Homework 1: k-Nearest Neighbor

Jef Harkay

October 8, 2011

1 Datasets Used

I used four datasets instead of the two datasets that were required, but I only did this because I was curious about getting more results. In the results section, I will only report on two datasets because there were three that had very similar outcomes.

Iris

Average Accuracy: ~94%

Attributes (in file's order)

1. sepal length (cm)
2. sepal width (cm)
3. petal length (cm)
4. petal width (cm)

Classifications

Iris Setosa

Iris Versicolor

Iris Virginica

Instances: 150

Ecoli

Average Accuracy: ~82%

Attributes (in file's order)

1. **mcg**: McGeoch's method for signal sequence recognition
2. **gvh**: Von Heijne's method for signal sequence recognition
3. **aac**: Score of discriminant analysis of the amino acid content of outer membrane and periplasmic proteins
4. **alm1**: Score of the ALOM membrane spanning region prediction program
5. **alm2**: Score of ALOM program after excluding putative cleavable signal regions from the sequence

Classifications

Cytoplasm (cp)

Inner Membrane without Signal Sequence (im)

Perisplasm (pp)

Inner Membrane, Uncleavable Signal Sequence (imU)

Outer Membrane (om)

Outer Membrane Lipoprotein (omL)

Inner Membrane Lipoprotein (imL)

Inner Membrane, Cleavable Signal Sequencer (imS)

Instances: 336

Other Info: I took out attributes 1, 4, and 5 in the original file because 1 is the actual name of each ecoli strain, 4 basically stays a constant number throughout, and 5 is definitely a constant number throughout. I also had to replace each space with commas, so my program could process the data.

Glass Identification

Average Accuracy: ~89%

Attributes (in file's order)

1. Refractive Index
2. Sodium weight %
3. Magnesium weight %
4. Aluminum weight %
5. Silicon weight %
6. Potassium weight %
7. Calcium weight %
8. Barium weight %
9. Iron weight %

Classifications

Building Windows Float Processed

Building Windows Non Float Processed

Vehicle Windows Float Processed

Containers

Tableware

Headlamps

Instances: 214

Other Info: I got rid of attribute 1 in the original file because it's just an Id number. I also changed each classification number at the end of the text file to the corresponding name. Building Windows Float Processed was 1, BWNFP was 2, etc.

Statlog (Vehicle Silhouettes)

Average Accuracy: ~50%

Attributes (in file's order)

1. Compactness
2. Circularity
3. Distance Circularity
4. Radius Ratio
5. Axis Aspect Ratio
6. Max Length Aspect Ratio
7. Scatter Ratio
8. Elongatedness

9. Axis Rectangularity
10. Max Length Rectangularity
11. Scaled Variance (Major Axis)
12. Scaled Variance (Minor Axis)
13. Scaled Radius of Gyration
14. Skewness About Major Axis
15. Skewness About Minor Axis
16. Kurtosis About Minor Axis
17. Kurtosis About Major Axis
18. Hollows Ratio

Classifications

Opel

Saab

Bus

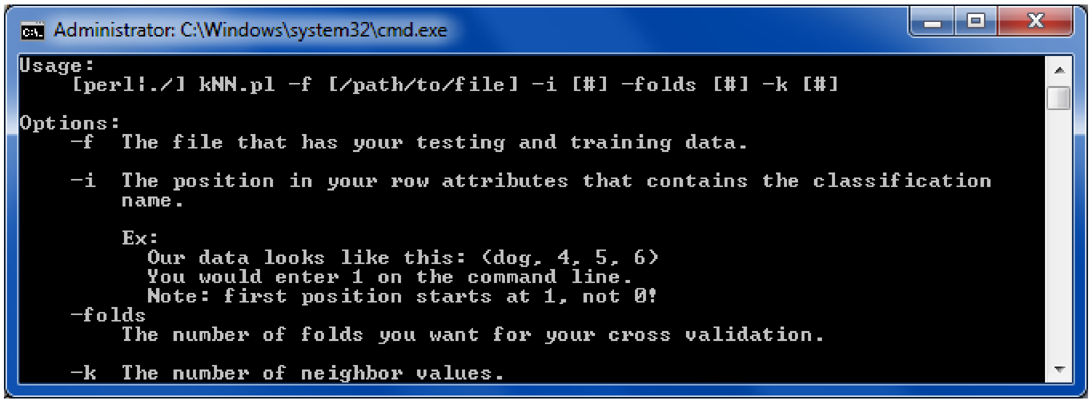
Van

Instances: 282

Other Info: I combined three of their given datasets online (xaa.dat, xab.dat, and xac.dat). I also had to add commas instead of spaces between each attribute.

2 How to Run kNN.pl

To see kNN.pl's usage statement, just run kNN.pl. It will warn you if you have an error in any of the parameters you supplied.



```
Usage:
    [perl|./] kNN.pl -f [/path/to/file] -i [#] -folds [#] -k [#]

Options:
    -f  The file that has your testing and training data.

    -i  The position in your row attributes that contains the classification
        name.

        Ex:
          Our data looks like this: (dog, 4, 5, 6)
          You would enter 1 on the command line.
          Note: first position starts at 1, not 0!
    -folds
        The number of folds you want for your cross validation.

    -k  The number of neighbor values.
```

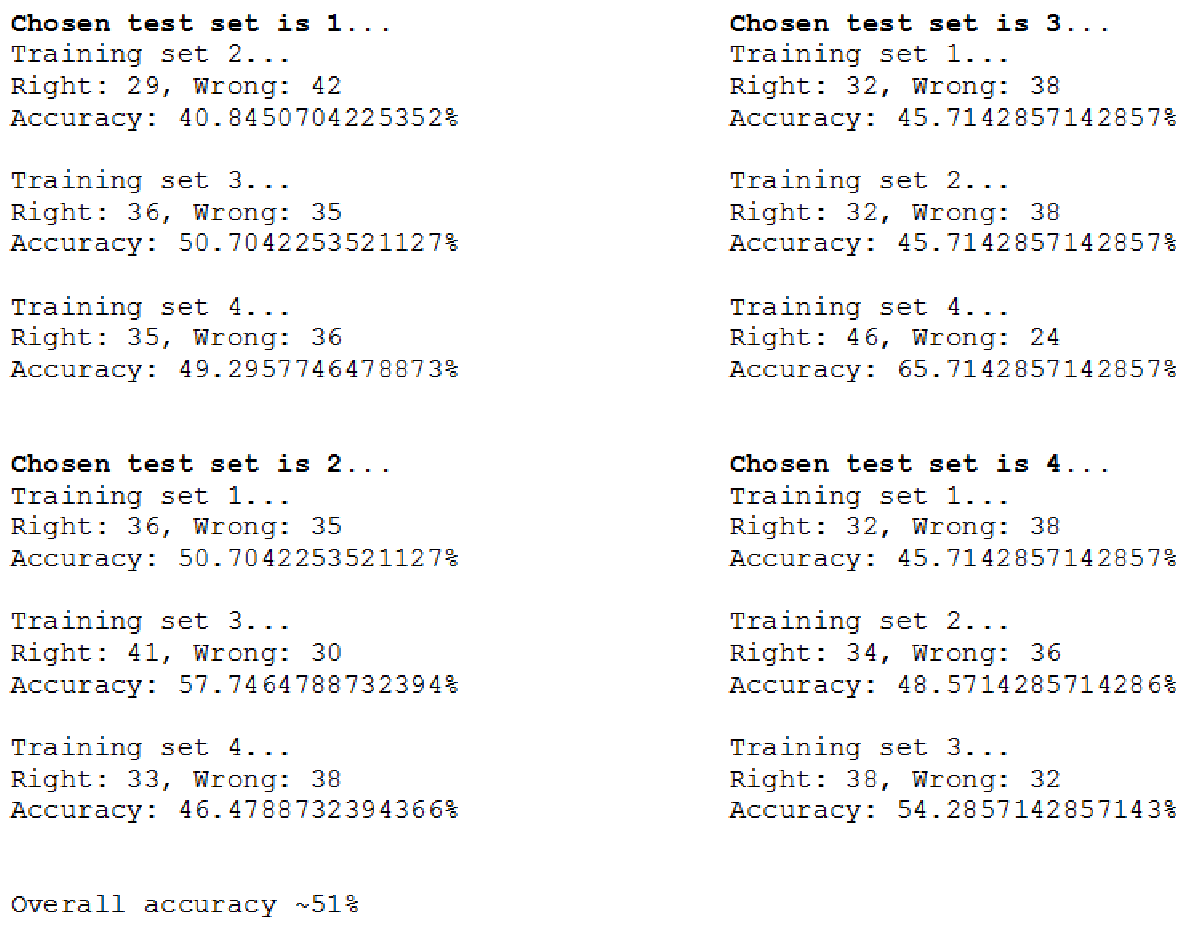
2.1 Sample Runs

```
Chosen test set is 1...
Training set 2...
Right: 37, Wrong: 1
Accuracy: 97.3684210526316%

Training set 3...
Right: 38, Wrong: 0
Accuracy: 100%

Training set 4...
Right: 36, Wrong: 2
Accuracy: 94.7368421052632%

Chosen test set is 2...
Training set 1...
Right: 31, Wrong: 7
Accuracy: 81.5789473684211%

Training set 3...
Right: 35, Wrong: 3
Accuracy: 92.1052631578947%

Training set 4...
Right: 38, Wrong: 0
Accuracy: 100%
```

```
Chosen test set is 3...
Training set 1...
Right: 31, Wrong: 6
Accuracy: 83.7837837837838%

Training set 2...
Right: 34, Wrong: 3
Accuracy: 91.8918918918919%

Training set 4...
Right: 36, Wrong: 1
Accuracy: 97.2972972972973%

Chosen test set is 4...
Training set 1...
Right: 33, Wrong: 4
Accuracy: 89.1891891891892%

Training set 2...
Right: 36, Wrong: 1
Accuracy: 97.2972972972973%

Training set 3...
Right: 36, Wrong: 1
Accuracy: 97.2972972972973%
```

```
Overall accuracy ~94%
```

Figure 1: 'perl kNN.pl -folds 4 -i 5 -f iris.txt -k 4'



```
Chosen test set is 1...
Training set 2...
Right: 29, Wrong: 42
Accuracy: 40.8450704225352%

Training set 3...
Right: 36, Wrong: 35
Accuracy: 50.7042253521127%

Training set 4...
Right: 35, Wrong: 36
Accuracy: 49.2957746478873%


Chosen test set is 2...
Training set 1...
Right: 36, Wrong: 35
Accuracy: 50.7042253521127%

Training set 3...
Right: 41, Wrong: 30
Accuracy: 57.7464788732394%

Training set 4...
Right: 33, Wrong: 38
Accuracy: 46.4788732394366%

Chosen test set is 3...
Training set 1...
Right: 32, Wrong: 38
Accuracy: 45.7142857142857%

Training set 2...
Right: 32, Wrong: 38
Accuracy: 45.7142857142857%

Training set 4...
Right: 46, Wrong: 24
Accuracy: 65.7142857142857%


Chosen test set is 4...
Training set 1...
Right: 32, Wrong: 38
Accuracy: 45.7142857142857%

Training set 2...
Right: 34, Wrong: 36
Accuracy: 48.5714285714286%

Training set 3...
Right: 38, Wrong: 32
Accuracy: 54.2857142857143%


Overall accuracy ~51%
```

Figure 2: ‘perl kNN.pl -folds 4 -i 19 -f vehicles.txt -k 4‘

2.2 Cross Validation

Cross validation is required to run kNN.pl. You can pick the fold number by using the ’-folds’ flag for kNN.pl. So let’s say you picked k=4:

1. kNN.pl creates a hash with set1, set2, set3, and set4.
2. The dataset gets split by the fold number.

   NOTE: If there’s an uneven split, then some sets will be one row bigger.
3. Each set gets assigned a random row of the dataset until all values are accounted for.

3 Iris Results

Table 1: k=1

Test Set	Train Set	Right	Wrong	Accuracy
1	2	35	3	93%
1	3	34	4	90%
1	4	36	2	95%
2	1	37	1	98%
2	3	35	3	93%
2	4	35	3	93%
3	1	35	2	95%
3	2	34	3	92%
3	4	34	3	92%
4	1	37	0	100%
4	2	37	0	100%
4	3	36	1	98%
	Final	425	25	95%

Table 2: k=4

Test Set	Train Set	Right	Wrong	Accuracy
1	2	38	0	100%
1	3	35	3	93%
1	4	36	2	95%
2	1	35	3	93%
2	3	36	2	95%
2	4	34	4	90%
3	1	36	1	98%
3	2	36	1	98%
3	4	32	5	87%
4	1	34	3	92%
4	2	34	3	92%
4	3	32	5	87%
	Final	418	32	93%

Table 3: k=7

Test Set	Train Set	Right	Wrong	Accuracy
1	2	36	2	95%
1	3	35	3	93%
1	4	34	4	90%
2	1	37	1	98%
2	3	37	1	98%
2	4	38	0	100%
3	1	33	4	90%
3	2	33	4	90%
3	4	32	5	87%
4	1	36	1	98%
4	2	33	4	90%
4	3	36	1	98%
	Final	420	30	94%

4 Statlog (Vehicle Silhouettes) Results

Table 4: k=1

Test Set	Train Set	Right	Wrong	Accuracy
1	2	39	32	55%
1	3	42	29	60%
1	4	37	34	53%
2	1	38	33	54%
2	3	38	33	54%
2	4	37	34	53%
3	1	39	31	56%
3	2	38	32	55%
3	4	36	34	52%
4	1	40	30	58%
4	2	42	28	60%
4	3	42	28	60%
	Final	468	378	56%

Table 5: k=4

Test Set	Train Set	Right	Wrong	Accuracy
1	2	38	33	54%
1	3	34	37	48%
1	4	30	41	43%
2	1	42	29	60%
2	3	42	29	60%
2	4	37	34	53%
3	1	29	41	42%
3	2	36	34	52%
3	4	37	33	53%
4	1	24	46	35%
4	2	29	41	42%
4	3	35	35	50%
	Final	413	433	49%

Table 6: k=7

Test Set	Train Set	Right	Wrong	Accuracy
1	2	32	39	46%
1	3	38	33	54%
1	4	35	36	50%
2	1	35	36	50%
2	3	35	36	50%
2	4	38	33	54%
3	1	32	38	46%
3	2	37	33	53%
3	4	37	33	53%
4	1	33	37	48%
4	2	37	33	53%
4	3	40	30	58%
	Final	429	417	51%

5 Thoughts on k-Nearest Neighbor

k-Nearest Neighbor is a very simple algorithm and because it's simple, it's useful if you have a dataset with a small amount of attributes. It makes sense too because less attributes means less it has to take into consideration–so it likes simple data!

Advantages:

- Simple to implement for numerical data.
- Performs well for small datasets.

Disadvantages:

- Computationally expensive (especially for large datasets/attributes).
- Hard to implement for data that has meaning/strings.
- Appears to have bad classifications for datasets with a large amount of attributes.

A major setback to this algorithm is that the decision is based on the highest "voted" answer. It can easily mistake a value that's in the wrong neighborhood–meaning the classification with more values usually wins.

Another thing I noticed is that the k value really didn't change much. I'm not sure if it's the data I used or if Euclidean distance is just that good, but there seemed to be very little difference between k values. As you can see in Figures 1-6, the accuracies all seem relatively the same within the dataset.

Lastly, the algorithm appears to perform poorly on a dataset with a large number of attributes. The vehicles dataset seems to have legitimate data, so computing the Euclidean distance shouldn't make that much of a difference, so I've come to the conclusion that k-Nearest Neighbor isn't good for high dimension datasets.

6 Appendix

```perl
use warnings;
use strict;
use POSIX;
use Pod::Usage;
use Getopt::Long;
use Math::Complex;
use List::Util 'shuffle';

my $file;
my $folds;
my $index;
my $k;
my $tot_right = 0;
my $tot_rows = 0;
GetOptions( "f=s" => \$file ,
            "folds=i" => \$folds ,
            "i=i" => \$index ,
            "k=i" => \$k );
pod2usage(1) unless $file and $folds and $index and $k;

my %sets = ();
my @data = ();

open my $fh, "<", $file or die "Can't open that!  $!\n";
setup($folds);

#<$fh>;  # Skips first 2 lines of the file... the attribute names,
#<$fh>;  # followed by a blank line.
while (my $line = <$fh>) {
  $line =~ s/\p{IsC}//g;
  next if !$line;
  #chomp $line;
  my @items = split(/,\s*/, $line);
  push(@data, [@items]);
}

@data = shuffle(@data); # randomizing data
split_data(\@data, $folds);
my $chosen = 1;
my $overall_acc = 0;
for (my $chosen = 1; $chosen <= $folds; $chosen++) {
  #print "Chosen test set is $chosen...\n";
  $overall_acc += euclid($chosen, $index, $folds);
  print "\n";
```

```
}
print "\\hline\n";
print " & Final & $tot_right & ", $tot_rows - $tot_right, " & ", ceil($overall_acc
#print "Total right: $tot_right, total wrong: ", $tot_rows - $tot_right, "\n";
#print "Overall accuracy ~", ceil($overall_acc / $folds), "%\n";

# Splits the data in the number of folds... if the fold divides evenly,
# then all arrays will be same number, otherwise, the beginning sets will have more
sub split_data {
  my ($data, $fold) = @_;
  my $set_number = 1;
  for (my $i = 0; $i < scalar(@{$data}); $i++) {
    push(@{$sets{"set$set_number"}}, @{$data}[$i]);
    $set_number = 1 if $set_number++ == $fold;
  }
}

sub setup {
  my $fold = shift;
  for (my $i = 1; $i <= $fold; $i++) {
    $sets{"set$i"} = ();
  }
}

sub euclid {
  my ($chosen, $indexes, $folds) = @_;
  my $num_attr = scalar(@{$sets{"set$chosen"}[0]});
  my $test_rows = scalar(@{$sets{"set$chosen"}});
  my $train_attr;
  my $count = 0;
  my $accuracy = 0;
  for (my $train = 1; $train <= $folds; $train++) {
    next if $train == $chosen;
    #print "Training set $train...\n";
    my $train_rows = scalar(@{$sets{"set$train"}});
    my $right = 0;
    for (my $test_row = 0; $test_row < $test_rows; $test_row++) {
      my @out;
      my $test_attr = $sets{"set$chosen"}[$test_row][$index-1];
      for (my $train_row = 0; $train_row < $train_rows; $train_row++) {
        my $sum = 0;
        for (my $col = 0; $col < $num_attr; $col++) {
          if ($index == $col+1) {
            $train_attr = $sets{"set$train"}[$train_row][$col];
            next;
          }
          $sum += ($sets{"set$chosen"}[$test_row][$col] - $sets{"set$train"}[$train
        }
        push (@out, [sqrt($sum), $train_attr]);
      }
      @out = sort {$a->[0] <=> $b->[0]} @out;
      @out = splice(@out, 0, $k);
      my $decision = decide(@out);
      $right++ if $decision eq $test_attr;
    }
    print "$chosen & $train & $right & ", $test_rows - $right, " & ", ceil(($right
    #print "Right: $right, Wrong: ", $test_rows - $right, "\n";
    $count++;
    $accuracy += ($right / $test_rows) * 100;
    $tot_rows += $test_rows;
    $tot_right += $right;
    #print "Accuracy: ", ($right / $test_rows) * 100, "%\n";
    #print "\n";
  }
  return $accuracy / $count;
}

sub decide {
  my @top_k = @_;
  my %top_k;
  for (my $i = 0; $i < $k; $i++) {
    if ($top_k{$top_k[$i][1]}) {
      $top_k{$top_k[$i][1]}++;
    }
    else {
      $top_k{$top_k[$i][1]} = 1;
    }
  }
  my @top = sort {$top_k{$a} < $top_k{$b}} keys %top_k;
  return $top[0];
}

__END__

=head1 SYNOPSIS

[perl|./] kNN.pl -f [/path/to/file] -i [#] -folds [#] -k [#]

=head1 OPTIONS
```

```
=over 4

=item B<-f>

The file that has your testing and training data.

=item B<-i>

The position in your row attributes that contains the classification name.

=begin text

        Ex:
          Our data looks like this: (dog, 4, 5, 6)
          You would enter 1 on the command line.
          Note: first position starts at 1, not 0!

=end text

=item B<-folds>

The number of folds you want for your cross validation.

=item B<-k>

The number of neighbor values.

=back

=head1 DESCRIPTION

This program is an implementation of the k-Nearest Neighbor algorithm.

=cut
```