# Adaptive k-Nearest Neighbor Classification Based on a Dynamic Number of Nearest Neighbors

Stefanos Ougiaroglou[1] Alexandros Nanopoulos[1] Apostolos N. Papadopoulos[1]
Yannis Manolopoulos[1] Tatjana Welzer-Druzovec[2]

[1] Department of Informatics, Aristotle University, Thessaloniki 54124, Greece
{stoug,ananopou,papadopo,manolopo}@csd.auth.gr
[2] Faculty of Electrical Eng. and Computer Science, University of Maribor, Slovenia
welzer@uni-mb.si

**Abstract.** Classification based on $k$-nearest neighbors ($k$NN classification) is one of the most widely used classification methods. The number k of nearest neighbors used for achieving a high precision in classification is given in advance and is highly dependent on the data set used. If the size of data set is large, the sequential or binary search of NNs is inapplicable due to the increased computational costs. Therefore, indexing schemes are frequently used to speed-up the classification process. If the required number of nearest neighbors is high, the use of an index may not be adequate to achieve high performance. In this paper, we demonstrate that the execution of the nearest neighbor search algorithm can be interrupted if some criteria are satisfied. This way, a decision can be made without the computation of all k nearest neighbors of a new object. Three different heuristics are studied towards enhancing the nearest neighbor algorithm with an early-break capability. These heuristics aim at: (i) reducing computation and I/O costs as much as possible, and (ii) maintaining classification precision at a high level. Experimental results based on real-life data sets illustrate the applicability of the proposed method in achieving better performance than existing methods.

*Keywords:* $k$NN classification, multidimensional data, heuristics.

## 1 Introduction

Classification is the data mining task [10] which constructs a model, denoted as *classifier*, for the mapping of data to a set of predefined and non-overlapping classes. The performance of a classifier can be judged according to criteria such as its accuracy, scalability, robustness, and interpretability. A key factor that influences research on classification in the data mining community (and differentiates it from classical techniques from other fields) is the emphasis on scalability, that is, the classifier must work on large data volumes, without the need for experts to extract appropriate samples for modeling. This fact poses the requirement for closer coupling of classification techniques with database techniques. In this paper we are interested in developing novel classification algorithms that are accurate *and* scalable, which moreover can be easily integrated to existing database systems.

Existing classifiers are divided into two categories [12], *eager* and *lazy*. In contrast to an eager classifier (e.g., decision tree), a lazy classifier [1] builds no general model until a new sample arrives. A $k$-nearest neighbor ($k$NN) classifier [7] is a typical example of the latter category. It works by searching the training set for the $k$ nearest neighbors of the new sample and assigns to it the most common class among its $k$ nearest neighbors. In general, a $k$NN classifier has satisfactory noise-rejection properties. Other advantages of a $k$NN classifier are: (i) it is analytically tractable, (ii) for $k = 1$ and unlimited samples the error rate is never worse than twice the Bayes' rate, (iii) it is simple to implement, and (iv) it can be easily integrated into database systems and exploit access methods that the latter provide in the form of indexes.

Due to the aforementioned characteristics, $k$NN classifiers are very popular and find many applications. With a naive implementation, however, the $k$NN classification algorithm needs to compute all distances between training data and a testing datum, and needs additional computation to get $k$ nearest neighbors. This negatively impacts the scalability of the algorithm. For this reason, recent research [5] has proposed the use of high-dimensional access methods and techniques for fast computation of similarity joins, which are available in existing database systems, to reduce the cost of searching from linear to logarithmic. Nevertheless, the cost of searching the $k$ nearest neighbors, even with a specialized access method, still increases significantly with increasing values of $k$.

For a given test datum, depending on which training data comprise its neighborhood, we may need a small or a large $k$ value to determine its class. In other words, in some cases, a small $k$ value may suffice for the classification, whereas in other cases we may need to examine larger neighborhoods. Therefore, the appropriate $k$ value may vary significantly. This introduces a trade-off: By posing a global and adequately high value for $k$, we attain good accuracy, but we pay high computational cost, since as described, the complexity of nearest neighbor searching increases for higher $k$ values. Higher computational cost reduces the scalability to large data sets. In contrast, by keeping a small $k$ value, we get low computational cost, but this may result to lower accuracy. What is, thus, required is an algorithm that will combine good accuracy and low computational cost, by locally adapting the required value of $k$. In this work, we propose a novel framework for a $k$NN classification algorithm that fulfills the aforementioned property. We also examine techniques that help us for finding the appropriate $k$ value in each case. Our contributions are summarized as follows:

– We propose a novel classification algorithm based on a non-fixed number of nearest neighbors, which is less time consuming than the known $k$-NN classification, without sacrificing precision.
– Three heuristics are proposed that aim at the early-break of the $k$-NN classification algorithm. This way, significant savings in computational time and I/O can be achieved.
– We apply the proposed classification scheme to large data sets, where indexing is required needed. A number of performance evaluation tests are conducted towards investigating the computational time, the I/O time and the precision achieved by the proposed scheme.

The rest of our work is organized as follows. The next section briefly describes related work in the area and summarized our contributions. Section 3 studies in detail the proposed early-break heuristics, and presents the modified $k$-NN classification algorithm. Performance evaluation results based on two real-life data sets are given in Section 4. Finally, Section 5 concludes out work and briefly discusses future work in the area.

## 2 Related Work

Due to its simplicity and good performance, $k$NN classification has been studied thoroughly [7]. Several variations were developed [2], like the distance-weighted $k$NN, which puts emphasis on nearer neighbors, and the locally-weighted averaging, which uses kernel width to controls the size of neighborhood that has large effect. All such approaches propose adaptive schemes to improve the accuracy of $k$NN classification in the case where not all attributes are similar in their relevance to the classification task. In our research we are interested in improving the scalability of $k$NN classification.

Also, $k$NN classification has been combined with other methods and, instead of predicting a class with simple voting, prediction is done by another machine learner (e.g., neural-network) [3]. Such techniques can be considered complementary to our work. For this reason, to keep comparison clear, we did not examine such approaches.

Böhm and Krebs [5] proposed an algorithm to compute the $k$-nearest neighbor join using the multipage index (MuX), a specialized index structure for the similarity join. Their algorithm can be applied to the problem of $k$NN classification and can increase its scalability. However, it is based on a fixed number of $k$, which (as described in Introduction) if it is not tuned appropriately, it can negatively impact the performance of classification.

## 3 Adaptive Classification

### 3.1 The Basic Incremental $k$-NN Algorithm

An algorithm for incremental computation of nearest neighbors using the R-tree family [9, 4] has been proposed in [11]. The most important property of this method is that the nearest neighbors are determined in their order of their distance from the query object. This enables the discovery of the $(k+1)$-th nearest neighbor if we have already determine the previous $k$, in contrast to the algorithm proposed in [13] (and enhanced in [6]) which requires a fixed value of $k$.

The incremental nearest neighbors search algorithm maintains a priority queue. The entries of the queue are Minimum Bounding Rectangles (MBRs) and objects which will be examined by the algorithm and are sorted according to their distance from the query point. An object will be examined when it reaches the top of the queue. The algorithm begins by inserting the root elements of the R-tree in the priority queue. Then, it selects the first entry and inserts its children. This procedure is repeated until the first data object reaches the top of the queue. This object is the first nearest neighbor. Figure 1 depicts the Incr-$k$NN algorithm with some modifications, towards adapting the algorithm for classification purposes. Therefore, each object of the test set is a query point and each object of the training set, contains an additional attribute which indicates the class where the object belongs to. The R-tree is built using the objects of the training set.

The aim of our work is to perform classification by using a smaller number of nearest neighbors than $k$ if this is possible. This will reduce computational costs and I/O time. However, we do not want to harm precision (at least not significantly). Such an early-break scheme can be applied since Incr-$k$NN determines the nearest neighbors in increasing distance order from the query point. The modifications performed to the original incremental $k$-NN algorithm are summarized as follows:

```
Algorithm Incr-$k$NN (QueryPoint $q$, Integer $k$)
1.  $PriorityQueue$.enqueue(roots children)
2.  $NNCounter = 0$
3.  while $PriorityQueue$ is not empty and $NNCounter \leq k$ do
4.      $element = PriorityQueue$.dequeue()
5.      if $element$ is an object or its MBR then
6.          if element is the MBR of $Object$ and $PriorityQueue$ is not empty
            and objectDist($q$, $Object$) $>$ $PriorityQueue$.top then
7.              $PriorityQueue$.enqueue($Object$, ObjectDist($q$, $Object$))
8.          else
9.              Report element as the next nearest object (save the class of the object)
10.             $NNCounter$++
11.             if early-break conditions are satisfied then
12.                 Classify the new object $q$ in the class where the most nearest neighbors
                    belong to and break the while loop. $q$ is classified using
                    $NNCounter$ nearest neighbors
13.             endif
14.         endif
15.         else if $element$ is a leaf node then
16.             for each entry (Object, MBR) in $element$ do
17.                 $PriorityQueue$.enqueue ($Object$, dist($q$, $Object$))
18.             endfor
19.         else /*non-leaf node*/
20.         for each entry $e$ in $element$ do
21.             $PriorityQueue$.enqueue($e$, dist($q$, $e$))
22.         endfor
23.     endif
24. end while
25. if no early-break has been performed then // use $k$ nearest neighbors
26.     Find the major class (class where the most nearest neighbors belong to)
27.     Classify the new object $q$ to the major class
28. endif
```

**Fig. 1.** Outline of Incr-$k$NN algorithm with early break capability.

- We have modified line 3 of the algorithm. The algorithm accepts a maximum value of $k$ and is executed until either $k$ nearest neighbors are found (no early break) or the heuristics criteria are satisfied (early break). Specifically, we added the condition $NNCounter \leq k$ in while command (line 3).
- We have added the lines 10,11,12 and 13. In this point, the algorithm retrieves a nearest neighbor and checks for early break. Namely, the while loop of the algorithm breaks if the criteria defined by the heuristic that we use are satisfied. So, the new item is classified using $NNCounter$ nearest neighbors, where $NNCounter < k$. Lines 11, 12 and 13 are replaced according to the selected heuristic.
- We have added the lines 25, 26, 27 and 28. These lines perform classification taking into account $k$ nearest neighbors. This code is executed only when the heuristic did not manage to perform an early-break.

## 3.2 Early-Break Heuristics

In this section, we present three heuristics that interrupt the computation of nearest neighbors when some criteria are satisfied. The classification performance (precision and execution cost) depends on the adjustments of the various heuristics parameters. The parameter $MinnNN$ is common to all heuristics and defines the minimum number of nearest neighbors which must be used for classification. After the retrieval of $MinnNN$ nearest neighbors, the check for early-break is performed. The reason for the use of $MinNN$ is that a classification decision is preferable when it is based on a minimum number of nearest neighbors, otherwise precision will be probably poor. These criteria depend on which proposed heuristic is used. The code of each heuristic replaces lines 11 and 12 of the Incr-$k$NN algorithm depicted in Figure 1.

**Simple Heuristic (SH)** The first proposed heuristic is very simple. According to this simple heuristic, the early-break is performed when the percentage of nearest neighbors that vote the major class is greater than a predefined threshold. We call this threshold $PMaj$.

For example, suppose that we have a data set where the best precision is achieved using 100 nearest neighbors. Also, suppose that we define that $PMaj = 0.9$ and $MinNN$=7. The Incr-$k$NN is interrupted when 90% of NNs vote a specific class. If this percentage is achieved when the algorithm examines the tenth NN (9 out of 10 NNs vote a specific class), then we avoid the cost of searching the rest 90 nearest neighbors. Using the Incr-$k$NN algorithm, we ensure that the first ten neighbors which have been examined, are the nearest.

Furthermore, if the simple heuristic fails to interrupt the algorithm because $PMaj$ is not achieved, it will retry an early-break after finding the next $TStep$ nearest neighbors.

**Independent Class Heuristic (ICH)** The second early-break heuristic is the Independent Class Heuristic (ICH). This heuristic does not use the $PMaj$ parameter. The early-break of Incr-$k$NN is based on the superiority of the major class. Superiority is determined by the difference between the sum of votes of the major class and the sum of votes of all the other classes. The parameter $IndFactor$ (Independency Factor) defines the superiority level of the major class that must be met in order to perform an early-break. More formally, in order to apply an early-break, the following condition must be satisfied:

$$SVMC > IndFactor \cdot \sum_{i=1}^{n} SVC_i - SVMC \qquad (1)$$

where $SVMC$ is the sum of votes of major class, $n$ is the number of classes and $SVC_i$ is the sum of votes of class $i$.

For example, suppose that our data set contains objects of five classes, we have set $IndFactor$ to 1 and the algorithm has determined 100 NNs. Incr-$k$NN will be interrupted if 51 NNs vote a specific class and the rest 49 NNs vote the other classes. If the value of $IndFactor$ is set to 2, then the early-break is performed when the major class has more than 66 votes.

Studying the Independent Class Heuristic, we conclude that the value of the $IndFactor$ parameter should be adjusted by taking into account the number of classes and the class distribution of data set. In the case of a normal distribution, we accept the following rule: when the number of classes is low, $IndFactor$ should be set to a high value. On the other hand, when there are many classes, $IndFactor$ should be set to a lower value.

In ICH, parameter $TStep$ is used in the same way as in the SH heuristic. Specifically, when there is a failure in interruption, the early-break check is again activated after determining the next $TStep$ nearest neighbors.

**M-times Major Class Heuristic (MMCH)** The last heuristic that we present is termed M-Times Major Class Heuristic (MMCH). The basic idea is to stop the Incr-$k$NN when $M$ consecutive nearest neighbors, which vote the major class, are found. In other words, the while-loop of Incr-$k$NN algorithm terminates when the following sequence of nearest neighbors appears:

$$NN_{x+1}, NN_{x+2}, ..., NN_{x+M} \in MajorClass \tag{2}$$

However, this sequence is not enough to for an early-break. In addition, the $PMaj$ parameter is used in the same way as in the SH heuristic. Therefore, MMCH heuristic breaks the while loop when the percentage of nearest neighbors that vote the major class is greater than $PMaj$ and there is a sequence of $M$ nearest neighbors that belong to the major class. We note that MMCH does not require the $TStep$ parameter.

## 4 Performance Evaluation

In this section, we present the experimental results on two real-life data sets. All experiments have been conducted on an AMD Athlon 3000+ machine (2000 MHz), with 512 MB of main memory, running Windows XP Pro. The R*-tree, the incremental $k$NN algorithm, and the classification heuristics have been implemented in C++.

### 4.1 Data Sets

The first data set is the Pages Blocks Classification (PBC) data set and contains 5,473 items. Each item is described by 10 attributes and one class attribute. We have used the first five attributes of the data set and the class attribute. We reduced the number of attributes considering that the most dimensions we use, the worst performance the family of R-trees has. Each item of the data set belongs to one of the five classes. Furthermore, we have divided the data set into two subsets. The first subset contains 4,322 items used for training and the second contains the rest 1,150 items used for testing purposes.

The traditional $k$NN classification method achieves the best possible precision when $k = 9$. However, this value was very low and so the proposed heuristics can not reveal their full potential. Therefore, we have added noise in the data set in order to make the use of a higher $k$ value necessary. Particularly, for each item of the train set, we modified the value of the class attribute with probability 0.7 (the most noise is added, the highest value of $k$ is needed to achieve the best classification precision). This fact forced the algorithm to use a higher $k$ value. This way, we constructed a data set where the best $k$ value is 48. This means that the highest precision value is achieved when 48 nearest neighbors contribute to the voting process. This is illustrated in Figure 2(a), which depicts the precision value accomplished by modifying $k$ between 1 and 48.

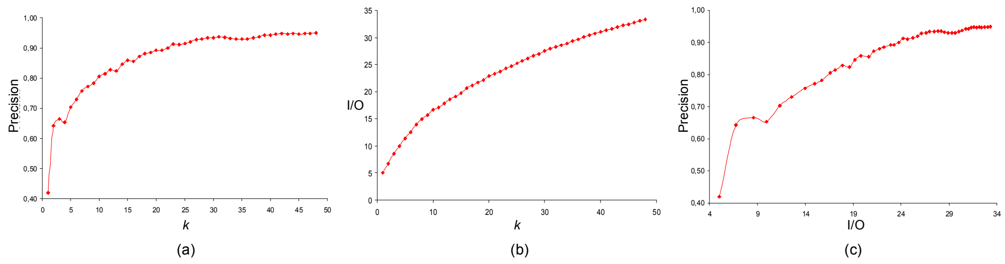


**Fig. 2.** Precision vs $k$, I/O vs $k$ and precision vs I/O for PBC data set.

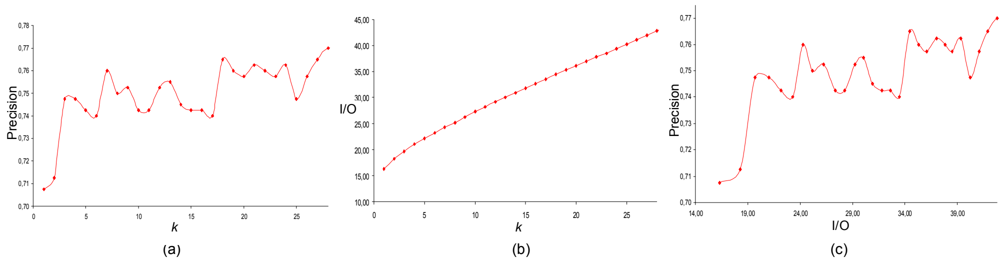


**Fig. 3.** Precision vs $k$, I/O vs $k$ and precision vs I/O for LRI data set.

As expected, the higher the value of $k$, the higher the number of I/O operations. This phenomenon is illustrated by Figure 2(b). We note that if we had used a lower value for $k$ ($k$ ¡ 48), we would have avoided a significant number of I/Os and therefore the search procedure would be less time-consuming. For example, if we set $k = 30$, then we avoid 5.84 I/Os for the classification of one item of the test set without significant impact on precision. Figure 2(c) combines the two previous graphs. Particularly, this figure shows how many I/Os the algorithm requires in order to accomplish a specific precision value.

The second data set is the Letter Image Recognition Data set [8], which contains 20,000 items. We have used 15,000 items for training and 5,000 items for testing. Each item is described by 17 attributes (one of them is the class attribute) and represents an image of a capital letter of the English alphabet. Therefore, the data set has 26 classes (one for each letter). The data set objective is to identify the capital letter represented by the items of the test set using a classification method.

As in the case of the PBC data set, we have reduced the number of dimensions (attributes). In this case, dimensionality reduction has been performed by using Principal Component Analysis (PCA) on the 67% of the original data. The final number of dimensions have been set to 5 (plus the class attribute).

Figure 3 illustrates the data set behavior. Specifically, Figure 3(a) shows the precision achieved for $k$ ranging between 1 and 28. We notice that the best precision (77%) is achieved when $k = 28$. Figure 3(b) presents the impact of $k$ on the number of I/Os. Almost 43 I/O operations are required to classify an item of the test set for $k = 28$. Finally, Figure 3(c) combines the two previous graphs illustrating the relation between precision and the number of I/O operations. By observing these figures, we conclude that if we had used a lower $k$ value, then we could have achieved better execution time by keeping precision at high levels.

## 4.2 Determining Parameters Values

Each heuristic uses a number of parameters. These parameters must be adjusted so that the best performance is achieved (or the best balance between precision and execution time). In this section, we present a series of experiments which demonstrates the behavior of the heuristics for different values of the parameters. We keep the best values (e.g. the parameters that manage to balance execution time and precision) for each heuristic and use these values in a subsequent section where heuristics are compared.

**Pages Blocks Classification Data Set** Initially, we are going to analyze the $MinNN$ parameter. It is a parameter that all heuristics use. Recall that $MinNN$ is the minimum number of NNs that should be used for classification. After determining these NNs, the heuristics are activated. Figure 4 show how the heuristics performance (precision and I/O) is affected by modifying the value of $MinNN$. The values of the other parameters have as follows: $TStep = 4$, $IndFactor = 1$, $PMaj = 0.6$, $MTimes = 4$, $k = 48$.

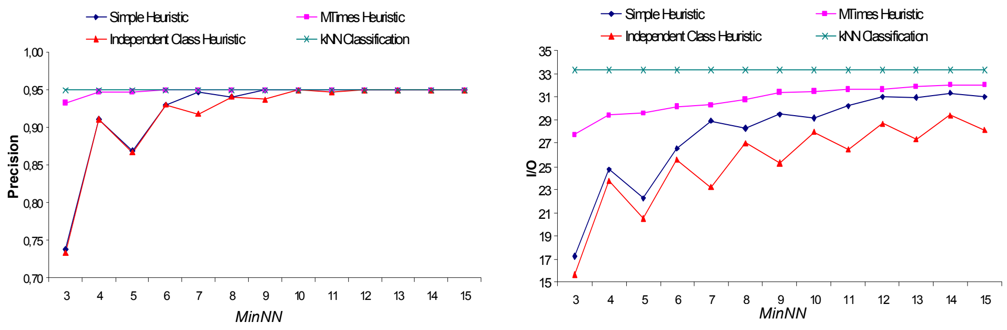


**Fig. 4.** Impact of $MinNN$ for PBC data set.

By observing the results it is evident that precision is least affected when the MMCH heuristic is used. Therefore, for this heuristic, we set $MinNN = 4$, which is the value that provides the best balance between I/O and precision. In contrast, the precision of the other two heuristics is significantly affected by the increase of $MinNN$. We decide to define MinNN = 11 for the Independency Class Heuristic and $MinNN = 7$ for the Simple Heuristic. Our decision is justified by the precision and I/O measurements provided for these parameters values.

We continue our experiments by finding the best value for $IndFactor$. Recall that this parameter is used only in ICH heuristic. We modify $IndepFactor$ from 0.4 to 4 and calculate the precision achieved. Figure 5 illustrates that the best balance between precision and I/O is achieved when $IndFactor = 1$. The values of the other parameters are as follows: $MinNN = 11$, $TStep = 4$, $k = 48$.

Next we study the impact of the parameter $MTimes$, which is used only by the MMCH heuristic. As it is depicted in Figure 6 for $MTimes = 3$, the precision level will be high enough and the number of I/O operations is relatively low. So we keep this value as the best possible for this parameter. The values of the other parameters have as follows: $MinNN = 4$, $PMaj = 0.6$, $k = 48$.

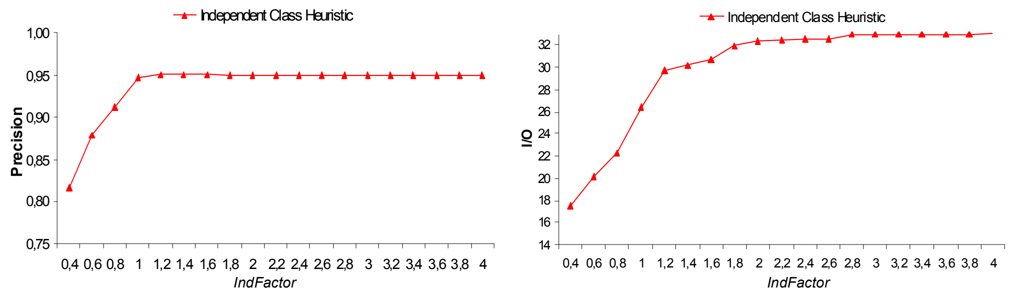


**Fig. 5.** Impact of $IndFactor$ on the performance of ICH heuristic for PBC data set.

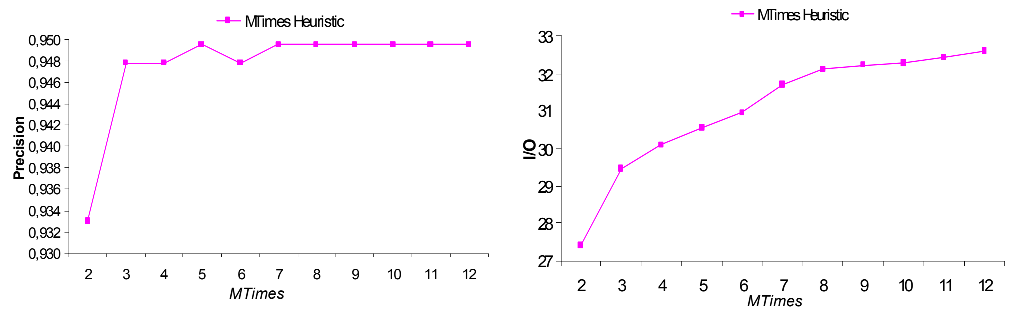


**Fig. 6.** Impact of $MTimes$ on the performance of MMCH heuristic for PBC data set.

Next, we study the impact of $TStep$ parameter on the performance of SC and ICH, since MMCH does not use this parameter. Figure 7 depicts the results. The values of the rest of the parameters have as follows: $MinNN = 6$, $IndFactor = 1$, $PMaj = 0.6$, $MTimes = 3$, $k = 48$. Both SH and ICH heuristics achieve the best precision when $TStep = 4$. In fact, SH achieves the same precision for $TStep = 3$ or $TStep = 4$, but less I/Os are required when $TStep = 4$. Since MMCH and $k$NN classification are not affected by $TStep$, their graphs are parallel to the $TStep$ axis. Finally, it is worth to note that although MMCH needs significantly less I/Os than $k$NN classification, the precision that MMCH achieves is the same as that of $k$NN classification.

**Letter Image Recognition Data Set** Next, we repeat the same experiments using the LIR data set. The impact of $MinNN$ is given in Figure 8. It is evident that all heuristics achieve higher precision than $k$NN classification. By studying Figure 8 we determine that the best $MinNN$ values for the three heuristics are: $MinNN = 7$ for SH, $MinNN = 12$ for ICH and $MinNN = 4$ for MMCH. These values achieve the best balance between precision and I/O processes. The values of the other parameters have as follows: $TStep = 2$, $IndFactor = 1$, $PMaj = 0.6$, $MTimes = 3$, $k = 28$.

Figure 9 depicts the results for the impact of $IndFactor$. We see that $IndFactor = 1$ is the best value since the ICH heuristic achieve the best possible precision value and at the same time saves almost ten I/O operations per query. The values of the other parameters are as follows: $MinNN = 12$, $TStep = 5$, $k = 28$.

Next, we consider parameter $MTimes$, which is used only by the MMCH heuristic. We define $MTimes = 3$, which results in 13 I/O savings for each query and achieves

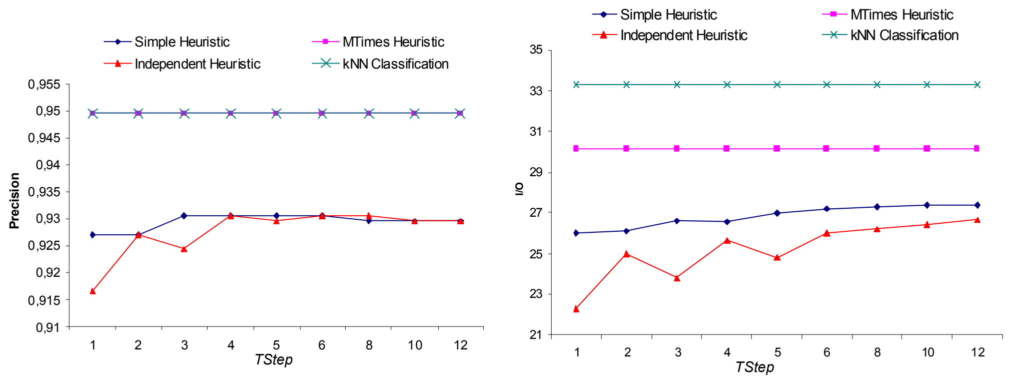


**Fig. 7.** Impact of $TStep$ parameter for PBC data set.

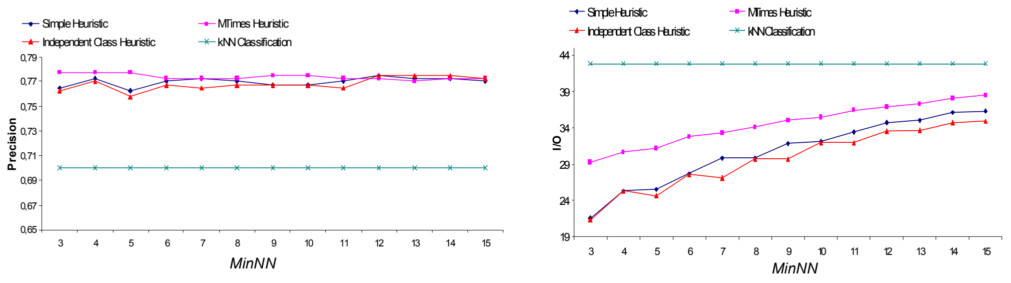


**Fig. 8.** Impact of $MinNN$ for LIR data set.

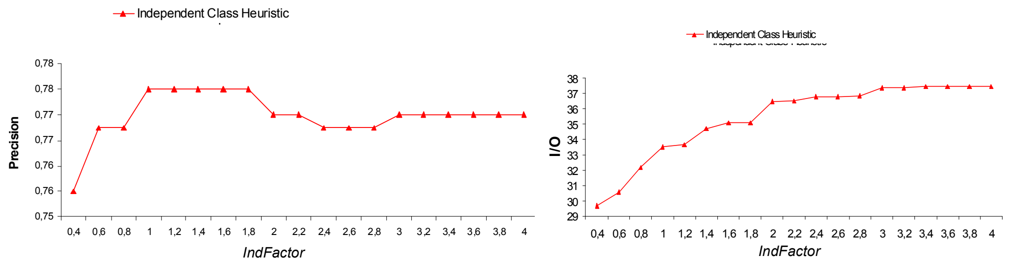


**Fig. 9.** Impact of $IndFactor$ on the performance of ICH heuristic for LIR data set.

the best possible precision value. The results are illustrated in Figure 10. The values of the other parameters have as follows: $MinNN = 4$, $PMaj = 0.6$, $k = 28$.

Finally, in Figure 11 we give the impact of $TStep$. We set $TStep = 4$ for SH and $TStep = 5$ for ICH, since these values give adequate precision and execution time. The values of the rest of the parameters have as follows: $MinNN = 4$, $IndFactor = 1$, $PMaj = 0.6$, $MTimes = 3$, $k = 28$.

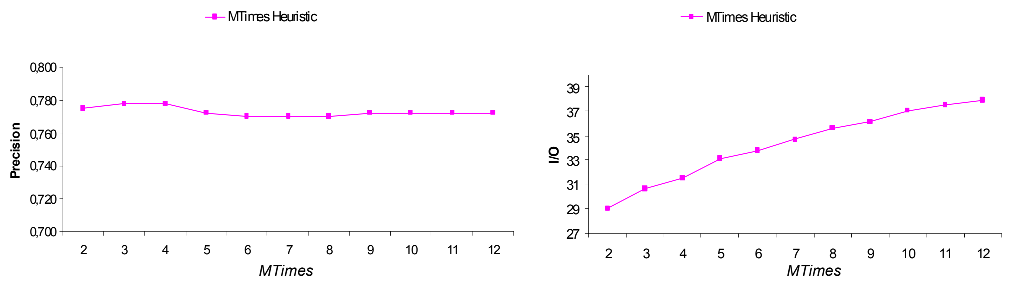


**Fig. 10.** Impact of $MTimes$ on the performance of MMCH heuristic for LIR data set.

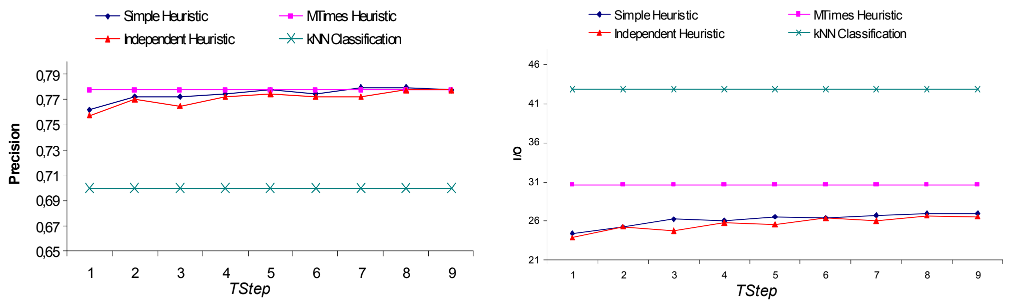


**Fig. 11.** Impact of $TStep$ parameter for LIR data set.

## 4.3 Comparison of Heuristics

In this section we study how the heuristics compare to each other and to the traditional $k$NN classification, by setting the parameters to the best values for each heuristic, as they have been determined in the previous section.

**Pages Blocks Classification Data Set** Figure 12 depicts the performance results vs $PMaj$. When $PMaj = 0.6$, precision is about the same for all heuristics and very close to that achieved by traditional $k$NN classification. However, our heuristics require significant less I/O for achieving this precision. When this precision value is accomplished there is no reason to find more nearest neighbors and therefore valuable computational savings are achieved.

According to our results, ICH achieves a precision value of 0.947 ($k$NN's precision is 0.9495) while it saves about 6.88 I/Os for the classification of one item. Particularly, when we use ICH, we find 31.29 nearest neighbors on average instead of 48. Similarly, when $PMaj = 0.6$, SH achieves precision equal to 0.948 (very close to that of $k$NN) by performing an early-break when 38.5 NNs on average have been found (almost 10 less than $k$NN requires). Therefore, SH saves about 4.385 I/Os per each item of the test set. Finally, the same precision value (0.948) is achieved by MMCH. However, this heuristic saves less number of I/Os than SH. Specifically, MMCH spends 3.875 I/O less than $k$NN because it finds 39.767 NNs on average.

Figures 13 and 14 summarize the results of this experiment using bar charts which have been produced by setting $PMaj = 0.6$ and maintain the same values for the other parameters. Figure 13 shows that the precision achieved by the heuristics is very close to that of $k$NN, whereas the number of required I/Os is significant less than that of

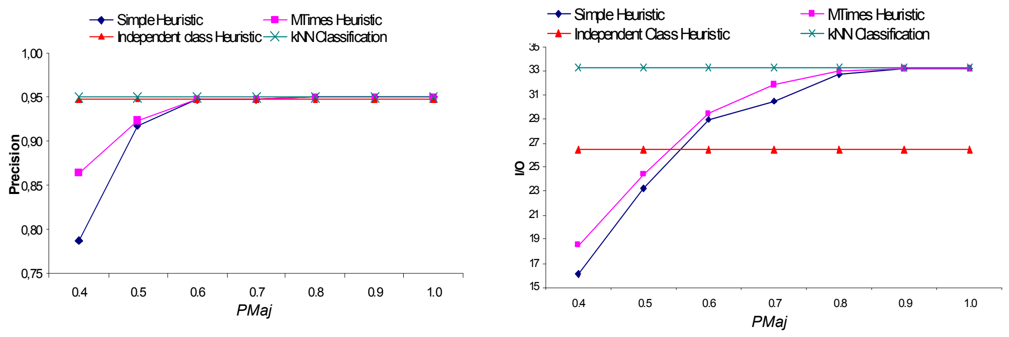


**Fig. 12.** Precision and number of I/Os.

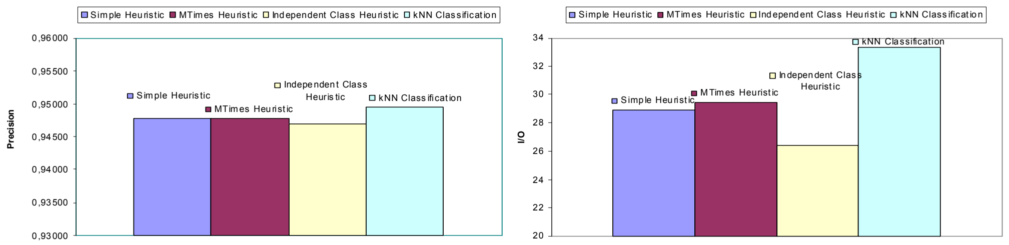


**Fig. 13.** Precision and number of I/Os for $PMaj = 0.6$

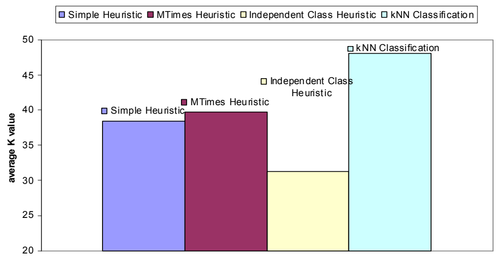


**Fig. 14.** Required number of nearest neighbors for $PMaj = 0.6$.

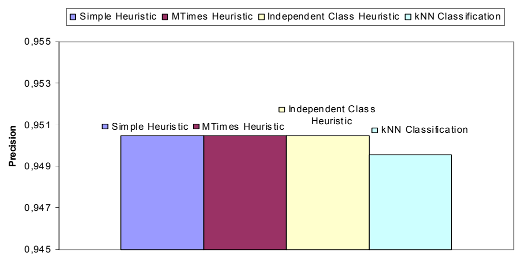
$k$NN. Figure 14 presents the number of nearest neighbors retrieved on average by each method.

We can not directly answer the question which heuristic is the best. The answer depends on which measure is more critical (precision or execution time). If a compromise must be made, Figure 12 will help. We notice that ICH shows the best performance because it achieves a precision that is very close to the precision of $k$NN posing the minimum number of I/Os. To declare a winner between SH and MMCH we notice that when $PMaj > 0.6$ the heuristics achieves almost the same precision, but MMCH poses more I/Os than SH.

As we have already mentioned, we try to find the parameters values that provide a compromise between precision and execution time. However, if one of these measures is more important than the other, the parameters can be adjusted to reflect this preference. Suppose that execution time is more critical than precision (when for example a quick-and-dirty scheme should be applied due to application requirements). If we se $PMaj = 0.5$, then 23.245 I/Os per query will be required (instead of 33.32 required by $k$NN) by finding 25.72 NNs instead of 48. However, this means that precision will be significantly smaller than that of $k$NN (we saw that when $PMaj = 0.6$, the difference between the precision of SH and $k$NN is minor). Similar results are obtained for MMCH when $PMaj = 0.5$. On the other hand, if precision is more critical than time, we can adjust the heuristics parameters towards taking this criticality into account. In this case, it is possible that the heuristics achieve better precision than $k$NN, with execution time overhead. In any case, early-break heuristics always require less execution time than $k$NN.

By considering the impact of $MinNN$ shown in Figure 4, it is evident that MMCH can achieve a slightly better precision than $k$NN. Specifically, if we set $MinNN = 8$, $PMaj = 0.6$ and $MTimes = 3$, then MMCH achieves a precision value equal to 0.950435, whereas 30.793 I/Os are required (while $k$NN requires 33.3174 I/Os per query and achieves a precision of 0.94956). The early-break heuristic is able to avoid 2.5244 I/O per query, whereas at the same time achieves better precision when $k$ is adjusted to ensure the best precision value. Although the number of saved I/Os may seems small, note that this savings are performed per classified item. Since the PBC test set contains 1,150 items, we realize that the overall number of saved I/Os is 2903, which is significant.

Similar considerations apply to the other two heuristics. For example, ICH can outperform the precision of $k$NN when $IndyFactor = 1.2$, $TStep = 4$, and $MinNN = 11$ (see Figure 5). However, because of the increase of $IndFactor$ from 1 to 1.2, the I/O requirements are increased from 26.44 to 29.75. Finally, if we set $MinNN = 11$



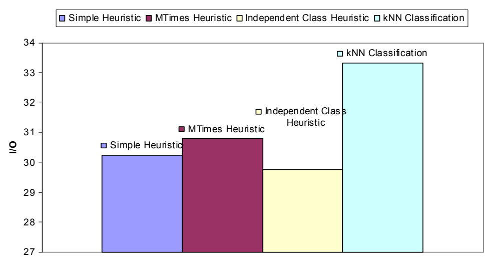


**Fig. 15.** Precision and number of I/Os ($MinNN = 11$).

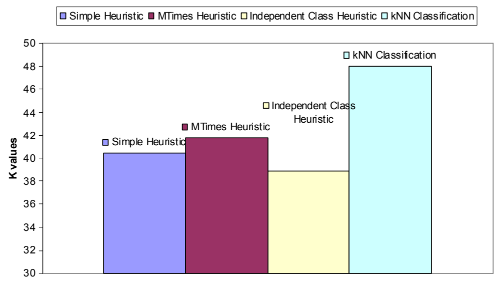


**Fig. 16.** Required number of nearest neighbors ($MinNN = 11$).

instead of 7, SH also outperforms the precision of $k$NN (see Figure 4). These results are illustrated in Figures 15 and 16.

**Letter Image Recognition Data Set** We close this section by presenting the comparison results using the LIR data set. Similar conclusions can be drawn as in the case of the PBC data set. The results are given in Figures 17, 18 and 19.

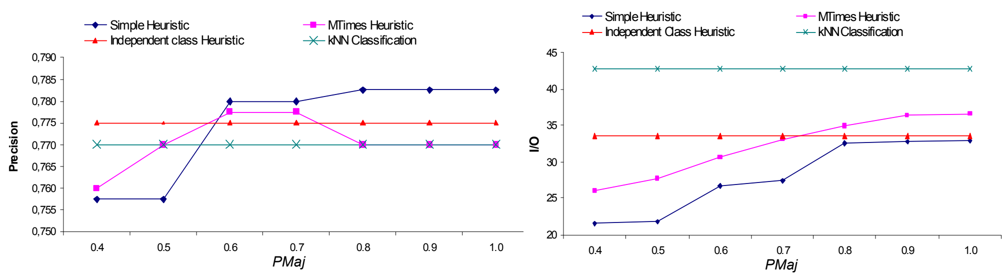


**Fig. 17.** Precision and number of I/Os vs $PMaj$.

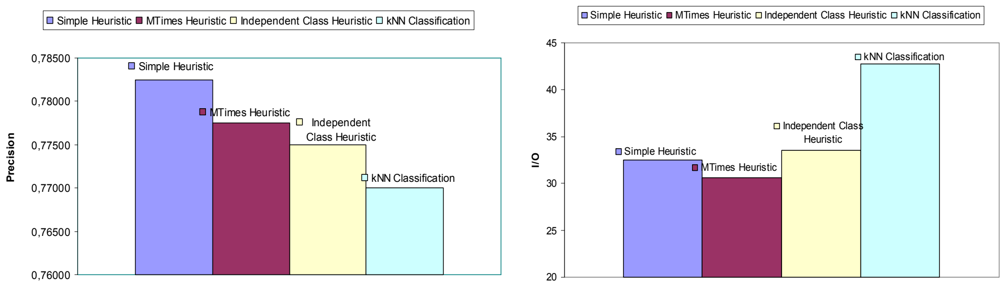


**Fig. 18.** Precision and number of I/Os for $PMaj = 0.8$.

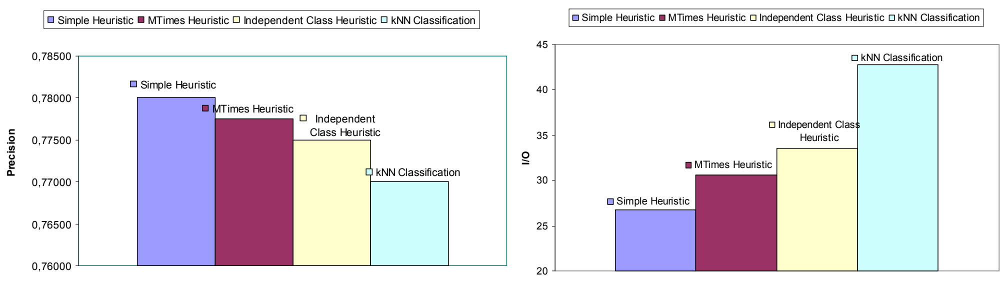


**Fig. 19.** Precision and number of I/Os for $PMaj = 0.6$.

We note that when we $PMaj > 0.5$ (see Figure 17), the three heuristics accomplish better precision than $k$NN and manage to reduce execution time. More specifically, SH achieves a precision of 0.7825 with 32.53 I/Os on average for $PMaj = 0.8$, whereas $k$NN a precision of 0.77 spending 42.81 I/Os (see Figure 18). Also, if we set $PMaj = 0.6$, SH achieves a precision of 0.78 spending 26.7775 I/Os on average (see Figure 19).

ICH, which is not affected by the $PMaj$ parameter, achieves a precision of 0.775 spending 33.52745 I/Os on average. Finally, MMCH achieves the best balance between precision and execution time we set $PMaj = 0.6$. Particularly, the heuristic accomplishes a precision of 0.7775 and spends 30.6525 I/Os on average (see Figure 18). Comparing the three heuristics using the Letter Image Recognition data set, we conclude that the simple heuristic has the best performance since it achieves the best possible precision spending the least possible number of I/Os.

## 5 Conclusions

In this paper, an adaptive $k$NN classification algorithm has been proposed, which does not require a fixed value for the required number of nearest neighbors. This is achieved by incorporating an early-break heuristic into the incremental $k$-nearest neighbor algorithm. Three early-break heuristics have been proposed and studied, which use different conditions to enforce an early-break. Performance evaluation results based on two real-life data sets have shown that significant performance improvement may be achieved, whereas at the same time precision is not reduced significantly (in some cases precision is even better than that of $k$NN classification). We plan to extend our work towards: (i) incorporating more early-break heuristics and (ii) studying incremental $k$NN classification by using subsets of dimensions instead of the whole dimensionality.

## References

1. D. W. Aha. Editorial. *Artificial Intelligence Review (Special Issue on Lazy Learning)*, 11(1-5):1–6, 1997.
2. C. Atkeson, A. Moore, and S. Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11(1-5):11–73, 1997.
3. C. Atkeson and S. Schaal. Memory-based neural networks for robot learning. *Neurocomputing*, 9:243–269, 1995.
4. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference*, pages 590–601, 1990.
5. C. Boehm and F. Krebs. The k-nearest neighbour join: Turbo charging the kdd process. *Knowledge and Information Systems*, 6(6):728–749, 2004.
6. K.L. Cheung and A. Fu. Enhanced nearest neighbour search on the r-tree. *ACM SIGMOD Record*, 27(3):16–21, 1998.
7. B.V. Dasarathy. *Nearest Neighbor Norms: NN Pattern Classification Techniques*. IEEE Computer Society Press, 1991.
8. P.W. Frey and D.J. Slate. Letter recognition using holland-style adaptive classifiers. *Machine Learning*, 6(2):161–182, 1991.
9. A. Guttman. R-trees: A dynamic index structure for special searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, 1984.
10. J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.
11. G.R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, 1999.
12. M. James. *Classification Algorithms*. John Wiley & Sons, 1985.
13. N. Rousopoulos, S. Kelley, and F. Vincent. Nearest neigbor queries. In *Proceedings of the ACM SIGMOD Conference*, pages 71–79, 1995.