Empirical Comparison of Supervised Learning Algorithms

Vishnu Nandurkar

Abstract:

In this paper, an analysis of three supervised learning algorithms (Boosted Trees, Random Forests, and Logistic Regression) is presented across three datasets. The aim is to demonstrate that each algorithm's expected performance aligns with the results reported in this paper. The changes in each algorithm's performance across all trials will be explained. The amount of training and test data is varied to see how each algorithm performs with different amounts of training data. The optimal hyperparameters for each algorithm, after tuning, will also be reported.

1. Introduction:

Classification is a fundamental application of supervised machine learning algorithms. These algorithms have wide array of applications such as medical diagnosis, prediction tools and detection systems. Throughout the past two decades, several such algorithms have been made, and they all are proficient in different applications. They all have their strengths and weaknesses, and this paper aims to highlight three supervised algorithms. They are Boosted trees (XGBoost), Random Forests and Logistic Regression. These three have been chosen because when their results are compared to each other, some important details about them are revealed.

For evaluation, these three datasets from the UCI data repository[1] were chosen: ADULT dataset, Breast Cancer Wisconsin dataset and Bank Marketing dataset. They are all binary classification problems, but these datasets vary in number and type of features and total size. This helps in effectively testing an algorithm's performance on a variety of problems which reduces bias giving more robust results.

The performance metric used in this paper is the accuracy score. The tuned hyperparameters obtained after cross validation are also reported alongside the accuracy score. This is done to provide more information regarding the tendency of an algorithm to perform better on average with a certain combination of hyperparameters.

The algorithms performed as expected in this paper. Performance of XGBoost is on average better compared to the two other algorithms on all three datasets across all combinations of data partitions. Next, the random forest algorithm's performance on a few of the partitions stood out, but the training time takes much longer compared to the other two algorithms.

2. Methodology:

First, the dataset was divided into two sets, one for training and one for testing. This division was based on three types of partitions. The training and test sets were partitioned as follows: (20%, 80%), (50%, 50%), (80%, 20%). Three trials were conducted for each kind of partition. A dictionary called 'parameter grid' containing possible values for each hyperparameter was defined before cross-validation. K-fold cross-validation (k = 5) was done on the training set for hyperparameter tuning. GridSearchCv from

sklearn.model_selection was used to find the best possible hyperparameter combination from the parameter grid for a given trial and partition. Then, the best hyperparameters were used to generate predictions for the training and test data. Based on these predictions, accuracy scores were generated for training, testing, and cross-validation (this accuracy score was based on the best set of hyperparameters as well). Average accuracy scores were calculated for the three trials of each partition. Optimal hyperparameters were also recorded and reported for each trial.

A pipeline (sklearn.Pipeline) was made for preprocessing the data and applying the required classifier. This just helps organize code and maintain a cohesive workflow. The pipeline was composed of two steps. The first was the preprocessing step, in which all numerical columns were standardized using StandardScaler from sklearn.preprocessing, and categorical columns (if any) were one-hot encoded using OneHotEncoder from sklearn.preprocessing. This two preprocessing components were defined in a 'transformers' list in a ColumnTransformer() from sklearn.compose. This helps in applying these two preprocessing methods to the required columns of the dataset.

The second step of the pipeline was the classification step in which the required classifier was defined with certain fixed hyperparameters that don't affect the learning process. For example, fixing the random_state hyperparameter ensures that observed differences in performance are due to changes in hyperparameters from the parameter grid and not random variations in the training process. For all classifiers, random_state was set to 42+ith trial, where i = 1, 2, or 3. To handle the class imbalance (significantly more items in the negative class) in the third dataset, the class_weight hyperparameter for logistic regression and random forest was set to 'balanced'. To handle the same problem with the third dataset for XGBoost classifier, the count of negative samples was divided by the count of positive samples and the scale_pos_weight hyperparameter was set to this ratio. This fixed the unbalancing as the minority positive class was given more importance (weight) which prevents the model from ignoring it.

Below are listed the hyperparameters related to the learning process for each algorithm and the range of values GridSearchCv used for tuning:

Classifier	Hyperparameter	Description of Hyperparameter	Values in parameter grid
XGBoost	n_estimators	Maximum number of boosting trees	100, 150 & 200
	learning_rate	The step-size shrinkage to reduce overfitting	0.1 & 0.2
	max_depth	Maximum height of the tree	3, 4 & 5
	subsample	Fraction of samples used to build each tree.	0.8 & 1.0
Logistic Regression	C	Regularization strength.	0.001, 0.01, 0.1, 1, 10 & 100
	penalty	Type of regularization	'l2' & 'l1'
	solver	Optimization algorithm used to find the model's coefficients.	'lbfgs', 'saga' & 'liblinear'
Random Forest	n_estimators	Number of trees	50, 100 & 200
	max_depth	Maximum depth of each individual tree	None, 5, 10, 15, 25 & 30

	min_samples_split	Minimum number of samples required to split an internal node.	2, 5 & 10
	min_samples_leaf	Minimum number of samples required to be at the led node.	1, 2 & 4

3. Datasets:

1. **ADULT**: This dataset has a binary classification task: predicting whether each individual's annual income is over or under 50k dollars. The features, which are a mix of categorical and numerical, are information about each individual (census data). The dataset is 32561x15, with the last column as the target variable. So, in total, there are 14 features. After dropping rows with missing values, the size of the dataset used for training was 30162x15.

2. **Breast Cancer Wisconsin (BCW):** This dataset is used for binary classification: predicting whether a patient has breast cancer. The features are description of cell nuclei present in the image. The features are all numerical/continuous. The dimensions of the dataset are 569x32. The first column, the ID column, didn't assist with classification and was dropped. Another column is the diagnosis column, which indicates whether the tumor was present. So, there are 30 features in the dataset. There are no missing/incomplete values so now rows were dropped.

3. **Bank Marketing (BM):** This dataset is used for binary classification: predicting whether a given client would subscribe to a term deposit. The features (mix of categorical and numerical) provide information regarding the clients who were contacted. The total size of the dataset is 41188x21. The 'cons.price.idx' were removed because it contained missing values and didn't provide any valuable information for classification. The 'duration' column in BM strongly affects the target attribute and so to get a realistic predictive model, this feature was dropped. For example, if the duration = 0 then the target variable automatically becomes a 'no'. So, 18 features were used for training the model.

4. Experimental Results:

Below is the **average test set** accuracies of all the models on the three datasets across all partitions:

20/80 split:

	ADULT	BCW	BM
XGBoost	**0.861**	0.953	0.866
Logistic Regression	0.844	**0.966**	0.822
Random Forest	0.854	0.945	**0.877**

50/50 split:

	ADULT	BCW	BM
XGBoost	**0.868**	0.971	0.852
Logistic Regression	0.844	**0.977**	0.831
Random Forest	0.857	0.959	**0.873**

Below are the **average training set** accuracies of all the models on the three datasets across all partitions:

20/80 split:

	ADULT	BCW	BM
XGBoost	0.890	**1.00**	**0.982**
Logistic Regression	0.855	0.982	0.823
Random Forest	**0.896**	0.997	0.923

50/50 split:

	ADULT	BCW	BM
XGBoost	0.885	**1.00**	**0.944**
Logistic Regression	0.852	0.989	0.831
Random Forest	**0.897**	0.998	0.901

80/20 split:

	ADULT	BCW	BM
XGBoost	**0.865**	0.965	0.849
Logistic Regression	0.846	**0.977**	0.827
Random Forest	0.859	0.956	**0.870**

80/20 split:

	ADULT	BCW	BM
XGBoost	0.885	**1.00**	**0.897**
Logistic Regression	0.850	0.988	0.826
Random Forest	**0.901**	0.999	0.890

Below given are the **average cross validation accuracies**:

20/80 split	ADULT	BCW	BM
XGBoost	**0.867**	0.924	0.870
Logistic Regression	0.851	**0.974**	0.820
Random Forest	0.860	0.938	**0.876**

50/50 split	ADULT	BCW	BM
XGBoost	**0.870**	0.961	0.855
Logistic Regression	0.850	**0.974**	0.830
Random Forest	0.861	0.960	**0.872**

80/20 split	ADULT	BCW	BM
XGBoost	**0.870**	0.969	0.850
Logistic Regression	0.849	**0.980**	0.827
Random Forest	0.861	0.965	**0.869**

(Bolded are the best avg acc are best accuracies in any given column).

5. Analysis:

The results show that a model's performance depends heavily on the dataset and the amount of available data. XGBoost and Random Forests usually achieve the best test scores on the more complex ADULT and BM datasets, but they often overfit, especially with less training data. This is shown by the big gap between their nearly perfect training scores and lower test results. Logistic Regression, in contrast, generalizes well, especially on the simpler, linearly separable BCW dataset. On this dataset, it often does better than the more complex models and shows little overfitting. The impact of different split ratios is clear: with less training data (20/80 split), tree-based models overfit more, while more data (50/50 and 80/20 splits) leads to more stable test results and a smaller gap between training and test scores. Cross-validation scores usually match the test results, but can sometimes be too optimistic, so it is important to use hold-out validation. These findings show that the model with the highest test accuracy is not always the best choice. For example, XGBoost overfits badly on BCW, even though it gets perfect training scores. So, choosing a classifier entails balancing accuracy, simplicity, and generalization. Logistic Regression works well for simpler problems, while ensemble methods like Random Forest are better for complex tasks if there is enough data and regularization to prevent overfitting.

6. Conclusion:

The analysis found that no single algorithm is always the best choice. The best model depends on the dataset and the amount of data available. XGBoost usually had the highest test accuracy, especially with larger training sets, but it often overfitted, especially on simpler datasets like BCW. In those cases, Logistic Regression performed better and was more generalizable. Random Forest struck a good balance between performance and stability on the more challenging BM dataset, though it needed more computing power. These results show that factors such as dataset complexity, linear separability, and dataset size are important when selecting a classifier. Practitioners should consider context and look beyond accuracy, also considering overfitting, interpretability, and computational efficiency when choosing supervised learning algorithms.

References:

[1] *UCI Machine Learning Repository*. (n.d.). http://archive.ics.uci.edu/ml/

[2] Caruana, R., Niculescu-Mizil, A., & Department of Computer Science, Cornell University, Ithaca, NY 14853 USA. (2006). An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd International Conference on Machine Learning*. https://www.cs.cornell.edu/~caruana/ctp/ct.papers/caruana.icml06.pdf

Appendix:

Optimized Hyperparameters are printed in this code repository:

https://github.com/Subtilizer0102/Cogs_118a_proj/tree/main