Random Feature Subset Selection for Ensemble Based Classification of Data with Missing Features

Joseph DePasquale and Robi Polikar*

Signal Processing and Pattern Recognition Laboratory, Electrical and Computer Engineering, Rowan University, 201 Mullica Hill Rd, Glassboro, NJ 08028 USA
depasq63@students.rowan.edu, polikar@rowan.edu

Abstract. We report on our recent progress in developing an ensemble of classifiers based algorithm for addressing the missing feature problem. Inspired in part by the random subspace method, and in part by an AdaBoost type distribution update rule for creating a sequence of classifiers, the proposed algorithm generates an ensemble of classifiers, each trained on a different subset of the available features. Then, an instance with missing features is classified using only those classifiers whose training dataset did not include the currently missing features. Within this framework, we experiment with several bootstrap sampling strategies each using a slightly different distribution update rule. We also analyze the effect of the algorithm's primary free parameter (the number of features used to train each classifier) on its performance. We show that the algorithm is able to accommodate data with up to 30% missing features, with little or no significant performance drop.

1 Introduction

One of the most frustrating problems encountered in field implementation of an automated decision making system is to get caught unprepared for partial loss of data. Unless it was designed to be robust against such a potential loss, there is nothing a classifier can do when faced with processing a data instance with missing components. The partial loss of field data need not even be catastrophic: e.g. if a single sensor malfunctions (loss of one feature) during data collection, the entire data cannot be processed by such classifiers. This problem is hardly rare: bad sensors, failed pixels, malfunctioning equipment, signal saturation, data corruption, etc. are all familiar scenarios in practical applications.

The missing feature problem has been well researched. The oldest, and perhaps most commonly used solution is to substitute a meaningful estimate of the missing value, such as the k-nearest neighbors of the missing value [1]. However, such *data imputation* techniques require that the training data be sufficiently dense for the estimate to be a faithful representative of the missing value. Such a requirement, however, is rarely met in practice, even for datasets with modest number of features. Other approaches with sound theoretical underpinnings are also available that provide

* Corresponding author.

M. Haindl, J. Kittler, and F. Roli (Eds.): MCS 2007, LNCS 4472, pp. 251–260, 2007.

precise performance guarantees under certain conditions. These are typically probabilistic approaches, based on density estimation. Therefore, they require that certain a priori knowledge regarding the underlying data distributions be known or estimated, which also requires a sufficiently dense database. Such knowledge is often vague or non-existent, and inaccurate choices may lead to inferior performance, particularly for large datasets. Classical Bayesian estimation and expectation maximization based techniques fall into this category [2].

Other approaches use neuro-fuzzy algorithms, which provide, perhaps, a more natural setting for dealing with the missing data. In such approaches, unknown values of the data are either estimated, or a classification is made using the existing features, by calculating the fuzzy membership of the data point to its nearest neighbors, clusters, or hyperboxes. The parameters of the clusters and hyperboxes are determined from the existing data points. Algorithms based on general fuzzy min-max neural networks [3], or ARTMAP and fuzzy c-means clustering [4] are examples of this approach.

More recently, ensemble based approaches have been proposed to address the missing feature problem. For example, Melville *et al*. show that the algorithm DECORATE, which generates artificial data (with no missing values) from existing data (with missing values) is quite robust to missing data [5]. On the other hand, Juszczak and Duin [6] propose combining an ensemble of one-class classifiers trained on a single feature. This approach is capable of handling any combination of missing features, with the fewest number of classifiers possible. The approach can be very effective so long as single features can reasonably estimate the underlying decision boundaries, which is not always plausible.

We have previously proposed an alternative approach, Learn^{++}.MF, that trains multi-class classifiers using a random subset of the feature space, where the number of features is a free parameter of the algorithm. Any instance missing a feature is then classified as the majority vote of those classifiers whose training data did not include the missing features. This approach essentially combines the random feature selection in random subspace methods [7], with the distribution update rule of AdaBoost inspired Learn^{++} [8]. The original algorithm built on this premise was crude, but could handle up to 10% missing data using a specific feature distribution update rule [9]. In this contribution, we formalize the algorithm, and extend our work by i) analyzing the effect of different update rules; ii) analyzing the effect of the algorithm's primary free parameter, the number of features used in each subset; and iii) evaluating the algorithm with up to 30% missing features. These analyses provide us with informative clues on how such parameters should be selected, and under which conditions the algorithm can be expected to perform well.

2 Learn^{++}.MF

Ensemble of classifies hints at a trivially intuitive approach for the missing feature problem that can guarantee a reasonable performance for any number and any combination of missing features: simply create one (or more) classifier(s) with every possible combination of the available features, and use those classifiers whose training

features did not include the missing ones. This exhaustive approach is of course becomes practically impossible even for a modest number of features, as the number of classifiers grows exponentially with the number of features. However, the probability of a particular feature combination being missing also diminishes exponentially as the number of features increase. Therefore, trying to account for every possible combination is hardly an effective use of computing resources. On the other hand, Juszczak and Duin's approach, training one class-classifiers trained with each feature separately, sits on the other end of the spectrum, and offers the fewest number of classifiers that can handle any feature combination (at a cost of potential performance drop due to single feature training).

Learn^{++}.MF, recognizing the inefficiency of trying to accommodate every feature combination as well as the difficulty of obtaining a good classifier using a single feature, offers a compromise: it trains an ensemble of classifiers with a random subset of the features, where the number of features is a free parameter of the algorithm. It also uses an iterative distribution update rule so that feature combinations not previously accounted for are more likely to be selected next (see, however, the effect of using different update rules in Section 3).Doing so, Learn^{++}.MF can achieve classification performances with little or no loss (compared to a fully intact data) even when large portions of data are missing.

The pseudocode of the algorithm is given in Figure 1. The inputs to the algorithm are the training data, a supervised classification algorithm, a sentinel value *sen* to represent missing values, the number of classifiers to be generated, T , and the number of features (*nof*) to be used to train each classifier. The algorithm maintains a distribution $\boldsymbol{D}$ over the features to determine which features should be more likely to be selected next. This distribution is initialized to be uniform so that each feature has equal probability of being selected to train the first classifier. During the t^{th} iteration, the algorithm draws a random bootstrap sample of *nof* features according to then current feature distribution $\boldsymbol{D}_t$. The indices of these features are then placed in a list, called $\boldsymbol{F}_{selection}(t)$. This list allows the algorithm to keep track of which features have been used for each classifier, so that appropriate classifiers can be called during testing depending on the then available set of features. Classifier C_t, is trained using the features in $\boldsymbol{F}_{\text{selection}}(t)$.

The distribution $\boldsymbol{D}_t$ is then updated by reducing the weights of those features that have just been used. This gives other features a better chance to be selected into the next feature subset. T such classifiers are iteratively generated, each using a different subset of *nof* features. In this current version, we do not check the performance of each classifier, since the distribution update rule is applied to features, and not to actual training data instances (as in AdaBoost). Interleaving two distribution update rules, one on features and one on training data, is however being considered as future work.

During testing, a given instance $\mathbf{z}_i$ is first checked for its missing features $\boldsymbol{M}(i)$, which were previously flagged with a sentinel. The algorithm then cross-checks the features in $\boldsymbol{M}(i)$ against those available in $\boldsymbol{F}_{\text{selection}}(t)$ for each classifier C_t. Classifiers whose feature lists do not include any of those in $\boldsymbol{M}(i)$ are then combined using majority voting.

Inputs:

- Sentinel value *sen*.
- **BaseClassifier** and the number of classifiers, *T*.
- Training data set $\boldsymbol{S}=\{(\mathbf{x}_i, \mathbf{y}_i) \mid i=1,\ldots,N\}$ with *N* instances with *f* features, each.
- Number of features used to train each classifier, *nof*.

Initialize $D_1(j) = 1/f, \forall j, j = 1, \cdots, f$ (1)

Do for *t = 1, 2,..., T*:

1. Normalize $\boldsymbol{D_t}$ so that it is a proper distribution**.**
2. Draw a bootstrap sample of *nof* features from $\boldsymbol{D_t}$ to be placed in $\boldsymbol{F}_{selection}(t)$
3. Call **BaseClassifier** to generate classifier C_t using features in $\boldsymbol{F}_{selection}(t)$
4. Update the distribution $\boldsymbol{D_t}$ using a suitable update rule

Validation / Testing

Given test data $\boldsymbol{Z}=\{\mathbf{z}_i\}$, i=1,…,*M*

Do for *i = 1,2,...,M*:

$$\boldsymbol{M}(i) = \left\{\arg\left(\mathbf{z}_i(j) == sen\right)\right\}, \ \forall j, j = 1, \cdots, f \ . \quad (2)$$

$$\boldsymbol{\mathcal{E}}(\mathbf{z}_i) = \arg\max_{y \in Y} \sum_{t:C_t(\mathbf{z}_i)=y} \left[\!\left[\left(\boldsymbol{M}(i) \cap \boldsymbol{F}_{selection}(t)\right) = \varnothing\right]\!\right] \quad (3)$$

Fig. 1. Pseudocode of the Learn^{++}.MF algorithm

The original distribution update rule we had previously used simply reduced the distribuion weight of each feature by a factor f , where f is the total number of features. While such a distribution update rule certainly makes intuitive sense, it is possible to devise other rules as well. For example, is 1/*f* the correct scaling factor to reduce the feature weights? How about *1*/*nof*, since the actual number of features used by each classifier is *nof*? Or *f*/*nof*, as the ratio of the two? Or does it really matter to have a database specific update rule? How about using a strict bootstrap sampling from a uniform distribution? The different distribution update rules can all be represented as in Equation (4), where the parameter β is one of 1/*f*, 1/*nof* or *n*/*nof*.

$$D_{t+1}\left(F_{selection}(t)\right) = \beta * D_t\left(F_{selection}(t)\right) \quad (4)$$

The algorithm was implemented with the above mentioned update rules, which was also compared to uniform sampling. Furthermore, note that the primary free parameter of Learn^{++}.MF is the number of features *nof* used to train each classifier (and to a lesser extend, the number of classifiers *T* to be generated). In order to determine the effect of *nof* on algorithm behavior, several values of *nof* were also evaluated for each database tested.

3 Experiemental Results

We present the implementation results on three datasets: Wisconsin Breast Cancer and Wine databases from UCI [10], and the real-world volatile organic compound

(VOC) identification database. The wine database was selected for its similarity to the VOC database in terms of its feature size. This allows to check for repeatability on datasets of similar size. Missing features were simulated by removing a certain percetange (% missing features – PMF) of values from the entire dataset and replace them with a sentinel. PMF was varied from 0 to 30%. All results are averages of 10 indepent trials, each randomly splitting the training and test data into two equal partitions.

Table 1 shows the total number of features *f*, the different values of *nof*, and the total number of classifiers generated (*T*) for each database. Note that for each chosen value of *nof*, the ensemble can accommodate up to *f-nof* features missing at a time. Hence a classifier trained with, say 3 of 12 features, can accommodate *any and all* of 1 through 9-way combination of the remaining 9 features being missing. This is how Learn^{++}.MF avoids using prohibitively large number of classifiers.

Table 1. Number of features (*nof*) and classifiers (*T*) used for each dataset

Dataset	*f*	nof_1	nof_2	nof_3	nof_4	Nof_4	*T*
VOC	12	3	4	5	6	---	200
WBC	30	10	12	14	16	---	1000
WINE	13	3	4	5	6	7	200

3.1 VOC Database

The VOC database consists of the responses of 12 quartz crystal microbalances type chemical sensors to 12 volatile organic compounds (VOC), including toluene, xylene, hexane, octane, methanol, tricholoroethylene (TCE), among others. Figure 2 illustrates the performance of the algorithm in four rows, one for each distribution update rule of *nof/f*, *1/nof*, *1/f*, and *uniform selection* respectively. For each update rule, we provide two plots, classifier performance with respect to percent missing features (PMF), and the percent instances processed (PIP – explained below) with respect to PMF. Performances for different values of *nof* are indicated using different line styles on each plot.

From the ensemble performance plots (on the left of Fig. 2), we make the following observations. First and foremost, as expected, there is a general decline in the ensemble classification performance as the percentage of missing features increase. However, this decline is very mild, the worst case being from 96% to 93%. In most cases, the differences are not even statistically significant. Hence, the algorithm can easily handle as much as 30% (perhaps even higher, but not tested yet) of missing data with little or no performance drop. Second, algorithm seems to do better with the first two distribution update rules, however, the differences were only significant for certain *nof* values.

The performance plots tell only half of the story, however. We also need to consider the amount of data that can be processed by the algorithm, on which *nof* has a

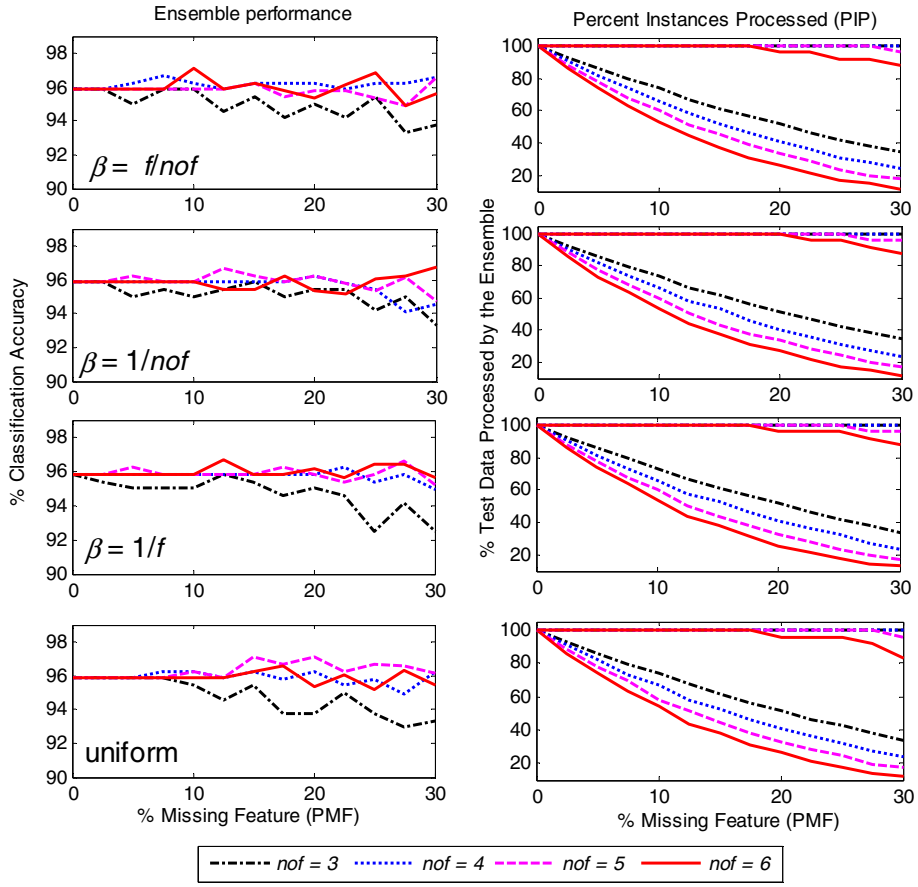

Fig. 2. Learn^{++}.MF performance on VOC database

dramatic impact, which is even more pronounced on higher dimensional data. As previously mentioned, Learn^{++}.MF does not guarantee that all possible combinations of features can be accommodated as missing data. Since features are selected at random, there may be certain feature combinations not represented by any of the classifiers trained in the ensemble. Instances with those exact feature combinations cannot be processed by any of the classifiers in the ensemble, and hence cannot be processed. Learn^{++}.MF attempts to minimize the number of such instances. The plots on the right side of Figure 2 provide a graphical representation of this issue, as it plots percentage of instances that can be processed, for different PMF values. Note that two sets of plots are given: the group of curves showing an exponentially decaying characteristic on the lower side of the plot represent the average PIP if we were to use a single classifier, averaged over 10 trials. The group of curves on the upper side of the plot show

the PIP when the Learn^{++}.MF ensemble is used. Notice that a substantially larger percentage of instances can be processed by the ensemble.

More interesting to notice, however, is the impact of the *nof* parameter on PIP. The figure indicates that the smaller the *nof*, the larger the PIP. In fact, for *nof*=3 and *nof*=4, PIP was 100% even for with 30% of the data missing. For *nof*=6, PIP was 100% for up to 20% missing data, but dropped to 85% - 87% for 30% missing data. This observation makes sense: when a large number of features are used for training, then so many features are required at the time of testing, and hence fewer missing features can be accommodated.

3.2 Wine Database

The wine database consists of 13 features of various chemical analysis results to predict three types of wine origins. This database was used due to its similarity in size to the VOC database (both in terms of number of features, and total data size), and same number of classifiers (200) was generated. This allows us to test for the repeatability of the algorithm's performance behaviors and trends over different datasets of similar size. The results, formatted similar to that of VOC dataset are illustrated in Figure 3.

Tested using five different values of *nof*, the algorithm shows very similar performance and behavior trends on this database, as it did for the VOC database. Specifically, we note that the ensemble performance is far more resistant to missing features when fewer *nof* values are used. For example, when only 3 or 4 features are used, there is no statistically significant performance drop even when 30% of data are missing, particularly for the first two distribution update rules of *f*/*nof* and 1/*nof*. There is some performance drop when larger values of *nof* are used, however, these are very modest. We also see similar behavior with the percentage of data that can be processed. The ensemble can process 100% of the data for *nof*=3 and *nof*=4, even when 30% of the data are missing. For *nof*=7, PIP drops to about 80% when 30% of the data are missing. In all cases, however, the algorithm – on average – is capable of processing substantially larger portion of the data than a single classifier can process alone. Finally, we do observe that the algorithm performs marginally better using the *f*/*nof* and 1/*nof* update rules than the original 1/*f* and the uniform distribution; however, the differences are rarely statistically significant.

3.3 WBC Database

The WBC database includes 30 features used as a diagnostic biomarker for distinguishing between benign and malignant tumors. Many of the trends observed for the VOC and Wine data can also be seen on this database, which has a larger feature size. Specifically, we observe in Figure 4 that the algorithm can accommodate up to 30% missing features with practically no loss of performance for *nof*=10, and only about

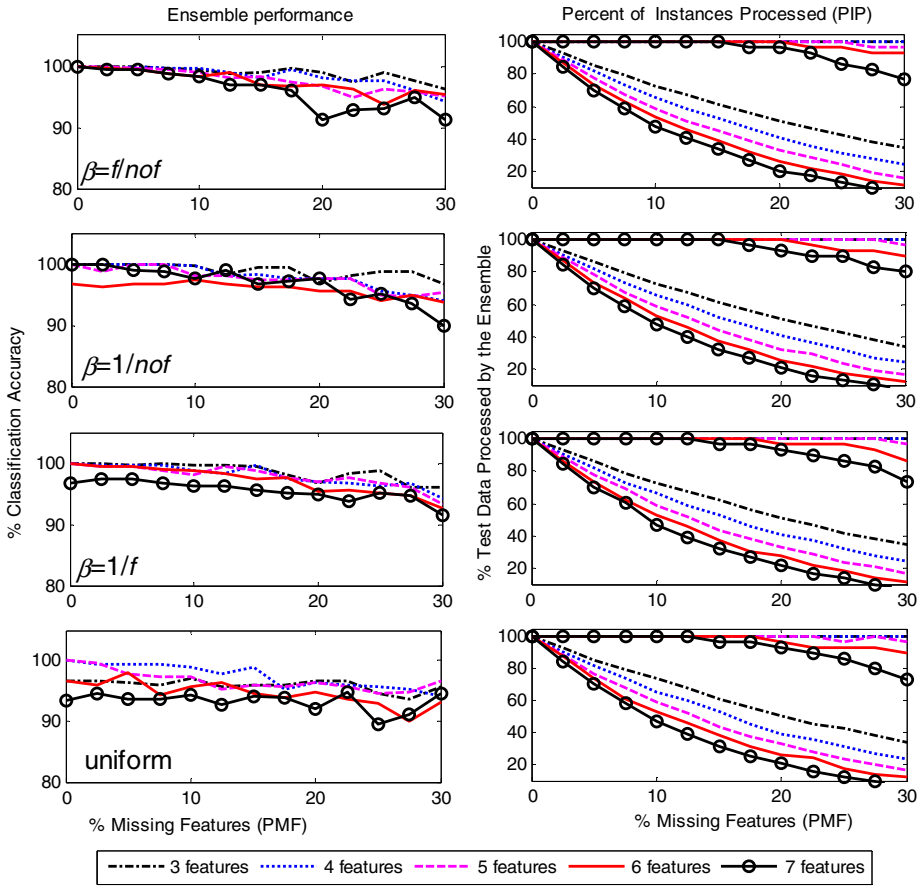

Fig. 3. Learn^{++}.MF performance on WINE database

7% loss for *nof*=16. Hence the selection of *nof* once again shows its impact: there is less of a performance drop for larger PMF values, when fewer features are used for training.

In other words, using fewer features during training, if adequate to model the data, generates an ensemble that is more resistant to missing data.

The PIP plots also display a now familiar trend: using fewer features in training not only gives better performance (as seen on performance plots), but also allows the ensemble to classify a larger percentage of instances as amount of missing data increases. Furthermore, while the drop in PIP is negligible up to 15% missing data for all values of *nof*, the differences becomes more substantial at higher PMF values, due to larger feature size of this database. In fact, with *nof*=16 features, the PIP displays a steep drop from 100% for a PMF of 15% to about 35% for a PMF of 30%. On the other hand, when fewer features are used for training, for example *nof*=10, PIP barely drops a couple percentage points even for 30% missing data.

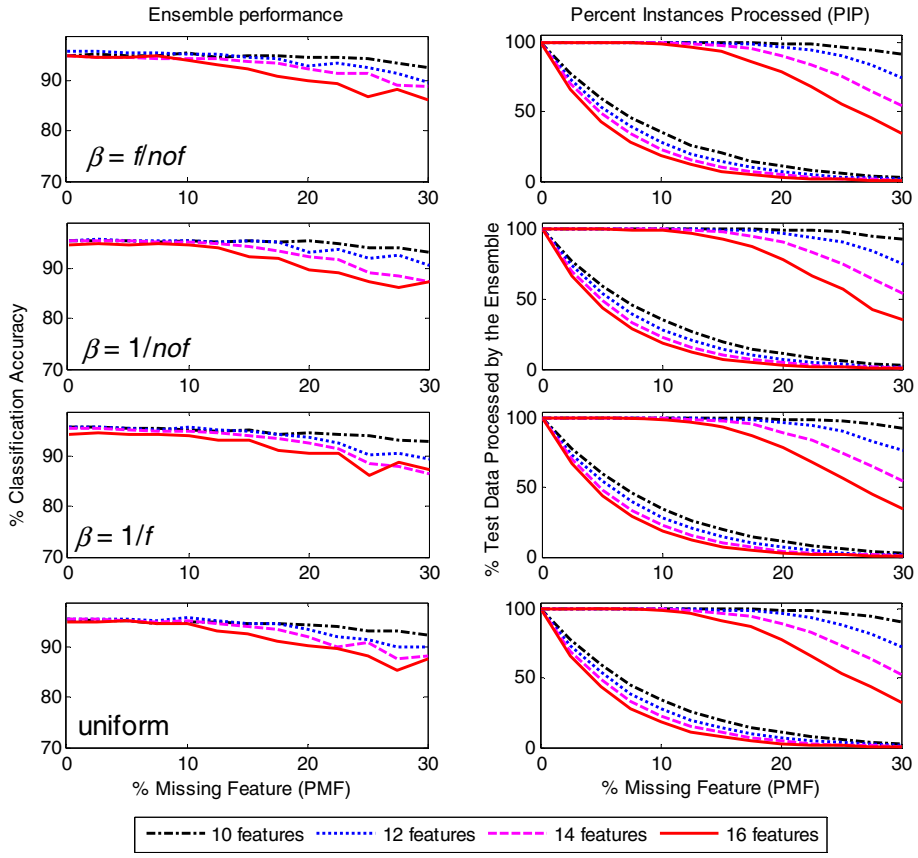

Fig. 4. Learn^{++}.MF performance on Wisconsin breast cancer database

4 Conclusions

In this paper, we described an ensemble based algorithm to address the missing feature problem in classification applications. The approach consists of generating a large number of classifiers using different subsets of the features, and then using only those classifiers whose training features did not include the missing features in the test instance. We observe that the algorithm works rather well with as much as 30% of the data missing.

The newer distribution update rules *f*/*nof* and 1/*nof* do add some performance gain to the algorithm, though the impact of the update rule was at best modest. The impact of the number of features used to train the classifiers, however, is quite dramatic: using fewer features to train the individual classifiers provides a better performance *and* can process a higher portion of the data, particularly when larger percentage of data is missing. PIP can be increased by increasing classifier count, *T*, if computational resources allow to do so. Of course, larger feature sets also require larger number of

classifiers to be generated, however reduced dimensionality in using feature subspaces allow faster training of each classifier, and offsets some of the computational burden of the algorithm. Based on our empirical observations, the number of classifiers required for good performance is typically on the order of 20-30 per dimensionality (feature) of the original data.

Notice that we have only considered the case of test data having missing features; however, the very nature of the algorithm also allows training with missing data. The random feature subsets would then be drawn from the available features only. Finally, we should add that the algorithm makes an implicit assumption: there is redundancy in the features that is distributed randomly. Of course, the identity of the redundant features are unknown to us, since otherwise, they would not have been part of the data. This is not an overly restricting assumption, as it is met by many real-world applications. It is those applications for which this algorithm is expected to perform well.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. ECS-0239090.

References

1. Morin, R.L., Raeside, D.E.: A reappraisal of distance-weighted k-nearest neighbor classification for pattern recognition with missing data. IEEE Trans. Systems, Man and Cybernetics 11 (1981) 241-243.
2. Dempster, A., Laird N., Rubin, D.R.: Maximum-likelihood from incomplete data via the EM algorithm (with discussion), Jour. of the Royal Statistical. Society, Series B, (1997) 1-38.
3. Gabrys, B.: Neuro-Fuzzy Approach to Processing Inputs with Missing Values in Pattern Recognition Problems, International Journal of Approximate Reasoning 30 (2002) 149-179.
4. Lim, C., Leong, J., Kuan, M.: A Hybrid Neural Network System for Pattern Classification Tasks with Missing Features. IEEE Transactions on Pattern Analysis and Machine Intelligence 27, (2005) 648-653
5. Melville, P., Shah, N., Mihalkova L., and Mooney R.: Experiments on ensembles with missing and noisy data, MCS 2004, Lecture Notes in Computer Science, 3077, (2004) 293-302.
6. Juszczak P. and Duin, R.P.W: Combining One-Class Classifiers to Classify Missing Data, MCS2004, Lecture Notes in Computer Science 3077 (2004) 92-101.
7. Ho, T.K.: The Random Subspace Method for Constructing Decision Forests, IEEE Transactions on Pattern Analysis and Machine Intelligence 20 (1998) 832-844.
8. Polikar, R., Udpa, L., Udpa, S., Honavar, V.: Learn++: An incremental learning algorithm for supervised neural networks. IEEE Transactions on System, Man and Cybernetics (C), 31 (2001) 497-508.
9. Krause S., Polikar, R.: An Ensemble of Classifiers Approach for the Missing Feature Problem, Int. Joint Conf. on Neural Networks 1 (2003) 553-556
10. Blake, C.L., Merz, C.J: UCI Machine Learning Repository," [Online Document], Accessed: 25 Nov 2006. http://www.ics.uci.edu/~mlearn/MLRepository.html