Fitting

Machine Learning with R

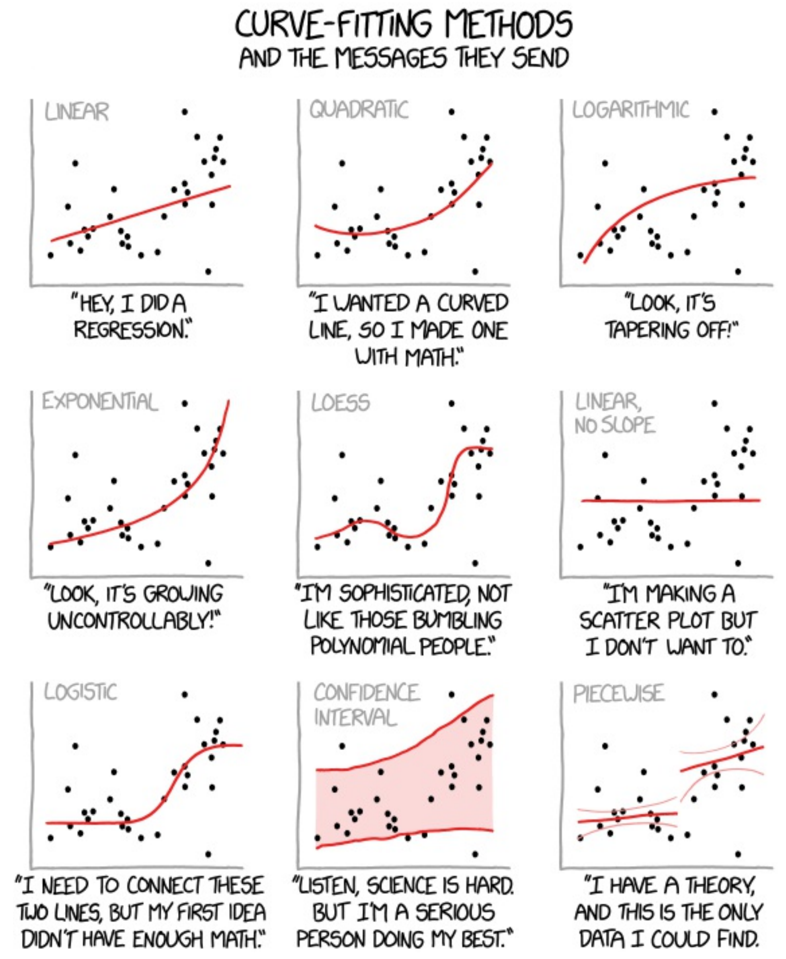
Basel R Bootcamp

May 2019

Fitting

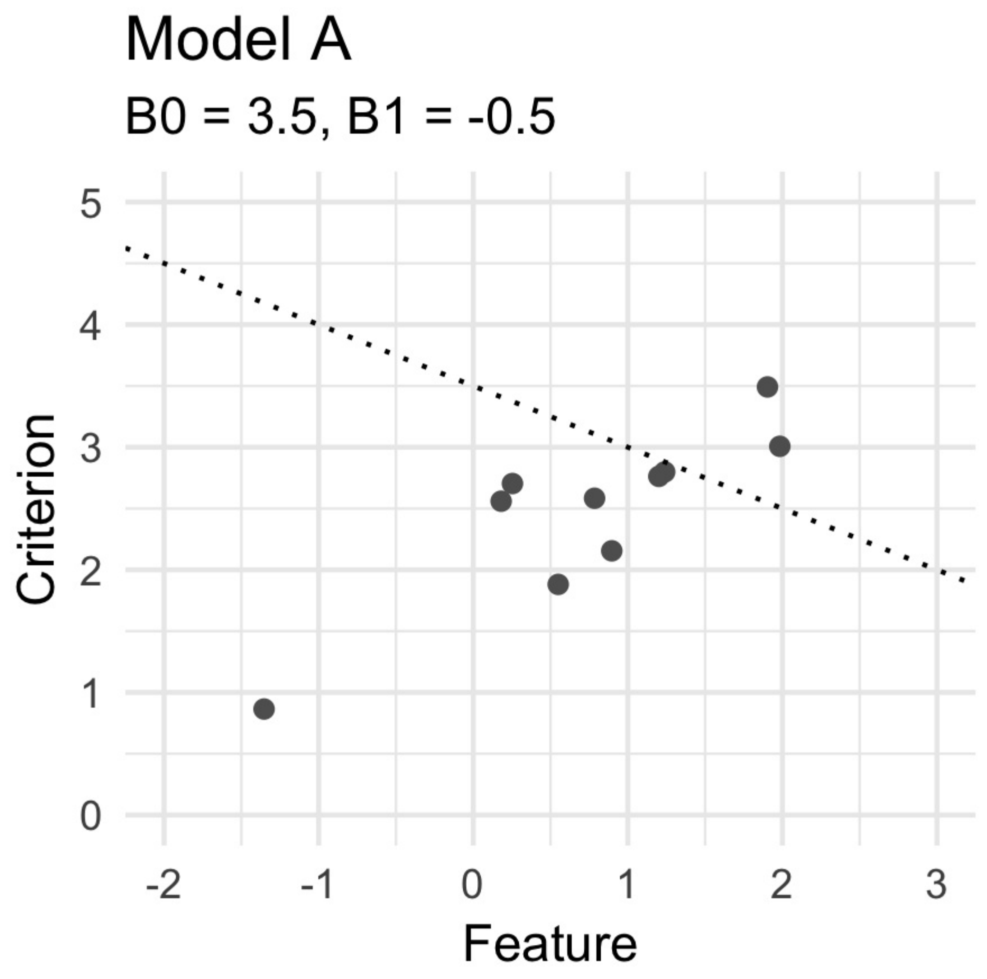
Models are actually **families of models**, with every parameter combination specifying a different model.

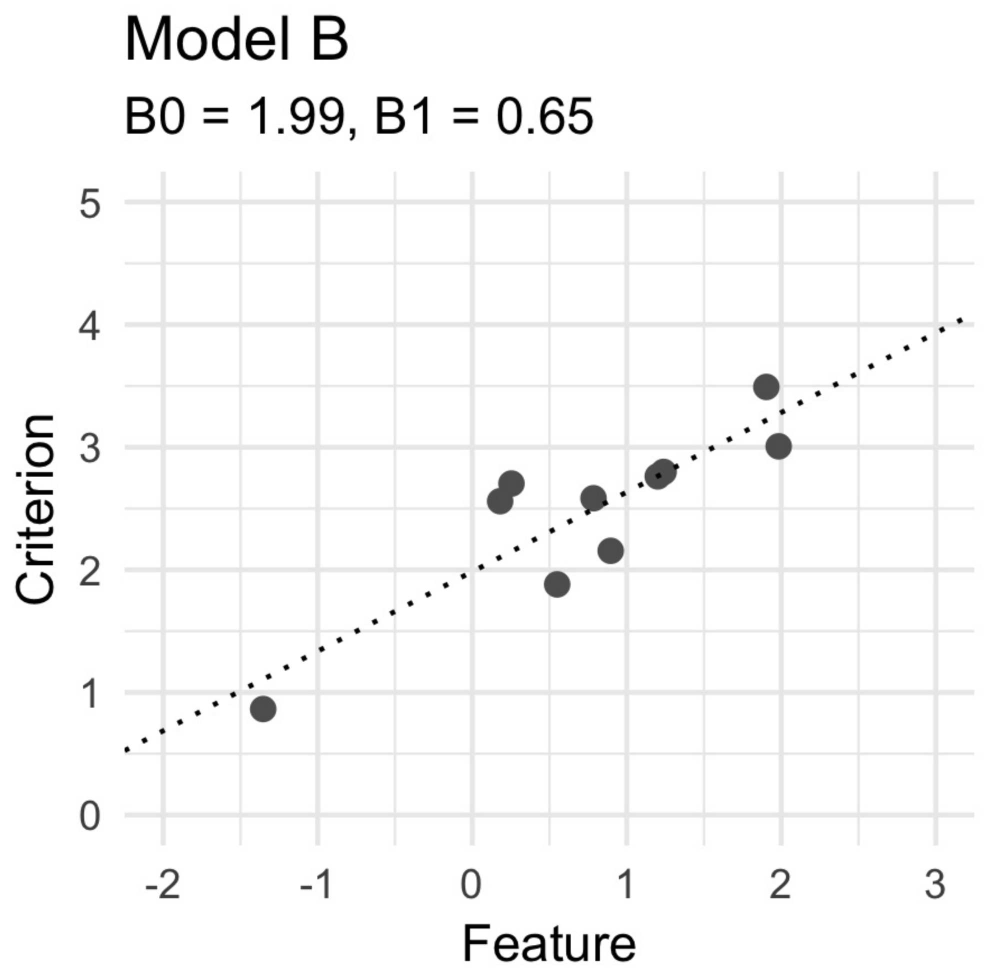
To fit a model means to **identify** from the family of models **the specific model that fits the data best**.


CURVE-FITTING METHODS
AND THE MESSAGES THEY SEND
LINEAR
"HEY, I DID A REGRESSION."
QUADRATIC
"I WANTED A CURVED LINE, SO I MADE ONE WITH MATH."
LOGARITHMIC
"LOOK, IT'S TAPERING OFF!"
EXPONENTIAL
"LOOK, IT'S GROWING UNCONTROLLABLY!"
LOESS
"I'M SOPHISTICATED, NOT LIKE THOSE BUMBLING POLYNOMIAL PEOPLE."
LINEAR, NO SLOPE
"I'M MAKING A SCATTER PLOT BUT I DON'T WANT TO."
LOGISTIC
"I NEED TO CONNECT THESE TWO LINES, BUT MY FIRST IDEA DIDN'T HAVE ENOUGH MATH."
CONFIDENCE INTERVAL
"LISTEN, SCIENCE IS HARD. BUT I'M A SERIOUS PERSON DOING MY BEST."
PIECEWISE
"I HAVE A THEORY, AND THIS IS THE ONLY DATA I COULD FIND."

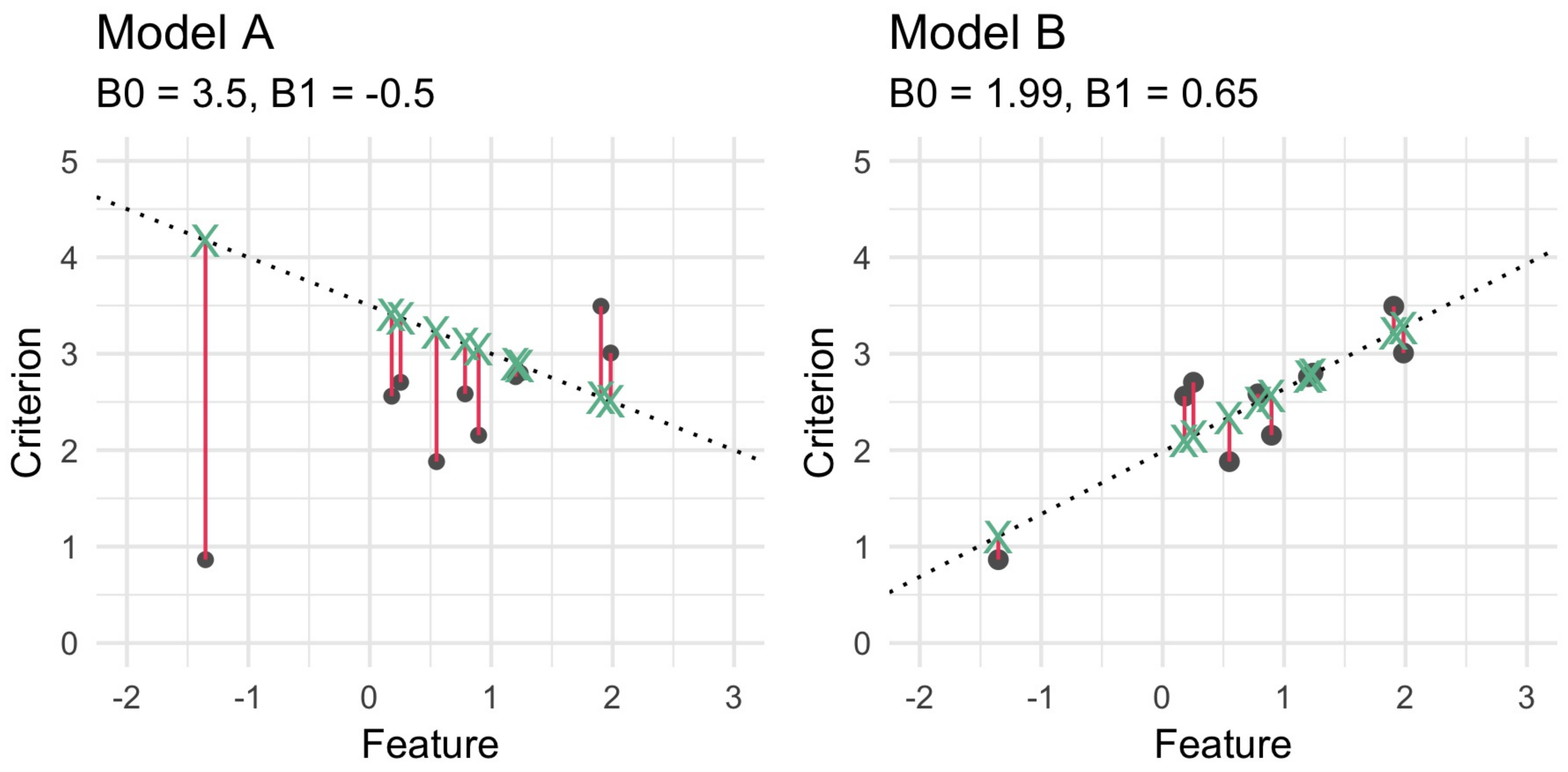
adapted from explainxkcd.com

Which of these models is better? Why?





Which of these models is better? Why?



Loss function

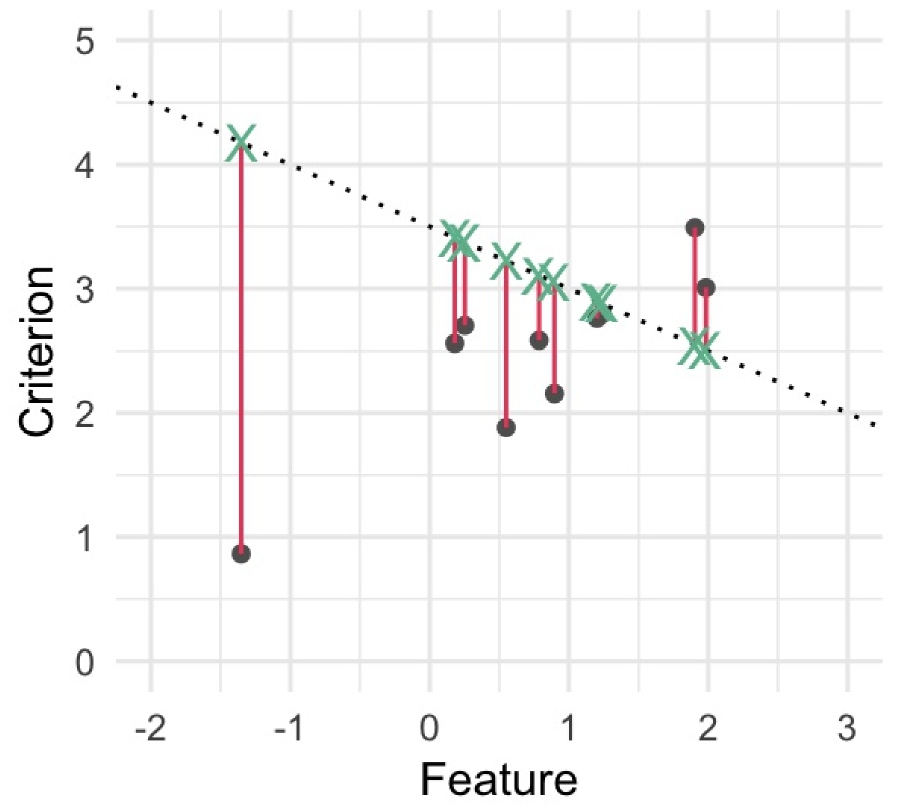
Possible **the most important concept** in statistics and machine learning.

The loss function defines some **summary of the errors committed by the model**.

$$Loss = f(Error)$$

Two purposes

Purpose	Description
Fitting	Find parameters that minimize loss function.
Evaluation	Calculate loss function for fitted model.



Regression

Decision Trees

Random Forests

Regression

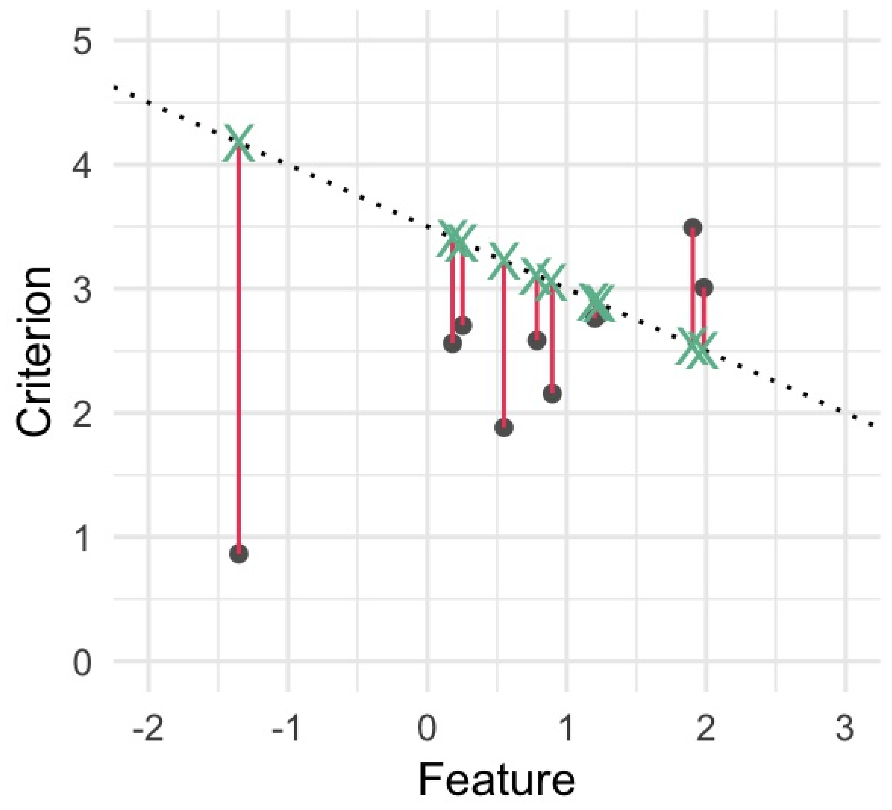
In **regression**, the criterion Y is modeled as the **sum** of **features** $X_1, X_2, ...$ **times weights** $\beta_1, \beta_2, ...$ plus β_0 the so-called the intercept.

$$\hat{Y} = \beta_0 + \beta_1 \times X_1 + \beta_2 \times X2 + ...$$

The weight β_i indiciates the **amount of change** in $\hat{Y}$ for a change of 1 in X_i.

Ceteris paribus, the **more extreme** β_i, the **more important** X_i for the prediction of Y (Note: the scale of X_i matters too!).

If $\beta_i = 0$, then X_i **does not help** predicting Y



Regression

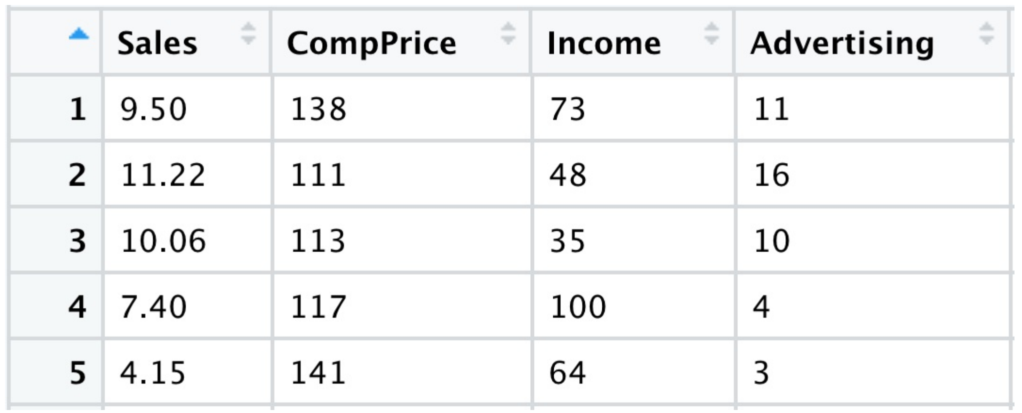
In **regression**, the criterion Y is modeled as the **sum** of **features** $X_1, X_2, ...$ **times weights** $\beta_1, \beta_2, ...$ plus β_0 the so-called the intercept.

$$\hat{Y} = \beta_0 + \beta_1 \times X_1 + \beta_2 \times X2 + ...$$

The weight β_i indiciates the **amount of change** in $\hat{Y}$ for a change of 1 in X_i.

Ceteris paribus, the **more extreme** β_i, the **more important** X_i for the prediction of Y (Note: the scale of X_i matters too!).

If $\beta_i = 0$, then X_i **does not help** predicting Y



	Sales	CompPrice	Income	Advertising
1	9.50	138	73	11
2	11.22	111	48	16
3	10.06	113	35	10
4	7.40	117	100	4
5	4.15	141	64	3

$$\begin{aligned}\hat{Y} =& 3.88 + \\ & .015 * CompPrice + \\ & .014 * Income + \\ & .111 * Advertising\end{aligned}$$

Regression loss

Mean Squared Error (MSE)

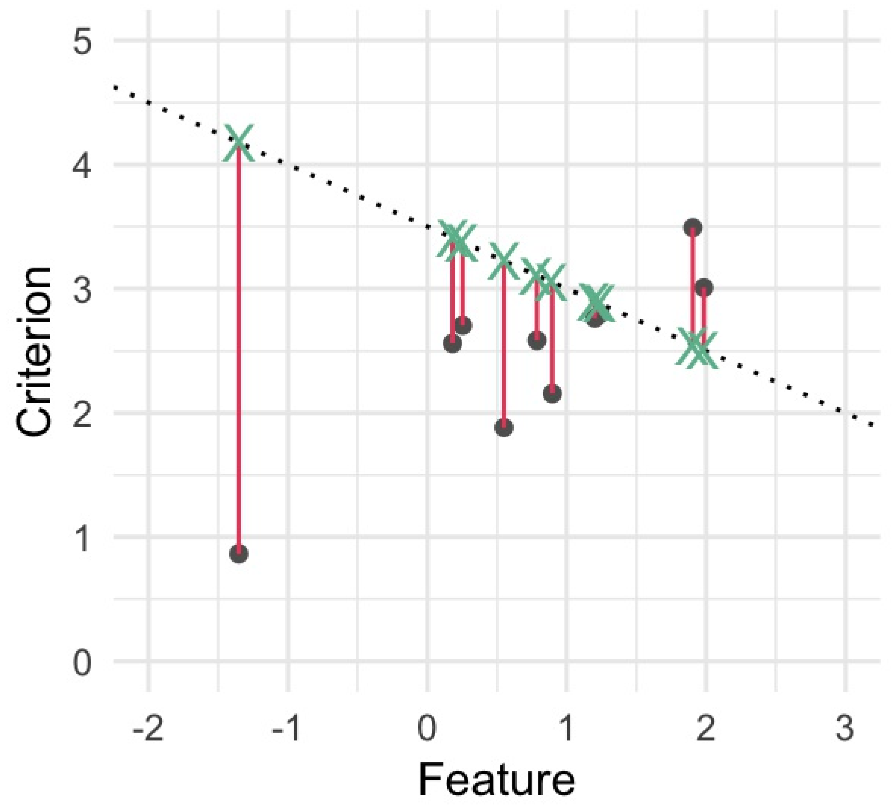
Average squared distance between predictions and true values?

$$MSE = \frac{1}{n} \sum_{i \in 1,\ldots,n} (Y_i - \hat{Y}_i)^2$$

Mean Absolute Error (MAE)

Average absolute distance between predictions and true values?

$$MAE = \frac{1}{n} \sum_{i \in 1,\ldots,n} |Y_i - \hat{Y}_i|$$



Fitting

There are two fundamentally different ways to find the set of parameters that minimizes loss.

Analytically

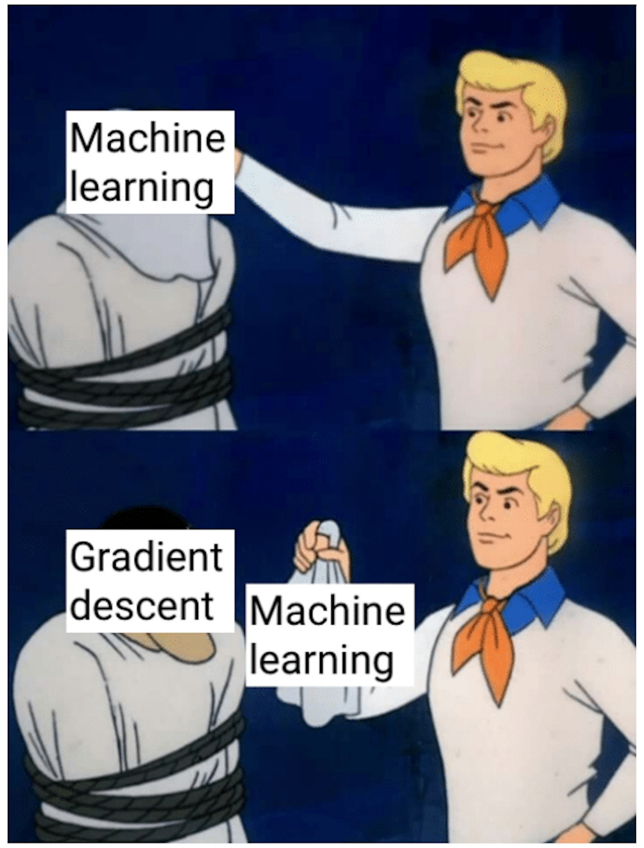
In rare cases, the parameters can be **directly calculated**, e.g., using the *normal equation*:

$$= (\quad {}^{T} \quad)^{-1} \quad {}^{T}$$

Numerically

In most cases, parameters need to be found using a **directed trial and error**, e.g., *gradient descent*:

$${}_{n+1} = \quad {}_{n} + \gamma \nabla F(\quad {}_{n})$$



adapted from me.me

Fitting

There are two fundamentally different ways to find the set of parameters that minimizes loss.

Analytically

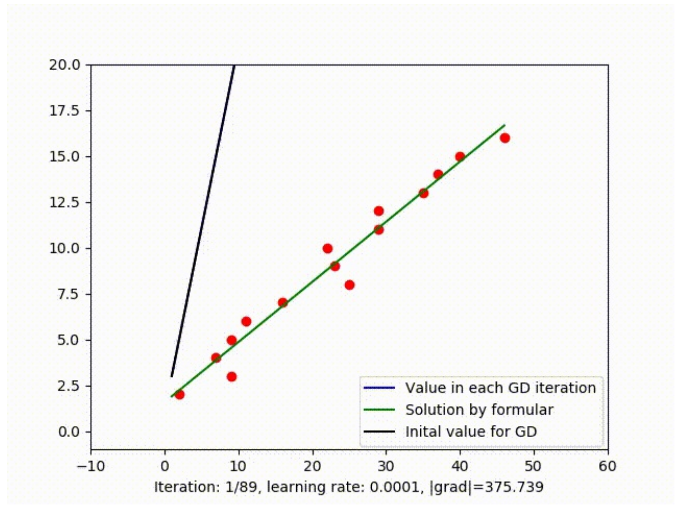
In rare cases, the parameters can be **directly calculated**, e.g., using the *normal equation*:

$$= (\quad ^T \quad)^{-1} \quad ^T$$

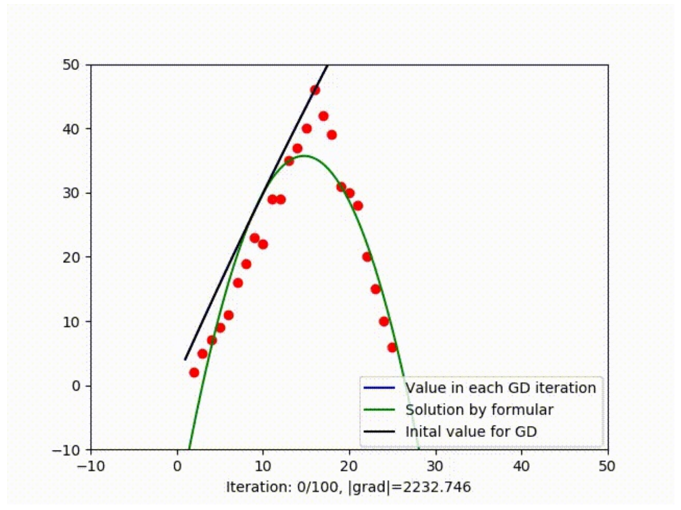
Numerically

In most cases, parameters need to be found using a **directed trial and error**, e.g., *gradient descent*:

$$_{n+1} = \quad _n + \gamma \nabla F(\quad _n)$$



adapted from dunglai.github.io



adapted from dunglai.github.io

2 types of supervised problems

There are two types of supervised learning problems that can often be approached using the same model.

Regression

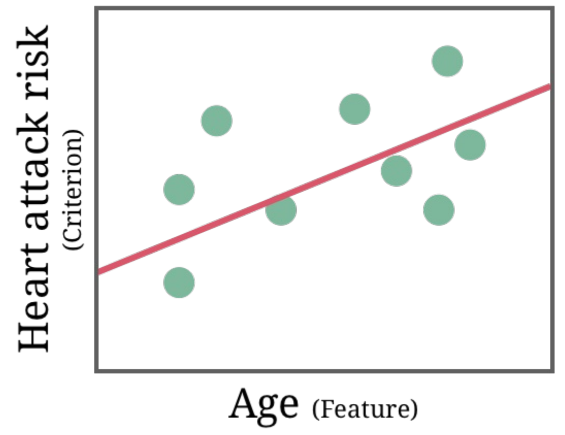
Regression problems involve the **prediction of a quantitative feature**.

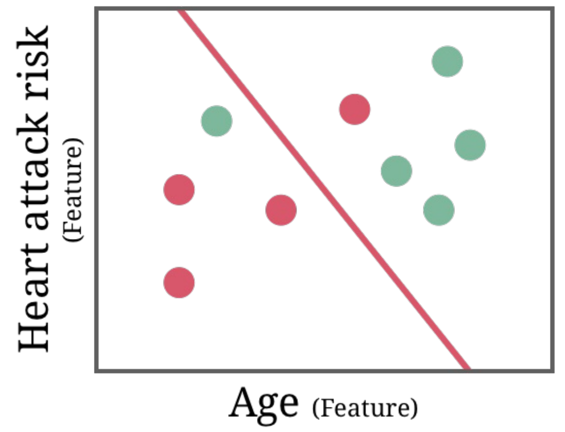
E.g., predicting the cholesterol level as a function of age.

Classification

Classification problems involve the **prediction of a categorical feature**.

E.g., predicting the type of chest pain as a function of age.





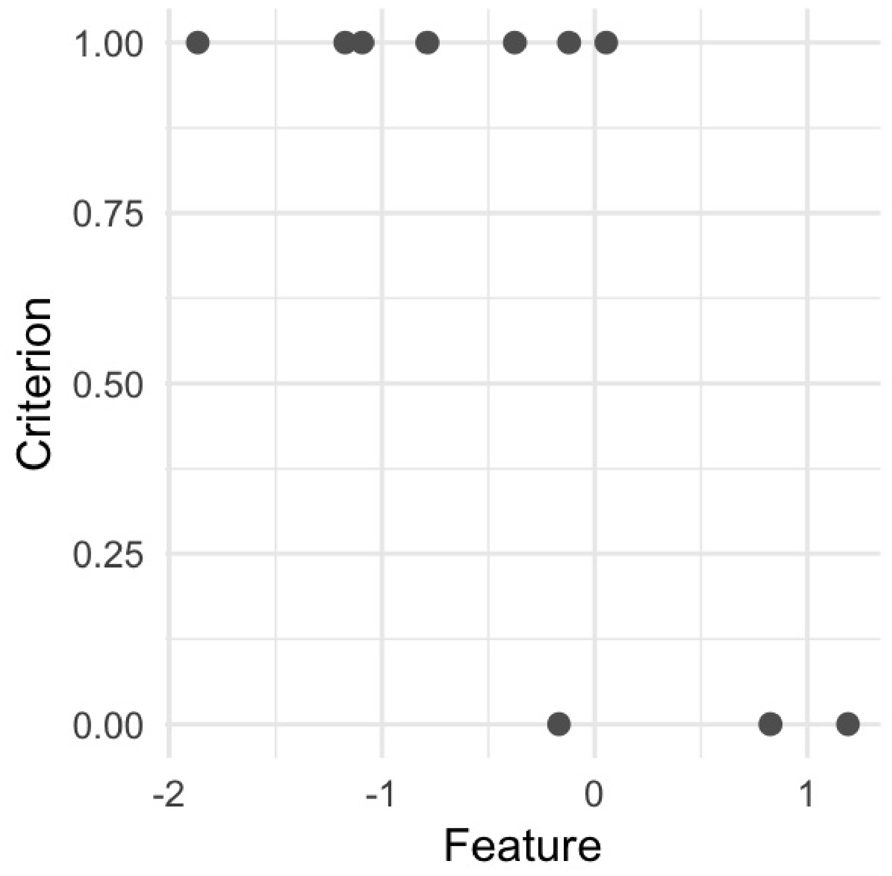
Logistic regression

In logistic regression, the class criterion $Y \in (0,1)$ is modeled also as the **sum of feature times weights**, but with the prediction being transformed using a **logistic link function**:

$$\hat{Y} = Logistic(\beta_0 + \beta_1 \times X_1 + \ldots)$$

The logistic function **maps predictions to the range of 0 and 1**, the two class values.

$$Logistic(x) = \frac{1}{1 + exp(-x)}$$



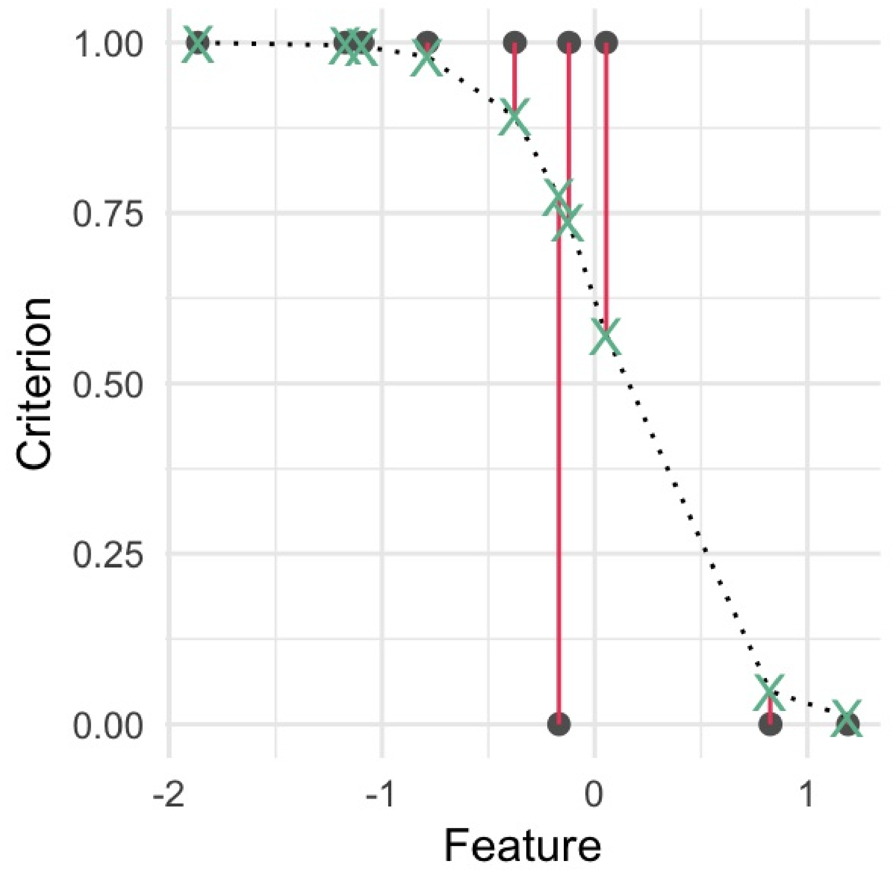
Logistic regression

In **logistic regression**, the class criterion $Y \in (0,1)$ is modeled also as the **sum of feature times weights**, but with the prediction being transformed using a **logistic link function**:

$$\hat{Y} = Logistic(\beta_0 + \beta_1 \times X_1 + \ldots)$$

The logistic function **maps predictions to the range of 0 and 1**, the two class values.

$$Logistic(x) = \frac{1}{1 + exp(-x)}$$



Classification loss - two ways

Distance

Logloss is **used to fit the parameters**, alternative distance measures are MSE and MAE.

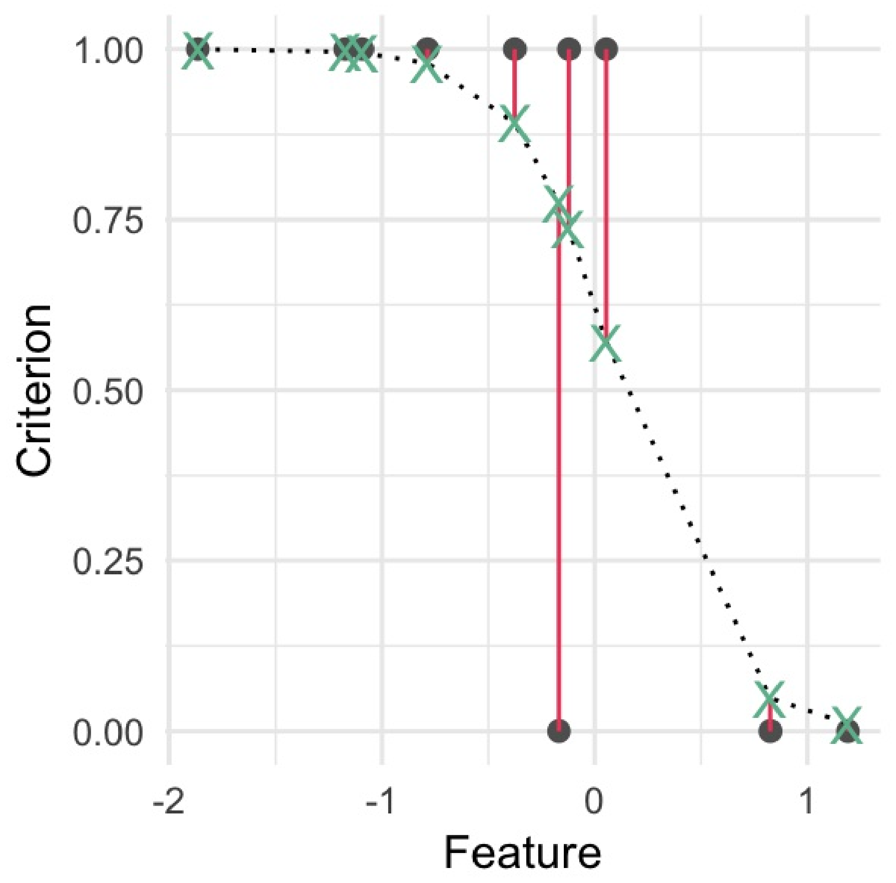
$$LogLoss = -\frac{1}{n}\sum_i^n (log(\hat{y})y + log(1-\hat{y})(1-y))$$

$$MSE = \frac{1}{n}\sum_i^n (y-\hat{y})^2,\; MAE = \frac{1}{n}\sum_i^n |y-\hat{y}|$$

Overlap

Does the **predicted class match the actual class**. Often preferred for **ease of interpretation**.

$$Loss_{01} = \frac{1}{n}\sum_i^n I(y \neq \lfloor \hat{y} \rceil)$$



Confusion matrix

The confusion matrix **tabulates prediction matches and mismatches** as a function of the true class.

The confusion matrix permits specification of a number of **helpful performance metrics**.

Confusion matrix

	ŷ = 1	ŷ = 0
y = 1	True positive (TP)	False negative (FN)
y = 0	False positive (FP)	True negative (TN)

Accuracy: Of all cases, what percent of predictions are correct?

$$Acc. = \frac{TP + TN}{TP + TN + FN + FP} = 1 - Loss_{01}$$

Sensitivity: Of the truly Positive cases, what percent of predictions are correct?

$$Sensitivity = \frac{TP}{TP + FN}$$

Specificity: Of the truly Negative cases, what percent of predictions are correct?

$$Specificity = \frac{TN}{TN + FP}$$

Confusion matrix

The confusion matrix **tabulates prediction matches and mismatches** as a function of the true class.

The confusion matrix permits specification of a number of **helpful performance metrics**.

Confusion matrix

	"Default"	"Repay"
Default	TP = 3	FN = 1
Repay	FP = 1	TN = 2

Accuracy: Of all cases, what percent of predictions are correct?

$$Acc. = \frac{TP + TN}{TP + TN + FN + FP} = 1 - Loss_{01}$$

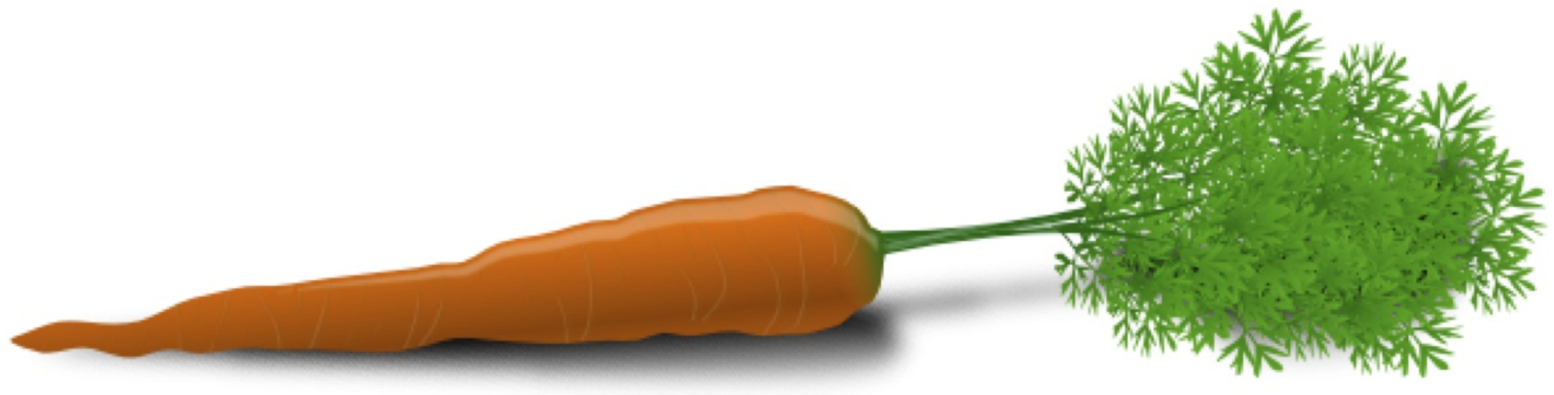
Sensitivity: Of the truly Positive cases, what percent of predictions are correct?

$$Sensitivity = \frac{TP}{TP + FN}$$

Specificity: Of the truly Negative cases, what percent of predictions are correct?

$$Specificity = \frac{TN}{TN + FP}$$

Let's fit regression models with `caret`!



caret

caret's key fitting functions

Function	Description
`trainControl()`	Choose settings for how fitting should be carried out.
`train()`	Specify the model and find *best* parameters.
`postResample()`	Evaluate model performance (fitting or prediction) for regression.
`confusionMatrix()`	Evaluate model performance (fitting or prediction) for classification.

```
# Step 1: Define control parameters
#   trainControl()

ctrl <- trainControl(...)

# Step 2: Train and explore model
#   train()

mod <- train(...)
summary(mod)
mod$finalModel   # see final model

# Step 3: Assess fit
#   predict(), postResample(), fon

fit <- predict(mod)
postResample(fit, truth)
confusionMatrix(fit, truth)
```

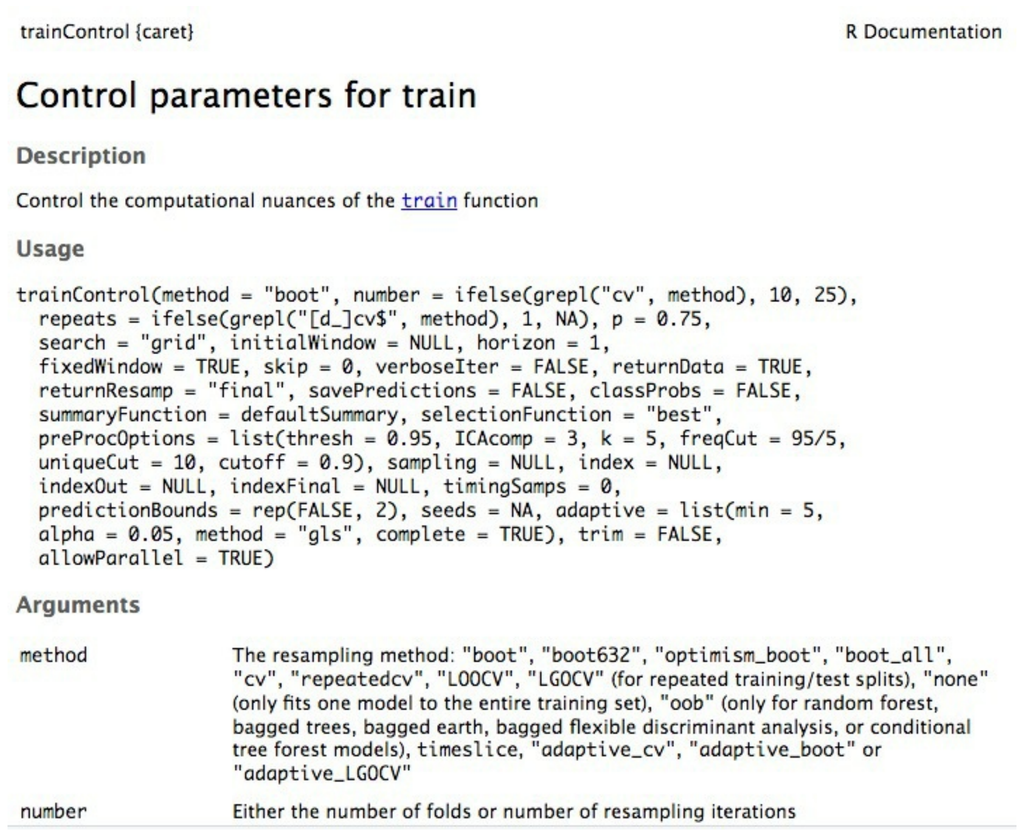
trainControl()

`trainControl()` controls how `caret` fits an ML model.

For now, set `method = "none"` to keep things simple. More in the session on **optimization**.

```r
# Fit the model without any
#  advanced parameter tuning methods

ctrl <- trainControl(method = "none")
```

```r
?trainControl
```



trainControl {caret} R Documentation

Control parameters for train

Description

Control the computational nuances of the train function

Usage

```r
trainControl(method = "boot", number = ifelse(grepl("cv", method), 10, 25),
  repeats = ifelse(grepl("[d_]cv$", method), 1, NA), p = 0.75,
  search = "grid", initialWindow = NULL, horizon = 1,
  fixedWindow = TRUE, skip = 0, verboseIter = FALSE, returnData = TRUE,
  returnResamp = "final", savePredictions = FALSE, classProbs = FALSE,
  summaryFunction = defaultSummary, selectionFunction = "best",
  preProcOptions = list(thresh = 0.95, ICAcomp = 3, k = 5, freqCut = 95/5,
  uniqueCut = 10, cutoff = 0.9), sampling = NULL, index = NULL,
  indexOut = NULL, indexFinal = NULL, timingSamps = 0,
  predictionBounds = rep(FALSE, 2), seeds = NA, adaptive = list(min = 5,
  alpha = 0.05, method = "gls", complete = TRUE), trim = FALSE,
  allowParallel = TRUE)
```

Arguments

`method`	The resampling method: "boot", "boot632", "optimism_boot", "boot_all", "cv", "repeatedcv", "LOOCV", "LGOCV" (for repeated training/test splits), "none" (only fits one model to the entire training set), "oob" (only for random forest, bagged trees, bagged earth, bagged flexible discriminant analysis, or conditional tree forest models), timeslice, "adaptive_cv", "adaptive_boot" or "adaptive_LGOCV"
`number`	Either the number of folds or number of resampling iterations

train()

train() is the fitting **workhorse** of caret, offering you **200+ models** just by changing the **method** argument!

train()'s key arguments

Argument	**Description**
form	Formula specifying features and criterion.
data	Training data.
method()	The model (algorithm).
trControl()	Control parameters for fitting.
tuneGrid(), preProcess()	Cool stuff for later.

```r
# Fit a regression model predicting Price

income_mod <-
  train(form = income ~ ., # Formula
        data = baselers,   # Training data
        method = "glm",    # Regression
        trControl = ctrl)  # Control Param's
income_mod
```

```
Generalized Linear Model

1000 samples
  19 predictor

No pre-processing
Resampling: None
```

train()

train() is the fitting **workhorse** of caret, offering you **200+ models** just by changing the **method** argument!

train()'s key arguments

Argument	**Description**
form	Formula specifying features and criterion.
data	Training data.
method()	The model (algorithm).
trControl()	Control parameters for fitting.
tuneGrid(), preProcess()	Cool stuff for later.

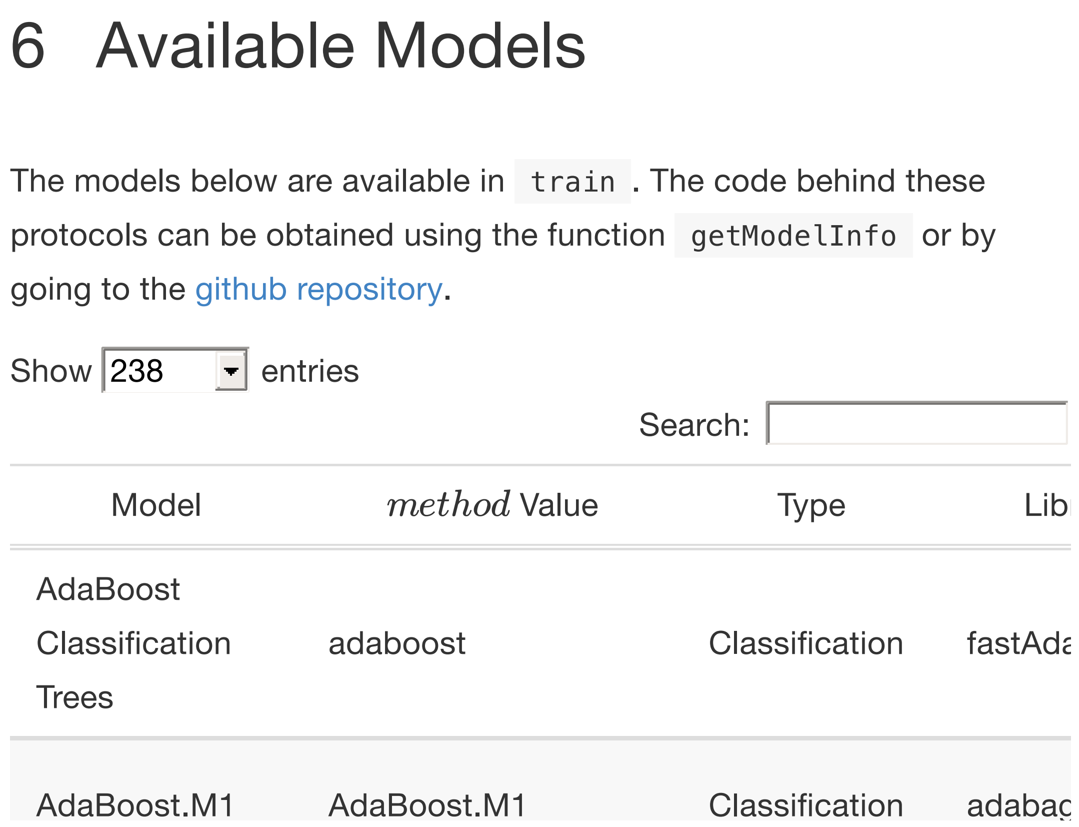
```r
# Fit a random forest predicting Price

income_mod <-
  train(form = income ~ .,# Formula
        data = baselers,  # Training data
        method = "rf",    # Random Forest
        trControl = ctrl) # Control Param's
income_mod
```

```
Random Forest

1000 samples
  19 predictor

No pre-processing
Resampling: None
```

train()

train() is the fitting **workhorse** of caret, offering you **200+ models** just by changing the **method** argument!

Find all 200+ models here.



6 Available Models

The models below are available in `train`. The code behind these protocols can be obtained using the function `getModelInfo` or by going to the github repository.

Show 238 entries

Search:

Model	*method* Value	Type	Lib
AdaBoost Classification Trees	adaboost	Classification	fastAda
AdaBoost.M1	AdaBoost.M1	Classification	adabag

train()

The criterion must be the right type:

`numeric` **criterion** = **Regression**
`factor` **criterion** = **Classification**!

```
# A tibble: 5 x 5
  Default   Age Gender Cards Education
    <dbl> <dbl> <chr>  <dbl>     <dbl>
1       0    45 M          3        11
2       1    36 F          2        14
3       0    76 F          5        12
4       1    25 M          2        17
5       1    36 F          3        12
```

```
# Will be a regression task

loan_mod <- train(form = Default ~ .,
                  data = Loans,
                  method = "glm",
                  trControl = ctrl)

# Will be a classification task

load_mod <- train(form = factor(Default) ~ .,
                  data = Loans,
                  method = "glm",
                  trControl = ctrl)
```

.$finalModel

The train() function returns a list with a key object called finalModel - this is your **final machine learning model**!

Access the model with mod$finalModel and **explore** the object with generic functions:

Function	Description
summary()	Overview of the most important results.
names()	See all named elements you can access with $.

```
# Create a regression object
income_mod <-
  train(form = income ~ age + height,
        data = baselers)  # Training data

# Look at all named outputs
names(income_mod$finalModel)
```

```
[1] "coefficients"  "residuals"     "fitted.values"
[4] "effects"       "R"             "rank"
 [ reached getOption("max.print") -- omitted 28 entries ]
```

```
# Access specific outputs
income_mod$finalModel$coefficients
```

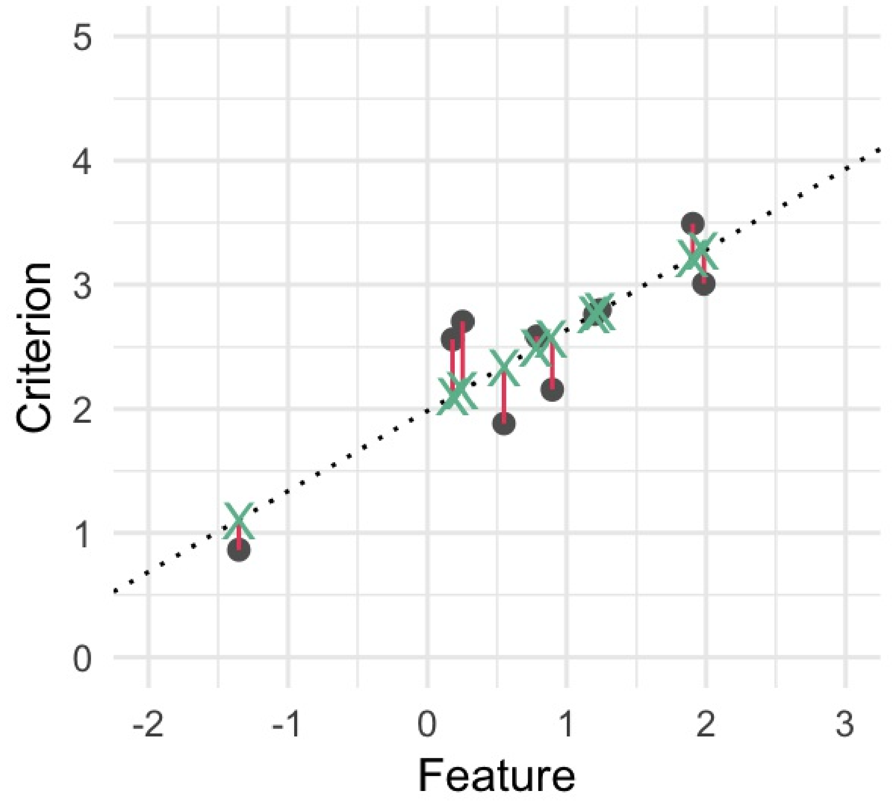
```
(Intercept)         age      height
    177.084     151.786       3.466
```

predict()

The `predict()` function **produces predictions** from a model. Simply put model object as the first argument.

```
# Get fitted values
glm_fits <- predict(object = income_mod)
glm_fits[1:8]
```

```
   1     2     3     4     5     6     7     8 
5508  6960  6982  8645  5325 10648  8663  4592 
```

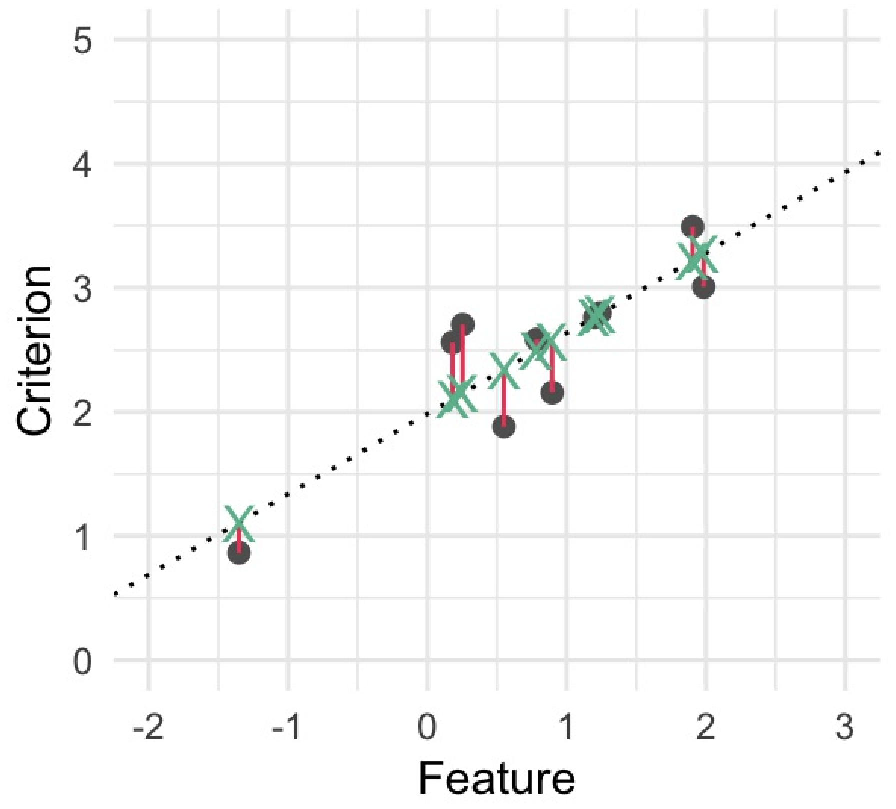


postResample()

The postResample() function **gives a simple summary** of a models' performance in a **regression task**. Simply put the predicted values and the true values inside the function.

```
# evaluate
postResample(glm_fits,
                baselers$income)
```

```
    RMSE  Rsquared       MAE
1173.079     0.821   937.113
```



confusionMatrix()

The `confusionMatrix()` does the same for a models' performance in a **classification task**. Simply put the predicted values and the true values inside the function.

```r
# eyecor to factor
baselers$eyecor <- factor(baselers$eyecor)

# run glm model for classification
eyecor_mod <-
  train(form = eyecor ~ age + height,
        data = baselers,
        method = "glm",
        trControl = ctrl)

# evaluate
confusionMatrix(predict(eyecor_mod),
                baselers$eyecor)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  no yes
       no    0   0
       yes 353 647

               Accuracy : 0.647
                 95% CI : (0.616, 0.677)
    No Information Rate : 0.647
    P-Value [Acc > NIR] : 0.514

                  Kappa : 0

 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.000
            Specificity : 1.000
         Pos Pred Value :   NaN
         Neg Pred Value : 0.647
             Prevalence : 0.353
         Detection Rate : 0.000
   Detection Prevalence : 0.000
      Balanced Accuracy : 0.500

       'Positive' Class : no
```

Practical