Letter Recognition using k-NN and Boosting

Daniel T. Babalola

Abstract

In this paper, I carry out a letter recognition task using the k-NN algorithm. I examine how the error of this k-NN algorithm varies with the value of k and pick an optimal k based on the results. I also examine how the number of training samples used to train the k-NN, influences the error given one k value. I also implement a boosting algorithm on this k-NN. The results show how well the k-NN is suited for this classification task and how boosting can be used to reduce the number of training samples required to train the k-NN.

1. INTRODUCTION

The main point of this paper was to see how well the k-NN algorithm performed on multi-classification tasks as well as to see how it could be improved to overcome its computational drawbacks. Various machine learning algorithms have been developed and improved over the last couple of decades. Many tasks once thought impossible for computers are now done with tremendous ease. Emulating various human approaches to solving problems, computers are now being to perform much "smarter" than ever thought possible (Chase & Simon, 1973). In this paper, I explore the problem of letter recognition. This is a popular learning task that has been investigated by various authors in previous years. The reason k-NN was the model of choice was because of the unique way in which it carries out learning. Like the human approach, it classifies future learning examples using previously recognized instances. This makes it well suited for a task such as letter recognition, where same letters always have the same characteristics regardless of font, size and color.

This paper investigates the performance of the k-NN by trying to see how the learning algorithm performs in different configurations. We perform tests with increasing values of k to see how well k performs with a larger number of neighbors being considered for classification. In theory, it already known that the choice of k has no definite formula and could vary based on the nature of the task being worked on or the representation in the dataset. Theoretically, a very large k could increase the 'smoothing' effect of the voters so much that the k-NN is unable to correctly classify most examples, especially when there is limited training data. Conversely, a very small k runs the risk of increasing the bias of the algorithm as it is more prone to not recognizes faults in the data such as outliers and noise. Also, we explore how necessary it is to have a large number of training examples required for training. In practice, it is always advisable to have as much training data as

possible as this increases the chances of the dataset representing future data much more accurately. Also, a large number of training examples helps to reduce the effects of noise on the whole learning process. In this paper, I performed tests on increasing values of n training samples and saw just how important, the number of training samples could prove to be in any learning task.

In this paper, I decided to also implement boosting to the traditional k-NN algorithm. Boosting has already been recognized as a way to improve the performance of existing learning algorithm. It has been shown to work really well with weak learners i.e. learning algorithms that classify only slightly better than random guessing. With boosting, a weak learner can be made to perform just as good as a powerful learner. This works by running several instances of the learning algorithm over varied distributions of the data set. Unlike bagging, which simply selects random portions of the dataset, boosting works by altering the distribution after each iteration. Therefore, with each completed round, more emphasis is placed on misclassified examples, forcing the weak learner to correctly predict even the hardest of training samples. Boosting has been successfully carried out on several algorithms, a few of which include, decision trees, k-NN, SVM etc. In this paper, I developed and implemented a boosting algorithm and compared its performance with its base model. It is seen that boosting, does in fact improve its performance as would be elaborated in further sections.

To conduct these experiments, I used the Letter Recognition Data Set found at the UCI Machine Learning Repository website. It consists of 20000 black and white capital letters of the English alphabet. Created by David Slate (Frey & Slate, 1991), these data samples were based on 20 different fonts and then randomly distorted. Each data sample contains 16 numeric attributes which were then scaled into a range of integer values from 0 through 15. The attributes contained basic descriptions of each character and included information such as total number of pixels, height and width of the box, etc. Following convention, these 20000 samples were split into two: 16000 as the training set and 4000 as the test set. The training set was then further split into two:12000 as the actual training set and the remaining 4000 were used as the validation set. This way, we could run several tests on our training sample before performing our final run on the test set.

2. BACKGROUND

In 1951, Fix and Hodges came up with a non-parametric method for pattern classification. This later became known as the k-NN (Fix & Hodges, 1951). It remains one of the simplest yet effective classification methods. The k-NN is usually used at the start of any classification study when there is little or no knowledge about the data distribution. It works on the assumption that similar data tend to possess similar characteristics and so would be located around each other on a dimensional plane. Based on this assumption, the algorithm classifies new instances by computing the k nearest training examples and then assigning the label of the majority of these points.

Soon enough, people started to carry out research with the aim of improving the performance of the k-NN. Because, even though it seemed to perform quite well on many classification tasks, it still consumed a lot of memory and computation resources as every round of classification required that the k-NN compute the distance of all the training points available. In 1968, Hart introduced the Condensed Nearest Neighbor Rule, CNN as it is known today. The underlying algorithm was the same, however, it worked at reducing the number of training data required by the k-NN (Hart, 1968). To do this, a smaller subset of the original sample set was created. A random sample from

the original set was added to this smaller set. On each round of iteration, the round ends when the smaller subset fails to classify a sample from the training set, this sample is then added to the smaller subset. At the end of the training, the smaller subset, though containing fewer examples than the original sample set at the beginning, would now be able to classify future instances just as good as the original sample set. This way, we could easily speed up the time required to classify future instances.

In 1972, Gates introduced the reduced nearest neighbor rule which claimed to perform even better than the CNN (Gates, 1972). Just like the CNN, Gates also tried to reduce the number of samples required to correctly classify future samples. It tries to modify the CNN even further by removing samples that may not be needed. His results showed that this was indeed possible.

Finally, in 1996, Freund and Schapire introduced a new boosting algorithm called Adaboost (Freund & Schapire, 1996). Their boosting algorithm, more specifically Adaboost.M2, focused on adjusting the distribution of examples based on an error measure called the *pseudo-loss*. This way, the learner was forced to focus not only on hard-to-classify examples but also on incorrect labels. Adaboost.M2 was combined with the k-NN classifier in this paper. Here, just like with the CNN, the goal was to reduce the number of samples required to correctly classify future instances. The algorithm worked by building a prototype set. At each iteration, ten candidate prototypes were selected at random, the candidate that caused the largest decrease in pseudo-loss was then added to the prototype set. This was repeated until the prototype set had reached a specified size. Adaboost.M2 outperformed both the traditional k-NN and the CNN as it focused on selecting samples based on the pseudo-loss, and the distribution of training examples. This way, a sufficiently high accuracy could be achieved with a minimal amount of training examples. Their work is what has inspired my research; that is, to explore in more detail how boosting affects the k-NN.

3. METHODOLOGY

As previously discussed, the k-NN works by storing all the training data. This is done in a single unit of time. Most of the work is done in classifying new instances. The algorithm first calculates its distance from all the training points, it then selects its k nearest neighbors and finally, uses a majority vote to predict its label. in this paper, I implemented the k-NN algorithm using a package called *scikit-learn 0.22* (Pedregosa, et al., 2011), a software module based on the Python programming language. It's a very powerful module which makes it possible to readily use several machine learning algorithms. By using the *KNeighborsClassifier* class, the k-NN can easily be implemented in only a few words. Scikit learn also allows for various manipulations such as defining the number of neighbors to consider for a majority vote (a feature which would be heavily referenced as we go on), adjust the weights of the nearest neighbors, select the distance measure used to select the nearest neighbors and much more. Its methods are shown in *figure 1* below.

Methods

`fit`(self, X, y)	Fit the model using X as training data and y as target values
`get_params`(self[, deep])	Get parameters for this estimator.
`kneighbors`(self[, X, n_neighbors, ...])	Finds the K-neighbors of a point.
`kneighbors_graph`(self[, X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
`predict`(self, X)	Predict the class labels for the provided data.
`predict_proba`(self, X)	Return probability estimates for the test data X.
`score`(self, X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
`set_params`(self, **params)	Set the parameters of this estimator.

Figure 1: Methods in scikit-learn KNeighbors Classifier

For the implementation of this k-NN algorithm, I used the standard Euclidean distance metric to find our k nearest neighbors. This was to keep the algorithm as basic as possible to allow for more focus on the variables I intent to work with, k and n. The Euclidean metric is one of many ways to calculate the distance of new instances from training samples. It works by simply finding the squared root of the sum of the squared differences between the new instance and previously existing points. It is shown by the formula below:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Others include the Manhattan distance which works by calculating the distance between real vectors using the sum of their absolute difference, and the Minkowski distance which is a mere generalization of the Euclidean and Manhattan distance.

The boosting algorithm was also implemented in python. Following from the algorithm Adaboost.M2 worked on by Schapire and Freund, the algorithm works in a similar way.

Boosting algorithm

Input: sequence of training samples with labels
K-NN algorithm
Integer ***n*** specifying size of random checks
Integer ***p*** specifying prototype size

Steps:

1. Select ***n*** random points
2. For each point + prototype set, train on remaining examples
3. Select point which causes largest decrease in error and add to prototype set ***P***
4. Repeat until ***P*** is of defined size ***p***

I implemented this algorithm to build of the scikit learn framework. Initially, the prototype set is empty. Then n, in this case 10, random samples are selected. The candidate which causes the lowest decrease in training error is then added to the prototype set P. This process is then repeated until P is of expected size. Figure 2 below shows a screenshot of the training process. It's important to note that only candidates that cause a decrease in error are selected, as opposed to those which simply have the lowest error out of the ten points. Else, the algorithm builds a prototype set consisting of limited labels. This kind of prototype set performs poorly on both the validation and test set.

```
Length at the moment: 4
Current error: 0.8641666666666666
Lowest training error out of ten random points is:  0.841
Attribute value of selected candidate:  [12926    25     4     9     6
     4    10     1     8     6    10]
Round completed: 5

Length at the moment: 5
Current error: 0.841
Lowest training error out of ten random points is:  0.8216666666666667
Attribute value of selected candidate:  [10409    21     6     7     5
    11     7     3    10     1     8]
Round completed: 6
```

Figure 2: A quick glance at the boosting algorithm undergoing training

Just like Adaboost, this algorithm influences the k-NN first by constantly randomizing the samples being selected. And second, by forcing the k-NN to select labels that cause a decrease in the training error. This has a similar effect to adjusting the distribution of training samples on each round as the k-NN is forced to always select the most 'relevant' training samples. It should be noted however that because this is a sequential process, i.e. all n samples must have been trained with before the sample with the lowest point can be chosen, and this process is repeated as each point is added to the prototype set, the boosting process takes up a lot of time and computation resources.

4. RESULTS AND DISCUSSION

The first experiment I carried out was to see how well the k-NN performed with varying values of k. As said earlier, it is expected that as k becomes increasingly large, the training error grows as well. This was exactly the case. I trained with a value of k from 1 to 100. As shown in figure 3 below.

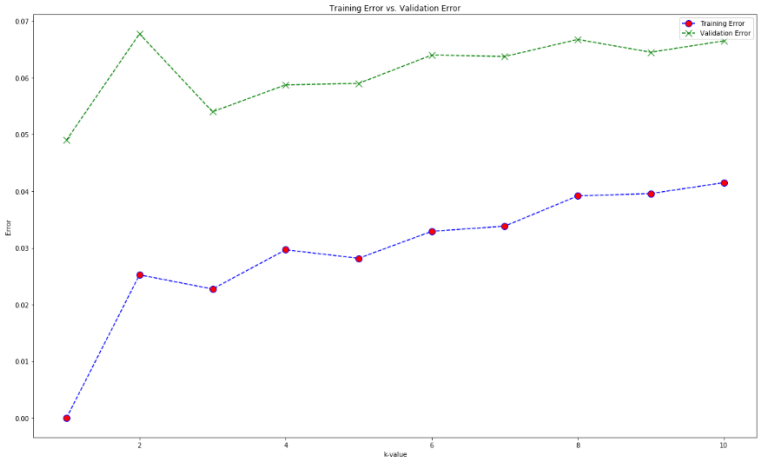

Figure 3: Training and Validation error as the value of k grows

I decided to use a k value of 3 as a value of 1 seemed more prone to variance. Next, I trained the k-NN using increasing sizes of samples. This was done by randomly selecting *n* portions of the training data and fitting it directly in to the k-NN, with a k value of 3. Just like in theory, there is a direct relationship with the number of training samples available and the error expected. The k-NN performs consistently better as the number of samples used to train grows. Figure 4 and table 1 below shows how the value of n affects the performance of the k-NN.

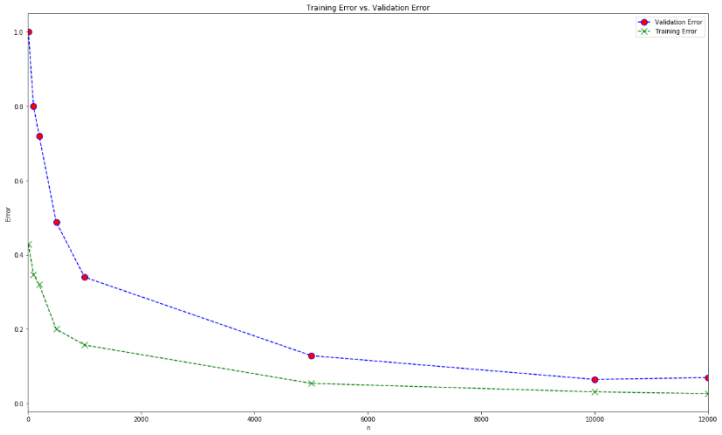

Figure 4: Training and Validation error as the number of training samples, n, grows

It must be noted however that after a certain number of training samples, here about 5000 samples, the effects of adding more samples seems to have a limited boost on the overall performance of the k-NN. In our experiment, the addition of 7000 training examples to the existing 5000 barely reduced the validation error by 5%.

n	E(train)	E(val)
10	42.9	100.0
100	34.7	80.0
200	32.0	72.0
500	20.0	48.8
1000	15.7	34.0
5000	5.4	12.9
10000	3.1	6.5
12000	2.7	7.0

Table 1: Training and Validation error as n grows

At the end of the first round of experiments, with a k value of 3 and 12000 training examples, I was able to achieve and accuracy of 93.8% i.e. a test error of 6.2%. This was also with a training error of 2.7% and a validation error of 7%. An accuracy this high shows that the k-NN performs really well on the selected task.

When we carried out boosting on the k-NN, we were able to significantly reduce the number of training samples required to still get a similar training error. For example, with only 200 examples, we were able to achieve an error of 48.3%, a big improvement from the 72% achieved without boosting. Just like the effects of increasing the sample size however, it is also seen than the gain from boosting also reduced as the prototype set increased. There was a gain of 23.7% at 200 examples but only 9.7% at 1000 examples. Nevertheless, with boosting, the k-NN performed consistently better at the same number of examples. Due to constraint on computation resources, I was only able to build a prototype set of 2000 samples. This set achieved an error of 17.4%, a considerable improvement given that the traditional k-NN achieves this error at about 5000 training samples. Figure 5 and Table 2 below illustrate this.

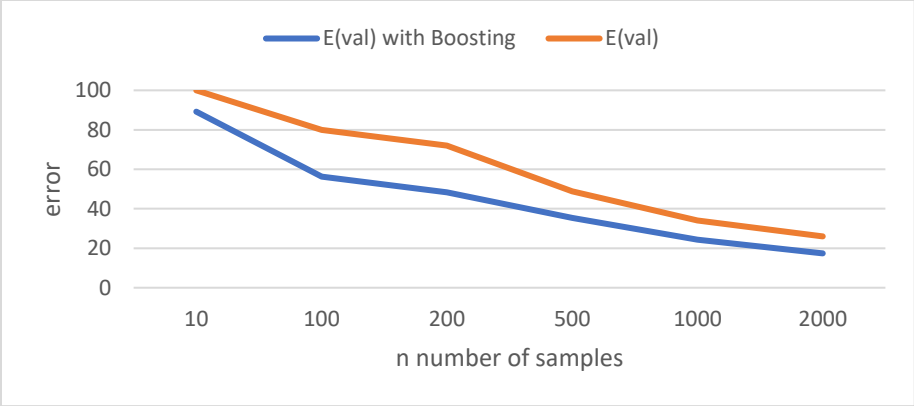

Figure 5: Validation Error with and without Boosting

n	E(val)	E(val) with Boosting
10	100.0	89.2
100	80.0	56.2
200	72.0	48.3
500	48.8	35.4
1000	34.0	24.3
2000	26.0	17.4

Table 2: Validation Error with and without Boosting

With or without boosting, it is also seen that most of the mislabels occurred mainly on structurally similar letters such as (F,P), (G,C), (B,R) etc.

5. CONCLUSION

We have seen how successful the k-NN algorithm performed on this learning task. An accuracy of about 94% percent shows that the k-NN is indeed suited for classification tasks such as these, be it binary or multi-classification. We have also seen that the choice of k is very important as a larger k could lead to a higher training and validation error. Also, just like with other machine learning algorithms, it is desirable to have as many training examples as possible as this only improves the accuracy. We see how there was a consistent decrease in the error as the number of training samples grew.

However, a large number of training examples also poses some form of drawback to the k-NN which, being a lazy-learner, relies on performing new computations when each new instance is being classified. We circumvent this by boosting. This way, we can significantly reduce the number of examples required to correctly classify future data. Through boosting, we achieve similar accuracy with way less examples, thus reducing the amount of time spent computing distances which in turn improves the overall performance of the k-NN algorithm. It is seen that we can achieve a considerably lower error at similar sample sizes just by implementing the boosting algorithm. Furthermore, the boosting algorithm reduces the overall bias of the k-NN as several portions of the instance space are used to build the prototype set.

This shows that a boosted k-NN retains all the benefits of the traditional k-NN while avoiding most of its limitations, the chief of which is the lengthy computation time. It is worth considering however if the additional computation time required to boost the k-NN algorithm is more beneficial than working with the traditional algorithm. This could depend on a lot of factors such as nature of

task, number of training examples and available computational resources. Because the boosting process could take up a lot of time, it might turn out to be detrimental as opposed to using all the training samples available. Moreover, there exist other ways to improve the performance of the traditional k-NNs. Methods involving, deleting unnecessary attributes or even faster though not more efficient ways of reducing the number of training samples needed. In future, I would say that there is still a need to look into even more efficient boosting methods for the k-NN. This way, we retain not just the simplicity of classification but also greatly improve the practicality of use.

Acknowledgements

I would like to thank the UCI Machine Learning Repository for their careful attempt at maintaining various datasets. Thank you to Prof. Chang for the lectures we've had on Machine learning thus far. They provided me with the knowledge required to carry out this research.

References

Chase, W. G., & Simon, H. A. (1973). Perception in chess. *Cognitive Psychology, 4(1)*, 55–81.

Fix, E., & Hodges, J. (1951). *Discriminatory Analysis, Nonparametric Discrimination: Consistency Properties.* Technical Report 4, USAF School of Aviation Medicine, Randolph Field.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. *Machine Learning: Proceedings of the Thirteenth International Conference*, (pp. 148-156).

Frey, P. W., & Slate, D. J. (1991, March). Letter Recognition Using Holland-style Adaptive Classifiers. *Machine Learning Vol 6 #2*.

Gates, G. W. (1972, May). The reduced nearest neighbor rule. *IEEE Trans. Information Theory, vol. IT-18*, 431-433.

Hart, P. E. (1968, May). The condensed nearest neighbor rule. *IEEE Trans. Inform. Theory, vol. IT-14*, 515-516.

Pedregosa, Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., & Mathieu Blondel, P. P. (2011). Scikit-learn: Machine Learning in Python. *JMLR 12*, 2825-2830.