university of groningen

University Medical Center Groningen

University of Groningen

Feature selection and intelligent livestock management

Alsahaf, Ahmad

DOI:
10.33612/diss.145238079

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2020

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Alsahaf, A. (2020). *Feature selection and intelligent livestock management*. [Thesis fully internal (DIV), University of Groningen]. https://doi.org/10.33612/diss.145238079

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 30-03-2023

The content of this chapter was based on:

Alsahaf, A., Petkov, N., Shenoy, V., & Azzopardi, A. (2020). A Framework for Feature Selection Through Boosting (Under review).

Chapter 4

A Framework for Feature Selection Through Boosting

Abstract

As the dimensions of datasets in predictive modelling continue to grow, feature selection becomes increasingly practical. Datasets with complex feature interactions and high levels of redundancy still present a challenge to existing feature selection methods. We propose a novel framework for feature selection that relies on boosting, or sample re-weighting, to select sets of informative features in classification problems. We compare the proposed method to standard feature selection algorithms on a number of benchmark datasets. We show that the proposed approach reaches higher accuracies with fewer features on most of the tested datasets.

4.1 Introduction

The presence of irrelevant and redundant features in a dataset can lower the performance of predictive models, and prohibit the interpretability of their outcomes. Feature selection is the sub-field of machine learning concerned with identifying and selecting relevant features, and removing irrelevant and redundant ones.

Reducing the number of features in any dataset can be achieved with principally different approaches. Therefore, many methods of feature selection have been proposed in literature. The most popular taxonomy of those methods divides them into three broad categories: filter methods, wrapper methods, and embedded methods [Guyon and Elisseeff, 2003].

This categorization pertains to how the selection process and the associated prediction task are connected. Filter methods rank features independently of any prediction model, whereas wrapper methods evaluate the performance of candidate feature subsets on a pre-chosen predictor. Finally, embedded methods reduce the number of features in conjunction with solving a prediction problem [Guyon and Elisseeff, 2003; Dash and Liu, 1997; Tang et al., 2014].

Since an exhaustive search of all possible feature subsets is intractable for large datasets, the wrapper approach always makes use of some search strategy for finding candidate subsets of features to evaluate with the wrapped predictor [Tang et al., 2014]. Popular examples include hill-climbing, best-first, and genetic algorithms [Kohavi and John, 1997].

In this paper, we propose a greedy forward selection algorithm that uses embedded feature importance scores of tree ensemble models for choosing the candidate features. In addition, we use the concept of boosting by sample re-weighting to update the importance scores, and thus the search space, after every iteration.

The embedded feature importance scores of tree-based ensembles are powerful starting points for feature selection [Tuv et al., 2009]. This is largely due to the following factors: First, the versatility of those models; being scale invariant, scalable to large datasets, and able to handle numerical, categorical, and missing data. Second, the fact that the intrinsic importance scores can be derived at no additional cost over that of model training.

In a decision tree, the importance of a feature is defined as the total value of a node-splitting criterion that the feature is responsible for (e.g. Information Gain or the Gini Index). When used in ensembles, it is commonly defined as the sum or average of the splitting criterion that a feature causes across all trees [Breiman, 2002; Louppe et al., 2013].

In an ensemble of trees, like random forest [Breiman, 2001a], the importance scores of two or more redundant features will be spread evenly among them, due

to feature sub-sampling and bootstrapping. Using those scores for feature selection in large datasets could lead to falsely selecting a feature with many of its redundant copies [Genuer et al., 2010].

Other issues with tree-derived feature scores include the bias towards categorical features of high cardinality [Strobl et al., 2007], sensitivity to hyper-parameters [Genuer et al., 2010], and inconsistencies [Lundberg et al., 2018]. Inconsistencies refer to cases where the assigned importance of a feature decreases when its true impact on model performance has increased.

For the importance scores of tree models to be used reliably for feature selection, these shortcomings should be overcome. This can be partially achieved through boosting, or sample re-weighting.

Boosting algorithms, like AdaBoost and its derivatives [Freund and Schapire, 1997], rely on a sequential procedure of varying the sample weights of weak classifiers, typically decision-trees, based on the accuracy of previous boosting rounds. This leads the classifiers in later rounds of the algorithm to focus on samples that were misclassified in earlier rounds. Then, each classifier votes to determine the final outcome of the ensemble.

Through sample re-weighting, the process of boosting also affects the feature importance scores (and rankings) produced by the classifier being boosted. This presumably happens in such a way that in later boosting rounds, some features which initially ranked poorly, appear in top ranked positions due to becoming effective in classifying samples which were misclassified in earlier rounds.

This property of boosting has been exploited in the past to design feature selection algorithms [Das, 2001; Tieu and Viola, 2004; Liu et al., 2009]. In short, this is done by relying on a weight-sensitive feature ranking algorithm, and a sequential procedure of adding features, often one at a time. At each round, the ranking algorithm is applied to a re-weighted version of the training data. And the re-weighting is done based on the classification error produced by features selected so far. The best feature from each round, according to the feature ranking, is added to the selected subset.

In this paper, we expand on the use of boosting for feature selection, and propose an algorithm which we call FeatBoost. Compared to previous methods, we introduce the several contributions: 1) a sample weighting strategy which weights each sample according to its prediction probability. In contrast, previous methods up-weight all misclassified samples by the same amount, 2) a modular algorithm architecture, which decouples feature ranking from selection. This overcomes inconsistencies in feature rankings, and potentially increases the robustness of the selected subsets, 3) a sample weighting reset strategy, which prevents premature stopping of the algorithm.

Article structure

In section 4.2, we describe the novel elements of the proposed approach in relation to existing algorithms, and give an overview of recent, relevant developments in feature selection with tree ensembles. We then give the details of the proposed algorithm in Section 4.3. Then, we give the experimental settings in Section 4.4, followed by the results in Section 4.5. Finally, we discuss the implications of the results in Section 4.6.

Notations and definitions

We introduce the notations used in the rest of the paper. Whenever possible, we unify notations describing the different methods we compare. Therefore, the notations do not necessarily reflect those used in the original works.

Each instance of feature selection is solved on a given dataset $\mathcal{D}$ with p features, n samples, and a discrete target output $\boldsymbol{y}$ with n_c classes. A prediction of $\boldsymbol{y}$ is referred to as $\hat{\boldsymbol{y}}$. The subset of selected features for the given output is denoted by $\mathcal{X}$.

In methods that take as input the number of desired features to select, or the maximum number of features that can be selected, this number is denoted as p'. The actual number of features selected is denoted as p^*. If a method selects features sequentially, or produces a rank for the features in $\mathcal{X}$, then $\mathcal{X}^i$ refers to $\mathcal{X}$ at the ith iteration, for $i = 1, \ldots, p^*$.

For the sake of clarity, we stress here the difference between two types of boosting that are present in the proposed algorithm. The first may occur within the ensemble model that is used to generate the feature importance rankings - if a boosting method was used for that purpose - while the second is an outer layer of boosting (or sample re-weighting) performed specifically for the feature selection process, and bears no direct functional relation to the first type. From this point onward, mentions of sample re-weighting or boosting in this paper refer to the second type, unless otherwise specified.

4.2 Proposal and related work

In this section, we elaborate on the use of boosting or similar procedures in feature selection by highlighting a number of existing methods, and how the proposed algorithm differs from them. We then consider, more broadly, methods that use the embedded feature importance scores of decision-tree models as bases for feature selection.

4.2.1 Boosting and feature selection

Early examples of directly using boosting for feature selection include the following: Boosted Decision Stump Feature Selection (BDSFS) [Das, 2001], Boosting Image Retrieval [Tieu and Viola, 2004], and Boosted Mutual Information Feature Selection (BMIFS) [Liu et al., 2009]. Recent examples include Adaptive Boosting for Feature Selection (ABFS) [Barddal et al., 2019].

Dash and Liu [1997] used a tree stump as a base classifier, and followed the sample re-weighting strategy from AdaBoost for the purpose of feature selection. Namely, all training samples are initially given a weight of $\frac{1}{n}$, with n being the number of training samples. Then, at each subsequent iteration, the sample weights are given as a function of the classification error from the previous iteration. All misclassified samples are equally up-weighted according to Eq. 4.1. The best feature from every iteration, according to Information Gain, is selected.

$$\begin{aligned} \alpha &= \log \frac{1 - err}{err} \\ \omega_j^{i+1} &= \omega_j^i \cdot exp(\alpha); \quad \forall\, j = 1, \cdots, n \\ \omega_j^{i+1} &= \frac{\omega_j^{i+1}}{\sum_{j=1}^{n} \omega_j^{i+1}}; \quad \forall\, j = 1, \cdots, n \end{aligned} \tag{4.1}$$

where err is the classification error from the previous iteration and ω_j^i is the sample weight for sample j at the ith iteration.

Boosted Image Retrieval follows a similar procedure to BDSFS, adapted for the purpose of image retrieval. The BMIFS method differs in that the error used for updating sample weights is estimated from an information metric, Mutual Information, and not from a base classifier. Barddal et al. [2019] use this form of feature selection to the solve the problem of feature drift in data streams.

Our approach follows a framework that is similar to the aforementioned methods, but differs from them in a number of ways. First, we use a different sample re-weighting strategy. The AdaBoost weighting strategy, used in BDSFS (Eq. 4.1), re-weights all misclassified samples by the same amount, which disregards how far a sample is from being correctly predicted. To improve this, we weight each sample inverse-proportionally to its prediction probability, according to Eq. 4.2.

$$\begin{aligned}\alpha^{\mathbf{i}} &= -\sum_{c=1}^{n_c} \mathbf{Y_c} \log(\mathbf{P_c}) \\ \alpha_j^i &= \alpha_j^i / \alpha_j^{i-1}; \quad \forall\, j = 1, \cdots, n \\ \omega_j^{i+1} &\leftarrow \omega_j^i \cdot \alpha_j^i; \quad \forall\, j = 1, \cdots, n \\ \omega_j^{i+1} &= \frac{\omega_j^{i+1}}{\sum_{j=1}^{n} \omega_j^{i+1}}; \quad \forall\, j = 1, \cdots, n\end{aligned} \tag{4.2}$$

where $\mathbf{Y_c}$ is a one-hot encoded matrix indicating the correct class for each sample, $\mathbf{P_c}$ is an $n \times n_c$ matrix containing the class probabilities for each sample, obtained from a classifier, and ω_j^i is the sample weight for sample j at the ith iteration.

For a given sample, the associated weighting term α_j is decreased as the probability of the correct class for that sample approaches 1, and increased as it approaches zero. Therefore, samples which are far from being correctly classified are given higher weights in the next iteration.

We use a gradient boosting trees model, XGBoost, as a base learner for obtaining the feature scores [Chen and Guestrin, 2016]. Despite being computationally demanding compared to individual trees, ensembles of trees are more predictive in large datasets, and their feature importance scores reflect more complex interactions. XGBoost is a powerful example of such models, and it outperforms traditional tree-ensemble models in many applications [Luckner et al., 2017; Alsahaf et al., 2018b; Murauer and Specht, 2018].

Moreover, we introduce a few procedural changes, some of which were inspired by Iterative Input Selection (IIS) [Galelli and Castelletti, 2013], a tree-based forward selection method for regression problems (See 4.A).

For instance, in FeatBoost, we use a two-step process to select the best feature at each iteration. First, the top ranked features are obtained from the embedded feature scores of a tree model trained on all features. Then, we use a classifier to evaluate the classification performance of the top-ranked features obtained from the embedded scores.

This strategy is used in IIS, where evaluations of a regressor determine the best feature at each iteration [Galelli and Castelletti, 2013]. We use this approach of model evaluation in FeatBoost by testing m of the top ranked features at each iteration. However, instead of evaluating each of the candidate features individually as a single-input model, we append each feature to the selected features thus far, and evaluate the classification accuracy of the resulting models. Namely, at the ith iteration, we evaluate m models of order i. With this approach, the algorithm could be viewed as a step-wise greedy search in which the search space in each iteration is reduced from p features, to a user-specified number of features. And those candi-

date features change through the process of boosting.

The justification for this two-step process is the following: First, choosing the top feature from the feature ranking may not be reliable in the presence of feature redundancy in large datasets. Second, using model evaluations decouples the selection from the feature ranking algorithm, making it more robust [Galelli and Castelletti, 2013]. Moreover, this process solves the issue of inconsistency highlighted by Lundberg et al. [2018].

We make FeatBoost more modular by allowing the model choice for the evaluation step to be different than the model used for feature ranking. This additional decoupling of the two procedures - ranking and model evaluation - could lead to improvements in computational efficiency, by choosing the second model to be a computationally efficient one, as opposed to the first model, which produces the feature rankings.

Another element of IIS which we use in FeatBoost is that once a feature is selected, it is not dropped from the list of candidate features of subsequent iterations. This way, a feature may be selected twice, which translates to an automatic stopping condition for the algorithm.

The main motivation for using this strategy is that future iterations on re-weighted samples will rank new features in the presence of all other features. This means that if a feature ranks higher as a result of sample re-weighting, it does so in interaction with features that were selected before, and those that might be selected after. And the only difference between rankings of different iterations comes from sample re-weighting, and not from explicitly removing features from the list of candidates once they have been selected.

We make further adjustments to the boosting process to make it more adapted to feature selection. When boosting for the purpose of classification, as in AdaBoost, it is not necessary that each classifier is trained on a highly different sample distribution. In other words, if sample weights do not change significantly after a boosting round, this will not necessarily hinder performance, as the final classification will be determined by a majority vote of all classifiers.

On the other hand, in a feature selection context like the proposed method, a relevant feature is added at each boosting round. Therefore, classification error is expected to decrease with rounds. Consequently, the difference in α (Eq. 4.1 and 4.2) between consecutive rounds will decrease. If the difference becomes low enough, sample weights swill stop changing, and therefore the desired variation in the top ranked features will stop or diminish, causing the algorithm to prematurely terminate.

We solve this problem with two strategies. First, we base the re-weighting of samples not on the performance of the base classifier of the current iteration, but

on the relative performance between the current iteration and the preceding one, hence the normalization step of α in the second line of Eq. 4.2. Second, we use the following *reset* scheme: At any given iteration, if an existing feature is selected again, or the selected feature causes no increase in performance, we re-initialize the samples to have equal weights, and repeat the iteration with the new weights. In effect, this reboots the algorithm with a non-empty feature set, which could allow for the selection of additional useful features.

4.2.2 Ensemble tree models and feature selection

The simplest way to use the embedded scores of ensemble tree models for feature selection is by thresholding, or by only retaining features with non-zero scores. An alternative approach is to use a simple forward selection procedure; relying on the feature rankings to introduce one feature at a time to the selected subset, if the features causes a significant gain in performance [Genuer et al., 2010]. This approach will suffer from the various inconsistencies and biases of those scores (see Section 4.1).

More elaborate ways of using these scores have been proposed. One such example is to select features according to their importance scores when compared to artificial features, designed to trick the ranking algorithm [Kursa et al., 2010; Tuv et al., 2009]. A popular example of this approach is the Boruta method [Kursa et al., 2010]. Boruta works by creating shadow features, which are copies of the original features whose values are shuffled across samples, then computing the feature importance scores of the set of original plus shadow features using a random forest classifier. Original features that score lower than the top-scoring shadow features are deemed irrelevant, and are subsequently removed. The process is repeated until all the remaining original features are relevant. This method, by design, does not solve issues of redundancy, as it selects all relevant features [Kursa et al., 2010].

Tree-based models can also be used in combination with other approaches to improve their feature selection capabilities. Rao et al. [2019] combine gradient boosted trees with artificial bee colony algorithms for feature selection. Peker et al. [2015] combine the scores from random forest with the filter method ReliefF for selecting feature extracted from EEG signals.

Other methods attempt to solve the issues with tree-based feature scores not through external procedures, but by modifying, or redefining the importance scores themselves to address particular weaknesses [Lundberg et al., 2018; Nguyen et al., 2015; Strobl et al., 2008, 2007].

4.3 Methodology

For a given dataset $\mathcal{D}$ with p features, n training samples, and output $\boldsymbol{y}$ with n_c classes, the FeatBoost algorithm proceeds as follows: First, the selected subset of features, $\mathcal{X}$, is initialized to empty, and a user selected tree-based classifier, H_1, is trained on all samples with initial weights equal to $\frac{1}{n}$, to produce a ranking of all features. We choose H_1 to be an XGBoost classifier for the remainder of the paper.

Then, a user-specified number, m, of the top ranking features are evaluated and compared as single-input classifiers in k-fold cross validation, using either H_1, or a different classifier, H_2.[1] The best performing feature with H_2, according to an appropriate metric (e.g. classification accuracy, F-score, or area under the ROC curve) is added to the selected subset $\mathcal{X}$. In iterations other than the first, classifier H_2 is used to evaluate each of the m features appended to the features selected so far, $\mathcal{X}^{i-1}$.

Finally, H_1 is trained on all selected features, and its prediction probabilities are used to update the sample weights for the following iteration according to Eq. 4.2.

The algorithm stops if the increase in accuracy of $\mathcal{X}$ with respect to the previous iteration is below a user-defined threshold, ϵ, or if a feature is selected twice.

The reset scheme functions as follows: If at iteration i, a feature is selected which already belongs to $\mathcal{X}^{i-1}$, or if the feature does not improve classification performance, the algorithm normally terminates, and $\mathcal{X}^{i-1}$ is taken as the final subset. Under the reset scheme, this is temporarily overcome by resetting the sample weights to their initial values, and repeating iteration i. This will lead to the feature ranking being equal to that of the first iteration, albeit with an initial $\mathcal{X}$ that is not empty. This could lead the model evaluation step (with H_2) to select a feature that is partially redundant to one or more features in $\mathcal{X}^{i-1}$, but nonetheless having additional predictive value. If that occurs, the algorithm resumes its normal course from iteration i until a stopping condition is reached again. Otherwise, if the reset scheme does not lead to selecting a new useful feature, the algorithm stops.[2]

The pseudocode of the algorithm, along with the details of weighting and reset strategies are given in Algorithm 1.

4.4 Experimental settings and evaluation

In this section, we describe the data and experimental settings. Then, we briefly describe Boruta and ReliefF, the methods we compared FeatBoost with.

[1]Note that H_2 does not need to be tree-based, nor sensitive to sample weights, since it is not used in the feature ranking process.

[2]A software implementation of the algorithm is available at `https://github.com/amjams/FeatBoost`

input : Dataset $\mathcal{D}$ with p features, n samples, and output $\boldsymbol{y}$ with n_c classes.
output: $\mathcal{X}^i$ (a subset of selected features at the ith iteration).
1 **initialize:** $i \leftarrow 1$, $\mathcal{X}^0 \leftarrow \phi$, reset $\leftarrow 0$, and sample weights $\omega_j^0 \leftarrow \frac{1}{n} \quad \forall\, j = 1, \cdots, n$;
2 **choose:** stopping conditions $\{\epsilon, p'\}$, parameters m, k, and classifiers H_1 and H_2.
3 **while** $i < p'$ **and** *reset* $\leqslant 1$ **do**
4   **fit** H_1 to $\mathcal{D}$ with ω^i and rank all features;
5   **fit** H_2 to $\{\mathcal{X}^{i-1}, x\}$ for all x in the top m ranked features;
6   **find** the top performing feature x^i, in cross-validation;
7   **compute** $\Delta Acc = Acc[H_2(\mathcal{X}^{i-1}, x^i)] - Acc[H_2(\mathcal{X}^{i-1})]$, in cross-validation;
8   **if** $\Delta Acc > \epsilon$ **and** $x^i \notin \mathcal{X}^i$ **then**
9   **set** $\mathcal{X}^i \leftarrow \{\mathcal{X}^{i-1}, x^i\}$;
10   **fit** H_1 to $\mathcal{X}^i$ to find the class probabilities, $\mathbf{P_c}$, and compute $\alpha^{\mathbf{i}}$:
          $\alpha^{\mathbf{i}} = -\sum_{c=1}^{n_c} \mathbf{Y_c} \log(\mathbf{P_c})$;
11   **re-normalize** $\alpha_j^i = \alpha_j^i / \alpha_j^{i-1}; \quad \forall\, j = 1, \cdots, n$;
12   **update** $\omega_j^{i+1} \leftarrow \omega_j^i \cdot \alpha_j^i; \quad \forall\, j = 1, \cdots, n$;
13   **re-normalize** $\omega_j^{i+1} = \frac{\omega_j^{i+1}}{\sum_{j=1}^{n} \omega_j^{i+1}}; \quad \forall\, j = 1, \cdots, n$;
14   reset $\leftarrow 0$;
15   i += 1
16   **else**
17   **re-initialize weights** $\omega_j^i \leftarrow \frac{1}{n} \quad \forall\, j = 1, \cdots, n$;
18   reset += 1
19   **end**
20 **end**

Algorithm 1: Pseudocode of FeatBoost.

We applied each algorithm to 16 real datasets, and one artificial dataset, Madelon, which was designed to benchmark feature selection algorithms [Guyon et al., 2006]. Table 4.1 contains a description of the datasets, and their dimensions. The datasets represent multiple domains, including large biomedical and text data.

On each dataset, we apply an m-by-k-fold cross-validated selection procedure, with $m = 3$, and $k = 10$: We split each dataset into ten equally sized folds, and apply each feature selection algorithm to the training folds separately. Then, we use the selected features to train a classifier on the training folds, and then test them on the held-out test folds, on which the classification performance is evaluated. The performance is measured in terms of the average classification accuracy of the selected subsets on the test data, and the computation time of the feature selection algorithm. We repeat the entire procedure $m = 3$ times with random shuffles of the sample set, for a total of 30 runs of each algorithm. A similar validation procedure is used by Song et al. [2013].

In cases where an algorithm selects more than 100 features, we only evaluate the accuracies of the first 100. Moreover, since each algorithm produces multiple subsets, we try to exclude those that could skew the average performance at the validation stage. Therefore, for each algorithm, we evaluate only the resulting subsets with a number of features equal to or larger than the mode of all subset sizes for that algorithm. Subsets which are larger than the mode are truncated to have a number of features equal to the mode.

We used a Nearest Neighbor classifier to validate the selected subsets. A Nearest Neighbor classifier is a sensible choice for validating feature subsets, as its performance is more likely to suffer from the inclusion of irrelevant, redundant, or noisy features, when compared to more complex classifiers [Loughrey and Cunningham, 2005]. Validation with a Gaussian Naive Bayes classifier and an XGBoost classifier with default parameters are also given in 4.D.

Additional experiments are given in the appendices. In 4.C, we examine the effect of the reset and weighting strategies. In 4.B, we show the advantage of the two-step selection procedure over selection based on feature ranking only.

4.4.1 Compared methods

XGBoost: XGB-FS

Since we build FeatBoost around a specific feature importance score, one derived from an XGBoost classifier, then a suitable benchmark to compare against is the same base score but with a simpler threshold. For that purpose, we define the first method in the comparison to be the feature importance scores from the same classifier used for ranking in FeatBoost, with the threshold being the mean of all feature scores. That is, features with an importance score lower than the mean of all feature scores are discarded. We denote this by XGB-FS in the rest of the paper.

Boruta

The second method we compare against is Boruta. In the comparison, we use XGBoost instead of random forest as the base ranking algorithm for Boruta. That way, we are able to achieve a fairer comparison, and determine which of the approaches makes better use of the same underlying feature scores. Moreover, since Boruta does not rank the elements of the selected subset by default, we post-rank them with an additional fitting of the classifier. This allows us to compare the performance of all methods iteratively with each added feature.

ReliefF

Finally, we compare FeatBoost to the ReliefF algorithm [Kira and Rendell, 1992]. ReliefF is a powerful filter-based approach which belongs to the Relief family of feature selection methods [Urbanowicz et al., 2018; Kira and Rendell, 1992]. We use it in the comparison to represent a baseline of filter-based approaches.

Since Relief-based methods are feature ranking algorithms, a suitable threshold is needed in order to use them for feature subset selection. As in XGB-FS, we used the mean value of the scores as a lower threshold.
We configured the algorithms as follows:

1. **XGB-FS:** We configured XGB-FS with 100 trees, and a maximum tree depth of 20. We set the maximum depth parameter to a high value in order to detect higher order feature interactions that might be present in some datasets [Johnson, 2009]. We set the remaining parameters of the classifier to their default values.[3]

2. **FeatBoost:** We used FeatBoost in two configurations. In the first, we set H_1 and H_2 to be the same XGBoost classifier in XGB-FS. In the second, we set H_2 as a Nearest Neighbor classifier. We set the remaining parameters as follows: $k = 3$, $m = 50$, $p' = 100$, and $\epsilon = 10^{-18}$.

3. **Boruta:** We used Boruta with the same classifier used in XGB-FS, and default settings otherwise. We implemented Boruta with the BorutaPy Python package, which we modified to be compatible with XGBoost.

4. **ReliefF:** We used ReliefF with a number of neighbors equal to 10 and implemented the algorithm with the Skrebate Python package.

4.5 Results and discussion

The results of the feature selection comparison are summarized in Fig. 4.1, which shows the average classification accuracies of the subsets selected by the compared algorithms. The average computation times of each algorithm on all datasets are shown in Table 4.2.

In all datasets, FeatBoost selects better performing features in the leading ranks than the feature importance score on which it is based, XGB-FS. This shows that sample re-weighting and the model evaluation performed in FeatBoost, improve the performance of the base ranking. Moreover, the automatic stopping conditions in

[3]Default values as per XGBoost's stable Python release 0.90

Table 4.1: Descriptions of datasets used in the comparison.

Name	#Features	#Samples	#Classes	Domain
Madelon	500	2600	2	Synthetic
Isolet	617	1560	26	Speech recognition
Basehock	4862	1993	2	Text
PCMAC	3289	1943	2	Text
Relathe	4322	1427	2	Text
Coil20	1024	1440	20	Face image
ORL	1024	400	40	Face image
Orlraws10P	10394	100	10	Face image
Pixar10P	10000	100	10	Face image
WarpAR10P	2400	130	10	Face image
WarpPIE10P	2420	210	10	Face image
GlI	22283	85	2	Biomedical
GLI-BRA	49151	180	4	Biomedical
TOX	5748	171	4	Biomedical
SMK-CAN	19993	187	2	Biomedical
CLL-SUB	11340	111	3	Biomedical
Colon	2000	62	2	Biomedical

FeatBoost lead to selecting significantly smaller subsets than the mean-value threshold that we used in XGB-FS.

In most datasets, FeatBoost outperforms Boruta and ReleifF as well, reaching higher accuracies with fewer features. This is most apparent in Coil20, Isolet, and Orl. In those datasets, the performance of feature subsets selected by FeatBoost converges significantly faster than the second best method.

In terms of computation time, XGB-FS is predictably the most efficient approach across all datasets, followed by ReliefF. As for FeatBoost and Boruta, we observe that the former, when configured with an XGBoost wrapped classifier, is slightly faster than Boruta, which uses the same classifier to provide its base feature ranking. On the other hand, when FeatBoost uses a KNN classifier, it becomes significantly faster than Boruta.

It is worth noting that this increase in efficiency in FeatBoost - obtained by using KNN instead of XGBoost as the evaluation classifier - does not sacrifice the performance of the algorithm. In fact, this configuration performs better, perhaps unsurprisingly, when the validation classifier is also KNN (Fig. 4.1). It also performs well when XGBoost is the validation classifier (Fig. 4.4). This demonstrates that the modular architecture of the algorithm can take advantage of a powerful ranking algorithm, that of XGBoost, while using a simpler and more efficient evaluation classifier.

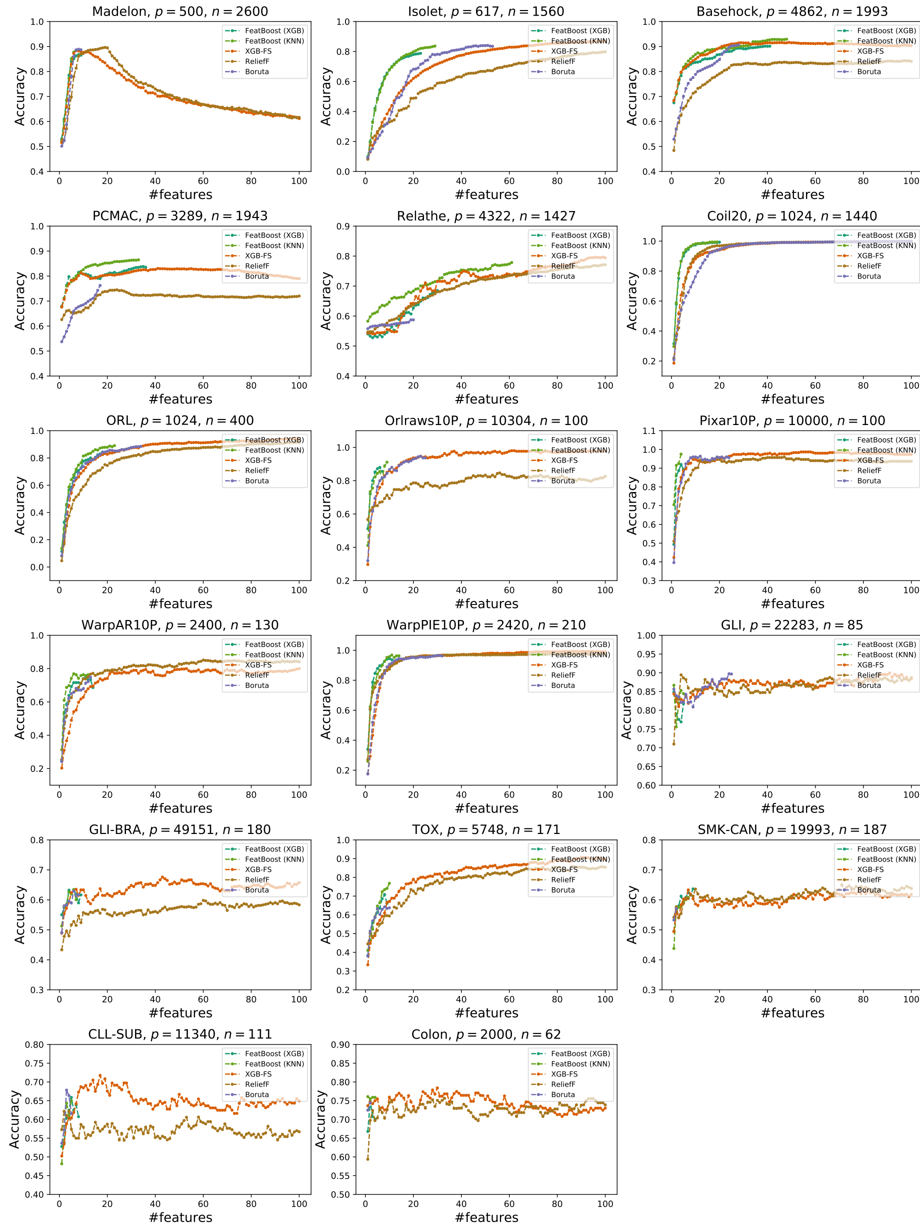

Figure 4.1: Accuracies of selected subsets validated with a KNN classifier

Table 4.2: The average computation time in minutes of each feature selection algorithm.

	XGB-FS	FeatBoost (XGB)	FeatBoost (KNN)	ReliefF	Boruta
Madelon	0.250	58.707	3.556	1.022	20.813
Isolet	1.035	168.452	52.527	0.717	154.914
Basehock	1.542	217.516	73.329	4.342	131.142
PCMAC	1.108	186.073	45.071	2.809	105.667
Relathe	1.076	163.376	60.920	2.773	91.732
Coil20	1.181	333.852	36.673	2.193	241.225
ORL	0.556	168.383	18.179	0.571	102.347
Orlraws10P	0.477	12.915	6.432	6.159	48.654
Pixar10P	0.374	6.912	3.400	5.869	41.150
WarpAR10P	0.158	15.463	2.969	2.025	13.746
WarpPIE10P	0.245	22.914	5.270	3.281	24.119
GLI	0.126	1.663	0.958	1.210	7.859
GLI-BRA	2.816	58.686	47.257	15.370	142.591
TOX	0.337	16.148	6.646	1.744	16.513
SMK-CAN	0.430	7.135	4.068	2.438	18.232
CLL-SUB	0.290	7.907	4.324	1.539	15.362
Colon	0.009	0.717	0.092	0.054	0.661
Average	0.706	85.107	21.863	3.183	69.219

4.6 Conclusion

We have shown that the proposed FeatBoost algorithm, which is based on boosting, and a stage-wise greedy procedure, is able to use the feature importance scores derived from an ensemble of decision-trees to select high performing feature subsets for classification problems. The resulting subsets outperform those obtained by simple thresholding of the baseline scores. They also outperform in most cases the Boruta algorithm, which we configured to use the same feature scores as a basis, and improved with a post-ranking of the selected subsets.

Moreover, we have shown that the usefulness of the algorithm is not limited to $(n \gg p)$ datasets. Out of the 17 datasets we tested, 10 were $(n \ll p)$, and FeatBoost outperformed or matched Boruta's performance in 8 of them.

The computational cost of the algorithm is sensitive to several design choices, most importantly, the ranking algorithm, the parameter m, and the wrapped classifier. We have shown that changing the latter from XGBoost to the much simpler Nearest Neighbor improved the speed of the algorithm without significantly affecting the performance of the selected features.

The proposed boosting framework, and the two-step selection procedure, circumvent the weaknesses in the importance scores of tree-based models externally,

without changing how the scores are computed. The algorithm is therefore independent of the particular scores being used, or the wrapped classifier. It would be useful to investigate the use of other classifiers, and to test if other ranking algorithms, like the filter based ReliefF, would react similarly within the same algorithm.

Appendix: Background

4.A Iterative Input Selection

Tree-based Iterative Input Selection (IIS) [Galelli and Castelletti, 2013], is a step-wise forward selection method for regression problems. The method has three main steps. First, a non-linear model-based ranking algorithm is used to determine the topmost relevant features at every iteration. An Extra-Trees regressor is used for that purpose [Geurts et al., 2006]. Second, from the top ranked features at every iteration, the one that performs best individually on the continuous target output $\boldsymbol{y}$ is selected and added to $\mathcal{X}$, using single-input-single-output (SISO) model evaluations. Similarly, Extra-Trees models are used in that step. Finally, the performance of the selected features is evaluated at every step on a non-linear regression model (also an Extra-Trees regressor), and the output is adjusted for the following iteration. The adjustment is done by setting the output of the next iteration as the prediction residual of the current iteration (Eq. 4.3). Selection stops when a feature has been selected twice or performance stops improving.

$$y_j^{i+1} = r_j^i = y_j^i - \hat{y}_j^i;\ \forall\, j = 1, \cdots, n \tag{4.3}$$

Appendix: Additional Results and Experiments

4.B The effect of model evaluations on selection

In this section, we show the effect of using model evaluations to select the best feature at every iteration, instead of selecting the top ranked feature. We do this by comparing the subset accuracy of FeatBoost with $m = 50$ and $m = 1$. The second option means that the top ranked feature by XGBoost is the selected feature, and no model evaluations are used. The results are shown in Fig. 4.2. We note that when $m = 50$, the feature selected at each iteration is not necessarily the top-ranked one, and that the algorithm performs better than the case with $m = 1$. We disabled the reset scheme in this experiment for simplicity.

4.C Weighting strategy and reset

We demonstrate with an experiment the effects of two major elements in FeatBoost: the proposed weighting strategy, and the weighting reset scheme. We do so by applying FeatBoost to all the benchmark datasets, once with the default weighting strategy that is given in Algorithm 1, and once with a sample weighting equation

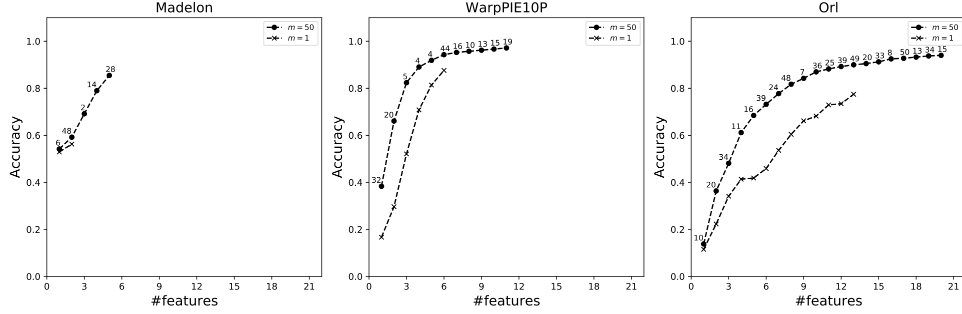

Figure 4.2: Feature subset accuracy of FeatBoost with $m = 50$ and $m = 1$, on datasets Madelon, WarpPIEp, and Orl. The number atop the plot line indicates the order, from the XGBoost ranking, of the feature that had the highest model accuracy, and was thus selected by the algorithm.

similar to AdaBoost, i.e. the one used in BDSFS (Eq. 4.1). We use an extended version of the equation which accounts for multi-class cases [Hastie et al., 2009], and follow the same normalization approach for the term α which we use in the default weighting strategy of FeatBoost. The weighting strategy is given in Eq. 4.4.

$$\begin{aligned} \alpha^i &= \log \frac{1 - err}{err} + \log(n_c - 1) \\ \alpha^i &= \alpha^i / \alpha^{i-1} \\ \omega_j^{i+1} &= \omega_j^i \cdot exp(\alpha^i);\ \forall\ j = 1, \cdots, n \\ \omega_j^{i+1} &= \frac{\omega_j^{i+1}}{\sum_{j=1}^{n} \omega_j^{i+1}};\ \forall\ j = 1, \cdots, n \end{aligned} \tag{4.4}$$

In both cases, we indicate the iteration at which the first successful reset occurs. In other words, this shows when a stopping criterion is first reached. The results are in Fig. 4.3. Therein, we indicate with a dotted line segment, the performance obtained by features that were selected after the first successful reset. If a line has no dotted segment, this means that the algorithm reached its end without a successful reset.

4.D Validation with other classifiers

In this section, we give the results of the main comparison (Section. 4.4) with two additional validation classifiers, XGBoost, and Gaussian Naive Bayes. The results are shown Fig. 4.4 and Fig. 4.5.

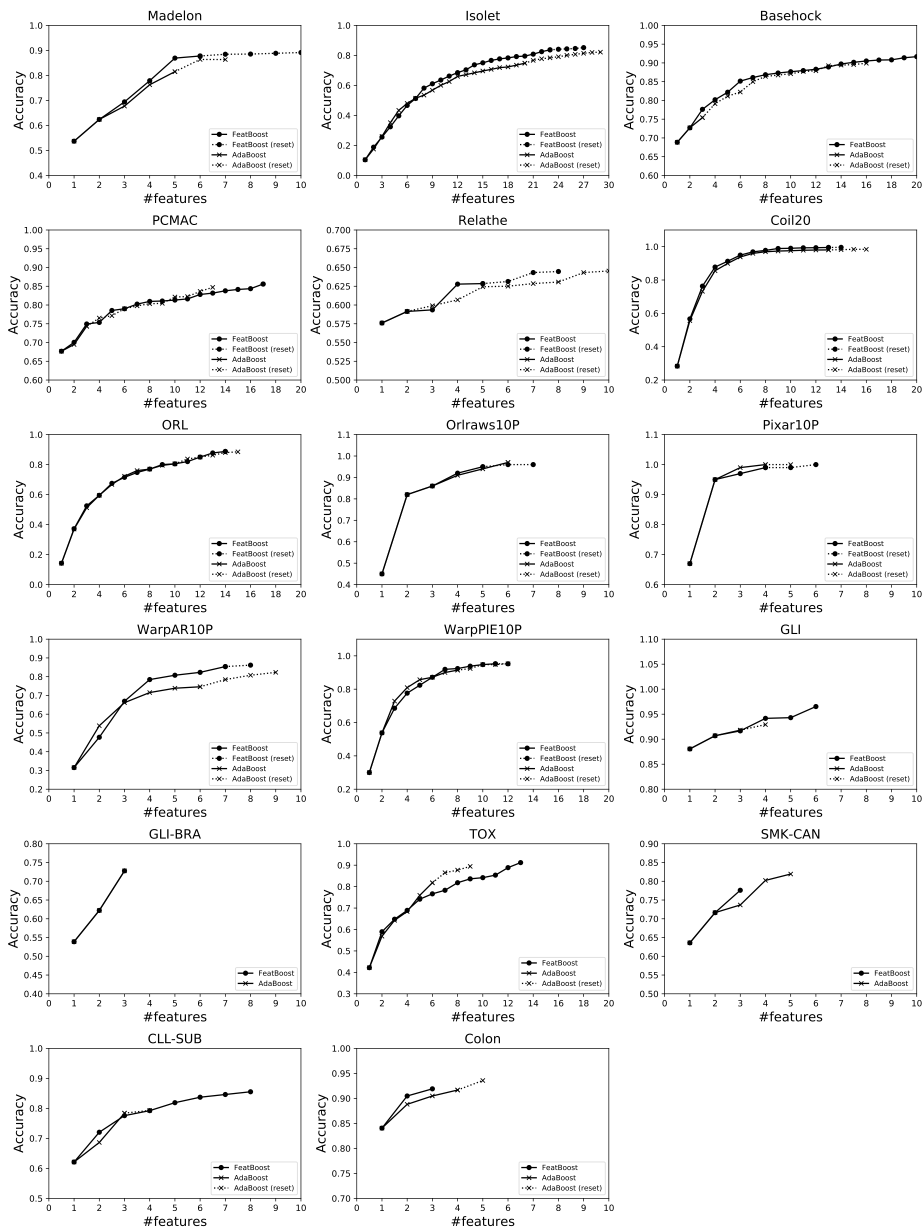

Figure 4.3: The effect of the reset strategy on the selected subsets

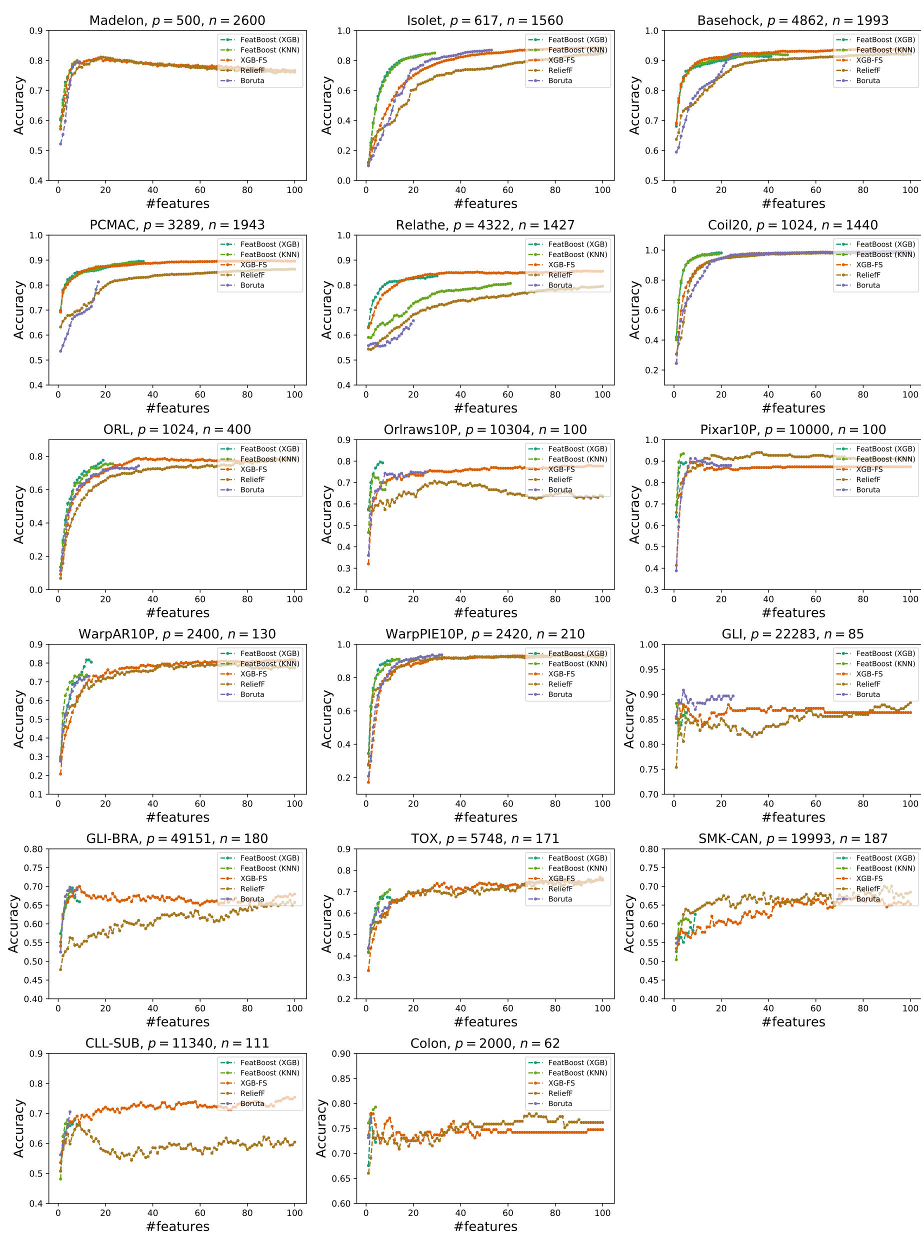

Figure 4.4: Accuracies of selected subsets validated with an XGBoost classifier

to AdaBoost, i.e. the one used in BDSFS (Eq. 4.1). We use an extended version of the equation which accounts for multi-class cases [Hastie et al., 2009], and follow the same normalization approach for the term α which we use in the default weighting strategy of FeatBoost. The full weighting strategy is given in Eq. 4.5.

$$\begin{aligned}
\alpha^i &= \log \frac{1 - err}{err} + \log(n_c - 1) \\
\alpha^i &= \alpha^i / \alpha^{i-1} \\
w_j^{i+1} &= w_j^i \cdot exp(\alpha^i);\ \forall\, j = 1, \cdots, n \\
w_j^{i+1} &= \frac{w_j^{i+1}}{\sum_{j=1}^{n} w_j^{i+1}};\ \forall\, j = 1, \cdots, n
\end{aligned} \tag{4.5}$$

In both cases, we indicate the iteration at which the first successful reset occurs. In other words, this shows when a stopping criterion is first reached. The results are in Fig. 4.6. Therein, we indicate with a dotted line segment, the performance obtained by features that were selected after the first successful reset. If a line has no dotted segment, this means that the algorithm reached its end without a successful reset.

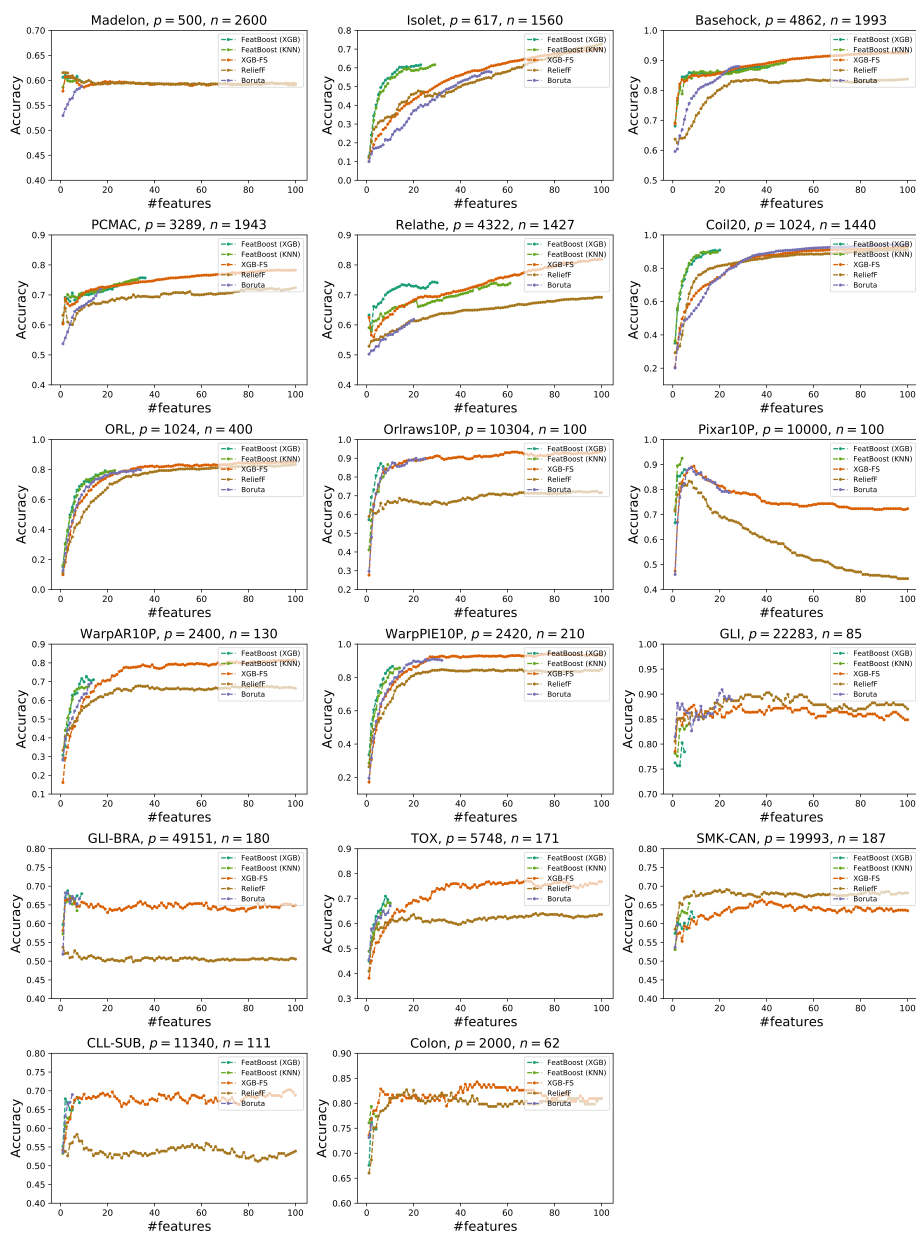

Figure 4.5: Accuracies of selected subsets validated with a Gaussian Naive Bayes classifier

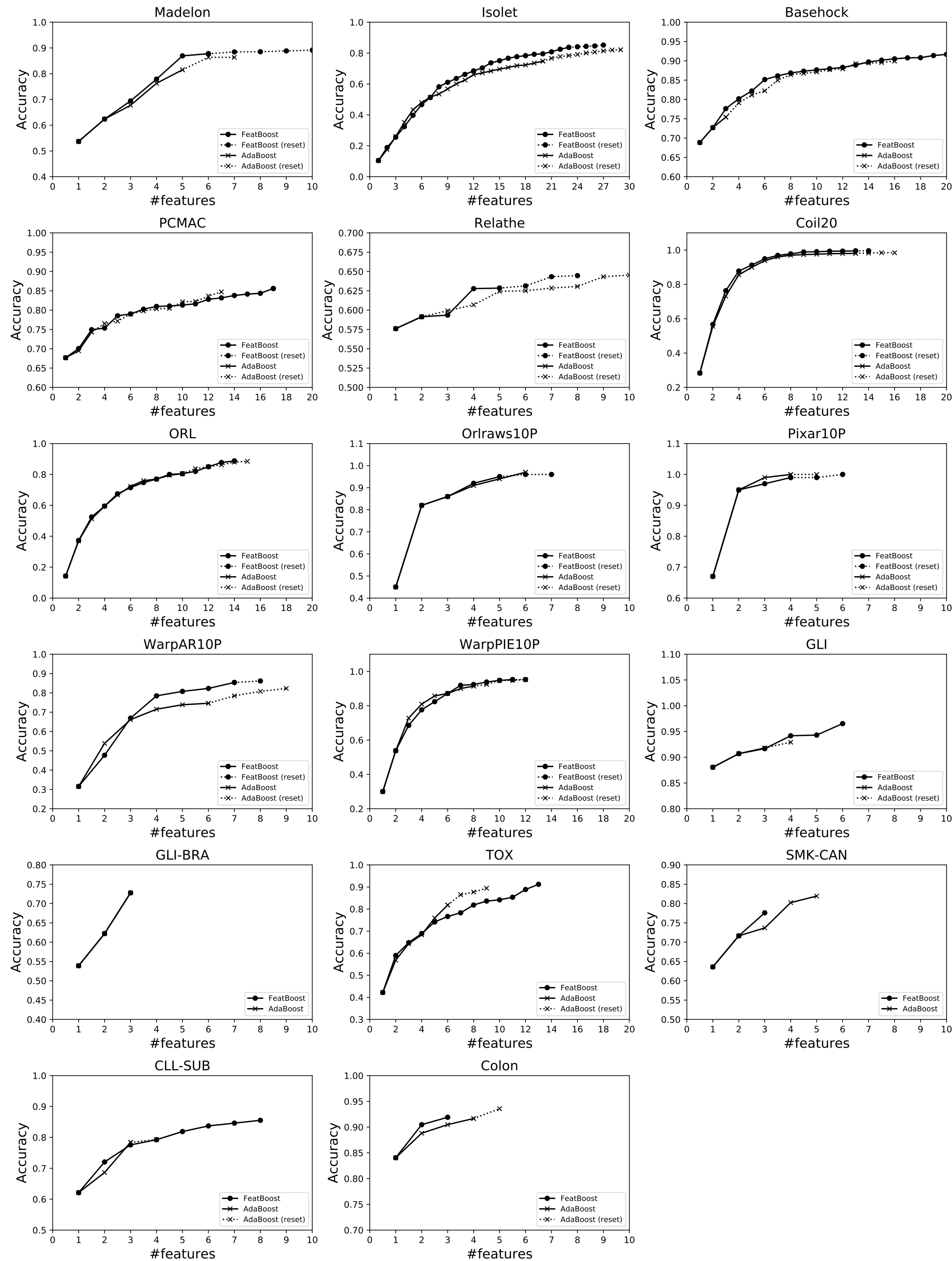

Figure 4.6: The training accuracy of FeatBoost using its default weighting strategy and an AdaBoost weighting strategy on all the benchmark datasets