A Primer on ML

ABFER 2017 – Singapore
Masterclass part II
Matt Taddy

ML is fast semiparametrics with Bayesian heuristics to avoid overfit

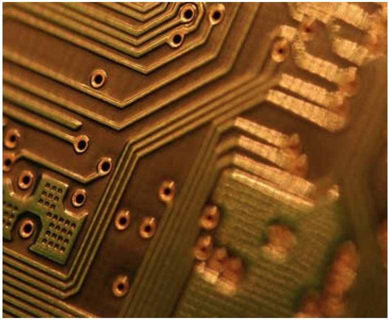
The plan

Out-of-Sample vs In-Sample performance

Regularization paths, the lasso, and penalty selection

Trees, forests, and regularization via the bootstrap

Example: Semiconductor Manufacturing Processes



Very complicated operation
Little margin for error.

Hundreds of diagnostics
Useful or debilitating?

We want to focus reporting and better predict failures.

$\mathbf{x}$ is 200 input signals, y has 100/1500 failures.

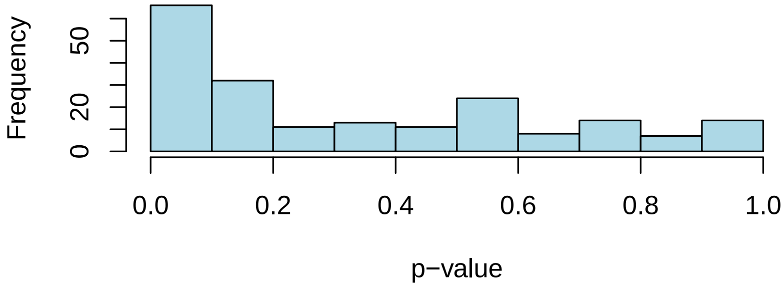
Logistic regression for failure of chip i is

$$p_i = \mathrm{p}(\texttt{fail}_i|\mathbf{x}_i) = e^{\alpha+\mathbf{x}_i\boldsymbol{\beta}}/(1+e^{\alpha+\mathbf{x}_i\boldsymbol{\beta}})$$

Semiconductor FDR Control

The full model has $R^2 = 0.56$ (based on *binomial* deviance).

The p-values for these 200 coefficients:



Some are clustered at zero, the rest sprawl out to one.

FDR of $q = 0.1$ yields $\alpha = 0.0122$ p-value rejection cut-off.

Implies 25 ‘significant’, of which approx 22-23 are true signals.

semiconductors

A *cut* model, using only these 25 signals, has $R^2_{cut} = 0.18$.
This is much smaller than the full model's $R^2_{full} = 0.56$.

In-Sample (IS) R^2 *always* increases with more covariates.
This is exactly what MLE $\hat{\beta}$ is fit to maximize.
But how well does each model predict *new* data?

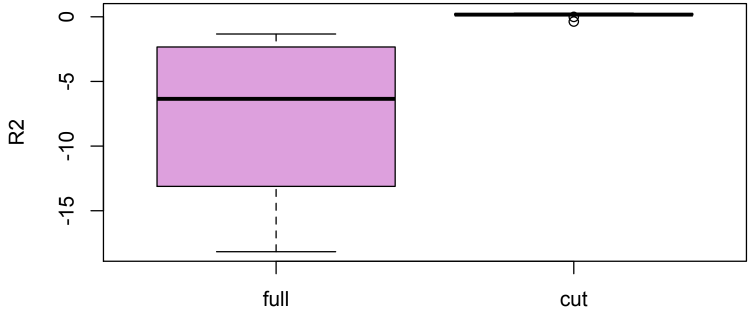
An out-of-sample (OOS) experiment

- split the data into 10 random subsets ('folds').
- Do 10x: fit model $\hat{\beta}$ using only 9/10 of data,
  and record R^2 on the left-out subset.

These OOS R^2 give us a sense of how well each
model can predict data that it has not already seen.

OOS experiment for semiconductor failure

We gain predictive accuracy by *dropping* variables.



Cut model has mean OOS R2 of 0.09, about 1/2 in-sample R2.

The full model is terrible. It is overfit and worse than $\bar{y}$.
Negative R2 are more common than you might expect.

OOS experimentation

All that matters is Out-of-Sample R^2.
We don't care about In Sample R^2.

Using OOS experiments to choose the best model is called *cross validation*. It will be a big part of our big data lives.

Selection of 'the best' model is at the core of all big data.

But before getting to selection, we first need strategies to build good sets of candidate models to choose amongst.

Naive forward stepwise regression

The `step()` function in R executes a common routine:

- Fit all univariate models. Choose that with highest (IS) R^2 and put that variable – say $x_{(1)}$ – in your model.
- Fit all bivariate models including $x_{(1)}$ ($y \sim \beta_{(1)} x_{(1)} + \beta_j x_j$), and add x_j from one with highest R^2 to your model.
- Repeat: max R^2 by adding one variable to your model.

You stop when some model selection rule (AIC) is lower for the current model than for any of the models that add one variable.

The problem with Subset Selection

`step()` is very slow (e.g., 90 sec for tiny semiconductors)

This is true in general with subset selection (SS):

Enumerate candidate models by applying maximum likelihood estimation for subsets of coefficients, with the rest set to zero.

SS is slow because adding one variable to a regression can change fit dramatically: *each model must be fit from scratch*.

A related subtle (but massively important) issue is stability.

MLEs have high *sampling variability*: they change a lot from one dataset to another. So which MLE model is 'best' changes a lot.

$\Rightarrow$ predictions based upon the 'best' model will be have high variance. And big variance leads to big expected errors.

Regularization

The key to contemporary statistics is regularization:

depart from optimality to stabilize a system.

Common in engineering: I wouldn't drive on an optimal bridge.

We minimize deviance

$$\min -\frac{2}{n}\log \mathsf{LHD}(\boldsymbol{\beta})$$

Regularization

The key to contemporary statistics is regularization:

depart from optimality to stabilize a system.

Common in engineering: I wouldn't drive on an optimal bridge.

We minimize deviance plus a cost on the size of coefficients.

$$\min -\frac{2}{n}\log \mathsf{LHD}(\boldsymbol{\beta}) + \lambda \sum_k |\beta_k|$$

This particular cost gives the 'lasso': the new least squares.

Decision theory: Cost in Estimation

Decision theory is based on the idea that choices have costs.
Estimation and hypothesis testing: what are the costs?

Estimation:

Deviance is the cost of distance between data and the model.
Recall: $\sum_i (y_i - \hat{y}_i)^2$ or $-\sum_i y_i \log(\hat{p}_i) - (1 - y_i) \log(1 - \hat{p}_i)$.

Testing:

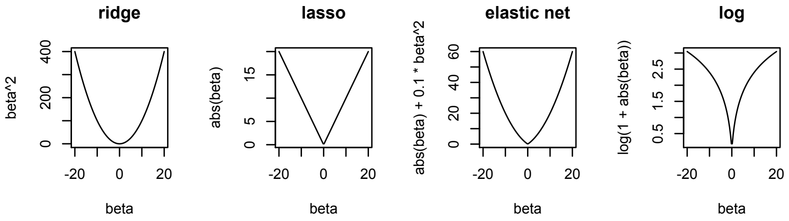
Since $\hat{\beta}_j = 0$ is *safe*, it should cost us to decide otherwise.

$\Rightarrow$ The cost of $\hat{\beta}$ is deviance plus a penalty away from zero.

[Sparse] Regularized Regression

$$\min \left\{ -\frac{2}{n} \log \text{LHD}(\boldsymbol{\beta}) + \lambda \sum_j c(\beta_j) \right\}$$

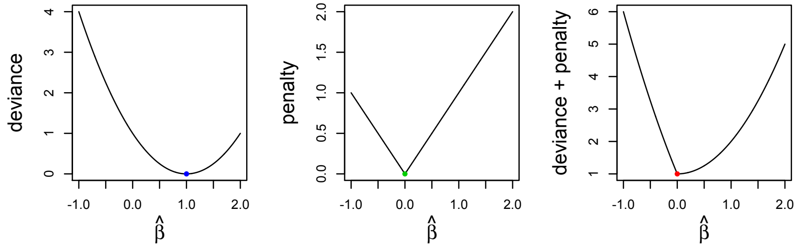
$\lambda > 0$ is the penalty weight, c is a cost (penalty) function.
$c(\beta)$ will be lowest at $\beta = 0$ and we pay more for $|\beta| > 0$.



Options: ridge β^2, lasso $|\beta|$, elastic net $\alpha\beta^2 + |\beta|$, $\log(1 + |\beta|)$.

Penalization can yield **automatic variable selection**

The minimum of a smooth + pointy function can be at the point.



Anything with an absolute value (e.g., lasso) will do this.

There are MANY penalty options and far too much theory.

Think of lasso as a baseline, and others as variations on it.

Lasso Regularization Paths

The lasso fits $\hat{\boldsymbol{\beta}}$ to minimize $-\frac{2}{n}\log \text{LHD}(\boldsymbol{\beta}) + \lambda \sum_j |\beta_j|$.

We'll do this for a *sequence* of penalties $\lambda_1 > \lambda_2 ... > \lambda_T$.

Then we can apply model selection tools to choose best $\hat{\lambda}$.

Path estimation:

Start with big λ_1 so big that $\hat{\boldsymbol{\beta}} = \mathbf{0}$.

For $t = 2 \ldots T$: update $\hat{\boldsymbol{\beta}}$ to be optimal under $\lambda_t < \lambda_{t-1}$.

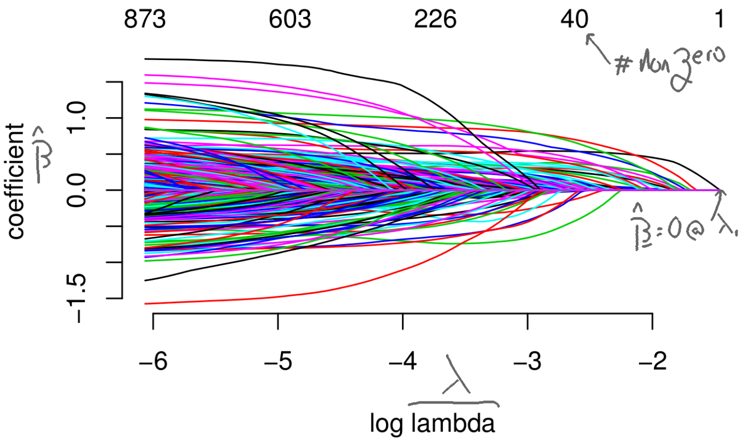
Since estimated $\hat{\boldsymbol{\beta}}$ changes smoothly along this path:

- It's fast! Each update is easy.
- It's stable: optimal λ_t may change a bit from sample to sample, but that won't affect the model much.

It's a better version of forward stepwise selection.

Path plots

The whole enterprise is easiest to understand visually.



The algorithm moves *right to left*.
The *y*-axis is $\hat{\boldsymbol{\beta}}$ (each line a different $\hat{\beta}_j$) as a function of λ_t.

Example: Comscore web browser data

The previous plot is household log-online-spend regressed onto % of time spent on various websites (each β_j a different site).

Comscore (available via WRDS) records info on browsing and purchasing behavior for annual panels of households.

I've extracted 2006 data for the 1000 most heavily trafficked websites and for 10,000 households that spent at least 1$.

Why do we care? Predict consumption from browser history.

e.g., to control for base-level spending, say, in estimating advertising effectiveness. You'll see browser history of users when they land, but likely not what they have bought.

Lasso Software

There are many packages for fitting lasso regressions in R.

`glmnet` is most common. `gamlr` is my contribution.
These two are very similar, and they share syntax.

Big difference is what they do beyond a simple lasso:.
  `glmnet` does an 'elastic net': $c(\beta) = |\beta| + \nu\beta^2$.
  `gamlr` does a 'gamma lasso': $c(\beta) \approx \log(\nu + |\beta|)$.

Since we stick mostly to lasso, they're nearly equivalent for us.
`gamlr` just makes it easier to apply some model selection rules.

Both use the `Matrix` library representation for sparse matrices.

Running a lasso

Once you have your `x` and `y`, running a lasso is easy.

```
spender <- gamlr(xweb, log(yspend))
plot(spender)  # nice path plot
spender$beta[c("mtv.com","zappos.com"),]
```

And you can do logistic lasso regression too

```
gamlr(x=SC[,-1], y=SC$FAIL,  family="binomial")
```

You should make sure that `y` is numeric $0/1$ here, not a factor.

Some common arguments

- `verb=TRUE` to get progress printout.
- `nlambda`: T, the length of your λ grid.
- `lambda.min.ratio`: λ_T/λ_1, how close to MLE you get.

See `?gamlr` for details and help.

Size Matters

Penalization means that scale matters. e.g., $x\beta$ has the same effect as $(2x)\beta/2$, but $|\beta|$ is twice as much penalty as $|\beta/2|$.

You can multiply β_j by $\mathrm{sd}(x_j)$ in the cost function to standardize.

That is, minimize $-\frac{2}{n}\log \mathrm{LHD}(\boldsymbol{\beta}) + \lambda \sum_j \mathrm{sd}(x_j)|\beta_j|$.

$\Rightarrow$ β_j's penalty is calculated per effect of 1SD change in x_j.

`gamlr` and `glmnet` both have `standardize=TRUE` by default.
You *only* use `standardize=FALSE` if you have good reason.

Regularization and Selection

The lasso minimizes $-\frac{2}{n}\log \mathrm{LHD}(\boldsymbol{\beta}) + \lambda \sum_j |\beta_j|$.

This ‘sparse regularization’ auto-selects the variables.

Sound too good to be true? You need to choose λ.

Think of $\lambda > 0$ as a signal-to-noise filter: *like squelch on a radio.*

We’ll use cross validation or information criteria to choose.

Path algorithms are key to the whole framework:

$\star$ They let us quickly enumerate a set of candidate models.

$\star$ This set is stable, so selected ‘best’ is probably pretty good.

Model Selection: **it is all about prediction.**

A recipe for model selection.

1. Find a manageable set of candidate models
   (i.e., such that fitting all models is fast).
2. Choose amongst these candidates the one with
   best predictive performance *on unseen data*.

1. is what the lasso paths provide.
2. Seems impossible! But it's not . . .

First, define predictive performance via 'deviance'.

Then, we need to *estimate* deviance for a fitted model applied to *new independent observations* from the true data distribution.

Out-of-sample prediction experiments

We already saw an OOS experiment with the semiconductors. Implicitly, we were estimating predictive deviance (via R^2).

The procedure of using such experiments to do model selection is called Cross Validation (CV). It follows a basic algorithm:

For $k = 1 \ldots K$,

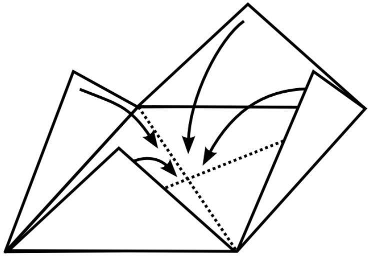
- Use a subset of $n_k < n$ observations to 'train' the model.
- Record the error rate for predictions from this fitted model on the left-out observations.

We'll usually measure 'error rate' as deviance. But alternatives include MSE, misclass rate, integrated ROC, or error quantiles.

You care about both average and spread of OOS error.

K-fold Cross Validation

One option is to just take repeated random samples.
It is better to 'fold' your data.



- Sample a random ordering of the data
  (important to avoid order dependence)
- Split the data into K folds: 1st 100/K%, 2nd 100/K%, etc.
- Cycle through K CV iterations with a single fold left-out.

This guarantees each observation is left-out for validation, and lowers the sampling variance of CV model selection.

Leave-one-out CV, with $K = n$, is nice but takes a long time.
$K = 5$ to 10 is fine in most applications.

CV Lasso

The lasso path algorithm minimizes $-\frac{2}{n}\log \mathrm{LHD}(\boldsymbol{\beta}) + \lambda_t \sum_j |\beta_j|$ over the sequence of penalty weights $\lambda_1 > \lambda_2 \ldots > \lambda_T$.

This gives us a path of T fitted coefficient vectors, $\hat{\boldsymbol{\beta}}_1 \ldots \hat{\boldsymbol{\beta}}_T$, each defining deviance for new data: $-\log \mathrm{p}(\mathbf{y}^{new} \mid \mathbf{X}^{new}\hat{\boldsymbol{\beta}}_t)$.

Set a sequence of penalties $\lambda_1 \ldots \lambda_T$.
Then, for each of $k = 1 \ldots K$ folds,

- Fit the path $\hat{\boldsymbol{\beta}}_1^k \ldots \hat{\boldsymbol{\beta}}_T^k$ on all data *except* fold k.
- Get fitted deviance *on left-out data*: $-\log \mathrm{p}(\mathbf{y}^k \mid \mathbf{X}^k\hat{\boldsymbol{\beta}}_t)$.

This gives us K draws of OOS deviance for each λ_t.

Finally, use the results to choose the 'best' $\hat{\lambda}$, then re-fit the model to *all of the data* by minimizing $-\frac{2}{n}\log \mathrm{LHD}(\boldsymbol{\beta}) + \hat{\lambda} \sum_j |\beta_j|$.

CV Lasso

Both `gamlr` and `glmnet` have functions to wrap this all up.
The syntax is the same; just preface with `cv.`

```
cv.spender <- cv.gamlr(xweb, log(yspend))
```

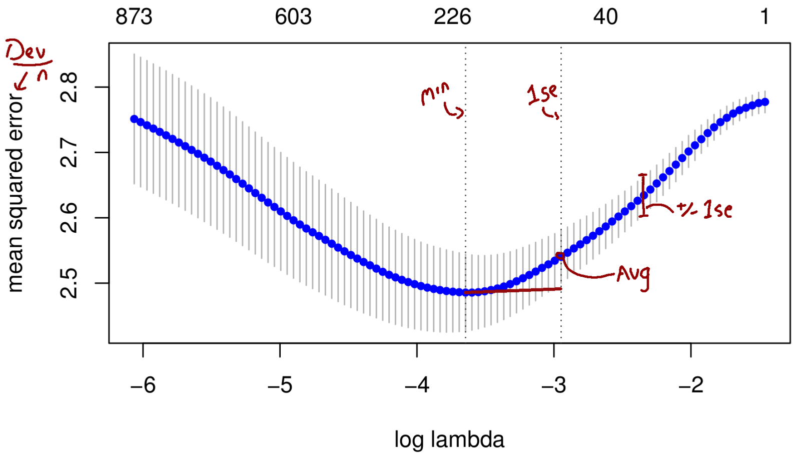
Then, `coef(cv.spender)` gives you $\hat{\boldsymbol{\beta}}_t$ at the ‘best’ λ_t

- `select="min"` gives λ_t with *min average OOS deviance*.
- `select="1se"` defines best as *biggest* λ_t *with average OOS deviance no more than 1SD away from the minimum.*

`1se` is default, and balances prediction against false discovery.
`min` is purely focused on predictive performance.

CV Lasso

Again, the routine is most easily understood visually.



Both selection rules are good; `1se` has extra bias for simplicity.

Alternatives to CV: **Information Criteria**

Many 'Information Criteria' out there: AICc, AIC, BIC, ...
These approximate distance between a model and 'the truth'.
You can apply them by choosing the model with minimum IC.

Most common is Akaike's $\text{AIC} = \text{Deviance} + 2df$.

df = 'degrees of freedom' used in your model fit.
For lasso and MLE, this is just the # of nonzero $\hat{\beta}_j$.

AIC overfits in high dimensions

The AIC is atually estimating OOS deviance: what your deviance would be on another *independent* sample of size n.

IS deviance is too small, since the model is tuned to this data. Some deep theory shows that IS - OOS deviance $\approx 2df$.

$\Rightarrow$ AIC $\approx$ OOS deviance.

Its common to claim this approx (i.e., AIC) is good for 'big n'.
Actually, its only good for big n/df.

In Big Data, df (# parameters) can be huge. Often $df \approx n$.
In this case the AIC will be a bad approximation: it overfits!

AIC corrected: AICc

AIC approximates OOS deviance, but does a bad job for big *df*.

In linear regression an improved approx to OOS deviance is

$$\text{AICc} = \text{Deviance} + 2df\,\mathbb{E}\left[\frac{\sigma^2}{\hat{\sigma}^2}\right] = \text{Deviance} + 2df\frac{n}{n - df - 1}$$

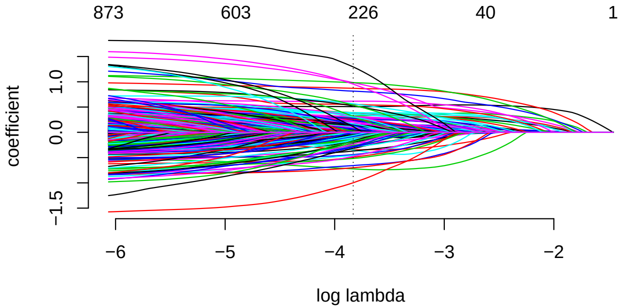
This is the corrected AIC, or AICc.

It also works nicely in logistic regression, or for any `glm`.

Notice that for big n/df, AICc $\approx$ AIC. So *always* use AICc.

gamlr uses AICc

It's marked on the path plot



And it is the default for `coef.gamlr`

```
B <- coef(spender)[-1,]
B[c(which.min(B),which.max(B))]
    cursormania.com shopyourbargain.com
          -0.998143            1.294246
```

Another option: **Bayes IC**

The BIC is Deviance + $\log(n) \times df$.

This *looks* just like AIC, but comes from a very different place.

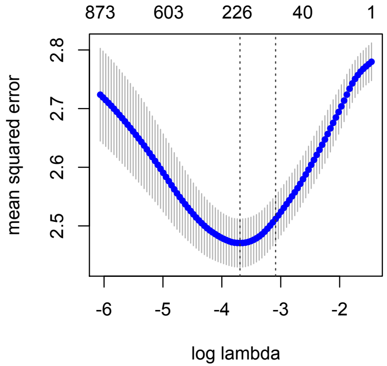
$BIC \approx -\log \mathrm{p}(M_b|\mathrm{data})$, the 'probability that model b is true'.

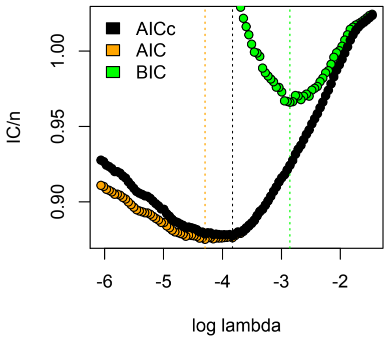
$$\mathrm{p}(M_b|\mathrm{data}) = \frac{\mathrm{p}(\mathrm{data}, M_b)}{\mathrm{p}(\mathrm{data})} \propto \underbrace{\mathrm{p}(\mathrm{data}|M_b)}_{\text{LHD}} \underbrace{\mathrm{p}(M_b)}_{\text{prior}}$$

The 'prior' is your probability that a model is true *before* you saw any data. BIC uses a 'unit-info' prior: $\mathrm{N}\left[\hat{\beta}, \frac{2}{n}\mathrm{var}(\hat{\beta})^{-1}\right]$

AIC[c] tries to approx OOS deviance.
BIC is trying to get at the 'truth'.

IC and CV on the Comscore Data

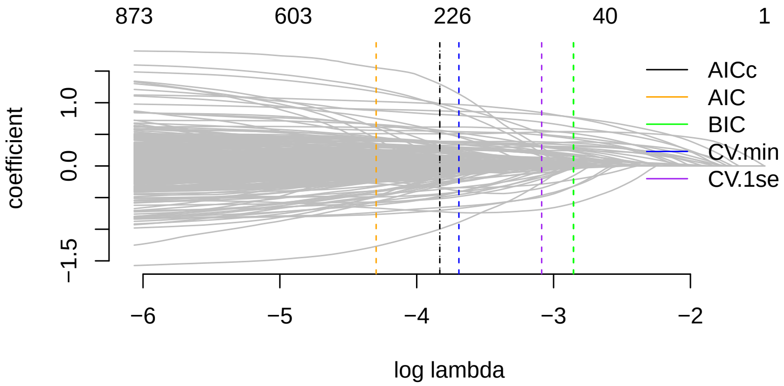




The take home message: AICc curve looks like CV curve.

In practice, BIC works more like the 1se CV rule.
But with big *n* it chooses too simple models (it underfits).

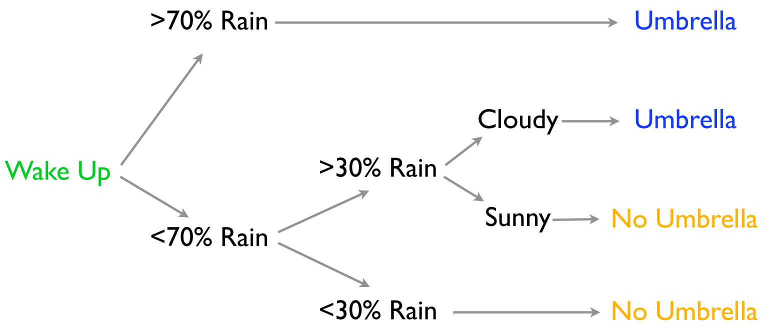
IC and CV on the Comscore Data



With all of these selection rules, you get a range of answers.
If you have time, do CV. But AICc is fast and stable.
If you are worried about false discovery, tend towards BIC/1se.

Decision Trees

Trees are great: nonlinearity, deep interactions, heteroskedasticity.



The ‘optimal’ decision tree is a statistic we care about (s.w.c.a).

CART: greedy growing with optimal splits

Given node $\{\mathbf{x}_i, y_i\}_{i=1}^n$ and DGP weights $\boldsymbol{\theta}$, find x to minimize

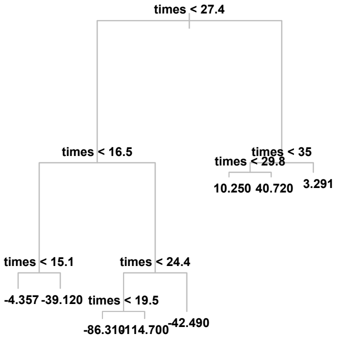
$$
\begin{aligned}
|\boldsymbol{\theta}|\sigma^2(x, \boldsymbol{\theta}) = & \sum_{k \in \mathrm{left}(x)} \theta_k (y_k - \mu_{\mathrm{left}(x)})^2 \\
& + \sum_{k \in \mathrm{right}(x)} \theta_k (y_k - \mu_{\mathrm{right}(x)})^2
\end{aligned}
$$

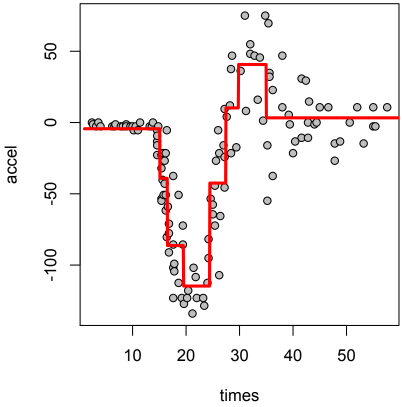
for a regression tree. Classification impurity can be Gini, etc.

Trees are awesome

They automatically learn non-linear response functions and will discover interactions between variables.

Example: Motorcycle Crash Test Dummy Data
x is time from impact, y is acceleration on the helmet.





Unfortunately, deep tree structure is so unstable that optimal depth is not easily chosen via cross validation

- tree predictions are non-smooth wrt the tuning parameter (tree depth).
- Breiman 2001: Cross-validation fails for such tuning

Instead, we can average over a bootstrapped sample of trees:

- repeatedly re-sample the data, with-replacement,
  to get a 'jittered' dataset of n observations.
- for each resample, fit a CART tree.
- when you want to predict y for some $\mathbf{x}$,
  take the average prediction from this forest of trees.

Real structure that persists across datasets shows up in the average. Noisy useless signals will average out to have no effect.

This is a Random Forest

Random Forests

- Sample B subsets of the data + variables:
  e.g., observations $1, 5, 20, \dots$ and inputs $2, 10, 17, \dots$
- Fit a tree to each subset, to get B fitted trees is $\mathcal{T}_b$.
- Average prediction across trees:
  - for regression average $\mathbb{E}[y|\mathbf{x}] = \frac{1}{B}\sum_{b=1}^{B} \mathcal{T}_b(\mathbf{x})$.
  - for classification let $\{\mathcal{T}_b(\mathbf{x})\}_{b=1}^{B}$ vote on $\hat{y}$.

The observation resample is usually *with-replacement*, so that this is taking the *average of bootstrapped trees* (i.e., 'bagging')

Understanding Random Forests

Recall how CART is used in practice.

- Split to lower deviance until leaves hit minimum size.
- Create a set of candidate trees by pruning back from this.
- Choose the best among those trees by cross validation.

Random Forests avoid the need for CV.

Each tree 'b' is not overly complicated because
you only work with a limited set of variables.

Your predictions are not 'optimized to noise' because
they are averages of trees fit to many different subsets.

RFs are a great go-to model for nonparametric prediction.

Deeper: Distribution-free Bayesian nonparametrics

Find some *statistic of the DGP* that you care about:

- derive from first principles, e.g. moment conditions
- *an algorithm* that we know works, e.g. CART
- think about geometric projections, e.g. OLS

Call this statistic $\theta(g)$ where $g(\mathbf{z})$ is the DGP (e.g., for $\mathbf{z} = [\mathbf{x}, y]$).

Then you write down a flexible model for the DGP g, and study properties of the posterior on $\theta(g)$ induced by the posterior over g.

A flexible model for the DGP

Say $\mathbf{z} = [\mathbf{x}, y]$ is a single *independent* data point.

Each data point assumes one of a *finite* number of possible values, $[\boldsymbol{\zeta}_1 \dots \boldsymbol{\zeta}_L]$, with probabilities proportional to $[\boldsymbol{\theta}_1 \dots \boldsymbol{\theta}_L]$.

$$g(\mathbf{z}) = \frac{1}{|\boldsymbol{\theta}|} \sum_{l=1}^{L} \theta_l \mathbb{1}_{[\mathbf{z}=\zeta_l]}$$

We complete specification with a conjugate prior on the weights:

$$\frac{\boldsymbol{\theta}}{|\boldsymbol{\theta}|} \sim \mathrm{Dir}(a) \propto \frac{1}{|\boldsymbol{\theta}|^{L(a-1)}} \prod_l \theta_l^{a-1} \quad \text{where} \quad a, \theta_l > 0.$$

This is the Dirichlet-multinomial sampling model (Ferguson 1973).

Now you've observed some data, say $\mathbf{Z} = \{\mathbf{z}_1 \dots \mathbf{z}_n\}$.
(say every $\mathbf{z}_i = [\mathbf{x}_i, y_i]$ is unique).

The posterior over weights has $\theta_l \overset{ind}{\sim} \mathrm{Exp}\left(a + \mathbb{1}_{[\zeta_l \in \mathbf{Z}]}\right)$.

A convenient limiting case

$a \to 0$ leads to $\mathrm{p}(\theta_l = 0) = 1$ for $\zeta_l \notin \mathbf{Z}$.

In this case, we can focus on only the *observed support* and write the posterior for our DGP

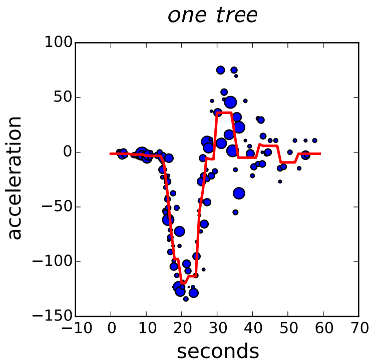
$$g(\mathbf{z}) = \tfrac{1}{|\boldsymbol{\theta}|} \sum_{i=1}^{n} \theta_i \mathbb{1}[\mathbf{z} = \mathbf{z}_i], \quad \theta_i \overset{iid}{\sim} \mathrm{Exp}(1).$$

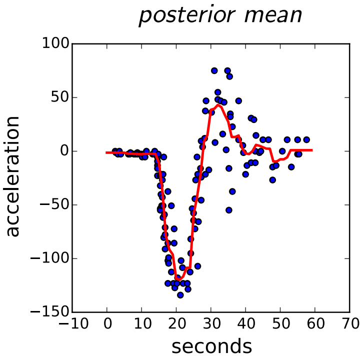
This is just the Bayesian bootstrap. (Rubin 1981)

Bayesian Forests: a posterior for CART trees

For $b = 1 \dots B$:

- draw $\boldsymbol{\theta}^b \overset{iid}{\sim} \mathrm{Exp}(\mathbf{1})$
- run weighted-sample CART to get $\mathcal{T}_b = \mathcal{T}(\boldsymbol{\theta}^b)$





Random Forest ≈ Bayesian forest ≈ posterior over CART fits.

Random Forests in R

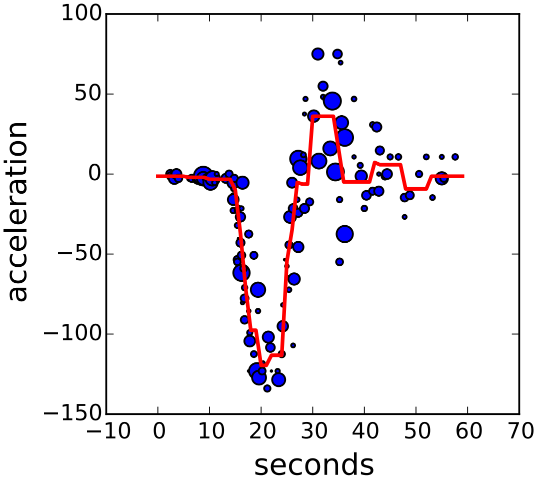
R has the `randomForest` package,
which works essentially the same as `tree`

```
   spamrf <- randomForest(spam ~ ., data=xemail)
```

For big datasets, use $\mathtt{x} = \mathtt{x}, \ \mathtt{y} = \mathtt{y}$ like in `gamlr`.

Unfortunately, you lose the interpretability of a single tree.
However, if you set `importance=TRUE`, `Random Forest` will evaluate each $\mathcal{T}_b$'s performance on the *left-out sample* (recall each tree is fit on a sub-sample). This yields nice OOS stats.

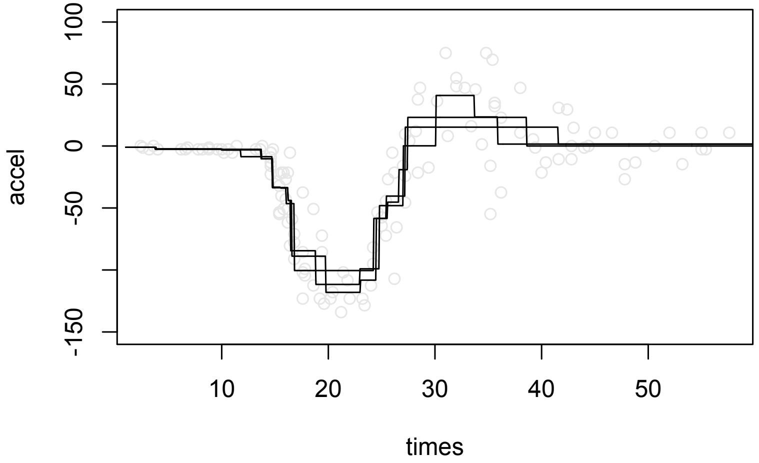
They can be slow (due to many tree fits) but
they can also be fit in parallel or on distributed data...



Fitting CART to sampled-with-replacement data is equivalent to randomly weighting your observations in the deviance calculations.

(size proportional to weight in this picture).

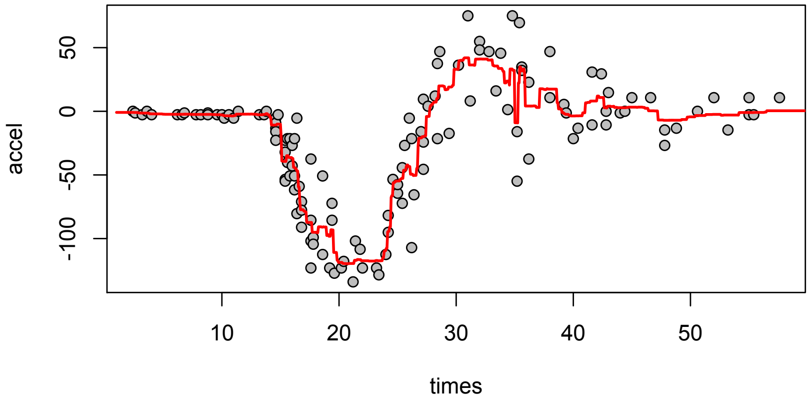
Random Trees for the Motorcycle Data



If you fit to random subsets of the data,

you get a slightly different tree each time.

Model Averaging with Random Forests



Averaging many trees yields a single response surface.

Still looks like a bit of overfit to me, which remains a danger.

A larger example: California Housing Data

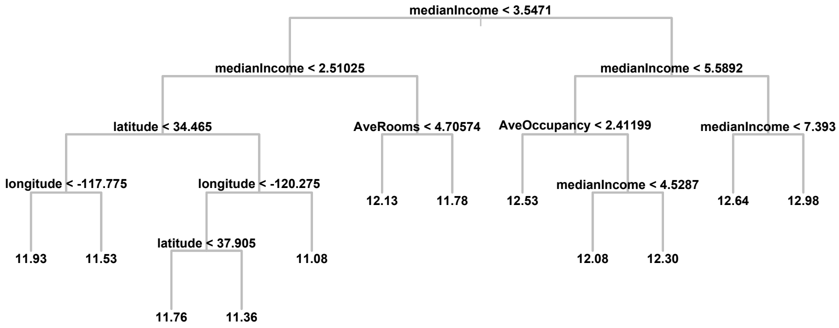
Median home values in census tracts, along with

- Latitude and Longitude of tract centers.
- Population totals and median income.
- Average room/bedroom numbers, home age.

The goal is to predict log(MedVal) for census tracts.

Difficult regression: Covariate effects change with location.
How they change is probably not linear.

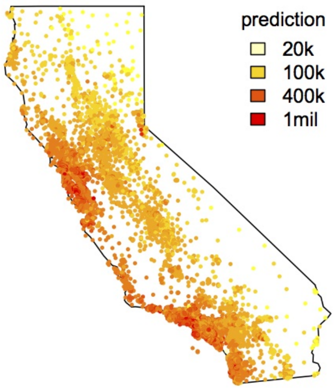
CART Dendrogram for CA housing

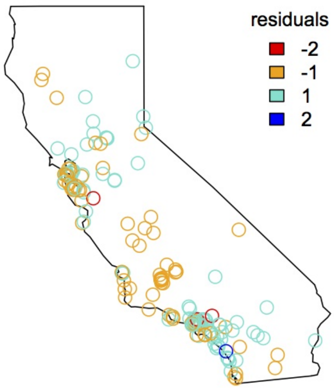


Income is dominant, with location important for low income.

Cross Validation favors the most complicated tree: 12 leaves.

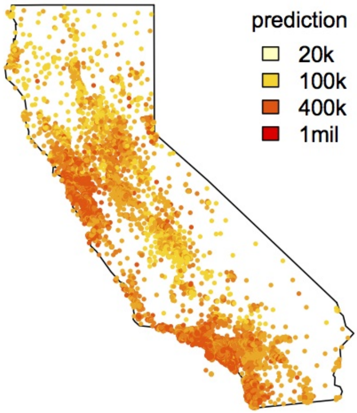
LASSO fit for CA housing data

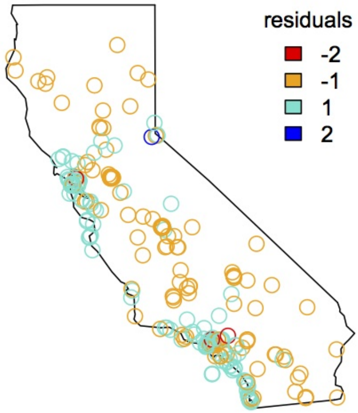




Looks like over-estimates in the Bay, under-estimates in OC.

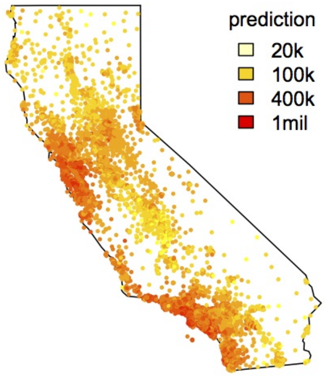
CART fit for CA housing data

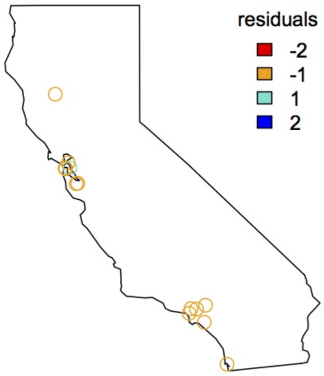




Under-estimating the coast, over-estimating the central valley?

randomForest fit for CA housing data

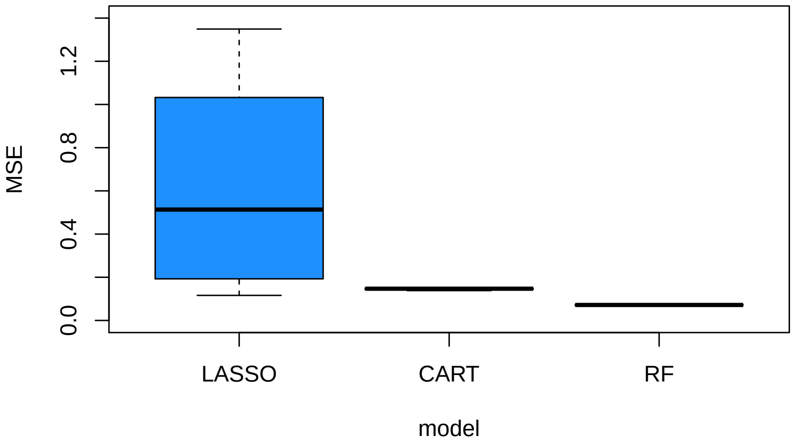




No big residuals! (although still missing the LA and SF effects)

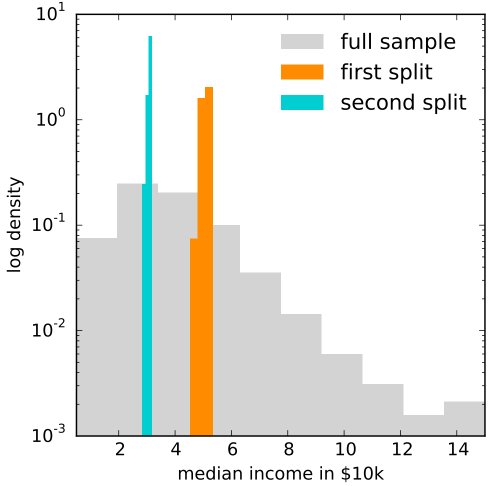
Overfit? From out-of-sample prediction it appears not.

CA housing: out-of-sample prediction



Trees outperform LASSO: gain from nonlinear interaction.
RF is better still than CART: benefits of model averaging.

Aside: trunks are stable (Taddy+ icml 2015)



- sample tree occurs 62% of the time.
- 90% of trees split on income twice, and then latitude.
- 100% of trees have 1st 2 splits on median income.

Empirically and theoretically: trees are stable, at the trunk.

Random Forest bottlenecks

RFs (or BFs) are awesome, but when the data are too big to fit in memory or on a single machine they get extremely expensive.
(e.g., Google PLANET, Panda 2009).

A common solution is a 'sub-sampling forest': instead of drawing with-replacement, or re-weighting, draw $m \ll n$ sub-samples.

This defeats the whole purpose of trees: these are rules that are designed to grow in complexity with the amount of data available.
(that's why we bother storing so much data!)

If you starve the individual trees of data you lose.

Empirical Bayesian Forests (EBF)

RFs are expensive. Sub-sampling hurts bad.

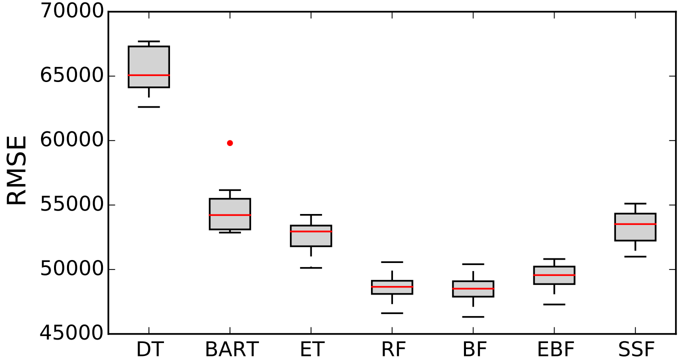
Instead:

- fit a single tree to a shallow trunk.
- Map data to each branch.
- Fit a full forest on the smaller branch datasets.

Since the trunks are all similar for each tree in a full forest, our EBF looks nearly the same at a fraction of computational cost.

It's an updated version of classical Empirical Bayes:
use plug-in estimates at high levels in a hierarchical model,
focus effort at full Bayesian learning for the the hard bits.

OOS predictive performance on California Housing



Here EBF and BF give nearly the same results. *SSF does not.*

EBFs crunch more data faster without hurting performance.

Roundup on Tree-based learning

We've seen two techniques for building tree models.

- CART: recursive partitions, pruned back by CV.
- randomForest: average many simple CART trees.

There are many other tree-based algorithms.

- Boosted Trees: repeatedly fit simple trees to residuals.
  Fast, but it is tough to avoid over-fit (requires full CV).
- Bayes Additive Regression Trees: mix many simple trees.
  Robust prediction, but suffers with non-constant variance.

Trees are poor in high dimension, but fitting them to low dimension factors (principle components) is a good option.

Roundup on Nonlinear Regression and Classification

Many other *semiparametric learning* algorithms

- Neural Networks (and deep learning):
  many recursive logistic regressions.
- Support Vector Machines:
  Project to HD, then classify.
- Gaussian Processes, splines, wavelets, etc:
  Use sums of curvy functions in regression.

Some of these are great, but all take a ton of tuning.
Lots of work on automating this tuning for deep nets

For now, nothing's better out-of-the-box in low-D than forests.
But: a lasso will tend to work better in high-D