Damage Rate Forecasting

Capstone Project

Dane Anderson

January 17, 2019

I. Definition

Project Overview

(All Numbers mentioned in this section are fictional and are provided purely for demonstration purposes, they hold no value and should only be regarded as hypothetical data for the purposes of explaining the importance)

Businesses work best when there's a plan in place. Without a plan, it can incur costs that are unnecessary and possibly avoidable. For example, if a company lends out assets to other businesses, these businesses are required to return these assets back to their original owners. When it comes to the biggest pieces of the financial analysis for a company that pools their assets, the rate at which their assets come back broken, the cost to make new ones, and the rate in which they are being sent and returned are of the utmost importance. Once the finance team determines this information, they can begin their own proposals to plan the allocation of money to these tasks for the year. In the supply chain industry, these assets don't always make their way back to their original owners in the best conditions. Sometimes they are broken and need repairing. If you are part of a business where all they do is pool assets across the supply chain around the world, a major part of your cost analysis is how much your assets get damaged and need to be repaired.

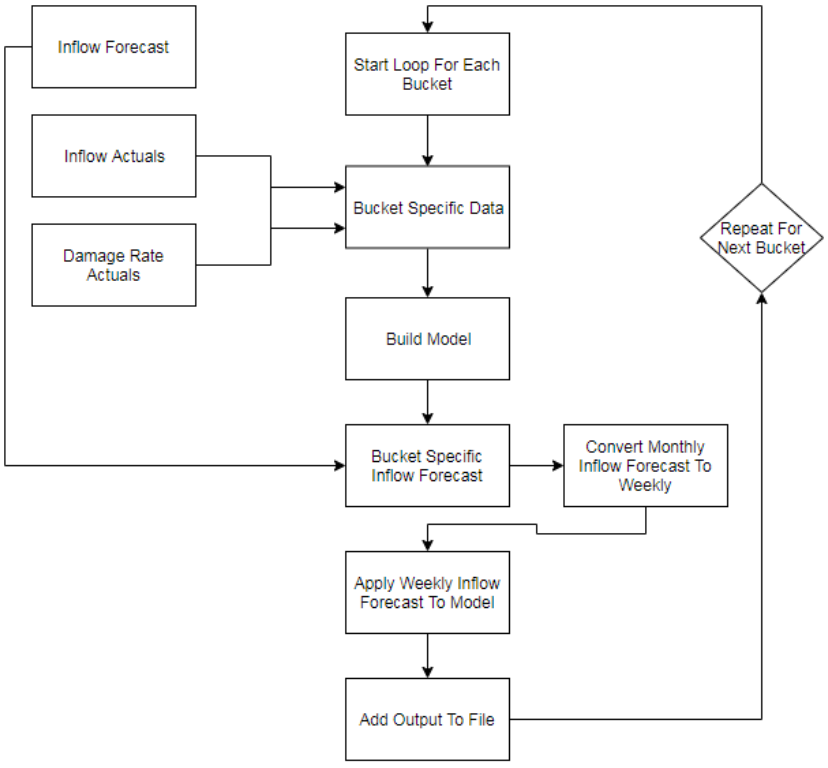
Project Overview

Mathematically, the damage rate equation is very simple. It is as follows: assets damaged / all assets returned. The number that represents volume of assets returning to the company is called 'Inflows' and will be represented as such throughout the report. The full equation used to plan expected costs regarding repairs per asset returned is: (damaged/damaged+working)*repaircost. This simple number tells everyone that for every 100 assets that are returned, X amount are needing to be repaired. There is also a consistent cost to repair these assets, for the sake of ambiguity and for this capstone project, I will put that consistent price to be $4. In turn, if the damage rate for the entire business for this week is .4, then you multiply .4 by 4, which ends up being $1.60. We now know that if we allocate $1.60 towards every expected asset that is returned to us for the month, we should have a strong idea how much money we should set aside for repairs. These repair costs include materials,

labor, and shipping. All three are fairly easily obtained if there is a proper amount of time coordinated in advance. Otherwise, this process can be very costly, mostly because labor last minute is a premium, shipping large amounts of assets last minute is a premium, and sometimes materials are in deficit which, due to supply and demand elasticity, price goes up. A cost to repair an asset goes from $4 if planned properly and goes up to $8 if not. If a network has 30 million assets returning in a given month and a damage rate of .4 is planned, but a damage rate of .6 (12 million) actually happens, the company now has to spend .2 (6 million) of the damage rate at $8 per asset. $48M is paid out with a .4 rate and $48M is paid from the .2 rate due to the premium. The final 1/3rd of the repairs cost the same as the first 2/3rd. Having this kind of spread is akin to flipping a coin every month based on the previous months. If we have an accurate forecast, this can save a considerable amount of money for the organization that would allow them to place it in other places like employee bonuses.

In order to get an accurate plan for damage rate, you must have a forecast. There are many ways to get there though. You could always do an ARIMA forecast model but those are univariate. Multivariate forecast models are proven to have consistent positive results, especially when using a machine learning approach. The best regression technique so far is the XGBoost model. Extreme Gradient Boosting Regression is good because it is not sensitive to noise or outliers and can pick up patterns and make a great output with a small amount of data engineering or hyperparameter tuning.

The map for completing this project is as follows:



Metrics

I will use error bias and accuracy as an evaluation metric. This can tell us how far off from reality the forecast was. This should be a satisfactory measurement to determine model success. In the business, if the model reaches a 95% accuracy, then it is considered satisfactory, but of course, it's always suggested to strive to perfection. Another possible way to calculate an evaluation metric is how far the output deviates from the mean per time period. If a non-programmer, non-statistician attempted to make a forecast, they would just create some sort of rolling 4 week mean and apply it to each further week. We will compare these. In response to feedback, f1 score was no chosen because f1 score is built for classification accuracy rather than regression accuracy. Expounding upon accuracy, it takes each time step and obtains the absolute value delta of reality and the forecast, then divides by reality to get a percentage score. This score can show how close the model got to reality for each timestep. Error bias is similar. It takes the delta for each timestep and calculates the mean of the deltas.

II. Analysis

Data Exploration

The data used for this project are split into five files. Forecast of inflows, the actual damage rate from the past (split between each bucket and the entire network), the actual inflow volume from the past, and the calendar lookup table. For further clarification, CalendarDay in Damage Rate & Inflows Actuals tables is the first day of the week (Sunday) and Date in the calendar table is the same, first day of the week (Sunday). CM is Calendar Month written out with the first 3 letters of the word, CW is calendar week, CY is calendar year, CM2 is numerical notation of the calendar month.

An issue came up when extracting the inflow forecast from the source, it didn't have a linked data field, just calendar month and another notation for year, FY or Fiscal Year. This caused a lot of headache when trying to use python to fix this rather than fixing it at the source since we don't know who else uses the source and It'd be selfish to me to change it for my own needs if someone else already uses it the way it is.

Exploratory Visualization

Below is a look at what each table looks like and how it connects to each other table. In the code, CalendarDay is converted to Date, then turned into a datetime object then placed into the index for quick comparisons, joining etc. Calendar is used to connect Inflow Actuals to inflow forecast using CalendarDay in Inflows Actuals linked to Date in Calendar to extract the year and month from the calendar to derive a date into the Inflows Forecast table. After that, everything becomes related to each other.

Damage Rate Actuals

Index	CalendarDay	Bucket	DamageRate
0	7/11/2013	1	0.7349
1	1/23/2016	1	0.8749
2	1/22/2016	1	0.8763
3	1/21/2016	1	0.8446
4	7/10/2013	1	0.7348
5	1/15/2016	1	0.8342
6	1/24/2016	1	0.8520

Inflows Actuals

Index	CalendarDay	Bucket	Inflows
0	8/23/2017	1	213227
1	2/23/2017	1	142148
2	7/26/2017	1	217904
3	2/24/2017	1	151656
4	1/19/2018	1	242536
5	5/29/2018	1	211245

Inflow Forecast

Index	FY	CM	Bucket	Fcst
0	FY19	Jan	2	19171096.0000
1	FY19	Jan	4	499448.0000
2	FY19	Jan	1	6469085.0000
3	FY19	Jan	3	3724686.0000
4	FY19	Feb	2	15173932.0000
5	FY19	Feb	4	479276.0000
6	FY19	Feb	1	4133146.0000

Calendar

Index	Date	CM	CW	CY	CM2
2022-12-25 00:00:00	2022-12-25 00:00:00	Dec	53	2022	12
2022-12-18 00:00:00	2022-12-18 00:00:00	Dec	52	2022	12
2022-12-11 00:00:00	2022-12-11 00:00:00	Dec	51	2022	12
2022-12-04 00:00:00	2022-12-04 00:00:00	Dec	50	2022	12
2022-11-27 00:00:00	2022-11-27 00:00:00	Dec	49	2022	12
2022-11-20 00:00:00	2022-11-20 00:00:00	Nov	48	2022	11
2022-11-13 00:00:00	2022-11-13 00:00:00	Nov	47	2022	11

Average Damage Rate Per Week

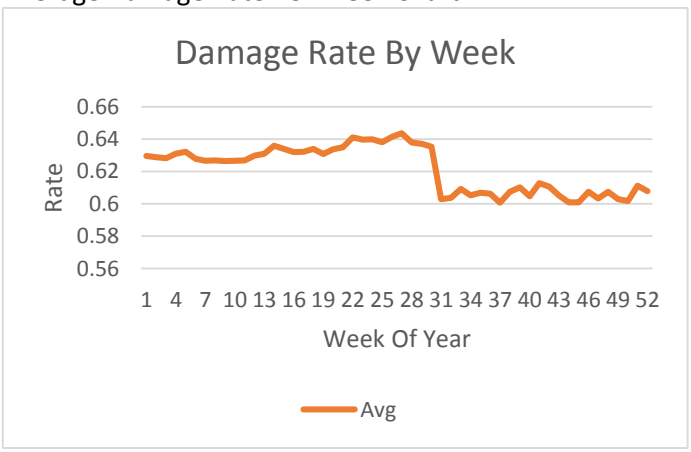
Week	Avg	Week	Avg
1	0.629615	27	0.643611
2	0.628911	28	0.637854
3	0.628141	29	0.63703
4	0.630918	30	0.63533
5	0.632125	31	0.602979
6	0.627905	32	0.603751
7	0.626633	33	0.609149
8	0.626889	34	0.605302
9	0.626447	35	0.606964
10	0.626647	36	0.606326
11	0.626757	37	0.600813
12	0.629771	38	0.607543
13	0.630963	39	0.610316
14	0.636007	40	0.604724
15	0.633906	41	0.61287
16	0.631926	42	0.610541
17	0.632093	43	0.605056
18	0.634052	44	0.600944
19	0.630711	45	0.600957
20	0.633739	46	0.60754
21	0.634911	47	0.603324
22	0.641074	48	0.607492
23	0.639779	49	0.603
24	0.639932	50	0.601629
25	0.638116	51	0.611228
26	0.641484	52	0.607933

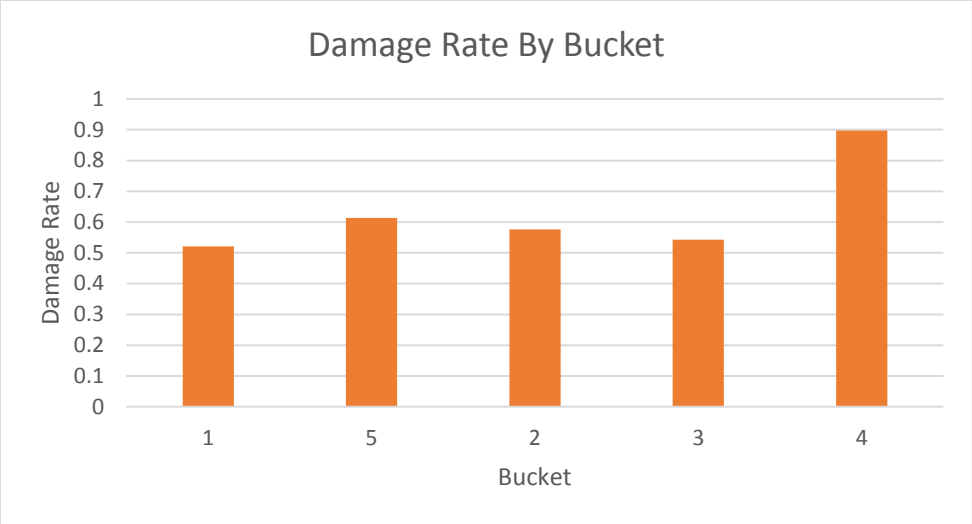
Average Damage Rate Per Bucket

Bucket	Avg
1	0.52068
5	0.613249
2	0.575594
3	0.542377
4	0.897224

- **Kurtosis**
- By Week : -1.602
- By bucket : 3.97
- By year: 3.13
- **Average : 62.21%**
- **Min : 60%**
- **Max : 64.31%**
- **Median: 62.68%**

Average Damage Rate Per Week Chart





Algorithms and Techniques

When it comes to determining what strategies to employ, the solution is rather apparent for those who have used regression algorithms before. For other projects done in my field of work, there have been many various strategies attempted to create a forecast. Some of those attempts include using MatLab, a Recurrent Neural Network, or even some standard regression models like Croston-Fourier, AVS-Graves, or even a Moving Average. MatLab is too expensive to use in an enterprise solution to just pick up without corporate approval, recurrent neural networks are too sensitive to seasonality, outliers and require an enhanced form of stationarity which in turn removes a lot of the patterns from the datasets, thus causing the output of an RNN to be flat and only deviate from this mean by decimals. Standard regression models are good for quick and small tasks that don't require a lot of variation between each time period and sometimes can skew in a way that is not wanted. After many months of testing different models and solutions, the best option ended up being DMLC's XGBoost regression package. It's easy to use out of the box, we used grid search to easily obtain optimal hyperparameters and create quick regression forecasts for nearly anything.

When implementing the code, I ran the XGBoost algorithm with the regression package. I then built a library of parameters to use in the gridsearch function. For N Estimators which is the number of rounds applied to the algorithm, I tried 5, 10, 50, and 100. 10 ended up being the best for this. For Learning rate, which is the controller for determining the weight of all the new trees made. I tested .1, .5, and 1 and .5 ended up being the optimal learning rate. Lastly, I tested Max Depth, that's the limit of how big a tree can get inside the algorithm (or how complex the pattern recognition needed to be). After testing 1, 5, 10, 20, 30, and 40, 20 ended up being the optimal number. After it ran, -.003 accuracy was the best it could achieve using the optimal parameters. Explicitly, boosting is when you use an ensemble of weaker prediction models like decision trees and then marry them together to obtain a more robust outcome. In

other words, it makes new models over and over with each iteration modifying the last to correct the errors made by the previous model. XGBoost comes into play when we want to apply these models in a C background instead of python that talks directly on a hardware level to optimize performance. It's great because it can weigh data points that are hard to predict.

Benchmark

When handling a regression model, having a benchmark not obtainable in the real world. For example, you can't have an answer key if you are forecasting data 5 months in the future. Since this is the case, I am using SciKitLearn's train-test-split function to gather the best benchmark available.

Bucket 1

y_pred - NumPy array

	0
0	0.702481
1	0.820294
2	0.714241
3	0.906388
4	0.826731
5	0.69354
6	0.601112
7	0.831438
8	0.494216
9	0.734017
10	0.709239
11	0.669385

y_test - NumPy array

	0
0	0.522243
1	0.59629
2	0.638567
3	0.603144
4	0.640172
5	0.718866
6	0.660697
7	0.660187
8	0.606864
9	0.657249
10	0.60191
11	0.651393

Bucket 2

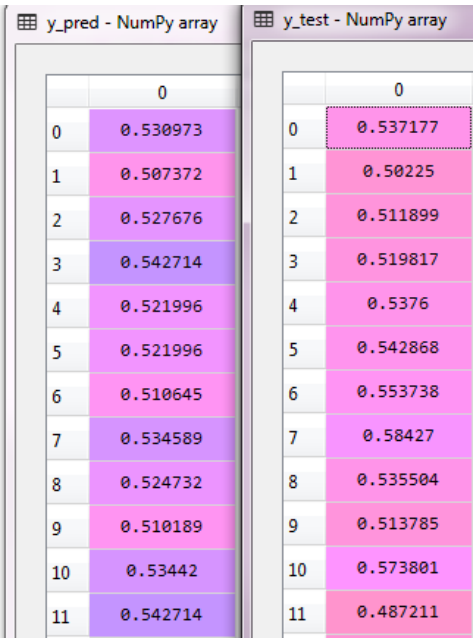
y_pred - NumPy array

	0
0	0.565014
1	0.584279
2	0.601165
3	0.604864
4	0.601165
5	0.584279
6	0.603516
7	0.603516
8	0.604864
9	0.604864
10	0.604864
11	0.585627

y_test - NumPy array

	0
0	0.559228
1	0.58913
2	0.570499
3	0.579303
4	0.572494
5	0.588395
6	0.573946
7	0.558619
8	0.562106
9	0.569387
10	0.549316
11	0.598786

Bucket 3



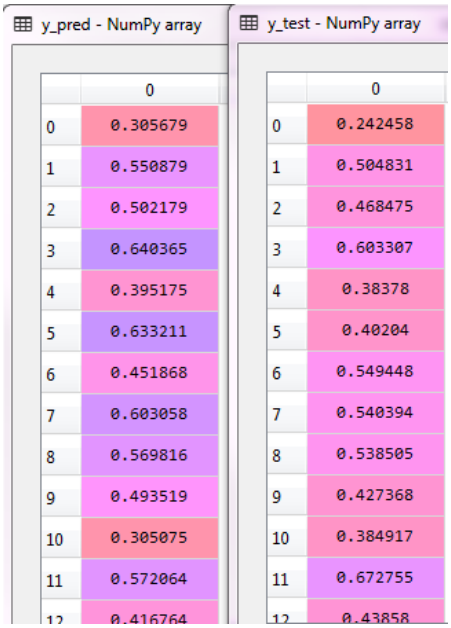
y_pred - NumPy array

	0
0	0.530973
1	0.507372
2	0.527676
3	0.542714
4	0.521996
5	0.521996
6	0.510645
7	0.534589
8	0.524732
9	0.510189
10	0.53442
11	0.542714

y_test - NumPy array

	0
0	0.537177
1	0.50225
2	0.511899
3	0.519817
4	0.5376
5	0.542868
6	0.553738
7	0.58427
8	0.535504
9	0.513785
10	0.573801
11	0.487211

Bucket 4



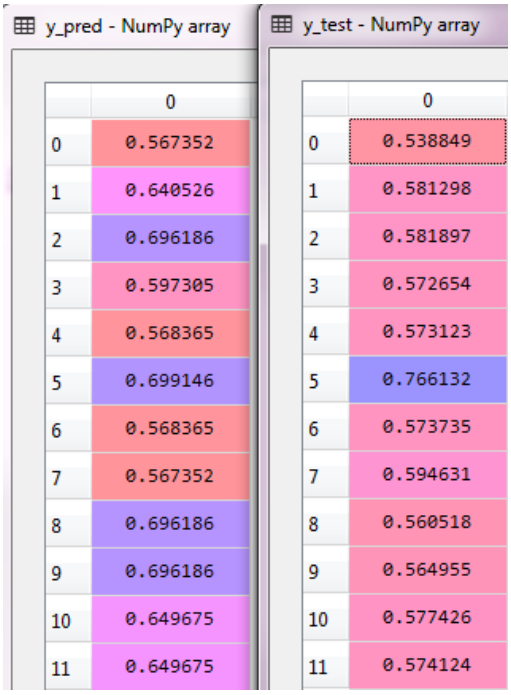
y_pred - NumPy array

	0
0	0.305679
1	0.550879
2	0.502179
3	0.640365
4	0.395175
5	0.633211
6	0.451868
7	0.603058
8	0.569816
9	0.493519
10	0.305075
11	0.572064
12	0.416764

y_test - NumPy array

	0
0	0.242458
1	0.504831
2	0.468475
3	0.603307
4	0.38378
5	0.40204
6	0.549448
7	0.540394
8	0.538505
9	0.427368
10	0.384917
11	0.672755
12	0.43858

Bucket 5



y_pred - NumPy array

	0
0	0.567352
1	0.640526
2	0.696186
3	0.597305
4	0.568365
5	0.699146
6	0.568365
7	0.567352
8	0.696186
9	0.696186
10	0.649675
11	0.649675

y_test - NumPy array

	0
0	0.538849
1	0.581298
2	0.581897
3	0.572654
4	0.573123
5	0.766132
6	0.573735
7	0.594631
8	0.560518
9	0.564955
10	0.577426
11	0.574124

As you might have noticed, Bucket 1 is a bit of a troublemaker. The average delta between each week represented is around .2 rather than .08 in the others. Understanding why is knowledge obtained by knowing who the bucket is and how they manage they manage our assets. When investigating into their weekly historical volume, you would see that bucket 1 has a highly irregular trend. You would also notice that bucket 1 has a higher damage rate than the

others. This is another business knowledge case where this bucket is known to re-use our assets until they are on the verge of breaking, or broken.

III. Methodology

Data Preprocessing

Most of my code is preprocessing the data. We all know that in the enterprise environment, source data is not always in the same format you need it in. Therefore, we need to do a lot of manipulation of the data. Out of the 250 lines of code, close to 230 of it is data preprocessing. In order to explain the preprocessing steps completed, I will reformat the explanation into outline form.

- Ingest Damage Rate Actuals
    - Converts the date field to a datetime object
        - Set is as index
    - Adds week and year from the daily calendar version
    - Reorders the columns
    - Renames the columns
    - Changes the rate field to float
    - Concatenates damage rate actuals network and damage rate actuals
    - Reorders the columns
- Ingest Damage Rate Actuals Network
    - Converts the date field to a datetime object
        - Set is as index
    - Adds week and year from the daily calendar version
    - Reorders the columns
    - Renames the columns
    - Changes the rate field to float
    - Concatenates damage rate actuals network and damage rate actuals
    - Reorders the columns
- Ingest Inflows Actuals
    - Reorder columns
    - Turn date field into datetime object
        - Set it as index
    - Create second inflows table for the entire network
        - Group the original table by the index and sum the inflows keeping the first date to make an aggregation of all the buckets.
        - Add the Week month and year fields from the calendar table
        - Drop the date field since it's in the index now
    - Add the week month and year fields from the calendar table

    - Drop the date field since it's in the index now
    - Concatenate the network table to the original and reorder the columns since concatenation messes with the order
- Ingest Inflows Monthly Forecast
    - Adding the year to 2019 since we know this will never change unless we get a new forecast table and the data expands past 2019. The business need only wanted half the current calendar year
    - The fcst column which is the actual forecast number is another datatype, converting it to int using 'astype'
    - Using merge to add all the monthly calendar dataset columns to the table
    - Setting the date field that was added from the calendar as the index and sorts it descending
    - Reorders the columns
    - Creates the network version
        - Group by index just like the actuals table
        - Adds the name for the bucket, 5
        - Concatenates the network to the original table
        - Reorders the columns
- Convert Inflows Monthly Forecast Data Into Weekly
    - Extract monthly forecast per bucket to iterate each one
    - Drop the bucket field
    - Resample into weekly and forwardfill everything
    - Manually make the final week since forwardfilling doesn't do it automatically
        - Extract the final week from the resampled table
        - Drop the final week row from the newly made weekly forecast table
        - Assign the final week date field to the index
        - Uses the index and adds 6 rows with the new 6 days to make the whole week
        - Adds it to the newly made weekly forecast from monthly
    - Manually make the list of week numbers used in the forecast table since forward filling doesn't do it right
    - Apply this list to the forecast
    - Reorders the columns
    - Renames the columns
    - Create the weekly ratios to apply it to the monthly forecast to get the weekly forecast
        - Starts a loop by iterating through each year
            - Makes a list of the month numbers for this year
            - Starts a loop by iterating through each month
                - Extracts the actual data for this month and year

        - Resets the index
        - Groups by week to get a sum of each week from the daily inflow actuals
        - Takes the sum of the entire month and stores it as a variable
        - Creates a ratio for the month by dividing each week of the month by the total of that month variable made in the above step
        - Appends those numbers to the main table
        - Fills the NA values (weekends or holidays) with zero
        - Iterates through each month for each year to obtain all the ratios
    - Drops the ratios that are larger than 32% since the only way to get 33% is by having only 3 days which is not possible in a full week, so this actively removes the data with incomplete weeks in the first and last week of the dataset
    - Groups by week and calculates the average ratio to create a table of week and ratio
  - Links the table mentioned in the above step to the first monthly to weekly table full of the same data
  - Multiplies that ratio for the week number of the year to create the weekly forecast from monthly
- Ingest Calendar
  - Rename and reorganize the table
  - Turn the date field into a datetime object
    - set it as the index
  - Create a second calendar from the original and turn it into a daily calendar
    - Resample by day and forwardfill the null values
    - Assign the date field to the index since resampling messes the original up
  - Create a third calendar from the original and turn it into a monthly calendar
    - Sort the index
    - Group it by Month and Year taking the first entry of each other column
    - Set the index to date column since grouping it messes up the original
    - Remove all the other columns except Date, Month and Year

Now that we have all of the data preprocessed, we are ready to build the model and apply it to the newly made weekly inflows forecast.

Implementation

Using the cleaned damage rate actuals table, we strip the labels and train-test-split it by 20% test and 80% train. We then apply the XGBRegressor object from XGBoost package. We then apply the built parameters library into the gridsearch and let it output the best parameters. Use those parameters in the regressor object and fit it to the train and test set that the train-test-split made. We then extract the prediction with the XTest object using the predict method on the regressor.

Finally, we take the fitted regressor object and apply weekly inflows forecast table with calendar data. This outputs the damage rate forecast. We then do a few minor data processing touches like change data types, drop tables, reassign buckets, re-add calendar field date. Lastly, we repeat the process for each bucket and append them to each other and output the file as a CSV to be used by the dashboarding tool of choice.

Refinement

For refinement, using gridsearch allowed me to automate the guesswork on finding the best parameters. Throughout testing, I did adjust the gridsearch parameters library if the chosen parameter was either the largest or smallest of the choices. The initial strategy of using XGBoost worked well and didn't further exploration of different models.

IV. Results

Model Evaluation and Validation

The evaluation of the model has been evaluated and testing of other models were mentioned previously in the report. The parameters of the model have been represented and explained in the methodology section. To evaluate how close the forecast aligns with reality, you can see it in the image below:

FY	FW	ML Fcst	S&OP Fcst	Actuals	ML Vs S&OP	ML Vs Act	S&OP Vs Act	R3WK Acc Avg
2019	1	90.7021%	89.9910%	91.1972%	99.2160%	99.4571%	98.6774%	
2019	2	90.6937%	89.8900%	92.0366%	99.1138%	98.5409%	97.6677%	
2019	3	90.5669%	91.3040%	90.4767%	99.1927%	99.9004%	99.0939%	99.2995%
2019	4	90.3139%	90.2940%	89.6978%	99.9780%	99.3178%	99.3397%	99.2531%
2019	5	90.2201%	91.5060%	91.1573%	98.5947%	98.9719%	99.6189%	99.3967%
2019	6	90.0654%	91.4050%	91.9192%	98.5344%	97.9833%	99.4406%	98.7577%
2019	7	90.1685%	89.9910%	92.4196%	99.8031%	97.5643%	97.3722%	98.1732%
2019	8	90.1685%	90.1930%	92.7153%	99.9728%	97.2531%	97.2795%	97.6002%
2019	9	90.1769%	91.7080%	91.6690%	98.3305%	98.3723%	99.9574%	97.7299%
2019	10	89.8071%	91.9100%	91.9833%	97.7120%	97.6342%	99.9204%	97.7532%
2019	11	89.8071%	91.8090%	90.4921%	97.8195%	99.2430%	98.5657%	98.4165%
2019	12	89.5029%	93.1220%	90.2282%	96.1136%	99.1961%	96.8925%	98.6911%
2019	13	89.4809%	92.0110%	89.8259%	97.2502%	99.6160%	97.6251%	99.3517%
2019	14	89.4119%	92.2130%	90.3785%	96.9624%	98.9305%	98.0106%	99.2475%
2019	15	90.4404%	92.7180%	90.2399%	97.5435%	99.7783%	97.3272%	99.4416%
2019	16	90.6058%	91.6070%	88.8203%	98.9071%	98.0293%	96.9579%	98.9127%
2019	17	89.7026%	92.4150%	90.8460%	97.0650%	98.7414%	98.3022%	98.8497%

The S&OP Fcst column numbers are the numbers that another department currently uses to forecast the damage rate. They do the entire process manually and spend about 4 hours per month gaining a consensus on it, whereas the machine learning approach is done automatically. You can see that the accuracy between ML and S&OP shows that the machine learning approach is more accurate. The data begins July 1 2018 which is FY 2019 FW1,

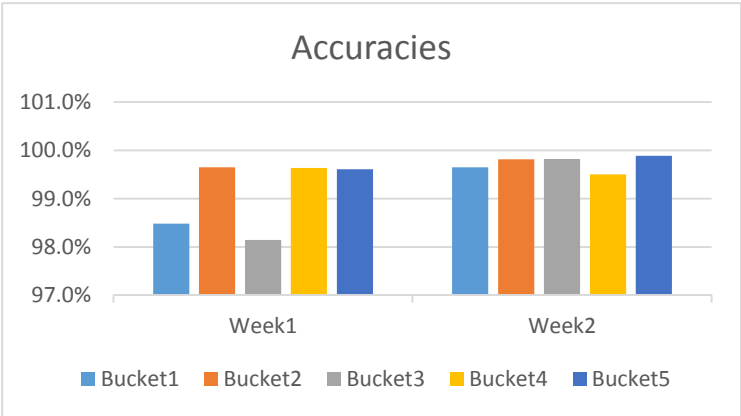
Justification

There is reason to believe that the machine learning model can be trusted in terms of accuracy. It has also been reviewed by managers and the vice president of the division and many are impressed and excited with the accuracy and all say that it solves the problem.

V. Conclusion

Free-Form Visualization

To the right is a graph that shows the last 2 weeks' accuracy per bucket. There isn't much of an insight to give about it, other than how close to 100% it is. This is mostly just used as a supplement to understand the breakdown in the distribution of accuracy.



Reflection

Businesses rely on having plans set in place to determine how much money is required to carry out tasks months, even a year in advance. In respects to damage rate, finance requires at least 6 months of forward looking numbers to plan the amount of trucks, assets etc. at the cheapest rate. When the rate of damaged assets exceeds beyond what is planned, costs can increase at an alarming rate. Currently, the business relies on manually calculating these rates by hours of consensus meetings, with this new model, they could no longer need these meetings and can save time doing other tasks. In reflection of the entire process, one of the hardest parts was dealing with data that is not consistent across the environment. The forecast data for inflows, for example, was formatted as monthly yet I needed it as weekly. Devising a plan to extrapolate these values took a considerable amount of time and thought. Eventually, it was agreed upon that the best route was to gather a generalized week-of-month ratio since most of the supply chain is seasonal. All-in-all, the final model satisfied expectations with both myself and leadership in my company when proposed.

Improvement

The old saying ‘there’s more than one way to skin a cat’ holds true to almost any aspect of programming. It especially holds true to this project. The biggest improvement that could be made is re-organize which data gets forecasted. Instead of forecasting the rate of damage of the asset, why not forecast the number of assets that get damaged? Then apply the weekly inflow forecast estimate derived from the monthly number. This could possibly obtain a different number, even a more accurate one. It could obtain a more accurate one possibly because of the math used to get to the rate of damage (damaged / inflows). If we apply that formula to the forecasted number, it’s possible to obtain a different outcome. When it comes to other algorithms used for the model, it’s entirely possible to explore image recognition model building. We could try to use the image recognition model to study the graphs of the historical rate of damage and then use an RNN on top of the CNN to come up with some numbers that match the flow of the charts originally fed into the CNN. Currently, without further extensive testing, I feel like my final solution is the new benchmark and is the best-in-class breed for solving the problem.