Forecasting a US Municipal Green Bond Index Using Tree Based Machine Learning Models

Vincent Jia
Supervised by Prof. Tony Wirjanto

August 18th, 2025

Abstract

We attempt to forecast the S&P Global Municipal Green Bond Index price over three different time periods. These time periods are the "Covid" time period (Jan 2nd, 2019 to May 31st, 2023), the "post-Covid" time period (June 1st, 2023 to Dec 31st, 2024), and "full" time period ("Covid" + "post-Covid" time period). We attempt this using four different tree based machine learning models. These are Random Forest, XGBoost, LightGBM, and CatBoost. We find that the strongest predictor of municipal green bond index prices over all three time periods is conventional bond index prices.

Acknowledgments

I'd like to thank my advisor Prof. Tony Wirjanto, Mom, Dad, Darren Jia, my tutor Gavin Orok, and all the classmates and professors who have made Waterloo so enjoyable

Dislclaimer

Some of the code used in this paper was generated with the help of generative AI.

Contents

1 Introduction

In *The role of major markets in predicting the U.S. municipal green bond market performance: New evidence from machine learning models* [16] by Kocaarslan and Soytas, the authors use tree based machine learning methods to explain the prices of municipal green bonds. Specifically, they want to explain the prices of municipal green bonds during the pre-Covid-19 time period (June 30th, 2014 to Dec 31st 2019), and the Covid-19 time period (Jan 1st 2020 to Jan 21st 2022). To do this, they use tree based machine learning models. Specifically they use Random Forest, Extreme Gradient Boosting (XGBoost), Light Gradient Boosting Machine (LightGBM), and Categorical Boosting (CatBoost).

In this paper, we will attempt to forecast the prices of the same municipal green bond index used by Kocaarslan and Soytas by using the same tree based machine learning models. One small difference is that our data set will be slightly different with the most important adjustment being due to the LIBOR to SOFR transition. The three time periods that we are interested in are the "Covid" time period (Jan 2nd, 2019 to May 31st, 2023), the "post-Covid" time period (June 1st, 2023 to Dec 31st, 2024), and "full" time period ("Covid" + "post-Covid" time period).

2 Literature Review

Green bonds have been extensively studied by financial economists. Kocaarslan (2023) is building on the results of past studies, but attempting to extend their results by selecting a wide range of variables to predict changes in municipal green bond prices. Specifically, he has included conventional bonds, energy commodities, stocks, and various measures of macroeconomic risk (like different types of interest rates and volatility indices).

In Ehlers et al (2017) [12] they find that green bonds tend to be issued at a higher price than conventional bonds. This means that the yield on green bonds is lower. This supposed difference between the yield on green bonds and conventional bonds is called the "greenium".

Financial economists are interested in the connectedness between green bonds and other asset classes. In Naeem et al [20], the authors conclude that, "there are some return and volatility connections between green stock markets, US sectors and commodities".

There are also studies specifically focusing on green bonds during the Covid-19 period. One such paper is Abakah et al [1]. In this paper, they show that the connectedness of green bonds and conventional asset classes increased during the outbreak of Covid-19. This is because during Covid-19, investors shifted their portfolios from stocks and commodities to safer assets like green bonds. They also show that green bonds are strongly correlated with treasury bonds, aggregate bonds, and bond indices, and green bonds are not highly connected to energy indices.

With this background, we are interested in assessing the relationship between green bonds and various assets classes on the post Covid time frame. We are also interested in investigating whether using SOFR as an input variable, instead of LIBOR based input variables, will improve the performance of our models for municipal green bond index prices.

3 Author's Implementation and Results

Kocaarslan and Soytas (2023) [16] attempt to use tree based machine learning techniques to explain prices in municipal green bond indices. Specifically they use Random Forest, Extreme Gradient Boosting (XGBoost), Light Gradient Boosting Machine (LightGBM), and Categorical Boosting (CatBoost).

Kocaarslan and Soytas (2023) are interested in looking at the difference in drivers of municipal green bond index prices over two distinct time periods. These are the "pre-Covid" and "Covid" time periods. The "pre-Covid" time period is from June 30th, 2014 to Dec 31st, 2019. The "Covid" time period is from Jan 1st, 2020 to Jan 21st, 2022.

Kocaarslan and Soytas (2023) conclude that during the "pre-Covid" period, the key driver of municipal green bond index prices was the traditional bond market. Over the "Covid" period, the key driver of municipal green bond index prices was the S&P 500.

3.1 Authors' Data

Kocaarslan and Soytas (2023) select an extensive amount of factors to predict municipal green bond index prices. They are as follows [16]:

1. MGB: Standard & Poor's Municipal Green Bond Index
2. AB: Standard & Poor's US Aggregate Bond Index
3. SP500: S&P500 Index
4. EN: Standard & Poor's GSCI Energy Index
5. DEF: Default spread, the difference between the yields on the BAA-rated and AAA-rated corporate bonds (US companies)
6. TERM: Term Spread, the difference between the yields on the US 10-year treasury bond and the US 3-month treasury-bill.
7. FFR: US Federal Funds Rate
8. TED: TED Spread, the difference between the 3-month LIBOR based on US dollars and the 3-month treasury-bill.
9. USD: The trade weighted US dollar index
10. VIX: Chicago Board Options Exchange (CBOE) Volatility Index for the S&P 500.
11. EVZ: CBOE Euro Currency Volatility Index

12. OVX: CBOE Crude Oil Volatility Index

13. GVZ: CBOE Gold Volatility Index

14. USEPU: US Economic Policy Uncertainty Index

Data sources 1-4 were obtained from the "S&P Global" website. Data sources 5-9 were obtained from the "FRED (Federal Reserve Economic Data)" website maintained by the Federal Reserve Bank of St. Louis. Data sources 11 - 13 were obtained on the "CBOE" website. Data source 14 comes from a website maintained by a professor at the University of Wisconsin-Madison [2].

However, due to being only given a brief description of the data, finding the exact same data source as Kocaarslan and Soytas (2023) have has been difficult. We had difficulty finding the DEF data set. TED is now deprecated because of the LIBOR to SOFR transition. The specific index Kocaarslan used for USD is also deprecated. Finally, the data from the USEPU website cannot be easily exported into an excel file. In our own implementation, some adjustments in the data sets had to be made. This will be discussed more in a later section.

Notice that data sources 1-4 are index prices. Data sources 5-8 are all related to different interest rates. Source 9 measures the purchasing power of the US dollar when buying foreign goods. Sources 10-14 are all different types of volatility indices.

3.2 Authors' Implementation

The authors apply Random Forest, XGBoost, LightGBM, and CatBoost to their chosen data set. Interestingly, they do not do any data pre-processing, and keeps all the data at the level it was imported at. They do not make any of the data stationary or normalize (e.g. from -1 to 1) any of the data. In the papers, generally we assume that the data $\mathcal{D} = (\boldsymbol{x}_k, y_k)_{k=1,2,\ldots,n}$ contains i.i.d. examples $(\boldsymbol{x}_k, y_k)$ sampled from some unknown distribution $P(\cdot, \cdot)$. Since we are working with time series data, the assumption of i.i.d. examples $(\boldsymbol{x}_k, y_k)$ cannot be correct, because time series data contains strong autocorrelation.

Since the authors keep the data at the level it was imported at, and we generally know that stocks follow a geometric Brownian motion, some of the author's data is definitely not stationary. Since stock data is non-stationary, and we know non-stationarity implies non-i.i.d., we know that stock data is non-i.i.d. This is a problem because our tree learning models work under the assumption of all i.i.d. data. We should keep this limitation in mind when interpreting the authors' results.

So, the authors have the data partitioned into two time periods. The "pre-Covid" time period is from June 30th, 2014 to Dec 31st, 2019. The "Covid" time period is from Jan 1st,

Table 1
Descriptive statistics.

	MGB	AB	SP500	EN	DEF	TERM	FFR	TED	USD	VIX	EVZ	OVX	GVZ	USEPU
Panel A. Sub-sample 1: 06/30/2014–12/31/2019														
Mean	111.58	190.50	2408.39	194.65	0.92	1.30	1.01	0.32	110.92	15.04	8.65	35.50	14.57	81.65
Median	112.39	190.05	2363.64	188.26	0.90	1.30	0.91	0.29	112.20	13.97	8.13	33.05	14.37	72.92
Maximum	126.16	210.79	3240.02	349.08	1.54	2.64	2.45	0.68	119.23	40.74	15.05	78.97	28.37	336.71
Minimum	99.36	178.03	1829.08	98.90	0.55	−0.52	0.06	0.13	93.14	9.14	4.31	15.59	8.88	3.32
Std. dev.	6.60	7.55	364.39	43.70	0.21	0.77	0.83	0.11	5.66	4.14	2.41	10.75	3.39	43.28
Skewness	0.31	0.97	0.31	1.08	0.82	−0.42	0.41	0.80	−1.37	1.53	0.59	0.82	0.60	1.69
Kurtosis	2.57	3.71	1.71	4.83	3.26	2.30	1.66	2.91	4.60	6.51	2.54	3.50	2.82	7.66
Jarque-Bera	30.99	236.28	114.66	447.30	154.64	66.28	137.53	144.14	563.95	1210.54	88.65	165.49	81.26	1848.74
Probability	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Panel B. Sub-sample 2: 01/01/2020–21/01/2022														
Mean	129.89	220.85	3760.88	185.74	0.90	0.98	0.22	0.20	115.43	24.45	6.86	51.67	19.04	218.05
Median	130.87	221.87	3768.36	187.75	0.80	1.01	0.08	0.14	114.65	22.01	6.53	39.44	17.73	177.23
Maximum	135.12	225.57	4793.54	287.77	1.99	1.73	1.60	1.42	126.14	82.69	19.31	325.15	48.98	807.66
Minimum	114.02	207.76	2237.40	63.76	0.62	−0.20	0.04	0.06	110.54	12.10	4.13	27.66	10.91	22.25
Std. dev.	4.05	3.57	617.48	55.45	0.29	0.50	0.43	0.24	3.50	10.29	1.91	34.72	5.27	141.66
Skewness	−0.85	−1.34	−0.15	−0.07	1.45	−0.24	2.73	3.51	0.99	2.46	2.47	3.49	1.94	1.40
Kurtosis	3.38	4.37	1.96	1.96	4.86	1.88	8.56	14.95	3.31	10.99	13.13	17.59	8.70	4.87
Jarque-Bera	62.82	189.31	24.50	23.10	247.25	30.59	1263.38	4001.52	83.91	1832.41	2646.25	5452.78	991.98	236.01
Probability	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Notes: Table 1 depicts the summary statistics of the variables used for analysis. VIX, EVZ, OVX, and GVZ represent the implied volatilities (uncertainties) in stock, currency, oil, and gold markets, respectively. USEPU refers to the U.S. economic policy uncertainty. DEF, TERM, FFR, TED, and USD represent the default spread, term spread, federal funds rate, TED spread, and trade-weighted U.S. dollar index, respectively. MGB, AB, SP500, and EN represent the U.S. municipal green bond, U.S. aggregate bond (conventional bond), U.S. stock, and energy commodity markets, respectively.

Figure 1: Descriptive statistics of Author's dataset

2020 to Jan 21st, 2022.

The authors use four different metrics to assess the quality of his models. These are mean absolute error (MAE), mean squared error (MSE), root mean squared error (RMSE), and R^2.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - Y_i) \tag{1}$$

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - Y_i)^2 \tag{2}$$

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - Y_i)^2} \tag{3}$$

$$R^2 = \frac{\sum_{i=1}^{N}(\hat{Y}_i - \bar{Y}_i)^2}{\sum_{i=1}^{N}(Y_i - \bar{Y}_i)^2} \tag{4}$$

In Equation (4), $\hat{Y}_i$ refers to the predicted Y_i, Y_i is an observed value, and $\bar{Y}_i$ is the mean of observed values.

Kocaarslan and Soytas (2023) say that over the "pre-Covid" and "Covid" data sets, they randomly partition 75% of the data into the training set and 25% of the data into the testing set. Notice how by doing this, all the data in the testing set may actually be data from earlier in time than the training set. Generally, we want our testing set to be later in time than our training data. Otherwise, we would be using future data to predict past results which is *data leakage*. If we are doing this, there is generally no real world application to our results.

Kocaarslan and Soytas (2023) then say they use standard cross validation to tune the hyperparameters for his models. These hyperparameters were not shown in their paper.

Table 3
The performance of machine learning models.

Models	MAE	MSE	RMSE	R^2
Panel A. Sub-sample 1: 06/30/2014–12/31/2019				
Random forest	0.215800688	0.092037391	0.303376649	0.997994154
XGBoost	0.209915778	0.078377402	0.279959643	0.998291858
LightGBM	0.216477977	0.09874724	0.314240736	0.997847921
CatBoost	0.219363826	0.082117232	0.286561044	0.998210353
Panel B. Sub-sample 2: 01/01/2020–21/01/2022				
Random forest	0.372423926	0.519685317	0.720892029	0.975070365
XGBoost	0.371137153	0.431235065	0.656684905	0.97931338
LightGBM	0.442456484	0.686650907	0.828644017	0.96706092
CatBoost	0.353521705	0.365348864	0.604440952	0.982473983

Notes: Table 2 shows the forecasting performance of the four machine learning models: Mean Absolute Error (MAE), Mean Square Error (MSE), Root Mean Square Error (RMSE), and the coefficient of determination (R^2).

Figure 2: Author's metrics [16]

From the metrics provided in Figure (2), we can see that almost all the models by

Kocaarslan and Soytas have a R^2 of close to 0.99. This is most likely due to data leakage and suggests the models they have fit are overfitted.

Regardless, we see for "sub-sample 1" which is the "pre-Covid" time period, the best performing model is XGBoost across all metrics, mean absolute error, mean squared error, root mean squared error, and R^2. Over the "Covid" period, the best performing model is CatBoost over all metrics.

3.3 Authors' Results

Kocaarslan and Soytas (2023) show the *SHAP summary plots* (also called *beeswarm plots*) and *variable importance plots* of the XGBoost model fitted over the "pre-Covid" data set, and of the CatBoost model fitted over the "Covid" data set. The specific packag he uses is called "shap" which is why the Shapley values are labelled "shap values".

Specifically, these plots show what has driven predictions of the municipal green bond index price over the testing set in the two different models.

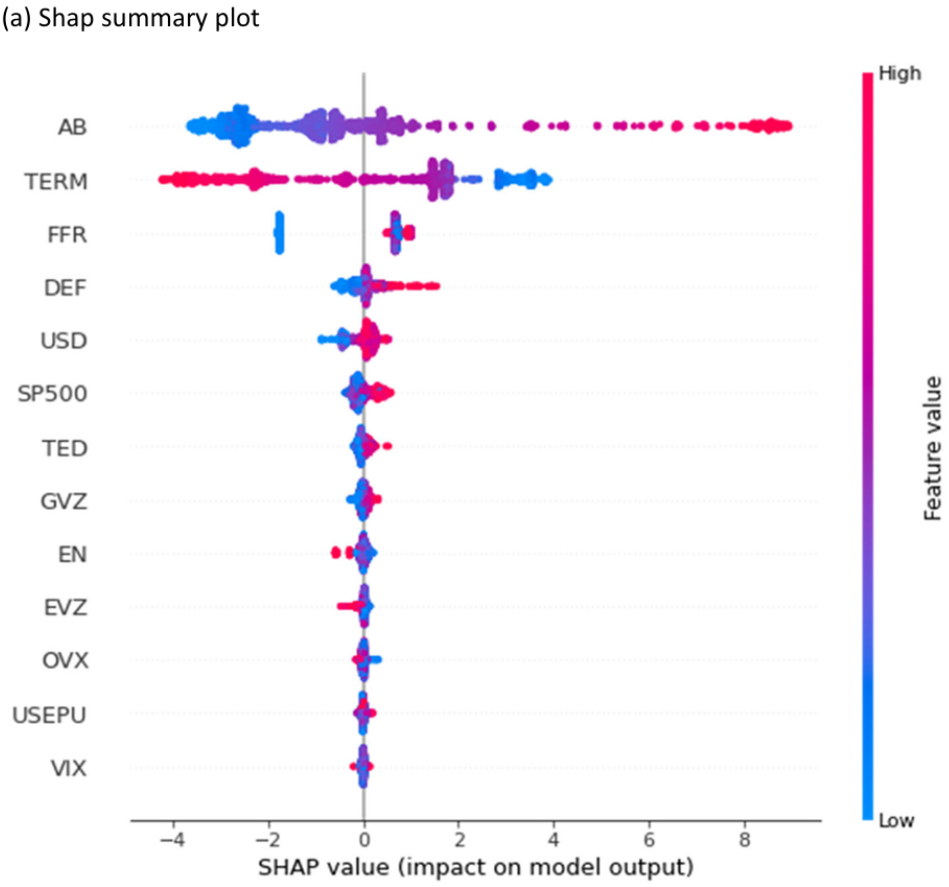


Figure 3: XGBoost summary plot over "pre-Covid" period [16]

(b) Variable importance plot

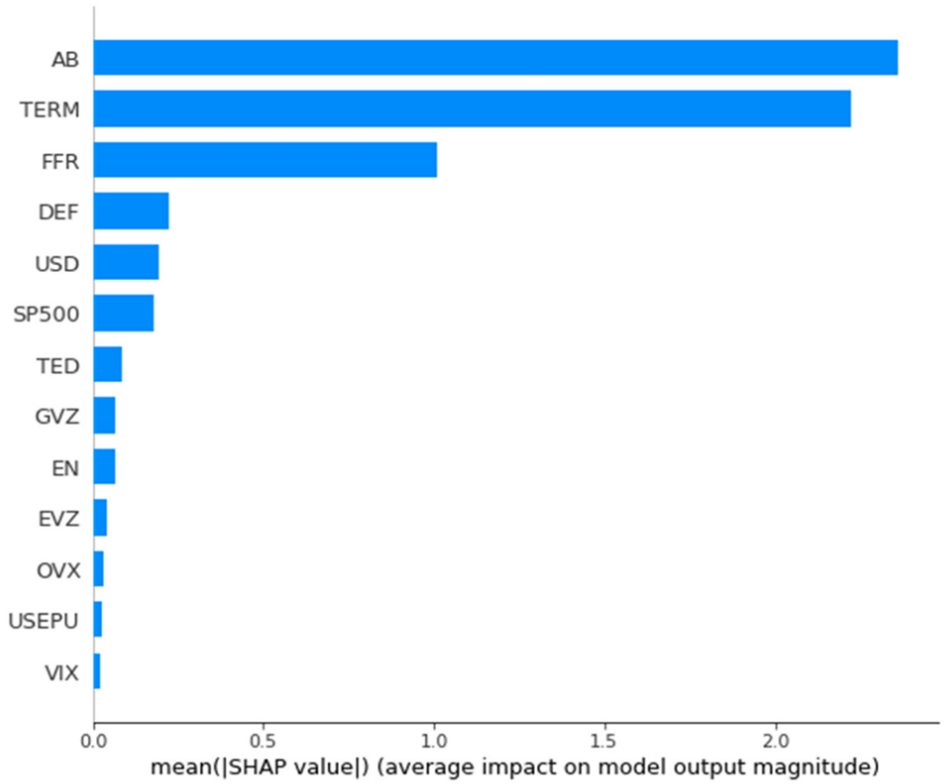


Figure 4: XGBoost variable importance plot over "pre-Covid" period [16]

Let us interpret Figures 3 and 4. The way to read Figure (3) is that every "dot" in the figure is a data point in the testing data. Where you have many data points "bunching up" at a specific point in the x-axis, that is because there are many data points with that particular shap value. The color gradient tells you how to interpret the shap summary plot. For example, we can see that when the value of AB (Aggregate bond index) is high (red color), the value of municipal green bonds increases relative to the expected value of municipal green bonds over all the training instances. When AB is low (blue color), we see that the price of MGB falls relative to the expected value of municipal green bonds over all the training instances. This is to be expected, as green bonds and traditional bonds are close substitutes for each other.

Figure (4) shows the variable importance plot. We can directly see the similarity between Figures (4) and (3). This is because, in the variable importance plot, we take the absolute values of each point in the shap summary plot, and then take the average of all those points. So, we can see that the order from top to bottom of the variables (in this case) are still the same. Thus, the variable importance plot shows us which variables contribute the most to predicting shap values over the testing set. Thus, we can see that AB and TERM are by far

the largest two contributors to MGB over the testing set.

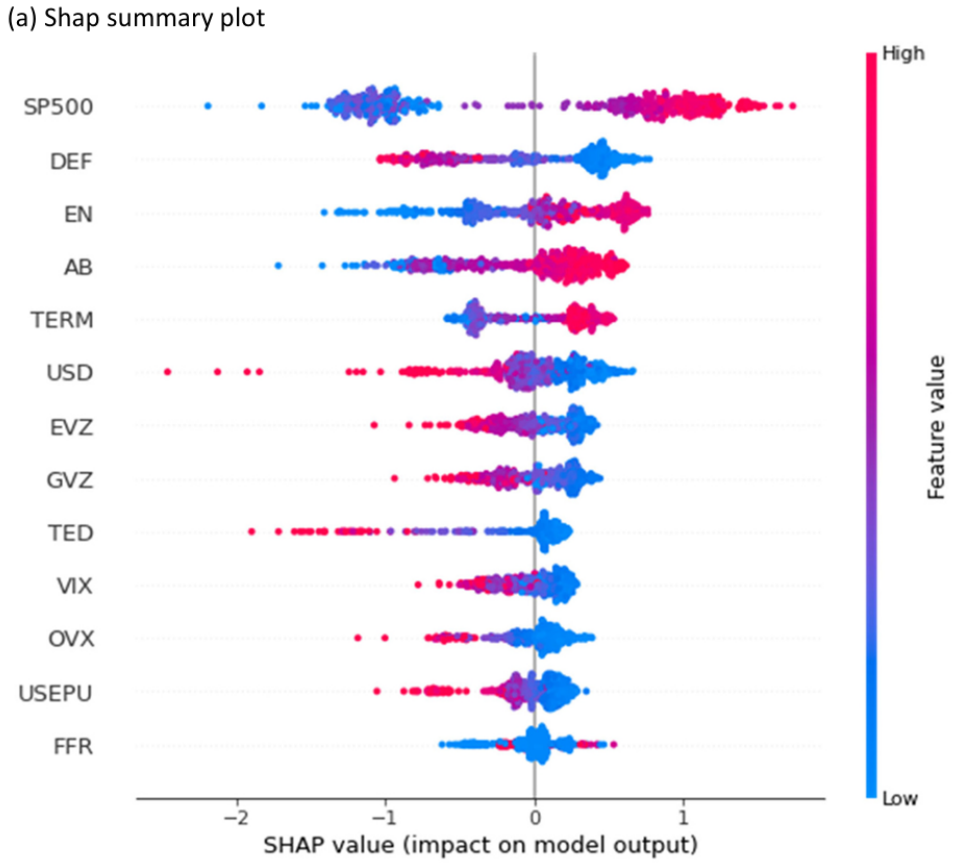


Figure 5: CatBoost summary plot over "Covid" period [16]

(b) Variable importance plot

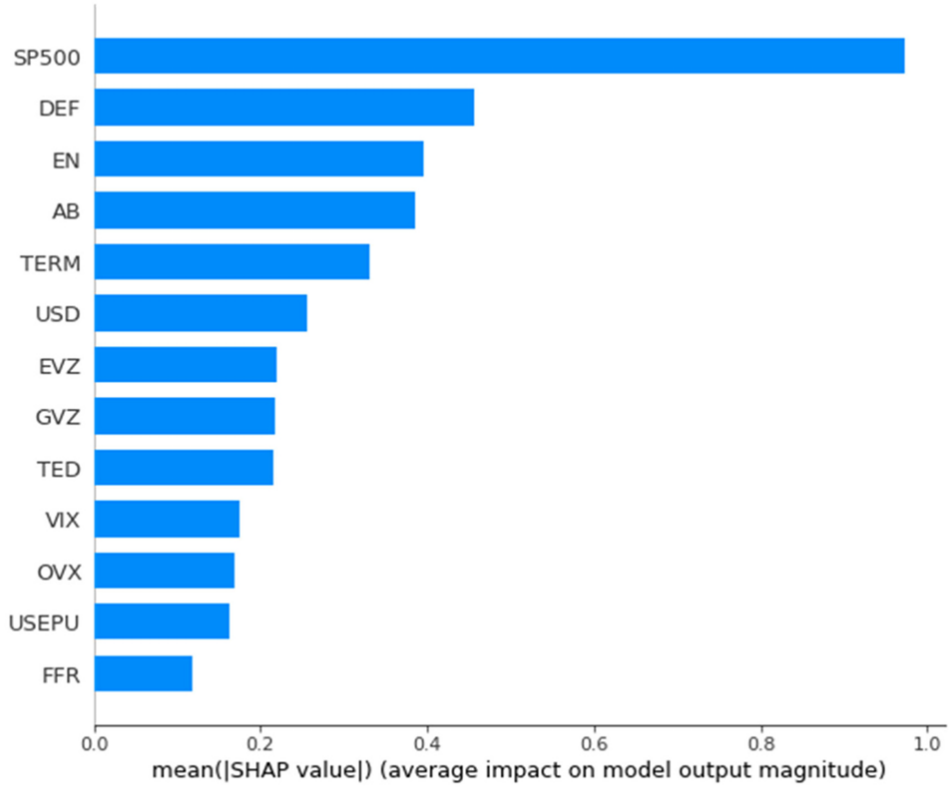


Figure 6: CatBoost variable importance plot over "Covid" period [16]

Let us interpret Figures 5 and 6. We see in Figure (5), that when SP500 is high (low), the value of municipal green bonds increases (decreases) relative to the expected value of municipal green bonds over all the training instances. When DEF is high (low), the value of municipal green bonds decreases (increases) relative to the expected value of municipal green bonds over all the training instances

From Figure 6, we see the SP500 and DEF are the biggest contributors to predicting MGB over the "Covid" period.

The authors provide *partial dependence plots* which display changes in shap values for a joint distribution of predictor variables. They select a portion of the most interesting partial dependence plots.

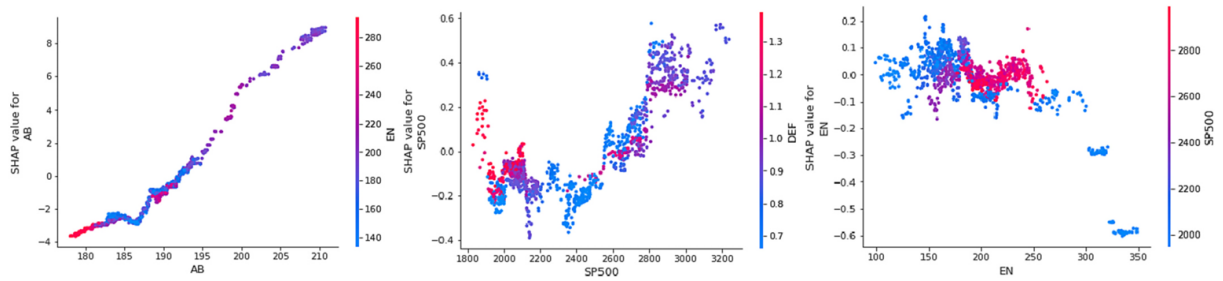


Figure 7: Partial dependence plots over "pre-Covid" period [16]

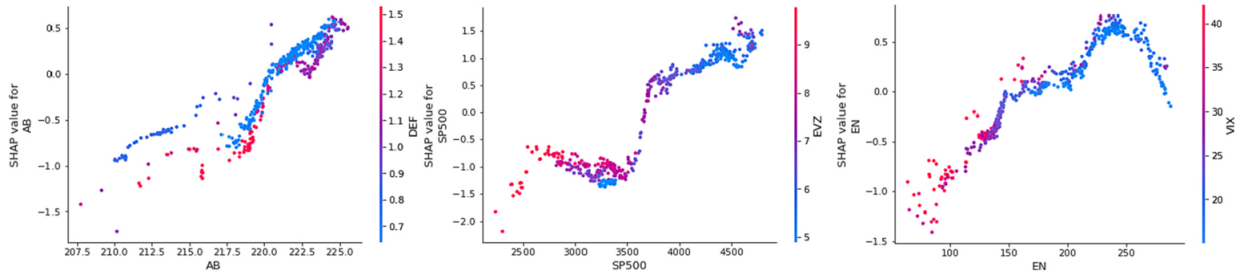


Figure 8: Partial dependence plots over "Covid" period [16]

The way to interpret Figures 7 and 8 is by imagining we are working with a 3D graph, but the 3rd dimension is being encoded by color. Let's work with a specific clear example.

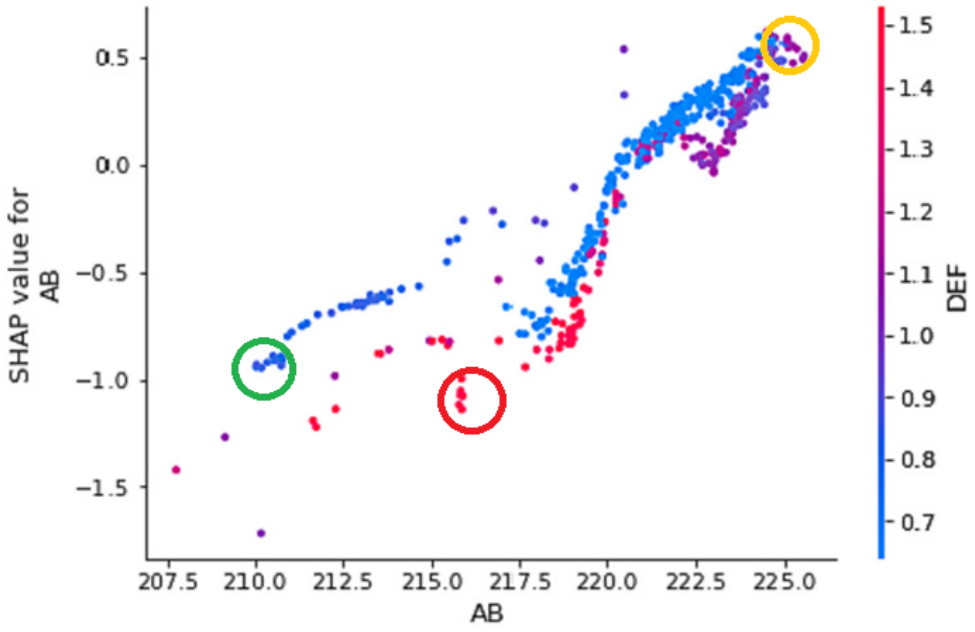


Figure 9: Partial dependence plots of AB and DEF over "Covid" period [16]

In Figure 9, let your eyes follow the color gradient from blue to red. When the dots are blue(red), that means the value of DEF is low (high). Values of AB are on the x-axis and are increasing from left to right. So, we can see when AB and DEF are low (green circle), that the Shapley value is around -1. When the value of AB is around 225 and DEF is around 1.1 (orange circle), the Shapley value is around 0.5. When the value of AB is around 217 and DEF is around 1.5, the Shapely value is around -1.

The fact that the Shapley values can go from positive to negative when considering joint effects means we should be skeptical in accepting the results of our *Shap summary plots* at face value. The variables in the *Shap summary plots* are not independent of each other, but the shap summary plots calculate Shapley values when holding the other variables constant. So, if in the real world, if we see the S&P 500 rapidly increase, we must be aware that some other variables like (VIX) may also increase. So the positive increase in Shapley value from the S&P increasing may be partially offset by the decrease in Shapley value from VIX increasing.

4 Our Implementation

4.1 Our Data

We could not perfectly replicate the results in *The role of major markets in predicting the U.S. municipal green bond market performance: New evidence from machine learning models* [16] due to not being able to find the exact same data.

As mentioned before, we had issues with the DEF, TED, USD, and USEPU datasets. Of these four, the biggest issue was the TED data set being deprecated in 2022 due to the LIBOR to SOFR transition. As such, we have decided to use SOFR in order to do analysis on more recent data.

Our dataset spans from January 2nd, 2019 to Dec 31st, 2024. This is a period of about 6 years. We also had to account for the Covid shock. I've split my data into a "Covid" and "post-Covid" period. The "Covid" period is from Jan 2nd, 2019 to May 31st, 2023. The "post-Covid" period is from June 1st, 2023 to Dec 31st, 2024. We also have the "full" data set which spans from January 2nd, 2019 to Dec 31st, 2024.

We will perform a forecasting exercise over the "Covid", "post-Covid" and "full" data sets in order to predict the price of the municipal green bond index.

In addition to SOFR, we have added three new data sets. These are "bb_lithium", "bb_rare_earths" and "bb_crude_oil". These data sources were obtained through a Bloomberg Terminal and correspond with China Lithium Carbonate Spot Price, MVIS Global Rare Earths Index, and West Texas Intermediate Crude Oil Futures. We have these new data sets because we were was curious if these resources which are frequently are talked about in the context of green finance, would affect the price of the municipal green bond index.

Also, the USD variable that Kocaarslan corresponded to was "trade weighted US dollar index". The "fred_usd" variable we have is the newer "Nominal Broad US Dollar Index"

The exact data used, url to the data sources, and the full code is available in the Github link at the end of this paper.

1. sp_mgb - S&P US Municipal Green Bond Index. Kocaarslan's MGB.
2. sp_ab - S&P US Aggregate Bond Index. Kocaarslan's AB.
3. sp_500 - S&P 500. Kocaarslan's SP500.
4. sp_en - S&P GSCI Energy. Kocaarslan's EN
5. nyf_sofr - Secured Overnight Financing Rate from the New York Fed website.
6. fred_term - Interest rate on the US 10 year treasury minus the interest rate on the 3 month treasury. Found on FRED. Kocaarslan's TERM.

7. fred_ffr - Effective Federal Funds Rate. Found on FRED. Kocaarslan's FFR.
8. fred_usd - Nominal Broad US Dollar Index. Found on FRED. In practice, very similar to Kocaarslan's USD.
9. fred_evz - CBOE Eurocurrency ETF Volatility Index. Found on FRED, not on CBOE. Kocaarslan's EVZ.
10. cboe_vix - CBOE Volatility Index of the S&P 500. Kocaarslan's VIX.
11. cboe_ovx - CBOE Crude Oil ETF Volatility Index. Kocaarslan's OVX.
12. cboe_gvz - CBOE Gold ETF Volatility Index. Kocaarslan's GVZ.
13. bb_lithium - China Lithium Carbonate Spot Price.
14. bb_rare_earths - MVIS Global Rare Earths Index
15. bb_crude_oil - West Texas Intermediate Crude Oil Futures

4.2 Data Processing

In order to process the data, we had to make some variables stationary. Remember, in theory the variables should be i.i.d. in order to apply the tree based machine learning models. However, since we are working with time series data, we cannot achieve i.i.d. data due to the data being auto-correlated (the past data points influence the future data). In this situation, we can at least make the data stationary to make the data "look more" like i.i.d. data.

At this point, before we have done any processing, the data looks as follows.

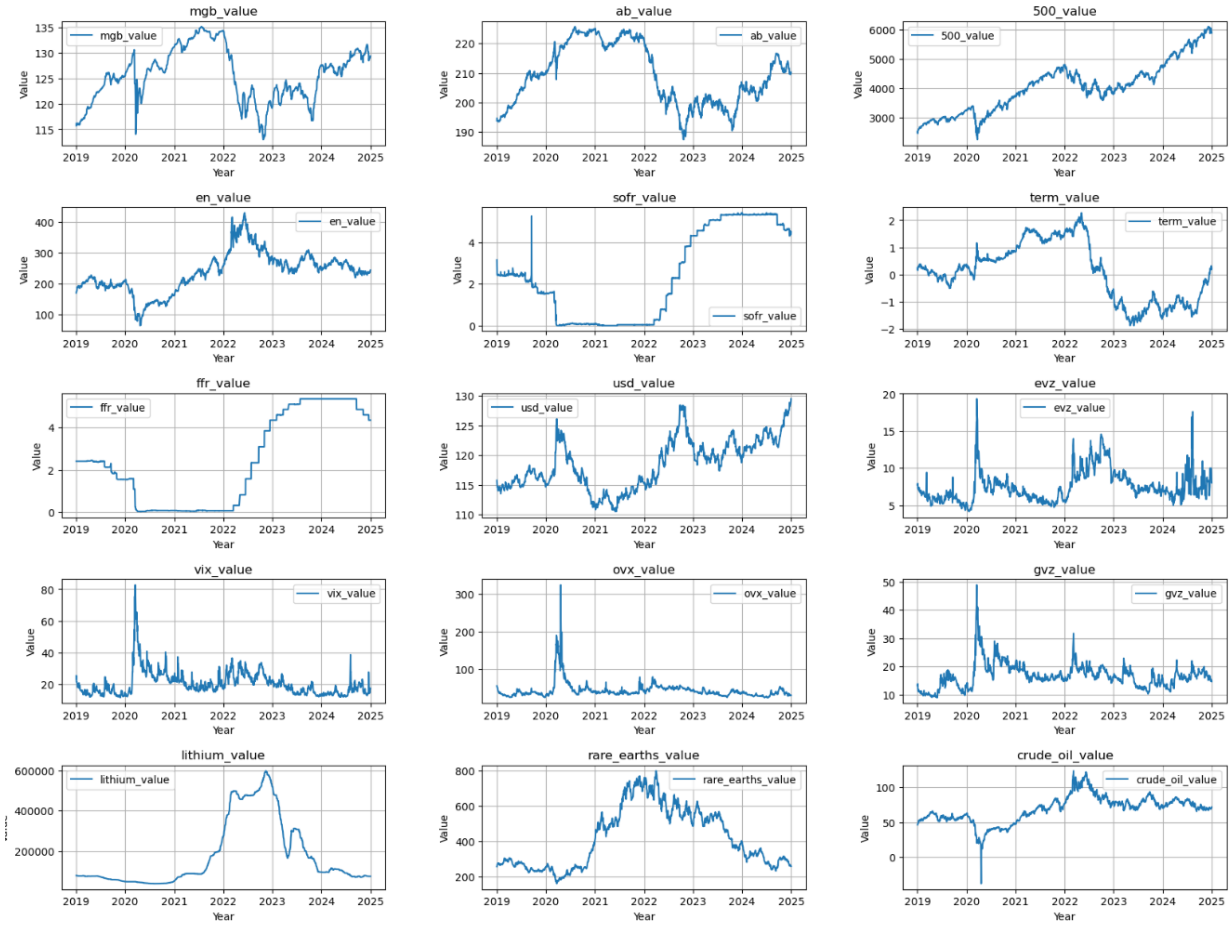


Figure 10: Graph of data before processing

We want to check if some specific variables are non-stationary. There are some variables that we want to keep unchanged, even if they are nonstationary. These are nyf_sofr, fred_term, fred_ffr, fred_evz, cboe_vix, cboe_ovx, cboe_gvz. These are because these are either interest rates or volatility indices. The meaning in these data sets is directly encoded by their level, and are typically interpreted directly from their level by financial people.

The data that we will test for stationarity is: sp_mgb, sp_ab, sp_500, sp_en, fred_usd, bb_lithium, bb_rare_earths, and bb_crude_oil.

4.2.1 Augmented Dickey-Fuller Stationarity Test

We will use the Augmented Dickey-Fuller (ADF) test to test for stationarity on sp_mgb, sp_ab, sp_500, sp_en, fred_usd, bb_lithium, bb_rare_earths, and bb_crude_oil.

Remember that the null hypothesis H_0 is: the data is not stationary. The alternative hypothesis H_a is: the data is stationary. We want to see p-values of less than 0.05 so we can reject the null hypothesis of non-stationarity.

Variable	ADF Statistic	p-value	Stationary
Municipal Green Bonds	-2.3816	0.1470	No
Aggregate Bonds	-1.6639	0.4498	No
S&P 500	-0.5221	0.8876	No
Energy Index	-1.4851	0.5409	No
Lithium	-1.7897	0.3856	No
Rare Earths	-1.2642	0.6454	No
Crude Oil	-1.8880	0.3377	No
USD	-1.1556	0.6924	No

Figure 11: ADF test for stationarity

We can see none of the data we tested on is stationary. This is not surprising.

4.2.2 Log Differencing vs Differencing

Now, we want to either log difference or difference the data to make it stationary. Log differencing is defined as $X_t = \ln(\frac{P_t}{P_{t-1}})$ where P_t is the value today (called P because the data is usually the price of some stock), and P_{t-1} is the value yesterday, of some particular data. Differencing is defined as $X_t = P_t - P_{t-1}$.

For sp_mgb, sp_ab, sp_500, and sp_en, we will take the log index. This is done out of the suspicion that these data sets follow a geometric Brownian motion. In theory, we know that stocks follow a geometric Brownian motion. For sp_mgb, and sp_ab, using the "eyeball test" over 10 years history (not shown), it seems reasonable to take the log difference.

For fred_usd, bb_lithium, bb_rare_earths, and bb_crude_oil, we will just take the difference. This is because we know this enough to make the data stationary, so there is no need to take the log difference.

Variable	ADF Statistic	p-value	Stationary
Log Diff Municipal Green Bonds	-12.0593	0.0000	Yes
Log Diff Aggregate Bonds	-26.6512	0.0000	Yes
Log Diff S&P 500	-8.8680	0.0000	Yes
Log Diff Energy Index	-6.5998	0.0000	Yes
Diff Lithium	-4.6601	0.0001	Yes
Diff Rare Earths	-32.1106	0.0000	Yes
Diff Crude Oil	-8.8678	0.0000	Yes
Diff USD	-9.4220	0.0000	Yes

Figure 12: ADF test for stationarity after log differencing/differencing

So, our data is now stationary. One last thing, we will normalize the data for the newly log differenced/ differenced data from -100 to 100. The reason -100 to 100 was chosen instead of -1 to 1 is just so the SHAP plots are more readable later on. Let's take one last look at our data.

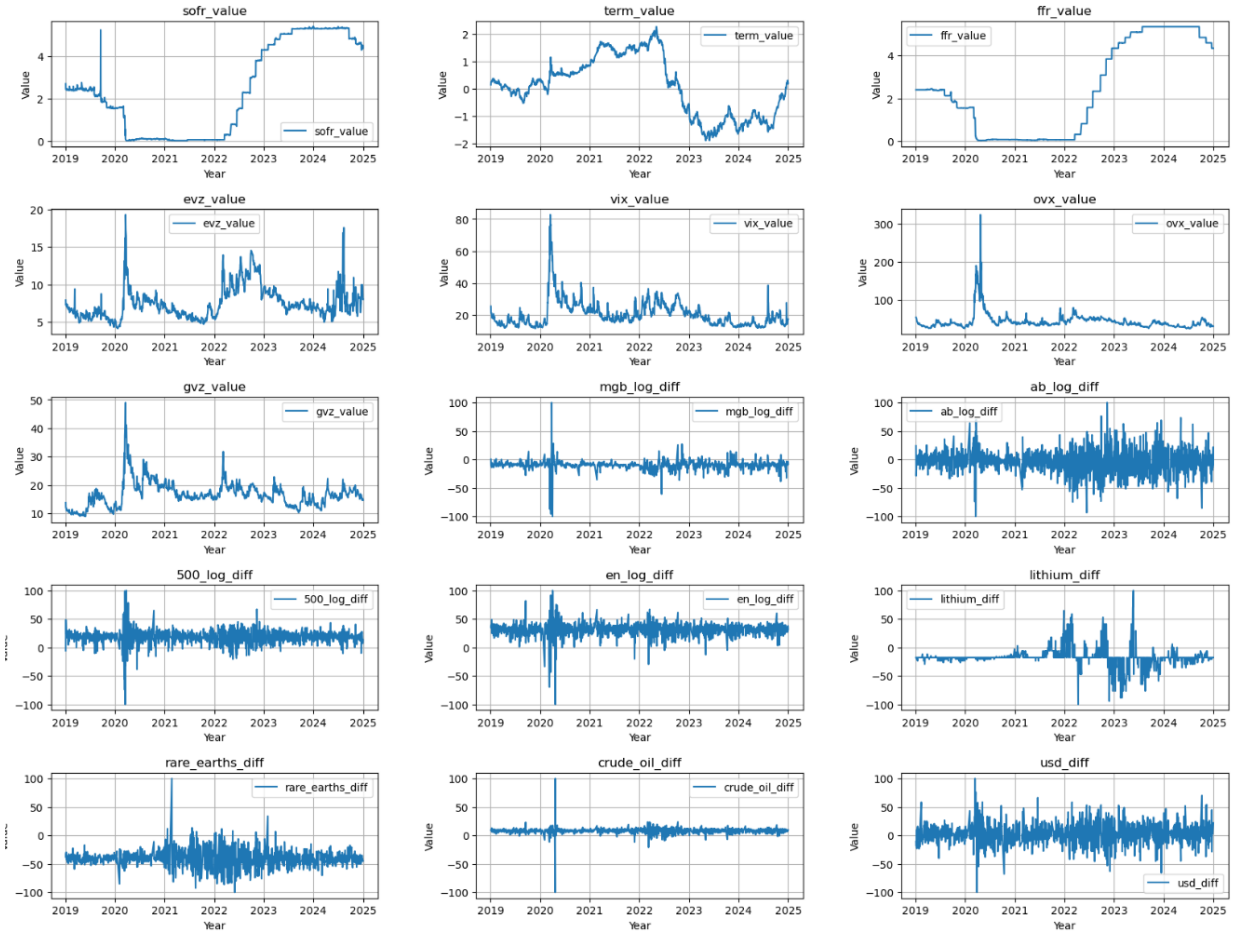


Figure 13: Graph of data after processing

4.3 Hyperparameter Tuning

Because we are working with time series data, we cannot randomly select a portion of our data to be the training set and a portion to be a testing set.

So, over the "Covid", "post-Covid", and "full" periods, we will select the first 80% of the data to be the training set, and the last 20% of the data be the testing set.

Now, for the "Covid", "post-Covid" and "full" periods, we want to do hyperparameter tuning over the training set.

4.3.1 Time Series Cross Validation

To do this, we need to use a special type of cross validation technique called time series cross validation. This will be clearer with a figure.

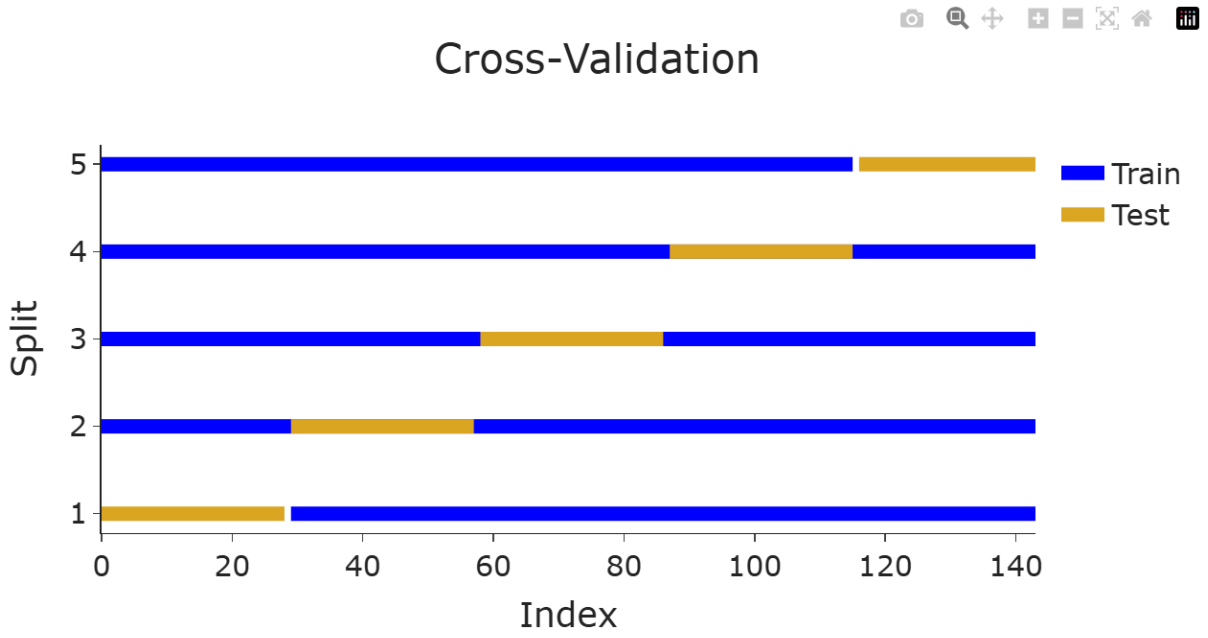


Figure 14: Standard cross validation on training set [14]

Let's imagine that in Figure 14, that we have a training set of just over 140 elements. Under standard 5 fold cross validation (there are 5 equally spaced testing blocks), we would train our model over all the blue data, and test over the yellow data. Then we would average the performance of the model across all the testing splits. This helps prevent overfitting, and makes use of all the training data.

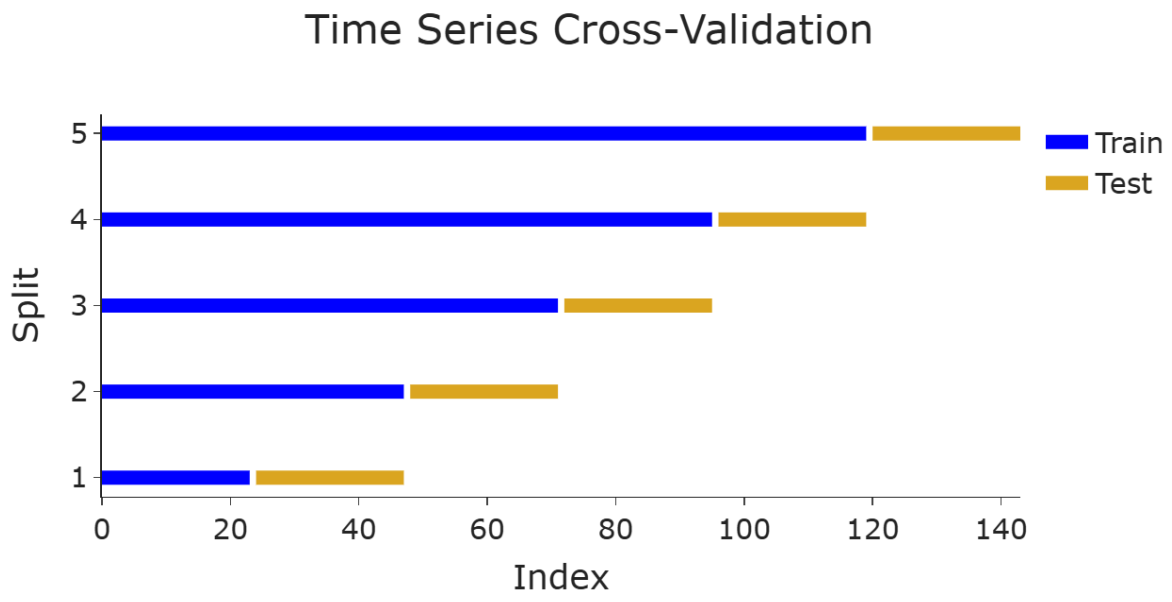


Figure 15: Time series cross validation on training set [14]

In Figure 15, we have an implementation of time series cross validation. The reason that we cannot cannot partition our data the way we have under figure 14 is because time series data is ordered. We cannot use future data to predict the past, because that would be *data leakage*. As such, when doing our 5 fold cross validation, we must only use data before the testing blocks (denoted in yellow). Then, we will average the performance of our model over all the testing splits.

When performing a time series cross validation, we want to make sure to include the default parameters of the models in our hyperparameter testing set. This way we will know if a set of custom parameters will outperform the default parameters. Of course, for reproducibility of our result, we have set a seed for all of our models. We will do 5 fold time series cross validation for our data.

4.3.2 Default Parameters for Tree Models

Random Forest Default Parameters

The default parameters are [24]:

1. n_estimators = 100. n_estimators is the number of trees created by our model.
2. max_depth = None. max_depth = None means there is no max depth on for our decision trees.
3. criterion = "squared_error". This is just a squared error loss.

The combination of hyperparameters that we want to test for Random Forest will be created from all possible combinations of the following choices:

1. max_depth = 6, 8, 10, None
2. n_estimators = 75, 100, 125

XGBoost Default Parameters

The default parameters are [10] [11]:

1. n_estimators = 100. This means we create 100 decision trees.
2. max_depth = 6. The max depth of our decision tree is 6.
3. learning_rate = 0.3. This is α in $F^t = F^{t-1} + \alpha h^t$
4. objective = 'reg:squarederror'. This is just squared error loss.

The combination of hyperparameters that we want to test for XGBoost will be created from all possible combinations of the following choices:

1. max_depth = 4,5,6
2. n_estimators = 75, 100, 125
3. learning_rate = 0.01, 0.02. 0.3

LightGBM Default Parameters

The default parameters are [8] [9]:

1. n_estimators = 100. This means we create 100 decision trees.
2. max_depth = -1. This means there is no max depth for our decision trees.
3. learning_rate = 0.1. This is α
4. num_leaves = 31. This is the maximum number of leaves we can have in a tree.
5. objective = None. When using LightGBM for regression (which we are), this is set to objective = regression. regression is an alias for L2 loss and mean squared error.

The combination of hyperparameters we want to test for LightGBM will be created from all possible combinations of the following choices:

1. max_depth = 4,6,8,-1

2. n_estimators = 100, 150, 200

3. learning_rate = 0.05, 0.075, 0.1

4. num_leaves = $2^{\text{max depth}}$ if max_depth > 4. 31 otherwise. The reason why this code is a bit weird is because we need to account for max_depth = -1.

CatBoost Default Parameters
The default parameters are [7]:

1. iterations = 1000. This means we create 1000 decision trees.

2. max_depth = 6. This means our decision tree can have max depth of 6.

3. learning_rate = None. CatBoost is unique because the learning rate is automatically detected based on dataset properties and number of iterations.

4. loss_function = 'RMSE'

The combination of hyperparameters we want to test for CatBoost will be created from all possible combinations of the following choices:

1. max_depth = 6,7,8

2. iterations = 250, 500, 1000

3. learning_rate = 0.01, 0.3. *Small note*: I have tried learning rate 0.02 as well. 0.01 is preferred over 0.02 and 0.3 in all runs of the code I've done.

4.4 Metrics for Model Evaluation

We used 5 different metrics to evaluate model fit. These were mean squared error (MSE), R^2, root mean square error (RMSE), mean absolute error (MAE), and mean absolute percentage error (MAPE).

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - Y_i)^2 \tag{5}$$

$$R^2 = \frac{\sum_{i=1}^{N}(\hat{Y}_i - \bar{Y}_i)^2}{\sum_{i=1}^{N}(Y_i - \bar{Y}_i)^2} \tag{6}$$

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - Y_i)^2} \tag{7}$$

$$MAE = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - Y_i) \tag{8}$$

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{Y_i - \hat{Y}_i}{Y_i}\right| \cdot 100\% \tag{9}$$

where $\hat{Y}_i$ refers to the predicted Y_i, Y_i is an observed value, and $\bar{Y}_i$ is the mean of observed values.

5 Our Results

5.1 Optimal Hyperparameters

	Covid Period	Post Covid Period	Full Period
Random Forest	max_depth = 8, n_estimators = 100	max_depth = None, n_estimators = 100	max_depth = 8, n_estimators = 100
XGBoost	max_depth = 4, n_estimators = 125, learning_rate = 0.01	max_depth = 4, n_estimators = 100, learning_rate = 0.01	max_depth = 5, n_estimators = 75, learning_rate = 0.01
LightGBM	max_depth = 8, n_estimators = 100, learning_rate = 0.05, num_leaves = 256	max_depth = -1, n_estimators = 100, learning_rate = 0.05, num_leaves = 31	max_depth = 6, n_estimators = 100, learning_rate = 0.05, num_leaves = 64
CatBoost	max_depth = 8 iterations = 250 learning_rate = 0.01	max_depth = 6 iterations = 1000 learning_rate = 0.01	max_depth = 8 iterations = 250 learning_rate = 0.01

Table 1: Table of optimal hyperparameters

5.2 Forecasting the Data

Because there are too many models, (12 total), we will first display the performance of each model at forecasting. Then, for each time period "Covid", "post-Covid" and "full", we will display the forecast of the best model.

	MSE	R^2	RMSE	MAE	MAPE
Random Forest	**52.2143**	**0.1908**	**7.2259**	**4.7556**	**1.38%**
XGBoost	60.0546	0.0693	7.7495	5.0784	1.64%
LightGBM	70.7216	-0.0961*	8.4096	6.0639	1.49%
CatBoost	58.7331	0.0897	7.6638	5.3160	1.60%

Table 2: Metrics of tree models during "Covid"

The value of -0.0961 for the R^2 of LightGBM is *not* a typo. In theory, R^2 cannot be negative, but the way that R^2 is calculated in scikit learn can be negative if our model fit is exceptionally bad.

	MSE	R^2	RMSE	MAE	MAPE
Random Forest	**31.7455**	**0.3286**	**5.6343**	**3.7514**	**1.11%**
XGBoost	36.2397	0.2336	6.0199	3.8957	1.35%
LightGBM	34.6740	0.2667	5.8885	4.0992	1.27%
CatBoost	34.1706	0.2773	5.8456	3.8821	1.18%

Table 3: Metrics of tree models during "post-Covid"

	MSE	R^2	RMSE	MAE	MAPE
Random Forest	29.0826	0.2533	5.3928	3.4206	1.20%
XGBoost	30.9919	0.2043	5.5670	3.5291	1.24%
LightGBM	**23.3112**	**0.4015**	**4.8282**	**3.3778**	**1.05%**
CatBoost	29.2612	0.2487	5.4094	3.4514	1.24%

Table 4: Metrics of tree models during "full"

Thus, we can see the optimal models are: Random Forest for the "Covid" period, Random Forest for the "post-Covid" period, and LightGBM for the "full" period. Let's show those graphs.

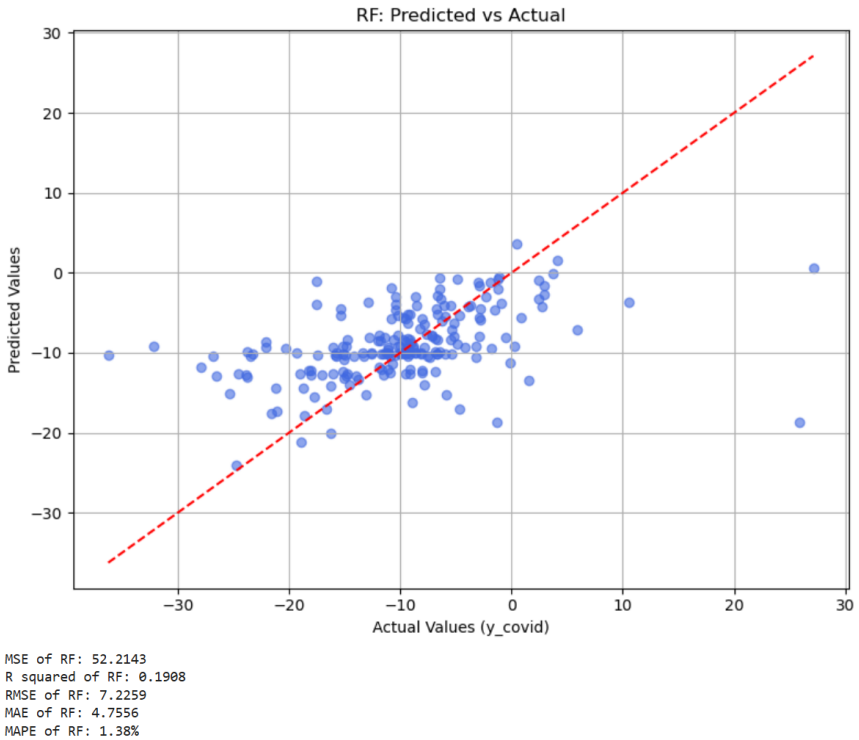
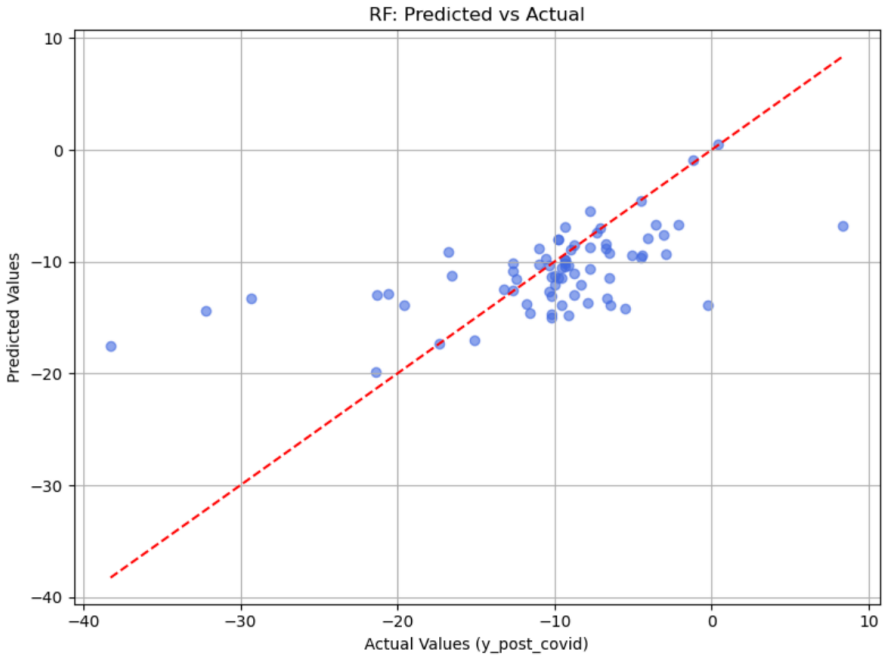


Figure 16: Best model on "Covid" time period, max_depth = 8, n_estimators = 100



MSE of RF: 31.7455
R squared of RF: 0.3286
RMSE of RF: 5.6343
MAE of RF: 3.7514
MAPE of RF: 1.11%

Figure 17: Best model on "post-Covid" time period, max_depth = None, n_estimators = 100

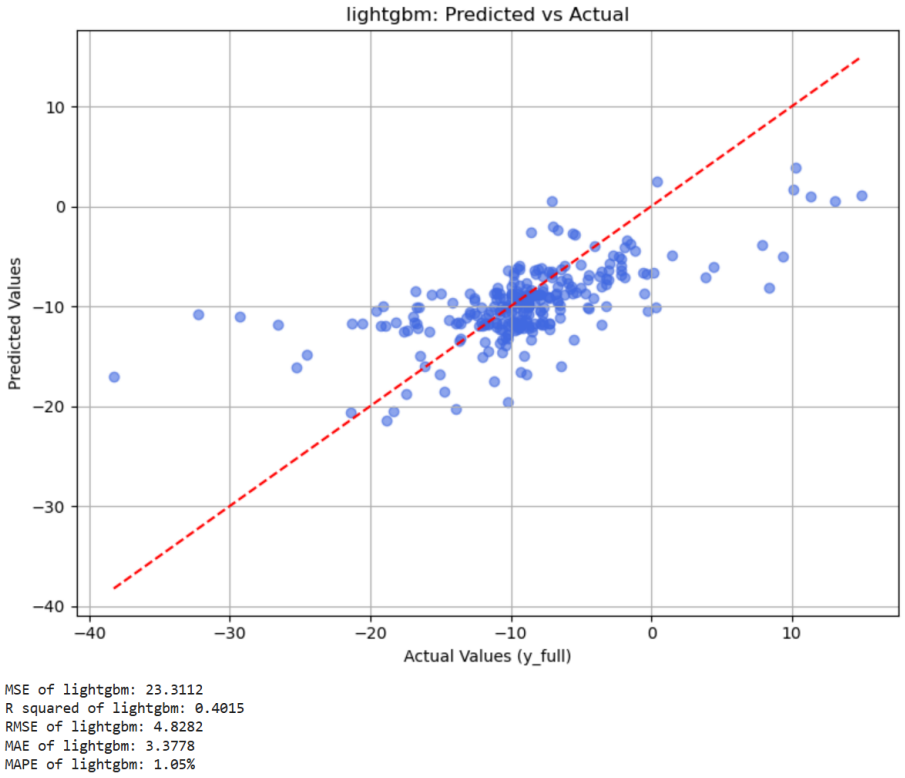


Figure 18: Best model on "full" time period, max_depth = 6, n_estimators = 100, learning_rate = 0.05, num_leaves = 64

5.3 Different Types of SHAP Plots

There are 12 different models. However, we will not show all of these due to space constraints. For the models we will show, we will show 4 types of SHAP plots. These are the SHAP summary plots, SHAP variable importance plots, SHAP dependence plots, and SHAP heatmaps. For the 4 different types of SHAP plots, the only one we haven't seen before are SHAP heatmaps.

5.3.1 SHAP Heatmaps

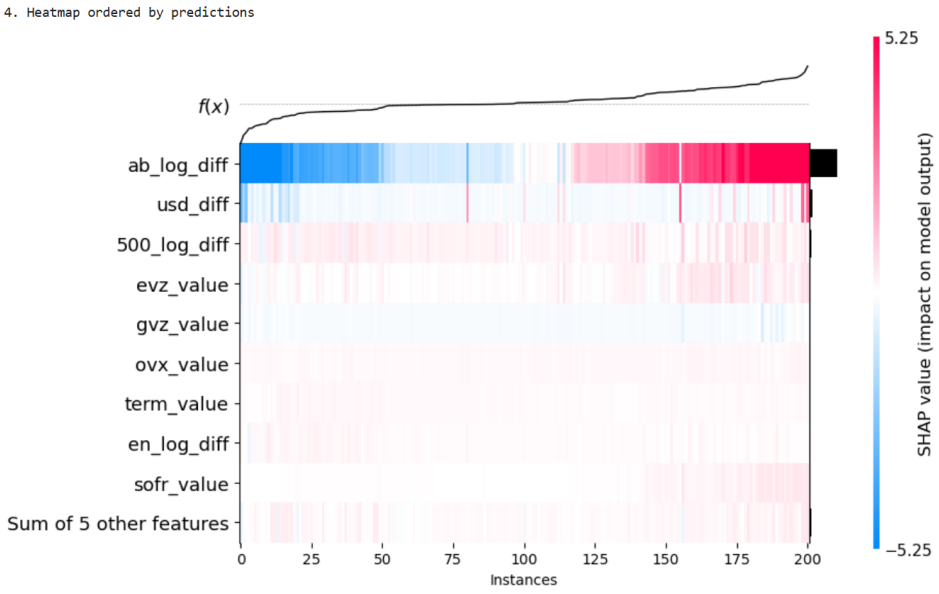


Figure 19: SHAP heatmap for Random Forest on "Covid" period

Above is an illustrative example of a SHAP heatmap. The way to read the heatmap is that $f(x)$ is the predicted value of our variable mgb_log_diff over the testing set. The intensity of the color (dark blue vs dark red) shows how much the SHAP value changes. A dark blue color means SHAP values decrease significantly, while a dark red color means SHAP values increase significantly. We can see from Figure 19, that ab_log_diff has the greatest effect on determining the value of mgb_log_diff over the testing set.

5.4 SHAP Plots of the Optimal Model over Different Time Periods

Remember that over the optimal model over "Covid" was Random Forest, over "post-Covid" was Random Forest, and over "full" was lightgbm. We will compare how each of the models differ.

5.4.1 SHAP Plots of Optimal Model over "Covid"(RF)

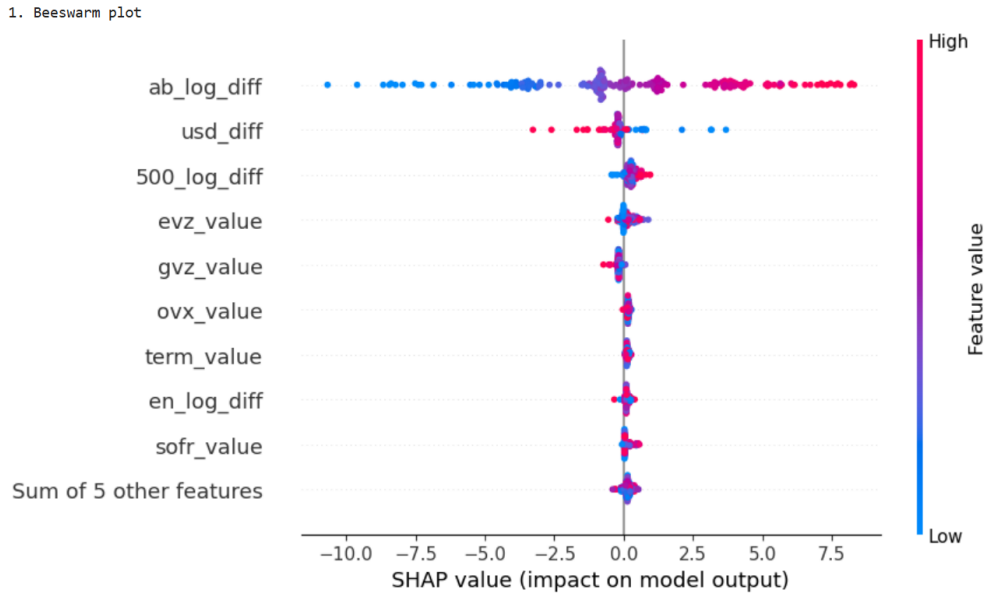


Figure 20: SHAP Summary plot of RF over "Covid" time period

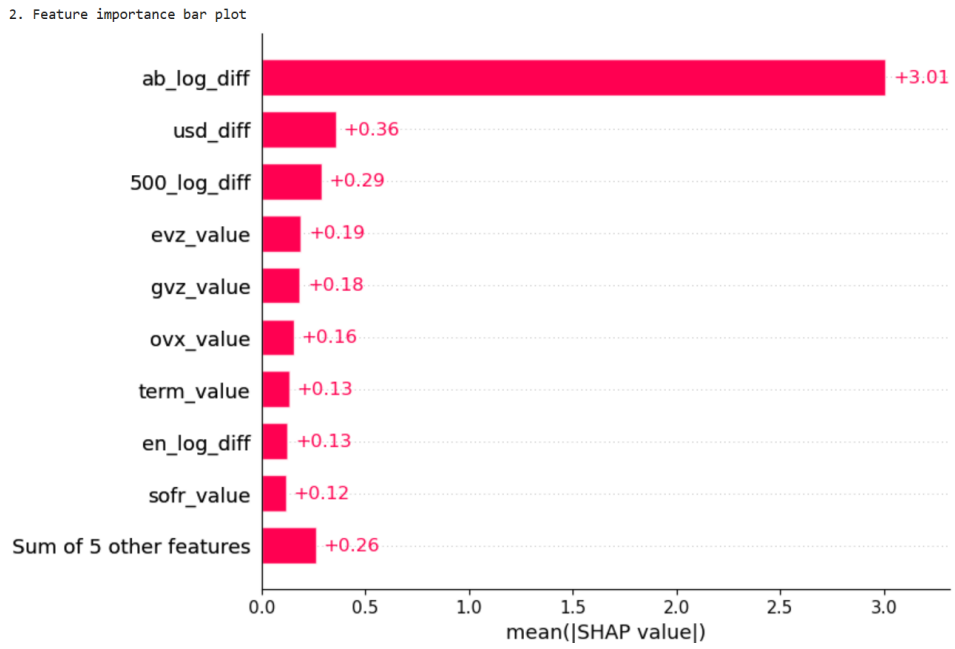


Figure 21: SHAP variable importance plot of RF over "Covid" time period

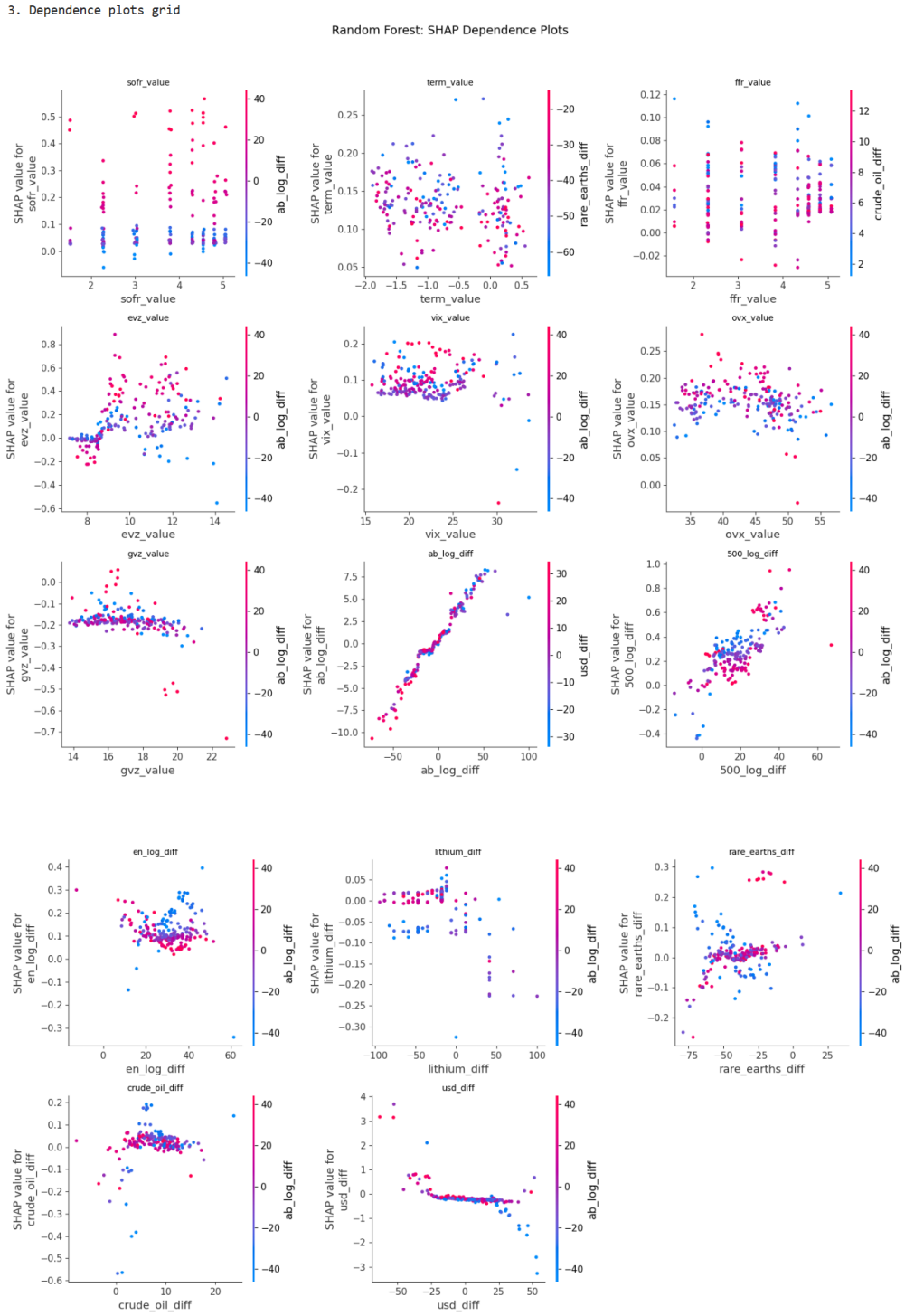


Figure 22: SHAP dependence plot of RF over "Covid" time period

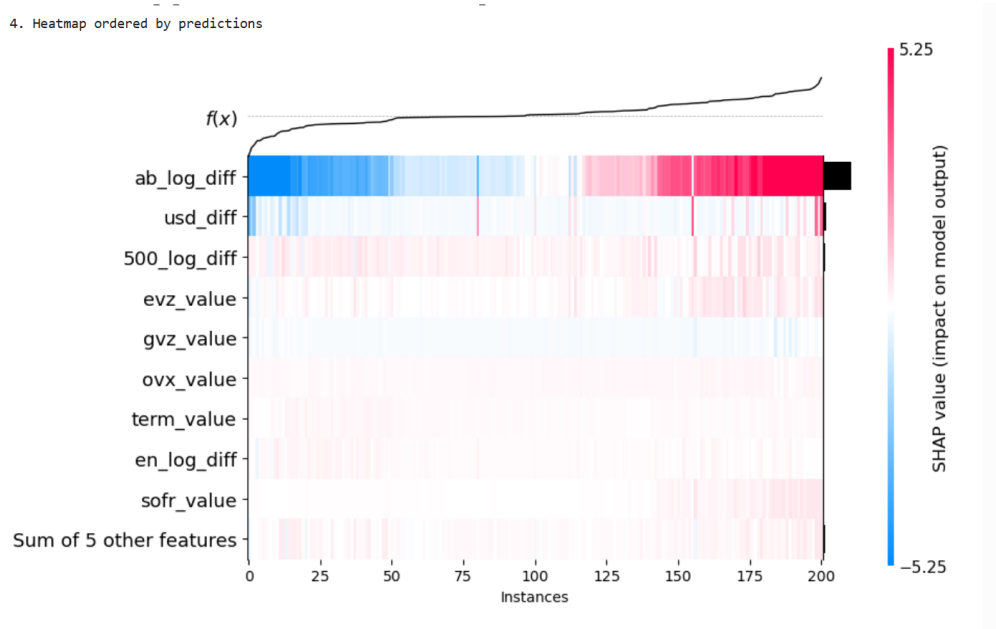


Figure 23: SHAP heatmap of RF over "Covid" time period

Over the "Covid" period, it is evident that ab_log_diff is the largest predictor of mgb_log_diff (log differed municipal green bond index prices). This is not unexpected. The 2nd and 3rd most influential predictors were usd_diff, and 500_log_diff.

5.4.2 SHAP Plots of Optimal Model over "Post-Covid"(RF)

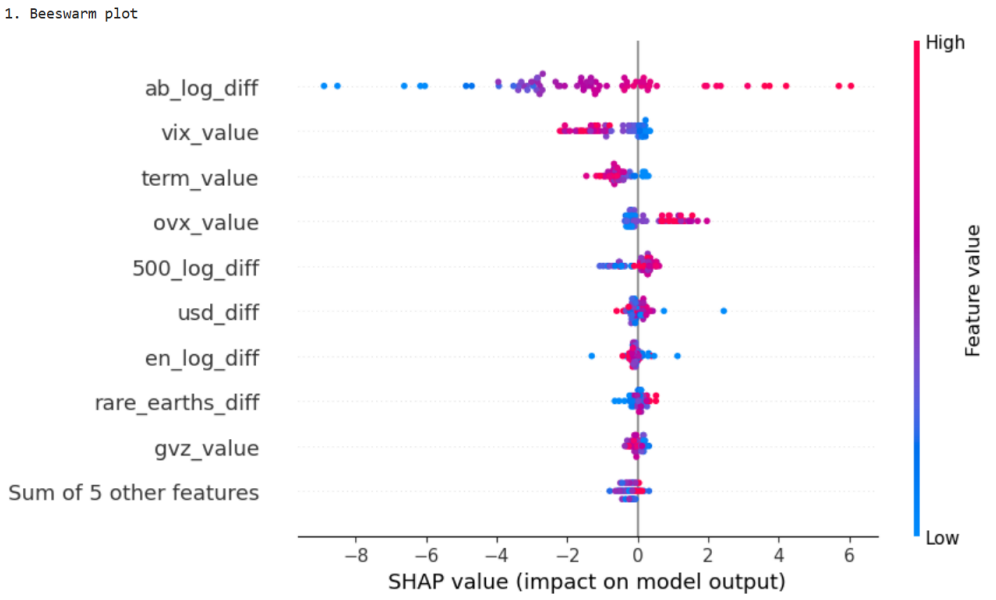


Figure 24: SHAP Summary plot of RF over "post-Covid" time period

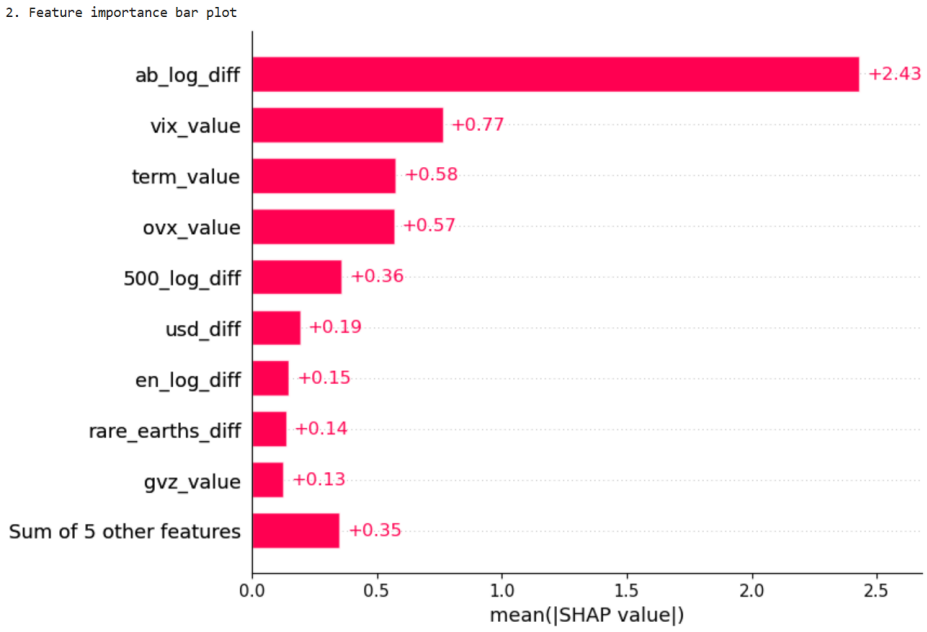
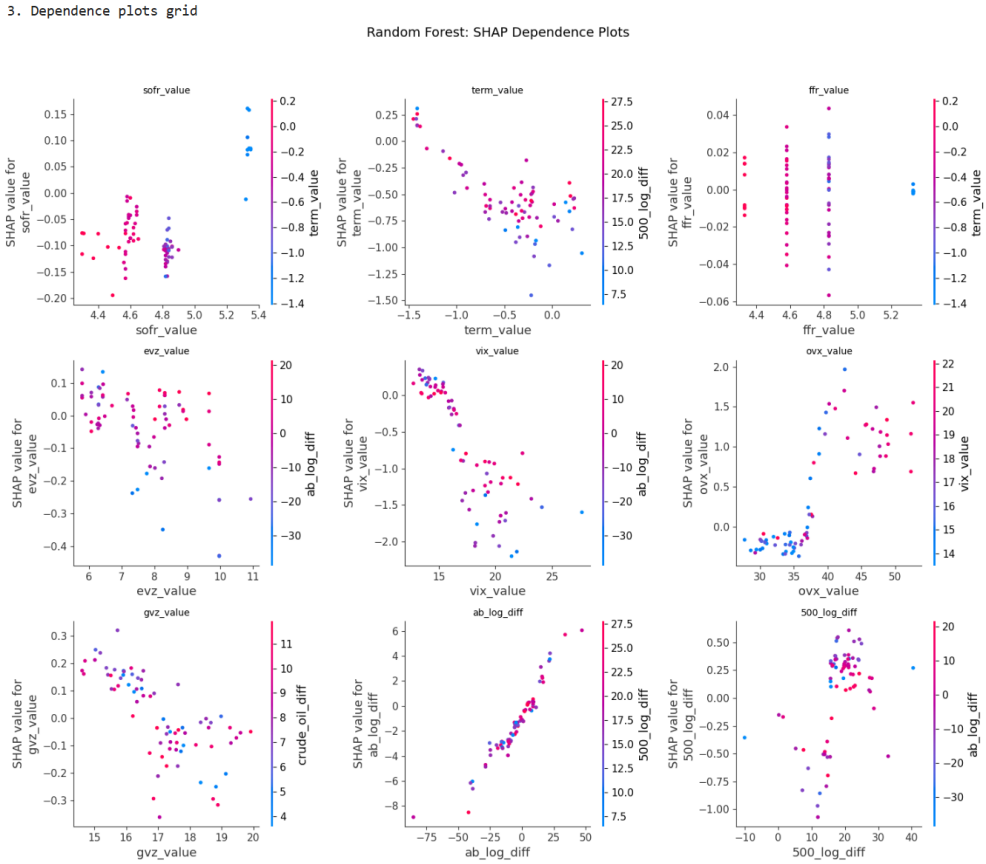


Figure 25: SHAP variable importance plot of RF over "post-Covid" time period



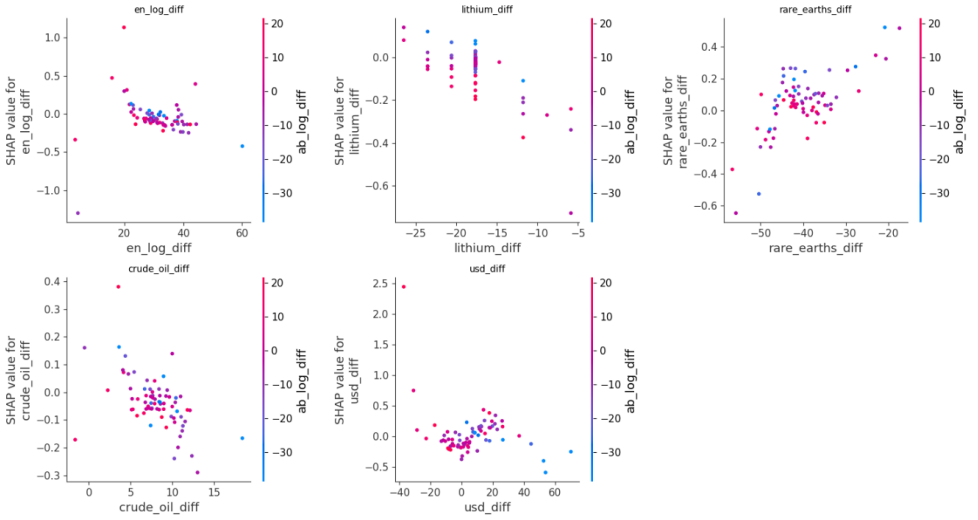


Figure 26: SHAP dependence plot of RF over "post-Covid" time period

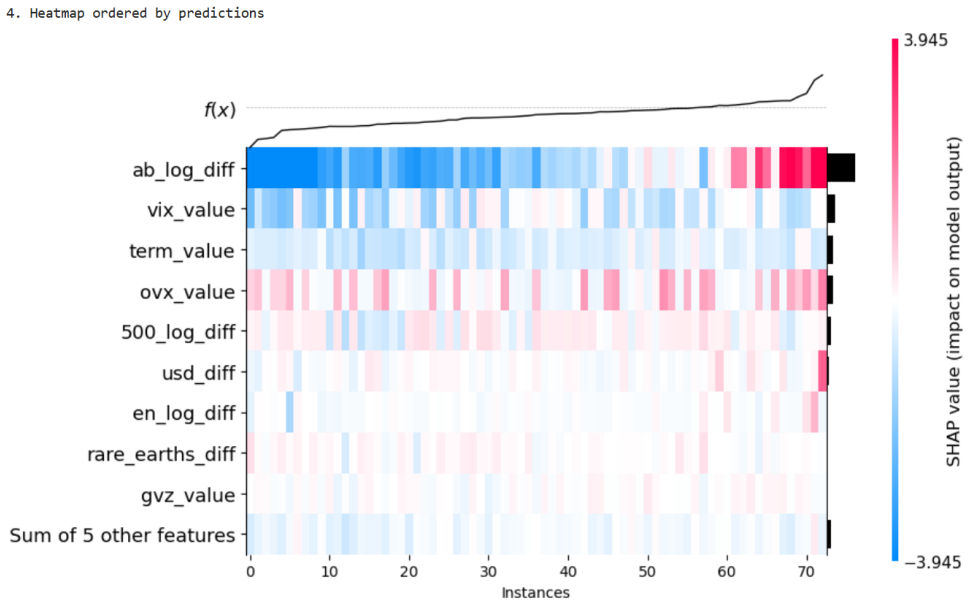


Figure 27: SHAP heatmap of RF over "post-Covid" time period

Over the "post-Covid" time period, the biggest predictor of mgb_log_diff was ab_log_diff. However, the 2nd and 3rd most influential predictors during this time were vix_value and term_value.

5.4.3 SHAP Plots of Optimal Model over "Full"(LightGBM)

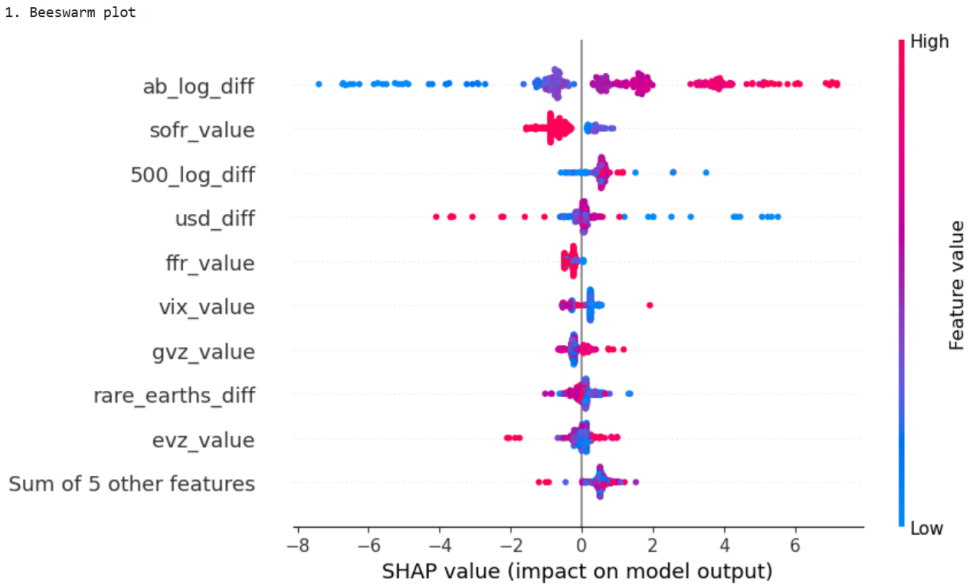


Figure 28: SHAP Summary plot of LightGBM over "full" time period

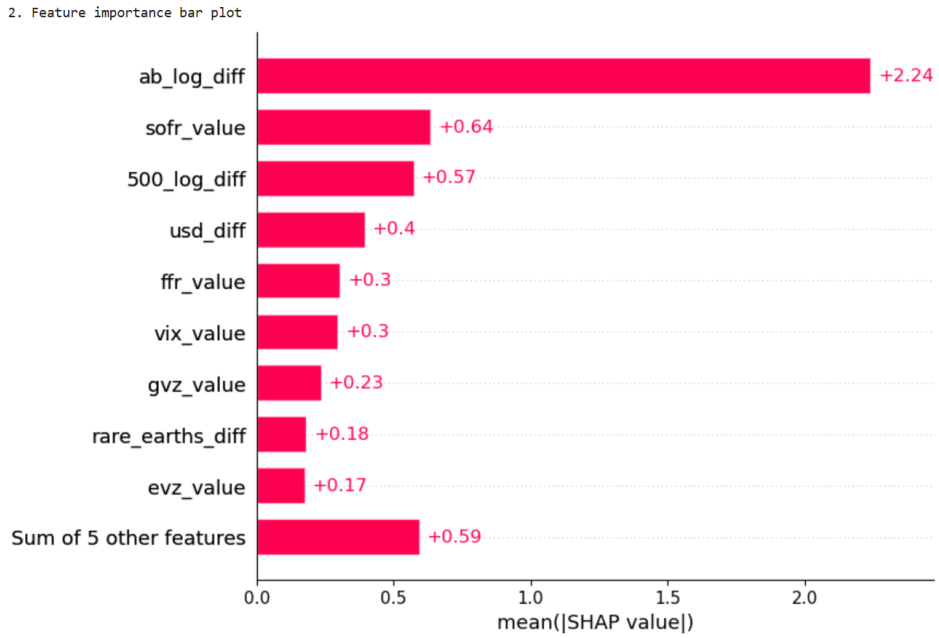


Figure 29: SHAP variable importance plot of LightGBM over "full" time period

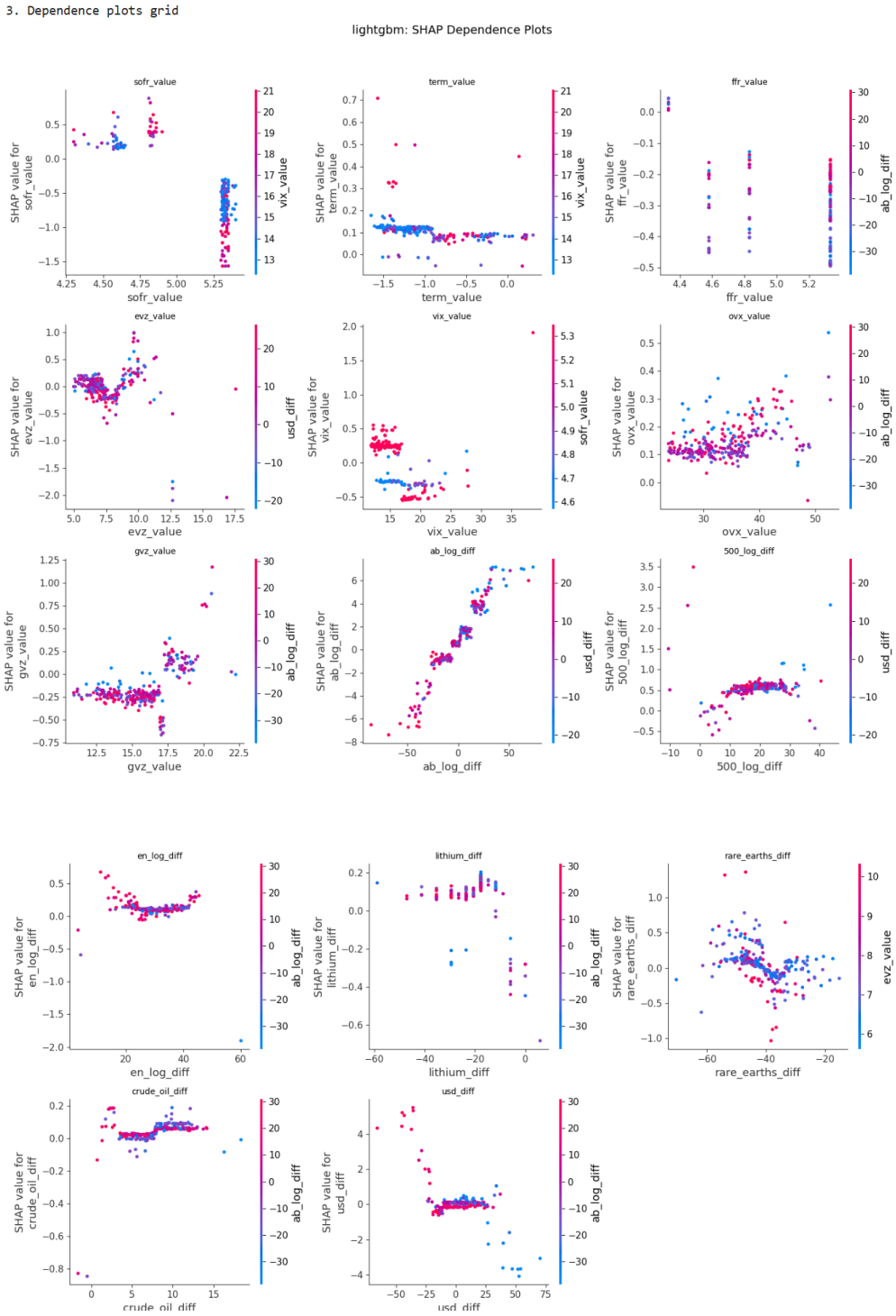


Figure 30: SHAP dependence plot of LightGBM over "full" time period

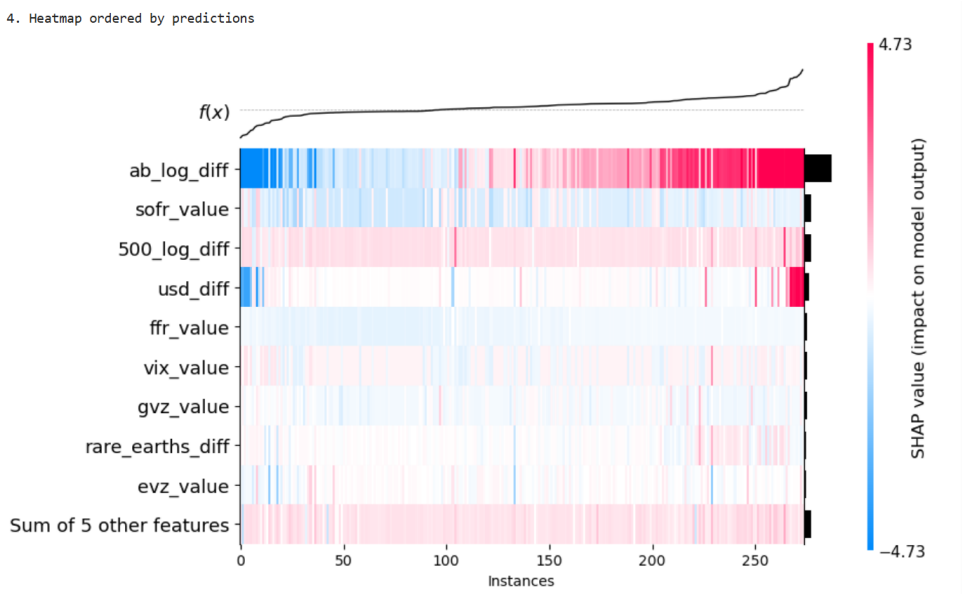


Figure 31: SHAP heatmap of LightGBM over "full" time period

Over the "full" time period, the biggest predictor of mgb_log_diff was ab_log_diff. However, the 2nd and 3rd most influential predictors during this time were sofr_value and 500_log_diff.

5.5 SHAP Plots for All Models over the "Full" Period

For the sake of completeness, here are all of the models over the "full" period except LightGBM (because we have already shown those plots).

5.5.1 Graphs for Random Forest

The below graphs are for the "full" period for Random Forest.

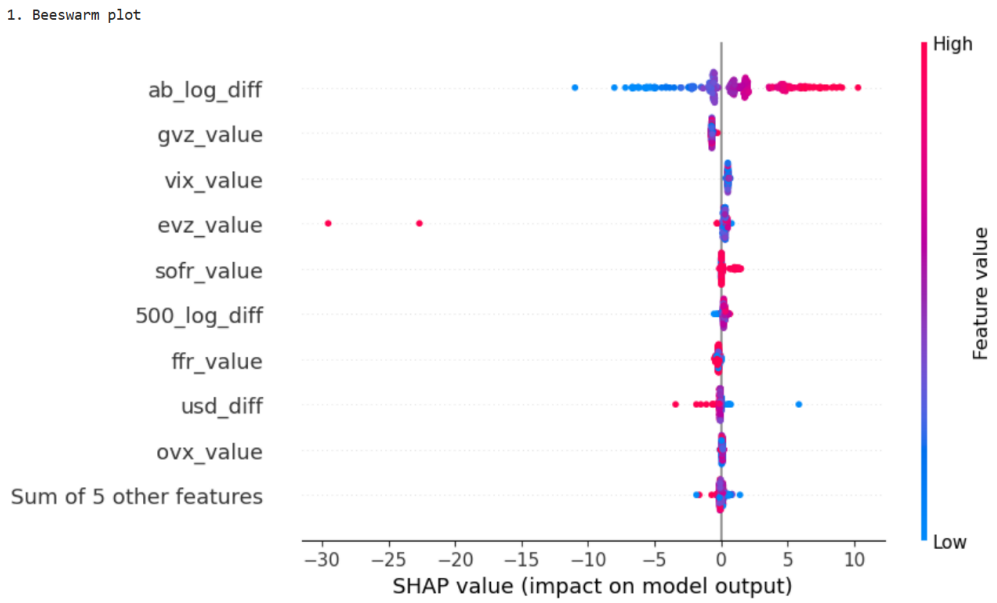


Figure 32: SHAP Summary plot of RF over "full" time period

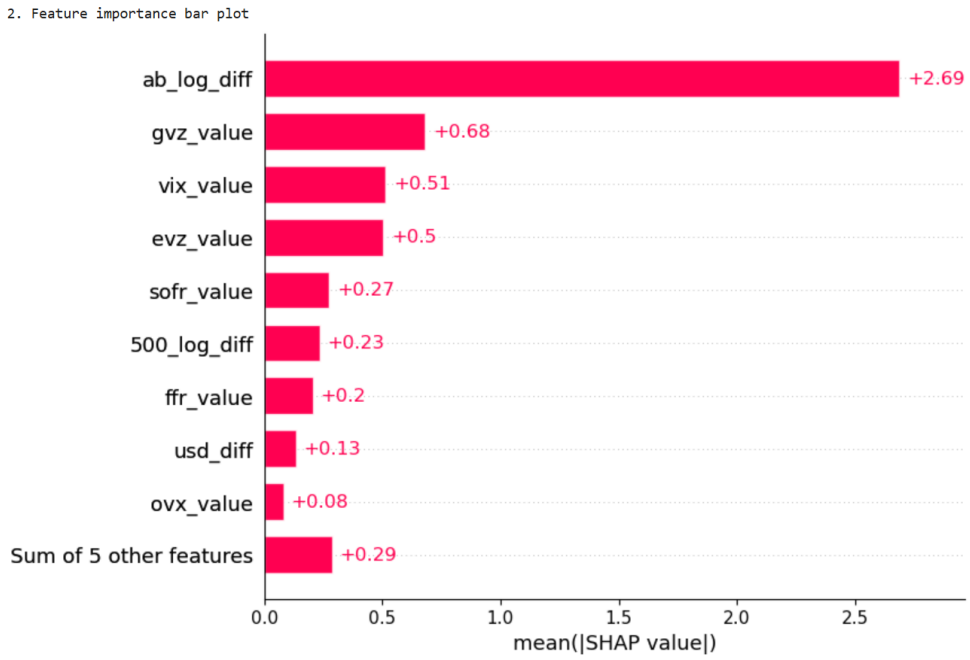


Figure 33: SHAP variable importance plot of RF over "full" time period

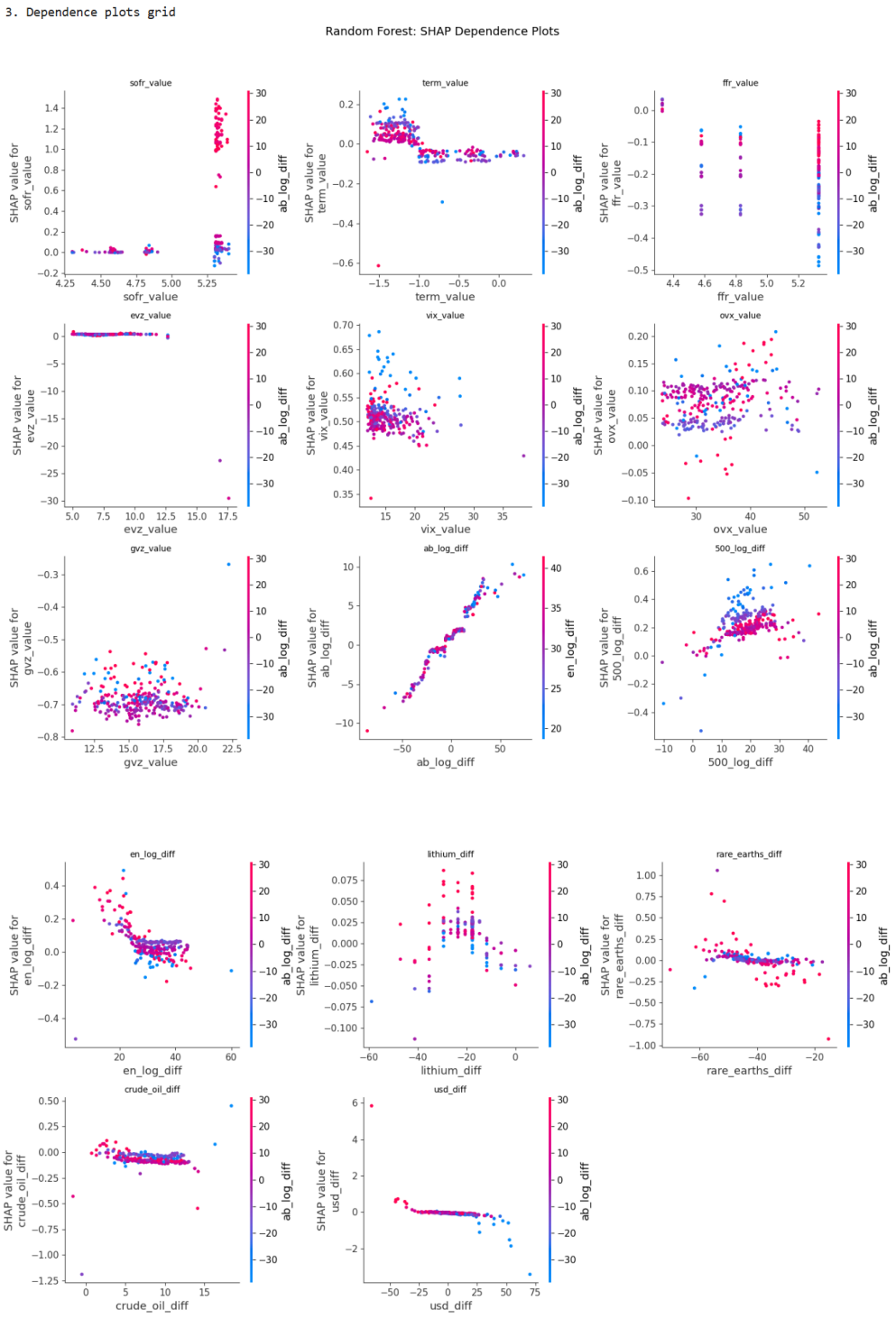


Figure 34: SHAP dependence plot of RF over "full" time period

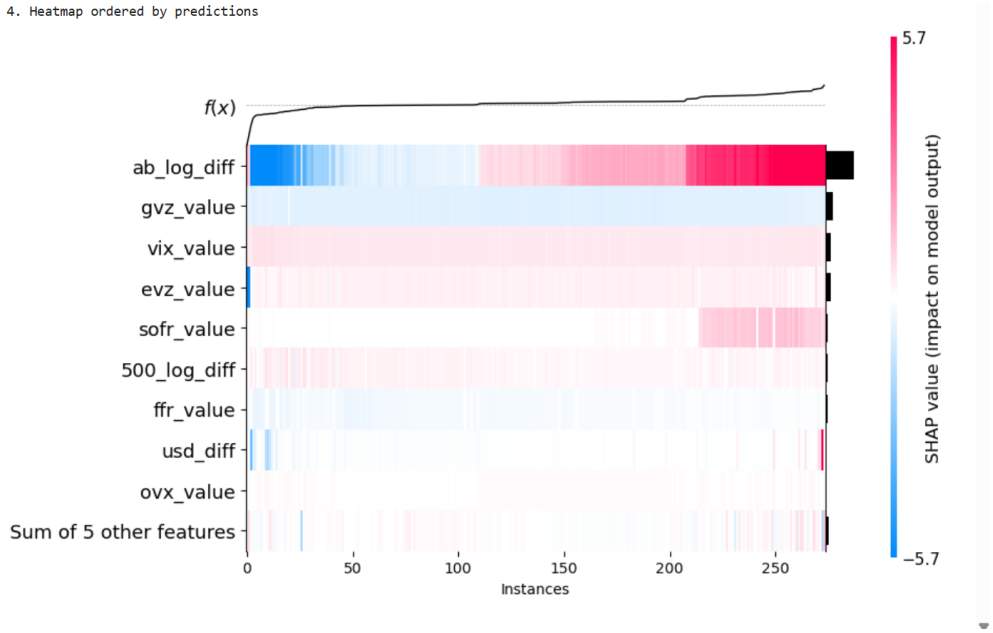


Figure 35: SHAP heatmap of RF over "full" time period

5.5.2 Graphs for XGBoost

The below graphs are for the "full" period for XGBoost.

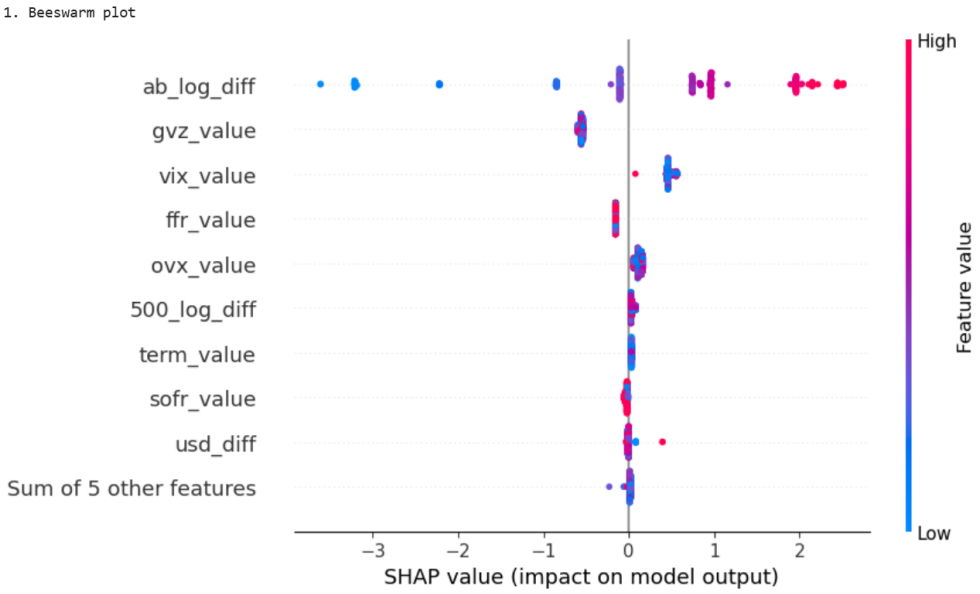


Figure 36: SHAP Summary plot of XGBoost over "full" time period

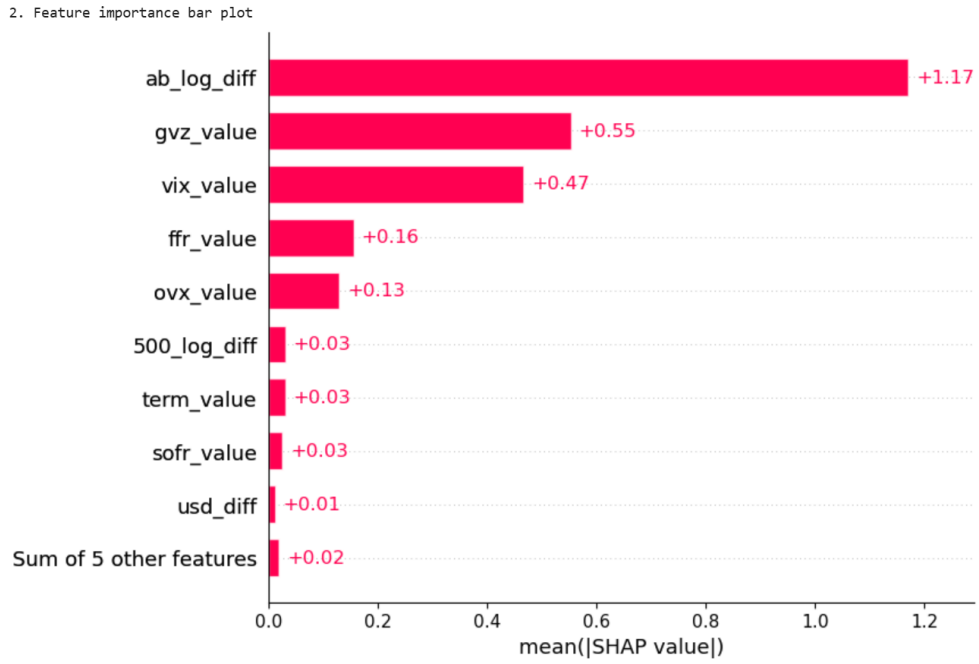
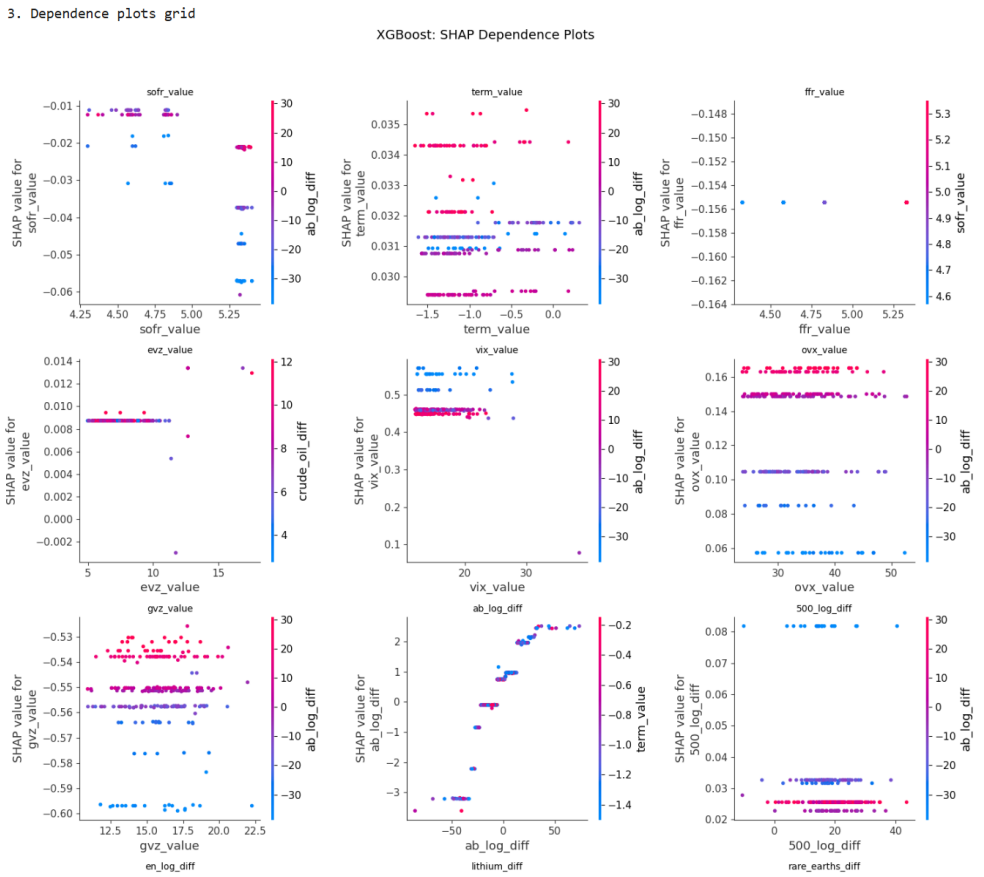


Figure 37: SHAP variable of importance plot XGBoost over "full" time period



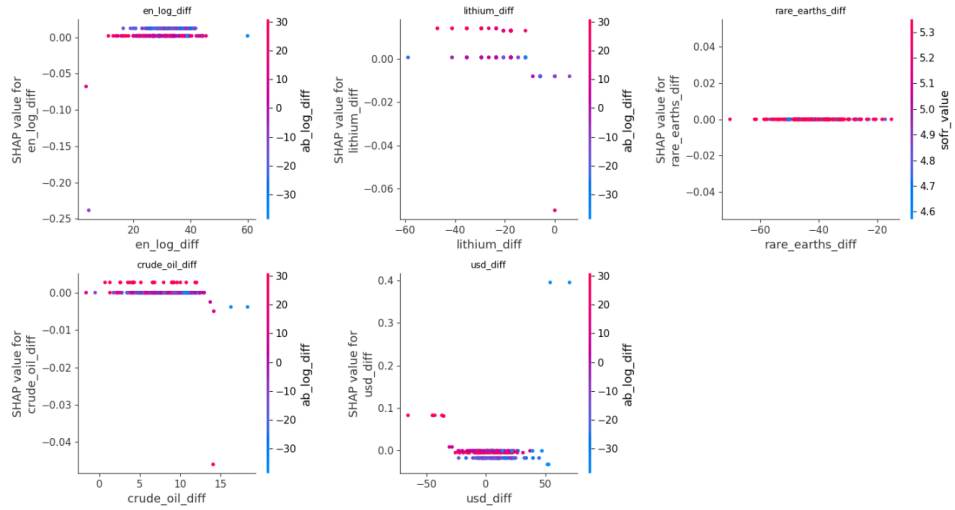


Figure 38: SHAP dependence plot of XGBoost over "full" time period

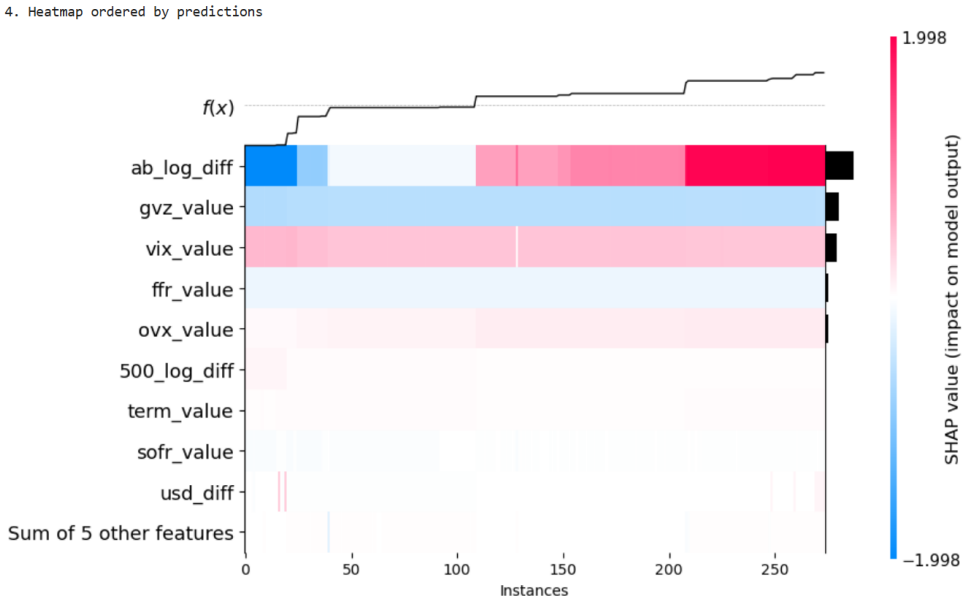


Figure 39: SHAP heatmap of XGBoost over "full" time period

5.5.3 Graphs for CatBoost

The below graphs are for the "full" period for CatBoost.

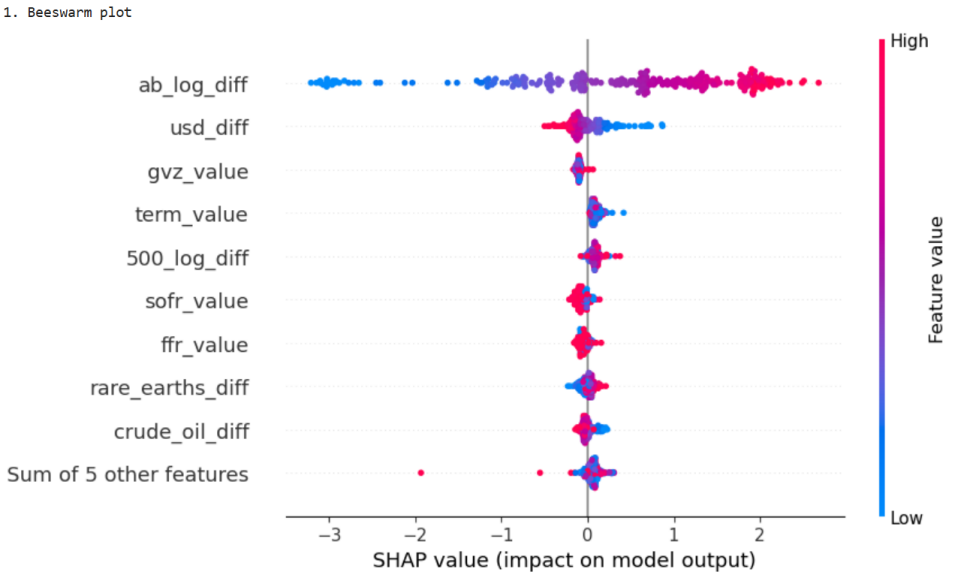


Figure 40: SHAP Summary plot of CatBoost over "full" time period

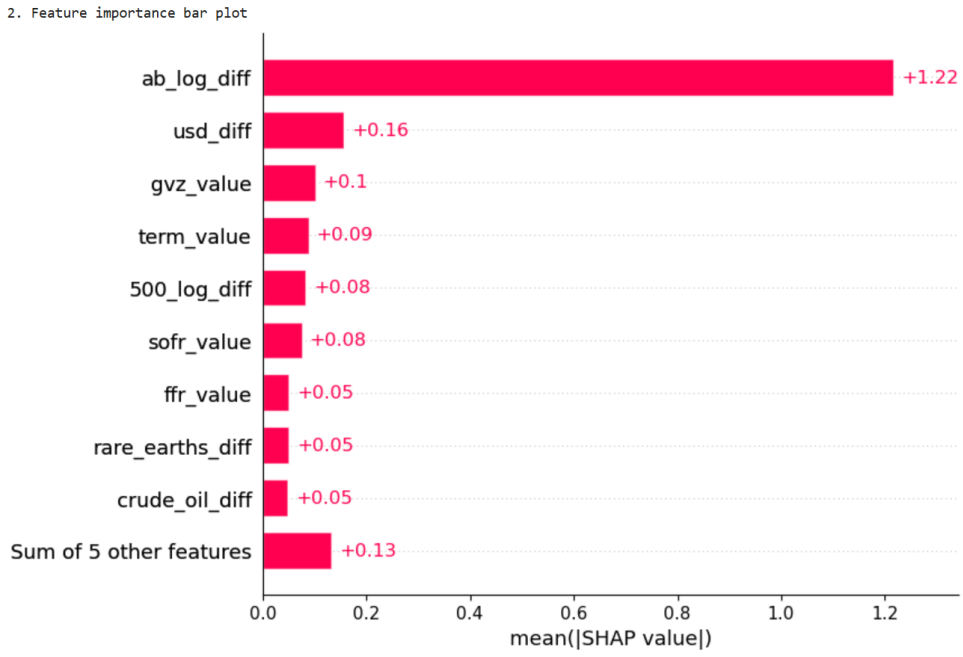


Figure 41: SHAP variable importance plot of CatBoost over "full" time period

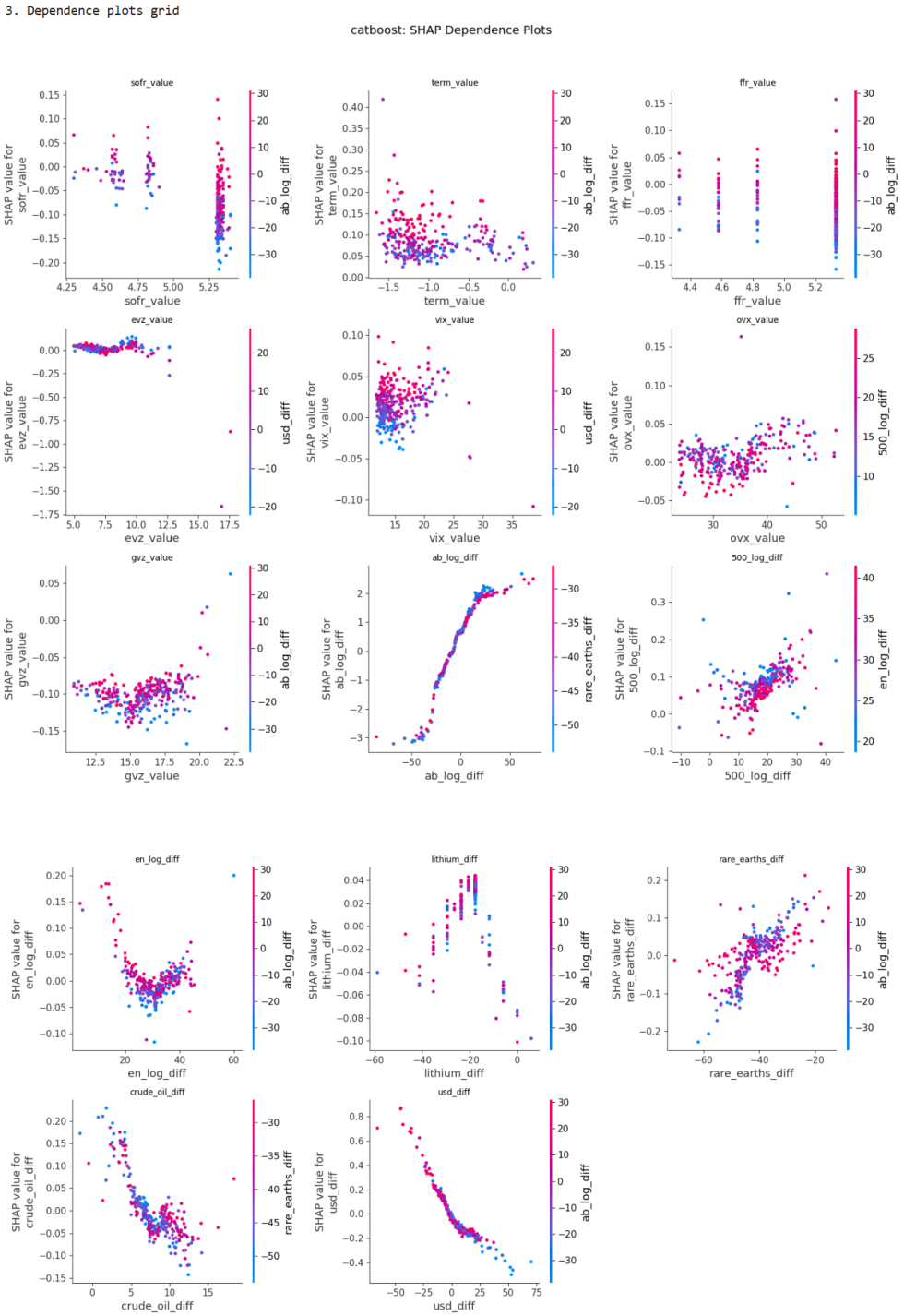


Figure 42: SHAP dependence plot of CatBoost over "full" time period

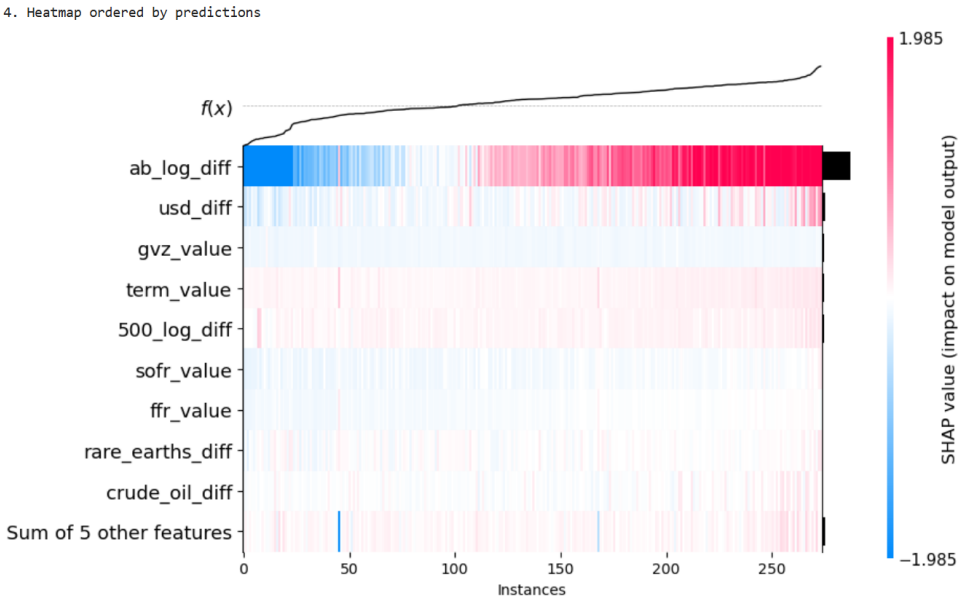


Figure 43: SHAP heatmap of CatBoost over "full" time period

6 Critical Evaluation of Methods

There are many ways that our results are not ideal and can be improved.

6.1 Time Series Data is Inherently not i.i.d.

One of the biggest issues with our model is that we have violated the i.i.d. assumption in models like Random Forest, XGBoost, LightGBM, and CatBoost. However, because our data is time series data, this is not something we can easily remedy. This is is due to the autocorrelation in time series data. Even if we make the data stationary, we cannot make the data points independent of each other. In our model, we have kept some variables at the level they were imported. There were, nyf_sofr, fred_term, fred_ffr, fred_evz, cboe_vix, cboe_ovx, cboe_gvz. We did this because the information is encoded through the level of the data. This also goes against the theory behind our tree based models.

6.2 Machine Learning is a Black Box

Another issue is interpreting machine learning output and having a good sense of the "correctness" of our results. The main difference between machine learning and classic statistics is causality. A machine learning may produce a model with a high R^2 on a testing set, but that doesn't mean you can fully trust the model. As the example done in Kocaarslan showed, a high R^2 may actually be a sign of *data leakage*, and not a supremely trained model. There is no specification test we can do to check if our models are "good", which is unlike in classical statistics. Generally, there is no way to produce a confidence interval when doing machine learning. We can only hope that our sample size over the training set is large enough that our results are "good enough". Shapley values are also only locally calculated. The Shapley summary plots are calculated by changing only one variable at a time, and holding all other variables constants. As such, the SHAP summary plots cannot capture joint effects. When looking at SHAP dependence plots, we can only capture the joint effect of two variables at a time. However, the SHAP dependence plots are not very human readable. Even when the SHAP dependence plots are readable, explaining the *why* behind why two variables have a joint effect is difficult.

6.3 Hyperparameter Tuning Can be Improved

The hyperparameter tuning that we've done can be significantly improved. This is because we have barely scratched the surface of all the options available for hyperparameter tuning. Generally, we have only touched the most basic of hyperparameters like n_estimators, and max_depth. Due to the limitation of our hardware, it was not feasible to do cross validation

with too many hyperparameters. Running cross validation for CatBoost would take 15 minutes on our hardware. For this same reason, we had limited our default k-fold cross validation amount to 5.

7 Conclusion

In this paper, we have attempted to replicate similar results to Kocaarslan's *The role of major markets in predicting the U.S. municipal green bond market performance: New evidence from machine learning models* [16]. In this paper, we have used Random Forest, XGBoost, LightGBM, and CatBoost in order to forecast the price of a municipal green bond index over three distinct time periods. These time periods are the "Covid" time period (Jan 2nd, 2019 to May 31st, 2023), the "post-Covid" time period (June 1st, 2023 to Dec 31st, 2024), and "full" time period("Covid" + "post-Covid" time period). Overall, we find that the best models for forecasting over the "Covid" and "post-Covid" time period were Random Forest, and the best model over the "full" period was lightGM. Over all models, and over all time periods, we see that by far the biggest predictor of municipal green bond index prices was the aggregate bond index by S&P Global which tracks conventional bonds.

References

[1] Emmanuel Joel Aikins Abakah, Aviral Kumar Tiwari, Aarzoo Sharma, and Dorika Jeremiah Mwamtambulo. Extreme connectedness between green bonds, government bonds, corporate bonds and other asset classes: Insights for portfolio investors. *Journal of Risk and Financial Management*, 2022.

[2] Scott Baker. Economic policy uncertainty index, 2016.

[3] Leo Breiman. Bagging predictors. *Machine Learning*, 1996.

[4] Leo Breiman. Random forests. *Machine Learning*, 2001.

[5] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794. ACM, August 2016.

[6] Wikipedia contributors. Graph coloring. Wikipedia, The Free Encyclopedia, 2025. Accessed: 2025-08-03.

[7] CatBoost Developers. Catboostregressor, 2025.

[8] LightGBM Developers. lightgbm.lgbmregressor, 2025.

[9] LightGBM Developers. Parameters, 2025.

[10] XGBoost Developers. Python api reference, 2025.

[11] XGBoost Developers. Xgboost parameters, 2025.

[12] Torsten Ehlers and Frank Packer. Green bond finance and certification. *BIS Quarterly Review*, 2017.

[13] Shubham Gandhi. Why catboost works so well: The engineering behind the magic, 2025.

[14] Egor Howell. Cross-validation for time series forecasting — python tutorial, 2024.

[15] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: a highly efficient gradient boosting decision tree. NIPS'17, page 3149–3157, Red Hook, NY, USA, 2017. Curran Associates Inc.

[16] Baris Kocaarlsan and Ugur Soytas. The role of major markets in predicting the u.s. municipal green bond market performance: New evidence from machine learning models. *Technological Forecasting and Social Change*, 2023.

[17] Martin Krzywinski and Naomi Altman. Classification and regression trees. *Nature Methods*, 2017.

[18] L Lovasz and K.Vesztergombi. Discrete mathematics: Lecture notes, 1999. Yale University, Spring 1999.

[19] Scott M. Lundberg, Gabriel G. Erion, and Su-In Lee. Consistent individualized feature attribution for tree ensembles, 2019.

[20] Muhammad Abubakr Naeem, Sitara Karim, Gazi Salah Uddin, and Juha Junttila. Small fish in big ponds: Connections of green finance assets to commodity and sectoral stock markets. *International Review of Finanical Analysis*, 2022.

[21] A Data Odyssey. The mathematics behind shapley values, 2023. YouTube video.

[22] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features, 2018.

[23] Donald Rubin. The bayesian bootstrap, 1981.

[24] scikit-learn developers. Randomforestregressor, 2025.

[25] L. S. Shapley. A value for n-person games. *Theory Games*, 1952.

[26] TutorialsPoint. Introduction to trees. https://www.tutorialspoint.com/discrete_mathematics/introduction_to_trees.htm, 2025. Accessed: July 30, 2025.

8 Appendix: Link to Code

The accompanying code to the Master's Research Paper can be found on Github: Github Code For Master's Research Paper

9 Appendix: Theory Behind Tree Learning Models

9.1 Decision Trees

To implement a decision tree, we first need a basic understanding of trees. Let us build up the definition of decision tree.

9.1.1 Graphs

Our presentation below is based on Lovasz and Vesztergombi (1999) [18].

Definition 9.1 (**Graphs**)**.** A graph consists of a set of *nodes* (or *points*, or *vertices*, all these names are in use), and some pairs of these (not necessarily all pairs) are connected by *edges* (or *lines*). The set of nodes of a graph G is usually denoted by V; the set of edges, by E. Thus we write $G = (V, E)$ to indicate that the graph G has node set V and edge set E. [18]

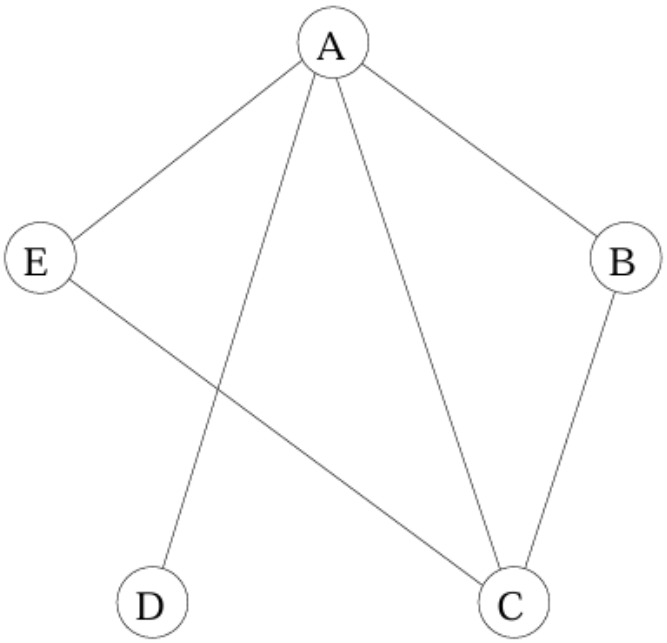


Figure 44: Example graph [18]

In the example graph Figure 44 above, we can see that $V = \{A, B, C, D, E\}$. For simplicity, if some edge $e \in E$ connects the two nodes $\{A, B\}$, we will denote that edge as ab. So $E = \{ab, ac, ad, ae, bc, ce\}$. So, our graph can be written as $G = (\{A, B, C, D, E\}, \{ab, ac, ad, ae, bc, ce\})$

Definition 9.2 (**Subgraph**)**.** A graph H is called a *subgraph* of a graph G if it can be obtained from G by deleting some of its edges and nodes. [18]

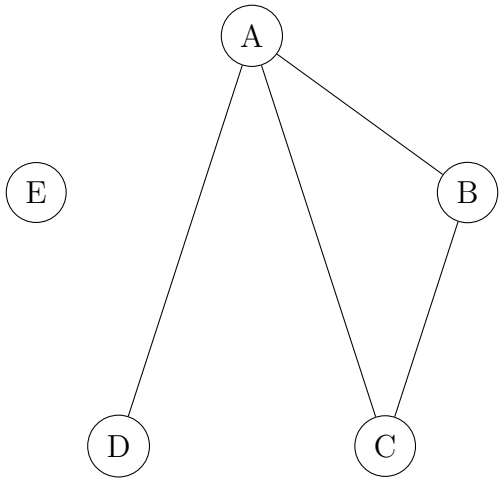


Figure 45: A simple subgraph of figure 44

We can see in Figure 45, that we have created a simple subgraph of G in Figure (44). Notice that we have removed elements ae and ce from E. So, $H = (\{A, B, C, D, E\}, \{ab, ac, ad, bc\})$.

Definition 9.3 (**Path**)**.** Let us draw n nodes in a row and connect the consecutive ones by an edge. This way we obtain a graph with $n - 1$ edges, which is called a *path* [18].

Definition 9.4 (**Cycle**)**.** If we also connect the last node to the first, we obtain a *cycle* [18].

Definition 9.5 (**Connected Graph**)**.** A graph G is *connected* if every two nodes of the graph can be connected by a path in G. To be more precise, a graph G is connected if for every two nodes u and v, there exists a path with endpoints u and v which is a subgraph of G. [18]

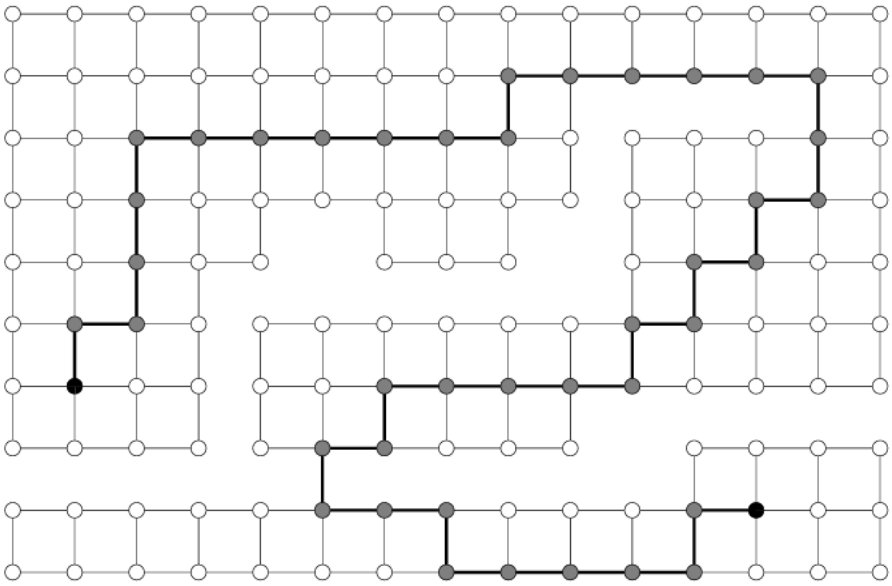


Figure 46: Example of a connected graph [18]

In Figure 46 above, we see an example of a connected graph. Every single node u can be accessed by a path starting in any other node v.

We are now ready to define a tree.

9.1.2 Trees

Definition 9.6 (**Trees**)**.** A graph $G = (V, E)$ is called a *tree* if it is connected and contains no cycle as a subgraph [18].

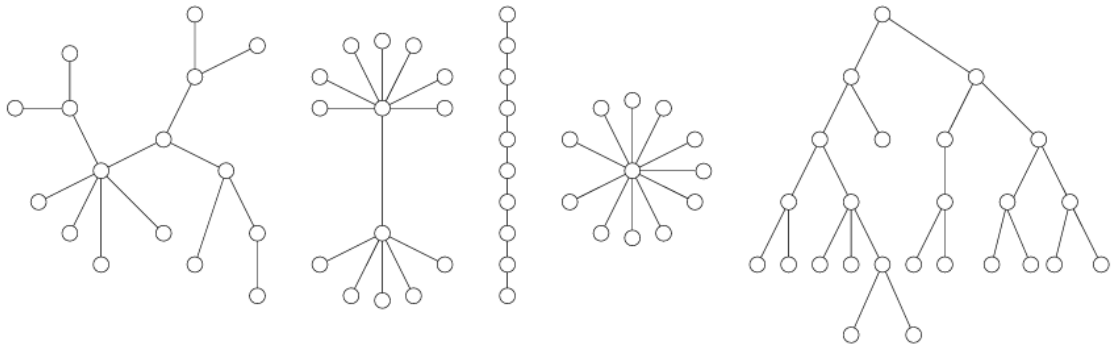


Figure 47: Example trees [18]

Often, we want to add label a specific node of our tree as a *root node*. A tree with a specified *root node* is called a *rooted tree*. The reason that we want our trees to be rooted trees is because clearly labeled nodes help when explaining complicated ideas.

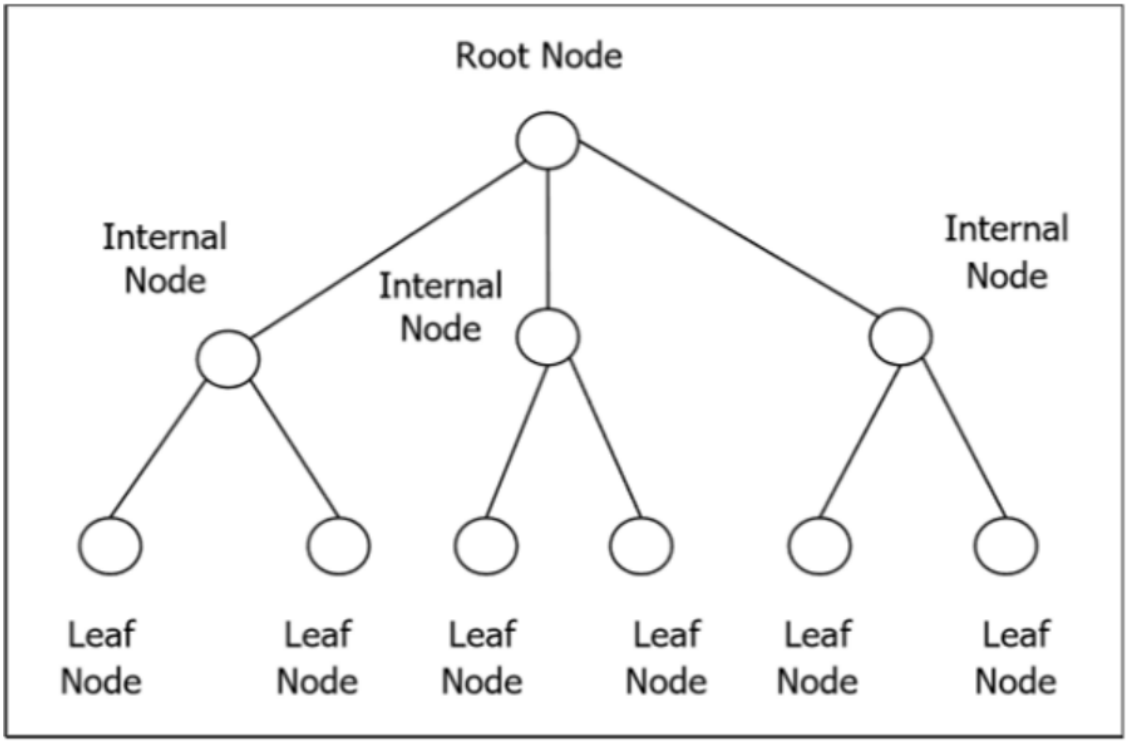


Figure 48: Example trees [26]

By convention, we write the root node on top of the tree. In Figure 48, we see that the root node has exactly 3 child nodes.

Definition 9.7 (**Types of Nodes**)**.** We will elaborate on the different types of nodes.

1. *Child nodes* are the nodes directly connected to the node in the level above it.
2. *Parent node* is the node directly connected to the child nodes in the level below it. Each child node can only have one parent node.
3. *Leaf nodes* are nodes are nodes on the bottom level of the tree, and do not have their own child nodes.
4. *Internal nodes* are nodes which are neither root nodes nor leaf nodes.

Now, we discuss decision trees.

9.1.3 Decision Trees

According to Prokhorenkova et al (2018)[22], we have the following definition of a decision tree.

Definition 9.8 (**Decision Tree**). A decision tree h can be written as

$$h(\boldsymbol{x}) = \sum_{j=1}^{J} b_j \mathbf{1}_{\{\boldsymbol{x} \in R_j\}} \tag{10}$$

where R_j are disjoint regions corresponding to the leaves of the tree.

Let us interpret the Equation (10). $\boldsymbol{x}$ is a vector of m features. We can write $\boldsymbol{x}$ as $\boldsymbol{x} = (x^1, x^2, ..., x^m)$ where the superscript on x is a way of denoting the k^{th} feature where $k = 1, 2, .., m$. The tree h has J total leaves where $j \leq J$ and $j \in \mathbb{N}$. b_j is the value of the j^{th} leaf. The indicator function $\mathbf{1}_{\{\boldsymbol{x} \in R_j\}}$ tells us which region R_j, the vector $\boldsymbol{x}$ is "sorted into" in tree h.

The above definition will be clearer with the following simple example.

Example 9.9 (**Simple Decision Tree**).

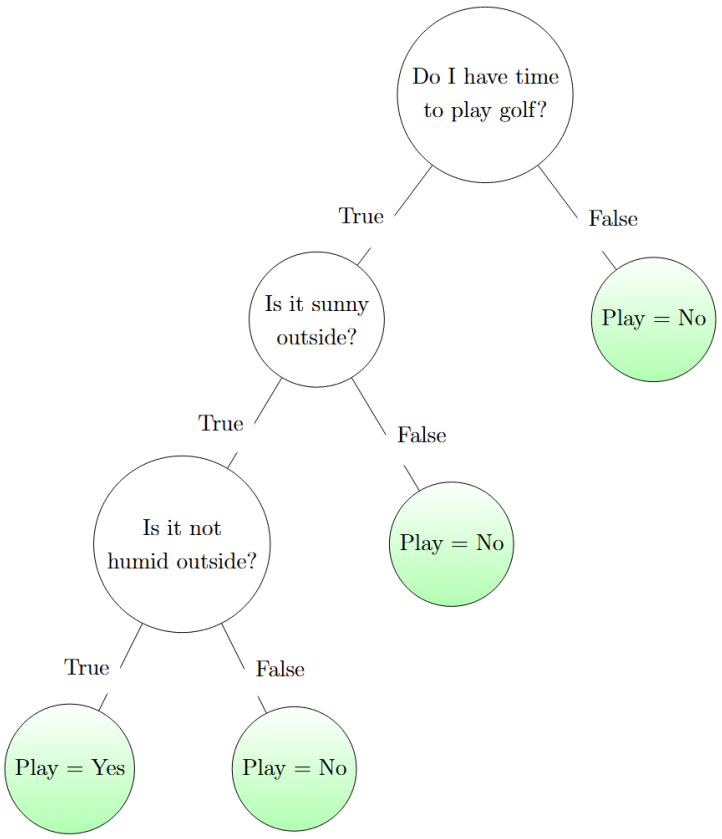


Figure 49: My simple decision tree

We will be working with a classification decision tree. The green nodes are leaf nodes. In this simple example, $\boldsymbol{x}$ has three features. These are "Do I have enough time to play golf?", "Is is sunny outside?" and "Is it not humid outside?". Set the label of these variables to x^1, x^2, x^3 respectively. For convenience, let us use one hot encoding. Let True = 1, and False = 0.

Let's assume that on this particular day, we have time to play golf, it's sunny outside, and it's humid outside. We can encode this as $\boldsymbol{x} = (x^1 = \text{True}, x^2 = \text{True}, x^3 = \text{False}) = (x^1 = 1, x^2 = 1, x^3 = 0)$. We can also encode b_j in this our classification tree. Let's read the leaves from left to right. Let $b_j = 1$ when "Play = Yes", and $b_j = 0$ when "Play = No". So if B is the set $\{b_1, b_2, ..., b_j, ..., b_J\}$ where J is the total number of leaves, then our particular example has $B = \{b_1 = 1, b_2 = 0, b_3 = 0, b_4 = 0\}$.

Thus, we can reinterpret the decision tree as follows.

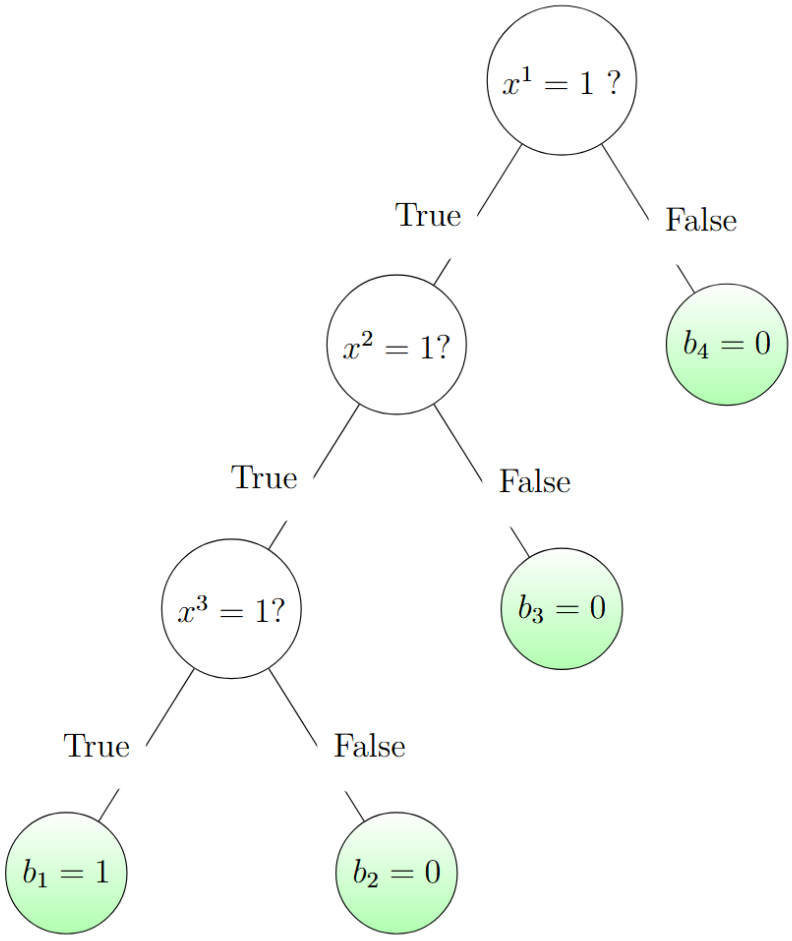


Figure 50: Reinterpreted simple decision tree

Let us find $h(\boldsymbol{x})$ for our tree. When working with decision trees, we start from the root node and "walk down" the decision tree until we reach a leaf node. By convention, we like the left path from a parent node to denote "True" and the right path to denote "False". Remember that $\boldsymbol{x} = (x^1 = 1, x^2 = 1, x^3 = 0)$.

1. Is $x^1 = 1$? Since $x^1 = 1$ is true, we take the path to the left child node.
2. Is $x^2 = 1$? Since this is true, we take the path to the left child node.
3. Is $x^3 = 1$? Since this node is false, we take the path to the right child node.
4. We arrive at $b_2 = 0$

Thus, we can conclude that

$$h(\boldsymbol{x}) = \sum_{j=1}^{J} b_j \mathbf{1}_{\{\boldsymbol{x} \in R_j\}} \tag{11}$$

$$= b_1 \mathbf{1}_{\{\boldsymbol{x} \in R_1\}} + b_2 \mathbf{1}_{\{\boldsymbol{x} \in R_2\}} + b_3 \mathbf{1}_{\{\boldsymbol{x} \in R_3\}} + b_4 \mathbf{1}_{\{\boldsymbol{x} \in R_4\}} \tag{12}$$

$$= 1 \cdot 0 + 0 \cdot 1 + 0 \cdot 0 + 0 \cdot 0 \tag{13}$$

$$= 0 \tag{14}$$

when $\boldsymbol{x} = (x^1 = 1, x^2 = 1, x^3 = 0)$

Now that we know how a decision tree works, we can discuss the Classification and Regression Tree (CART) algorithm.

9.2 Classification and Regression Trees (CART) Algorithm

CART is a powerful algorithm used to grow decision trees for both classification and regression. CART is directly referenced in the papers [4] and [5]. CART as a concept is also used in [15] and [22].

CART is best explained with an example.

9.2.1 CART for Classification

Example 9.10 (CART Simple Example).

For this example, we will show CART for classification. Let's assume that we have one continuously independent predictor variable X, and one categorically dependent variable Y.

Let us say X is some generic continuous variable (the amount of some hypothetical gene) that predicts a person's eye color. Let us also say the potential categories of Y are blue,

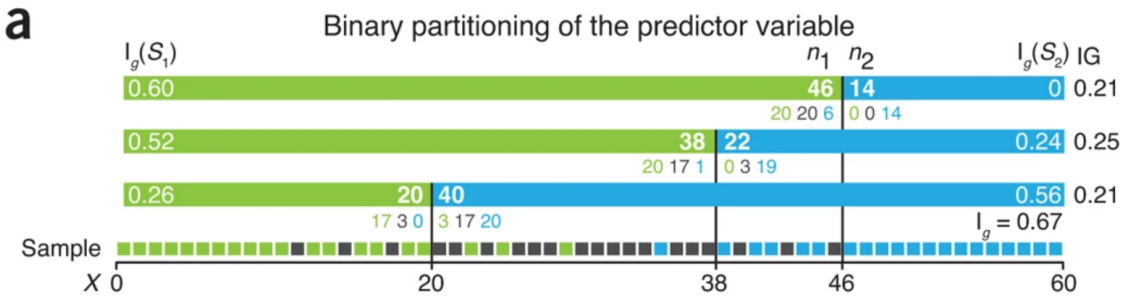


Figure 51: CART Diagram

gray, and green. Using a sample size of $n = 60$, let's build a decision tree classifier that predicts the color Y depending on the amount of X.

In Figure (51), for convenience, we have set the values of X to be between 0 and 60 and our sample $x_1, x_2, ...x_n$ is uniformly distributed. We want to pick the optimal $x \in X$ such that x will give use the most information in deciding whether a person has green eyes or blue eyes. In this example, we see that low values of X correspond to green eyes, while high values of X correspond to blue eyes.

To to select the optimal x, we want to maximize the information gain IG.

Definition 9.11 (**Information Gain**).

$$IG(S_1, S_2) = I(S) - \frac{n_1}{n} I(S_1) - \frac{n_2}{n} I(S_2) \tag{15}$$

x is selected so that x is the midpoint be two adjacent observed values of X. S is the set of all samples $\{x \in X\}$, S_1 is the set of samples $\{X < x\}$, and S_2 is the set of samples $\{X > x\}$. n_1 is the number of samples in set S_1, n_2 is the number of samples in set S_2, and n is the number of samples in S [17].

When maximizing $IG(S_1, S_2)$, we are attempting to split S_1 and S_2 so that S_1 contains mostly X's which predict green eyes, and S_2 contains mostly X's which predict blue eyes.

To maximize $IG(S_1, S_2)$, we need to specific what our impurity function $I(S)$ is. $I(S)$ measures the amount of class mixing in a subset.

Definition 9.12 (**Impurity Function - Gini Index**). Let $I_g(S) = \sum_{i=1}^{n} p_i(1 - p_i)$ where p_i is the fraction of data points of class i in a subset S [17].

We may be more familiar with the Gini Index written in the alternative form $I_g(S) = 1 - \sum_{i=1}^{n} (p_i)^2$. We can easily confirm that $\sum_{i=1}^{n} p_i(1 - p_i) = 1 - \sum_{i=1}^{n} (p_i)^2$.

$$\sum_{i=1}^{n} p_i(1-p_i) \tag{16}$$

$$= \sum_{i=1}^{n} (p_i - p_i^2) \tag{17}$$

$$= \sum_{i=1}^{n} p_i - \sum_{i=1}^{n} p_i^2 \tag{18}$$

$$= 1 - \sum_{i=1}^{n} p_i^2 \quad \text{We know that probabilities } \sum_{i=1}^{n} p_i \text{ must sum to 1} \tag{19}$$

The Gini index is a fairly standard and simple metric used for classification trees. The Gini index $I_g(S)$ has a minimum value of 0 when $p_i = 1$ for some $i = 1, 2, ..., n$. This means that every element in S is part of class i.

In this example, the information gain using Gini index IG_g is calculated for all possible splits in X using a computer. The maximum information gain IG_g is found at $x = 38$ [17]. We will confirm the correctness of their IG_g number for illustrative purposes. Remember that we have three classes i (green, gray, and blue). Let us set $i = 1$ to be green, $i = 2$ to be gray, and $i = 3$ to blue.

Calculating IG_g

In S_1, we have 20 green samples, 17 gray samples, and 1 blue sample. In S_2, we have 0 green samples, 3 gray samples, and 19 blue samples.

First, we will find $I_g(S_1)$, $I_g(S_2)$ and $I_g(S)$.

$$I_g(S_1) = 1 - \sum_{i=1}^{3} (p_i)^2 \quad \text{for all classes } i \tag{20}$$

$$= 1 - p_1^2 - p_2^2 - p_3^2 \tag{21}$$

$$= 1 - \left(\frac{20}{38}\right)^2 - \left(\frac{17}{38}\right)^2 - \left(\frac{1}{38}\right)^2 \tag{22}$$

$$\approx 0.522 \tag{23}$$

$$I_g(S_2) = 1 - \sum_{i=1}^{3} (p_i)^2 \quad \text{for all classes } i \tag{24}$$

$$= 1 - p_1^2 - p_2^2 - p_3^2 \tag{25}$$

$$= 1 - \left(\frac{0}{22}\right)^2 - \left(\frac{3}{22}\right)^2 - \left(\frac{19}{22}\right)^2 \tag{26}$$

$$\approx 0.2355 \tag{27}$$

$$I_g(S) = 1 - \sum_{i=1}^{3} (p_i)^2 \quad \text{for all classes } i \tag{28}$$

$$= 1 - p_1^2 - p_2^2 - p_3^2 \tag{29}$$

$$= 1 - \left(\frac{20}{60}\right)^2 - \left(\frac{20}{60}\right)^2 - \left(\frac{20}{60}\right)^2 \tag{30}$$

$$= \frac{2}{3} \tag{31}$$

Now, we can calculate $IG_g(S_1, S_2)$

$$IG_g(S_1, S_2) = I(S) - \frac{n_1}{n} I(S_1) - \frac{n_2}{n} I(S_2) \tag{32}$$

$$= \frac{2}{3} - \frac{38}{60}(0.522) - \frac{22}{60}(0.2355) \tag{33}$$

$$\approx 0.249 \tag{34}$$

Thus, we can confirm results in [17] are correct.

Once the first split is chosen, we split each of the two subsets created by the first split. We continue splitting this way until we reach a stopping condition. Example stopping conditions include reaching the max depth of the tree, maximum number of leaves, minimum number of samples per leaf, or a desired purity percentage in your leaves. By splitting our subset, we are naturally following the logic of decision tree, where the nodes of the tree correspond to subsets of the data and the tree's branches correspond to partitioning a variable above or below a certain value. Once we have reached our stopping condition for splitting, we will look at subsets we have created, and assign each subset the class $i \in \{1, 2, 3\}$ which appears most often in that subset [17].

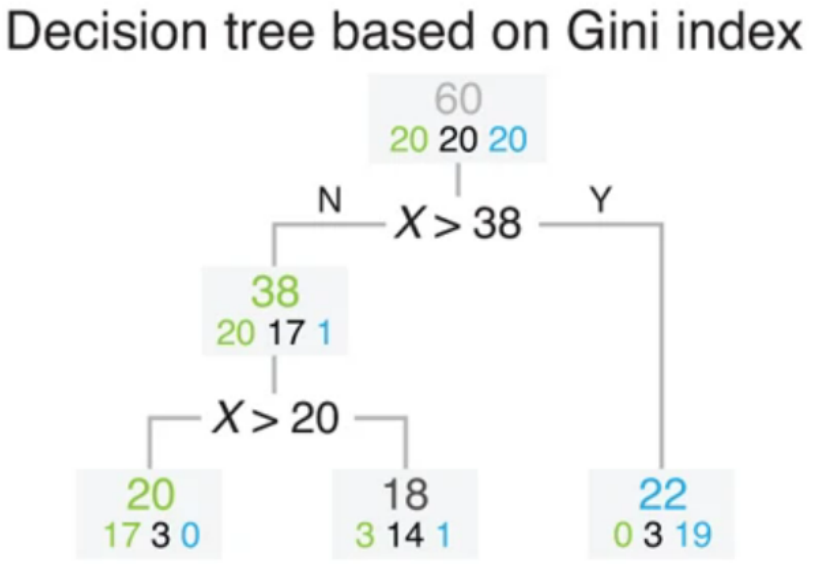


Figure 52: Finished CART tree

In Figure (52), we can see our finished decision tree generated by CART.

9.2.2 CART for Regression

CART for regression is analogous to CART for classification. Let's assume that both X and Y are continuous random variables. We want to use X to predict Y.

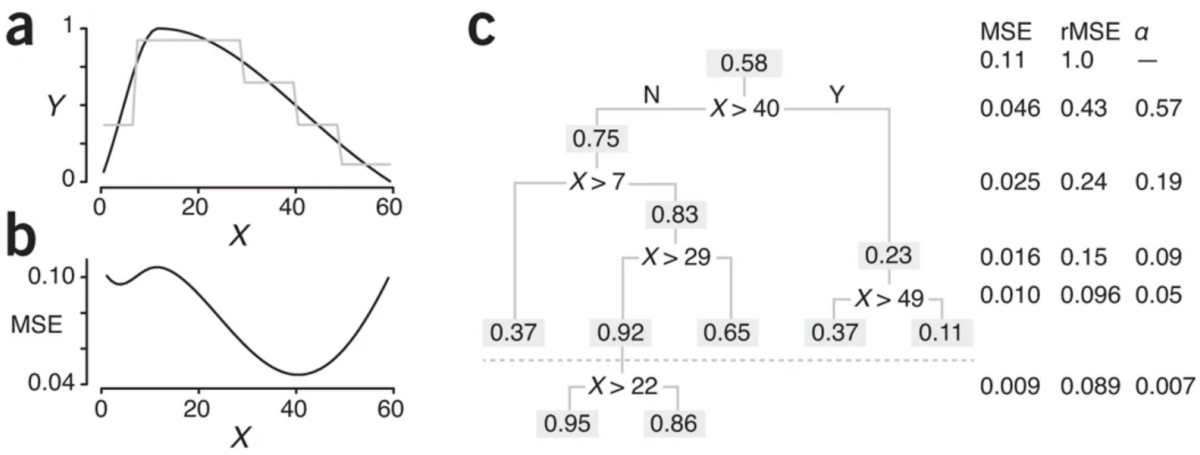


Figure 53: Diagrams for CART for regression

In Figure (53a), we see that our goal is to fit the black line, which is a real function $y = f(x)$, with an approximate gray line of step functions. We want to partition $S = \{x \in X\}$

so that over each subset $S_j \subset S$, we pick a $\bar{y}$ so that the mean square error

$$MSE = \sum_{i=1}^{n} \frac{(\bar{y} - y_i)^2}{n}$$

is minimized. Here, $\sum_{i=1}^{n}$ means that we are summing over all values $i = 1, ..., n$ in subset S_j. Also, y_i is the value of the real y in the interval S_j, $\bar{y}$ is the mean of y_i over the subset S_j, and n is the number of data points within our particular subset S_j[17].

Let's assume that we have partitioned S into S_1 and S_2, where S_1 is the set of samples $\{X < x\}$, and S_2 is the set of samples $\{X > x\}$. Also let n_1 be the number of samples in set S_1, n_2 be the number of samples in set S_2, and n be the number of samples in S. Then, in an analogous way to minimizing $IG(S_1, S_2)$ in the CART for classification example, we want to minimize:

$$MSE_T = \frac{n_1 MSE_1}{n} + \frac{n_2 MSE_2}{n} \tag{35}$$

where MSE_T stands for mean squared error (MSE) total, MSE_1 is the MSE over S_1, and MSE_2 is the MSE over S_2. We will greedily find the $x \in X$ to minimize the RHS of line 35. Just like in the case for classification, we want to iteratively split the newly created subsets S_1 and S_2 until we reach a stopping condition. We can see our finished regression tree in Figure 53c.

A useful stopping criterion used in Figure 53c is requiring that each split improves the relative error by at least α, which is a predetermined constant [17]. In Figure 53c, we can see that relative MSE (rMSE) is just the MSE but normalized so that the rMSE equals 1 before any splits happen. If $\alpha = 0.01$, then the split $X > 22$ in Figure 53c would not be accepted due to reducing rMSE by an insufficient amount.

For our paper, we will be using only CART to grow regression trees, as all our variables are numeric and not categorical.

9.3 Gradient Boosting for Decision Trees

Gradient boosting is an ensemble machine learning algorithm that involves adding up many weak learners, which are models with lower predictive power, (in this case decision trees) to form a strong predictive model. Gradient boosting for decision trees is directly referenced in the papers [5], [15], and [22].

Gradient boosting for decision trees is best explained with an example. We will use the example provided in [22]

Example 9.13 (Gradient Boosting for Decision Trees).

Let's assume that we observe a dataset of examples $\mathcal{D} = \{(\boldsymbol{x}_k, y_k)\}_{k=1,2,...,n}$ where $\boldsymbol{x}_k = (x_k^1, x_k^2, ..., x_k^m)$ is a row vector of m features and $y_k \in \mathbb{R}$ is the target which we wish to predict. If we are doing gradient boosting for classification between two states, then y_k is a binary variable with values either 0 or 1. If we are doing gradient boosting for regression, which is the focus of our paper, then y_k can take any values in $\mathbb{R}$ [22]. The intuition behind this set up is $\{\boldsymbol{x}_k\}_{k=1,2,...,n}$ will be our independent predictors, and $\{y_k\}_{k=1,2,...,n}$ will be our dependent variable when setting up a prediction model.

Let us elaborate this point further. We can write $\mathcal{D} = \{(\boldsymbol{x}_k, y_k)\}_{k=1,2,...,n}$ more explicitly as

$$\mathcal{D} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), ..., (\boldsymbol{x}_n, y_n)\}$$

Notice that for $\boldsymbol{x}_k$ where $k \in \{1, 2, .., n\}$, that in $\boldsymbol{x}_k = (x_k^1, x_k^2, ..., x_k^m)$, the superscript on x_k denotes the l^{th} feature for $l = 1, 2, ..., m$. Also, notice that $\boldsymbol{x}_k$ is in the k^{th} example $(\boldsymbol{x}_k, y_k)$. [22].

Let's assume that each example $(\boldsymbol{x}_k, y_k)$ is independent and identically distributed according to some unknown distribution $P(\cdot, \cdot)$. Now, we want to find a function $F : \mathbb{R}^m \to \mathbb{R}$ (an ensemble of decision trees) which minimizes the expected loss $\mathcal{L} := E[L(y, F(\boldsymbol{x}))]$. Here $L(\cdot, \cdot)$ is a smooth loss function of our choice (usually squared error) and $(\boldsymbol{x}, y)$ is a test example sampled from $P(\cdot, \cdot)$ independently of the training set $\mathcal{D}$ [22].

Remember that our prior notation for decision trees, $h(\boldsymbol{x}) = \sum_{j=1}^{J} b_j \mathbf{1}_{\{\boldsymbol{x} \in R_j\}}$. From now on, let $h(\cdot)$ represent an generic decision tree.

The gradient boosting procedure iteratively builds a sequence of approximations $F^t : \mathbb{R}^m \to \mathbb{R}$ where $t = 0, 1, 2...$ in a greedy fashion. Specifically, $F^t = F^{t-1} + \alpha h^t$ where $\alpha \in (0, 1]$ is a constant and $h^t : \mathbb{R}^m \to \mathbb{R}$ (called our *base predictor* or *weak learner*) is a tree. Intuitively, we construct F^t by adding a series of trees h^t together. If we want to be more explicit, we can write F^t as:

$$F^t = \alpha(h^0 + h^1 + h^2 + ...h^t)$$

where h^t belongs to a family of functions H. In our case, we will assume H is the family of all trees with domain $\mathbb{R}^m$ that produces a real-valued output.

We want to choose h^t in order to minimize the expected loss [22]:

$$h^t = \underset{h \in H}{\arg\min} \left(\mathcal{L}(F^{t-1} + h)\right) = \underset{h \in H}{\arg\min} \left(E\left[L\left(y, F^{t-1}(\boldsymbol{x}) + h(\boldsymbol{x})\right)\right]\right) \tag{36}$$

Let's elaborate further on Equation (36). Remember that $(\boldsymbol{x}, y)$ are independently sampled from $P(\cdot, \cdot)$ and we are taking the expectation $E\left[L\left(y, F^{t-1}(\boldsymbol{x}) + h(\boldsymbol{x})\right)\right]$ over all possible $(\boldsymbol{x}, y)$ pairs. On the right most side of Equation (36), intuitively, we want to greedily find the next best tree h so that $E\left[L\left(y, F^{t-1}(\boldsymbol{x}) + h(\boldsymbol{x})\right)\right]$ is minimized.

Let's provide a simple example to clarify the RHS of Equation (36). Assume that our loss function $L(\cdot, \cdot)$ is just squared error. That is, for some y and predicted $\hat{y}$, $L(y, \hat{y}) = (y - \hat{y})^2$. Then, $\mathcal{L}(F) = E[L(y, F(\boldsymbol{x}))] = E[(y - F(\boldsymbol{x}))^2]$

The method of actually finding the next best tree h^t to satisfy

$$h^t = \arg\min_{h \in H} \left(E\left[L\left(y, F^{t-1}(\boldsymbol{x}) + h(\boldsymbol{x}) \right) \right] \right)$$

is usually done numerically by a computer. The minimization problem is usually approached using the *Newton method* using a second-order approximation of $\mathcal{L}(F^{t-1} + h)$ at F^{t-1} or by taking a *negative gradient step* [22].

Let's give some intuition behind negative gradient steps.

9.3.1 Negative Gradient Steps

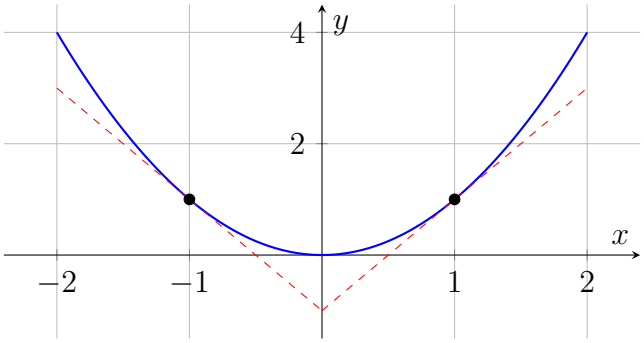


Figure 54: Intuition for negative gradient steps

In the Figure (54) above, imagine that our blue line is our function L where are graph is in the 2D plane. At $x = -1$, imagine the gradient $\nabla L = -c$ where $-c$ is a negative constant. At $x = 1$, imagine the gradient $\nabla L = c$ where c is a positive constant. Now, lets define our negative gradient function $NG := -\alpha \nabla L$ where $\alpha \in (0, 1]$ is a constant which represents our *learning rate*.

So, at $x = -1$, $NG = -\alpha \nabla L = (-\alpha)(-c) = \alpha c$. Respectively, at $x = 1, NG = -\alpha c$. The interpretation is that, at $x = -1$, the negative gradient function is telling us to move αc amount on the x-axis to the *right* to find our desired absolute (or sometimes unfortunately local) minimum. At $x = 1$, the negative gradient is telling us to move αc to the *left* to find the desired absolute (or local) minimum.

In [22], the gradient step h^t is chosen in such a way that $h^t(\boldsymbol{x})$ approximates $-g^t(\boldsymbol{x}, y)$ where

$$g^t(\boldsymbol{x}, y) := \left. \frac{\partial L(y, s)}{\partial s} \right|_{s = F^{t-1}(\boldsymbol{x})}$$

In plain words, $g^t(\boldsymbol{x}, y)$ is the gradient of L, taken at the point $s = F^{t-1}(\boldsymbol{x})$. Thus, $-g^t(\boldsymbol{x}, y)$ is the negative gradient of L.

Usually, a least squares approximation is used [22]:

$$h^t = \arg\min_{h \in H} E\left[\left(-g^t(\boldsymbol{x}, y) - h(\boldsymbol{x})\right)^2\right] \tag{37}$$

9.4 Bagging (Bootstrap + Aggregate)

The technique of bagging, which was first published in 1996 by Breiman [3] is a shorthand that stands for "bootstrap aggregating." The technique of bagging is used in *Random Forest*[4].

9.4.1 Bootstrap

First we need to discuss a bootstrap technique. We will cite Rubin (1981) [23] to explain the standard bootstrap technique.

Suppose that we have sample size of n independent and identically distributed realizations of a random variable X. Let us denote these realizations with $x_1, x_2, ..., x_n$. Let's assume we want to estimate a parameter ϕ of the distribution of X. Our estimate of ϕ will be denoted with $\hat{\phi}$. The bootstrap distribution of $\hat{\phi}$ is generated by taking repeated *bootstrap replications* from $x_1, x_2, ..., x_n$. One *bootstrap replication* from $x_1, x_2, ..., x_n$ is a simple random sample of size n from $x_1, x_2, ..., x_n$ *with replacement*. One bootstrap replication of $\hat{\phi}$ is the value of $\hat{\phi}$ calculated on one bootstrapped sample. In theory, after taking all possible bootstrapped samples, $x_1, ..., x_n$, which would lead to finding all the probabilities of all possible $\hat{\phi}$'s, we can then create a bootstrapped distribution of $\hat{\phi}$ [23].

9.4.2 Aggregate Decisions

We will provide a simple example for aggregate decisions for classification.

Intuitively, we can think of aggregate decisions for classification as "first past the post voting". Imagine that we have three candidates "Alice", "Brandon", and "Chris" from three different political parties. If Alice wins 40% of votes, Brandon 35%, and Chris 25%, then Alice would win the election and Brandon and Chris would get nothing.

We can now formalize our reasoning. Let's imagine that we have m number of classes denoted by i. Let i be the i^{th} class where $i = 1, 2, ..., m$. Let's say we have a sample $x_1, x_2, ..., x_n$ of n observations. Each of these observations x_k, where $k = 1, 2, ..., n$, is a vote for specific class i. Let $N(i) = |\{x_k = i, \text{ for } k = 1, 2, ..., n\}|$ where $|\cdot|$ is the cardinality of a set. Then, we would calculate $N(i)$ for all classes $i = 1, 2, ..., m$. Then our aggregate

decision function is

$$F_A(x_1, x_2, ..., x_n) := \arg\max_i (N(i))$$

That is to say, F_A (where A stands for aggregate) just picks the class with the most occurrences in the sample $x_1, x_2, ..., x_n$.

We can now describe the concept of bagging.

9.4.3 How Bagging Works

We will reference Breiman (1996) [3] to explain bagging.

Let's assume that we have a learning set $\mathcal{L}$ that consists of data $\{(y_n, \boldsymbol{x}_n), n = 1, 2, ..., N\}$ where the y_n's are either a class label ($y_n = 1, 2, 3, ...$) or a numerical value (in the case of our project, $y_n \in \mathbb{R}$). Let's also assume that $\boldsymbol{x}_n$ is a row vector with m features. So, $\boldsymbol{x}_n = (x_n^1, x_n^2, ..., x_n^m)$. Notice that the superscript on x_n denotes the l^{th} feature for $l = 1, 2, ..., m$. [3]

Assume that we have a procedure for using this learning set to form a predictor $\varphi(\boldsymbol{x}, \mathcal{L})$. Here $\boldsymbol{x}$ is a new input, a single row vector which may or may not come from the same distribution $P(\cdot, \cdot)$ which is used to generate $\mathcal{L}$. We want to use our predictor $\varphi(\boldsymbol{x}, \mathcal{L})$ to predict y.

Now suppose that we have a given sequence of learning sets $\{\mathcal{L}_k\}$ each consisting of N independent observations from the same underlying distribution as $\mathcal{L}$ — in our case this is $P(\cdot, \cdot)$ [3]. If we want to be explicit, we can write $\{\mathcal{L}_k\}$ as $\{\mathcal{L}_k\}_{k=1,2,...}$.

Our objective is to use the $\{\mathcal{L}_k\}$ to get a better predictor than the single learning set predictor $\varphi(\boldsymbol{x}, \mathcal{L})$. Our restriction is that all the information we have is the sequence of predictors $\{\varphi(\boldsymbol{x}, \mathcal{L}_k)\}$

If y is numerical, simple and clear procedure is to replace $\varphi(\boldsymbol{x}, \mathcal{L})$ with the average of $\varphi(\boldsymbol{x}, \mathcal{L}_k)$ over all possible learning sets (which in the theoretical case may be infinite). In other words $\varphi_A(\boldsymbol{x}) = E_{\mathcal{L}}[\varphi(\boldsymbol{x}, \mathcal{L})]$ where $E_{\mathcal{L}}$ denotes the expectation over $\mathcal{L}$, and the subscript A in φ_A denotes aggregation [3].

If y is categorical, then $\varphi(\boldsymbol{x}, \mathcal{L})$ should predict a class $j \in \{1, 2, ..., J\}$. One way to predict a class is by voting. We will aggregate $\varphi(\boldsymbol{x}, \mathcal{L}_k)$ by the "first past the post" voting system. Let $N_j = \text{num}\{k; \varphi(\boldsymbol{x}, \mathcal{L}_k) = j\}$ [3]. Let's interpret N_j. For the sake of convenience and real world application, let us assume that we have a finite K total number of learning sets. Let's index the learning sets by $s = 1, 2, ..., K$. Each $\varphi(\boldsymbol{x}, \mathcal{L}_s)$ for $s = 1, 2, ..., K$ evaluates to a certain class $j \in \{1, 2, ..., J\}$. So $\text{num}\{s; \varphi(\boldsymbol{x}, \mathcal{L}_s) = j\}$ can be interpreted as the number of learning sets $\mathcal{L}_s$ for $s = 1, 2, ..., K$ which evaluate to class j.

Next, we want to take $\varphi_A(\boldsymbol{x}) = \arg\max_j(N_j)$, that is, we want to choose the class $j \in \{1, 2, ..., J\}$ which is chosen most often among learning sets $\{\mathcal{L}_s\}_{s=1,2,...,K}$ [3].

Unfortunately, in the real world, we usually only have a single learning set $\mathcal{L}$ sampled from underlying distribution $P(\cdot, \cdot)$, and we do not have the replicate sets $\{\mathcal{L}_s\}_{s=1,2,...,K}$. To imitate the process of sampling $\{\mathcal{L}_s\}_{s=1,2,...,K}$ from $P(\cdot, \cdot)$, we can use the bootstrap technique [3]. We can take repeated bootstrap samples, which we denote as $\{\mathcal{L}^{(B)}\}$, from $\mathcal{L}$ and form $\{\varphi(\boldsymbol{x}, \mathcal{L}^{(B)})\}$.

Let φ_B stand for our predictor formed under *bagging*. φ_B is analogous to φ_A, where φ_A is the predictor formed under aggregation with the replicate sets $\{\mathcal{L}_s\}_{s=1,2,...,K}$ sampled from $P(\cdot, \cdot)$. If y is any numerical, take φ_B as [3]:

$$\varphi_B(\boldsymbol{x}) = \text{avg}_B(\boldsymbol{x}, \mathcal{L}^{(B)})$$

If y is a class label, let the $\{\varphi(\boldsymbol{x}, \mathcal{L}^{(B)})\}$ vote to form $\varphi_B(\boldsymbol{x})$. Since we have used the process of both *bootstrapping* and *aggregating*, we call our process *bagging*.

Breiman (1996) [3] claims that bagging can produce superior results than just using a single learning set $\mathcal{L}$.

9.4.4 Why Bagging can Improve Predictions

Breiman (1996) [3] defines the word "stability" in the context of bagging. A procedure is "stable" if small changes in a replicate of $\mathcal{L}$ produce small changes in φ. A procedure is "unstable" if small changes in a replicate of $\mathcal{L}$ produce large changes in φ. Breiman (1996) [3] claims that improvement in prediction will occur when using bagging on "unstable" procedures where a small change in $\mathcal{L}$ can result in large changes in φ [3]. He demonstrates that empirically in his paper through simulation.

9.5 Random Forest

Random Forest is a technique created by Breiman (2001)[4]. This technique makes use of both the bagging technique and the CART algorithm.

9.5.1 Random Forest Algorithm

Significant improvements in classification accuracy have resulted from growing an ensemble of trees and letting them vote for the most popular class. In order to grow these ensembles, often random vectors are generated that govern the growth of each tree in the ensemble [4].

To generate the k^{th} tree, a random vector Θ_k is generated. This random vector Θ_k is generated from some distribution $P(\cdot)$ and is independent of past random vectors $\Theta_1, \Theta_2, ..., \Theta_{k-1}$. Let's assume we have some training set T, and we are currently working with the random vector Θ_k. Then using both T and Θ_k, we can create a classifier $h(\boldsymbol{x}, \Theta_k)$ to predict some y,

where $\boldsymbol{x}$ is a vector of features. After a large number of trees is generated, they vote for the most popular class. Breiman (2001) calls this procedure, *Random Forests* [4].

Definition 9.14 (**Random Forests**)**.** A *Random Forest* is a classifier consisting of a collection of tree-structured classifiers $\{h(\boldsymbol{x}, \Theta_k), k = 1, 2, ...\}$ where the $\{\Theta_k\}$ are independently identically distributed random vectors and each tree casts a unit vote for the most popular class at input $\boldsymbol{x}$[4].

9.5.2 Bagging in Random Forest

Breiman (2001) [4] describes how he uses *bagging* in Random Forest. Assume we have an original training set T. Then, let's assume we can bootstrap some new training sets $\{T_s^{(B)}\}_{s=1,2,...,K}$ where the superscript (B) stands for bootstrapped. For each training set, $T_s^{(B)}, s = 1, 2, ..., K$, we can construct tree-based classifiers $h(\boldsymbol{x}, T_s^{(B)}), s = 1, 2, ..., K$ and let these vote to form the bagged predictor.

The simplest Random Forest using both *bagging* and *random feature selection* described by Breiman (2001) [4] is as follows: "The simplest Random Forest with random features is formed by selecting at random, at each node, a small group of input variables to split on. Grow the tree using CART methodology to maximum size and do not prune" [4].

Let's interpret that statement. Let's assume that we are working with tree $h(\boldsymbol{x}, T_s^{(B)})$ where s is a specific number in $\{1, 2, ..., K\}$. When constructing our tree h using CART methodology, at *each possible node* in the tree, we select a random subset $S \subset T_s^{(B)}$ of possible features to split that node on. Remember, at each specific node, we can only split that node with 1 chosen feature. So, at each node in the tree, we look through the newly chosen random subset S, and choose the best feature $f \in S$ that will split the tree the best according to some CART criteria (like information gain calculated using Gini index).

9.5.3 Properties of Random Forest for Regression

For our paper, we will exclusively be using Random Forests for a regression setting.

As such, we should mention some properties of Random Forest so we have some idea about the predictive power of our results. Not all these properties will be exclusive to Random Forest for the regression setting.

(i) Random Forests converge. This result explains why Random Forests do not overfit as more trees are added, but produce a limiting value of the generalization error [4].

(ii) It can be shown that an upper bound for the generalization error can be found. According to Breiman (2001) [4]:

Theorem 2.3. *An upper bound for the generalization error is given by*

$$PE^* \leq \bar{\rho}(1 - s^2)/s^2.$$

Here, PE^* is a generalization error. The exact definition of $\frac{\bar{\rho}(1-s^2)}{s^2}$ is quite involved and would require a lengthy explanation. Note that $\bar{\rho} \in [-1, 1]$ and $s \in [-1, 1]$. In practice however, $\bar{\rho}$ should not be below 0, otherwise the interpretation of *Theorem 2.3* would not make sense. To fully understand *Theorem 2.3* fully, refer to pages 7, 8, and 9 of Breiman (2001)[4]. The important thing to note here is that the generalization error, also known as *out-of-sample error*, is bounded. This means that the error of our model when we are looking at new data (like testing data), is bounded. Notice that if s is closer to 1, as opposed to being closer to 0, our upper bound on the generalization error decreases. Heuristically, we can think of s as the strength of individual trees in the forest.

(iii) For Random Forests for the regression setting, the generalization error of the forest is less than the generalization error of a single tree. According to Breiman(2001)[4]:

Theorem 11.2. *Assume that for all* Θ, $EY = E_{\mathbf{X}}h(\mathbf{X}, \Theta)$. *Then*

$$PE^*(forest) \leq \bar{\rho}PE^*(tree)$$

where $\bar{\rho}$ *is the weighted correlation between the residuals* $Y - h(\mathbf{X}, \Theta)$ *and* $Y - h(X, \Theta')$ *where* Θ, Θ' *are independent.*

From Breiman(2001)[4], we know that $PE^*(forest)$, $PE^*(tree) \geq 0$. The interpretation of thereom 11.2 is, when $\bar{\rho} \in [0, 1]$, the generalization error for a Random Forest is bounded by the generalization error for a single tree. This makes an intuitive sense, as sampling more trees should give us a better predictors than sampling only a single tree. Having more trees would mean our model incorporates more information.

9.6 Extreme Gradient Boosting (XGBoost)

XGBoost is a technique created by Chen and Guestrin (2016)[5]. XGBoost makes use of both the CART algorithm and gradient boosting for decision trees.

Chen and Guestrin (2016) describe the many practical benefits of XGBoost. XGBoost's benefits include

1. a novel tree learning algorithm for handling sparse data

2. a theoretically justified weighted quantile sketch procedure to handle instance weights in approximate tree learning

3. parallel and distributed computing

4. and out-of-core computation that enables data scientists to produces hundred millions of examples on a desktop [5].

For the sake of focused discussion, we will only discuss the tree boosting method employed by XGBoost as described by Chen and Guestrin(2016).

9.6.1 XGBoost Algorithm - Regularized Learning Objective

Let's assume that we have a data set $\mathcal{D}$ with n examples, and m features. Let $\mathcal{D} = \{(\boldsymbol{x_i}, y_i)\}_{i=1,2,...,n}$ where $\boldsymbol{x}_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}$. Let's also assume ϕ is a tree learning ensemble model that uses K additive functions to predict the output. Specifically:

$$\hat{y}_i = \phi(\boldsymbol{x}_i) = \sum_{k=1}^{K} f_k(\boldsymbol{x}_i), \quad f_k \in \mathcal{F} \tag{38}$$

In Equation (38), $\mathcal{F} = \{f(\boldsymbol{x}) = w_{q(\boldsymbol{x})}\}$ where $q : \mathbb{R}^m \to T$ and $w \in \mathbb{R}^T$ [5].

Let us explain Equation (38) more clearly. First, we need to clarify the notation $q : \mathbb{R}^m \to T$. T represents the number of leaves in a generic tree f. The notation for T from the Equation (38) comes from set theory. If $|T| = 5$, then we should interpret $q : \mathbb{R}^m \to T$ as $q : \mathbb{R}^m \to \{1, 2, 3, 4, 5\}$. So, we can interpret q as a function which takes in a row vector $\boldsymbol{x} = (x^1, x^2, ..., x^m)$ where the superscript on x denotes the l^{th} feature where $l = 1, 2, ..., m$. Without loss of generality, we can label our leaves on a generic decision tree f from left to right as $1, 2, ..., |T|$. So, q is a function which takes in a data point $\boldsymbol{x}$ and then "sorts" that data point into a specific leaf. Thus, we can interpret q as the "tree structure".

We can interpret w as a vector containing the weights corresponding to each leaf. In math notation $w = (w_1, w_2, ..., w_{|T|})$ where the subscript on w denotes the j^{th} weight for $j = 1, 2, ..., |T|$. Notice, that j can also be used to denote specific leaves. Note that the weights $(w_1, w_2, ..., w_{|T|})$ *do not* have to add up to 1. So we can finally now interpret $\mathcal{F} = \{f(\boldsymbol{x}) = w_{q(\boldsymbol{x})}\}$. $\mathcal{F}$ is the class of functions, specifically decision trees f, which follow the structure that an input $\boldsymbol{x}$ of dimension $\mathbb{R}^m$ will return a weight $w_{q(\boldsymbol{x})}$ where the dimension of w is $\mathbb{R}^T$. In our case, the function $q : \mathbb{R}^m \to T$ determines how an input $\boldsymbol{x}$ will map to a specific leaf $j = 1, 2, ..., |T|$. Thus, in a roundabout way, q determines $w_{q(\boldsymbol{x})}$.

Chen and Guestrin (2016) state $\mathcal{F}$ is "also known as CART" [5].

So, when looking at Equation (38), we see that f_k is a generic decision tree which belongs to the class of decision trees that look like $\mathcal{F}$. That is, f_k is a decision tree with some

independent tree structure q and leaf weights w. So our interpretation of Equation (38) finally is that we have K many decision trees f_k where $k = 1, 2, ..., K$. In each decision tree, we put in an input $\boldsymbol{x}_i$. Each decision tree $f_k(\boldsymbol{x}_i)$ will return some scalar weight value $w_{q(\boldsymbol{x}_i)}$. We then add those weights $w_{q(\boldsymbol{x}_i)}$ together for each decision tree f_k, and that is how we find $\hat{y}_i$, our predicted output for $\boldsymbol{x}_i$.

This will be much clearer with a simple example.

Example 9.15 (Simple application example of decision trees in XGBoost).

We will use the example provided by Chen and Guestrin (2016) [5].

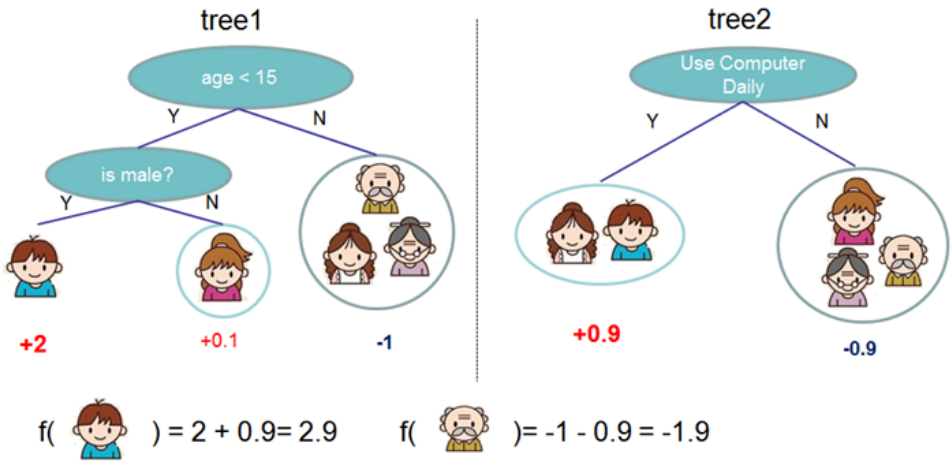


Figure 55: Simple XGBoost example

In Figure 55, we can see we have two simple decision trees. Let's assume that we want to find the $\phi(\text{“young boy”})$. From “tree 1”, the value of “young boy” is 2, and from “tree 2”, the value of “young boy” is 0.9. Thus,

$$\phi(\text{“young boy”}) = \sum_{k=1}^{2} f_k(\text{“young boy”}) = 2 + 0.9 = 2.9$$

Similarly, we can see $\phi(\text{“elderly man”}) = -1.9$

Let us get back to describing XGBoost. To learn the set of functions $\{f_k\}_{k=1,2,...,K}$ used by XGBoost, we want to minimize the following *regularized* objective [5]:

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \tag{39}$$

$$\text{where } \Omega(f) = \gamma|T| + \frac{1}{2}\lambda\|w\|^2 \tag{40}$$

Let me elaborate on Equations (39) and (40). Here, l is a differentiable convex loss function of our choice (like squared error). $\hat{y}_i$ is the prediction when using $\hat{y}_i = \phi(\boldsymbol{x}_i)$ from Equation (38). y_i is our target prediction [5]. $\|\cdot\|$ is the euclidean norm so $\|w\| = \sqrt{w_1^2 + w_2^2 + ... + w_{|T|}^2}$. Thus, $\|w\|^2 = w_1^2 + w_2^2 + ... + w_{|T|}^2$. Ω is some function than penalizes the complexity of f_k for $k = 1, 2, ..., K$. γ and λ are two constants such that $\gamma, \lambda > 0$. Intuitively, we can think of $\gamma|T|$ and $\frac{1}{2}\lambda\|w\|^2$ as a penalties for having too many leaves.

Intuitively, when $\mathcal{L}(\phi)$ is minimized, we will have found some trees $f_k, k = 1, 2, ..., K$ which predict the data well, but are also not too complex.

9.6.2 XGBoost Algorithm - Gradient Tree Boosting

The regularization objective in Equations (39) and (40) cannot be optimized using traditional optimization methods. Instead, the model must be trained in an additive manner [5].

Formally, let $\hat{y}_i^{(t)}$ be the prediction of the i^{th} instance at the t^{th} iteration [5]. Remember, our original dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1,2,...,n}$. We want to minimize the following objective:

$$\mathcal{L}^{(t)} = \left[\sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{(t-1)} + f_t(\boldsymbol{x}_i)\right)\right] + \Omega(f_t) \tag{41}$$

by greedily adding the f_t that improves $\mathcal{L}^{(t)}$ the most.

Using second order approximation (also known as *Taylor expansion*), we can approximate $\mathcal{L}^{(t)}$ as [5]:

$$\mathcal{L}^{(t)} \approx \left(\sum_{i=1}^{n} \left[l\left(y_i, \hat{y}_i^{(t-1)}\right) + g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i)\right]\right) + \Omega(f_t) \tag{42}$$

where

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l\left(y_i, \hat{y}_i^{(t-1)}\right) \quad \text{and} \quad h_i = \partial^2_{\hat{y}_i^{(t-1)}} l\left(y_i, \hat{y}_i^{(t-1)}\right)$$

Notice in Equation (42), that at time t, we already know the value of $l(y_i, \hat{y}_i^{(t-1)})$. Thus, since $l(y_i, \hat{y}_i^{(t-1)})$ is a constant at time t, we can remove it to because it is not useful to find f_t.

So, let us denote our simplified objective function $\widetilde{\mathcal{L}}^{(t)}$. Then:

$$\widetilde{\mathcal{L}}^{(t)} := \left(\sum_{i=1}^{n} \left[g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i)\right]\right) + \Omega(f_t) \tag{43}$$

Now, let's define $I_j = \{i | q(\boldsymbol{x}_i) = j\}$ as the *instance set* of leaf j. This statement is a bit complicated. Let's show an example. We'll use the diagram provided by Chen and Guestrin (2016) [5].

Example 9.16 (Instance set example for decision trees in XGBoost).

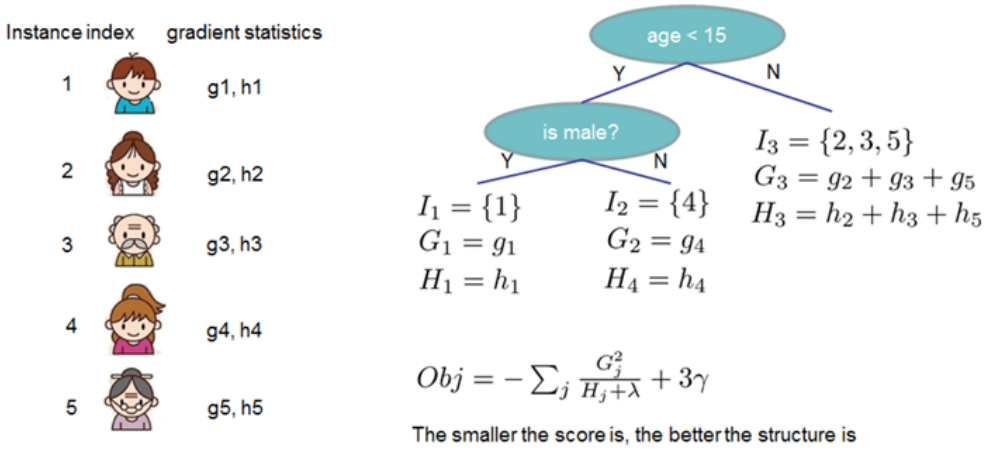


Figure 56: XGBoost example about instance sets

In Figure 56, we can see that each picture ("young boy", "mother", "elderly man", "young girl", "elderly woman") correspond to an instance index $i \in \{1, 2, 3, 4, 5\}$. We can think of "young boy" $= \boldsymbol{x}_1$, "mother" $= \boldsymbol{x}_2$, "elderly man" $= \boldsymbol{x}_3$, "young girl" $= \boldsymbol{x}_4$, and "elderly woman" $= \boldsymbol{x}_5$. We see that we have three leaves in our example decision tree. Without loss of generality, label the index of the leaves, which we will denote as j, from left to right from 1 to 3.

So, using the formula $I_j = \{i \text{ such that } q(\boldsymbol{x}_i) = j\}$, we can conclude $I_1 = \{1\}$, $I_2 = \{4\}$, and $I_3 = \{2, 3, 5\}$.

Let's get back to describing XGBoost. For Equation (43), let us rewrite it by expanding $\Omega(f_t)$ [5].

$$\widetilde{\mathcal{L}}^{(t)} := \left(\sum_{i=1}^{n} \left[g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i) \right] \right) + \Omega(f_t) \quad \text{From Equation (43)} \tag{44}$$

$$= \left(\sum_{i=1}^{n} \left[g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i) \right] \right) + \left(\gamma |T| + \frac{1}{2} \lambda \sum_{j=1}^{|T|} w_j^2 \right) \quad \text{Expand } \Omega(f_t) \tag{45}$$

Let's make use of the notation I_j we just created. Using clever reasoning, we can go from Equation (45) to:

$$\widetilde{\mathcal{L}}^{(t)} = \sum_{j=1}^{|T|} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma |T| \tag{46}$$

To see why this is true, we will show the simple example that

$$\sum_{i=1}^{n} g_i f_t(\boldsymbol{x}_i) = \sum_{j=1}^{|T|} (\sum_{i \in I_j} g_i) w_j \tag{47}$$

Since, g_i is on both sides of Equation (47), we can ignore it. The most important thing to show is that we are summing over the same i's.

$$\sum_{i=1}^{n} f_t(\boldsymbol{x}_i) = \sum_{j=1}^{|T|} \sum_{i \in I_j} w_j \tag{48}$$

On the LHS, f_t takes in some data $\boldsymbol{x}_i$ and output the weight of that data. For convenience, let us now denote that weight as w_i. Thus, on LHS, the order we are doing the summation is over the order the data $\boldsymbol{x}_i$ is inputted.

On the RHS, we are instead doing the order of our summation over leaves. We start at leaf $j = 1$, and end at leaf $j = |T|$ where $|T|$ is the total number of leaves. Then, for each $\boldsymbol{x}_i$ sorted into a generic leaf j, we count the number of i's that are in that leaf. This is $\sum_{i \in I_j}$. And since a leaf can only have one weight, we add the weight w_j, $|\{i \in I_j\}|$ times where $|\cdot|$ denotes cardinality.

Now that Equation (46) makes sense, we can continue describing XGBoost.

For a fixed leaf structure $q(\boldsymbol{x})$, we can compute the optimal weight w_j^* of leaf j by [5]:

$$w_j^* = -\left(\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \right) \tag{49}$$

This calculation is not actually too difficult to do. If we want to check ourselves, take the derivative with respect to w_j of

$$\left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] \tag{50}$$

in Equation (46), and set the LHS to 0.

Now that we know w_j^*, if we go back to Equation (46) and substitute w_j^* in place of w_j, we get [5]:

$$\widetilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{|T|} \left[\frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} \right] + \gamma |T| \tag{51}$$

If we want to check ourselves, the calculation is not difficult but a bit tedious.

Equation (51) can now be used as a scoring function to measure the quality of a tree structure q [5]. Think of Equation (51) as analogous to some impurity score when splitting using the CART algorithm. Once again, remember that we want $\widetilde{\mathcal{L}}^{(t)}$ to be as small as possible.

In practice, since it is too difficult to find the optimal q to reduce $\widetilde{\mathcal{L}}^{(t)}(q)$ to be small as possible, we use a greedy algorithm to determine splits [5].

Assume that I_L and I_R are the *instance sets* of left and right node after the split. Letting $I = I_L \cup I_R$, then the loss reduction after the split is given by:

$$\mathcal{L}_{\text{split loss reduction}} = \frac{1}{2}\left[\left(\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda}\right) + \left(\frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda}\right) - \left(\frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right)\right] - \gamma \quad (52)$$

We will briefly show where this result comes from. We need to reference Equation (51) for:

$$\widetilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2}\sum_{j=1}^{|T|}\left[\frac{(\sum_{i\in I_j} g_i)^2}{(\sum_{i\in I_j} h_i) + \lambda}\right] + \gamma|T| \quad (53)$$

Let $\widetilde{\mathcal{L}}_{\text{original}}$ be the loss before the split. Let $\widetilde{\mathcal{L}}_{\text{split}}$ be the loss after the split.

$$\mathcal{L}_{\text{split loss reduction}} \quad (54)$$

$$= \widetilde{\mathcal{L}}_{\text{original}} - \widetilde{\mathcal{L}}_{\text{split}} \quad (55)$$

$$= \left[-\frac{1}{2}\left(\frac{(\sum_{i\in I} g_i)^2}{(\sum_{i\in I} h_i) + \lambda}\right) + \gamma|T|\right] - \left[-\frac{1}{2}\left(\frac{(\sum_{i\in I_L} g_i)^2}{(\sum_{i\in I_L} h_i) + \lambda}\right) - \frac{1}{2}\left(\frac{(\sum_{i\in I_R} g_i)^2}{(\sum_{i\in I_R} h_i) + \lambda}\right) + \gamma(|T|+1)\right] \quad (56)$$

$$= \frac{1}{2}\left[\left(\frac{(\sum_{i\in I_L} g_i)^2}{(\sum_{i\in I_L} h_i) + \lambda}\right) + \left(\frac{(\sum_{i\in I_R} g_i)^2}{(\sum_{i\in I_R} h_i) + \lambda}\right) - \left(\frac{(\sum_{i\in I} g_i)^2}{(\sum_{i\in I} h_i) + \lambda}\right)\right] - \gamma \quad \text{By rearrangement} \quad (57)$$

In practice, the formula in Equation (52) is used to evaluate split candidates. We want $\mathcal{L}_{\text{split loss reduction}}$ to be positive. Remember $\mathcal{L}_{\text{split loss reduction}} = \widetilde{\mathcal{L}}_{\text{original}} - \widetilde{\mathcal{L}}_{\text{split}}$. If $\widetilde{\mathcal{L}}_{\text{original}}$ is greater than $\widetilde{\mathcal{L}}_{\text{split}}$, this means that we have reduced our loss by splitting.

9.7 Light Gradient Boosting Machine (LightGBM)

Light Gradient Boosting Machine is a technique created by Ke et al (2017) [15]. LightGBM makes use of both the CART methodology and gradient boosting decision trees.

An issue with other gradient boosting decision tree methods is that as the dimension and number of data points increase, efficiency and scalability are unsatisfactory. To solve this problem, Ke et al (2017) propose two methods.

These are *Gradient-Based One Sided Sampling* (GOSS) and *Exclusive Feature Bundling*(EFB).

Ke et al (2017) [15] states when LightGBM is run on real data, LightGBM speeds up the training process up to 20 times when compared to methods like XGBoost.

9.7.1 LightGBM Algorithm - Gradient-Based One Sided Sampling

When working with large data sets, one intuitive idea to speed up calculation is to reduce the number of data instances and number of features.

There is the often quoted "stylized fact" that "80% of effects come from 20% of causes." We will follow this same intuition, and attempt to sample instances which contribute more to information gain more often, and instances which contribute less to information gain less often.

Ke et al (2017) notices that instances with larger gradients (in absolute value terms) will contribute more to information gain. Ke et al (2017) calls these "under-trained instances"[15]. The idea behind the name "under-trained instances" is that you want to start training the model where gradient values are larger. When the absolute gradient values are small, those instances will not contribute a lot to information gain. These instances with small absolute gradients are what Ke et al (2017) calls "well-trained".

Sampling a smaller subset of the original dataset is called *down sampling*. Thus, when down sampling, we should keep those instances with large absolute gradients and only randomly sample a portion of the instances with smaller absolute gradients [15].

The pseudocode for GOSS is as below. We will not go line by line through the pseudocode.

Algorithm 2: Gradient-based One-Side Sampling

Input: I: training data, d: iterations
Input: a: sampling ratio of large gradient data
Input: b: sampling ratio of small gradient data
Input: $loss$: loss function, L: weak learner
models $\leftarrow$ {}, fact $\leftarrow \frac{1-a}{b}$
topN $\leftarrow$ a $\times$ len(I) , randN $\leftarrow$ b $\times$ len(I)
for $i = 1$ ***to*** d **do**
    preds $\leftarrow$ models.predict(I)
    g $\leftarrow$ $loss$(I, preds), w $\leftarrow$ {1,1,...}
    sorted $\leftarrow$ GetSortedIndices(abs(g))
    topSet $\leftarrow$ sorted[1:topN]
    randSet $\leftarrow$ RandomPick(sorted[topN:len(I)], randN)
    usedSet $\leftarrow$ topSet + randSet
    w[randSet] $\times =$ fact $\triangleright$ Assign weight $fact$ to the small gradient data.
    newModel $\leftarrow$ L(I[usedSet], $-$ g[usedSet], w[usedSet])
    models.append(newModel)

Figure 57: LightGBM GOSS Pseudocode

When working with a large data set, one may be tempted to entirely discard the instances with small gradients. However, the data distribution will be changed by doing so, and the accuracy of our model will decrease [15]. GOSS proposes sampling all the instances with large gradients and perform random sampling on the instances with small gradients.

Specifically, GOSS does the following [15].

1. Sort the data instances according to their absolute gradients and select the top $a \times 100\%$ of *total instances*. Total instances is total number of instances in the whole data set. Here $a \in (0, 1]$

2. In the data set with small gradients, randomly sample $b \times 100\%$ of those instances with small gradients. Here, $b \in (0, 1]$

3. Amplified the sampled data with small gradients by a constant $\frac{1-a}{b}$.

Theoretical Analysis

The theoretical analysis for LightGBM is quite complicated and not thoroughly explained. I'll leave you with the key points. We will quote heavily from Ke et al (2017)[15].

Let's assume that we have a *standard gradient boosting decision tree* which learns a function from the input space $\mathcal{X}^s$ to the gradient space $\mathcal{G}$. Also assume that we have a

training set with n independent and identically distributed instances $\{x_1, x_2, ..., x_n\}$, where each x_i is a vector with dimension s in space $\mathcal{X}^s$. In each iteration of gradient boosting, the negative gradients of the loss function with respect to the output of the model are denoted as $\{g_1, g_2, ...g_n\}$ [15]. Then, the quantity we want to maximize, our information gain function, is as below.

Definition 3.1 *Let O be the training dataset on a fixed node of the decision tree. The variance gain of splitting feature j at point d for this node is defined as*

$$V_{j|O}(d) = \frac{1}{n_O}\left(\frac{\left(\sum_{\{x_i \in O: x_{ij} \leq d\}} g_i\right)^2}{n^j_{l|O}(d)} + \frac{\left(\sum_{\{x_i \in O: x_{ij} > d\}} g_i\right)^2}{n^j_{r|O}(d)}\right),$$

where $n_O = \sum I[x_i \in O]$, $n^j_{l|O}(d) = \sum I[x_i \in O : x_{ij} \leq d]$ *and* $n^j_{r|O}(d) = \sum I[x_i \in O : x_{ij} > d]$.

Figure 58: Standard GBDT used information gain function

Ke et al (2017) calls information gain by the name *variance gain*. At each node, we want to split according to the optimal feature j^* and at the optimal point d_{j^*} [15].

For *GOSS*, we first want to sort our instances by the absolute value of their gradients, with larger gradients first and smaller gradients last. Set A is the subset of top $a \times 100\%$ of total instances with larger gradients. For the remaining set A^C consisting of $(1-a) \times 100\%$ of the instances, we call A^C the set of smaller gradients. In this set of smaller gradients A^C, we want to randomly sample a subset B with size $b \times |A^C|$ where $|\cdot|$ denotes cardinality. Finally, we want to split the instances to find the estimated variance gain (information gain) over the subset $A \cup B$[15].

Ke et al (2017) uses the below complicated method:

$$\tilde{V}_j(d) = \frac{1}{n}\left(\frac{\left(\sum_{x_i \in A_l} g_i + \frac{1-a}{b}\sum_{x_i \in B_l} g_i\right)^2}{n^j_l(d)} + \frac{\left(\sum_{x_i \in A_r} g_i + \frac{1-a}{b}\sum_{x_i \in B_r} g_i\right)^2}{n^j_r(d)}\right), \quad (1)$$

where $A_l = \{x_i \in A : x_{ij} \leq d\}$,$A_r = \{x_i \in A : x_{ij} > d\}$,$B_l = \{x_i \in B : x_{ij} \leq d\}$,$B_r = \{x_i \in B : x_{ij} > d\}$, and the coefficient $\frac{1-a}{b}$ is used to normalize the sum of the gradients over B back to the size of A^c.

Figure 59: LightGBM used information gain function

Finally, Ke et al (2017) provides the following theorem for approximation error in GOSS.

Theorem 3.2 *We denote the approximation error in GOSS as* $\mathcal{E}(d) = |\tilde{V}_j(d) - V_j(d)|$ *and* $\bar{g}_l^j(d) = \frac{\sum_{x_i \in (A \cup A^c)_l} |g_i|}{n_l^j(d)}, \bar{g}_r^j(d) = \frac{\sum_{x_i \in (A \cup A^c)_r} |g_i|}{n_r^j(d)}$. *With probability at least* $1 - \delta$, *we have*

$$\mathcal{E}(d) \leq C_{a,b}^2 \ln 1/\delta \cdot \max \left\{ \frac{1}{n_l^j(d)}, \frac{1}{n_r^j(d)} \right\} + 2DC_{a,b} \sqrt{\frac{\ln 1/\delta}{n}}, \tag{2}$$

where $C_{a,b} = \frac{1-a}{\sqrt{b}} \max_{x_i \in A^c} |g_i|$, *and* $D = \max(\bar{g}_l^j(d), \bar{g}_r^j(d))$.

Figure 60: LightGBM theorem for approximation error

The key point, the asymptotic approximation ratio of GOSS is $\mathcal{O}\left(\frac{1}{n_l^j(d)} + \frac{1}{n_r^j(d)} + \frac{1}{\sqrt{n}}\right)$ [15]. If the approximation error is dominated by the $\frac{1}{\sqrt{n}}$ term – highlighted in Figure (60), then as $n \to \infty$, the approximation error approaches 0.

9.7.2 LightGBM Algorithm - Exclusive Feature Bundling

Exclusive feature bundling is a technique to reduce the number of features for sparse data. Assume we have a data set where we have many categorical variables. Let's assume these categorical variables are stored by one hot encoding. Ke et al (2017) claims that in a sparse feature space, many features are mutually exclusive — they never take nonzero values simultaneously. These features which are mutually exclusive, can be safely bundled into a single feature, which Ke et al calls an *exclusive feature bundle*.

The pseudocode for exclusive feature bundling is below:

Algorithm 3: Greedy Bundling

```
Input: F: features, K: max conflict count
Construct graph G
searchOrder ← G.sortByDegree()
bundles ← {}, bundlesConflict ← {}
for i in searchOrder do
    needNew ← True
    for j = 1 to len(bundles) do
        cnt ← ConflictCnt(bundles[j],F[i])
        if cnt + bundlesConflict[i] ≤ K then
            bundles[j].add(F[i]), needNew ← False
            break
    if needNew then
        Add F[i] as a new bundle to bundles
Output: bundles
```

Algorithm 4: Merge Exclusive Features

```
Input: numData: number of data
Input: F: One bundle of exclusive features
binRanges ← {0}, totalBin ← 0
for f in F do
    totalBin += f.numBin
    binRanges.append(totalBin)
newBin ← new  Bin(numData)
for i = 1 to numData do
    newBin[i] ← 0
    for j = 1 to len(F) do
        if F[j].bin[i] ≠ 0 then
            newBin[i] ← F[j].bin[i] + binRanges[j]
Output: newBin, binRanges
```

Figure 61: LightGBM pseudocode for exclusive feature bundling

Exclusive feature bundling is better explained with a simple example.

Example 9.17 (Exclusive Feature Bundling Graph Coloring Example).

Ke et al (2017) compares the problem of exclusive feature bundling to the *graph coloring problem*. The graph coloring problem is a classic math problem, where if we have some graph $G = (V, E)$, we want to find the minimum number of colors so that all directly connected vertices are a different color.

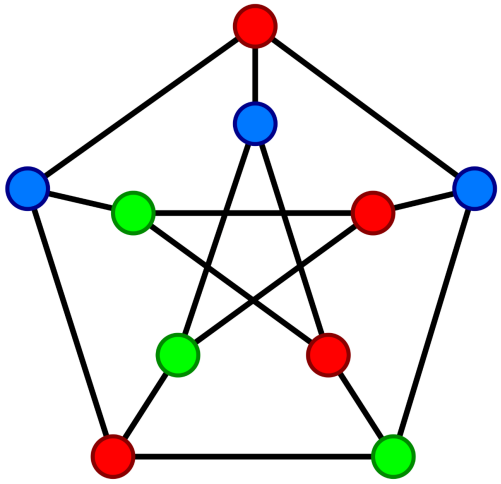


Figure 62: LightGBM graph coloring problem example [6]

Let's hypothetically say that we are trying to do some sentiment analysis. We will go onto the internet, onto some forum where people talk about the stock market, and gauge whether market sentiment is positive or negative. Let's say that each instance is a comment on this forum.

Instance Number	**A**	**B**	**C**	**D**	**E**
Instance 1	1	0	0	1	1
Instance 2	0	1	0	0	1

Table 5: Simple graph coloring problem example

In Table 5, let's say that each letter A, B, C, D, E stands for a particular word. Let's say $A =$ "up", $B =$ "down", $C =$ "call", $D =$ "put", $E =$ "volatility". We will use one-hot encoding. So $A = 1$ means that the word "up" shows up in that particular instance, and $A = 0$ means that "up" doesn't show up.

In these graphs, if two words like A, B show up in the same instance, then we will draw an edge ab between points A and B.

Let's graph these two instances.

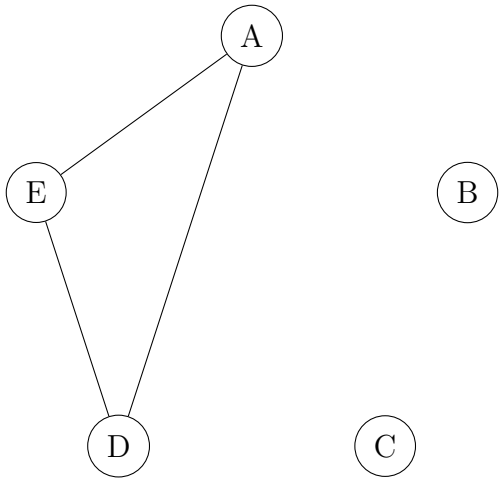


Figure 63: Graph of instance 1

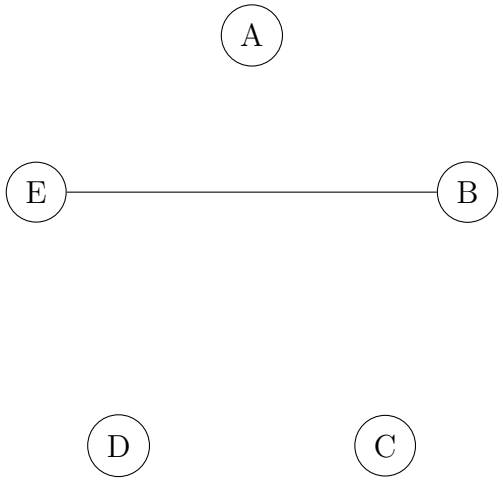


Figure 64: Graph of instance 2

Now, let us add Figures 63 and 64 together. We get:

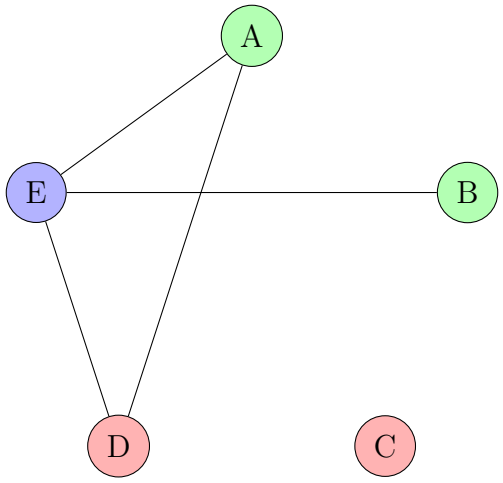


Figure 65: Graph of instance 1 combined with instance 2

In Figure 65, we have added instance 1 and instance 2 together. For this simple example, we can see using our eyes that we need at least three colors so no directly connected vertices have the same color. This is the same idea behind *exclusive feature bundling*, where each same colored node is a part of the same *exclusive feature bundle*.

We will now describe the algorithm used to create exclusive feature bundles proposed in LightGBM.

Example 9.18 (**Exclusive Feature Bundling Algorithm Example**)**.**

First, we want to construct a graph with weighted edges, whose weights correspond to the total conflict between features. Let's assume that our graph looks like below:

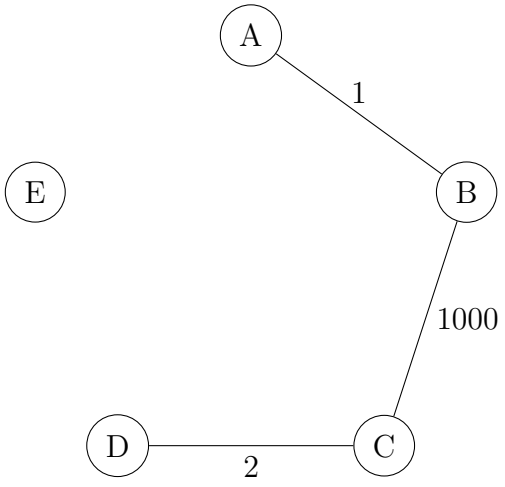


Figure 66: Graph with weighted edges

In Figure 66, we see that A and B have 1 conflict, B and C have 1000 conflicts, and C and D have 2 conflicts.

Second, we want to sort the features by their *degrees* in the graph in the descending order. In graph theory, *degrees* means the number of edges connected to a vertex.

So, we will sort our features like so:

Feature	**Degrees of feature**
B	2
C	2
A	1
D	1
E	0

Table 6: Degrees of our feature

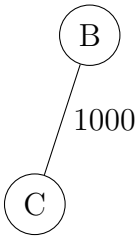
Ke et al (2017) has this variable γ to denote the maximal conflict rate allowed in each bundle. For the sake of simplicity, I will create my own analogous K which is just the raw total number of conflicts allowed. I will set $K = 5$.

Now, we want to check each feature in the ordered list, and either assign it to an existing bundle (where the amount of conflict allowed is controlled by γ), or create a new bundle [15].

Let's go through our example step by step.

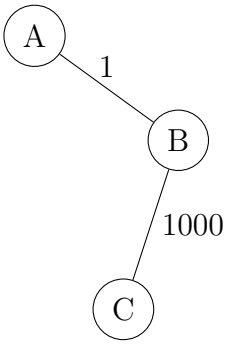
1. Since B is at the top of our list, we will create a node B. Since there are no "bundle groups" yet, B will go into $G_1 = \{B\}$.

2. We will create the node C.



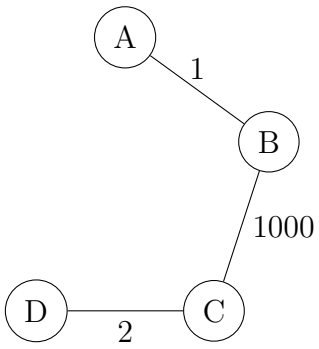
   Since C and B have 1000 conflicts, and $1000 > 5 = K$, we will have to put C into a different "bundle group". So $G_1 = \{B\}, G_2 = \{C\}$.

3. We will create the node A.



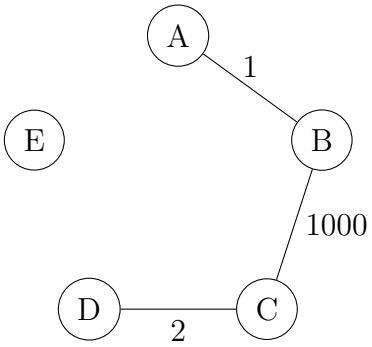
We see that A and B have only 1 conflict, so $1 < 5 = K$. Thus, A should go into an existing bundle. For the sake of convenience, let's put A into G_1 since G_1 was created first. So $G_1 = \{B, A\}$ and $G_2 = \{C\}$.

4. We will create the node D.



Since nodes C and D only have 2 conflicts and $2 < 5 = K$, D should go into an existing bundle. Let's put D into G_1. $G_1 = \{B, A, D\}$ and $G_2 = \{C\}$.

5. We will create the node E.



Since node E has no conflict with any nodes, we can put it into an existing bundle. Let $G_1 = \{B, A, D, E\}$ and $G_2 = \{C\}$.

This is just a simple illustrive example of the exclusive feature bundling algorithm. The actual algorithm used in LightGBM is more efficient, because it does not build a graph, but orders by the count of non-zero values [15]. The pseudocode for this more efficient algorithm is not shown, but since the only difference is in ordering strategy, the pseudocode in Figure (61) is mostly unchanged.

The last thing to comment on is Algorithm 4: Merge Exclusive Features in Figure (61). The key idea here is handle issues where the features from the same bundle can take the same values. Lets say that A and B are from the same bundle. If feature A takes values from $[0, 10)$ and feature B takes values from from $[0, 20)$, then we want to offset B so the values of B are from $[10, 30)$. Then we can merge features A and B, so that the range of $A + B$ is $[0, 30)$ [15].

9.8 Categorical Boosting (CatBoost)

CatBoost is a technique created by Prokhorenkova et al (2018) [22]. Catboost makes use of both the CART algorithm and gradient boosting for decision trees.

CatBoost is a gradient algorithm specifically designed to work categorical variables with minimal pre-processing. Generally, when working with categorical variables, one must do one hot encoding, but you do not need to do so with CatBoost.

Key benefits of CatBoost are *ordered boosting*, and an innovative algorithm for processing categorical features [22]. Both these techniques were created to fight *prediction shift*, which is a type of *target leakage*. *Target leakage* is when a model during training, learns information in the test set, which ideally it should not learn.

To explain CatBoost thoroughly, we must first explain *target statistics*, and *ordered boosting*.

9.8.1 Target Statistics

For CatBoost, reference the notation for decision trees used in Section 9.3 , Gradient Boosting for Decision Trees. As a refresher, $\mathcal{D} = \{(\boldsymbol{x}_k, y_k)\}_{k=1,2,...,n}$ where $\boldsymbol{x}_k = (x_k^1, x_k^2, ..., x_k^m)$ is a row vector of m features and $y_k \in \mathbb{R}$ is the target which we wish to predict. Notice that for $\boldsymbol{x}_k$ where $k \in \{1, 2, .., n\}$, that in $\boldsymbol{x}_k = (x_k^1, x_k^2, ..., x_k^m)$, the superscript on x_k denotes the l^{th} feature for $l = 1, 2, ..., m$.

When using categorical variables, *one hot encoding* leads to an infeasibly large number of new features as our data size increases. For example, categorical variables like "user ID" where "user IDs" are unique would lead to a new category column for each new user under one-hot encoding.

A way to deal with this issue is to use *target statistics*.

Prokhorenkova et al (2018) states, "an effective and efficient way to deal with a categorical feature i is to substitute the category x_k^i of the k^{th} training example with *one* numeric feature equal to some *target statistic* $\hat{x}_k^i$. Commonly, it estimates the expected target y conditioned by the category: $\hat{x}_k^i \approx E[y|x^i = x_k^i]$ " [22]

Let's explain the above statement clearly. Let's imagine that our categorcal feature i is colors. So, let's say in our data set the only two colors available to x^i are "red" and "blue". So $x^i \in \{$"red", "blue"$\}$. Let's assume we are working with the k^{th} example where $\boldsymbol{x}_k = (x_k^1, x_k^2, ..., x_k^m)$. So let's imagine x_k^i is the realization of the i^{th} feature where $i \in \{1, 2, ...m\}$, x^i is a categorical variable, and we are working with the k^{th} training example (what Ke calls instances). So the two potential values of x_k^i can be either "red" or "blue".

So we can now interpret the statement $\hat{x}_k^i \approx E[y|x^i = x_k^i]$. Let's say that if $x^i =$ "red", then $E[y] = 100$ where y is from either the training or testing set. Also, let's say that if $x^i =$ "blue", then $E[y] = 50$ where y is from either the training or testing set. This information is exactly what $\hat{x}_k^i \approx E[y|x^i = x_k^i]$ is trying to encode. Notice that in this paper, the training set and tests are *not* mutually exclusive. The training set can be a subset of the test set. This is not conventional, so it should be kept in mind.

Greedy Target Statistic A straightforward method to estimate the expected value of y on any data set (training or testing) given $x^i = x_k^i$, is to use the training examples' y_k's.

Usually, this estimate is noisy for low-frequency categories, and one smooths it by some prior p. Thus, we would write:

$$\hat{x}_k^i = \frac{(\sum_{j=1}^n \mathbf{1}_{\{x_j^i = x_k^i\}} \cdot y_j) + ap}{(\sum_{j=1}^n \mathbf{1}_{\{x_j^i = x_k^i\}}) + a} \tag{58}$$

where $a > 0$. A common setting for p is the average target value in the training set [22].

However, we can see a serious issue with this method. We have just feature engineered $\hat{x}_k^i$, but this feature takes in the input y_k from the training data points $\{(\boldsymbol{x}_k, y_k)\}_{k=1,2,...,z}$ where $z \leq n$. In a way, we are indirectly using y_k to predict y_k, because $\hat{x}_k^i$ takes in y_k, and also helps predict y_k. This is *target leakage* because feature $\hat{x}_k^i$ is computed using y_k, the target of $\boldsymbol{x}_k$.

This problem leads to conditional shift. The distribution of $\hat{x}^i|y$ differs over the training and test examples [22]. Here, $\hat{x}^i \approx E[y|x^i]$ where x^i is a random variable. Here y is just any generic y over the training and test examples. The intuition is this: when running over the testing set, we would not know the real y's. However, when running over the training set, you would know the real y_k's and you are thus overfitting the data because you are using information that your model would not know in the testing set.

Because of the issue with conditional shift, we specify that we want the following property [22].

P1 $\mathbb{E}(\hat{x}^i \mid y = v) = \mathbb{E}(\hat{x}^i_k \mid y_k = v)$, *where* $(\mathbf{x}_k, y_k)$ *is the k-th training example.*

The interpretation of the above **P1** is as follows.

1. y is a generic target. y can be in the training set or the testing set.
2. y_k is a realized value in the training set.
3. v is just a generic value.
4. $E[\hat{x}^i|y = v]$ is the expectation of our target statistic $\hat{x}^i$ on any value in the training or test set, conditioned on $y = v$, where y can be in the training or test set.
5. $E[\hat{x}^i_k|y_k = v]$ is the expectation of our target statistic $\hat{x}^i_k$ where the target statistic is in the training set, conditioned on $y_k = v$, where y_k is also in the training set.

Holdout Target Statistic

Prokhorenkova et al (2018) suggests a way to fix the data leakage problem is through the *holdout target statistic*.

The general idea is that we want to calculate the target statistic for $\boldsymbol{x}_k$ on a subset of examples $\mathcal{D}_k \subset \mathcal{D} \setminus \{\boldsymbol{x}_k\}$ excluding $\boldsymbol{x}_k$ using:

$$\hat{x}^i_k = \frac{(\sum_{\boldsymbol{x}_j \in \mathcal{D}_k} \mathbf{1}_{\{x^i_j = x^i_k\}} \cdot y_j) + ap}{(\sum_{\boldsymbol{x}_j \in \mathcal{D}_k} \mathbf{1}_{\{x^i_j = x^i_k\}}) + a} \tag{59}$$

We want to partition the training dataset into two parts $\mathcal{D} = \widehat{\mathcal{D}}_0 \sqcup \widehat{\mathcal{D}}_1$. Now, set $\mathcal{D}_k = \widehat{\mathcal{D}}_0$, and use Equation (59) for calculating the target statistic, and $\widehat{\mathcal{D}}_1$ for training [22].

However, we want to meet the following condition.

P2 *Effective usage of all training data for calculating TS features and for learning a model.*

Leave-one-out Target Statistic

The *leave one out target statistic* works as follows: $\mathcal{D}_k = \mathcal{D} \setminus \boldsymbol{x}_k$ are training examples for $\boldsymbol{x}_k$ and $\mathcal{D}_k = \mathcal{D}$ are testing examples. However, Prokhorenkova says that this method still has target leakage [22].

Ordered Target Statistic

CatBoost introduces the concept of "artificial time" by making a random permutation σ of the training examples. Then for each example, we use all the available "history" to compute the target statistic. Specifically, let $\mathcal{D}_k = \{\boldsymbol{x}_j : \sigma(j) < \sigma(k)\}$ for a training example, and $\mathcal{D}_k = \mathcal{D}$ for a test one [22]. Prokhorenkova claims that this method satisfies both conditions **P1** and **P2**.

9.8.2 Ordered Boosting

Prokhorenkova et al (2018) claims that even standard gradient boosting leads to *prediction shift*. Prediction shift is defined as follows [22]:

The conditional distribution $F^{t-1}(\boldsymbol{x}_k)|\boldsymbol{x}_k$ for a training example $\boldsymbol{x}_k$ is shifted, in general, from the distribution $F^{t-1}(\boldsymbol{x})|\boldsymbol{x}$ for a test sample $\boldsymbol{x}$.

To solve the issue of *prediction shift*, Prokhorenkova proposes the idea of *ordered boosting*. Ordered boosting resembles the ordered target statistic method.

The theoretical setup of the ordered boosting method is as follows.

Theory for Ordered Boosting

Before we begin, let us define our residual function as $r^{t-1}(\boldsymbol{x}_k, y_k) := y_k - F^{t-1}$. For now, let us assume that we want to find $r^{t-1}(\boldsymbol{x}_k, y_k)$ because it is analogous to finding a gradient $-g^t(\boldsymbol{x}_k, y_k)$.

Let us first work with simple hypothetical world where we have unlimited access to training data. At each step of boosting, we sample a new dataset $\mathcal{D}_t$ independently and obtain unshifted residuals by applying the current model to new training examples [22]. If we have unlimited data, then we don't need to worry about prediction shift.

In practice, training data is limited. Prokhorenkova states, "Assume we learn a model with I number of trees. To make the residual $r^{I-1}(\boldsymbol{x}_k, y_k)$ unshifted, we need to have F^{I-1} trained without the example $\boldsymbol{x}_k$. Since we need unbiased residuals for all training examples, no examples may be used for training F^{I-1}, which at first glance makes the training process impossible. However, it is possible to maintain a set of models differing by examples used for their training. Then, for calculating the residual on an example, use a model trained without it. In order to construct such a set of models, we can use the ordering principle previously applied to target statistics" [22]

Let's explain a simple example of ordered boosting.

Example 9.19 (**Ordered Boosting Simple Example**)**.**

The actual implementation of CatBoost is much more complicated, but this illustrative example captures the main idea behind *ordered boosting*.

First let's assume that we have some data $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), (\boldsymbol{x}_3, y_3), (\boldsymbol{x}_4, y_4)\}$. We want to permute the data. Let's use the tuple σ to encode how we want to permute our data. Let $\sigma = (4, 2, 1, 3)$. For this simple example, let us assume that we only have 1 permutation σ. In the full CatBoost implementation, there are multiple permutations $\sigma_1, ..., \sigma_s$ where s is a chosen number. $\sigma = (4, 2, 1, 3)$ means the new order of our data $\mathcal{D}$, which we will call $\mathcal{D}' = \{(\boldsymbol{x}_4, y_4), (\boldsymbol{x}_2, y_2), (\boldsymbol{x}_1, y_1), (\boldsymbol{x}_3, y_3)\}$. For the sake of clarity when doing further discussion, we will relabel the indices in $\mathcal{D}'$ so that $(\boldsymbol{x}_4, y_4) = (\boldsymbol{x}'_1, y'_1), (\boldsymbol{x}_2, y_2) = (\boldsymbol{x}'_2, y'_2), (\boldsymbol{x}_1, y_1) = (\boldsymbol{x}'_3, y'_3),$ and $(\boldsymbol{x}_3, y_3) = (\boldsymbol{x}'_4, y'_4)$.

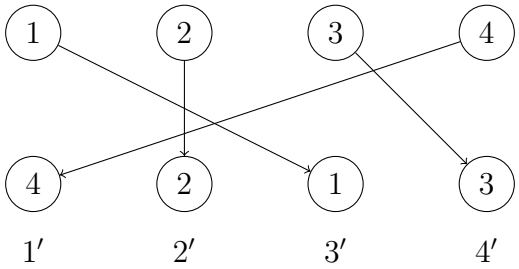


Figure 67: Idea behind permutation

Here, we want to train n different supporting models $M_1, ..., M_n$ such that the model M_i is learned only using the first i examples in the permuted data and n is the total number of data points.

So, we create the notation $\mathcal{D}'_i$ where i denotes how many of the first i examples of the permuted data set are in the data set.

So, in our case

$$\mathcal{D}'_1 = \{(\boldsymbol{x}'_1, y'_1)\} \tag{60}$$

$$\mathcal{D}'_2 = \{(\boldsymbol{x}'_1, y'_1), (\boldsymbol{x}'_2, y'_2)\} \tag{61}$$

$$\mathcal{D}'_3 = \{(\boldsymbol{x}'_1, y'_1), (\boldsymbol{x}'_2, y'_2), (\boldsymbol{x}'_3, y'_3)\} \tag{62}$$

$$\mathcal{D}'_4 = \{(\boldsymbol{x}'_1, y'_1), (\boldsymbol{x}'_2, y'_2), (\boldsymbol{x}'_3, y'_3), (\boldsymbol{x}'_4, y'_4)\} \tag{63}$$

So, we will train model M_1 using $\mathcal{D}'_1$, M_2 using $\mathcal{D}'_2$, M_3 using $\mathcal{D}'_3$, and M_4 using $\mathcal{D}'_4$.

Prokhorekova et al (2018) states, "At each step, in order to obtain the residual for the j^{th} sample, we use the model M_{j-1} [22]".

So, to find r_2, which I define as the residual between y'_2 and the predicted $M_1(\boldsymbol{x}'_2)$, we need to use M_1 and $(\boldsymbol{x}'_2, y'_2)$. To find r_3, we need to use M_2 and $(\boldsymbol{x}'_3, y'_3)$. To find r_4, we need to use M_3 and $(\boldsymbol{x}'_4, y'_4)$.

Prokhorenkova et al (2018) provides a more explicit diagram:

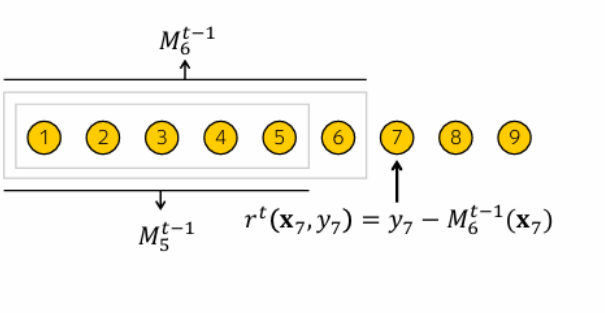


Figure 1: Ordered boosting principle, examples are ordered according to σ.

Figure 68: Residual calculation example [22]

In Figure 68, we can see that the data points are already permuted. In this example, we can see that $r^t(\boldsymbol{x}_7, y_y) = y_7 - M_6^{t-1}(\boldsymbol{x}_7)$.

The reason we want to find the residual is because Prokhorenkova et al (2018) states, "We formally analyze the problem of prediction shift in a simple case of a regression task with the quadratic loss function $L(y, \hat{y}) = (y - \hat{y})^2$. In this case, the negative gradient $-g^t(\boldsymbol{x}_k, y_k)$ can be substituted by the residual function $r^{t-1}(\boldsymbol{x}_k, y_k) := y_k - F^{t-1}(\boldsymbol{x}_k)$[22]". A bit more explicitly, it can be shown $-g^t(\boldsymbol{x}_k, y_k) = 2(y_k - F^{t-1}(\boldsymbol{x}_k))$ but the constant 2 is not critical for the implementation of ordered boosting.

Let us quickly show $-g^t(\boldsymbol{x}_k, y_k) = 2(y_k - F^{t-1}(\boldsymbol{x}_k)$. Let's assume that we are working with a generic $\boldsymbol{x}$ and y.

$$g^t(\boldsymbol{x}, y) := \frac{\partial L(y, \hat{y})}{\partial \hat{y}}\bigg|_{\hat{y}=F^{t-1}(\boldsymbol{x})} \quad \text{By definition of } g^t(\boldsymbol{x}, y) \tag{64}$$

$$= \frac{\partial (y - \hat{y})^2}{\partial \hat{y}}\bigg|_{\hat{y}=F^{t-1}(\boldsymbol{x})} \quad \text{Sub in } L(y, \hat{y}) = (y - \hat{y})^2 \tag{65}$$

$$= -2(y - \hat{y})\bigg|_{\hat{y}=F^{t-1}(\boldsymbol{x})} \tag{66}$$

$$= -2(y - F^{t-1}(\boldsymbol{x})) \tag{67}$$

$$\implies -g^t(\boldsymbol{x}, y) = 2(y - F^{t-1}(\boldsymbol{x})) \tag{68}$$

Remember from Equation (37) that:

$$h^t = \arg\min_{h \in H} E\left[\left(-g^t(\boldsymbol{x}, y) - h(\boldsymbol{x})\right)^2\right]$$

where the Equation above is how we select the optimal tree h^t to greedily reduce our expected loss $\mathcal{L}$ from line 36.

More explicitly, Equation (36) is:

$$h^t = \arg\min_{h \in H} \left(\mathcal{L}(F^{t-1} + h) \right) = \arg\min_{h \in H} \left(E \left[L \left(y, F^{t-1}(\boldsymbol{x}) + h(\boldsymbol{x}) \right) \right] \right) \tag{69}$$

In practice the expectation in Equation (37) is unknown and approximated using the same data set $\mathcal{D}$ [22].

So, in practice,

$$h^t = \arg\min_{h \in H} \frac{1}{n} \sum_{k=1}^{n} \left[\left(-g^t(\boldsymbol{x}_k, y_k) - h(\boldsymbol{x}_k) \right)^2 \right] \tag{70}$$

So using Equation (70), where we know the negative gradient $-g^t(\boldsymbol{x}_k, y_k)$ can be substituted by the residual function $r^{t-1}(\boldsymbol{x}_k, y_k) := y_k - F^{t-1}$, we can effectively do gradient boosting. We will use Equation (70) to find the next best tree h^t to reduce $\mathcal{L}$.

Then, we can use $F^t = F^{t-1} + \alpha h^t$ to train our best ensemble learner F^t. Notice that the constant 2 that we omitted in $-g^t(\boldsymbol{x}_k, y_k) = 2(y_k - F^{t-1}(\boldsymbol{x}_k))$ is not too significant, because we can adjust the constant α.

9.8.3 Implementation of Ordered Boosting

Below is the pseudocode for CatBoost. We will not go through the code line by line.

CatBoost has two modes, *ordered* and *plain*. *Plain CatBoost* is just the standard gradient boosting decision tree algorithm using ordered target statistics. *Ordered CatBoost* is an efficient implementation of the pseudocode in Algorithm 1 of Figure (69).

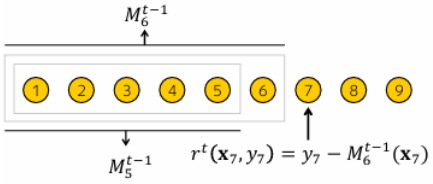


Figure 1: Ordered boosting principle, examples are ordered according to σ.

Algorithm 1: Ordered boosting

input : $\{(\mathbf{x}_k, y_k)\}_{k=1}^n$, I;

$\sigma \leftarrow$ random permutation of $[1, n]$;
$M_i \leftarrow 0$ for $i = 1..n$;
for $t \leftarrow 1$ **to** I **do**
  for $i \leftarrow 1$ **to** n **do**
    $r_i \leftarrow y_i - M_{\sigma(i)-1}(\mathbf{x}_i)$;
  for $i \leftarrow 1$ **to** n **do**
    $\Delta M \leftarrow$ $LearnModel((\mathbf{x}_j, r_j) : \sigma(j) \le i)$;
    $M_i \leftarrow M_i + \Delta M$;
return M_n

Algorithm 2: Building a tree in CatBoost

input : M, $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, α, L, $\{\sigma_i\}_{i=1}^s$, $Mode$

$grad \leftarrow CalcGradient(L, M, y)$;
$r \leftarrow random(1, s)$;
if $Mode = Plain$ **then**
  $G \leftarrow (grad_r(i)$ for $i = 1..n)$;
if $Mode = Ordered$ **then**
  $G \leftarrow (grad_{r,\sigma_r(i)-1}(i)$ for $i = 1..n)$;
$T \leftarrow$ empty tree;
foreach *step of top-down procedure* **do**
  foreach *candidate split* c **do**
    $T_c \leftarrow$ add split c to T;
    if $Mode = Plain$ **then**
      $\Delta(i) \leftarrow \text{avg}(grad_r(p)$ for $p : leaf_r(p) = leaf_r(i))$ for $i = 1..n$;
    if $Mode = Ordered$ **then**
      $\Delta(i) \leftarrow \text{avg}(grad_{r,\sigma_r(i)-1}(p)$ for $p : leaf_r(p) = leaf_r(i), \sigma_r(p) < \sigma_r(i))$ for $i = 1..n$;
    $loss(T_c) \leftarrow \cos(\Delta, G)$
  $T \leftarrow \arg\min_{T_c}(loss(T_c))$
if $Mode = Plain$ **then**
  $M_{r'}(i) \leftarrow M_{r'}(i) - \alpha\,\text{avg}(grad_{r'}(p)$ for $p : leaf_{r'}(p) = leaf_{r'}(i))$ for $r' = 1..s, i = 1..n$;
if $Mode = Ordered$ **then**
  $M_{r',j}(i) \leftarrow M_{r',j}(i) - \alpha\,\text{avg}(grad_{r',j}(p)$ for $p : leaf_{r'}(p) = leaf_{r'}(i), \sigma_{r'}(p) \le j)$ for $r' = 1..s$, $i = 1..n$, $j \ge \sigma_{r'}(i) - 1$;
return T, M

Figure 69: CatBoost pseudocode [22]

We will address some key points about the practical CatBoost implementation.

Oblivious Decision Trees

CatBoost uses *oblivious decision trees* as base predictors. This means that the same splitting criterion is used across an entire level of the tree [22].

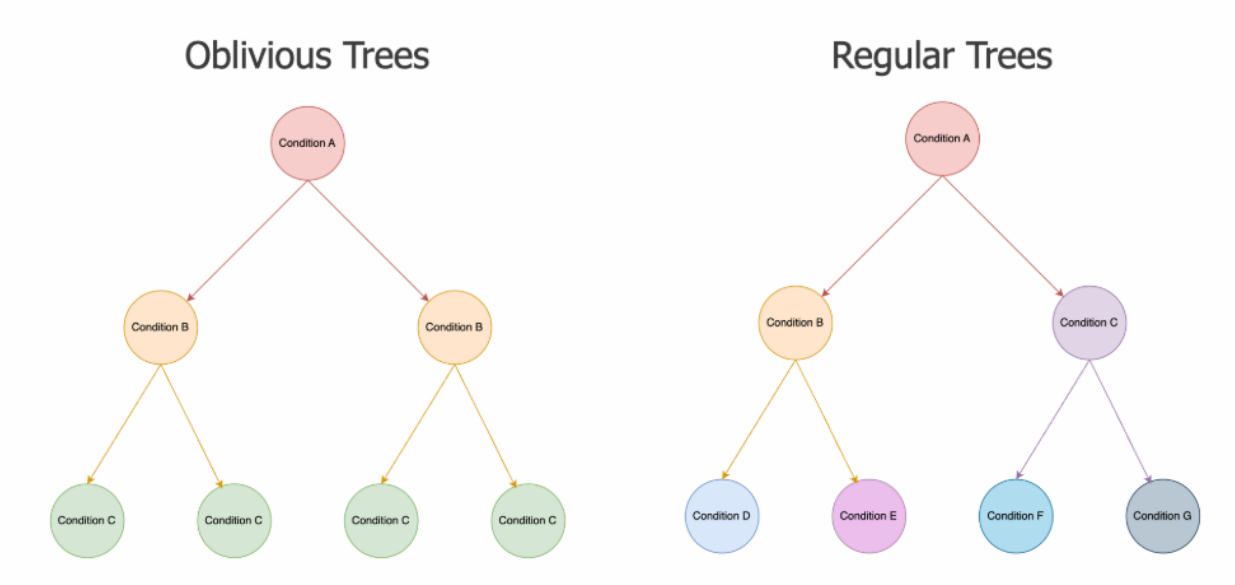


Figure 70: Oblivious decision tree [13]

The procedure to build this tree is described in Algorithm 2 in figure 69. There trees are balanced, less prone to overfitting, and allow speeding up execution at testing time significantly [22].

9.9 Shapley Values

Shapley values a technique used to explain how the input features of a machine learning models locally contribute to the final prediction.

9.9.1 Original Shapley Value

The original Shapley values were created by Shapley (1952) [25]. The intuition behind Shapley values is they explain the marginal contribution of each input feature when doing prediction.

The original formula for Shapley value is as follows:

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(M - |S| - 1)!}{M!} [f_x(S \cup \{i\}) - f_x(S)] \tag{71}$$

I've used the notation from Lundberg et al (2019)[19] but the idea is the same as in Shapley (1952).

Let us explain Equation (71) thoroughly.

1. N is the set of all input features.

2. S is the subset of the set of input features N.

3. M is the number of features in set N.

4. We can think of f_x as a function which calculates value of a *coalition*. In this case $f_x(S \cup \{i\})$ calculates the value of the *coalition*, which for our purposes is just a set, $S \cup \{i\}$.

This will be clearer with a simple example.

Example 9.20 (**Simple Shapley value example**)**.**

The example below draws from the video by *A Data Odyssey*[21].

Let's imagine that we have simple two player game. There players are denoted as P_1 and P_2. Let's say there is a competition where 1st prize wins 10,000 dollars, 2nd prize wins 7,500 dollars, and 3rd prize wins 5000 dollars.

Let's say that players 1 and 2 work together and win the first place prize. Let's denote the coalition of players 1 and 2 as C_{12}. So $C_{12} = 10,000$. However, players 1 and 2 want to split the money fairly based on their individual contributions to the team. Let's imagine we have a time machine, and we can go back in time and see what would happen if players 1 and 2 worked individually instead of together.

Let's say that these are our new coalition values. $C_1 = 7500, C_2 = 5000$, and $C_0 = 0$ where C_0 is the coalition of no players. Now, we can calculate the *marginal contribution* for both players. The *marginal contribution* of a player is the increase in a coalition's value due to that player joining the coalition.

Player 1 contribution when joining coalition

$$C_{12} - C_2 = 10000 - 5000 = 5000 \tag{72}$$

$$C_1 - C_0 = 7500 - 0 \tag{73}$$

Player 2 contribution when joining coalition

$$C_{12} - C_1 = 10000 - 7500 = 2500 \tag{74}$$

$$C_2 - C_0 = 5000 - 0 \tag{75}$$

Let's list the total number of permutations of ways groups can form. In this case, it's just 2!.

Let the expected marginal contribution of P_1 be

$$E[MC(P_1)] = P(C_{12} - C_2) \cdot (C_{12} - C_2) + P(C_1 - C_0) \cdot (C_1 - C_0)$$

Group label	1st to join	2nd to join
Group 1	1	2
Group 2	2	1

Table 7: Permutations of way group can form

where $P(C_{12} - C_2)$ is the probability of player 1 joining a group with player 2 already in it, and $P(C_1 - C_0)$ is the probability of joining an empty group.

Let us assume that the probability of joining either "Group 1" or "Group 2" is uniformly distributed. We can see that in Table (7), that under "Group 1", player 1 joins an empty group. So $P(C_1 - C_0) = \frac{1}{2}$. We see that under "Group 2", player 1 joins a group with player 2 already in it. So $P(C_1 - C_0) = \frac{1}{2}$.

Thus,

$$E[MC(P_1)] = P(C_{12} - C_2) \cdot (C_{12} - C_2) + P(C_1 - C_0) \cdot (C_1 - C_0) \tag{76}$$

$$= \frac{1}{2} \cdot 5000 + \frac{1}{2}7500 \tag{77}$$

$$= 6250 \tag{78}$$

Through a similar calculation for P_2, we obtain:

$$E[MC(P_2)] = P(C_{12} - C_1) \cdot (C_{12} - C_1) + P(C_2 - C_0) \cdot (C_2 - C_0) \tag{79}$$

$$= \frac{1}{2} \cdot 2500 + \frac{1}{2} \cdot 5000 \tag{80}$$

$$= 3750 \tag{81}$$

Notice how $E[MC(P_1)] + E[MC(P_2)] = 6750 + 3750 = 10000 = C_{12}$. This is not a coincidence.

9.9.2 Shapley Value for Trees

In Lundberg et al (2019)[19], they extend the classic notion of shapley value to trees. They call their new method SHAP which stands for SHapley Additive exPlanations.

The proposed method by Lundberg et al (2019) still uses Equation (71).

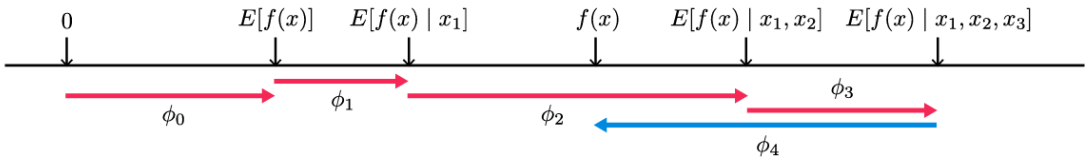


Figure 2: SHAP (SHapley Additive exPlanation) values explain the output of a function f as a sum of the effects ϕ_i of each feature being introduced into a conditional expectation. Importantly, for non-linear functions the order in which features are introduced matters. SHAP values result from averaging over all possible orderings. Proofs from game theory show this is the only possible consistent approach where $\sum_{i=0}^{M} \phi_i = f(x)$. In contrast, the only current individualized feature attribution method for trees satisfies the summation, but is inconsistent because it only considers a single ordering [24].

Figure 71: SHAP for trees [19]

Above is an idea of how SHAP works. To apply shapley values for trees, we must estimate $E[f(x)|x_S]$. In Figure 71, we see that $x_S = \{x_1, x_2, x_3, x_4\}$. Once we have estimated $E[f(x)|x_S]$, we want to set $f_x(S) = (E[f(x)|x_S]$ and use Equation (71) [19].

Below is the illustrative algorithm to estimate $E[f(x)|x_S]$.

Algorithm 1 Estimating $E[f(x) \mid x_S]$

```
procedure EXPVALUE(x, S, tree = {v, a, b, t, r, d})
    procedure G(j, w)
        if v_j ≠ internal then
            return w · v_j
        else
            if d_j ∈ S then
                return G(a_j, w) if x_{d_j} ≤ t_j else G(b_j, w)
            else
                return G(a_j, w r_{a_j}/r_j) + G(b_j, w r_{b_j}/r_j)
            end if
        end if
    end procedure
    return G(1, 1)
end procedure
```

Figure 72: SHAP illustrative algorithm [19]

We will not go through this line by line, as it is actually quite difficult and dense. The full algorithm used to calculate Shapley values for trees is below.

Algorithm 2 Tree SHAP

procedure TS(x, $tree = \{v, a, b, t, r, d\}$)
  ϕ = array of $len(x)$ zeros
  procedure RECURSE(j, m, p_z, p_o, p_i)
    m = EXTEND(m, p_z, p_o, p_i)
    if $v_j \neq internal$ **then**
      for $i \leftarrow 2$ to $len(m)$ **do**
        $w = sum(\text{UNWIND}(m, i).w)$
        $\phi_{m_i} = \phi_{m_i} + w(m_i.o - m_i.z)v_j$
      end for
    else
      $h, c = x_{d_j} \leq t_j \ ? \ (a_j, b_j) : (b_j, a_j)$
      $i_z = i_o = 1$
      k = FINDFIRST($m.d, d_j$)
      if $k \neq$ nothing **then**
        $i_z, i_o = (m_k.z, m_k.o)$
        m = UNWIND(m, k)
      end if
      RECURSE(h, m, $i_z r_h / r_j$, i_o, d_j)
      RECURSE(c, m, $i_z r_c / r_j$, 0, d_j)
    end if
  end procedure
  procedure EXTEND(m, p_z, p_o, p_i)
    $l = len(m)$
    $m = copy(m)$
    $m_{l+1}.(d, z, o, w) = (p_i, p_z, p_o, l = 0 \ ? \ 1 : 0)$
    for $i \leftarrow l - 1$ to 1 **do**
      $m_{i+1}.w = m_{i+1}.w + p_o m_i.w(i/l)$
      $m_i.w = p_z m_i.w[(l - i)/l]$
    end for
    return m
  end procedure
  procedure UNWIND(m, i)
    $l = len(m)$
    $n = m_l.w$
    $m = copy(m_{1 \ldots l-1})$
    for $j \leftarrow l - 1$ to 1 **do**
      if $m_i.o \neq 0$ **then**
        $t = m_j.w$
        $m_j.w = n \cdot l/(j \cdot m_i.o)$
        $n = t - m_j.w \cdot m_i.z((l - j)/l)$
      else
        $m_j.w = (m_j.w \cdot l)/(m_i.z(l - j))$
      end if
    end for
    for $j \leftarrow i$ to $l - 1$ **do**
      $m_j.(d, z, o) = m_{j+1}.(d, z, o)$
    end for
    return m
  end procedure
  RECURSE(1, [], 1, 1, 0)
  return ϕ
end procedure

Figure 73: Full SHAP algorithm [19]

9.9.3 SHAP Interaction Values

SHAP interaction values are a way to quantify the change in Shapley values when we have joint predictors. The *SHAP interaction values* can capture the interaction effects between two variables, and how they change the Shapley value. When using *SHAP dependence plots* in python, we are using SHAP interactions values.

The SHAP interaction value is calculated as follows [19]:

$$\Phi_{i,j} = \sum_{S \subseteq N \ \{i,j\}} \frac{|S|!(M - |S| - 2)!}{2(M-1)!} \nabla_{ij}(S) \tag{82}$$

where $i \neq j$, and

$$\nabla_{ij}(S) = f_x(S \cup \{i,j\}) - f_x(S \cup \{i\}) - f_x(S \cup \{j\}) + f_x(S) \tag{83}$$

$$= f_x(S \cup \{i,j\}) - f_x(S \cup \{i\}) - [f_x(S \cup \{i\}) - f_x(S)] \quad \text{Non-trivial result} \tag{84}$$

Let's interpret Equations (82)- (84). $\Phi_{i,j}$ is the *SHAP interaction value* between features i and j.

Once again:

1. N is the set of all input features
2. S is a subset of N
3. M is the number of input features
4. $f_x(S) := E[f(x)|x_S]$

Intuitively, we can break the expression in Equation (84) into two parts. These are $f_x(S \cup \{i,j\}) - f_x(S \cup \{i\})$, and $f_x(S \cup \{i\}) - f_x(S)$.

$f_x(S \cup \{i,j\}) - f_x(S \cup \{i\})$ intuitively is the value of a coalition $S \cup \{i,j\}$, minus the value of a coalition $S \cup \{i\}$. So $f_x(S \cup \{i,j\}) - f_x(S \cup \{i\})$ measures the marginal contribution of player j joining a coalition.

Likewise, we can see that $f_x(S \cup \{i\}) - f_x(S)$ measures the marginal contribution of player i joining a coalition.