The XGBoost Boosting Algorithm and an Application to the Forest Cover Type Dataset

Matthias Quinn

Matthew Brigham

STA 685 - Advanced Data Models

Fall 2021

Abstract

This project will focus on the famous XGBoost system and its application on a moderately large dataset. Starting with a history of the system, then exploring the algorithm itself, and finally ending with an application to forest cover types, this project will hopefully provide a framework on which to base further research and applications.

Table of Contents

Introduction

Tree-based methods are popular due to their superior performance on tabular data. They work by repeatedly dividing the predictor space into regions in order to make predictions for both classification and regression problems. The most basic type of tree is the decision tree. The decision tree is simple to implement and to interpret. However, it usually doesn't perform as well as other supervised methods in terms of accuracy. Bagging, boosting, and ensembling are all ways to improve the performance of decision trees.

Boosting is a method of iteratively improving a model. One benefit of boosting is that both the bias and variance of the model is controlled, as opposed to just one. There are several popular algorithms for boosting: AdaBoost, Gradient Boosting, XGBoost, and others. One of the issues with most boosting algorithms is the computation time, which is why the XGBoost algorithm has gained so much traction over recent years.

From Tianqi Chen and Carlos Guestrin in March of 2014, XGBoost has been a monumental system in the field of advanced analytics and machine learning. XGBoost was initially a research project started by Tianqi Chen - and later Carlos Guestrin - at the University of Washington, who presented their paper at SIGKDD Conference in 2016. Since then, version 1.5.0 is available for public usage and has been implemented in more than 5 languages including: C++, Python, R, Java, Scala, and Julia. In addition, the system is available to all modern operating systems, including: Windows, OS X, Linux, and a variety of cloud platforms.

Background

Decision Trees

Decision Trees are simple clustering algorithms that split the predictor space into distinct, non-overlapping regions. Trees are easy to visualize, are flexible since they make no assumption about the functional form, and can model highly non-linear relationships. They can also be used in the context of regression and classification.

Generally speaking, the algorithm works by searching through the available predictors and selecting the one that splits the leads to the greatest reduction in residual sums of squares (RSS), called recursive binary splitting. However, the model does not look beyond a single split to see if other predictors may lead to a greater overall reduction of RSS over the course of many splits. This is called a greedy algorithm. Considering Figure 1 below, we see that decision trees are a top-down approach, meaning that they start with the full predictor space then, moving down, create rules that split the space into distinct regions. The end result is a set divided predictor space of non-overlapping regions, as seen on the left. The splitting process continues until a criterion is met, whether it be a defined maximal depth, number of observations per leaf, or number of observations per split.

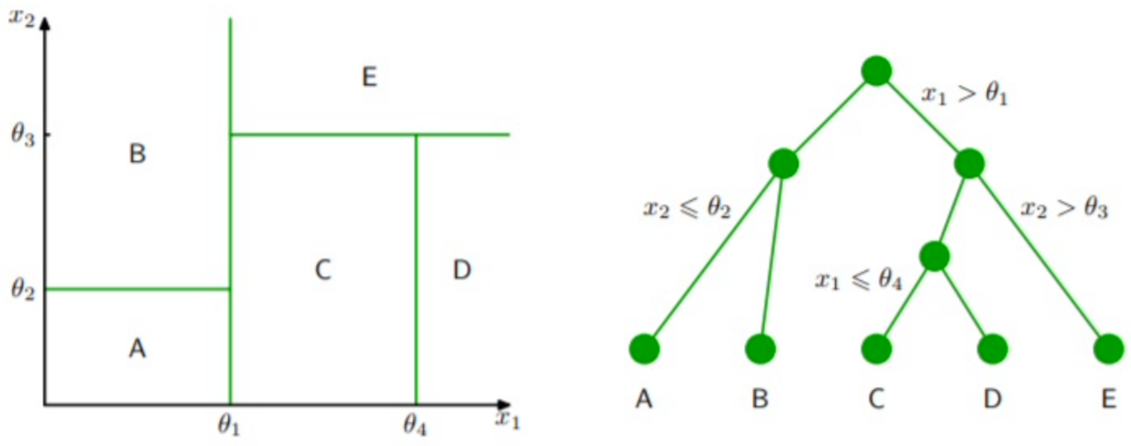

Figure 1: Decision trees divide the predictor space into non-overlapping regions by using splitting rules. The image on the left shows a divided predictor space. The tree on the right shows how splitting rules are used. (Image from Karlicek, 2021)

Tree pruning is a method that can improve the predictive accuracy of decision trees. Pruning refers to the reduction in depth of the tree. One concern about decision trees is overfitting by modeling noise or being too complex. Utilizing a process known as cost complexity pruning, the optimal tree depth may be determined. This process considers the prediction error and the number of terminal nodes, as illustrated in the equation below. Cost complexity pruning aims to minimize this equation. The tuning parameter, α, controls the complexity, or depth, of the tree. A large α will result in a deep, complex tree (less pruning) and a small α will result in a shallow tree (more pruning). In general, shallow trees cannot model all of the patterns in the data and will have higher prediction error at the cost of low complexity. Deep trees can model the patterns in the data at the cost of high complexity and potential overfitting. There exists a happy medium between the two.

$$\sum_{m=1}^{|T|} \quad \sum_{x_i \in R_m} \quad (y_i - \hat{y}_{R_m})^2 + \alpha|T|$$

Classification requires small modifications to the fitting process compared to regression. Notably, instead of considering RSS, classification trees may consider either classification error, entropy, or the Gini Index. When the goal of modelling is prediction accuracy, it is generally advisable to minimize classification error for the splits. Classification error looks at the proportion of observations in each region belonging to a certain class and compares this to the most common classification of observations in that region. The equation below is used to calculate the classification error given region *m* and class *k*.

$$CE = 1 - max(\hat{p}_{mk})$$

Entropy and the Gini Index are most commonly used for classification and are very similar in that they consider node purity. Node purity refers to the assortment of observations in a node. If all of the observations in a node have the same class label, then it is said to have high purity. If there are many different observation classes in a node, then it is not pure. High node purity is generally preferred. The equations for entropy and Gini Index are below. Just as regression trees aim to minimize RSS, classification trees aim to minimize entropy and Gini.

$$Gini = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

$$Entropy = -\sum_{k=1}^{K} \hat{p}_{mk} log(\hat{p}_{mk})$$

Decision trees can be advantageous since they are easy to interpret and can model a variety of data types. However, they have some weaknesses. A primary weakness is that they can have a high degree of variance. Another weakness is that they are not robust in the sense that the addition/removal of a single observation may result in a very different tree shape. There are several ways to handle these concerns, commonly bagging, random forests, and boosting. We will discuss boosting, in particular.

Boosting

Boosting is a general ensemble algorithm that can be used to improve the predictive accuracy of decision trees. Boosting assumes a generic function then fits a shallow tree of the residuals. After incorporating the new tree to get an updated function, another shallow tree is fit to the updated function's residuals. This process repeats until a criterion is met. Just like decision trees, the final resulting model is a single tree.

The boosting algorithm has three main steps:

1. Define an initial model to be fit to the data.
2. Iteratively fit B-many shallow trees to the residuals of the previous model and add them to the previous model
    a. Fit a tree with d-many splits to the prior model's residuals
    b. Update the previous model by adding a shrunken version of the shallow tree.
    c. Update the residuals by subtracting the shrunken version of the shallow tree
3. After B-many shallow trees have been fit to the residuals, the result is a single boosted model.

From the algorithm, we can see that the boosted model is a single tree that consists of a collection of weak learners (shallow trees). It is powerful because it can control for both bias and variance. Each addition of a weak learner aims to reduce the bias of the model. The collection of weak learners has the effect of reducing the variance of the model. Even though it controls for both, most of the focus is on bias reduction. If the initial function in the first step of the algorithm is $\hat{f}(x) = 0$, then the initial residuals are the *y*-values. When this is the case, the final boosted model will be the following equation, where $\widehat{f_b}(x)$ is the weak-learning decision tree from each iteration.

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \widehat{f_b}(x)$$

There are three tuning parameters for boosting: *B*, λ, and *d*. The parameter *B* represents the number of shallow trees that will be fit to the residuals. This value should be determined using cross-validation since a large value may result in overfitting. The parameter *B* is closely tied to the shrinkage parameter λ. The parameter λ is called the shrinkage parameter and this controls the rate at which the model learns. A small lambda value confers a small adjustment to each sequential model. If the value of λ is extremely small, the model will need a large value for *B* in order for the data to be fit accurately. The parameter *d* defines the number of splits in each of the shallow trees. When we are referring to these sequentially added shallow trees, we mean that the value of d is small.

There are several algorithms for implementing boosting. Some example algorithms are AdaBoost, Gradient Boosting, and XGBoost (there are others). AdaBoost is commonly recognized as the first algorithm that successfully implemented the boosting algorithm. Gradient Boosting is a generalization of AdaBoost, and XGBoost is an optimized gradient boosting algorithm. One problem with boosting is that it can be very slow, especially for large datasets. This is part of the reason why XGBoost has become so popular.

XGBoost

There are many algorithms to implement boosting. The first boosting algorithm is recognized as AdaBoost and was mostly used for binary classification. Gradient Boosting is a generalization of AdaBoost and allows for the use of different loss functions. This improvement meant that it could be applied to regression and multi-class classification problems. One issue with Gradient Boosting, which is common among boosting in general, is that the learning process can be very slow. XGBoost is an improved implementation of the Gradient Boosting algorithm optimized for speed and accuracy. It is optimized for speed and has been observed to be more accurate in many cases, which is how it got its name: eXtreme Gradient Boosting. The success of XGBoost is attributed to several key algorithmic improvements:

1. Regularized learning objective
2. Novel tree-based algorithm for handling sparse data
3. Weighted quantile sketch for learning
4. Optimized computer hardware utilization

To implement a boosting algorithm, there is a chosen loss function (usually based on prediction error) that gets minimized. Gradient Boosting uses gradients to accomplish this. XGBoost improves upon this process by using second derivatives. These second derivatives are used to help find optimal splits without looking at every possible split, which saves a lot of time.

XGBoost uses regularization of the loss function. Briefly, boosting works by iteratively adding weak learner shallow trees. This improvement helps to prevent overfitting by smoothing the solutions of terminal nodes in the shallow trees that are used for boosting. Regularization works by adding a penalty to a traditional loss function, usually based on errors or residuals. Common types of regularization are the l_1 and l_2 regularization. The loss function used by XGBoost is given below. The first term is a loss function that measures the difference between the predicted and actual values of *y*. The second term is the regularization term, or penalty, and uses both l_1 and l_2 regularization. The penalty works by reducing the complexity of the model. This rationale is analogous to cost complexity pruning in decision trees. There are alternative methods for regularization, but this method has been shown to be easier to implement for parallelization. Note that when the regularization term is set to zero, the loss function is identical to traditional gradient boosting.

$$Loss = \sum_{i} \quad l(y_i, \widehat{y_i}) + \sum_{k} \quad (\gamma T + \frac{1}{2}\lambda \left|\left|w\right|\right|^2)$$

The second major improvement on boosting is the use of second-order derivatives in the optimization of the loss function. The use of second-derivatives yields more information about how to minimize the loss function faster. Usually for large datasets, it is not possible to explore all possible tree structures. Therefore, the use of the second derivative helps to pick which splits are optimal. The loss function cannot be optimized using traditional methods. The designers of XGBoost proposed an additive training strategy that necessitates the use of the second derivative for large datasets.

A third major feature is shrinkage and feature subsampling to prevent overfitting. Shrinkage refers to the parameter λ used in the boosting algorithm described in the previous section. Feature subsampling is a technique that refers to the construction of the shallow trees during boosting. When these trees are constructed, at each split, only a fraction of the total features are considered. Random forests are known for employing this methodology as it decorrelates successive trees and allows for different patterns to be captured.

There are several splitting algorithms incorporated into the function that can be defined by the user. These splitting algorithms are listed below.

A. Exact Greedy Algorithm: This algorithm considers all of the possible splits for the features and picks a split based on the greatest reduction in the loss function. For continuous variables, the observations are sorted in ascending order then calculates split statistics (such as Sum of Squared Errors) between each observation. For n-many observations and m-many features, there are $(n-1)$ possible splits per feature and $m \cdot (n-1)$ possible splits. For large datasets, this is infeasible and requires the use of the approximate algorithm.

B. Approximate Algorithm: This algorithm improves upon the exact greedy algorithm and can work with large datasets. Instead of exploring all possible splits, the algorithm proposes split points based on quantiles and maps continuous features to these “bins”. Split statistics are calculated for the bins as a whole and then the split point is chosen based on which was best. The quantile strategy is distributable and recomputable, meaning it is faster. The designers also showed that this approximate quantile strategy can yield similar predictive performance as models that use the exact strategy. This quantile strategy is called weighted quantile sketch, and can be used for weighted data.

C. Sparsity-Aware Split Finding: Many datasets contain sparse data (such as missing data and zero-entries from one-hot encoding or other values). By incorporating a pattern recognition algorithm for sparse data, the model can run much faster for datasets with lots of sparse data. XGBoost assigns a default direction to each node when it is sparse. This “unified” approach allows for faster computation times.

The final improvement for XGBoost is the system optimization. Tree construction and sorting data (during the splitting algorithm) are the most computationally intensive and time consuming. Large datasets may not always be able to be modelled on a device. XGBoost was designed to be able to work efficiently on any device and in parallel with other devices in a distributed manner. To take the most advantage of a computer system, large datasets need to first be divided into blocks of data. XGBoost utilizes the following computational system improvements:

A. Column Block for Parallel Learning - To reduce the time of sorting, data is stored in blocks and only sorted once. The data is stored in a compressed column format.

B. Parallelization - The data blocks are distributed among the CPU cores. Therefore, collecting statistics from columns can be done using a parallel algorithm for finding splits.

C. Distributed Computing - Blocking the data allows for it to be distributed among different machines or disks. This allows the model to work on large datasets.

D. Cache Awareness - Storing gradient statistics in the cache of a CPU is computationally more efficient.

E. Blocks for Out-of-Core Computation - Dividing the data into blocks and using computer science techniques known as block compression and sharding allows for the model to work with very large datasets.

There are some limitations to XGBoost. As with most tree-based models, extrapolation can be problematic. There are ways to deal with this by employing density estimation. (Hooker, 2004) Another concern is when the number of features exceeds the number of observations.

In summary, XGBoost is a gradient boosting algorithm that is designed for speed and scalability. It incorporates many different algorithms that the user can choose to use depending on their data structure. XGBoost is commonly used to take advantage of these design features and has been shown to perform just as well, oftentimes better, than other gradient boosting algorithms.

Application: Forest Cover Type

Given forestry data from four wilderness areas in Roosevelt National Forest, classify the patches into one of 7 cover types, listed below:

1. Spruce/fir
2. Lodgepole Pine
3. Ponderosa Pine
4. Cottonwood/Willow
5. Aspen
6. Douglas/fir
7. Krummholz

Data Description

The forest cover type problem requires a prediction of the type of trees that are growing on a plot of land from a variety of descriptive features that affect which species are able to grow in those conditions. An effective model would be able to accurately predict the cover type, allowing researchers to make these predictions without using remotely sensed data. There are 12 predictor variables that can be used to predict the cover type. Table 1 below summarizes each of these predictors. Each observation in the dataset consists of measurements from these variables from a 30 meter by 30 meter plot of land in northern Colorado. There are 581,012 observations in this dataset. The dataset was released in 1998.

Variable Name	Type	Measurement Unit	Description
Elevation	Quantitative	Meters	Elevation in Meters
Aspect	Quantitative	Azimuthal Angle	Aspect in Degrees Azimuth
Slope	Quantitative	Degrees	Slope in Degrees
Horizontal Distance to Hydrology	Quantitative	Meters	Horizontal distance to nearest surface water features
Vertical Distance to Hydrology	Quantitative	Meters	Vertical distance to nearest surface water features
Horizontal Distance to Roadways	Quantitative	Meters	Horizontal distance to nearest roadway
Hillshade 9 AM	Quantitative	0-255 Index	Hillshade index at 9 AM, during summer solstice
Hillshade Noon	Quantitative	0-255 Index	Hillshade index at noon, during summer solstice
Hillshade 3 PM	Quantitative	0-255 Index	Hillshade index at 3 PM, during summer solstice
Horizontal Distance to Fire Points	Quantitative	Meters	Horizontal distance to nearest wildfire ignition points
Wilderness Area	Qualitative	0 (absent), 1 (present)	Wilderness area designation
Soil Type	Qualitative	0 (absent), 1 (present)	Soil type designation
Cover Type	Integer	1 to 7	Forest cover type designation

Table 1: Above is a summary of all of the features in the dataset. The dataset contains a total of 54 features because some of the variables are one-hot encoded, however there are only 13 truly unique features.

The data was collected from four different wilderness areas (out of a total 6 wilderness areas) within the Roosevelt National Forest in Northern Colorado. The cover type was determined by the US Forest Service Region 2 Resource Information System data. A wilderness area is an area that is relatively untouched by humans; this means that any ecological processes are the result of nature rather than forest management services. The four wilderness areas studied in Roosevelt National Forest were Neota, Rawah, Comanche, and Cache la Poudre. These four areas differ in elevation and geography, even though they are nearby, resulting in different species of trees covering the land. Neota consists mostly of spruce/fir. Rawah and Comanche are mostly lodgepole pine, with a smattering of spruce/fir and aspen. Cache la Poudre consists mostly of Ponderosa Pine, Douglas/Fir, and Cottonwood/Willow. It should be noted that approximately 85% of the observations consist of two types of classes. There are no missing values.

Tables 2-3 show some frequency distributions of the cover types and wilderness areas. From these, it can be seen that the dataset is dominated by the two cover types and two wilderness areas. This is attributable to the relative sizes of the wilderness areas. It is important to note the imbalance of observations as this may impact the bias of the model. The Rawah wilderness area is approximately 119.4 sq. miles; the Neota wilderness area is approximately 15.5 sq. miles; the Comanche wilderness area is approximately 104.4 sq. miles; the Cache la Poudre wilderness area is approximately 14.5 sq. miles. From Figure 1, there is a significant proportion of the total observations (approximately 20%) that are in the same soil type and cover type. Figure 2 displays a matrix of correlation values between all of the numeric variables. No significant conclusions regarding the relationship between variables can be concluded. Figure 3 shows that elevation does a great job distinguishing between the cover types. Other variables were explored, but there were no significant patterns identified.

Class	Cover Type	Frequency	Relative Frequency
1	Spruce/fir	211840	0.365
2	Lodgepole Pine	283301	0.488
3	Ponderosa Pine	35754	0.062
4	Cottonwood/Willow	2747	0.005
5	Aspen	9493	0.016
6	Douglas/fir	17367	0.03
7	Krummholz	20510	0.035

Table 2: A table of observation frequencies shows that the observations are dominated by Spruce/Fir and Lodgepole Pine.

Wilderness Area	Frequency
Rawah	260796
Neota	29884
Comanche Peak	253364
Cache la Poudre	36968

Table 3: A table of wilderness area frequencies shows that approximately 88% of the observations came from two of the four areas.

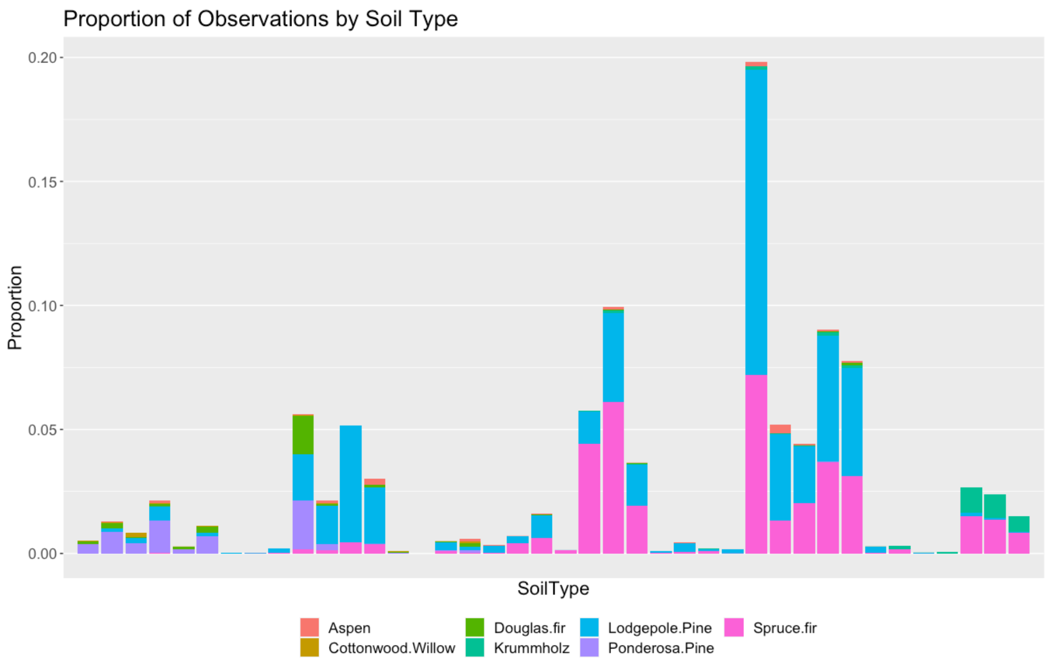

Figure 1: A plot of proportions of cover types grown on a specific soil type shows that some soils are correlated with cover type. Soil type 28 has a huge spike due to Lodgepole Pine. The proportion is out of the total number of observations.

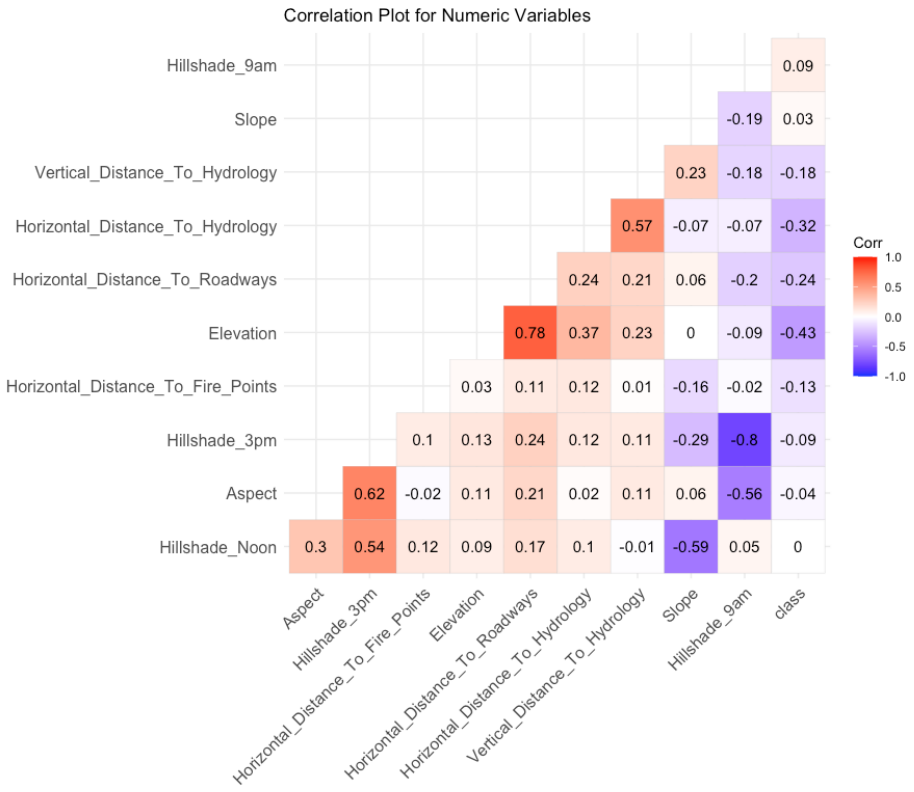

Figure 2: A matrix of correlation values between all numeric variables suggests that there aren’t any strong relationships with the class variable (cover type).

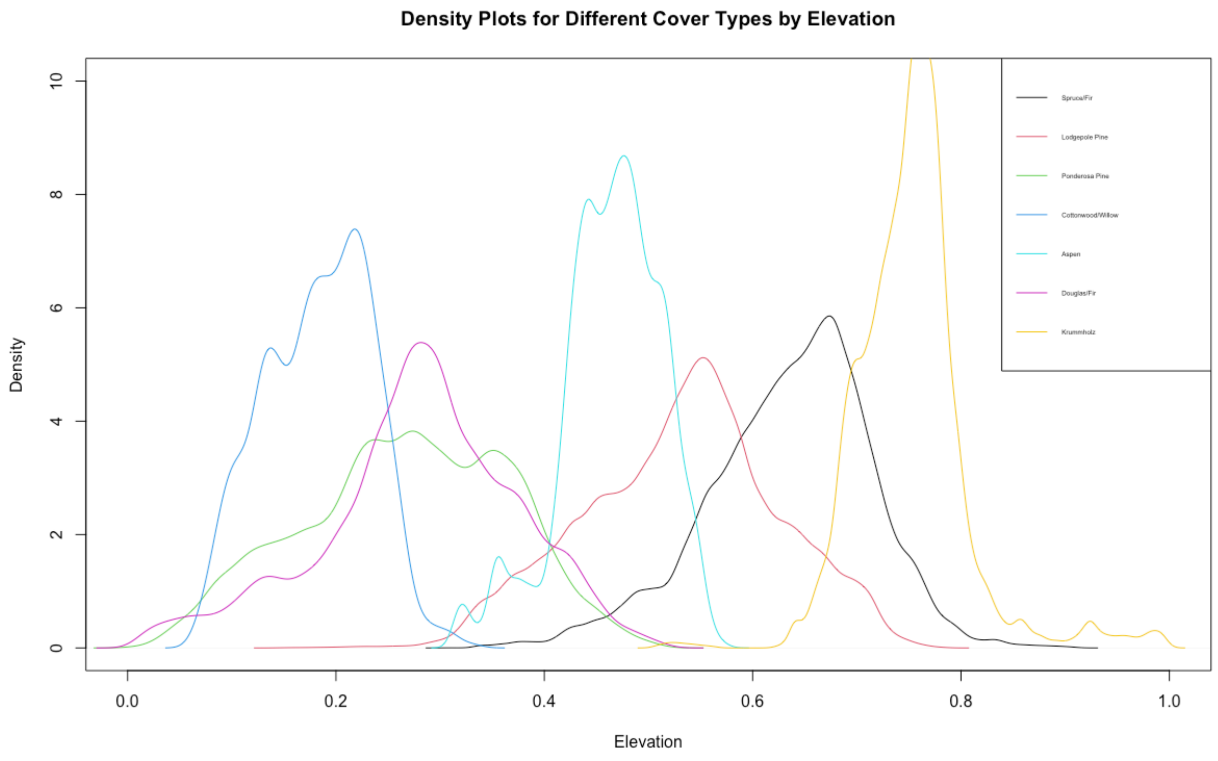

Figure 3: Density plots for the cover types when looking at the elevation. It can be observed that at certain elevations, certain tree species dominate over others. There are also several elevations where there may be a mix or overlap of cover types. Elevation clearly separates the cover types.

Methods

Feature Engineering

Several of the variables were binary variables that needed to be re-coded. The Soil Type feature was one-hot encoded and there were 40 different soil types, or features. To reduce complexity, these 40 features were combined into one column on a scale from 0 to 39. A similar method was applied to the Wilderness Area and Cover Type features. It is important in R to have these variables start at 0 for classification. In addition, the Euclidean Distance to Hydrology feature was calculated from the horizontal and vertical distances, as follows:

$$X_i = \sqrt{(HD_i^{\ 2} + VD_i^{\ 2})}$$

From the exploratory analysis, it is observed that there is an uneven distribution of the data in terms of cover type. Approximately 88% of the data were two of the seven possible cover types. One concern is that this may cause bias in the model towards these two types. To reduce this risk, a balanced random sample was performed so that each cover type was represented evenly. A random sample of size 5000 was chosen from each cover type and was collected for classification. The unused data was used as a test set. It should be noted that one of the cover types (Cottonwood/Willow) only has 2747 observations; since it has less than 5000 observations, an 80/20 train/test split was applied so that the classification accuracy could be determined for this specific cover type. The total size of the training set was 32,197. The test set had 548,815 observations.

Modeling

An XGBoost model was fit onto the training set using the variables: Elevation, Aspect, Slope, Horizontal Distance to Hydrology, Vertical Distance to Hydrology, Horizontal Distance to Roadways, Hillshade 9 AM, Hillshade Noon, Hillshade 3 PM, Horizontal Distance to Fire Points, Wilderness Area (re-coded), Soil Type (re-coded), and Euclidean Distance to Hydrology.

LDA, KNN, Multinomial Logistic Regression, and a Decision Tree will all be fitted and compared to XGBoost. Some of the metrics used for comparison are AUC, predictive accuracy on the test set, and computation time. The formula for classification accuracy is shown below. True positives and negatives are the correctly classified observations.

$$Accuracy = \frac{True\ Positives + True\ Negatives}{Total\ Classified\ Observations}$$

Feature importance can be measured by the gain, coverage, and frequency. A high value for gain implies that the variable contributed a lot to the model. A high value for coverage implies many observations in the various terminal nodes of the shallow trees come from a splitting rule based on that feature. Importance can also be measured based on the frequency that a feature appears in the shallow trees.

Results

The analysis consisted of fitting several models to the training set, then comparing their performance and efficiency. The classification performance of XGBoost can be analyzed using confusion matrices and ROC curves.

Table 4 below summarizes the fit and efficiency of each model tested. It is observed that XGBoost is significantly faster than the others and boasts the highest accuracy. Table 5 shows a confusion matrix for the training set. There were relatively few errors and they were all between two types of cover types.

Model	AUC	Accuracy	Time (sec)
LDA	0.93	0.635	23.16
KNN (k=5)	0.97	0.821	360.53
Multinomial Log. Regression	0.94	0.679	505.57
Decision Tree	0.87	0.553	35.53
XGBoost	0.98	0.868	3.07

Table 4: This table summarizes the performance and computational efficiency of each model when trained on the training set of 32,197 observations.

		Actual Class							
		1	2	3	4	5	6	7	Total
Predicted Class	1	5000	0	0	0	0	0	0	5000
	2	0	5000	0	0	0	0	0	5000
	3	0	0	4993	1	0	0	0	4994
	4	0	0	7	2196	0	0	0	2203
	5	0	0	0	0	5000	0	0	5000
	6	0	0	0	0	0	5000	0	5000
	7	0	0	0	0	0	0	5000	5000
	Total	5000	5000	5000	2197	5000	5000	5000	32197

Table 5: Confusion matrix for the XGBoost model. There are very few misclassifications (8) considering the number of observations, 32197. All of the misclassifications are on two types of forest covers.

Table 6 below shows the feature importances after fitting the XGBoost model. Feature importance can be measured by the gain, coverage, and frequency. For all three measures of importance, elevation was the most important variable. Considering all three measures, the three most important features are Elevation, Soil Type, and Horizontal Distance to Roadways. Figure 5 illustrates the same results in a plot.

Feature	Gain	Cover	Frequency
Elevation	0.35257	0.42363	0.14353
Soil Type	0.28779	0.13321	0.08592
Horizontal_Distance_To_Roadways	0.08601	0.09694	0.17689
Horizontal_Distance_To_Fire_Points	0.07676	0.09432	0.1688
Hillshade_9am	0.04415	0.03634	0.05896
EuclidDistHydro	0.03438	0.03553	0.04818
WildernessArea	0.02854	0.05631	0.031
Aspect	0.02102	0.02295	0.05964
Horizontal_Distance_To_Hydrology	0.01828	0.01179	0.05054
Hillshade_Noon	0.01706	0.02847	0.04885

Table 6: This table shows the variable importance of the feature. All three measures of importance should be maximized. For all of these measures, Elevation was the most important.

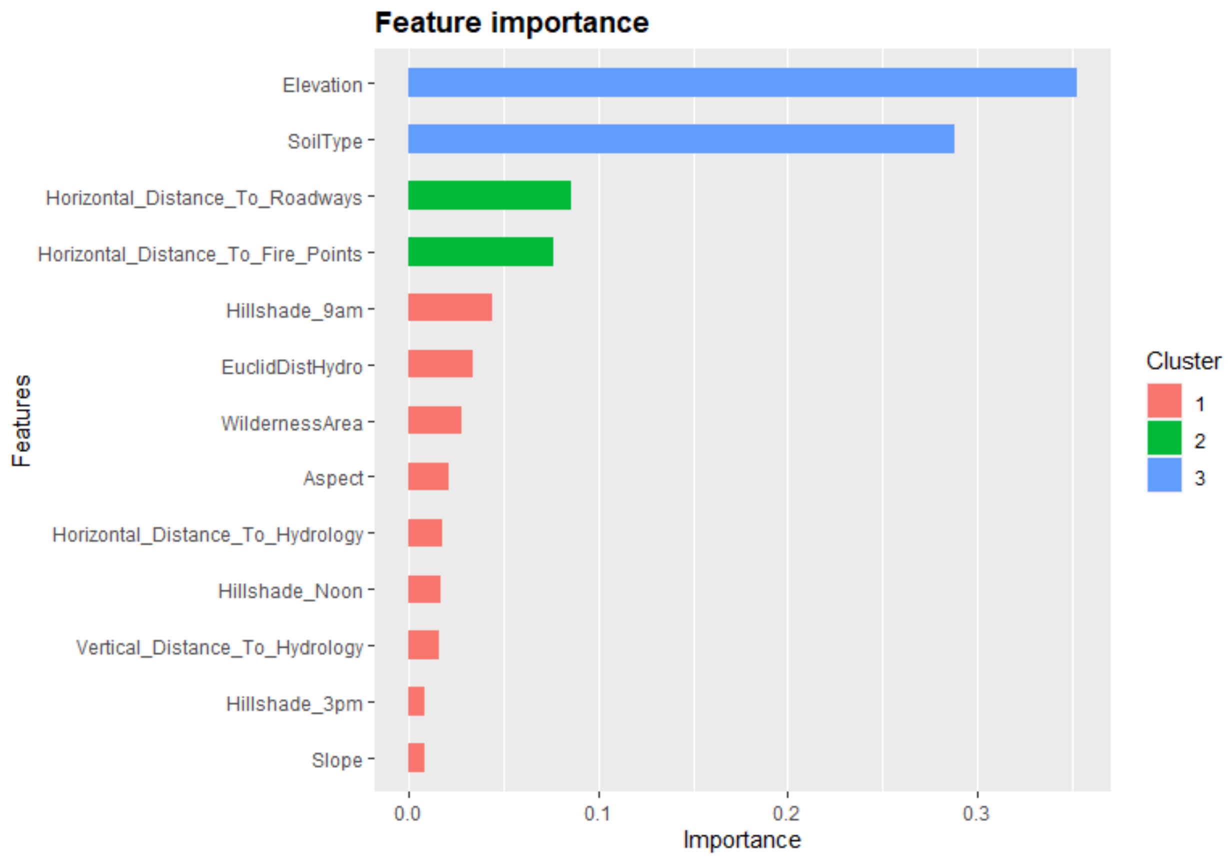

Figure 4: Variable importance plot from the XGBoost model. It is evident that the majority of the predictive power from the model comes from two main variables, the elevation and the type of soil.

Discussion

XGBoost is one of many algorithms for implementing boosting. XGBoost has gained popularity in recent years due to its primary design features: computational speed and high accuracy. To investigate XGBoost, we applied it to the cover type data set. The cover type data set isn't considered big data as it only has 581,012 observations, but it is large enough to present similar computational issues. The XGBoost model was fitted to a small sample of 32,197 observations and then tested on the remainder. The model performed well, having an accuracy of 86.8% and only taking approximately 3 seconds to fit. This performance was compared to other classification models: LDA, KNN, Multinomial Logistic Regression, and a Decision Tree. XGBoost not only fit the training data faster, it also had a higher accuracy. This outperformance illustrates its design features and its potential to be applied at scale to big data problems.

References

Chen Tianqi, Guestrin Carlos. August 2016 *XGBoost: A Scalable Tree Boosting System.* Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Association for Computing Machinery.

Ryza, S., Laserson, U., Owen, S., & Wills, J. (2017). *Advanced Analytics with Spark* (2nd ed.).

Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani. (2013). *An introduction to statistical learning* : with applications in R. New York :Springer,

Gonzalez, R. (2018). *More Features Than Observations* [web log]. Retrieved from https://rodrigo-gonzalez.com/blog/featureselection.

Hooker, G. (2004). Diagnosing extrapolation. Proceedings of the 2004 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '04. https://doi.org/10.1145/1014052.1014121

What is XGBoost? NVIDIA Data Science Glossary. (n.d.). Retrieved December 9, 2021, from https://www.nvidia.com/en-us/glossary/data-science/xgboost/.

Karlicek, O. (2021). *Application of Machine Learning in Portfolio Construction* (thesis). Charles University.

Blackard, J. A., Dean, D. J., & Anderson, C. W. (1998). Forest Cover Type Dataset. https://archive.ics.uci.edu/ml/datasets/covertype.

Abu-Rmileh, A. (2021, September 2). *Be careful when interpreting your features importance in xgboost!* Medium. Retrieved December 9, 2021, from https://towardsdatascience.com/be-careful-when-interpreting-your-features-importance-in-xgboost-6e16132588e7.

Reinstein, I. (n.d.). *XGBoost, a top machine learning method on Kaggle, explained*. KDnuggets. Retrieved December 9, 2021, from https://www.kdnuggets.com/2017/10/xgboost-top-machine-learning-method-kaggle-explained.html.

Appendix - R Code

```
set.seed(1234)
inTrain <- createDataPartition(df$Cover_Type,
              p = .75,
              list = FALSE,
              times = 1)
training <- df[inTrain, ]
testing <- df[-inTrain, ]

rm(inTrain)

n = 5000
df$Class <- df$Cover_Type
class1_ind = which(df$Class == 1)
class2_ind = which(df$Class == 2)
class3_ind = which(df$Class == 3)
class4_ind = which(df$Class == 4)
class5_ind = which(df$Class == 5)
class6_ind = which(df$Class == 6)
class7_ind = which(df$Class == 7)

class1_samp = sample(class1_ind, min(n, length(class1_ind)), replace = F)
class2_samp = sample(class2_ind, min(n, length(class2_ind)), replace = F)
class3_samp = sample(class3_ind, min(n, length(class3_ind)), replace = F)
class4_samp = sample(class4_ind, min(n, length(class4_ind)), replace = F)
class5_samp = sample(class5_ind, min(n, length(class5_ind)), replace = F)
class6_samp = sample(class6_ind, min(n, length(class6_ind)), replace = F)
class7_samp = sample(class7_ind, min(n, length(class7_ind)), replace = F)

indices = c(class1_samp, class2_samp, class3_samp, class4_samp, class5_samp,
class6_samp, class7_samp)
training <- df[indices, ]
testing <- df[-indices, ]

trControl <- trainControl(method = "repeatedcv",
           repeats = 2,
           number = 5,
           verboseIter = TRUE,
           allowParallel = TRUE,
           savePredictions = TRUE,
           preProcOptions = c("center", "scale"),
           classProbs = TRUE, # For ROC calculation
           summaryFunction = multiClassSummary,
            # sampling = "smote"
            )

mainFormula <- formula(CoverName ~ Elevation + Aspect + Slope +
Horizontal_Distance_To_Hydrology + Vertical_Distance_To_Hydrology +
Horizontal_Distance_To_Roadways + Hillshade_9am + Hillshade_Noon + Hillshade_3pm +
Horizontal_Distance_To_Fire_Points + WildernessArea + EuclidDistHydro + SoilType)

set.seed(5678)
ldaMod <- train(mainFormula,
       data = training,
       method = "lda",
       metric = "Accuracy",
       trControl = trControl)

set.seed(1234)
knnMod <- train(mainFormula,
       data = training,
       method = "knn",
       metric = "Accuracy",
       trControl = trControl)

set.seed(1234)
mnomMod <- train(mainFormula,
        data = training,
        method = "multinom",
        metric = "Accuracy",
        trControl = trControl,
        verbose = 0)

set.seed(1234)
dTreeMod <- train(mainFormula,
        data = training,
        method = "rpart",
        metric = "Accuracy",
        trControl = trControl)

resamps <- resamples(list(LDA = ldaMod, KNN = knnMod, multiNomial = mnomMod, dTree =
dTreeMod))

# Subtract by 1, since multi-class starts at 0 rn.
training$Cover_Type <- as.integer(training$Class) - 1
testing$Cover_Type <- as.integer(testing$Class) - 1
```

```
xTrain <- training %>%
 select(-Cover_Type, -CoverName, -Class) %>%
 as.matrix()
yTrain <- training %>%
 select(Cover_Type) %>%
 as.matrix()
xTest <- testing %>%
 select(-Cover_Type, -CoverName, -Class) %>%
 as.matrix()
yTest <- testing %>%
 select(Cover_Type) %>%
 as.matrix()

xgbTrain <- xgb.DMatrix(data=xTrain, label=yTrain)
xgbTest <- xgb.DMatrix(data=xTest, label=yTest)

# Define the parameters for multinomial classification
numClasses = length(levels(as.factor(df$Class)))

params = list(
 booster="gbtree",
 eta=0.001,
 max_depth= c(10),
 gamma=3,
 subsample=0.75,
 colsample_bytree=1,
 objective="multi:softprob",
 eval_metric = "auc",
 num_class = numClasses
)

xgbMod3 <- xgb.cv(params = params,
        data = xgbTrain,
        nrounds = 3,
        nfold = 2,
        verbose = FALSE,
        prediction = TRUE)

xgbPreds3 <- data.frame(xgbMod3$pred) %>%
 mutate(max_prob = max.col(., ties.method = "last"),
    label = training$Cover_Type + 1)
# confusion matrix of model
confusionMatrix(factor(xgbPreds3$max_prob),
factor(xgbPreds3$label),
mode = "everything")
```

Ref/Pred	Spruce Fir	Lodgepole Pine	Ponderosa Pine	Cottonwood	Aspen	Douglas Fir	Krummholz
Spruce Fir	3558	1032	0	0	14	4	346
Lodgepole Pine	849	3000	15	0	195	28	4
Ponderosa Pine	5	111	3746	126	82	623	0
Cottonwood	0	2	197	2510	0	127	0
Aspen	168	601	85	0	4601	117	7
Douglas Fir	30	214	957	111	108	4101	0
Krummholz	390	40	0	0	0	0	4643

Overall Statistics:

Accuracy: 0.7988

95% C.I.: (0.7944, 0.8032)

No Information Rate: 0.1527

P-value: <0.001

Variable Importances:

xgb.importance(model = xgbMod2)

Feature	Gain	Cover	Frequency
Elevation	0.352571897	0.42362815	0.14353100
SoilType	0.287792775	0.13321422	0.08591644
Horizontal_Distance_To_Roadways	0.086014876	0.09693740	0.17688679

Horizontal_Distance_To_Fire_Points	0.076757289	0.09432344	0.16880054
Hillshade_9am	0.044145245	0.03634326	0.05896226
EuclidDistHydro	0.034384586	0.03552973	0.04818059
WildernessArea	0.028542161	0.05631486	0.03099730
Aspect	0.021019391	0.02295444	.05963612
Horizontal_Distance_To_Hydrology	0.018275505	0.01178815	0.05053908
Hillshade_Noon	0.017061114	0.02847024	0.04885445
Vertical_Distance_To_Hydrology	0.015899327	0.03798679	0.05929919
Hillshade_3pm	0.008854905	0.01211883	.03335580
Slope	0.008680928	0.01039049	0.03504043