1 PQHS 471 Notes Week 1

1.1 Week 1 Day 1

1.1.1 Syllabus

Course goals: Understand the models, intuition, statistical underpinnings, strengths and weaknesses, assumptions and trade-offs of various machine learning approaches. Can run R/Python to analyze data. Technical details such as optimization algorithms and theoretical properties are not of primary interest.

Books, participation, homework, exams, etc.

ISLR	http://www-bcf.usc.edu/~gareth/ISL/
HOML	http://proquest.safaribooksonline.com/9781491962282?uicode=ohlink
NNDL	http://neuralnetworksanddeeplearning.com/
ESL	http://www.stanford.edu/~hastie/ElemStatLearn/
CASI	https://web.stanford.edu/~hastie/CASI/
DL	http://www.deeplearningbook.org/
MMDS	http://www.mmds.org/
R4DS	http://r4ds.had.co.nz

Prerequisites: Calculus, linear algebra, and some exposure to statistics (minimum EPBI/PQHS 431). Familiarity with matrices (such as explained in ISLR Chapter 1; DL Chapter 2).

Languages/environments: R, Python, git/GitHub (highly recommended), Linux (recommended)

IDEs: RStudio (https://www.rstudio.com/) (recommended), spyder (comes with Anaconda)

Useful websites: https://www.r-bloggers.com/, https://www.kaggle.com/

1.1.2 Overview of Data Science

Overlapping fields:

- Statistics: Supposedly synonymous to "data science". It covers all aspects of data-oriented activities: data collection, analysis, and interpretation. Historically it focuses on inference, including parameter estimation, hypothesis testing, and decision making. It studies function estimation (aka prediction) from the perspective of smoothing instead of prediction accuracy. Historically the field is confined to small datasets and limited by lack of computing power, and has thus developed certain tastes in its approaches: It prefers interpretable models and uses probabilistic thinking. These have made the field shy away from prediction-oriented tasks and analyses of large messy data.
- Computer Science:
  - Machine learning: Similar to Statistics, often using more colorful language. Supervised learning heavily focuses on prediction. Methods are often algorithmic rather than probabilistic.
  - Data mining: Detection of unknown relationships in large data, sometimes with the help of visualization. In Statistics, it is called exporatory data analysis (EDA), a notion introduced in the era of small data. DM problems are often less well defined than those in ML.
  - Artificial intelligence: Has much broader goals than data-oriented tasks. Modern "AI" successes are due to large training datasets and advances in computation. But the successes are task-specific and the algorithms learn in a cumbersome way, unlike humans.
- Other names: Biostatistics, Statistical learning, Pattern recognition, Predictive/data analytics, Business intelligence, etc.

Data science = Statistics/ML/DM + implementation with large datasets

Related fields:

- Mathematical optimization
- Informatics: Focused on data capture/retrieval, feature retrieval
- Natural language processing

Big data: Volume, Velocity and Variety. (data vs. information vs. knowledge)

1.1.3 R Packages and Datasets

Package installation: One can install R packges from CRAN using the R function `install.packages()`. For example, to install the ISLR package for our textbook, use `install.packages("ISLR")`. The default directory for package installation is the first element of `.libPaths()`. To install a package into a different directory, one can specify the directory using the argument `lib=`.

Alternatively one can download the .tar.gz file for a package and install it on the command line (e.g., on Linux, `R CMD INSTALL ISLR_1.2.tar.gz`). On my computers, which run Ubuntu Linux 16.04, I install packages into a system directory by starting an R session with `sudo -i R`.

If a package had been installed in the past, you may want to check its version to see if it is up to date.

```
packageVersion("ISLR")  ## installed version for a package; v1.2 (01/03/2018)
available.packages()["ISLR", "Version"]  ## latest version on CRAN
packageDescription("ISLR")  ## description of a package
```

`sessionInfo()` and `devtools::session_info()` give information about the current R session and loaded libraries.

```
library(ISLR)
sessionInfo()  ## check all relevant version information
devtools::session_info()  ## another version
```

The following libraries are used in the R Labs of ISLR. A simple way to install/update all these packages is:

```
install.packages(c("ISLR", "MASS", "ROCR", "akima", "boot", "car",
                   "class", "e1071" , "gam", "gbm", "glmnet", "leaps",
                   "pls", "randomForest", "tree"))
```

A few other packages (e.g., `caret`, `xgboost`) will be used in class demonstration. You can install them when you need them. Note that `tree` is not as useful as `rpart`.

Loading a library: use `library()` or `require()`. To unload a package, use `detach()`. By default, `library()` and `require()` search the `.libPaths()` to find the named package to load. Note that `library()` without any argument will list all installed R packages.

```
library(ISLR)
search()  ## show search path
ls("package:ISLR")  ## list all objects in ISLR (see Table 1.1)
```

Although objects in a library can be accessed without loading the library (by using `::`; e.g., `ISLR::Wage`), loading a library makes it a lot more convenient.

In addition to the datasets in the ISLR package, the book ISLR also uses these datasets: `Boston` in the `MASS` library, `USArrests` in base R, and `Advertising`, `Heart`, `Income1`, and `Income2` in .csv format from the book website. Some datasets are simulated: `Carseats`, `Credit`, `Default`, `Portfolio`, `Income1`, and `Income2`.

Exploring and using a dataset: Most datasets should have class `data.frame`. A data frame is a list in which all elements are vectors of the same length. Using the `Wage` dataset as an example:

```
?Wage  ## show its documentation
names(Wage); dim(Wage)  ## variable names and dataset dimensions
str(Wage)  ## variable structures
head(Wage)  ## first few observations
```

To access variables in a data frame, use `$` or `with()` for a single command, and `attach()` for multiple commands.

```
levels(Wage$education)
plot(Wage$age, Wage$wage)
with(Wage, plot(age, wage))  ## same plot; ensures all variables are from the same dataset
```

Below is an example of using `attach()`. Using `attach()` can lead to confusion and unexpected consequences. **Remember to `detach()`** at the end to avoid problems.

```
#### Figure 1.1, Wage data (also see ISLR Section 7.8)
attach(Wage)  ## use search() to check
par(mfrow=c(1,3))

## Fig 1.1. first panel
plot(age, wage, xlab='Age', ylab='Wage', col='grey')
agegrid = seq(10,80,1)
lines(agegrid, predict(loess(wage ~ age), agegrid), col="blue", lwd=3)

## Fig 1.1 second panel
plot(year, wage, xlab='Year', ylab='Wage', col='grey')
abline(lm(wage ~ year), col = "blue", lwd=3)

## Fig 1.1 third panel; 'education' is a factor
plot(education, wage, xlab='Education Level', ylab='Wage', col=2:6, xaxt='n')
axis(1, at=1:5, labels=1:5)

detach()  ## use search() to check
```

Functions: To learn the syntax and arguments of a function, use `?` (e.g., `?plot`). `?` is a shortcut for `help()`.

1.1.4 Git Introduction

An easy tutorial: http://product.hubspot.com/blog/git-and-github-tutorial-for-beginners
Broman's tutorial: http://kbroman.org/github_tutorial/
Book *Pro Git*: https://git-scm.com/book/en/v2

This git introduction is for Linux. Create a local repository (repo), a working directory.

```
mkdir TestGit
cd TestGit
git init  ## initialize the repo (creating .git/ directory)
git status  ## check file status (of current branch)
## Now create some files under TestGit/
git status
```

Staging environment: Files can be tracked and untracked. To stage a file (i.e., add a file to the staging environment/index), use `git add`. To untrack a file, use `git rm`.

```
git add file1 file2
git status
```

Every staged file (called a **blob**, binary large object) has a hashed copy under `.git/objects/` and a hash tag (a hex string of length 40). The file is named after its hash tag, with the first 2 hex symbols as directory name and the remaining 38 hex symbols as filename.

Commit: A commit is a snapshot of a branch. Commits make up the essence of a project and allow you to go back to the state of a project at any commit. To create a commit, use `git commit`. The first commit creates a branch called `master`.

```
git commit -m "my 1st commit!"
git config --global user.name "Joe Smith"  ## info added to ~/.gitconfig
rm -f ~/.gitconfig
git config user.name "Joe Smith"  ## info added to .git/config
git config user.email "me@abc.com"
git commit -m "my 1st commit!"
```

Every commit creates two hashed files under `.git/objects/`, one containing a **tree** of blobs and the other containing the hash tag of the tree and the commit message. `git commit --amend -m` allows amending with a new message. `git commit --amend` pops up the default editor for editing the message file `.git/COMMIT_EDITMSG`. To add an annotation tag (e.g. version number) to a commit, use `git tag`. A commit can be referred to with either an annotation tag or a hash tag (using the first few, at least 4, hex symbols as long as it is unique to the repository). To see the tree of a commit, use `git ls-tree [tag]`. A commit also contains a pointer to the previous commit. This chain of pointers is very useful for merging.

For those who are curious about the hashed files under `.git/objects/`: To show the type of a hashed file (e.g., blob, tree, commit, tag), use `git cat-file -t [tag]`. To show the content of a hashed file (e.g., file content, list of blobs, tree hash and commit message, tag info), use `git cat-file -p [tag]`.

Branch: A good work habit is to create a branch to work on; once the work is done, merge the changes in the branch to the master. To create a branch, use `git branch [branchname]`. To switch to a branch, use `git checkout [branchname]`. To create a branch and switch to it, one can use a single command `git checkout -b [branchname]`. When a new branch is created without specifying a start point, its hash tag is the same as the current branch (info in `.git/HEAD`). Branch hash tags are in `.git/refs/heads/`.

```
git branch  ## check branch status; current branch is starred
git checkout -b branch1  ##
git branch
```

Now change some files. `git diff` is to go through the list of files in the current branch, compare the version in the working directory with the last staged version, not with the version in the last commit.

```
git diff  ## Or, git diff [filename(s)]
git status
git add file2
git diff
git status
git commit -m "my 2nd commit"  ## create a commit for the current branch
```

To check the history of all the commits, use

```
git log --oneline --decorate --graph --all
```

To merge the content from a branch to the current branch:

```
git checkout master
git merge branch1  ## merge the content of branch1 into current branch, master
```

Note that switching to a branch triggers copying branch files to the working directory. To delete a local branch, use `git branch -d <branchname>`. To delete a remote branch, use `git push <remotename> --delete <branchname>`.

Github repository: Create a new, empty repository on github.com. Mine is called "FirstTest.git". Then on local machine, add a remote repo named `origin`. Information is added to `.git/config`. Version 1 below requires entering password for every push; Version 2 uses ssh key authentication to avoid this. To copy a branch to a remote repo, use `git push`. The hash tag for a branch of a remote repo is saved under `.git/refs/remotes/`. To copy a branch from a remote repo, use `git pull`, which is a `git fetch` followed by a `git merge`.

```
git remote add origin https://github.com/cxl791/FirstTest.git  ## ver 1
git remote remove origin
git remote add origin git@github.com:cxl791/FirstTest.git  ## ver 2
git remote -v  ## check remote repos
```

```
git push -u origin master
git push origin branch1

git pull origin master  ## copy the master branch from github.com to local repo
git log
git status
git checkout master
git pull
```

To clone a remote repository to your computer, use `git clone`. Make sure you are not in any repository when doing this. For example, to clone github user umutisik's repository Eigentechno:

```
git clone git://github.com/umutisik/Eigentechno
cd Eigentechno
ls -a
```

`git clone` names the remote repo `origin` by default. All files in a cloned repository are staged and packed (info in `.git/objects/pack/`). One can also fork a repository first on github.com, and then clone it locally. For example, after I have forked the repository Eigentechno on github.com, I can do

```
git clone git@github.com:cxl791/Eigentechno
cd Eigentechno
git remote add origin2 git://github.com/umutisik/Eigentechno
git pull origin2 master  ## Get the latest from the original source
git push origin master  ## Push changes to my repo on github.com
```

What not to commit: (1) No derived files. For a LaTeX manuscript, no .log, .dvi, .aux, etc. For R code to generate a figure, no figure file. (2) No binary files (.docx, .xlsx, etc). Git works best with text files (source code, text, markdown files) when there are merge conflicts. (3) Avoid very big files. Once a big file is committed, it take space and is hard to remove cleanly, even if you use `git rm` to remove it later. So, use `git commit -a` with caution.

Ignore file: Files you're not tracking can be indicated in an ignore file to avoid seeing them in the output of `git status`. A global ignore file can be created under home directory and set with command:

```
git config --global core.excludesfile ~/.gitignoreglobal
```

A repository can have its own ignore file `.git/info/exclude`. Every subdirectory can also have its own ignore file named `.gitignore`; the `.gitignore` file should be tracked. An example ignore file contains:

```
*~
.*~
.DS_Store
.Rhistory
.RData
```

SSH key authentication on github.com: Copy your key in `~/.ssh/id_rsa.pub`. Go to github.com, choose 'Settings', 'SSH keys', and then paste the key. To check if the key is added successfully, use `ssh -T git@github.com`. Details in http://kbroman.org/github_tutorial/pages/first_time.html

1.1.5 Python Installation, .ipynb/.py Files, Modules, and DataFrame

Anaconda: HOML requires python, various packages (e.g., numpy, scipy, pandas, scikit-learn), matplotlib, and Jupyter notebook (for running the book's code). The easiest way to install all relevant packages is to install Anaconda. I installed Anaconda version 3.6 (not 2.7) on my computer. On Linux, if you already have Anaconda installed, use `conda list` to list all the packages installed by Anaconda. To update Anaconda, first `conda update conda`, then `conda update anaconda`.

Alternatives of installation are the OS's own package management system or python's system `pip`.

HOML python code: Can be downloaded with `git clone https://github.com/ageron/handson-ml`.

.ipynb files are in the JSON format. On Linux, to open a .ipynb file, use `jupyter notebook [file.ipynb]`; to close it, use Ctrl-C in the same terminal. Jupyter's autosave feature can be annoying. To turn it off on Linux, create/edit file `~/.jupyter/custom/custom.js` to contain:

```
require(['base/js/namespace', 'base/js/events'], function (IPython, events) {
  events.on("notebook_loaded.Notebook", function () {
    IPython.notebook.minimum_autosave_interval = 0; // disable autosave
  });
});
```

The JSON format is clunky. I prefer to save the code as a .py file and play with it in spyder (similar to RStudio). To convert a .ipynb file to a .py file, use `jupyter nbconvert --to script [file.ipynb]`, or inside Jupyter, go to 'File', 'Download as', and choose 'Python'.

Loading modules: `<from [module1]> import [module2] <as [alias]>`. A package can contain many modules, which are organized as a tree. For example,

```
import sklearn
import numpy.random
import pandas as pd
from sklearn import pipeline
from sklearn import linear_model as linmod
from sklearn.base import BaseEstimator, TransformerMixin
```

Packages often used: **numpy** and **scipy** are for basic data manipulation; **pandas** adds functionalities for DataFrame; **sklearn** adds machine learning methods; **matplotlib** provides functionalities for graphics. Datasets in Scikit-Learn are NumPy arrays or SciPy sparse matrices.

DataFrame: A pandas dataset is called a DataFrame. Pandas focuses on tabular data structures, and when doing the operations (e.g., addition, subtraction, etc.) it looks at the indices (row names), not positions. For example,

```
df = pd.DataFrame(np.random.randn(5, 3), index=list('abcde'), columns=list('xyz'))
df[1:]/df[:-1]  ## a little unexpected results
df[1:]/df[:-1].values  ## .values makes it a numpy array, and the results are like in R
```

Below is a cross reference of some basic operations between Python and R:

pandas DF	R DF	pandas DF	R DF	pandas Series	R vec
df.head()	head(df)	len(df)	dim(df)[1]	v.value_counts()	table
df.info	str(df)	list(df)	names(df)		
df.describe()	summary(df)				
df.shape	dim(df)				
df.index	rownames(df)				
df.columns	names(df)				
df.sort()	df[order()]				

1.1.6 ISLR Chapter 1

Some R code examples:

```
#### Figure 1.2, Smarket data (also see ISLR Section 4.6)
?Smarket
str(Smarket)
head(Smarket)

## Fig 1.2 first panel
with(Smarket, plot(Direction, Lag1,
                   xlab="Today's Direction", ylab="Percentage change in S&P",
                   main='Yesterday', col=2:3))
```

```
#### Figure 1.4, NCI60 data (also see ISLR Section 10.6)
str(NCI60)
names(NCI60)
class(NCI60$data)
dim(NCI60$data)
NCI60$labs

## Fig 1.4 right panel (with different colors)
tmp = prcomp(NCI60$data, scale=T)
str(tmp)
pchcode = as.numeric(as.factor(NCI60$labs)) %% 4 + 20
pchcode[pchcode==22] = 24
pchcode[pchcode==20] = 22
colcode = as.numeric(as.factor(NCI60$labs)) %% 7+2
plot(tmp$x[,1], tmp$x[,2], pch=pchcode, col=colcode, bg=colcode)
```

1.1.7 Assignment

1. Matrix warm-up: ISLR Chapter 1; DL Chapter 2.
2. Install course relevant R packages; get familiar with RStudio.
3. R warm-up: ISLR Chapter 2 R Lab. (It shows 3 ways of plotting a function f(x,y): `contour()`, `image()`, `persp()`)
4. Warm-up reading: ISLR Chapter 1.
5. Reading for next lecture: ISLR Chapter 2; HOML Chapter 1.
6. Get familiar with git and python (not urgent).

1.2 Week 1 Day 2

ISLR Chapter 2 provides an overview of some basic issues in data analysis methods, including the trade-off between flexibility and interpretability among existing methods, and the trade-off between bias and variance in both regression and classification problems.

1.2.1 ISLR 2.1

A general form of a supervised model is

$$Y = f(X) + \epsilon, \tag{2.1}$$

where ϵ is independent of X and has expectation zero. This formulation reflects probabilistic thinking. "Statistical learning refers to a set of approaches for estimating f." — ISLR

Figure 2.1 displays sales (in thousands of units) for a product vs. advertising budgets (in thousands of dollars) for TV, radio, and newspaper media.

```
Advertising = read.table("Advertising.csv", header=T, sep=',', row.names=1)
str(Advertising)
## Fig 2.1 left panel
with(Advertising, plot(TV, sales))
abline(lm(sales ~ TV, data=Advertising))
```

- 2.1.1: Prediction vs. inference. **Reducible vs. irreducible errors**. Consider the squared error loss $L(y, \hat{y}) = (y - \hat{y})^2$. Given a fitted model $\hat{f}$ and predictor value x_0, the expected loss for the prediction at x_0 is

$$E(y_0 - \hat{f}(x_0))^2 = [f(x_0) - \hat{f}(x_0)]^2 + Var(\epsilon) = [Bias(\hat{f}(x_0))]^2 + Var(\epsilon). \tag{2.3}$$

When the variation of estimating $\hat{f}$ is taken into account, the expected loss for the prediction at x_0 is

$$E(y_0 - \hat{f}(x_0))^2 = Var[\hat{f}(x_0)] + [Bias(\hat{f}(x_0))]^2 + Var(\epsilon). \quad (2.7)$$

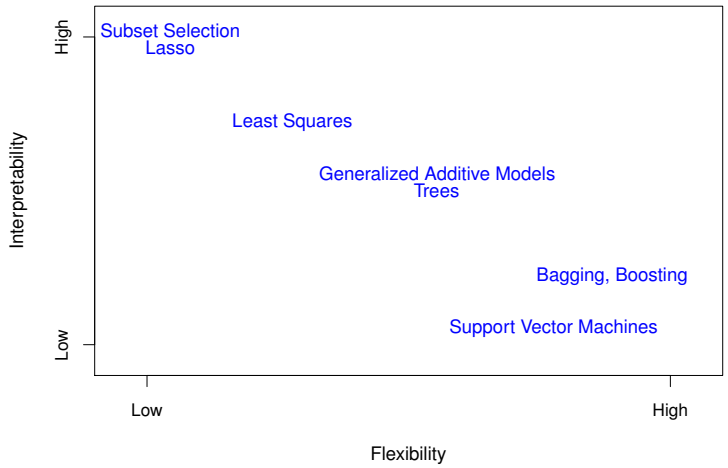
These are the three sources of variability in prediction: sampling, modeling, and irreducible error. Irreducible error provides a bound on the accuracy of our prediction. The bound is almost always unknown in practice.

- 2.1.2: Estimation of f: Parametric vs. nonparametric vs. semiparametric models. Data are split into training set and test set. Model fitting criterion (e.g., least squares, minimum cost, etc.). Overfitting.

Thin-plate spline using the `fields` package. Datasets `Income1` and `Income2` are simulated and they are different datasets!

```
library(fields)
library(rgl)  ## for #3D plots

Income2 = read.table("Income2.csv", header=T, sep=',', row.names=1)
names(Income2); dim(Income2); summary(Income2)

fit = with(Income2, Tps(cbind(Education, Seniority), Income, m=3, scale.type="range"))

ngrid = 100
gridedu = seq(10, 22, length.out=ngrid)
gridsen = seq(20, 188, length.out=ngrid)
grid2 = expand.grid(gridedu, gridsen)
grid2$pred = predict(fit, x=as.matrix(grid2))
names(grid2)

with(Income2, plot3d(Education, Seniority, Income, col=2, size=5))
plot3d(grid2$Var1, grid2$Var2, grid2$pred, add=T, alpha=.5)
```

- 2.1.3: Model flexibility vs. interpretability. Figure 2.7. With big data, traditional interpretable models tend to underfit, which derail their "interpretability". Artificial neural networks (ANNs) would be at the lower right corner (high flexibility but low interpretability).



- 2.1.4: supervised vs. unsupervised vs. semi-supervised learning (also see HOML Chapter 1)
- 2.1.5: regression vs. classification models

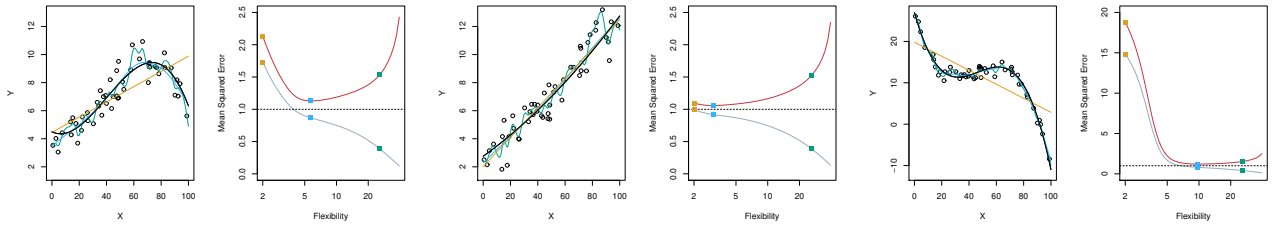
1.2.2 Demonstration of variantion $Var(\hat{f})$

We use `horsepower` vs `mpg` (in `Auto` dataset) as an example of joint distribution of (X, Y), and subsample the data. Suppose we have sampled only $n = 50$ cars and fitted quantile regression to data. The fitted model has variability from one sample to another. The smaller n becomes, the higher variability.

```
library(quantreg)  ## quantile regression package
str(Auto)
with(Auto, plot(horsepower, mpg))

## Repeat this 100 times (i.e., 100 replicates)
with(Auto, plot(horsepower, mpg))
for(ii in 1:100) {
  subidx = sample.int(392, 50)
  Autosub = Auto[subidx,]
  mod = rq(mpg ~ horsepower + I(horsepower^2), data=Autosub, tau=.5)
  idx = order(Autosub$horsepower)
  points(Autosub$horsepower[idx], fitted(mod)[idx], type='l')
}

mod = rq(mpg ~ horsepower + I(horsepower^2), data=Auto, tau=.5)
idx = order(Auto$horsepower)
points(Auto$horsepower[idx], fitted(mod)[idx], type='l', lwd=3, col=2)
```

1.2.3 ISLR 2.2

- 2.2.1: Continuous outcomes: MSE as a measure of the quality of fit; it effectively uses squared error as the measure of loss. Calculate training and test MSEs as functions of model flexibility (or model DF). Figures 2.9–2.11 show 3 scenarios, all with a single predictor, $n = 50$ in training data, the true curve (in black), and 3 fitted curves (from a linear model and 2 smoothing splines).

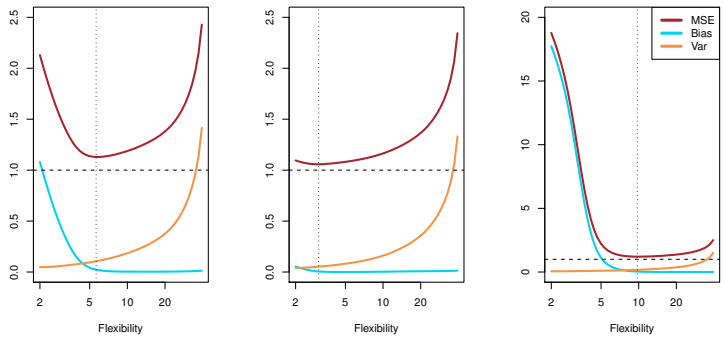


Using test data MSE to determine model complexity is a good approach. With a single predictor, linear models often underfit, but when dealing with many predictors, linear models may overfit and require regularization.

- 2.2.2: Bias–variance trade-off in test data MSE.

$$E(y_0 - \hat{f}(x_0))^2 = Var[\hat{f}(x_0)] + [Bias(\hat{f}(x_0))]^2 + Var(\epsilon). \tag{2.7}$$

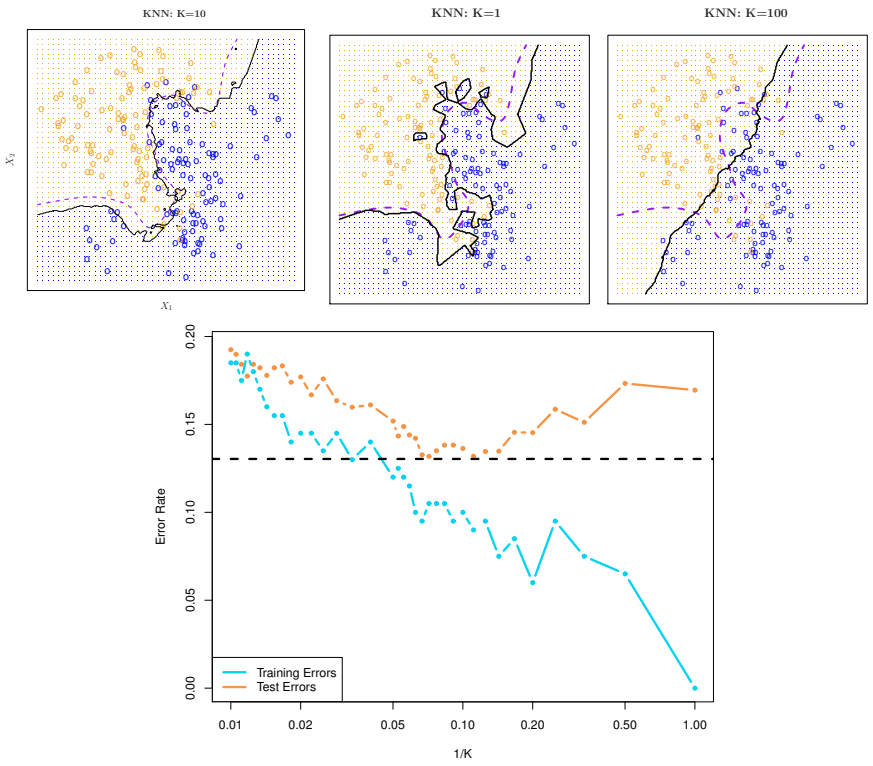
The three sources of variability are due to sampling, modeling, and irreducible error. Given a fixed dataset, a more complex/flexible model tends to reduce bias but increase variation. Figure 2.12 shows all three components for the simulation scenarios of Figures 2.9–2.11.



- 2.2.3: Discrete outcomes: Classification error rate (effectively using 0-1 loss as the measure of loss). Similar decomposition of sources of variation as in (2.7). Other popular loss functions for discrete outcomes are Gini index and cross-entropy (ISLR Chapter 8).

Bayes classifier: $f(x) = \arg\max_g P(Y = g|X = x)$, is the best classifier because it minimizes the classification error rate. It is often used as a reference in simulation studies; Figure 2.13 is an example. A Bayes classifier has a corresponding Bayes decision boundary and a Bayes error rate. (Note that this is not Bayesian statistics.) The Bayes classifier is often **unknown in practice.**

k-nearest neighbors (KNN) is a local estimation method, with k being a hyperparameter. KNN requires a measure of similarity (for defining neighborhoods). It is similar to moving-window approaches; in KNN k is fixed, while in moving-window approaches, window size is fixed. The smaller k the higher model flexibility; an example is in Figures 2.15–2.17 ($n = 200$). With high dimensional predictors, all local estimation methods suffer from the curse of dimensionality (ISLR Chapter 3.5). KNN is also applicable to continuous outcomes (ISLR 3.5).



1.2.4 HOML Chapter 1

Types of ML systems:

- From the problem perspective: Supervised vs. unsupervised vs. semisupervised learning. "Labels" are called outcomes in Statistics.
- From the algorithm perspective: Batch vs. online learning. Out-of-core learning.
- From the method perspective: Instant-based vs. model-based learning. In Statistics, "instant-based learning" is local estimation.

Reinforcement learning describes learning in a scenario where new observations (x, y) can be generated automatically by randomly trying a condition x and obtaining a result y. This can be very effective because the sample size n can be very large and the data can be truly random. Applications include games and robotics studies. A famous example is AlphaGo (and AlphaGo Zero) for the Go game.

Main challenges in ML/Statistics:

- Algorithm development vs. corpus development. "Unreasonable Effectiveness of Data" in language learning
- Sampling bias: when data are not representative (not randomly drawn from the target population). "Sampling noise" in this book is called sampling variation in Statistics.
- Data quality: Outliers, missing data, missing features.
- Too many (irrelevant) features. Feature engineering (selection, extraction, collection).
- Overfitting. Signs of overfitting. Regularization prevents overfitting.
- Underfitting.

The example behind Figure 1-23 is misleading: Regularization is known to push slope estimates closer to zero. Random missing of observations does not lead to an elevated slope (it leads to a higher variation in slope). The fact that the slope obtained from the full data is closer to zero than the slope for that specific subset is because the missing observations probably were not randomly chosen (as shown in the figure). With a single predictor, there is no ganrantee that regularization gives results closer to the truth; this is because a linear model with a single predictor is much more likely to underfit than to overfit. With many predictors or a very complex model (e.g., ANN), the model may overfit and regularization often gives better results.

Hyperparameters and validation:

- Hyperparameters fall in three categories:
  1. Choices of a model structure, e.g., number of included features, neighbor size.
  2. Choices of model fitting criteria, e.g., cost function, amount of regularization.
  3. Choices in model fitting process, e.g., parameter initialization, learning rate, mini-batch size, number of epochs.
- Some hyperparameters are also called *tuning parameters*.
- Hyperparameters can be selected using validation. Data can be split into training/validation/test sets, or training/test sets with cross-validation on the training set. Cross-validation is preferred to a standalone validation set unless the sample size is very large.

Training vs. validation vs. testing

1. Training set is for estimation of parameters;
2. Validation set or cross-validation is for selection of hyperparameters (selection criteria less formalized);
3. Test set is for evaluation of the final model, which has all parameters and hyperparameters determined.

1.2.5 Assignment

1. **Homework**: ISLR Chapter 2 Exercises 1, 3, 8
2. Reading for next lecture: ISLR 3.1–3.2
3. ISLR Chapter 3 R Labs
4. Try out the python code for HOML Chapter 1