Lecture 13: Classification

Announcements

- Reading
  - Chapter 24
  - Section 5.3.2 (list comprehension)
- Course evaluations
  - Online evaluation now through noon on Friday, December 16

Supervised Learning

- Regression
  - Predict a real number associated with a feature vector
  - E.g., use linear regression to fit a curve to data
- *Classification*
  - Predict a discrete value (label) associated with a feature vector

An Example (similar to earlier lecture)

Features **Label**

Name	Egg-laying	Scales	Poisonous	Cold-blooded	Number legs	Reptile
Cobra	1	1	1	1	0	1
Rattlesnake	1	1	1	1	0	1
Boa constrictor	0	1	0	1	0	1
Chicken	1	1	0	1	2	0
Guppy	0	1	0	0	0	0
Dart frog	1	0	1	0	4	0
Zebra	0	0	0	0	4	0
Python	1	1	0	1	0	1
Alligator	1	1	0	1	4	1

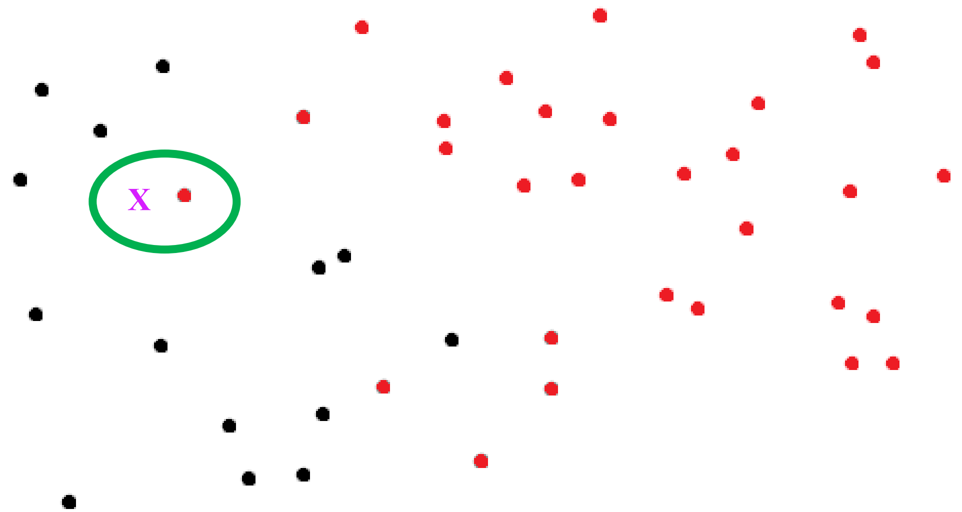
Distance Matrix

	cobra	rattlesnake	boa constrictor	chicken	guppy	dart frog	zebra	python	alligator
cobra	--	0.0	1.414	2.236	1.732	1.732	2.236	1.0	1.414
rattlesnake	0.0	--	1.414	2.236	1.732	1.732	2.236	1.0	1.414
boa constrictor	1.414	1.414	--	2.236	1.0	2.236	1.732	1.0	1.414
chicken	2.236	2.236	2.236	--	2.449	2.0	2.0	2.0	1.0
guppy	1.732	1.732	1.0	2.449	--	2.0	1.414	1.414	1.732
dart frog	1.732	1.732	2.236	2.0	2.0	--	1.414	2.0	1.732
zebra	2.236	2.236	1.732	2.0	1.414	1.414	--	2.0	1.732
python	1.0	1.0	1.0	2.0	1.414	2.0	2.0	--	1.0
alligator	1.414	1.414	1.414	1.0	1.732	1.732	1.732	1.0	--

Code for producing this table posted

Using Distance Matrix for Classification

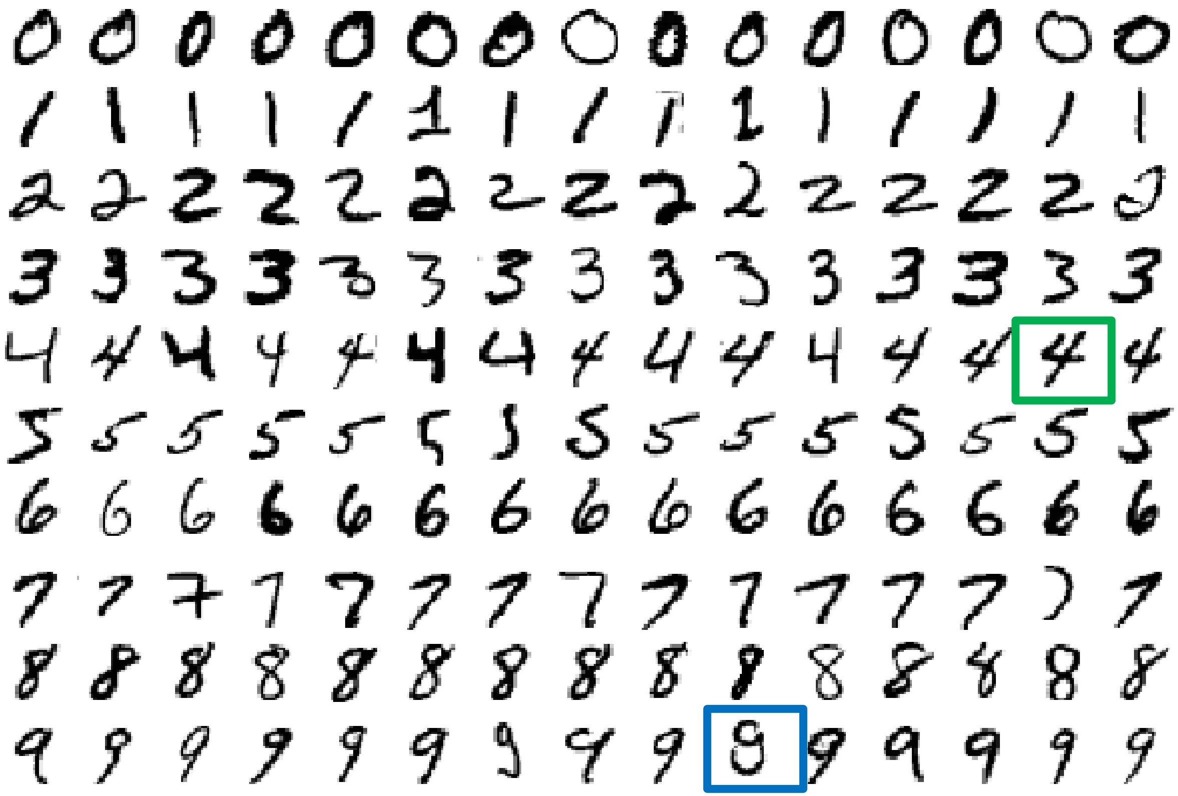
- Simplest approach is probably nearest neighbor
- Remember training data
- When predicting the label of a new example
  - Find the nearest example in the training data
  - Predict the label associated with that example



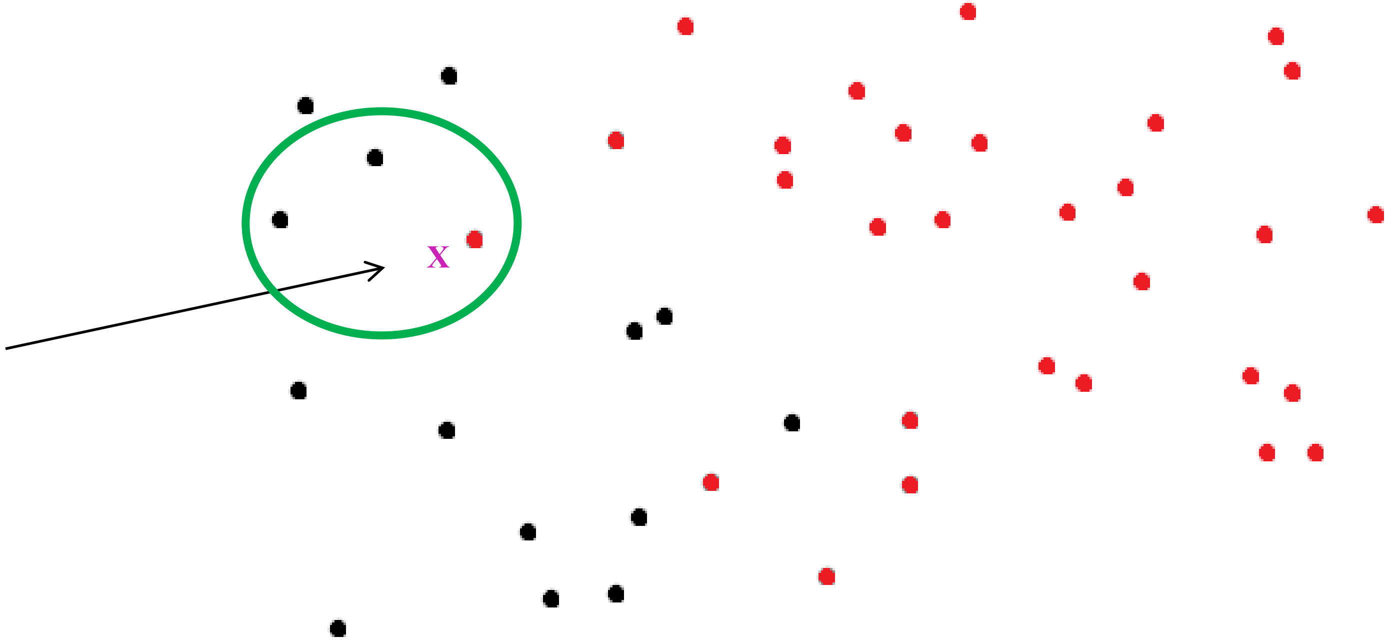
Distance Matrix

	cobra	rattlesnake	boa constrictor	chicken	guppy	dart frog	zebra	python	alligator	Label
cobra	--	0.0	1.414	2.236	1.732	1.732	2.236	1.0	1.414	R
rattlesnake	0.0	--	1.414	2.236	1.732	1.732	2.236	1.0	1.414	R
boa constrictor	1.414	1.414	--	2.236	1.0	2.236	1.732	1.0	1.414	R
chicken	2.236	2.236	2.236	--	2.449	2.0	2.0	2.0	1.0	~R
guppy	1.732	1.732	1.0	2.449	--	2.0	1.414	1.414	1.732	~R
dart frog	1.732	1.732	2.236	2.0	2.0	--	1.414	2.0	1.732	~R
zebra	2.236	2.236	1.732	2.0	1.414	1.414				
python	1.0	1.0	1.0	2.0	1.414	2.0				
alligator	1.414	1.414	1.414	1.0	1.732	1.732				

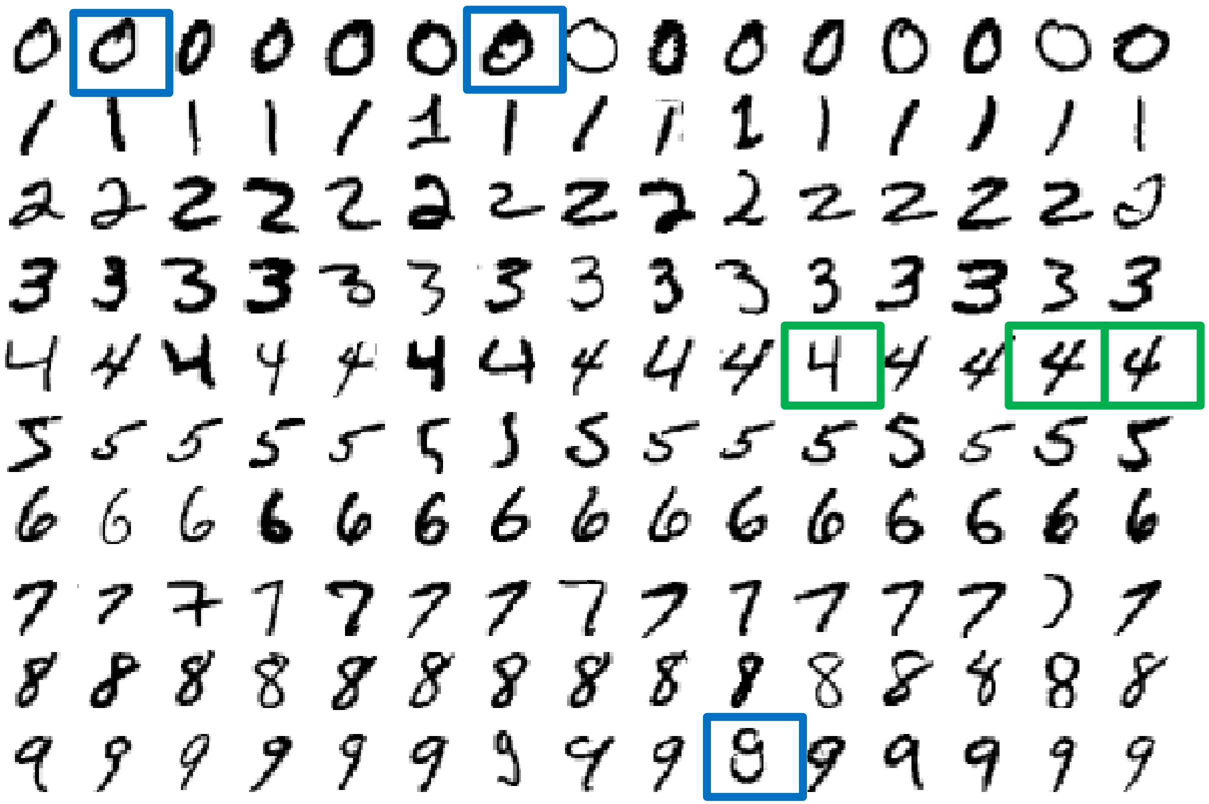
An Example



K-nearest Neighbors



An Example



Advantages and Disadvantages of KNN

- Advantages
  - Learning fast, no explicit training
  - No theory required
  - Easy to explain method and results
- Disadvantages
  - Memory intensive and predictions can take a long time
    - Are better algorithms than brute force
  - No model to shed light on process that generated data

The Titanic Disaster

- RMS Titanic sank in the North Atlantic the morning of 15 April 1912, after colliding with an iceberg. Of the 1,300 passengers aboard, 812 died. (703 of 918 crew members died.)
- Database of 1046 passengers
  - Cabin class
    - 1st, 2nd, 3rd
  - Age
  - Gender

Is Accuracy Enough

- If we predict “died”, accuracy will be >62% or passenger and >76% for crew members
- Consider a disease that occurs in 0.1% of population
  - Predicting disease-free has an accuracy of 0.999

Other Metrics

$$sensitivity = \frac{true\ positive}{true\ positive + false\ negative}$$

$$specificity = \frac{true\ negative}{true\ negative + false\ positive}$$

$$positive\ predictive\ value = \frac{true\ positive}{true\ positive + false\ positive}$$

$$negative\ predictive\ value = \frac{true\ negative}{true\ negative + false\ negative}$$

sensitivity = recall
specificity = precision

Testing Methodology Matters

- Leave-one-out
- Repeated random subsampling

Leave-one-out

```
def leaveOneOut(examples, method, toPrint = True):
    truePos, falsePos, trueNeg, falseNeg = 0, 0, 0, 0
    for i in range(len(examples)):
        testCase = examples[i]
        trainingData = examples[0:i] + examples[i+1:]
        results = method(trainingData, [testCase])
        truePos += results[0]
        falsePos += results[1]
        trueNeg += results[2]
        falseNeg += results[3]
    if toPrint:
        getStats(truePos, falsePos, trueNeg, falseNeg)
    return truePos, falsePos, trueNeg, falseNeg
```

Repeated Random Subsampling

```
def split80_20(examples):
    sampleIndices = random.sample(range(len(examples)),
                                  len(examples)//5)
    trainingSet, testSet = [], []
    for i in range(len(examples)):
        if i in sampleIndices:
            testSet.append(examples[i])
        else:
            trainingSet.append(examples[i])
    return trainingSet, testSet
```

Repeated Random Subsampling

```
def randomSplits(examples, method, numSplits, toPrint = True):
    truePos, falsePos, trueNeg, falseNeg = 0, 0, 0, 0
    random.seed(0)
    for t in range(numSplits):
        trainingSet, testSet = split80_20(examples)
        results = method(trainingSet, testSet)
        truePos += results[0]
        falsePos += results[1]
        trueNeg += results[2]
        falseNeg += results[3]
    getStats(truePos/numSplits, falsePos/numSplits,
             trueNeg/numSplits, falseNeg/numSplits, toPrint)
    return truePos/numSplits, falsePos/numSplits,\
             trueNeg/numSplits, falseNeg/numSplits
```

Let's Try KNN

```
def KNearestClassify(training, testSet, label, k):
    """Assumes training & testSet lists of examples, k an int
       Predicts whether each example in testSet has label
       Returns number of true positives, false positives,
          true negatives, and false negatives"""

knn = lambda training, testSet:\
             KNearestClassify(training, testSet,
                              'Survived', 3)
numSplits = 10
print('Average of', numSplits,
      '80/20 splits using KNN (k=3)')
truePos, falsePos, trueNeg, falseNeg =\
      randomSplits(examples, knn, numSplits)

print('Average of LOO testing using KNN (k=3)')
truePos, falsePos, trueNeg, falseNeg =\
      leaveOneOut(examples, knn)
```

Results

Average of 10 80/20 splits using KNN (k=3)

Accuracy = 0.766

Sensitivity = 0.67

Specificity = 0.836

Pos. Pred. Val. = 0.747

Average of LOO testing using KNN (k=3)

Accuracy = 0.769

Sensitivity = 0.663

Specificity = 0.842

Pos. Pred. Val. = 0.743

Considerably better than 62%

Not much difference between experiments

Logistic Regression

- Analogous to linear regression
- Designed explicitly for predicting probability of an event
  - Dependent variable can only take on a finite set of values
    - Usually 0 or 1
- Finds weights for each feature
  - Positive implies variable positively correlated with outcome
  - Negative implies variable negatively correlated with outcome
  - Absolute magnitude related to strength of the correlation
- Optimization problem a bit complex, key is use of a log function—won't make you look at it

Class LogisticRegression

```
import sklearn.linear_model
```

`fit`(sequence of feature vectors, sequence of labels)

Returns object of type LogisticRegression

`coef_`

Returns weights of features

`predict_proba`(feature vector)

Returns probabilities of labels

Building a Model

```
def buildModel(examples, toPrint = True):
    featureVecs, labels = [],[]
    for e in examples:
        featureVecs.append(e.getFeatures())
        labels.append(e.getLabel())
    LogisticRegression = sklearn.linear_model.LogisticRegression
    model = LogisticRegression().fit(featureVecs, labels)
    if toPrint:
        ...
    return model
```

Applying Model

```
def applyModel(model, testSet, label, prob = 0.5):
    testFeatureVecs = [e.getFeatures() for e in testSet]
    probs = model.predict_proba(testFeatureVecs)
    truePos, falsePos, trueNeg, falseNeg = 0, 0, 0, 0
    for i in range(len(probs)):
        if probs[i][1] > prob:
            if testSet[i].getLabel() == label:
                truePos += 1
            else:
                falsePos += 1
        else:
            if testSet[i].getLabel() != label:
                trueNeg += 1
            else:
                falseNeg += 1
    return truePos, falsePos, trueNeg, falseNeg
```

List Comprehension

`expr` **for** `id` **in** `L`

Creates a list by evaluating `expr` `len(L)` times with `id` in `expr` replaced by each element of `L`

```
L = [x*x for x in range(10)]
print(L)
L = [x*x for x in range(10) if x%2 == 0]
print(L)
```

Applying Model

```
def applyModel(model, testSet, label, prob = 0.5):
    testFeatureVecs = [e.getFeatures() for e in testSet]
    probs = model.predict_proba(testFeatureVecs)
    truePos, falsePos, trueNeg, falseNeg = 0, 0, 0, 0
    for i in range(len(probs)):
        if probs[i][1] > prob:
            if testSet[i].getLabel() == label:
                truePos += 1
            else:
                falsePos += 1
        else:
            if testSet[i].getLabel() != label:
                trueNeg += 1
            else:
                falseNeg += 1
    return truePos, falsePos, trueNeg, falseNeg
```

Putting It Together

```
def lr(trainingData, testData, prob = 0.5):
    model = buildModel(trainingData, False)
    results = applyModel(model, testData, 'Survived', prob)
    return results

numSplits = 10
print('Average of', numSplits, '80/20 splits LR')
truePos, falsePos, trueNeg, falseNeg =\
      divide80_20(examples, lr, numSplits)

print('Average of LOO testing using LR')
truePos, falsePos, trueNeg, falseNeg =\
      leaveOneOut(examples, lr)
```

Results

Average of 10 80/20 splits LR
Accuracy = 0.804
Sensitivity = 0.719
Specificity = 0.859
Pos. Pred. Val. = 0.767

Average of LOO testing using LR
Accuracy = 0.786
Sensitivity = 0.705
Specificity = 0.842
Pos. Pred. Val. = 0.754

Compare to KNN Results

Average of 10 80/20 splits using KNN (k=3)
Accuracy = 0.744
Sensitivity = 0.629
Specificity = 0.829
Pos. Pred. Val. = 0.728

Average of LOO testing using KNN (k=3)
Accuracy = 0.769
Sensitivity = 0.663
Specificity = 0.842
Pos. Pred. Val. = 0.743

Average of 10 80/20 splits LR
Accuracy = 0.804
Sensitivity = 0.719
Specificity = 0.859
Pos. Pred. Val. = 0.767

Average of LOO testing using LR
Accuracy = 0.786
Sensitivity = 0.705
Specificity = 0.842
Pos. Pred. Val. = 0.754

Performance not much difference
Logistic regression slightly better

Also provides insight about variables

Looking at Feature Weights

```
def buildModel(examples, toPrint = True):
    ...
    if toPrint:
        print('model.classes_ =', model.classes_)
        for i in range(len(model.coef_)):
            print('For label', model.classes_[1])
            for j in range(len(model.coef_[0])):
                print('   ', Passenger.featureNames[j], '=',
                      model.coef_[0][j])
    return model

buildModel(examples, True)
```

model.classes_ = ['Died' 'Survived']
For label Survived
  C1 = 1.66761946545
  C2 = 0.460354552452
  C3 = -0.50338282535
  age = -0.0314481062387
  male gender = -2.39514860929

Be wary of reading too much into the weights

Features are often correlated

Changing the Cutoff

```
random.seed(0)
trainingSet, testSet = split80_20(examples)
model = buildModel(trainingSet, False)
print('Try p = 0.1')
truePos, falsePos, trueNeg, falseNeg =\
                   applyModel(model, testSet, 'Survived', 0.1)
getStats(truePos, falsePos, trueNeg, falseNeg)
print('Try p = 0.9')
truePos, falsePos, trueNeg, falseNeg =\
                   applyModel(model, testSet, 'Survived', 0.9)
getStats(truePos, falsePos, trueNeg, falseNeg)
```

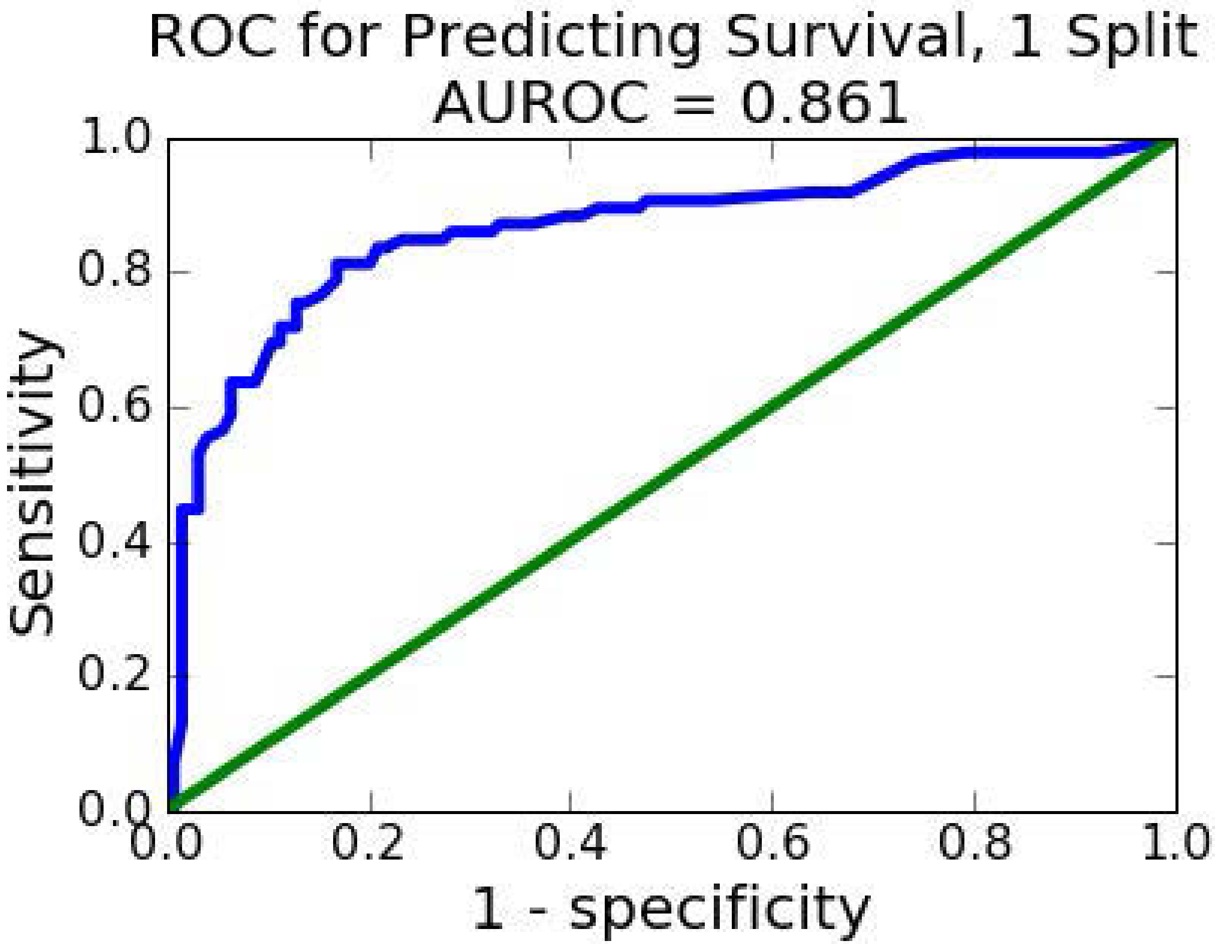
Try p = 0.1
 Accuracy = 0.493
 Sensitivity = 0.976
 Specificity = 0.161
 Pos. Pred. Val. = 0.444

Try p = 0.9
 Accuracy = 0.656
 Sensitivity = 0.176
 Specificity = 0.984
 Pos. Pred. Val. = 0.882

ROC (Receiver Operating Characteristic)

```
def buildROC(trainingSet, testSet, title, plot = True):
    model = buildModel(trainingSet, True)
    xVals, yVals = [], []
    p = 0.0
    while p <= 1.0:
        truePos, falsePos, trueNeg, falseNeg =\
                               applyModel(model, testSet,
                               'Survived', p)
        xVals.append(1.0 - specificity(trueNeg, falsePos))
        yVals.append(sensitivity(truePos, falseNeg))
        p += 0.01
    auroc = sklearn.metrics.auc(xVals, yVals, True)
    if plot:
        ...
    return auroc
```

Output


ROC for Predicting Survival, 1 Split
AUROC = 0.861
Sensitivity
1 - specificity
0.0
0.2
0.4
0.6
0.8
1.0