Combining knn and Divide and Conquer to reduce computational cost†

Quevedo J.R., Montañés E.

Artificial Intelligent Centre. Oviedo University. Viesques, E-33271 Gijón, Spain
{quevedo, elena}@aic.uniovi.es

Abstract. The main problem of using *knn* on large data sets is that its computational cost is $O(knn)=O(AN^2)$ for A attributes and N examples. In this paper it is proposed an instance-based Machine Learning System for continuous labelled examples based on *knn*. This new system, called Continuous *knn* Divide and Conquer (*ckndc*), incorporates a divide and conquer strategy that reduces its computational cost to $O(cknndc)=O(ANlogN)$. *cknndc* uses a *m5* based heuristic to select the best division in the sense of improving accuracy. Experimentally it is shown that *knn* and *cknndc* have similar accuracy.

1 Introduction

Instance-based approaches learn to remember a collection of prototypes. These prototypes might be just some of the training cases, or some hypothetical cases computed from one or more of them. The collection of prototypes that the algorithm *knn* collects is all training set. To evaluate a new test example, *knn* looks for the k nearest neighbours of the test example. This means that it iterates by all examples in the training set, so the order of *knn* is $O(knn)=(ANT)$ for N training examples, T test examples and A attributes. Then, supposing that $O(N)=O(T)$ then $O(knn)=(AN^2)$

In this paper it is proposed a new algorithm (*cknndc*) based on *knn* that uses divide and conquer to reduce the order to $O(cknndc)=O(ANlogN)$ for N examples and A attributes. In the divide phase it is used a heuristic based on the heuristic that *m5* [6] employs to partitioning a leaf when the tree is being constructed. The accuracy of *cknndc* is similar to *knn*.

2 Related Work

The accuracy and the computational cost of instance-based learning have been widely study [1]. To improve the computational cost there are several Example Selection techniques [2]. In this paper it is proposed another technique that does not select a

† The research reported in this paper has been supported in part under MCyT and Feder grant TIC2001-3579

subset of examples; otherwise it uses all of them. A divide and conquer algorithm is employed to reduce its computational cost.

It is very usual to apply divide and conquer method for model-based approaches, especially for decision trees [8] or for regression trees [6]. Domingos [3] proposes a "conquering without separating" induction strategy. In [7] it is also proposed strategies of combining divide and conquer model-based approaches with instance-based learning.

3 The ckndc algorithm

The general idea of applying divide and conquer over a data set consists of dividing recursively the set of training examples into smaller subsets and of applying *knn* to each subset. This idea presents two problems: firstly, the problem of which subset is applied when the algorithm evaluates a test example and secondly, the problem of dividing into subsets with accuracy similar to the accuracy of applying *knn*. In this section it is described *ckndc*, which solves these two problems and whose computational cost is $O(ckndc)=O(ANlogN)$ for N examples and A attributes.

The algorithm divides recursively the train set and the test set at the same time using a condition over an attribute for each single division. To find the best attribute and division it employs a heuristic based on the heuristic that *m5* uses to select the best division of a leaf when the tree is being constructed. The recursion ends when the number of examples of the train subset is below a fixed threshold (M). This threshold represents the maximum number of examples over the original *knn* is applied.

The *ckndc* algorithm is described as follows:

```
Error ckndc(ExamplesSet Train, ExamplesSet Test)
{
 if(#Test==0) return 0; //There is no error

 //If the number of examples of train is lower than M
 //the original knn is applied
 if(#Train<M) return cknn(Training, Test);

 //The train and test sets are divided into two subsets
 {Train1,Test1,Train2,Test2}=Divide(Train,Test);

 //Calculus of the error of each subset(recursive call)
 Err1=ckndc(Train1,Test1);
 Err2=ckndc(Train2,Test2);

 //To calculate and return the global error
 return (Err1*#Test1+Err2*#Test2)/(#Test1+#Test2);
}
```

ckndc looks for an example and attribute in the train set and takes its value to divide the train and the test sets. The division has two objectives: to make *ckndc*

faster and to keep similar accuracy. In subsection 3.1 it is detailed how to choose an example for each attribute in a very fast way and in the subsection 3.2 it is detailed the selection of the best division in order to get good accuracy.

3.1 Dividing the train and test set

It is important to notice that the division of the examples can not be done by its category because it is going to be applied both for train and test sets and test examples do not have information about its category.

The algorithm *cknndc* chooses a train example for each attribute to define a division. It is important the search of the example to be fast and the division to be near 50%. If the division splits the train set into two subsets with very different size the order of the algorithm increases. To select quickly an example that splits the set into two subsets with similar number of examples *cknndc* utilizes two algorithms, one for continuous attributes and another for symbolic ones.

To select an example for a continuous attribute *cknndc* takes one example randomly and calculates the number of examples of each subset. If the percent of the number of examples of the bigger subset is below a threshold (*P*) the example is accepted, else a new example is randomly taken. This search is repeated only *M* times at the most. The algorithm is described as follows:

```
Example SearchExampleCont(Examples SetExa, Attribute A)
{
 Continuous Min=+Inf,Max=-Inf;
 int iterations=0;
 do{
  iterations++;
  //Takes a random example from SetExa whose its value
  //for attribute A is between Min and Max
  Example E=RandomExample(SetExa,A,Min,Max);

  //Calculus of the number of examples higher and lower
  Percent Lower=PercentLower(SetExa,A,E);
  Percent Higher=PercentHigher(SetExa,A,E);

  //If E divides the set into two subsets with similar
  //number of examples, the example E is accepted
  if(MAX(Lower,Higher)<P) return E;

  //Update the interval of attribute A for the forward
  //examples (E.A is the value of attribute A for E)
  if(Lower>Higher) Max=E.A;
  if(Higher>Lower) Min=E.A;

 }while(iterations<M);
 return E;
}
```

To select an example for a symbolic attribute *cknndc* iterates for each value of the attribute and calculate the percent of examples of the set that has the same value. The value with the percent nearest to 50% is selected. The algorithm is described as follows:

```
Example SearchExampleSymb(Examples SetExa, Attribute A)
{
  //It is calculated the best value as the one with
  //best percent
  Percent BestPercent=100.0;
  Value BetsValue;
  for Value v=each value of attribute A
  {
    //The percent of examples that has the value 'v'
    Percent Per=PercentSame(SetExa,A,v);

    //The percent nearest to 50% is the best
    if(|Per-50|<|BestPercent-50|)
    {
     BestValue=v;
     BestPercent=Per;
    }
  }
 //It returns an example that has the value BestValue
 return FirstExampleOFValue(SetExa,BestValue,A);
}
```

3.2 Selecting the best division

In the latter subsection it is calculated a division for each attributes being the objective of this phase to select the division that gives the best accuracy. To carry out this issue *cknndc* employs a heuristic based in the heuristic of *m5* that uses to select the best division of a leaf when the tree is being constructed. The algorithm *m5* looks for the division that minimizes the sum of the standard deviation of the categories of each subset. This heuristic, which is fine to make regression trees, has a problem: the division could separate examples of its nearest neighbours for too many examples. A good criterion should tries to not separate the examples of its nearest neighbours.

The heuristic used in *cknndc* tries to not divide if the values of the examples for the attribute of the division are too near to the division (see figure 1). A better division is that which creates two subsets that group the nearest neighbours to each other.

The mean of the distance to the division (*MDD*) is the heuristic that *cknndc* utilizes to determine the quality of a division in the way of not separating examples from its neighbours. Combining this heuristic with m5`s heuristic it is obtained the *cknndc*'s heuristic *SelectDivision* (See equation 1 for a particular case of two attributes). This heuristic is the same of m5's but dividing each standard deviation by the *MDD*.

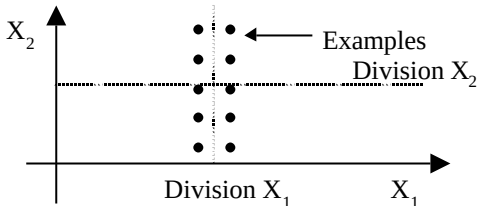

Fig. 1. This figure represents ten examples with two attributes. The nearest example of an example is the one at its left or right. There are two possible divisions, division X_1 and division X_2. It is better to choose division X_2 in order to group the examples with its nearest neighbours.

$$SelectDivision(A,e) = \frac{SD_1(A,e)}{MDD_1(A,e)} + \frac{SD_2(A,e)}{MDD_2(A,e)} \quad (1)$$

The heuristic *SelectDivision* is applied to each division. SD_i and MDD_i represent the standard deviation and the mean distance to the division of the two subsets that creates the division, that are called 1 and 2. The division finally accepted is the one whose *SelectDivision* is lower. The algorithm *Divide* is described as follows:

```
{Examples Train1, Test1, Train2, Test2} Divide(Examples
                                          Train, Test)
{
  Continuous LowerSelectDiv=+Inf;
  Attribute BestAttr;  //These variables remember the
  Example BestEx;      //best division
  for Attr=each Attribute
  {
   //To calculate the division
   Example ExDiv;
   if(IsSymbolicAttribute(Attr))
     ExDiv=SearchExampleSymb(Train,Attr);
   else
     ExDiv=SearchExampleCont(Train,Attr);

   //To calculate the best division
   if(SelectDivision(Attr,ExDiv)<LowerSelectionDiv)
   {
    LowerSelectionDivision= SelectDivision(Atr,ExDiv);
    BestAttr=Attr;
    BestEx=ExDiv;
   }
  }
 {Train1,Train2}=CreateSubSets(Train,BestAttr,BestEx);
 {Test1,Test2}=CreateSubSets(Test,BestAttr,BestEx);
 return { Train1, Test1, Train2, Test2};
}
```

3.3 The order of ckndc

The order of *cknndc* depends on *N*, *T* and *M*, which are respectively the number of examples in the train set, in the test set and the maximum number of examples over the original *knn* is applied. It is supposed that *N*, *T* and *M* are integers so that *T,N>M>1*. It also depends on the number of attributes (*A*). This program creates a binary tree of recursive calls, where in each node, except in the leaves, a *Divide* function is executed. In the leaves the original *knn* is executed. If it is supposed that the function *Divide* splits the set into two subsets with equal number of examples then in the *i* level of recursion the number of examples that *Divide* treats is $N/2^i$. As the number of examples in the leaves is *M*, the depth at the leaves is $log_2(N/M)$. The average of the number of examples in the nodes is called *NDiv* and is estimated as follows:

$$NDiv = \frac{\sum_{i=0}^{\log_2 \frac{N}{M}-1} 2^i \frac{N}{2^i}}{\sum_{i=0}^{\log_2 \frac{N}{M}} 2^i} = \frac{N \log_2 \frac{N}{M}}{\sum_{i=0}^{\log_2 \frac{N}{M}} 2^i} < \frac{N \log_2 \frac{N}{M}}{\sum_{i=0}^{\left\lceil \log_2 \frac{N}{M} \right\rceil} 2^i} = \frac{N \log_2 \frac{N}{M}}{2^{\left\lceil \log_2 \frac{N}{M} \right\rceil} 2-1} \tag{2}$$

$$\frac{N \log_2 \frac{N}{M}}{2^{\left\lceil \log_2 \frac{N}{M} \right\rceil} 2-1} < \frac{N \log_2 \frac{N}{M}}{2^{\log_2 \frac{N}{M}-1} 2-1} = \frac{N \log_2 \frac{N}{M}}{2^{\log_2 \frac{N}{M}} -1} = \frac{N \log_2 \frac{N}{M}}{\frac{N}{M}-1} = \frac{MN \log_2 \frac{N}{M}}{N-M} \tag{3}$$

The number of executions of *Divide* and *knn* are limited by *2N/M*, so *O(cknndc)=O(2N/M(O(Divide)+O(knn))*. The order of *Divide* is *O(Divide)= O(A(O(SearchExample)+O(SelectDivision))+O(CreateSubSets))*. The order of all these functions is the number of examples treated in each execution, which is *NDiv* and is limited by the expression in equation (3). Then, the order of *Divide* is shown in equation (4).

$$O(Divide) = O(A(NDiv + NDiv) + NDiv) = O(A \cdot NDiv) = O\left(A \frac{MN \log_2 \frac{N}{M}}{N-M} \right) \tag{4}$$

If *M* is supposed to be a constant, *O(Divide)* is simplified (see equation (5)).

$$O(Divide) = O\left(A \frac{MN \log_2 \frac{N}{M}}{N-M} \right) = O(A \log_2 N) \tag{5}$$

To calculate the order of *knn* it is necessary to estimate the number of examples that are considered as train and as test. The number of train examples is limited by *M*.

To estimate the number of examples of test it is supposed that the train set and the test set have the same distribution of examples. This supposition is generally accepted in Machine Learning. If the original train set has *N* examples and at the leaves the *knn* uses *M* for train, the same proportion of test examples *knn* would use. The number of examples that *knn* uses as test is called *TDiv* and equation (6) shows this value.

$$TDiv = \frac{T}{\frac{N}{M}} = M\frac{T}{N} \tag{6}$$

Equation (7) shows the order of *knn*, which is the product of the number of examples of train (*M*), the number of examples of test (*TDiv*) and the number of attributes (*A*). It is also supposed that *M* is constant.

$$O(knn) = O(M \cdot TDiv \cdot A) = O\left(AM^2\frac{T}{N}\right) = O\left(A\frac{T}{N}\right) \tag{7}$$

So the order of *ckNNdc* is shown in equations (8) and (9).

$$O(ckNNdc) = O(2\frac{N}{M}(O(Divide) + O(knn)) = O\left(N\left(O(A\log_2 N) + O\left(A\frac{T}{N}\right)\right)\right) \tag{8}$$

$$O\left(N\left(O(A\log_2 N) + O\left(A\frac{T}{N}\right)\right)\right) = O(AN\log_2 N) + O(AT)) \tag{9}$$

If it is supposed that *O(N)=O(T)*, then equation (10) shows the order of *ckNNdc*.

$$O(ckNNdc) = O(AN\log_2 N) \tag{10}$$

4 Experimental Evaluation

The well known heterogeneous data sets of the Torgo's repository at *LIACC* [5], described in table 1, are used to compare the performance of *knn* and *ckNNdc*. Each experiment consists of a Cross Validation (*CV*) with 10 folds. Besides, it is employed *MLC++*[4] with 2032 seed (our office's telephone) to make the experiments to be repeatable.

The value of *k* for both algorithms varies from 1 to 3. *ckNNdc* has two additional parameters: *M*, which determines the maximum number of examples that use the original *knn* and *PMax*, which determines the maximum percent of examples it could be in a subset. In our experiments *M* is set to 200 and *PMax* is set to 60%.

The result of a *CV* experiment is the Medium Average Deviation (*MAD*), but in table 2 it is shown the Relative Medium Average Deviation (*RMAD*) which is the *MAD* divided by the *MAD* of the system that always predicts the average function. Its execution time is shown in figure 2.

Table 1. List of the data sets of the Torgo's repository. The name, the number of examples (#Ex), the number of attributes (#Att) and the *MAD* of the system that always predict the average function (Av. MAD) are shown for each data set. Each data set is also numbered (Nº) to be referred forward using this number.

Nº Name	#Ex	#Att	Av.MAD	Nº Name	#Ex	#Att	Av.MAD
1 Abalone	4177	8	2,363	16 Diabetes	43	2	2,363
2 Ailerons	13750	40	0,0003	17 Elevators	16599	18	0,0046
3 Airpla.Com.	950	9	5,4852	18 Friedman Ex.	40768	10	4,0648
4 Auto-Mpg	398	4	6,5459	19 Housing	506	13	6,6621
5 Auto-Price	159	14	4600,65	20 Kinematics	8192	8	0,2156
6 Bank 32NH	8192	32	0,0903	21 Machine-Cpu	209	6	96,9004
7 Bank 8FM	8192	8	0,1236	22 MvExample	40768	10	8,8932
8 Cal. Hou.	20640	9	91174,5	23 PoleTele.	15000	48	37,2124
9 Cart Delve	40768	10	3,6069	24 Pumadyn(32)	8192	32	0,0235
10 Census(16)	22784	16	32428,2	25 Pumadyn(8)	8192	8	4,8659
11 Census(8)	22784	8	32428,2	26 Pyrimidines	74	27	0,0957
12 Com.Act	8192	21	10,6326	27 Servo	167	2	1,1662
13 Com.Act(s)	8192	12	10,6326	28 Triazines	186	60	0,1187
14 Delta Ailer.	7129	5	0,0003	29 Wisconsin	198	32	29,6833
15 Delta Eleva	9517	6	0,002				

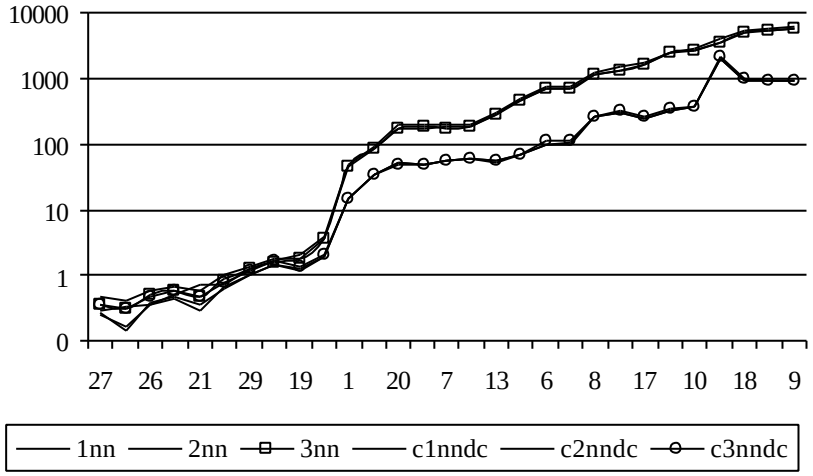

Fig. 2. Execution time of a Cross Validation with ten folders using *knn* and *cknndc* for $k=\{1,2,3\}$ in a logarithmic scale versus the data sets ordered by 1nn execution time.

In figure 2 it can be observed the great improvement of execution time between *knn* and *cknndc* for large data sets.

Table 2. *RMAD* of *knn* and *ckndc* with $k=\{1,2,3\}$. It is shown the *RMAD* for each data set of Torgo's repository and the average of all *RMADs* (*Av*.) over all data sets.

	Knn			*ckndc*		
	k=1	k=2	k=3	k=1	k=2	k=3
1	84.63%	74.44%	70.97%	85.28%	76.31%	73.03%
2	66.67%	33.33%	33.33%	66.67%	66.67%	33.33%
3	9.60%	8.49%	8.57%	11.35%	10.56%	10.82%
4	39.18%	36.76%	35.28%	39.08%	36.93%	35.50%
5	44.61%	41.84%	41.58%	44.61%	41.84%	41.58%
6	104.87%	96.23%	93.02%	109.63%	100.89%	97.23%
7	69.26%	60.68%	59.22%	76.46%	68.85%	67.80%
8	53.74%	48.56%	46.77%	55.88%	50.81%	49.26%
9	35.22%	32.30%	30.95%	58.68%	49.46%	46.86%
10	73.48%	65.84%	63.38%	76.78%	69.72%	67.53%
11	69.57%	62.46%	59.67%	71.41%	64.39%	61.49%
12	21.37%	18.86%	17.97%	23.77%	21.71%	21.22%
13	24.55%	21.81%	20.86%	26.68%	23.86%	23.07%
14	33.33%	33.33%	33.33%	33.33%	33.33%	33.33%
15	70.00%	65.00%	60.00%	70.00%	65.00%	60.00%
16	129.04%	98.00%	95.75%	129.04%	98.00%	95.75%
17	60.87%	54.35%	52.17%	65.22%	60.87%	58.70%
18	53.89%	43.97%	40.18%	59.68%	50.24%	47.44%
19	46.12%	40.08%	40.01%	46.38%	40.46%	41.55%
20	59.37%	49.03%	45.45%	63.08%	54.73%	51.58%
21	35.97%	32.09%	31.18%	35.09%	31.98%	31.18%
22	19.94%	15.96%	14.68%	22.27%	19.51%	19.07%
23	7.50%	7.18%	7.22%	10.01%	10.10%	10.78%
24	122.55%	105.96%	100.00%	124.68%	108.94%	102.55%
25	78.15%	67.96%	63.85%	81.18%	71.13%	67.82%
26	68.65%	64.26%	63.32%	68.65%	64.37%	63.32%
27	41.36%	67.62%	83.61%	40.92%	69.07%	83.61%
28	91.07%	81.97%	83.15%	91.49%	82.48%	83.15%
29	125.56%	101.10%	97.42%	125.56%	101.10%	97.42%
Av.	60.00%	52.74%	51.48%	62.51%	56.67%	54.34%

The average of the accuracy of *ckndc* is slightly worse than the average of the accuracy of *knn* for k={1,2,3}, but this difference is not relevant in contrast with the improvement reached in execution time.

5 Conclusions

The algorithm *cknndc*, presented in this paper, is an instance-based Machine Learning System based on *knn*. The main advantage is its computational cost, that is *O(cknndc)=(ANlogN)*. This improvement is caused by the application of a divide and conquer strategy over the train and test sets. To procure the best computational cost *cknndc* tries always to split the original set into two subsets with similar number of examples. To overcome the lost of accuracy that the divide and conquer strategy could cause, *cknndc* uses a *m5* based heuristic that determines the best division.

The experiments presented shown that the accuracy of *cknndc* is similar to the accuracy of *knn* for a well known and varied data sets.

In future work we will study the application of this technique to symbolic labeled examples. We will also study the possible use of combining this technique with example or attribute selection.

6 References

1. Aha, D.W., Kibler, D., Albert, M.K.: Instance based learning algorithms. Machine Learning, Vol. 6. (1991) 37-66
2. Blum A.L., Langley. P.:Selection of relevant features and examples in machine learning. Artificial Intelligence. (1997) 245-271
3. Domingos, P. Unifying instance-based and rulebased induction. Machine Learning 24. (1996) 141-168
4. Kohavi, R., John, G., Long, R., Manley, D., & Pfleger, K. (1994). MLC++: A machine learning library in C++. In Proc. of the 6th International Conference on Tools with Artificial Intelligence, 740-743. IEEE Computer Society Press.
5. Torgo. L: Regression Data Sets Repository at LIACC (University of Porto). http://www.ncc.up.pt/~ltorgo/Regression/DataSets.html
6. Quinlan, J.R. Learning with continuous classes. Proceedings of the Fifth Australian Joint Conference on Artificial Intelligence, In A. Adams and L.Sterling. World Scientific. Singapore (1992) 343-348
7. Quinlan, J.R. Combining instance-based and model-based learning. In Machine Learning: Proceedings of the Tenth International Conference. Morgan Kaufmann. Amherst, Massachusetts (1993) 236-243
8. Quinlan, J.R. "C4.5: Programs for Machine Learning". Morgan Kaufmann. (1993)