Classify Handwritten Digtits Using KNN and Random Forest

Tianyi Gu
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
tg1034@wildcats.unh.edu

Abstract

Natural handwriting is often a mixture of different "styles", some even hard to recognize by human. A reliable recognizer for such handwriting would greatly help. In this paper, we compare the accuracy and reliability of several different classifiers in recognizing handwritten digit. Classifiers such as K-Nearest Neighbors, Decision Tree, and Random Forest are applied to the problem and results are compared. Our results show that using K-Nearest Neighbors is the most accurate of the recognizer algorithms.

Introduction

Classification, a supervised learning approach, is probably the most widely used form of machine learning, and has been used to solve many interesting and often difficult real-world problems. In this paper, we study image classification. Now consider the harder problem of classifying images directly, where a human has not pre-processed the data. We might want to classify the image as a whole, e.g., is it an indoors or outdoors scene? is it a horizontal or vertical photo? does it contain a dog or not? This is called image classification. In the special case that the images consist of isolated handwritten letters and digits, for example, in a postal or ZIP code on a letter, we can use classification to perform handwriting recognition(Murphy, 2012).

Natural handwriting is often a mixture of different "styles", thin, fat, italic, some even hard recognized by human. A reliable recognizer for such handwriting would greatly improve interaction with pen-based devices, but its implementation presents technical challenges(LeCun *et al.*, 1995). Great algorithms have been developed in the area of handwritten digit recognition. In this paper we contrast the relative merits of each of the algorithms.

Approaches

The following are three approaches that can be used in determining the handwritten digit in an image. All of these approaches require supervised learning and a number of training examples.

K-Nearest Neighbors

The K nearest neighbor (KNN) classifier(Murphy, 2012) is a non-parametric classifier. This simply looks at the K points in the training set that are nearest to the test input x, counts how many members of each class are in this set, and returns that empirical fraction as the estimate. More formally,

$$P(y = c|x, D, K) = \frac{1}{K} \sum_{i \in N_k(x,D)} \mathbb{1}(y_i = c) \qquad (1)$$

where $N_k(x, D)$ are the (indices of the) K nearest points to x in D and $\mathbb{1}(e)$ is the indicator function defined as follow:

$$\mathbb{1}(e) = \begin{cases} 1 \text{ if } e \text{ is true} \\ 0 \text{ if } e \text{ is false} \end{cases} \qquad (2)$$

The most common distance metric to use is Euclidean distance (which limits the applicability of the technique to data which is real-valued), although other metrics can be used.

Decision Tree

Decision tree induction is one of the simplest and yet most successful forms of machine learning(Russell and Norvig, 2010). A decision tree represents a function that takes as input a vector of attribute values and returns a "decision"a single output value. The input and output values can be discrete or continuous. A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the input attributes, A_i and the branches from the node are labeled with the possible values of the attribute, A_i–Each leaf node in the tree specifies a value to be returned by the function. The decision tree representation is natural for humans.

Finding the optimal partitioning of the data is NP–complete (Hyafil and Rivest 1976), so it is common to use the greedy procedure. The greedy search used in decision tree learning is designed to approximately minimize the depth of the final tree. The idea is to pick the attribute that goes as far as possible toward providing an exact classification of the examples. All we need, then, is a formal measure of "fairly good" and "really useless" and we can implement the importance function. We will use the notion of information gain, which is defined in terms of entropy, the fundamental quantity in information theory (Shannon and Weaver, 1949).

Entropy is a measure of the uncertainty of a random variable; acquisition of information corresponds to a reduction in entropy.A random variable with only one value–a coin that always comes up headshas no uncertainty and thus its entropy is defined as zero; thus, we gain no information by observing its value. A flip of a fair coin is equally likely to come up heads or tails, 0 or 1, and we will soon show that this counts as "I bit" of entropy. The roll of a fair four-sided die has 2 bits of entropy, because it takes two bits to describe one of four equally probable choices. Now consider an unfair coin that comes up heads 99% of the time. Intuitively, this coin has less uncertainty than the fair coinif we guess heads we'll be wrong only 1% of the time–so we would like it to have an entropy measure that is close to zero, but positive.

An attribute A with d distinct values divides the training set E into subsets E_k,Each subset F_k has p_k positive examples and n_k negative examples, so if we go along that branch, we will need an additional $H(p_k/(p_k + n_k))$ bits of information to answer the question. A randomly chosen example from the training set has the kth value for the attribute with probability $((p_k + n_k)/(p + n))$,so the expected entropy remaining after testing attribute A is

$$Remainder(A) = \sum_{k=1}^{K} \frac{p_k + n_k}{p + n} H(\frac{p_k}{p_k + n_k}) \qquad (3)$$

The information gain from the attribute test on A is the expected reduction in entropy:

$$Gain(A) = H(\frac{p}{p + n}) - Remainder(A) \qquad (4)$$

In fact Gain(A) is just what we need to implement the importance function.

To prevent overfitting, we can stop growing the tree if the decrease in the error is not sufficient to justify the extra complexity of adding an extra subtree. However, this tends to be too myopic. For decision trees, a technique called decision tree pruning combats overfitting. Pruning works by eliminating nodes that are not clearly relevant. We start with a full tree and look at a test node that has only leaf nodes as descendants, If the test appears to be irrelevant-detecting only noise in the data, then we eliminate the test, replacing it with a leaf node. We repeat this process, considering each test with only leaf descendants, until each one has either been pruned or accepted as is.

To detect that a node is testing an irrelevant attribute, we can use a statistical significance test. Such a test begins by assuming that there is no underlying pattern (the so-called null hypothesis). Then the actual data are analyzed to calculate the extent to which they deviate from a perfect absence of pattern. If the degree of deviation is statistically unlikely (usually taken to mean a 5% probability or less), then that is considered to be good evidence fur the presence of a significant pattern in the data. The probabilities are calculated from standard distributions of the amount of deviation one would expect to see in random sampling.

Random Forest

One way to reduce the variance of an estimate is to average together many estimates(Murphy, 2012). For example, we can train M different trees on different subsets of the data, chosen randomly with replacement, and then compute the ensemble

$$f(x) = \sum_{m=1}^{M} \frac{1}{M} f_m(x) \qquad (5)$$

where f_m is the m'th tree. This technique is called bagging (Breiman 1996), which stands for bootstrap aggregating.

Unfortunately, simply re-running the same learning algorithm on different subsets of the data can result in highly correlated predictors, which limits the amount of variance reduction that is possible. The technique known as random forests (Breiman 2001a) tries to decorrelate the base learners by learning trees based on a randomly chosen subset of input variables, as well as a randomly chosen subset of data cases. Such models often have very good predictive accuracy (Caruana and Niculescu-Mizil 2006), and have been widely used in many applications (e.g., for body pose recognition using Microsofts popular kinect sensor (Shotton et al. 2011)).

Bagging is a frequentist concept. It is also possible to adopt a Bayesian approach to learning trees. In particular, (Chipman et al. 1998; Denison et al. 1998; Wu et al. 2007) perform approximate inference over the space of trees (structure and parameters) using MCMC. This reduces the variance of the predictions. We can also perform Bayesian inference over the space of ensembles of trees, which tends to work much better. This is known as Bayesian adaptive regression trees or BART (Chipman et al. 2010). Note that the cost of these sampling-based Bayesian methods is comparable to the sampling-based random forest method. That is, both approaches are farily slow to train, but produce high quality classifiers.

Digit Recognizer

In this paper, to contrast the relative merits of each of the algorithms, we implement these three approaches and train the classifiers to recognize handwritten digits.

Data Collection

A standard dataset used in this area is known as MNIST, which stands for "Modified National Institute of Standards". (The term "modified" is used because the images have been preprocessed to ensure the digits are mostly in the center of the image.)

The data files train.csv and test.csv contain gray-scale images of hand-drawn digits, from zero through nine.

Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive.

The training data set, (train.csv), has 785 columns. The first column, called "label", is the digit that was drawn by

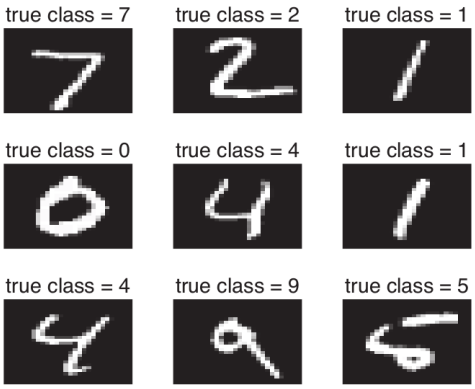

Figure 1: First 9 test MNIST gray-scale images

the user. The rest of the columns contain the pixel-values of the associated image.

Each pixel column in the training set has a name like pixelx, where x is an integer between 0 and 783, inclusive. To locate this pixel on the image, suppose that we have decomposed x as $x = i*28+j$, where i and j are integers between 0 and 27, inclusive. Then pixelx is located on row i and column j of a 28×28 matrix, (indexing by zero).

The test data set, (test.csv), is the same as the training set, except that it does not contain the "label" column. See Figure 1 for some example images.

K-Nearest Neighbors

To implement K-Nearest Neighbors for digit recognizer, for each training image, we calculate the distance from the test image

$$D = \sqrt{\sum_{pixel} (c_i - c_i')^2} \tag{6}$$

Then, we sort all training images to find k "nearest" images. So that we can calculate the fraction of each class are in this set using equation 1. At the end, we just need to return the class with maximum p value. Algorithm 1 is the pseudo-code for K-Nearest Neighbors.

Algorithm 1: KNN($TrainSet, TestSet, K$):

```
foreach image y ∈ TestSet do
    foreach image x ∈ TrainSet do
        x.d ← dis(x, y);
    end
    KNearestSet ← K nearest images in TrainSet;
    foreach image x ∈ KNearestSet do
        if x.l = c then c.count++;
    end
    foreach class c from 0 to 9 do
        if y.c.cout < c then y.c = c;
    end
end
return TestSet
```

Image	Attributes					class
	pixel1	pixel2	...	pixel783	pixel784	
X_1	76	188	...	1	199	0
X_2	234	18	...	1	45	8

Table 1: Attributes of images for decision tree

Image	Attributes					class
	pixel1	pixel2	...	pixel783	pixel784	
X_1	10-100	100-200	...	< 10	100-200	0
X_2	> 200	100-200	...	< 10	10-100	8

Table 2: Simplified attributes of images for decision tree

Decision Tree

We want a tree that is consistent with the examples and is as small as possible. Unfortunately, no matter how we measure size, it is an intractable problem to find the smallest consistent tree: there is no way to efficiently search through the 2^{2^n} trees. With some simple heuristics, however, we can find a good approximate solution: a small (but not smallest) consistent tree. Algorithm 2 adopts a greedy divide-and-conquer strategy: always test the most important attribute first. This test divides the problem up into smaller subproblems that can then be solved recursively. By "most important attribute" we mean the one that makes the most difference to the classification of an example. That way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be shallow.

Intuitively, we want to use catalogues rather than 256 integer numbers to branch each attribute, because that will make the tree too big so that it will be more probably overfit. For example, we can simplify table 1 to table 2 by using the following catalogues.

$$\begin{aligned} &catalogue1 : 0 - 10 \\ &catalogue2 : 10 - 100 \\ &catalogue3 : 100 - 200 \\ &catalogue4 : 200 - 255 \end{aligned}$$

In line 7 of Algorithm 2, we choose the attribute that minimizes the remaining information needed

$$\begin{aligned} Reminder(A) &= \sum_{i \in cata} \frac{e_i}{e} H(\frac{c0}{e_i}, \frac{c1}{e_i}, ..., \frac{c9}{e_i}) \\ &= \sum_{i \in cata} \frac{e_i}{e} (-\sum_{c} \frac{c}{e_i} \log \frac{c}{e_i}) \end{aligned} \tag{7}$$

Then we branch on that pixel. We keep doing this until no more examples or pixels and get the tree. After this, we can use the tree to determine the class for test image.

Random Forest

We implement Random Forest based on Decision Tree approach. The idea is we train M different trees on random subsets of the data and attributes and let every tree take a vote on classification for each test image by using equation 5

Algorithm 2: DTL($examples, attribs, default$) **return** a decision tree:

if *example is empty* **then return** default;
else if *all example have the same classification* **then**
    return the classification
else if *attributes is empty* **then**
    return MODE(examples)
else
    $best$ ←CHOOSE-ATTRIBUTE(attributes,examples);
    $tree$ ←a new decision tree with root test best;
    foreach *value* v_i *of best* **do**
        $examples_i$ ←{a new decision tree with root test best};
        $subtree$ ←DTL($examples_i$,attrebutes–best,MODE(examples));
        add a branch to tree with label v_i and subtree $subtree$;
    end
    return $tree$
end

Analysis

First, We split the training data set into two parts. One part for training and the other for testing. Second, we learn three classifiers respectively and do several analysis on each approach.

K-Nearest Neighbors

In the first experiment on K-Nearest Neighbors, we pick $k = 3$ and it turned out to perform very well, with only 0.03 misclassification rate. We still need to do model selection, means choosing K. As show in Figure 2 we do cross validation on 5 folds. The best $k = 3$,with misclassification rate=0.025.

Here are some of misclassified images as show in Figure 3. l:8 p:1 means the label of image is 8 but our prediction is 1. We can say most of them are hard to classify. However, we can also find some characteristic misclassified images like those "thin" 8 in Figure 4. KNN always predict them as 1. This might one of the weak point of KNN.

Decision Tree

The first experiment on Decision Tree, we use 5 catalogues to share the 0-255 value of each feature. The classifier work with missclassification rate of 0.29, this is so bad.

So then, we try to increase catalogue number. As show in Figure 5, as the catalogue number increase, the Decision Tree classifier even performs worse. The result tell us 3 catalogues are enough. However the misclassificaion rate still bad, which is about 0.2.

Then, we try to do pruning. We can measure the deviation by comparing the actual numbers of each catalogue

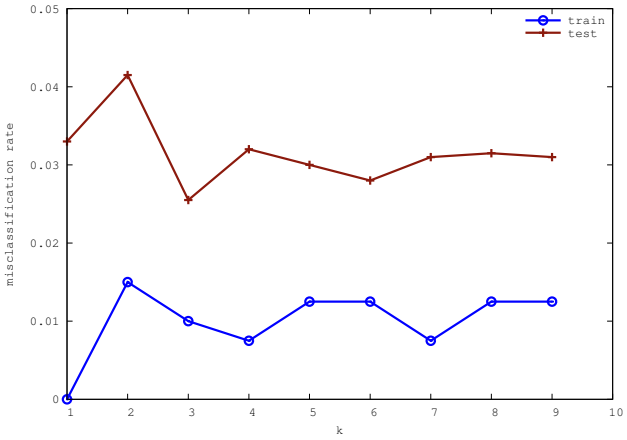

Figure 2: Misclassification rate vs K in a K-nearest neighbor classifier. On the left, where K is small, the model is complex and hence we overfit. On the right, where K is large, the model is simple and we underfit. Blue line: training set (size 38000). Red line: test set (size 4000).

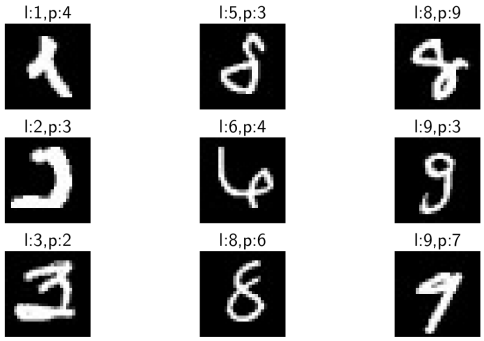

Figure 3: Some of misclassified images by KNN.

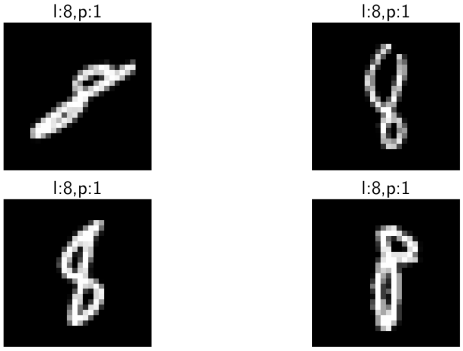

Figure 4: Characteristic misclassified images by KNN.

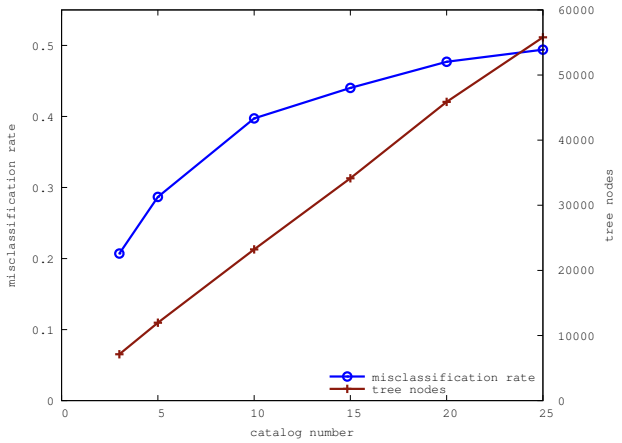

Figure 5: Misclassification rate vs catalogue number in a Decision Tree classifier. Blue line: misclassification rate. Red line: tree node number

examples in each subset, p_c with the expected numbers $\hat{p}_c$

$$\hat{p}_c = e_c \times \frac{e_i}{e} \tag{8}$$

A convenient measure of the total deviation is given by

$$D = \sum_{i=0}^{9} \sum_{c \in cata} \frac{(p_- \hat{p}_c)^2}{\hat{p}_c} \tag{9}$$

So if D is less than some ϵ, we prune this node. Algorithm 3 is the pseudocode of this approach. When the whole tree is built, we do pruning bottom up.

As shown in Figure 6, the best $\epsilon = 8$. But one decision tree still performs very bad.

Algorithm 3: DTLwithPruning($examples, attribs, default$) **return** a decision tree:

```
if example is empty then return default;
else if all example have the same classification then
    return the classification
else if attributes is empty then
    return MODE(examples)
else
    best ←CHOOSE-
    ATTRIBUTE(attributes,examples);
    tree ←a new decision tree with root test best;
    foreach value v_i of best do
        examples_i ←{a new decision tree with root
        test best};
        subtree ←DTLwithPruning(examples_i,attrebutes–
        best,MODE(examples));
        add a branch to tree with label v_i and subtree
        subtree;
    end
    best.D← get total deviation(best);
    return tree
end
```

Random Forest

The first experiment on Random Forest, we use 20 trees, 4000 random samples, 27 random features, and 3 catalogues. We end up with misclassification rate of 0.1435, which is better than single decision tree.

To improve the performance, we can try different parameters: tree number, sample number, features number, and catalogue number. As shown in Figure 7 8 9 and 10, the best parameters are 400 trees, 8000 random samples, 80 features, and 3 catalogues. The best misclassification rate is 0.0633.

Discussion

K-Nearest Neighbors classifier has the advantage that no training time and no brain on the part of the designer are required. However, the memory requirement and recognition time are large: the complete 38000 28 by 28 pixel training images(about 70 Megabytes) must be available at run time. As we can see in Algorithm 1, the time complexity of this approach is $O(|train||test||pixel|)$, when we do cross validation it takes about 10 hours.

A summary of the performance of our classifiers is shown in Figure 11. Although K-Nearest Neighbors and Random Forest both did well on the test set, K-Nearest Neighbors

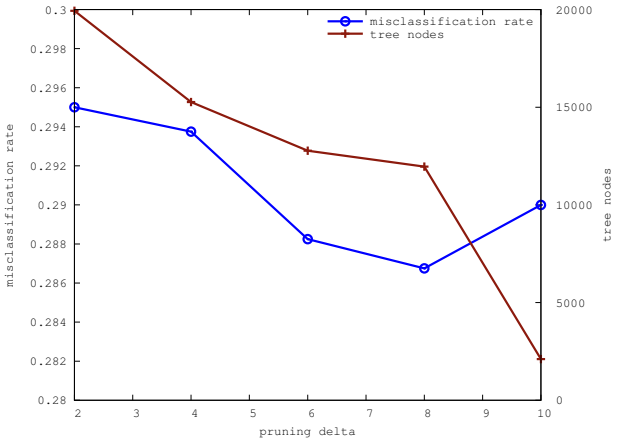

Figure 6: Misclassification rate vs pruning epsilon in a Decision Tree classifier. On the left,no pruning, the model is complex and hence we overfit. On the right, pruning too much, the model is simple and we underfit. Blue line: misclassification rate. Red line: tree node number

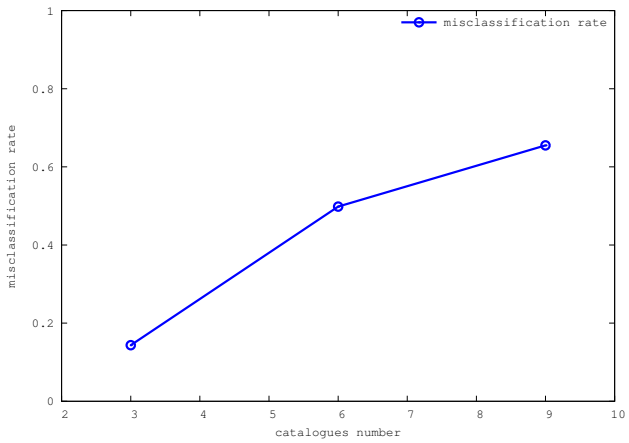

Figure 7: Misclassification rate vs catalogues number in a Random Forest classifier.

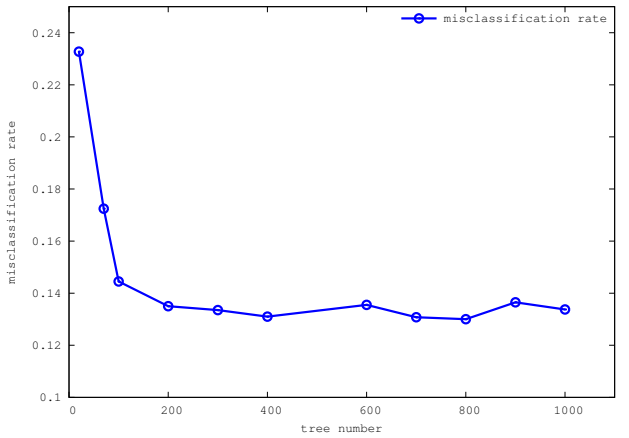

Figure 8: Misclassification rate vs tree number in a Random Forest classifier.

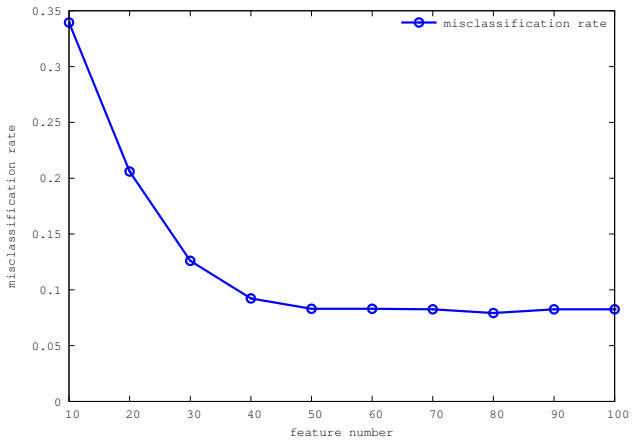

Figure 9: Misclassification rate vs feature number in a Random Forest classifier.

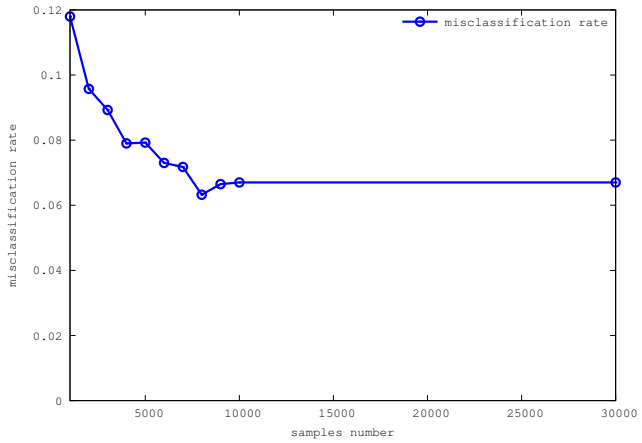

Figure 10: Misclassification rate vs sample number in a Random Forest classifier.

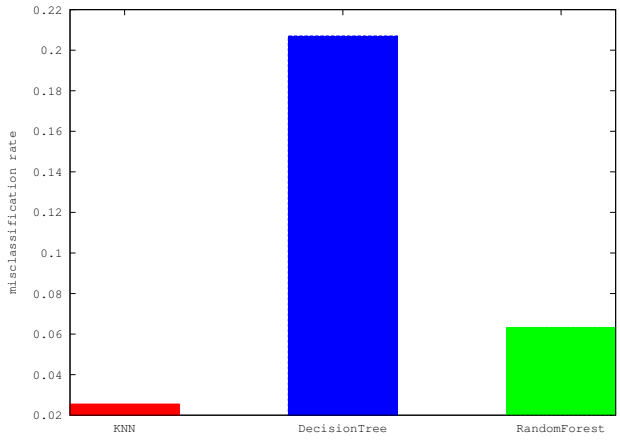

Figure 11: Misclassification rate comparison of K-Nearest Neighbors, Decision tree, and Random Forest classifiers.

is clearly the best, achieving a score of 0.025, followed by Random Forest at 0.0633.

Future Work

In the future we would like to run the same comparison but with more experiments. It is clear to see that both the K-Nearest Neighbors and the Random Forest perform not bad. I am curious if Random Forest would beat K-Nearest Neighbors given more trees and samples.

Also there are several machine learning approaches that I would like to try. For example, support vector machine and neural nets.

Acknowledgments

The authors would thank the members of the UNH AI Group and CS880 for their insightful comments.

References

Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, Cambridge, Massachusetts, 2012.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, 3rd Edition*. Pearson Education, Upper Saddle River, New Jersey, 2010.

Y. Bengio, Y. LeCun, C. Nohl and C. Burges. LeRec: A NN/HMM Hybrid for On-Line Handwriting Recognition. *Neural Computation*, 7(6):1289–1303, November 1995.

Y. LeCun, L. D. Jackel, L. Bottou, A. Brunot, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. A. Muller, E. Sackinger, P. Simard and V. Vapnik. Comparison of learning algorithms for handwritten digit recognition, in Fogelman, F. and Gallinari, P. (Eds). *International Conference on Artificial Neural Networks*, 53–60, EC2 & Cie, Pairs, 1995.