Homomorphic Encryption in R

Alexander C. Mueller PhD
CEO and Founder of Capnion

May 9, 2019

Who is the speaker?

30 second resume:

- an ancient metro Saint Louis townie
- grew up in University City
- University City High School
- B.A. Washington University (econ and math)
- Ph.D. University of Michigan (math)
- private sector data science
- founded data privacy company Capnion

The Counter-Earth

Orbiting exactly opposite the Earth you are familiar with is another Earth, alike in many ways but also different...

Agenda

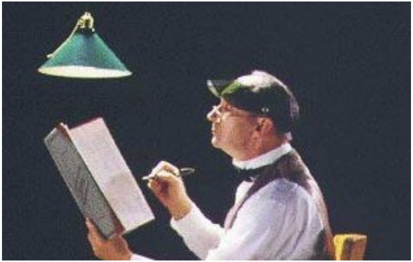
What and Why?

- What is homomorphic encryption?
- What's out there?
- Who cares? Use cases.

Examples and Application in R

- Basic keygen, encryption and decryption
- Easy in R, harder when R isn't R
- Wrinkles and how to flatten them
- An example classification problem
- Private models, private data

Back to Basics



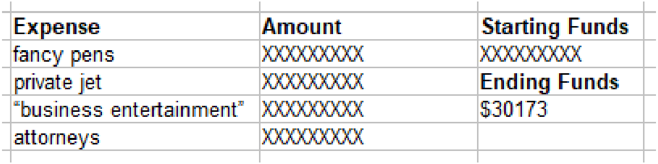
If only I could find someone with some **basic decency** and respect for other **people’s privacy** to help me out.

I am not asking for anything too crazy, just someone competent in **basic arithmetic** to help with some **bookkeeping.**

Is that person you? Would you **care to volunteer**?

Assignment: Basic Bookkeeping

Please go ahead and total the expenses by category.

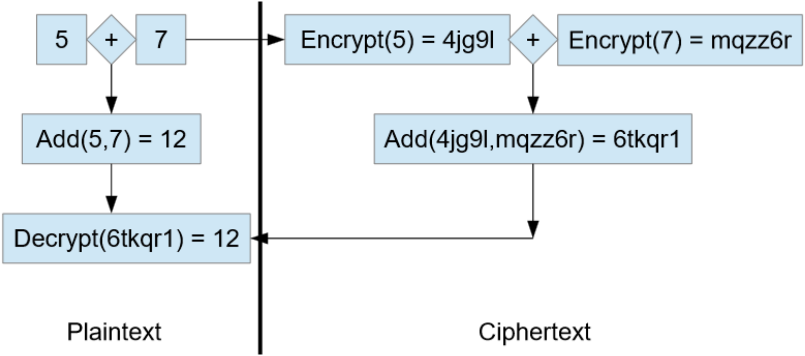


Expense	Amount	Starting Funds
fancy pens	XXXXXXXXX	XXXXXXXXX
private jet	XXXXXXXXX	**Ending Funds**
"business entertainment"	XXXXXXXXX	$30173
attorneys	XXXXXXXXX	

Easy, right? Thanks for handling this. It's so hard to find decent help these days. I need that jet deduction!

What's the problem here? Why is it hard to keep your private jet and business entertainment budget line items a secret?

Homomorphic Encryption



Homomorphic encryption allows one to do computations on encrypted data and get the “correct” answer after decryption.

History and Prophecy

Some landmark papers and people...

- (Gentry 2009) “A Fully Homomorphic Encryption Scheme”
- (Gennaro, Gentry, Parno, Raykova 2013) “Quadratic Span Programs and Succinct NIZKs without PCPs”

Craig Gentry won a MacArthur Fellowship (the “genius grant”) for his 2009 PhD thesis work on fully homomorphic encryption (able to compute an arbitrary algorithm on ciphertext) and was widely interviewed afterwards.

He said at the time, almost 10 years ago, that it might be 10 years before the technology really started to catch on...

What’s Out There?

- HomomorphicEncryption R Package
  - intuitive interface in R
  - http://www.louisaslett.com/HomomorphicEncryption/
- HElib - fully homomorphic encryption software library
  - the algorithms, close to the metal in C++
  - https://github.com/shaih/HElib
- PySEAL Python Package
  - intuitive interface in Python
  - https://github.com/Lab41/PySEAL
- OpenMined - crowdsourced, private machine learning
  - https://github.com/OpenMined

The HomomorphicEncryption Library

“For research purposes only”

- Created by academic statistician Louis Aslett [1]
- A wrapper for a C++ library, via Fan and Vercauteren [2]

Some notable limitations

- Exists for Mac and Linux but not Windows
- Pain around where R is really C under the hood
- (True of all HE:) More algebra, More problems

Homomorphic encryption is a new technology, and there is work still to be done bridging the space between basic artithmetic and scientific computing.

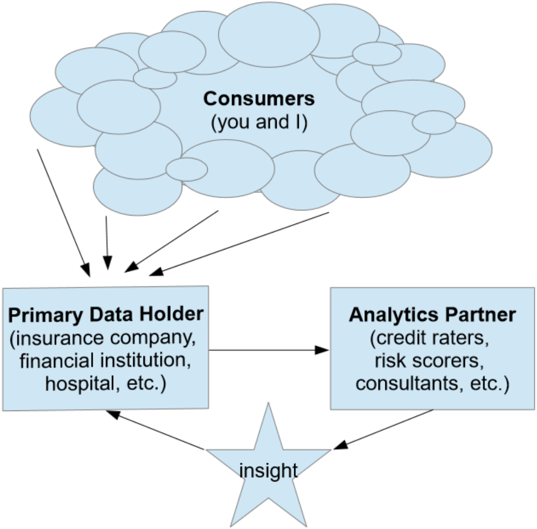
Business Layer: Analytics Partners

Businesses (i.e. insurance companies) often hand data off to **analytics partners** (i.e. risk scorers) to obtain some insight on that data. For a variety of reasons, the primary data holder might want to keep their data secret and / or the analytics partner might want to keep their model secret.

Depending on the industry, there are many businesses that fit into this rough category...

- **Risk scoring in insurance**
- Raters of consumer credit
- Clearing houses for financial transaction
- Many forms of consulting

Typical Data Flow



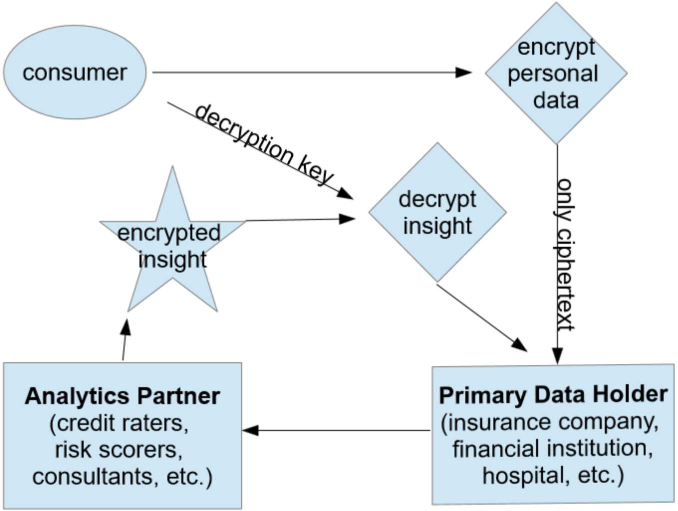
Consumers
(you and I)

Primary Data Holder
(insurance company, financial institution, hospital, etc.)

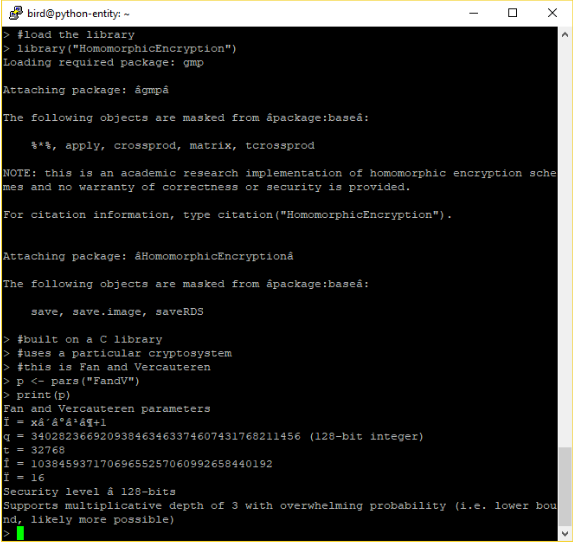
Analytics Partner
(credit raters, risk scorers, consultants, etc.)

insight

Data Flow with Homomorphic Encryption

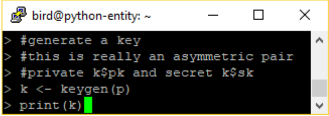

consumer
encrypt personal data
decryption key
decrypt insight
only ciphertext
encrypted insight
Analytics Partner (credit raters, risk scorers, consultants, etc.)
Primary Data Holder (insurance company, financial institution, hospital, etc.)

R Code 1: Getting Started

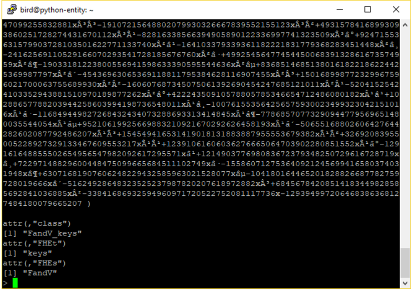


```
> #load the library
> library("HomomorphicEncryption")
Loading required package: gmp

Attaching package: âgmpâ

The following objects are masked from âpackage:baseâ:

    %*%, apply, crossprod, matrix, tcrossprod

NOTE: this is an academic research implementation of homomorphic encryption sche
mes and no warranty of correctness or security is provided.

For citation information, type citation("HomomorphicEncryption").


Attaching package: âHomomorphicEncryptionâ

The following objects are masked from âpackage:baseâ:

    save, save.image, saveRDS

> #built on a C library
> #uses a particular cryptosystem
> #this is Fan and Vercauteren
> p <- pars("FandV")
> print(p)
Fan and Vercauteren parameters
Ï = xâ´â°â¹â¶+1
q = 340282366920938463463374607431768211456 (128-bit integer)
t = 32768
Î = 10384593717069655257060992658440192
Ï = 16
Security level â 128-bits
Supports multiplicative depth of 3 with overwhelming probability (i.e. lower bou
nd, likely more possible)
>
```

R Code 2: Key Generation



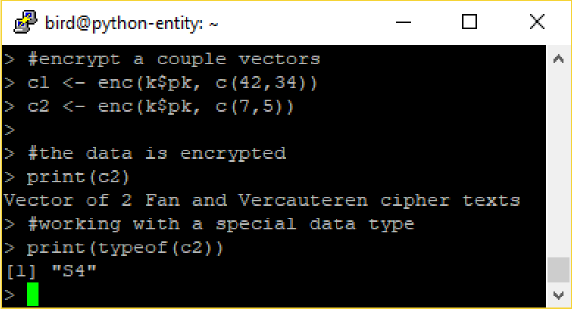
```
> #generate a key
> #this is really an asymmetric pair
> #private k$pk and secret k$sk
> k <- keygen(p)
> print(k)
```



```
47099255832881xÂ³Â³-19107215648802079930326667839552155123xÂ³Â²+4931578416899309
3860251728274431670112xÂ³Â¹-82816338566394905890122336997741323509xÂ³â°+92471553
631579903728103501622771133740xÂ²â¹-164103379339361182221831779368283451448xÂ²â¸
-24162569110529166070293541728185676760xÂ²â·+49925456477454450068391328616735749
59xÂ²â¶-19033181223800556941598633390595544636xÂ²âµ+8368514685138016182218622442
5369987797xÂ²â´-45436963065369118811795384628116907455xÂ²Â³+15016899877232996759
6021700063755689930xÂ²Â²-160607687345075061392690454247685121011xÂ²Â¹-5204152542
4103352943881510970189877262xÂ²â°+42224350910578805785346654712486080182xÂ¹â¹+10
288657788203944258603994198736548011xÂ¹â¸-10076155356425657593002349932304215101
6xÂ¹â·-116849449827268432434073288693313414845xÂ¹â¶-7786857077329094477956965148
0035544054xÂ¹âµ+95210619925669883210921670292626458193xÂ¹â´-50655168802606427644
282602087792486207xÂ¹Â³+154549416531419018131883887955553679382xÂ¹Â²+32692083955
005228927329133467609553217xÂ¹Â¹+12391061606036276665064703902280851552xÂ¹â°-129
161648855502654956547982092617295571xâ¹+121490377698083672379348250729616728719x
â¸+72297148829600448475099665684511102749xâ·-15586071275364092124569941658037403
1948xâ¶+63071681907606248229432585963021528077xâµ-104180164465201828826687782759
728019666xâ´-51624928648323525237987820207618972882xÂ³+6845678420851418344982858
5692841036885xÂ²-33841686932594960971720522752081117736x-12939499720646838636812
7484180079665207 )

attr(,"class")
[1] "FandV_keys"
attr(,"FHEt")
[1] "keys"
attr(,"FHEs")
[1] "FandV"
>
```

We end up with an asymmetric key pair with special powers.

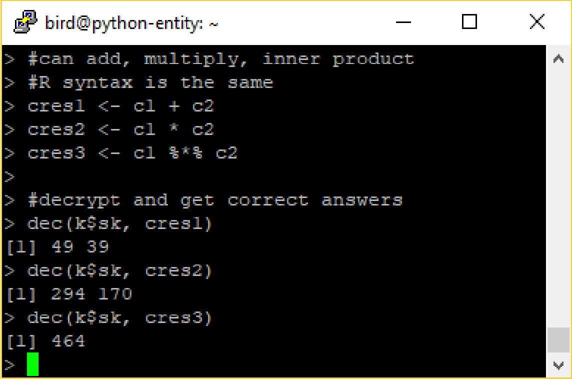
R Code 3: New Data Types



```
> #encrypt a couple vectors
> c1 <- enc(k$pk, c(42,34))
> c2 <- enc(k$pk, c(7,5))
>
> #the data is encrypted
> print(c2)
Vector of 2 Fan and Vercauteren cipher texts
> #working with a special data type
> print(typeof(c2))
[1] "S4"
>
```

We have new data types for individual pieces of ciphertext and also vectors thereof. They will do their best to pretend to be the analogous base R data types but they are not.

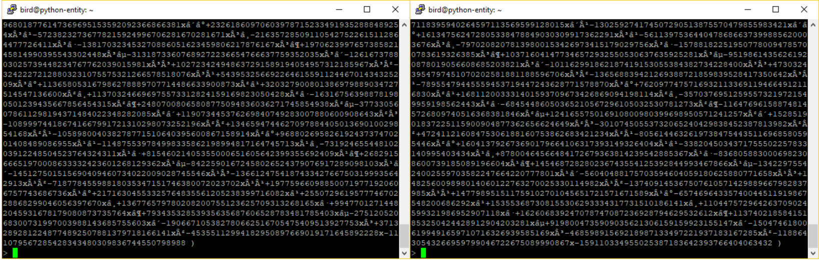
R Code 4: Operator Overloading



```
> #can add, multiply, inner product
> #R syntax is the same
> cres1 <- c1 + c2
> cres2 <- c1 * c2
> cres3 <- c1 %*% c2
>
> #decrypt and get correct answers
> dec(k$sk, cres1)
[1] 49 39
> dec(k$sk, cres2)
[1] 294 170
> dec(k$sk, cres3)
[1] 464
> 
```

R will add our encrypted numbers, and vectors thereof, using the usual syntax. Any code written purely in R using this syntax will run and get the “right” answer on encrypted data.

R Code 4.5: What really just happened?

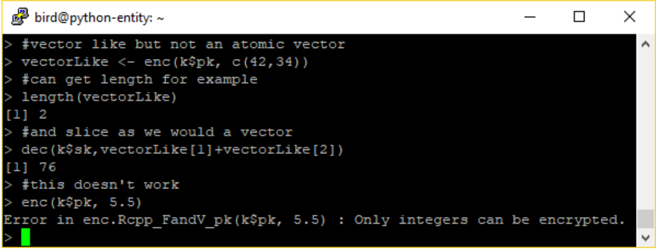


```
968018776147369695153592092366866381xâ´â°+232618609706039787152334919352888848925
4xÂ³â¹-57238232736778215924996706281670281671xÂ³â¸-216357285091105427522615112 86
4477726411xÂ³â·-138170323453270886051623459806217876167xÂ³â¶+1970623997657385821
4581499039554330244 8xÂ³âµ-31318733607689272236654766637759352035xÂ³â´-1261673788
030257394482347677620390159 81xÂ³Â³+102723424948637291589194054957312185967xÂ³Â²-
3242227212880323107557532126657851807 6xÂ³Â¹+543953256692264615591124467014343252
09xÂ³â°+113658053167986278889707714486633900873xÂ²â¹+320327900801386979889034727
514547136600xÂ²â¸+113703246696975573312824159169823050428xÂ²â·-16316756398878198
0501239435667856454315xÂ²â¶+248070080658087750948360362717458549 38xÂ²âµ-37733056
078611298194371484022348282085xÂ²â´+119073445376269840749283007880600908643xÂ²Â³
-108999744186741667991721310298073252196xÂ²Â²+1346594744627097884405013690100298
54168xÂ²Â¹-105898004038278771510640395600867158914xÂ²â°+968802695826192437374702
01408489086955xÂ¹â¹-114875539784998335862198994817164745713xÂ¹â¸-731924655448102
03912248504523764324311xÂ¹â·+815460214053550006516056423993556924 09xÂ¹â¶+2682915
66651970008633332423601268129362xÂ¹âµ-84225901672458026524379076917289098103xÂ¹â
´-145127501515690409460734022009028745546xÂ¹Â³-136612475418743342766750319993564
2913xÂ¹Â²-71877845598818035347151746380072023702xÂ¹Â¹+19775966098850071977192060
6757743686736xÂ¹â°+21716304553325764835561205238399716082xâ¹+2550729619577746702
28868299046056397670xâ¸+136776579780208200755123625709313268165xâ·+9947701271448
204593167817908087373576 4xâ¶+79343532853935635687606528783481785403xâµ-275120520
68300731997003988143685755603xâ´-190667105382780662516705475409513927753xÂ³+3713
289281224877489250788137971816614 1xÂ²-45355112994182950897669019171645892228x-11
1079567285428343480309836744550798988 )
>
```

```
711839594026459711356959912801 5xâ´Â¹-130259274174507290513875570479855983421xâ´â
°+161347562472805338478849030309917362291xÂ³â¹-561139753644047868663739988562000
3676xÂ³â¸-79702082078139800153426973415179029756xÂ³â·-15788182251950778009478570
0783619326385xÂ³â¶+103716041477346572932550530637635925281xÂ³âµ-951986143562619 2
0878019056608685203821xÂ³â´-101162991862187419153055384382734228400xÂ³Â³+4730324
395479745107020258188118859670 6xÂ³Â²-136568839421269388721859839528417350642xÂ³Â
¹-78955479445559453719447243628771578870xÂ³â°+762097747571693211336911946649121 1
6830xÂ²â¹+168112003331401593709673426869094198114xÂ²â¸-3570376951259557321972154
9959198562443xÂ²â·-68454486050365210567296105032530781273xÂ²â¶-11647696158874814
5726809740516368381846xÂ²âµ+124165575016910800980399698950571241257xÂ²â´+1528519
01837225115900904877362656624649xÂ²Â³-30107450553732065240429838452387813982xÂ²Â
²+47241121608475306188160753862683421234xÂ²Â¹-8056144632619738475443511696858059
5446xÂ²â°+160413792673690179664106317393149326404xÂ¹â¹-3382045034371755502257833
140995403434xÂ¹â¸+87800464566484172679363814239542885367xÂ¹â·-836805883000698230
860073918508919660 4xÂ¹â¶+145468728280236743554125392844993467866xÂ¹âµ-1342297554
24002559703582247664220777801xÂ¹â´-56040488175703594604059180625880771658xÂ¹Â³+1
48256009898014060122763270025330114982xÂ¹Â²-13740914536750761057142988967982837
985xÂ¹Â¹+147798951511759102701045651721571671589xÂ¹â°-6574696433574000445119198 67
548200686292xâ¹+153553687308155306293334317731510186141xâ¸+110447572964263709024
599321986952907118xâ·+162606839247078747087236928794629532612xâ¶+113740218584151
853250424428912904203281xâµ+919800473590903562130615915992315514 7xâ´-15047461800
6199491659710716326939585169xÂ³-46859891569218987133497221937183167285xÂ²-118864
3054326695979904672267508999086 7x-159110334955025387183642393766404063432 )
>
```

These polynomials are what we just “added” together. When we decrypted, whatever we did to those polynomials turned out to be the right thing in that we managed to compute addition of the plaintext correctly.

R Code 5: Some Wrinkles



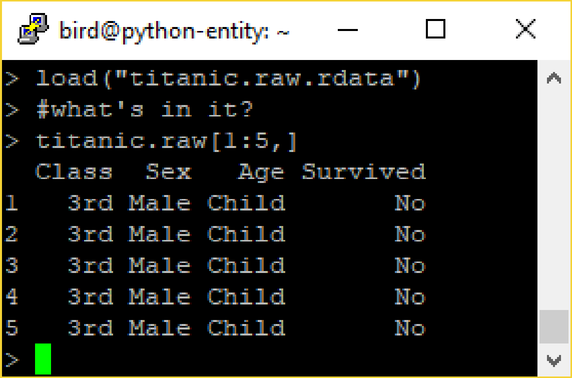
```
> #vector like but not an atomic vector
> vectorLike <- enc(k$pk, c(42,34))
> #can get length for example
> length(vectorLike)
[1] 2
> #and slice as we would a vector
> dec(k$sk,vectorLike[1]+vectorLike[2])
[1] 76
> #this doesn't work
> enc(k$pk, 5.5)
Error in enc.Rcpp_FandV_pk(k$pk, 5.5) : Only integers can be encrypted.
> 
```

The cipher **only works for integers**, which is a huge pain in the ass in practice but not so bad in principle.

You can **approximate by fractions**, and computations on fractions are just computations on integers as $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{db}$.

This is not the level of abstraction we are used to, though.

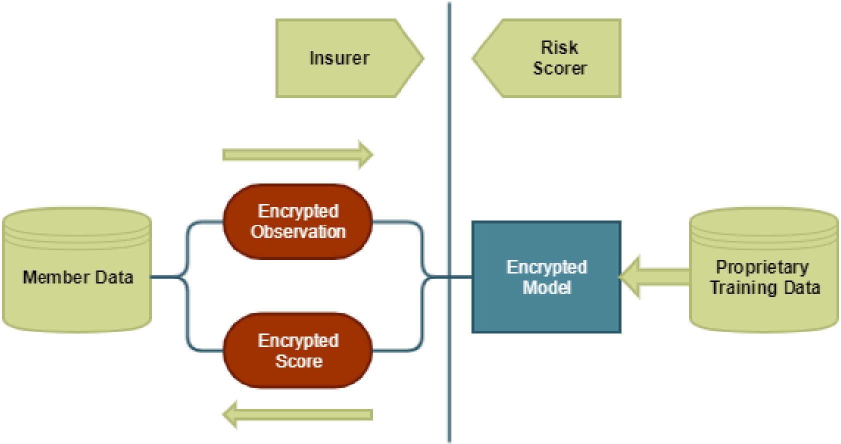
R Code 6: A Familiar Classification Problem



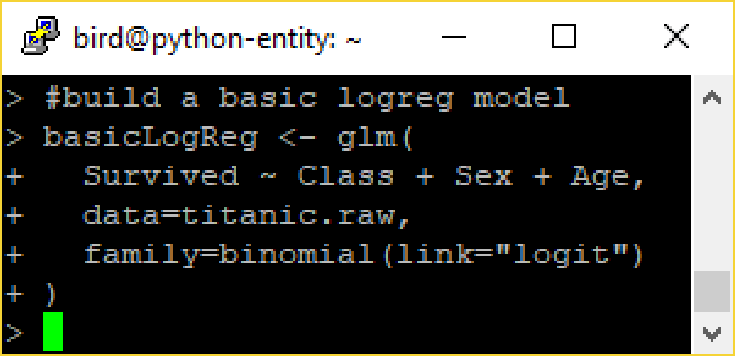
```
> load("titanic.raw.rdata")
> #what's in it?
> titanic.raw[1:5,]
  Class  Sex   Age Survived
1   3rd Male Child       No
2   3rd Male Child       No
3   3rd Male Child       No
4   3rd Male Child       No
5   3rd Male Child       No
> 
```

We’ll train a logistic regression model on the Titanic dataset, encrypt the data of that model, and compute survival odds **blind to both the observations used and the results.**

Encrypted Model Data Flow



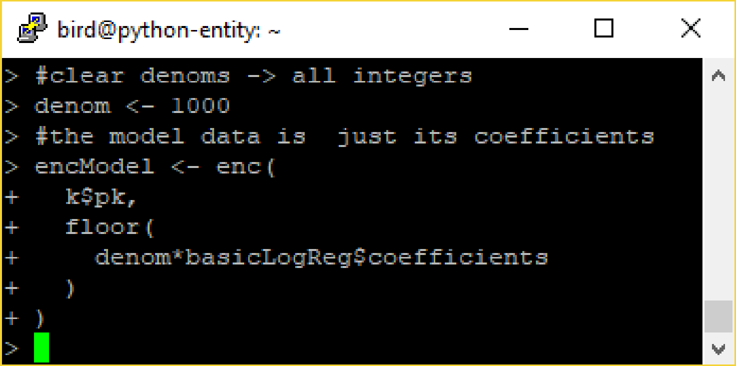
R Code 7: Train on Plaintext



```
> #build a basic logreg model
> basicLogReg <- glm(
+    Survived ~ Class + Sex + Age,
+    data=titanic.raw,
+    family=binomial(link="logit")
+ )
> 
```

Here we imagine we had some in-house data we could use to train our model as usual, but maybe we have clients that don’t want to share information about themselves.

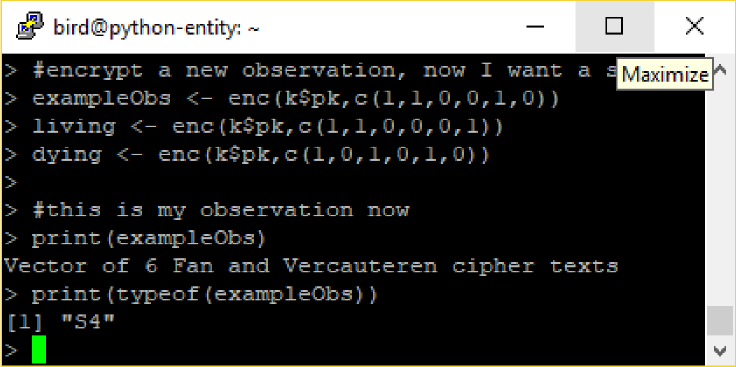
R Code 8: Encrypt the Model



```
> #clear denoms -> all integers
> denom <- 1000
> #the model data is  just its coefficients
> encModel <- enc(
+   k$pk,
+   floor(
+     denom*basicLogReg$coefficients
+   )
+ )
> 
```

A logistic regression model is determined entirely by its coefficients, which we can encrypt like anything else after clearing denominators (-ish) so we can work with integers.

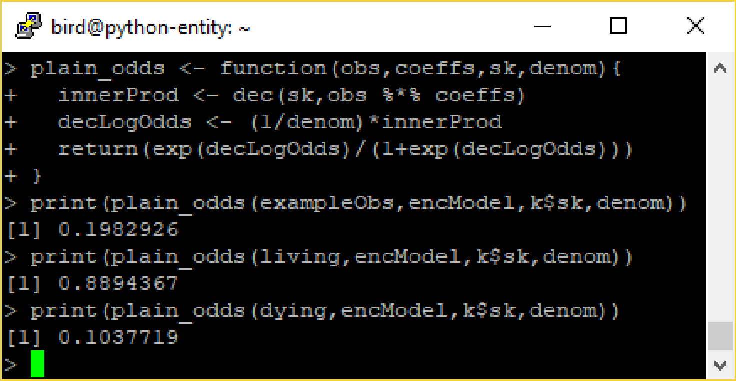
R Code 9: Encrypted Observations



```
> #encrypt a new observation, now I want a s
> exampleObs <- enc(k$pk,c(1,1,0,0,1,0))
> living <- enc(k$pk,c(1,1,0,0,0,1))
> dying <- enc(k$pk,c(1,0,1,0,1,0))
>
> #this is my observation now
> print(exampleObs)
Vector of 6 Fan and Vercauteren cipher texts
> print(typeof(exampleObs))
[1] "S4"
> 
```

We'll reformat a bit as we are going to write our own prediction function, and then we can take example observations and encrypt them so the model owner need not see our business.

R Code 10: Insights



```
> plain_odds <- function(obs,coeffs,sk,denom){
+   innerProd <- dec(sk,obs %*% coeffs)
+   decLogOdds <- (1/denom)*innerProd
+   return(exp(decLogOdds)/(1+exp(decLogOdds)))
+ }
> print(plain_odds(exampleObs,encModel,k$sk,denom))
[1] 0.1982926
> print(plain_odds(living,encModel,k$sk,denom))
[1] 0.8894367
> print(plain_odds(dying,encModel,k$sk,denom))
[1] 0.1037719
> 
```

We can now compute our model, defined by encrypted data, on encrypted observations to get an encrypted answer. When we decrypt, we see that we are getting good results.

Security Subtleties 1

Let's say I have an encrypted string and an algorithm that takes a plaintext search string and returns the indices where that string appears.

$$\text{search(“a”)} = \{2, 8, 13\}$$

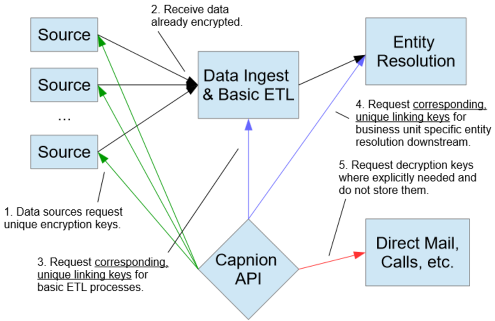
$$\text{search(“b”)} = \{1\}$$

$$\text{search(“c”)} = \{\}$$

and so on ...

It will never take me more than 26 tries before I can reconstruct **“bart skateboards”** as the plaintext.

Security Subtleties 2



Source
Source
...
Source
2. Receive data already encrypted.
Data Ingest & Basic ETL
Entity Resolution
4. Request corresponding, unique linking keys for business unit specific entity resolution downstream.
5. Request decryption keys where explicitly needed and do not store them.
1. Data sources request unique encryption keys.
3. Request corresponding, unique linking keys for basic ETL processes.
Capnion API
Direct Mail, Calls, etc.

Generate keys that enable **particular operations on particular sets of data**, and specifically keys that give information on **how to link records but not on how to decrypt** or do other computations

Questions

Any questions?

Feel free to contact me at acmueller@capnion.com

Slides are available at
https://github.com/capnion/ ...
random/blob/master/acm_rug_he_may19.pdf

References

L. J. M. Aslett, P. M. Esperança, and C. C. Holmes, “A review of homomorphic encryption and software tools for encrypted statistical machine learning,” tech. rep., University of Oxford, 2015.

J. Fan and F. Vercauteren, “Somewhat practical fully homomorphic encryption,” tech. rep., 2012.