BZAN X6 / ACCT 7374 – Advanced Modeling

Books:

- **R for Data Science.** Garret Grolemund and Hadley Wickham. http://r4ds.had.co.nz/
- - **An Introduction to Statistical Learning.** Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani. ISBN-13: 978-1461471370 *(pdf on blackboard)*
- - **Bayesian Data Analysis.** Andrew Gelman et. al. 3rd Edition ISBN 978-1439840955 *(2nd edition on blackboard – should get you by)*
- - **Causal Inference in Statistics - A Primer.** Judea Pearl ISBN-10: 1119186846, ISBN-13: 978-1119186847

Blackboard Additional Resources *(included JPM Analytics Strategy)*

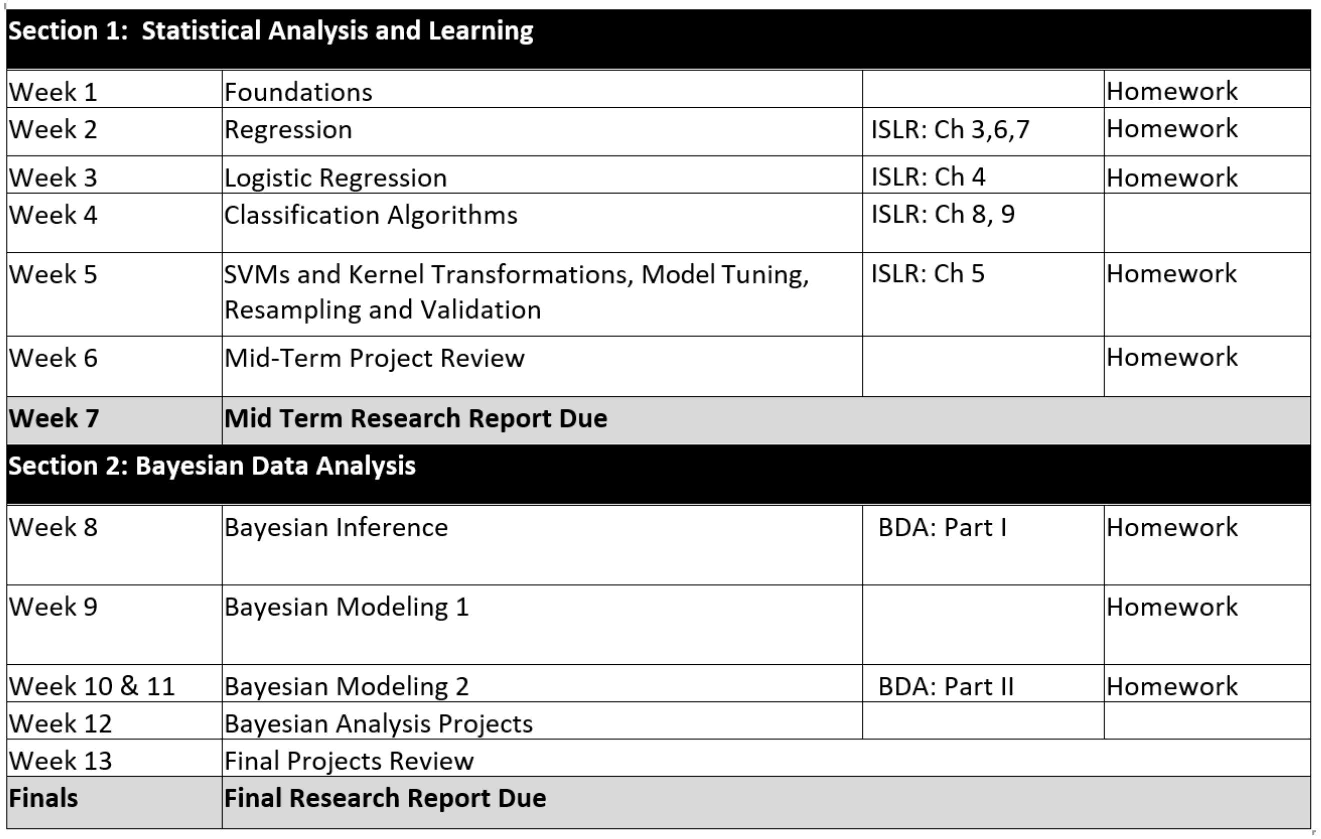
Class Materials:

https://github.com/ellenwterry *(code=>clone will update you each time to log in)*

BAUER COLLEGE OF BUSINESS UNIVERSITY of HOUSTON

BZAN X6 / ACCT 7374 – Advanced Modeling

Tentative Schedule *(very tentative considering all the confounding factors)*



Section 1: Statistical Analysis and Learning			
Week 1	Foundations		Homework
Week 2	Regression	ISLR: Ch 3,6,7	Homework
Week 3	Logistic Regression	ISLR: Ch 4	Homework
Week 4	Classification Algorithms	ISLR: Ch 8, 9	
Week 5	SVMs and Kernel Transformations, Model Tuning, Resampling and Validation	ISLR: Ch 5	Homework
Week 6	Mid-Term Project Review		Homework
Week 7	**Mid Term Research Report Due**		
Section 2: Bayesian Data Analysis			
Week 8	Bayesian Inference	BDA: Part I	Homework
Week 9	Bayesian Modeling 1		Homework
Week 10 & 11	Bayesian Modeling 2	BDA: Part II	Homework
Week 12	Bayesian Analysis Projects		
Week 13	Final Projects Review		
Finals	**Final Research Report Due**		

Research Reports must be submitted in RMD (https://rmarkdown.rstudio.com/) + PDFs - underlying code in supplemental R files. Plan to do presentations *(if time permits)*. Reports must include references to reading material *(keep up with reading material)*. Topics may be the same, but data and models must be original *(master data generation)*. Originality will be graded.

BAUER
COLLEGE OF BUSINESS
UNIVERSITY of HOUSTON

Advanced Modeling – Foundations

This section will introduce some of the foundational tools of advanced analytics that will be helpful throughout your career as an analyst. Analytics is a search for functions *(model selection)* and their parameters. In this intro, we'll focus on finding parameter values, using a range of methodologies:

Analytic:

- Ordinary Least Squares *(OLS)* and derivative based solutions

Linear Algebra:

- Normal Equations
- Matrix Decomposition *(linear solvers – includes single value and Chorlesky)*
  - QRH Decomposition

Computational:

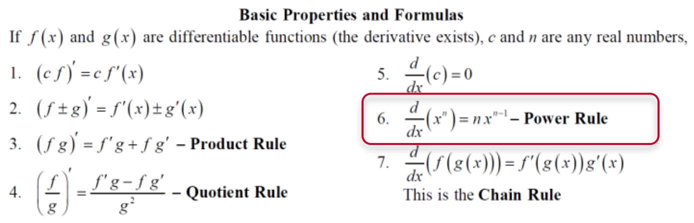
- Gradient Descent
- Maximum Likelihood

Sampling:

- Markov Chain Monte Carlo *(MCMC)*

Analytic Solutions

Calculus cheat sheet (blackboard)



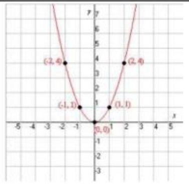
Basic Properties and Formulas

If $f(x)$ and $g(x)$ are differentiable functions (the derivative exists), c and n are any real numbers,

1. $(cf)' = cf'(x)$
2. $(f \pm g)' = f'(x) \pm g'(x)$
3. $(fg)' = f'g + fg'$ – **Product Rule**
4. $\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$ – **Quotient Rule**
5. $\frac{d}{dx}(c) = 0$
6. $\boxed{\frac{d}{dx}(x^n) = nx^{n-1}\text{ – Power Rule}}$
7. $\frac{d}{dx}\left(f\left(g(x)\right)\right) = f'\left(g(x)\right)g'(x)$

   This is the **Chain Rule**

Harold's cheat sheet (blackboard)

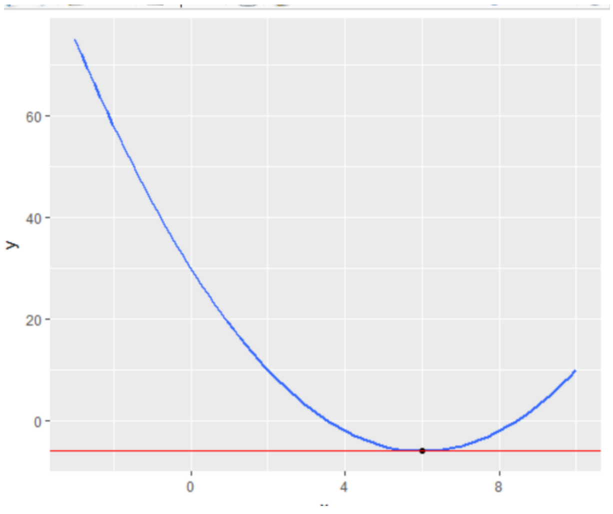
Quadratic or Square	$f(x) = x^2$	

Recall: the first derivative gets us the tangent, setting slope of the tangent to 0 gets us to the max or min of a quadratic convex function.

x^2 – 12x –10

0 = 2x – 12

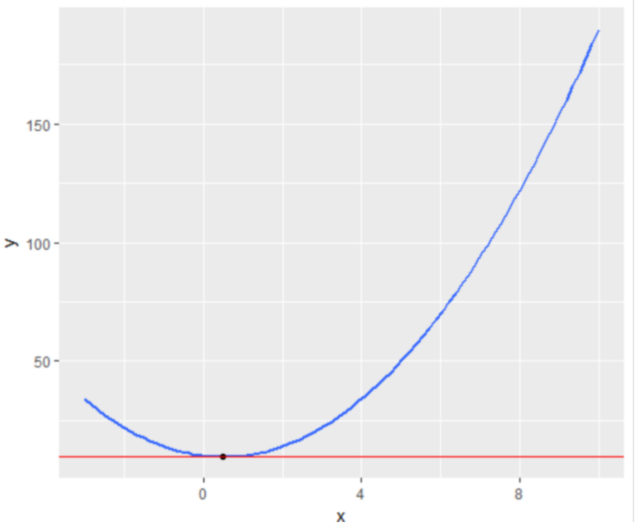
x = 6



$2x^2$ – 2x+10

0 = 4x – 2

x = 0.5

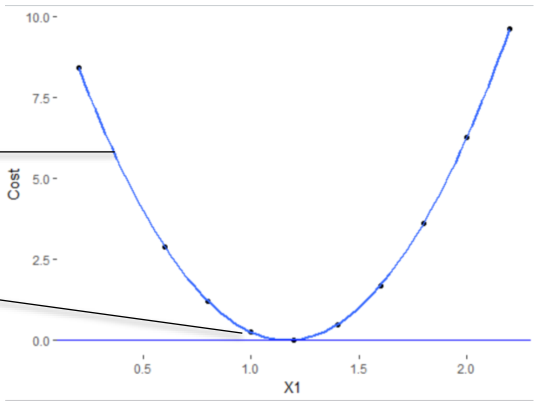


Derivative Based Optimization

Analytic Solutions:

Cost Function (assumed to be convex)

Minimum = first derivative set to 0



Derivatives

Basic Properties/Formulas/Rules

$\frac{d}{dx}(cf(x)) = cf'(x)$, c is any constant. $(f(x) \pm g(x))' = f'(x) \pm g'(x)$

$\frac{d}{dx}(x^n) = nx^{n-1}$, n is any number. $\frac{d}{dx}(c) = 0$, c is any constant.

$(fg)' = f'g + fg'$ – **(Product Rule)** $\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$ – **(Quotient Rule)**

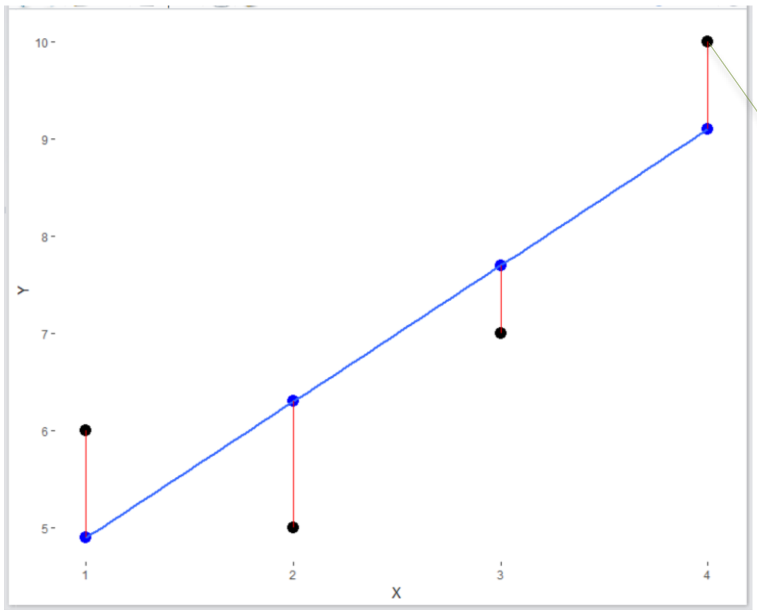
$\frac{d}{dx}(f(g(x))) = f'(g(x))g'(x)$ **(Chain Rule)**

$\frac{d}{dx}(\mathbf{e}^{g(x)}) = g'(x)\mathbf{e}^{g(x)}$ $\frac{d}{dx}(\ln g(x)) = \frac{g'(x)}{g(x)}$

See:
common_derivatives_integrals_cheatsheet

In Blackboard

Ordinary Least Squares



$\beta_0 + 1\beta_1 = 6$
$\beta_0 + 2\beta_1 = 5$
$\beta_0 + 3\beta_1 = 7$
$\beta_0 + 4\beta_1 = 10$

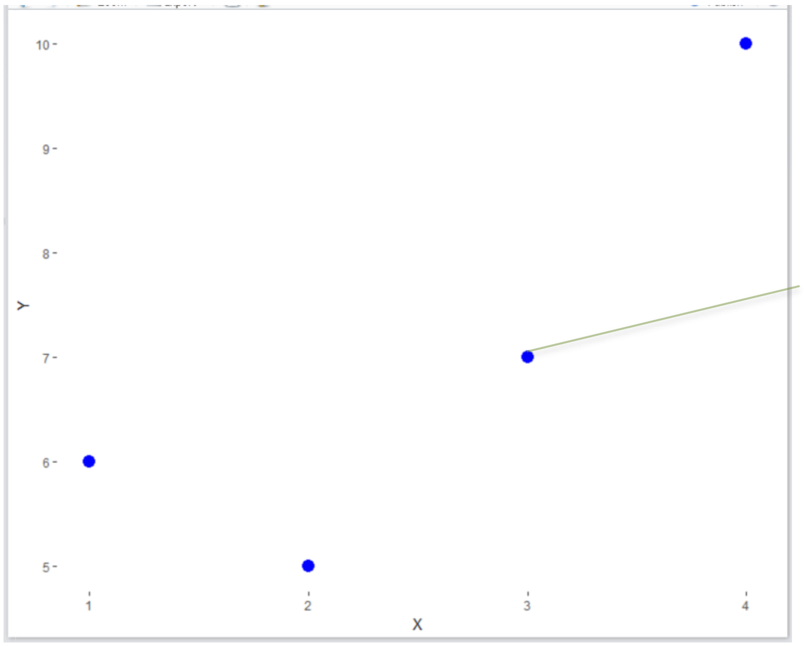
$S(\beta_1, \beta_2) =$

$[6 - (\beta_0 + 1\beta_1)]^2$
$[5 - (\beta_0 + 2\beta_1)]^2$
$[7 - (\beta_0 + 3\beta_1)]^2$
$[10 - (\beta_0 + 4\beta_1)]^2$

This is called the error, or residual
(the difference between the predicted value of the model and the actual value)

We want this to be as small as possible. So, we define error as a function and minimize.

Analytical Solution to Least Squares Regression



Taken from Wikipedia article (good!)
Also see common derivatives_integrals cheat sheet

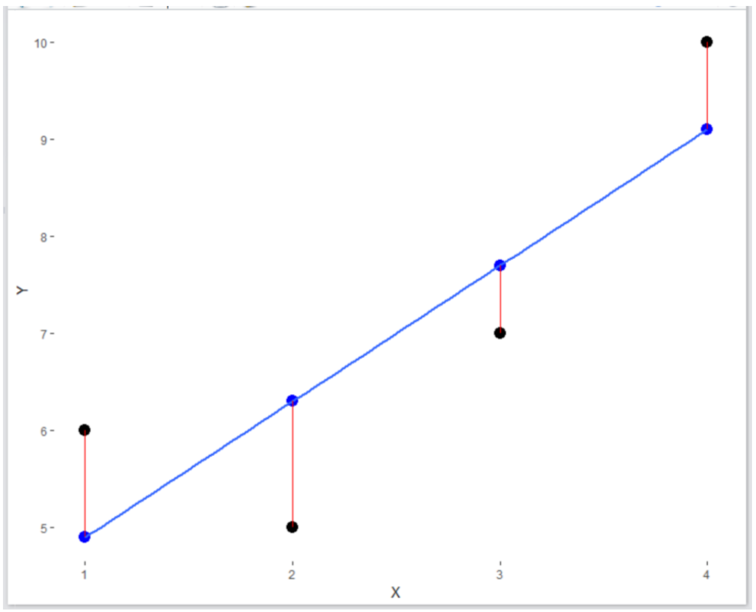
$\beta_0 + 1\beta_1 = 6$
$\beta_0 + 2\beta_1 = 5$
$\beta_0 + 3\beta_1 = 7$
$\beta_0 + 4\beta_1 = 10$

$$S(\beta_1, \beta_2) = \begin{array}{r} [6- (\beta_0 + 1\beta_1)]^2 \\ [5- (\beta_0 + 2\beta_1)]^2 \\ [7- (\beta_0 + 3\beta_1)]^2 \\ [10- (\beta_0 + 4\beta_1)]^2 \end{array}$$

$$= 4\beta_0^2 + 20\beta_0\beta_1 - 56\beta_0 + 30\beta_1^2 - 154\beta_1 + 210$$

$$\frac{\partial S}{\partial \beta_0} = 8\beta_0 + 20\beta_1 = 56$$

$$\frac{\partial S}{\partial \beta_1} = 20\beta_0 + 60\beta_1 = 154$$

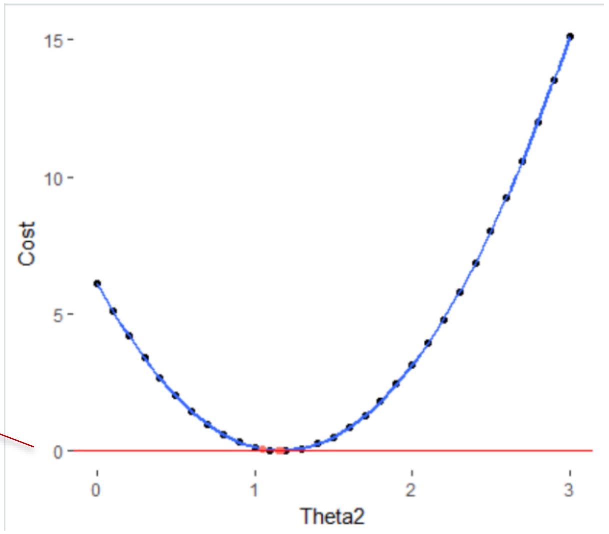


	β_0^2	$\beta_0\beta_1$	β_0	β_1^2	β_1	
$(6-\beta_0 - 1\beta_1)(6-\beta_0 - 1\beta_1)$	1	2	-12	1	-12	36
$(5-\beta_0 - 2\beta_1)(5-\beta_0 - 2\beta_1)$	1	4	-10	4	-20	25
$(7-\beta_0 - 3\beta_1)(7-\beta_0 - 3\beta_1)$	1	6	-14	9	-42	49
$(10-\beta_0 - 4\beta_1)(10-\beta_0 - 4\beta_1)$	1	8	-20	16	-80	100
	4	20	-56	30	-154	210

$$= 4\beta_0^2 + 20\beta_0\beta_1 - 56\beta_0 + 30\beta_1^2 - 154\beta_1 + 210$$

$$\frac{\partial S}{\partial \beta_0} = 0 = 8\beta_0 + 20\beta_1 - 56 \quad = 3.5$$

$$\frac{\partial S}{\partial \beta_1} = 0 = 20\beta_0 + 60\beta_1 - 154 \quad = 1.4$$



analytic derivative based solutions become intractable in the real world

Solving using elimination	8 b_0	20 b_1	=	56
(recall rules of linear algebra)	20 b_0	60 b_1	=	154
LCM 40:	40 b_0	100 b_1	=	280
	-40 b_0	-120 b_1	=	-308
Combine:		-20 b_1	=	-28
solve:		b_1	=	1.4
substitute	8 b_0	28	=	56
solve	8 b_0		=	28
	b_0		=	3.5

Notice how this gets easy when we get down to 1 variable. Once we have b_1, we can substitute b_1's value and solve b_0 (equality / multiplication property – LA is algebra)

This works for any number of variables as long as we can format the problem diagonally:

b_0	b_1	b_2	b_3	b_4	b_5	=	#
	b_1	b_2	b_3	b_4	b_5	=	#
		b_2	b_3	b_4	b_5	=	#
			b_3	b_4	b_5	=	#
				b_4	b_5	=	#
					b_5	=	#

or use R solve

```
X <- matrix( c(8, 20, 20, 60), nrow=2, ncol = 2)
B <- matrix( c(56, 154), nrow=2, ncol = 1)
solve(X, B)
```

```
solve(X, B)
[1,] 3.5
[2,] 1.4
```

In this case, b_5 is easy to find, then we can backsolve all the way up.

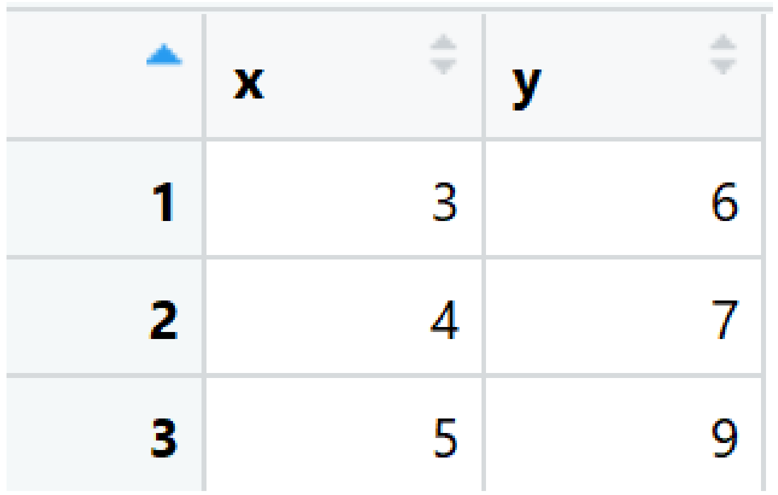
Computers are really good at this – and fast.

Keep this in mind, we'll revisit soon.

BAUER COLLEGE OF BUSINESS UNIVERSITY of HOUSTON

Class Exercise

Find the OLS parameters for the X and Y values presented:



	x	y
1	3	6
2	4	7
3	5	9

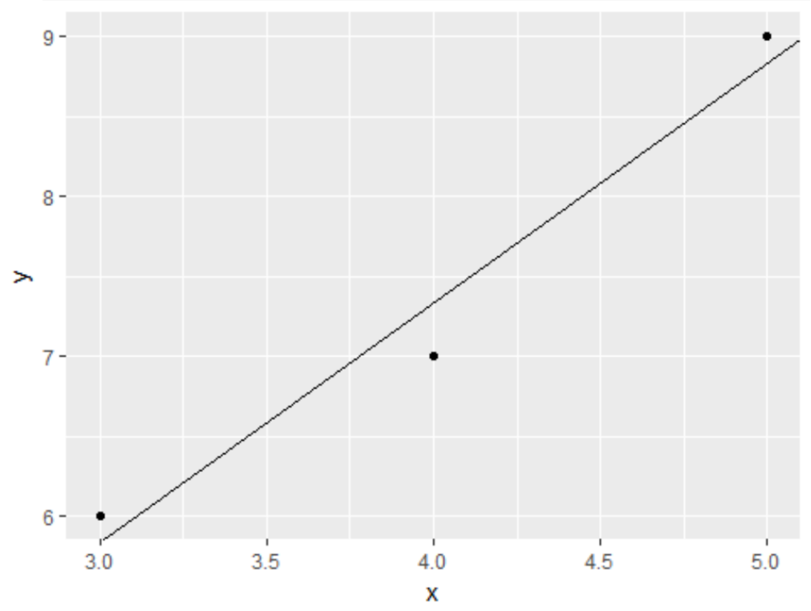
	β_0^2	$\beta_0\beta_1$	β_0	β_1^2	β_1	
$(6-\beta_0 - 3\beta_1)(6-\beta_0 - 3\beta_1)$	1	6	-12	9	-36	36
$(7-\beta_0 - 4\beta_1)(7-\beta_0 - 4\beta_1)$	1	8	-14	16	-56	49
$(9-\beta_0 - 5\beta_1)(9-\beta_0 - 5\beta_1)$	1	10	-18	25	-90	81
	3	24	-44	50	-182	166

$$= 3\beta_0^2 + 24\beta_0\beta_1 - 44\beta_0 + 50\beta_1^2 - 182\beta_1 + 166$$

$$6\beta_0 + 24\beta_1 - 44 = 0$$

$$24\beta_0 + 100\beta_1 - 182 = 0$$

Solving using elimination	6 b_0	24 b_1	=	44
(recall rules of linear algebra)	24 b_0	100 b_1	=	182
LCM 24:	24 b_0	96 b_1	=	176
	-24 b_0	-100 b_1	=	-182
Combine:		-4 b_1	=	-6
solve:		b_1	=	1.5
substitute	6 b_0		36 =	44
solve	6 b_0		=	8
	b_0		=	1.33



Linear Algebra Review

Linear Algebra is the basis of database, numerical, statistical analysis… all built on LA theory:

- We focused on the tidyverse and sql to build a high level understanding of data acquisition and transformation.
- You will need to more granular control and expanded mathematical function as you get into advanced analytics.
- Many essential packages in CRAN are older and don't understand the tidyverse. Most of these are the critical mathematical and statistical packages that you need for deeper analysis.
- Also, many of the packages get close to what you need and you will have to extend to deliver. You'll need LA…

Good review of LA theory / basics:

https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab

don't feel bad – I have to relearn this stuff every time!!

That said, this stuff is PFM!!

A matrix is m x n *(rows, columns)*, so A[2,1] is the second row, first column element in a matrix
(note: in R the first row and column is [1,1], in python (numpy), it's [0,0])

```
#create a matrix and a vector (several ways to do it)
A <- matrix (c(2, 2)) # this creates a vector
A <- cbind(A, c(3,1)) # then combine vectors with cbind or
A <- matrix (c(2,3), nrow =1)
A <- rbind(A, c(2,1)) # then combine rows
A <- matrix( c(2, 2, 3, 1), nrow=2, ncol = 2) # or just create the matrix at once.

B <- c(3,2) # vector 1
C <- c(0,1) # Vector 2
D <- A #easy to duplicate data structures
D <- A[,1] # or parts - notice that D becomes a vector
D <- cbind(D, A[,2]) # and back again
D <- t(D) #transpose a matrix
```

A matrix transposition rotates the matrix on the main diagonal (from 1,1)

(cbind and rbind are older version of bind_cols and bind_rows in tidyverse)

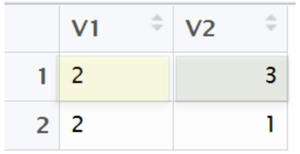
basic matrix operations

Operator or Function	Description
A + B	*Must be same structure*
A - B	*Addition / Subtraction always an element operation*
t(A)	Transpose
A * B	***Element*** multiplication *(the **product** of vectors or matrices – e.g., product * price)*
A %*% B	***Matrix*** multiplication
A %o% B	Outer product. **AB'**
crossprod(A,B) **crossprod(A)**	**A'B** and **A'A** respectively. (cross product gives a vector orthogonal to 2 vectors)

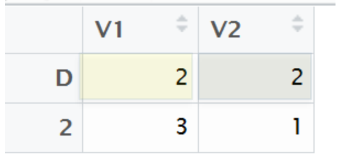
D <- A+D # has to be same structure
E <- B+C
E <- B-C

A[1,1] * B[1] = E[1,1]
A[1,2] * B[1] = E[1,2]…

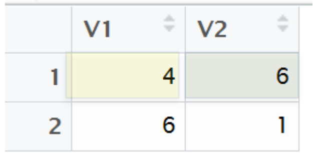
E <- A*B # elements



	V1	V2
1	2	3
2	2	1



	V1	V2
D	2	2
2	3	1



	V1	V2
1	4	6
2	6	1

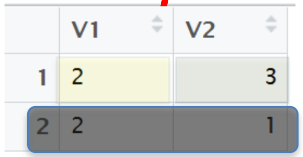
(A[1,1] * B[1,1]) + (A[1,2] * B[2,1]) = E[1,1]
(A[1,1] * B[1,2]) + (A[1,2] * B[2,2]) = E[1,2]…

(think linear equation)

G <- A **%*%** B

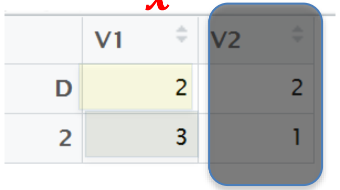
# also called dot product

β



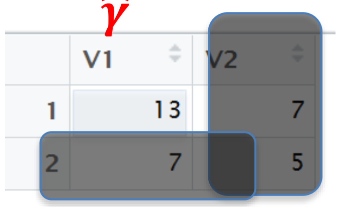
	V1	V2
1	2	3
2	2	1

x

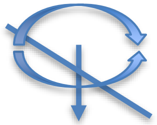


	V1	V2
D	2	2
2	3	1

$\hat{\gamma}$



	V1	V2
1	13	7
2	7	5



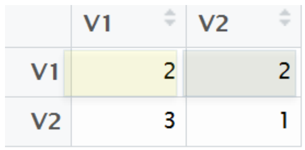
https://en.wikipedia.org/wiki/Transpose

t(A)

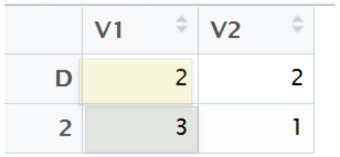
*We will **often** use dot product operations to solve linear equations. Keep in mind that it’s COLUMNS on the left * ROWS on the right*

H <- **crossprod** (A,B)

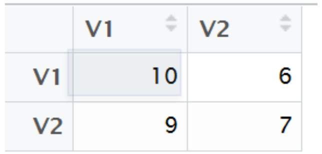
*#(equivalent to t(A) * B)*



	V1	V2
V1	2	2
V2	3	1



	V1	V2
D	2	2
2	3	1



	V1	V2
V1	10	6
V2	9	7

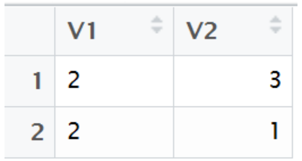
these are mechanisms for building dot product transformations

The resulting matrix can be used for analysis of the original data.

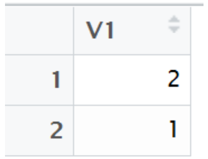
diagonals and determinants

diag(x) Creates diagonal matrix with elements of **x** in the principal diagonal

E <- **diag**(A) $\Longrightarrow$

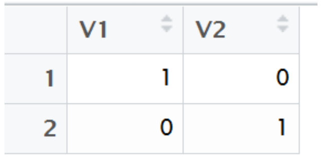


	V1	V2
1	2	3
2	2	1



	V1
1	2
2	1

I <- diag(2) $\Longrightarrow$

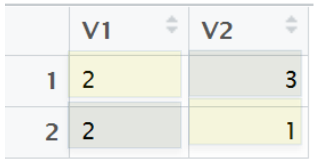


	V1	V2
1	1	0
2	0	1

If you feed it a matrix, it will give you back a vector of the diagonal
If you feed it a number, it will create an identity matrix with that number of rows

J <- det(A)

used to solve linear systems of equations



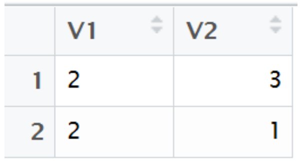
	V1	V2
1	2	3
2	2	1

(2*1)-(3*2) =

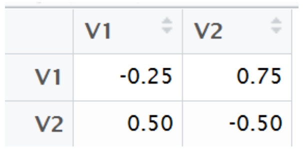
```
[1] -4
```

K <- **solve**(A)

used to achieve matrix division, among other things

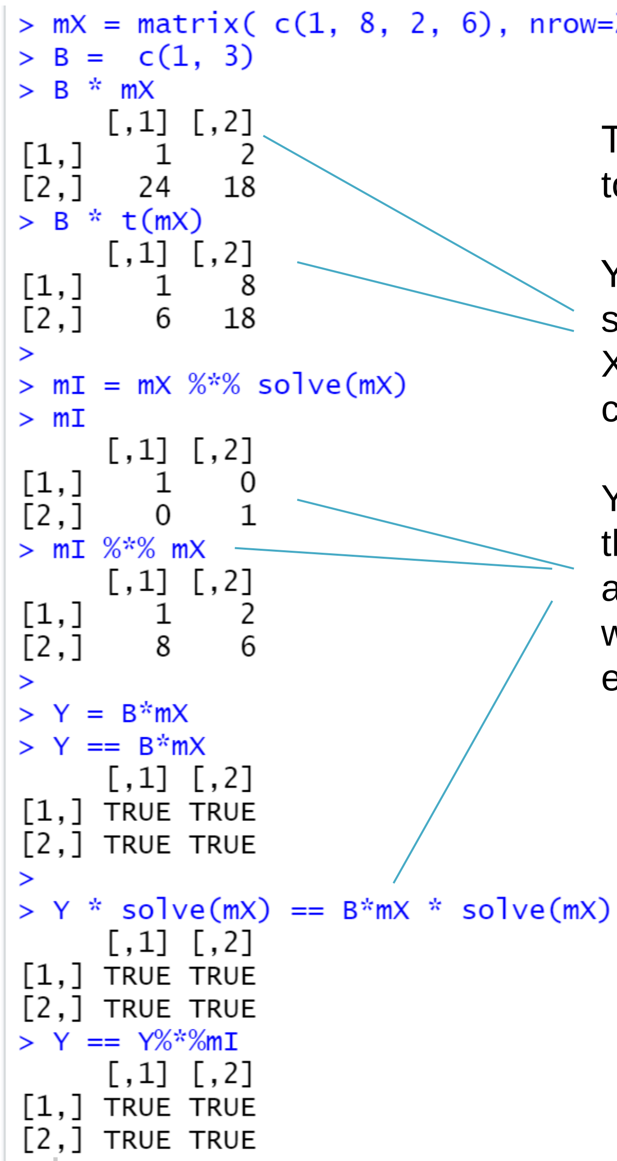


	V1	V2
1	2	3
2	2	1



	V1	V2
V1	-0.25	0.75
V2	0.50	-0.50

***To get the inverse (A)$^{-1}$: swap** the positions of a and d, put **negatives** in front of b and c, and **divide** everything by the determinant (**that's why you have solve ☺**) a matrix * it's inverse equals the identity (definition of inverse) $A^{-1}A = I$*
*Check to see if K*A is a matrix of ones (not the same as an identity matrix) L <- K*A*

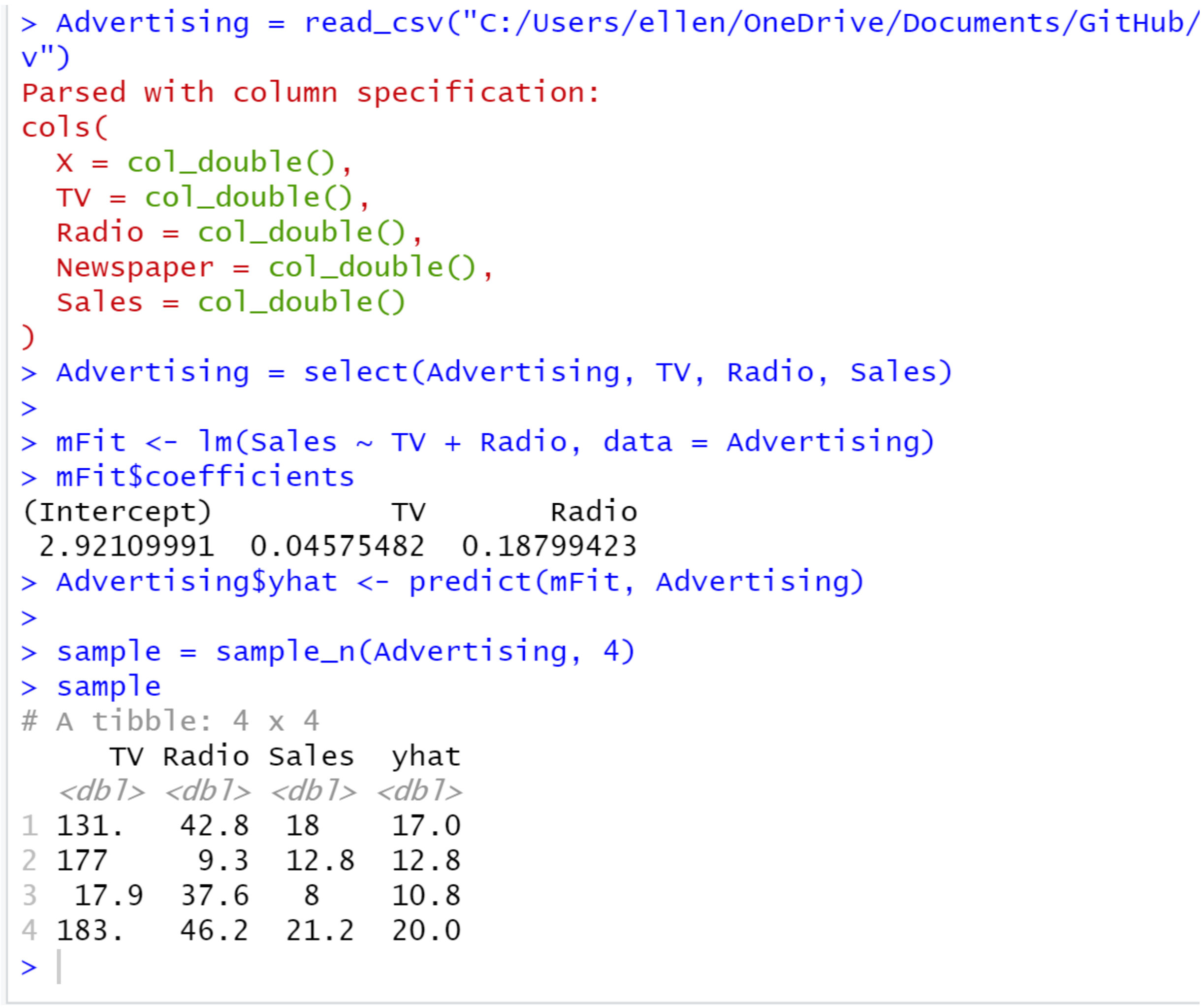


```
> mX = matrix( c(1, 8, 2, 6), nrow=2, ncol = 2)
> B =  c(1, 3)
> B * mX
     [,1] [,2]
[1,]    1    2
[2,]   24   18
> B * t(mX)
     [,1] [,2]
[1,]    1    8
[2,]    6   18
>
> mI = mX %*% solve(mX)
> mI
     [,1] [,2]
[1,]    1    0
[2,]    0    1
> mI %*% mX
     [,1] [,2]
[1,]    1    2
[2,]    8    6
>
> Y = B*mX
> Y == B*mX
     [,1] [,2]
[1,] TRUE TRUE
[2,] TRUE TRUE
>
> Y * solve(mX) == B*mX * solve(mX)
     [,1] [,2]
[1,] TRUE TRUE
[2,] TRUE TRUE
> Y == Y%*%mI
     [,1] [,2]
[1,] TRUE TRUE
[2,] TRUE TRUE
```

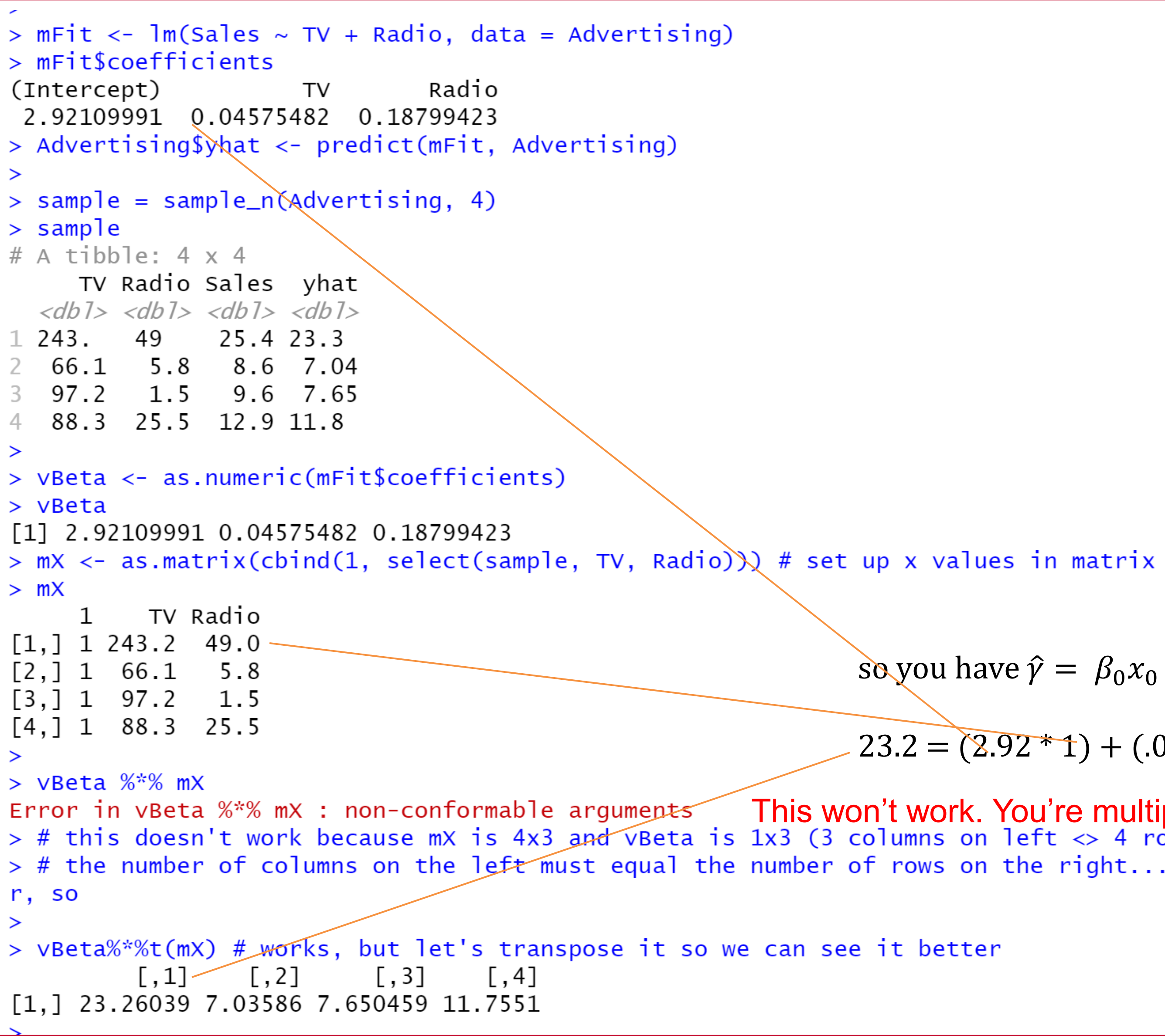
There are many different ways to apply matrix operations, but here's a few to help this sink in:

You'll use transpose ALL the time in linear equations – often because you simply have a vector of β values (B), and you want to multiply those by the X values (to get Y), and you have to line up your columns with β. As you can see, B*mX gets a very different result from B*t(mX).

You use inverse matrices to solve equations (matrix algebra generally has the same properties as regular algebra (identity, equality, inverse, associative, distributive, etc.) and you can use an inverse matrix like you would use a reciprocal in real numbers – to create a 1 on one side of an equation.



```
> Advertising = read_csv("C:/Users/ellen/OneDrive/Documents/GitHub/
v")
Parsed with column specification:
cols(
  X = col_double(),
  TV = col_double(),
  Radio = col_double(),
  Newspaper = col_double(),
  Sales = col_double()
)
> Advertising = select(Advertising, TV, Radio, Sales)
>
> mFit <- lm(Sales ~ TV + Radio, data = Advertising)
> mFit$coefficients
(Intercept)          TV       Radio
 2.92109991  0.04575482  0.18799423
> Advertising$yhat <- predict(mFit, Advertising)
>
> sample = sample_n(Advertising, 4)
> sample
# A tibble: 4 x 4
     TV Radio Sales  yhat
  <dbl> <dbl> <dbl> <dbl>
1 131.   42.8  18    17.0
2 177     9.3  12.8  12.8
3  17.9  37.6   8    10.8
4 183.   46.2  21.2  20.0
> 
```



```
> mFit <- lm(Sales ~ TV + Radio, data = Advertising)
> mFit$coefficients
(Intercept)          TV       Radio 
 2.92109991  0.04575482  0.18799423 
> Advertising$yhat <- predict(mFit, Advertising)
> 
> sample = sample_n(Advertising, 4)
> sample
# A tibble: 4 x 4
     TV Radio Sales  yhat
  <dbl> <dbl> <dbl> <dbl>
1 243.   49    25.4 23.3 
2  66.1   5.8   8.6  7.04
3  97.2   1.5   9.6  7.65
4  88.3  25.5  12.9 11.8 
> 
> vBeta <- as.numeric(mFit$coefficients)
> vBeta
[1] 2.92109991 0.04575482 0.18799423
> mX <- as.matrix(cbind(1, select(sample, TV, Radio))) # set up x values in matrix
> mX
     1    TV Radio
[1,] 1 243.2  49.0
[2,] 1  66.1   5.8
[3,] 1  97.2   1.5
[4,] 1  88.3  25.5
> 
> vBeta %*% mX
Error in vBeta %*% mX : non-conformable arguments
> # this doesn't work because mX is 4x3 and vBeta is 1x3 (3 columns on left <> 4 rows on right)
> # the number of columns on the left must equal the number of rows on the right... EXACTLY in that orde
r, so
> 
> vBeta%*%t(mX) # works, but let's transpose it so we can see it better
         [,1]    [,2]     [,3]    [,4]
[1,] 23.26039 7.03586 7.650459 11.7551
```

so you have $\hat{y} = \beta_0 x_0 + \beta_1 x_1$ ...

$23.2 = (2.92 * 1) + (.045 * 243) + (.18 * 49)$

This won't work. You're multiplying 3 columns by 4 rows

So, starting with the basic linear equation, and applying a little algebra:

$$X\,\beta = Y$$

$$X^T X\,\beta = X^T Y \qquad \textit{equality}$$

$$\underbrace{(X^T X)^{-1}\,(X^T X)}_{=1}\,\beta = (X^T X)^{-1}\,X^T Y \qquad \textit{reciprocal}$$

$$\beta = (X^T X)^{-1}\,(X^T Y)$$

and this gets us what we call the "normal equation"
Which solves for our parameters β

BAUER COLLEGE OF BUSINESS UNIVERSITY of HOUSTON

Normal Equations

Consider an **overdetermined** system *(more equations than unknowns)*

$$\sum_{j=1}^{n} X_{ij}\beta_j = y_i \quad (i = 1,2,\dots,m)$$

Of m linear equations in n unknown coefficients β_1 , $\beta_{2,} \dots \beta_m$ with m > n *(note: for a linear model as above, not all of X contains information on the data points. The* ***first column is populated with ones X_{i1} = 1*** *(intercept).* This can be written in matrix form as:

$$X\beta_j = y$$

$$X = \begin{bmatrix} X_{11} & X_{12} & \cdots & X_{1n} \\ X_{21} & X_{22} & \cdots & X_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ X_{m1} & X_{m2} & \cdots & X_{mn} \end{bmatrix} \quad \beta = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_2 \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_2 \end{bmatrix}$$

$$\hat{\beta} = \arg_{\beta} min\, S(\beta)$$

Such a system usually has no solution, so the goal is instead to find the coefficients β *which fit the equations "best" in the sense of solving the quadratic minimization problem (i.e., the least squares solution). This is called the normal equation:* $\hat{\beta} = \arg_{\beta} \min S(\beta)$

Normal Equations.ipynb
Normal Equations-R.ipynb

Note: arg min means position where S (the objective function) is minimized

Finding parameters using normal equations

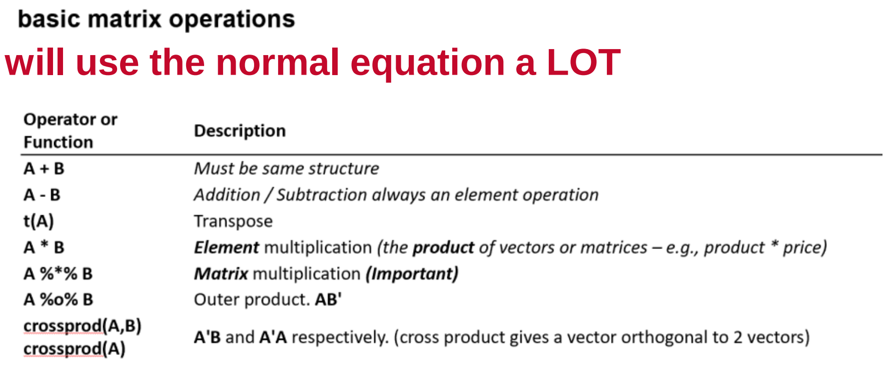
Solving this way, you need to *create a placeholder for the intercept term*

$$\beta = (X^TX)^{-1}\,(X^TY)$$

```
print(betaHat)
[1,] 3.5
[2,] 1.4
```

```
X <- cbind(1, mydata$X)
y <- mydata$Y
# we can solve this from the raw data by using a transpose
betaHat <- solve(t(X)%*%X) %*% t(X) %*%y
print(betaHat)
```

We will use the normal equation a LOT

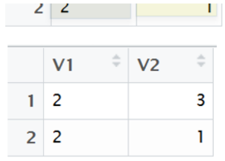


basic matrix operations

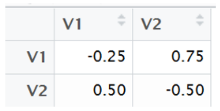
Operator or Function	Description
A + B	*Must be same structure*
A - B	*Addition / Subtraction always an element operation*
t(A)	Transpose
A * B	***Element*** multiplication *(the **product** of vectors or matrices – e.g., product * price)*
A %*% B	***Matrix*** multiplication ***(Important)***
A %o% B	Outer product. **AB'**
crossprod(A,B) **crossprod(A)**	**A'B** and **A'A** respectively. (cross product gives a vector orthogonal to 2 vectors)

K <- **solve**(A)

used to achieve matrix division, among other things



	V1	V2
1	2	3
2	2	1

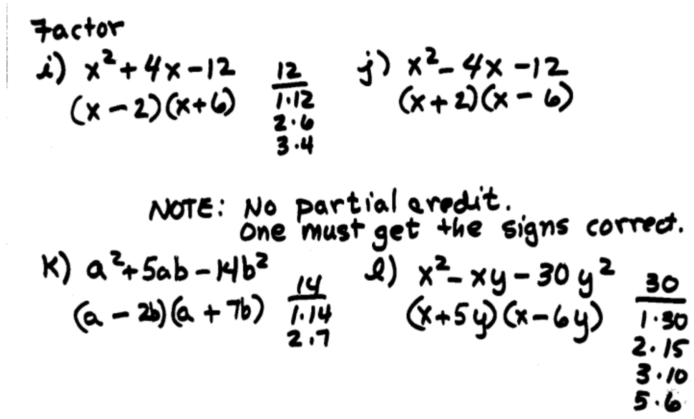


	V1	V2
V1	-0.25	0.75
V2	0.50	-0.50

*To get the **inverse $(A)^{-1}$:** **swap** the positions of a and d, put **negatives** in front of b and c, and **divide** everything by the determinant (**that's why you have solve** ☺) a matrix * it's inverse equals the identity (definition of inverse) $A^{-1}A = I$*
*Check to see if K*A is a matrix of ones (not the same as an identity matrix) L <- K*A*

Matrix Decomposition

Remember factoring in HS?



Factor

i) $x^2+4x-12$ $\quad \frac{12}{1\cdot 12}$ $2\cdot 6$ $3\cdot 4$

$(x-2)(x+6)$

j) $x^2-4x-12$

$(x+2)(x-6)$

NOTE: No partial credit. One must get the signs correct.

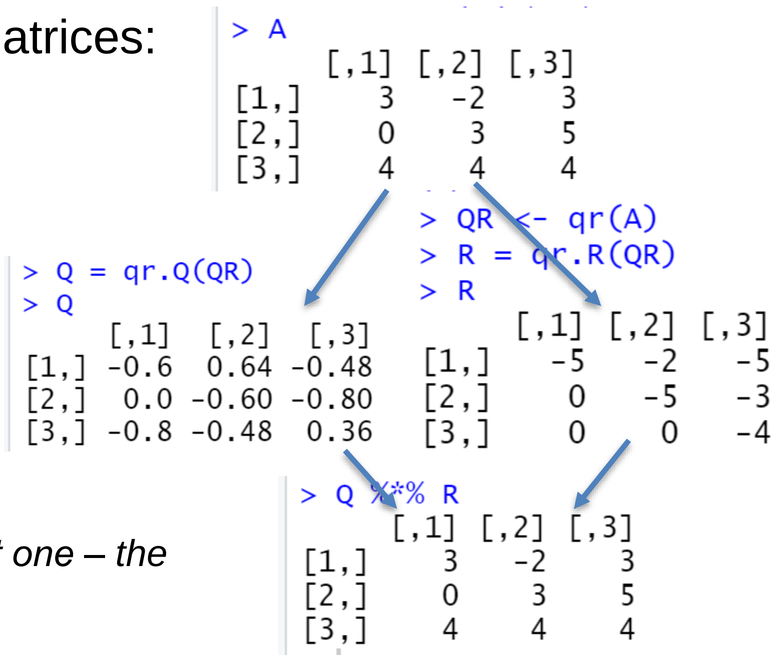
K) $a^2+5ab-14b^2$ $\quad \frac{14}{1\cdot 14}$ $2\cdot 7$

$(a-2b)(a+7b)$

l) $x^2-xy-30y^2$ $\quad \frac{30}{1\cdot 30}$ $2\cdot 15$ $3\cdot 10$ $5\cdot 6$

$(x+5y)(x-6y)$

You can do the same thing to matrices:



```
> A
     [,1] [,2] [,3]
[1,]    3   -2    3
[2,]    0    3    5
[3,]    4    4    4
```

```
> QR <- qr(A)
> R = qr.R(QR)
> R
     [,1] [,2] [,3]
[1,]   -5   -2   -5
[2,]    0   -5   -3
[3,]    0    0   -4
```

Q

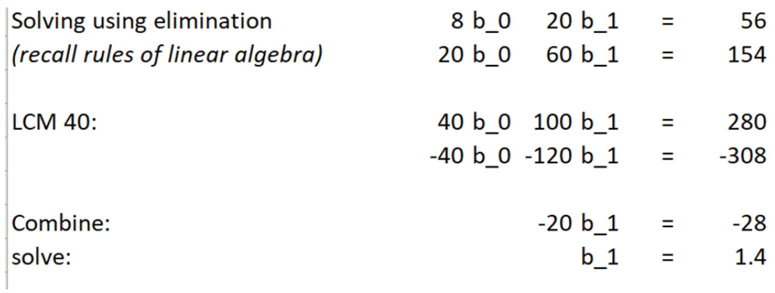
```
> Q = qr.Q(QR)
> Q
     [,1]  [,2]  [,3]
[1,] -0.6  0.64 -0.48
[2,]  0.0 -0.60 -0.80
[3,] -0.8 -0.48  0.36
```

R

```
> Q %*% R
     [,1] [,2] [,3]
[1,]    3   -2    3
[2,]    0    3    5
[3,]    4    4    4
```

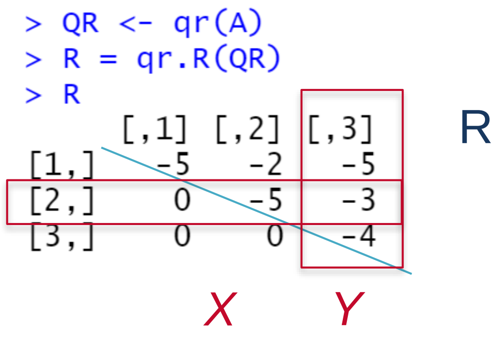
There are a number of ways to factor (or decompose) matrices, we'll take a look at one – the QR Householder decomposition

QR Intro.R



Solving using elimination	8 b_0	20 b_1	=	56
(recall rules of linear algebra)	20 b_0	60 b_1	=	154
LCM 40:	40 b_0	100 b_1	=	280
	-40 b_0	-120 b_1	=	-308
Combine:		-20 b_1	=	-28
solve:		b_1	=	1.4

Also recall how we just did solved an OLS equation system, by backsolving



```
> QR <- qr(A)
> R = qr.R(QR)
> R
     [,1] [,2] [,3]
[1,]   -5   -2   -5
[2,]    0   -5   -3
[3,]    0    0   -4
```

It turns out, that in a QR decomposition, the R matrix is diagonal, which makes **backsolving** easy

So here, β is -3/-5 = .6, then we substitute to solve the rest.

QR Decomposition

QR Decomposition *(factorization)* *[this is how lm works]*

Solve the linear system:

X

1	2	-1
0	1	2
0	0	1

B

a
b
c

Y

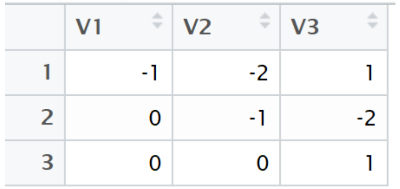
6
4
1

This is easy because we can see that c =1, which implies that b = 2, which implies that a = 3 (*backsolving*)
X is a triangular matrix and one of the nice things about triangular matrices is that it's easy to write a program to solve problems like this.

We can decompose any full rank matrix X *(full rank means the columns of the matrix are independent; i.e., no column can be written as a combination of the others)* can be "factored" into "Q" matrix and an R matrix, with R being an upper triangular matrix

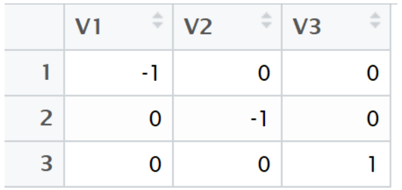
```
X <- matrix(c(1, 0, 0, 2, 1, 0, -1, 2, 1), nrow = 3)
> Y <- c(6,4,1)
> QR <- qr(X)
> Q <- qr.Q(QR)
> R <- qr.R(QR)
> betahat <- backsolve(R, crossprod(Q, Y))
➢ betahat

➢ [1,] 3
➢ [2,] 2
➢ [3,] 1
```

R



	V1	V2	V3
1	-1	-2	1
2	0	-1	-2
3	0	0	1

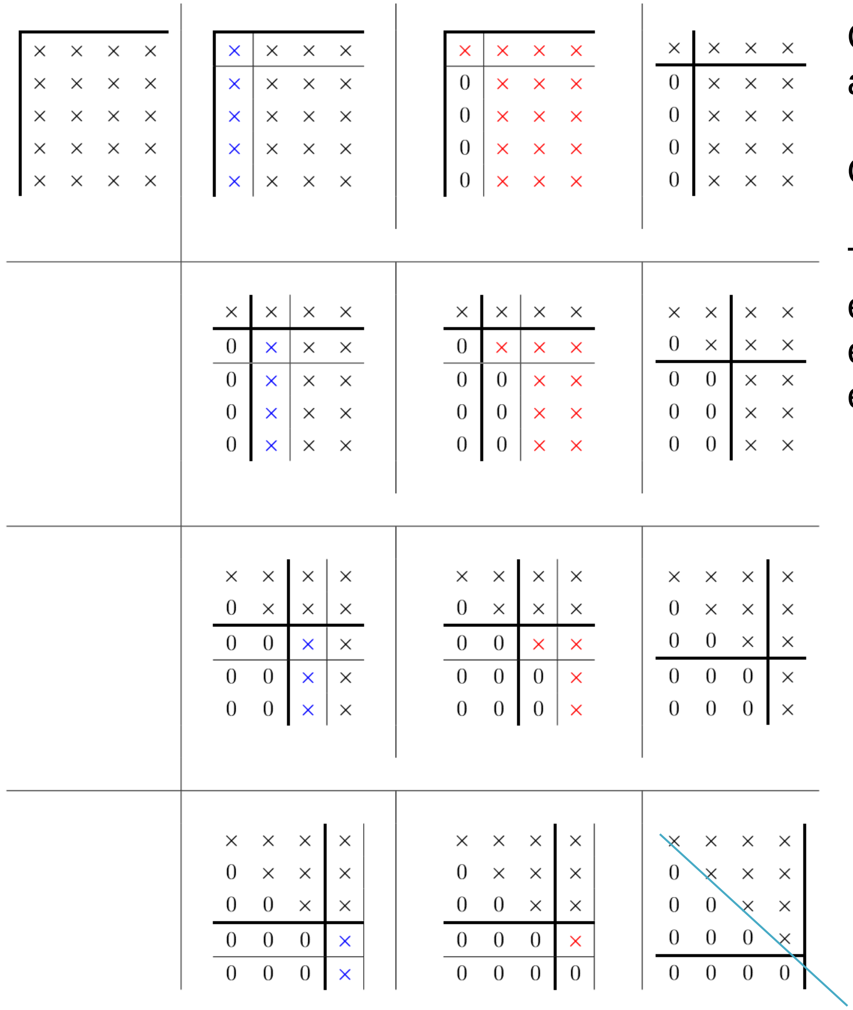
Q



	V1	V2	V3
1	-1	0	0
2	0	-1	0
3	0	0	1

Recall:	$(X^TX)\,\beta = X^T y$
Substituting the QR factors:	$(QR)^T (QR)\,\beta = (QR)^T y$
association:	$R^T (Q^TQ)\, R\,\beta = R^TQ^T y$
Eliminate identity *(remember, $Q^TQ = I$)*	$R^T R\,\beta = R^TQ^T y$
Multiplication Principle	$(R^T)^{-1} R^T R\,\beta = (R^T)^{-1} R^TQ^T y$
Elimination	$R\,\beta = Q^T y$

So now we're left with a much easier equation to solve



QR Householder factors a matrix into 2 matrices, Q and R.

QRH is used by lm – it’s very robust and extensible.

The QRH process iteratively eliminates all the elements in a column in a triangular pattern – with each column in the R matrix having ncol-1 non-zero elements left.

Remember, in matrix algebra, you can multiple rows and add them together like algebra, which we did with least squares earlier.

This process is really fast with computers.

1. Robert A. van de Geijn University of Texas at Austin

BAUER COLLEGE OF BUSINESS
UNIVERSITY of HOUSTON

Reference: Householder algorithm

```
X <- matrix(c(3, 0, 4, -2, 3, 4), nrow = 3, ncol = 2)
y <- matrix(c(3, 5, 4), nrow = 3, ncol = 1)

# Householder Function

 nr <- length(y)
 nc <- NCOL(X)

 for (j in seq_len(nc))
{
   id <- seq.int(j, nr)
   sigma <- sum(X[id,j]^2)
   s <- sqrt(sigma)
   diag_ej <- X[j,j]
   gamma <- 1.0 / (sigma + abs(s * diag_ej))
   kappa <- if (diag_ej < 0) s else -s
   X[j,j] <- X[j,j] - kappa
   if (j < nc)
     for (k in seq.int(j+1, nc))
     {
       yPrime <- sum(X[id,j] * X[id,k]) * gamma
       X[id,k] <- X[id,k] - X[id,j] * yPrime
     }
   yPrime <- sum(X[id,j] * y[id]) * gamma
   y[id] <- y[id] - X[id,j] * yPrime
   X[j,j] <- kappa
 }
RX <- as.matrix(X[1:ncol(X), ])
Ry <- as.matrix(y[1:ncol(X),])
```

(1) let z = the first column of the submatrix B, where B = $\widehat{A}_{k:m,k:n+1}$
(2) Construct a Householder transformation that zeros out z below the first entry in z:
  (a) $v = \text{sign}(z_1)||z||_2\, e + z$ %vector normal to a Householder "mirror"
  (b) $v = v/||v||_2$ % unit vector normal to a Householder "mirror"

$$(1)\ B = \widehat{A} = \begin{pmatrix} 3 & -2 & 3 \\ 0 & 3 & 5 \\ 4 & 4 & 4 \end{pmatrix},\ z = \begin{pmatrix} 3 \\ 0 \\ 4 \end{pmatrix}.$$

$$(2)\ (a)\ v = \text{sign}(z_1)||z||_2\, e + z = \text{sign}(3)\,5 \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 3 \\ 0 \\ 4 \end{pmatrix} = (+1)\,5 \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 3 \\ 0 \\ 4 \end{pmatrix} = \begin{pmatrix} 8 \\ 0 \\ 4 \end{pmatrix}.$$

norm

z_1

Vector e is combined in this operation below

For z in step 1), let sign(z_1) be +1 if $z_1 > 0$ and let sign(z_1) be -1 if $z_1 < 0$. z_1 is the first component of z. Also let e be a vector of the same dimension as z that is all zero except the first element is one. Here are details for the above algorithm:

This is for pedagogy – NOT industrial strength!

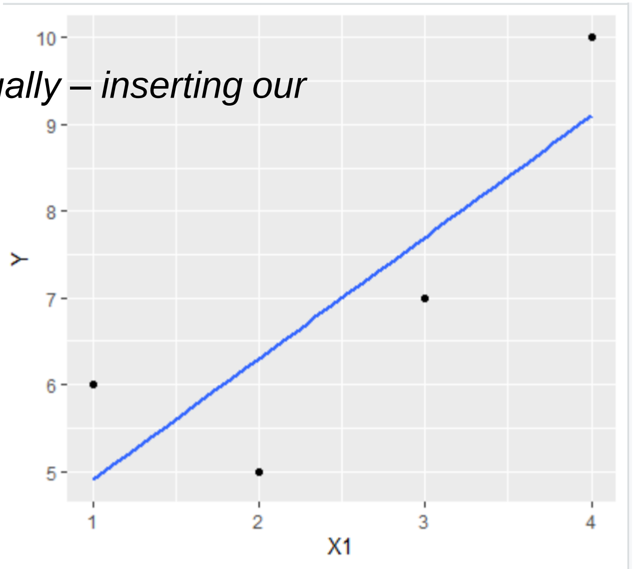
There's a file QR Wolkthrough.v 2.R, that prints out matrices for each loop so you can see the transformation effects

Householder Application

One Variable Application

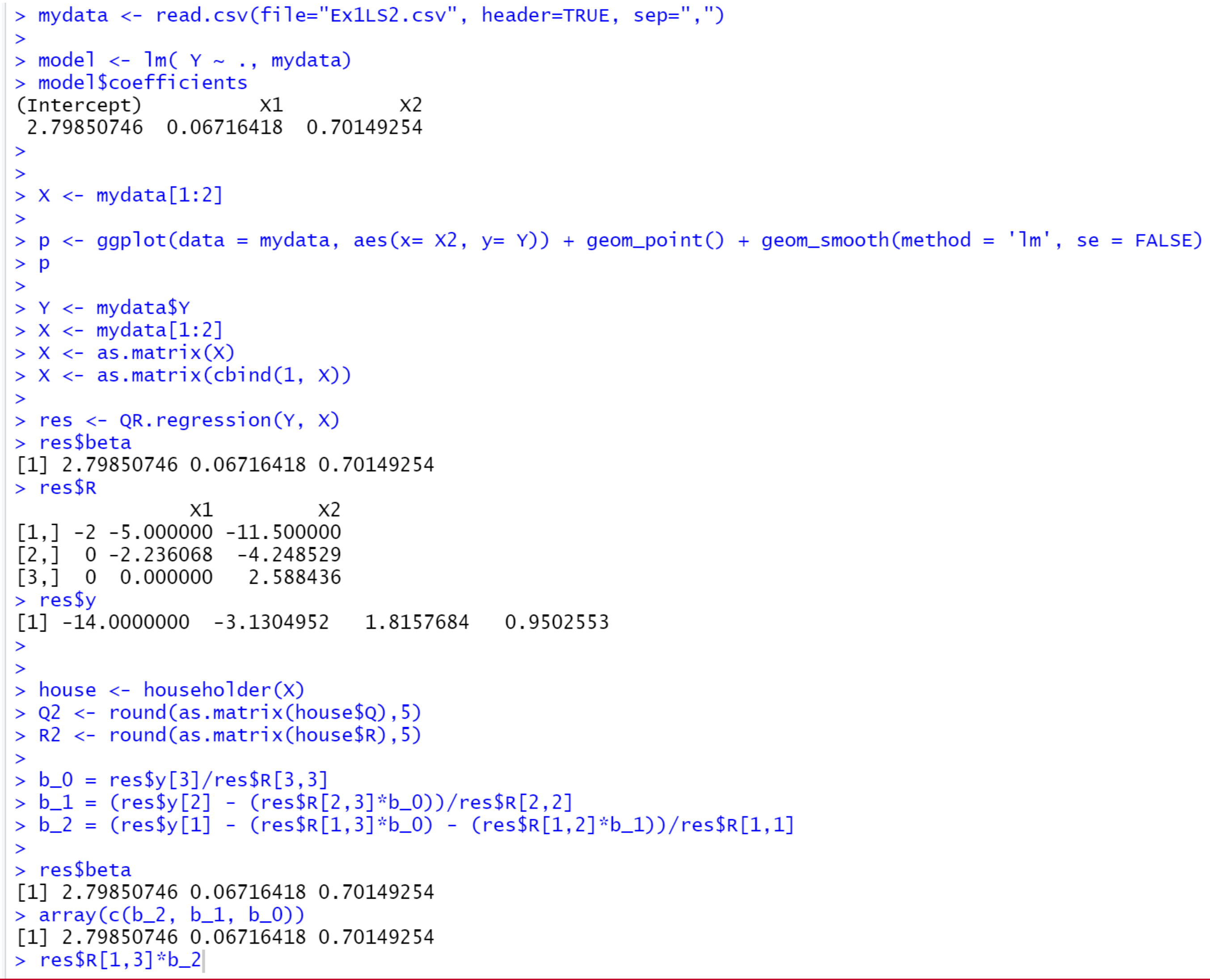
```
> mydata <- read.csv(file="Ex1LS2.csv", header=TRUE, sep=",")
> model <- lm( Y ~ X1 ,mydata)
> model$coefficients
(Intercept)          X1
        3.5         1.4
>
> p <- ggplot(data = mydata, aes(x= X1, y= Y)) + geom_point() + geom_smooth(method = 'lm', se = FALSE)
> p
>
> Y <- mydata$Y
> X <- mydata$X1
> X <- as.matrix(cbind(1, X))
>
> res <- QR.regression(Y, X)
>
> res$beta
[1] 3.5 1.4
>
> b_1 = res$y[2]/res$R[2,2]
> b_0 = (res$y[1] - res$R[1,2]*b_1)/res$R[1,1]
>
> array(c(b_0, b_1))
[1] 3.5 1.4
```

here, we're creating a model matrix manually – inserting our own 1 in the intercept column



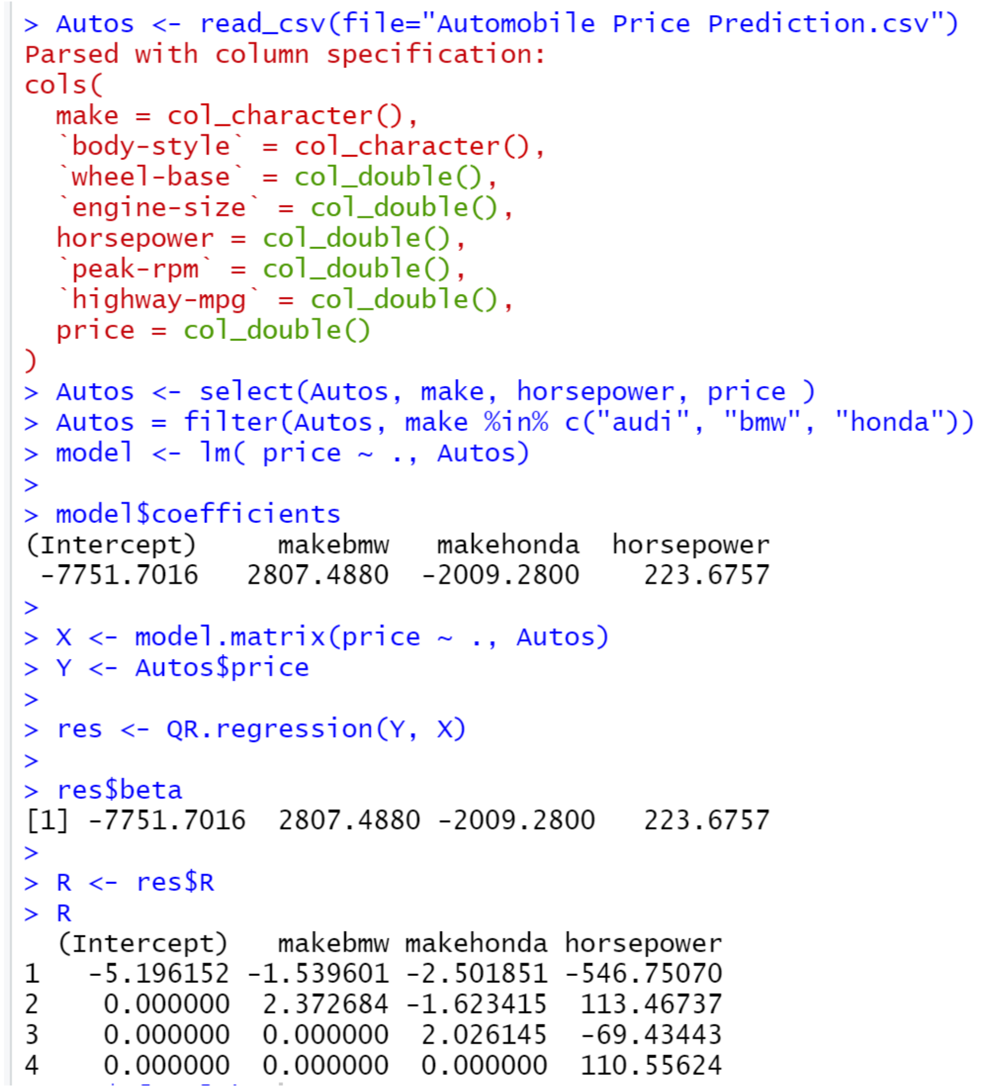
QR Application.R

Two Variable Application



```
> mydata <- read.csv(file="Ex1LS2.csv", header=TRUE, sep=",")
>
> model <- lm( Y ~ ., mydata)
> model$coefficients
(Intercept)          X1          X2 
 2.79850746  0.06716418  0.70149254 
>
>
> X <- mydata[1:2]
>
> p <- ggplot(data = mydata, aes(x= X2, y= Y)) + geom_point() + geom_smooth(method = 'lm', se = FALSE)
> p
>
> Y <- mydata$Y
> X <- mydata[1:2]
> X <- as.matrix(X)
> X <- as.matrix(cbind(1, X))
>
> res <- QR.regression(Y, X)
> res$beta
[1] 2.79850746 0.06716418 0.70149254
> res$R
              X1         X2
[1,] -2 -5.000000 -11.500000
[2,]  0 -2.236068  -4.248529
[3,]  0  0.000000   2.588436
> res$y
[1] -14.0000000  -3.1304952   1.8157684   0.9502553
>
>
> house <- householder(X)
> Q2 <- round(as.matrix(house$Q),5)
> R2 <- round(as.matrix(house$R),5)
>
> b_0 = res$y[3]/res$R[3,3]
> b_1 = (res$y[2] - (res$R[2,3]*b_0))/res$R[2,2]
> b_2 = (res$y[1] - (res$R[1,3]*b_0) - (res$R[1,2]*b_1))/res$R[1,1]
>
> res$beta
[1] 2.79850746 0.06716418 0.70149254
> array(c(b_2, b_1, b_0))
[1] 2.79850746 0.06716418 0.70149254
> res$R[1,3]*b_2
```

Auto data Application



```
> Autos <- read_csv(file="Automobile Price Prediction.csv")
Parsed with column specification:
cols(
  make = col_character(),
  `body-style` = col_character(),
  `wheel-base` = col_double(),
  `engine-size` = col_double(),
  horsepower = col_double(),
  `peak-rpm` = col_double(),
  `highway-mpg` = col_double(),
  price = col_double()
)
> Autos <- select(Autos, make, horsepower, price )
> Autos = filter(Autos, make %in% c("audi", "bmw", "honda"))
> model <- lm( price ~ ., Autos)
>
> model$coefficients
(Intercept)     makebmw   makehonda  horsepower 
 -7751.7016   2807.4880  -2009.2800    223.6757 
>
> X <- model.matrix(price ~ ., Autos)
> Y <- Autos$price
>
> res <- QR.regression(Y, X)
>
> res$beta
[1] -7751.7016  2807.4880 -2009.2800   223.6757
>
> R <- res$R
> R
  (Intercept)   makebmw makehonda horsepower
1   -5.196152 -1.539601 -2.501851 -546.75070
2    0.000000  2.372684 -1.623415  113.46737
3    0.000000  0.000000  2.026145  -69.43443
4    0.000000  0.000000  0.000000  110.55624
```

When you encounter categorical variables in matrix decomp solvers, you have to create dummy (or indicator) variables ISL pg 84.

The model.matrix function will do that for you, and create a vector for the intercept (with 1's for coefficients). Then you can run it through like any other data, but again, the algorithm will produce a coefficient for each dummy variable

BTW, you have to manually convert to dummy (or indicator variables when you do regression in Excel. And as you learned at the beginning of DA1 (I hope) Excel and Tableau make simple problems simpler and complex problems more complex. By orders of magnitude.

Other Matrix Decomposition Approaches

There are other matrix decomposition approaches that work like QR, but with a few differences:

Cholesky Decomp is faster than QR, but less stable and does not produce a covariance matrix. Common in Bayesian models due to computational requirements.

Single Value Decomposition is the most stable, but slowwwwwww. It's most often used during initial modeling when rank deficiencies are encountered:

You will encounter rank deficiency more often that you would like. It basically means that there is insufficient information to determine parameters, and it can occur for many reasons:

- ***Too little data*** to uniquely determine parameters.
- ***Too much data*** *(usually the problem is repetitive values here)*. Algorithms that use gradient and derivative methods will struggle with rep data (we're going to cover gradient descent soon, recall this concept then, and ask: "how would the derivative behave across stretches where the value is repetitive". This is a more common problem than might think – many dimensions have just a few values and millions of transactions in business systems.
- This brings up ***model selection***. You need to know how things work under the hood (which is why we're studying all this). You have a huge selection of models from which to choose and they all have different algorithms with different implementations of LA solvers, derivative based optimization *(e.g., gradient descent – next topic)*, non-derivative optimization, maximum likelihood *(last topic)*, on and on. You have to UNDERSTAND the way it works.

Gradient Descent

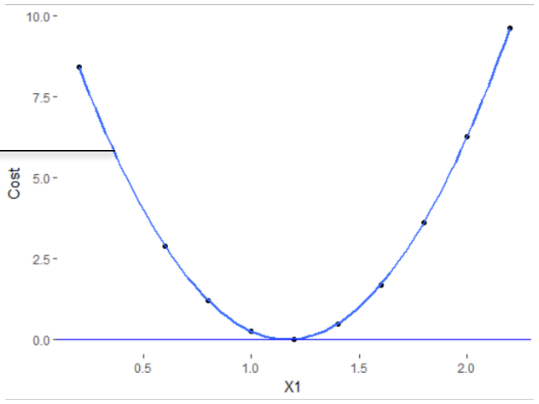
We're going to spend some time on gd - it is common throughout machine learning as an optimization component – from linear regression to neural networks.

Recall:

Analytic Solutions:

Cost Function (assumed to be convex)

$$\frac{\partial s}{\partial \beta_2} = 0 = 20\beta_1 + 60\beta_2 - 154 = 1.4$$



Holding β_1 constant

OLS becomes intractable in the real world

Cost function canonical expression $\frac{1}{2m}\sum_{i=1}^{m}(f_\theta(x_i) - y_i)^2$

θ theta is a vector [θ_1, θ_2, …. θ_n] that holds the parameters *(e.g., intercept$_0$+ slope$_1$ + slope$_2$…)*

note: $\frac{1}{2m}$ is not necessary to minimize, but is used to average the error so that models are more comparable (and to make derivative computation easier)

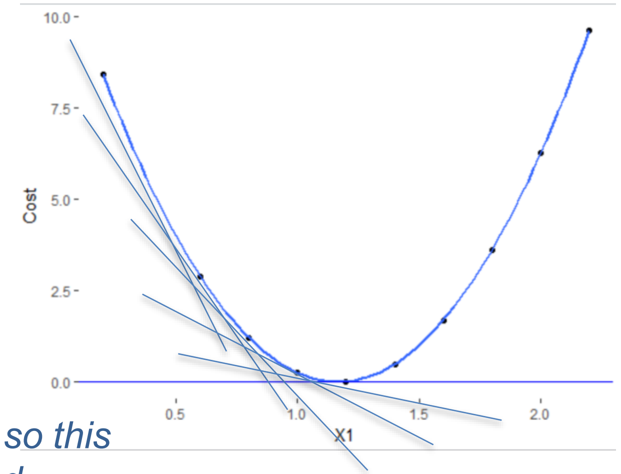
Gradient Descent v2.R

Objective function to be minimized

$$J(\theta_1, \theta_2) = \frac{1}{2}(f_\theta(xi) - yi)^2 = \frac{1}{2}(\theta_1 + \theta_2 x) - y)^2$$

Derivative

$$\frac{\partial J}{\partial \theta_2} \frac{1}{2}(\theta_1 + \theta_2 x) - y)^2 = x\,((\theta_1 + \theta_2 x) - y) = x\,(\hat{y} - y)$$



We're holding X, Y and theta1 constant here and just varying theta2, so this doesn't quite work out realistically, just to make it easier to understand

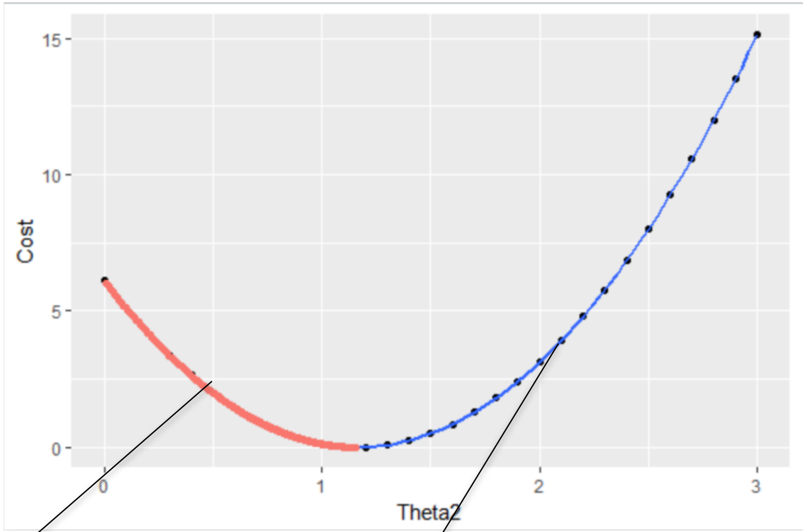
X	Y	theta1	theta2	Yhat = theta1 + (theta2*X)	SSE = 1/2(Y-Yhat)^2	*d* SSE/*d* theta2 = -(Y-Yhat)X
3	7	3.5	0.0	3.5	6.125	-10.5
3	7	3.5	0.2	4.1	4.205	-8.7
3	7	3.5	0.4	4.7	2.645	-6.9
3	7	3.5	0.6	5.3	1.445	-5.1
3	7	3.5	0.8	5.9	0.605	-3.3
3	7	3.5	1.0	6.5	0.125	-1.5
3	7	3.5	1.2	7.1	0.005	0.3
3	7	3.5	1.4	7.7	0.245	2.1
3	7	3.5	1.6	8.3	0.845	3.9
3	7	3.5	1.8	8.9	1.805	5.7
3	7	3.5	2.0	9.5	3.125	7.5
3	7	3.5	2.2	10.1	4.805	9.3

Note that slope turns positive ~ 1.2-1.4

So how do we figure out where the slope turns positive (and stop - that's our objective)?

Gradient descent attempts to minimize the error by solving a sequence of smaller minimization problems

$$\nabla J(\theta) = \frac{1}{2m}\sum_{i=1}^{m} (f_\theta(x_i) - y_i)^2$$



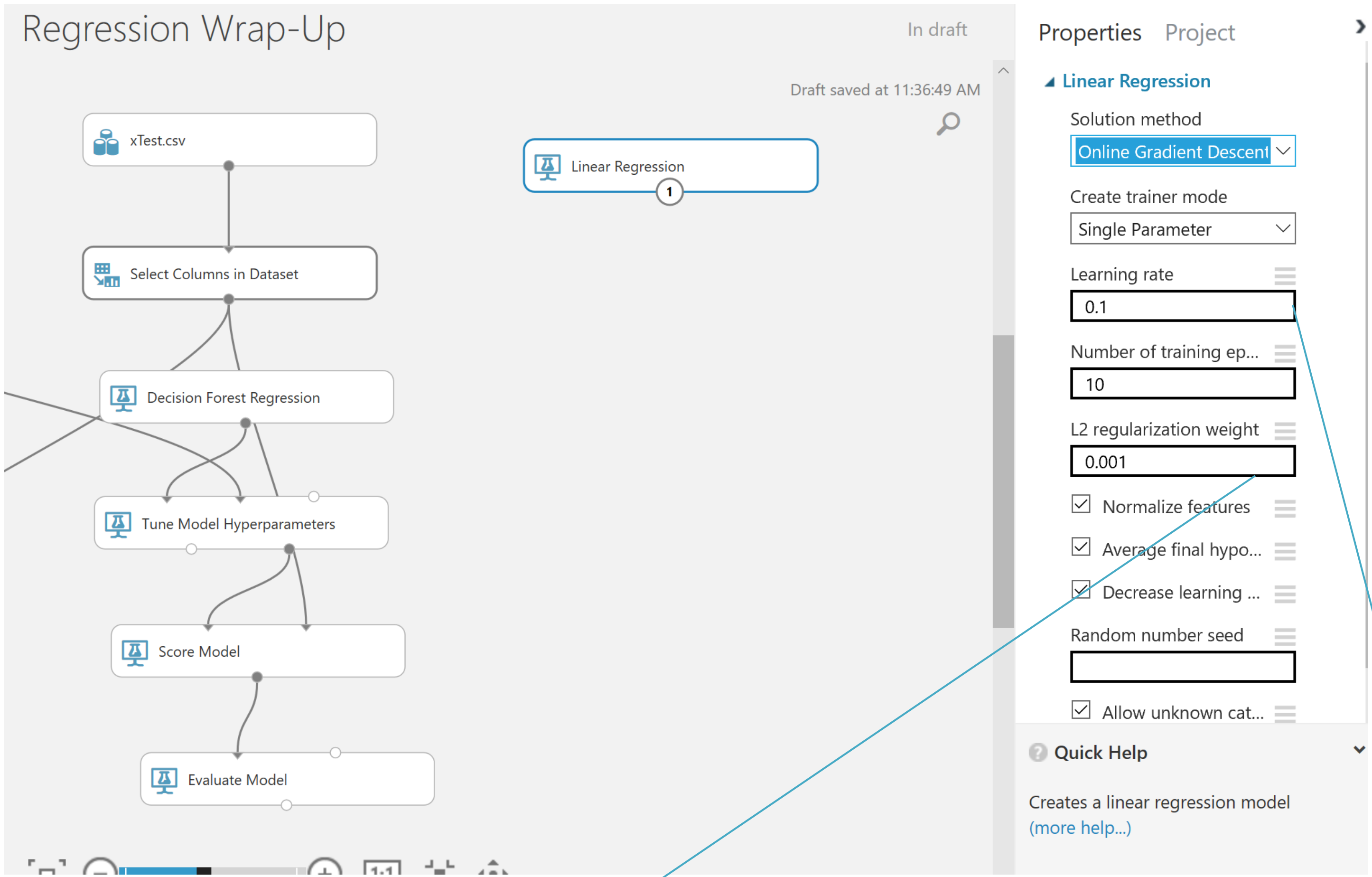
```
cost2 <- function(X, y, theta) {
  sum ((((X %*% theta)+Intercept) - y)^2 ) / (2*length(y))
}
```

Objective function to be minimized

1. Map out the cost function which measures MSE

2. Then, walk along the gradient until it turns positive.

(note, the denominator just scales and stabilizes the error term – the result will be the same, but large error values will cause many algorithms to error on infinity)


Regression Wrap-Up

In draft

Draft saved at 11:36:49 AM

xTest.csv
Select Columns in Dataset
Decision Forest Regression
Tune Model Hyperparameters
Score Model
Evaluate Model
Linear Regression

Properties Project

Linear Regression

Solution method: Online Gradient Descent

Create trainer mode: Single Parameter

Learning rate: 0.1

Number of training ep...: 10

L2 regularization weight: 0.001

☑ Normalize features

☑ Average final hypo...

☑ Decrease learning ...

Random number seed:

☑ Allow unknown cat...

Quick Help

Creates a linear regression model (more help...)

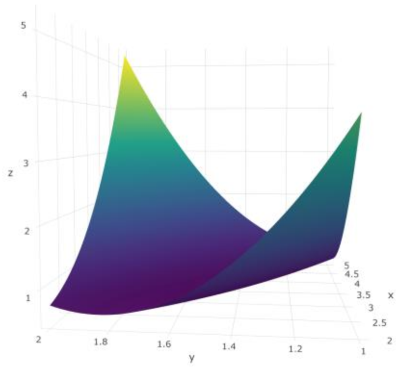
Gradient Descent is core in AI and common in advanced ML algorithms and environments. Even with lower level regression algorithms. The theory is the same – you need to understand the learning rate and threshold (here, they're using L2 regularization weight *(we cover this is in the regularization section). That said, you need to know each algorithm (they're all different implementations and each has it's own idea of how hyper-parameters are applied). How do you know what parameter value for what algorithm? RTFM!*

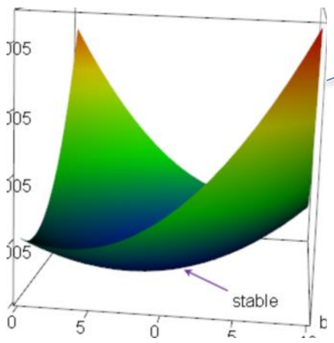
Optimization

Loss functions control model ***fit***. We assumed that they were convex *(a simplifying assumption that doesn't go too far ☺)*, and we found the optimal point using different methods:

- **Ordinary Least squares** *(which is a **derivative based** method)*. OLS often becomes ***unstable*** and ***intractable***. One way to mitigate instability is to add a ***regularization term*** *(coming up in this section)* to generalize (***see "Ridge Regression" - ISL Ch 6****).* OLS specifically, and derivative based optimization in general, will run into challenges in complex and non-linear data environments. This is one of the many benefits of regularization. ***QR Matrix Decomposition*** solves the ***tractability*** problem and produces good estimates in large, complex data. But it still relies on linear optimization.

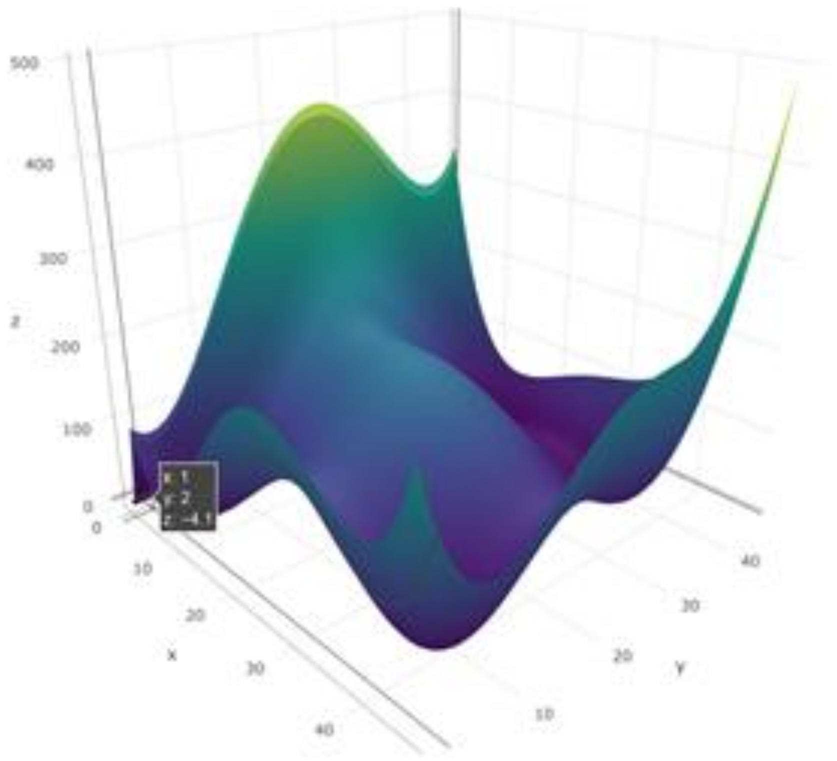
- **Gradient Descent**. GD *(also derivative based)* can be applied to a wider range of functions, but as functions become more complex, a global minimum becomes hard for GD to find.





In the real world, cost functions become complex with many “ridges”, resulting in local minima *(maxima)* and global minima. A derivative based solution will often hit a ridge and get “fooled” when it detects a change in slope, thinking it found the minima. The concept here is local and global minima, and there are many approaches to finding the global minima (Nelder-Mead, which is a type of random walk, is common), including swarm algorithms.

The point here is to build intuition about how your models (and the underlying algorithms) will attempt to estimate parameter values using derivative baed methods.



Maximum Likelihood Estimation *(MLE)*

By FAR the most common parameter estimation method utilized in analytics

Suppose there is a sample x_1, x_2, …, x_n of n iid observations, coming from a distribution with an unknown probability density function. It is however surmised that the function f(.) belongs to a certain family of distributions { f(·| θ), θ ∈ Θ } (where ***θ is a vector of parameters*** for this family), called the parametric model, so that f0 = f(·| θ0). The value θ0 is unknown and is referred to as the true value of the parameter vector. It is desirable to find an estimator which would be as close to the true value θ0 as possible. Either or both the observed variables xi and the parameter θ can be vectors. To use the method of maximum likelihood, one ***first specifies the joint density function for all observations***. For an independent and identically distributed sample, this joint density function is

$$f(x_1,x_2,\ldots,x_n \mid \theta) = f(x_1 \mid \theta) \times f(x_2 \mid \theta) \times \cdots \times f(x_n \mid \theta).$$

Now we look at this function from a different perspective by considering the observed ***values x_1, x_2, …, x_n to be fixed "parameters" of this function, whereas θ will be the function's variable and allowed to vary freely; this same function will be called the likelihood:***

$$\mathcal{L}(\theta\,;\,x_1,\ldots,x_n) = f(x_1,x_2,\ldots,x_n \mid \theta) = \prod_{i=1}^{n} f(x_i \mid \theta).$$

Note that In practice it is often more convenient when working with the natural logarithm of the likelihood function, called the log-likelihood *(please refer to your log transforms cheatsheet)*

$$\ln \mathcal{L}(\theta\,;\,x_1,\ldots,x_n) = \sum_{i=1}^{n} \ln f(x_i \mid \theta),$$

MLE.ipynb
MLE-R.ipynb

Maximum Likelihood Estimation *(MLE)*

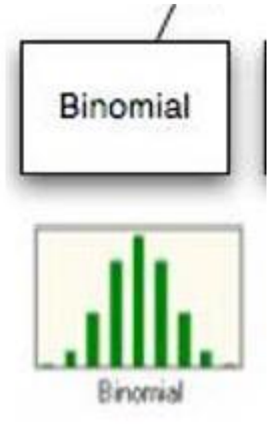
Likelihood is somewhat hard to get your head around, but it's an essential tool in modleling and parameter estimation. Here's some additional thoughts to help:

$\mathcal{L} \neq \mathbb{P}$	likelihood is not probability and it doesn't sum to 1
$\mathcal{L} \propto \mathbb{P}$	likelihood is proportional to probability *(we use the probability function (or log) to determine the likelihood)*.
$\mathcal{L}\,(\mathrm{H} \mid \mathrm{D}) = \mathrm{K} * \mathbb{P}\,(\mathrm{D} \mid \mathrm{H})$ $\mathcal{L}\,(\theta \mid \mathrm{D}) = f * \mathbb{P}\,(\mathrm{D} \mid \theta)$	The likelihood of a hypothesis *(or parameter θ)* given some data is proportional to the probability of obtaining data D given the hypothesis H is true *(multiplied by a constant or proportional function)*

The critical difference between probability and likelihood is the interpretation of what is fixed and what can vary. In the case of traditional probability $\mathbb{P}\,(\mathrm{D} \mid \mathrm{H})$ probability, the hypothesis is fixed and the data are free to vary. In the case of $\mathcal{L}\,(\mathrm{H} \mid \mathrm{D})$, the data are fixed and the hypothesis is free to vary

Likelihoods are not useful in isolation – they are used to compare parameters

Quick Basic Statistics Review:



Binomial Discrete Distribution

$$\frac{n!}{h!(n-h)!}\,p^h(1-p)^{n-h}$$

```
> p <- .5
> h <- 5 # sum(y)
> h
[1] 5
> x <- factorial(n)/(factorial(h)*factorial(n-h))
> x*(p)^h*(1-p)^(n-h)
[1] 0.2460938
```

Keep this in your head – we're coming back. Understand what it means, how it works…

$$\frac{10!}{5!(10-5)!}\,.5^5(1-.5)^{10-5} =$$

.24

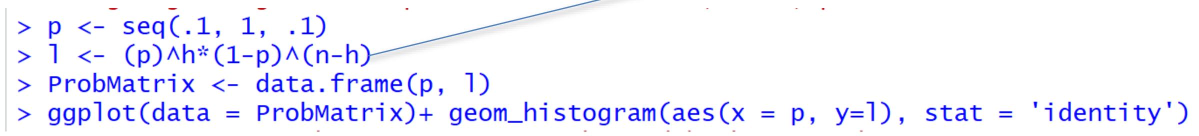
So this means that the probability of observing 5 1's in a set of 10 events is .24… we will plot this probability density…

One last thing about the binomial for now *(we'll come back to this later)*.

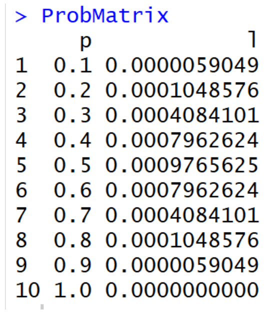
In working with the binomial pdf

$$\frac{n!}{\mathrm{h!(n-h)!}} p^h(1-p)^{\mathrm{n-h}}$$

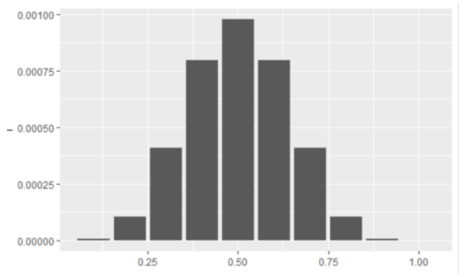
we can drop the ***constant*** $\frac{n!}{\mathrm{h!(n-h)!}}$ *(wrt p)* and just work with $p^h(1-p)^{\mathrm{n-h}}$ and still get to the same place *(as long as we're searching for p – the formula would be different if we're searching for h or n)*.

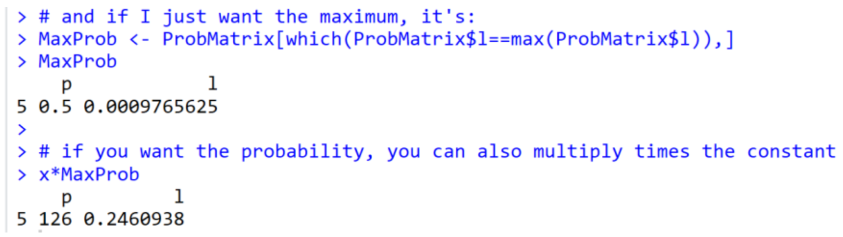


```
> p <- seq(.1, 1, .1)
> l <- (p)^h*(1-p)^(n-h)
> ProbMatrix <- data.frame(p, l)
> ggplot(data = ProbMatrix)+ geom_histogram(aes(x = p, y=l), stat = 'identity')
```



```
> ProbMatrix
     p            l
1  0.1 0.0000059049
2  0.2 0.0001048576
3  0.3 0.0004084101
4  0.4 0.0007962624
5  0.5 0.0009765625
6  0.6 0.0007962624
7  0.7 0.0004084101
8  0.8 0.0001048576
9  0.9 0.0000059049
10 1.0 0.0000000000
```

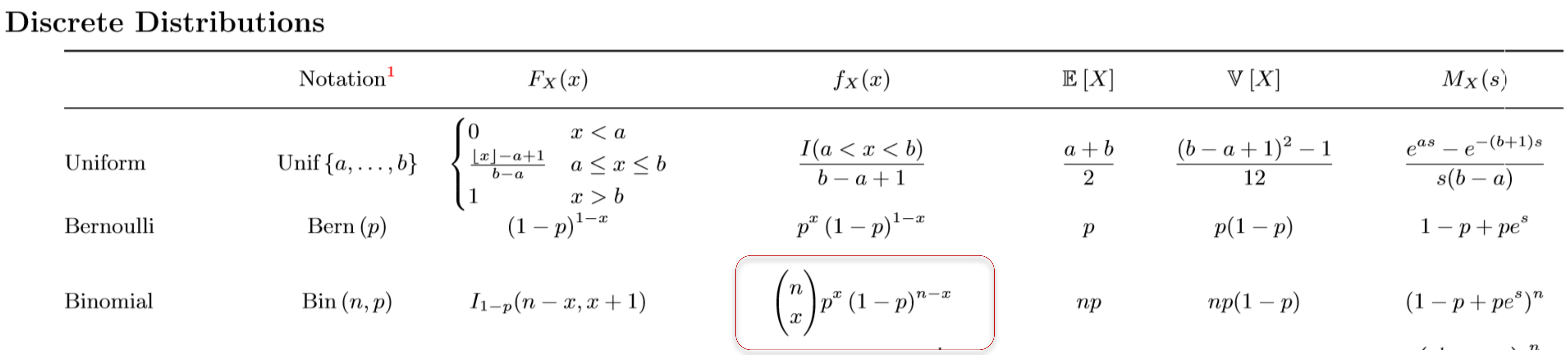




```
> # and if I just want the maximum, it's:
> MaxProb <- ProbMatrix[which(ProbMatrix$l==max(ProbMatrix$l)),]
> MaxProb
    p            l
5 0.5 0.0009765625
>
> # if you want the probability, you can also multiply times the constant
> x*MaxProb
    p         l
5 126 0.2460938
```

MLE is a technique that is applied to a range of modeling processes. Here, we are going to use MLE to estimate ***distribution parameters***, ***but the real power of MLE comes in estimating model parameters*** *(e.g., θ_1, θ_2, θ_3,… θ_n in a linear model – which we will study soon)* .

For this initial exercise, refer to your “Probability and Statistics” Cheatsheet on Blackboard and jot down the formula for a Binomial PDF.



Discrete Distributions

	Notation[1]	$F_X(x)$	$f_X(x)$	$\mathbb{E}[X]$	$\mathbb{V}[X]$	$M_X(s)$
Uniform	$\text{Unif}\{a,\ldots,b\}$	$\begin{cases} 0 & x<a \\ \frac{\lfloor x\rfloor - a+1}{b-a} & a\le x\le b \\ 1 & x>b \end{cases}$	$\frac{I(a<x<b)}{b-a+1}$	$\frac{a+b}{2}$	$\frac{(b-a+1)^2-1}{12}$	$\frac{e^{as}-e^{-(b+1)s}}{s(b-a)}$
Bernoulli	$\text{Bern}(p)$	$(1-p)^{1-x}$	$p^x(1-p)^{1-x}$	p	$p(1-p)$	$1-p+pe^s$
Binomial	$\text{Bin}(n,p)$	$I_{1-p}(n-x,x+1)$	$\boxed{\binom{n}{x}p^x(1-p)^{n-x}}$	np	$np(1-p)$	$(1-p+pe^s)^n$

So we have a function that describes our model *(in this case, just a probability distribution, but it could be another model function – e.g., linear equation)*. And now, we'll call this the likelihood function. And we translated those that to code, along with the log of the function which we call the log likelihood.

$$p^h(1-p)^{n-h}$$

N = 10
K = 4

$$\mathcal{L}(\theta\,;x_1,\ldots,x_n) = f(x_1,x_2,\ldots,x_n \mid \theta) = \prod_{i=1}^{n} f(x_i \mid \theta).$$

# The likelihood function
L = function(p,k,n) p^k*(1-p)^(n-k)

And we're going to simplify this again by taking the log of the function, and we'll call this the log likelihood *(see your log cheatsheets)*, which takes the exponents out *(not such a big deal here, but it gets complex later)*

$$\ln \mathcal{L}(\theta\,;x_1,\ldots,x_n) = \sum_{i=1}^{n} \ln f(x_i \mid \theta),$$

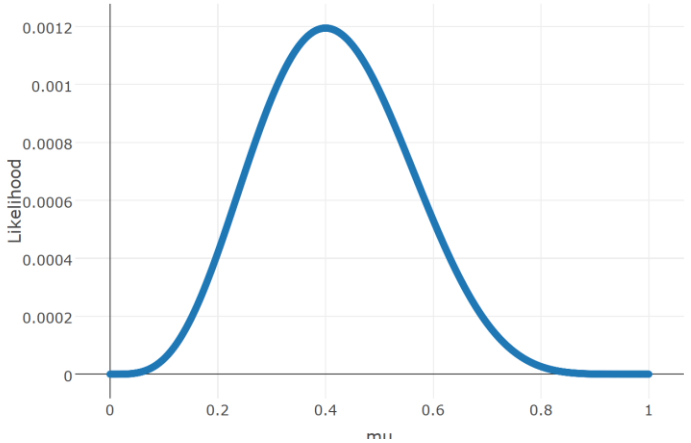
$$h\log(p) + (n-h) * log(1-p)$$

# The log-likelihood function
l = function(p,k,n) k*log(p) + (n-k)*log(1-p)

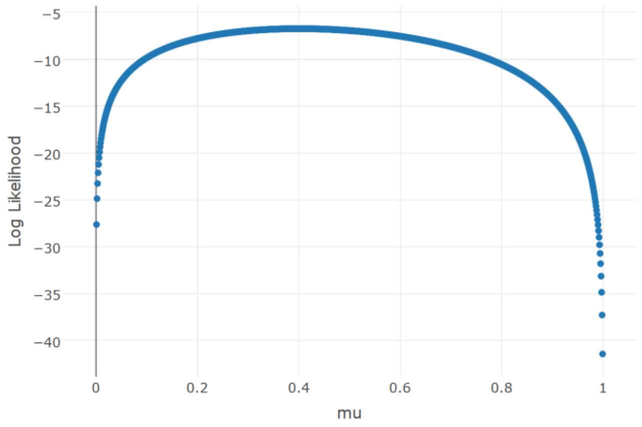
mu = seq(0,1,0.001)

And we'll plot this for a range of probabilities *(like before)* and so now we're looking for the mu of the distribution – or the ***parameter*** of the distribution density function.

```
# Plotting the Likelihood function
bNomLikePlot <- dLogL<-data.frame(mu, L(mu, K, N))
p <- plot_ly (x = ~ bNomLikePlot$mu, y = ~
bNomLikePlot$L.mu..K..N., type = 'scatter') %>% layout(xaxis =
list(title = "mu"), yaxis = list(title = "Likelihood"))
p
```

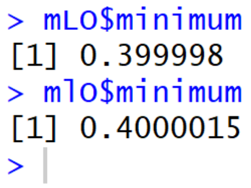


```
# Plotting the Log Likelihood function
bNomLogLikePlot <- dLogL<-data.frame(mu, l(mu, K, N))
p <- plot_ly (x = ~ bNomLogLikePlot$mu, y = ~
bNomLogLikePlot$l.mu..K..N., type = 'scatter') %>% layout(xaxis
= list(title = "mu"), yaxis = list(title = "Log Likelihood"))
p
```



We're not done yet. We're looking for the parameter that is MOST likely. So we need to find the optimal point. We can't just select the maximum from a table – this is continuous now. We could use a derivative, but we'll take the easy approach and use optimize *(which is a simple version of optum (next) for one variable.*

```
# The optimization functions in R finds the minimum, not the
maximum. We
# therefor must create new functions that return the negavive
likelihood and
# log-likelihood, and then minimize these:
# Minus likelihood:

mL = function(p,k,n) -p^k*(1-p)^(n-k)

# minus log-likelihood:

ml = function(p,k,n) -(k*log(p) + (n-k)*log(1-p))

# Using 'optimize'
#  simpler version of optim for one parameter.
#  we will use optim in the next exercise

mLO <- optimize(mL, interval = c(0,1), k=K, n=N)
mlO <- optimize(ml, interval = c(0,1), k=K, n=N)

mLO$minimum
mlO$minimum
```



```
> mLO$minimum
[1] 0.399998
> mlO$minimum
[1] 0.4000015
> 
```

Showing how the likelihood and the log likelihood find the same value here

Logarithm Properties

$\log_b b = 1 \qquad \log_b 1 = 0$

$\log_b b^x = x \qquad b^{\log_b x} = x$

$\log_b \left(x^r\right) = r \log_b x$

$\log_b (xy) = \log_b x + \log_b y$

$\log_b \left(\frac{x}{y}\right) = \log_b x - \log_b y$

See Algebra_Cheatsheet in Blackboard Resources.zip

xponent Properties

$^n a^m = a^{n+m} \qquad \frac{a^n}{a^m} = a^{n-m} = \frac{1}{a^{m-n}}$

$a^n)^m = a^{nm} \qquad a^0 = 1, \quad a \neq 0$

$ab)^n = a^n b^n \qquad \left(\frac{a}{b}\right)^n = \frac{a^n}{b^n}$

$^{-n} = \frac{1}{a^n} \qquad \frac{1}{a^{-n}} = a^n$

$\frac{a}{b}\Big)^{-n} = \left(\frac{b}{a}\right)^n = \frac{b^n}{a^n} \qquad a^{\frac{n}{m}} = \left(a^{\frac{1}{m}}\right)^n = \left(a^n\right)^{\frac{1}{m}}$

Properties of Radicals

$\sqrt[n]{a} = a^{\frac{1}{n}} \qquad \sqrt[n]{ab} = \sqrt[n]{a}\sqrt[n]{b}$

$\sqrt[m]{\sqrt[n]{a}} = \sqrt[nm]{a} \qquad \sqrt[n]{\frac{a}{b}} = \frac{\sqrt[n]{a}}{\sqrt[n]{b}}$

$\sqrt[n]{a^n} = a$, if n is odd

$\sqrt[n]{a^n} = |a|$, if n is even

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$\downarrow$$

$$\log[2\pi\sigma^{-½}] - \frac{(x-\mu)^2}{2\sigma^2}$$

$$\downarrow$$

$$-½ \log[2\pi\sigma] - \frac{(x-\mu)^2}{2\sigma^2}$$

$$\downarrow$$

$$-½ [\log(2\pi) - \log(\sigma)] - \frac{(x-\mu)^2}{2\sigma^2}$$

$$-\frac{1}{2}\;[\log(2\pi) - \log(\sigma)]\;-\;\frac{(x-\mu)^2}{2\sigma^2}$$

Transform to handle arrays

-(n/2)*log(2*pi) - (n/2)*log(sigma^2)

```
for (i in 1:n) {
    tmp = tmp + (data[i]-mu)**2
  }
  NLL = NLL + -(1/(2*(sigma^2)))*tmp
```

This compares the estimated mu to the actual data *(so the farther away the estimate, the larger the likelihood)*

And remember, the optimization is going to run a sequence of possible mu's through until it finds the minimum

Exercise 2: normal distribution 2 parameters. So now we'll keep using pdf's, but we'll find 2 parameters this time

Generate normal distribution with mean of 10 and standard deviation of 2

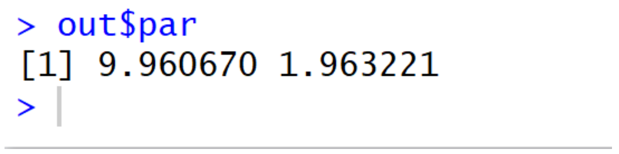
```
x <- rnorm(1000, 10, 2)
df2 <- density(x)
x2 <- data.frame(x = df2$x, y = df2$y)
ggplot(x2, aes(x,y)) + geom_line()

capMu <- matrix( nrow = 0, ncol = 3)

NLL <- function(theta,data) {
  mu = theta[1]
  sigma = theta[2]
  n = length(data)
  t <- n
  NLL = -(n/2)*log(2*pi) - (n/2)*log(sigma^2)
  tmp = 0
  for (i in 1:n) {
    tmp = tmp + (data[i]-mu)**2
  }
  NLL = NLL + -(1/(2*(sigma^2)))*tmp
  capMu <<- rbind(capMu, c(NLL, mu, sigma))
  -NLL
}
out = optim(par=c(9,1), fn=NLL, data=x)
out$par
```

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Gaussian density function from your Cheatsheet

(see why we use a log ☺)

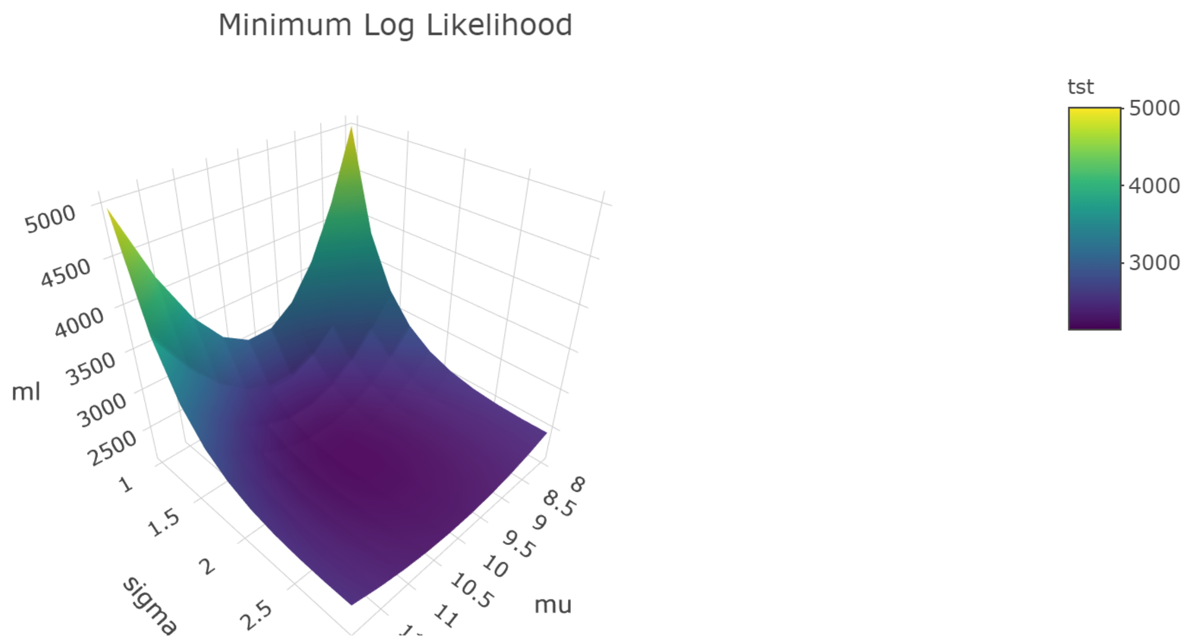
$$\log P(x) = \boxed{\log\left[\frac{1}{\sqrt{2\pi}\sigma}\right] \boxed{-\frac{(x-\mu)^2}{2\sigma^2}}}$$



```
> out$par
[1] 9.960670 1.963221
>
```

Using optim now – multiple parameters

```
mu1 = seq(8, 12, length.out = 10)
sigma1 = seq(1, 3, length.out = 10)

minLL = matrix( nrow = 0, ncol = 3)
for (i in 1:length(mu1)) {
  mu = mu1[i]
  for(j in 1:length(sigma1)){
    sigma = sigma1[j]
    ml = NLL(c(mu, sigma), x)
    minLL <<- rbind(minLL, c(mu, sigma, ml))
    }
  }
dfSurface <- data.frame(minLL)


tst = matrix(dfSurface$X3, nrow = 10)
x <- mu1
y <- sigma1
plot_ly() %>% add_surface(x = ~x, y = ~y, z = ~tst) %>%
layout(
  title = "Minimum Log Likelihood",
  scene = list(
    xaxis = list(title = "mu"),
    yaxis = list(title = "sigma"),
    zaxis = list(title = "ml")
  ))
```



So now we've used maximum likelihood to find the parameters of a normal distribution (μ,σ).

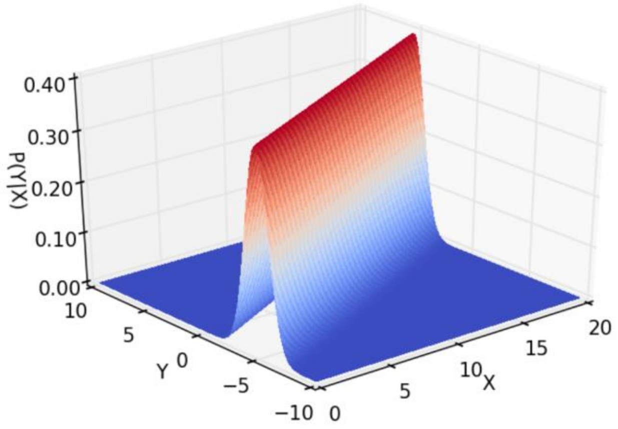
$$l(\beta,\sigma^2;y,X) = -\frac{N}{2}\ln(2\pi) - \frac{N}{2}\ln(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i - x_i\beta)^2$$

MLE Applied to Linear Regression

```
# linear likelihood function
linear.lik <- function(theta, y, X){
  n      <- nrow(X)
  k      <- ncol(X)
  beta   <- theta[1:k]
  sigma2 <- theta[k+1]^2
  e      <- y - X%*%beta
  logl   <- -.5*n*log(2*pi)-.5*n*log(sigma2) - ( (t(e) %*% e)/ (2*sigma2) )
  return(-logl)
}
```

error

cost function
(remember, errors are distributed normally)



```
# create some linear data

data.x <- rnorm(n = 100, mean = 10, sd = 2)

a.true <- 3
b.true <- 8

true.y <- data.x * a.true + b.true

err.sd.true <- 1  # Set noise sd
noise <- rnorm(n = 100, mean = 0, sd = 2)  # Generate noise

data.y <- true.y + noise  # Add noise to true (latent) responses

data <- data.frame(cbind(x = data.x, y = data.y))

lmData <- data
mod <- lm(data = lmData, y ~ x)
summary(mod)

linear.MLE <- optim(fn=linear.lik, par=c(1,1,1), lower =
                upper = c(Inf, Inf, Inf), hessian=TRUE,
                y=data$y, X=cbind(1, data$x), method = "



# Compare lm with MLE
linear.MLE$par[1]
mod$coefficients[1]
linear.MLE$par[2]
mod$coefficients[2]
```

```
Call:
lm(formula = y ~ x, data = lmData)

Residuals:
    Min      1Q  Median      3Q     Max 
-5.5237 -1.1794 -0.0114  1.2088  5.1827 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)   8.5672     1.0099   8.483 2.34e-13 ***
x             2.9112     0.0981  29.677  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.985 on 98 degrees of freedom
Multiple R-squared:  0.8999,	Adjusted R-squared:  0.8988 
F-statistic: 880.7 on 1 and 98 DF,  p-value: < 2.2e-16
```

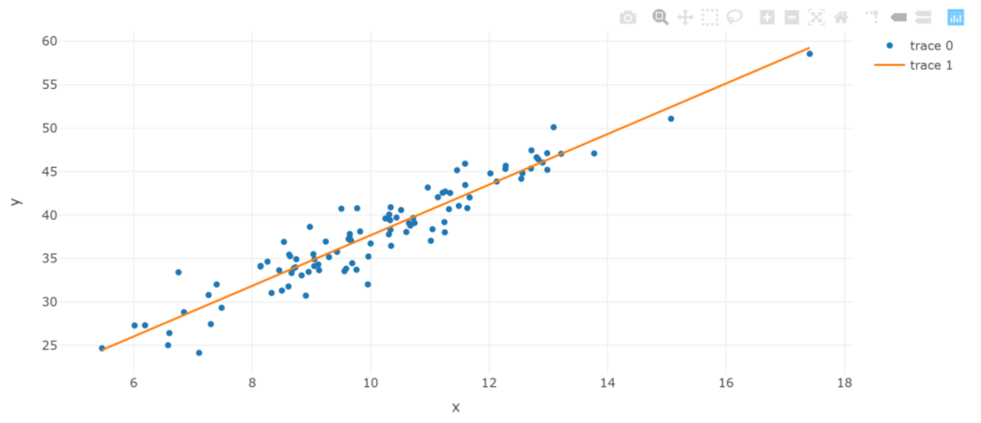
8.56716943685062

(Intercept): 8.56718494613546

2.91118004324675

x: 2.9111792281826

```
plot_ly(data = data, x = ~ x) %>%
add_markers(y = ~y) %>%
add_lines(x = ~x, y = fitted(mod))
```



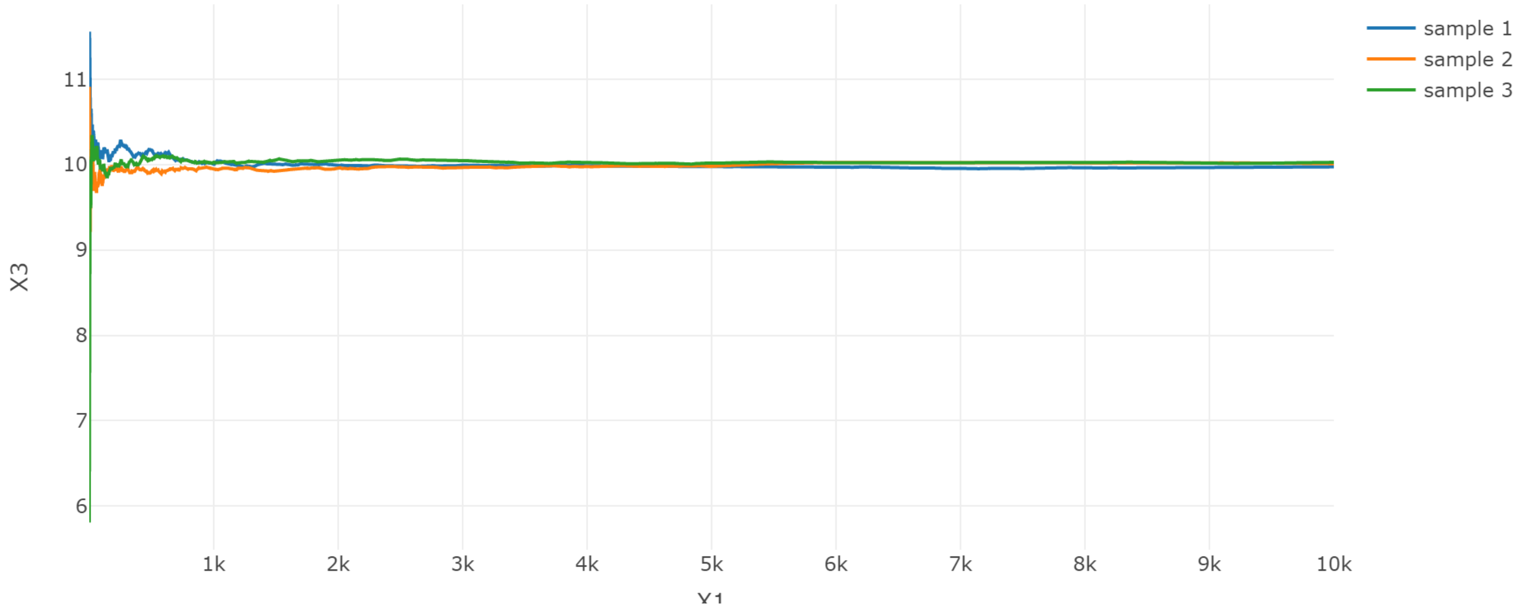
Sampling

- Why do we sample? We want to make probabilistic statements about predictions and projections
  - You need to define distribution parameters to make statements about probability
  - Predictions and projections are theoretical – parameters are not known Therefore, we create a projection based on inputs and experience, and sample the posterior *(i.e., sample the imaginary)*

Bayesian modeling was not possible before Sampling could be applied to large, complex populations – and we needed BIG machines to do it - so Bayesian methodologies were not common until ~ 2010 when clusters became available.

```
n = 10000
genSample = function()
{
track = matrix( nrow = n+1, ncol = 3)
for (i in 1:n) {
  track[i, 1] = i
  track[i, 2] = rnorm(1, 10, 2)
  track[i, 3] = mean(track[,2], na.rm =T)
}
  return(track)
}
```

Note how, as a sample size accumulates, the estimated mean gets closer to the “true” mean and more stable (central limit theorem).

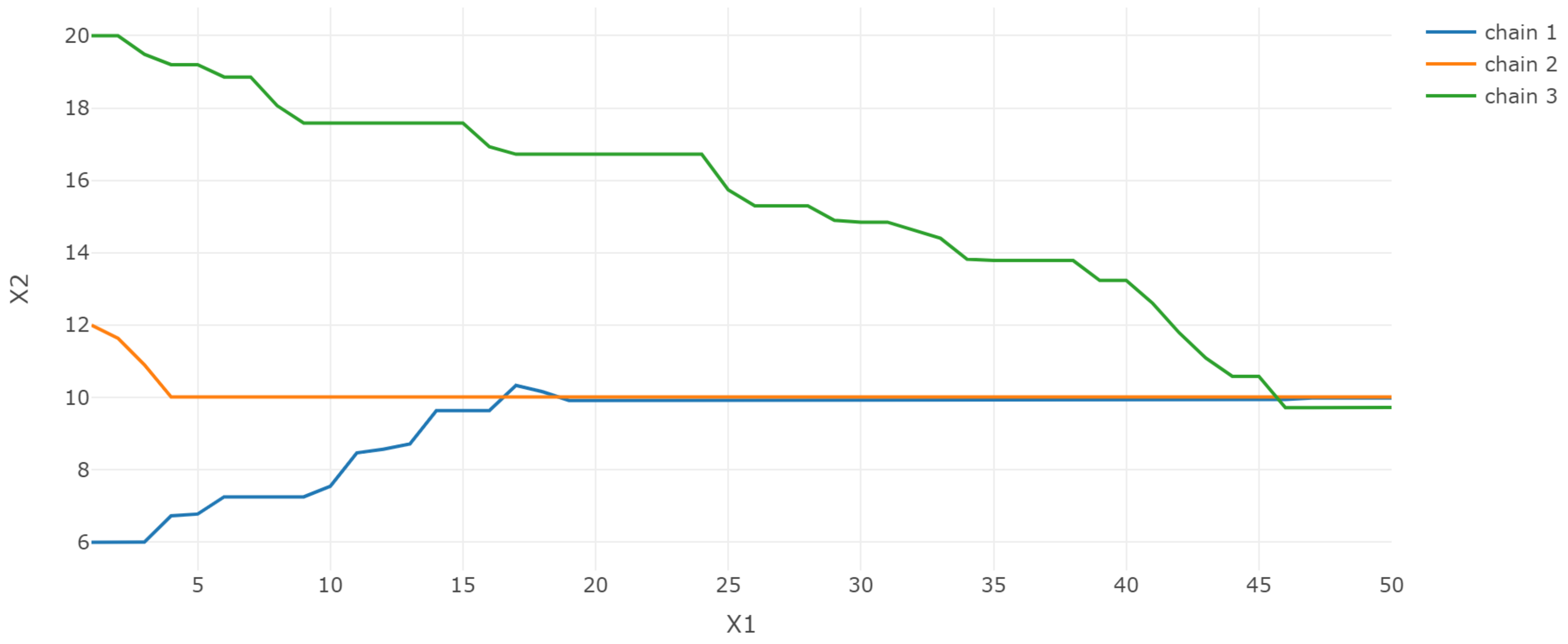


Here, we create a normally distributed population and use likelihood to test a randomly selected value. We then take another random value and compare NLL values. If the value is lower, we take the proposed value. Then we do it again…and again, each time we get closer and closer. It's a random walk. A Markov chain Monte Carlo (MCMC) method…

```
data = rnorm(1000, mean = 10, sd = 2)

# NLL from maximum likelihood

NLL <- function(theta,data) {
  mu = theta[1]
  sigma = theta[2]
  n = length(data)
  NLL = -(n/2)*log(2*pi) - (n/2)*log(sigma**2)
  tmp = 0
  for (i in 1:n) {
    tmp = tmp + (data[i]-mu)**2
  }
  NLL = NLL + -(1/(2*(sigma**2)))*tmp
  -NLL
}

proposalfunction <- function(param, t){
 return(runif(1,(param-t), (param+t)))
}
```

```
run_metropolis_MCMC <- function(startvalue, iterations, t){
  chain = array(dim = c(iterations+1,2))
  chain[1,2] = startvalue
  sigma = 2 # isolate the mean for demo purposes
  for (i in 1:iterations){
      chain[i, 1] = i
      proposal = proposalfunction(chain[i,2], t)
    if (NLL(c(chain[i,2],sigma), data) < (NLL(c(proposal,sigma), data))){
      chain[i+1,2] = chain[i,2]
    }else{
      chain[i+1,2] = proposal
    }
  }
  return(chain)
}

iterations = 100
chain = run_metropolis_MCMC(9, iterations, 1)
```

… we do this 3 times, starting with a different random number each time. They all converge near the "true" mean. We'll use a similar MCMC method when we get to Bayesian Analysis.


chain 1
chain 2
chain 3
X2
X1