Data Preprocessing

Data Preprocessing: Feature Engineering

- Feature Engineering
    - Dealing with Missingness
        - Imputation
    - Feature Filtering
    - Numerical Feature Engineering
        - Normalizing, Standardization
    - Categorical Feature Engineering
        - Lumping, One-hot/Dummy Encoding, Label Encoding
- Sequential Steps & Data Leakage

Data Preprocessing: Feature Engineering

- Data preprocessing: addition, deletion, or transformation of data
- Can make or break an algorithm’s predictive ability
- Deserves continued focus and attention

Prereqs: Packages and Data (Ames Housing)

```r
library(dplyr)
library(ggplot2)
library(rsample)
library(recipes)
```

```r
# ames data
ames <- AmesHousing::make_ames()

# split data
set.seed(123)
split <- initial_split(ames, strata = "Sale_Price")
ames_train <- training(split)
```

Data Preprocessing: Dealing with Missingness

- Missingness: most common data quality concerns
- **In real life, usually two reasons:**
  - *Informative missingness:* underlying cause for the missing value
  - *Missingness at random:* occur independent of the data collection process
- **Two steps:**
  - Visualization
  - Imputation

Data Preprocessing: Dealing with Missingness

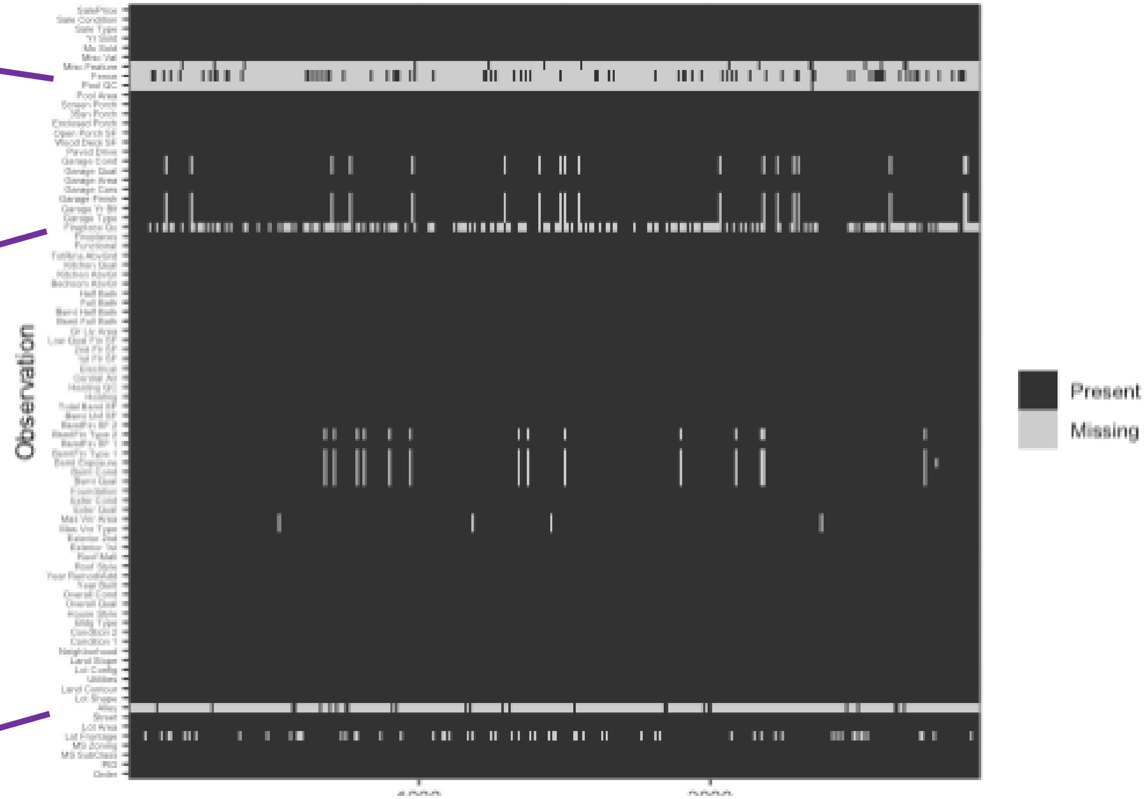
- **Visualization** of Missing Values
- Understand distribution of missing values in the datasets
- Help determine best approach to deal with them

- Efficient way to visualize: **Heat maps**

Data Preprocessing: Dealing with Missingness

Visualization of Missing Values

Misc Feature
Fence
Pool QF



Fireplace Qual

Misc Feature, Fence, Pool QF,
Fireplace Qual, Alley: Majority missing

Alley

Data Preprocessing: Dealing with Missingness

Imputation of Missing Values

- The process of replacing a missing value with a “best guess” value
- Should be one of the first feature engineering steps to take

- Three approaches (Will look at each in detail)
  - 1. Estimated statistic
  - 2. K-nearest neighbor (KNN)
  - 3. Tree-based (bagged trees)

Data Preprocessing: Dealing with Missingness

Imputation of Missing Values

Three approaches:

1. Estimated statistic

- Descriptive statistics
    - e.g. mean/median (for numeric), mode (for categorical)
    - Use that value to replace NAs
- Computationally efficient
- Does not consider any other attributes for a given observation when imputing

Data Preprocessing: Dealing with Missingness

Imputation of Missing Values

Three approaches:

2. K-nearest neighbor (KNN)

- Identify observations that are most similar, based on other available features
    - Use that value to replace NAs
- More accurate
- May be computationally burdensome

3. Tree-based (bagged trees)

- Similar to KNN, but using decision trees / bagged trees
- More accurate
- May be computationally burdensome

Imputation

- Estimated statistic
  - i.e. mean, median, mode
- K-nearest neighbor
- Tree-based (bagged trees)

step_impute_mean()
step_impute_median()
step_impute_mode()
step_impute_knn() (default k = 5)
step_impute_bag()

Data Preprocessing: Feature Filtering

- Common for datasets to have hundreds/thousands of features
- Model with too many features:
  - Non-informative features – affect model generalization
  - Harder to interpret – effect of important variables obscured
  - Costly to compute – significantly more time to train models

Data Preprocessing: Feature Filtering

- Deal with non-informative features: Eliminate Zero and near-zero variance variables

- Eliminate zero variance variable: *step_zv()*

- Eliminate near-zero variance variable:
  - *step_nzv(, freq_cut = B, unique_cut = A)*
  - Eliminate variables which satisfy **both of the followings**:
    - %unique values over the sample size ___ A%
    - Ratio of [most prevalent value's freq] to [second most prevalent value's freq] ___ B

- R default: A = 10, B = 95/5

Data Preprocessing: Feature Filtering

```
> caret::nearZeroVar(ames_train, saveMetrics = TRUE)
                     freqRatio percentUnique zeroVar   nzv
MS_SubClass           1.871287    0.73206442   FALSE     ?
MS_Zoning             4.865854    0.34163006   FALSE
Lot_Frontage          1.643216    5.75890678   FALSE
Lot_Area              1.000000   71.79111762   FALSE
Street              226.666667    0.09760859   FALSE
Alley                24.253165    0.14641288   FALSE
Lot_Shape             1.829060    0.19521718   FALSE
Land_Contour         19.500000    0.19521718   FALSE
Utilities          1023.000000    0.14641288   FALSE
Lot_Config            4.018919    0.24402147   FALSE
Land_Slope           22.159091    0.14641288   FALSE
```

Data Preprocessing: Feature Engineering

- Numeric Feature Engineering: Three issues
  - **Skewness**
  - **Outliers**
  - **Wide range in feature distribution**

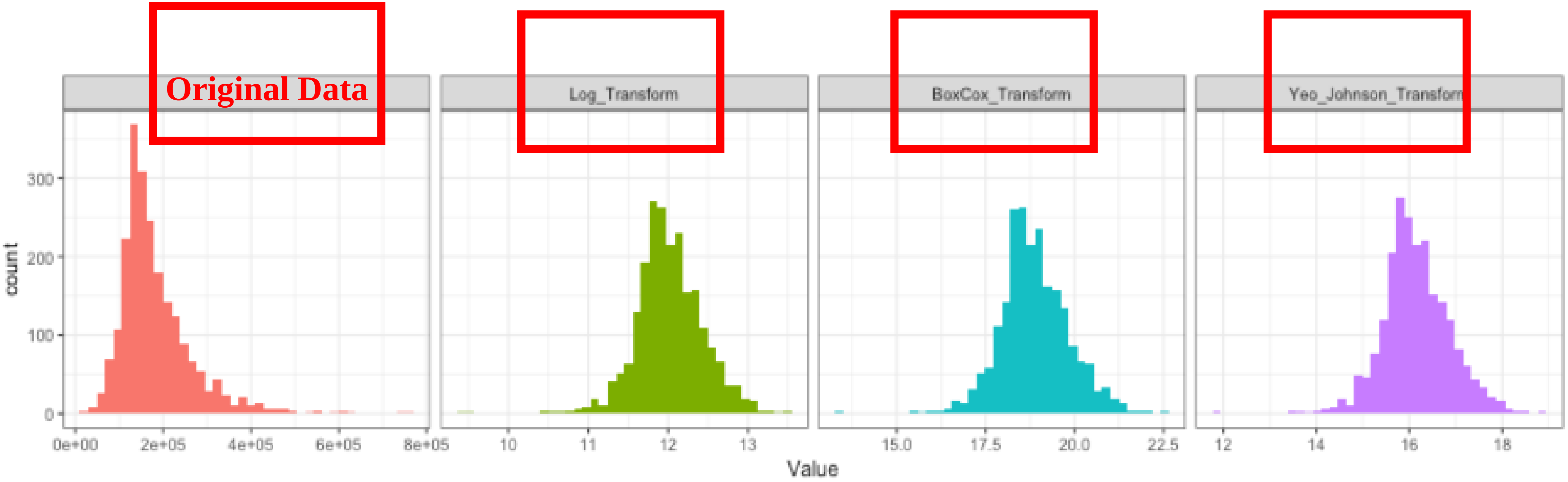
- Solutions
  - Deal with **Skewness, outliers**:
    - **Normalizing**
  - Deal with **wide range**:
    - **Standardization**

Data Preprocessing: Feature Engineering

Numeric Feature Engineering: Normalizing

```
step_log()
step_BoxCox()
step_YeoJohnson()
```

- **Response +ve: step_log(), step_BoxCox(). Box Cox transformation**
- **Response NOT +ve, but -ve response values are small: step_log(, offset = x) → adds offset to the value prior to applying log**
- **Otherwise: step_YeoJohnson(), Yeo Johnson Transformation**



Data Preprocessing: Feature Engineering

Numeric Feature Engineering: Standardization

- Always a good idea to standardize the features.
- Centering: numeric variables have zero mean
- Scaling: numeric variables have unit variance provides a common comparable unit of measure across all variables.

Data Preprocessing: Feature Engineering

Numeric Feature Engineering: Standardization

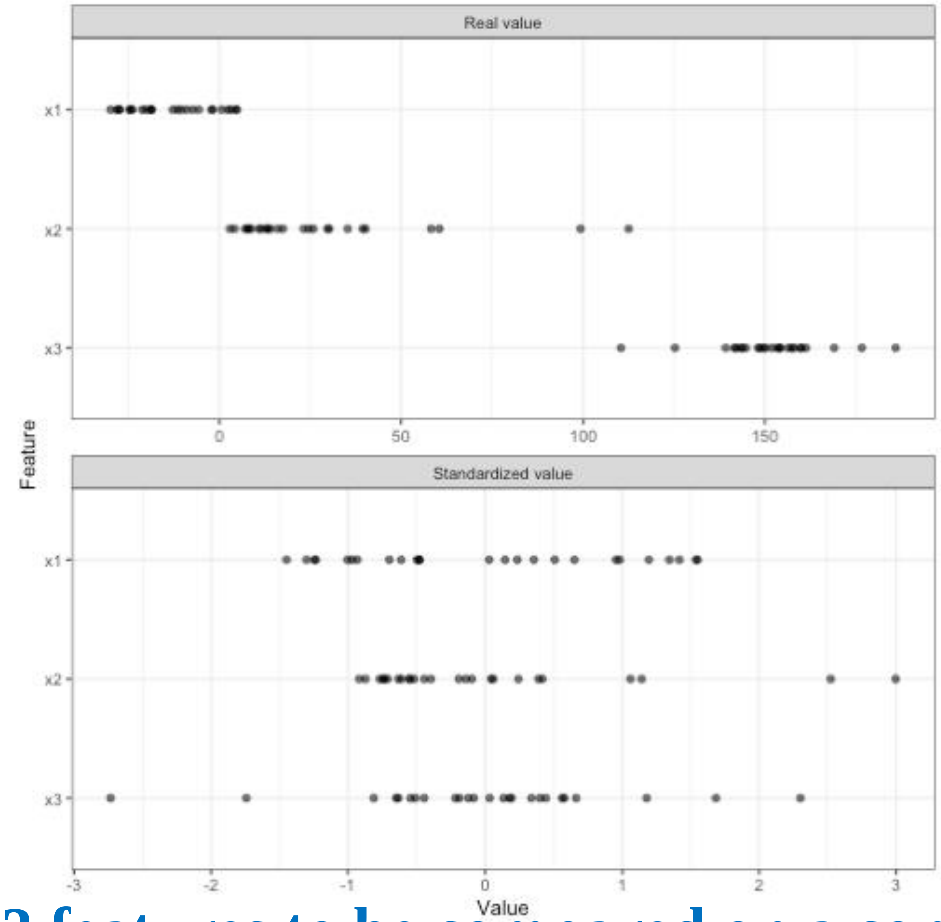
step_center()

step_scale()

x_1

x_2

x_3



x_1

x_2

x_3

Before Centering & Scaling

After Centering & Scaling

Allow the 3 features to be compared on a common value scale, regardless of real value diff

Data Preprocessing: Feature Engineering

Categorical Feature Engineering: Two Issues

- **Too few observations:**
  - Features contain levels with too few observations
- **Numerical input requirement:**
  - Many models require numerical inputs
- Solutions
  - Deal with **too few observations**:
    - **Lumping**
  - Deal with **numerical input requirement**:
    - **One-hot/Dummy/Label Encoding**

Data Preprocessing: Feature Engineering

Categorical Feature Engineering: Lumping

Relevant to: features contain **levels with too few observations**
e.g. Feature: neighborhood

- of the 28 unique neighborhoods in Ames housing data, several only have a few observations

```
count(ames_train, Neighborhood) %>% arrange(n)
## # A tibble: 28 x 2
##    Neighborhood                                  n
##    <fct>                                     <int>
##  1 Landmark                                      1
##  2 Green_Hills                                   2
##  3 Greens                                        7
##  4 Blueste                                       9
##  5 Northpark_Villa                              17
##  6 Briardale                                    18
##  7 Veenker                                      20
##  8 Bloomington_Heights                          21
##  9 South_and_West_of_Iowa_State_University      30
## 10 Meadow_Village                               30
## # … with 18 more rows
```

Data Preprocessing: Feature Engineering

Categorical Feature Engineering: Lumping

Collapse all levels that are observed in less than x% of the training sample into an “other” category. (e.g. x% = 1%)

use step_other() to do so.

Lumping should be used sparingly: may result in loss in model performance

Data Preprocessing: Feature Engineering

Categorical Feature Engineering: One-hot/Dummy Encoding

Transform categorical variables into numeric forms
(Some meta engines automate this process, e.g. h2o, caret)

One-hot encoding: transpose categorical variables s.t. each level of the feature is represented as a boolean value

Dummy encoding: similar to one-hot encoding but drop one of the levels

Note: one-hot encoding has **perfect collinearity** in features, while dummy encoding does not have

Data Preprocessing: Feature Engineering

Categorical Feature Engineering: One-hot/Dummy Encoding

id	x
1	a
2	c
3	b
4	c

One-hot/Dummy Encoding

step_dummy()

Dummy encoding

id	x.b	x.c
1	0	0
2	0	1
3	1	0
4	0	1

One-hot encoding

id	x.a	x.b	x.c
1	1	0	0
2	0	0	1
3	0	1	0
4	0	0	1

Data Preprocessing: Feature Engineering

Categorical Feature Engineering: One-hot/Dummy Encoding

- One-hot and dummy encoding are NOT good when:
  - have lots of categorical features
  - features have high cardinality
  - have ordinal (ranked) features

- E.g. Feature: Overall_Qual (ranges from Very_Poor to Very_Excellent)

```
ames_train %>% select(matches("Qual|QC|Qu"))
## # A tibble: 2,199 x 9
##    Overall_Qual Exter_Qual Bsmt_Qual Heating_QC Low_Qual_Fin_SF
##    <fct>        <fct>      <fct>     <fct>                <int>
##  1 Above_Avera… Typical    Typical   Typical                  0
##  2 Good         Good       Typical   Excellent                0
##  3 Average      Typical    Good      Good                     0
##  4 Above_Avera… Typical    Typical   Excellent                0
##  5 Very_Good    Good       Good      Excellent                0
##  6 Very_Good    Good       Good      Excellent                0
##  7 Good         Typical    Typical   Good                     0
##  8 Above_Avera… Typical    Good      Good                     0
##  9 Above_Avera… Typical    Good      Excellent                0
## 10 Good         Typical    Good      Good                     0
## # … with 2,189 more rows, and 4 more variables: Kitchen_Qual <fct>,
## #   Fireplace_Qu <fct>, Garage_Qual <fct>, Pool_QC <fct>
```

Data Preprocessing: Feature Engineering

Categorical Feature Engineering: Label Encoding

Label encoding

- Pure numeric conversion of levels of a categorical variable
- Deal with high-cardinality ordinal (ranked) features

```
count(ames_train, Overall_Qual)
## # A tibble: 10 x 2
##    Overall_Qual       n
##    <fct>          <int>
##  1 Very_Poor          3
##  2 Poor              12
##  3 Fair              29
##  4 Below_Average    166
##  5 Average          607
##  6 Above_Average    553
##  7 Good             458
##  8 Very_Good        266
##  9 Excellent         81
## 10 Very_Excellent    24
```

Label (Ordinal) Encoding

step_integer()

```
## # A tibble: 10 x 2
##    Overall_Qual     n
##           <dbl> <int>
##  1            1     3
##  2            2    12
##  3            3    29
##  4            4   166
##  5            5   607
##  6            6   553
##  7            7   458
##  8            8   266
##  9            9    81
## 10           10    24
```

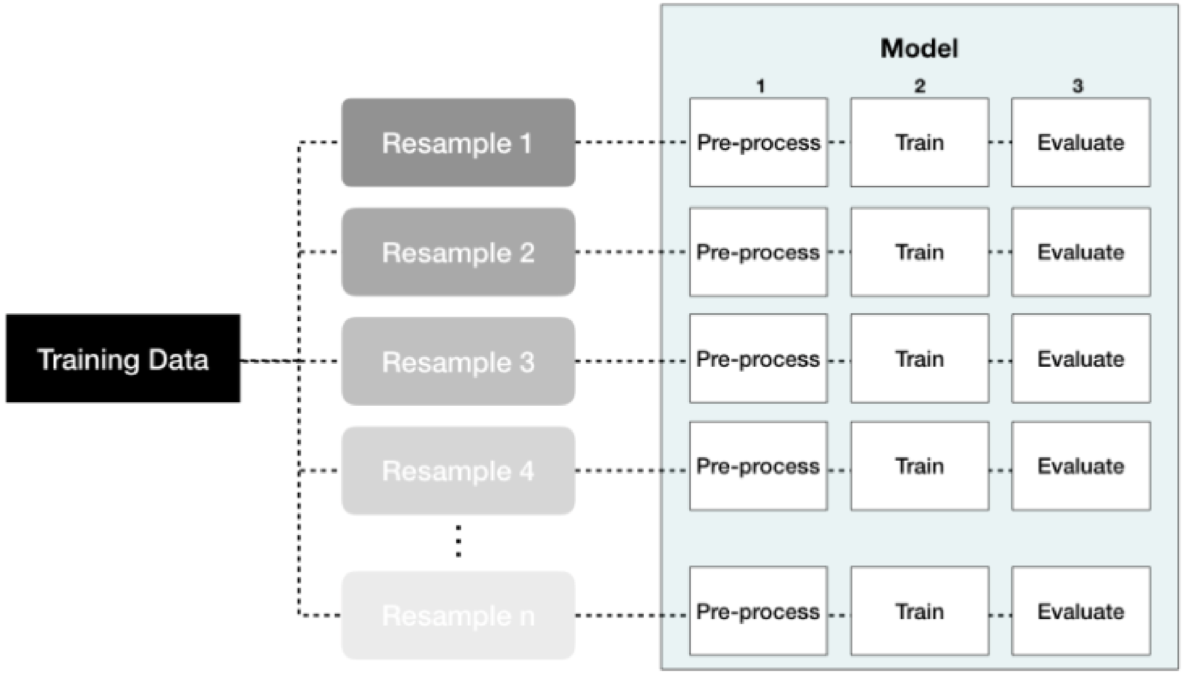
Data Preprocessing: Sequential Steps

1. Filter out zero or near-zero variance features
2. Perform imputation if required
3. Normalize to resolve numeric feature skewness
4. Standardize (center and scale) numeric features
5. Create dummy encoded features

Why step 3 before step 4?
Why step 4 before step 5?
If want to do lumping, should it be before or after step 5?

Data Preprocessing: Data Leakage

- Data leakage: info from outside the training data used to create the model.
- Often occurs during the data preprocessing period
- Feature engineering should be done **after resampling**



Data Preprocessing: Data Leakage

Data leakage: Info from outside the training data used to create the model.

E.g. Train data (x_i, y_i), i = 1…n, where x_i's have mean of 5 and var of 2

$$\text{Standardization: } \tilde{x} = (x - 5)/sqrt(2)$$

Suppose the model we obtain from train data is:

$$\hat{y} = 2\,\tilde{x} + 3$$

Now suppose we have validation data (x_j, y_j), j = 1…m, x_j's have mean of 4 and var of 3

For x, do we predict

$$\hat{y} = \frac{2(x-5)}{sqrt(2)} + 3 \quad \text{OR} \quad \hat{y} = \frac{2(x-4)}{sqrt(3)} + 3$$

Bring Memory back to 1st Class: KNN implementation in R

Load Data

```
# Loading iris dataset
iris.rawData <- iris

# Viewing iris dataset structure and attributes
summary(iris.rawData)
```

```
> summary(iris.data)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species  
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa    :50  
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolor:50  
 Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica :50  
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199                  
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800                  
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500                  
```

3 Classes of Iris Species: Setosa, versicolor, virginica

Data Standardization

```
# standardize data
standardize <- function(x) {
  return ( x - mean(x) )/( sd(x) )
}
# Only standardize the first 4 columns (5th column is label)
iris.standardizeData = iris.rawData
for(i in seq(1,4)){
  iris.standardizeData[,i] = standardize(iris.rawData[,i])
}
# Split into train & test set
set.seed(123)
split  <- rsample::initial_split(iris.standardizeData, prop = 0.7, strata = "Species")
iris.train  <- rsample::training(split)
iris.test   <- rsample::testing(split)
iris.trainFeatMat = iris.train[,1:4]
iris.trainLabel <- iris.train[,5]
iris.testFeatMat <- iris.test[,1:4]
iris.testLabel <- iris.test[,5]
```

Data Preprocessing: Implementation with recipe

```
blueprint <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_nzv(all_nominal())  %>%
  step_integer(matches("Qual|Cond|QC|Qu")) %>%
  step_center(all_numeric(), -all_outcomes()) %>%
  step_scale(all_numeric(), -all_outcomes())

prepare <- prep(blueprint, training = ames_train)

baked_train <- bake(prepare, new_data = ames_train)
baked_test <- bake(prepare, new_data = ames_test)
```

```
> c(ames_train[1:5,c('Lot_Frontage')],baked_train[1:5,c('Lot_Frontage')])
$Lot_Frontage
[1] 21 21 24 50 70

$Lot_Frontage
[1] -1.1320859 -1.1320859 -1.0408930 -0.2505543  0.3573985
```

Data Preprocessing: Implementation with recipe

Some Code Examples

Data Preprocessing: Putting the Process Together

1. Split into training vs testing data
2. **Create feature engineering blueprint**
3. Specify a resampling procedure
4. Create our hyperparameter grid
5. Execute grid search
6. Evaluate performance

```
# 1. stratified sampling with the rsample package
set.seed(123)
split  <- initial_split(ames, prop = 0.7, strata = "Sale_Price")
ames_train  <- training(split)
ames_test   <- testing(split)

# 2. Feature engineering
blueprint <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_nzv(all_nominal()) %>%
  step_integer(matches("Qual|Cond|QC|Qu")) %>%
  step_center(all_numeric(), -all_outcomes()) %>%
  step_scale(all_numeric(), -all_outcomes()) %>%
  step_dummy(all_nominal(), -all_outcomes(), one_hot = TRUE)

# 3. create a resampling method
cv <- trainControl(
  method = "repeatedcv",
  number = 10,
  repeats = 5
  )
```

Data Preprocessing: Putting the Process Together

1. Split into training vs testing data
2. **Create feature engineering blueprint**
3. Specify a resampling procedure
4. Create our hyperparameter grid
5. Execute grid search
6. Evaluate performance

```
# 4. create a hyperparameter grid search
hyper_grid <- expand.grid(k = seq(2, 25, by = 1))

# 5. execute grid search with knn model,     use RMSE as
preferred metric
knn_fit <- train(
  blueprint,
  data = ames_train,
  method = "knn",
  trControl = cv,
  tuneGrid = hyper_grid,
  metric = "RMSE"
  )

# 6. evaluate results
# print model results
knn_fit
```

Data Preprocessing: Putting the Process Together

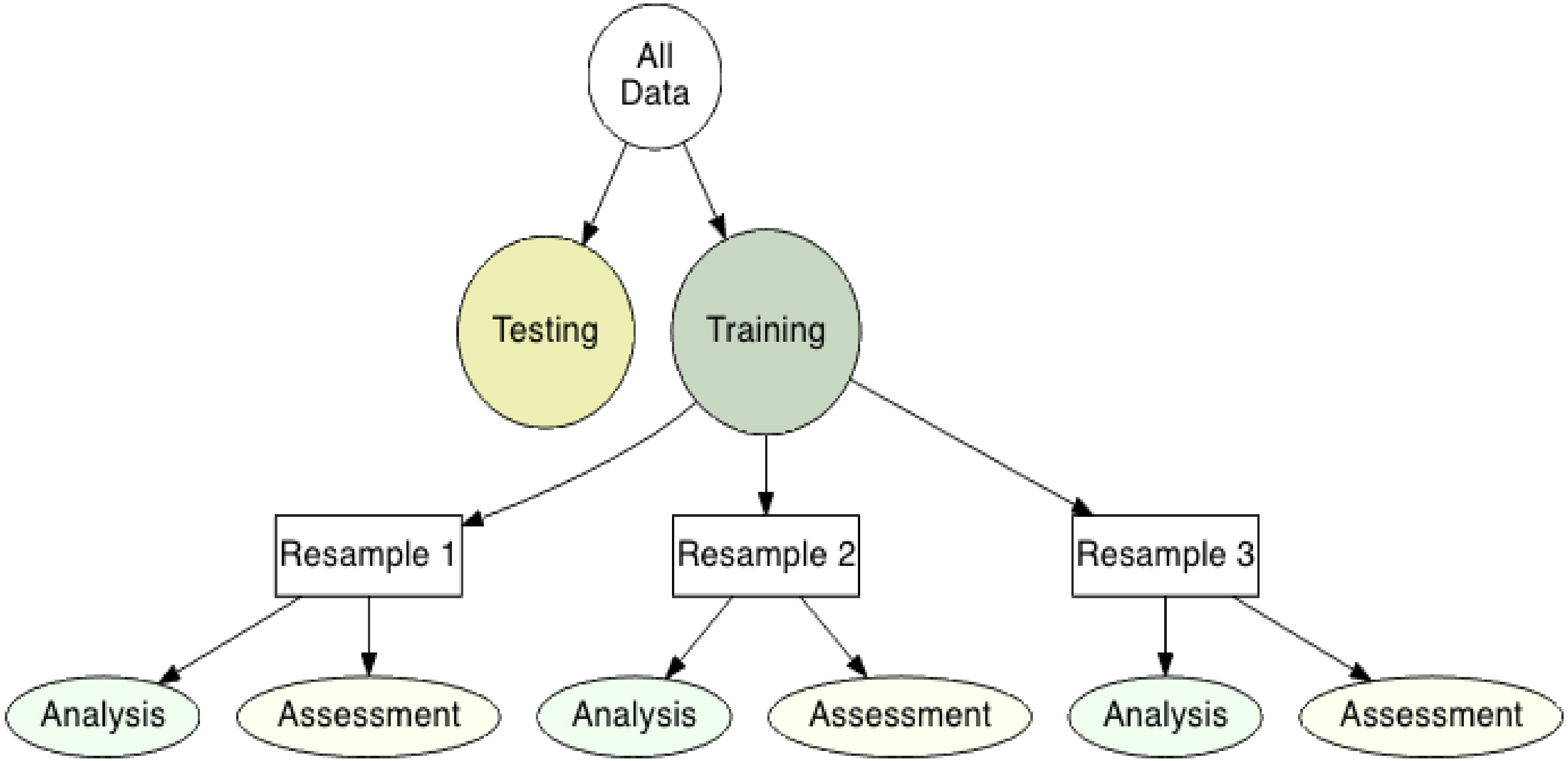
Note:

Can use **“predict”** on the resulting model object: knn_fit

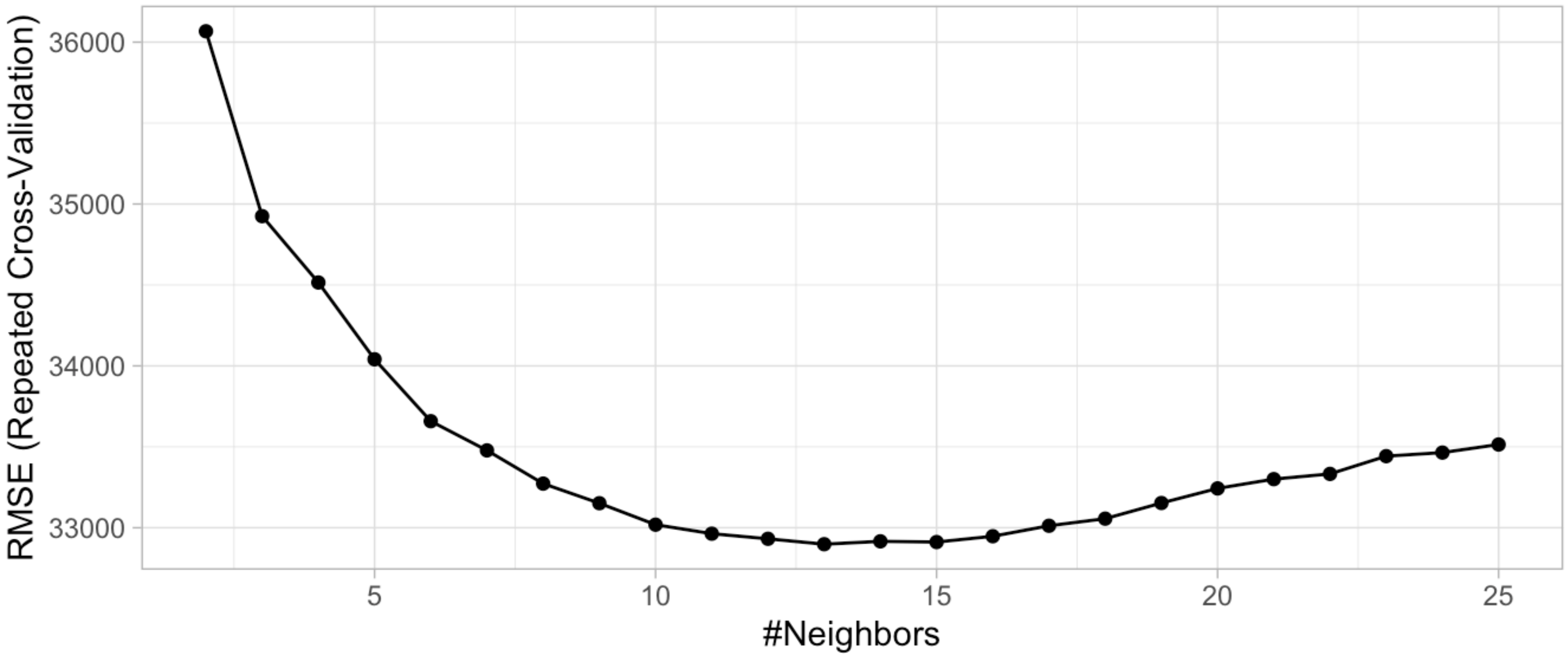
The model object is obtained by using **all the training data**

```
# 4. create a hyperparameter grid search
hyper_grid <- expand.grid(k = seq(2, 25, by = 1))

# 5. execute grid search with knn model,     use RMSE as
preferred metric
knn_fit <- train(
  blueprint,
  data = ames_train,
  method = "knn",
  trControl = cv,
  tuneGrid = hyper_grid,
  metric = "RMSE"
  )

# 6. evaluate results
# print model results
knn_fit
```

Revisit


All
Data
Testing
Training
Resample 1
Resample 2
Resample 3
Analysis
Assessment
Analysis
Assessment
Analysis
Assessment

Data Preprocessing: Putting the process together



New RMSE: ~33,000
Feature engineering alone reduced our error by >$10,000!

The ML Process: Summary

Sub-topic	Concepts	Demonstrated Package/Func
Data Splitting	Stratified Sampling	rsample::initial_split rsample::training rsample::testing
Specify Train Procedure	Direct vs Meta Engine	Meta: caret::train Direct: e.g. glm
Resampling Method	K-fold CV vs Bootstrap	caret::train (specify trControl)
Hyperparameter Tuning	Grid Search	expand.grid (built-in)
Model Performance Metric	e.g. RMSE, Accuracy	caret::train (specify metric = “RMSE” for regression / metric = “Accuracy” for classification)

Data Preprocessing: Summary

Sub-topic	Concepts	Demonstrated Package/Func
Normalizing	Variable Transform	recipes::step_log(), step_boxcox(), step_YeoJohnson()
Dealing with Missingness	Imputation	recipes::step_impute_mean(), step_impute_median(), step_impute_mode(), step_impute_knn(), step_impute_bag()
Feature Filtering	Near-zero-variance variable	recipes::step_zv(), step_nzv()
Numerical Feature Engineering	Standardization	recipes::step_center(), step_scale()
Categorical Feature Engineering	Lumping, One-hot/Dummy Encoding, Label Encoding	recipes::step_other, step_dummy(), step_integer()

Why data scientists think data cleaning is the most difficult step?

End

```r
blueprint <- recipe(Sale_Price ~ ., data  = ames_train) %>%
  step_nzv(all_nominal()) %>%
  step_integer(matches("Qual|Cond|QC|Qu")) %>%
  step_center(all_numeric(), -all_outcomes()) %>%
  step_scale(all_numeric(), -all_outcomes()) %>%
  step_dummy(all_nominal(), -all_outcomes(), one_hot = TRUE)

# ----------------New: manually implemented cross-validation procedure--
rmse_records <- c()
k_max <- 25
repeats <- 5
cv_number <- 10
```

```r
for (k in 2:k_max){ #For each parameter set
  rmse_k_total <- 0

  for (rep in 1:repeats){ #For each repeat
    rmse_rep_total <- 0
    ames_train_in_rep <- ames_train[sample(nrow(ames_train)),] #Randomly shuffle the data
    folds <- cut(seq(1,nrow(ames_train_in_rep)),breaks=cv_number,labels=FALSE) #Create equal size

    for (i in 1:cv_number){  #For each fold
```

Train on folds ≠ i

Validate on fold i, get rmse

Update rmse_rep_total = rmse_rep_total + rmse

```r
    }

    rmse_rep <- rmse_rep_total/cv_number
    rmse_k_total <- rmse_k_total + rmse_rep
  }
  rmse_k <- rmse_k_total/repeats
  rmse_records <- append(rmse_records,rmse_k)
}

plot(2:k_max, rmse_records, xlab = "Neighbors", ylab = "RMSE")
```

```r
#Segmentdata by fold using the which() function
valIndexes <- which(folds==i,arr.ind=TRUE)
valData <- ames_train_in_rep[valIndexes, ]
trainData <- ames_train_in_rep[-valIndexes, ]

#Preprocessing
prepare <- prep(blueprint, training = trainData)
baked_train_full <- bake(prepare, new_data = trainData)
baked_val_full <- bake(prepare, new_data = valData)

#Create true labels and remove the outcome variable
train_true_label <- baked_train_full[,names(baked_train_full) %in% c("Sale_Price"),drop=TRUE]
val_true_label <- baked_val_full[,names(baked_val_full) %in% c("Sale_Price"),drop=TRUE]
baked_train <- baked_train_full[,!names(baked_train_full) %in% c("Sale_Price")]
baked_val <- baked_val_full[,!names(baked_val_full) %in% c("Sale_Price")]

#Fit the knn model
knn_pred <- knn.reg(baked_train, baked_val, train_true_label,k = k)
knn_pred <- knn_pred$pred

#Record performance
knn_pred <- as.numeric(as.character(knn_pred))
rmse <- sqrt(mean((val_true_label - knn_pred)^2))
rmse_rep_total <- rmse_rep_total + rmse
```

Train on folds ≠ i
Validate on fold i, get rmse
Update rmse_rep_total = rmse_rep_total + rmse