1.If the data is structured tabular data, I will recommend using boosting (XGBoost). The following are the reasons behind it:

Steps to compare models:

- In this paper first we investigate whether the proposed deep models have advantages when used for various tabular datasets. For real-world applications, the models must (1) perform accurately, (2) be trained and make inferences efficiently, and (3) have a short optimization time (fast hyperparameter tuning). Therefore, we first evaluate the performance of the deep models, XGBoost, and ensembles on various datasets.
- Next, we analyze the different components of the ensemble.
- Then, we investigate how to select models for the ensemble and test whether deep models are essential for producing good results or combining 'classical' models (XGBoost, SVM [Cortes and Vapnik] and CatBoost) is sufficient.
- In addition, we explore the tradeoff between accuracy and computational resource requirements.
- Finally, we compare the hyperparameter search process of the different models and demonstrate that XGBoost outperforms the deep models.

Dataset Features Classes	Samples Source Paper
Gesture Phase 32 5	9.8k OpenML DNF-Net
Gas Concentrations 129 6	13.9k OpenML DNF-Net
Eye Movements 26 3	10.9k OpenML DNF-Net
Epsilon 2000 2	500k PASCAL Challenge 2008 NODE
YearPrediction 90 1	515k Million Song Dataset NODE
Microsoft (MSLR) 136 5	964k MSLR-WEB10K NODE
Rossmann Store Sales 10 1	1018K Kaggle TabNet
Forest Cover Type 54 7	580k Kaggle TabNet
Higgs Boson 30 2	800k Kaggle TabNet
Shrutime 11 2	10k Kaggle New dataset
Blastchar 20 2	7k Kaggle New dataset

Table 1: Description of the tabular datasets

Model Name	Rossman	CoverType	Higgs	Gas	Eye	Gesture
XGBoost	490.18 ± 1.19	3.13 ± 0.09	21.62 ± 0.33	2.18 ± 0.20	**56.07**±0.65	80.64 ± 0.80
NODE	488.59 ± 1.24	4.15 ± 0.13	21.19 ± 0.69	2.17 ± 0.18	68.35 ± 0.66	92.12 ± 0.82
DNF-Net	503.83 ± 1.41	3.96 ± 0.11	23.68 ± 0.83	**1.44** ±0.09	68.38 ± 0.65	86.98 ± 0.74
TabNet	**485.12**±1.93	3.01 ± 0.08	**21.14**±0.20	1.92 ± 0.14	67.13 ± 0.69	96.42 ± 0.87
1D-CNN	493.81 ± 2.23	3.51 ± 0.13	22.33 ± 0.73	1.79 ± 0.19	67.9 ± 0.64	97.89 ± 0.82
Simple Ensemble	488.57 ± 2.14	3.19 ± 0.18	22.46 ± 0.38	2.36 ± 0.13	58.72 ± 0.67	89.45 ± 0.89
Deep Ensemble w/o XGBoost	489.94 ± 2.09	3.52 ± 0.10	22.41 ± 0.54	1.98 ± 0.13	69.28 ± 0.62	93.50 ± 0.75
Deep Ensemble w XGBoost	485.33 ± 1.29	**2.99** ±0.08	22.34 ± 0.81	1.69 ± 0.10	59.43 ± 0.60	**78.93** ±0.73
		TabNet			DNF-Net	

Model Name	YearPrediction	MSLR	Epsilon	Shrutime	Blastchar
XGBoost	77.98 ± 0.11	55.43±2e-2	11.12±3e-2	13.82 ± 0.19	20.39 ± 0.21
NODE	76.39 ± 0.13	55.72±3e-2	**10.39**±1e-2	14.61 ± 0.10	21.40 ± 0.25
DNF-Net	81.21 ± 0.18	56.83±3e-2	12.23±4e-2	16.8 ± 0.09	27.91 ± 0.17
TabNet	83.19 ± 0.19	56.04±1e-2	11.92±3e-2	14.94±, 0.13	23.72 ± 0.19
1D-CNN	78.94 ± 0.14	55.97±4e-2	11.08±6e-2	15.31 ± 0.16	24.68 ± 0.22
Simple Ensemble	78.01 ± 0.17	55.46±4e-2	11.07±4e-2	13.61±, 0.14	21.18 ± 0.17
Deep Ensemble w/o XGBoost	78.99 ± 0.11	55.59±3e-2	10.95±1e-2	14.69 ± 0.11	24.25 ± 0.22
Deep Ensemble w XGBoost	**76.19** ±0.21	**55.38**±1e-2	11.18±1e-2	**13.10**±0.15	**20.18**±0.16
		NODE		New datasets	

Table2. Result of the Tabular Datasets

Therefore, we have learned about the performance of recently proposed deep models for tabular datasets. In our analysis, the deep models were weaker on datasets that did not appear in their original papers, and they were weaker than XGBoost, the baseline model. Therefore, we proposed using an ensemble of these deep models with XGBoost. This ensemble performed better on these datasets than any individual model and the 'non-deep' classical ensemble. In addition, we explored possible tradeoffs between performance, computational inference cost, and hyperparameter optimization time, which are important in real-world applications. Our analysis shows that we must take the reported deep models' performance with a grain of salt. When we made a fair comparison of their performance on other datasets, they provided weaker results. Additionally, it is much more challenging to optimize deep models than XGBoost on a new dataset. However, we found that an ensemble of XGBoost with deep models yielded the best results for the datasets we explored.

Consequently, when researchers choose a model to use in real-life applications, they must consider several factors. Apparently, under time constraints, XGBoost may achieve the best results and be the easiest to optimize. However, XGBoost alone may not be enough to achieve the best performance; we may need to add deep models to our ensemble to maximize performance.

In conclusion, despite significant progress using deep models for tabular data, they do not outperform XGBoost on the datasets we explored, and further research is probably needed in this area. Taking our findings into account, future research on tabular data must systematically check the performance on several diverse datasets. Our improved ensemble results provide another potential avenue for further research. Comparing models requires including details about how easy it is to identify the most appropriate hyperparameters. Our study showed that some deep models require significantly more attempts to find the correct configuration. Another line of

research could be developing new deep models that are easier to optimize and may compete with XGBoost in terms of this parameter.

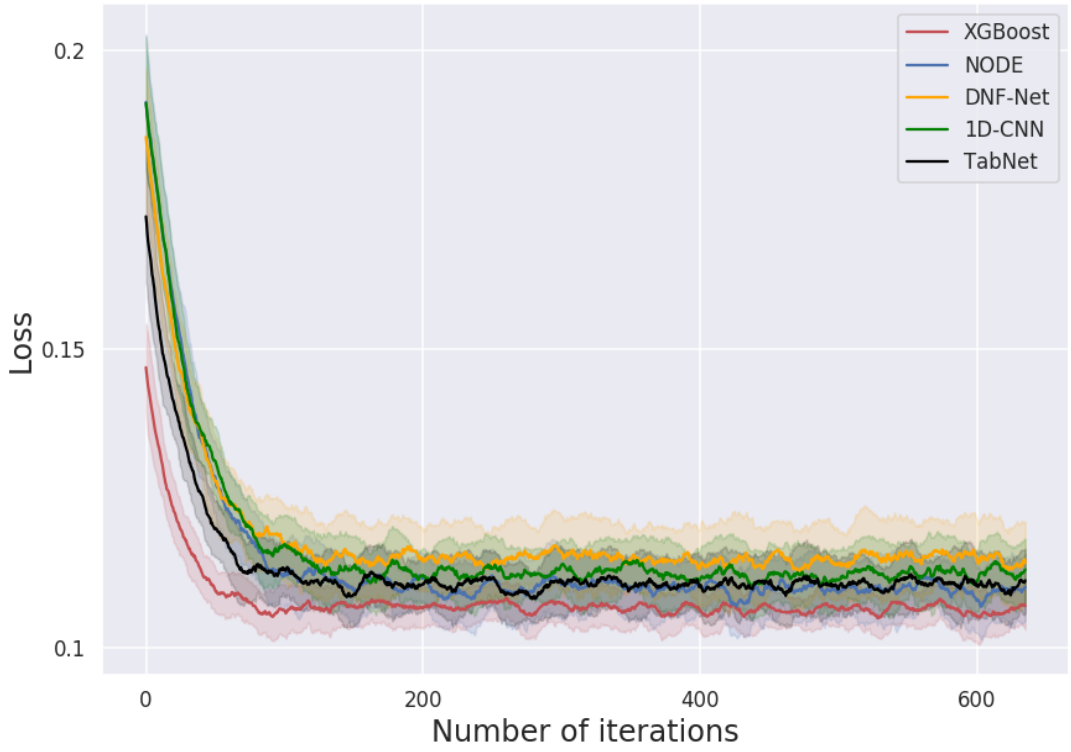


Figure 1: The Hyper-parameters optimization process for different models.

2.

Self Averaging: It depends on the type of variable that the Random Forest is predicting, and perhaps on the specific implementation of Random Forest. Following is an overview of the simplest techniques.

Continuous Target

In the case of a continuous target variable yy, each of the trees in an ensemble will generate a prediction $\hat{y}i$. The naivest way to combine the results from trees in an ensemble is to take the mean of all predictions.
Suppose you have an ensemble of 10 trees. Then the combined prediction $\hat{y}$ would be computed as

$$\hat{y}=1/10\sum i=1/10\hat{y}i$$

Where each tree has a weight ωiωi. The weight could be determined, for example, by the tree's performance. This approach might slightly improve accuracy in some cases, but it's easier to overfit.

Categorical Target

In the case where your random forest is predicting the value of a categorical variable, you can allow each tree to act like a member of a "committee" and cast a vote.

Suppose a categorical variable has three possible values, AA, BB, or CC, and your random forest has 10 trees. You generate a prediction with each tree, and that prediction counts as one vote. For example, suppose 5 of your trees predict A, 3 of them predict BB, and 2 of them predict C. The combined prediction will be A.
Notice that you may run into cases where there is a tie. Suppose 5 of your trees predict AA, and the other five predict B. Which prediction do you go with?
One option is to predict the majority class in the case of a tie. In the example above, if the training data contains more B examples than A examples, then we would predict BB.
Another option for resolving ties is to weight votes by the accuracy of the tree. Suppose again that you have five trees that vote for A and five trees that vote for BB. If you observe that, on average, the trees voting for B are more reliable, you may give their votes extra weight and predict B.
Hence, it proves that random forests are self-averaging.

Interpolating algorithm:

Algorithm 2: Random Forests Hastie et al. (2009)

1. For $b = 1$ to B:
   (a) Draw a bootstrap sample $\mathbf{X}^*$ of size N from the training data
   (b) Grow a decision tree T_b to the data $\mathbf{X}^*$ by doing the following recursively until the minimum node size n_{min} is reached:
      i. Select m of the p variables
      ii. Pick the best variable/split-point from the m variables and partition
2. Output the ensemble $\{T_b\}_b^B$

Let $\hat{C}_b(\mathbf{x}^*)$ be predicted class of tree T_b. Then $\hat{C}_{rf}^B(\mathbf{x}^*) = \text{majority vote}\{\hat{C}_b(\mathbf{x}^*)\}_1^B$.

It is a widely held belief by statisticians that if a classifier interpolates all the data, that is, it fits all the training data without error, then it cannot be consistent and should have a poor generalization error rate. In this section, we demonstrate that there are interpolating classifiers that defy this intuition: in particular random forests will serve as leading example of such classifiers.

Spiked-smooth Classifier: Let's understand the concept of spike-smooth classifier with the help of the example present in the paper.

Suppose we are trying to predict a continuous response y based on real-valued x observations. Let us assume that the true underlying model is $y = x + \varepsilon$, where ε is a mixture of a point mass at zero and some heavy-tailed distribution. In other words, we'll assume that most points in a given training set reflect the true linear relationship between y and x, but a few observations will be noise points. This is analogous to the types of probability models we typically consider in classification settings, such as those found in later sections of the paper. Figure 1 shows hypothetical training data: note that the only "noise" point is found at $x = 0.4$. We then consider fitting three models to this data: two interpolating functions, given by the blue and black lines, and an ordinary regression fit given by the red line. The first thing to notice is that the two interpolating fits differ only from the true target mean model $y = x$ only at the noise point $x = 0.4$. In contrast, the fit of the regression line deviates from the underlying target over the entire range of x. The one noise point corrupted the entire fit of the regression line, while the interpolating lines were able to minimize the influence of the noise point by adapting to it only very locally. Moreover, one should note that between the two interpolating fits, the blue line interpolates more locally than the black line, and thus its overall fit is even less influenced by the noise point. This simplified example is of course meant to be didactic, but we will show throughout the rest of this paper that in practice AdaBoost and random forest do indeed produce

fits similar to the blue line.

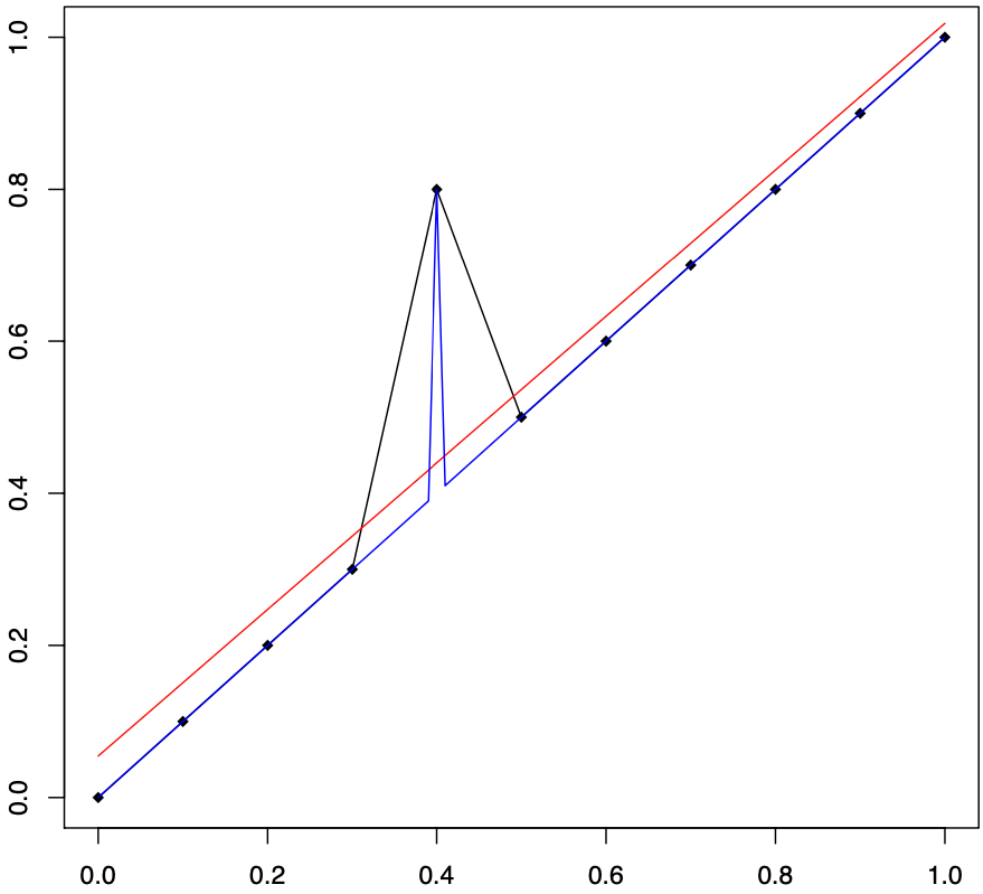
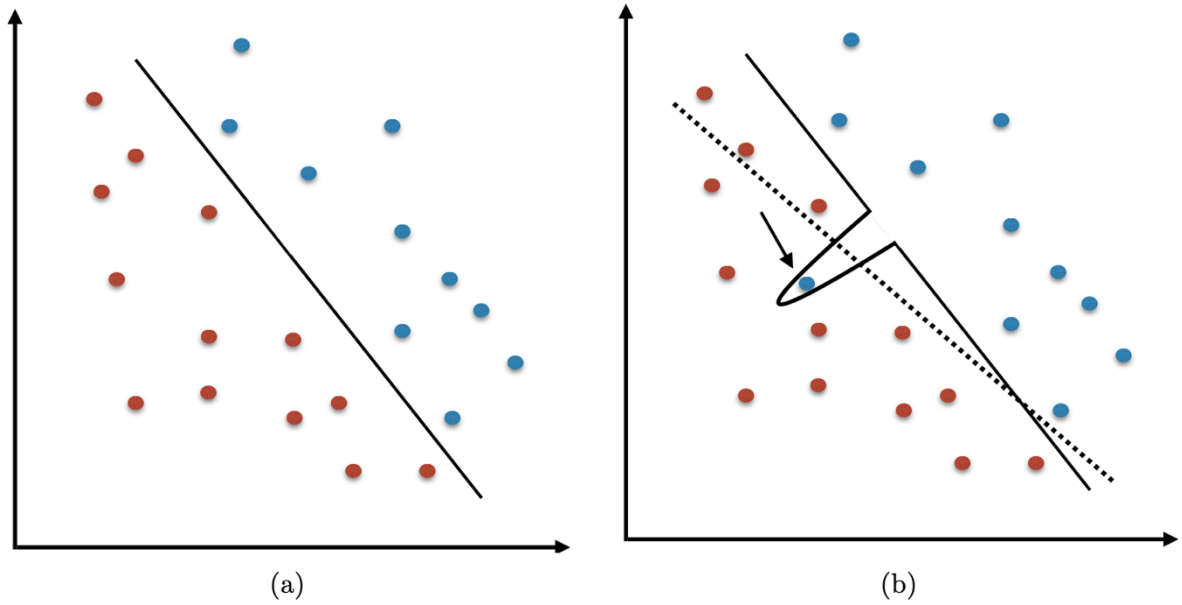


Figure 1: Three estimated regression functions with two interpolating fits (black and blue) and ordinary least squares fit (red).

While it is conceptually clear that it is desirable to produce fits like the blue interpolating line in the previous example, one may wonder how such fits can be achieved in practice. In the classification setting, we will argue throughout this paper that this type of fit can be realized through the process of averaging interpolating classifiers. We will refer to the decision surface produced by this process as being spiked-smooth. The decision surface is spiked in the sense that it is allowed to adapt in very small regions to noise points, and it is smooth in the sense that it has been produced through averaging. A technical definition of a spiked-smooth decision surface would lead us too far astray. It may be helpful instead to consider the types of classifiers that do not produce a spiked-smooth decision surface, such as a logistic regression. Logistic regression separates the input space into two regions with a hyperplane, making a constant prediction on each region. This surface is not spiked- smooth because it does not allow for local variations: in large regions (namely, half-spaces), the classifier's predictions are constrained to be the same

sign. Figure 2 provides a graphical illustration of this intuition.



3. Ada boost as a self-averaging and interpolating algorithm:

Self Averaging:

we explain why the additional iterations in boosting way beyond the point at which perfect classification of the training data (i.e interpolation) has occurred actually has the effect of smoothing out the effects of noise rather than leading to more and more overfitting. To the best of our knowledge, this is a novel perspective on the algorithm. To explain our key idea, we will recall the pure noise example from before with p = .8, d = 20 and n = 5000.

Recall that the classifier produced by AdaBoost corresponds to $I[f_M(x) > 0]$ where

$$f_M(x) = \sum_{m=1}^{M} \alpha_m G_m(x)$$

as defined earlier. Taking M = 1000 which was successful in our example let us rewrite this as

$$f_{1000}(x) = \sum_{m=1}^{1000} \alpha_m G_m(x) = \sum_{j=1}^{10} \sum_{k=1}^{100} \alpha_{100(j-1)+k} G_{100(j-1)+k}(x) = \sum_{j=1}^{10} \sum_{k=1}^{100} h_k^j(x)$$

where

$$h_k^j(x) \equiv \alpha_{100(j-1)+k} G_{100(j-1)+k}(x).$$

Now define

$$h_K^j(x) \equiv \sum_{k=1}^{K} h_k^j(x)$$

and note that for every j ∈ {1,...,10} and every K ∈ {1,...100} that $I[h^j_K(x) > 0]$ is itself a classifier made by linear combinations of classification trees.

The second classifier in the decomposition is simply AdaBoost weight carried over from the first classifier. Since re- weighting the training data does not prevent AdaBoost from obtaining zero training error, the second classifier also interpolates eventually, as does the third, and so on.

Decomposing boosting in this way offers an explanation of why the additional iterations lead to robustness and better performance in noisy environments rather than severe over- fitting. In this example, AdaBoost for 100 iterations is an interpolating classifier. It makes some errors, mostly near the points in the training data for which the label differs from the Bayes rule, although these are localized. Boosting for 1000 iterations is thus a point-wise weighted average of 10 interpolating classifiers. The random errors near the points in the training data for which the label differ from the Bayes' rule cancel out in the ensemble average and become even more localized. Of course, the final classifier is still an interpolating classifier as it is an average of 10 interpolating classifiers. In this way, boosting is self- smoothing, self-averaging or self-bagging process that reduces overfitting as the number of iterations increase.

Interpolating:

We will now consider a second simulation to further illustrate how this self-averaging property of AdaBoost helps prevent overfitting and improves performance. In this simulation we add signal while retaining significant random noise. Let n = 400, d = 5 and sample x_i distributed I uniform on $[0,1]^5$. The true model from for the simulation is

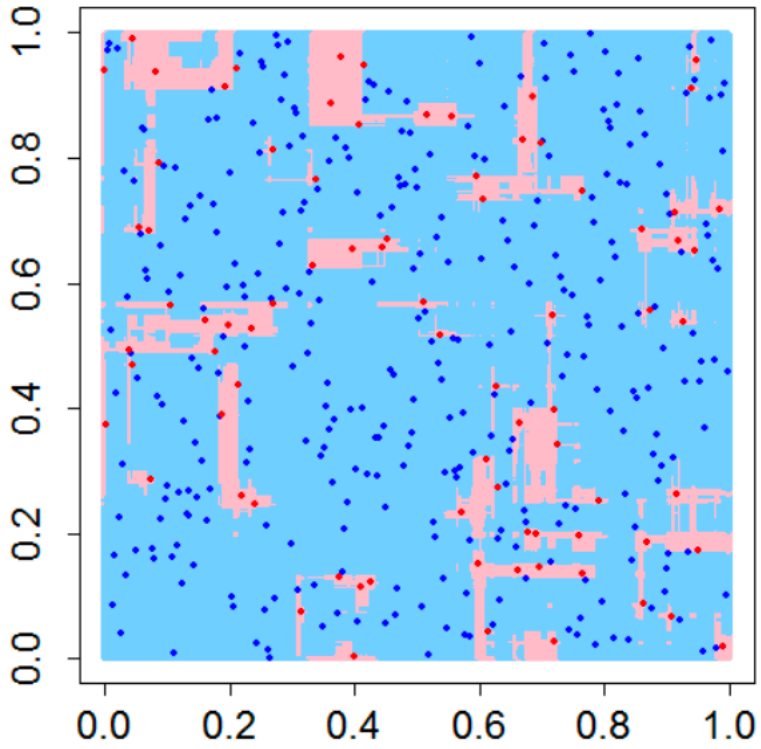
$$\mathbb{P}(y = 1|\mathbf{x}) = .2 + .6\ \mathrm{I}\left[\sum_{j=1}^{2} x_j > 1\right].$$

The Bayes' error is 0.20 and the optimal Bayes' decision boundary is the diagonal of the unit square in x_1 and x_2. Even with this small sample size, AdaBoost interpolates the training data after 10 iterations. So we boost for 100 iterations which decomposes into ten sets of ten. In this example The correlations ranged from 0.4 to 0.56, with an average value of 0.488. As a comparison, we also considered a similar calculation for the decomposition produced from a random forest with 500 trees, and from bagging 1000 depth 8 trees. As with AdaBoost, we can also decompose these classifiers into a sum of interpolating classifiers: 25 sub-ensembles of 20 trees in the case of the random forest, and 10 sub-ensembles of trees in the case of bagged trees. The correlations for the bagged trees ranged from 0.9 to 0.94, with an average value of 0.92. This proves Adaboost is an interpolating algorithm too.

Spike Smooth Classifier:

The implementation of AdaBoost is carried out according to the algorithm described earlier. The base learners used are trees fit by the rpart package (Therneau and Atkinson, 1997) in R. The trees are grown to a maximum depth of 8, meaning they may have at most $2^8 = 256$ terminal nodes. This will be the implementation of AdaBoost we will consider throughout the remainder of this paper.

We will consider again the "pure noise" model as described in the previous section, where the probability that y is equal to +1 for every x is some constant value $p > .5$. For the training data we will take $n = 400$ points uniformly sampled on $[0, 1]^2$ according to a Latin Hypercube using the midpoints as before. For the corresponding y values in training data we will randomly choose 80 points to be −1's so that $P(y = 1|x) = .8$.



Performance of AdaBoost on a pure noise response surface with P (y = 1|x) = .8 and n = 400 training points. Regions classified as +1 are colored light blue and regions classified as −1 are colored pink. The training data is displayed with blue points for y = +1 and red points for y = −1. Since the Bayes' rule would be to classify every point as +1, we judge the performance of the classifiers by the fraction of the unit square that matches the Bayes' rule. AdaBoost performs substantially better classifying 87% of the square as +1 after 100 iterations (which is long after the training error equals zero). This is evidence of boosting's robustness to noise. Visually, it is obvious that the AdaBoost classifier is spiked-smooth, which allows it to be less sensitive to noise points.

4. From the paper we get to know why AdaBoost and random forests are successful self-averaging, interpolating and spike-smoothed classifiers. The literature on AdaBoost focuses on classifier margins and boosting's interpretation as the optimization of an exponential likelihood function. A random forest is another popular ensemble method for which there is substantially less explanation in the literature. We introduce a novel perspective on AdaBoost and random forests that proposes that the two algorithms work for similar reasons. While both classifiers achieve similar predictive accuracy, random forests cannot be conceived as a direct optimization procedure. Rather, random forests are self- averaging, interpolating algorithm which creates what we denote as a spiked-smooth classifier, and we view AdaBoost in the same light. We conjecture that both AdaBoost and random forests succeed because of this mechanism. We provide a number of examples to support this explanation. In the process, we question the conventional wisdom that suggests that boosting algorithms for classification require regularization or early stopping and should be limited to low complexity classes of learners, such as decision stumps.

We conclude that boosting should be used like random forests: with large decision trees, without regularization or early stopping.

AdaBoost is an undeniably successful algorithm and random forests is at least as good, if not better. But AdaBoost is as puzzling as it is successful; it broke the basic rules of statistics by iteratively fitting even noisy data sets until every training set data point was fit without error. Even more puzzling, to statisticians at least, it will continue to iterate an already perfectly fit algorithm which lowers generalization error. The statistical view of boosting understands AdaBoost to be a stage wise optimization of an exponential loss, which suggest (demands!) regularization of tree size and control on the number of iterations. In contrast, a random forest is not an optimization; it appears to work best with large trees and as many iterations as possible. It is widely believed that AdaBoost is effective because it is an optimization, while random forests works well because it works. Breiman conjectured that "it is my belief that in its later stages AdaBoost is emulating a random forest". This paper sheds some light on this conjecture by providing a novel intuition supported by examples to show how AdaBoost and random forest are successful for the same reason.

A random forests model is a weighted ensemble of interpolating classifiers by construction. Although it is much less evident, we have shown that AdaBoost is also a weighted ensemble of interpolating classifiers. Viewed in this way, AdaBoost is a "random" forest of forests. The trees in random forests and the forests in the AdaBoost each interpolate the data without error. As the number of iterations increase the averaging of decision surface because smooths but nevertheless still interpolates. This is accomplished by whittling down the decision boundary around error points. We hope to have cast doubt on the commonly held belief that the later iterations of AdaBoost only serve to overfit the data.

Instead, we argue that these later iterations lead to an "averaging effect", which causes AdaBoost to behave like a random forest.

A central part of our discussion also focused on the merits of interpolation of the training data, when coupled with averaging. Again, we hope to dispel the commonly held belief that interpolation always leads to overfitting. We have argued instead that fitting the training data in extremely local neighborhoods actually serves to prevent overfitting in the presence of averaging. The local fits serve to prevent noise points from having undue influence over the fit in other areas. Random forests and AdaBoost both achieve this desirable level of local interpolation by fitting deep trees. It is our hope that our emphasis on the "self-averaging" and interpolating aspects of AdaBoost will lead to a broader discussion of this classifier's success that extends beyond the more traditional emphasis on margins and exponential loss minimization.

5. Random Forest is an independent and equal-vote learning structure. Adaboosting is a sequentially dependent, weighted learning structure. Both of them can achieve "self-averaging, interpolating, and spiked-smooth classification".

In my opinion another good example for self averaging, interpolating and spike smoothed classifier can be XG-boost.

XGBoost is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework. In prediction problems involving unstructured data (images, text, etc.) artificial neural networks tend to outperform all other algorithms or frameworks. However, when it comes to small-to-medium structured/tabular data, decision tree-based algorithms are considered best-in-class right now. Please see the chart below for the evolution of tree-based algorithms over the years.

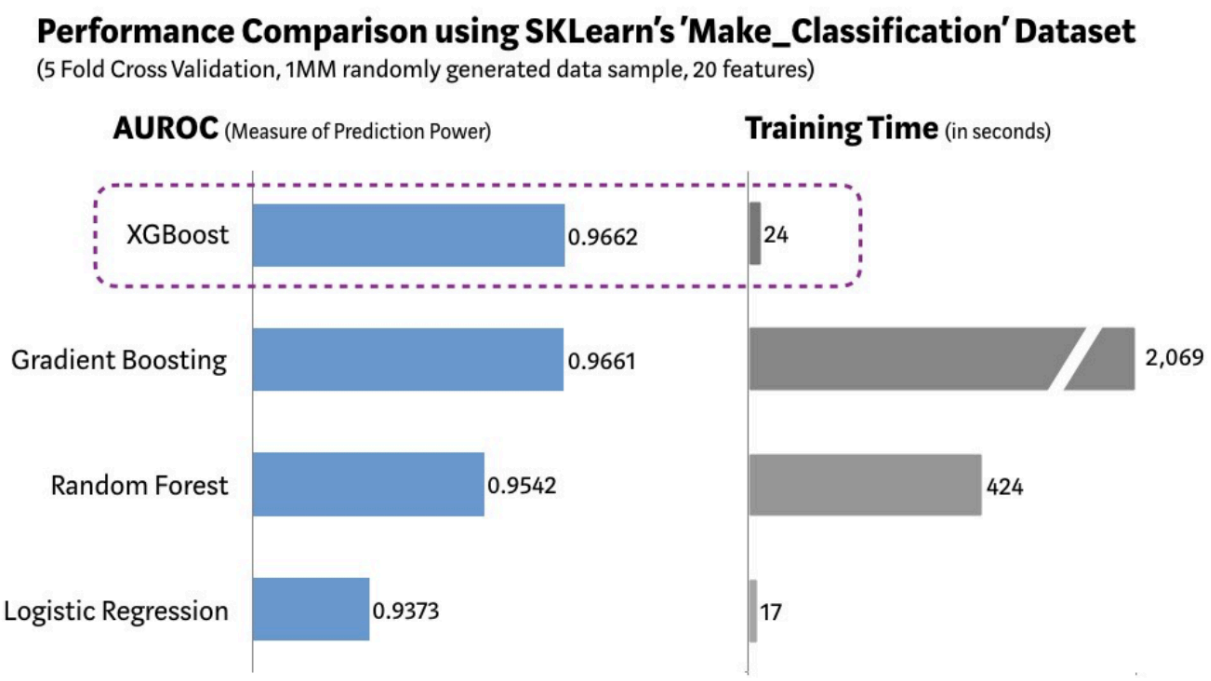
Algorithmic Enhancements:

1. **Regularization**: It penalizes more complex models through both LASSO (L1) and Ridge (L2) regularization to prevent overfitting.

2. **Sparsity Awareness**: XGBoost naturally admits sparse features for inputs by automatically 'learning' best missing value depending on training loss and handles different types of sparsity patterns in the data more efficiently.

3. **Weighted Quantile Sketch:** XGBoost employs the distributed weighted quantile algorithm to effectively find the optimal split points among weighted datasets.

4. **Cross-validation**: The algorithm comes with built-in cross-validation method at each iteration, taking away the need to explicitly program this search and to specify the exact number of boosting iterations required in a single run.



XGBoost vs. Other ML Algorithms using SKLearn's Make_Classification Dataset

As demonstrated in the chart above, XGBoost model has the best combination of prediction performance and processing time compared to other algorithms. Other rigorous benchmarking studies have produced similar results. No wonder XGBoost is widely used in recent Data Science competitions.

Advantages:

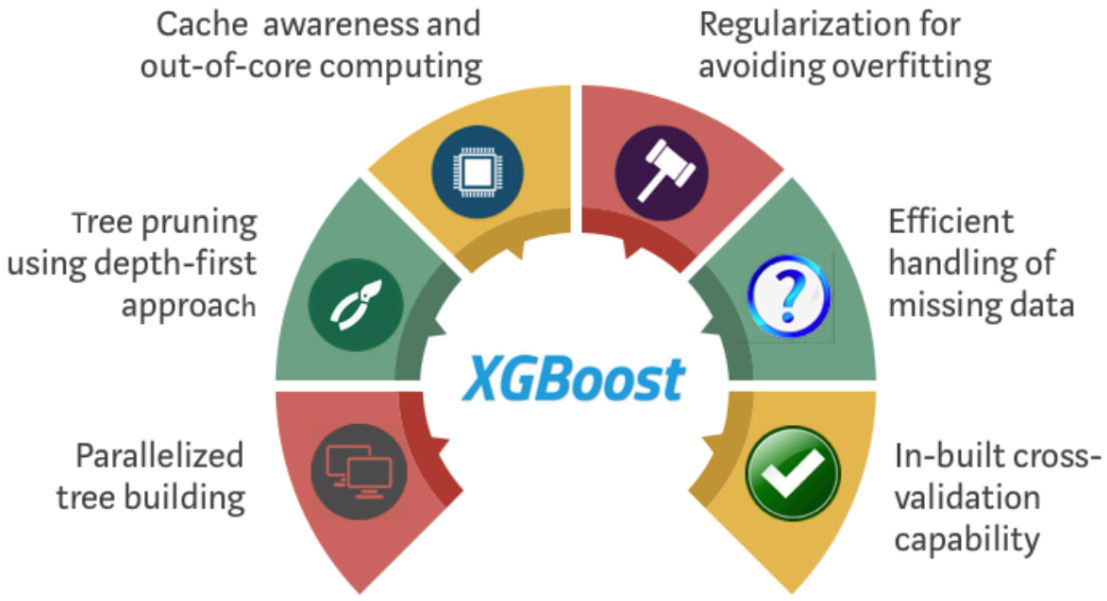
- It is Highly Flexible
- It uses the power of parallel processing
- It is faster than Gradient Boosting
- It supports regularization
- It is designed to handle missing data with its in-build features.
- The user can run a cross-validation after each iteration.
- It Works well in small to medium dataset

XGBoost:

Taking the best parts of AdaBoost and random forests and adding additional features:

- **Sequential tree growing**
- Minimizing loss function **using gradient descent**
- **Parallel processing** to increase speed
- **Regularization** parameter

Overview of the most relevant features of the XGBoost algorithm. Source: Julia Nikulski.



How XGBoost optimizes standard GBM algorithm