K-tree: A HEIGHT BALANCED TREE STRUCTURED VECTOR QUANTIZER

Shlomo Geva
Machine Learning Research Centre
School of Computing Science
Queensland University of Technology
GPO Box 2434, Brisbane
Queensland 4001
Australia
s.geva@qut.edu.au

Abstract: We describe a clustering algorithm for the design of height balanced trees for vector quantisation. The algorithm is a hybrid of the *B-tree* and the *k-means* clustering procedure. *K-tree* supports on-line dynamic tree construction. The properties of the resulting search tree and clustering codebook are comparable to that of codebooks obtained by *TSVQ*, the commonly used recursive *k-means* algorithm for constructing vector quantization search trees. The *K-tree* algorithm scales up to larger data sets than *TSVQ*, produces codebooks with somewhat higher distortion rates, but facilitates greater control over the properties of the resulting codebooks. We demonstrate the properties and performance of *K-tree* and compare it with *TSVQ* and with *k-means*.

1. INTRODUCTION

Vector quantisation is a powerful computational procedure that has many uses in signal and image processing, data compression, and in statistical and machine learning applications. A comprehensive coverage of vector quantisation methods can be found in [1] and numerous applications are described in [2] and in the current literature.

Vector quantisation is encountered in two basic forms, which depend on the application. In data compression one is concerned with the identification of a set of cluster vectors (or a **codebook**) that typify vectors that are to be compressed. In such cases one usually applies an unsupervised cluster discovery procedure, such as *k-means* [3]. In classification problems, on the other hand, supervised learning algorithms are used. The codebook is constructed from a set of labelled data. The procedure is then concerned not only with clustering, but also with correct labelling of the codebook vectors, as for instance in LVQ [4]. This paper is concerned with unsupervised clustering, although the *K-tree* algorithm is also useful in the context of supervised clustering and classification.

The *k-means* algorithm [3] is a classical clustering algorithm. The procedure is also known, in various variations and extensions, as *GLA*, the Generalized Lloyd Algorithm, or as LBG, after the authors who described it [1]. Given a training data set **T**, the objective is to design a codebook **C** having *N* codebook vectors that minimise the quantization distortion error of the codebook. The quantisation distortion is most commonly defined as the average (mean square) distortion resulting from replacing each training vector by its nearest codebook vector. The *Euclidean distance* is usually applied in measuring the distortion error, but other measures, such as the *absolute error* or the *worst case* error, are also encountered.

The *k-means* procedure starts by an initial estimation (usually random selection) of ***N*** vectors from **T**. The procedure then proceeds in two iterated steps until convergence. In the first step the training set is partitioned into ***N*** subsets $\{\mathbf{T}_1..\mathbf{T}_N\}$ defined by the Voronoi tessellation of **T** by the current codebook vectors $\{\mathbf{C}_1..\mathbf{C}_N\}$. Each vector $\mathbf{C}_i$ is the nearest codebook vector to the training vectors in partition $\mathbf{T}_i$. The second step is the re-estimation of codebook vectors - each vector $\mathbf{C}_i$ is re-assigned the mean value of all the vectors in partition $\mathbf{T}_i$. These two steps are iterated until some termination condition is met. Usually, until the procedure converges into a stable configuration. It can be shown that each iteration must reduce or leave unchanged the average distortion [1].

A common variation to the basic *k-means* algorithm involves *splitting* and supports incremental construction of a codebook. The procedure starts with the application of *k-means* to two codebook vectors. Once convergence is reached a small perturbation is applied to a copy of each codebook vector thus splitting the codebook into four vectors. The clustering algorithm is then re-applied to the new codebook. The process of splitting and clustering is continued until the desired number of codebook vectors is reached, or until the distortion error of the codebook falls below a pre-determined value.

In many applications large codebooks are involved and an efficient search procedure is required for implementation. The *TSVQ* algorithm is a recursive partitioning procedure that builds a search-tree. The procedure starts with the generation of a codebook of ***N*** vectors $\{\mathbf{C}_1..\mathbf{C}_N\}$ through the use of the *k-means* algorithm. Each of the corresponding partitions $\{T_1..T_N\}$ is then further processed by the application of *k-means* to vectors in each partition $\mathbf{T}_i$. The process is repeated recursively until the desired tree depth is reached. A common variation relaxes the requirement of having a balanced tree and nodes are recursively split until a desired distortion rate is achieved at all leaves. The leaf nodes of the tree define the final clustering codebook. The search for a codebook vector nearest to a given input vector starts at the root, progressing through the tree, selecting the next branch by nearest neighbor comparisons, until the nearest leaf node vector is identified. It should be noted that the *TSVQ* search tree does not guarantee the identification of the nearest leaf-level codebook vector. However, when it is not the nearest

neighbor, another *nearby* vector is identified instead, and the error is usually of no significant practical importance. The *TSVQ* tree is not necessarily height balanced and leaf nodes are usually of unequal size. The distortion error of codebooks obtained by *TSVQ* is higher than that of codebooks designed by *k-means* and having the same number of clusters.

2. THE *K-tree* ALGORITHM

The problem of constructing a search tree in the context of *keyed* access to records in a file is very well studied and understood. A classical abstract data type for the bottom-up construction of height balanced search trees is the *B-tree.* A *B-tree* is built by the insertion of keys into the tree at the leaf level. Insertions proceed sequentially, and the tree is accessible as a search tree at any point during its construction. This is in contrast to the *TSVQ* procedure where the tree is built from the top down. We now describe how this algorithm is adapted to apply in vector quantisation problems.

We start with the definition of a *B-tree* of order *m*:

1. All leaves are on the same level.
2. All internal nodes except the root have at most *m* nonempty children, and at least *m/2* nonempty children.
3. The number of keys in each internal node is one less than the number of its nonempty children, and these keys partition the keys in the children to form a search tree.
4. The root has at most *m* children, but may have as few as *2* if it is not a leaf, or none, if the tree consists of the root alone.

A *K-tree* of order *m* is defined as follows:

1. All leaves are on the same level.
2. All internal nodes, including the root, have at most *m* nonempty children, and at least 1 nonempty children.
3. Codebook vectors (clusters) act as search keys.
4. The number of keys in each internal node is equal to the number of its nonempty children, and these keys partition the keys in the children to form a *nearest neighbour* search tree.
5. The level immediately above the leaf level forms the clustering ***codebook level***.
6. Leaf nodes contain data vectors, or references to data vectors.

K-tree nodes store real valued vectors. The search through the tree is based on nearest neighbour comparisons, rather than on key comparisons. The nature of the distance measure used in nearest neighbour comparisons is in general an implementation dependent choice. We have used the Euclidean distance.

A *K-tree* consists of two kinds of nodes. All internal branch nodes, starting from the root, down to the last level above the leaf level, contain mean-

vectors (clusters). The leaf nodes of a *K-Tree* store the training data vectors (or references to them). The clustering codebook is contained at the **codebook level** - the 2nd last level of the tree. All branches above that level define the search tree, and all leaves below that level store data vectors. If the *K-tree* is used to search the original data set (for example, if the data vectors are used in nearest neighbor classification) then the search terminates at the leaf level, with the nearest data vector. If the application only requires the nearest codebook vector (eg. in signal compression) then the search terminates at the codebook level (the leaf level can in fact be discarded).

2.1 *K-tree* construction

In order to explain the *K-tree* construction procedure it is convenient to start from an already existing tree. Insertion proceeds as follows: An input vector **x** is presented for insertion. The tree is searched to identify the leaf node containing the nearest leaf (**data)** vector. In the simplest case the number of data vectors stored at the leaf node is less than N, the tree order, and the vector is inserted at that node. All branch vectors on the path leading to that leaf are also updated since the search tree must reflect the new cluster structure that had changed slightly now that an additional vector was inserted. Consider the node just above the leaf. It contains the mean vector of the leaf into which **x** was inserted. This mean vector must be updated to reflect the new cluster mean. In a similar manner, the node above the branch node just considered needs to reflect the insertion, and again the mean vector of that branch is updated at the level above it. The process continues all the way to the root node. This procedure lends itself to efficient recursive implementation similar to that of a *B-tree*.

Now consider the insertion of a data vector resulting in a full leaf node. That leaf node requires a split. The *k-means* algorithm is applied to the data vectors at the leaf, to generate two new clusters ($k=2$) in place of the original cluster. The two new cluster means now replace the single cluster mean at the parent node. This of course can result in a full parent node. In that case the parent node is split into two via *k-means*, except that now the clustering is applied to cluster mean vectors at an internal branch instead of to data vectors. The two new cluster means are then pushed up the tree. In the case that the root node itself becomes full the tree depth increases – a new root node is created, receiving the two new cluster means.

Starting from an empty tree, the *K-tree* is initialised with a root node and a single leaf node consisting of a single data vector. Insertion of data vectors now proceeds as described above. Initially the tree tends to grow in depth rapidly. However, the capacity of a tree consisting of a root node with N clusters and the corresponding leaf nodes is N^2, and after another root node split the maximum capacity of the leaf level of the *K-tree* is N^3. As the *K-tree* capacity grows very rapidly with increased depth, it reaches its final depth (for a given number of training vectors and tree order) long before all the vectors have been inserted. Given a data set of M vectors, a *K-tree* of order N

reaches its full depth after about M / N insertions. From that point on the tree depth remains fixed and insertions occur with an almost constant rate of node splits. The insertion time is not constant though. It slowly increases since the average number of comparisons along the search path increases. The number of comparisons along the search path of a *K-tree* is of the order $O(N\log_N(m))$ where N is the tree order and m the number of vectors already stored in the tree. Consequently, the procedure scales up very well to very large training sets.

The *K-tree* approach to the construction of a clustering search tree is radically different to the traditional recursive partitioning approach. To visually demonstrate the structure of the resulting codebook we have built a *K-tree* Using a set of 5,000 vectors drawn from a random normal distribution with a mean of 0 and a variance of 1. Figure 1 depicts the Voronoi tessellation of the **codebook level** vectors in the *K-tree*. Figure 2 depicts the distribution of the same number of clusters, obtained by the direct application of *k-means* to the same set of training vectors. The result is qualitatively very similar, albeit the *k-means* procedure leads to a lower distortion rate as expected.

3. NODE SPLITTING STRATEGIES

It is evident that the choice of the tree order N is arbitrary, and influences the final structure of the codebook. It may be desirable to allow more flexibility in the construction of the tree, and use some other criteria that is more closely related to the training set characteristics, in deciding when a node should split. One such criterion is the **local** distortion rate of the leaf node. The local distortion is measured in the same manner as the global distortion defined as the average (mean square) distortion resulting from replacing each training vector by its nearest codebook vector. Rather than define a node as full when the number of vectors exceeds N, it can be defined as full when its distortion rate exceeds a threshold D. With this modifications the dynamic behavior of the tree as it is constructed is somewhat different. In the initial stages splits occur very rapidly as the tree does not yet cover the input domain adequately. Consequently almost every insertion causes the local distortion to exceed D. Many near-empty nodes are initially distributed over input space. Once this has occurred the tree becomes more stable, and splits become less and less frequent.

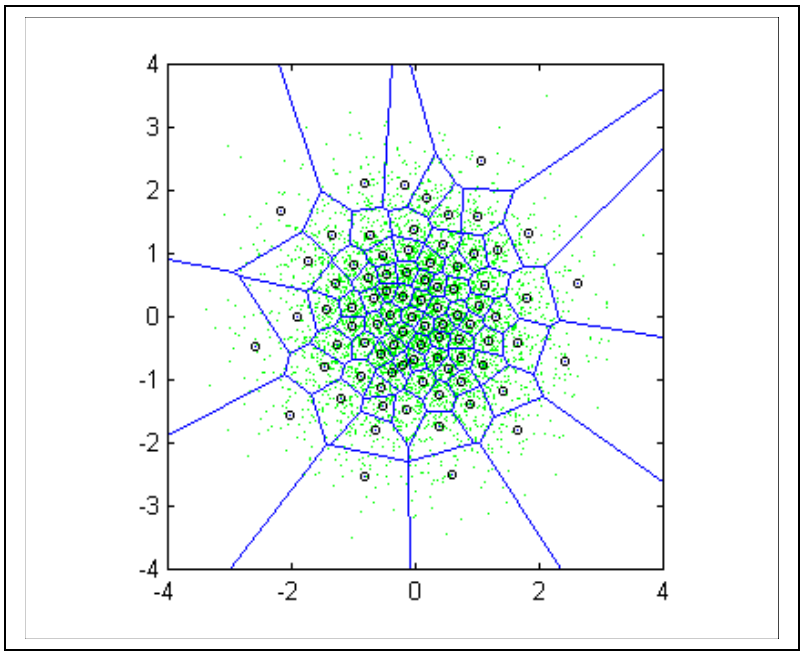

Figure 1: *K-tree* codebook

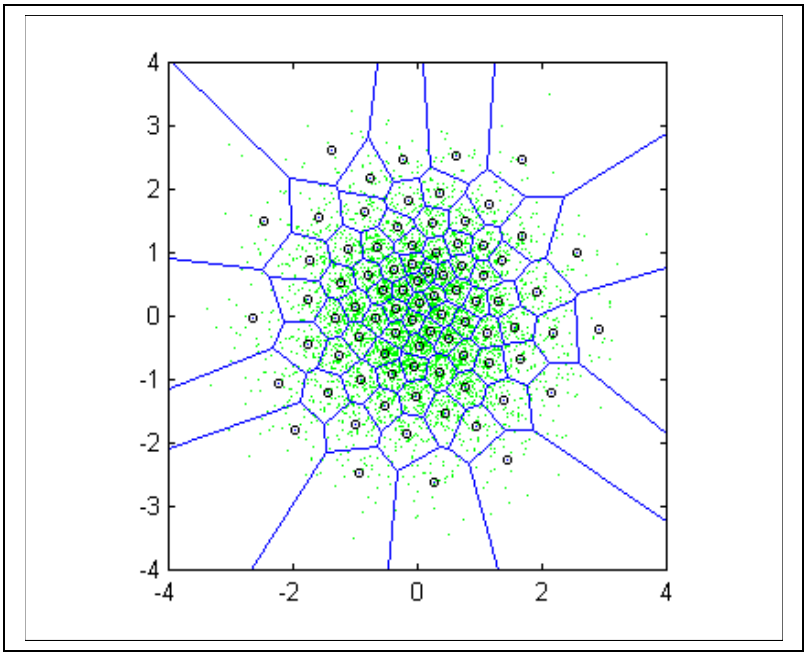

Figure 2: *k-means* codebook

One could also relax the limit of *N* entries at-most for leaf level nodes (note that branch nodes are still limited to *N* entries). The leaf nodes can instead only split when the local distortion exceeds *D*, rather than when their number exceeds *N*. Consequently nodes that correspond to densely populated regions in input space contain more data vectors. This also means that the distribution of codebook vectors will not match the distribution of data vectors in the training set and will be biased towards the more 'novel' regions of the input space. Figure 3 complements figures 1 and 2, and depicts the distribution of codebook vectors with *D = 0.15* and *N=100* at internal branch nodes, and with no limitation on the number of vectors at leaf nodes. A comparison of Figures 2 and 3 reveals that less *K-tree* clusters are dedicated to the densely populated regions and more clusters are dedicated to the sparsely populated regions, than is the case with the *k-means* clusters. The codebooks are now qualitatively different. The overall distortion rate of the *K-tree* codebook is higher than that of the *k-means* codebook, but the variance in the local distortion of the *K-tree* clusters is lower than of the *k-means* clusters.

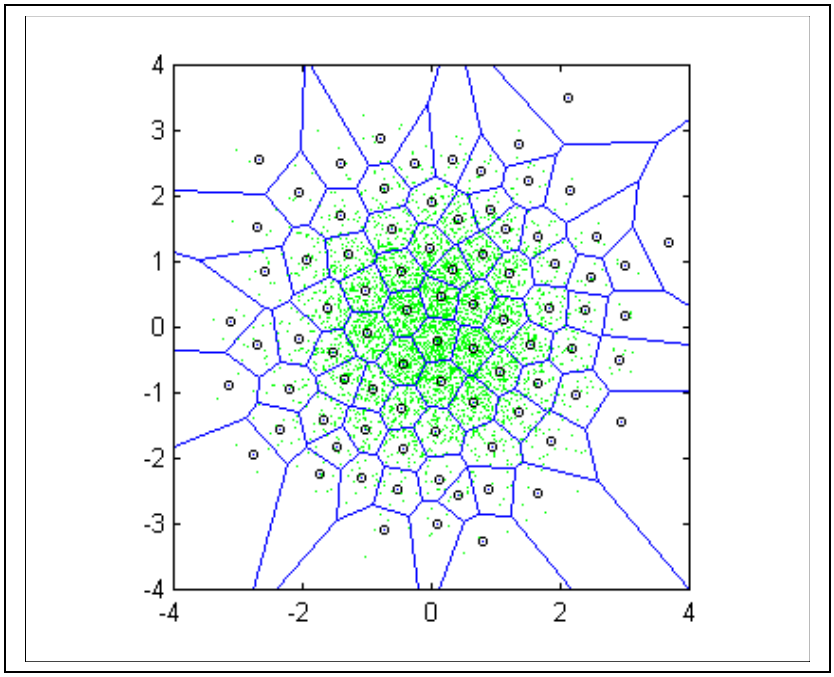

Figure 3: *K-tree* with controlled local distortion

4. PERFORMANCE COMPARISONS

In this section we compare the performance of the *K-tree* with *k-means* and with *TSVQ*. In the first set of experiments we compare the time complexity and scalability of the methods with simple artificial data sets. We have conducted clustering experiments with 3-dimensional vectors from a random normal distribution with a zero mean and a variance of 1. Starting from a set of 1,000 vectors the tree order was held constant, at *N=50*, while the number of vectors was increased in successive experiments, by doubling the training set size, from 1,000 to 256,000 vectors. Codebooks generated by *TSVQ* and by *K-tree* were of almost identical size. In each experiment we generated the *k-means* codebook with the same number of clusters as obtained in the codebook level of the *K-tree*. In this manner all comparisons relate to the same number of codebook clusters and are therefore meaningful. Each clustering experiment was repeated 10 times to obtain an average execution time. The results are depicted in Figure 4. The *K-tree* procedure is the only procedure that scales well to very large data sets. It is computationally feasible with data sets that are several orders of magnitude larger than the other methods can feasibly handle.

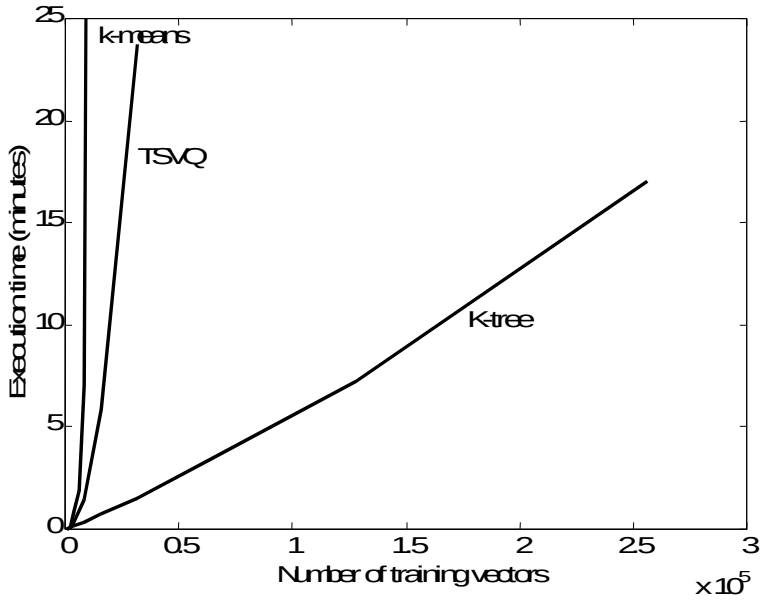

Figure 4: Scalability of *K-tree, TSVQ*, and *k-means* to large data sets

In the second set of experiments we compare the global distortion rates of the codebooks as we increase their size, by reducing *N*, the tree order, and keeping the size of the training set constant. Again, the size of the *k-means*

codebook is set to the same size as that of the codebook level of the *K-tree*. All three methods were implemented in MATLAB programs that call the same function implementation of the *k-means* procedure where they spend most of the execution time. The training set consisted of 3924, 20-dimensional, speech cepstrum vectors. The results are depicted in figure 5. As the number of clusters increases, the distortion of the codebooks decreases. The *k-means* codebooks always outperform the *TSVQ* and the *K-tree* codebooks, as expected. There is an inherent trade-off between distortion and search efficiency when constructing the trees.

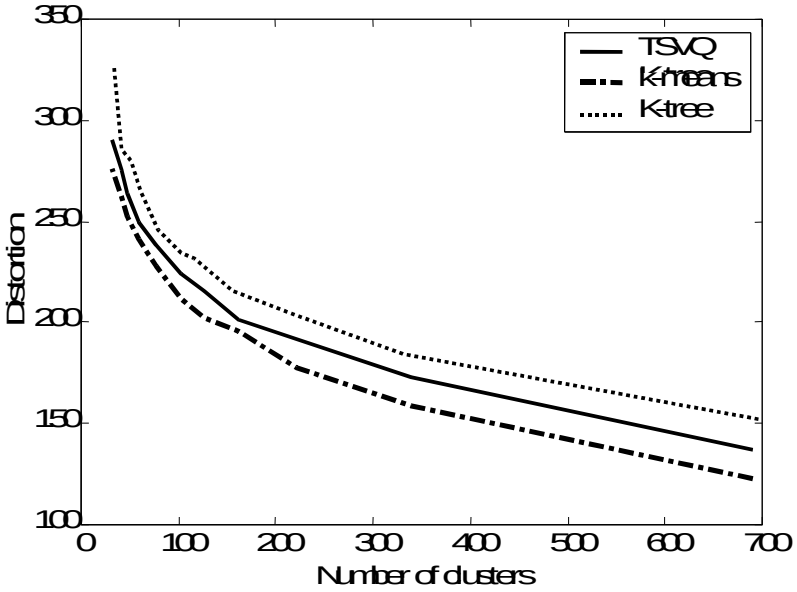

Figure 5: Comparison of codebook clustering distortion

5. CONCLUSION

We introduced the *K-tree* clustering algorithm for the construction of height balanced search trees for vector quantization. The algorithm is hybrid of two classical algorithms, *B-tree* and *k-means*. The bottom-up approach of *K-tree* represents a significant deviation from the top-down recursive partitioning approach that is at the basis of numerous variations on *TSVQ* and *k-means*. The procedure lends itself to efficient recursive implementation and the performance of the *K-tree* was compared to that of *TSVQ*. K-tree is computationally more efficient for trees of higher order (node capacity) and consequently it also scales up to data sets that are several orders of magnitude larger. *K-tree* builds trees having higher distortion rates than *TSVQ* for the same number of clusters and tree size so there is a trade off. By construction, the *K-tree* is a height-balanced tree. *K-tree* has the advantage over top-down

tree building methods in that the search tree is a dynamic structure and the codebook can be used at any time during tree construction. This allows the deployment of an adaptive tree structured codebook. Although we have not described the deletion procedure, deletion of vectors from a *K-tree* is a straightforward extension of a deletion of keys in a *B-tree*. Future work will aim to study the utility of *K-trees* in dynamic domains and the utility of various node-splitting strategies in controlling the codebooks' local distortion rates.

References:

1. A. Gersho and R. Gray. **Vector Quantisation and Signal compression**, 1992, Kluwer Academic Publishers. ISBN 0-7923-9181-0
2. Huseyin abut, Editor, **Vector Quantisation**, EEE Selected Reprints Series. IEEE Press, 1990, ISBN 0-87942-265-3
3. J. MacQueen. "Some methods for classification and analysis of multivariate observations," In *Proc. of the Fifth Berkeley Symposium on Math. Stat. and Prob.,* volume 1, pp. 281-296, 1967.
4. Kohonen T. **Self-Organization and Associative Memory**, 2nd Ed, 1988, Springer-Verlag, ISBN 3-540-18314-0 2.