Recap: Decision Trees

Decision Trees — Prediction

Root node
Decision node
Leaf node

$x_1 < 1.3$

- $x_2 < 0.2$
- $x_2 < 0.3$

- $x_1 < -1.3$
  - $t$
  - $f$

- $t$
  - $f$

- $t$
  - $f$

- $t$
  - $f$
  - $t$
  - $f$
  - $t$
  - $f$

Scatter plot with decision boundaries.
Recap: Decision Trees

Decision Trees — Prediction

Root node

Decision node

$x_1 < 1.3$

Decision node

$x_2 < 0.2$

$x_1 < -1.3$

Decision node

$x_2 < 0.3$

Leaf node
Recap: Decision Trees

Root node

Decision node

Leaf node

\[ x_1 < 1.3 \]

\[ x_2 < 0.2 \]

\[ x_1 < -1.3 \]

\[ x_2 < 0.3 \]
Recap: Decision Trees

Root node

Decision node

Leaf node

\[ x_1 < 1.3 \]

\[ x_2 < 0.2 \]

\[ x_2 < 0.3 \]

\[ x_1 < -1.3 \]
Recap: Decision Trees

- Decision Trees

- Prediction

- Root node

- Decision node

- Leaf node

- $x_1 < 1.3$

- $x_2 < 0.2$

- $x_2 < 0.3$

- $x_1 < -1.3$
Recap: Decision Trees

- **Root node**
  - $x_1 < 1.3$

- **Decision node**
  - $x_2 < 0.2$
  - $x_2 < 0.3$

- **Leaf node**
  - $x_1 < -1.3$

The diagram illustrates a decision tree with nodes and arrows indicating the decision criteria for each level. The tree splits data points into regions based on the conditions specified in the nodes.
Recap: Decision Trees

Decision Trees — Prediction

-2 0 2 4
-1
0
1
2

Root node

Decision node

Leaf node

\( x_1 < 1.3 \)

\( x_2 < 0.2 \)

\( x_2 < 0.3 \)

\( x_1 < -1.3 \)

\( x_1 \)

\( x_2 \)
Recap: Regularization for Decision Trees

Trees can overfit if they are too large so we usually look to regularize or prune them.

![Accuracy vs Size of Tree](chart.png)

### Pruning
- Collapsing down some nodes to a single leaf node
- Start at the leaves and step upwards deciding whether to collapse based on some metric
- Smaller tree that is less prone to overfitting
- Computational expensive and unreliable
Ensembles: Bootstrapping, Bagging, and Random Forests
Individual Trees are Not Good Predictive Models
Learn Multiple Trees and Average Their Predictions Instead

\[ p(Y|X) = \frac{1}{L} \sum_{t=1}^{L} p_t(Y|X) \]
Different Trees will Make Different Errors: Averaging Can Cancel Errors Out
Decision ensembles (also known as forests) are fundamentally more powerful models than individual trees:

- They can have more complex decision boundaries.
- They reduce the variation on predictions: we can average away the errors as not all trees will make the same “mistakes.”
- They are less prone to overfitting.
- No need to prune.

Figure: Decision surface for a random forest.
Forests have a huge number of applications

Image Classification

Motion Detection

Processing Satellite Images

[Image Credits: Bosch et al and Gram-Hansen et al]
Microsoft Kinect Pose Recognition

https://youtu.be/lntbRsi81U8?t=41
Forests are extremely easy to use

Number of trees (bigger the better)

```
RF = TreeBagger(200,XTrain,YTrain); % Training
preds = predict(RF,XTest); % Predict
```

- Applicable to wide range of problems and data types
- Extremely fast
Random forests and their extensions give spectacular out-of-the-box performance: 7 out of the top 20 classifiers from a survey of 180 classifiers on 82 datasets are based on them.


Ensembling is much more general than just the concept of decision tree ensembles. We can take almost any machine learning model and convert it into an ensemble (e.g. we might also construct an ensemble of neural networks). The general idea is always the same: train multiple instances of the model and then combine their predictions to a single prediction (usually by averaging). To ensure that we do not just end up with the same predictions as a single model, we need to introduce some form of randomness into the training process. To construct a good ensemble we need to ensure we get a good balance between the diversity of the constituent models (i.e. how correlated the errors they make are) and their individual predictive power. Ensembling almost always improves the performance compared to a single model at the expense of increased cost. Machine learning “competitions” (e.g. Kaggle) are usually won by some sort of ensemble method, often an ensemble of neural network.
Key Questions

- How should we introduce randomness into the training process?
  - Though this can vary depending on the base model, our focus will be on a universally applicable method called bagging and some additional methods that are specific to decision trees.

- How can we control the diversity vs predictive power trade-off?
  - This will effectively boil down to controlling how much randomness we instill.
Bootstrapping
Model Variability

- To have an effective ensemble, our individual models need to make different errors in their predictions: we need diversity.
- Some algorithms have instability in their training such that small changes in data can lead to large differences in the learned model.
- Decision trees and neural networks are unstable algorithms, k-nearest neighbors are stable.
- **Bootstrapping** allows us to exploit instabilities to produce a diverse set of models by training each on a slightly different dataset.
**Bootstrap Sampling**

- Samples a new dataset from the old dataset by **resampling with replacement**
- Each new sample is drawn independently from the full old dataset
Bootstrap Sampling

- Samples a new dataset from the old dataset by **resampling with replacement**
- Each new sample is drawn independently from the full old dataset
Bootstrap Sampling

❖ Samples a new dataset from the old dataset by **resampling with replacement**

❖ Each new sample is drawn independently from the full old dataset
Bootstrap Sampling

- Samples a new dataset from the old dataset by **resampling with replacement**
- Each new sample is drawn independently from the full old dataset
Bootstrap Sampling

❖ Samples a new dataset from the old dataset by resampling with replacement
❖ Each new sample is drawn independently from the full old dataset
Bootstrap Sampling

- Samples a new dataset from the old dataset by **resampling with replacement**
- Each new sample is drawn independently from the full old dataset
Bootstrap Sampling

- Samples a new dataset from the old dataset by **resampling with replacement**
- Each new sample is drawn independently from the full old dataset
Bootstrap Sampling

- Samples a new dataset from the old dataset by **resampling with replacement**
- Each new sample is drawn independently from the full old dataset
Bootstrap Sampling

- Samples a new dataset from the old dataset by **resampling with replacement**
- Each new sample is drawn independently from the full old dataset
Bootstrap Sampling

❖ Samples a new dataset from the old dataset by **resampling with replacement**
❖ Each new sample is drawn independently from the full old dataset
Bootstrap for Decision Trees

- The **bootstrap** is itself a way to assess the variance of estimators.
- Fit multiple trees, each on a **bootstrapped sample**.

```r
> n <- nrow(Pima.tr)
> bss <- sample(1:n, n, replace=TRUE)
> sort(bss)
[1]  2  4  4  5  6  7  9 10 11 12 12 12 12 13 13 15 15 15 15 20 ...

> tree_boot <- rpart(Pima.tr[bss,8] ~ ., data=Pima.tr[bss,-8],
control=rpart.control(xval=10)) ## 10-fold CV
```
Example: Bootstrap for Regression Trees

- Regression for Boston housing data.
- Predict median house prices based only on crime rate.
- Use decision stump—the simplest tree with a single split at root (note this for the purposes of the example on not good practice in general)
Example: Bootstrap for Regression Trees

- We are aiming to fit a predictor \( \hat{f}(x) \) and have data \( \{(x_i, y_i)\}_{i=1}^n \).
- Assess the variance of \( \hat{f}(x) \) by taking \( B = 20 \) bootstrap samples of the original data, and obtaining bootstrap estimators
  \[
  \hat{f}^b(x), \quad b = 1, \ldots, B
  \]
- Each tree \( \hat{f}^b \) is fitted on the resampled data \( (x_{j_i}, y_{j_i})_{i=1}^n \) where each \( j_i \) is chosen randomly from \( \{1, \ldots, n\} \) with replacement.
Bagging
Creating Diversity: Bagging

Original Dataset

Dataset 1

Dataset 2

Dataset L

Train Tree 1

Train Tree 2

Train Tree L

Ensembles: Bootstrapping, Bagging, and Random Forests

Bagging
Average Predictions over Trees

\[
p(Y|X) = \frac{1}{L} \sum_{t=1}^{L} p_t(Y|X)
\]
Bagging

**Bagging (Bootstrap Aggregation):** average across all $B$ predictive models (e.g. trees) fitted on different bootstrap samples.

1. For $b = 1, \ldots, B$:
   1. Draw indices $(j_1, \ldots, j_n)$ from the set $\{1, \ldots, n\}$ with replacement.
   2. Fit the model, and form predictor $\hat{f}^b(x)$ based on bootstrap sample $(x_{j_1}, y_{j_1}), \ldots, (x_{j_n}, y_{j_n})$

2. Form bagged estimator

$$\hat{f}_{Bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^b(x)$$
Bagging smooths out the drop in the estimate of median house prices.

In general, bagging reduces the variance of predictions.
Variance Reduction Through Aggregation

Consider an ideal world where we can draw a complete new dataset at will without having to perform bootstrapping, such that we can construct estimators $\hat{f}_m$ each based on different independent datasets of size $n$ from the true joint distribution of $X, Y$.

The aggregated estimator would then be

$$\hat{f}_{ag}(x) = \frac{1}{B} \sum_{m=1}^{B} \hat{f}_m(x) \rightarrow \bar{f}(x) = \mathbb{E}_D[\hat{f}(x)] \quad \text{as } B \rightarrow \infty$$

where expectation is with respect to independent datasets of size $n$.

Using the standard bias-variance decomposition and the law of large numbers, the squared-loss to the optimal classifier $f_*$ for input $x$ is

$$\mathbb{E}_D[(f_*(x) - \hat{f}_{ag}(x))^2] = (f_*(x) - \bar{f}(x))^2 + \mathbb{E}_D[(\bar{f}(x) - \hat{f}_{ag}(x))^2]$$

$$= (f_*(x) - \bar{f}(x))^2 + \frac{1}{B} \mathbb{E}_D[(\bar{f}(x) - \hat{f}(x))^2]$$

$$\rightarrow (f_*(x) - \bar{f}(x))^2 \quad \text{as } B \rightarrow \infty.$$

Aggregation reduces the squared loss by reducing the variance of $\hat{f}(x)$. 
Variance Reduction Through Bagging

- For bagging, our setup is similar, but our estimators are now based on the bagging estimators $\hat{f}^b$ trained on resampled versions of the dataset.

We thus have

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{m=1}^{B} \hat{f}^b(x) \to \tilde{f}(x, D) \triangleq \mathbb{E}_{j_1, \ldots, j_n} [\hat{f}^1(x) | D] \quad \text{as } B \to \infty$$

where the expectation is now over the resampling indices.

- We thus have (not examinable):

$$\mathbb{E}[(f_*(x) - \hat{f}_{bag}(x))^2] = \mathbb{E}_D \left[ \mathbb{E} \left[ (f_*(x) - \hat{f}_{bag}(x))^2 \big| D \right] \right]$$

$$= \mathbb{E}_D \left[ \mathbb{E} \left[ (f_*(x) - \tilde{f}(x, D))^2 \big| D \right] \right] + \mathbb{E}_D \left[ \mathbb{E} \left[ (\tilde{f}(x, D) - \hat{f}_{bag}(x))^2 \big| D \right] \right]$$

$$= \mathbb{E}_D \left[ (f_*(x) - \tilde{f}(x, D))^2 \right] + \frac{1}{B} \mathbb{E} \left[ (\tilde{f}(x, D) - \hat{f}^1(x))^2 \right]$$

$$\to \mathbb{E}_D \left[ (f_*(x) - \tilde{f}(x, D))^2 \right] \quad \text{as } B \to \infty.$$
In general, the variance of $\tilde{f}(x, \mathcal{D})$ will be less than that of $\hat{f}(x)$, i.e. the base model trained on the full dataset. It is still non-zero as $\tilde{f}(x, \mathcal{D})$ still varies with the dataset.

Similarly the variance of the bagging estimator $\hat{f}_{bag}(x)$ is generally less than that of $\hat{f}(X)$:

$$\mathbb{E} \left[ (\hat{f}_{bag}(x) - \mathbb{E} [\hat{f}_{bag}(x)])^2 \right] \lesssim \mathbb{E} \left[ (\hat{f}(x) - \mathbb{E} [\hat{f}(x)])^2 \right]$$

This variance reduction comes at the cost of a small increase in bias, in general:

$$\left( \mathbb{E} [\hat{f}(x)] - f_*(x) \right)^2 \leq \left( \mathbb{E} [\hat{f}_{bag}(x)] - f_*(x) \right)^2$$

Bagging is most useful for flexible estimators with high variance and low bias.
Variance Reduction in Bagging

- Deeper trees have higher complexity and variance, but lower bias.
- Compare bagging trees of depth 1 and 3.
Out-of-bag Test Error Estimation

- How well does bagging to? Can we estimate generalization performance, and tune hyperparameters?
- Sledgehammer approach: cross-validation.

For each $v = 1, \ldots, V$,
- fit $\hat{f}_{Bag}$ on the training samples.
- predict on validation set.
- Compute the CV error by averaging the loss across all test observations.

<table>
<thead>
<tr>
<th>$i=1$</th>
<th>$i=2$</th>
<th>$i=3$</th>
<th>$i=4$</th>
<th>$i=5$</th>
<th>$i=6$</th>
<th>$i=7$</th>
<th>$i=8$</th>
<th>$i=9$</th>
<th>$i=10$</th>
<th>$i=11$</th>
<th>$i=12$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v=1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v=2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v=3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v=4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Out-of-bag Test Error Estimation

But to fit $\hat{f}_{Bag}$ on the training set for each $v = 1, \ldots, V$, we have to train on $B$ bootstrap samples!

<table>
<thead>
<tr>
<th>$i=1$</th>
<th>$i=2$</th>
<th>$i=3$</th>
<th>$i=4$</th>
<th>$i=5$</th>
<th>$i=6$</th>
<th>$i=7$</th>
<th>$i=8$</th>
<th>$i=9$</th>
<th>$i=10$</th>
<th>$i=11$</th>
<th>$i=12$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v=1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b=1$</td>
<td>$b=2$</td>
<td>$b=3$</td>
<td>$b=4$</td>
<td>$b=5$</td>
<td>$b=1$</td>
<td>$b=2$</td>
<td>$b=3$</td>
<td>$b=4$</td>
<td>$b=5$</td>
<td>$b=5$</td>
<td>$b=5$</td>
</tr>
</tbody>
</table>

| $v=2$ |       |       |       |       |       |       |       |       |        |        |        |
|       |       |       |       |       |       |       |       |       |        |        |        |
|       |       |       |       |       |       |       |       |       |        |        |        |
| $b=1$ | $b=2$ | $b=3$ | $b=4$ | $b=5$ | $b=1$ | $b=2$ | $b=3$ | $b=4$ | $b=5$  | $b=5$  | $b=5$  |

| $v=3$ |       |       |       |       |       |       |       |       |        |        |        |
|       |       |       |       |       |       |       |       |       |        |        |        |
|       |       |       |       |       |       |       |       |       |        |        |        |
| $b=1$ | $b=2$ | $b=3$ | $b=4$ | $b=5$ | $b=1$ | $b=2$ | $b=3$ | $b=4$ | $b=5$  | $b=5$  | $b=5$  |

| $v=4$ |       |       |       |       |       |       |       |       |        |        |        |
|       |       |       |       |       |       |       |       |       |        |        |        |
|       |       |       |       |       |       |       |       |       |        |        |        |
| $b=1$ | $b=2$ | $b=3$ | $b=4$ | $b=5$ | $b=1$ | $b=2$ | $b=3$ | $b=4$ | $b=5$  | $b=5$  | $b=5$  |

Out-of-bag Test Error Estimation

Idea: test on the “unused” data points in each bootstrap iteration to estimate the test error.

\[
\hat{f}_{ooob}(x_1) = \frac{1}{4} \sum_{b \in \{3,4,8,10\}} \hat{f}^b(x_1)
\]
Out-of-bag Test Error Estimation

Idea: test on the “unused” data points in each bootstrap iteration to estimate the test error.

\[
\hat{f}^{\text{oob}}(x_2) = \frac{1}{3} \sum_{b \in \{2,8,10\}} \hat{f}^b(x_2)
\]
Out-of-bag Test Error Estimation

- For each $i = 1, \ldots, n$, the out-of-bag sample is:

  $$
  \tilde{B}_i = \{b : x_i \text{ is not in training set}\} \subseteq \{1, \ldots, B\}.
  $$

- Construct the out-of-bag estimate at $x_i$:

  $$
  \hat{f}^{\text{ooob}}(x_i) = \frac{1}{|\tilde{B}_i|} \sum_{b \in \tilde{B}_i} \hat{f}^b(i_i)
  $$

- Out-of-bag risk:

  $$
  R^{\text{ooob}} = \frac{1}{n} \sum_{i=1}^{n} L(y_i, \hat{f}^{\text{ooob}}(x_i))
  $$
Ensembles: Bootstrapping, Bagging, and Random Forests

Bagging

Out-of-bag Test Error Estimation

- We need $|\tilde{B}_i|$ to be reasonably large for all $i = 1, \ldots, n$.
- The probability $\pi^{\text{oob}}$ of an observation NOT being included in a bootstrap sample $(j_1, \ldots, j_n)$ (and hence being ‘out-of-bag’) is:

$$\pi^{\text{oob}} = \prod_{i=1}^{n} \left(1 - \frac{1}{n}\right) \quad \text{as} \quad n \to \infty \quad \frac{1}{e} \approx 0.367.$$ 

- Hence $E[|\tilde{B}_i|] \approx 0.367B$
- In practice, number of bootstrap samples $B$ is typically between 100 and 1000, meaning that the number $|\tilde{B}_i|$ of out-of-bag samples will be approximately in the range $35 - 400$.
- The obtained test error estimate is asymptotically unbiased for large number $B$ of bootstrap samples and large sample size $n$.
- For finite $B$ it will generally overestimate the test error as it is effectively using a smaller ensemble.
Example: Boston Housing Dataset

- Apply out of bag test error estimation to select optimal tree depth and assess performance of bagged trees for Boston Housing data.
- Use the entire dataset with \( p = 13 \) predictor variables.

```r
n <- nrow(BostonHousing)  # n samples
X <- BostonHousing[, -14]
Y <- BostonHousing[, 14]
B <- 100
maxdepth <- 3
prediction_oob <- rep(0, length(Y))  # vector with oob predictions
numbertrees_oob <- rep(0, length(Y))  # number pf oob trees
for (b in 1:B) {  # loop over bootstrap samples
  subsample <- sample(1:n, n, replace=TRUE)  # "in-bag" samples
  outofbag <- (1:n)[-subsample]  # "out-of-bag" samples
  treeboot <- rpart(Y ~ ., data=X, subset=subsample, control=rpart.control(maxdepth=maxdepth, minsplit=2))  # fit tree on "in-bag" samples
  prediction_oob[outofbag] <- prediction_oob[outofbag] + predict(treeboot, newdata=X[outofbag,])
  numbertrees_oob[outofbag] <- numbertrees_oob[outofbag] + 1
}
# final oob-prediction is average across all "out-of-bag" trees
prediction_oob <- prediction_oob / numbertrees_oob
```
Example: Boston Housing Dataset

```r
plot(prediction_oob, Y, xlab="PREDICTED", ylab="ACTUAL")
```

For depth $d = 1$.  

For depth $d = 10$. 
Example: Boston Housing Dataset

- Test error / out-of-bag error as a function of tree depth $d$:

<table>
<thead>
<tr>
<th>tree depth $d$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>10</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>single tree $\hat{f}$</td>
<td>60.7</td>
<td>44.8</td>
<td>32.8</td>
<td>31.2</td>
<td>27.7</td>
<td>26.5</td>
<td>27.3</td>
</tr>
<tr>
<td>bagged trees $\hat{f}_{Bag}$</td>
<td>43.4</td>
<td>27.0</td>
<td>22.8</td>
<td>21.5</td>
<td>20.7</td>
<td>20.1</td>
<td>20.1</td>
</tr>
</tbody>
</table>

- Without bagging, the optimal tree depth seems to be $d = 10$.
- With bagging, we could also take the depth up to $d = 30$. 
Bagging: Summary

- Bagging reduces variance and helps prevent overfitting
  - In general, we do not need to prune when making forests and can often get away without including any early stopping criteria
- Almost always improves performance in practice (for non–trivial data) compared to using a single tree
  - Because they can have more complicated decision boundaries, bagged trees are generally fundamentally more powerful models
- However, bagged trees lose much of the interpretability of single trees
- Bagging is a general strategy: decision trees are the most common base model, but can also be applied to other things like neural networks.
For some base models, bagging is not best way of creating an ensemble

- Some models already have natural instabilities that can be exploited without needing to take subsets of the data, e.g. the instability in neural network training with respect to the initialization of the weights and biases
- In some scenarios, the gains from variance reduction are outweighed by the losses from increased bias due to using subsets of the data

In other scenarios, bagging does not give enough diversity and we can get further gains by introducing additional randomization into the training process in addition to (or instead of) bagging

Another common way of introducing diversity is to randomize the way features are accessed by the classifier, such as by manipulating the form of the features (e.g. through using random rotations) or restricting access to certain features during the model training (e.g. using random subspacing)
Random Forests
In general, bagging trees creates insufficient diversity: it is usually beneficial to further randomize the training process.

Random Forests (RFs) are an extension of bagged trees that introduce an additional randomization: random subspacing (we’ll cover this next).

- Implemented in `randomForest` library in R.

This tends to give a better trade off between diversity and predictive power of individual trees.

RFs and their extensions remain some of the best performing models in machine learning, particularly in the context of “out-of-the-box” use because they typically do not require careful setting of any hyperparameters.
Random forests and their extensions give spectacular out-of-the-box performance: 7 out of the top 20 classifiers from a survey of 180 classifiers on 82 datasets are based on them.

Random Subspacing

- At each node randomly select a random subset of features to use for the split search.
- The number of features selected, $m_{\text{try}}$, is a hyperparameter.
- Typically only select a small number of features (e.g. $m_{\text{try}} = \log_2 p$ or $m_{\text{try}} = p^{0.5}$).
- This means different trees will split along different dimensions at each node.
- Difference at earlier nodes lead to different data partitions at later nodes and thus further diversity.
Random Subspacing

• At each node randomly select a random subset of features to use for the split search
• The number of features selected, \( m_{\text{try}} \), is a hyperparameter
• Typically only select a small number of features (e.g., \( m_{\text{try}} = \log_2 p \) or \( m_{\text{try}} = p^{0.5} \))
• This means different trees will split along different dimensions at each node
• Difference at earlier nodes lead to different data partitions at later nodes and thus further diversity
Random Subspacing

- At each node randomly select a random subset of features to use for the split search
- The number of features selected, $m_{\text{try}}$ is a hyperparameter
- Typically only select a small number of features (e.g. $m_{\text{try}} = \log_2 p$ or $m_{\text{try}} = p^{0.5}$)
- This means different trees will split along different dimensions at each node
- Difference at earlier nodes, lead to different data partitions at later nodes and thus further diversity
Random Subspacing

- At each node randomly select a random subset of features to use for the split search
- The number of features selected, $m_{\text{try}}$ is a hyperparameter
- Typically only select a small number of features (e.g. $m_{\text{try}} = \log_2 p$ or $m_{\text{try}} = p^{0.5}$)
- This means different trees will split along different dimensions at each node
- Difference at earlier nodes, lead to different data partitions at later nodes and thus further diversity
Random Subspacing

- At each node randomly select a random subset of features to use for the split search.
- The number of features selected, $m_{\text{try}}$ is a hyperparameter.
- Typically only select a small number of features (e.g. $m_{\text{try}} = \log_2 p$ or $m_{\text{try}} = p^{0.5}$).
- This means different trees will split along different dimensions at each node.
- Difference at earlier nodes lead to different data partitions at later nodes and thus further diversity.
Random Forests = Bagging + Random Subspacing
1. Use bagging: generate a separate bootstrap dataset to train each individual tree on
2. Train each tree by recursively greedily choosing the best split (as per the split criterion) from set of randomly selected features until a stopping criterion is reached

\[ \text{InfoGain} = 0.95 \]
3. Make predictions by aggregating the predictions made by individual trees (note bootstrapping is not used for evaluation)
Looking at the Boston Housing data again

library(randomForest)
library(MASS)
data(Boston)

y <- Boston[,14]
x <- Boston[,1:13]

?randomForest
Random Forests

> rf <- randomForest(x,y)
> print(rf)
>
Call:
randomForest(x = x, y = y)
Type of random forest:  regression
Number of trees:  500
No. of variables tried at each split:  4

Mean of squared residuals:  10.26161
% Var explained:  87.84

Can plot the predicted values (out-of-bag estimation) vs. true values by

> plot( predict(rf), y)
> abline(c(0,1),col=2)

Same if treating the training data as new data

> plot( predict(rf,newdata=x), y)
Out-of-bag error.

\texttt{> plot(predict(rf), y)}
\texttt{> abline(c(0,1), col=2)}

Training error.

\texttt{> plot(predict(rf, newdata=x), y)}
\texttt{> abline(c(0,1), col=2)}
Try `mtry` 2 (i.e. search over two features at each node)

```r
> (rf <- randomForest(x, y, mtry=2))
Call:
  randomForest(x = x, y = y, mtry = 2)
Type of random forest:  regression
Number of trees:  500
No. of variables tried at each split:  2

Mean of squared residuals:  12.17176
```

Try `mtry` 4

```r
> (rf <- randomForest(x, y, mtry=4))
Call:
  randomForest(x = x, y = y, mtry = 4)
Type of random forest:  regression
Number of trees:  500
No. of variables tried at each split:  4

Mean of squared residuals:  10.01574
```
And \texttt{mtry} 8 and 10.

\begin{verbatim}
> (rf <- randomForest(x,y,mtry=8))
Call:
randomForest(x = x, y = y, mtry = 8)
Type of random forest: regression
Number of trees: 500
No. of variables tried at each split: 8

Mean of squared residuals: 9.552806

> > (rf <- randomForest(x,y,mtry=10))
Call:
randomForest(x = x, y = y, mtry = 10)
Type of random forest: regression
Number of trees: 500
No. of variables tried at each split: 10

Mean of squared residuals: 9.774435
\end{verbatim}

\textbf{mtry} the main tuning parameter and typically performance is not sensitive to its choice (can use \texttt{tuneRF} to select it automatically)
Hyperparameters — Number of Trees

- More the merrier — limited by computational budget, having too many does not cause Bagging to overfit
- 500 trees is a typical choice
- More trees gives a smoother surface

Computational Cost

- Training in $O(n(\log(n))^2 B m_{\text{try}})$ for $n \times p$ data with $B$ trees
- Prediction in $O(\log(n) B)$
- These are not a typos, their costs are actually independent of $p$!
  - We may wish to increase $m_{\text{try}}$ (and potentially also $B$) as $p$ increases we can indirectly increase cost with larger $p$
  - Typically though, we set $m_{\text{try}}$ to be a sublinear function of $p$, with $m_{\text{try}} = \lfloor \log_2 p \rfloor$ and $m_{\text{try}} = \lfloor \sqrt{p} \rfloor$ being common choices
- RFs are thus cheaper to train than bagged trees, substantially so for large $p$
- For large $p$, training a RF can even be cheaper than training a single decision tree, particularly if we need to use pruning for the latter
Tree ensembles have better performance, but decision trees are more interpretable.

How to interpret a forest of trees?

Idea: denote by $\hat{e}$ the out-of-bag estimate of the loss when using the original data samples. For each variable $k \in \{1, \ldots, p\}$,

- permute randomly the $k$-th predictor variable to generate a new set of samples $(\tilde{X}_1, Y_1), \ldots, (\tilde{X}_n, Y_n)$, i.e., $\tilde{X}_{\tau(i)}^{(k)} = X_{\tau(i)}^{(k)}$, for a permutation $\tau$.
- compute the out-of-bag estimate $\hat{e}_k$ of the prediction error with these new samples.

A measure of importance of variable $k$ is then $\hat{e}_k - \hat{e}$, the increase in error rate due to a random permutation of the $k$-th variable.
Example for Boston Housing data.

```r
rf <- randomForest(x, y, importance=TRUE)
varImpPlot(rf)
```
Random Forests Pros and Cons

+ Very easy to use and applicable to a wide range of problems
+ Fast at train and test time
+ State-of-the-art for many applications
+ Particularly good for small to medium datasets
+ Work very well “out-of-the-box”, i.e. without tuning

- Typically worse than deep neural nets for huge datasets
- Little flexibility to tune to specific problems
- Good performance can required data preprocessing using some sort of feature extractor
Extensions

- Various further extensions to random forests exist.
- Most look to increase diversity even more by adding further randomness to the training process.
  - **Extremely Randomized Trees** forms an ensemble where the tree structures are predominantly or even complete random.
  - Other methods split along *randomized hyperplanes* rather than using individual features to create further diversity.

References:

Hyperplane Splits

(a) Single CART
(b) RF with 200 Trees

(c) Single CCT
(d) CCF with 200 Trees

Rainforth and Wood (2014)
Useful Packages / Further Reading

- TreeBagger (Matlab) — [https://uk.mathworks.com/help/stats/treebagger.html](https://uk.mathworks.com/help/stats/treebagger.html)
- C++ — [https://github.com/bjoern-andres/random-forest](https://github.com/bjoern-andres/random-forest)
- R — [https://www.tutorialspoint.com/r/r_random_forest.htm](https://www.tutorialspoint.com/r/r_random_forest.htm)
- Weka (stand alone) — [http://www.cs.waikato.ac.nz/ml/weka/](http://www.cs.waikato.ac.nz/ml/weka/) - Gui with java back end, operates on csv files and allows lots of algorithms to be used at once