Cross-Validation & Model Selection

NYC Data Science Academy

Outline

- **Overfitting and underfitting**
- Estimating the future performance
- Metrics for measuring performance

Evaluating model performance

The metrics such as:

- MSE or RMSE for regression problems
- Accuracy for classification problems

are used as quantitative measurement for the model performance. However, the result can be misleading if a model is evaluated incorrectly.

To introduce the right process, we need to introduce:

- Training and test data
- Overfitting and underfitting

Training and Testing

Training is the process of searching for the model that best represents the data. Very often, what to be done is:

```
### These are not real code
train(model, X, y)
```

Testing on the other hand is the process to evaluate how well a model represents the data. Very often, what we need to do is:

```
### These are not real code
y_pred = predict(model, X)
error = distance(y_pred, y)
```

Training and Test data

The purpose of most of the supervised learning is to predict for new observations collected in the future.

Training data is used to fit the model. So a model usually performs well on the training data.

To (reasonably) estimate the model performance for the data collected in the future, we usually need another dataset which is prevented from being seen by the model in the training process. This is often called the test data.

Overfitting and Underfitting

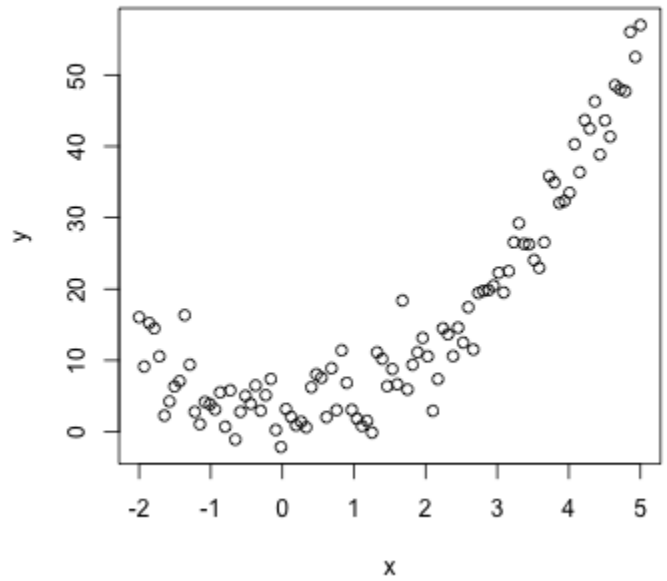
Overfitting: occur when a statistical model describes random error or noise instead of the underlying relationship. Overfitting generally occurs when a model is excessively complex, such as having too many parameters. In this case, we often say that the model has high variance.

Underfitting: occur when a statistical model or machine learning algorithm cannot capture the underlying trend of the data. Intuitively, underfitting occurs when the model or the algorithm does not fit the data well enough. In this case, we often say that the model has high bias.

Performance of Regression

Let's create a toy dataset to illustrate:

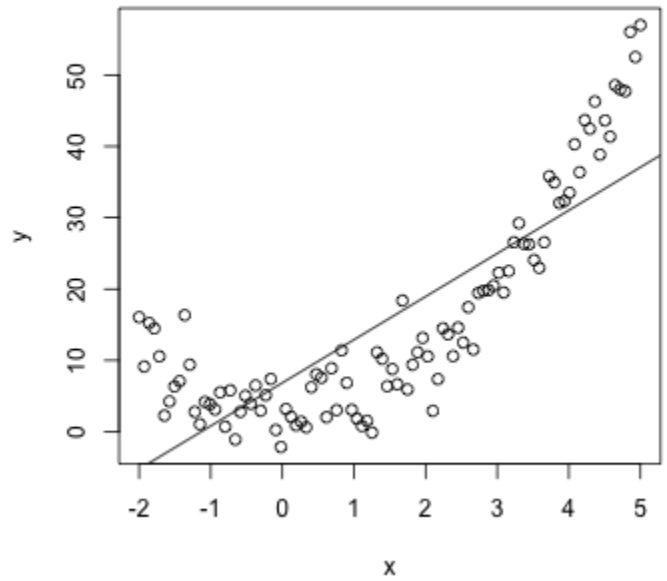
```
set.seed(0); x = seq(-2, 5, length=100)
noise = rnorm(100); y = 3 + 2*x^2 + 4*noise
plot(x, y)
```



Underfit

- Model fails to capture the main pattern.

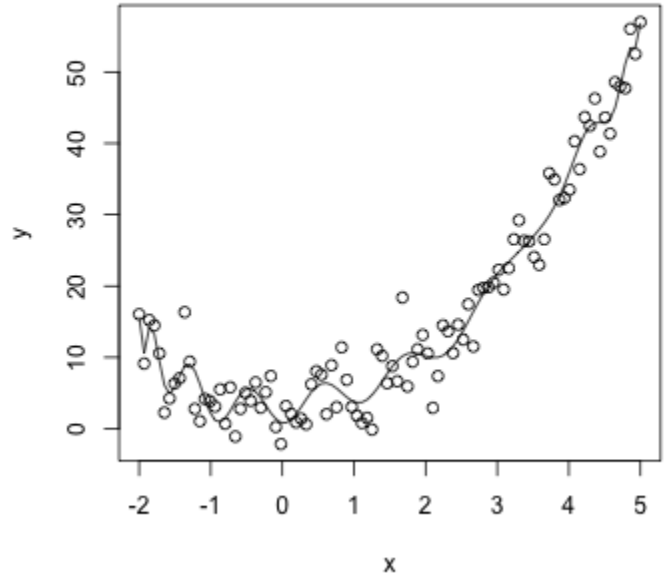
```
plot(x, y)
model1 = lm(y ~ x)
abline(model1)
```



Overfit

- Model captures random fluctuation.

```
model2 = lm(y ~ poly(x,20))
plot(x, y)
lines(x, model2$fitted.values)
```



Learning curve

There is bias-variance tradeoff. Introducing more complexity to a model helps to reduce the bias, but very often it also increases the variance at the meanwhile.

The best model often requires the right amount of complexity in between the two extreme of very high bias and very high variance.

Learning curve

We split the data into two portion:

```
dat = data.frame(x,y)
set.seed(1)
index = sample(1:100, 50)
train = dat[index,]
test = dat[-index,]
```

Learning curve

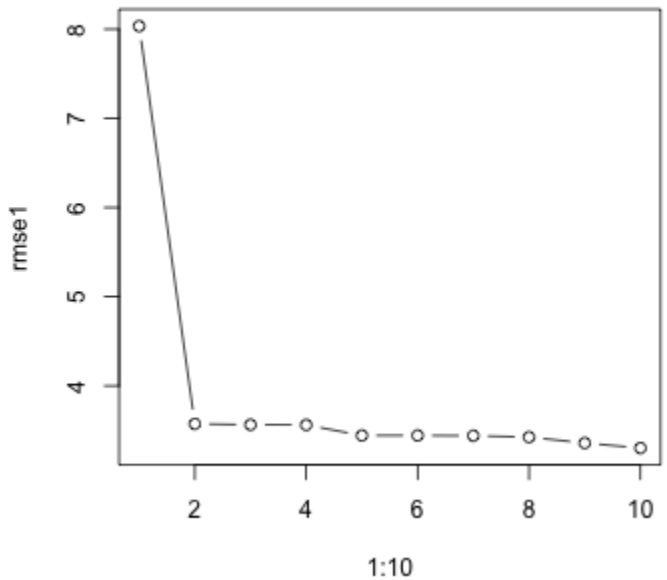
A simple way to introduce complexity is to use higher order polynomials in regression. Below we compute the rmse of regression models with different order in the polynomials:

```
rmse_train = function(n) {
    model = lm(y ~ poly(x,n), data=train)
    pred = predict(model)
    rmse = sqrt(mean((train$y-pred)^2))
    return(rmse)
}
rmse1 = sapply(1:10, rmse_train)
```

Learning curve

Ploting the performance against the complexity results in a learning curve. Below is the learning curve for training data:

```
plot(1:10, rmse1, type='b')
```



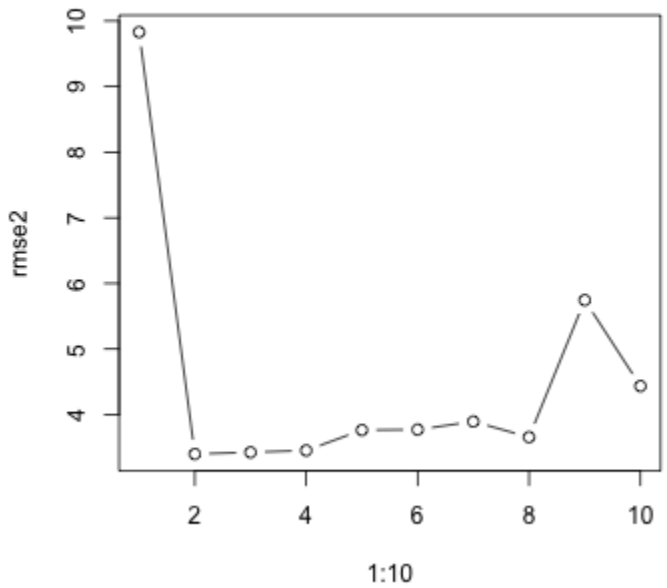
Learning curve

We do the same thing to test data:

```
rmse_test = function(n) {
    model = lm(y ~ poly(x, n), data=train)
    pred = predict(model, newdata=test)
    rmse = sqrt(mean((test$y-pred)^2))
    return(rmse)
}
rmse2 = sapply(1:10, rmse_test)
```

Learning curve

```
plot(1:10, rmse2, type='b')
```



Learning curve

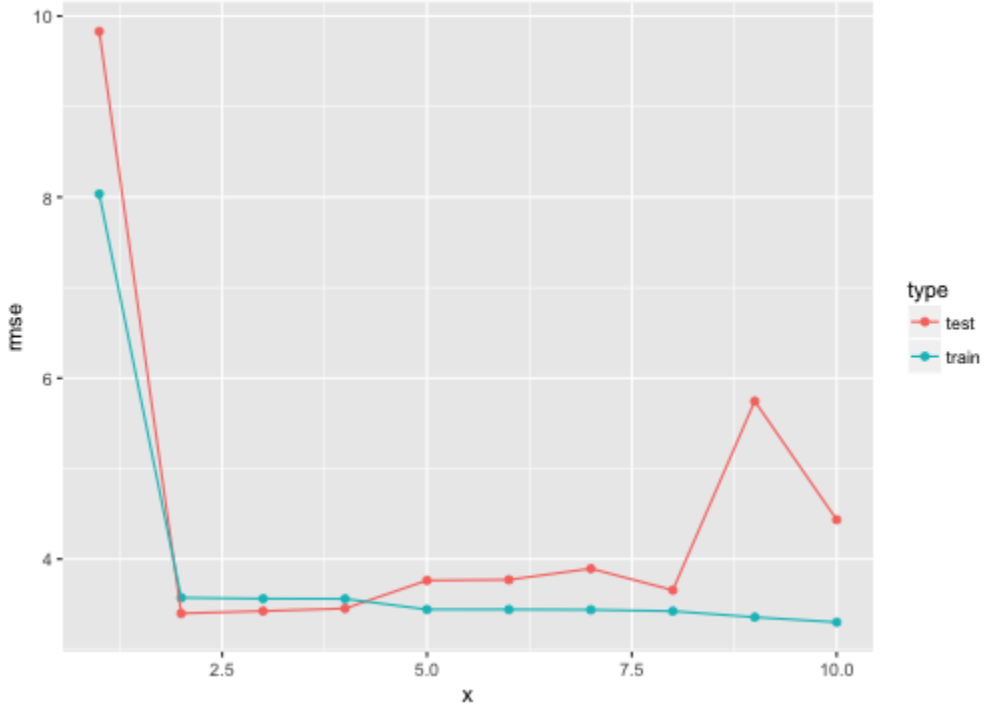
Overfitting can be detected when the learning curves start to deviate:

```
plotdata = data.frame(rmse=c(rmse1,rmse2),
                      type=rep(c('train','test'), each=10),
                      x=rep(1:10,times=2))

library(ggplot2)
p = ggplot(plotdata, aes(x=x,y=rmse,group=type,color=type))
p = p + geom_point() + geom_line()
```

Learning curve

```
print(p)
```



Outline

- Overfitting and underfitting
- **Estimating the future performance**
- Metrics for measuring performance

Model evaluation

We saw that the performance on the training dataset is in general different from that on a test dataset. Since the purpose is to be able to predict, we do want to evaluate a model on a dataset that has never been seen in the training process, namely, the test dataset!

Holdout method

To have a test dataset, it's most intuitive to split the data randomly as we did when we plot the learning curve. The ratio between training and test are often 70% ~ 30% or 80% ~ 20%.

- We will demonstrate the method with the `credit` dataset:

```
credit = read.csv("./data/credit.csv")
View(credit)
```

Holdout method

So the process should be:

```
set.seed(0)
index = sample(1:nrow(credit), size= nrow(credit)*0.7)

### Training
logit = glm(default~., data = credit[index, ], family = 'binomial')

### Testing
prob = predict(logit, credit[-index, ], type="response")
mean((prob>=0.5) == (credit$default[-index]=='yes'))
```

```
## [1] 0.7433333
```

Holdout method

Randomly holding the test portion is simple and intuitive.

- However, is there any potential issue with this method?

Class imbalance

```
y = c(rep('a', 9990), rep('b', 10))

set.seed(2)
index = sample(1:10000, size= 7000)
label_train = y[index]
label_test = y[-index]
print(mean(label_train=='b'))
```

```
## [1] 0.001285714
```

```
print(mean(label_test=='b'))
```

```
## [1] 0.0003333333
```

Full usage of the dataset

The other issue is that the test portion is never used in the training process. So the performance is often underestimated by the holdout method.

The createFolds function

To fully use the whole dataset, cross validation is often implemented.

- Insead of splitting the data into two portions, we partition the data into k folds. Five to ten folds are often used. We use k=5 in our demonstration:

```
library(caret)
folds = createFolds(credit$default, 5)
str(folds)
```

```
## List of 5
##  $ Fold1: int [1:200] 2 3 13 16 19 20 29 33 40 48 ...
##  $ Fold2: int [1:200] 4 5 17 18 25 28 32 38 50 55 ...
##  $ Fold3: int [1:200] 1 6 8 9 10 15 21 26 30 31 ...
##  $ Fold4: int [1:200] 12 22 24 27 39 41 42 49 60 64 ...
##  $ Fold5: int [1:200] 7 11 14 23 36 37 45 46 52 54 ...
```

The createFolds function

Exercise: The `createFolds` function actually helps with the imbalance. Try creating folds with the imbalanced data we had before, confirm that the training and test set have the same distribution of each class.

The training and evaluation

```
n=5
accuracy = numeric(n)

for(i in 1:n){
  index = -folds[[i]]

  logit = glm(default~., data = credit[index, ],
              family = 'binomial')
  prob = predict(logit, credit[-index, ], type="response")
  accuracy[i] = mean(
    (prob>=0.5) == (credit$default[-index]=='yes')
    )

}
accuracy
```

```
## [1] 0.755 0.710 0.750 0.795 0.705
```

The training and evaluation

Very often we use:

- the average of `accuracy` to evaluate the performance.
- the standard deviation of `accuracy` to measure the variability of the model.

```
print(mean(accuracy)); print(sd(accuracy))
```

```
## [1] 0.743
```

```
## [1] 0.03684427
```

Training and evaluation

The package `caret` actually provides a convenient function to proceed with cross validation:

```
ctrl = trainControl(method = "cv", number = 5)
logit_cv = train(default ~ ., data=credit,
                 method = "glm", trControl = ctrl)

logit_cv$results
```

```
##   parameter Accuracy    Kappa AccuracySD   KappaSD
## 1      none    0.749 0.361941  0.0389551 0.1042983
```

Hyperparameters optimizations

Cross validation is often implemented together with grid search to optimize hyperparameters.

- Hyperparameters are the parameters that have to be decided in advance of training and they remain constant throughout the training process. The parameter `lambda` in our shrinkage panelty is an example.

Selecting the hyperparameter for the best performance of a model is called hyperparameter optimization or tuning parameters. The `train` function from `caret` does that automatically.

Hyperparameters optimization

```
ctrl = trainControl(method = "cv", number = 5)
tune.grid = expand.grid(lambda = (0:10)*0.1, alpha=0)
logit_shrinkage = train(default ~ ., data=credit, method = "glmnet",
              metric = "Accuracy", trControl = ctrl,
              preProc=c('center', 'scale'), tuneGrid = tune.grid)
logit_shrinkage$results
```

```
##    alpha lambda Accuracy       Kappa  AccuracySD    KappaSD
## 1      0    0.0    0.751 0.351291247 0.024341323 0.07309308
## 2      0    0.1    0.737 0.251971678 0.019235384 0.05290967
## 3      0    0.2    0.726 0.178698110 0.014747881 0.04184828
## 4      0    0.3    0.722 0.133448331 0.018234583 0.05544787
## 5      0    0.4    0.717 0.095928904 0.017535678 0.06342533
## 6      0    0.5    0.703 0.029285560 0.011510864 0.04017283
## 7      0    0.6    0.704 0.028810830 0.008944272 0.02990123
## 8      0    0.7    0.702 0.017148889 0.006708204 0.02141434
## 9      0    0.8    0.703 0.013907285 0.002738613 0.01269556
## 10     0    0.9    0.702 0.009271523 0.002738613 0.01269556
## 11     0    1.0    0.700 0.000000000 0.000000000 0.00000000
```

Final evaluation

From the cross validation above we see that the best `lambda` is 0, and the accuracy is out 0.75. Does that mean we will have around 75% accuracy for some future dataset?

Final evaluation

From the cross validation above we see that the best `lambda` is 0, and the accuracy is out 0.75. Does that mean we will have around 75% accuracy for some future dataset?

- Notice that portions we used the test the model (which gives us 75%) were actually used to choose `lambda`. Therefore they are not proper for testing anymore.

A common process is shown below:

Hyperparameters optimization

```
set.seed(0)
index = sample(1:nrow(credit), size= nrow(credit)*0.8)
train_data = credit[index, ]
test_data = credit[-index, ]
tune.grid = expand.grid(lambda = (0:10)*0.1, alpha=0)
ctrl = trainControl(method = "repeatedcv", number = 5, repeats = 5)
logit_shrinkage = train(default ~ ., data=train_data,
                        method = "glmnet", metric = "Accuracy",
                        preProc=c('center', 'scale'),
                        trControl = ctrl,  tuneGrid = tune.grid)
```

Final evaluation

From cross validation we see that `lambda = 0` is the best. It can actually be returned:

```
logit_shrinkage$bestTune
```

```
##   alpha lambda
## 1     0      0
```

and the object `m_rf` can be used for prediction:

```
predict.train(logit_shrinkage, test_data)
```

```
##   [1] no  no  no  no  no  yes yes no  no  no  ...
##  [18] no  no  yes no  no  no  no  yes no  no  ...
##  [35] no  no  no  no  no  no  no  yes no  no  ...
##  [52] yes no  no  yes yes no  yes no  no  no  ...
##  [69] no  yes no  yes no  yes no  no  no  yes  ...
##  ... ...
```

Final evaluation

The performance of this final model should be evaluated with the held out test dataset

```
mean(predict.train(logit_shrinkage, test_data)==test_data$default)
```

```
## [1] 0.755
```

Outline

- Overfitting and underfitting
- Estimating the future performance
- **Metrics for measuring performance**

Other metrics

We havce been using:

- MSE or RMSE for regression problems
- Accuracy for classification problems

There are actually a lot more choices that we can use, such as `Kappa` we saw from the `caret` package. We will introduce some common ones below, and we will separate the ones for regression and the ones for classification.

Performance of Regression

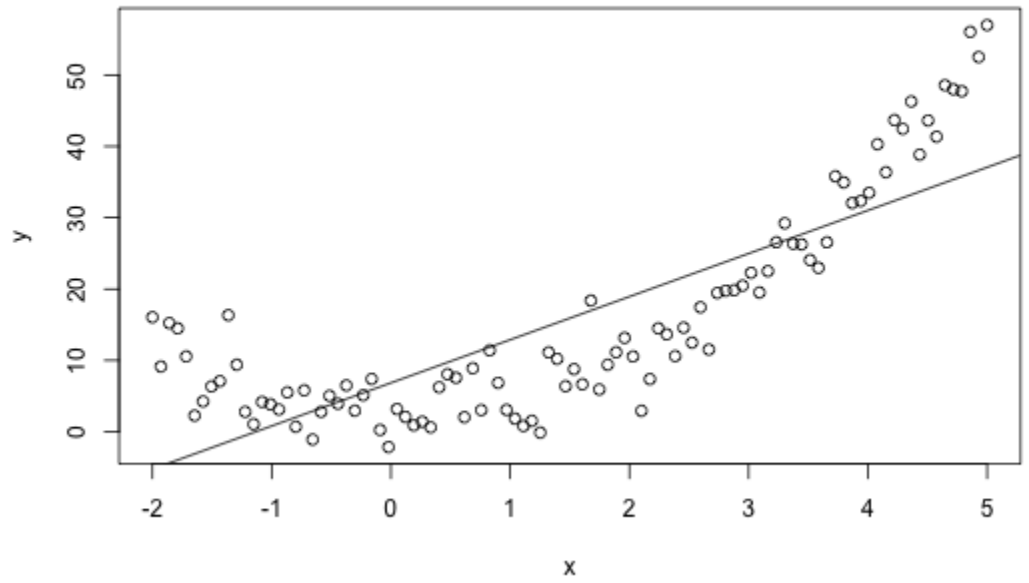
We start with regression problems and recall that we have created a toy dataset.

```
set.seed(0)
x = seq(-2, 5, length=100)
noise = rnorm(100)
y = 3 + 2*x^2 + 4*noise
```

Performance of Regression

Simple linear regression is a typical regression model:

```
model1 = lm(y~x); pred1 = predict(model1)
plot(x, y); model1 = lm(y~x); pred1 = predict(model1)
abline(model1)
```

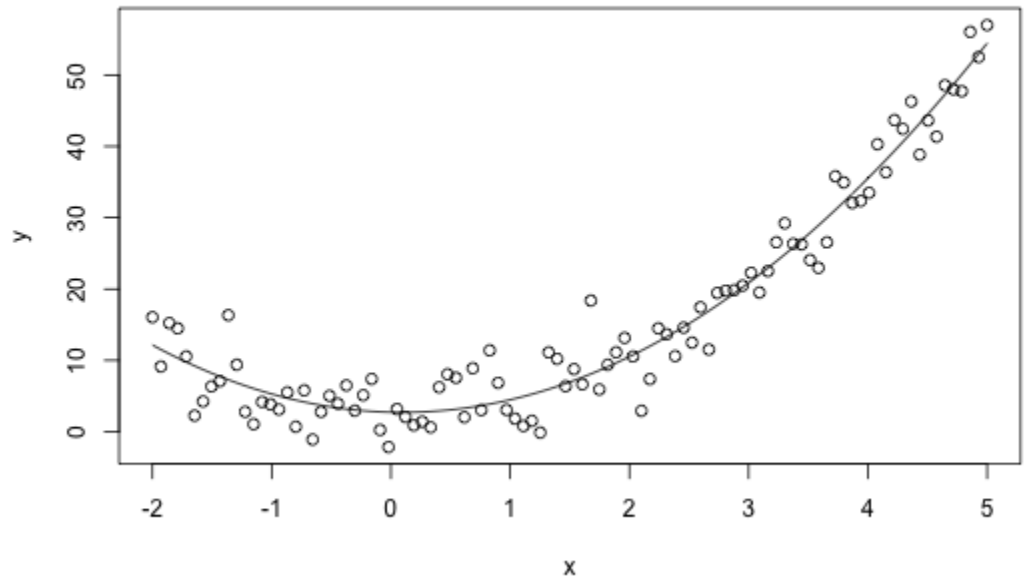


Performance of Regression

We can make the model a little more flexible.

```
model2 = lm(y~x+I(x^2)); pred2 = predict(model2)
```

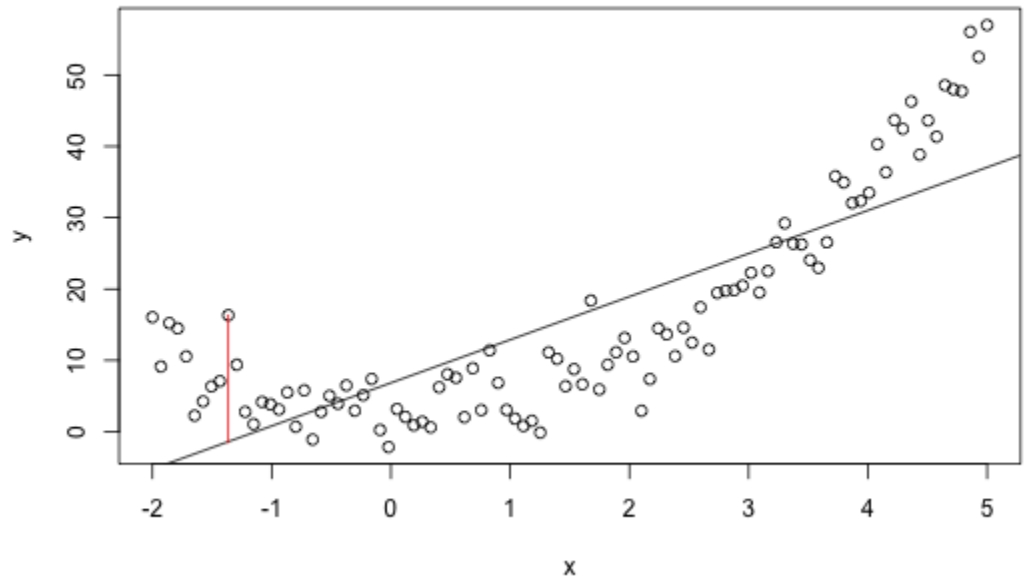
Compare to the last one, which performs better?



Single error:

It's easy to measure the performance of a model at a single point. The difference is simply a vector:

```
y - pred1
```



Aggregating the errors

Below are the common quantity to measure the performance of regression models. All of them are obtained by aggregating single errors.

- MAE
- MSE
- RMSE
- R^2

MAE

Mean absolute error:

```
mae1 = mean(abs(y-pred1))
mae1
```

```
## [1] 7.088283
```

```
mae2 = mean(abs(y-pred2))
mae2
```

```
## [1] 2.787181
```

MSE

Mean square error:

```
mse1 = mean((y-pred1)^2)
mse1
```

```
## [1] 76.16359
```

```
mse2 = mean((y-pred2)^2)
mse2
```

```
## [1] 12.0212
```

RMSE

Root mean square error:

```
rmse1 = sqrt(mean((y-pred1)^2))
rmse1
```

```
## [1] 8.727175
```

```
rmse2 = sqrt(mean((y-pred2)^2))
rmse2
```

```
## [1] 3.467161
```

R squared

The coefficient of determination: R squared

```
r2_1 = 1 - mean((y-pred1)^2)/mean((y-mean(y))^2)
r2_1
```

```
## [1] 0.6664287
```

```
r2_2 = 1 - mean((y-pred2)^2)/mean((y-mean(y))^2)
r2_2
```

```
## [1] 0.9473511
```

Performance of classification

- Confusion Matrix
  - Accuracy
  - Sensivitity
  - Specificity
  - Precision
  - Kappa
- ROC and AUC

Performance of classification

Three elements are essential to evaluate a classifier:

- Actual class values
- Predicted class values
- Estimated probabilities of the prediction

Performance of classification

We will demonstrate the evaluation with the data below:

```
dat = read.csv('data/school.csv',
                colClasses = c('factor','factor','numeric','factor'))
model = glm(y ~ . , dat, family='binomial')
pred.prob = predict(model, type='response')
pred.class = ifelse(pred.prob > 0.5, 1, 0)
```

Confusion Matrix

```
cmat = table(dat$y, pred.class)
row.names(cmat) = c('actual_0','actual_1')
cmat
```

```
##            pred.class
##               0   1
##   actual_0 589  76
##   actual_1 117 218
```

Can we identify the items below in the confusion matrix?

- True Positive (TP): Correctly classified as the class of interest
- True Negative (TN): Correctly classified as not the class of interest
- False Positive (FP): Incorrectly classified as the class of interest. It's often called 'Type I' error.
- False Negative (FN): Incorrectly classified as not the class of interest. It's often called 'Type II' error.

Accuracy

The measure of accuracy is the proportion of correct classifications out of total classifications.

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$$

```
accuracy = (cmat[2,2] + cmat[1,1]) / sum(cmat)
print(accuracy)
```

```
## [1] 0.807
```

```
error.rate = 1 - accuracy
print(error.rate)
```

```
## [1] 0.193
```

Sensitivity/Recall

The measure of sensitivity is the proportion of positive examples that were correctly classified.

$$\text{sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Note From the formula we also see that sensitivity is actually the true positive rate.

```
sensitivity = cmat[2,2] / (cmat[2,1] + cmat[2,2])
sensitivity
```

```
## [1] 0.6507463
```

Specificity

The measure of specificity is the proportion of negative examples that were correctly classified.

$$\text{specificity} = \frac{TN}{TN + FP}$$

Note From the formula we also see that specificity is actually the true negative rate.

```
specificity = cmat[1,1] / (cmat[1,1] + cmat[1,2])
specificity
```

```
## [1] 0.8857143
```

Precision

The measure of precision is the proportion of positive examples that are truly positive.

$$\text{precision} = \frac{TP}{TP + FP}$$

```
precision = cmat[2,2] / (cmat[1,2] + cmat[2,2])
precision
```

```
## [1] 0.7414966
```

F-score

There is often trade-off between precision and recall. To strike a balance one often optimizes the F-score:

$$\text{F-score} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

```
fscore = (2 * precision*recall)/(recall + precision)
fscore
```

```
## [1] 0.6931638
```

Kappa statistics

The kappa statistic adjusts accuracy by accounting for the possibility of a correct prediction by chance alone.

```
pmat = cmat / sum(cmat)
p = addmargins(pmat)
p
```

```
##              pred.class
##                   0     1   Sum
##    actual_0 0.589 0.076 0.665
##    actual_1 0.117 0.218 0.335
##    Sum      0.706 0.294 1.000
```

Kappa statistics

The table we saw in the previous page was summarized from what we actually observed. If instead, we make the prediction randomly, what should the values in the table be?

```
ep
```

```
##              pred.class
##                0     1     Sum
##    actual_0 ?     ?     0.665
##    actual_1 ?     ?     0.335
##    Sum      0.706 0.294 1
```

Kappa statistics

The expected probability can be computed as below:

```
ep[1,1] = ep[1,3] * ep[3,1]
ep[1,2] = ep[1,3] * ep[3,2]
ep[2,1] = ep[2,3] * ep[3,1]
ep[2,2] = ep[2,3] * ep[3,2]
ep
```

```
##              pred.class
##                       0       1     Sum
##    actual_0 0.46949 0.19551 0.66500
##    actual_1 0.23651 0.09849 0.33500
##    Sum      0.70600 0.29400 1.00000
```

Kappa statistics

The probability of our model to be correct (accuracy) is:

```
pr.actual = p[1,1] + p[2,2]
pr.actual
```

```
## [1] 0.807
```

The probability to be correct with a random model:

```
pr.expected = ep[1,1] + ep[2,2]
pr.expected
```

```
## [1] 0.56798
```

Kappa statistics

Kappa statistics indicates how much our model improves from a random model. Note that κ is standardized by the range to possible improvement.

$$\kappa = \frac{\text{pr.actual} - \text{pr.expected}}{1 - \text{pr.expected}}$$

- Poor agreement $\approx$ Less than 0.20
- Fair agreement $\approx$ 0.20 to 0.40
- Moderate agreement $\approx$ 0.40 to 0.60
- Good agreement $\approx$ 0.60 to 0.80
- Very good agreement $\approx$ 0.80 to 1.00

Confusion matrix with caret

```
confusionMatrix(pred.class, dat$y, positive='1')
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 589 117
##          1  76 218
##
##                Accuracy : 0.807
##                  95% CI : (0.7811, 0.831)
##     No Information Rate : 0.665
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5533
##  Mcnemar's Test P-Value : 0.003986
##
##             Sensitivity : 0.6507
##             Specificity : 0.8857
##          Pos Pred Value : 0.7415
```

ROC

- If the target variable is binary categorical, in addition to a confusion matrix, we can also evaluate the model with what's called an ROC curve.
- Many models allow the prediction output to be the probability. The confusion matrix takes 0.5 as the critical point to determine which samples are positive and which are negative, meanwhile the sensitivity TPR and the specificity TNR can be calculated.
- In addition to the training parameters of models, the selection of critical point will also influence TPR and TNR greatly. Sometimes you can select the critical point according to the specific problem and needs.
- If selecting a series of critical points, we will get the corresponding TPR and TNR. Concatenate these points represented by TPR and TNR to get the ROC curve.
- The ROC curve can help us clearly understand the performance of a classifier, and facilitates the performance comparison of different classifiers.

Building the ROC

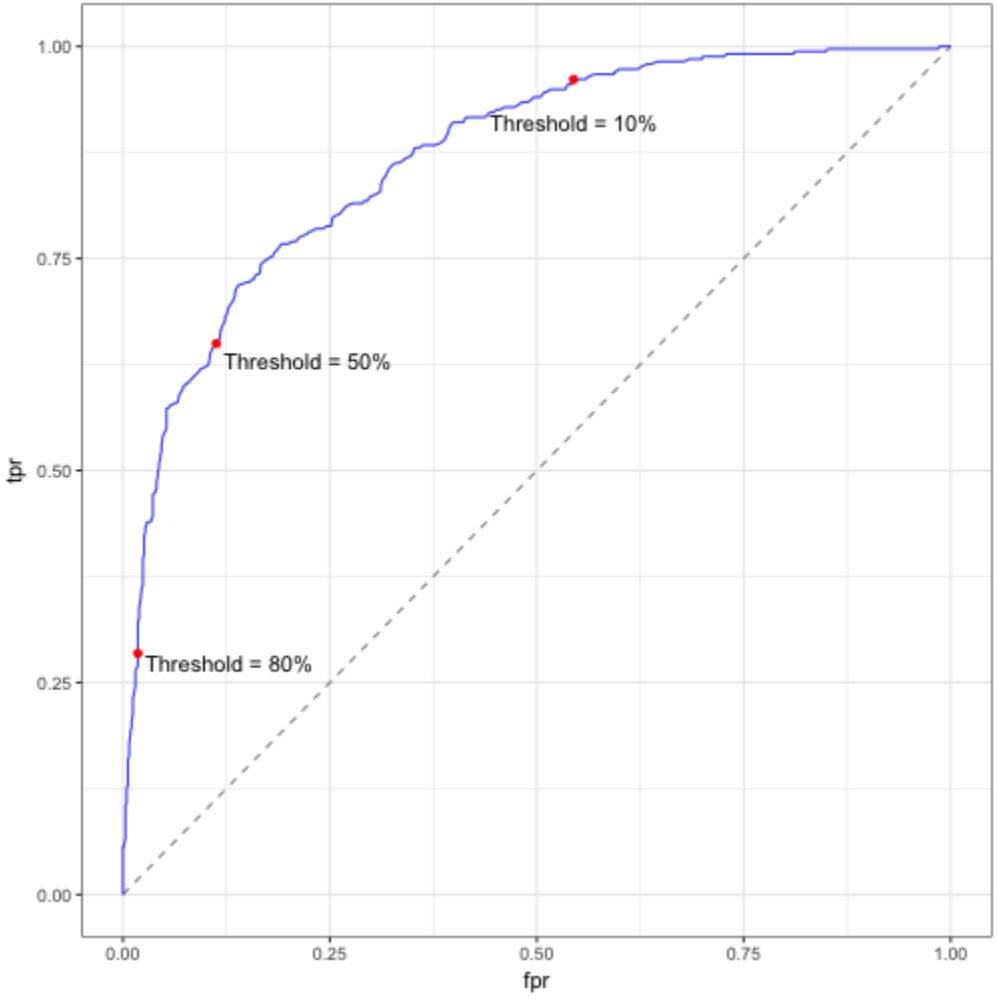
```
#prediction probability prob and the actual results
preObs = data.frame(prob=pred.prob, obs=dat$y)
#sort descending according to the predicted probability
preObs = preObs[order(-preObs$prob), ]
head(preObs)
```

```
##          prob obs
## 688 0.9880548   1
## 595 0.9870100   1
## 504 0.9861706   1
## 567 0.9810796   1
## 682 0.9790099   1
## 692 0.9662373   1
```

Building the ROC

```
n = nrow(preObs)
tpr = fpr = rep(0,n)
#calculate TPR and FPR according to different thresholds;
#draw ROC curve
for (i in 1:n) {
    threshold = preObs$prob[i]
    tp = sum(preObs$prob > threshold & preObs$obs == 1)
    fp = sum(preObs$prob > threshold & preObs$obs == 0)
    tn = sum(preObs$prob < threshold & preObs$obs == 0)
    fn = sum(preObs$prob < threshold & preObs$obs == 1)
    tpr[i] = tp / (tp + fn) # true positive rate (sensitivity)
    fpr[i] = fp / (tn + fp) # false positive rate (1-specificity)
}
```

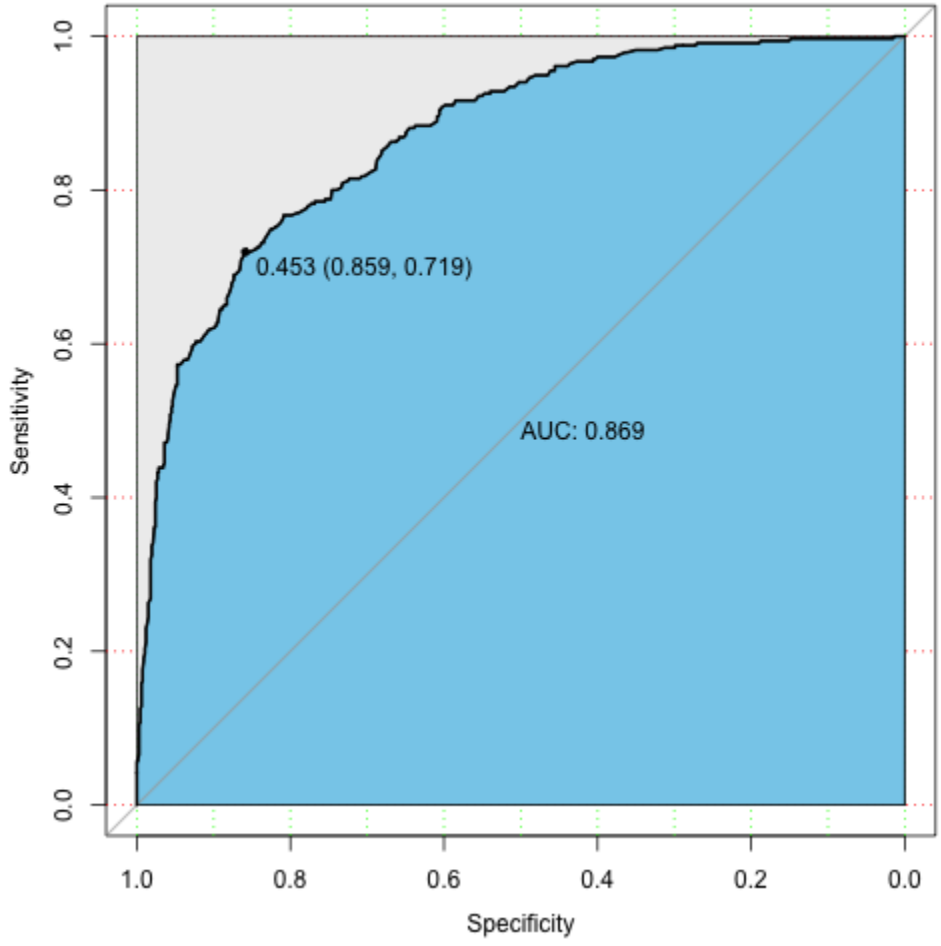
Building the ROC



AUC ROC

```
library(pROC)
modelroc = roc(preObs$obs, preObs$prob)
plot(modelroc, print.auc=TRUE, auc.polygon=TRUE,
     grid=c(0.1, 0.2), grid.col=c("green", "red"),
     max.auc.polygon=TRUE, auc.polygon.col="skyblue", print.thres=TRU
```

AUC ROC


0.453 (0.859, 0.719)
AUC: 0.869
Sensitivity
Specificity
1.0
0.8
0.6
0.4
0.2
0.0

AUC ROC

The object `modelroc` also returns quantities of interest:

```
auc(modelroc)
```

```
## Area under the curve: 0.8687
```

```
ci(modelroc)
```

```
## 95% CI: 0.8457-0.8918 (DeLong)
```