Classification by ALH-Fast Algorithm

Tao Yang**, Vojislav Kecman†, Longbing Cao

Faculty of Engineering and Information Technology, University of Technology Sydney, Sydney 2007, Australia; † Department of Computer Science, The Virginia Commonwealth University (VCU), Richmond, VA 23284-3068, USA

Abstract: The adaptive local hyperplane (ALH) algorithm is a very recently proposed classifier, which has been shown to perform better than many other benchmarking classifiers including support vector machine (SVM), K-nearest neighbor (KNN), linear discriminant analysis (LDA), and K-local hyperplane distance nearest neighbor (HKNN) algorithms. Although the ALH algorithm is well formulated and despite the fact that it performs well in practice, its scalability over a very large data set is limited due to the online distance computations associated with all training instances. In this paper, a novel algorithm, called ALH-Fast and obtained by combining the classification tree algorithm and the ALH, is proposed to reduce the computational load of the ALH algorithm. The experiment results on two large data sets show that the ALH-Fast algorithm is both much faster and more accurate than the ALH algorithm.

Key words: classification; adaptive local hyperplane (ALH); decision tree

Introduction

Recently, a series of local manifold-based classifiers have been proposed to improve the K-nearest neighbor (KNN) classifier, including nearest feature line (NFL) [1], K-local hyperplane distance nearest neighbor (HKNN) [2], nearest neighbor line (NNL) [3], and center-based nearest neighbor (CNN) [4]. These algorithms implicitly assume the local data cloud belongs to the same class and they try to approximate the local data cloud by some kind of low-dimensional manifolds. The essential difference between them lies in the manifold approximation method. Among the local manifold-based methods, the HKNN algorithm has been shown to perform very well in a variety of applications [2,5,6]. The basic idea of the HKNN algorithm is to approximate the local data cloud of each class by a so-called “local hyperplane”. Specifically, in HKNN, K-nearest neighbors of a query are first selected from each class as the class prototypes by the KNN method, and then a local hyperplane is constructed to approximate the local manifold of each class based on these class prototypes. Hence, the class label of the query is assigned according to the shortest distance between the query and the local hyperplane of each class. For more details about HKNN, please refer to Ref. [2].

The adaptive local hyperplane (ALH) algorithm is a very recently proposed classifier [7], which is both the extension and the improvement of the K-local hyperplane distance nearest neighbor (HKNN) algorithm. ALH has been shown to perform better than many other benchmarking classifiers including support vector machines (SVM), KNN, linear discriminant analysis (LDA) and HKNN algorithms over many benchmarking data sets [7]. Encouraged by such performance, ALH has been applied in face recognition tasks and the experimental results obtained for two benchmarking face data sets (the Cambridge ORL face data and the Yale face data set) are shown in Ref. [8]. Again, ALH has outperformed five other face recognition classifiers.
(KNN, SVM, NFL, and NNL algorithms). In addition, (for the ORL face data set with a 5/5 split of data in the training and test sets), ALH has outperformed the following nine approaches: Eigenface method, Discriminant Eigenface, Waveletface+NN, Discriminant Waveletface+NN, Discriminant Waveletface+MLP, Discriminant Waveletface+NFL, Discriminant Waveletface+NFP, Discriminant Waveletface+NFS as well as the HKNN algorithm combined with various feature extraction methods. Finally, the ALH algorithm’s performance has been presented on small medical data sets in Ref. [9]. The experimental results on the two cancer datasets demonstrate that ALH outperforms, on average, all the other four benchmarking classifiers (mega-trend diffusion classifier (MTD), back-propagation-network (BPN), SVMs, and decision trees (DT)) for learning small medical data sets.

Although the ALH algorithm is well formulated and despite the fact that it performs well in practice, its scalability over a very large data set is limited due to its testing speed and storage requirements. In this paper, we propose a novel classification algorithm, called ALH-Fast, in order to reduce both the testing speed and storage requirement of the ALH algorithm. The basic idea of the ALH-Fast algorithm is to combine the ALH and classification tree algorithms in an efficient way. Specifically, the classification tree is constructed over the training data set first, and then the ALH algorithm is applied at each leaf of the constructed tree. That is, a new query that falls into a particular leaf of the classification tree is classified by ALH; only training instances that fall into that leaf are used for constructing ALH models.

1 ALH Algorithm

In the general classification or pattern recognition problems, we are given a training set of, say, $l$ instances with $d$ input features. Each training instance belongs to one of a small set of labels with $C$ classes, and it can be denoted as $x_i = (x_{i1}, \ldots, x_{id})^T$ with known class label $y_i = c$, for $i = 1, \ldots, l$ and $c = 1, \ldots, C$. The objective of the classification tasks is to predict the class label of an unlabeled query with an input vector $q = (q_1, \ldots, q_d)^T$.

The origins of the ALH algorithm are in the class of KNN algorithms and it is a significant improvement of the HKNN algorithm. The basic mechanism of ALH can be summarized as nearest neighbor (NN) selection, local hyperplane construction, and query classification. The K NNs of $q$ are selected first and all the other training instances will be ignored in later computations. The K NNs of $q$ are selected by using the weighted Euclidean distance,

$$D(x_i, q) = \sqrt{\sum_{j=1}^d w_j(x_{ij} - q_j)^2} \quad (1)$$

where $w_j$ refers to the weight of feature $j$, and it is found by using the exponential weighting scheme on the ratio of between-group to within-group sum-of-squares (RBWSS) for feature $j$. Specifically, $w_j$ is defined as follows:

$$w_j = \frac{\exp(TR_j)}{\sum_{j'}\exp(TR_{j'})} \quad (2)$$

$$R_j = r_j / \max(r_j) \quad (3)$$

$$r_j = \frac{\sum_{i: y_i = c} I(y_i = c)(x_{ij} - \bar{x}_c)^2}{\sum_{i: y_i = c} I(y_i = c)(x_{ij} - \bar{x}_c)^2}, \forall j = 1, \ldots, d \quad (4)$$

where $T$ is a positive parameter that controls the influence of $R_j$ on $w_j$. If $T = 0$, then $w_j = 1/d$ and all the features are assigned with equal weight. On the other hand, when $T$ is large, a change in $R_j$ will be exponentially reflected in $w_j$. The notation of $I(\cdot)$ represents the indicator function, $\bar{x}_{cj}$ denotes the $j$-th component of class centroid of class $c$, and $\bar{x}_j$ denotes the $j$-th component of the grand class centroid.

The NN rule is subject to the assumption that class conditional probabilities are locally constant. This assumption becomes invalid for data sets with high dimensions and finite instances due to the curse of dimensionality. The weighting scheme imposed on the NN selection can resolve this severe bias caused by the curse of dimensionality, and therefore, it is of great significance to any classification tasks with high dimensional and sparse data sets. In choosing the $K$ NNs there is a difference between ALH and HKNN. Unlike in HKNN which keeps same $K$ for all the classes, in ALH the value of $K$ refers to $K$ NNs to $q$ from all the classes. Hence, there can be from 0 to $K$ NNs of a certain class within the $K$-neighborhood of $q$. It is straightforward to see that the procedure used here is more sensible than the procedure used in HKNN when the sample sizes of each class of the data set differ a lot.

Unlike the KNN algorithm, ALH does not classify
q by using the majority voting scheme on its NNs. Instead, a local hyperplane is constructed for each class to find a virtually enriched training set around q. The local hyperplane for class c around q, LH_c(q), is defined as the hyperplane passing through all the NNs of q in class c, and it can be used to fantasize the unseen instances based on a local linear approximation of the manifold of class c. This idea of the local hyperplane is first proposed in HKNN. Formally, LH_c(q) is defined as

\[ LH_c(q) = \{ s | s = \sum_{i=1}^{n_c} \alpha_i V_{ij} + m \} \tag{5} \]

where \( m = \frac{1}{n_c} \sum_{i=1}^{n_c} \mathbf{p}_i \), \( \mathbf{p}_i \) is the i-th NN of class c, \( V \) is the \( d \times n_c \) matrix whose i-th column is defined as: \( V_{ij} = \mathbf{p}_i - m \), \( n_c \) is the number of prototypes in class c, and \( a = (a_1, \ldots, a_{n_c})^T \) are solved by minimizing the distance between q and LH_c(q) with regularization, which leads to the minimization of

\[ J_c(q) = \min_{\alpha} \sum_{j=1}^{d} w_j (V_{ij} a + m_j - q_j)^2 + \lambda a^T a = \min_{\alpha} (s - q)^T W (s - q) + \lambda a^T a \tag{6} \]

where \( V_j \) is the j-th row of \( V \), \( s \in LH_c(q) \), \( W \) is the diagonal matrix with \( W(j,j) = w_j \) and \( \lambda \) is the regularization parameter. It can be shown that the minimization of Eq. (6) will be achieved by solving the equation given below for \( a \),

\[ (U^T V + \lambda I_n) a = U^T (q - m) \tag{7} \]

where \( U^T = V^T W \). Derivation of Eq. (7) starts equal to zero the derivative of the cost function \( J_c(q) \) with respect to \( a \) as follows:

\[ \frac{\partial J_c(q)}{\partial a} = \frac{\partial J_c(q)}{\partial s} \frac{\partial s}{\partial a} + \frac{\partial J_c(q)}{\partial \lambda} \tag{8} \]

which leads to

\[ V^T (W(s - q)) + \lambda I_n a = 0 \tag{9} \]

and with \( s = V a + m \) we arrive at

\[ V^T (W(V a + m - q)) + \lambda I_n a = 0 \tag{10} \]

leading to

\[ (V^T V + \lambda I_n) a = V^T W(q - m) \tag{11} \]

or, with \( U^T = V^T W \) we get the final expression given in Eq. (7),

\[ (U^T V + \lambda I_n) a = U^T (q - m) \tag{12} \]

Finally, the class label of the query q is assigned as:

\[ f(q) = \arg \min_c J_c(q) \]

The designing of the “local hyperplanes” in the ALH algorithm given above goes as follows. First, they are computed only for the classes which members have been selected as the NNs around q. A tiny example is introduced here to explain the nature of the local hyperplanes. Say, a training data set has 9 different classes with 10 input features and \( K = 12 \). The 12 NNs to q are distributed as follows: 3 in class 1, 4 in class 3, 2 in classes 5 and 6, and 1 in class 7. Then, the following local hyperplanes are interpolated in the 10-dimensional feature space: the first one through 3 data points from class 1, the next one through 4 data points in class 3, the third local hyperplane (which is now a straight line) through 2 data points in class 5, and again one straight line is drawn through 2 data points in class 6, and finally the last “local hyperplane” is now a point coinciding with a single data point in class 7. Next, the parameter vectors \( a \) (which define the closest projection points \( s \) to q) are computed for each local hyperplane, the nearest points \( s \) on the local hyperplanes are found and the distances between q and all the s points are determined. The query q is assigned to the class for which the distance to its s is the smallest. Note that in this example classes 2, 4, 8, and 9 are eliminated as the candidates due to the fact that none of its member has been within the 12 NNs to q.

A graphical illustration of the ALH algorithm is introduced in Fig. 1 where a classification problem with two features and three classes is sketched, the query q will be assigned to the class 3 because \( d_3 < d_1 \) and \( d_3 < d_2 \) (\( d_1, d_2, \) and \( d_3 \) refer to the weighted distances between the query and the local hyperplanes for each class). Note that the user-friendly software for implementing the ALH algorithm can be downloaded by
clicking here. This software can be freely used for all noncommercial purposes. If utilized, the reference to Ref. [7] would be appreciated.

2  ALH-Fast Algorithm

In the ALH algorithm, the distances between the query to all the training instances need to be computed online and the whole training data set needs to be stored in memory. Hence, the major computational deficiencies of the ALH algorithm are in its testing speed and storage requirement. Unlike the testing speed, the training speed of ALH can be very fast because only the feature weights need to be computed by using the RBWSS criteria. However, the testing speed is much more important than the training speed in real-life applications of the classification tools because the testing phase is done online. These deficiencies of ALH will not be an important issue for the small-to-medium sized data sets with a modern computer. However, it will be very beneficial to resolve the above-mentioned problems for ALH on very large data sets. The proposed ALH-Fast algorithm tries to reduce the computational load of ALH by combing the classification tree and ALH algorithms.

The classification tree (CART) algorithm \cite{10} partitions the feature space into a number of mutually exclusive cuboid regions, whose edges are aligned with the feature space axes. The feature space is sequentially split into two subregions based on the values of one feature input corresponding to the traversal of a binary tree. In order to determine the structure of the tree, the choice of feature for splitting as well as the threshold value for the corresponding split are determined by the exhaustive search so as to minimize some splitting criterion. Here, we choose to use the Gini index as the splitting criteria. The Gini index for node $t$ is defined as

$$G_t = \sum_{c=1}^{C} p_c (1 - p_c)$$

(13)

where $p_c$ is the proportion of instances within node $t$ assigned to class $c$. If all the instances within node $t$ are from the same class, then this node is said to be a pure node. It can be seen that $G_t = 0$ for a pure node because $p_c = 1$ for $c = c'$ and $p_c = 0$ for $c \in \{1, \ldots, C\} \setminus \{c\}$. It can be seen that the splitting criteria of the Gini index encourages the splitting which allows for a high proportion of instances assigned to one class.

If all the instances within node $t$ are not from the same class, then this node is said to be an impure node. Impure nodes must have $N_s$ or more instances to be split, and the existence of impure nodes are used to avoid the problem of overfitting. Note that $N_s$ is a hyperparameter for the classification tree algorithm and it controls the size of the tree. If $N_s$ is large, then the size of the tree will be small; otherwise, the size of the tree will be large. The construction of the tree will be stopped when there is less than $N_s$ instances within each region and then the resulting tree is pruned based on a criterion that balances the sum of the Gini index and the number of the nodes (model complexity). Specifically, the pruning criterion for a subtree $H$ is defined as

$$P(H) = \sum_{t \in H} G_t(H) + \rho |H|$$

(14)

where $\rho$ is a regularization parameter which controls the trade-off between the classification performance and the model complexity, and the value of $\rho$ is determined by some sort of resampling techniques. The final subregions produced by the classification tree are called the “leaves” of the tree. The class membership of a leaf is assigned by using the majority voting scheme, which assigns the subregion to class $c$ if the number of instances within class $c$ is the greatest. When classifying the query, the leaf it falls into is determined by using the constructed tree and then the class membership of the query is determined as the same as that of the subregion.

In one perspective, the classification tree algorithm can be regarded as a combination of majority voting classifiers in which only one classifier is responsible for making predictions at a given point in feature space. The splitting is hard and the naive majority voting classifier will not be suitable for the data sets with complex structure. The basic idea of the ALH-Fast algorithm is to replace the majority voting algorithm with the ALH algorithm. In other words, the ALH algorithm is applied within every leaf of a constructed tree in ALH-Fast. Specifically, the tree is constructed first to split the training data set into several subregions (leaves). When classifying the query, the leaf into which the query falls will be found first, and then the ALH algorithm will be used to make the final decision for the query.
The combination of the classification tree and ALH algorithms can enhance the power of both individual algorithms. Firstly, the tree is constructed before the ALH algorithm is applied, so only the training instances within the leaf in which the query falls into are retained. Hence, in ALH-Fast, only a fraction of the training instances are retained and hence the amount of distance computation will be reduced and only part of the data is stored according to the size of the tree. Secondly, the ALH-Fast algorithm may also be even more accurate than the ALH algorithm because the irrelevant instances for classifying $q$ can be ignored at the beginning, and these irrelevant instances may interrupt the construction of the local hyperplanes in ALH. Thirdly, ALH-Fast is also expected to work better than the classification tree algorithm. It is not rare that there are still some information left in the leaves of the tree, but further splitting will over-split the data. Because the ALH algorithm has been demonstrated to perform better than many benchmarking classifiers for small data sets\cite{11}, it is reasonable to use the classification tree algorithm as the tool to mine the global structure of the data while the ALH algorithm as the tool to mine the local structure of the data.

3 Experiments

The proposed ALH-Fast algorithm is examined on two real data sets with a large number of instances. We have used the forest covertype data set and shuttle data sets from the UC Irvine Machine Learning Repository. The covertype data set consists of 581,012 instances with 54 features, and there are seven classes which need to be classified. The first 15,120 instances are used for training, and the rest of the instances are used for testing. The shuttle data set has already been split into training and test sets. The training set consists of 43,500 instances and the test set consists of 14,500 instances. There are 9 features and seven classes in the shuttle data set.

The classification accuracies and testing time for several classifiers are compared on these two data sets (CPU: Core (TM) 2, 1.86 GHz, RAM: 2.00 GB). The benchmarking classifiers LDA, SVM with Gaussian kernel, and ALH are also applied on the two data sets. We applied LDA by using the Statistics Toolbox in MATLAB, SVM by using the ISDA software written in C++\cite{12}, ALH by using our alh software written in MATLAB, and ALH-Fast by our own MATLAB routines. Note that the testing time comparison between SVM and ALH/ALH-Fast is not strictly fair because the ALH and ALH-Fast can be faster if the C++ language is used. There are hyperparameters in the SVM, ALH, and ALH-Fast algorithms, which we tuned by using the last 25% of the training set as a validation set. Both SVM and ALH have three hyperparameters: $C$ (the penalty parameter), $\sigma$ (the Gaussian kernel shape), and $b$ (the bias term) for SVM and $K$, $T$, and $\lambda$ for ALH. ALH-Fast has four hyperparameters $K$, $T$, $\lambda$, and $N_c$. Note that although ALH-Fast has one more hyperparameter than ALH, the hyperparameter tuning procedure for ALH-Fast is usually faster than ALH because the number of instances in the training set can be greatly reduced by using the classification tree.

The experimental results for the covertype data set are summarized in Table 1. Note that the results for the first five classifiers are borrowed from Ref. [13]. It can be seen that ALH-Fast achieves the highest testing accuracy among eight different methods. In particular, ALH-Fast is more than 11 times faster than ALH, the classifier with the second highest testing accuracy. Also, ALH-Fast is much more accurate and faster than the very popular SVM classifier. Here, the optimized hyperparameters are $C=10$, $\sigma=0.6$, $b=0$ for SVM, $K=8$, $T=2$, $\lambda=20$ for ALH and $N_c=720$, $K=7$, $T=3$, $\lambda=40$ for ALH-Fast. Note that ALH-Fast can be even faster than SVM if ALH-Fast is implemented in C++.

The experimental results for the shuttle data set are summarized in Table 2. ALH-Fast achieves the highest testing accuracy among four different classifiers. For the testing speed comparison, ALH-Fast is also much faster than ALH. Although ALH-Fast is slower than SVM here, the difference will be smaller if ALH-Fast is also implemented in C++. Interestingly, SVM, ALH, and ALH-Fast are all more accurate than the traditional LDA on two data sets. Note that a crucial difference between the ALH and SVM is that SVM, while using kernel functions, maximizes the margin in a virtual or feature space and not in the original input space, while ALH aims exactly at maximizing the margin in the original input space. Here, the optimized hyperparameters are $C=75$, $\sigma=0.32$, $b=0$ for SVM, $K=12$, $T=2$, $\lambda=0.1$ for ALH, and $N_c=40$,
4 Conclusions

In this paper, we propose a novel classification algorithm, ALH-Fast. This novel algorithm greatly reduces the computational load of the ALH algorithm while retaining the same level of classification accuracy. The ALH-Fast algorithm is developed by combining the ALH and classification tree algorithms in an efficient way. The experiments on two large data sets show that ALH-Fast performs better than ALH in both the testing time and classification accuracy.

References