Prediction Trees

CART, Bagging, and Random Forest

EN5422/EV4238 | Fall 2023

w07_Prediction_Trees.pdf

(Week 7 – 2/2)

Contents

1 Bagging Trees

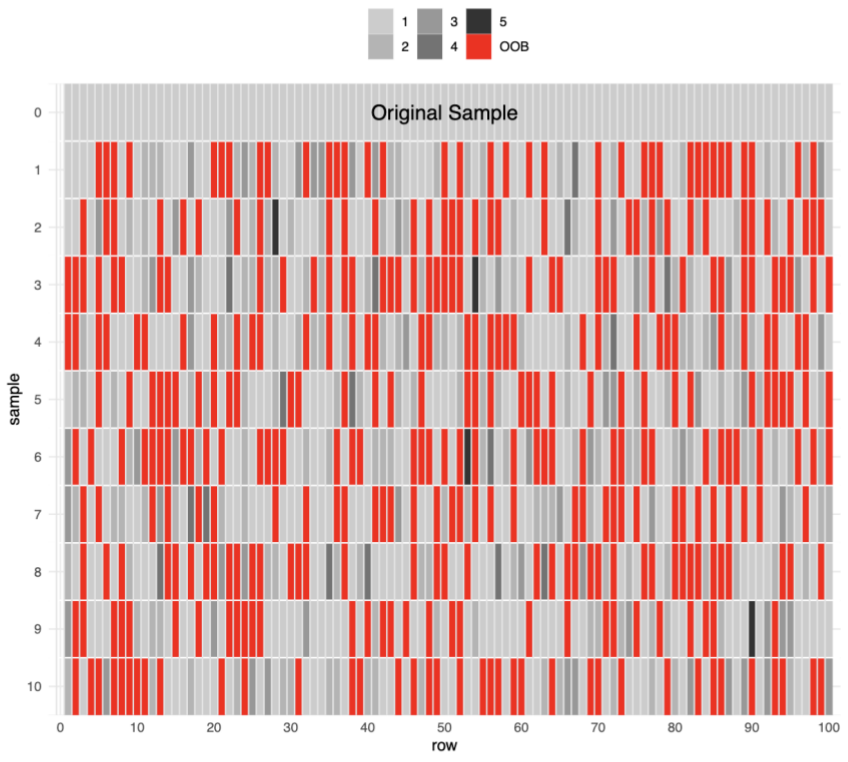
1.1 Better Trees

- Because of the instability of trees, they are great candidates for methods like bagging that will reduce the variance.
- Grow a set of B trees from a bootstrap samples and average their predictions:

$$\hat{f}(x) = \frac{1}{B}\sum_{b=1}^{B} T(x; \hat{\theta}_b)$$

where $\hat{\theta}_b = \hat{\theta}(\mathcal{D}_b)$ are all the estimated parameters for fitting the tree (split variables, cut points, and terminal (leaf) node values) from the bootstrap sample $\mathcal{D}_b$.

- Details and discussion can be found in Breiman's article "Bagging Predictors" (1996, Machine Learning).
  - Bagging = **B**ootstrap **Agg**regat**ing**
  - Lots of advice on when Bagging will help and when it won't.
  - Bagging will help with variance reduction, but not bias.
- Bagging produces an *ensemble model*
- Aggregation of Bagged Predictors:
  - For regression: use the average predictions.
  - For classification: use the average of the predicted class probabilities (majority vote is possible too, but be careful about class imbalance or unequal misclassification costs)



1.1.1 Variance Reduction with Bagging

Note

A helpful probability cheatsheet can be found here:
https://github.com/wzchen/probability_cheatsheet/blob/master/probability_cheatsheet.pdf

Properties of Variance/Covariance

$$Var(X) = E(X^2) - \left(E(X)\right)^2 = Cov(X,X)$$
$$Cov(X_1,X_2) = E(X_1X_2) - E(X_1)(X_2)$$
$$Cor(X_1,X_2) = \frac{Cov(X_1,X_2)}{\sqrt{Var(X_1)V(X_2)}}$$
$$Var(X_1 + X_2) = Var(X_1) + Var(X_2) + 2Cov(X_1,X_2)$$

$$Var\left(a\sum_{i=1}^{p} X_i\right) = a^2\sum_{i=1}^{p} V(X_i) + 2a^2\sum_{i<j} Cov(X_i,X_j)$$

Variance Reduction

- Let θ be something we want to estimate (e.g., $\theta = f(x)$) and $\hat{\theta}$ is an estimate.
- Suppose we have M models to estimate θ which produces the estimates $\{\hat{\theta}_1,\hat{\theta}_2,\dots,\hat{\theta}_M\}$.
- One way to make an *ensemble prediction* is from the average.

$$\bar{\theta} = \frac{1}{M}\sum_{i=1}^{M}\hat{\theta}_i$$

- The **expected value** of the ensemble is:

$$\mathrm{E}[\bar{\theta}] = \frac{1}{M}\sum_{i=1}^{M}\mathrm{E}[\hat{\theta}_i]$$

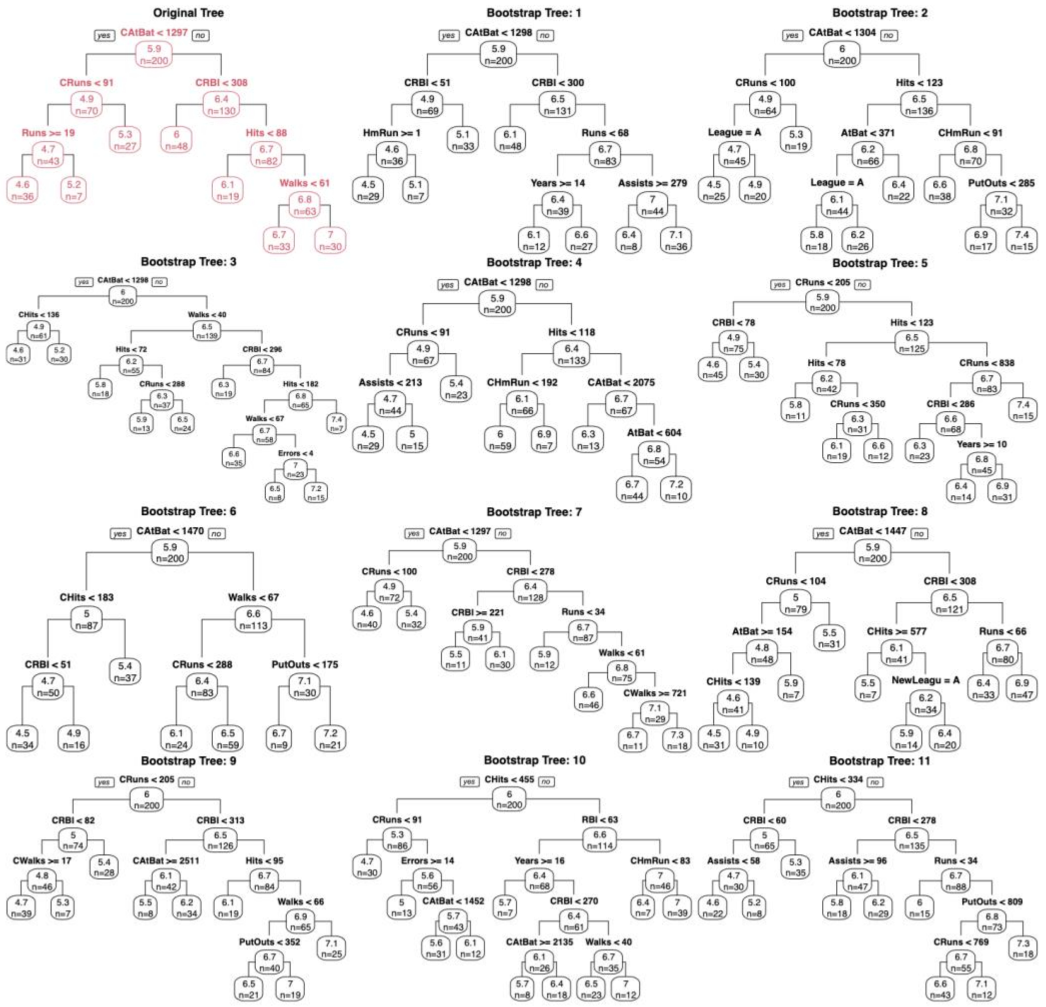
- The variance of the ensemble is:

$$Var(\bar{\theta}) = \frac{1}{M^2}\sum_{i=1}^{M} Var(\hat{\theta}_i) + \frac{2}{M^2}\sum_{i<j}\sqrt{Var(\hat{\theta}_i)Var(\hat{\theta}_i)}Cor(\hat{\theta}_i,\hat{\theta}_j)$$

- Thus, to reduce the variance, we want to use models that have low correlation
  - If $Cor(\hat{\theta}_i,\hat{\theta}_j) = 0 \;\; \forall i,j$, then variance is minimized (for example, when the models are *independent*)

- If $Cor(\hat{\theta}_i, \hat{\theta}_j) = 1 \ \forall i, j$, then there is no (variance reduction) benefit of using an ensemble.
- In Bagging, each *model* is a tree fit with a bootstrap sample.
- Four unstable models, like trees, the bagged models will have low correlation, but for more stable models, like linear regression, the bagged models will maintain high correlation.

1.2 Bagging Trees



(ESL pg 587) "The essential idea in bagging is to average many noisy but approximately unbiased models, and hence reduce the variance."

- Thus when Bagging trees, grow deep trees to reduce bias and use many bootstrap samples to reduce variance.

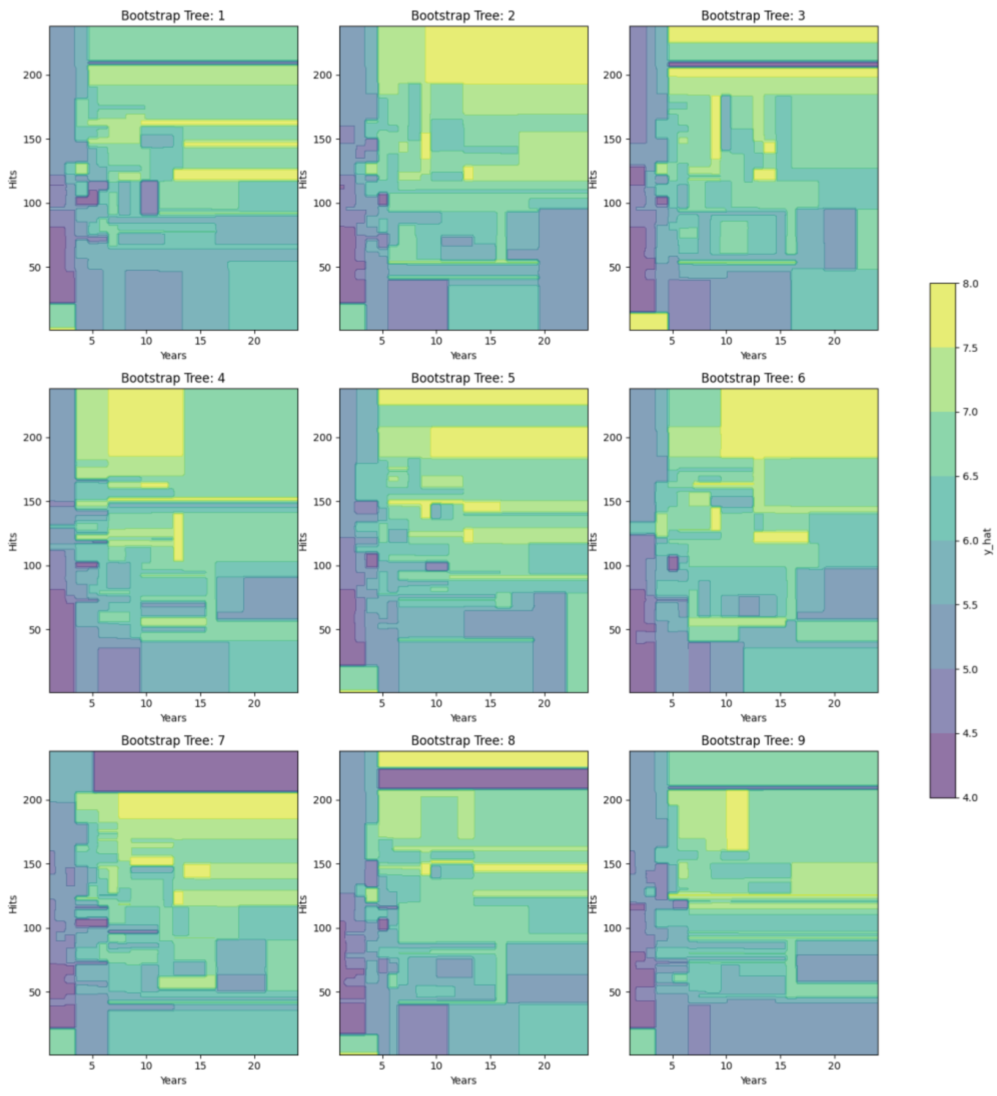


Figure 1: Bagging in 2 dimensions (Years and Hits)

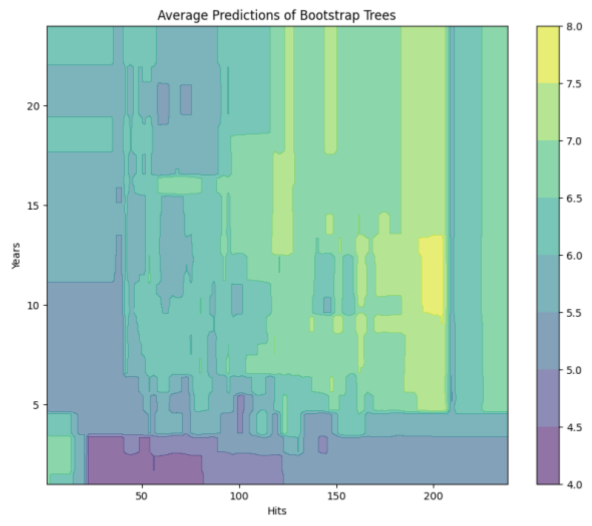


Figure 2: Average of bootstrap predictions

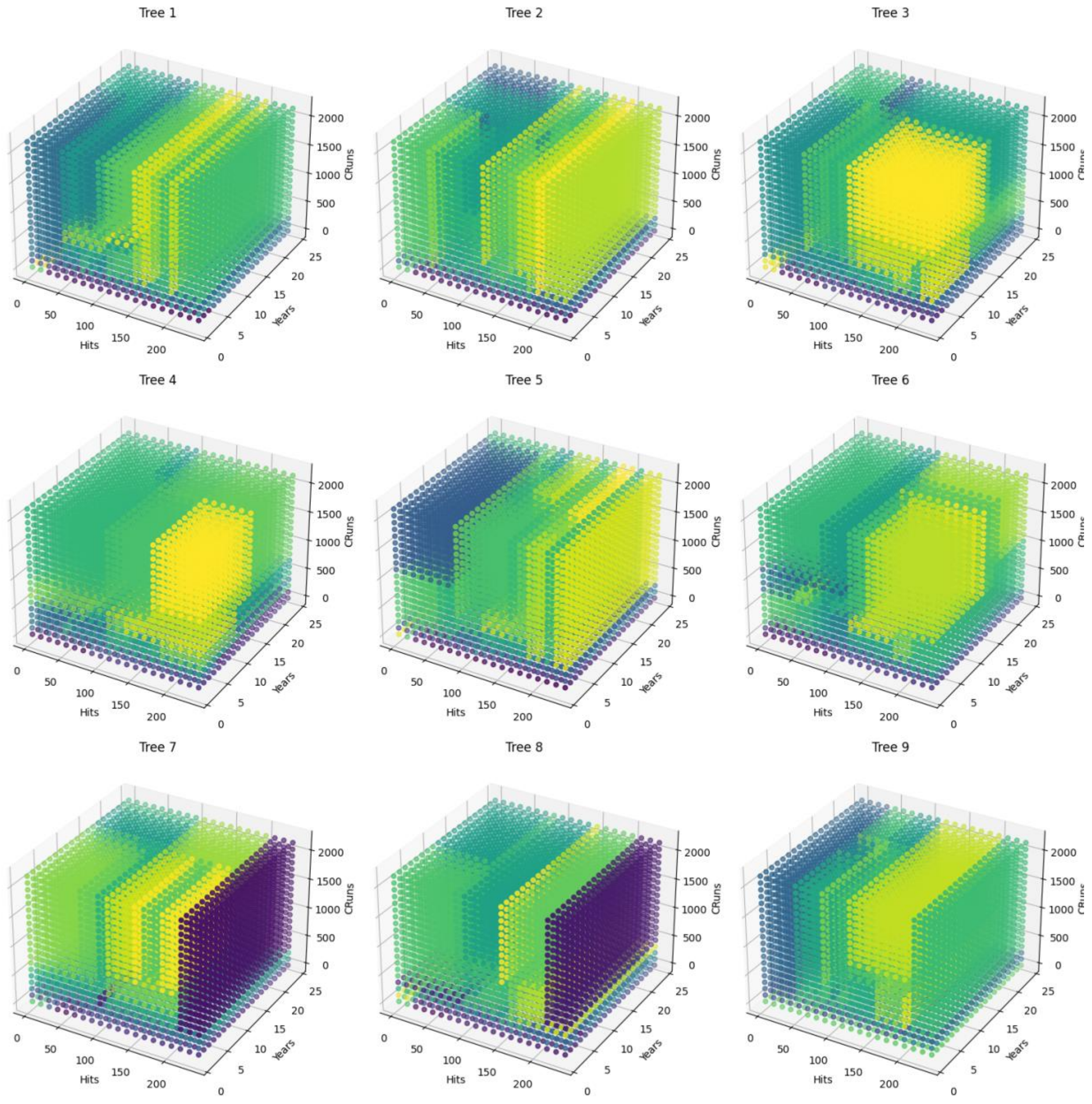


Figure 3: Bagging in 3 dimensions (Years, Hits, and CRuns)

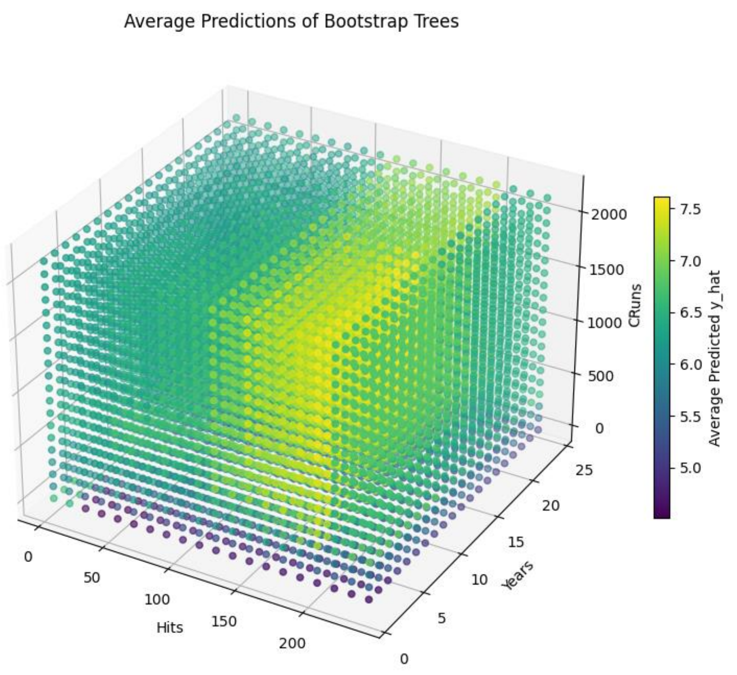


Figure 4: Average of bootstrap predictions (3 dimensions)

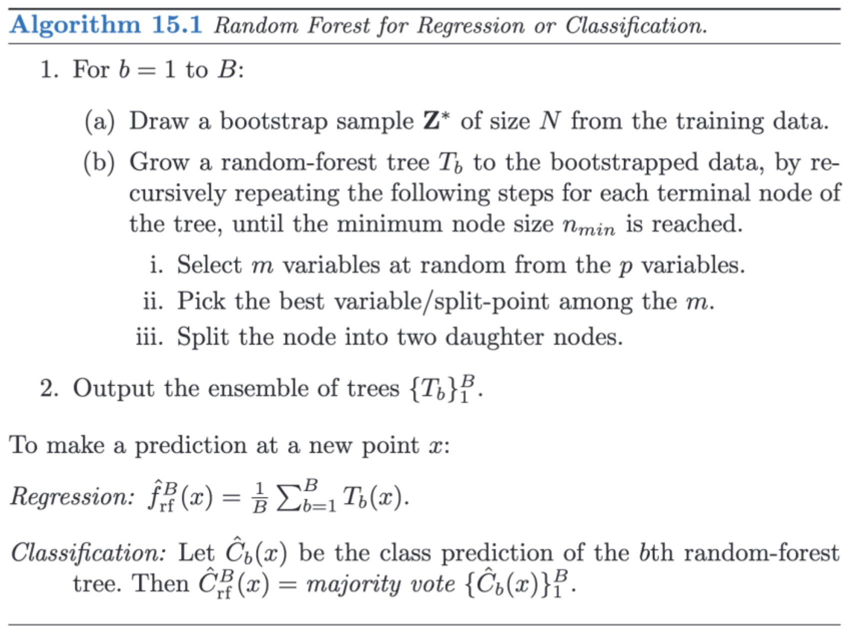
Code Example: Bagging Trees

```
from sklearn.ensemble import BaggingRegressor
from sklearn.tree import DecisionTreeRegressor
import matplotlib.pyplot as plt
import numpy as np

# Select only "Hits" and "Years" from the dataset
X_subset = hitters_data[['Years', 'Hits']]

# Fit the bagging regressor on the subset of features
bagging_regressor = BaggingRegressor(base_estimator=DecisionTreeRegressor(),
n_estimators=12, random_state=42)
bagging_regressor.fit(X_subset, y)

# Extract individual trees from the bagged model
trees = bagging_regressor.estimators_

# Prepare the grid for "Years" and "Hits"
years_range = np.linspace(X_subset['Years'].min(), X_subset['Years'].max(),
100)
hits_range = np.linspace(X_subset['Hits'].min(), X_subset['Hits'].max(),
100)
years_grid, hits_grid = np.meshgrid(years_range, hits_range)

# Set up the plots
fig, axes = plt.subplots(3, 3, figsize=(15, 15))  # Create a 3x3 subplot
structure

# Plot predictions from each bootstrap tree
for i, ax in enumerate(axes.flatten()):
    if i < len(trees):  # Check if the tree exists
        # Predict y_hat using the ith tree
        y_hat = trees[i].predict(np.c_[years_grid.ravel(),
hits_grid.ravel()])
        y_hat_grid = y_hat.reshape(years_grid.shape)

        # Create the contour plot on the current subplot axis
        contour = ax.contourf(years_grid, hits_grid, y_hat_grid, alpha=0.6,
cmap='viridis')
        ax.set_title(f'Bootstrap Tree: {i + 1}')
        ax.set_xlabel('Years')
        ax.set_ylabel('Hits')

# Adjust layout to prevent overlap
fig.tight_layout()

# Add a color bar to the right of the subplots with modifications
cbar = fig.colorbar(contour, ax=axes.ravel().tolist(), shrink=0.5)  # Make
the color bar 50% shorter
cbar.set_label('y_hat')  # Set the title of the legend to 'y_hat'

plt.show()
```

2 Random Forest

2.1 Random Forest

Random Forest is a modification of bagging that attempts to build *de-correlated trees* by considering a restricted set of features for splitting.



Algorithm 15.1 *Random Forest for Regression or Classification.*

```
1. For $b = 1$ to $B$:
    (a) Draw a bootstrap sample $\mathbf{Z}^*$ of size $N$ from the training data.
    (b) Grow a random-forest tree $T_b$ to the bootstrapped data, by re-
        cursively repeating the following steps for each terminal node of
        the tree, until the minimum node size $n_{min}$ is reached.
          i. Select $m$ variables at random from the $p$ variables.
         ii. Pick the best variable/split-point among the $m$.
        iii. Split the node into two daughter nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$.
```

To make a prediction at a new point x:

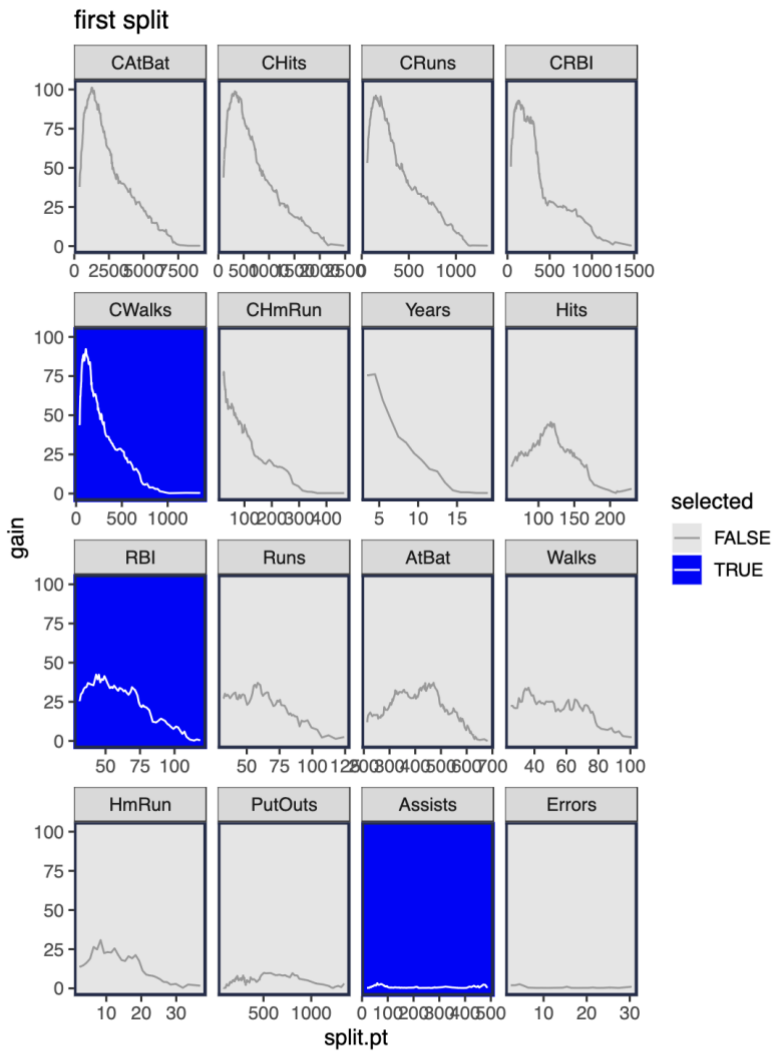
Regression: $\hat{f}_{\text{rf}}^B(x) = \frac{1}{B}\sum_{b=1}^B T_b(x)$.

Classification: Let $\hat{C}_b(x)$ be the class prediction of the bth random-forest tree. Then $\hat{C}_{\text{rf}}^B(x) = \textit{majority vote } \{\hat{C}_b(x)\}_1^B$.

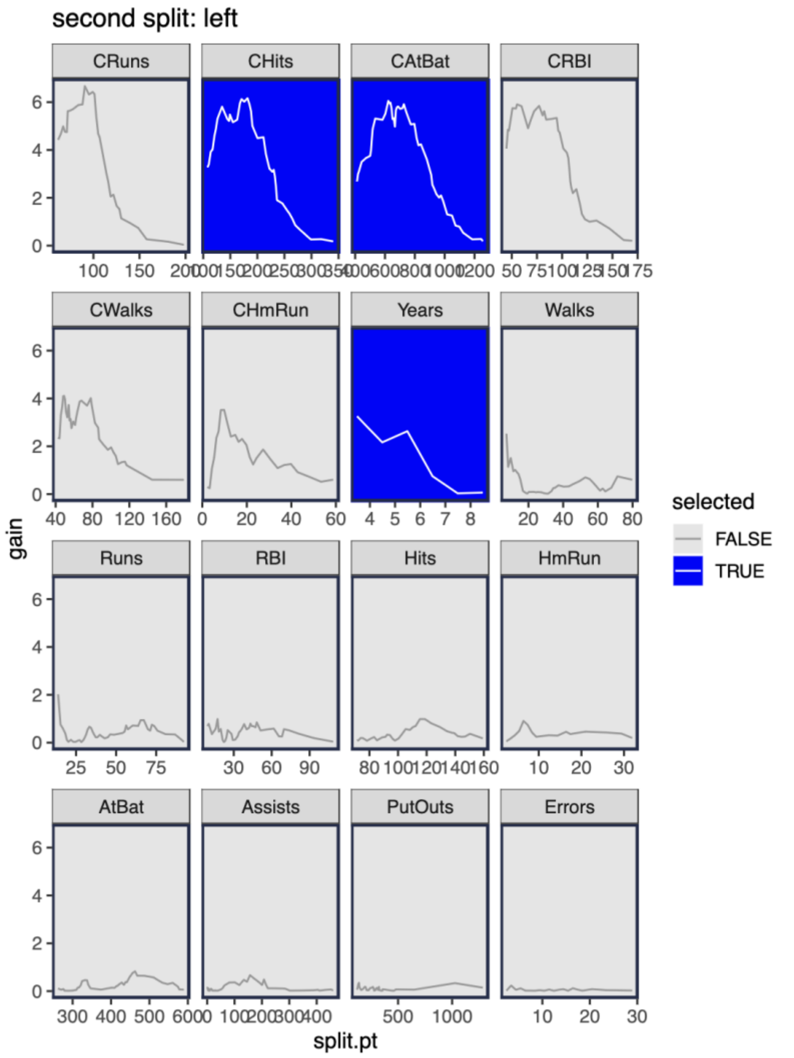
Note: I recommend aggregating the probabilities for classification trees instead of majority vote.

2.1.1 Illustration of Restricted Set of Features for Splitting

var	split.pt	n.L	n.R	est.L	est.R	SSE.L	SSE.R	SSE	gain
CAtBat	1296.5	70	130	4.942	6.433	20.06	48.35	68.42	101.109
CHits	331.0	71	129	4.964	6.432	21.93	48.93	70.86	98.661
CRuns	153.0	68	132	4.946	6.408	21.02	52.56	73.58	95.950
CRBI	140.0	73	127	5.013	6.427	26.54	50.24	76.79	92.738
CWalks	**111.0**	**74**	**126**	**5.026**	**6.431**	**27.59**	**49.88**	**77.47**	**92.053**
CHmRun	25.5	75	125	5.107	6.393	40.31	51.74	92.06	77.470
Years	4.5	69	131	5.062	6.358	32.33	61.31	93.64	75.881
Hits	117.5	114	86	5.497	6.459	73.89	50.25	124.14	45.389
RBI	**43.5**	**83**	**117**	**5.366**	**6.297**	**55.13**	**72.29**	**127.42**	**42.109**
Runs	58.5	113	87	5.534	6.401	78.66	53.93	132.59	36.932
AtBat	472.5	123	77	5.571	6.454	85.27	47.37	132.63	36.893
Walks	36.5	95	105	5.479	6.302	62.42	73.28	135.71	33.820
HmRun	8.5	94	106	5.495	6.279	69.21	69.70	138.91	30.611
PutOuts	508.0	174	26	5.826	6.482	141.56	18.22	159.79	9.737
Assists	**60.5**	**111**	**89**	**5.799**	**6.050**	**89.94**	**76.47**	**166.40**	**3.121**
Errors	4.5	71	129	5.763	5.992	47.98	119.15	167.13	2.396



var	split.pt	n.L	n.R	est.L	est.R	SSE.L	SSE.R	SSE	gain
CRuns	91.0	43	27	4.698	5.331	12.136	1.274	13.41	6.651
CHits	**182.5**	**45**	**25**	**4.721**	**5.340**	**12.713**	**1.193**	**13.91**	**6.155**
CAtBat	**624.0**	**35**	**35**	**4.648**	**5.236**	**11.302**	**2.725**	**14.03**	**6.034**
CRBI	55.5	35	35	4.652	5.232	11.434	2.737	14.17	5.890
CWalks	49.5	33	37	4.686	5.170	11.784	4.179	15.96	4.099
CHmRun	10.0	35	35	4.718	5.166	12.749	3.807	16.56	3.505
Years	**3.5**	**46**	**24**	**4.787**	**5.240**	**13.689**	**3.131**	**16.82**	**3.241**
Walks	7.5	3	67	5.838	4.902	5.198	12.345	17.54	2.518
Runs	14.0	3	67	5.742	4.906	5.963	12.090	18.05	2.008
RBI	17.5	7	63	5.296	4.903	8.443	10.643	19.09	0.975
Hits	115.5	57	13	4.886	5.189	18.843	0.243	19.09	0.974
HmRun	6.5	40	30	4.844	5.073	16.212	2.953	19.16	0.896
AtBat	463.5	58	12	4.893	5.179	19.021	0.227	19.25	0.813
Assists	158.0	51	19	4.883	5.100	10.136	9.272	19.41	0.653
PutOuts	140.5	25	45	5.034	4.891	13.077	6.653	19.73	0.331
Errors	3.5	22	48	4.859	4.980	4.290	15.553	19.84	0.218



2.2 Random Forest Tuning

There are two primary tuning parameters for Random Forest:

1. Variety: *m* controls the number of predictors that are evaluated for each split (this is named *max_features* argument in Scikit-Learn library)
2. Complexity: The depth/size of the trees are controlled by setting the *minimum number of observations in the leaf nodes* (*min_sample_leaf*) or the *depth* (*max_depth*) of the tree or the number of leaf nodes (*max_leaf_nodes*)

Code Example: Random Forest

```
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor(
    n_estimators=100,
    max_features='sqrt', # Maximum features
    max_depth=10,  # Maximum depth of each tree
    min_samples_leaf=5,  # Minimum samples at each leaf
    max_leaf_nodes=50  # Maximum number of leaf nodes per tree
)
```

max_features:

1) Integer: Specify the number of features to consider directly.
2) Float: A fraction (percentage) of features to consider at each split.
3) "auto": This option will use all the features (equivalent to None).
4) "sqrt": Uses the square root of the number of features.
5) "log2": Uses the base-2 logarithm of the number of features.

min_samples_leaf: This parameter sets the minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

max_depth: This parameter controls the maximum depth of each tree. The tree will not be split beyond this depth, limiting the complexity of each individual tree. A very deep tree can be more expressive and capture more information about the data, but it can also lead to overfitting.

max_leaf_nodes: This parameter sets the maximum number of leaf nodes a tree can have. When this is set, the best nodes are chosen as part of the splitting process, adding a level of complexity control by limiting the number of splits that can occur.

Your Turn #1

How do these tuning parameters relate to the bias/variance trade-off?

1. **max_features (Number of Features to Consider When Looking for the Best Split)**:
    - **Influence on Bias**: If `max_features` is set too low, the algorithm might not consider enough information at each split, potentially missing important patterns in the data and leading to higher bias (underfitting). The trees might become too simplistic and not capture the complexity of the data.
    - **Influence on Variance**: A higher `max_features` value means that each tree in the forest has a higher chance of being similar to others as they are all considering most of the same features for splitting. This could lead to higher variance in the model, as the trees become less diversified and the ensemble may overfit the training data. Conversely, a lower value for `max_features` encourages diversity among the trees in the forest, as each tree is trained on a different subset of features, which can lead to a reduction in variance (less overfitting).
2. **max_depth (Maximum Depth of the Tree)**:
    - **High max_depth**: Allows the trees to grow deeper, potentially making the model more complex and fit more closely to the training data. This can lead to lower bias (as the model is better at capturing complexities in the data) but higher variance (as the model might start capturing noise in the training data, leading to overfitting and poor generalization to new data).
    - **Low max_depth**: Results in shallower trees, which might not capture all the complexities or patterns in the data, leading to higher bias (underfitting) but lower variance.
3. **min_samples_leaf (Minimum Number of Samples Required at a Leaf Node)**:
    - **High min_samples_leaf**: Requires more samples at each leaf node, which smooths the model's predictions and generally leads to simpler models. This can increase bias but decrease variance.
    - **Low min_samples_leaf**: Allows trees to make decisions based on very few samples, leading to more complex, finely-tuned models that might capture noise, thus decreasing bias but increasing variance.
4. **max_leaf_nodes (Maximum Number of Leaf Nodes)**:
    - **High max_leaf_nodes**: Allows more leaf nodes in the tree, enabling the model to make more fine-grained distinctions about the data, potentially decreasing bias but increasing variance.
    - **Low max_leaf_nodes**: Restricts the model to fewer leaf nodes, which simplifies the model, potentially increasing bias but decreasing variance.

- The tuning parameters can be determined from cross-validation or OOB error

- In Python's Scikit-Learn library, the default values for the `RandomForestClassifier` and `RandomForestRegressor` in terms of the number of features considered at each split (`max_features`) and the minimum number of samples required at a leaf node (`min_samples_leaf`) are as follows:

1. **RandomForestClassifier**:
   - **`max_features`**: The default is `'auto'`, which is equivalent to `sqrt(n_features)` where `n_features` is the number of features.
   - **`min_samples_leaf`**: The default is `1`, meaning that a leaf node can have the minimum of 1 sample.
2. **RandomForestRegressor**:
   - **`max_features`**: The default is also `'auto'`, but unlike in classification, this means that all features (`n_features`) are considered at each split, not a subset. This is different from the R `randomForest` package, where the default is one-third of the number of features for regression.
   - **`min_samples_leaf`**: The default value is also `1`, like in the classifier, allowing for very fine-grained leaves.

Note
• These defaults in Scikit-Learn's implementation of Random Forests are based on empirical evidence that they generally perform well across a variety of datasets. However, they might not be optimal for every specific case, and it's often beneficial to tune these parameters, especially in cases where the dataset is very large, has many features, or when the data is imbalanced.   • It is important to highlight that while these defaults provide a good starting point, optimal performance usually requires adjusting these parameters to fit the characteristics of the specific dataset you're working with. This can be done through techniques like *grid search* or *random search* combined with cross-validation.

- The *number of trees* is another tuning parameter, but want this to be as large as possible (subject to computational and memory constraints)

2.3 OOB error

For each observation (x_i, y_i), construct its OOB prediction by averaging only those trees corresponding to bootstrap samples in which observation *i* did not appear.

$$\hat{f}(x_i) = \frac{1}{N_B(i)} \sum_{b=1}^{B} \mathbb{1}(x_i \in \mathrm{OOB(b)}) \cdot \mathrm{T}(\mathrm{x_i}; \hat{\theta}_\mathrm{b})$$

where $N_B(i)$ is the number of trees with observation *i* out-of-bag.

- Recall that there is a 37% chance that any observation is out-of-bag in any bootstrap sample.

- Thus, $N_B(i) \approx 0.37B$ (the number of trees used to estimate the OOB error is about 37% of the total number of trees in the forest).
    - More encouragement to use *many* trees in the forest

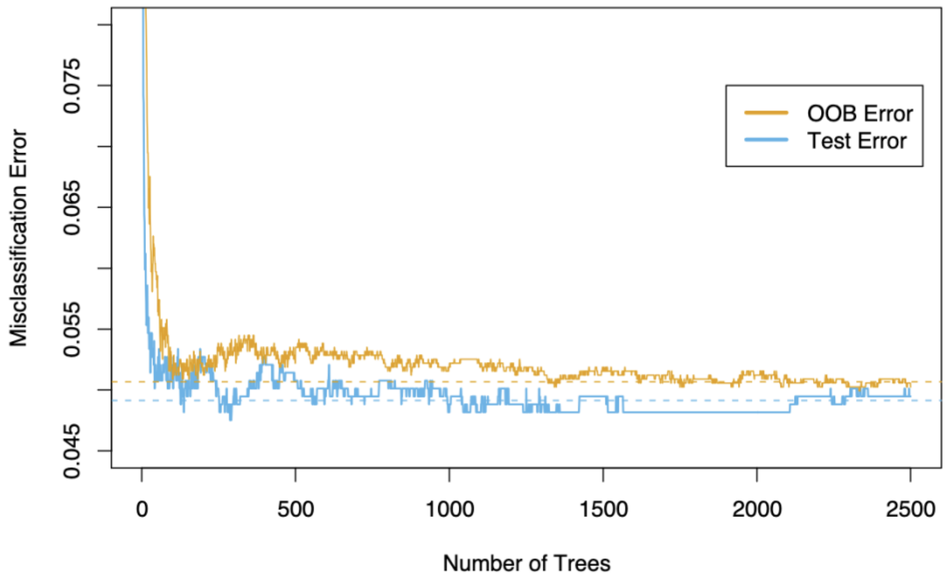


FIGURE 15.4. OOB *error computed on the* spam *training data, compared to the test error computed on the test set.*

2.4 Variable Importance

At each split in each tree, the improvement in the split-criterion is the importance measure attributed to the splitting variable, and is accumulated over all the trees in the forest separately for each variable.

The importance of predictor *j* in a single tree T:

$$\mathcal{I}_j(T) = \sum_t \text{gain}\ (t) \cdot \mathbb{1}(\text{ split } t \text{ uses feature } j)$$

That is, the importance of feature *j* in tree *T* is the total *gain* from all splits involving feature *j*. In the equation, the sum is over all splits *t* in tree *T*.

The importance of predictor *j* in a forest is the average importance from all tress in the forest:

$$\mathcal{I}_j = \frac{1}{B}\sum_{b=1}^{B} \mathcal{I}_j(T_b)$$

- Note: a final normalizing step may transform importance scores to sum to 1
- There are other ways to measure feature importance, like permutation.

2.5 Random Forest and k-NN

Random Forests (especially with fully grown trees) are similar to *k*-NN methods, but adaptively determines the neighbors.

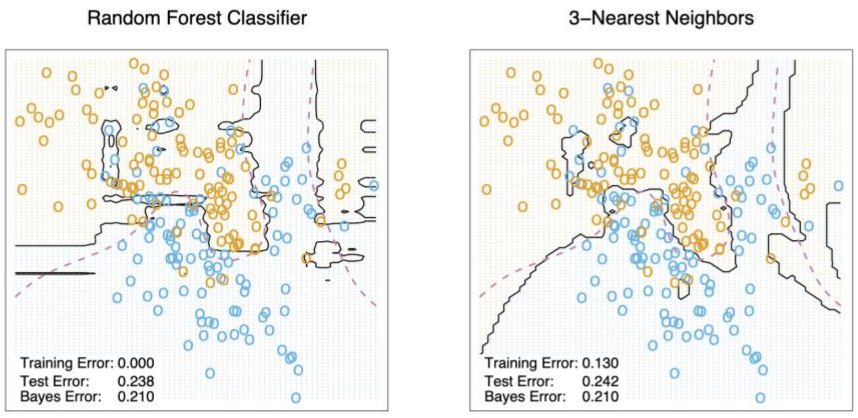


FIGURE 15.11. *Random forests versus 3-NN on the mixture data. The axis-oriented nature of the individual trees in a random forest lead to decision regions with an axis-oriented flavor.*