Automated Learning and Adaptive Knowledge Systems

Homework

1D Learning

Roberto Valenti, 0493198

Introduction

The purpose and goal of this assignment was to design and implement a small system to predict a class of a single numerical variable in a binomial class problem. In order to do this, it was requested to implement or to think of one or more "1D classifiers", discussing their computational complexities in the learning phases, the performances on the given dataset, the differences and similarities with related or alternative methods.
This report is structured as follows: before describing the classifiers and the implementation details, there will be an introduction to the suggested dataset in order to understand some of the choices that have been made during the design and implementation phases.
Together with some of the theory that lies behind the implemented classifiers, the implementation choices and details will be discussed. The evaluation session will show some of the results obtained by the classifiers on the features of the dataset, comparing their performance and trying to understand the reasons behind their success or their failure.
Together with the meaning of the classifiers' evaluation, a more detailed discussion about the relation between the dataset and the classifiers over the assigned task will follow in the pertinent "Discussion" session, which will lead to the conclusions drawn from this assignment.

Dataset

The suggested dataset consists of 16 dimensions (columns) each describing a variable in the process of getting a loan. The two class labels (getting or not getting a loan) are described by the last column (16). The only usable (real-valued) attributes are on columns 2, 3, 8, 11, 14, 15, and some of these presents missing data (labeled as "?"). For the purpose of the assignment and the requested dimensionality, the missing data is pruned away during the loading phase. One alternative could be to place the value in the middle of the assigned class' distribution, but this solution won't affect considerably the outcome of the classifier. The idea is to use only one of the available attributes and try to have the best discrimination as possible. Of course, this depends on which kind of classifier is used and on the correct tuning of its parameters. A note on the available features is that they are dense and somewhat discreet in the real space. This means that the two classes are very close together (if not overlapping) and that instead of using the full real space, some of the data points have the same values and different class labels, which will cause some errors in all applicable classifier. Another characteristic of some feature is that are grouped by class. This is likely to cause some troubles during the classifier's validation, unless the data is shuffled before the testing procedure.

Implementation

The chosen programming environment is java. An easier choice could be Matlab given its build-in functions to do matrix operations, but by doing so a lot of information about the complexity of the algorithm is lost inside those pre-implemented functions, not giving a precise idea of the difficulties of the task.

An important part of the implementation choice is the use of the "Classifier" java interface which connects all the implemented classifiers as a single classifier allowing to perform the same operation on all the different classifiers, using them in a single and generic class. Through this interface it is possible to easily add new classifier to the implementation, allowing future extension of the full program.
The "Classifier" java interface enforces the following fields in each implemented classifier:

- **test** (testdata);
- **classify** (data);
- **crossvalidate** (folds);
- **findparameter** ();
- **getdata** ();

On the basis of this skeleton, the following classifiers were implemented:

- **Linear;**
- **K-nearest neighbors;**
- **Parzen;**
- **Adaboost;**
- **Distance Majority;**

The data operation such as load, initialize weights, split, sort, shuffle, distance from point etc... are all grouped in the "Data" class so that is easy to distinguish between data operation and classifiers operations.
Following, some implementation details on each implemented classifier are discussed.

Linear

This is the easier and most straight forward classifier that could be built on the given task.
The theory behind this classifier can be related to the results of a Bayesian learner [1] [2]: given two classes, we can find a Gaussian distribution for each of them. When a new unlabeled data point is tested, a maximum likelihood of the point being generated from one of the distribution will assign its class label. In a two-dimensional environment, it is possible to draw a line (linear or curve) between the two classes where the class likelihood is equal, which can represent the discriminant function between the two classes (i.e. each new unlabeled points above this line will be classified as class one and the ones under it as class two). In the 1D version, this threshold is a dot. The implemented algorithm searches for this threshold in the dataset, positioning it between each possible data point in a sorted sequential manner. Some might argue that finding the density model using the mean and covariance and have a search for the decision boundary can perform computationally better (especially in big datasets) than the used search method. In this case the dataset is not so big, and the chosen method is trivial to be used for boosting.

K-nearest-neighbors

The original algorithm described in [1] and [2] was readapted for the given task. In fact, since there is only one dimension, the Kth nearest neighbor can be easily found by ordering the data per class, taking the Kth element of this array. If the Kth element of class 1 is closer than the Kth element of class 2, it means that class 1 is more dense near that point, so the new point can be labeled as class 1 (and vice – versa)
This classifier requires a parameter K, which is the number of near neighbors to consider. For the purpose of this assignment the parameter is searched by sequentially trying all of the possible values (1,..,N ,N = size of the dataset). This process is computationally expansive since N classification of the full dataset has to be made in order to find the value that gives less error as possible. The only heuristic used to reduce the computation time is the assumption that the best value of K will be at most half of the size of the dataset ($1<K<N/2$). Another important assumption was made about the classifier's parameter: the K that performs the best on the training set is likely to perform well on the test set.

Parzen

This estimator is a kernel method [1] [2] which work by building a distribution (or kernel) over each labeled data point (Figure 1) and given an unlabeled data point evaluates its class likelihood by summing those distributions (Figure 2). A key factor of this method is choosing the correct smoothing parameter for the distributions: a very small smoothing parameter will result on a line on each point in the dataset (overfitting the data) while a big smoothing parameter will loose information about important characteristics of the data.
Since it is very difficult to find the correct parameter by trial and error, a parameter estimator was implemented. The implementation consists in finding the distances between samples and their kth nearest neighbors (in this case k=20) and use this has the smoothing parameter (Jones et al. [4]). Of course, more sophisticated methods could be used to define a specific smoothing parameter dependent of the position in the space [1].

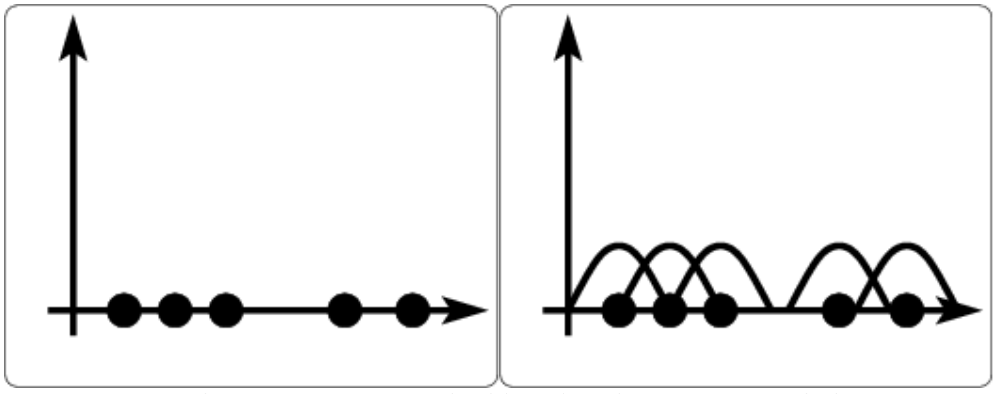

***Figure 1**: The parzen estimator builds a distribution over each data point*

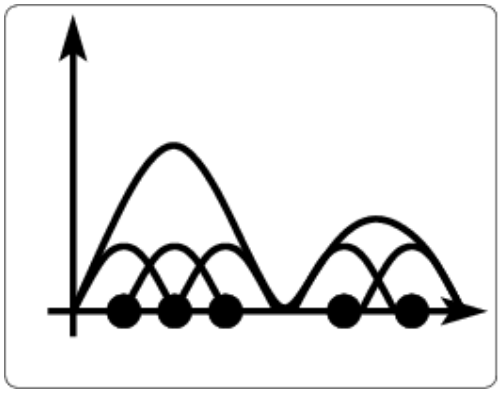

***Figure 2**: The parzen estimator sums the distributions to estimate the full distribution of the data.*

Adaboost

The implementation of this algorithm is consistent with the one described by Freund and Schapire in [3]. The only task influence in the implementation is the choice of the weak learner. In fact, since the algorithm should deal with a single dimension, a good choice of a weak classifier is the linear discriminant discussed before. The linear discriminant searches for the decision boundary instead of deriving it from the distribution, which allows for an easy search without considering and tuning the parameters of the data distributions. The combined strong classifier will partition the instance spaces and a new instance is tested on majority voting. The resulting partition could easily be liked to decision tree learning.

Distance majority

This is a variant of the distance based K-nn algorithm that was inspired by [2] and implemented from scratches. The concept of this classifier is very easy: given a labeled dataset with label L={-1,1} and an unlabeled data point, the algorithm calculate the distances between the unlabeled point and the each of data point in the dataset.
For the N closest point, the algorithm sums the multiplication of the label and the distance
This majority voting [2] will output a number. If this number is bigger than 0, than there are major votes for class 1, and the element should be classified as class "1" ("-1" otherwise). The only parameter needed is the distance to consider, and this is found with the same procedure described for the K-nn algorithm.
In this algorithm, the weight of a label is defined by its distance from the data point. The first version of this algorithm used the inverse of the distance as a weight for the label but, surprisingly, assigning the weight proportionally to the distance of the data gave better performances.

Interface

To better visualize and play with the parameters of the implemented classifiers, a graphical interface was added to the project. This interface allows the user to choose the favorite classifier, load data and labels (even form other files than the given dataset), find and tune the parameters of each classifier, cross-evaluate its performances, display the chosen dataset, classify a single element, decide whether or not to shuffle the dataset etc.. (Figure 3)

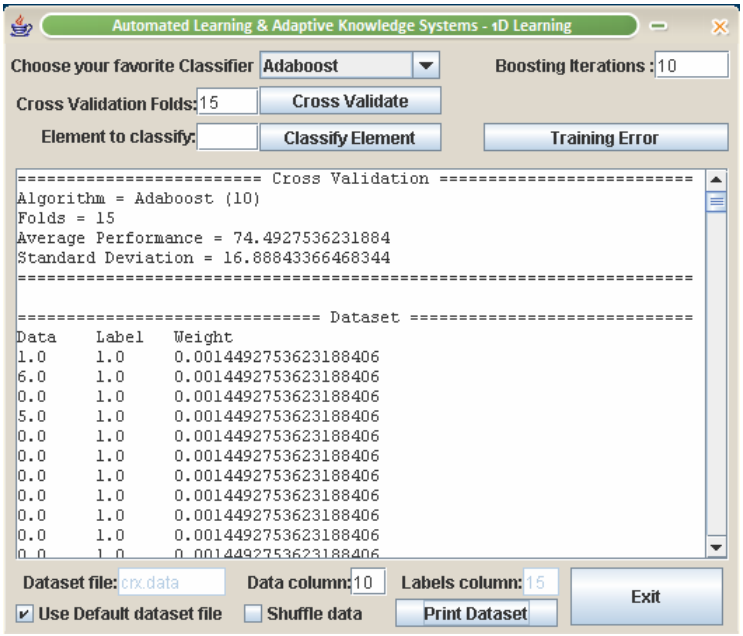

***Figure 3**: The 1D-learner interface on Adaboost*

The interface is designed to help the user choosing the correct action for the current program state. For instance, in figure 3 (Adaboost) there is no way to look for the boosting iteration automatically, while in figure 4 (Parzen) the name of the classifier's parameter changed to "Smoothing" and a special button appeared to give the possibility to estimate the best smoothing parameter for the dataset.

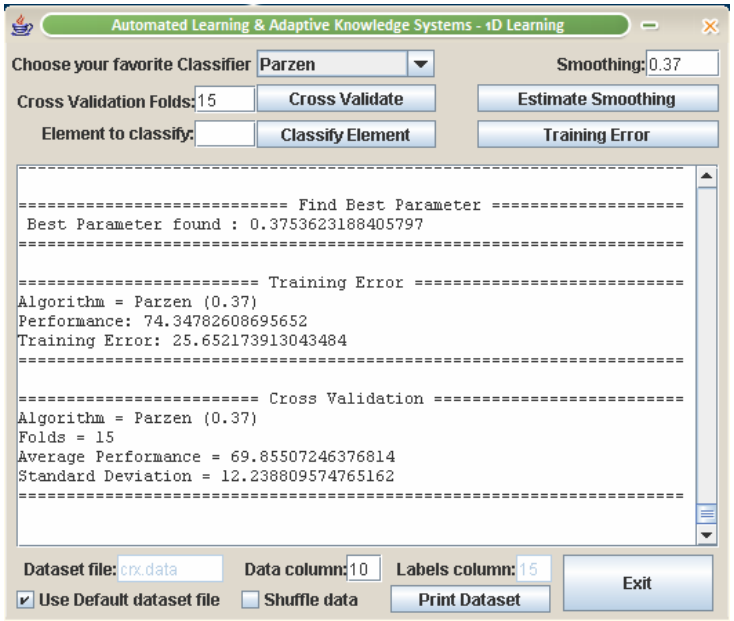

***Figure 4**: The 1D-learner interface on Parzen*

In the interface's console, the user can easily compare outputs of different classifiers with different parameters and feature, since the output contains all the relevant information.
Furthermore the interface notifies the user when an entered value is invalid, and warns for computationally expansive operations requested by the user.

Evaluation

In order to evaluate the different classifiers on the different data, an n-folds cross validation was implemented. In n-folds cross validation, the dataset is shattered in n parts and one of them is choose to be used as test set, while the rest n-1 parts are used for training. By repeating the process n times and selecting each time one different part for testing and the other n-1 for training, all the values on the original training set participate exactly once as a part of the test set and n-1 times as part of the training set. Keeping track and averaging the error over all the n iterations gives a good estimate on the performance of the selected classifier over unseen data, and shows whether or not the classifier over-fitted the data on the training set.

Letting the parzen estimator overfit the data by choosing a very small smoothing parameter, it is possible to define a boundary which represent an estimation of the best performance obtainable on the specified dataset. The rationale behind this claim is that an overfitted parzen estimator will have a training error of 0% in the case that the classes are completely separable (two data points belonging to different classes cannot have the same numerical value). This is not the case of the used dataset, meaning that even the best classifier will always have some errors. As stated previously, the parzen with a very small smoothing parameter builds a peak on top of each datapoint. By using the training set as the test set, each point is classified using the class with the highest sum of peaks, generating misclassification for the other class. Note that this procedure doesn't give the perfect estimation of the maximum performances obtainable, but does give a good evaluation standard to compare the other implemented classifiers.

In this view, table 1 represents the estimation of the best performances obtainable on the all possible (numerical) features of the used dataset. The numeration starts from 0, so the feature one is represented in the second column of the dataset.

Feature 1	Feature 2	Feature 7	Feature 10	Feature 13	Feature 14
~80	~72	~73	~74	~71	~78

***Table 1**: The Performance boundaries on the different features*

Table 2 represents the results obtained by all the classifiers on all the possible features of the dataset, in 15-folds cross validation with randomly shuffled data. Shuffling the dataset was required by the nature of the data itself, which is slightly grouped by class label. In this case the cross validation procedure could strip out a lot of important class feature. Shuffling the dataset improved both the average performances and the standard deviation.

On each field, the first value represents the classifer's performance, the second its standard deviation and the bracketed value (where applicable) represents the value of the used parameter. Since a method to estimate the number of iteration required by Adaboost was not implemented, the table displays the results after 100 iterations, 1.000 iteration and 10.000. Of course 10.000 iterations are a lot more than the required iteration in order to obtain the same results but, by doing so, it ensures the algorithm's convergence. An important note is that all the classifiers used in this evaluation didn't use random guessing in uncertain situations (for instance when a new data point has the same value of the discriminant point) but it assigned an arbitrary class (always "-1") allowing the comparisons of the results with the defined performance boundaries, without considering the "luck" factor.

	Feature 1	Feature 2	Feature 7	Feature 10	Feature 13	Feature 14
Linear	59.85/5.9	62.75/6.4	69.56/7.4	74.49/5.4	60.14/8.8	69.56/4.7
K-nn (K)	52.44/6.4 (1)	57.53/7.6 (5)	69.56/6.8 (18)	73.18/8.3 (1)	60.14/9.0 (16)	69.42 6.3 (16)
Parzen (smoothing)	57.33/6.4 (0,91)	61.73/7.3 (0.39)	68.98/7.2 (0.30)	72.75/4.5 (0.37)	62.96/8.3 (13.90)	66.23/7.2 (549.18)
Adaboost (100)	64.00/8.3	63.76/7.2	69.56/6.7	74.49/7.0	63.40 8.1	70.72/7.4
Adaboost (1K)	68.44/8.2	65.65/5.8	72.02/5.3	74.49/5.8	65.62 7.0	72.02/6.5
Adaboost (10K)	74.51/5.8	72.75/3.6	73.76/7.6	74.49/4.3	71.11 6.8	78.26/6.2
Distance (distance)	57.62/7.1 (28)	58.69/3.8 (61)	68.11/5.5 (83)	74.49/4.8 (27)	58.51/6.0 (44)	69.42/5.6 (16)

***Table 2**: Results of the experiments with all the classifiers over all the features*
Format: "perf/std (param)"

Discussion

Looking at the obtained results it is possible to note that almost all classifiers perform decently on feature 10. In fact, the linear discriminant performs as good as all the other classifiers on this feature, which makes it the most discriminative feature. The fact that overall performances of the Linear discriminant are the same as the K-nn (or even slightly better) can be seen as a proof that the two classes are merged together in almost every the features.
Compared with the poor performances of the K-nn algorithm, the "Distance Majority" performs slightly better on few of the data features. The main problem of the latter is that it overfits the training data so that it is very difficult to find the correct parameter that will be valid when testing with unseen data. An adaptive version of the classifier could be considered to split the instance space in different parts, each of which can have its own parameter. This procedure could be applied to the parzen estimator as well, which has similar performance of both the previous discussed classifiers.
Sadly, the linear discriminant performs overall better and is faster and easier to implement than the K-nn, the parzen and the Distance Majority algorithm. Considering this, the boosted linear discriminant is the classifier of choice for the given task. In fact, by looking at the results, Adaboost always reaches the defined performance boundary (beside on feature 1) after a considerable amount of iterations. This result was expected since it is a property of Adaboost to reduce the training error and even the generalization error (it is possible to see this on feature 10) to a minimum. The only problem of the algorithm is the long training time if compared with the other implemented classifiers. If this is not relevant for a task (i.e. the task doesn't require on-line training) than this classifier should give the maximum performance obtainable on the data. The only way to further improve the performances is to increase the dataset maximum performance (table 1) by performing some kind of operations with the data. An experimented idea was to "filter" the data by assigning all weights to the instances and to group same-valued class instances in only one instance with the sum of the weights. This should preserve the weight distribution on the data and should allow for faster classifiers. This filtering procedure worked as well as the unfiltered data but it wasn't compatible with all the classifiers, so it was removed from the final version (is still kept in the source code as proof of concept). Another possible data filtering can be done by combining more features together or by completely removing overlapping class instances, letting the algorithms decide how to assign the class label on these uncertain instances just by looking only at the error-free instances.

Conclusions

A full application was developed in order to easily test the performances of the family of implemented classifiers. One of the implemented algorithms proved to be very efficient and to obtain the maximum out of the suggested dataset. The best overall performance was 78.26% obtained by Adaboost on the 14^{th} feature (marked in red in table 2), while the most easy-to-discriminate feature appeared to be the 10^{th} feature, where all algorithms have the same performance. The other algorithms seemed to fail in all other features, maybe due the nature of the dataset. In fact, to further improve these results, one should think about performing some operations on the data prior to the learning phase. Some ideas were suggested, but not implemented, so this is left as a future work. As a final remark, one should think about what is the tradeoff between a straightforward implementation of a linear discriminant on feature 10 and a more computational training of Adaboost on feature 14 in order to improve the results of just 4%. Of course the answer to this is task depended, and for the given task (getting or not getting a loan) Adaboost on feature 14 should be the best choice.

References

[1] Andrew Webb, *"Statistical Pattern Recognition",* Wiley, 2002.

[2] Tom M. Mitchell, *"Machine Learning",* McGraw-Hill,1997

[3] Yoav Freund and Robert E. Schapire, *"A short introduction to boosting"*, Journal of Japanese Society for Artificial Intelligence, 14:771-780, 1999.

[4] Jones, M.C., Marron, J.S., and Sheather, S.J. *"A brief survey of bandwidth selection for density estimation"*. Journal of the American Statistical Association, 91:401-407, 1996