TRUMAN STATE UNIVERSITY

Professional Program in
DATA SCIENCE

PDAT615G: Machine Learning

Module 1 – Introduction

DISTINCT BY DESIGN

Module 1: Introduction

- Supervised, unsupervised and reinforcement learning
- Regression models
    - Single/multiple variable for numeric prediction
    - Logistic regression for classification
- Review of R programming techniques
    - Defining functions
    - Vectorized commands & loop-like constructions

Types of Machine Learning

Supervised learning trains a model to use certain features, or variables, to predict a response.

- Training data contains both explanatory variables and the response.
- Models are trained to be able to use only the explanatory variables to predict new responses.
- Regression is an example of a supervised learning technique.

Unsupervised learning analyzes patterns in data without a particular response variable specified.

- Clustering is one example. What groupings appear in the data?
- Dimension reduction is another. Do fewer variables describe the data just as well?

Reinforcement learning creates an algorithm by rewarding “good” behavior and punishing “bad” behavior.

- Reinforcement learners need
  - a way to measure the desirability of the end state,
  - methods to explore the space of all possible actions, and
  - methods to find optimal strategies.
- Examples might include
  - a game-playing algorithm, or
  - a planner for supply-chain management.

Regression Models

Linear regression models can be used to predict a numeric response variable.

Linear regression models predict the value of one variable (the response, or dependent variable) based on other variables (the predictor, or independent, variables):

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k + \varepsilon.$$

- **Response (dependent) variable:** denoted Y

  The response variable Y must be a numeric variable.

- **Explanatory (independent) variables:** denoted $x_1, x_2, \ldots, x_k$

  Predictors $x_1, \ldots, x_k$ can be numeric or categorical variables.

If we have **MORE THAN ONE** independent variable, say x_1 and x_2, or x_1, x_2 and x_3, the model is a *multiple linear regression model*.

Regression coefficients are chosen to minimize the sum of squared residuals.

- We assume a model form:

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k + \varepsilon.$$

- The β's are estimated by $b_0, b_1, \ldots$, giving the regression equation to calculate $\hat{y}$, the predicted value of y.

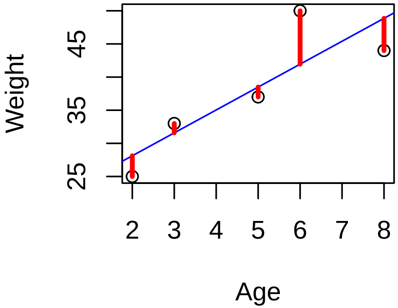
$$\hat{y} = b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_k x_k.$$

- The coefficients $b_0, b_1, \ldots, b_k$ are chosen to minimize

$$\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Regression models are calculated with `lm`, and predicted values obtained with `predict`.

```r
df <- data.frame(
  Age = c(2, 5, 3, 8, 6),
  Weight = c(25, 37, 33, 44, 50)
)
wt.fit <- lm(Weight ~ Age, data=df)
df$Pred.Wt <- predict(wt.fit)
with(df, {
  plot(x=Age, y=Weight)
  abline(wt.fit, col="blue")
  segments(x0=Age, x1=Age,
           y0=Weight, y1=Pred.Wt,
           col="red", lwd=3)
})
coef(wt.fit)
```



```
(Intercept)         Age
    21.2105      3.4561
```

Logistic regression predicts the likelihood of a Yes/No categorical variable.

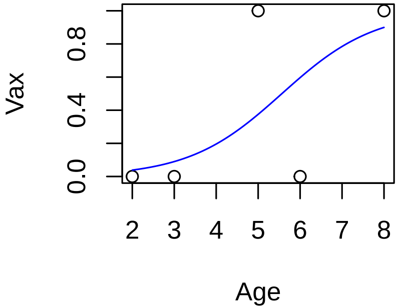
- Suppose p represents the probability of a “Yes” response.
- Perhaps p is related to one or more predictor variables.
- Logistic regression models the *log odds* as a linear function of one or more variables:

$$\log\left(\frac{p}{1-p}\right) = \beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\varepsilon, \text{ or } p = \frac{1}{1+e^{-(\beta_0+\beta_1 x_1+\cdots+\varepsilon)}}$$

- Note: Functions of the form $p = \frac{1}{1+e^{-z}}$ asymptotically approach 0 and 1 in an “S”-shaped curve, which is good for modeling probabilities.

Logistic regression models are calculated with the `glm` command and `family="binomial"`.

```r
# Note: glm accepts 0/1 or
# factor variables.
df$Vax <- c(0, 1, 0, 1, 0)
vax.fit <- glm(Vax ~ Age, data=df,
               family="binomial")
with(df, plot(x=Age, y=Vax))
# Plot the curve
df.curve <- data.frame(
  Age = seq(2, 8, by=0.01)
)
df.curve$p <- predict(vax.fit,
  newdata=df.curve, type="response")
with(df.curve,
  lines(x=Age, y=p, col="blue"))
```

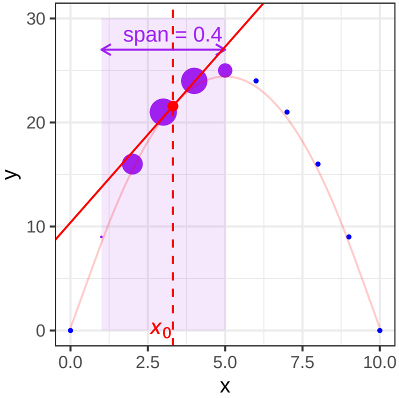


LOESS creates a "local" regression model near each point by using only near points in model construction.

- Historically, two related terms have been used.
- LOWESS – LOcally Weighted Scatterplot Smoothing
  - `lowess()`: Calculates a fit for one *x* and one *y*. Outputs predicted points for plotting.
  - Defaults to first-degree (linear) fitting.
- LOESS – LOcally Estimated Scatterplot Smoothing
  - Interface similar to `lm()`.
  - Outputs a model for use with `predict()`.
  - Generalizes LOWESS. Can accommodate multiple predictor variables.
  - Defaults to second-degree polynomial fitting.

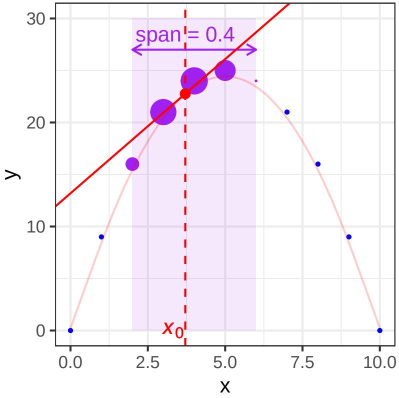
Example: Degree 1 LOESS Fit, 40% Span

- We want to find a function $\hat{f}(x)$ that fits the data.
- Step 1: For a base point x_0, find the closest 40% of the data points.
- Step 2: Weight each point within the span according to its distance from x_0.
- Step 3: Calculate a weighted linear regression using those points.
- Step 4: $\hat{f}(x_0)$ is the point on that line above x_0
- Step 5: Repeat for other x's.



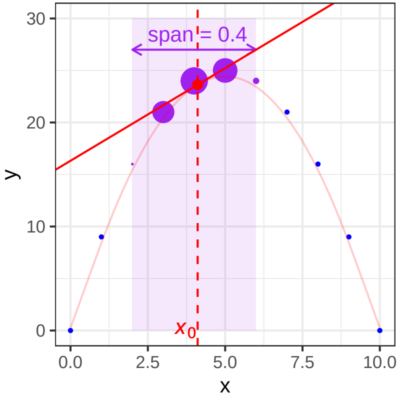
Example: Degree 1 LOESS Fit, 40% Span

- We want to find a function $\hat{f}(x)$ that fits the data.
- Step 1: For a base point x_0, find the closest 40% of the data points.
- Step 2: Weight each point within the span according to its distance from x_0.
- Step 3: Calculate a weighted linear regression using those points.
- Step 4: $\hat{f}(x_0)$ is the point on that line above x_0
- Step 5: Repeat for other x's.



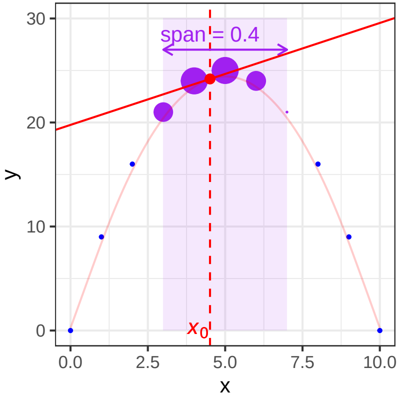
Example: Degree 1 LOESS Fit, 40% Span

- We want to find a function $\hat{f}(x)$ that fits the data.
- Step 1: For a base point x_0, find the closest 40% of the data points.
- Step 2: Weight each point within the span according to its distance from x_0.
- Step 3: Calculate a weighted linear regression using those points.
- Step 4: $\hat{f}(x_0)$ is the point on that line above x_0
- Step 5: Repeat for other x's.



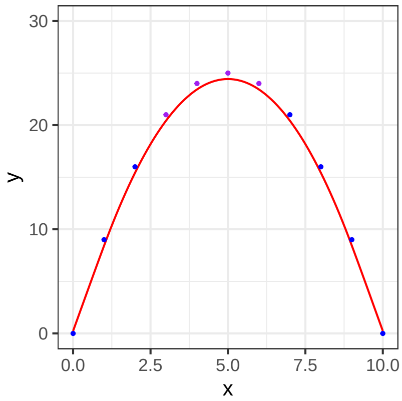
Example: Degree 1 LOESS Fit, 40% Span

- We want to find a function $\hat{f}(x)$ that fits the data.
- Step 1: For a base point x_0, find the closest 40% of the data points.
- Step 2: Weight each point within the span according to its distance from x_0.
- Step 3: Calculate a weighted linear regression using those points.
- Step 4: $\hat{f}(x_0)$ is the point on that line above x_0
- Step 5: Repeat for other x's.

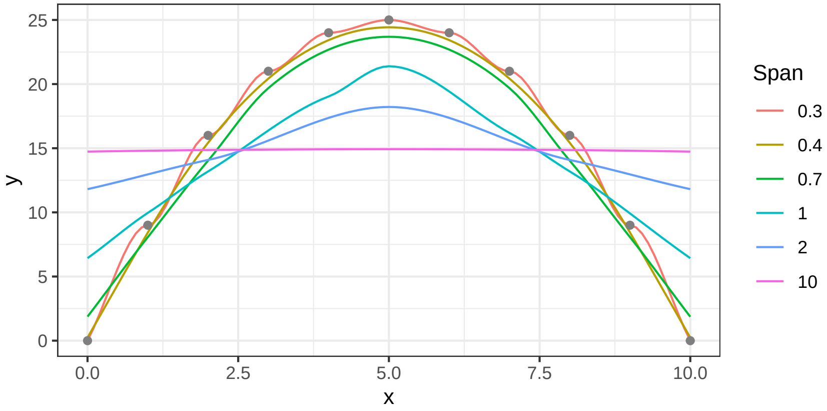


Example: Degree 1 LOESS Fit, 40% Span

- We want to find a function $\hat{f}(x)$ that fits the data.
- Step 1: For a base point x_0, find the closest 40% of the data points.
- Step 2: Weight each point within the span according to its distance from x_0.
- Step 3: Calculate a weighted linear regression using those points.
- Step 4: $\hat{f}(x_0)$ is the point on that line above x_0
- Step 5: Repeat for other x's.



Increasing the span moves from a more “local” to a more “global” fit.



Note: Increasing span larger than 1 considers all the data points and evens out the weights, approaching a standard linear regression.

Evaluating Predictive Models

Mean squared error (MSE) or root mean squared error (RMSE) are often used with numeric predictors.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y - \hat{y})^2$$

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y - \hat{y})^2}$$

RMSE is measured in the same units as the data.

```
df
```

```
  Age Weight Pred.Wt Vax
1   2     25  28.123   0
2   5     37  38.491   1
3   3     33  31.579   0
4   8     44  48.860   1
5   6     50  41.947   0
```

```
sqrt(mean((df$Weight - df$Pred.Wt)^2))
```

```
[1] 4.5267
```

```
library(caret)
RMSE(pred=df$Pred.Wt, obs=df$Weight)
```

```
[1] 4.5267
```

Accuracy, Sensitivity and Specificity give information about classifiers.

	Act. Pos	Act. Neg
Pr. Pos	A	B
Pr. Neg	C	D

Accuracy Percentage of correct predictions. $\left(\frac{A+D}{A+B+C+D}\right)$

Sensitivity Percentage of actual positives correctly predicted to be positive. $\left(\frac{A}{A+C}\right)$

Specificity Percentage of actual negatives correctly predicted to be negative. $\left(\frac{D}{B+D}\right)$

```r
df$Prob.Vax <-
  predict(vax.fit, type="response")
df$Pred.Vax <-
  as.numeric(df$Prob.Vax > 0.50)
# Accuracy
mean(df$Vax == df$Pred.Vax)
```

```
[1] 0.6
```

```r
# Sensitivity
with(df[df$Vax==1, ],
  mean(Vax == Pred.Vax))
```

```
[1] 0.5
```

```r
# Specificity
with(df[df$Vax==0, ],
  mean(Vax == Pred.Vax))
```

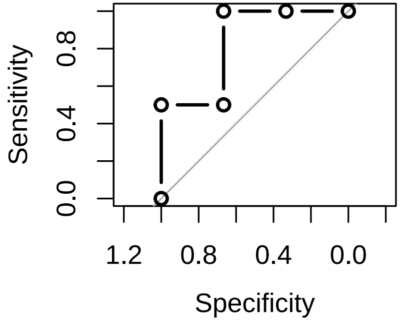
```
[1] 0.66667
```

For classifiers that use a probability cut-off, the ROC curve describes the trade-off between sensitivity and specificity.

- For a set of thresholds between 0 and 1, predict the binary response.
- Calculate the pair (Specificity, Sensitivity).
- Plot the point on the graph.
- When AUC ≈ 1, a threshold can be chosen to make both sensitivity and specificity high.

```
  Vax Prob.Vax
1   0 0.038597
2   0 0.090001
3   1 0.375094
4   0 0.596565
5   1 0.899743
```

```r
library(pROC)
vax.roc <- roc(Vax ~ Prob.Vax, data=df
plot(vax.roc, type="b")
```



```r
auc(vax.roc)
```

```
Area under the curve: 0.833
```

Predictive models should be evaluated on data not used to train the model.

Simplest: Create training and testing sets.

```r
# Base R: The `sample` command
# selects random indices.
ind <- sample(1:nrow(df),
        round(0.70*nrow(df)))
df.train <- df[ind, ]
df.test <- df[-ind, ]
df.train[, 1:4]
```

```
  Age Weight Pred.Wt Vax
3   3     33  31.579   0
5   6     50  41.947   0
1   2     25  28.123   0
4   8     44  48.860   1
```

```r
df.test[, 1:4]
```

```
  Age Weight Pred.Wt Vax
2   5     37  38.491   1
```

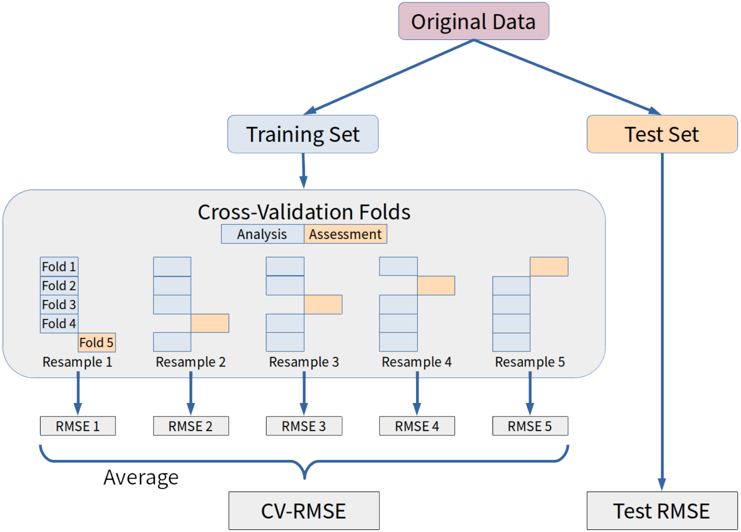
```r
# Caret's `createDataPartition`
# creates indices as well.
# Given a factor, it will try to
# split them evenly.
ind <- createDataPartition(df$Vax,
                p=0.7, list=FALSE)
df.train <- df[ind, ]
df.test <- df[-ind, ]
df.train[, 1:4]
```

```
  Age Weight Pred.Wt Vax
1   2     25  28.123   0
2   5     37  38.491   1
4   8     44  48.860   1
5   6     50  41.947   0
```

```r
df.test[, 1:4]
```

```
  Age Weight Pred.Wt Vax
3   3     33  31.579   0
```

Cross-validation creates many “analysis” and “assessment” sets from the same data set.



Review of R Programming Techniques

Putting commonly-used code in a function can speed coding and prevent copy/paste mistakes.

```
funct.name <- function(<arguments>){...code...}
```

- Multiple commands are placed within braces { }
- Default values can be defined by setting function arguments equal to a value within the definition.
- The `return` keyword specifies the function return value. Otherwise the last value calculated in the function.
- The `warning` and `stop` functions can be used to throw warning and error messages.

```r
sumn <- function(x, n=length(x)){sum(x[1:n], na.rm=TRUE)}
sumn(1:5, n=4)
```

```
[1] 10
```

If dealing with outside data/user input, it's good form to verify inputs. With your own code, it could help debug.

```r
sumn <- function(x, n=length(x)){
  if(!is.numeric(x) || length(dim(x)))
    stop("Requires one-dimensional
          numeric vector.")
  if(!is.numeric(n) || length(n) > 1)
    stop("n must be a single
          numeric value.")
  if(round(n) != n){
    n <- round(n)
    warning("n must be an integer
             and will be rounded.")
  }
  if(n < 1 || n > length(x))
    stop("n must be an integer between
          1 and length of x.")
  return(sum(x[1:n], na.rm=TRUE))
}
```

```r
a <- 1:5
b <- c(1, 2, NA, 4, 5)
sumn(a)
```

```
[1] 15
```

```r
sumn(a, n=3)
```

```
[1] 6
```

```r
sumn(b, n=3)
```

```
[1] 3
```

```r
sumn(a, n=3.2)
```

```
Warning in sumn(a, n = 3.2): n must be
             and will be rounded.
[1] 6
```

Vectorized functions are usually more efficient than loops. Many R functions are vectorized.

Square roots of the whole numbers from 1 to 5.

```
a <- 1:5
a
```

```
[1] 1 2 3 4 5
```

```
sqrt(a)
```

```
[1] 1.0000 1.4142 1.7321 2.0000 2.2361
```

Single values are repeated as needed in vectorized arithmetic.

```
# Note: This is a long way to
# calculate variance.
a
```

```
[1] 1 2 3 4 5
```

```
a.mean <- mean(a)
a.mean
```

```
[1] 3
```

```
a - a.mean
```

```
[1] -2 -1  0  1  2
```

```
sum((a - a.mean)^2) / (length(a) - 1)
```

```
[1] 2.5
```

The s/lapply family of functions iterate functions over vector input. Usually more efficient than a loop.

- lapply always returns a list.
- sapply tries to return a simplified object.

```r
a <- 1:5
lapply(a, FUN=sqrt)
```

```
[[1]]
[1] 1

[[2]]
[1] 1.4142

[[3]]
[1] 1.7321

[[4]]
[1] 2
```

```r
sapply(a, FUN=sqrt)
```

```
[1] 1.0000 1.4142 1.7321 2.0000 2.2361
```

```r
# Applied to an anonymous function.

sapply(a, FUN=function(x){sqrt(x)+2})
```

```
[1] 3.0000 3.4142 3.7321 4.0000 4.2361
```

The purrr package (in tidyverse) gives the map family of commands, which are similar to s/lapply.

```r
map(1:5, sqrt)
```

```
[[1]]
[1] 1

[[2]]
[1] 1.4142

[[3]]
[1] 1.7321

[[4]]
[1] 2

[[5]]
[1] 2.2361
```

```r
# Note that the "suffix" gives the
# desired type of output.
map_dbl(1:5, sqrt)
```

```
[1] 1.0000 1.4142 1.7321 2.0000 2.2361
```

```r
# The "formula" format specifies
# an anonymous function compactly.
# using `.x` as the input.
map_dbl(1:5, ~ sqrt(.x) + 2)
```

```
[1] 3.0000 3.4142 3.7321 4.0000 4.2361
```

Sometimes a loop is just easier, especially if the function is complicated. So don't feel bad for using one if you have to!

```r
for(i in 1:5){
  # Note that `print` is needed
  # inside a loop to show results.
  print(sqrt(i))
}
```

```
[1] 1
[1] 1.4142
[1] 1.7321
[1] 2
[1] 2.2361
```

```r
# To store loop results, it's good to
# initialize variables first.
a.sqrt <- numeric(5)
# Note that these "type" commands with
# a number input give empty vectors.
a.sqrt
```

```
[1] 0 0 0 0 0
```

```r
for(i in 1:5){
  a.sqrt[i] <- sqrt(i)
}
a.sqrt
```

```
[1] 1.0000 1.4142 1.7321 2.0000 2.2361
```