Theory: Bias-Variance and Ensemble

Dr Liew How Hui

Jan 2022

Reminder

- Submission of **Report** + **Programming Code**:
  - Week 11 Wednesday.
  - In proper formats, marks will be deducted for bad format.
  - By Email from the Group Leader.
- Oral Presentation (Week 11 to Week 13?):
  - Arrangement(?): In reverse order of registration — the latest to register will present first, the first group will present last or have the right to choose time slot.
  - Presentation: 18–20 minutes? In class recording.
  - Submission of **presentation slides**.

14 groups 1 week 5+2 groups (Week 12 + Week 13)?

Reminder (cont)

Presentation is based on registration date in reverse order.

Week 11/12:

1. Team karipap
2. Paradise (2022.02.09)
3. Homosapiens (2022.02.09)
4. K thx bye (2022.02.08)
5. PM for details (2022.02.07)
6. Insight (2022.02.07)
7. Invaders (2022.02.07)

Week 12/13:

11. White-hat Hackers (2022.02.04)
12. The Steady Gang (2022.01.26)
13. We Grow, We Glow (2022.01.26)
14. Valentine (2022.01.26)
15. ALPHA (2022.01.24)
16. Malatsssssss (2022.01.24)
17. Teletubbies (2022.01.25)

Week 1–8 Revision

General 'formulation' of the 'unknown' model (Topic 1):

$$Y = f(\mathrm{X}) + \epsilon, \quad f(\mathrm{X}) = f(X_1, \ldots, X_p). \tag{1}$$

Here, the ϵ is the randomness of the real-world data generation 'mechanism' and has a mean of $\mathbb{E}[\epsilon] = 0$

The 'unknown' model is only 'observed' with the data

$$D = \{(\mathrm{x}_1, y_1), \ldots, (\mathrm{x}_n, y_n)\}.$$

If $f(\mathrm{X}) = 0$, that means $Y = \epsilon$ and the 'unknown' model is random and hence 'unpredictable'! So predictive modelling is useful when D is generated from $f(\mathrm{X}) \neq 0$ with $\mathrm{Var}[\epsilon]$ reasonably small.

Week 1–8 Revision (cont)

Let D_s be a subset of D, A class of predictive models trained on D_s can be expressed as

$$\widehat{Y} = h_{D_s}(\mathrm{X}) = h_{D_s}(X_1, \cdots, X_p) \tag{2}$$

We intend to approximate $f(vX)$ using $h_{D_s}(\mathrm{X})$. The functions f and h_D are all deterministic functions.

We have learned the following classes of predictive models:

- kNN ($k \in \mathbb{Z}$); wkNN is mentioned (nonparametric);
- LR; Multilogit & ANN are mentioned (parametric);
- Naive Bayes & LDA (parametric);
- Decision tree models (nonparametric).

Week 1–8 Revision (cont)

Things to take note: When training predictive models with data, we may need to perform data scaling when the predictive models (as well as the unsupervised learning methods) are based on distance and optimisation.

- kNN: Based on distance/similarity measure, data scaling is recommended.
- LR, Multilogit & ANN: Based on optimisation, data scaling is recommended.
- LDA: The matrix performs terribly to low-dimensional data, data scaling is recommended.
- Naive Bayes, Decision tree models: Data scaling is not necessary.

Other considerations: SMOTE (Synthetic Minority Over-sampling TEchnique, based on NN) may be useful for the training of models with unbalanced data, etc.

Week 1–8 Revision (cont)

The **measurement** for the **differences** between Y from the 'unknown' model (1) and the $\widehat{Y}$ from the predictive model (2), called '**loss**', depends on the type of Y.

For classification problems, $Y \in \{1, 2, \cdots, K\}$ and a possible loss function is

$$\ell(Y, \widehat{Y}) = \mathbb{E}[\#(Y \neq \widehat{Y})].$$

For regression problems, $Y \in \mathbb{R}$ and a possible loss function is the mean squared loss:

$$\ell(Y, \widehat{Y}) = \mathbb{E}[(Y - \widehat{Y})^2].$$

Another possible loss function is $\ell(Y, \widehat{Y}) = \mathbb{E}[|Y - \widehat{Y}|]$.

Outline

Theoretical Questions

Can we 'guess' the predictive model $h_D(\mathrm{X})$ without really depending on the data D??? Yes, we can. When the D is fixed (not random). Physicists perform measurements to obtain data and then try to fit the data using mathematical formulas (which they guessed).

Suppose we are working with regression problems and using mean squared loss, then

$$\begin{aligned}
&\mathbb{E}[(Y-\widehat{Y})^2] \\
&= \mathbb{E}[(f(\mathrm{X})+\epsilon-h_D(\mathrm{X}))^2] \\
&= \mathbb{E}[(f(\mathrm{X})-h_D(\mathrm{X}))^2]+2\mathbb{E}[\epsilon(f(\mathrm{X})-h_D(\mathrm{X}))]+\mathbb{E}[\epsilon]^2 \\
&= (f(\mathrm{X})-h_D(\mathrm{X}))^2+\mathrm{Var}[\epsilon] = (\mathrm{bias})^2+\mathrm{variance}
\end{aligned}$$

Here the variance is the 'noise' or 'irreducible error'.

Bias-Variance Decomposition

The difference between the 'unknown' model $f(\mathrm{X})$ and the predictive model $h_D(\mathrm{X})$ is called the **bias**.

The randomness of the unknown model is the **variance**. It cannot be eliminated and $\mathrm{Var}[\epsilon] > 0$.

Therefore, choosing a good predictive model means to find the one with zero or very small bias!!! But how to we know $f(\mathrm{X}) - h_D(\mathrm{X})$ when we don't know f?

- For theoretical investigation, we can choose any $f(\mathrm{X})$ and generate a random set of D to test how good are the predictive models;
- For real-world data, we need to apply holdout method (train-test set) or cross-validation method.

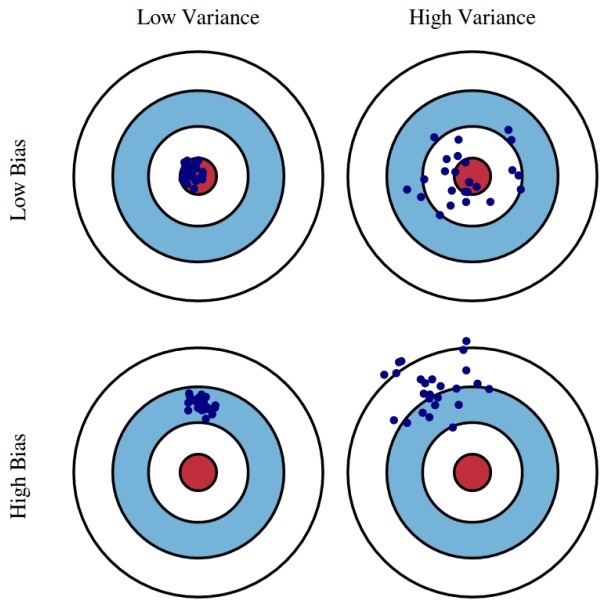
Bias-Variance Decomposition (cont)

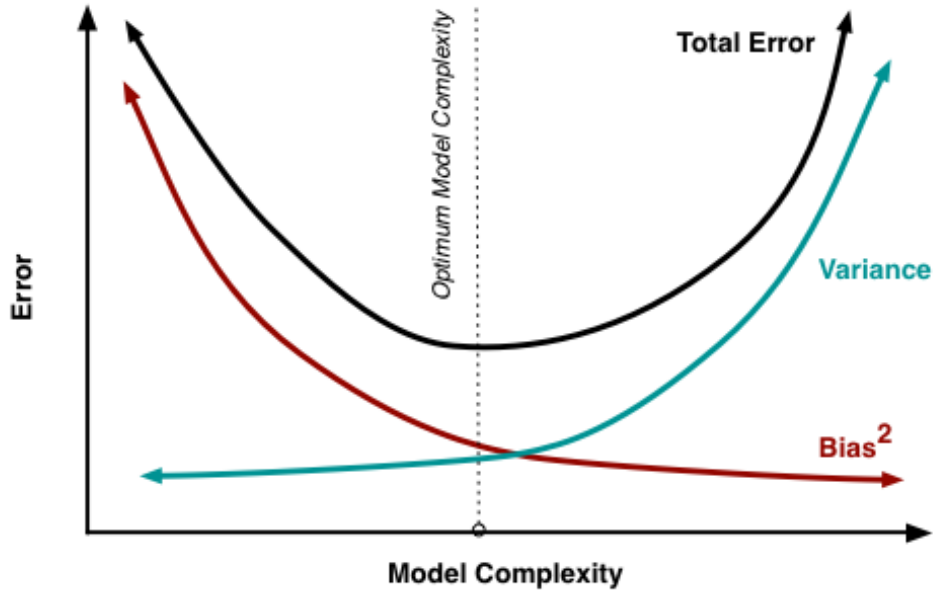
What we are interested is to obtain a **generalisation** h_D from D which matches f but for real-world data, we **don't know** f. So we resolve to calculate the following values to try to guess the error/loss (e.g. $\mathbb{E}[(Y - \widehat{Y})^2]$).

- Holdout method: D is splitted to training data D_t and validation data D_v. We can calculate
  - training error $= \frac{1}{n_t} \sum_{x_t \in D_t} (y_t - h_{D_t}(x_t))^2$
  - testing error $= \frac{1}{n_v} \sum_{x_v \in D_v} (y_v - h_{D_t}(x_v))^2$
  - Underfit: training error is large
  - Overfit: training error is low (or zero) but testing error is large (noise is fitted)
  - Nice fit: both errors are not large.
- K-fold CV: err1, err2, ..., errK has mean and stdev
- Bootstrap (?)

Theory (cont)

Figure on the left: Graphical illustration of bias and variance;
Figure on the right: The variation of Bias and Variance with the model complexity. This is similar to the concept of overfitting and underfitting. More complex models overfit while the simplest models underfit.





Theory — Advanced

What happens to the bias-variance when we have more data D_1, D_2 . . . for training?
Assume that D_1, D_2, . . . are i.i.d random and let $\mathbb{E}_D[h_D(\mathrm{x})] = \overline{h}(\mathrm{x})$, the bias-variance we obtained earlier generalises below.

Bias-Variance Decomposition

$$\underbrace{\mathbb{E}_{\mathrm{x},y,D}\left[\left(y - h_D(\mathrm{x})\right)^2\right]}_{\text{Expected Test Error}} = \underbrace{\mathbb{E}_{\mathrm{x}}\left[\left(f(\mathrm{x}) - \overline{h}(\mathrm{x})\right)^2\right]}_{\text{Bias}^2} + \underbrace{\mathbb{E}_{\mathrm{x},D}\left[\left(h_D(\mathrm{x}) - \overline{h}(\mathrm{x})\right)^2\right]}_{\text{Variance (due to Data)}} + \underbrace{\mathbb{E}_{\mathrm{x},y}\left[\epsilon^2\right]}_{\text{Noise}}. \tag{3}$$

Theory (cont)

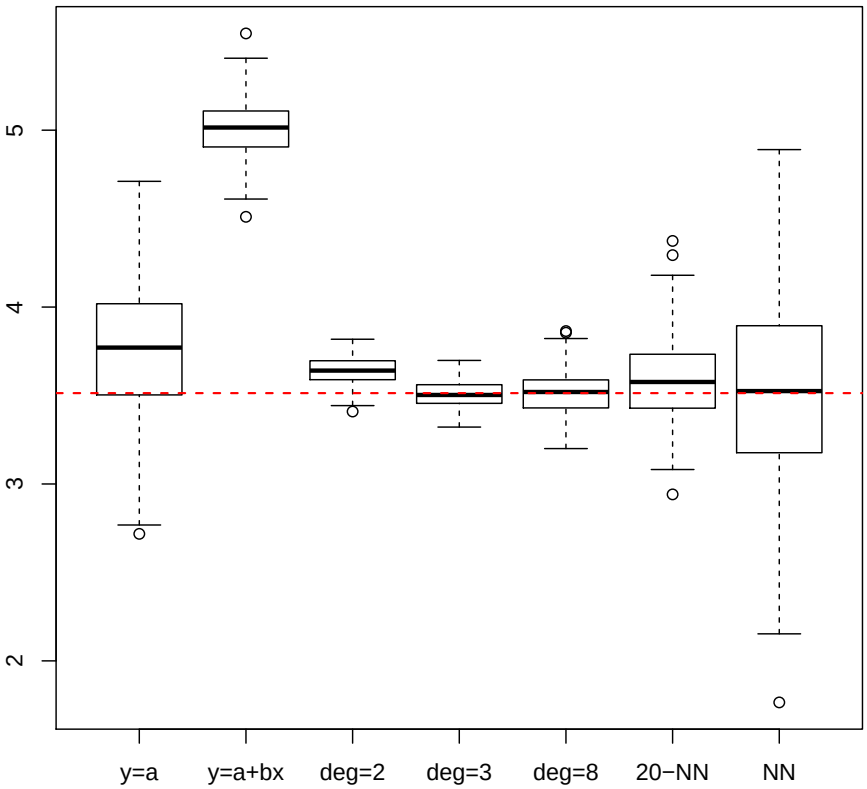
The "Variance" captures how much the classifier changes if we train on a different training set. How "over-specialised" is the classifier to a particular training set (i.e. is it overfitting)?

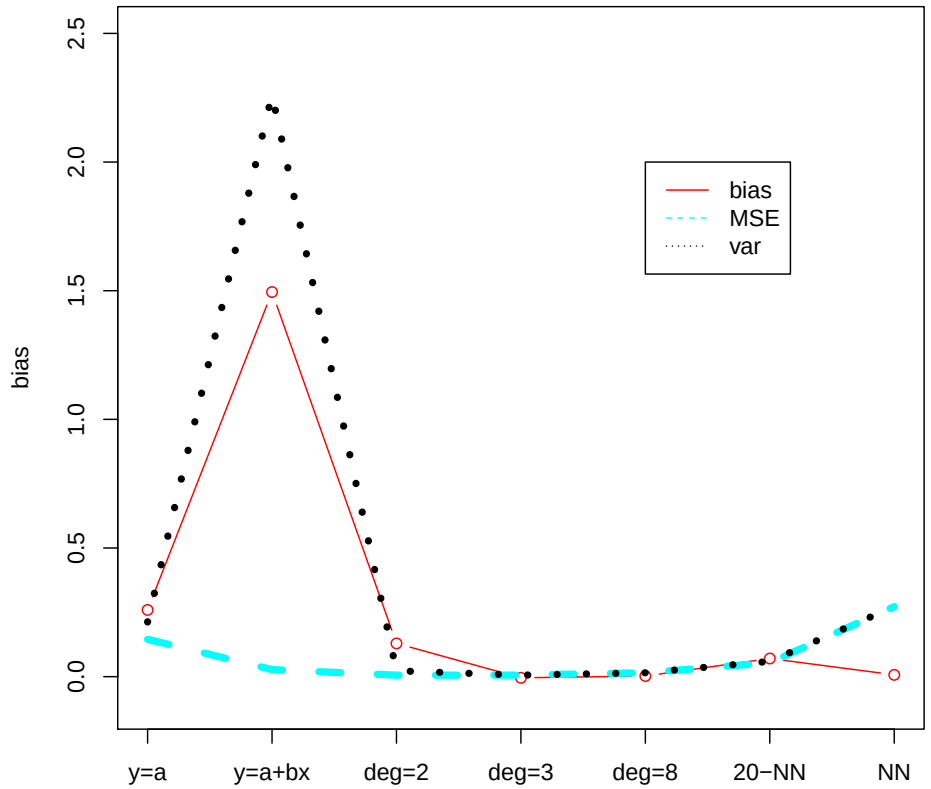
What is the inherent error that we obtain from the classifier even with infinite training data? This is due to the classifier being "biased" to a particular kind of solution (e.g. linear classifier).

How large is the data-intrinsic "noise"? This error measures ambiguity due to our data distribution and feature representation. We can never beat this, it is an aspect of the data.

Theory (cont)

Consider a theoretical model $f(x) = x^2 + 1.2x^3$, $0 \leq x \leq 2$, $\sigma = 0.5$ with training datasets $D_1, \ldots, D_{250}$. The predictions $h_{D_i}(1.2)$, $i = 1, \ldots, 250$ for various polynomial regression and kNN regression models are:





Outline

Resampling Methods

Ideally, we would want to draw large, non-repeated, samples from a population in order to create a sampling distribution for a statistic. However, limited resources prevent us from getting the ideal statistic.

Resampling means that we can draw small samples over and over again from the same population D. As well as saving time and money, the samples can be quite good approximations for population parameters.

Holdout, K-fold CV, Bootstrap are all **resampling methods** used to measure the generalisation error (performance). This section tries to develop the mathematical formulations for resampling methods.

Cross-Validation

Assume a given dataset

$$D = \{(\mathrm{x}_1, y_1), \ldots, (\mathrm{x}_n, y_n)\} \sim P(X, Y)$$

is drawn i.i.d. from some (unknown) distribution $P(X, Y)$ and the responses $y_i \in \mathbb{R}$.

Suppose the data D is used to train a supervised learning model $\widehat{h}_D$ and suppose that we have a new test data

$$(\mathrm{x}_1^{new}, y_1^{new}), \cdots, (\mathrm{x}_L^{new}, y_L^{new}) \sim P. \tag{4}$$

Cross-Validation (cont)

Loss function = "error measurement"

$$\frac{1}{L}\sum_{i=1}^{L}\ell(y_i^{new}, \widehat{h}_D(\mathrm{x}_i^{new})) \rightarrow \mathbb{E}[\ell(Y^{new}, \widehat{h}_D(X^{new}))] \quad (5)$$

where the right is the *theoretical test set error* for a predictive model training on a particular training set $\widehat{h}_D$. The *theoretical generalisation error* is defined as the average of theoretical test set error over all possible training sets, i.e.

$$\mathbb{E}_D\left[\mathbb{E}_{(X^{new}, Y^{new})}[\ell(Y^{new}, \widehat{h}_D(X^{new}))]\right]. \quad (6)$$

Cross-Validation (cont)

For classification problems, the usual form of test error (5) and ℓ is

$$\frac{1}{L}\sum_{i=1}^{L} I(y_i^{new} \neq \widehat{h}_D(\mathrm{x}_i^{new})).$$

For regression problems, the usual form of test error (5) and ℓ is

$$\frac{1}{L}\sum_{i=1}^{L} (y_i^{new} - \widehat{h}_D(\mathrm{x}_i^{new}))^2.$$

Cross-Validation (cont)

The estimate of the generalisation error (6)

1. Split-test validation: $\approx \frac{1}{n}\sum_{i=1}^{n} \ell(y_i^{new}, \widehat{h}_D(\mathrm{x}_i^{new}))$
2. Leave-one-out cross-validation (LOOCV):
   $\approx \frac{1}{n}\sum_{i=1}^{n} \ell(y_i^{new}, \widehat{h}_{n-1}^{(-i)}(\mathrm{x}_i^{new}))$
3. K-fold CV: $\approx \frac{1}{K}\sum_{i=1}^{K} \frac{1}{|B_k|}\sum_{i\in B_k} \ell(y_i^{new}, \widehat{h}_{n-|B_k|}^{(-B_k)}(\mathrm{x}_i^{new}))$

Cross-Validation (cont)

Weaknesses of

1. split-test validation: Splitting the sample once is pretty crude and unstable as an approximation to (6) because it involves the random choice of split.
2. LOOCV: $\ell(Y, \widehat{h}_{n-1}^{(-i)}(X)$ is an unbiased estimate of (6), so it is low bias (it may be high-variance, but not always).
3. K-fold CV: $\ell(Y, \widehat{h}_{n-|B_k|}^{(-B_k)}(X)$ is slightly bias estimate of (6) (it may be less higher-variance, but not always).

Bootstrap

Bootstrapping is a type of resampling where large numbers of smaller samples of the same size are repeatedly drawn, **with replacement**, from a single original sample.

R Example:

```
# Suppose data Data={1,2,...,10}
set.seed(123)
# Sample WITHOUT REPLACEMENT
idx = sample(10, 7)  # 3 10  2  8  6  9  1
# Sample WITH REPLACEMENT (can repeat)
idx.boostrap = sample(10, 7, replace=TRUE)
# 4  6  9 10  5  3  9
```

Bootstrap (cont)

Suppose $Data = \{1, 5, 8, 3, 7\}$. Estimate the standard error of the sample median (a 'statistic' without formula. Standard error of the sample mean is a multiple of the sample standand deviation).

Using 10000 bootstrap in R:

```
Data = c(1,5,8,3,7)
B = 1000  # bootstrapping sample size
# create an array to store all medians
the.median = rep(0,B)
set.seed(12)
for(i in 1:B) {
  bootstrap = sample(Data, replace=T)
  the.median[i] = median(bootstrap)
}
cat("mean(median)=",mean(the.median),"\n")
cat("stderr(median)=",sd(the.median),"\n")
```

Bootstrap (cont)

Specialised bootstrapping methods:

- Jacknife bootstrap: Similar to LOOCV?
- Block/Spatial bootstrap: Similar to K-fold?

Disadvantages:

- Often used incorrectly
- Typically not useful for correlated (dependent) data
- Missing data, censoring, data with outliers are also problematic

Outline

Training vs Testing Errors

Suppose the data is split into "training" D_{train} and "testing" D_{test}.

For D_{train}, the regression error is

$$RSME_{train} = \sqrt{\frac{1}{|D_{train}|} \sum_{(x,y) \in D_{train}} (y - h_{D_{train}}(x))^2}$$

For D_{test}, the regression error is

$$RSME_{test} = \sqrt{\frac{1}{|D_{test}|} \sum_{(\tilde{x},\tilde{y}) \in D_{test}} (\tilde{y} - h_{D_{train}}(\tilde{x}))^2}$$

Training vs Testing Errors (cont)

Theoretically, the more data the better the "trained" model, we expect.

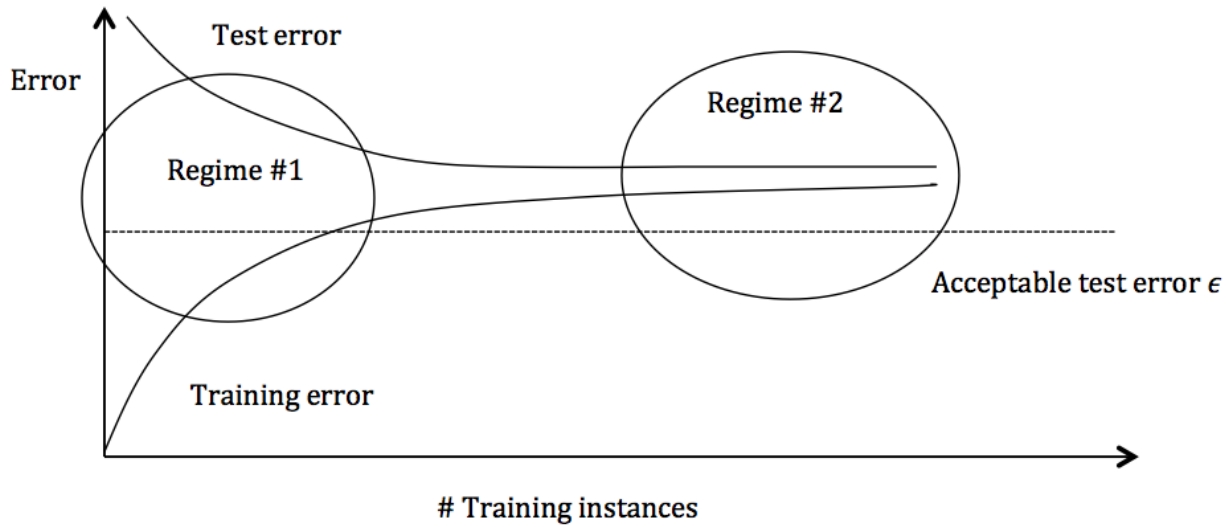
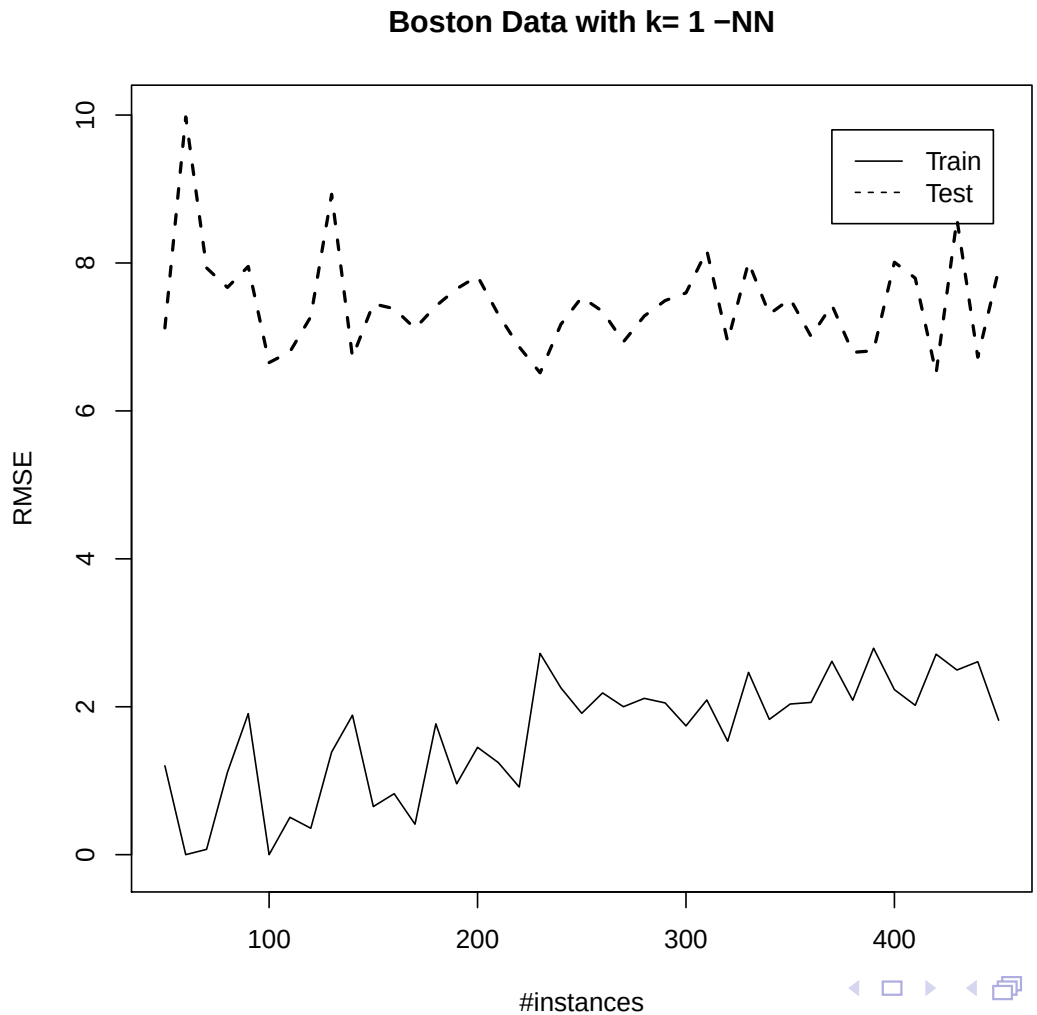


Figure: Test and training error as the number of training instances ↑. Region 1: High variance; Region 2: High bias

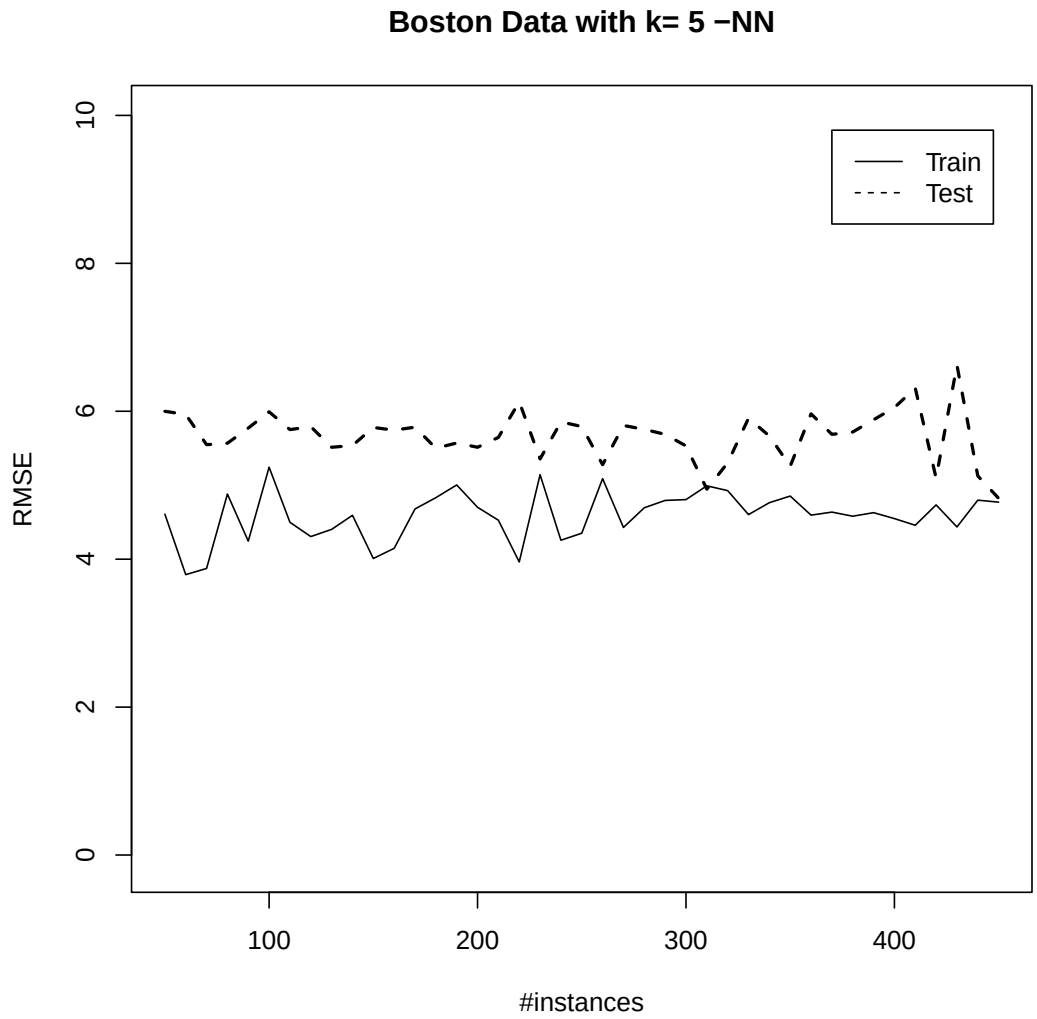
Remedy 1: Add more data / reduce the model complexity
Remedy 2: Use a more complex model

Training vs Testing Errors (cont)

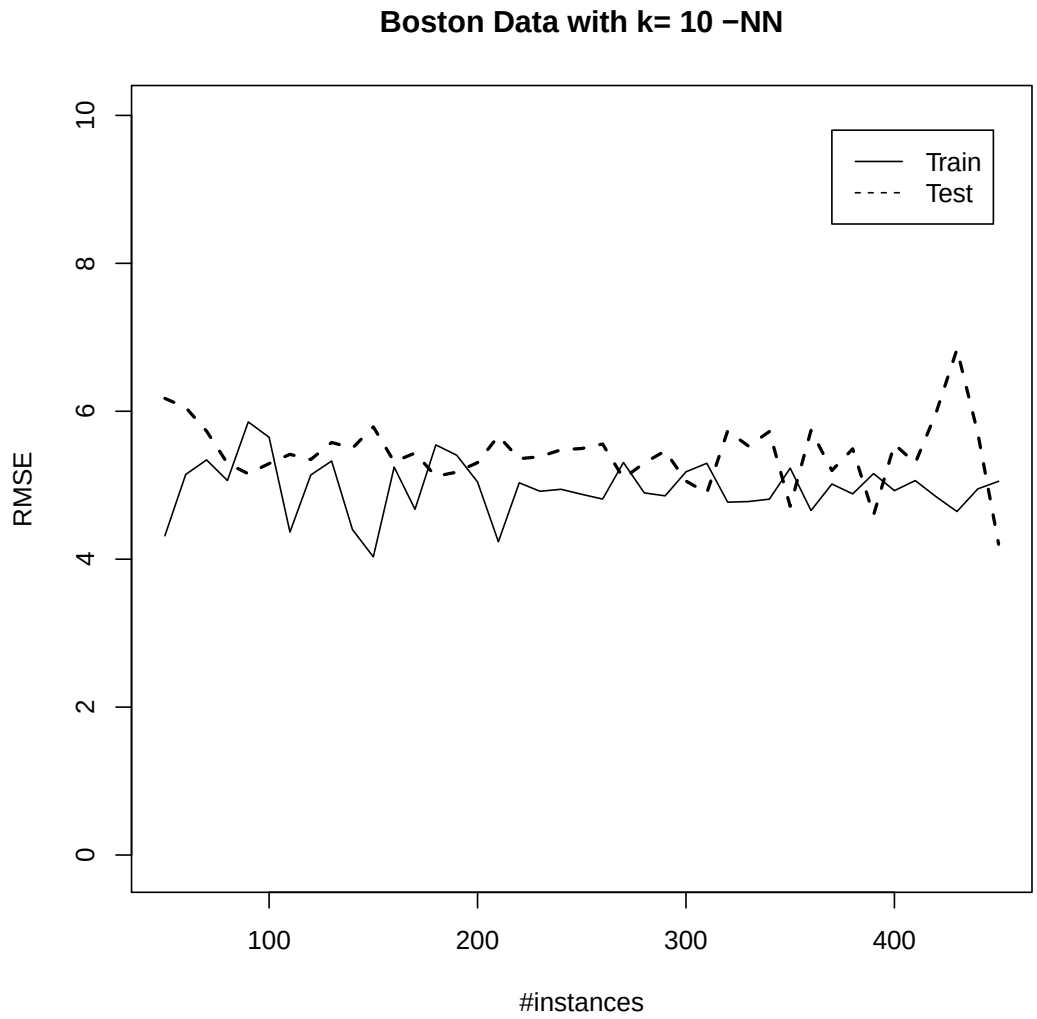
In reality, increasing number of rows in D may not produce 'smooth' curve as in the previous slide.



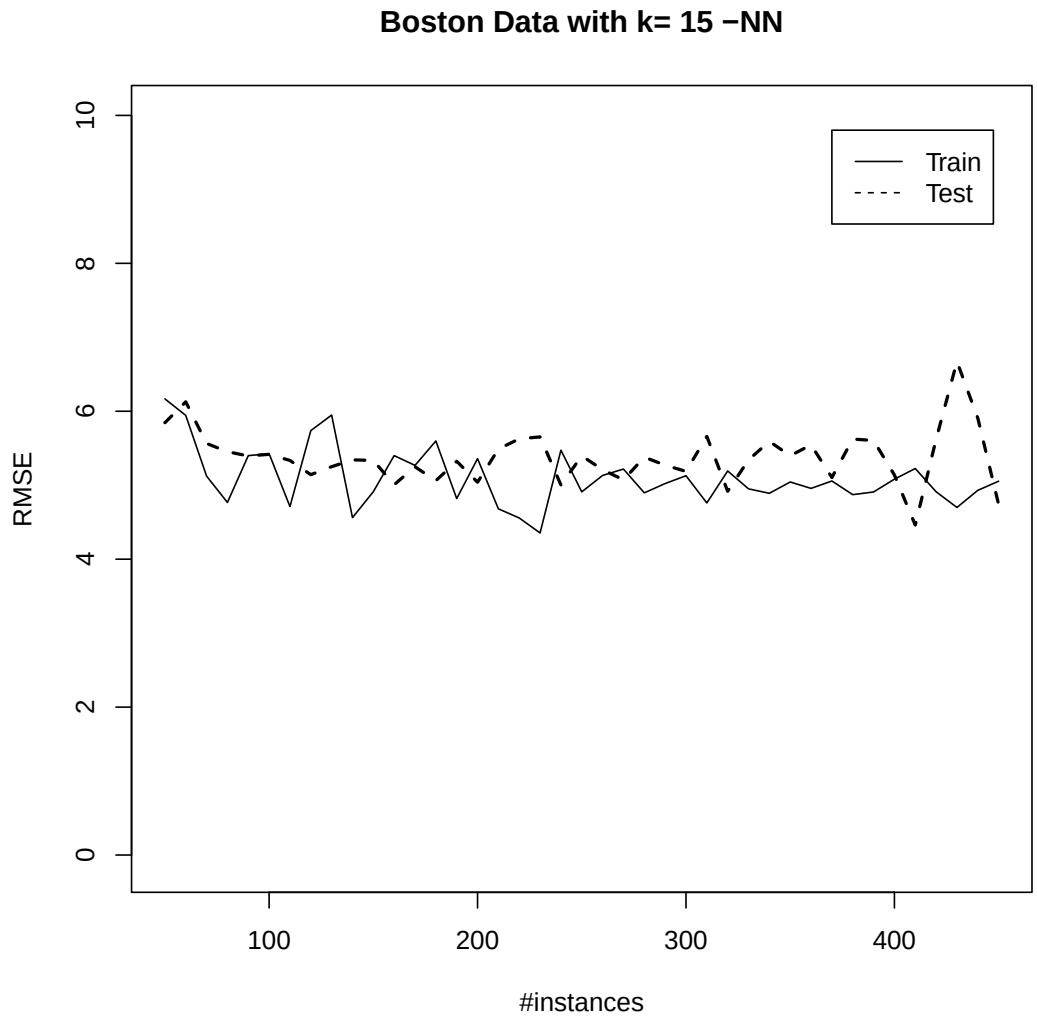
Training vs Testing Errors (cont)



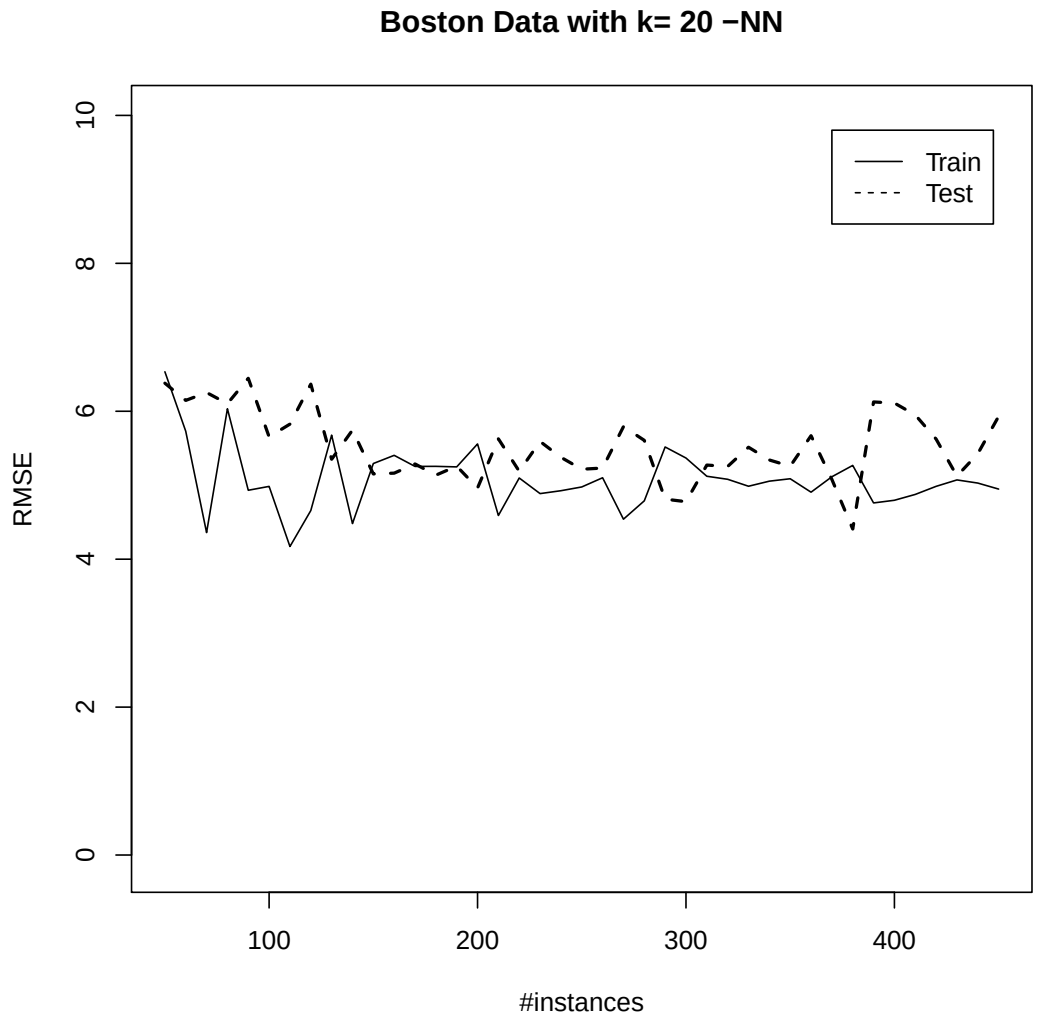
Training vs Testing Errors (cont)



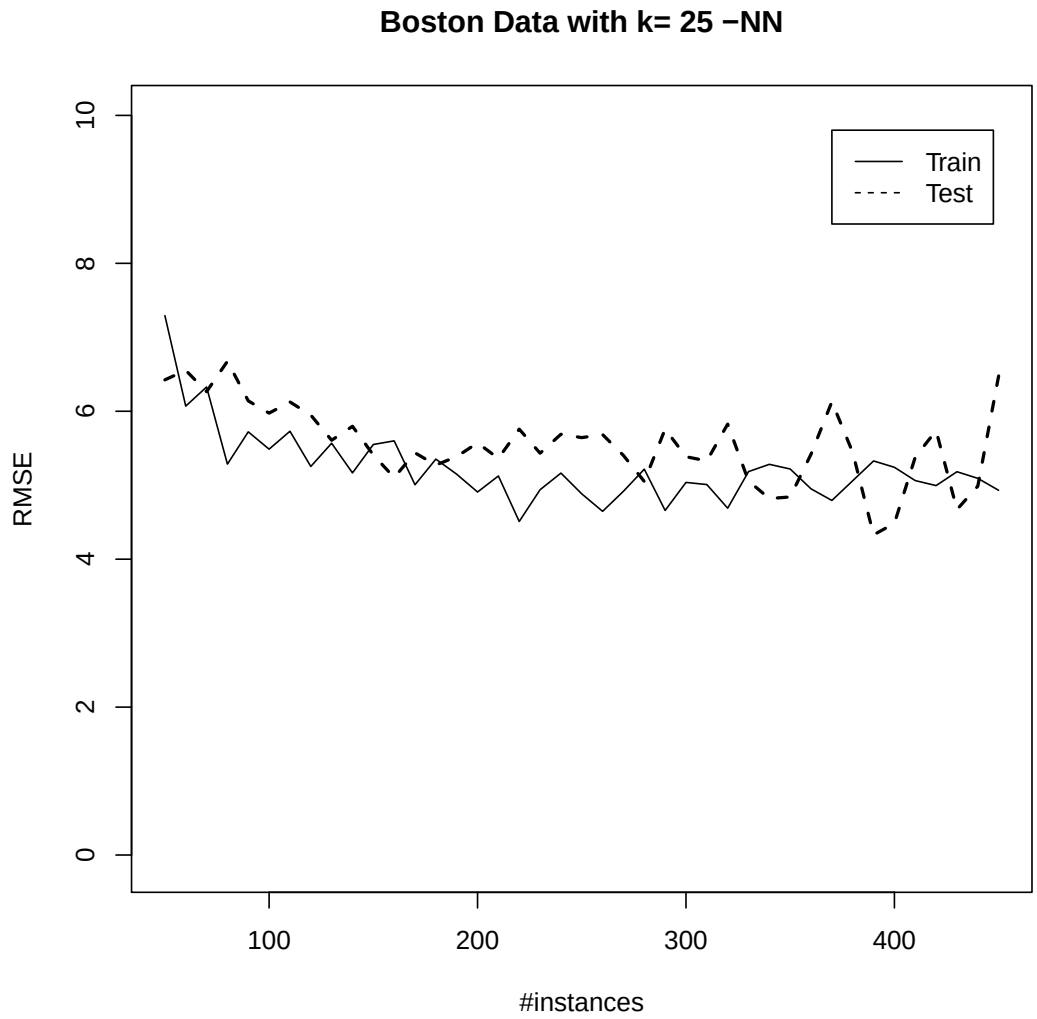
Training vs Testing Errors (cont)



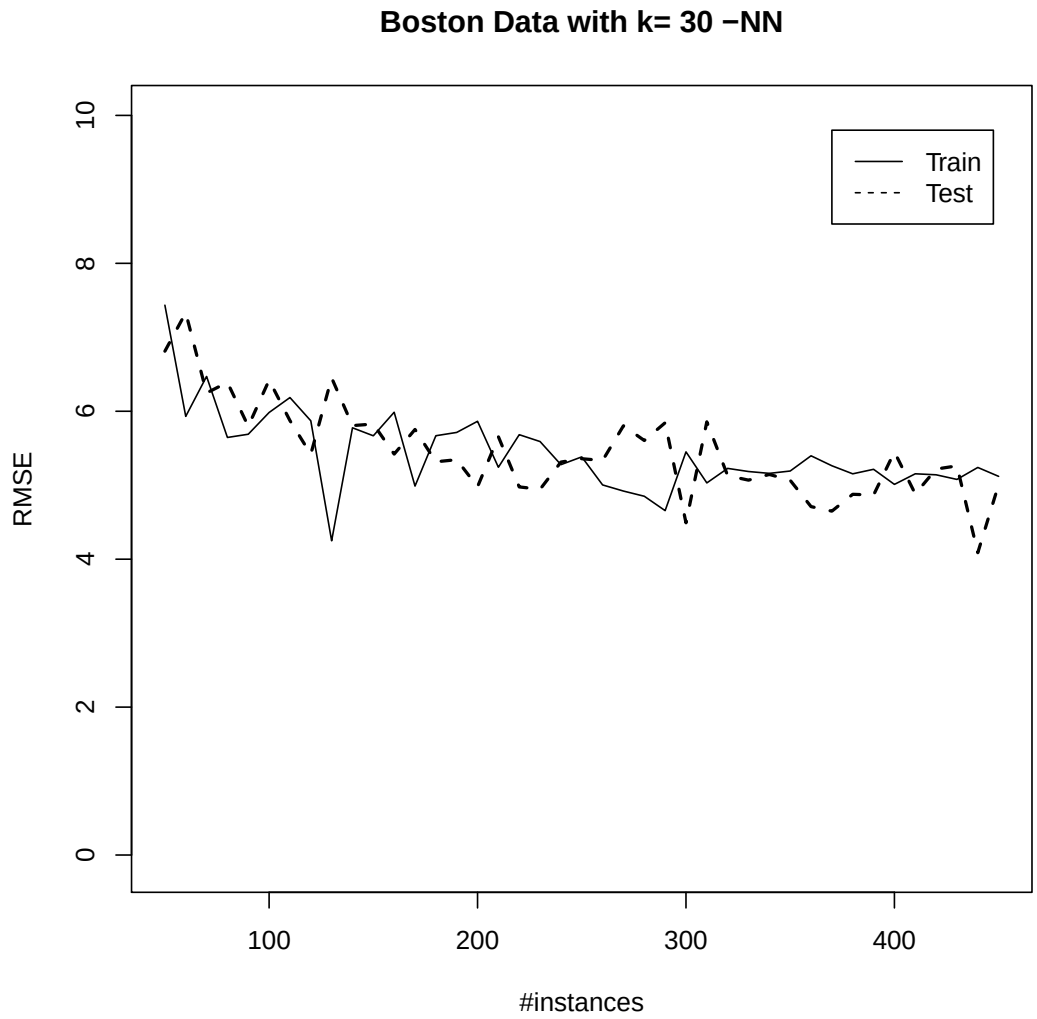
Training vs Testing Errors (cont)



Training vs Testing Errors (cont)

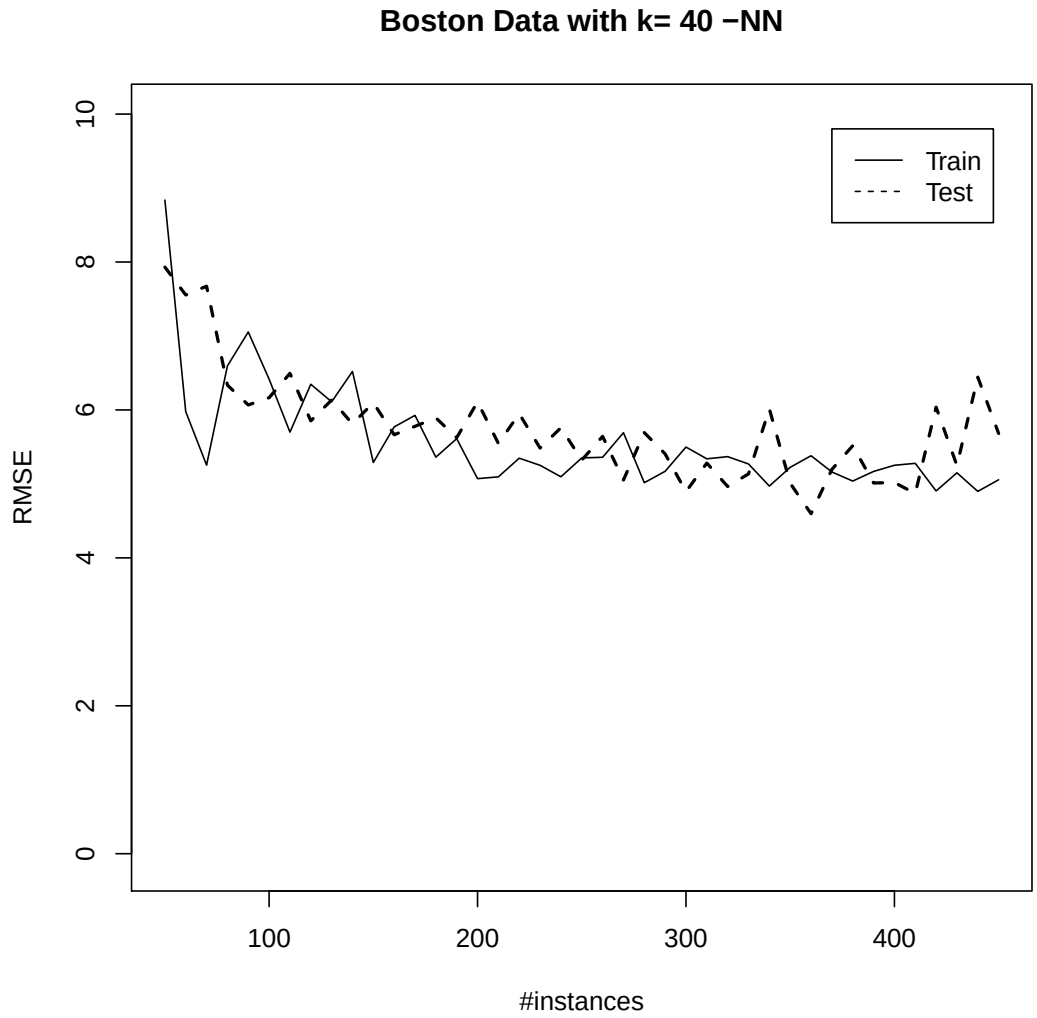


Training vs Testing Errors (cont)



Training vs Testing Errors (cont)

These are based on an adaption from Practical 2.



Training vs Testing Errors (cont)

The main use of "Training vs Testing Errors" is to capture the "best" hyperparameters as illustrated in the last part of Practical 2.

$k = 1$: Training error low, Testing error high $\rightarrow$ Overfitting
$k = 5$: Training error higher, Testing error lower $\rightarrow$ improving
$k = 10$: Training error higher, Testing error lower $\rightarrow$ OK?
$k = 15$: Training error higher, Testing error lower $\rightarrow$ OK?
$k = 20$: Training error higher, Testing error lower $\rightarrow$ OK?
$k = 25$: Training error higher, Testing error lower $\rightarrow$ OK?
$k = 30$: Training error higher, Testing error lower $\rightarrow$ OK?
$k = 40$: Training error higher, Testing error higher $\rightarrow$ Underfitting?

Outline

Ensemble Methods

`https://en.wikipedia.org/wiki/Ensemble_learning` or "Ensemble Methods" use multiple learning algorithms to obtain a more "flexible" model.

- Ensemble methods are not models but "meta-algorithms".
- They "combine" multiple "predictive models" h_1, ..., h_B to yield a single consensus prediction.
- Variations:
  - Booststrap: Bagging, Random Forest
  - Boosting: AdaBoost, Gradient Boost, ...

Outline

Bagging

Bagging = Boostrap Aggregation.

A Bagging classifier/regressor is an ensemble meta-estimator that fits base classifiers/regressors each on random subsets of the original dataset and then aggregate their individual predictions (by voting/averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

Bagging in R

It is similar to the Boostrapping method mentioned in an earlier slide.

```
B=100
bagged_models=list()
set.seed(12)
for(i in 1:B) {
  bootstrap = sample(idx.train, size=length(idx.train), replace=T)
  bagged_models = c(bagged_models, list(rpart(Y~.,
    d.f.train[bootstrap],control=rpart.control(minsplit=6))))
}
bagged_result=NULL
i=0
for (from_bag_model in bagged_models) {
  if (is.null(bagged_result))
    bagged_result=predict(from_bag_model,new_X)
  else
    bagged_result=
      (i*bagged_result+predict(from_bag_model,new_X))/(i+1)
  i=i+1
}
```

Bagging in R (cont)

Pictorial illustration:

B	C	D	E	F	G
Original	Bootstrap1	Bootstrap2	Bootstrap3	Bootstrap4	Bootstrap5
Index	D1.Index	D2.Index	D3.Index	D4.Index	D5.Index
1	9	1	7	9	7
2	9	7	5	6	1
3	3	5	6	5	6
4	8	10	9	9	2
5	10	7	2	10	1
6	7	9	5	4	2
7	10	9	8	6	4
8	9	10	2	8	5
9	3	7	1	6	6
10	4	5	9	6	3
	h_D1	h_D2	h_D3	h_D4	h_D5
Predict(X)	A	A	B	A	B

The ipred library provides the implementation.

```
bagging(y, X=NULL, nbagg=25,
        method=c("standard","double"), coob=TRUE,
        control=rpart.control(minsize=2, cp=0), ...)
```

Bagging in Python

Python's sklearn.ensemble provides BaggingClassifier and BaggingRegressor as well as VotingClassifier, VotingRegressor, StackingClassifier, StackingRegressor.

```
sklearn.ensemble.BaggingClassifier(base_estimator=None,
  n_estimators=10, max_samples=1.0, max_features=1.0,
  bootstrap=True, bootstrap_features=False,
  oob_score=False, warm_start=False, n_jobs=None,
  random_state=None, verbose=0)
```

Bagging in Python (cont)

- When random subsets of the dataset are drawn as random subsets of the samples **without replacement**, then this algorithm is known as **Pasting**.

  R's sample(nrow(d.f))

- If samples are **drawn with replacement**, then the method is known as **Bagging**.

  R's sample(nrow(d.f), replace=TRUE)

- When random subsets of the dataset are drawn as random subsets of the features, then the method is known as **Random Subspaces**.

- When base estimators are built on subsets of both samples and features, then the method is known as **Random Patches**.

Random Forest

A **random forest** is a meta estimator that fits a number of decision tree classifiers on various **sub-samples of the dataset** (#rows is n but #cols$= m < p$) and uses averaging to improve the predictive accuracy and control over-fitting.

mtry = 2

Original					Bootstrap1			Bootstrap2			Bootstrap3		
Index	C1	C2	C3	C4	D1.Index	C1	C2	D2.Index	C4	C2	D3.Index	C3	C1
1					9			1			7		
2					9			7			5		
3					3			5			6		
4					8			10			9		
5					10			7			2		
6					7			9			5		
7					10			9			8		
8					9			10			2		
9					3			7			1		
10					4			5			9		
					h_D1			h_D2			h_D3		
Predict(X)					A			A			B		

Random Forest (cont)

According to the paper "Bias in random forest variable importance measures: Illustrations, sources and a solution", bioinformatics face the so-called "small n large p"' problem: Usual data sets in genomics often contain 10^3 of genes or markets that serve as predictor variables but only for a comparatively small number n of subjects or tissue types.

Logistic regression and neural networks won't work but the random forest works fine.

Random Forest in R

R's `randomForest` package provides:

```
randomForest(formula, data=NULL, ..., subset, na.action=na.fail)

randomForest(x, y=NULL,  xtest=NULL, ytest=NULL, ntree=500,
  mtry=if (!is.null(y) && !is.factor(y))
  max(floor(ncol(x)/3), 1) else floor(sqrt(ncol(x))),
  replace=TRUE, classwt=NULL, cutoff, strata,
  sampsize = if (replace) nrow(x) else ceiling(.632*nrow(x)),
  nodesize = if (!is.null(y) && !is.factor(y)) 5 else 1,
  maxnodes = NULL,
  importance=FALSE, localImp=FALSE, nPerm=1,
  proximity, oob.prox=proximity,
  norm.votes=TRUE, do.trace=FALSE,
  keep.forest=!is.null(y) && is.null(xtest), corr.bias=FALSE,
  keep.inbag=FALSE, ...)
```

It implements Breiman's random forest algorithm (based on Breiman and Cutler's original Fortran code) for classification and regression.

Random Forest in R (cont)

An implementation of the random forest and bagging ensemble algorithms utilizing ctree (conditional inference trees) as base learners. The variable selection of ctree is unbiased while the randomForest which is based on CART trees is highly biased.

```
party::cforest(formula, data=list(), subset=NULL, weights=NULL,
    controls = cforest_unbiased(),
    xtrafo = ptrafo, ytrafo = ptrafo, scores = NULL)

partykit::cforest(formula, data, weights, subset, offset, cluster,
    strata, na.action = na.pass,
    control = ctree_control(teststat = "quad", testtype = "Univ",
        mincriterion = 0, saveinfo = FALSE, ...),
    ytrafo = NULL, scores = NULL, ntree = 500L,
    perturb = list(replace = FALSE, fraction = 0.632),
    mtry = ceiling(sqrt(nvar)), applyfun = NULL, cores = NULL,
    trace = FALSE, ...)
```

Random Forest in Python

Python's sklearn.ensemble provides RandomForestClassifier and RandomForestRegressor.

```
sklearn.ensemble.RandomForestClassifier(
  n_estimators=100, criterion='gini',
  max_depth=None, min_samples_split=2,
  min_samples_leaf=1, min_weight_fraction_leaf=0.0,
  max_features='auto', max_leaf_nodes=None,
  min_impurity_decrease=0.0, min_impurity_split=None,
  bootstrap=True, oob_score=False, n_jobs=None,
  random_state=None, verbose=0,
  warm_start=False, class_weight=None,
  ccp_alpha=0.0, max_samples=None)
```

Random Forest in Python (cont)

Remark on the class RandomForestClassifier:

- The default values for the parameters controlling the size of the trees (e.g. “max_depth“, “min_samples_leaf“, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.
- The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data, “max_features=n_features“ and “bootstrap=False“, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, “random_state“ has to be fixed.

Outline

Boosting Methods

In boosting, multiple models are **trained sequentially** and each model learns from the errors of its predecessors.

The two famous boosting algorithms are

- AdaBoost = Adaptive Boosting [Freund et al., 1996, Freund and Schapire, 1997]
- Gradient Boosting = Gradient Descent + Boosting [Friedman et al., 2000, Friedman, 2001]

Other variations: XGBoost (eXtreme Gradient Boosting), LightGBM

AdaBoost

Given $(x_1, y_1), \ldots, (x_n, y_n)$, $x_i \in \mathbb{R}^p$ (numeric inputs), $y_i \in \{-1, 1\}$ (not 0,1 like logistic regression).

Let $D_1[i] = \frac{1}{n}$ (assume every row is equally important)

For t=1,. . . ,T:

- Use $(x, y) \sim D_t$ to obtain $h_t : X \to \{-1, 1\}$
- Calculate the error $\epsilon_t = \mathbb{E}[h_t(x_i) \neq y_i]$
- Let $\alpha_t = 0.5 \ln((1 - \epsilon_t)/\epsilon_t)$
- $$D_{t+1}[i] := \frac{D_t[i]}{Z_t} \times \begin{cases} e^{-\alpha_t}, & h_t(x_i) = y_i \\ e^{\alpha_t}, & h_t(x_i) \neq y_i \end{cases}$$
  $$= D_t[i] \exp(-\alpha_t y_i h_t(x_i))/Z_t, \ i = 1, \cdots, n.$$
  Here: Z_t is the sum to turn $D_{t+1}[i]$ to probability

AdaBoost (cont)

The trained AdaBoost model is

$$H(t) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\text{x})\right).$$

In Adaboost, "shortcomings" are identified by high-weight datapoints.

Data

Index	X1	X2	X3	Y	D1[i]	Err
1					0.1	
2					0.1	
3					0.1	Yes
4					0.1	
5					0.1	
6					0.1	Yes
7					0.1	
8					0.1	
9					0.1	
10					0.1	

Stumps:

Choose X1, X2, X3 for h_1

Let's say X2 wins out

X2	Y	D2[i] formula	D2[i]'	D2[i]
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(J17)	0.2	0.25
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(J17)	0.2	0.25
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(-J17)	0.05	0.063
		'=G8*EXP(-J17)	0.05	0.063

h_1

0.8

err1 0.2

Z_1

alpha_1=

0.693147

Choose X1, X2, X3 for h_2

Let's say X3 wins out

Index	X3	Y
3		
6		
3		
6		
6		
1		
10		
9		
7		
4		

h_2

AdaBoost in R

The adabag library has many dependencies due to it reliance on caret library and rpart.

```
boosting(formula, data, boos = TRUE,
  mfinal = 100, coeflearn = 'Breiman',
  control,...)
```

Here, boos: if TRUE (by default), a bootstrap sample of the training set is drawn using the weights for each observation on that iteration. If FALSE, every observation is used with its weights.

coeflearn: if Breiman (by default), alpha=$0.5\ln((1-\epsilon_t)/\epsilon_t$ is used. If Freund, alpha=$\ln((1-\epsilon_t)/\epsilon_t)$ is used. In both cases the AdaBoost.M1 algorithm is used and 'alpha' is the weight updating coefficient. On the other hand, if coeflearn is 'Zhu' the SAMME algorithm is implemented with alpha=$\ln((1-\epsilon_t)/\epsilon_t)+\ln(T-1)$.

AdaBoost in R (cont)

Simple Example:

```
library(adabag)

set.seed(1)
d.f   = iris
form  = Species~.
idx   = sample(nrow(d.f), 0.7*nrow(d.f))
train = d.f[ idx,]
test  = d.f[-idx,]
model_adaboost = boosting(form, data=train,
  boos=TRUE, mfinal=50)
yhat  = predict(model_adaboost, test)
print(table(yhat$class, test$Species))
```

AdaBoost in Python

Python's Scikit-learn only supports the SAMME AdaBoost algorithm (Zhu, H. Zou, S. Rosset, T. Hastie, "Multi-class AdaBoost", 2009) and AdaBoost.R2 algorithm (Drucker, "Improving Regressors using Boosting Techniques", 1997).

```
class sklearn.ensemble.AdaBoostClassifier(
  base_estimator=None, *, n_estimators=50,
  learning_rate=1.0, algorithm='SAMME.R',
  random_state=None)

class sklearn.ensemble.AdaBoostRegressor(
  base_estimator=None, *, n_estimators=50,
  learning_rate=1.0, loss='linear',
  random_state=None)
```

Gradient Boosting

Let the loss function $L(y_i, \gamma)$ be differentiable.

Initialise model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

For $m = 1$ to M:

- Compute so-called pseudo-residuals:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

- Fit a base learner (e.g. tree) closed under scaling $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.

Gradient Boosting (cont)

- Compute multiplier γ_m by solving the 1D optimisation problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

- Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

The final trained Gradient Boosting model is $F_M(x)$.

Gradient Boosting (cont)

In Gradient Boosting, "shortcomings" are identified by gradients.

	Data					Residue	h1(X) trains on green colour		
Index	X1	X2	X3	Y	F0(X)	Y-F0(X) ri1	F1(X)=F0(X)+h1(X)	Y-F1(X)	ri2
1				10	55	-45 r11			
2				20	55	-35 r21			
3				30	55	-25 r31			
4				40	55	-15 r41			
5				50	55	-5 r51			
6				60	55	5 r61			
7				70	55	15 r71			
8				80	55	25 r81			
9				90	55	35 r91			
10				100	55	45 r10,1			

h2(X) trains on new residue ri2

...

Gradient Boosting in R

The Gradient Boosting 'classifier' and 'regressor' (input must be numeric):

```
gbm(formula = formula(data), distribution="bernoulli",
  data = list(), weights, var.monotone = NULL,
  n.trees = 100, interaction.depth = 1,
  n.minobsinnode = 10, shrinkage = 0.1,
  bag.fraction = 0.5, train.fraction = 1,
  cv.folds = 0, keep.data = TRUE,
  verbose = FALSE,
  class.stratify.cv = NULL,
  n.cores = NULL
)
```

Gradient Boosting in Python

The Gradient Boosting Tree classifier:

```
class sklearn.ensemble.GradientBoostingClassifier(*,
  loss='deviance', learning_rate=0.1, n_estimators=100,
  subsample=1.0, criterion='friedman_mse',
  min_samples_split=2, min_samples_leaf=1,
  min_weight_fraction_leaf=0.0, max_depth=3,
  min_impurity_decrease=0.0,
  init=None, random_state=None,
  max_features=None, verbose=0, max_leaf_nodes=None,
  warm_start=False, validation_fraction=0.1,
  n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0)
```

Gradient Boosting in Python

The Gradient Boosting Tree regressor:

```
class sklearn.ensemble.GradientBoostingRegressor(*,
  loss='squared_error', learning_rate=0.1,
  n_estimators=100,
  subsample=1.0, criterion='friedman_mse',
  min_samples_split=2, min_samples_leaf=1,
  min_weight_fraction_leaf=0.0, max_depth=3,
  min_impurity_decrease=0.0,
  init=None, random_state=None, max_features=None,
  alpha=0.9, verbose=0, max_leaf_nodes=None,
  warm_start=False, validation_fraction=0.1,
  n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0)
```