K-Nearest Neighbors

Geoffrey Ulman
Homework 11
CSI873

December 2011

1 Results

The minimum testing error rate for k-nearest neighbors with uniform weights was 0.241 and occurred with $k = 4$. The minimum testing error rate for k-nearest neighbors with weights decaying by distance according to Equation 1 was 0.217 and occurred with $k = 7$.

$$w_i = \frac{1}{d\left(x_q, x_i\right)^2 + \epsilon}, \epsilon = 1 \tag{1}$$

The trend for larger k for both the uniform and decaying weight runs is shown on Figure 1. It indicates that the error rate of 0.217 obtained by the decaying weight knn classifier with $k = 7$ is actually the best performing for all k for this handwriting classification problem. Even including all training samples weighted by distance does not improve the error rate. In fact, with an error rate of 0.410 it is a significantly worse performer than the runs with small k values.

Table 1: Uniform Weight Error

K	Error	95% Confidence Interval	
		Lower Bound	Upper Bound
1	0.254	0.212	0.296
2	0.266	0.223	0.309
3	0.259	0.216	0.301
4	0.241	0.200	0.283
5	0.273	0.230	0.316
6	0.280	0.237	0.324
7	0.268	0.225	0.311

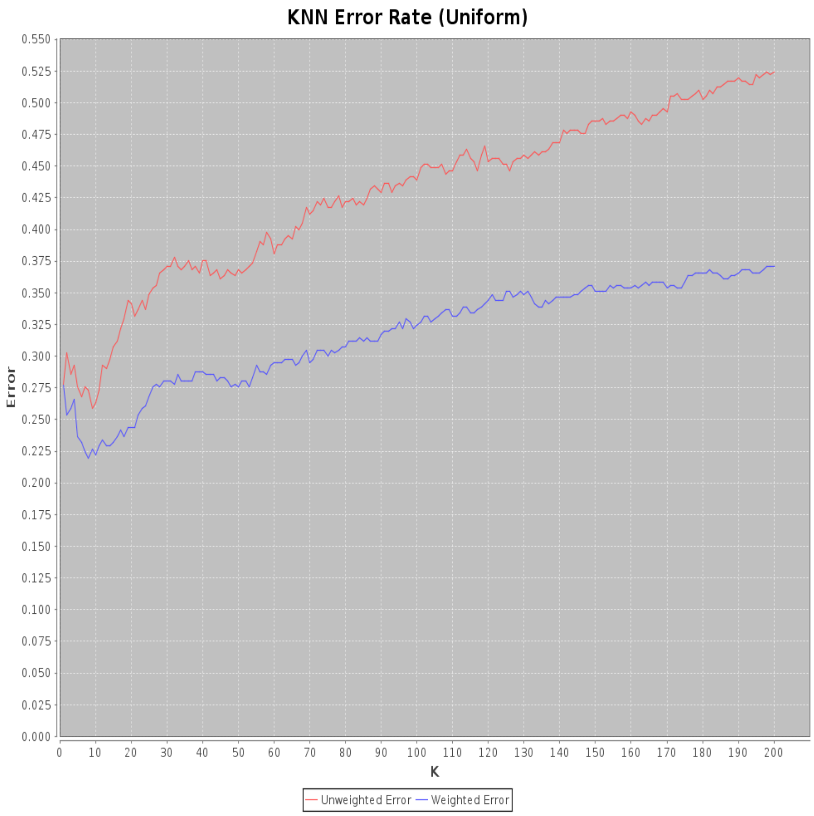

Figure 1: Missclassification Error by K

Table 2: Decaying Weight Error

K	Error	95% Confidence Interval	
		Lower Bound	Upper Bound
1	0.254	0.212	0.296
2	0.244	0.202	0.285
3	0.227	0.186	0.267
4	0.234	0.193	0.275
5	0.237	0.195	0.278
6	0.229	0.189	0.270
7	0.217	0.177	0.257
410	0.410	0.362	0.457

References

[1] Tom M. Mitchell, *Machine Learning*, WCB McGraw-Hill, Boston, 1997.

```java
package edu.gmu.classifier.knearest;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;

import javax.swing.JFrame;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.xy.DefaultXYDataset;

import com.google.common.collect.Ordering;
import com.google.common.collect.TreeMultimap;

import edu.gmu.classifier.io.DataLoader;
import edu.gmu.classifier.io.TrainingExample;

public class KNearest
{
	public static void main( String[] args ) throws IOException
	{
		// load training and testing data
		List<TrainingExample> dataListTrain = DataLoader.loadDirectoryTrain( "/home/ulman/CSI873/midterm/data" );
		List<TrainingExample> dataListTest = DataLoader.loadDirectoryTest( "/home/ulman/CSI873/midterm/data" );

		System.out.println( "Testing Data" );
		runKNNClassifier( dataListTrain, dataListTest );

		System.out.println( "Training Data" );
		runKNNClassifier( dataListTrain, dataListTrain );
	}

	public static interface WeightCalculator
	{
		public double getWeight( double distance );
	}

	public static void runKNNClassifier( List<TrainingExample> dataListTrain, List<TrainingExample> dataListTest )
	{
		// calculate the distance from each testing example to each training example and store
		// them in an ordered set, making it efficient to retrieve the k closest
		Map<TrainingExample,TreeMultimap<Integer,TrainingExample>> outerMap = new
HashMap<TrainingExample,TreeMultimap<Integer,TrainingExample>>( );
		for ( TrainingExample test : dataListTest )
		{
			TreeMultimap<Integer,TrainingExample> map = getDistancesFrom( test, dataListTrain );
			outerMap.put( test, map );
		}

		int maxk = 7;

		// create jfreechart dataset for plotting purposes
		DefaultXYDataset dataset = new DefaultXYDataset( );
		double[][] seriesData = new double[2][maxk];
		double[][] seriesData2 = new double[2][maxk];

		// create a weight calculator which returns uniform weights regardless of distance
		WeightCalculator uniformWeight = new WeightCalculator( )
		{
			@Override
			public double getWeight( double distance )
			{
				return 1.0;
			}
		};

		System.out.println( "Uniform Weight" );

		for ( int k = 1 ; k <= maxk ; k++ )
		{
			double error = calculateErrorRate( outerMap, dataListTest, uniformWeight, k );

			seriesData[0][k-1] = k;
			seriesData[1][k-1] = error;
		}

		// create a weight calculator which returns weights that decay with distance
		WeightCalculator decayWeight = new WeightCalculator( )
		{
			@Override
			public double getWeight( double distance )
			{
				return 1.0 / ( distance * distance + 1.0 );
```

```
			}
		};

		System.out.println( "Decaying Weight" );

		for ( int k = 1 ; k <= maxk ; k++ )
		{
			double error = calculateErrorRate( outerMap, dataListTest, decayWeight, k );

			seriesData2[0][k-1] = k;
			seriesData2[1][k-1] = error;
		}

		calculateErrorRate( outerMap, dataListTest, decayWeight, dataListTest.size( ) );

		dataset.addSeries( "Unweighted Error", seriesData );
		dataset.addSeries( "Weighted Error", seriesData2 );
		JFreeChart chart2 = ChartFactory.createXYLineChart( String.format( "KNN Error Rate (Uniform)" ), "K", "Error", dataset,
PlotOrientation.VERTICAL, true, false, false );
		ChartPanel chartPanel2 = new ChartPanel( chart2 );
		JFrame frame2 = new JFrame( );
		frame2.setSize( 1000, 1000 );
		frame2.add( chartPanel2 );
		frame2.setVisible( true );
	}

	// calculate and print the error rate for the given k
	public static double calculateErrorRate( Map<TrainingExample,TreeMultimap<Integer,TrainingExample>> outerMap,
							List<TrainingExample> dataListTest, WeightCalculator weightCalc, int k )
	{
		int correct = 0;
		for ( TrainingExample test : dataListTest )
		{
			int predicted_digit = pickDigit( pickLowestK( outerMap.get( test ), k ), weightCalc );
			if ( predicted_digit == test.getDigit( ) ) correct++;
		}

		double errorRate = 1.0 - ( (double) correct / (double) dataListTest.size( ) );
		double errorInterval = 1.96 * Math.sqrt( errorRate * ( 1 - errorRate ) / dataListTest.size( ) );

		System.out.printf( "K: %d Error Rate: %.3f Train Interval: (%.3f, %.3f)%n", k, errorRate, errorRate - errorInterval,
errorRate + errorInterval );

		return errorRate;
	}

	// given a weighting scheme and a set of nearby training examples, use a simple weighted
	// voting scheme to classify the sample in question
	public static int pickDigit( Collection<Entry<Integer,TrainingExample>> list, WeightCalculator weightCalc )
	{
		double[] digitCounts = new double[10];

		for ( Entry<Integer,TrainingExample> entry : list )
		{
			TrainingExample example = entry.getValue( );
			Integer distance = entry.getKey( );

			digitCounts[example.getDigit( )] += weightCalc.getWeight( distance );
		}

		return getLargestIndex( digitCounts );
	}

	//returns the index of the largest entry in the array
	public static int getLargestIndex( double[] array )
	{
		double max = 0;
		int index = 0;

		for ( int i = 0 ; i < array.length ; i++ )
		{
			double data = array[i];

			if ( data > max )
			{
				max = data;
				index = i;
			}
		}

		return index;
	}

	// given a sorted map containing the distances from a test example to all training examples, choose the k lowest
	public static Collection<Entry<Integer,TrainingExample>> pickLowestK( TreeMultimap<Integer,TrainingExample> map, int k )
	{
		Collection<Entry<Integer,TrainingExample>> list = new ArrayList<Entry<Integer,TrainingExample>>( k );
		int count = 0;

		for( Entry<Integer,TrainingExample> example : map.entries( ) )
```

```
		{
			if ( count++ == k ) break;

			list.add( example );
		}

		return list;
	}

	// calculates the distance between each training example and the test example, returns the values in a sorted map
	public static TreeMultimap<Integer,TrainingExample> getDistancesFrom( TrainingExample example, List<TrainingExample>
dataListTrain )
	{
		TreeMultimap<Integer,TrainingExample> map = TreeMultimap.create( Ordering.natural( ), Ordering.arbitrary( ) );

		for ( TrainingExample data : dataListTrain )
		{
			map.put( getDistance( example, data ), data );
		}

		return map;
	}

	// the distance between two training examples is defined as the number of pixels which differ
	public static int getDistance( TrainingExample e1, TrainingExample e2 )
	{
		double[] d1 = e1.getInputs( );
		double[] d2 = e2.getInputs( );

		int count = 0;
		for ( int i = 0 ; i < d1.length ; i++ )
		{
			if ( d1[i] != d2[i] )
				count++;
		}

		return count;
	}
}
```