Prototype-Based Approximate Nearest Neighbor Search

IGOR DOZORETS∗ , ISAK GATH†∗ and HADAS SHACHNAI∗
∗Dept. of Computer Science and †Dept. of Biomedical Engineering
Technion, Israel Institute of Technology
32000 Haifa
ISRAEL
hadas@technion.ac.il, isak@biomed.technion.ac.il

Abstract : - The principles of the *p*rototype model for speaker recognition have been applied in a new version of the *A*pproximate Nearest Neighbor schema. An analytical expression for the space complexity, using the *p*rototype model, has been developed. It has also been empirically proven that increasing the error bounds when using the *A*pproximate Nearest Neighbor search speeds up the prototype-based search, with minimal reduction of search accuracy.

Key-Words: - Approximate Nearest Neighbors; Prototype-Based Search; Space and Time Complexities; Speaker Recognition.

1 Introduction

The main idea of the present work is derived from our earlier studies [1] on the problem of *Speaker Recognition or Verification* stated as follows:
Given a database S of n human voices and a speaker voice q, find a voice $p \in S$ closest to q in a minimal number of steps.

The problem is well-known and several models of speaker recognition have been suggested and studied in order to solve it efficiently. One such model, the *prototype model* [2], states that voices are recognized by the human listener *not* by any fixed number of features, but rather by those features, that significantly deviate from the population average, termed `prototype`. Human voices can be represented as vectors in a metric space X of dimension d and then the problem can be reduced to the *Nearest Neighbor* problem:
Given a set S of n data points in a metric space, X, preprocess these points so that, given any query point $q \in X$, the data point nearest to q can be reported quickly.

When very large databases of voices are involved the search complexity may be critical, in particular if a real-time application is required.

In the present study findings from the field of *Speaker Recognition* will be used, in order to obtain an efficient implementation of the general *Nearest Neighbor (NN) Search*. An analytical expression of the space complexity for the problem solution that uses the *prototype model* (i.e *prototype-based* solution) will be given. It will also be empirically proven that enlarging the error bounds, when using the *Approximate Nearest Neighbor* search techniques [3, 4], speeds up the prototype-based search, with minimal reduction of search accuracy.

2 Prototype Transformation of the Data

The main idea of the *prototype transformation* is related to the problem of speaker recognition. Accordingly, there is no fixed number of features being memorized by a listener, but rather some variable number of features being extracted from a speaker and related to their deviation from a predefined "population average" termed the `prototype`. Hence, the prototype is defined as a vector in a d dimensional space (the same as the database), with

each coordinate being the average of the corresponding feature over the entire database. The transformation process includes two steps:

1. For each vector j in the database, and for each feature i in the vector calculate the *deviation from the prototype*:

$$\delta_i^j = F_i^j - F_i^p, \tag{1}$$

where F_i^j and F_i^p are the values of the $i-th$ feature in the $j-th$ voice vector and in the prototype vector, respectively.

2. If $|\delta_i^j| \leq \Delta_i$, where Δ_i, a predefined constant value is the `tolerance` of the $i-th$ feature F_i in a voice vector, do not store its value in the transformed database (*obliterate it!*). Otherwise, store the value of δ_i^j instead of F_i^j in the database.

The probability p_i of the i-th feature to be preserved is:

$$p_i = 2(1 - G_i(\mu_i + \Delta_i)), \tag{2}$$

where $G_i(x)$ is the cumulative distribution function of the i-th feature, μ_i is the mean of the i-th feature and Δ_i is the tolerance of that feature. For Gaussian distributed features (independent) and when the tolerance values are measured in fractions of STDs, equation (2) can be written as:

$$p_i = 2(1 - \Phi(\eta_i)), \tag{3}$$

where $\Phi(x)$ is the cumulative distribution function of a Normal random variable with zero mean and variance 1, and η_i is the tolerance of the corresponding feature, measured in STD fractions.

The database of the feature vectors is stored in a simple table format, with each row, having one field for speaker identification (e.g., name), and d other fields being the values of the features under consideration.

3 Search Techniques and Complexities

High-dimensional *Nearest Neighbor* (NN) problems arise when objects are represented by vectors of d numeric features. It can be assumed that the metric space X is a $d-dimensional$ space over the real numbers, R^d and distances are measured using any of the Minkowski distance metric.

There are several data structures that support an efficient nearest neighbor search in high dimensions, for example the *kd-trees* and the *bd-trees* [5]. It has been shown [6] that $O(n)$ space and $O(\log n)$ query times are achievable, on average, through the use of these data structures. The bd-trees defined in [5] are based on spatial decompositions of space and achieve both exponential cardinality and geometric size reduction. However, the constant factors hidden in the asymptotic running time grow at least as fast as 2^d (depending on the metric) and therefore,*Approximate Nearest Neighbor* (ANN) search can be a better strategy.

3.1 Approximate Nearest Neighbor (ANN) Search

In spite of many attempts to eliminate the exponential dependencies on d in the nearest neighbor search running times, no known method achieves the simultaneous goals of roughly linear space and logarithmic query time, for any fixed dimension greater than 2. Therefore, an alternative approach of finding *approximate* nearest neighbors should be considered:

Consider a set S of data points in R^d and a query point $q \in R^d$. Given $\epsilon > 0$, we say that a point $p \in S$ is a *(1 + ϵ)-approximate nearest neighbor* of q if $dist(p,q) \leq (1+\epsilon)dist(p^*,q)$, where p^* is the true nearest neighbor to q and $dist$ is some metric in R^d. In other words, p is within relative error ϵ of the true nearest neighbor. In more general terms, for $1 \leq k \leq n$, a $k-th$ $(1+\epsilon)$-approximate nearest neighbor of q is a data point whose relative error from the true $k-th$ nearest neighbor q is

ϵ. Space requirements for the bd-trees based ANN search [5] are completely independent of ϵ and are asymptotically optimal for all parameter settings, since dn storage is needed only to store the data points. In addition, the preprocessing is independent of both ϵ and the metric. This implies that once the data structure has been constructed it can be reused for search, using various values of ϵ and different metrics. Setting $\epsilon = 0$ will cause the algorithm to search for the true nearest neighbor.

Although the time complexity of the bd-trees based ANN search still depends exponentially on dimension in the `worst case` (in contrast to the `average case` performance), it proves to be extremely effective in practical terms.

4 Prototype-Based Search

4.1 Database Modifications

In order to efficiently search the "prototype-transformed" databases the search algorithms, as well as the data structures, must be substantially modified. Three types of problems have to be tackled:

1. Each coordinate of the prototype transformed feature vector has its own "special" meaning, being a "deviation from a *certain* population average", and thus, a method for labeling the vector entries has to be formulated.

2. The dimensions of the prototype transformed vectors change after the transformation and there is no way to predict these vector dimensions. On the other hand, the search techniques described earlier operate in some R^d space, where the dimension d is fixed beforehand.

3. The general procedure of the prototype transformation may result in obliterating *all* the features in some vectors, in particular when a vector possesses features which are very "close" to the corresponding features of a calculated prototype. This may significantly reduce the search accuracy. Therefore, a vector with all of its features `obliterated` after the transformation must be handled separately.

The solution to the first problem is by storing in the k, $1 \leq k \leq d$ elements of the `coords` field of the basic ANN-type point [4] only the k features preserved after the transformation process, and in the d elements of the `members` field storing boolean variables, each one indicating if the corresponding feature has been obliterated.

Regarding the second problem, in order to adopt the structure to the basic bd-tree construction procedure, some constant value may be returned when an obliterated feature is referenced, e.g., the average value for that particular feature, the corresponding prototype feature, or zero.

The third issue is resolved by slightly altering the basic transformation procedure. When the transformation procedure encounters a vector with *all* of its features being so close to the prototype such that all of them have to be obliterated, *one* single feature, nevertheless, will be stored. The most natural feature to be stored is the one deviating *most significantly* from the prototype.

4.2 Space Complexity

The space complexity of the prototype model is calculated as follows: Assuming that p_i is the probability that the i-th feature survives (depending on η_i), the total number of coordinates needed to be stored in the transformed database, $M_{d,n}(\eta_i)$, is:

$$M_{d,n}(\eta_i) = n\left(\sum_{i=1}^{d} p_i + \prod_{i=1}^{d}(1-p_i)\right) + O(n) \; , \quad (4)$$

The product expresses the fact that the transformation process does not produce vectors with all of their features obliterated and the last term, $O(n)$, is the overhead introduced by the `members` field for identification of the obliterated features.

When η_i is constant for all values of i, $1 \leq i \leq d$ (equal tolerances), the values of p_i are also constant for all the i's. Denoting the values of p_i by p and the values of η_i by η, equation (4) is given in a simplified form:

$$M_{d,n}(\eta) = n(dp + (1-p)^d) + O(n) \ , \qquad (5)$$

Thus, for example, for $\eta = 1.0$, $d = 16$ and $n = 10^4$, the total space savings will be 1MB with only 19.5KB overhead introduced by the $O(n)$ term.

4.3 Preprocessing Complexity

An additional overhead in constructing the search tree is introduced, due to the prototype transformation. This includes calculation of the prototype and transformation of the data and the query points. However, transformation of the query points set is performed only once, *before* the query is run, and is, therefore, considered as a part of the preprocessing stage.

It can be assumed that the prototype consists of the features' averages, and the overhead is proportional to $dn + dm$, where d is the space dimension and n and m are the number of the data points and that of the query points, respectively. The process of constructing the search tree, including the overhead for the transformation, requires therefore, $O(dn \log n + dm)$ time.

5 Performance

5.1 Constructing the Data Structure

The preprocessing stage of constructing the bd-tree is accomplished in time proportional to $dn \log n$. For a small number of queries the overall construction and search times will be high, even when compared to the brute-force search strategy (which does not require preprocessing). However, whenever the number of queries is sufficiently large, the overhead introduced by the preprocessing stage becomes negligible. For example, when the number of query points is the same as the number of the data points, preprocessing for two dimensions is about 25 percent of the total computation time, while for eight dimensions that fraction is between three and five percent.

5.2 Constructing the BD-tree

For the purpose of the experiments, several databases, each consisting of 10,000 artificial voices with various dimensions, were generated and the bd-trees constructed using the *sliding midpoint* splitting rule [4].

The experiments show that the tree depth grows with tolerance. The reason is that the higher the tolerance, the more clustered the data set is, and when the data set becomes more clustered, more shrinking nodes are required to preserve the *packing constraint* [4].

Construction time is strictly dependent on tree dimensionality. In addition, the total time required to construct the bd-tree depends largely on the value of the tolerance when transforming the database.

5.3 Searching the Data Structure

The empirical results given hereby describe the behaviour of the search techniques studied under different conditions. However, several parameters remain invariant in all the experiments, unless explicitly stated otherwise:
The number of data points and query points are 10,000 and 2000, respectively. The number of nearest neighbors searched for each query is 1 and the bucket size [4] is 1. The metric employed is the L_∞, defined by:

$$dist(F^j, F^k) = \max_{1 \leq i \leq d} |F_i^j - F_i^k| \qquad (6)$$

where j and k are two points (vectors).
The choice of the L_∞ metric in the simulations has been dictated by the relative simplicity of the theoretically derived search time complexity for this type of metric. L_∞ has been shown to be superior to other Minkowski metrics, as far as time complexity of the kd-tree search procedures is concerned [6].

In all the simulations the bd-trees under study were constructed using the *sliding midpoint* splitting rule and the *standard* shrinking

rule. The search has been performed using the *priority* search technique [3]. All the results were averaged over the 2000 queries. The statistical uncertainty of these averages is quite small, approximately two percent in the worst cases.

5.4 Dependence on the size of the data

The search complexity of the bd-tree structure is proportional to $\log n$, where n is the size of the data set, with constant factors rising exponentially with some $k, 1 \leq k \leq d$, where k is the dimensionality of the intrinsic geometry of the data.

In order to examine the logarithmic dependency on n of the prototype-based search process, the 4-dimensional data was employed, with n varying from 500 to 10,000. Figure 1 demonstrates the average number of points visited during the search for the true ($\epsilon = 0$) nearest neighbor. It can be clearly seen that the growth is logarithmic in n, in addition to the superiority of the prototype search, in particular with tolerance equal to 0.5.

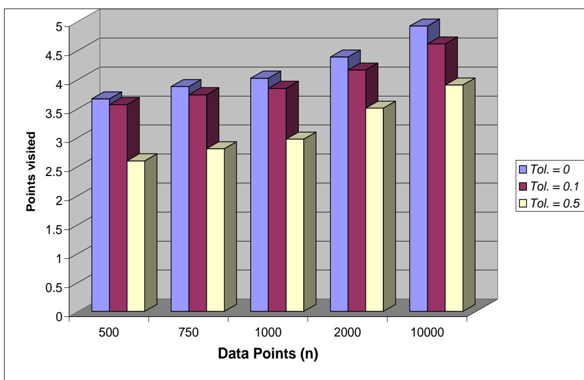

Figure 1: Average number of points visited per query versus data size and tolerance in bd-trees.

5.5 Searching with different error bounds

The empirically measured parameters, related to the complexity of the search with different error bounds ($\epsilon > 0$), are depicted in Figure 2. A marked improvement in the performance of the search technique, based on the bd-trees, is evident for $\epsilon \geq 1$. It is worthwhile

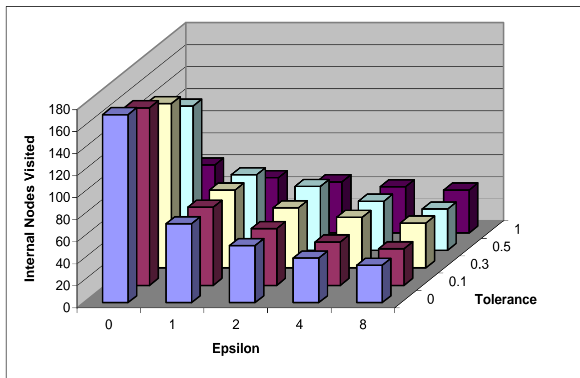

Figure 2: Average number of internal nodes visited per query, in 16-dimensional bd-trees, versus error bound and tolerance.

noting the unexpectedly good performance of the search, compared to the worst case prediction in [5]. For example, for $\epsilon = 1$ and dimension 16, the upper bound on the running time given in [5] is 10^{32}, whereas the plot in Figure 2 shows that the average number of internal nodes visited in the search process is 71.64. The plot also reveals that even for the low error bounds the number of internal nodes visited is reduced by a factor of 4 over the exact case. This high complexity reduction rate is retained even when the tolerance grows.

Increasing the tolerance in itself reduces the search complexity. For example, for $\epsilon = 0$, increasing the tolerance to 1.0 reduces the number of points visited in the 16-dimensional bd-trees search by a factor of 5. However, as the plot reveals, there is little or no improvement (in running time) when tolerance values are increased in the cases of $\epsilon > 0$.

5.6 Search error rates

For the purpose of error rate measurements, the average *rank error*, defined as the absolute value of the difference between the reported rank of the nearest neighbor and its true position among the nearest neighbors, and the percentage of true nearest neighbors found during the search have been computed.

It has been found that the rank error rises quickly with the tolerance, resulting in a particular high jump between the values of 0.3 and 0.5 for all values of ϵ. However, from Figure 3 it can be seen that even with this high average error rank (369) more than one third of the true nearest neighbors were detected. When tolerance is set to zero and ϵ is set to

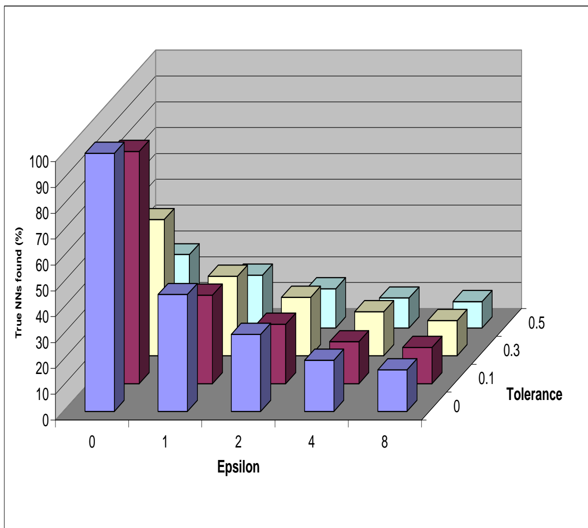

Figure 3: The fraction of true nearest neighbors found searching 8-dimensional bd-trees, versus error bounds and tolerances.

1, the plot shows that also in this case more than one third of the true nearest neighbors were found, however, the rank error is as low as 2.

6 Conclusions

The data employed as a training set in the present study is derived from experiments dealing with recognition of familiar voices. However, the results of the ANN prototype-based search obtained here may be applied to any high-dimensional nearest neighbor problem.

It could be verified through multiple simulation experiments that the prototype transformation of the data resulted in significant savings in storage requirements. The prototype-based search introduces a `tradeoff` between search **complexity** and search **accuracy**. It is highly effective in search time requirements when the data set is clustered **close** to the "average", resulting in obliteration of most of the features. The relative accuracy reduction introduced by the prototype-based search remains reasonable only when the query data is relatively **far** from the prototype, i.e. remote from the data set "average".

However, the gains in search complexity, achieved by obliterating the features not deviating significantly from the prototype, are of minor importance when higher error bounds are used in the search and when a large fraction of true nearest neighbors is required.

References

[1] Y. Lavner, I. Gath and J. Rosenhouse, Acoustic features and perceptive processes in the identification of familiar voices, *Speech Communication*, Vol. 30, 2000, pp. 9-26.

[2] Y. Lavner, J. Rosenhouse and I. Gath, The prototype model in speaker identification, *EUROSPEECH'99*, Budapest, Sept. 1999.

[3] S. Arya and D. Mount, Approximate nearest neighbor queries in fixed dimensions, In *SODA'93*, pp. 271-280.

[4] D. Mount and S. Arya, A library for approximate nearest neighbor searching, *CGC 2-nd annual workshop on computational geometry*, 1997.

[5] S. Arya, D. Mount, N. Netanyahu, R. Silverman and Wu A., An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions, *SODA'94*, pp. 573-582.

[6] J.H. Friedman, J.L. Bentley, and R.A. Finkel, An algorithm for finding best matches in logarithmic expected time, *ACM Transactions on Mathematical Software*, Vol. 3, 1977, pp. 209-226.