Lecture 15 - Feb 28

Random Forests
K-Fold Cross-Validation
Subset Selection

Reading

L. Breiman. Bagging Predictors. *Machine Learning* **24**, 123–140 (1996). https://doi.org/10.1023/A:1018054314350

T. Dietterich. Ensemble Methods in Machine Learning. In *Proceedings of the First International Workshop on Multiple Classifier Systems (MCS '00)*. https://dl.acm.org/doi/10.5555/648054.743935

L. Breiman. Random Forests. *Machine Learning* **45**, 5–32 (2001). https://doi.org/10.1023/A:1010933404324

T. Hastie, et al. *Elements of Statistical Learning*. Ch 15: Random Forests

Ensemble Learning

Last week, we saw that **bagging** (**b**ootstrap **agg**regation) averages multiple classifiers or regressors fitted to different bootstrapped data samples, which has the effect of reducing the variance of the individual models and regularizing their fits to improve generalization (test performance).

This week, we continue our work with ensemble models, of which bagging is a special case. All ensembles are involved with training multiple supervised ML models and combining their predictions such that performance of the whole is better than any of the individual parts.

Let's consider classification first.

In classification, ensembles are generally better than their parts (the individual classifiers) if some of the following three properties are satisfied:

1. Classifiers are **accurate** (better than random guessing)
2. Classifiers are **diverse** (make mistakes on *different* examples)
3. Classifiers are **cheap** (each is not too computationally costly)

The situation with regression is similar, but errors are smooth rather than binary, so ideal regressors for ensembles have somewhat low error, have errors dispersed differently across the test set, and can be trained cheaply.

Decision trees satisfy all three conditions. Trees are not the most accurate models, but they do learn (**accurate**). Trees have very high variance (**diverse**). Trees train in very little time, so ensembles of thousands of trees are viable (**cheap**). As such, trees are ideal classifiers for ensembling.

It should be noted that ensembling can be useful even if some of the conditions above are net met: for example, neural networks are very expensive to train, but they are accurate and diverse, and we do see smaller but substantial performance gains from ensembling neural networks.

We cover two sub-topics this week:

1. **Random forests** uses bagging with *de-correlated* trees to solve overfitting more effectively than simple bagging.
2. **Boosting** throws out bootstrapping in favor of sequential samples that push the individual models to focus on points where the model makes errors. This is primarily used to solve underfitting.

Each of these represent state-of-the-art models for the types of data where they are most effective.

Today, we cover **random forests**. A random forest is made up of bagged decision trees which train only on random subsets of the featues of the training data as inputs. This has the effect of making the trees less correlated to one another, which allows the ensemble to have lower variance than the individual models -- and lower than simple bagged trees. This has an exceptional regularizing effect (i.e. it improves generalization to test data).

Recall: Bagging

Consider a regression problem. Fit model to training data

$$Z = \{(x_1, y_1), \ldots, (x_N, y_N)\}$$

with prediction $\hat{f}(x)$. Bootstrap aggregation, or bagging, averages this prediction over a collection of bootstrap samples, reducing variance.

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$$

Bagging reduces variance of an estimated prediction function
Works well for high-var, low-bias models like trees.
Random forests (Breiman, 2001) builds a large collection of
de-correlated trees and averages them.

Trees are ideal for bagging. Since each tree generated in
bagging is identically distributed, expectation of a mean of B
trees is the same as each tree
$\Rightarrow$ bias is same, but var. reduction can help.

(Variance of)
Average of B i.i.d. RVs is $\frac{\sigma^2}{B}$. If just i.d. with
correlation $\rho > 0$, the variance of the average is

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

As $B \to \infty$, right term $\to 0$

As $B \to \infty$, right term $\to 0$

As B increases, the second term disappears, but the first remains, and hence the size of the correlation of pairs of bagged trees limits the benefits of averaging. The idea in random forests (Algorithm 15.1) is to improve the variance reduction of bagging by reducing the correlation between the trees, without increasing the variance too much. This is achieved in the tree-growing process through random selection of the input variables.

Specifically, when growing a tree on a bootstrapped dataset:

Before each split, select $m \leq p$ of the input variables at random as candidates for splitting.

RF Algorithm

① for $b = 1$ to B

(a) Draw bootstrap sample Z^*

(b) Grow tree T_b to Z^* by recursively doing (i)-(iii) until min node size n_{min} is reached

(i) Select m random features

(ii) Pick best variable + split-point among m

(iii) Split into two daughter nodes

Making a prediction for x:

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

Classification: $\hat{C}_b(x)$ = predicted class of bth tree

$\hat{C}_{rf}(x)$ = majority vote of $\hat{C}_1(x), \ldots, \hat{C}_B(x)$

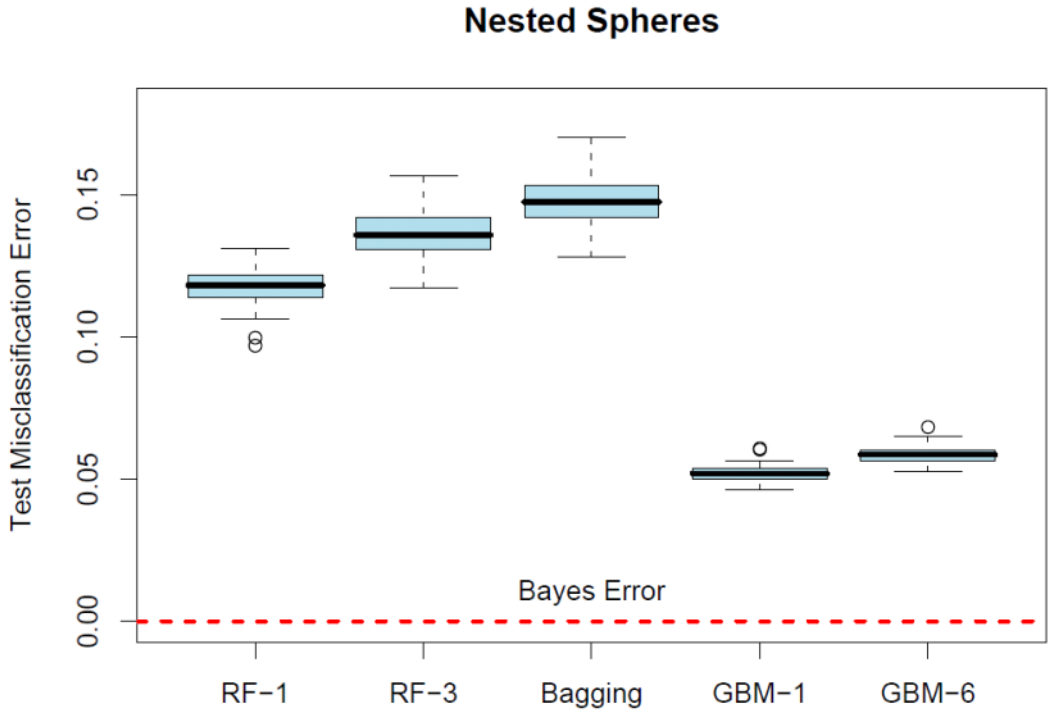
m is a hyperparameter, $1 \leq m \leq \sqrt{d}$ usually

reducing m reduces correlation between trees since fewer features will be shared.

Not all estimators can be improved by shaking up the data like this. It seems that highly nonlinear estimators, such as trees, benefit the most. For bootstrapped trees, ρ is typically small (0.05 or lower is typical; see Figure 15.9), while σ^2 is not much larger than the variance for the original tree. On the other hand, bagging does not change *linear* estimates, such as the sample mean (hence its variance either); the pairwise correlation between bootstrapped means is about 50% (Exercise 15.4).

RFs tend to work well with little tuning, but gradient boosted trees are often better

The authors make grand claims about the success of random forests: "most accurate," "most interpretable," and the like. In our experience random forests do remarkably well, with very little tuning required. A random forest classifier achieves 4.88% misclassification error on the `spam` test data, which compares well with all other methods, and is not significantly worse than gradient boosting at 4.5%. Bagging achieves 5.4% which is significantly worse than either (using the McNemar test outlined in Exercise 10.6), so it appears on this example the additional randomization helps.



15.3 Details of RFs

- Classification: $1 \le m \le \lfloor \sqrt{d} \rfloor$
- Regression: $5 \le m \le \lfloor \frac{d}{3} \rfloor$

defaults recommended by Breiman et al. (Hyperparameters)

these are not always good... example in ESL had $m = 6 > \lfloor 8/3 \rfloor = 2$ and performed much better

15.3.1 Out of Bag (OOB) Samples

For each z_i, construct RF predictor only with trees corresponding to bootstrap samples not including z_i

An OOB error estimate is almost identical to that obtained by N-fold cross-validation; see Exercise 15.2. Hence unlike many other nonlinear estimators, random forests can be fit in one sequence, with cross-validation being performed along the way. Once the OOB error stabilizes, the training can be terminated.

15.3.4 RFs and Overfitting

When d is large but only a few vars are important, RFs perform poorly with small m since good vars will usually not be in each tree

(Hypergeometric probabilities for # vars in tree)

Feature 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 left plot

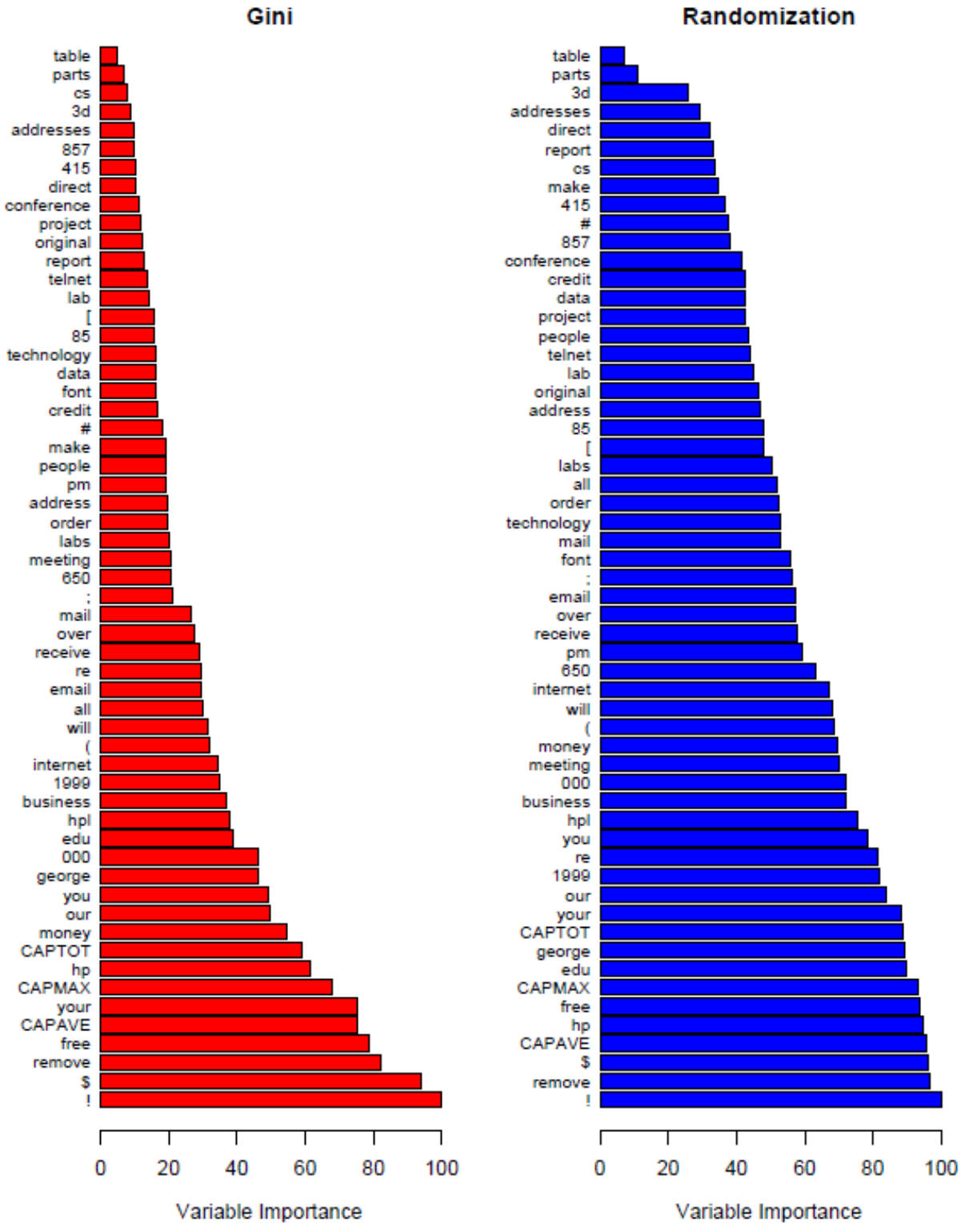


FIGURE 15.5. *Variable importance plots for a classification random forest grown on the* spam *data. The left plot bases the importance on the Gini splitting index, as in gradient boosting. The rankings compare well with the rankings produced by gradient boosting (Figure 10.6 on page 354). The right plot uses* OOB *randomization to compute variable importances, and tends to spread the importances more uniformly.*

Random forests also use the OOB samples to construct a different *variable-importance* measure, apparently to measure the prediction strength of each variable. When the bth tree is grown, the OOB samples are passed down the tree, and the prediction accuracy is recorded. Then the values for the jth variable are randomly permuted in the OOB samples, and the accuracy is again computed. The decrease in accuracy as a result of this permuting is averaged over all trees, and is used as a measure of the importance of variable j in the random forest. These are expressed as a percent of the maximum in the right plot in Figure 15.5. Although the rankings of the two methods are similar, the importances in the right plot are more uniform over the variables. The randomization effectively voids the effect of a variable, much like setting a coefficient to zero in a linear model (Exercise 15.7). This does not measure the effect on prediction were this variable not available, because if the model was refitted without the variable, other variables could be used as surrogates.

Hyperparameter Effects

When the number of variables is large, but the fraction of relevant variables small, random forests are likely to perform poorly with small m. At each split the chance can be small that the relevant variables will be selected. Figure 15.7 shows the results of a simulation that supports this claim. Details are given in the figure caption and Exercise 15.3. At the top of each pair we see the hyper-geometric probability that a relevant variable will be selected at any split by a random forest tree (in this simulation, the relevant variables are all equal in stature). As this probability gets small, the gap between boosting and random forests increases. When the number of relevant variables increases, the performance of random forests is surprisingly robust to an increase in the number of noise variables. For example, with 6 relevant and 100 noise variables, the probability of a relevant variable being selected at any split is 0.46, assuming $m = \sqrt{(6 + 100)} \approx 10$. According to Figure 15.7, this does not hurt the performance of random forests compared with boosting. This robustness is largely due to the relative insensitivity of misclassification cost to the bias and variance of the probability estimates in each tree. We consider random forests for regression in the next section.

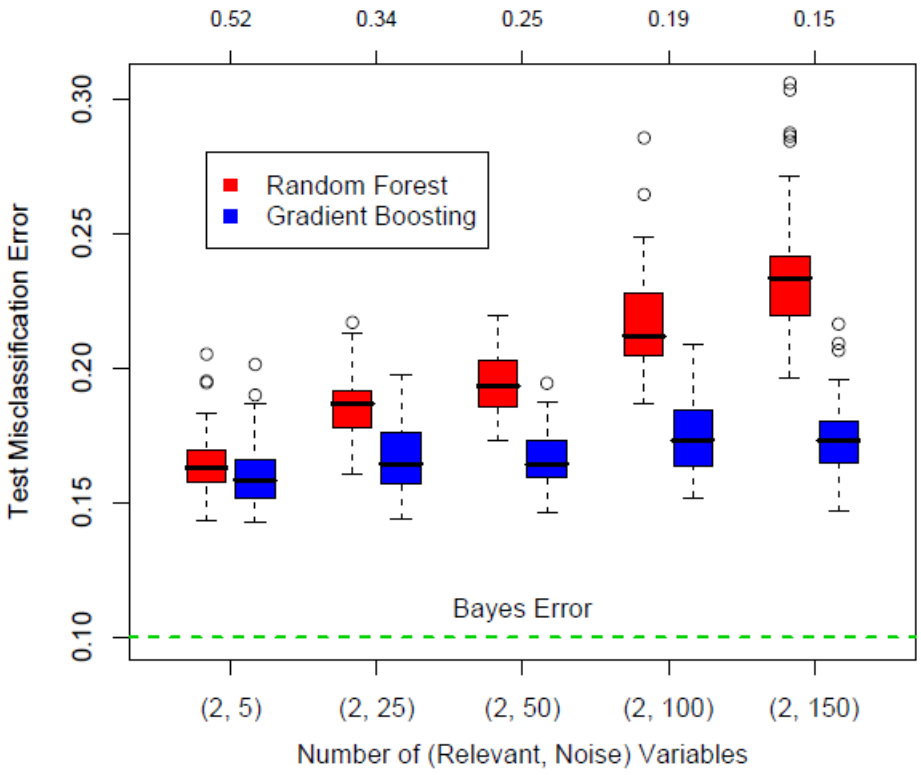


FIGURE 15.7. *A comparison of random forests and gradient boosting on problems with increasing numbers of noise variables. In each case the true decision boundary depends on two variables, and an increasing number of noise variables are included. Random forests uses its default value* $m = \sqrt{p}$. *At the top of each pair is the probability that one of the relevant variables is chosen at any split. The results are based on* 50 *simulations for each pair, with a training sample of* 300, *and a test sample of* 500.

Figure 15.8 shows the modest effect of depth control in a simple regression example. Classifiers are less sensitive to variance, and this effect of overfitting is seldom seen with random-forest classification.

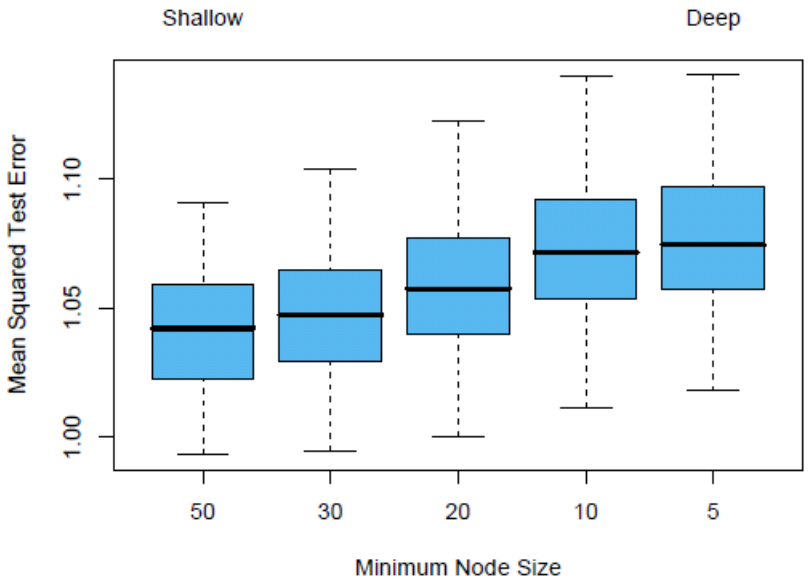


FIGURE 15.8. *The effect of* tree size *on the error in random forest regression. In this example, the true surface was additive in two of the* 12 *variables, plus additive unit-variance Gaussian noise. Tree depth is controlled here by the minimum node size; the smaller the minimum node size, the deeper the trees.*

$\rho(x)$ is the *sampling* correlation between any pair of trees used in the averaging:

$$\rho(x) = \text{corr}[T(x; \Theta_1(\mathbf{Z})), T(x; \Theta_2(\mathbf{Z}))], \tag{15.6}$$

where $\Theta_1(\mathbf{Z})$ and $\Theta_2(\mathbf{Z})$ are a randomly drawn pair of random forest trees grown to the randomly sampled $\mathbf{Z}$;

Figure 15.9 shows how the correlation (15.6) between pairs of trees decreases as m decreases: pairs of tree predictions at x for different training sets $\mathbf{Z}$ are likely to be less similar if they do not use the same splitting variables.

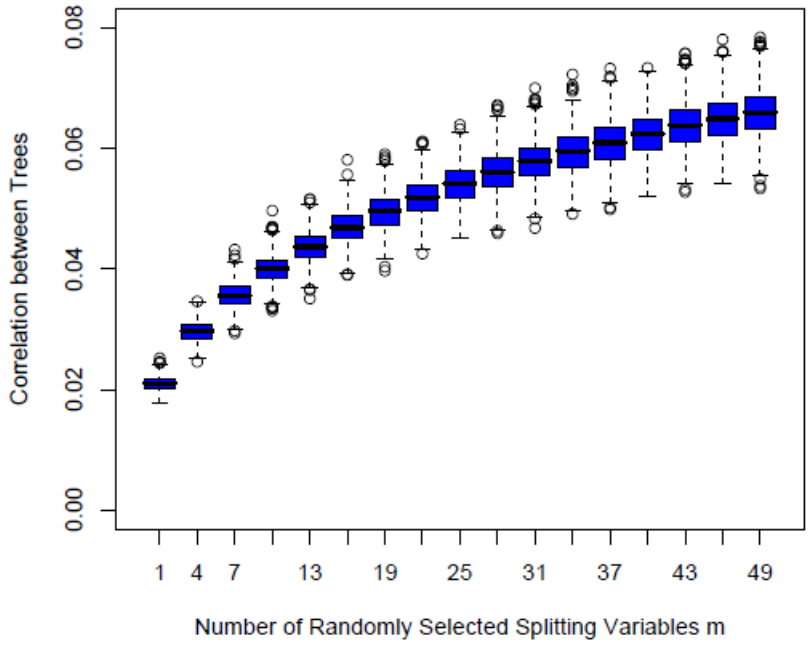


FIGURE 15.9. *Correlations between pairs of trees drawn by a random-forest regression algorithm, as a function of m. The boxplots represent the correlations at 600 randomly chosen prediction points x.*

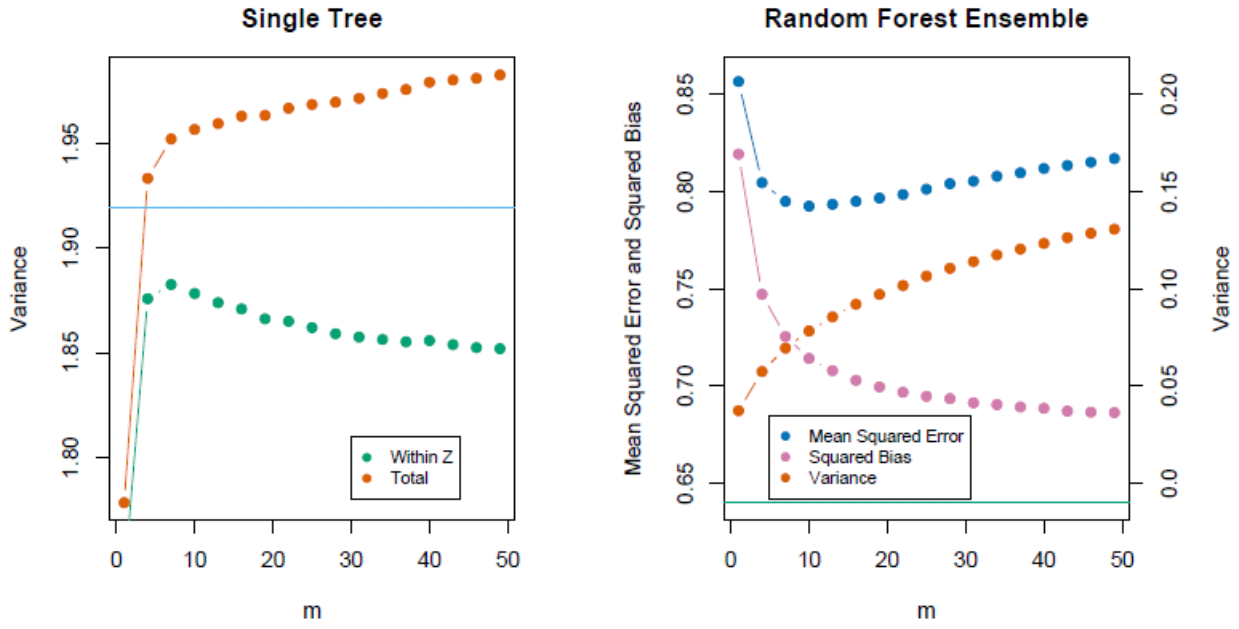


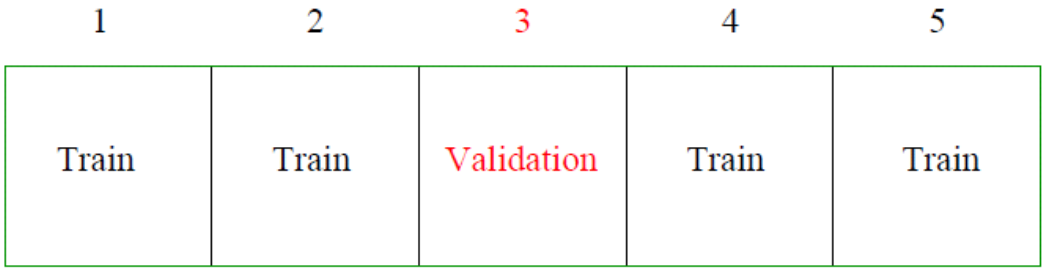
FIGURE 15.10. *Simulation results. The left panel shows the average variance of a single random forest tree, as a function of m. "Within* $\mathbf{Z}$*" refers to the average within-sample contribution to the variance, resulting from the bootstrap sampling and split-variable sampling (15.9). "Total" includes the sampling variability of* $\mathbf{Z}$*. The horizontal line is the average variance of a single fully grown tree (without bootstrap sampling). The right panel shows the average mean-squared error, squared bias and variance of the ensemble, as a function of m. Note that the variance axis is on the right (same scale, different level). The horizontal line is the average squared-bias of a fully grown tree.*

K-Fold Cross-Validation

Probably the simplest and most widely used method for estimating prediction error is cross-validation. This method directly estimates the expected extra-sample error $\mathrm{Err} = \mathrm{E}[L(Y, \hat{f}(X))]$, the average generalization error when the method $\hat{f}(X)$ is applied to an independent test sample from the joint distribution of X and Y. As mentioned earlier, we might hope that cross-validation estimates the conditional error, with the training set $\mathcal{T}$ held fixed. But as we will see in Section 7.12, cross-validation typically estimates well only the expected prediction error.

7.10.1 K-Fold Cross-Validation

Ideally, if we had enough data, we would set aside a validation set and use it to assess the performance of our prediction model. Since data are often scarce, this is usually not possible. To finesse the problem, K-fold cross-validation uses part of the available data to fit the model, and a different part to test it. We split the data into K roughly equal-sized parts; for example, when $K = 5$, the scenario looks like this:



For the kth part (third above), we fit the model to the other $K-1$ parts of the data, and calculate the prediction error of the fitted model when predicting the kth part of the data. We do this for $k = 1, 2, \ldots, K$ and combine the K estimates of prediction error.

Here are more details. Let $\kappa : \{1, \ldots, N\} \mapsto \{1, \ldots, K\}$ be an indexing function that indicates the partition to which observation i is allocated by the randomization. Denote by $\hat{f}^{-k}(x)$ the fitted function, computed with the kth part of the data removed. Then the cross-validation estimate of prediction error is

$$\mathrm{CV}(\hat{f}) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, \hat{f}^{-\kappa(i)}(x_i)). \tag{7.48}$$

Typical choices of K are 5 or 10 (see below). The case $K = N$ is known

Here are more details. Let $\kappa : \{1, \ldots, N\} \mapsto \{1, \ldots, K\}$ be an indexing function that indicates the partition to which observation i is allocated by the randomization. Denote by $\hat{f}^{-k}(x)$ the fitted function, computed with the kth part of the data removed. Then the cross-validation estimate of prediction error is

$$\mathrm{CV}(\hat{f}) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, \hat{f}^{-\kappa(i)}(x_i)). \tag{7.48}$$

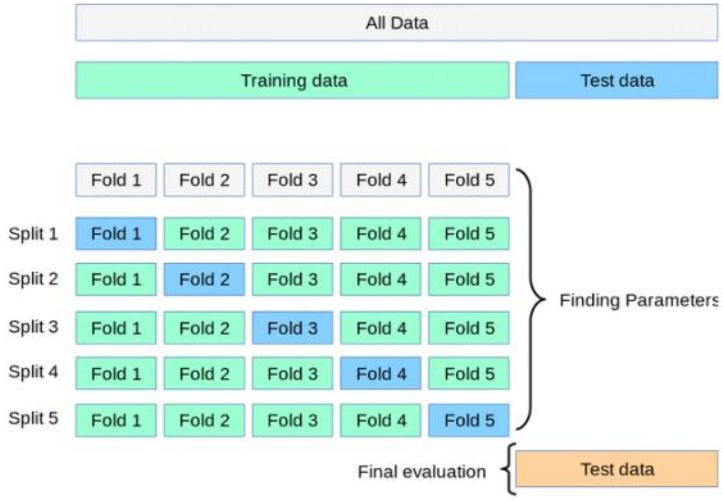
Typical choices of K are 5 or 10 (see below). The case $K = N$ is known as *leave-one-out* cross-validation. In this case $\kappa(i) = i$, and for the ith observation the fit is computed using all the data except the ith.

Given a set of models $f(x, \alpha)$ indexed by a tuning parameter α, denote by $\hat{f}^{-k}(x, \alpha)$ the αth model fit with the kth part of the data removed. Then for this set of models we define

$$\mathrm{CV}(\hat{f}, \alpha) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, \hat{f}^{-\kappa(i)}(x_i, \alpha)). \tag{7.49}$$

The function $\mathrm{CV}(\hat{f}, \alpha)$ provides an estimate of the test error curve, and we find the tuning parameter $\hat{\alpha}$ that minimizes it. Our final chosen model is $f(x, \hat{\alpha})$, which we then fit to all the data.

The performance measure reported by *k*-fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive, but does not waste too much data (as is the case when fixing an arbitrary validation set), which is a major advantage in problems such as inverse inference where the number of samples is very small.



When evaluating different settings ("hyperparameters") for estimators, such as the `C` setting that must be manually set for an SVM, there is still a risk of overfitting *on the test set* because the parameters can be tweaked until the estimator performs optimally. This way, knowledge about the test set can "leak" into the model and evaluation metrics no longer report on generalization performance. To solve this problem, yet another part of the dataset can be held out as a so-called "validation set": training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.

A solution to this problem is a procedure called cross-validation (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV. In the basic approach, called *k*-fold CV, the training set is split into *k* smaller sets (other approaches are described below, but generally follow the same principles). The following procedure is followed for each of the *k* "folds":

- A model is trained using $k - 1$ of the folds as training data;
- the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy).

The performance measure reported by *k*-fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive, but does not waste too much data (as is the case when fixing an arbitrary validation set), which is a major advantage in problems such as inverse inference where the number of samples is very small.

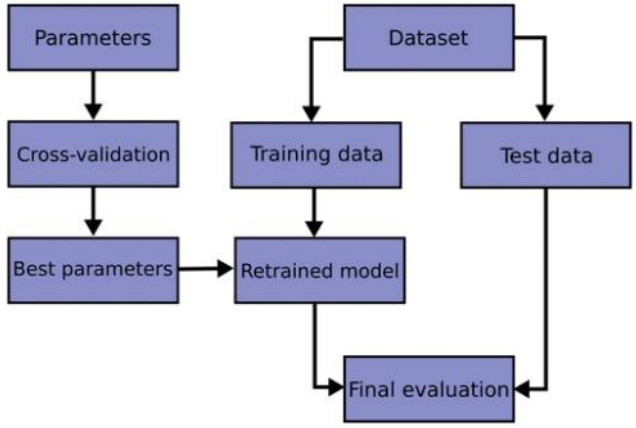
Algorithm 22.2: K-fold Cross-Validation

CROSS-VALIDATION(K, $\mathbf{D}$):

1 $\mathbf{D} \leftarrow$ randomly shuffle $\mathbf{D}$
2 $\{\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_K\} \leftarrow$ partition $\mathbf{D}$ in K equal parts
3 **for** $i \in [1, K]$ **do**
4 $\quad M_i \leftarrow$ train classifier on $\mathbf{D} \setminus \mathbf{D}_i$
5 $\quad \theta_i \leftarrow$ assess M_i on $\mathbf{D}_i$
6 $\hat{\mu}_\theta = \frac{1}{K} \sum_{i=1}^{K} \theta_i$
7 $\hat{\sigma}_\theta^2 = \frac{1}{K} \sum_{i=1}^{K} (\theta_i - \hat{\mu}_\theta)^2$
8 **return** $\hat{\mu}_\theta, \hat{\sigma}_\theta^2$

3.1. Cross-validation: evaluating estimator performance

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called **overfitting**. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a **test set** `X_test, y_test`. Note that the word "experiment" is not intended to denote academic use only, because even in commercial settings machine learning usually starts out experimentally. Here is a flow-chart of typical cross validation workflow in model training. The best parameters can be determined by grid search techniques.



In scikit-learn a random split into training and test sets can be quickly computed with the `train_test_split` helper function. Let's load the iris data set to fit a linear support vector machine on it:

```
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> from sklearn import datasets
>>> from sklearn import svm

>>> X, y = datasets.load_iris(return_X_y=True)
>>> X.shape, y.shape
((150, 4), (150,))
```

We can now quickly sample a training set while holding out 40% of the data for testing (evaluating) our classifier:

```
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.4, random_state=0)

>>> X_train.shape, y_train.shape
((90, 4), (90,))
>>> X_test.shape, y_test.shape
((60, 4), (60,))

>>> clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.96...
```

Issues with least squares estimates:

(1) prediction accuracy: low bias, high variance
↳ Can be improved by shrinking coefficients sometimes which sacrifices some bias for less var.

(2) Interpretation: with many variables, we may prefer a subset of most important variables

There are several methods for subset selection

3.3.1 Best-Subset Selection

Find the best subset of size $k=1,\dots,d$ with smallest SSE.

(brute force/leaps + bounds)

3.3.2 Fwd- + Bwd-Stepwise Selection

Best-subset selection is too slow for large d...

Best-subset selection is too slow for large d...

Fwd-stepwise selection fits intercept, then adds the predictor that improves fit most + makes a path to k-subset sequentially

(greedy search)

Bwd-stepwise starts with all vars + deletes greedily 1-by-1

Stepwise searches have lower variance

Some solutions consider fwd + bwd moves at each step to chose the best step, sometimes using AIC criterion

With dummies, smart to add or drop the whole group

3.3.3 Fwd-Stagewise Regression

Even more constrained than stepwise search...

0. Start by fitting $\theta_0 = \bar{y}$.

while an unused predictor is correlated with residual

- select most highly correlated one
- compute simple linear regression coef of residual on predictor
- add this to current coefficient

Works well with large d sometimes

Aviation Example

```
from itertools import combinations

import numpy as np
import pandas as pd

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split, cross_val_score
```

```
data = pd.read_csv('data.csv', sep = ',')
data = data.drop(columns = ['Country'])
data
```

```
pax = '# Pax Per 1 Pop'

for i, var in enumerate(data.columns.to_numpy()):
    print(i, var)

# output is the pax
y = data[pax]

# all data except pax are predictors
X = data.drop(columns = pax)
```

```
0 Population
1 Consumer price index: General
2 Patents in force (number)
3 Gross domestic expenditure on R & D: as a percentage of GDP (%)
4 Unemployment rate - Total
5 Tourist/visitor arrivals (thousands)
6 Tourism expenditure (millions of US dollars)
7 Balance of Payments: Current account (millions of US dollars)
8 Balance imports/exports (millions of US dollars)
9 Public expenditure on education (% of government expenditure)
10 Public expenditure on education (% of GDP)
11 Total population of concern to UNHCR (number)
12 GDP per capita (US dollars)
13 Labour force participation - Total
14 # Pax Per 1 Pop
```

```
# find the "best" linear regression model
def best_subset(estimator, X, y, max_size=15, cv=5):
    n_features = X.shape[1]

    # create all subsets of the features/attributes
    subsets = (combinations(range(n_features), k + 1) for k in range(min(n_features, max_size)))

    # find the best model of each subset size
    best_size_subset = []

    # for each subset size
    for subsets_k in subsets:
        best_score = -np.inf
        best_subset = None

        # for each subset
        for subset in subsets_k:

            # fit the model on variables in subset
            estimator.fit(X.iloc[:, list(subset)], y)

            # get the subset with the best r^2 score among subsets of the same size
            score = estimator.score(X.iloc[:, list(subset)], y)

            # if the score is the best we have found at this size, save and print it
            if score > best_score:
                print(subset, score)
                best_score, best_subset = score, subset

        # first store the best subset of each size
        best_size_subset.append(best_subset)
```

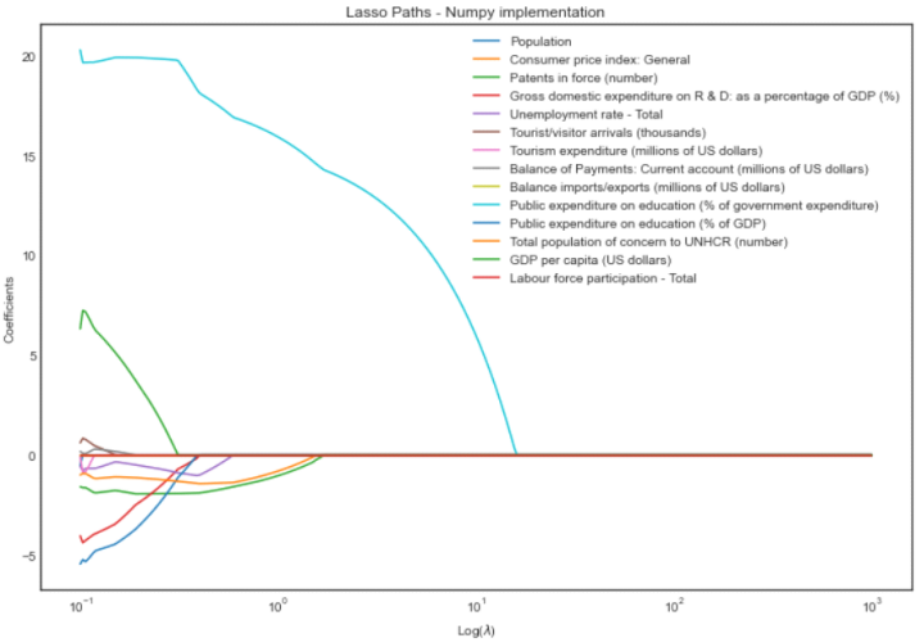
```
    # compare best subsets of each size
    best_score = -np.inf
    best_subset = None
    list_scores = []

    # find the "best of the best" models
    for subset in best_size_subset:

        # find the mean of r^2 scores from cross-validation and save it
        score = cross_val_score(estimator, X.iloc[:, list(subset)], y, cv=cv).mean()
        list_scores.append(score)

        # save the score and model if it's the best
        if score > best_score:
            print(score, subset)
            best_score, best_subset = score, subset

    # return the best model, best cross-validated r^2 score, all best models (by r^2)
    # of each size, and their cross-validated r^2 scores
    return best_subset, best_score, best_size_subset, list_scores
```

```
lm = LinearRegression()
print(best_subset(lm, X, y, max_size=15, cv=5))
```

```
pax = '# Pax Per 1 Pop'

# output is the pax
y = data[pax].to_numpy().reshape(-1, 1)

# all data except pax are predictors
X = data.drop(columns = pax).to_numpy()
```

```
# Initialize variables
m,n = X.shape
initial_theta = np.ones((n,1))
theta_list = list()
lamda = np.logspace(0,4,300)/10 #Range of lambda values

#Run lasso regression for each lambda
for l in lamda:
    theta = coordinate_descent_lasso(initial_theta,X,y,lamda = l, num_iters=100)
    theta_list.append(theta)

#Stack into numpy array
theta_lasso = np.stack(theta_list).T

#Plot results
n,_ = theta_lasso.shape
plt.figure(figsize = (12,8))

for i in range(n):
    plt.plot(lamda, theta_lasso[i], label = data.columns[i])

plt.xscale('log')
plt.xlabel('Log($\\lambda$)')
plt.ylabel('Coefficients')
plt.title('Lasso Paths - Numpy implementation')
plt.legend()
plt.axis('tight')
```

```
(0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 12, 13) 0.9999974853451451
(0, 1, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13) 0.9999985851641283
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12) 1.0
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13) 1.0
-2.946695607703091 (9,)
-2.147337137077943 (3, 9)
-0.7012090059421493 (1, 3, 9)
0.2960446747622578 (1, 3, 7, 9)
0.7454435861894739 (0, 2, 3, 5, 6, 9, 10, 12, 13)
((0, 2, 3, 5, 6, 9, 10, 12, 13), 0.7454435861894739, [(9,), (3, 9), (1, 3, 9), (1, 3, 7, 9), (1, 3, 7, 9, 12), (3, 5, 6, 9, 10, 12), (0, 3, 5, 6, 9, 10, 12),
(0, 2, 3, 5, 6, 9, 10, 12), (0, 2, 3, 5, 6, 9, 10, 12, 13), (0, 1, 2, 3, 5, 6, 7, 9, 10, 12), (0, 1, 2, 3, 5, 6, 7, 9, 10, 11, 12), (0, 1, 3, 4, 5, 6, 8, 9, 1
0, 11, 12, 13), (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)], [-2.946695607703091, -2.147337137077943, -0.70120
90059421493, 0.2960446747622578, -0.028525771201065076, -1.9439177309390097, -0.2594129238284899, 0.051931505619169725, 0.7454435861894739, 0.607857569248572
1, -13.79336921080726, -6.916133988260947, -132.3862873633211, -839.6190273686864])
```


Lasso Paths - Numpy implementation
Population
Consumer price index: General
Patents in force (number)
Gross domestic expenditure on R & D: as a percentage of GDP (%)
Unemployment rate - Total
Tourist/visitor arrivals (thousands)
Tourism expenditure (millions of US dollars)
Balance of Payments: Current account (millions of US dollars)
Balance imports/exports (millions of US dollars)
Public expenditure on education (% of government expenditure)
Public expenditure on education (% of GDP)
Total population of concern to UNHCR (number)
GDP per capita (US dollars)
Labour force participation - Total
Coefficients
20
15
10
5
0
−5
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
Log(λ)