Evaluating algorithms and kNN

Training set and test set

- **Training:** used to train classifier, and create the model
- **Test:** used to evaulate the model

NEVER test model using training data! Often yields overly optimistic results. Might make sense to split data into independent sets (as seen before), train on one, and test with another

10-Fold Cross Validation

Common data science technique where data is split into **10 distinct groups**. At each phase, **9 of the 10 partitions** are used to train the model while **the other partition is used as our test set** and then **average the results**

This method can yield **different results** based on the partitions, which can be suboptimal.

n-Fold Cross Validation (Leave-one-out)

Same approach as last time, except each iteration we leave **one datapoint** out of our training set, and then test on this datapoint. Benefit of this approach is that it's **deterministic**, meaning you will get the same results every time.

Drabacks of leave-one-out:

- **Huge** computational cost to this method.
- **Stratification:** sample in each *bucket* should be representative of **entire population!** LOO does not satisfy this stratification, since it necessarily leaves only one data element out.

Confusion Matrix

Matrix with "actual class" as the rows and "predicted class" as the columns. This means the **diagnols represent correct predictions** while the other values are incorrect.

Straighforward to look at confusion matrix and get a glimpse at **where your classifier is failing**.

Lets grab the classifier code from last chapter, modify it to perform 10-fold cross validation, and show a confusion matrix.

In [199]:
```
class Classifier:

    #add "K" parameter
    def __init__(self, bucketPrefix, testBucketNumber, dataFormat, k=1):
        self.k = k

        #same stuff as always
        self.medianAndDeviation = []
        self.data = []

        #load and format datatypes per column
        self.format = dataFormat.strip().split(' ')

        ##load in each training set that ISNT OUR TEST BUCKET!
        for i in range(1,11):
            if i != testBucketNumber:
                filename = "%s-%02i" % (bucketPrefix, i)
                # reading the data in from the file
                f = open(filename)
                lines = f.readlines()
                f.close()
                self.data = []
                for line in lines[1:]:
                    fields = line.strip().split('\t')
                    ignore = []
                    vector = []
                    for i in range(len(fields)):
                        if self.format[i] == 'num':
                            vector.append(float(fields[i]))
                        elif self.format[i] == 'comment':
                            ignore.append(fields[i])
                        elif self.format[i] == 'class':
                            classification = fields[i]
                    self.data.append((classification, vector, ignore))
        self.rawData = list(self.data)
        # get length of instance vector
        self.vlen = len(self.data[0][1])
        # now normalize the data
        for i in range(self.vlen):
            self.normalizeColumn(i)


    #takes test bucket information and runs it based on curre
    def testBucket(self, bucketPrefix, bucketNumber):
        #load test file
        filename = "%s-%02i" % (bucketPrefix, bucketNumber)
        f = open(filename)
```

```
        lines = f.readlines()
        totals = {}
        f.close()
        #load data for each line
        for line in lines:
            data = line.strip().split('\t')
            vector = []
            classInColumn = -1
            for i in range(len(self.format)):
                if self.format[i] == 'num':
                    vector.append(float(data[i]))
                elif self.format[i] == 'class':
                    classInColumn = i
            #make predictions based on training data
            theRealClass = data[classInColumn]
            classifiedAs = self.classify(vector)
            totals.setdefault(theRealClass, {})
            #tally up totals
            totals[theRealClass].setdefault(classifiedAs, 0)
            totals[theRealClass][classifiedAs] += 1
        return totals

    def getMedian(self, alist):
        """return median of alist"""
        if alist == []:
            return []
        blist = sorted(alist)
        length = len(alist)
        if length % 2 == 1:
            # length of list is odd so return middle element
            return blist[int(((length + 1) / 2) -  1)]
        else:
            # length of list is even so compute midpoint
            v1 = blist[int(length / 2)]
            v2 =blist[(int(length / 2) - 1)]
            return (v1 + v2) / 2.0


    def getAbsoluteStandardDeviation(self, alist, median):
        """given alist and median return absolute standard deviation"""
        sum = 0
        for item in alist:
            sum += abs(item - median)
        return sum / len(alist)


    def normalizeColumn(self, columnNumber):
        col = [v[1][columnNumber] for v in self.data]
        median = self.getMedian(col)
        asd = self.getAbsoluteStandardDeviation(col, median)
        self.medianAndDeviation.append((median, asd))
        for v in self.data:
            v[1][columnNumber] = (v[1][columnNumber] - median) / asd


    def normalizeVector(self, v):
        """We have stored the median and asd for each column.
```

```
        We now use them to normalize vector v"""
        vector = list(v)
        for i in range(len(vector)):
            (median, asd) = self.medianAndDeviation[i]
            vector[i] = (vector[i] - median) / asd
        return vector


    ###
    ### END NORMALIZATION
    ####################################################



    def manhattan(self, vector1, vector2):
        """Computes the Manhattan distance."""
        return sum(map(lambda v1, v2: abs(v1 - v2), vector1, vector2))

    def euclidean(self, vector1, vector2):
        return sum(map(lambda v1, v2: (v1 - v2) ** 2, vector1, vector2))
 ** (0.5)

    def kNN(self, itemVector):
        neighbors = sorted([(self.manhattan(itemVector, item[1]), item)
                            for item in self.data])[:self.k]
        votes = {}
        for neighbor in neighbors:
            votes.setdefault(neighbor[1][0], 0)
            votes[neighbor[1][0]] += 1

        sortedVotes = sorted(votes, votes.get, reverse=False)
        prediction = sortedVotes[0]

        return prediction

    def classify(self, itemVector):
        """Return class we think item Vector is in"""
        return(self.kNN(self.normalizeVector(itemVector)))

#INTERESTING! create new classifier for each validation iteration
def tenfoldCrossValidation(bucketPrefix, dataFormat,kN=1):
    results = {}
    for i in range(1,11):
        #create and train a new classifier for each combination of 9 buc
kets
        c = Classifier(bucketPrefix, i, dataFormat,k=kN)
        test = c.testBucket(bucketPrefix, i)
        for (key, value) in test.items():
            results.setdefault(key, {})
            for (categoryKey, categoryValue) in value.items():
                results[key].setdefault(categoryKey, 0)
                results[key][categoryKey] += categoryValue
    categories = list(results.keys())
    categories.sort()

    print( "\n Classified as: ")
    header = " "
```

```
    subheader = " +"
    for category in categories:
        header += "  " + category + " "
        subheader += "----+"
    print (header)
    print (subheader)
    total = 0.0
    correct = 0.0
    for category in categories:
        row = category + " |"
        for c2 in categories:
            if c2 in results[category]:
                count = results[category][c2]
            else:
                count = 0
            row += "%2i |" % count
            total += count
            if c2 == category:
                correct += count
        print(row)
    print(subheader)
    print("\n%5.3f percent correct!!!" %((correct * 100) / total))
    print("Total of %i instances." % total)

tenfoldCrossValidation("mpgData/mpgData", "class num num num num num com
ment",1) #1 nearest neighbor
tenfoldCrossValidation("mpgData/mpgData", "class num num num num num com
ment",2) #2 nearest neighbors

#HOLY CRAP IT WORKED THAT IS AWESOME!
#for some reason, the classifier I wrote is mysteriously 5% less accurat
e than the one he wrote? hm
```

```
 Classified as:
   10   15   20   25   30   35   40   45
 +----+----+----+----+----+----+----+----+
10 |  0 |13 |  0 |  0 |  0 |  0 |  0 |  0 |
15 |  0 |62 |23 |  1 |  0 |  0 |  0 |  0 |
20 |  0 |11 |62 |17 |  1 |  5 |  0 |  0 |
25 |  0 |  1 |13 |35 |17 |13 |  0 |  0 |
30 |  0 |  0 |  6 |21 |23 |  7 |  6 |  0 |
35 |  0 |  0 |  2 |12 |19 |  2 |  3 |  0 |
40 |  0 |  0 |  1 |  2 |  7 |  0 |  1 |  0 |
45 |  0 |  0 |  0 |  0 |  6 |  0 |  0 |  0 |
 +----+----+----+----+----+----+----+----+

47.194 percent correct!!!
Total of 392 instances.

 Classified as:
   10   15   20   25   30   35   40   45
 +----+----+----+----+----+----+----+----+
10 |  0 |13 |  0 |  0 |  0 |  0 |  0 |  0 |
15 |  0 |63 |22 |  1 |  0 |  0 |  0 |  0 |
20 |  0 |12 |60 |24 |  0 |  0 |  0 |  0 |
25 |  0 |  0 |15 |53 |11 |  0 |  0 |  0 |
30 |  0 |  0 |  6 |32 |25 |  0 |  0 |  0 |
35 |  0 |  0 |  2 |17 |19 |  0 |  0 |  0 |
40 |  0 |  0 |  1 |  3 |  7 |  0 |  0 |  0 |
45 |  0 |  0 |  0 |  0 |  6 |  0 |  0 |  0 |
 +----+----+----+----+----+----+----+----+

51.276 percent correct!!!
Total of 392 instances.
```

Kappa statistic

Compares performance of classifier to classifier that makes decisions based solely on chance.

- Take **row totals**, which represent amount of true datapoints within that class
- Take **column totals** which represent total predicted values for classifier
- Predict (Column Total / Total Values) * Amount for Class classifications in our confusion matrix for each class

$$k = \frac{P(c) - P(r)}{1 - P(r)}$$

Kappa statistic is given by the above formula, where P(c) is accuracy of the real classifier and P(r) is the accuracy of our (semi?) random classifier. How do we interperet the value given by this heuristic?

- **Less than 0:** less than random chance!!!! an octupus pointing at a piece of paper could have done better than your code did.
- **0.01-0.20:** slightly good
- **0.21-0.4:** fair performance
- **0.41-0.6:** moderate performance
- **0.61-0.8:** substantially good performance
- **0.81-1.00:** near perfect performance

Improvements to the Nearest Neighbor Algorithm

- **Rote classifier:** not useful in practice, but it only classifies EXACT MATCHES to datapoints it is trained on
- **Nearest neighbor** classifier can be seen as an extension of this... the ***IF IT WALKS LIKE A DUCK, AND IT QUACKS LIKE A DUCK, IT PROBABLY IS A DUCK*** approach
- Nearest neighbor approach has the same problem as it did with recommendations... **sensitive to outliers**. k-Nearest Neighbors mitigates this sensitivity to outliers.

When predicting a **discrete class**, the k-nearest neighbors **cast votes** for which class the item in question should be in, and randomly selects if there is a tie. In cases with **numeric values**, as we saw in recommender systems earlier, we can weight averages based on user similarity.

For given distances, we should 1) replace distance with **inverse distance** (smaller become larger) and 2) **divide each inverse distance with the sum of all inverse distances**.