# The Evaluation of Management Practices

## 6. Predictions and Machine Learning

Dirk Sliwka, Jesper Armouti-Hansen

University of Cologne

Fall Term 2021

## Course content

1. Introduction
2. Regression
3. Classification
4. Model selection and assessment
5. Tree-based methods
6. Model interpretation
7. Final remarks and outlook

## 1. Introduction

## What is Machine Learning?

It is hard to find one accepted definition of machine learning (ML).

- For our purposes, we will treat the terms machine learning and statistical learning as identical.
- In this respect, the goal is to cover methods that are able to predict an outcome of interest based on inputs.
  - Example: Predict a house's selling price based on the features of it.
  - Another example: Predict whether an employee will leave a company based on his/her age, wage,...
- Both of the examples are instances of what we call *supervised learning*. There are, however, other types of learning problems.

## Types of Machine Learning

**Supervised Learning**

- The task of learning a function that maps an input to an output based on example input-output pairs.
- The learning method learns from a training sample consisting of a set of input-output observations.
- In general, there are two types of supervised learning problems: *regression* and *classification*.
- In a regression problem, the outcome is quantitative:
  - Example: Predict a house's selling price based on the features of it.
- In a classification problem, the outcome takes values in a finite unordered set.
  - Example: Predict whether an employee will leave a company based on his/her age, wage,...

## Unsupervised Learning

- We observe inputs but no outputs
- We can seek to understand the relationship between the variables or between the observations
- For example, in a market segmentation study we might observe multiple characteristics for potential customers and attempt to see if customers can be clustered into groups based on these characteristics
- If we have high-dimensional data, we might use unsupervised methods to project our inputs onto a lower dimensional space
- This can be a pre-processing step for a supervised learning method
- We do not cover unsupervised learning here.

## Starting point

- Dependent variable (output) $Y$.
  - In a regression problem, $Y$ is quantitative (i.e., $Y \in \mathbb{R}$).
  - In a classification problem, $Y$ takes values in a finite unordered set.
- Independent variables (inputs) are denoted by the $p$-dimensional vector $X$.
- $(X, Y)$ follows the joint distribution $P$. Furthermore, $P_{Y|X}$ denotes the conditional distribution of $Y$ given $X$.
- We have a sample $\mathcal{D} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ of known observation drawn from $P$.
- Goal:
  - Estimate a function $f$ that accurately predicts $Y$ given $X$.
  - Understand which independent variables affects the dependent variable, and how.
  - Assess the quality of our predictions and make inferences.

## 2. Regression

## Regression problem

- Note that one of our goals is to assess the quality of our predictions based on the function $f$ that makes predictions of $Y$ given $X$.
- To do this, we need some measurement of what a "good" and "bad prediction" is. Specifically, we need a "loss" function that penalizes "bad" predictions that are far away from the outcome.
- A common choice in a regression problem is to use the squared loss:

$$\ell(f(x), y) = (y - f(x))^2 \tag{1}$$

Suppose we knew $P_{Y|X}$. The following property follows:

**CEF prediction property**

Let $f(x) = E[Y|X = x]$, i.e., the conditional expectation function (CEF). Furthermore, let $m$ be any function. It follows that the CEF solves:

$$f(x) = \arg\min_{m(X)} E[(Y - m(X))^2|X = x] \tag{2}$$

- Hence, if the squared loss is our measurement to which we evaluate the quality of our predictions, we can do no better than the CEF.
- That is, if we knew the CEF, we could make predictions on which values $Y$ would take for different values of $X$.

In addition, recall the following property of the CEF:

**CEF decomposition property**

Let $f(x) = E[Y|X = x]$, i.e., the conditional expectation function (CEF). We can decompose $Y$ as follows:

$$Y = f(x) + \epsilon \tag{3}$$

where

1. $\epsilon$ is mean independent of $X$: $E[\epsilon|X = x] = 0$.
2. $\epsilon$ is uncorrelated with any function of $X$.

- Hence, $f$ captures all of the variation in $Y$ that is predictable by $X$.
- However, usually "noise" remains which means that we cannot perfectly predict $Y$ given $X$, even if we knew the CEF.

## The irreducible error

Suppose we have estimated $f(x)$ by $\hat{f}(x)$ for $X = x$. Then we have

$$E[(Y - \hat{f}(X))^2 | X = x] = \underbrace{(f(x) - \hat{f}(x))^2}_{\text{reducible}} + \underbrace{Var[\epsilon]}_{\text{irreducible}} \qquad (4)$$

- We focus on techniques for estimating $f$ with the aim of minimizing the reducible error.
- The irreducible error provides an upper bound on the accuracy, but is almost always unknown.

## Estimating $f$ with $\mathcal{D}$ - KNN regression

With our training data, we might attempt to directly apply the concept of $E[Y|X = x]$ by asking, at each point $x$, for the average $y$.

- Since we rarely have sufficient data points to do this, we can settle for:

$$\hat{f}(x) = \frac{1}{K} \sum_{x_i \in N_K(x)} y_i \qquad (5)$$

  Where $N_K(x)$ is a neighborhood containing the $K$ "closest" known points of $x$ in $\mathcal{D}$. Note:

- Expectation is approximated by averaging over sample data;
- Conditioning at a point is relaxed to conditioning on some region "close" to the target point.
- This approach is called $K$ nearest neighbors (KNN) regression.

## Estimating $f$ with $\mathcal{D}$ - Linear regression

Another (parametric) approach is to assume that $f$ (CEF) is approximately linear:

$$f(X) \approx \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p \tag{6}$$

- Although $f$ is almost never linear, such an assumption is reasonable in many settings.
- Based on our sample $\mathcal{D}$ we thus estimate:

$$\hat{f}(x) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_p x_p \tag{7}$$

- See the previous lecture slides on regression for further details.

## Parametric vs. Non-parametric methods

Broadly speaking, we can assign any of the regression (and classification) methods into one of the following two groups:

1. Parametric methods
   - We make an assumption about the functional form of $f$.
   - For example, if we assume $f$ is linear, we only need to estimate $p+1$ coefficients as opposed to an arbitrary $p$-dimensional function.
2. Non-parametric methods
   - No explicit assumptions about the functional for of $f$.
   - Attempts to give an estimate of $f$ close to observed data points subject to pre-specified constraints.

Both methods have their pros and cons as we shall see.

# Example

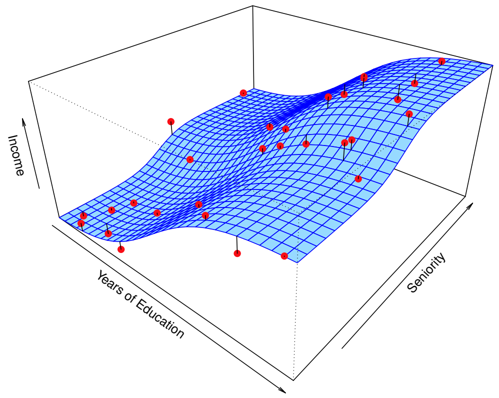


Figure: Simulated data: Income as a function of years of education and seniority. The blue surface represents the true underlying relationship (i.e. $f$). Red dots indicate observed values for 30 individuals (i.e. $\mathcal{D}$) (See ISLR p. 18)

Parametric estimation:

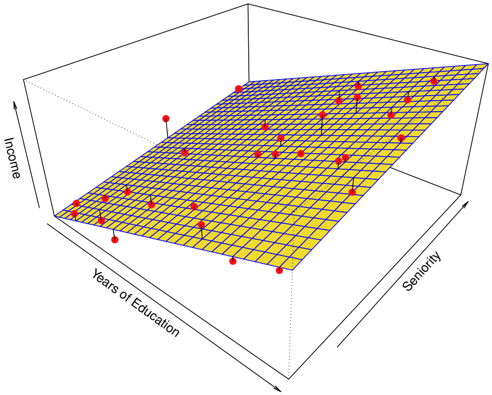


Figure: A linear model fit by least squares:
$\hat{f} = \hat{\beta}_0 + \hat{\beta}_1 * \textit{education} + \hat{\beta}_2 * \textit{seniority}$ (See ISLR p. 22)

## Non-parametric estimation:

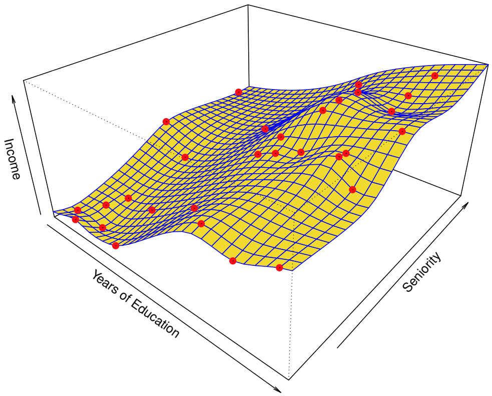


Figure: A rough thin-plate spline fit to the same income data (See ISLR p. 24)

## Prediction Accuracy vs. Model Interpretability

Why would we ever choose to use more restrictive models instead of a very flexible approach?

- More complex models require more data.
- Recall that one of our goals is to understand how the independent variables vary with $Y$.
  - Some of the models become so complex that understanding how any individual input is associated with the output becomes difficult
- We might overfit with highly flexible methods. We will see this shortly.

## Assessing Model Accuracy

Why introduce many different methods?

- No Free Lunch: No one method dominates all others over all possible data sets.

Hence an important task is deciding on the best method for a given data set. To decide on a method, we need a metric to evaluate the quality of the predictions.

- We could compute the mean squared error directly in our sample $\mathcal{D}$:

$$MSE_{\mathcal{D}} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{f}(x_i))^2 \tag{8}$$

- However, as $Y = f(x) + \epsilon$, this will in most cases be a biased estimate of the expected squared error.
- Specifically, it may substantially underestimate the error of complex models that are fitting the "noise".

- Instead, a better approach is to have a designated hold-out or test set $\mathcal{D}_{Te} = \{(x_1, y_1), \ldots, (x_M, y_M)\}$ to estimate it:

$$MSE_{\mathcal{D}_{Te}} = \frac{1}{M} \sum_{i=1}^{M} (y_i - \hat{f}(x_i))^2 \tag{9}$$

- This test MSE will then serve as a unbiased estimate of the expected squared error.
- However, notice that this cannot be done without a cost:
  - To construct a test set, we need to split our sample $\mathcal{D}$ into a training set $\mathcal{D}_{Tr}$ on which we fit our model, and a test set $\mathcal{D}_{Te}$ on which we evaluate its predictions.
- Let us consider some simulated examples to understand why this is necessary.

# Example

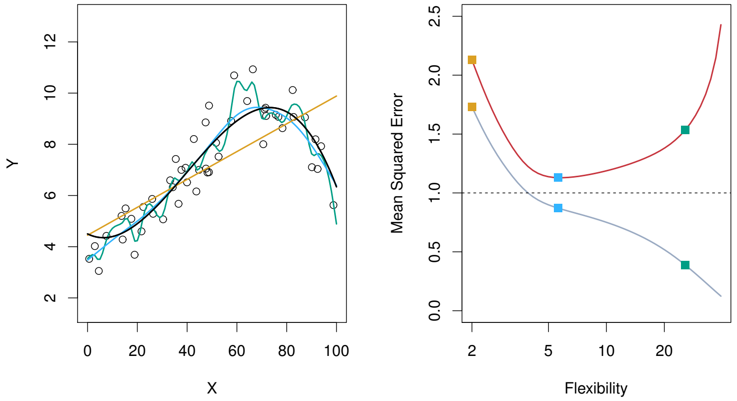


Figure: Simulated data: true $f$ (black), linear regression line (orange), and two smoothing splines (blue and green) (See ISLR p. 31)

## Example

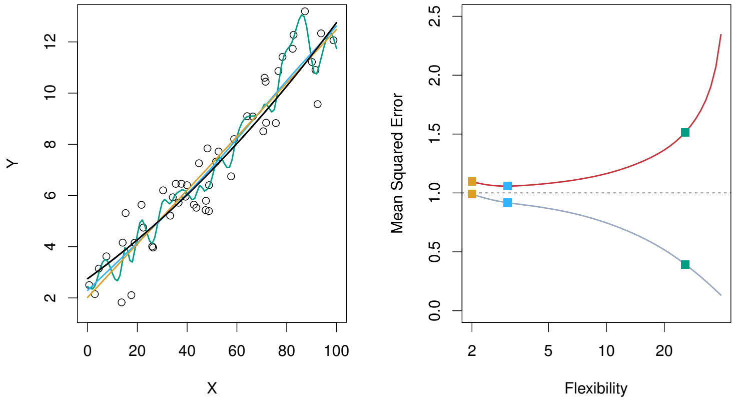


Figure: Simulated data: true *f* (black), linear regression line (orange), and two smoothing splines (blue and green) (See ISLR p. 33)

## Example

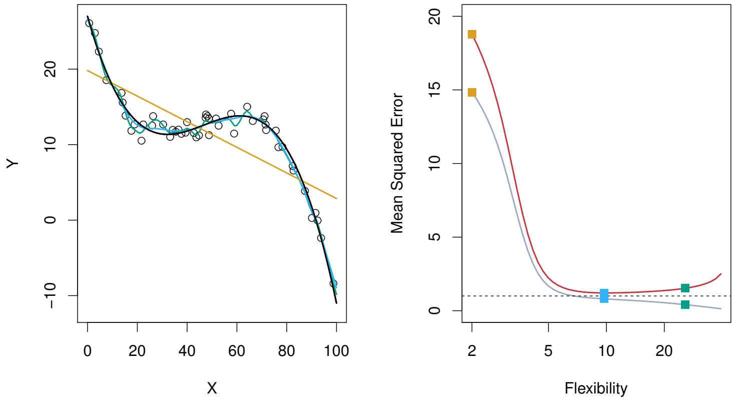


Figure: Simulated data: true $f$ (black), linear regression line (orange), and two smoothing splines (blue and green) (See ISLR p. 33)

## The Bias-Variance Tradeoff

We saw that the test MSE tends to be U-shaped

- The shape is the result of two competing forces
- More formally, given we draw a test observation $(x_0, y_0)$ from $P$, the expected squared prediction error is given by

$$
\begin{aligned}
E[(y_0 - \hat{f}(x_0))^2] = & \underbrace{Var[\epsilon]}_{\text{irreducible error of } f} \\
& + \underbrace{Var[\hat{f}(x_0)]}_{\text{variance of } \hat{f}} + \underbrace{[f(x_0) - E[\hat{f}(x_0)]]^2}_{\text{bias}^2 \text{ of } \hat{f}}
\end{aligned}
$$

- $E[(y_0 - \hat{f}(x_0))^2]$ is the expected squared error
  - i.e. the average test MSE if we repeatedly estimated $f$ using a large number of training sets, and tested each at $(x_0, y_0)$

A comparison of bias and variance in the three cases :

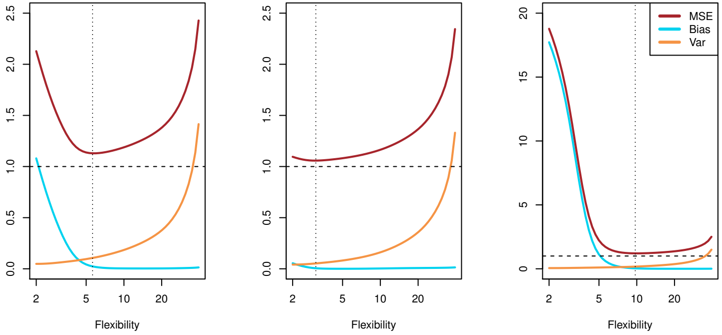


Figure: Squared bias (blue), variance (orange), $Var(\varepsilon)$ (dashed), and test MSE (red) for the three data sets (See ISLR p. 36)

## Python - Scikit-Learn

- Suppose we have a dataframe *df* consisting of two columns *a* and *b*.
- We wish to apply a model that predicts *b* based on *a*.
- To do this, we first define *y* as a vector and *X* as a matrix:

```
y = df[‘b’]
X = df[[‘a’]]
```

- To perform a linear regression using Scikit-Learn, we can do as follows:

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression().fit(X,y)
```

- Suppose we want to make a prediction at *X=[[1]]*:

```
pred = lr.predict([[1]])
```

- To perform e.g., 10NN regression in Scikit-Learn, we can do as follows:

```
from sklearn.neighbors import KNeighborsRegressor
knn = KNeighborsRegressor(n_neighbors=10).fit(X,y)
```

- Predictions can be made in the same manner as for linear regression.

## Your task - The Bias-Variance Tradeoff

- Write a script that generates a fictitious data set with 10,000 observations:

  ```
  n=10000
  df = pd.DataFrame(index=range(n))
  ```

- Generate a uniformally distributed random variable *age* with 18 and 70 as the lower and upper boundary, respectively:

  ```
  df['age'] = np.random.uniform(18,70,size=n)
  ```

- Generate a random variable *income* (yearly, in 1,000s) as a function of *age*:

  ```
  d['income']=(2*df['age']-0.002*df['age']**2+
  np.random.normal(0,10,size=n)
  ```

- Note that in this setting, $Y =$ `df['income']`.

Conceptual questions:

1. What is the CEF $f$?
2. What is the irreducible error?
3. What is $f(age)$ for $age = 50$?

## Your task - The Bias-Variance Tradeoff (cont'd)

Exercises:

1. Suppose you estimate $f$ by $\hat{f} = \hat{\beta}_0 + \hat{\beta}_1 * age$. Attempt to estimate the (squared) bias and variance of $\hat{f}$ at $age = 50$, based on a sample size of 10,000 by resampling the data 100 times.
2. Suppose you estimate $f$ by a 5NN regression. Attempt to estimate the (squared) bias and variance of $\hat{f}$ at $age = 50$, based on a sample size of 10,000 by resampling the data 100 times. Compare your result to 1. Which of the two has a higher bias and variance, respectively.

## 3. Classification

## Classification problem

Now $Y$ takes values in a finite and unordered set $\mathcal{Y}$.

- Suppoe $\mathcal{Y}$ contains $K$ elements numbered $1, ..., K$
- Let $p_k(x) = Pr(Y = k|X = x)$, for $k = 1, ..., K$
- Suppose we knew these conditional probabilities.
- Then, the *Bayes optimal classifier* at $x$ given by

$$f(x) = j \text{ if } p_j(x) = \max\{p_1(x), ..., p_K(x)\} \tag{10}$$

is optimal in the sense that it minimizes the expected misclassification rate:

$$E[\mathbb{1}_{\{Y \neq f(X)\}}] \tag{11}$$

## Estimating $f$ with $\mathcal{D}$ - KNN classification

Given our sample $\mathcal{D}$, we might attempt to apply a KNN once again to

1. approximate the conditional distribution of $Y$ given $X$

$$\hat{p}_j(x) = \frac{1}{K} \sum_{x_i \in N_K} \mathbb{1}_{\{y_i = j\}} \tag{12}$$

2. predict that a given observation belongs to the class with highest estimated probability

$$\hat{f}(x) = j \text{ if } \hat{p}_j(x) = \max\{\hat{p}_1(x), ..., \hat{p}_K(x)\} \tag{13}$$

Thus, KNN gives an estimate for the conditional probabilities as well as for the decision boundary

## Example

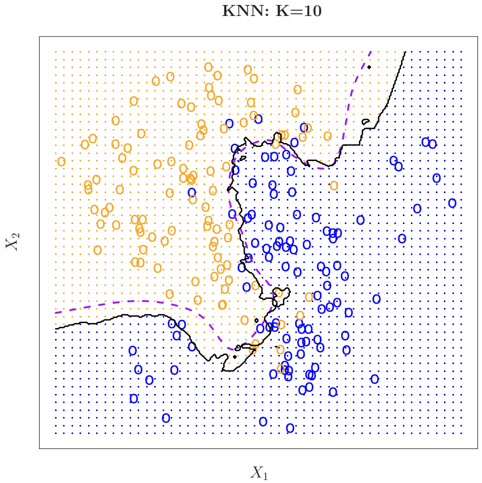


Figure: Bayes decision bounday (dashed), 10-NN decision boundary (black) (See ISLR p. 41)

# Example

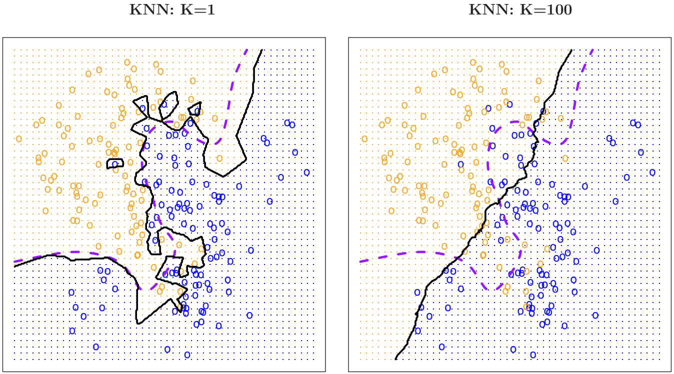


Figure: Bayes decision bounday (dashed), Left: 1-NN decision boundary (black), Right: 100-NN decision boundary (See ISLR p. 41)

# Example

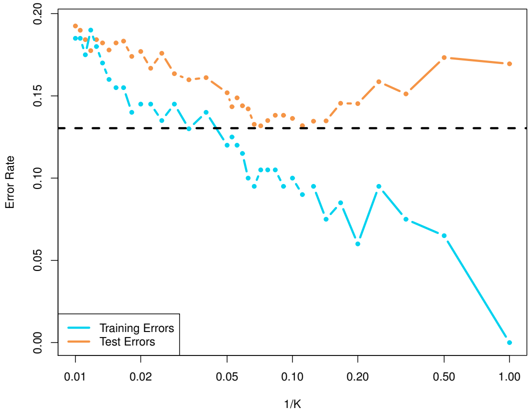


Figure: KNN training error rate (blue, 200 obs), test error rate (orange, 5,000 obs), Bayes error rate (dashed) (See ISLR p. 42)

## Estimating $f$ with $\mathcal{D}$ - Linear regression

- Suppose $\mathcal{Y} = \{0, 1\}$. In this case, we could consider using linear regression to estimate $p_1$.
- Since, in the population, we have $E[Y|X = x] = p_1(x) = Pr(Y = 1|X = x)$, regression seems to be good for this task.
- However, if the range of $X$ is not limited, we will see probability estimates below zero and above one.
- If we are only concerned with classification, then the linear regression might still be fine.
- However, in the case of multiple unordered classes, there is no straightforward way of applying linear regression.

## Estimating $f$ with $\mathcal{D}$ - Logistic regression

- To avoid getting probability estimates outside $[0, 1]$, we model $p(X)$ using a function which lies in the interval for all values of $X$.
- In the case of logistic regression, we use the logistic or sigmoid function.

$$\hat{p}_1(x) = \frac{exp(\hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_p x_p)}{1 + exp(\hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_p x_p)} \tag{14}$$

- To estimate (14) we use maximum likelihood.
- An additional benefit is that we can extend logistic regression to the case of multiclass classification.

# Example

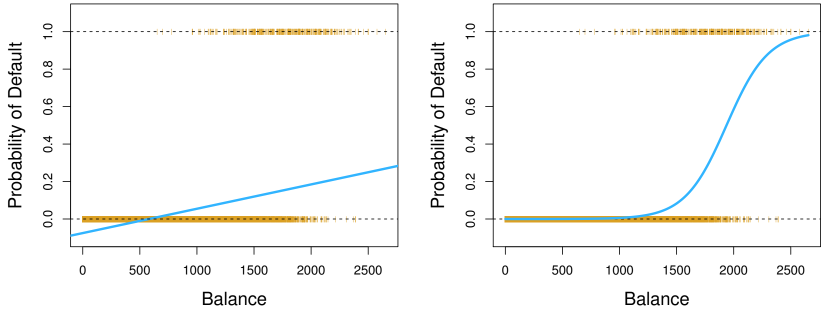


Figure: Left: Estimated probability of default using linear regression. Right: Same estimation using logistic regression (See ISLR p. 131)

- Some methods build structured models for $f(x)$.
  - e.g., support vector machines.
- Other methods, as we have seen here, build structured models for representing $p_k(x)$.
  - e.g., logistic regression.

## Python - Scikit-Learn

- Suppose we have a training set $(X_{train}, y_{train})$ and a test set $(X_{test}, y_{test})$.
- To fit a logistic regression on the training set using Scikit-Learn, we can do as follows:

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression().fit(X_train,y_train)
```

- We can estimate the misclassification rate on the test set as follows:

```
from sklearn.metrics import accuracy_score
preds = logreg.predict(X_test)
1-accuracy_score(y_test,preds)
```

- To make a probabilistic prediction at e.g., [[1]]:

```
pred = logreg.predict_proba([[1]])
```

- To perform e.g., 10NN classification in Scikit-Learn, we can do as follows:

```
from sklearn.neighbors import KNeighborsClassifier
knn =
KNeighborsClassifier(n_neighbors=10).fit(X_train,y_train)
```

- Making predictions and calculating the misclassification rate then follows in the same manner as for the logistic regression.

## Your task - Classification

- Import the *Default* data set:

```
path_to_df = 'https://raw.githubusercontent.com/
armoutihansen/EEMP2021/main/datasets/Default.csv'
df = pd.read_csv(path_to_df)
```

- Let $X$ consist only of the variable *balance* and let $y$ be a binary indicator that equals 1 if the customer defaulted on his debt and 0 otherwise:

```
X=df[['balance']]
y=(df['default']=='Yes')*1
```

Exercises:

1. Fit the linear regression: $\hat{f}(balance) = \hat{\beta}_0 + \hat{\beta}_1 * balance$ on $X, y$. What is the estimated probability of default given a balance of 200? Is this estimate sensible?
2. Now fit a logistic regression on $X, y$ and once again estimate the probability of default given a balance of 200. Is this a more sensible estimate?

## Your task - Classification (cont'd)

- Now split the sample into a training set and test set:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X,y,test_size=0.5)
```

3. Fit a 1NN classifier on the training set. Now calculate the misclassification rate on the training set and test set, respectively. Why is there a difference between these two estimates?
4. Fit a logistic regression on the training set and calculate the misclassification rate on the test set. Does this imply that the logistic regression is better or worse than the 1NN classifier in this problem? Why do you think that is?

## 4. Model selection and assessment

## Motivation

- Assessment of the general performance of our model is what we truly care about.
- The *generalization performance* refers to a model's prediction capability on independent test data.
    - Or, more generally, its capability in the population.
- If we know methods generalization performance, we
    1. can select the best of our considered models;
    2. know the best model's performance on unseen test data.

## Definitions

- Based on our choice of $\ell$, we would like to know our model's general performance
- The *generalization error* is the error over an independent sample
$$Err_{\mathcal{D}} = E[\ell(Y, \hat{f}(X))|\mathcal{D}] \tag{15}$$
i.e. the expected error of our model, given that $(X, Y)$ are drawn from their joint distribution $P$, conditional on being trained on $\mathcal{D}$.
- Estimation of $Err_{\mathcal{D}}$ is our goal, however it turns out that our tools for estimation better estimate the *expected generalization error*
$$Err = E[\ell(Y, \hat{f}(X))] = E[E[\ell(Y, \hat{f}(X))|\mathcal{D}]] = E[Err_{\mathcal{D}}] \tag{16}$$
i.e. the expected error over everything that is random – including the sample $\mathcal{D}$.

- The *training error* is simply the average loss in $\mathcal{D}$

$$\bar{err} = \frac{1}{N}\sum_{i=1}^{N} \ell(y_i, \hat{f}(x_i)) \qquad (17)$$

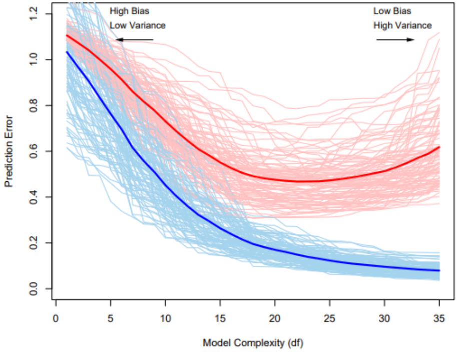


Figure: Light blue curves show $\bar{err}$ while light red curves show $Err_{\mathcal{D}}$ for 100 samples of size 50, as model complexity is increased. The solid curves show $E[\bar{err}]$ and $Err$, respectively (See ESL p. 220)

## Goals

- Often our model will have a hyper/tuning parameter or parameters.
  - e.g. # of neighbors for the KNN method.
- We wish to find the parameter(s) that minimizes the generalization error.
- However, we do in fact have two separate goals in mind:

1. **Model selection**: estimating the performance of a model with different hyper parameters in order to choose the best one.
2. **Model assessment**: having chosen a final model, estimating its prediction error (generalization error).

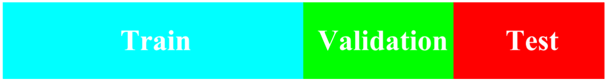
## Validation-set approach (without model selection)

**Idea:** Estimate the generalization error by holding out a subset of the known observations $\mathcal{D}$ from the fitting process, and then evaluate our model's predictions on those held out observations.

- Randomly divide the sample $\mathcal{D}$ into two parts: a training set $\mathcal{D}_{Tr}$ and a test (or validation) set $\mathcal{D}_{Te}$.
- The model is fit on the training set, and the fitted model is used to predict the responses for the observations in the validation set.
- The resulting test error provides an estimate of the expected generalization error.

## Validation-set approach (with model selection)

- Best approach: randomly divide $\mathcal{D}$ into three parts:



- We
  - fit our model with its hyper parameters on the training data $\mathcal{D}_{Tr}$ ($\sim$ 50%)
  - estimate error for model selection on the validation data $\mathcal{D}_{val}$ ($\sim$ 25%)
  - estimate the expected generalization error for our best model on the test data $\mathcal{D}_{Te}$ ($\sim$ 25%)
- If we use the test data for both model selection and assessment, we will usually underestimate the true test error of the best model.

## K-fold CV

- The error estimates of the validation approach may heavily depend on the splits of the sample.
- An approach of dealing with this is to instead use *K-fold Cross Validation* (CV).

1. Randomly split the data into $K$ roughly equal parts
2. For each $k = 1, ..., K$:
   - leave the $k$th part out and fit the model using the other $K - 1$ parts: $\hat{f}^{-k}(x)$
   - calculate the error of $\hat{f}^{-k}(x)$ on the held out $k$th part
3. Average the $k$ errors
   - Define the index function $\kappa : \{1, ..., N\} \rightarrow \{1, ..., K\}$ (allocating membership)
   - Then, the K-fold cross validation estimate of *Err* is:

$$CV(\hat{f}) = \frac{1}{N}\sum_{i=1}^{N} \ell(y_i, \hat{f}^{-\kappa(i)}(x_i)) \tag{18}$$

## What value should we choose for $K$?

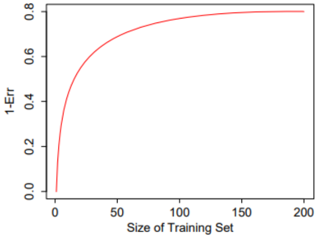


Figure: Hypothetical learning curve for a classifier on a given task: a plot of $1 - Err$ versus the size of the training set $N$ (See ESL p. 243)

- The optimal choice of $K$ depends on the slope of the learning curve
- In general, 5- or 10-fold CV are recommended as a good compromise between bias and variance.

## Choosing hyper parameters with CV

- Often we consider models with hyper parameter(s) $\alpha$
  - e.g. number of neighbors in KNN regression or classification.
- Denote by $\hat{f}^{-k}(x, \alpha)$ the $\alpha$th model fit with the $k$th part of data removed
- Then, we have

$$CV(\hat{f}, \alpha) = \frac{1}{N} \sum_{i=1}^{N} \ell(y_i, \hat{f}^{-\kappa(i)}(x_i, \alpha)) \tag{19}$$

- Usually, instead of choosing the $\hat{\alpha}$ which minimizes (19), we apply the "one standard error rule":

*choose the most parsimonious model with error no more than one standard error above the best error*

## Example - Classification

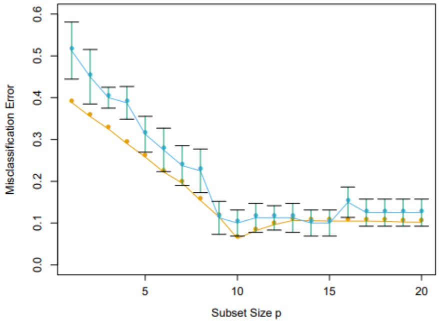


Figure: Test error (orange) and 10-fold CV curve (blue) from a single training set (See ESL p. 244)

- Which model would be chosen here?

## CV for both model selection and assessment

- Recall that in the validation set approach:
  - If we use the test data for both model selection and assessment, we will risk underestimating the true test error of the best model.
- The same holds with cross validation.
- To deal with this problem, two approaches are used in practice:
  1. Leave out a test (or validation) set, use CV for model selection and estimate the generalization error on the test set.
  2. Nested K-fold CV: Model selection on inner CV and model assessment on outer cv.
- Here we opt for option 1.

## Python - Scikit-Learn

- Suppose we have a training set $(X_{train}, y_{train})$ and a test set $(X_{test}, y_{test})$.
- Suppose we wish to perform 5-fold CV to determine the optimal number of neighbors $(K)$ in a KNN regression. For any $K$, we can calculate the CV error as follows:

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import cross_val_score
knn = KNeighborsRegressor(n_neighbors=k)
cv = cross_val_score(knn, X_train, y_train, cv=5,
scoring='neg_mean_squared_error')*-1
```

- *cv* contains 5 error estimates, one for each fold. We can then use numpy to calculate the average error:

```
np.mean(cv)
```

- We will do this procedures for all $K$s that we consider (e.g., in a *for* loop) and then select the $K$ with the lowest average CV error.

## Python - Scikit-Learn (cont'd)

- After having found the optimal $K$, our task is normally to estimate the expected generalization error. This is done by fitting the optimal model on the whole training set and then evaluating its predictions on the test set. In a regression problem where we use the squared error, we can do as follows:

```
from sklearn.metrics import mean_squared_error as MSE
knn = KNeighborsRegressor(n_neighbors=best_k).fit(X_train,
y_train)
preds = knn.predict(X_test)
MSE(preds,y_test)
```

## Your task - Cross Validation

- Write a script that generates a fictitious data set with 10,000 observations:

  ```
  n=10000
  df = pd.DataFrame(index=range(n))
  ```

- Generate a uniformally distributed random variable *age* with 18 and 70 as the lower and upper boundary, respectively:

  ```
  df[‘age’] = np.random.uniform(18,70,size=n)
  ```

- Generate a random variable *income* (yearly, in 1,000s) as a function of *age*:

  ```
  d[‘income’]=(2*df[‘age’]-0.002*df[‘age’]**2+
  np.random.normal(0,10,size=n)
  ```

- Note that in this setting, $Y =$ `df[‘income’]`.

Exercises:

1. Split the data into a training set and a test set, where the test set consists of 25% of the observations.

## Your task - Cross Validation (cont'd)

Your task now is to perform model selection on the training set using 5-fold CV in order to find the optimal # of neighbors in a KNN regression, where $K \in [10, 11, \ldots, 100]$.

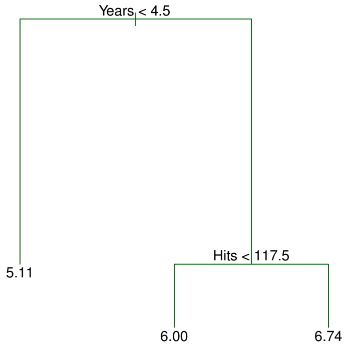
2. Calculate and store the average CV error for each $K$ and plot these errors using matplotlib.
3. Based on these CV errors, which $K$ is optimal?
4. Fit the KNN regression with the optimal K on the training set and estimate its expected generalization error on the test set.

## 5. Tree-based methods

## Introduction to Decision Trees

- Decision trees (DTs) are versatile ML algorithms that can perform both classification and regression tasks.
- a DT is a tree-based method – these involve dividing the inputs into a number of simple regions.
- After this split, we typically make predictions based on the mean or mode output value in the regions.
- The set of splitting rules used to divide the input space can be summarized in a tree.
- DTs are also the fundamental components of *Random Forests* and other ensemble methods, which are among the most powerful ML algorithms available.

# Example - DT regression



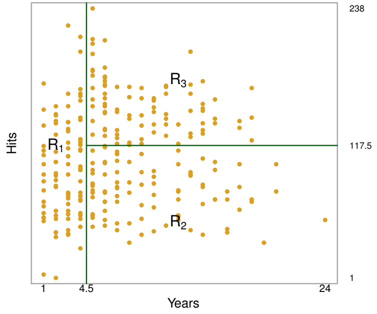


Figure: Right: A regression tree for predicting the log salary of a baseball player, based on years in major league and number of hits made in the previous year- Left: The three-region division on the inputs from the regression tree (See ISLR pp. 304–5)

## Terminology

- The regions (e.g., $R_1, R_2, R_3$) are known as *terminal nodes* or *leaves* of the three
- The points along the three where the inputs are split are known as *internal nodes* (e.g., Years$<$4.5 and Hits$<$117.5)
- The initial node (Years$<$4.5) is also sometimes referred to as the *root node*
- The segments of the tree that connects the nodes are *branches*
- Note: The tree is usually displayed upside down

## Prediction via splitting

- How do we build regression trees?
- Roughly speaking, there are two steps:
  1. We divide the inputs $X_1, \ldots, X_p$ into $J$ distinct and non-overlapping regions, $R_1, \ldots, R_J$
  2. For every observation that lies in $R_j$, we predict the mean of the output values from the training observations in $R_j$
- But how do we divide $X_1, \ldots, X_p$ into $R_1, \ldots, R_J$?
  - We find $R_1, \ldots, R_J$ that minimizes

$$\sum_{j=1}^{J} \sum_{i \in R_j} \left(y_i - \hat{y}_{R_j}\right)^2 \tag{20}$$

where $\hat{y}_{R_j}$ is the mean response for the training observations within the $j$th region

- Unfortunately, that division approach is computational infeasible (it is a NP-Complete problem)
- Instead, one usually apply a *top-down, greedy* approach known as *recursive binary splitting*
- We choose the input $X_j$ and the cutpoint $s$ such that the regions $\{X|X_j < s\}$ and $\{X|X_j \geq s\}$ lead to the largest possible reduction in the sum of squared errors.
- That is, we seek $j, s$ that define the half-spaces

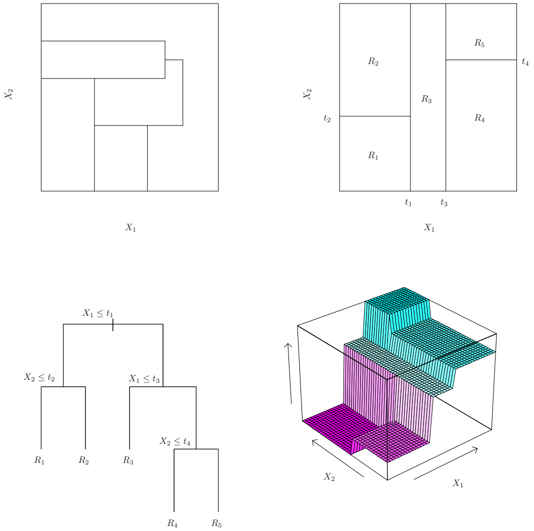
$$R_1(j,s) = \{X|X_j < s\}, R_2(j,s) = \{X|X_j \geq s\} \tag{21}$$

  such that we minimize

$$\sum_{i:x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2 \tag{22}$$

- We iterate the process of splitting the (sub)spaces until a stopping criterion is reached (e.g. *max depth* or *min observations per leaf*)
- If no such criterion is given, the process will continue until no improvement can be made

# Example


$X_2$
$X_1$
$R_2$
$R_3$
$R_5$
$R_4$
$R_1$
$t_2$
$t_4$
$t_1$
$t_3$
$X_1 \leq t_1$
$X_2 \leq t_2$
$X_1 \leq t_3$
$X_2 \leq t_4$
$R_1$
$R_2$
$R_3$
$R_4$
$R_5$
$X_2$
$X_1$

## Example – Simple regression with/without restrictions

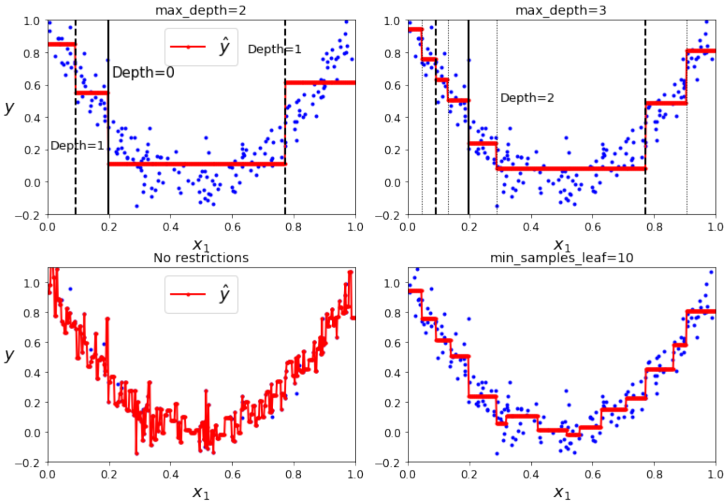


Figure: See HOML p. 178

## Decision Trees - Classification

- A Decision Tree for classification is build the same way as with regression
- We apply the recursive binary splitting algorithm to sequentially segment our feature space
- Naturally, we will no longer be minimizing the sum of squared errors. We could use the misclassification rate:

  $$E_m = 1 - \max_k(\hat{p}_{mk}) \tag{23}$$

  where $\hat{p}_{mk}$ is the proportion of observations in the $m$th region belonging to the $k$th class – our estimate of $p_{mk}$
- However, it turns out that $E_m$ is suboptimal for tree-growing – We rather need a measure of node impurity

## Gini index – Node impurity

- In practice we use the *Gini index* to build trees
- The *Gini index* value in region $R_m$ is a measure of total variance across the $K$ classes and is given by

$$G_m = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) \tag{24}$$

  which is often referred to as *node impurity* – a small value indicates that a node contains predominantly observations from a single class
- Thus, when deciding to split $X$, we seek $j$ and $s$ that define half-spaces

$$R_1(j,s) = \{X|X_j < s\}, R_2(j,s) = \{X|X_j \geq s\} \tag{25}$$

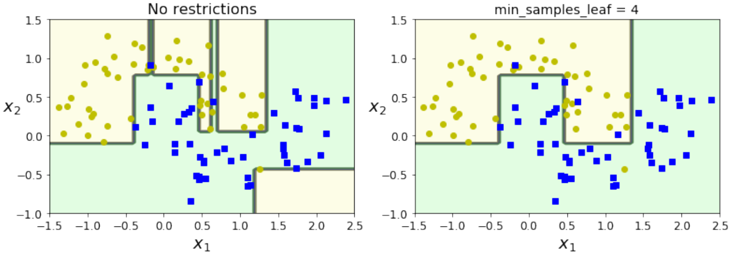
  such that we minimize the weighted sum of node impurity:

$$\frac{N_1}{N} G_1 + \frac{N_2}{N} G_2 \tag{26}$$

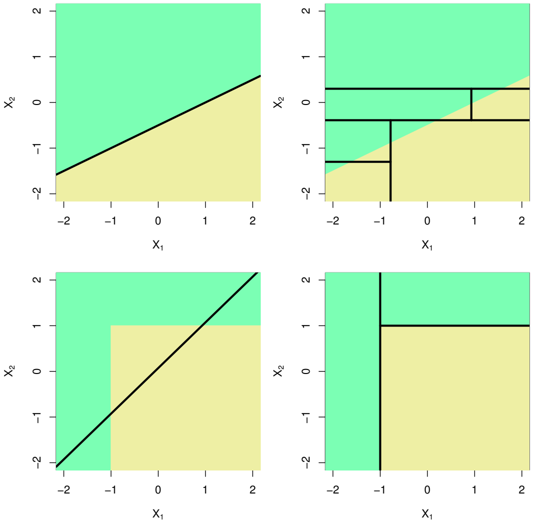
- If we cannot gain purity in making this split, the preceding node is declared as a terminal node/leaf

# Example – Classification with/without restrictions

- As with regression trees, we risk overfitting if we do not specify terminal criteria



# Trees vs. Linear Models



## Instability

Decision Trees are:

- simple and easy to interpret, as well as versatile.

However,

- they are very sensitive to small variations in the data;
    - removing one observation may substantially change the tree.
    - thus, they suffer from high variance.
- The decision boundaries have to be orthogonal to the feature axis.
    - they are sensitive to the rotation of the data.
- In many applications, their predictive ability is below that of other well-known ML methods.
- However, as we shall see, we can use DTs as building block to construct more powerful methods.

## Python - Scikit-Learn

- Suppose we have a training set $(X_{train}, y_{train})$ and a test set $(X_{test}, y_{test})$.
- Suppose we wish to fit a decision tree on the training set. If it is a regression problem, we can do as follows:

```
from sklearn.tree import DecisionTreeRegressor
dt = DecisionTreeRegressor(max_depth=2).fit(X_train,y_train)
preds = dt.predict(X_test)
from sklearn.metrics import mean_squared_error as MSE
print(MSE(y_test,preds)
```

- To plot the tree, we can do as follows:

```
from sklearn import tree
tree.plot_tree(dt, feature_names=X_train.columns)
```

- The procedure is similar for classification. Here, we use a *DecisionTreeClassifier* and *accuracy_score*.

## Ensemble Learning

- Consider the following simplified version of the *Condorcet Jury Theorem*:
- Suppose there are $N$ voters on a jury
- For simplicity, let each voter's probability of being correct be $p$
- Furthermore, let $M$ be the probability that the majority is correct
- Then, under certain conditions,

$$p > \frac{1}{2} \Rightarrow M > p \tag{27}$$

and

$$\lim_{N \to \infty} M \to 1 \tag{28}$$

- Thus, we can think of training a set of models on the training data to improve our prediction accuracy – This is (roughly) what ensemble learning methods are about.

## Bagging (Bootstrap Aggregating)

- The effectiveness of using a set of models instead of one is the better the less they are correlated. In turn this lowers the variance of our estimate at a point $x_0$
  - As an illustration, suppose we have $N$ independent observations $Z_1, \ldots Z_n$ with variance $\sigma^2$
  - Then the variance of the mean $\bar{Z}$ is given by $\sigma^2/N$

- One way to decrease the variance would be to train a set of models from the same method (e.g. a Decision Tree) on $B$ different training samples drawn from the population

- We then train each model on each of the sets, and then predict according to

$$\hat{f}_{avg}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(x) \tag{29}$$

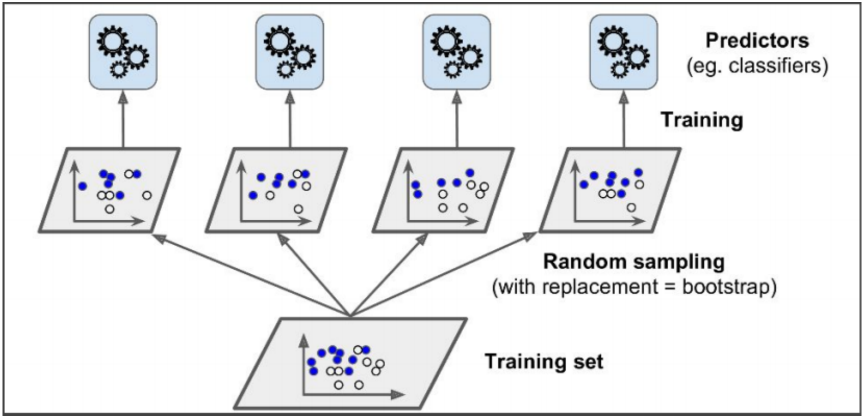
  in the regression setting, and by majority voting in the classification setting.

- Usually, we do not have access to multiple training sets
- Instead, we could split up our training set into $B$ partitions and train a model on each set – This is called *pasting*
- Alternatively, we could sample $B$ sets, with replacement, from our training set to create $B$ bootstrap samples with the same size as our training set
- Then we train a method on each set and then predict according to

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{b}(x) \tag{30}$$

  in the regression setting, and by majority voting in the classification setting
- With Decision Trees, we grow each tree deep and leave them unpruned
- Thus, they have high variance, but low bias – Averaging then lowers the variance

## Example – Bagging



# Example – Bagging

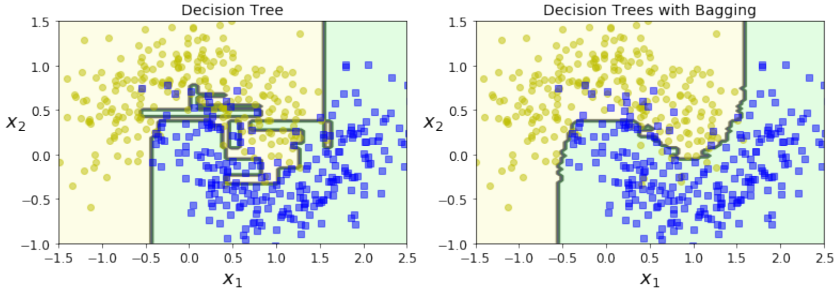


Figure: A single Decision Tree vs. a bagging ensemble of 500 trees (See HOML p. 189)

## Random Forest

- A Random Forest is an ensemble of Decision Trees, generally trained via the bagging method
- However, it provides an improvement over bagged trees by using a small tweak that further decorrelates the trees
  - Recall that ensemble methods increase in accuracy as correlation decreases
- The trick: each time a split in a tree is considered, a random sample of $m < p$ predictors are considered
- Thus, instead of searching for the best feature when splitting a node, it searches for the best among the random sample of $m$ predictors
- The result is greater tree diversity which trades of a higher bias with lower variance compared to bagged trees
- Often, one sets $m = \lceil \sqrt{p} \rceil$

## Python - Scikit-Learn

- Suppose we have a training set $(X_{train}, y_{train})$ and a test set $(X_{test}, y_{test})$.
- Suppose we wish to fit a Random Forest on the training set. If it is a regression problem, we can do as follows:

  ```
  from sklearn.ensemble import RandomForestRegressor
  ```

- Then, we initialize the object specifying our desired hyper parameters:

  ```
  rf = RandomForestRegressor(n_estimators=1000,
  max_depth=None, max_features='sqrt', random_state=181,
  n_jobs=-1)
  ```

- Then we can fit and evaluate the predictions of our model:

  ```
  rf.fit(X_train,y_train)
  preds = rf.predict(X_test)
  from sklearn.metrics import mean_squared_error as MSE
  print(MSE(y_test,preds))
  ```

- The procedure is similar for classification. Here, we use a *RandomForestClassifier* and *accuracy_score*.

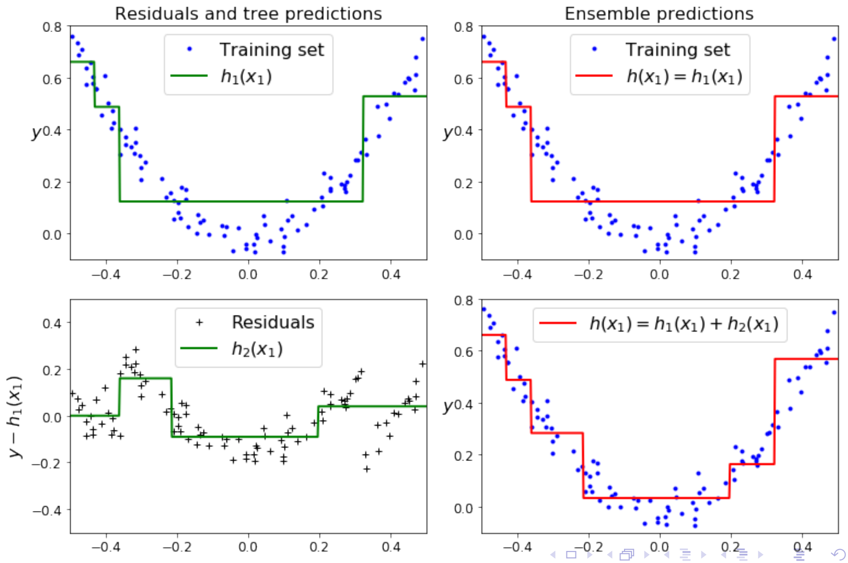
## Boosting

- Boosting is an ensemble method that can combine several *weak learners* into one *strong learner*
    - Weak learner: Computationally simple model, that performs slightly above chance
- Idea: Train models sequentially on the (modified) training data, where each succeeding model tries to correct its predecessor
- In general, models that learn slowly tend to perform well

## Gradient Boosting

- Gradient Boosting works by sequentially adding models to an ensemble.
- Each of the models attempts to correct for its predecessor.
- That is, if $\hat{f}_1$ is trained on $(X, y)$, then $\hat{f}_2$ is trained on $(X, r_1)$ where $r_1 = y - \hat{f}_1$, and so on.
- One may then stop the collection if no further improvement is observed for several rounds, or once a certain number of trees have been reached

# Gradient Boosting – Example



## Python - Scikit-Learn

- Suppose we have a training set $(X_{train}, y_{train})$ and a test set $(X_{test}, y_{test})$.
- Suppose we wish to fit a Gradient Boosting regressor on the training set. We can do as follows:

```
from sklearn.ensemble import GradientBoostingRegressor
```

- Then, we initialize the object specifying our desired hyper parameters:

```
gb = GradientBoostingRegressor(n_estimators=100,
learning_rate=0.1, max_depth=3, random_state=181, n_jobs=-1)
```

- Then we can fit and evaluate the predictions of our model:

```
gb.fit(X_train,y_train)
preds = gb.predict(X_test)
from sklearn.metrics import mean_squared_error as MSE
print(MSE(y_test,preds)
```

- The procedure is similar for classification. Here, we use a *GradientBoostingClassifier* and *accuracy_score*.

## Python - Scikit-Learn

- Suppose we have a training set $(X_{train}, y_{train})$ and a test set $(X_{test}, y_{test})$.
- When we perform model selection on the training set using ensemble methods (e.g. boosting and bagging), we usually have a number of hyper parameters to choose from.
- For the random forest, we usually care most about (i) the number of estimators and (ii) the number of inputs available for consideration when a decision tree in the ensemble makes a split.
- For the gradient boosting, we usually care most about (i) the number of estimators and (ii) the maximum depth of each tree in the ensemble.
- To perform model selection by CV in sklearn, we can do as follows:

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV
gb = GradientBoostingRegressor()
param_grid = {'n_estimators':np.arange(100,500),
'max_depth':np.arange(1,2)}
cv = GridSearchCV(gb, param_grid,
scoring='neg_mean_squared_error', cv=5,
n_jobs=-1).fit(X_train,y_train)
```

## Python - Scikit-Learn (cont'd)

- To see the best model:

```
print(cv.best_estimator)
```

- To estimate the generalization error:

```
preds = cv.best_estimator_.predict(X_test
from sklearn.metrics import mean_squared_error as MSE
print(MSE(y_test,preds)
```

- Then, we initialize the object specifying our desired hyper parameters:
- The procedure is similar for the random forest. Here we use *RandomForestRegressor*.
- The procedure is also similar for classification. Here we use *scoring='accuracy'*.

## Python - Scikit-Learn

- If we optimize over all of these hyper parameters using CV, the process might take too long.
- Solution: Instead of trying out all possible combinations of the hyper parameters, we randomly choose a sample of parameter candidates.
- Hence, we do not fit each possible candidate model, but fit *n* models consisting of randomly chosen hyper paramaters from our defined parameter grid.

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import RandomizedSearchCV
gb = GradientBoostingRegressor()
param_grid = {'n_estimators':np.arange(100,500),
'max_depth':np.arange(1,2)}
cv = RandomizedSearchCV(gb, param_grid, n_iter=100,
scoring='neg_mean_squared_error', cv=5,
n_jobs=-1).fit(X_train,y_train)
```

## Your task

1. Import the *default* data into a dataframe *df* and convert the column *student* into a binary variable using:
`df['student'] = pd.get_dummies(df['student'], drop_first=True)`
2. Define $y$ to be the column *default* and let $X$ consist of the columns *balance, income, student*.
3. Divide $X, y$ into a training set and a test set, where the test set consists of 25% of the observations.
4. Fit a decision tree classifier on the training set, where you limit the maximum depth to be 3. Plot the tree and estimate the expected misclassification rate using the test set.
5. Using 5-fold CV on the training set, attempt to find the optimal random forest classifier, using an appropriate parameter grid and random search. Estimate the expected misclassification rate of the optimal random forest using the test set.

## 6. Model interpretation

## Model interpretation

How can we interpret our estimated models?

- We cannot conclude causal effects, rather our models may reveal how the conditional expectation (in a regression problem) and the class conditional probabilities (in a classification problem) of the dependent variable varies with the inputs.
- We could simply check the correlation coefficients, but this is only a linear measure.
- We could check the $\beta$ coefficients of a linear regression, but the model may not be very predictive and the measure is also limited.

## Python - Scikit-Learn

- To access the beta coefficients in a linear regression *lr*, we can do as follows:
- To get the intercept ($\hat{\beta}_0$):

  `print(lr.intercept_)`

  `To get the remaining coefficients (ordered as in X_train):`

  `print(lr.coef_)`

If we fit a Decision Tree, we can see how the feature space is segmented and how the model predicts.

- This should give an idea about the association between variables.
- However, single Decision Trees rarely provide the best fit.

Usually, more complex models such as Random Forests are better predictors, but interpretation becomes more difficult. Here we will cover two methods that allow us to retain some interpretability:

1. Partial dependence: Shows how a single or pair of input(s) vary with the output.
2. Variable importance: Shows which of the inputs were important for predicting the output.

## Partial dependence

Partial dependence (PDP) tell us about how the dependent variable and one of the predictors interact.

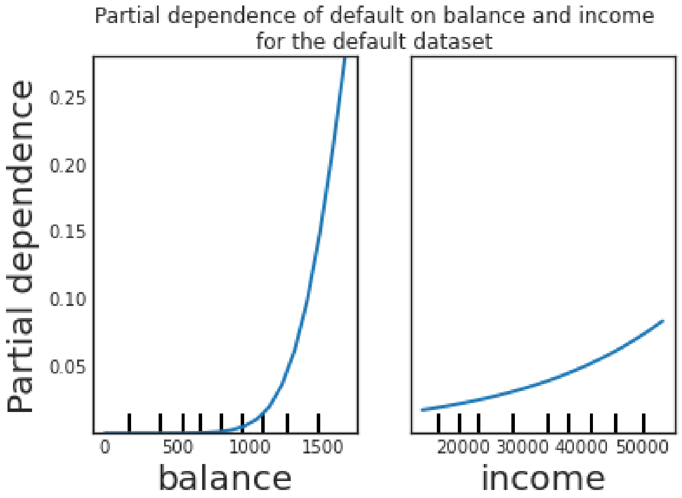
- Note: Partial dependence assumes that the predictors are independent of each other. This is often not true, so we must be careful when interpreting the results.

We can perform one-way partial dependence and two-way partial dependence in Scikit-Learn.

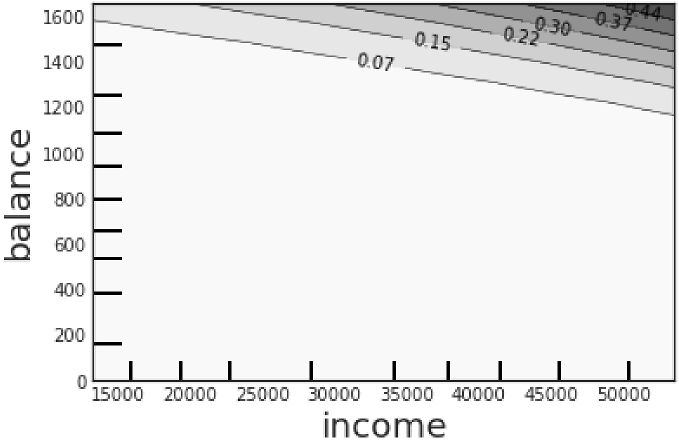
- One-way PDP tells us how the dependent variable interact with the considered input keeping all other inputs constant.

- Two-way PDP tells us how the dependent variable jointly interact with the two considered inputs keeping all other inputs constant.

## Example – One-way PDP



## Example – Two-way PDP



## Python - Scikit-Learn

- Suppose we have fitted a Random Forest classifier *rf* on the *default* data and want to plot one-way and two-way partial dependence. We can do as follows:

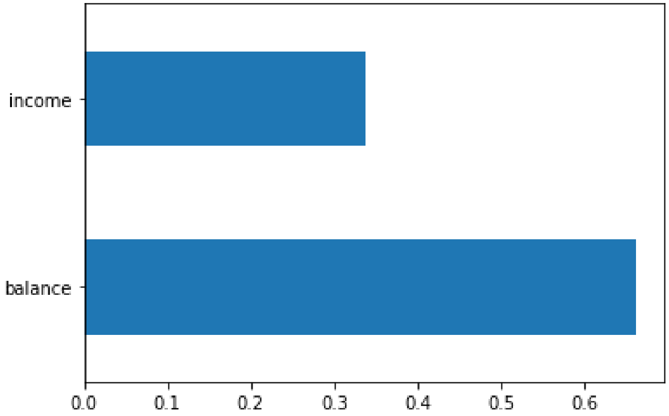
  ```
  from sklearn.inspection import PartialDependenceDisplay
  features = ['balance', 'income', ('balance', 'income')]
  PartialDependenceDisplay.from_estimator(rf, X_train,
  features, kind='average');
  ```

- The procedure is similar for the gradient boosting and for regression problems.

## Variable Importance

- Alternatively, it is possible to obtain an overall summary of the importance of each input using the sum of squared errors (regression) or the Gini index (classification).
- For example, for bagged regression trees, we can record the total amount that the error decreases due to splits over a given input, averaged over all $B$ trees.
- If the value is relatively high for a given feature, then this feature is important.

## Example – Variable importance



## Python - Scikit-Learn

- Suppose we have fitted a Random Forest classifier *rf* on the *default* data and want to plot the feature importance. We can do as follows:

  ```
  import matplotlib.pyplot as plt
  fig = pd.Series(rf.feature_importances_,
  index=X_train.columns).plot(kind='barh')
  plt.show();
  ```

- The procedure is similar for the gradient boosting and for regression problems.

## References

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An introduction to statistical learning (Vol. 112). **Chapter 2,4,5**

Friedman, J., Hastie, T., & Tibshirani, R. (2001). The elements of statistical learning (Vol. 1, No. 10). **Chapters 2,7**