FAST APPROXIMATE NEAREST NEIGHBORS

by
Matthew R. Casey

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Computer Science and Engineering

Lehigh University
May 2006

This thesis is accepted in partial
fulfillment of the requirements for the degree of
Master of Science.

(Date)

Henry S. Baird (Thesis Advisor)

Hank F. Korth (Chair, CSE Dept)

Preface

An investigation into algorithms for fast approximate nearest-neighbors (kNN) classification is reported. The principal results are empirical performance evaluation and analysis of a family of non-adaptive k-d tree data structures sped up by means of hash techniques. This research is motivated by a compelling need in pattern recognition theory and practice for classification methods trainable in reasonable time on large data sets (say, many millions of samples) while allowing extremely fast classification (ideally, at I/O rates). Experiments were carried out on data arising in digital libraries: images of pages of books containing several types of content (machine-print text, handwriting, photographs, etc). The particular problem was to classify every pixel in these images into one of the content types. Our choosing to classify pixels (rather than regions) gave us ready access to ground-truthed data sets of dauntingly large size. The non-adaptive k-d tree we used only approximates true kNN but it allows, in principle, high speed search. The growth of the k-d tree data structure on these data was not exponential in the size of the training set, as can happen in the worst case, but was a low-order polynomial (approximately cubic) in practically important operating regimes. Inevitably, tradeoffs of accuracy for speed are drastic at the extremes: the coarsest data structures run at brute-force (exhaustive search speeds), whereas finer-grained data structures allow speed-ups of several orders of magnitude. Happily, large speed-ups (of factors of 100 or more) do not always sacrifice much accuracy. The investigation includes testing these methods on a variety of data sets, as well as several variations of the classification technique.

Acknowledgments

I would like to thank my advisor, Henry Baird, for his generous support, guidance, and encouragement throughout development of this project and my graduate career in general. I would also like to thank the other members of the DICE project from Summer 2005: Michael Moll, Jean Nonnemaker, and Don Delorenzo. Without their work in building an infrastructure for classification, this project would not have been feasible.

Finally, I'd like to thank my family for their support throughout my academic career and anywhere it takes me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer vision (CV) systems, and document image analysis (DIA) systems in particular, are notoriously either overspecialized and fragile, or robust but ruinously expensive (*e.g.* national postal code readers)[Pav00]. The goal of highly versatile document analysis systems, capable of performing useful functions across the great majority of document images, remains elusive even after decades of effort [NS96][Nag00]. Certainly users—*e.g.* in the digital libraries, web search, and intelligence communities—avidly desire such a technology.

For this reason we have chosen a *versatility first* research strategy: that is, we strive to invent document image analysis capabilities that will, as its highest priority, work reliably across the broadest possible range of cases. This diversity embraces documents and images that: are written in many languages, typefaces, typesizes, and writing styles; possess a wide range of printing and imaging qualities; obey a myriad of geometric and logical layout conventions; are carefully prepared as well as hastily sketched; may be acquired by geometrically accurate flat-bed scanners, or snapped with hand-held cameras under accidental lighting conditions; are expressed as full color, grey-level, and black-and-white; are of any size or resolution (digitizing spatial sampling rate); and are presented in many image file formats (TIFF, JPEG, PNG, etc), both lossless and lossy.

We focus attention on basic DIA capabilities that are, arguably, the most broadly useful in that they need to be applied early in the document-processing pipeline. These are the *document image content extraction* tools, able to locate and characterize regions containing handwriting, machine-print text, graphics, line-art, logos, photographs, noise, and other "content types." By "locate" we mean describe the regions in the image where each content type dominates. By "characterize' we mean report estimates of a few basic properties of the regions: *e.g.* for machine-print, report skew angle, number of lines of text, number of words, and perhaps type size and type family. Although high accuracy is always appreciated, for tools as broadly applicable as these will be, a lower threshold of competency may often be sufficient.

We intend to classify documents at the pixel level, consistent with the versatility goals of the project. This means that training and testing samples can rapidly reach millions with only a few documents. In our case, this is a benefit, since it minimizes the number of documents needed to simulate a large classification system. Although we won't constrain our systems to a small number of classes, our initial experiments will classify each pixel as being handwritten text, machine print, photography or "unknown".

1.1 Our Classification Problem

A little formality at the outset will help to make our technical choices clear later on. We define our problem as follows, in terms that are widely understood within the pattern recognition and machine learning R&D communities. For a given set of testing document images, we wish to classify each pixel; thus our basic data (both for training and testing) consists of sets of pixel *samples*, each represented by a fixed number of numeric *features* computed by image-processing methods operating on a small region containing the pixel. The features, and their number—we call it d, the dimensionality of the feature space—was determined by an engineering exercise, outside the scope of this thesis, reported in [BMN$^+$06].

Classification problem:

Given **T**, and a previously unseen sample **x**,
find the most probable class c for **x**.

Adopting a Bayesian approach, we choose $c_{max} = argmax_{c_j \in \mathbf{C}}\{P(c_j|\mathbf{x}, \mathbf{T})\}$ where $P(c_j|\mathbf{x}, \mathbf{T})$ denotes the posterior probability of class c_j given the new observation **x** and the prior knowledge expressed in the training data **T**. The *k-nearest neighbors*(kNN) algorithm is:

```
K Nearest Neighbors algorithm
INPUT: $\mathbf{T}$, $k \in \mathbf{N}^+$, and $\mathbf{x}$
OUTPUT: $c \in \mathbf{C}$
Step 1. within $\mathbf{T}$, find $k$ nearest neighbors of $\mathbf{x}$
        that is, find a set $Y \subset \mathbf{T}$, $|Y| = k$, s.t.
            $\forall \mathbf{y} \in Y \ \forall \mathbf{z} \in T - Y \ \ \|\mathbf{x} - \mathbf{y}\| \leq \|\mathbf{x} - \mathbf{z}\|$
Step 2. within $Y$, use the ground-truth classes assigned to samples in $\mathbf{T}$ find
the most frequently occurring class, breaking ties at random.
```

For any particular set of features, assuming that the training data is representative, the kNN algorithm famously enjoys a valuable theoretical property: as the size of a representative training set increases, the error rate of kNN approaches to no worse than twice the Bayes error[DHS01], which is the minimum achievable by any classifier given the same information. This remarkable limiting performance is due, fundamentally, to its not committing to any approximate parametric model of the per-class distributions. This algorithm also generalizes directly to more than two classes (unlike several competing methods such as neural nets and classification trees). Finally, it has has often been competitive in accuracy (if rarely in speed) with more recently developed classifiers including support-vector machines. For these reasons, we consider kNN would be a nearly ideal classifier for our problem, were it were not for the high cost in both time and space of naive implementations and the difficulty of crafting algorithms that closely approximate its performance in high dimensional features spaces.

We have implemented 5NN (kNN for $k = 5$) and, in small scale experiments, have seen it achieve pixel classification rates on the content extraction problem greater than 98% correct. Interestingly, even when the per-pixel classification error rate is as high as 25%, the principal regions and their dominant content types are evident to the eye and (we expect) could be extracted robustly by simple image post-processing[BMN$^+$06].

1.2 Approximations to kNN

There are many ways to approximate the performance of kNN, including reformulation as the

Radius Search problem

Given $\mathbf{T}$, $r \in \mathbf{R}$, and $\mathbf{x}$,
find all points of $\mathbf{T}$ within radius r of $\mathbf{x}$: that is, the set $Y_r \equiv \{ \mathbf{y} \in \mathbf{T} \ s.t. \ \|\mathbf{x} - \mathbf{y}\| \leq r \}$

If $r \ll s$, we can expect that finding Y_r will be easier than finding Y_s. (Of course, generally $|Y_r| < |Y_s|$ also, in which case reporting the result of the search will take *longer*. But we choose to neglect runtimes that depend on the size of the output, for reasons that will become clear later.) If $|Y_r| = k$, then radius search is equivalent to kNN. If $|Y_r| \approx k$, then we will say that radius search *approximates* kNN. (Of course, this notion of approximation does not address the key issue, which is how much less accurate radius search classification is than kNN classification.)

It might be even cheaper to compute approximations to radius search.

Nested Radius Search problem

Given $\mathbf{T}$, $\{r_i\}_{i=1,\ldots,s} \in \mathbf{R}$, $r_1 < r_2 < \ldots < r_s$, and $\mathbf{x}$,
find sets $\subset \mathbf{T}$ within radii r_i of $\mathbf{x}$: that is,

$$Y_{r_i} \equiv \{ \mathbf{y} \in \mathbf{T} \ s.t. \ \|\mathbf{x} - \mathbf{y}\| \leq r_i \}$$

Note that $Y_{r_1} \subseteq Y_{r_2} \subseteq \ldots \subseteq Y_{r_s}$. If $r_k < r_l$, then it may be possible, through judicious choice of data structures and algorithms, to compute Y_{r_l} faster than Y_{r_k}.

Approximate Radius Search problem

Given $\mathbf{T}$, $r \in \mathbf{R}$, $\epsilon \in \mathbf{R}$ and $\mathbf{x}$,
find sets Y^L and $Y^U \subset \mathbf{T}$ such that

$$Y_{r-\epsilon} \subseteq Y^L \subseteq Y_r \subseteq Y^U \subseteq Y_{r+\epsilon}$$

That is, Y^L and Y^U approximate Y_r within ϵ. As $\epsilon \to 0$, $Y^L \to Y_r$ (from below) and $Y^U \to Y_r$ (from above), and so Approximate Radius Search converges to Radius Search.

When ϵ is large, it should be easier to solve Approximate Radius Search than to solve Radius Search.

How can Approximate Radius Search be used for classification? Well, if it happens that $|Y^L| = |Y^U|$, then $Y^L = Y^U = Y_r$ and, exactly as in kNN, we can choose the most frequently occurring class in that set. Lets call the frequency of occurrence of each class c_j in Y^L "f_j^L" and similarly, in Y^U, "f_j^U". Let the most frequently occurring classes in these sets be $maxc^L$ and $maxc^U$; if these happen to be the same class, we will of course choose that class. But if they differ, then we have several policy choices:

- we can compute average frequencies for each c_j, *e.g.* $f_j^{aver} \equiv (f_j^L + f_j^U)/2$, and then choose the class with the highest average frequency;
- we can use the value k from kNN to assist in the decision, for example, if $|Y^L|$ is closer to k than Y^U, we choose the most frequently occurring class in Y^L (and vice versa).

Thus Approximate Radius Search can be used as a basis for classification and offers a range of engineering tradeoffs between accuracy and speed.

1.3 Approximate Radius Search under the Infinity Norm

With the Infinity Norm as our metric, then Radius Search becomes a multidimensional range search (or "region query") in which the search regions are hypercubes of radius r centered on the query sample $\mathbf{x}$. Multidimensional range searching is an intensively explored topic[Sam90][Ben75a] by researchers in the computational geometry, machine learning, pattern recognition, and graphics research communities. However, special requirements of pattern classification may lead us in fresh directions. For example, in the literature, range searching is commonly discussed as a generalization of *point searching*, which in turn is characterized as a generalization of dictionaries to multidimensional data. Dictionaries are classically designed to support three basic operations: *insert* a single data item, *find* an item (returning its associated metadata, or returning 'not found'), and *delete* an item. Data structures and algorithms for dictionaries are thus *dynamic*: designed to process an arbitrary sequence of these three operations efficiently

By contrast, in classification, insertions are performed all at once in an offline batch operation (the “training stage”) in which all the data (training samples) are simultaneously available. Deletion never occurs (although we may choose to prune (“edit) them or summarize sets of data points statistically rather than storing all of them explicitly). Thus the data structure is *static*. Finding is the only operation which is online and must run fast; and it may also be possible to batch a large set of find operations and so exploit the similarity of sequences of queries for points that are isogenous. Thus much of the literature on dynamic multidimensional search is not directly relevant to classification, and may benefit from a fresh perspective.

Let us review techniques for multidimensional search that may assist us in choosing, adapting, or crafting new data structures and algorithms suitable for classification.

1.4 Adaptive *k*-d Trees

One multidimensional search technique with broad applicability is Bentley’s *k*-d trees[Ben75b]. The variant of k-d trees most relevant to classification seems to be the *adaptive k*-d tree[FBF77]. Briefly, these partition the set of points recursively in stages: at each stage one of the partitions is divided into two subpartitions; we will assume that it is possible to choose cuts that achieve *balance*, that is, to divide into subpartitions containing roughly the same number of points. Division is by cutting along one of the dimensions $i \in \{1, \ldots, d\}$, *i.e.* by choosing a threshold value and assigning each of the partition’s points **x** to subpartition (a) or (b) according to whether its x_i component value is (a) less than, or (b) greater than or equal to the threshold. (In some implementations, the threshold corresponds to a component value for one of the points, which is then stored in the interior node of the search tree; but will we assume here that all points are stored in leaf nodes). Generally, at each stage a different dimension is cut: one simple strategy is to cycle (and if necessary recycle) through the dimensions in a fixed order; another strategy is to cut the currently most populous partition. Cutting proceeds until all partitions contain few enough points to invite a final fast sequential search. The final partitions are generally hyperrectangles (not always hypercubes) with orthogonal sides (parallel to the coordinate axes of $\mathbf{R}^d$).

Balancing each cut ensures that find operations execute in $\Theta(\log n)$ time in the worst case. Consistent with a guarantee of such logarithmic-time finds, Bentley's k-d tree construction achieves an asymptotically minimum number of cuts and thus a minimum number of partitions (close to n/p, where p is the number of points in the final partitions; note that, in our context where $p = 10$ this is still huge, $\approx 10^8$). The threshold value chosen for each cut depends on the distribution of points within the partition to be cut, so the thresholds are not *independently predictable*: that is, none (after the first) can be computed without knowledge of the cut thresholds in some earlier stages. Further, given a previously unseen **x** it is not possible to compute the d upper and lower bounds of its k-d hyperrectangle (within which it lies) without traversing the k-d tree. Thus locating **x**'s k-d hyperrectangle *requires* $\Theta(\ \log n\ )$ time.

The pruning power of k-d trees speeds up range searches. Given a query point **x** and a radius r, defining a search hypercube, it is straightforward to generalize the find algorithm to explore all k-d tree nodes whose subtrees overlap the search hypercube. The asymptotic runtime of such variants has been studied: [LW77] reports that the worst-case number of tree nodes explored is $\Theta(\ d\ n^{1-1/d}\ )$. In our context, this may (or may not) promise much improvement over brute-force search, since (neglecting the multiplicative constant) $d\ n^{1-1/d} \approx 100(10^9)^{0.99} = 100(10^{8.91}) = 10^{10.91} \gg 10^8 \approx n$. Of course this may be a pessimistic bound for several reasons. Whether or not k-d tree range searches are efficient for classification may ultimately be decidable only by experiment.

1.5 Multidimensional Tries

An alternative data structure that assists multidimensional search is a recursive partitioning of the space using *fixed cuts*. Suppose that for each dimension $i \in \{1, \ldots, d\}$, lower and upper bounds L_i and U_i on the values of the components $\mathbf{x}_i$ are known. Then one may choose to place cuts at midpoints of these ranges, *i.e.* at $(L_i + U_i)/2$, and later, when cutting those partitions, recursively cut at the midpoint of the (now smaller) ranges. A search trie can be constructed in a manner exactly analogous to k-d trees with the exception that the distribution of the data is ignored in choosing cut thresholds. It will no longer be possible to guarantee balanced cuts and thus most of the time and space optimality properties of k-d trees are lost. However, there are gains: for example, the values of the cut thresholds can be predicted (they all are of course predetermined by $\{L_i, U_i\}_{i=1,\dots,d}$). As a consequence, if the total number of cuts r is known, the hyperrectangle within which any query (test) sample **x** lies can be computed in $\Omega(r)$ time, and in some realistic models of computation in $\Omega(1)$ time—in either case, extremely fast, faster than computing a single point-to-point distance using the metric. We will call these recursive midpoint-cut trie hyperrectangles *partitions* (they are often called 'bins' or 'cells' in the kNN literature).

Chapter 2

Bit-Interleaving Addressing in k-d Trees

2.1 Motivation

Since samples can be mapped to partitions in $\Omega(r)$, we hope to use this mapping in order to restrict the number of distance computations that need to be performed. The goal is to find a partition size that will not degrade accuracy severely, but will gain a significant decrease in search time.

2.2 Bit-Interleaved Addressing

Partitions resulting from tries as above can be addressed using *bit-interleaving*[Sam90]. Let $< d_k, m_k >_{k=1,\ldots,r}$ be a sequence of cuts, where, at cut k, d_k is the dimension chosen to be cut and m_k is the midpoint of the partition chosen for that cut. Among the partitions of feature space that result, a test sample x will fall in a partition that can be described by a sequence of decisions $b_k = (x_{d_k} > m_k)$ taking on the value False (‘0’) or True (‘1’) as x lies below or above the cut respectively. Equivalently, any bit-sequence $< b_k >_k = 1, \ldots, r$ of length r determines the boundaries of some partition, and so can be thought of as an address for it. This process is illustrated in Figure 2.1.

To summarize: given a test sample x and a sequence of r cuts, we can compute in $\Theta(r)$ time the bit-interleaved address of the partition within which x lies.

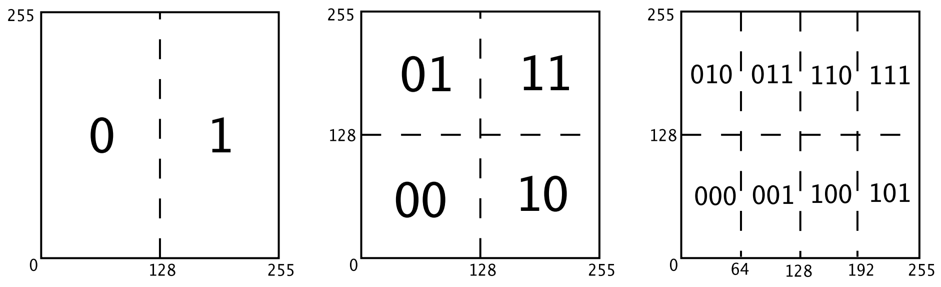

Figure 2.1: A two dimensional example of progressive bit-interleaving. The first dimension is split on its most significant bit, granting all points a bit-interleaved address of 0 or 1. Next, the same split is performed on the second dimension, appending another bit to the address. Then there's another split on the first, continuing until the desired number of splits (bit-interleaved address size) has been attained

We expect that training and test data have a skewed (nonuniform) distribution in feature space, and specifically when r is large that only a small fraction of the resulting small partitions will be occupied by *any* training data. If this assumption holds true in practice, only a few distinct bit-interleaved addresses will occur in even a very large training set, and so it may be feasible to use the bit-interleaved address as a key to a very sparse hash table. This would allow an $O(1)$ lookup once the address was known, and a 5NN search would then be performed using the points in the partition.

Chapter 3

Experiments

Experiments on our document images where $d = 15$, using $n = 10,000,000$ training samples, varying the number of bits $r < 75$ suggest that the number of occupied partitions, as a function of the length of their bit-interleaved address, is asymptotically roughly cubic. For $r = 50$ the absolute number observed was about 2,100,000, which is of course easily manageable by single-stage hashing where the hash table is contained within main memory.

Also, we have systematically tested the accuracy and speedups of hashed kD trees, where $d = 15$ (again), $n = 1,565,695$ training samples and $254,181$ test samples. When a brute-force 5-NN program is run, it achieves a per-pixel correct classification rate of 78% (and the results look very good to the eye); of course, it runs slowly, requiring about 400 billion distance calculations. By using bit-interleaved address of length 40 bits, hashing speeds up the calculation by a factor of 99.7 for a slight drop in per-pixel accuracy, to 68%. The range of tradeoffs between speed-up and accuracy is shown in Figure 3.1.

Accuracy remains high (above 60%) until somewhere between 40 bits (5 bytes) and 48 bits; thereafter it falls rapidly. The speed-up factor improves exponentially across the range.

In the above data, errors are of two types. In the first type, we hash into a cell which contains some training samples, but 5NN does not do as well as expected: this means of course that not all of the five nearest neighbors lie within this hash cell. Errors of

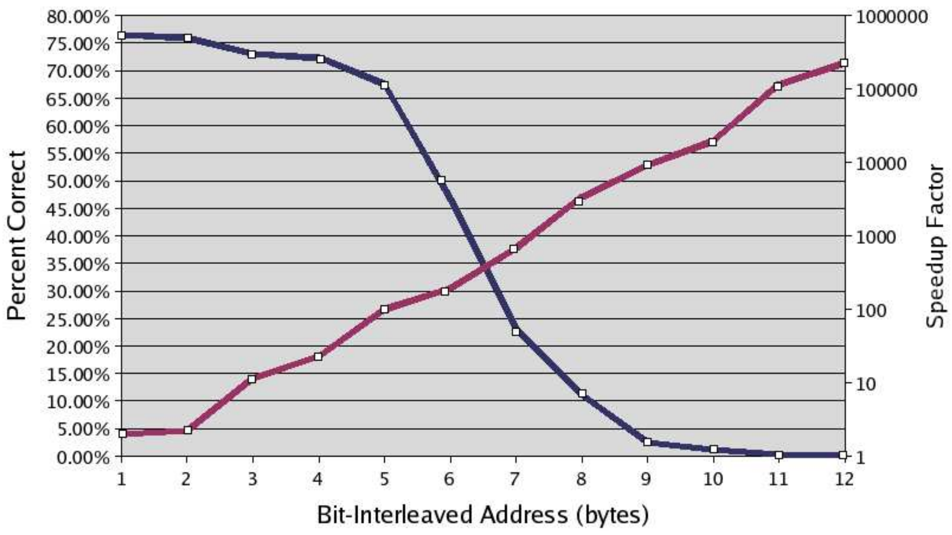

Figure 3.1: Tradeoffs between accuracy (percent of pixels correctly classified, labeled on the left scale) and speed-up (factor of decrease in the number of distances computed, compared to brute-force; labeled on the right scale, using a log-scale), as a function of the size of the bit-interleaved address (in bytes).

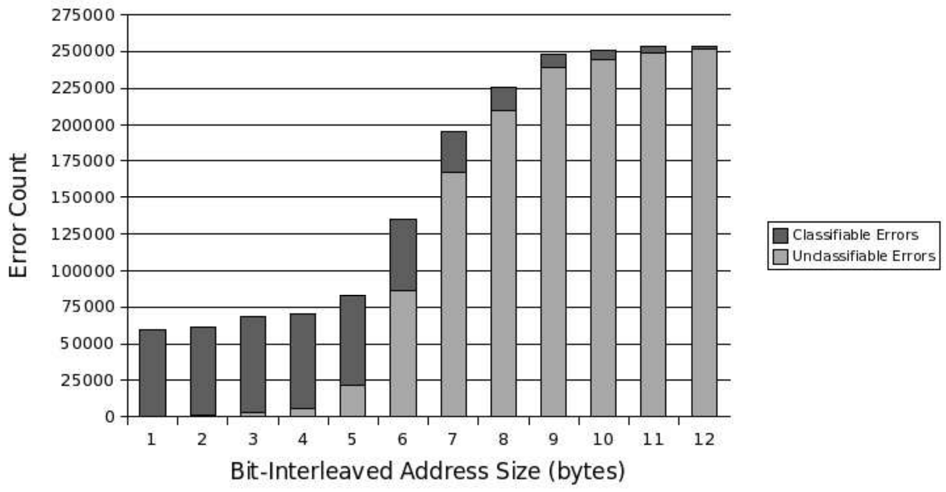

Figure 3.2: Contributions to the error rate caused by hashing into an empty bucket (Unclassifiable Errors) and finding the wrong class within a cell (Classifiable Errors) for varying sizes of bit-interleaved address in bytes.

the second type occur when the cell contains *no data*: that is, no training points hashed into that cell. Errors of the second type are in the minority as long as the bit-interleaved address is less than about 50 bits; but above that, they rise rapidly and dominate the error count (see Figure 3.2). Clearly these errors will be reduced by the addition of more training data. It is interesting to note that, if we do not count the second type of errors, then accuracy remains high (above 60%) until more than 64 bits are used, for a speed-up of 3401.

The rise and rapid dominance of errors of both types are to be expected when the volume of a hash cell falls below a certain threshold. We can roughly estimate this threshold as follows. We have observed that the distance from a probe point to the farthest of its five nearest neighbors is less than 15 for about 95% of the points. When we use at least 45 bits of bit-interleaved address, we have in effect guaranteed that each of the 15 dimensions have been cut three times, which gives hash cells that are $256/2^3 = 32$ on a side. This is at the boundary of the unpleasant case when the farthest of the five nearest neighbors is likely to fall outside the cell. This rough analysis seems to explain the rapid fall-off of accuracy above 48 bits. In order to better understand how this fall-off would behave with different types of data, we set out to perform the same experiment with even more data.

3.1 Expanded Training

By adding a few more documents, we increased the size of our training set by a factor of eight. As shown in Figure 3.3, we observed a consistent loss in accuracy along with an increase in speedup. The loss in accuracy can most likely be attributed to the new training documents having less in common with the testing document than the old training documents.

In order to better understand the increase in speedup, it is important to understand that the speedup factor is computed as the number of distance computations performed in “brute-force” kNN divided by the number of distance computations performed by the hashing classifier. Thus, Figure 3.3 shows that the number of computations performed by the hashing classified does not grow as quickly as that of kNN. We believe that the larger data set experienced a greater speedup because the data added was less relevant to the testing data than the original training set. Thus, most of the new data hashed into cells that never came in contact with testing data. This wouldn’t matter with traditional kNN, but with our hashing method, the classification can be performed much more quickly.

We also plotted the number of cells used for varying BIA sizes (Figure 3.4). Although we’d need to perform this analysis on a wider variety of data sets in order to make strong claims about the behavior, it is encouraging that the method has not yielded an exponential blowup in the number of cells used. Previous analysis had shown the expansion to be approximately cubic in the area between BIA sizes 2 and 6.

As we gathered more data, the decision was made to increase the size of the training data set while keeping the testing data set limited to one document. Thus, since each pixel is a data item, the training set grew very quickly. While this rapid growth would be a problem in a real-world system, we *wanted* a large data set, so this works to our advantage. On the other hand, the size of the data got to the point where the system was

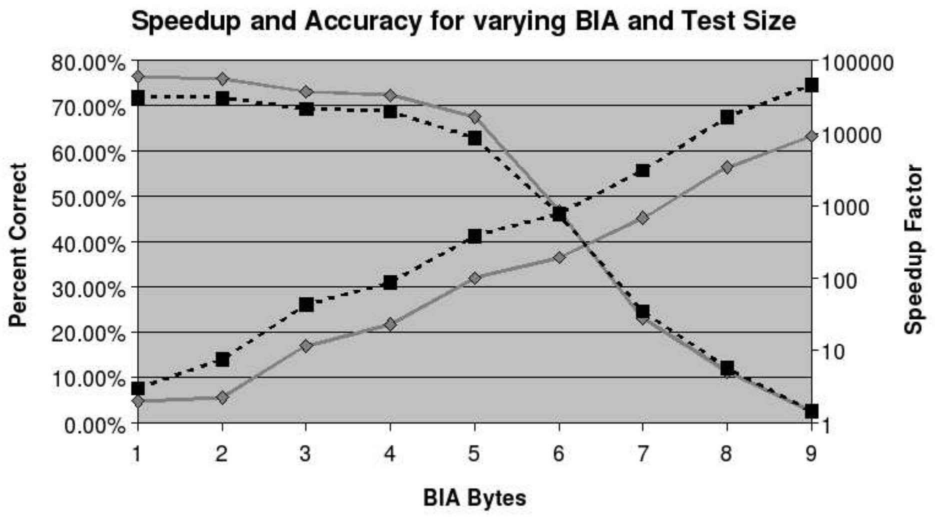

Figure 3.3: Speedup factor and percent correct for our initial, smaller, test (solid line with diamonds) and our second, larger, test (dashed line with squares).

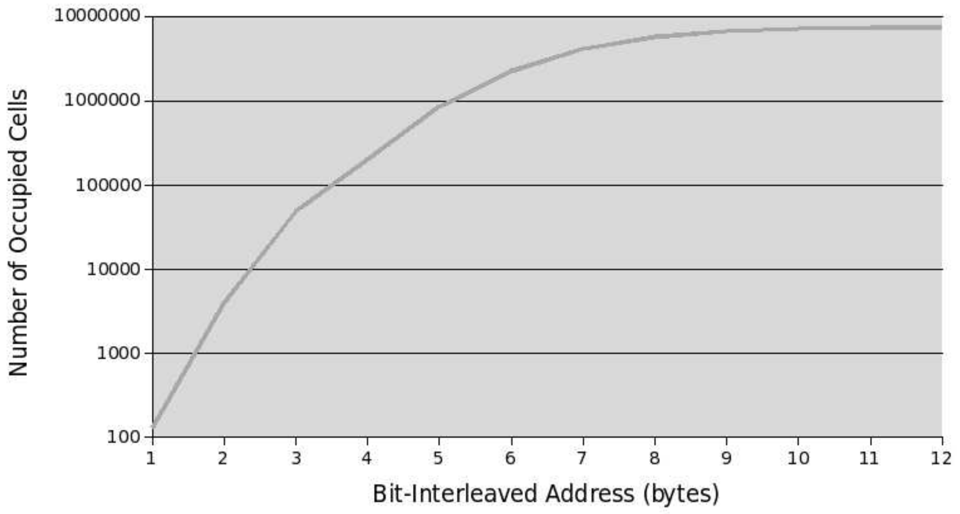

Figure 3.4: The number of cells used for varying BIA, with an exponential scale. This curve is sub-exponential, which is promising.

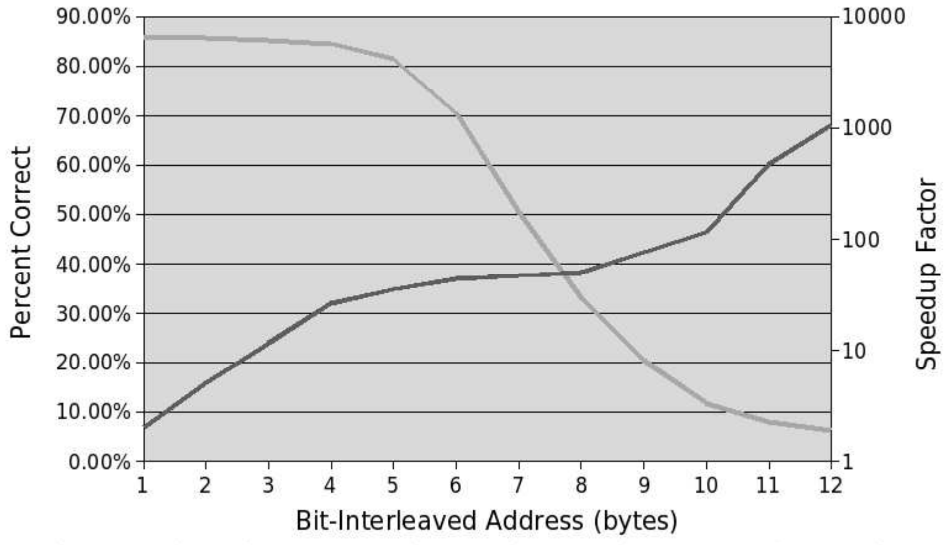

Figure 3.5: Tradeoffs between accuracy and speed-up, as a function of the size of the bit-interleaved address (in bytes) using the new data set. This new data yielded far smaller speedups than the older data.

thrashing and the process was no longer IO-bound.

3.2 New Training and Testing

One of the most important problems we had with the larger data set was that the documents added were from a different source than the original set, thus calling into question any inferences gained from those experiments. With this in mind, we set out to create a similarly sized set of documents that could be used for testing that would avoid duplicating this problem. Since classification at pixel granularity expands very quickly, we only needed a few documents, thus they were created by hand. A small set of typefaces were used, mixed with a few photographs and some handwriting. We again chose to use one document for testing and the others for training. This data set was used for the remainder of the experiments.

The results from the subsequent experiments were less promising, though equally interesting. As Figure 3.5 shows, accuracy dropped off in almost exactly the same way as as it did with the older data (see Figure 3.1), but speedup did not increase as quickly as we had hoped. We believe this behavior to be caused by the similarity among the documents used. Since the training and testing set have so much in common (they were generated in the same program, scanned by the same scanner, etc.), many of the points are likely to hash into many of the same cells, even as the size of the BIA increases. Thus, since the points are still hashing into occupied cells, speedup is minimized because more distance computations must be done. The lack of an exponential increase in speedup between 5 and 8 bytes is especially significant, and again most likely symptomatic of the characteristics of the data set. This sort of result suggests that this method is best suited to data sets that have a wide variety of document qualities. When this is the case, it is more likely that each cell will have a smaller number of training points compared to data sets with homogeneous data. Further testing on different types of data sets must be done in order to better understand this phenomenon.

Chapter 4

Filtered Training

Performing classification with the larger data sets was problematic because the memory usage of the classifier became excessive. This caused the system to thrash, significantly slowing classification. In order to combat this behavior, we propose a method of *filtered training* that classifies data without keeping the training set in memory. With this method, we first look at the (relatively small) testing data. Each testing point is stored in the appropriate location in the hash table (just as the training data had been stored in Bit-Interleaved Classification). When the training data is then loaded, points that hash into empty cells can safely be ignored since there's no testing data near them. Training points that hash into cells that already contain testing points must then *annotate* those testing points with their class and distance from that testing point (if they're among the closest k training points). This annotation means that the storage requirements are now independent of the size of the training data ($O(m)$ where m is the size of the testing data, when d is held constant), since only one training point must be stored in memory at a time. Meanwhile, the amount of hashing operations and distance computations remain exactly the same.

We found this technique to be very useful, as it ensured that classification with large training sets was once again CPU-bound. While precise performance characterization is not the goal of this thesis, we do believe that this filtering method has the potential to deliver a significant performance boost in many cases. We tested filtering on the two large data sets used in previous experiments. As expected, were able to complete our experiments more quickly than before, and most importantly the amount of memory used was

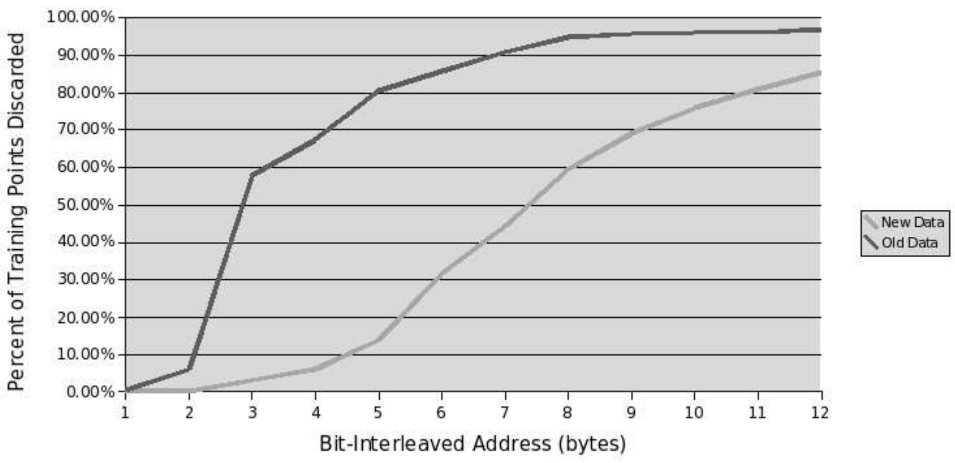

Figure 4.1: The percentage of training points discarded during filtered training for varying bytes of BIA. As expected, the older data was able to more quickly shed a large portion of its data, resulting in a larger speedup.

vastly reduced. Additionally, the output was identical except for some tie-breaking instances. This difference is not significant because the tie-breaking procedure is undefined in both methods, and should not bias the performance.

Filtered training allowed use to take a different perspective on the data and examine how many training points were never used. Although this sort of analysis was already possible with older classifiers, filtered training lends itself to this data collection because an unused training sample is one that hashes into an empty cell and is subsequently discarded. Figure 4.1 shows the percentage of training points discarded using the two most recent data sets. Not surprisingly, the data set that enjoyed huge speedups very quickly (old data) was able to discard a large number of training points very quickly, while the new data set did not discard as many points. This supports our intuition that the homogeneity of the new data set is what caused the relatively poor performance, though further experimentation should be done in order to refine this claim.

Chapter 5

Conclusion

5.1 Discussion

We have investigated algorithms for fast approximate nearest-neighbors classification of data arising in a digital libraries context where each pixel in an image of a document is to be classified according to the dominant content type—machine-print text, handwriting, photographs, etc—present in the small region around the pixel. Our principal results are empirical performance evaluation and analysis of a family of non-adaptive k-d tree data structures sped up by means of hash techniques. The non-adaptive k-d tree we used only approximates true kNN but it allows, in principle, high speed search. We investigated bit-interleaved addressing for k-D tree bins, to allow fast access via hashing. We observed that the growth of the k-d tree data structure on these data was not exponential in the size of the training set, as is possible in the worst case, but was a low-order polynomial and as a result hash-tabledata structure sizes were tractable in practice, though we expect that for larger experiments, some form of paging of hash-tables and training data will be required to avoid serious thrashing. We also investigated "filtering" methods, where test data are hashed in first, then training data is read and compared to it (this reverses the traditional order): this led to vastly reduced main-memory demands with resulting speed-ups, and with no loss in accuracy. For these methods we measured a range of tradeoffs of accuracy for speed: while they are drastic at the extremes, we found that large speed-ups (of factors of 100 or more) do not always sacrifice much accuracy.

5.2 Future Work

As we examined the results of our work, specifically the sources of errors, many ideas arose regarding other methods that may help to alleviate these errors. We hope that these directions may be explored in the future.

5.2.1 Fallback Hashing

As partition size decreases, unclassifiable errors (when a test point is in an empty partition) rapidly dominate the error rate (see Figure 3.2). In these cases, it may be advantageous to have some sort of fallback procedure instead of giving up. Specifically, we would like to "zoom out" and search a cell that has more training points. This will sacrifice some of the speedup afforded by the small partition size, but we hope that the price paid in speed may be worthwhile if it buys a significant increase in accuracy.

One of the first drawbacks we discovered when designing this method was that it could not be used effectively with filtered training. For example, if we allowed 8 levels of "zooming", each testing point would have to be listed 8 times. Additionally, each training point would have to annotate the points in all 8 of the cells that it hashes in to (since an arbitrary testing point may meet an arbitrary training point at any zoom level). This leads to some very serious problems if a testing point and training point meet at multiple zoom levels, not to mention that it sacrifices a lot of the speed gained by hashing.

For fallback hashing, we again chose to discretize the number of bits in the BIA used (r) into bytes, though there isn't a technical reason a more granular approach couldn't be used. As with the original hashing scheme, we choose a value for r to represent the maximum size of the BIA used when hashing. Instead of placing each training point in the corresponding cell from hashing on r bits, we place it in cells from hashing on 8, 16, 24... r bits. When testing, the test point is hashed to a cell using r bits of its BIA. If no points are in that cell, the classifier, "zooms out" to $r - 8$ bits and attempts to classify there, repeating the zooming operation as needed.

There is plenty of room for variation with this method, such as providing fewer discrete levels of fallback in order to conserve memory. While we were unable to complete the implementation of this method in time for publication, we believe it to be the most promising avenue for improvement.

5.2.2 Push-Forward Hashing

This method is based upon Fallback Hashing and Filtered Training. The goal is to vastly reduce or eliminate unclassifiable errors while keeping space complexity linear on the size of the testing set. In this method, we wish to classify the k nearest neighbors and put each testing point into a cell containing at least j training points (a support baseline).

For this explanation, we will consider BIA sizes to be in bytes, though there is no technical reason why other granularities couldn't be used. We start by hashing each of the testing points into cells using BIA size 1. Each testing point contains k votes (with distance & class, the same as with Filtered Training) and j *promotions* (which specify *destination* BIA levels, to be explained later).

Once all of the testing points have been inserted into the hash table, the training points area read in, one at a time. Each training point is first hashed into a cell using BIA size 1. The point then follows the same procedure used in Filtered Training to annotate the testing points in that cell with votes. In addition to this annotation, the training point offers promotions to the testing points where appropriate. A promotion to BIA size b means that the corresponding testing point and training point have identical values for the first b bytes of their BIA (and no more). A promotion is added to a testing point's list only if the promotion is greater than the minimum on the list. This means that a testing point can move to the cell with BIA size $MIN(promotions)$ and still have at least j training points in the same cell (though not physically stored there).

For example, if a testing point is moved to a cell with BIA size 3, it needs to do something in order to ensure that subsequent training points make it down there. This is done with a "bread crumbs" bit. This bit is set in any cell where a testing point has been moved to another BIA level. When training points see this bit set in their current cell, they first classify the points in the current cell (as described above), and then increment their BIA size and continue the operation. A cell that has been emptied by promotions will still exist as a hash entry because of this bread crumbs bit.

This method has some unique characteristics. The classifier will perform more slowly on the earlier points, and will continue to speed up as classification continues. It also does more manipulation with the BIA values (for determining promotions), which is an increased computational cost. Different ordering of the training points may also affect the results, since a pair of points on opposite sides of a BIA split border might meet if the testing point has not yet been promoted, but would most likely not meet if the training point arrives late in the process. Though slightly unpredictable, this behavior should do no worse than Fallback Hashing.

5.2.3 Cross-Listing

Another, perhaps less important, source of errors in our approach is *border issues*. The bit-interleaved classifier only searches the list of points in the same partition as the test point (x), but it doesn't search points in adjacent partitions. Some of the points in these adjacent partitions may be closer to x than the points in its own partition. Thus, if we can find a way to consider points in the same partition as x as well as points on the borders of adjacent partitions, it may be possible to improve accuracy of the bit-interleaved classifier. There is some concern that any attempt to solve such border issues will result in a significant increase in the time and space requirements for classification, but we believe it to be worthy of investigation.

Bibliography

[Ben75a] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[Ben75b] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[BMN$^+$06] H. S. Baird, M. A. Moll, J. Nonnemaker, M. R. Casey, and D. L. Delorenzo. Versatile document image content extraction. In *Proc., SPIE/IS&T Document Recognition & Retrieval XII Conf.*, San Jose, CA, January 2006.

[DHS01] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification, 2nd Edition*. Wiley, New York, 2001.

[FBF77] Jerome H. Freidman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[LW77] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Inf.*, 9:23–29, 1977.

[Nag00] George Nagy. Twenty years of Document Image Analysis in PAMI. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):38–62, 2000.

[NS96] G. Nagy and S. Seth. Modern optical character recognition, 1996.

[Pav00] T. Pavlidis. Thirty years at the pattern recognition front, September 2000.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts, 1990.