

# Feature Engineering



Applied Machine Learning in R  
Pittsburgh Summer Methodology Series

Lecture 2-A

July 20, 2021

# Overview

# Feature Engineering



## Prepare the predictors for analysis

- *Extract* predictors
- *Transform* predictors
- *Re-encode* predictors
- *Combine* predictors
- *Reduce* predictor dimensionality
- *Impute* missing predictor values
- *Select* and drop predictors

# Motivation

**Features** are descriptions of the data points that help to predict the outcomes

- We may need to extract features from "raw" or "low-level" data (e.g., images)
- We may need to address issues with missing data and feature distributions

There are many potential ways to **encode** or "represent" the features/predictors

- e.g., adding, dropping, transforming, and combining predictors<sup>1</sup>
- predictor encoding can have a big **impact on predictive performance**<sup>2</sup>
- The optimal encoding depends on both the **algorithm** and the **relationships**

[1] Some algorithms can learn their own, complex feature representations

[2] Some algorithms are more sensitive to feature encoding than others

# Examples of Feature Encodings



## When an event or observation occurred

- The numeric year (2021)
- The numeric month (7)
- The numeric day of the month (20)
- The numeric day of the year (201)
- Days since a reference (*diagnosis +2*)
- The day of the week (*Tuesday*)
- The season of the year (*Summer*)
- The type of day (*weekday*)
- The presence of a holiday (*FALSE*)

# Issues to Navigate in Feature Engineering



- predictors with **non-normal** distributions
- predictors with vastly **different scales**
- predictors with extreme **outliers**
- predictors with **missing** or censored values
- predictors that are **correlated** with one another
- predictors that are **redundant** with one another
- predictors that have zero or **near-zero variance**
- predictors that are **uninformative** for a task
- predictors with **uncertainty** or unreliability

# Recipes for Feature Engineering



All of the predictor engineering steps can be done "by hand" in R

The `caret` package provides some basic convenience tools

We will be learning the **`recipes`** package from `tidymodels`

1. Initiate a recipe by declaring data and roles using `recipe()`
2. Add one or more preprocessing steps using `step_*()`
3. Prepare/estimate the preprocessing steps using `prep()`
4. Apply these steps to the training and testing data with `bake()`

# Example Dataset: `titanic`

```
# Import and preview the (custom) titanic dataset
titanic <- read.csv("titanic.csv")
glimpse(titanic)
```

```
## Rows: 963
## Columns: 7
## $ survived <chr> "yes", "yes", "no", "no", "yes", "yes", "yes", "no", "no", "y~
## $ pclass <chr> "1st", "1st", "1st", "1st", "1st", "1st", "1st", "1st", "1st", "1st"~
## $ sex <chr> "female", "male", "male", "female", "male", "female", "female"~
## $ age <dbl> 29.0000, 0.9167, 30.0000, 25.0000, 48.0000, 63.0000, 53.0000,~
## $ sibsp <int> 0, 1, 1, 1, 0, 1, 2, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0~
## $ parch <int> 0, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0~
## $ fare <dbl> 211.3375, 151.5500, 151.5500, 151.5500, 26.5500, 77.9583, 51.~
```

- `survived`: *Did passenger survive?* {FALSE, TRUE}
- `pclass`: *Passenger class* {1st, 2nd, 3rd}
- `sex`: *Passenger sex* {female, male}
- `age`: *Passenger age (years)*
- `sibsp`: *Siblings/spouses Aboard (#)*
- `parch`: *Parents/children Aboard (#)*
- `fare`: *Cost of passenger fare (\$)*



# Recipes for Feature Engineering

```
# First, let's split the data into a training and testing set
index <- caret::createDataPartition(y = titanic$survived, list = FALSE, p = 0.8)
titanic_train <- titanic[index, ]
titanic_test <- titanic[-index, ]
```

```
# Initiate a recipe for predicting survived from all other variables
titanic_recipe <- recipe(titanic, formula = survived ~ .)
```

```
# We could also be explicit with: formula = survived ~ pclass + sex + age + sibsp + parch + fare
```

```
titanic_recipe %>% print()
```

```
## Data Recipe
```

```
##
```

```
## Inputs:
```

```
##
```

```
##      role #variables
```

```
## outcome      1
```

```
## predictor     6
```

# Recipes for Feature Engineering

```
# summary() provides a bit more information than print()  
titanic_recipe %>% summary()
```

variable	type	role	source
pclass	nominal	predictor	original
sex	nominal	predictor	original
age	numeric	predictor	original
sibsp	numeric	predictor	original
parch	numeric	predictor	original
fare	numeric	predictor	original
survived	nominal	outcome	original

Now we are ready to add some preprocessing (i.e., feature engineering) steps to the recipe!

# Common Steps (Lecture Roadmap)

- **Adding predictors**
  - Calculated predictors
  - Categorical predictors
  - Interaction Terms
- **Transforming predictors**
  - Centering and Scaling
  - Addressing Non-normality
  - Adding Non-linearity
- **Reducing predictors**
  - Nero-Zero Variance
  - Multicollinearity
  - Dimensionality Reduction
- **Advanced Steps**
  - Feature Extraction
  - Dealing with Missing Values
  - Feature Selection

Adding predictors

# Calculated predictors

Some variables will need to be calculated from existing values and variables

- You may choose to score an instrument from item-level data
- You may choose to encode a predictor as the ratio of two values
- You may choose to calculate sums, means, counts, proportions, etc.

We will show you some basic steps for calculating variables within {recipes}

For more advanced/complex data wrangling, we recommend you read

- *R for Data Science: Visualize, Model, Transform, Tidy, and Import Data* by Wickham and Grolemund (book for purchase or online for free)

# Calculated predictors

```
# Add a step to the recipe to calculate new predictors from existing predictors
cp_recipe <-
  titanic %>%
  recipe(survived ~ .) %>%
  step_mutate(
    numfamily = sibsp + parch,
    over70 = age > 70
  ) %>%
  prep(training = titanic_train, log_changes = TRUE)
```

```
## step_mutate (mutate_P9l7c):
##   new (2): numfamily, over70
```

# Calculated predictors

```
# We can ask for a summary of cp_recipe to see what happened  
cp_recipe %>% summary()
```

variable	type	role	source
pclass	nominal	predictor	original
sex	nominal	predictor	original
age	numeric	predictor	original
sibsp	numeric	predictor	original
parch	numeric	predictor	original
fare	numeric	predictor	original
survived	nominal	outcome	original
numfamily	numeric	predictor	derived
over70	logical	predictor	derived

# Calculated predictors

```
# If this is the only step in our recipe, we can bake() the recipe  
# This will allow us to generate updated training and testing sets  
cp_train <- bake(cp_recipe, new_data = titanic_train)  
cp_test <- bake(cp_recipe, new_data = titanic_test)
```

```
# Let's see if it worked  
glimpse(cp_test)
```

```
## Rows: 191  
## Columns: 9  
## $ pclass    <fct> 1st, 1st, 1st, 1st, 1st, 1st, 1st, 1st, 1st, 1st, 1st, ~  
## $ sex       <fct> male, male, male, female, female, female, male, female, male~  
## $ age       <dbl> 48, 26, 25, 35, 30, 58, 42, 45, 41, 48, 41, 36, 76, 30, 39, ~  
## $ sibsp     <int> 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, ~  
## $ parch     <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, ~  
## $ fare      <dbl> 26.5500, 30.0000, 91.0792, 135.6333, 164.8667, 26.5500, 26.5~  
## $ survived  <fct> yes, yes, yes, yes, yes, yes, yes, no, yes, no, no, yes, yes, yes~  
## $ numfamily <int> 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 3, 1, 0, 2, 0, 1, 1, 1, 1, ~  
## $ over70    <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALS~
```



# Categorical predictors

Categorical predictors can be re-encoded into multiple binary (0 or 1) predictors

In `titanic`, the categorical variable **sex** takes on the value *male* or *female*

## Dummy Coding

sex	sex_male
female	0
male	1

Efficient and avoids redundancy  
Good for GLM-based methods

## One-Hot Encoding

sex	sex_female	sex_male
female	1	0
male	0	1

Simple and easy to interpret  
Good for tree-based methods

# Categorical predictors in R

```
# Add a step to the recipe to create dummy codes for pclass and sex
dc_recipe <-
  titanic %>%
  recipe(survived ~ .) %>%
  step_dummy(pclass, sex) %>%
  prep(training = titanic_train, log_changes = TRUE)
```

```
## step_dummy (dummy_wBuxU):
##   new (3): pclass_X2nd, pclass_X3rd, sex_male
##   removed (2): pclass, sex
```

[1] As a shortcut, we could also have used `step_dummy(all_nominal_predictors())`.

# Categorical predictors in R

```
# It is easy to modify this recipe for one-hot encoding
oh_recipe <-
  titanic %>%
  recipe(survived ~ .) %>%
  step_dummy(pclass, sex, one_hot = TRUE) %>%
  prep(training = titanic_train, log_changes = TRUE)
```

```
## step_dummy (dummy_JiVU7):
##   new (5): pclass_X1st, pclass_X2nd, pclass_X3rd, sex_female, sex_male
##   removed (2): pclass, sex
```

[1] Note that `step_dummy()` is used for both dummy coding and one-hot encoding (with different arguments).

# Interaction Terms

Interaction terms allow the meaning of one predictor to depend on other predictors

In this way, interaction terms allow predictor "effects" to be **contingent** or **conditional**

e.g., perhaps having parents or children on board the Titanic helps you predict survival...  
but the effects differs depending on whether the passenger is a man or a woman

Interaction terms are literally **products** (i.e., multiplications) of two or more predictors

In order to include categorical variables in interaction terms, dummy code them first

$$survived \sim parch + sex\_dummy + parch \times sex\_dummy$$

# Interaction Terms in R

```
# Add interaction term between parch and the sex_male dummy code
it_recipe <-
  titanic %>%
  recipe(survived ~ .) %>%
  step_dummy(pclass, sex) %>%
  step_interact(~ parch:sex_male) %>%
  prep(training = titanic_train, log_changes = TRUE)
```

```
## step_dummy (dummy_kFkkr):
##   new (3): pclass_X2nd, pclass_X3rd, sex_male
##   removed (2): pclass, sex
##
## step_interact (interact_9Lvhh):
##   new (1): parch_x_sex_male
```

- [1] `step_interact()` requires you to use tilde (`~`) and the colon operator (`:`) to specify interaction terms.
- [2] We also need to anticipate what the name of the dummy code variable will be and include that.
- [3] We could include higher-order interactions by adding more colons and predictors (e.g., `x1:x2:x3`).

# Interaction Terms in R

```
# Bake the recipe and preview the updated training set  
bake(it_recipe, new_data = titanic_train) %>% glimpse()
```

```
## Rows: 772  
## Columns: 9  
## $ age      <dbl> 29.0000, 0.9167, 30.0000, 25.0000, 63.0000, 53.0000, ~  
## $ sibsp    <int> 0, 1, 1, 1, 1, 2, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, ~  
## $ parch    <int> 0, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, ~  
## $ fare     <dbl> 211.3375, 151.5500, 151.5500, 151.5500, 77.9583, 51.4~  
## $ survived <fct> yes, yes, no, no, yes, yes, no, no, yes, yes, yes, ye~  
## $ pclass_X2nd <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~  
## $ pclass_X3rd <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~  
## $ sex_male  <dbl> 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, ~  
## $ parch_x_sex_male <dbl> 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, ~
```

# Comprehension check

## Question 1

**What is the correct order in which to add the {recipe} functions to a pipeline?**

- a) recipe > prep > step(s) > bake
- b) recipe > step(s) > prep > bake
- c) prep > step > bake > recipe
- d) prep > recipe > step > bake

## Question 2

**How many dummy codes are needed to encode a variable with five (5) categorical levels?**

- a) Six (6)
- b) Five (5)
- c) Four (4)
- d) One (1)

# Transforming predictors



# Normalizing

Predictors with vastly different means and SDs can cause problems for some algorithms

**Centering** a predictor involves changing its mean to 0.0

- This is accomplished by subtracting the mean from every observation

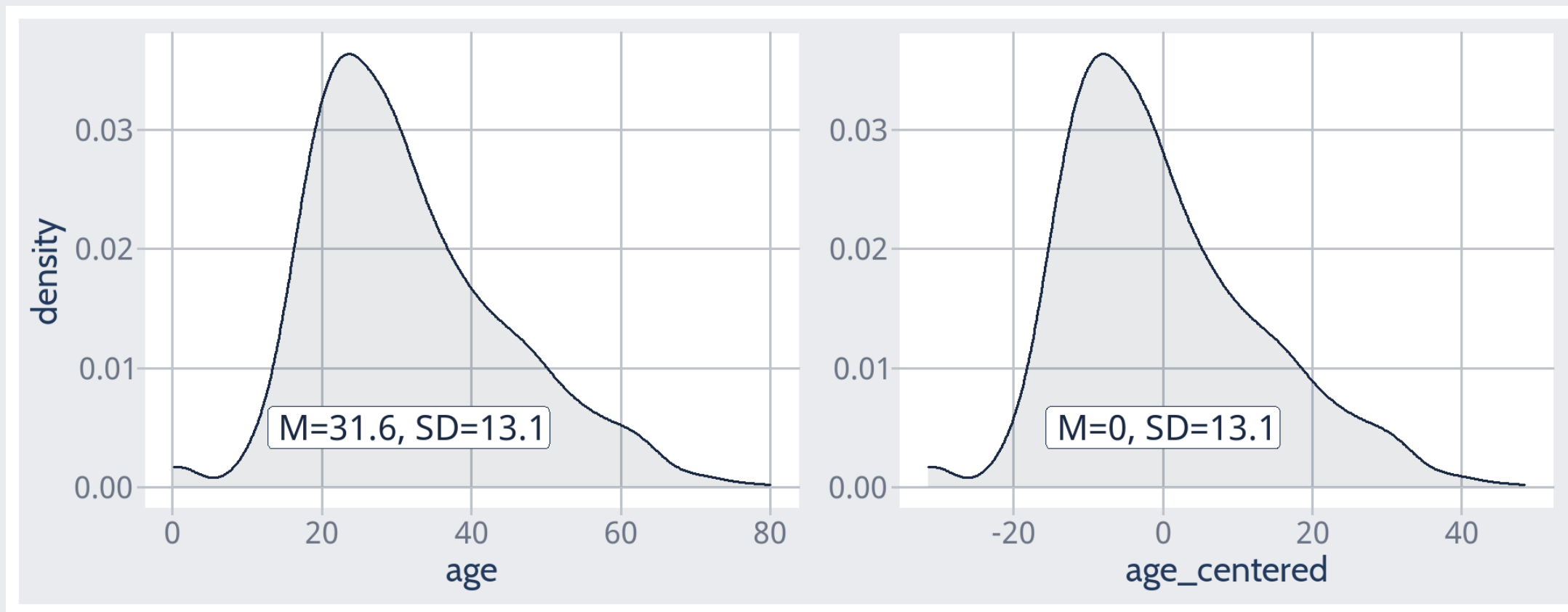
**Scaling** a predictor involves changing its standard deviation (and variance) to 1.0

- This is accomplished by dividing each observation by the standard deviation

**Normalizing** a predictor involves centering it and then scaling it

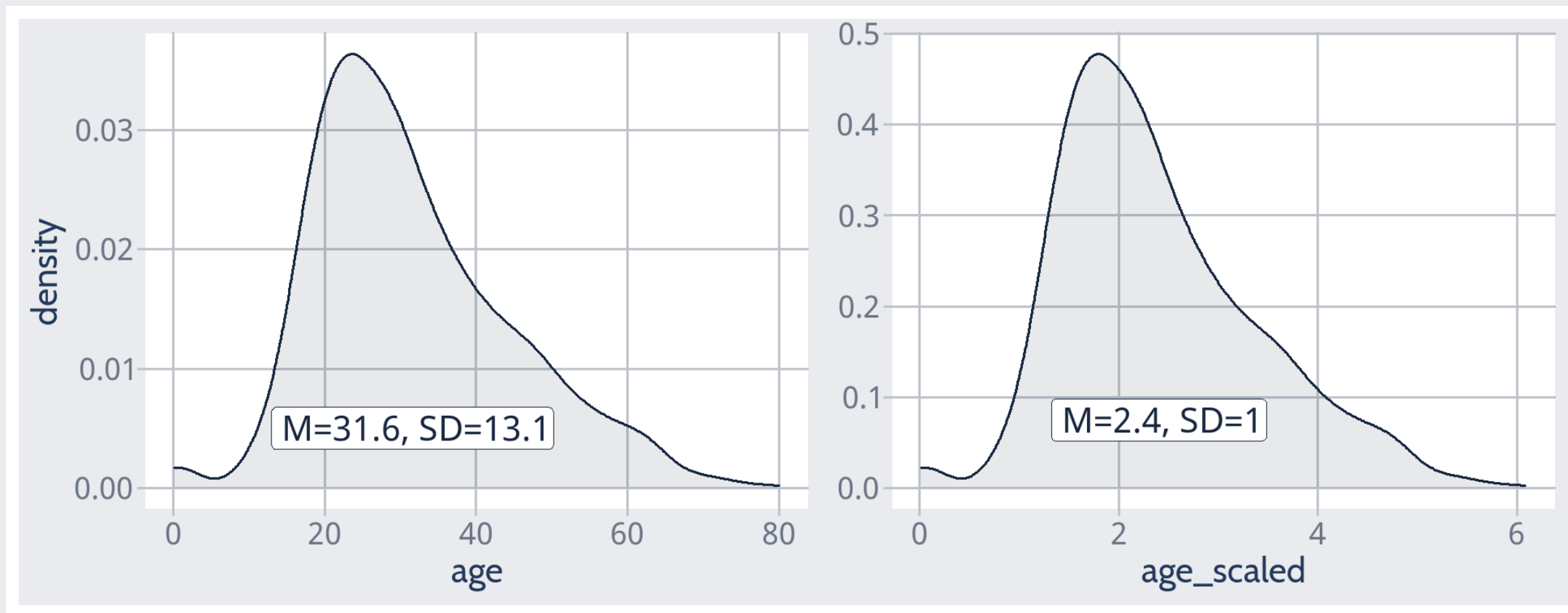
- This is also sometimes called "standardizing" or  $z$ -scoring the predictor

# Centering Visualized



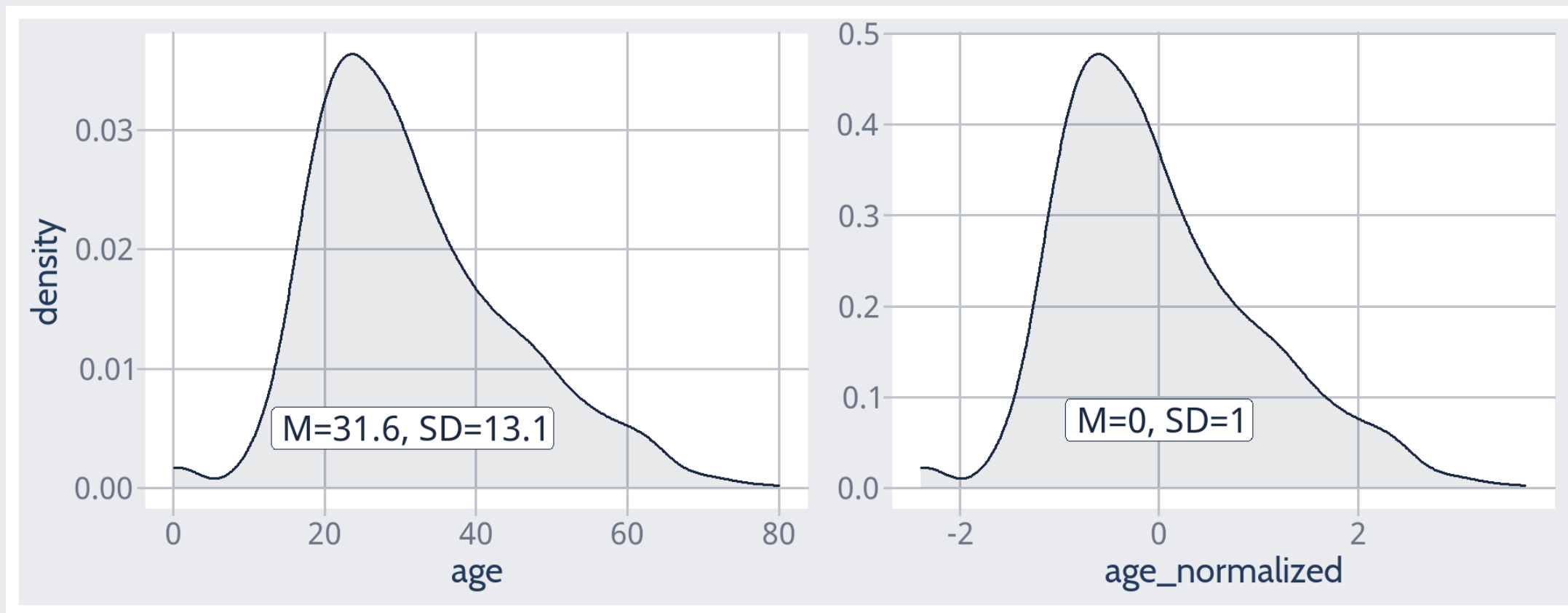
The mean is now 0 but the shape and SD of the distribution are unchanged (i.e., it has been shifted left).

# Scaling Visualized



The SD is now 1 and the mean is lower, but the shape of the distribution is unchanged.

# Normalizing Visualized



The mean is now 0 and the SD is now 1, but the shape of the distribution is unchanged.

# Normalizing in R

```
# Normalize the age variable using the training set mean and SD
norm_recipe <-
  titanic %>%
  recipe(survived ~ .) %>%
  step_normalize(age) %>%
  prep(training = titanic_train, log_changes = TRUE)
```

## step\_normalize (normalize\_tVirW): same number of columns

```
# Because of prep(), bake() uses the training set mean and SD to normalize the testing data2
norm_test <- bake(norm_recipe, new_data = titanic_test)
```

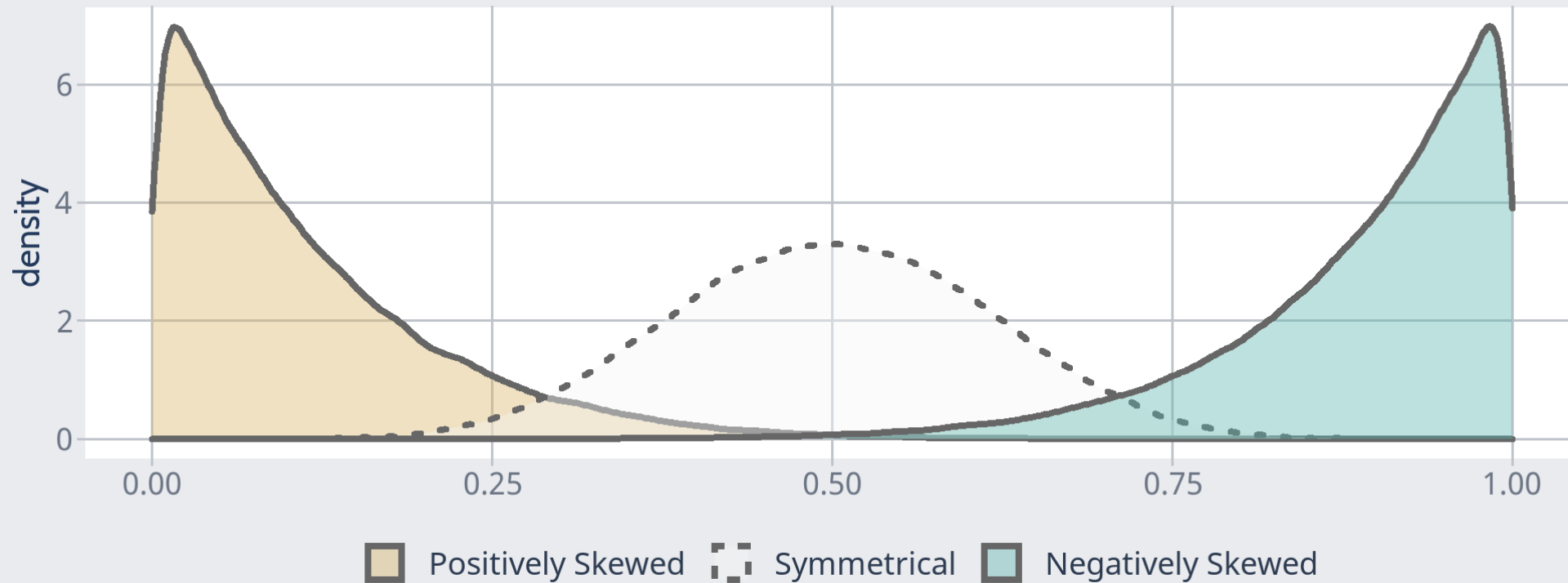
[1] We could also have used `step_center()` and/or `step_scale()` instead of `step_normalize()`.

[2] This is important to accurately estimating out-of-sample performance on truly novel data.

# Addressing Non-normality

A **skewed** distribution is one that is not symmetric (i.e., it has a "heavy tail")

A **bounded** distribution is one that cannot go beyond certain boundary values



# Addressing Non-normality

Specific transformations (e.g., log, inverse, logit) can help address specific issues

The Box-Cox and Yeo-Johnson approaches employ **families of transformations**

Box-Cox cannot be applied to negative or zero values, but **Yeo-Johnson** can

$$x_{(yj)}^* = \begin{cases} ((x + 1)^\lambda - 1)/\lambda & \text{if } \lambda \neq 0, x \geq 0 \\ \log(x + 1) & \text{if } \lambda = 0, x \geq 0 \\ -[(-x + 1)^{2-\lambda} - 1]/(2 - \lambda) & \text{if } \lambda \neq 2, x < 0 \\ -\log(-x + 1) & \text{if } \lambda = 2, x < 0 \end{cases}$$

# Addressing Non-normality in R

```
# Add step to apply the Yeo-Johnson transformation to fare
yj_recipe <-
  titanic %>%
  recipe(survived ~ .) %>%
  step_YeoJohnson(fare) %>%
  prep(training = titanic_train, log_changes = TRUE)
```

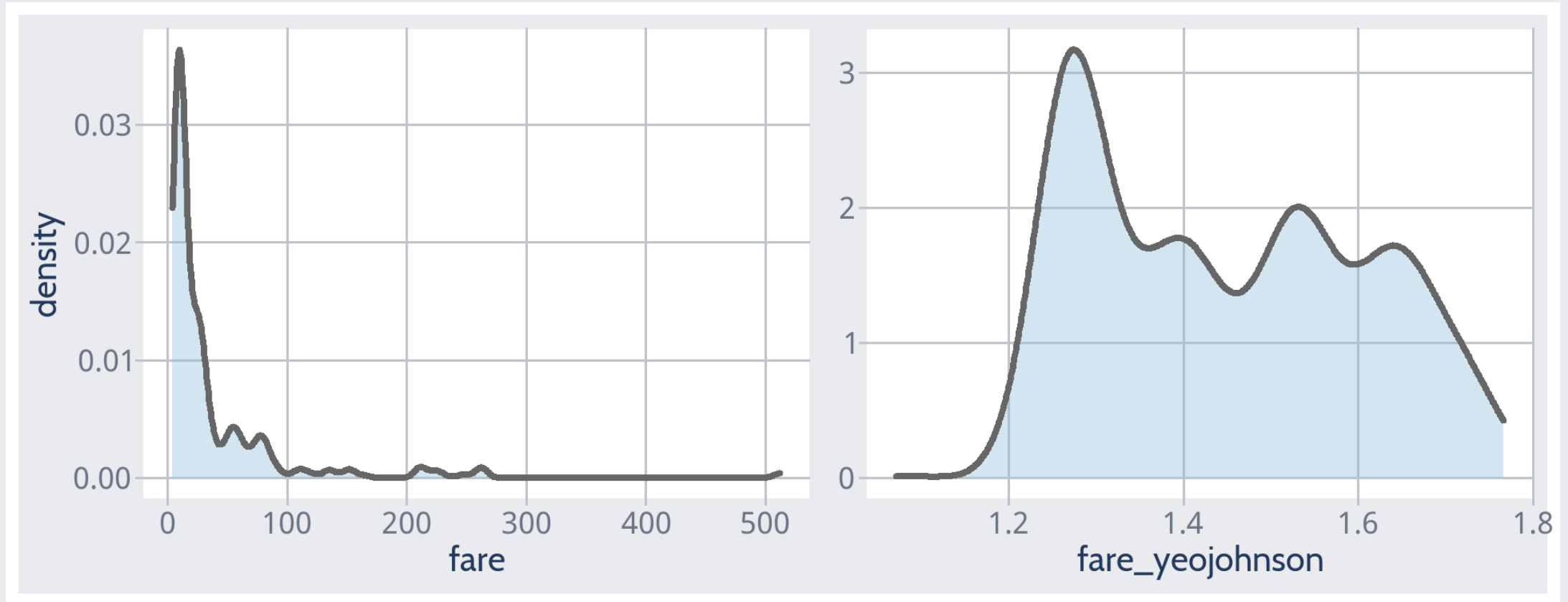
```
## step_YeoJohnson (YeoJohnson_lQ5Zw): same number of columns
```

- [1] If you would like to use specific transformations, use: `step_log()`, `step_inverse()`, `step_sqrt()`, etc.
- [2] As with normalizing, use `prep` to estimate  $\lambda$  from training set and use it when you `bake` the testing set.



# Addressing Non-normality in R

Bake the recipe using the training data and then plot the transformed variable



# Adding Nonlinearity

Many relationships between features and labels are non-linear in nature

- *e.g., perhaps survival was lowest for young adults and higher for children and elders*

Successful prediction will require us to **model that nonlinearity** in such cases

Many algorithms can capture nonlinearity easily but others need our help

- For these algorithms, we can provide help through feature engineering
- This typically means adding **nonlinear expansions** of existing predictors<sup>1</sup>

[1] If you are familiar with polynomial (e.g., quadratic or cubic) regression, you already have relevant experience!

# Adding Nonlinearity in R

```
# Add step to add orthogonal polynomial basis functions
nl_recipe <-
  titanic %>%
  recipe(survived ~ .) %>%
  step_poly(age, degree = 2) %>%
  prep(training = titanic_train, log_changes = TRUE)
```

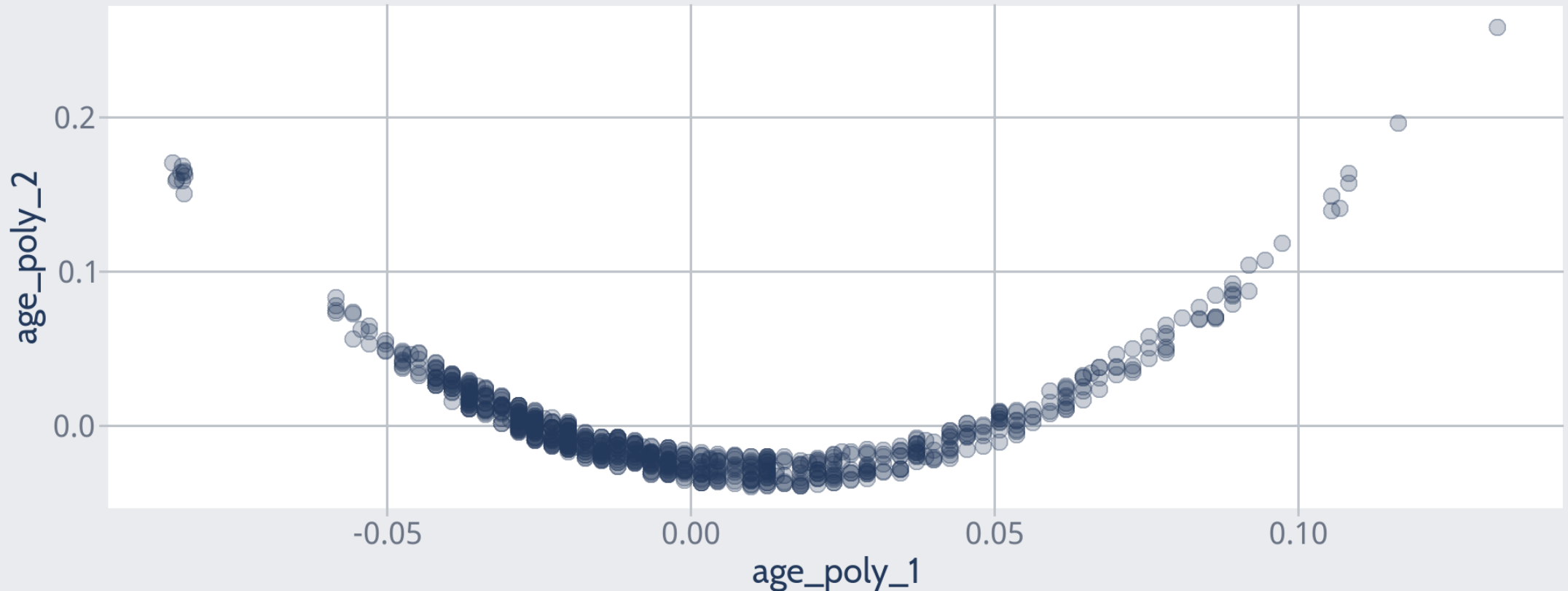
```
## step_poly (poly_egqAk):
##   new (2): age_poly_1, age_poly_2
##   removed (1): age
```

[1] Note that, by specifying `degree = 2`, we are creating a quadratic expansion; more flexibility can be added.

[2] Additional nonlinear expansions are also available: `step_ns()`, `step_bs()`, and `step_hyperbolic()`.

# Adding Nonlinearity in R

Bake the recipe and plot the polynomial terms against one another (with vertical jitter).



# Comprehension check

## Question 3

**I want to transform two predictors to have the same variance. Which would NOT achieve this?**

- a) Centering both
- b) Scaling both
- c) Normalizing both
- d) Dividing each by its SD

## Question 4

**Which of the following issues would the Yeo-Johnson transformation NOT help with?**

- a) Positive skew
- b) Negative skew
- c) Outlier values
- d) Categorical data

Reducing predictors

# Zero and Near-Zero Variance Predictors

**Zero variance predictors** take on only a single value in the sample

- These predictors are **uninformative** and may lead to **modeling problems**

**Near-zero variance predictors** take on only a few unique values with low frequencies

- These predictors can easily become zero-variance predictors during resampling

For many algorithms, we want to **detect** and **remove** both types of predictors

(This may not be necessary for algorithms with built-in *predictor selection*)

# Zero and Nero-Zero Variance Predictors in R

```
# Calculate predictors then detect and remove zero-variance predictors
zv_recipe <-
  titanic %>%
  recipe(survived ~ .) %>%
  step_mutate(species = "homo sapiens", under20 = age < 20, over70 = age > 70) %>%
  step_zv(all_predictors()) %>%
  prep(training = titanic_train, log_changes = TRUE)
```

```
## step_mutate (mutate_bX0DQ):
##   new (3): species, under20, over70
##
## step_zv (zv_lzXyg):
##   removed (1): species
```



# Zero and Near-Zero Variance Predictors in R

```
# Calculate predictors then detect and remove near-zero-variance predictors
nzv_recipe <-
  titanic %>%
  recipe(survived ~ .) %>%
  step_mutate(species = "homo sapiens", under20 = age < 20, over70 = age > 70) %>%
  step_nzv(all_predictors()) %>%
  prep(training = titanic_train, log_changes = TRUE)
```

```
## step_mutate (mutate_t9yCp):
##   new (3): species, under20, over70
##
## step_nzv (nzv_r1mGK):
##   removed (2): species, over70
```

# Multicollinearity

**Highly correlated predictors** can lead to problems for some algorithms/procedures

- The model has to randomly choose between the predictors, leading to **instability**
- Model predictions may be fine, but model **interpretation** will often be obfuscated
- The cutoff for "problematically high" correlations varies (e.g., 0.5 to 0.9 or higher)

Predictors that are **linear combinations** of other predictors are similarly problematic

- Occurs if a predictor variable can be predicted from the other predictor variables
- This is why dummy coding is preferred to one-hot encoding for some algorithms

For many algorithms, we want to **detect** and **remove** redundant predictors

(This may not be necessary for algorithms with *regularization* or *predictor selection*)

# Multicollinearity in R

```
# Add some predictors with high correlations and linear dependency
mc_titanic <- titanic %>% mutate(
  wisdom = 100 + 0.25 * age + rnorm(nrow(.)), # highly correlated with age
  family = sibsp + parch                     # linear combo of sibsp and parch
)
mc_train <- mc_titanic[index, ]
```

```
glimpse(mc_train)
```

```
## Rows: 772
## Columns: 9
## $ survived <chr> "yes", "yes", "no", "no", "yes", "yes", "no", "no", "yes", "y~
## $ pclass    <chr> "1st", "1st", "1st", "1st", "1st", "1st", "1st", "1st", "1st"~
## $ sex       <chr> "female", "male", "male", "female", "female", "female", "male~
## $ age       <dbl> 29.0000, 0.9167, 30.0000, 25.0000, 63.0000, 53.0000, 71.0000,~
## $ sibsp     <int> 0, 1, 1, 1, 1, 2, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0~
## $ parch     <int> 0, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0~
## $ fare      <dbl> 211.3375, 151.5500, 151.5500, 151.5500, 77.9583, 51.4792, 49.~
## $ wisdom    <dbl> 106.81232, 99.32269, 106.95356, 107.87199, 116.61815, 112.817~
## $ family    <int> 0, 3, 3, 3, 1, 2, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 2, 2, 0, 0, 0~
```

# Multicollinearity in R

```
# Estimate correlation matrix for the numeric predictors  
mc_train %>% select(where(is.numeric), -survived) %>% cor() %>% round(digits = 2)
```

```
##           age sibsp parch fare wisdom family  
## age      1.00 -0.11 -0.01 0.18   0.96  -0.07  
## sibsp   -0.11  1.00  0.33 0.16  -0.10   0.80  
## parch   -0.01  0.33  1.00 0.24   0.00   0.82  
## fare     0.18  0.16  0.24 1.00   0.17   0.25  
## wisdom   0.96 -0.10  0.00 0.17   1.00  -0.06  
## family  -0.07  0.80  0.82 0.25  -0.06   1.00
```

# Multicollinearity in R

```
# Detect and remove predictors that are highly correlated with another predictor
hc_recipe <-
  mc_titanic %>%
  recipe(survived ~ .) %>%
  step_corr(all_numeric_predictors(), threshold = 0.9) %>%
  prep(training = mc_train, log_changes = TRUE)
```

```
## step_corr (corr_eps1a):
##   removed (1): age
```

- [1] If we want to consider correlations with categorical variables, we can add `step_dummy()` to the pipeline.
- [2] We could have also lowered the threshold to 0.8 in order to drop the `family` variable here.

# Multicollinearity in R

```
# Detect and remove predictors that are linear combinations of other predictors
lc_recipe <-
  mc_titanic %>%
  recipe(survived ~ .) %>%
  step_impute_knn(all_predictors()) %>%
  step_lincomb(all_numeric_predictors()) %>%
  prep(training = mc_train, log_changes = TRUE)
```

```
## step_impute_knn (impute_knn_UkKYS): same number of columns
##
## step_lincomb (lincomb_3BcHI):
##   removed (1): family
```

# Dimensionality Reduction

Each feature/predictor included can be considered an additional "dimension"

**Dimensionality reduction** techniques try to find a smaller set of predictors to use

- If successful, little information from the original set of predictors will be lost
- Most techniques create new predictors as *functions of the original predictors*

**Principal Components Analysis (PCA)** is a commonly used technique

- The new predictors (PCs) are *linear combinations* of the original predictors
- The PCs are *uncorrelated* with one another, thus addressing multicollinearity
- PCs are extracted until a target amount of variability is explained (e.g., 75%)

[1] Predictors should be normalized (i.e., centered and scaled) before PCA is used.

[2] PCA is unsupervised (doesn't know the labels) but there are supervised techniques.

# Dimensionality Reduction in R

```
# Normalize the numeric predictors and then do PCA
pca_recipe <-
  titanic %>%
  recipe(survived ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_pca(all_numeric_predictors(), threshold = 0.75) %>%
  prep(training = titanic_train, log_changes = TRUE)
```

```
## step_normalize (normalize_mAFIy): same number of columns
##
## step_pca (pca_9Y9CU):
##   new (3): PC1, PC2, PC3
##   removed (4): age, sibsp, parch, fare
```

- [1] Note that PCA is most effective when there are many correlated predictors, so this is a weak use of it.
- [2] Note that higher thresholds (e.g., 0.95) are often used with algorithms that can handle many predictors.
- [3] `step_kpca_poly()` and `step_kpca_rbf()` are nonlinear, and `step_pls()` is supervised.



# Advanced Topics

# Feature Extraction

Feature extraction involves generating features from "raw" data

For raw **text** data, natural language processing techniques can be used

- *e.g., sentiment, word frequencies, word relationships, topic modeling, syntax*

For raw **image** and video data, computer vision techniques can be used

- *e.g., edges, corners, blobs, ridges, objects, curvature, shape, motion, color*

For raw **audio** data, acoustic signal processing techniques can be used

- *e.g., rhythm, stress, intonation, pitch, loudness, glottal flow, spectral density*

[1] One of the strengths of deep learning is its ability to learn its own feature representations from raw data.

# Dealing with Missing Values

There are several approaches to handling missing values in predictors

- Some algorithms (e.g., tree-based techniques) handle missing data inherently
- Another option is to drop predictors with any (or a lot of) missing values
- Or we can **impute** or estimate the missing values based on the other predictors

There are many techniques for imputing missing predictor values

- Some are very simple (e.g., using the mean or median) and others more complex
- We can also use a linear model or even machine learning to impute missing values
- {recipes} provides functions: `step_impute_mean()`, `step_impute_knn()`, etc.

[1] Missing data tends to be more problematic for inferential modeling than predictive modeling.

[2] When imputing, it is a good idea to use cross-validation to capture the uncertainty in the imputations.

# Feature Selection

Feature selection is focused on removing uninformative or redundant predictors

- Models with fewer predictors may be more interpretable, accurate, and efficient

**Wrapper methods** compare models with different combinations of predictors

- These are algorithms that search for combinations that optimize performance
- These methods tend to perform well but can be computationally expensive

**Filter methods** evaluate predictors outside of the context of the predictive model<sup>1</sup>

- Only predictors that seem informative, relevant, or unique will be retained
- These methods don't perform as well but are computationally efficient

[1] Note that we have already learned some basic filter methods (e.g., `step_corr()` and `step_nzv()`).

# Live Coding Activity

# Live Coding Activity

I will show you a new dataset and the process of feature engineering in RStudio

- You can follow along in your own RStudio (easiest with 1 large or 2+ monitors)
- Or you can download the `Day_2A_Activity.Rmd` and follow along
- In either case, you will need to download the `airsatisfaction.csv` file

Afterward, there will be an activity where you will modify my code to achieve goals

If you have questions, please post them in chat or in the workshop Slack channel

[1] All files can be accessed from the workshop website: <https://osf.io/3qhc8/>

Time for a Break!

10:00