

PML_4-1 Regularized Regression

Basic idea

- Fit a regression model
- Penalize (or shrink) large coefficients

Pros:

- Can help with the bias/variance tradeoff
- Can help with model selection

Cons:

- May be computationally demanding on large data sets
- Does not perform as well as random forests and boosting

This lecture is about **Regularized** regression. We learned about linear regression and generalized linear regression previously. For the basic idea here is to fit one of these regression models and then, penalize or shrink the large coefficients corresponding with some of the predictor variables. Why, because it might help with the bias variance tradeoff iff certain variables are highly coordinated (correlated) with each other. For example, you might not want to include them both in the linear regression model as they will have a very high variance. Then leaving one of them out might slightly bias your model. You might lose a little bit of prediction capability, but you'll save a lot on the variants and therefore improve your prediction to error. It can also help with model selection in certain cases for regular organization techniques like the lasso. It may be computationally demanding on large data sets. In general, it does not perform quite as well as random forests or boosting, when applied to prediction in the wild, for example, in cattle competitions.

A motivating example

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \epsilon$$

where X_1 and X_2 are nearly perfectly correlated (co-linear). You can approximate this model by:

$$Y = \beta_0 + (\beta_1 + \beta_2) X_2 + \epsilon$$

The result is:

- You will get a good estimate of Y
- The estimate (of Y) will be biased
- We may reduce variance in the estimate

Suppose we fit a very simple regression model with an outcome Y and we're trying to predict it with two covariants, X_1 and X_2 , with intercept term β_0 , a constant, plus another constant β_1 times X_1 plus another constant β_2 times X_2 . Assume that X_1 and x_2 are nearly perfectly correlated – they're almost exactly the same variable, often called co-linear. You can then approximate this more complicated model by just including only X_1 and multiplying it by both the coefficients for X_1 and X_2 . It won't be exactly right, because X_1 and X_2 aren't exactly the same variable but it will be very close to right, if X_1 and X_2 are very similar to each other. The result may be that you still get a very good estimate of Y , almost as good as you would've got by including both predictors in the model. The result will be a little bit biased because one of the predictors was left out. But the variance may be reduced if those two variables are highly correlated with each other.

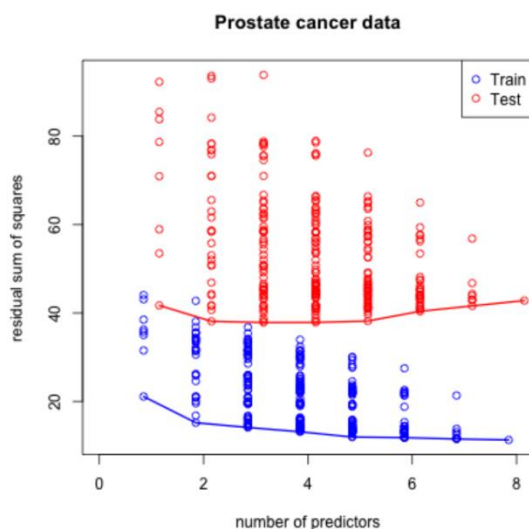
Prostate cancer

```
library(ElemStatLearn); data(prostate)
str(prostate)
```

```
'data.frame':  97 obs. of  10 variables:
 $ lcaVOL : num  -0.58 -0.994 -0.511 -1.204 0.751 ...
 $ lweight: num  2.77 3.32 2.69 3.28 3.43 ...
 $ age    : int   50 58 74 58 62 50 64 58 47 63 ...
 $ lbph   : num  -1.39 -1.39 -1.39 -1.39 -1.39 ...
 $ svi    : int    0 0 0 0 0 0 0 0 0 0 ...
 $ lcp    : num  -1.39 -1.39 -1.39 -1.39 -1.39 ...
 $ gleason: int    6 6 7 6 6 6 6 6 6 6 ...
 $ pgg45  : int    0 0 20 0 0 0 0 0 0 0 ...
 $ lpsa   : num  -0.431 -0.163 -0.163 -0.163 0.372 ...
 $ train  : logi   TRUE TRUE TRUE TRUE TRUE TRUE ...
```

Here's an example, with, prostate cancer data set and the elements of statistical learning library. Load the prostate data and look at that data set (97 observations on ten variables). Let's make a prediction about prostate cancer, PSA, based on a large number of predictors in the data set. This is very typical of what happens when you build these models in practice. Let's predict with all possible combinations of predictor variables. We go to the linear regression model for the outcome where we build one regression model for every possible combination of vectors.

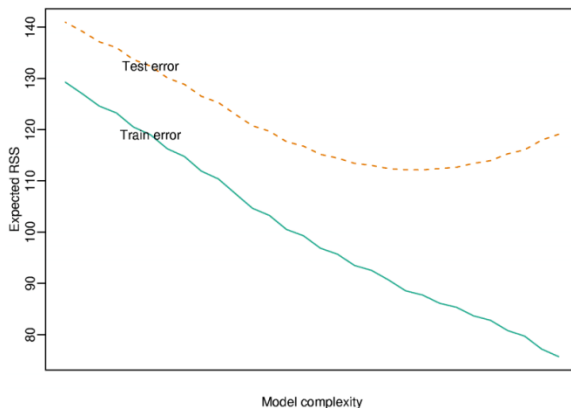
Subset selection



Then we can see as the number of predictors increases from left to right, the training set error always goes down. It has to down. As you include more predictors, the training set (blue) error will always decrease. But this is a typical pattern of what you observe with real data. The test set data on the other hand, as the number of predictors increases, the test set (red) error goes down but then eventually it hits a plateau, and it starts to go back up again. This is because we're overfitting the data in the training set, and eventually, we may not want to include so many predictors in our model.

Most common pattern

This is an incredibly common pattern. Basically any measure of model complexity on the X-axis **versus** the expected Residual Sum of Squares of the predicted error (Y-axis). You can see that in the training set almost always the error goes monotonically down, As you build more and more complicated models, the training error will always decrease. But on a testing set, the error will decrease for a while, eventual hit a minimum. And then, start to increase again as the model gets too complex and overfits the data.



Model selection approach: split samples

In general, the best approach might be to split samples when you have enough data and you have enough computation time. The idea is divide your data into training, test, and validation sets. Treat the validation set as the test data and you train every possible competing model – all possible subsets on the training data. Then pick the one that works best on the validation data set.

But now that we've used the validation set for training, we need to assess the error rate on a completely independent set. So we appropriately assess this performance by applying our prediction to the new data in the test set. Sometimes you may re-perform the splitting and analysis several times in order to get a better average estimate of what the out of sample error rate will be. But there are two common problems with this approach, one is that there might be limited data. Here we're breaking the data set up into three different data sets. It might not be possible to get a very good model fit when we split the data that finely. Second, is the computational complexity. If you're trying all possible subsets of models it can be very complicated, especially if you have a lot of predictor variables.

Model selection approach: split samples

- No method better when data/computation time permits it
- Approach
 1. Divide data into training/test/validation
 2. Treat validation as test data, train all competing models on the train data and pick the best one on validation.
 3. To appropriately assess performance on new data apply to test set
 4. You may re-split and re-perform steps 1-3
- Two common problems
 - Limited data
 - Computational complexity

<http://www.biostat.jhsph.edu/~ririzarr/Teaching/649/>

<http://www.cbcb.umd.edu/~hcorrada/PracticalML/>

Decomposing expected prediction error

Assume $Y_i = f(X_i) + \epsilon_i$

$$EPE(\lambda) = E[\{Y - \hat{f}_\lambda(X)\}^2]$$

Suppose \hat{f}_λ is the estimate from the training data and look at a new data point $X = x^*$

$$\begin{aligned} E[\{Y - \hat{f}_\lambda(x^*)\}^2] &= \sigma^2 + \{E[\hat{f}_\lambda(x^*)] - f(x^*)\}^2 + \text{var}[\hat{f}_\lambda(x_0)] \\ &= \text{Irreducible error} + \text{Bias}^2 + \text{Variance} \end{aligned}$$

Another approach is to try to decompose the prediction error to see if there's another way that we can directly get at including only the variable that need to be included in the model. Assume that the variable y can be predicted as a function of x , plus some error term, the expected prediction error is the expected difference between the outcome and the prediction of the outcome squared. \hat{f}_λ is the estimate from the training set using a particular set of tuning parameters λ . Then if we look at a

new point, x^* , and calculate the distance between our observed outcome, Y , and the prediction on the new data point, $\hat{f}_\lambda(x^*)$. $E[\{Y - \hat{f}_\lambda(x^*)\}^2]$ can be decomposed after some algebra into irreducible error, σ^2 , the bias $\{E[\hat{f}_\lambda(x^*)] - f(x^*)\}^2$, the difference between our expected prediction and the truth), and the variance of our estimate, $\text{var}[\hat{f}_\lambda(x_0)]$. Whenever you're building a prediction model, the goal is to reduce the **Irreducible Error**, σ^2 . This is the expected mean squared error between outcome and prediction. The irreducible error usually can't usually be reduced. It's a value that's just part of the data you're collecting. But you can trade off bias and variance and that's what the idea behind regularized regression does.

Another issue for high-dimensional data

```
small = prostate[1:5,]
lm(lpsa ~ ., data = small)
```

```
Call:
lm(formula = lpsa ~ ., data = small)
```

Coefficients:

(Intercept)	lcavol	lweight	age	lbph	svi	lcp
9.6061	0.1390	-0.7914	0.0952	NA	NA	NA
gleason	pgg45	trainTRUE				
-2.0871	NA	NA				

Another issue for high dimensional data is shown by a simple example of what happens with a lot of predictors. Take a small subset of the prostate data such that I only have five observations in my training set. It has more than five predictor variables. So I fit a linear model relating the outcome to all of these predictor variables. There are more than five so some of them will get estimates and some of them will be NA. In other words, R won't estimate them because there are more predictors than samples resulting in a design matrix that cannot be inverted.

Hard thresholding

- Model $Y = f(X) + \epsilon$
- Set $\hat{f}_\lambda(X) = x'\beta$
- Constrain only λ coefficients to be nonzero.
- Selection problem is after choosing λ figure out which $p - \lambda$ coefficients to make nonzero
- One approach to dealing with this problem and the other problem of trying to select our model is we could take this model that we have, $Y = f(X) + \epsilon$, and assume that it has a linear form, $\hat{f}_\lambda(X) = x'\beta$, a linear regression model like before. And then constrain only the lambda of the coefficients to be non-zero. After we pick lambda, suppose there are only three non-zero coefficients. Then we have to try all the possible combinations of three coefficients that are non-zero and fit the best model. That's still computationally quite demanding.

Regularization for regression

If the β_j 's are unconstrained:

- They can explode
- And hence are susceptible to very high variance

To control variance, we might regularize/shrink the coefficients.

$$PRSS(\beta) = \sum_{j=1}^n (Y_j - \sum_{i=1}^m \beta_{1i} X_{ij})^2 + P(\lambda; \beta)$$

where **PRSS** is a **penalized form of the sum of squares**. Things that are commonly looked for

- Penalty reduces complexity
- Penalty reduces variance
- Penalty respects structure of the problem

Another method is to use regularized regression. If the coefficients, β_j 's, that we're fitting in the linear model are unconstrained, i.e., we don't claim that they have any particular form, they may explode if you have very highly correlated variables that you're using for prediction. And so, they can be susceptible to high variance and high variance means that you'll get predictions that aren't as accurate. To control the variance we might regularize, or shrink the coefficients. What we might want to minimize, is a distance between our outcome that we have and our linear model. So here, this is the distance, $(Y_j - \sum_{i=1}^m \beta_{1i} X_{ij})$, between the outcome and the linear model fit squared. That's the residual sum of squares. Then you might also add a penalty term, $P(\lambda; \beta)$. The penalty will basically shrink beta coefficients back down if they are too big. The penalty is usually used to reduce complexity. It can be used to reduce variance and it can respect some of the structure in the problem if the penalty is set up in the right way.

Ridge regression

Solve:

$$\sum_{i=1}^N \left(y_i - \beta_0 + \sum_{j=1}^p x_{ij} \beta_j \right)^2 + \lambda \sum_{j=1}^p \beta_j^2$$

Is equivalent to solving

$$\sum_{i=1}^N \left(y_i - \beta_0 + \sum_{j=1}^p x_{ij} \beta_j \right)^2$$

subject to

$$\sum_{j=1}^p \beta_j^2 \leq s$$

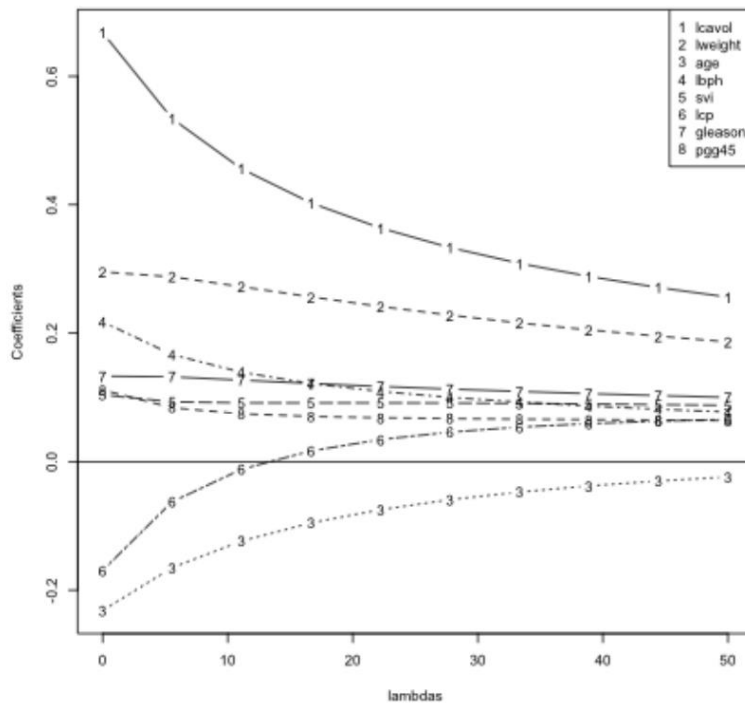
where s is inversely proportional to λ . Inclusion of λ makes the problem non-singular even if $X^T X$ is not invertible.

The first approach that was used in this sort of penalized regression is to fit the regression model. Here again, we're penalizing a distance between our outcome y and our regression model,

$(y_i - \beta_0 + \sum_{j=1}^p x_{ij} \beta_j)$. And then we also have a term, $\lambda \sum_{j=1}^p \beta_j^2$, lambda times the sum of the beta j 's squared. What does this mean? If β_j^2 's are really big then $\sum_{j=1}^p \beta_j^2$ will get too big and we won't get a very good fit. The whole quantity will end up being very big, so it basically requires that some β_j^2 's be small. It's actually equivalent to solving this problem where we're looking for the smallest sum of

differences squared, $\sum_{i=1}^N (y_i - \beta_0 + \sum_{j=1}^p x_{ij}\beta_j)^2$, subject to a particular constraint $\sum_{j=1}^p \beta_j^2 \leq s$. The idea is that the inclusion of the λ coefficient may also even make the problem non-singular. Even when the x transpose x is not invertible. In other words, in that model fit where we have more predictors than we do observations the ridge regression model can still be fit.

Ridge coefficient paths



This is what the coefficient paths looks like. So what do I mean by coefficient path. For every different choice of λ , that penalized regression problem on the previous page, as λ increases that means that we penalize the big β 's more and more. We start off with the β 's being equal to a certain of values (Y-axis) where $\lambda = 0$. That's just a standard linear with regression values. As λ increases, all of the coefficients get closer to 0 because we're penalizing the coefficients and making them smaller.

Tuning parameter λ

- λ controls the size of the coefficients
- λ controls the amount of *regularization*
- As $\lambda \rightarrow 0$ we obtain the least square solution
- As $\lambda \rightarrow \infty$ we have $\hat{\beta}_{\lambda=\infty}^{ridge} = 0$

So the tuning parameter λ controls the size of the control coefficients. Lambda controls the amount of regularization. As λ get's closer and closer to 0, we basically go back to the least square solution which is what you get from a standard linear model. And as λ goes to infinity, gets really big, it penalizes the coefficients, and the conditioned coefficients go toward 0 as the tuning parameter gets really big. So taking that parameter can be done with cross-validation or other techniques to try to pick the optimal tuning parameter. That trades off bias for variance.

Lasso

$$\sum_{i=1}^N (y_i - \beta_0 + \sum_{j=1}^p x_{ij}\beta_j)^2 \quad \text{subject to} \quad \sum_{j=1}^p |\beta_j| \leq s$$

also has a Lagrangian form $\sum_{i=1}^N (y_i - \beta_0 + \sum_{j=1}^p x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^p |\beta_j|$

For orthonormal design matrices (not the norm!) this has a closed form solution

$$\hat{\beta}_j = \text{sign}(\hat{\beta}_j^0)(|\hat{\beta}_j^0| - \gamma)^+$$

but not in general.

A similar approach can be done with a slight change of penalty. So again, here we might be solving the least squares problem. This is the standard trying to identify the beta values that make the distance to the outcome smallest, $(y_i - \beta_0 + \sum_{j=1}^p x_{ij}\beta_j)$. We can constrain it subject to the sum of the absolute value of the β_j 's being less than some value, s . You can also write that as a penalized regression of this form, so we're trying to solve the penalized sum of squares. So for ortho normal design matrix or the normal design matrix, the idea is that this actually has a closed form solution. And the closed form solution is basically, take $|\beta_j|$ and subtract off a gamma value, γ , and take only the positive part. In other words if γ is bigger than $|\hat{\beta}_j^0|$ then this will be a negative number and you're taking only the positive part so you set $(|\hat{\beta}_j^0| - \gamma)^+ = 0$. If it's a positive, $|\hat{\beta}_j^0| > \gamma$ then $(|\hat{\beta}_j^0| - \gamma)$ will be a smaller positive number by the amount, γ . Then multiply it by the sign of the original coefficient.

This is basically saying the lasso shrinks all of the coefficients and sets some of them to exactly 0. Some analysts like this approach because it both shrinks coefficients and by setting some exactly to 0, it performs model selection for you in advance.

Notes and further reading

- [Hector Corrada Bravo's Practical Machine Learning lecture notes](#)
- [Hector's penalized regression reading list](#)
- [Elements of Statistical Learning](#)
- In caret methods are:
 - ridge
 - lasso
 - relaxo

There are really good set of lecture notes from Hector Corrada Bravo. He also has a very nice list on the large number of penalized regression models. And in the Elements of Statistical Learning book covers this penalized regression idea a quite extensive detail.

In caret, if you want to fit these models, you can set the method to ridge, lasso or relaxo to fit different kinds of penalized regression models.

Code for subset selection graphics

```
####
# regression subset selection in the prostate dataset
library(ElmStatLearn)
data(prostate)

covnames <- names(prostate[ -(9:10)])
y <- prostate$lpsa
x <- prostate[,covnames]

form <- as.formula(paste("lpsa~", paste(covnames, collapse="+"), sep=""))
summary(lm(form, data=prostate[prostate$train,]))

set.seed(1)
```

```

train.ind <- sample(nrow(prostate), ceiling(nrow(prostate))/2)
y.test <- prostate$lpca[-train.ind]
x.test <- x[-train.ind,]

y <- prostate$lpca[train.ind]
x <- x[train.ind,]

p <- length(covnames)
rss <- list()
for (i in 1:p) {
  cat(i)
  Index <- combn(p,i)

  rss[[i]] <- apply(Index, 2, function(is) {
    form <- as.formula(paste("y~", paste(covnames[is], collapse="+"), sep=""))
    isfit <- lm(form, data=x)
    yhat <- predict(isfit)
    train.rss <- sum((y - yhat)^2)

    yhat <- predict(isfit, newdata=x.test)
    test.rss <- sum((y.test - yhat)^2)
    c(train.rss, test.rss)
  })
}

png("Plots/selection-plots-01.png", height=432, width=432, pointsize=12)
plot(1:p, 1:p, type="n", ylim=range(unlist(rss)), xlim=c(0,p), xlab="number of predictors", ylab="residual
sum of squares", main="Prostate cancer data")
for (i in 1:p) {
  points(rep(i-0.15, ncol(rss[[i]])), rss[[i]][1, ], col="blue")
  points(rep(i+0.15, ncol(rss[[i]])), rss[[i]][2, ], col="red")
}
minrss <- sapply(rss, function(x) min(x[1,]))
lines(1:p-0.15, minrss, col="blue", lwd=1.7)
minrss <- sapply(rss, function(x) min(x[2,]))
lines(1:p+0.15, minrss, col="red", lwd=1.7)
legend("topright", c("Train", "Test"), col=c("blue", "red"), pch=1)
dev.off()

##
# ridge regression on prostate dataset
library(MASS)
lambdas <- seq(0,50,len=10)
M <- length(lambdas)
train.rss <- rep(0,M)
test.rss <- rep(0,M)
betas <- matrix(0,ncol(x),M)
for(i in 1:M){
  Formula <- as.formula(paste("y~",paste(covnames,collapse="+"),sep=""))
  fit1 <- lm.ridge(Formula,data=x,lambda=lambdas[i])
  betas[,i] <- fit1$coef

  scaledX <- sweep(as.matrix(x),2,fit1$xm)
  scaledX <- sweep(scaledX,2,fit1$scale,"/")
  yhat <- scaledX%*%fit1$coef+fit1$ym
  train.rss[i] <- sum((y - yhat)^2)

  scaledX <- sweep(as.matrix(x.test),2,fit1$xm)
  scaledX <- sweep(scaledX,2,fit1$scale,"/")
  yhat <- scaledX%*%fit1$coef+fit1$ym
  test.rss[i] <- sum((y.test - yhat)^2)
}

png(file="Plots/selection-plots-02.png", width=432, height=432, pointsize=12)
plot(lambdas,test.rss,type="l",col="red",lwd=2,ylab="RSS",ylim=range(train.rss,test.rss))
lines(lambdas,train.rss,col="blue",lwd=2,lty=2)
best.lambda <- lambdas[which.min(test.rss)]
abline(v=best.lambda+1/9)
legend(30,30,c("Train", "Test"),col=c("blue", "red"),lty=c(2,1))

```



```
dev.off()

png(file="Plots/selection-plots-03.png", width=432, height=432, pointsize=8)
plot(lambdas,betas[1,],ylim=range(betas),type="n",ylab="Coefficients")
for(i in 1:ncol(x))
  lines(lambdas,betas[i,],type="b",lty=i,pch=as.character(i))
abline(h=0)
legend("topright",covnames,pch=as.character(1:8))
dev.off()

#####
# lasso
library(lars)
lasso.fit <- lars(as.matrix(x), y, type="lasso", trace=TRUE)

png(file="Plots/selection-plots-04.png", width=432, height=432, pointsize=8)
plot(lasso.fit, breaks=FALSE)
legend("topleft", covnames, pch=8, lty=1:length(covnames), col=1:length(covnames))
dev.off()

# this plots the cross validation curve
png(file="Plots/selection-plots-05.png", width=432, height=432, pointsize=12)
lasso.cv <- cv.lars(as.matrix(x), y, K=10, type="lasso", trace=TRUE)
dev.off()
```

PML_4-2 Combining Predictors

Key ideas

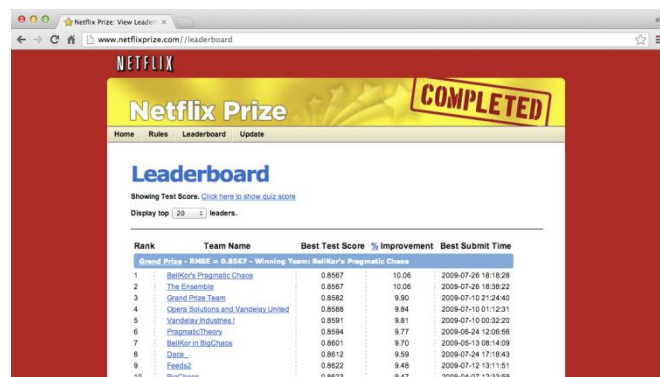
- You can combine classifiers by averaging/voting
- Combining classifiers improves accuracy
- Combining classifiers reduces interpretability
- Boosting, bagging, and random forests are variants on this theme

This lecture is about combining predictors, sometimes called **ensembling** methods in learning. The key idea here is that you can combine classifiers by averaging or voting; these can be very different classifiers. For example, you can combine a boosting classifier with a random forest with a linear regression model. In general combining classifiers together improves accuracy but it can also reduce interpretive abilities. You have to be really careful the gain that you get is worth the loss on inter-operability. Boosting, bagging, and random floor, they're all basically a variance on this theme that you can average different classifiers together. They are all examples of the same classifier being averaged in every case.

Netflix prize

BellKor = Combination of 107 predictors

An example of how successful this can be, the Netflix prize was a million dollar prize that was a major prediction contest in 2009. The team that ended up winning was a blended combination of 107 different Machine Learning algorithms combined together to get the best score in this competition.



The screenshot shows the Netflix Prize Leaderboard with a 'COMPLETED' banner. The table lists the top 10 teams, their best test scores, percentage improvements, and best submit times.

Rank	Team Name	Best Test Score	% Improvement	Best Submit Time
1	BellKor's Pragmatic Chaos	0.8567	10.06	2009-07-26 18:18:28
2	The Ensemble	0.8567	10.06	2009-07-26 18:38:22
3	Grand Prize Team	0.8562	9.90	2009-07-10 21:24:40
4	Genes Solutions and Vandelay United	0.8558	9.84	2009-07-10 01:12:31
5	Vandelay Industries	0.8591	9.81	2009-07-10 00:32:20
6	PragmaticTheory	0.8594	9.77	2009-06-24 12:06:56
7	BellKor in BigChaos	0.8601	9.70	2009-09-13 08:14:09
8	Qeios	0.8612	9.59	2009-07-28 17:18:43
9	FeedSift	0.8622	9.48	2009-07-12 13:11:51
10	BraChaos	0.8623	9.47	2009-04-07 12:33:59

Heritage health prize - Progress Prize 1

The Heritage Health prize was also won by methods that were ensembling for multiple different regression models and machine learning methods. The Heritage Health prize was a \$3 million prize designed to try to predict whether people would go back to the hospital based on their hospitalization record. Below are both of the links to the first progress prizes for the two teams that were in the lead.

Market Makers : "There were four distinct steps in creating the solution;

Manipulating the raw data into a suitable format for modelling: The data provided for use in the predictive models was each claim made by a patient, so there could be multiple records per patient. The objective is to predict the Days in Hospital, a single value for each patient per year. The claim level data was aggregated to a patient level to generate modelling data sets

Predictive Modelling: Predictive models were built utilizing the data sets created in Step 1. Numerous mathematical techniques were used to generate a set of candidate solutions.

Ensembling: The individual solutions produced in Step 2 were combined to create a single solutions that was more accurate than any of its components.

Future Proofing: The prediction data is one year ahead of the data provided to build the models. Events such as medical advances or policy changes might have subsequently occurred that may affect the model. Techniques can be used to help ensure the model works as efficiently as possible in the ‘future’.”

Mestrom: “**Introduction** My milestone 1 solution to the Heritage Health Prize with a RMSLE score of 0.457239 on the leaderboard consists of a linear blend of 21 result[s]. These are mostly generated by relatively simple models which are all trained using stochastic gradient descent. First in section 2, I proved a description of the way the data is organized and the features that were used. Then in section 3 the training method and the post-processing steps are described and the features that were used. Then in section 3, the training method and the post-processing steps are described. In section 4 each individual model is briefly described, all the relevant meta-parameter setting can be found in appendix Parameter settings. Finally the weights in the final blend are given in section 5.”

Basic intuition - majority vote

Suppose we have 5 completely independent classifiers. If accuracy is 70% for each, $\left[\binom{5}{3} = 10\right]$:

- $\binom{5}{3} \times (0.7)^3 (0.3)^2 + \binom{5}{4} \times (0.7)^4 (0.3)^1 + \binom{5}{5} (0.7)^5 = 0.837$
- 83.7% majority vote accuracy (probability of best (3 of 5) or (4 of 5) or (5 of 5)).

With 101 independent classifiers

- 99.9% majority vote accuracy

The basic intuition here is like a majority vote. Suppose we have five completely independent classifiers with an accuracy of 70% for each classifier. Then we can think that the accuracy for majority vote would be 10 times [five choose three] the majority vote being right for three out of the five classifiers, for four out of the five classifiers [five choose four], or for all five classifiers [five choose five]. That calculates to be 83.7% majority vote accuracy. With 101 independent classifiers you get 99.9% majority vote accuracy.

Approaches for combining classifiers

1. Bagging, boosting, random forests
 - Usually combine similar classifiers
2. Combining different classifiers
 - Model stacking
 - Model ensembling

One approach is using similar classifiers and combining them together using something like bagging, boosting, or random forests. Another approach is to combine different classifiers using model stacking or model ensembling.

Example with Wage data

Create training, test, and validation sets

```
library(ISLR); data(Wage); library(ggplot2); library(caret);
Wage <- subset(Wage, select=-c(logwage))
# Create a building data set and validation set
inBuild <- createDataPartition(y=Wage$wage,
```

```

                                p=0.7, list=FALSE)
validation <- Wage[-inBuild,]; buildData <- Wage[inBuild,]
inTrain <- createDataPartition(y=buildData$wage,
                                p=0.7, list=FALSE)
training <- buildData[inTrain,]; testing <- buildData[-inTrain,]

```

This is an example of model stacking with the wage data. Build the wage data set and leave out the **logwage** variable since we're trying to predict wage, log wage would be a very good predictor. We build the **validation** set and, we also build a **training** set and a **test** set to create three different data sets in these few lines of code. The importance of creating three sets will be clear in the following.

Wage data sets

```

dim(training)
[1] 1474 11
dim(testing)
[1] 628 11
dim(validation)
[1] 898 11

```

The training set is the largest of the data sets with 1,474 examples. The test set has 628 and the validation set has 898 samples in it. In the training set build using a GLM, a linear model. And with `~.` predicting **wage** using all the other variables. Then build a separate model, also in the training data and use method equals random forest, **method="rf"**. These two steps fit two different prediction models to the same data set. Then plot those predictions versus each other.

Build two different models

```

mod1 <- train(wage ~.,method="glm",data=training)
mod2 <- train(wage ~.,method="rf",
              data=training,
              trControl = trainControl(method="cv"),number=3)

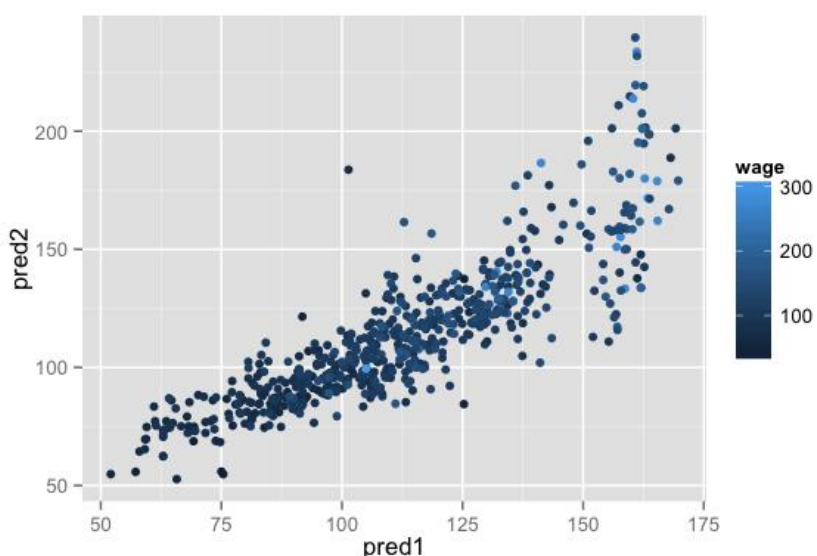
```

Predict on the testing set

```

pred1 <- predict(mod1,testing); pred2 <- predict(mod2,testing)
qplot(pred1,pred2,colour=wage,data=testing)

```



Pred1 is the predictions from the linear model, and pred2 are the predictions from the random forest. And you can see that they, they're close to each other, but they don't actually agree with each other. And, neither of them perfectly correlates with the wage variable, which is the color of the dots in this particular picture. Then fit a model that combines the predictors.

Fit a model that combines predictors

```
predDF <- data.frame(pred1,pred2,wage=testing$wage)
combModFit <- train(wage ~.,method="gam",data=predDF)
combPred <- predict(combModFit,predDF)
```

Build a new data set consisting of the predictions from model one and the predictions from model two. And then create a wage variable, `wage=testing$wage`, on the test data set, `predDF`. Then fit (`train`) a new model, `combModFit`, which relates the wage variable to the two predictions, `wage ~. .`. Now, instead of just fitting a model that relates the co-variants to the outcome, two separate prediction models have been fitted to create the outcome. That is, a regression model relating the outcome to the predictions for those two models to predict from the combined data set on new samples.

Testing errors

```
sqr(sum((pred1-testing$wage)^2))
[1] 827.1
sqr(sum((pred2-testing$wage)^2))
[1] 866.8
sqr(sum((combPred-testing$wage)^2))
[1] 813.9
```

How well do I do on the test set? See that for example on the test set the first predictor has an error of 827.1 and the second predictor has an error of 866, whereas the combined predictor has an error that's lower, 813.9.

And I also want to try it out on the validation set because remember I used the test set to try to blend the two models together.

Predict on validation data set

```
pred1V <- predict(mod1,validation); pred2V <- predict(mod2,validation)
predVDF <- data.frame(pred1=pred1V,pred2=pred2V)
combPredV <- predict(combModFit,predVDF)
```

The model fit on the test set is not a good representation of the out of sample error. So now create a prediction of the first model on the validation set (`pred1V`) and a prediction of the second model on the validation set (`pred2V`). Then create a data frame that contains those two predictions (`combPredV`). Next predict using the combined model (`combModFit`) on the predictions in the validation data set (`predVDF`). The covariance being passed in the model are the predictions from the two different models.

Evaluate on validation

```
sqr(sum((pred1V-validation$wage)^2))
[1] 1003
sqr(sum((pred2V-validation$wage)^2))
[1] 1068
sqr(sum((combPredV-validation$wage)^2))
[1] 999.9
```

The error from model one when used it by itself is 1003. The error for my second model is 1068. And the combined model has a lower error (999.9) even on the validation data set. Stacking models in this way can improve accuracy by blending the strengths of different models together.

Notes and further resources

- Even simple blending can be useful
- Typical model for binary/multiclass data
 - Build an odd number of models
 - Predict with each model

- Predict the class by majority vote
- This can get dramatically more complicated
 - Simple blending in caret: [caretEnsemble](#) (use at your own risk!)
 - Wikipedia [ensemble learning](#)

Even simple blending like that presented here can be really useful and can improve accuracy. The typical model blending for binary multiclass data involves building an *odd number* of models and then predicting the outcome with each model. Next assign the ultimate class label based on majority vote.

This can get dramatically more complicated. There is a caret package ensemble that ensembles some of the caret models together. But it's still in development, so use at your own risk. The Wikipedia page linked describes more about ensemble learning.

Recall - scalability matters

One important point to keep in mind when doing model ensembling is that this can lead to increases in computational complexity. It turns out that even though Netflix paid a million dollars to the team that won the prize; the solution was never actually implemented. It was too computationally intensive to apply to specific data sets. Recall from the earliest lectures in this class that trade-offs with *scalability versus accuracy* occur that you have to pay attention to when building these prediction models.

<http://www.techdirt.com/blog/innovation/articles/20120409/03412518422/>
<http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html>

PML_4-3 Forecasting

This lecture is about forecasting that is a very specific class of prediction problem. It's typically applied to things like time series data. For example, this is the stock of information for Google on the NASDAQ, and so is this symbol GOOG.

Time series data



<https://www.google.com/finance>

The price for this stock goes up and down over time. This introduces very specific kinds of dependent structure and additional challenges that must be taken into account when performing prediction.

What is different?

- Data are dependent over time
- Specific pattern types
 - Trends - long term increase or decrease
 - Seasonal patterns - patterns related to time of week, month, year, etc.
 - Cycles - patterns that rise and fall periodically
- Subsampling into training/test is more complicated
- Similar issues arise in spatial data
 - Dependency between nearby observations
 - Location specific effects
- Typically goal is to predict one or more observations into the future.
- All standard predictions can be used (with caution!)

First, the data are dependent over time and that alone makes prediction a bit more challenging than when you have independent examples. There are also specific pattern types that should be paid attention to:

1. trends, such as long term increases or decreases,
2. seasonal patterns that are very common in this stock price data. For example, seasonal patterns over weeks, months, years, etc.
3. cycles, patterns that rise and fall periodically over a period that's longer than a year, for example.

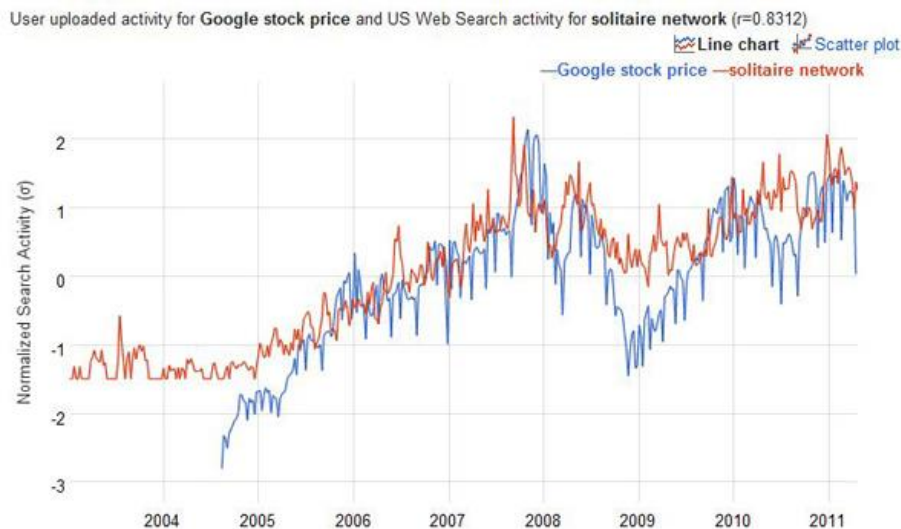
Here, the *subsampling* and the *training* and *test* [sets] can be more complicated because you can't just randomly assign samples into training and test. You have to take advantage of the fact that there are actually specific times that are being sampled and that points are dependent in time.

Similar issues arise in predictions of spatial data. For example, there's dependency between nearby observations and there may be location-specific effects that have to be modeled when doing prediction. Typically, the goal is to predict one or more observations into the future and all standard prediction algorithms can be used but you have to be cautious about how you use them.

Beware spurious correlations!

<http://www.google.com/trends/correlate>

<http://www.newscientist.com/blogs/onepercent/2011/05/google-correlate-passes-our-we.html>

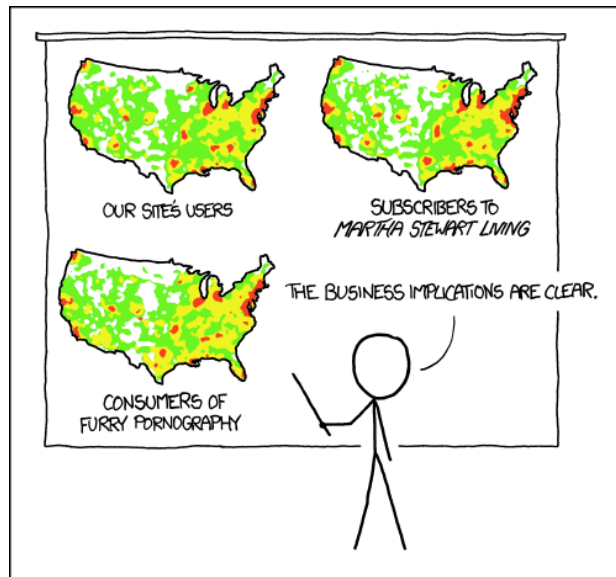


One thing to be aware of is spurious correlations. Time series can often be correlated for reasons that do not make them good for predicting one from the other. You can go to Google Correlate to correlate (the frequency) different words over time. For example, here you can see a correlation between the Google stock price, shown in blue, and solitaire network, which is in red. These two [stock price and search] don't necessarily have anything to do with each other at all, but they have a very high correlation. You might think to be able to predict one from the other, even though in the future, they might diverge substantially because they aren't necessarily related to each other at all.

Also common in geographic analyses

It's also very common in geographic analysis. Below is a cartoon from xkcd (<http://xkcd.com/1138/>) that shows that heat maps, particularly population-based heat maps, had very similar shapes because of the place where many people live. For example, the users of a particular site or the subscribers to a particular magazine or the consumers of a particular type of website may all appear in very similar places because the highest density in population in the United States is over here on the Eastern

seaboard. Thus you see very similar heat maps of a large number of individuals at all of those different places.

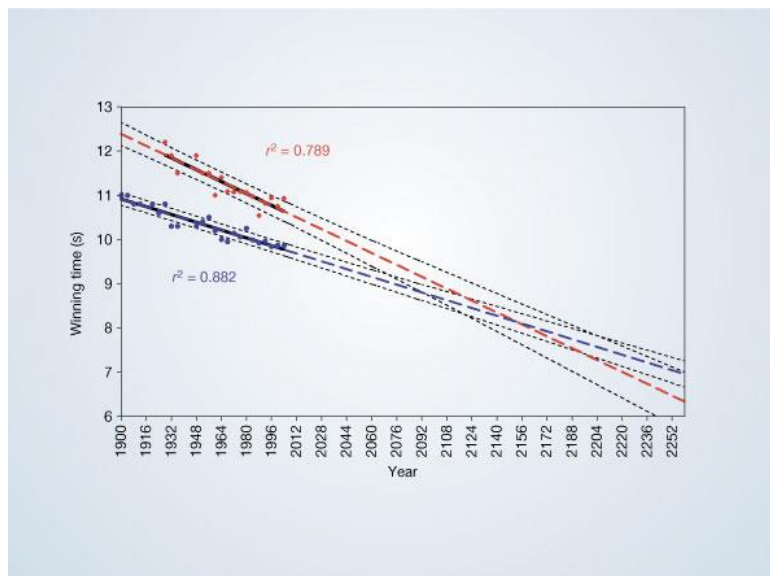


PET PEEVE #208:
GEOGRAPHIC PROFILE MAPS WHICH ARE
BASICALLY JUST POPULATION MAPS

Beware extrapolation!

You should also beware of extrapolation. A kind of a funny example shows what happens if you extrapolate time series out without being careful about what could happen. The graph below shows on a long scale the winning time of a large number of races that occurred at the Olympics. The blue times are men and the red times are women.

The authors of this paper extrapolated out into the future and said that in 2156 women would run faster than men in the sprint. And while we don't know when that may or may not occur, one thing this [simple] extrapolation points out is that it is very dangerous. Eventually at some time in the future, both men and women will be predicted to run negative times for the 100 meters. Clearly, you have to be very careful about how far out you extrapolate from your data.



<http://www.nature.com/nature/journal/v431/n7008/full/431525a.html>

Google data

```
library(quantmod)
from.dat <- as.Date("01/01/08", format="%m/%d/%y")
to.dat <- as.Date("12/31/13", format="%m/%d/%y")
getSymbols("GOOG", src="google", from = from.dat, to = to.dat)
```

```
[1] "GOOG"
```

```
head(GOOG)
```

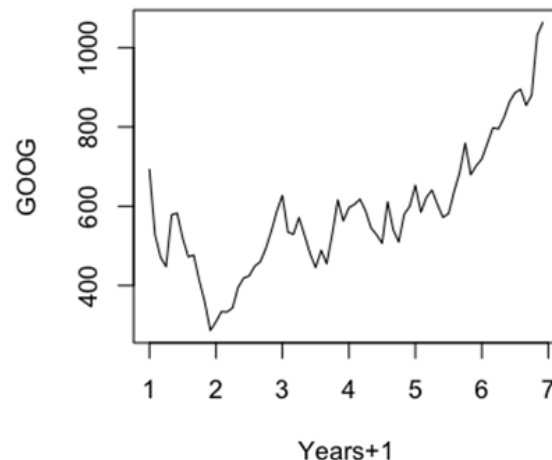
	GOOG.Open	GOOG.High	GOOG.Low	GOOG.Close	GOOG.Volume
2008-01-02	692.9	697.4	677.7	685.2	4306848
2008-01-03	685.3	686.9	676.5	685.3	3252846
2008-01-04	679.7	681.0	655.0	657.0	5359834
2008-01-07	653.9	662.3	637.4	649.2	6404945
2008-01-08	653.0	660.0	631.0	631.7	5341949
2008-01-09	630.0	653.3	622.5	653.2	6744242

Here is a quick example of some forecasting using the **quantmod** package with some Google data. Load the **quantmod** package and a bunch of data from the Google stock symbol, from the Google finance data set, **src="google"**. Look at the Google variable, **head(GOOG)** to get the open, high, low, close, and volume information for a particular Google stock from the 1st of January, 2008 to December 31st, 2013.

Summarize monthly and store as time series

```
mGoog <- to.monthly(GOOG)
googOpen <- Op(mGoog)
ts1 <- ts(googOpen, frequency=12)
plot(ts1, xlab="Years+1", ylab="GOOG")
```

I can summarize this monthly (**mGoog**) and store it as a time series, **ts1**. Use the **to.monthly()** function to convert **GOOG** to a monthly time series and take just the opening information, **googOpen**, and then create a time series object using the **ts()** function in R. The plot shows the monthly opening prices for Google over a period of seven years.



Example time series decomposition

- Trend - Consistently increasing pattern over time
- Seasonal - When there is a pattern over a fixed period of time that recurs.
- Cyclic - When data rises and falls over non fixed periods

<https://www.otexts.org/fpp/6/1>

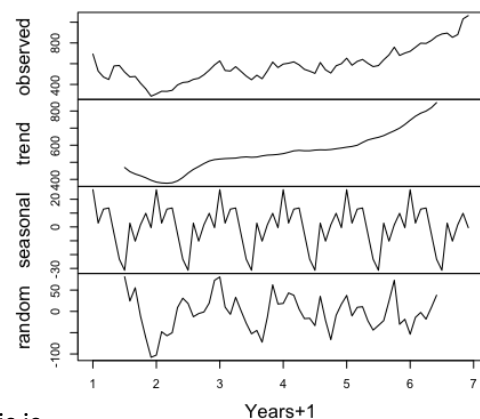
An example time series decomposition would decompose this time series into: 1) a trend, any kind of consistent pattern, 2) a seasonal pattern over time, and 3) cyclic patterns where the data rises and falls over non fixed periods.

Decompose a time series into parts

```
plot(decompose(ts1), xlab="Years+1")
```

One way to do this is with the **decompose** function in R. So if I decompose this in an additive way, then I can see that there's a trend variable that appears to be an upward trend of the Google stock price. There also appears to be a seasonal pattern, as well as a more of a random cyclical pattern in the data set. This is

Decomposition of additive time series



decomposing the series [in the top row] into a [three] series of different patterns in the data.

Training and test sets

```
ts1Train <- window(ts1,start=1,end=5)
ts1Test <- window(ts1,start=5,end=(7-0.01))
ts1Train
```

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
1	692.9	528.7	471.5	447.7	578.3	582.5	519.6	472.5	476.8	412.1	357.6	286.7
2	308.6	334.3	333.3	343.8	395.0	418.7	424.2	448.7	459.7	493.0	537.1	588.1
3	627.0	534.6	529.2	571.4	526.5	480.4	445.3	489.0	455.0	530.0	615.7	563.0
4	596.5	604.5	617.8	588.8	545.7	528.0	506.7	611.2	540.8	509.9	580.1	600.0
5	652.9											

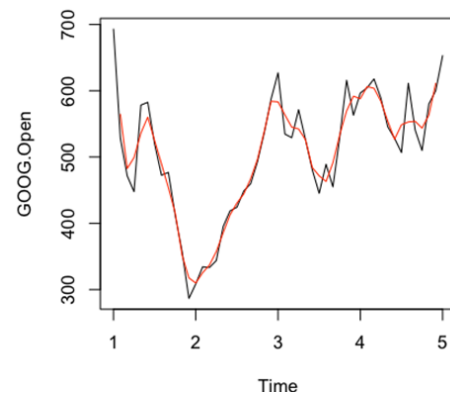
The training and test sets are specified to have consecutive time points. The training set starts at time point 1 and ends at time point 5. The test set that is the next consecutive sets of points after that, [5 to 7-0.01]. This builds training set and applies it to a test set that have consecutive time points that show the trends that I've observed in my data.

Simple moving average

$$Y_t = \frac{1}{2 * k + 1} \sum_{j=-k}^k y_{t+j}$$

```
plot(ts1Train)
lines(ma(ts1Train,order=3),col="red")
```

There are a couple different ways for doing forecasting. One is to do a simple moving average, which in another words, basically averages up all of the values for a particular time point. The prediction will be the average of the previous time points out to a particular time.



Exponential smoothing

Example - simple exponential smoothing

$$\hat{y}_{t+1} = \alpha y_t + (1 - \alpha) \hat{y}_{t-1}$$

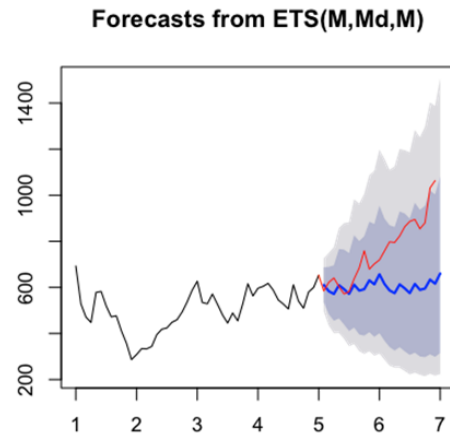
You can also do exponential smoothing. In other words, basically you weight near-by time points as higher values or by more heavily than time points that are farther away. There is a large number of different classes of smoothing models that you can choose.

	Seasonal Component		
Trend	N	A	M
Component	(None)	(Additive)	(Multiplicative)
N (None)	(N,N)	(N,A)	(N,M)
A (Additive)	(A,N)	(A,A)	(A,M)
A _d (Additive damped)	(A _d ,N)	(A _d ,A)	(A _d ,M)
M (Multiplicative)	(M,N)	(M,A)	(M,M)
M _d (Multiplicative damped)	(M _d ,N)	(M _d ,A)	(M _d ,M)

Exponential smoothing

```
ets1 <- ets(ts1Train,model="MMM")
fcast <- forecast(ets1)
plot(fcast); lines(ts1Test,col="red")
```

For exponential smoothing, you can fit a model where you have a different choices for the different types of trends that you might want to fit. And then when you forecast, you can get a prediction that comes out of your forecasting model. And you can also get sort of a prediction bounds for what are the possible values that you could get from that prediction.



Get the accuracy

```
accuracy(fcast,ts1Test)
```

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1	Theil's U
Training set	0.9464	48.78	39.35	-0.3297	7.932	0.3733	0.07298	NA
Test set	156.1890	205.76	160.78	18.1819	18.971	1.5254	0.77025	3.745

You can get the accuracy using the **accuracy** function to get the accuracy of your forecast using your test set. It will give you root mean square to error and other metrics that are more appropriate for forecasting.

Notes and further resources

- [Forecasting and timeseries prediction](#) is an entire field
- Rob Hyndman's [Forecasting: principles and practice](#) is a good place to start
- Cautions
 - Be wary of spurious correlations
 - Be careful how far you predict (extrapolation)
 - Be wary of dependencies over time
- See [quantmod](#) or [quandl](#) packages for finance-related problems.

If you want more information there's an entire field dedicated to forecasting and time series prediction. I would highly recommend Rob Hyndman's [Forecasting: principles and practice](#). This is a free book that's online and it's really, really good, with a lot of information about how to get started in forecasting.

The cautions are to be wary of spurious correlations. Be very careful about how far you predict out into the future with extrapolation. And be wary of dependencies like seasonal effects over time.

For more information on financial prediction and financial forecasting, the [quantmod](#) and [quandl](#) packages are also very useful.

PML_4-4 Unsupervised Prediction

You often know what the labels are for the examples covered so far in this class. In other words, these have been supervised classification, predicting an outcome that you know what it is.

Key ideas

- Sometimes you don't know the labels for prediction
- To build a predictor
 - Create clusters
 - Name clusters
 - Build predictor for clusters
- In a new data set
 - Predict clusters

But sometimes you don't know the labels for prediction so you have to discover those labels in advance. So, one way to do that is to create clusters from the data that you've observed, to add names to those clusters and then build a predictor for those clusters. In a new data set, you're going to predict the cluster and apply the name that you come up with in the previous data set.

This adds several levels of difficulty to the prediction problem. First, creating the clusters is not a perfectly noiseless process. Second, coming up with the right names, that is, interpreting the clusters well is an incredibly challenging problem. If you've taken exploratory data analysis in the data science specialization, you'll understand why.

Building a predictor for the clusters is basically using the algorithms that we've learned throughout the course of this class, as well as predicting on those clusters.

Iris example ignoring species labels

```
data(iris); library(ggplot2)
inTrain <- createDataPartition(y=iris$Species,
                               p=0.7, list=FALSE)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
dim(training); dim(testing)
```

```
[1] 45 5
```

Here is a quick example of how you could do this. Load the `iris` data and the `ggplot` library. Then create a `training` and `testing` set for the `iris` data just like in previous examples but we could ignore the species clusters. One thing that can be done is to perform actually a k-means clustering. Recall *k-means clustering* from the exploratory data analysis section of the data science specialization or look it up on Wikipedia.

Cluster with k-means

```
kMeans1 <- kmeans(subset(training,select=-c(Species)),centers=3)
training$clusters <- as.factor(kMeans1$cluster)
qplot(Petal.Width,Petal.Length,colour=clusters,data=training)
```

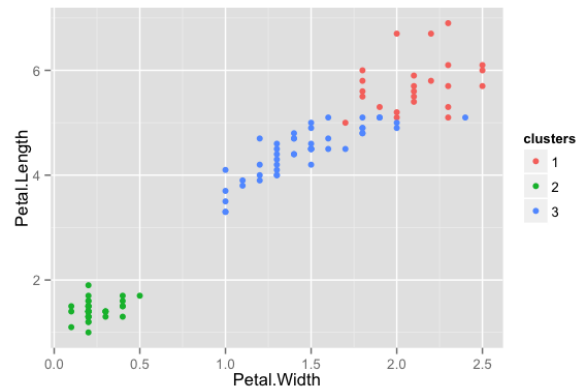
The basic idea here is to create three different clusters by telling it to create three different clusters, ignoring the species information, `kMeans1` then convert to factor. Then assign those clusters to different colors and plot those three different clusters that are created by performing the k-means clustering. Note the red, blue and green clusters on the graphic below. These clusters are relatively

close to the clusters you would get if you used the species labels themselves, but that's not a typical outcome. Often it's very hard to label the clusters that you get.

Compare to real labels

```
table(kMeans1$cluster, training$Species)
```

	setosa	versicolor	virginica
1	0	1	23
2	35	0	0
3	0	34	12



In this case, make a table of the cluster versus species and note that cluster 2 corresponds to the setosa species. Cluster 3 corresponds to the versicolor species and cluster 1 corresponds to the virginica species. So in general, those species names would be unknown and you would have to come up with names for each of the clusters.

Build predictor

```
modFit <- train(clusters ~., data=subset(training, select=-c(Species)), method="rpart")
table(predict(modFit, training), training$Species)
```

	setosa	versicolor	virginica
1	0	0	21
2	35	0	0
3	0	35	14

Then fit a model that relates the cluster variable (**clusters**) that has just been created to all the predictor variables (**~.**). It can be done in the training set, in this case with a classification tree ("**rpart**"). Then do a prediction in a, a training set and note the good result of predicting this green cluster 2 and the blue cluster 3 but cluster 1 and cluster 3 sometimes get mixed up in the prediction model. This is because of both error or variation in the prediction building and error and variation in the cluster building. It ends up being a quite a challenging problem to do unsupervised prediction in this way.

Apply on test

```
testClusterPred <- predict(modFit, testing)
table(testClusterPred, testing$Species)
```

	setosa	versicolor	virginica
1	0	0	13
2	15	0	0
3	0	15	2

Next apply the predictor (**modFit**) on the test data set. In general when predicting on the test data set, the labels are unknown but here the labels are shown. The code is predicting on a new data set and making a table versus the actual known species. The results show the model does a quite reasonable job here of predicting the different species into different clustered labels. In general, this is a quite hard problem, so care must be taken in the labeling the clusters and performing the unsupervised analysis while understanding how that works.

Notes and further reading

- The `c1_predict` function in the `clue` package provides similar functionality
- Beware of over-interpretation of clusters!
- This is one basic approach to [recommendation engines](#)
- [Elements of statistical learning](#)
- [Introduction to statistical learning](#)

The `c1_predict` function in the `clue` package provides similar functionality to what has been described here, but in general it makes a little bit more sense to sort of build your own approach if you're doing unsupervised prediction because you really need to think carefully about how you're going to define the clusters.

Be very wary of over interpretation of this type of clusters. This is in fact an exploratory technique. And so the clusters may change depend on the way that you sample the data.

This is one basic approach to building things like *recommendation engines* where you identify clusters of people that have similar tastes and assign their tastes to new individuals.

You can read more about unsupervised prediction in the [Elements of statistical learning](#) and in the [Introduction to statistical learning](#).