

# SOCIAL DATA SCIENCE

## STATISTICAL LEARNING

---

Sebastian Barfort

August 15, 2016

University of Copenhagen  
Department of Economics

**Cross validation:** Split data in test and training data. Train model on training data, test it on test data

**Supervised Learning:** Models designed to infer a relationship from **labeled** training data.

- linear model selection (OLS, Ridge, Lasso)
- Classification (logistic, KNN, CART)

**Unsupervised Learning:** Models designed to infer a relationship from **unlabeled** training data.

- PCA

# Cross Validation

Statistical learning models are designed to minimize **out of sample error**: the error rate you get on a new data set

Key ideas

- Out of sample error is what you care about
- In sample error  $<$  out of sample error
- The reason is overfitting (matching your algorithm to the data you have)

## OUT OF SAMPLE ERROR (CONTINUOUS VARIABLES)

Mean squared error (MSE):

$$\frac{1}{n} \sum_{i=1}^n (\text{prediction}_i - \text{truth}_i)^2$$

Root mean squared error (RMSE):

$$\sqrt{\frac{1}{n} \sum_{i=1}^n (\text{prediction}_i - \text{truth}_i)^2}$$

**Question:** what is the difference?

```
library("readr")
gh.link = "https://raw.githubusercontent.com/"
user.repo = "johnmyleswhite/ML_for_Hackers/"
branch = "master/"
link = "05-Regression/data/longevity.csv"
data.link = paste0(gh.link, user.repo, branch, link)
df = read_csv(data.link)
```

---

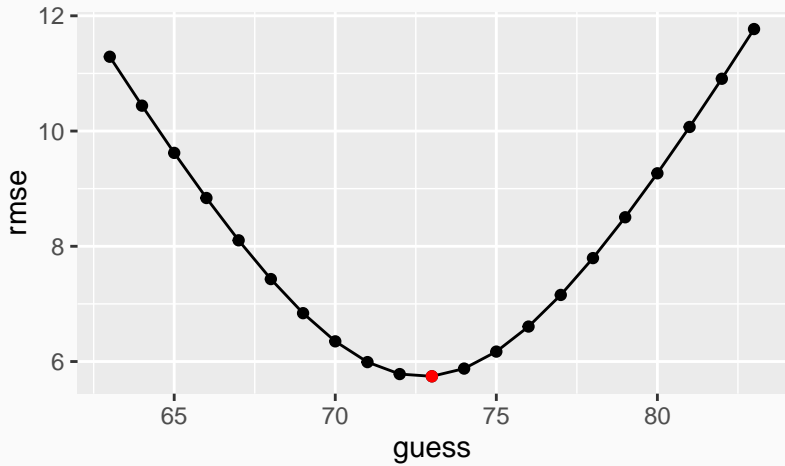
Smokes	AgeAtDeath
1	75
1	72
1	66
1	74
1	69
1	65

---

Let's look at RMSE for different guesses of age of death

```
add_rmse = function(i){  
  df %>%  
    mutate(sq.error = (AgeAtDeath - i)^2) %>%  
    summarise(mse = mean(sq.error),  
              rmse = sqrt(mse),  
              guess = i)  
}  
  
df.rmse = 63:83 %>%  
  map_df(add_rmse)
```





```
df.rmse %>%  
  filter(rmse == min(rmse))
```

```
## # A tibble: 1 x 3  
##       mse      rmse guess  
##   <dbl>    <dbl> <int>  
## 1 32.991 5.743779     73
```

```
df %>%  
  summarise(round(mean(AgeAtDeath), 0))
```

```
## # A tibble: 1 x 1  
##   round(mean(AgeAtDeath), 0)  
##                               <dbl>  
## 1                               73
```

## OUT OF SAMPLE ERROR (DISCRETE VARIABLES)

One simple way to assess model accuracy when you have discrete outcomes (republican/democrat, professor/student, etc) could be the mean classification error

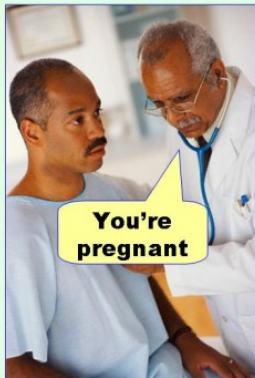
$$\text{Ave}(I(y_0 \neq \hat{y}_0))$$

But assessing model accuracy with discrete outcomes is often not straightforward.

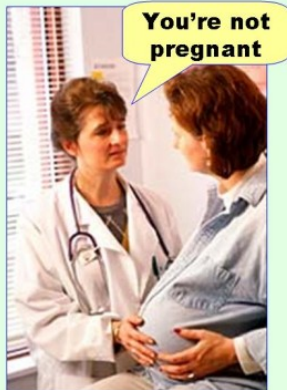
Alternative: ROC curves

## TYPE 1 AND TYPE 2 ERRORS

**Type I error**  
(false positive)



**Type II error**  
(false negative)



Accuracy on the training set (resubstitution accuracy) is optimistic

A better estimate comes from an independent set (test set accuracy)

But we can't use the test set when building the model or it becomes part of the training set

So we estimate the test set accuracy with the training set

Remember the bias-variance tradeoff

Why not just randomly divide the data into a test and training set?

Two drawbacks

1. The estimate of the RMSE on the test data can be highly variable, depending on precisely which observations are included in the test and training sets
2. In this approach, only the training data is used to fit the model. Since statistical models generally perform worse when trained on fewer observations, this suggests that the RMSE on the test data may actually be too large

One very useful refinement of the test-training data approach is **cross-validation**

## K-FOLD CROSS VALIDATION

1. Divide the data into  $k$  roughly equal subsets and label them  $s = 1, \dots, k$ .
2. Fit your model using the  $k - 1$  subsets other than subset  $s$
3. Predict for subset  $s$  and calculate RMSE
4. Stop if  $s = k$ , otherwise increment  $s$  by 1 and continue

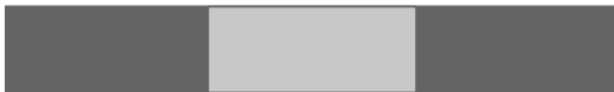
The  $k$  fold CV estimate is computed by averaging the mean squared errors ( $MSE_1, \dots, MSE_k$ )



$$CV_k = \frac{1}{k} \sum_{i=1}^k MSE_i$$

Common choices for  $k$  are 10 and 5.

CV can (and should) be used both to find tuning parameters and to report goodness-of-fit measures.





 Testing  
 Training

```
true_model = function(x){  
  2 + 8*x^4 + rnorm(length(x), sd = 1)  
}
```

```
df = data_frame(  
  x = seq(0, 1, length = 50),  
  y = true_model(x)  
)
```

---

x	y
0.0000000	1.497808
0.0204082	2.131533
0.0408163	1.921105
0.0612245	2.886897
0.0816327	2.117326
0.1020408	2.319497

---

We want to search for the correct model using a series of polynomials of different degrees.

```
my_model = function(pol, data = df){  
  lm(y ~ poly(x, pol), data = data)  
}
```

```
model.1 = my_model(pol = 1)
```

Table 3:

---

	y
poly(x, pol)	13.624*** (1.349)
Constant	3.731*** (0.191)
N	50
R-squared	0.680
Adj. R-squared	0.673
Residual Std. Error	1.349 (df = 48)
F Statistic	101.968*** (df = 1; 48)

---

\*\*\*p < .01; \*\*p < .05; \*p < .1

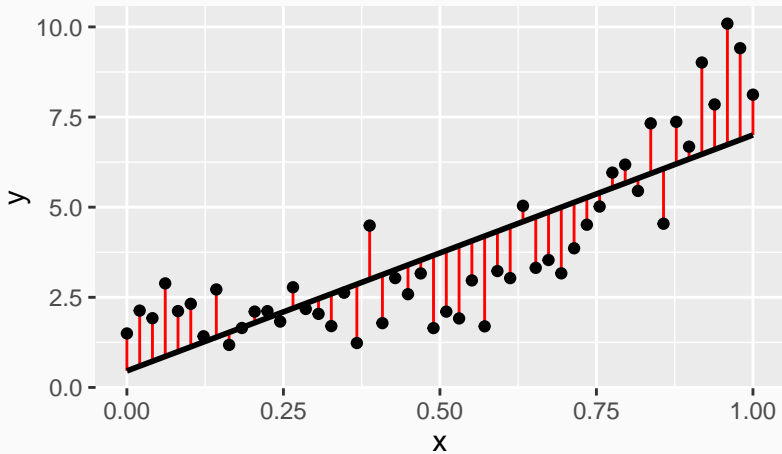
```
add_pred = function(mod, data = df){  
  data %>% add_predictions(mod, var = "pred")  
}  
  
df.1 = add_pred(model.1)
```



---

x	y	pred
0.0000000	1.497808	0.4594252
0.0204082	2.131533	0.5929425
0.0408163	1.921105	0.7264597
0.0612245	2.886897	0.8599769
0.0816327	2.117326	0.9934941
0.1020408	2.319497	1.1270114

---

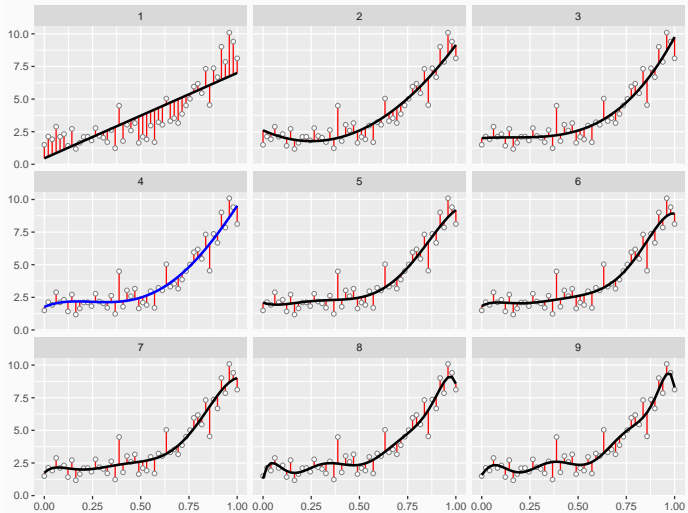


```
# Estimate polynomial from 1 to 9
models = 1:9 %>%
  map(my_model) %>%
  map_df(add_pred, .id = "poly")
```

---

poly	x	y	pred
1	0.0000000	1.497808	0.4594252
1	0.0204082	2.131533	0.5929425
1	0.0408163	1.921105	0.7264597
1	0.0612245	2.886897	0.8599769
1	0.0816327	2.117326	0.9934941
1	0.1020408	2.319497	1.1270114

---



```
models.rmse = models %>%  
  mutate(error = y - pred,  
          sq.error = error^2) %>%  
  group_by(poly) %>%  
  summarise(  
    mse = mean(sq.error),  
    rmse = sqrt(mse)  
  ) %>%  
  arrange(rmse)
```

Which model is best?

poly	rmse
9	0.7285253
8	0.7437169
7	0.7751253
6	0.7760149
5	0.7838361
4	0.7949836
3	0.8007723
2	0.8410190
1	1.3219606

```
gen_crossv = function(pol, data = df){  
  data %>%  
    crossv_kfold(10) %>%  
    mutate(  
      mod = map(train, ~ lm(y ~ poly(x, pol),  
                           data = .)),  
      rmse.test = map2_dbl(mod, test, rmse),  
      rmse.train = map2_dbl(mod, train, rmse)  
    )  
}
```



```
set.seed(3000)
df.cv = 1:10 %>%
  map_df(gen_crossv, .id = "degree")
```

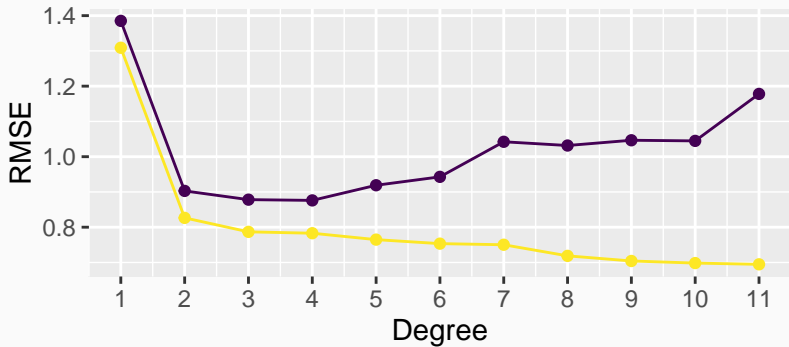
```
## # A tibble: 5 x 5
##   degree  .id      mod rmse.test rmse.train
##   <chr> <chr>   <list>   <dbl>     <dbl>
## 1     1    01 <S3: lm> 1.4698002 1.310390
## 2     1    02 <S3: lm> 1.4291296 1.312430
## 3     1    03 <S3: lm> 1.4155343 1.311204
## 4     1    04 <S3: lm> 1.4696419 1.310855
## 5     1    05 <S3: lm> 0.7654701 1.371687
```

```
df.cv.sum = df.cv %>%  
  group_by(degree) %>%  
  summarise(  
    m.rmse.test = mean(rmse.test),  
    m.rmse.train = mean(rmse.train)  
  )
```

---

degree	var	value
1	m.rmse.test	1.3691875
1	m.rmse.train	1.3146356
2	m.rmse.test	0.8334279
2	m.rmse.train	0.8376921
3	m.rmse.test	0.8898723
3	m.rmse.train	0.7949651
4	m.rmse.test	0.8644459

---



# Supervised Learning

**Supervised learning:** Models designed to infer a relationship from *labeled* training data.

**Labelled data:** For each observation of the predictor variables,  $x_i, 1, \dots, n$  there is an associated response measurement  $y_i$

- When the response measurement is discrete: **classification**
- when the response is continuous: **regression**

The problem with overfitting comes from our model being too complex

**Complexity:** models are complex when the number or size of the coefficients is large

One approach: punish the model for doing this

This approach is called **regularization**



Two popular models build on this approach: Ridge and Lasso

The approach is similar: include a **loss function** in the OLS minimization problem to prevent overfitting

$$\sum_{i=1}^n (y_i - \beta_0 - \sum_{j=1}^p b_j x_{ij})^2 + \text{LOSS}$$

- Ridge uses the L2 norm:  $\alpha \sum_{j=1}^p \beta_j^2$
- Lasso uses the L1 norm:  $\alpha \sum_{j=1}^p |\beta_j|$

This turns out to be very important

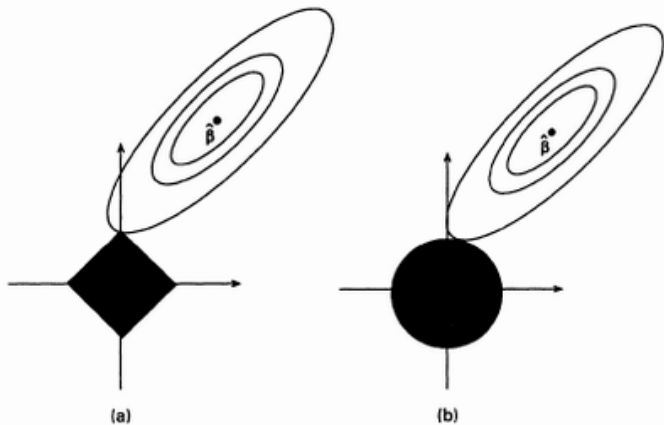


Fig. 2. Estimation picture for (a) the lasso and (b) ridge regression

L1 regularization gives you sparse estimates (and therefore performs some form of variable selection)

```
lm.fit = my_model(pol = 1)
l2.norm = sum(coef(lm.fit)^2)
l1.norm = sum(abs(coef(lm.fit)))
print(paste0("l2.norm is ", l2.norm))
```

```
## [1] "l2.norm is 199.539437019912"
```

```
print(paste0("l1.norm is ", l1.norm))
```

```
## [1] "l1.norm is 17.3549167871978"
```

Regularization methods are implemented in R in the `glmnet` package (although it might also be worth checking out the newer `caret` and `mlr` packages)

```
library("glmnet")
```

`alpha` controls the norm.

`alpha = 1` is the Lasso penalty,

`alpha = 0` is Ridge

```
x = poly(df$x, 9)
y = df$y

out = glmnet(x, y)
```

The output contains three columns

- **Df**: tells you how many coefficients in the model ended up being nonzero
- **%Dev**: essentially a R2 for the model
- **Lambda**: the loss parameter

Because **Lambda** controls the values that we get from the model, it is often referred to as a *hyperparameter*

Large **Lambda**: heavy penalty for model complexity

## PICKING $\lambda$

Which  $\lambda$  minimizes RMSE in our test data?

```
cal_rmse = function(prediction, truth){  
  return(sqrt(mean( (prediction - truth) ^2)))  
}
```

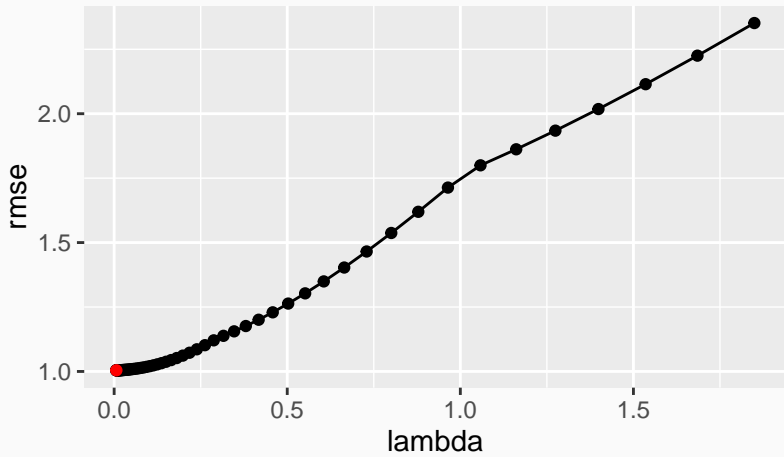
```
performance = function(i){  
  prediction = predict(glmnet.fit,  
                       poly(test.df$x, 9),  
                       s = i)  
  
  truth = test.df$y  
  RMSE = cal_rmse(prediction, truth)  
  return( data.frame(lambda = i,  
                     rmse = RMSE))  
}
```

Create test and training data

```
n = nrow(df)
indices = sort(sample(1:n, round(.5*n)))
training.df = df[indices, ]
test.df = df[-indices, ]
glmnet.fit = glmnet(poly(training.df$x, 9),
                    training.df$y)
lambdas = glmnet.fit$lambda

perf.df = lambdas %>%
  map_df(performance)
```





```
best.lambda = perf.df %>%  
  filter(lambda == min(lambda))  
glmnet.fit = glmnet(poly(df$x, 9), df$y)
```

```
coef(glmnet.fit, s = best.lambda$lambda)
```

```
## 10 x 1 sparse Matrix of class "dgCMatrix"  
##                1  
## (Intercept)  3.597608  
## 1            411.377377  
## 2            225.909396  
## 3             66.361665  
## 4             8.249294  
## 5              .  
## 6              .  
## 7              .  
## 8              .  
## 9              .
```

The Lasso model ended up using only 4 nonzero coefficients even though the model had the ability to use 9

Selecting a simpler model when more complicated models are possible is exactly the point of regularization

# Classification

When we are trying to predict discrete outcomes we are effectively doing classification

We saw this yesterday with the logit example

Now a different approach: **Classification and Regression Trees**

Decision trees can be applied to both regression and classification problems

They are intuitive, but run the danger of overfitting (what happens if you grow the largest possible decision tree for a given problem?)

Therefore, people usually use extensions such as random forests

---

**Algorithm 8.1** *Building a Regression Tree*

---

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.
  2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of  $\alpha$ .
  3. Use K-fold cross-validation to choose  $\alpha$ . That is, divide the training observations into  $K$  folds. For each  $k = 1, \dots, K$ :
    - (a) Repeat Steps 1 and 2 on all but the  $k$ th fold of the training data.
    - (b) Evaluate the mean squared prediction error on the data in the left-out  $k$ th fold, as a function of  $\alpha$ .Average the results for each value of  $\alpha$ , and pick  $\alpha$  to minimize the average error.
  4. Return the subtree from Step 2 that corresponds to the chosen value of  $\alpha$ .
-



## **Advantages**

Easy to explain

Mimic the mental model we often use to make decisions

Can be displayed graphically

## **Main disadvantage**

Performance

```
library("jsonlite")  
food = fromJSON("~/git/sds_summer/data/food.json")
```

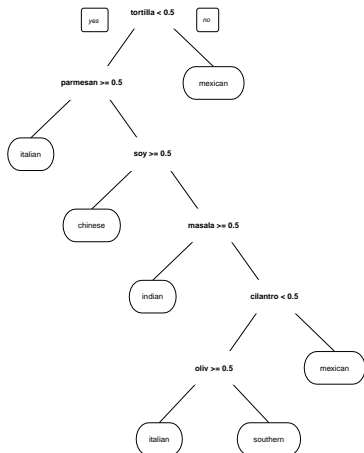
```
food$ingredients = lapply(food$ingredients,  
                           FUN=tolower)  
food$ingredients = lapply(food$ingredients,  
                           FUN=function(x)  
                             gsub("-", "_", x))  
food$ingredients = lapply(food$ingredients,  
                           FUN=function(x)  
                             gsub("[^a-z0-9_ ]", "", x))
```

```
library("tm")
combi_ingredients = c(Corpus(VectorSource(food$ingredient))
                     Corpus(VectorSource(food$ingredient)))
combi_ingredients = tm_map(combi_ingredients, stemDocument,
                           language="english")
combi_ingredientsDTM = DocumentTermMatrix(combi_ingredients)
combi_ingredientsDTM = removeSparseTerms(combi_ingredientsDTM)
combi_ingredientsDTM = as.data.frame(
  as.matrix(combi_ingredientsDTM))
combi = combi_ingredientsDTM
combi_ingredientsDTM$cuisine = as.factor(
  c(food$cuisine, rep("italian", nrow(food))))
```

```
trainDTM = combi_ingredientsDTM[1:nrow(food), ]  
testDTM = combi_ingredientsDTM[-(1:nrow(food)), ]
```

```
library("rpart")
set.seed(1)
model = rpart(cuisine ~ ., data = trainDTM,
              method = "class")
cuisine = predict(model, newdata = testDTM,
                  type = "class")
```

# DECISION TREE



Random Forest algorithms are so-called ensemble models  
This means that the model consists of many smaller models  
The sub-models for Random Forests are classification and regression trees



Breiman (1996) proposed bootstrap aggregating – “bagging” – to reduce the risk of overfitting.

The core idea of bagging is to decrease the variance of the predictions of one model, by fitting several models and averaging over their predictions

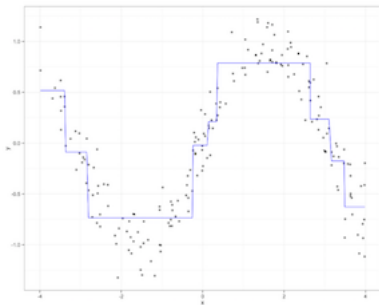
In order to obtain a variety of models that are not overfit to the available data, each component model is fit only to a bootstrap sample of the data

Random forests extended the logic of bagging to predictors.

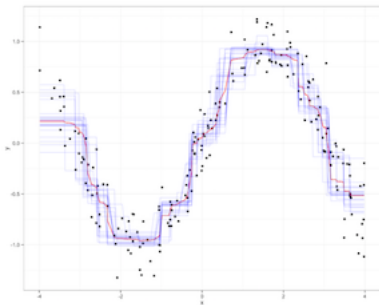
This means that, instead of choosing the split from among all the explanatory variables at each node in each tree, only a random subset of the explanatory variables are used

If there are some very important variables they might overshadow the effect of weaker predictors because the algorithm searches for the split that results in the largest reduction in the loss function.

If at each split only a subset of predictors are available to be chosen, weaker predictors get a chance to be selected more often, reducing the risk of overlooking such variables



(a) Approximation of the function using CART. The blue line displays the prediction from the tree.



(b) 25 randomly selected trees (shown in blue) in a Random Forest (prediction shown in red).

Jones & Linder. 2015.

# Unsupervised Learning

## Supervised

You have an outcome  $Y$  and some covariates  $X$

## Unsupervised

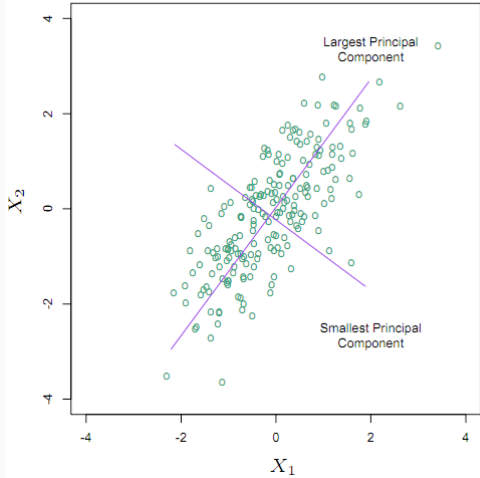
You have a bunch of observations  $X$  and you want to understand the relationships between them.

You are usually trying to understand patterns in  $X$  or group the variables in  $X$  in some way

You have a set of multivariate variables  $X_1, \dots, X_p$

- Find a new set of multivariate variables that are uncorrelated and explain as much variance as possible.
- If you put all the variables together in one matrix, find the best matrix created with fewer variables (lower rank) that explains the original data.

The first goal is statistical and the second goal is data compression.



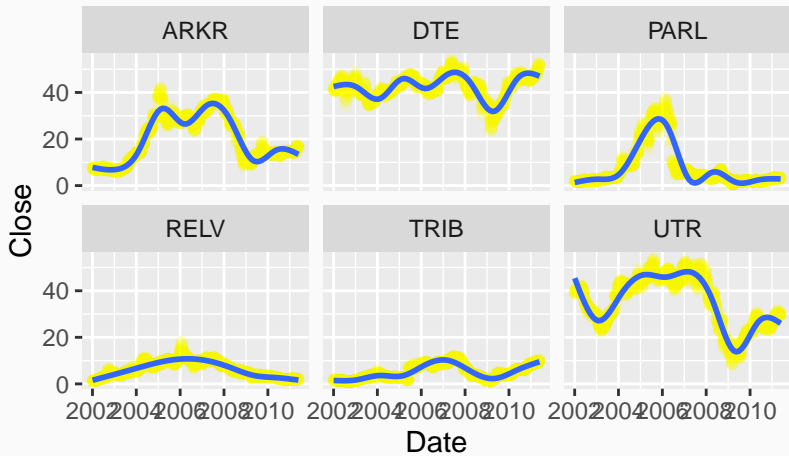
```
library("readr")
gh.link = "https://raw.githubusercontent.com/"
user.repo = "johnmyleswhite/ML_for_Hackers/"
branch = "master/"
link = "08-PCA/data/stock_prices.csv"
data.link = paste0(gh.link, user.repo, branch, link)
df = read_csv(data.link)
```



---

Date	Stock	Close
2011-05-25	DTE	51.12
2011-05-24	DTE	51.51
2011-05-23	DTE	51.47
2011-05-20	DTE	51.90
2011-05-19	DTE	51.91

---



Let's reduce the 25 stocks to 1 dimension and let's call that our *market index*

Dimensionality reduction: shrink a large number of correlated variables into a smaller number

Can be used in many different situations: when we have too many variables for OLS, for unsupervised learning, etc.

```
library("tidyr")  
df.wide = df %>% spread(Stock, Close)  
df.wide = df.wide %>% na.omit
```

```
pca = princomp(select(df.wide, -Date))
```

```
market.index = predict(pca)[, 1]
market.index = data.frame(
  market.index = market.index,
  Date = df.wide$Date)
```

---

market.index	Date
28.24125	2002-01-02
28.16625	2002-01-03
28.07273	2002-01-04
28.30203	2002-01-07
27.62799	2002-01-08

---

**Question:** How do we validate our index?

One suggestion: we can compare it to Dow Jones

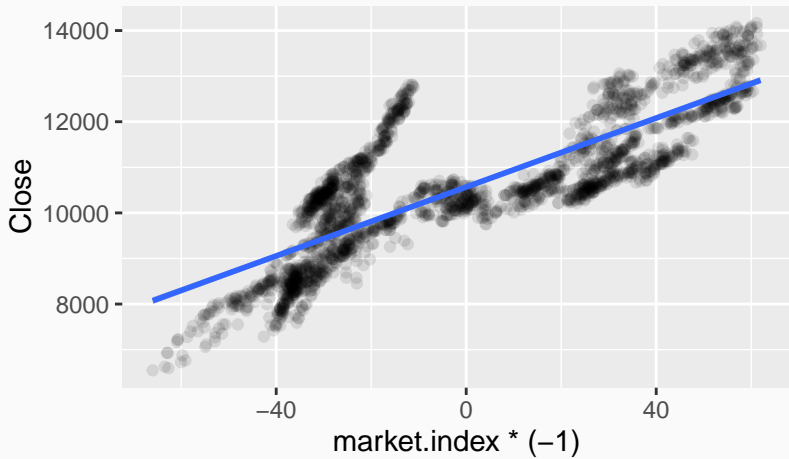


```
library("lubridate")
link = "08-PCA/data/DJI.csv"
data.link = paste0(gh.link, user.repo, branch, link)
dj = read_csv(data.link)
dj = dj %>%
  filter(ymd(Date) > ymd('2001-12-31')) %>%
  filter(ymd(Date) != ymd('2002-02-01')) %>%
  select(Date, Close)
market.data = inner_join(market.index, dj)
```

---

market.index	Date	Close
28.24125	2002-01-02	10073.40
28.16625	2002-01-03	10172.14
28.07273	2002-01-04	10259.74
28.30203	2002-01-07	10197.05
27.62799	2002-01-08	10150.55

---



```
market.data = market.data %>%  
  mutate(  
    market.index = scale(market.index * (-1)),  
    Close = scale(Close))  
market.data = market.data %>%  
  gather(index, value, -Date)
```

