[5] Big Data: Classification

Matt Taddy, University of Chicago Booth School of Business

faculty.chicagobooth.edu/matt.taddy/teaching

[5] Classification

K-nearest neighbors and group membership.

Binary classification: from probabilities to decisions.

- Misclassification, sensitivity and specificity.

Multinomial logistic regression: fit and probabilities.

DMR and distributed computing.

Nearest Neighbors

In classification, y is membership in a category $\{1, 2, ..., m\}$.

Each class observation y_i is accompanied by covariates $\mathbf{x}_i$.

The classification problem: given new $\mathbf{x}$ what is $\hat{y}$?

KNN: what is the most common class around x?

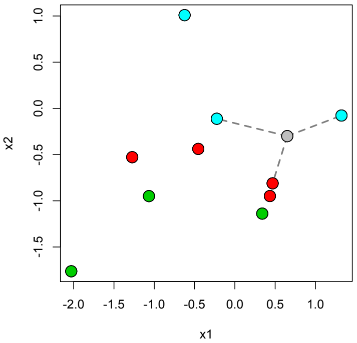
Take your K-nearest neighbors $\mathbf{x}_{i_1} \ldots \mathbf{x}_{i_k}$, and let them vote.

Nearness is in euclidean distance: $\sqrt{\sum_{j=1}^{p}(x_j - x_{ij})^2}$. (units!)

$\hat{y}$ is assigned the most common category in $\{y_{i_1} \ldots y_{i_k}\}$.

Since we're calculating distances on $\mathbf{X}$, scale Matters!
We'll use R's scale function to divide each x_j by $\mathrm{sd}(x_j)$
The new units of distance are in *standard deviations*.

Nearest Neighbors



K-NN's collaborative estimation:

Each neighbor votes.

Neighborhood is by shortest distance (shown as the dashed lines).

The relative vote counts provide a very crude estimate of probability.

For 3-nn, $\mathrm{p}(\text{blue}) = 2/3$, but for 4-nn or 2-nn, it's only $1/2$.
Sensitive to neighborhood size (think about extremes: *1* or *n*).

Forensic Glass Analysis

Classifying shards of glass

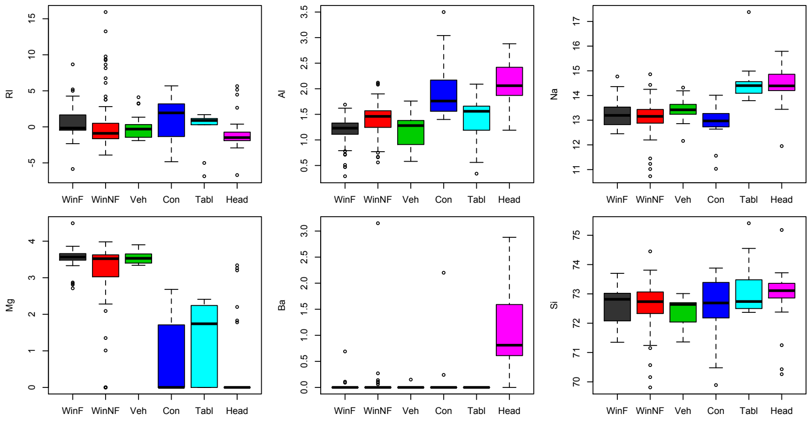
Refractive index, plus oxide %

Na, Mg, Al, Si, K, Ca, Ba, Fe.

6 possible glass types

`WinF`:	float glass window
`WinNF`:	non-float window
`Veh`:	vehicle window
`Con`:	container (bottles)
`Tabl`:	tableware
`Head`:	vehicle headlamp

Glass Data: characteristic by type



Some covariates are clear discriminators (Ba for headlamps, Mg for windows) while others are more subtle (Refractive Ind).

Nearest neighbors in R

Load the `class` package which includes function `knn`.

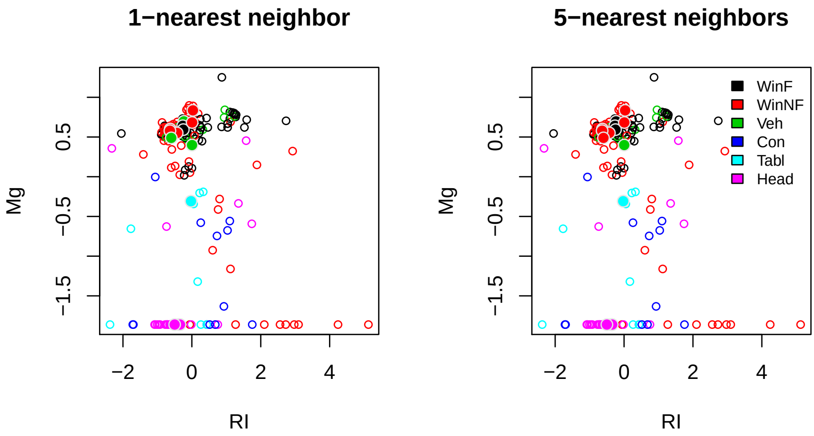
`train` and `test` are covariate matrices, `cl` holds known y's.

You set `k` to specify how many neighbors get to vote.

Specify `prob=TRUE` to get neighbor vote proportions.

```
knn(train=xobserved, test=xnew, cl=y, k=3)
nn1 <- knn(train=x[ti,], test=x[-ti,], cl=y[ti], k=1)
nn5 <- knn(train=x[ti,], test=x[-ti,], cl=y[ti], k=5)
data.frame(ynew,nn1,nn5)
          ynew     nn1     nn5
          WinF    WinF    WinF
           Con     Con    Head
          Tabl   WinNF   WinNF
```

KNN classification in the RI×Mg plane.



Open circles are observations and closed are predictions.

The number of neighbors matters!

Limits of KNN

Nearest neighbors is simple, but limited

There is no good way to choose K.

Cross-validation works, but is unstable: new data $\Rightarrow$ new K.

And the classification is *very* sensitive to K.

All you get is a classification, with only rough probabilities.

These are often zero or one: useless in making decisions.

Without probabilities, you cannot assess misclassification risk.

But the basic idea is the same as in logistic regression:
  Observations with similar $\mathbf{x}$'s should be classified similarly.

Probability, cost, and classification

Many decisions can be reduced to binary classification.

There are two ways to be wrong in a binary problem.

> False positive: predict $\hat{y} = 1$ when $y = 0$.
> False negative: predict $\hat{y} = 0$ when $y = 1$.

There will be costs associated with each type of error.

To make optimal decisions, you need to estimate probabilities.

Once you know $\hat{p}$, the probability of $y = 1$, you can assess risk.

We know how to estimate probabilities: logistic regression!

Credit Classification

Credit scoring is a classic problem of classification.

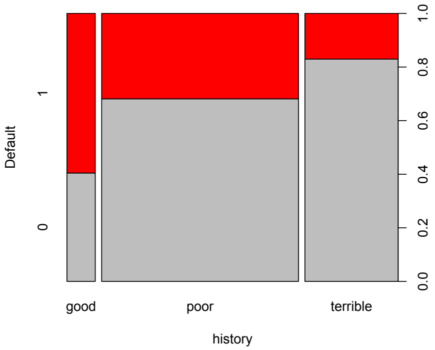
Take borrower/loan characteristics and previous defaults,
use these to predict performance of potential new loans.
Bond-rating is a multi-class extension of the problem.

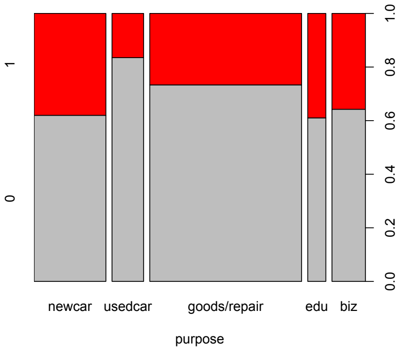
Consider the German loan/default data in `credit.csv`.

- Borrower and loan characteristics: job, installments, etc.
- Pretty messy data, needs a bit of a clean...

Choice Sampling

A caution on retrospective sampling



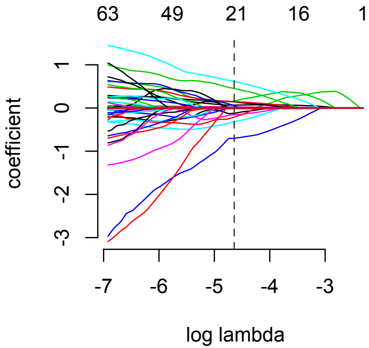


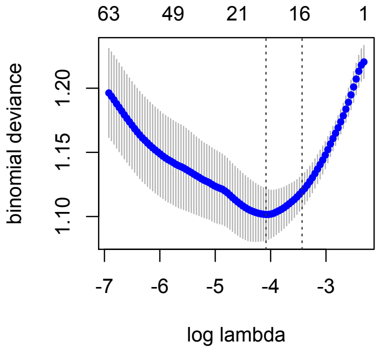
See anything strange here? Think about your data sources!

Conditioning helps here, but won't always solve everything...

German Credit Lasso

Create a numeric **x** and run lasso logistic regression.





```
> sum(coef(credscore)!=0) 13 # cv.1se
> sum(coef(credscore, s="min")!=0) 21 # cv.min
> sum(coef(credscore$gamlr)!=0) 21 # AICc
```

Using probabilities to make decisions

Say that, on average, for every 1$ loaned you make 25¢ in interest if it is repayed but lose 1$ if they default.

This gives the following action-*cost* matrix

	payer	*defaulter*
loan	-0.25	1
no loan	0	0

Suppose you estimate p for the probability of default.
Expected *profit* from lending is greater than zero if

$$(1-p)\frac{1}{4} - p > 0 \quad \Leftrightarrow \quad \frac{1}{4} > \frac{5}{4}p \quad \Leftrightarrow \quad p < 1/5$$

So, from this simple matrix you should lend whenever probability of default is less than 0.2.

FP and FN Rates

Any classification cutoff (e.g., our $p = 1/5$ rule, built from an expected profit/loss analysis) has some basic properties.

False Positive Rate: $\#$ misclassified as pos / $\#$ classified pos.

False Negative Rate: $\#$ misclassified as neg / $\#$ classified neg.

In-Sample rates for our $p = 1/5$ rule:

```
## false positive rate
> sum( (pred>rule)[default==0] )/sum(pred>rule)
[1] 0.6704289
## false negative rate
> sum( (pred<rule)[default==1] )/sum(pred<rule)
[1] 0.07017544
```

For comparison, a $p = \frac{1}{2}$ cut-off gives FPR=0.27, FNR=0.28.

Sensitivity and Specificity

Two more common classification rates are
sensitivity: proportion of true $y = 1$ classified as such.
specificity: proportion of true $y = 0$ classified as such.

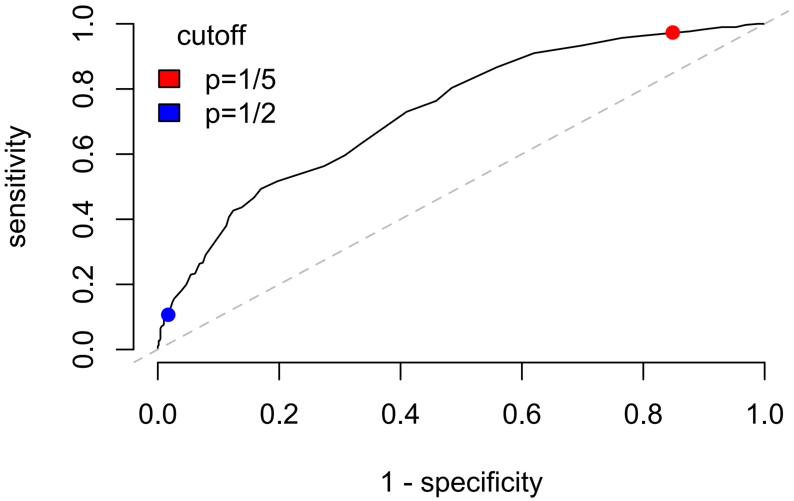
A rule is sensitive if it predicts 1 for most $y = 1$ observations, and specific if it predicts 0 for most $y = 0$ observations.

```
> mean( (pred>1/5)[default==1] )# sensitivity
[1] 0.9733333
> mean( (pred<1/5)[default==0] )# specificity
[1] 0.1514286
```

Contrast with FP + FN, where you are dividing by *total classified a certian way*. Here you divide by *true totals*.

Our rule is sensitive, not specific, because we lose more with defaults than we gain from a payer.

The ROC curve: sensitivity vs 1-specificity



From signal processing: Receiver Operating Characteristic.

A tight fit has the curve forced into the top-left corner.

Multinomial Logistic Regression

Probabilities are the basis for good cost-benefit classification.

We can get binary probabilities from binomial regression.

In *multi-class* problems, response y is one of K ‘categories’.

Rewrite $\mathbf{y}_i = [0, 1, \ldots, 0]$, where $y_{ik} = 1$ if response i is class k.

Then we need a model for

$$\mathbb{E}[y_{ik}|\mathbf{x}_i] = f(\mathbf{x}_i'\boldsymbol{\beta}_k).$$

$\Rightarrow$ we need a to fit regression coefficients $\boldsymbol{\beta}_k$ for *each* class.

To move to probabilites for multi-class problems, we need a Multinomial Regression for $\mathrm{p}(y_k = 1|\mathbf{x}) = p_k(\mathbf{x})$, $k = 1 \ldots K$.

Extend logistic regression via the multinomial logit:

$$\mathrm{p}(y_j = 1|\mathbf{x}) = p_j(\mathbf{x}) = \frac{e^{\mathbf{x}'\boldsymbol{\beta}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}'\boldsymbol{\beta}_k}}$$

Note separate coefficients for each class: $\boldsymbol{\beta}_k$.

Fitting this is called 'multinomial logistic regression'.

Multinomial Logistic Regression

Multinomial response is y_{ik} for $k \in 1 \dots K$: a *class id*.

Say k_i is the class for observation i, so $y_{k_i} = 1$.

Given class probabilities $\mathbf{p}_i = [p_{i1} \dots p_{iK}]$, we get

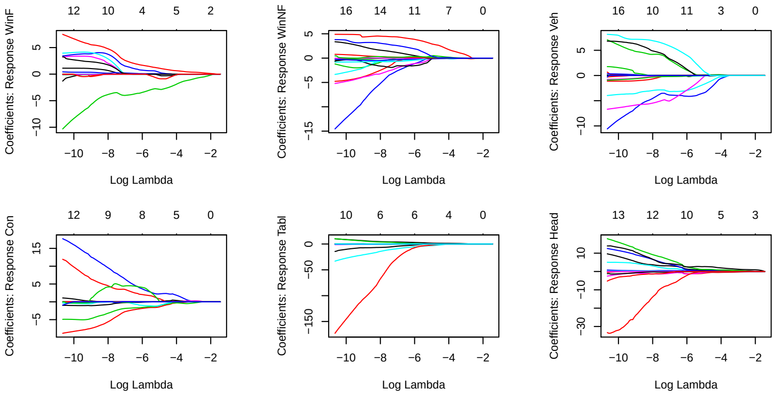
$$\mathrm{LHD} \propto \prod p_{ik_i} \quad \text{and} \quad \mathrm{Dev} \propto -2 \sum_i \log(p_{ik_i})$$

We'll fit these the usual way: penalized deviance minimization.

$$\min \left\{ -\frac{2}{n} \sum_i \log p_{ik_i}(\boldsymbol{\beta}) + \sum_k \sum_j \lambda |\beta_{kj}| \right\}$$

You can also have λ_k: different penalty for each class.

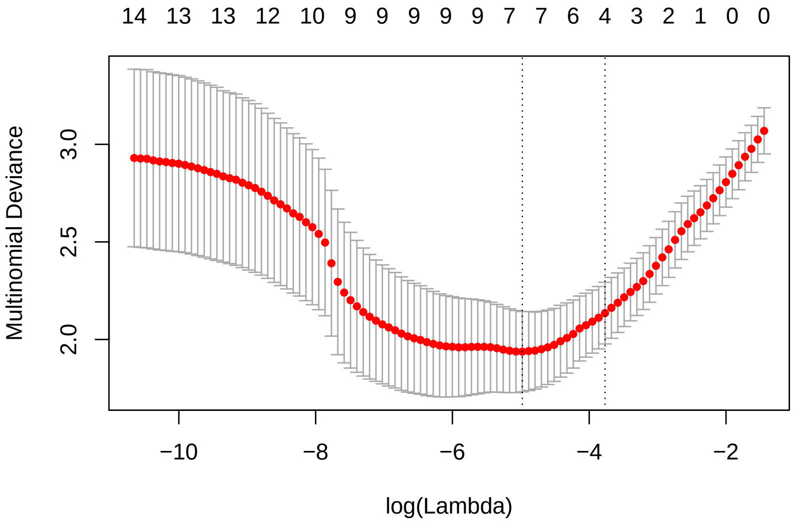
Fit the model in `glmnet` with `family="multinomial"`.



A separate path plot for every class.

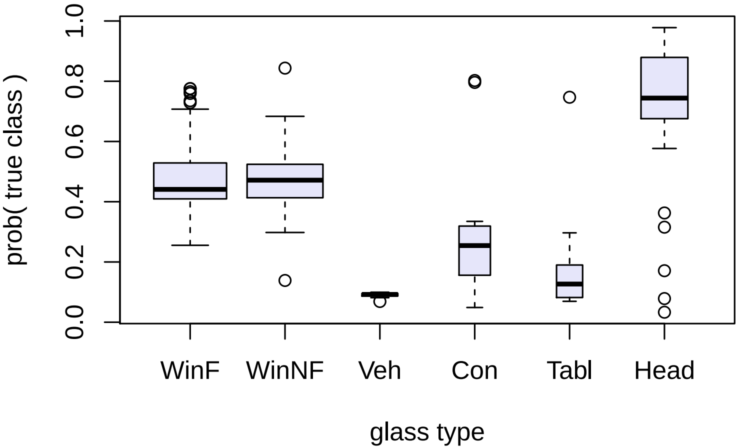
See `glass.R` for coefficients, prediction, and other details.

We can do OOS experiments on *multinomial deviance*.



And use this to choose λ (one shared for all classes here).

The 'fit plot' for multinomials: $\hat{p}_{ik_i}$, prob of true class, on k_i.



`Veh`, `Con`, `Tabls` have low fitted probabilities, but they are generally more rare in this sample (width of box is $\propto$ count).

MN classification via decision costs

Suppose a simple cost matrix has

	WinF	WinNF	Veh	Con	Tabl	Head
$\hat{k}$ =Head	9	9	9	9	9	0
$\hat{k}$ ≠Head	0	0	0	0	0	1

e.g. a court case where Head is evidence for the prosecution (innocent until proven guilty, and such).

Then expected cost of $\hat{k} \neq$ Head is greater than $\hat{k} =$ Head if

$$p_{\texttt{head}} > 9(1 - p_{\texttt{head}}) \quad \Leftrightarrow \quad p_{\texttt{head}} > 0.9$$

If you don't have asymmetric costs,
just use a maximum probability rule: $\hat{k} = \mathrm{argmax}_k \ \hat{p}_k$.
You can get this in R with `apply(probs,1,which.max)`.

Interpreting the MN logit

We're estimating a function that sums to one across classes. But now there are K categories, instead of just two.

The log-odds interpretation now compares between classes:

$$\log\left(\frac{p_a}{p_b}\right) = \log\left(\frac{e^{\mathbf{x}'\boldsymbol{\beta}_a}}{e^{\mathbf{x}'\boldsymbol{\beta}_b}}\right) = \mathbf{x}[\boldsymbol{\beta}_a - \boldsymbol{\beta}_b].$$

For example, with a one unit increase in `Mg`:

```
# odds of non-float over float drop by 30-40%
exp(B["Mg","WinNF"]-B["Mg","WinF"])
 0.6633846
# odds of non-float over Con increase by 60-70%
exp(B["Mg","WinNF"]-B["Mg","Con"])
 1.675311
```

Interpreting the MN logit

Interactions also translate directly.

You just need to look at differences across class coefficients:

In my run, the `RI:Mg` interaction coefficient is estimated as $\beta_{\texttt{winNF},\texttt{RI:Mg}} = -.05$ for `WinNF` and 0 for both `WinF` and `Con`.

So the above odds *decrease* by 5% under every extra unit `RI`.

($\exp(-0.05) \approx 0.95$; see code for more detail)

k-against-others odds are not as simple:

$$\log[p_k/(1-p_k)] = \mathbf{x}\boldsymbol{\beta}_k - \log \sum_{j \neq k} e^{\mathbf{x}\boldsymbol{\beta}_j}.$$

A nonlinear function of $\mathbf{x}$!

All we can say is something like

Unit increase in x_j multiplies odds numerator for class k by β_{kj}.

An alternative version of MN logit

You might have noticed: multinomial regression can be slow...

This is because everything needs to be done K times!

And each $\hat{\boldsymbol{\beta}}_k$ depends on the others: $p_{ik} = e^{\mathbf{x}'\boldsymbol{\beta}_k} / \sum_j e^{\mathbf{x}'\boldsymbol{\beta}_j}$.

It turns out that multinomial logistic regression is *very similar* to

$$\mathbb{E}[y_{ik}|\mathbf{x}_i] = \exp(\mathbf{x}_i'\boldsymbol{\beta}_k).$$

That is, K *independent* log regressions for each class k.

The full regression is $y_{ik} \sim \mathrm{Poisson}(\exp[\mathbf{x}_i'\boldsymbol{\beta}_k])$, which is the glm for 'count response'. Deviance is $\propto \sum_{i=1}^{n} \exp(\mathbf{x}_i'\boldsymbol{\beta}_k) - y_i(\mathbf{x}_i'\boldsymbol{\beta}_k)$.

Distributed Multinomial Regression

Since each $y_{ik} \sim \mathrm{Poisson}(e^{\mathbf{x}_i'\boldsymbol{\beta}_k})$ regression is independent, wouldn't it be faster to do these all at the same time? Yes!

`dmr` function in the `distrom` library does just this.
In particular, `dmr` minimizes

$$\sum_{i=1}^{n} \exp(\mathbf{x}_i'\boldsymbol{\beta}_k) - y_i(\mathbf{x}_i'\boldsymbol{\beta}_k) + \lambda_k \sum_j |\beta_{jk}|$$

along a path of λ_k *in parallel for every response class k*.
We then use AICc to get a different $\hat{\lambda}_k$ for each k.

You can use $\hat{\boldsymbol{\beta}}_1 \ldots \hat{\boldsymbol{\beta}}_K$ as if they are for a multinomial logit.
The intercepts differ from `glmnet`'s, but that's a wash anyways.

DMR

`dmr` is a faster way to fit multinomial logit in parallel.
It's based on `gamlr`, so the syntax will be familiar.

```
dmr(cl, covars, counts, ...)
```

- `covars` is **x**.
- `counts` is **y**. Can be a factor variable.
- `...` are arguments to `gamlr`.
- `cl` is a parallel socket cluster.

It takes `coef` and `predict` as you're used to.

The returned `dmr` object is actually a list of K gamlr objects, and you can call plot, etc, on each of these too if you want.

“ **to compute in parallel** ”
do many calculations at the same time on different processors.

Supercomputers have long used parallelism for massive speed. Since 2000’s, it has become standard to have many processor ‘cores’ on consumer machines. Even my phone has 4.

You can take advantage of this without even knowing.

- Your OS runs applications on different cores.
- Videos run on processing units with 1000s of tiny cores.

And numeric software can be set up to use multiple processors.
e.g., if you build R ‘from source’, you can set this up.

Parallel Computing in R

R's `parallel` library lets you take advantage of many cores.

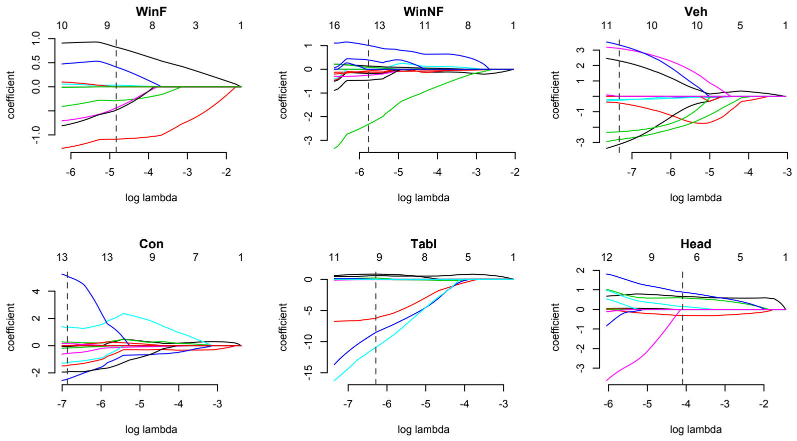
It works by organizing *clusters* of processors.

To get a cluster of cores do `cl <- makeCluster(4)`
You can do `detectCores()` to see how many you have.
If you're on a unix machine (mac/linux), you can ask for `makeCluster(4,type="FORK")` and it will often be faster.

After building `cl`, just pass it to `dmr` and you're off to the [parallel] races. Use `stopCluster(cl)` when you're done.

Note: this requires your computer is setup for parallelization. This *should* be true, but if not you can run `dmr` with `cl=NULL`.

DMR for glass data



The vertical lines show AICc selection: note it moves!

Note that `glmnet` `cv.min` rule chose $\log \hat{\lambda} \approx 5$.

Massive Text Regressions

The motivation for DMR actually comes from text mining.

The task is the estimate $\hat{\boldsymbol{\beta}}_k$ for $\mathbb{E}[y_{ik}|\mathbf{x}_i] = \exp(\mathbf{x}_i'\boldsymbol{\beta}_k)$
where y_{ik} is the count for word k in document i
and $\mathbf{x}_i$ are document attributes (author, date, etc).

Sounds simple: just regression for word counts.

However, you'll need to do this quickly for 100k-1mil words!
And the corpus of documents is far to big to fit on one computer.

Diversion: Data Distribution and Big Data

Parallel: many computations at once, using the same data.
Distributed: independent computations on many datasets.

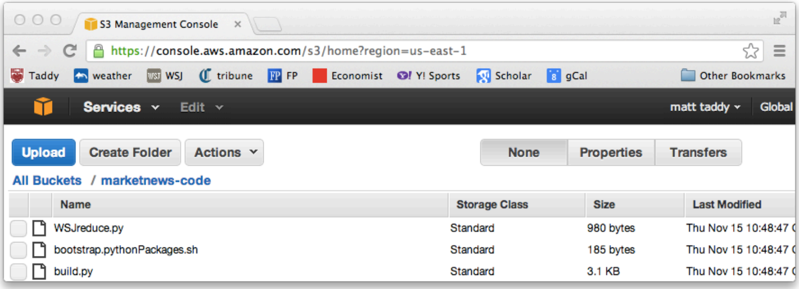
Parallel computing is good for large computations,
but for truly Big Data we need distributed algorithms.

The DMR idea is distributable.

Take advantage of massive bandwidth

Storage systems like Hadoop's HDFS and Amazon's S3 split your data into little pieces (e.g. 64kb of a file) that are kept wherever is convenient (often in more than one place).

You interact only with a map of where things are.



Name	Storage Class	Size	Last Modified
WSJreduce.py	Standard	980 bytes	Thu Nov 15 10:48:47
bootstrap.pythonPackages.sh	Standard	185 bytes	Thu Nov 15 10:48:47
build.py	Standard	3.1 KB	Thu Nov 15 10:48:47

Much of database theory is gone.

Just append new info and toss it in the cloud.

Distributed Analysis with MapReduce

Simple but powerful algorithm framework: *a recipe for recipes*.
Publicized by Google around 2004, but ideas pre-date that.

A Map-Reduce algorithm has three steps
  Map: calculate and sort relevant statistics by key.
  Partition: make sure records with the same key end up
    on the same machine (this is the hadoop magic).
  Reduce: apply some action by key.

Shared-key subsets are sorted and dispatched to ‘reducers’, each of which does some statistical analysis of the data.

Easy: Streaming Hadoop with S3 storage on Amazon EMR.

```
EMR -input s3://indir -output s3://outdir
    -mapper s3://map.py -reducer s3://reduce.R
```

MapReduce is great, but it is a pretty rigid recipe.

Many models are fit by *iteration*: update the model, look at what happened, then make another set of updates.

One can chain together MapReduce steps. But MR writes to disk after each run, so this will include tons of I/O slowness.

Instead: Spark! It is just like MR, except that the results and data stay ‘in memory’ on each machine until you are done.

Spark is getting closer to mainstream.
Runs on EMR, but you need to write in python or scala.
sparkR exists, but you need your own cluster...

Back to big text regressions

Consider raw documents stored in a distributed file system (i.e., many different machines) such as HDFS or Amazon S3.

A MapReduce Algorithm

Map: parse documents to output lines `word docid|count`.

Partition: All lines for given word k go to same machine.

Reduce: regress counts for k onto the doc attributes,
select optimal $\hat{\boldsymbol{\beta}}_k$ and send just this back to head.

Collect all $\hat{\boldsymbol{\beta}}_1 \ldots \hat{\boldsymbol{\beta}}_K$ and you've got your text model!

Massively scalable: build out not up.
Distributed computing is something we can discuss if you want with your project. The tools are open source and require tech savvy. You'll want to be solid on fundamentals before diving in.