinds of models.

Feature split selection with missing data

The way we handle learning missing data for decision trees is to modify the how we learn each decision stump. Specifically, we modify the function that tests for the best feature to split on. Remember that the feature split with the lowest classification error is chosen. In our examples from prior weeks, there were no missing values so we calculated the classification error for the a given feature split in one step. Now, if the feature has unknown values, we will need to figure out which branch of the split should get the unknown values. The algorithm will try associating the unknown values for the feature with each branch of the split, then choose the one with the lowest classification error as the proposed feature split. Then, as usual, we choose the actual feature to split on from all the proposed feature splits based on the lowest classification error.

So simply
- Missing values get added to one branch of the split
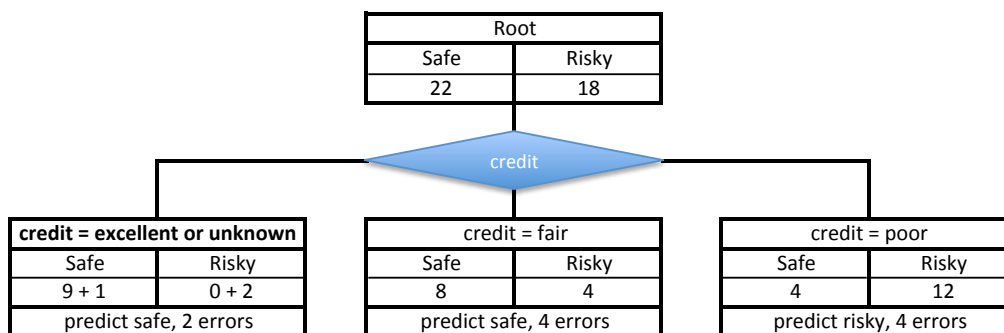- Classification error is used to choose which branch

For example, if we have this loan application data with N=40 data points, of which 3 of the credit rating values are unknown;

h1(x) credit	h2(x) term yrs	h3(x) income	y loan status
excellent	3	high	safe
?	5	low	risky
fair	3	high	safe
poor	5	high	safe
excellent	3	low	safe
?	5	low	safe
?	3	high	risky
...	...	...	...
fair	3	high	safe

N = 40 with 3 missing credit values

When we split on credit, we must decide which split branch (credit = excellent, credit = fair, or credit = poor) will include rows with unknown credit values. We can see in the table above that 1 of the unknown credit rating values are associated with a safe output value and 2 of the unknown credit rating values are associated the risky output values, so those output values will be added to the split branch when we calculate the classification error.

When we associated the unknown credit values with excellent credit branch, it looks like this;

Root	
Safe	Risky
22	18

credit

credit = excellent or unknown	
Safe	Risky
9 + 1	0 + 2
predict safe, 2 errors	

credit = fair	
Safe	Risky
8	4
predict safe, 4 errors	

credit = poor	
Safe	Risky
4	12
predict risky, 4 errors	

Classification Error = (2 + 4 + 4) / 40 = 0.25

The excellent credit branch now includes 1 row with a loan status of safe and 2 rows with loan status of risky. Those then become part of the calculation of classification error for the potential split; # mistakes in split / total data points.

We repeat this for the other two credit split branches;

Root	
Safe	Risky
22	18

credit

credit = excellent	
Safe	Risky
9	0
predict safe, 0 errors	

credit = fair or unknown	
Safe	Risky
8 + 1	4 + 2
predict safe, 6 errors	

credit = poor	
Safe	Risky
4	12
predict risky, 4 errors	

Classification Error = (0 + 6 + 4) / 40 = 0.25

Root	
Safe	Risky
22	18

credit

credit = excellent	
Safe	Risky
9	0
predict safe, 0 errors	

credit = fair	
Safe	Risky
8	4
predict safe, 4 errors	

credit = poor or unknown	
Safe	Risky
4 + 1	12 + 2
predict risky, 5 errors	

Classification Error = (0 + 4 + 5) / 40 = 0.225

We can see that if we associate the unknown values with the credit = poor branch, then we produce the credit split with the lowest classification error. That then becomes the credit split that we propose when we compare all possible feature splits.

This example was the first split, but we would use this same logic at every split; we use this logic when learning every decision stump. When considering a feature split, if there are missing values for that feature, we go through the process of finding the feature value to which we should associate the unknown values.

This example was for a categorical feature, but the same thing can be done with a numerical feature. Instead of associating unknown values with a category, we associate them with threshold splits (see week 3 for discussion of threshold splits). Note then that this would need to be taken into account deciding on the threshold split itself.

Here is the general algorithm;

Feature Splitting with Unknown Values
At each node with a subset of data M;
- For each feature $h_j(x)$, determine a proposed split
  - Split the data of M according to the feature $h_j(x)$ where the value $h_j(x)$ is not "unknown".
    - If there are no "unknown" values for the feature $h_j(x)$, then this is the proposed split; calculate the classification error for the split
    - If there are "unknown" values for feature $h_j(x)$
      - For each branch of the split
        - assign data points with "unknown" value for $h_j(x)$
        - calculate the classification error for the unknown split
      - Choose the unknown split with the lowest classification error as the proposed feature split
- Choose the proposed feature split $h_*(x)$ with the lowest classification error

What constitutes an unknown value?
You really need to look at the data to decide what is an unknown value. For a numerical feature, a blank is clearly unknown. However, sometimes when data is input or through some processing step, a zero or a -1 is used where data is unknown. For text data, blank may be an unknown value, but in some cases the empty string may be a legal value – you really need to understand the data before making that choice.