8-2022

Predicting Order Status using XGBoost

Kegan J. Penovich
Utah State University

Invista
Wichita, Kansas, US

Pre-emptively categorizing an order as on-time, early, or late

Using an XGBoost algorithm to predict whether a placed order will arrive early, on-time, or late to Invista.

Technical Report (Plan B)
May 2021

Kegan J. Penovich
Industrial and Applied Mathematics Masters Program
Department of Mathematics and Statistics
Utah State University

Table of Contents

Abstract

Pre-emptively categorizing an order as on-time, early, or late: Using an XGBoost algorithm to predict whether a placed order will arrive early, on-time, or late to Invista.

By

Kegan J Penovich

Utah State University, 2021

Logan, Utah

Major Professor: Dr. Joe Koebbe

Department of Mathematics and Statistics

Invista, a Koch subsidiary, is multinational producer of fibers, resins, and intermediaries, particularly nylon. To keep the company operating required them to take over 1.5 million orders over the course of $2 - \frac{1}{2}$ years, less than a third of which arrived on-time. Orders arriving other than when expected can cause many problems for any company. While arriving late is a clear problem, it also troublesome for them to arrive early. In the face of this it becomes important to be able to tell a-priori if an order will arrive on-time or not.

To address this problem, we made use of those 1.5 million orders to try and learn how to predict if an order would be on-time or not. There are many methods for doing so and we tried three approaches: Neural-Networks, Gradient Boosting, and Time series. In the end we found the Gradient Boosting algorithm worked the best. We utilized the popular XGBoost framework of Gradient boosting. This was made further appealing by the company having utilized this algorithm before.

Acknowledgements

Thanks to my committee, Joseph Koebbe, Tyler Brough, and Kevin Moon, the company who hired me for the internship Invista, and Andrew Brim for advising the internship team.

1. Introduction

Invista is a fiber, resin, and intermediates company centered in Wichita Kansas, though they operate many facilities in the United States and abroad. Due to its size, Invista requires hundreds of orders each day for products ranging from mechanical parts to chemical agents to building materials. An unfortunate reality of ordering in such volume and frequency is that some of those orders will not arrive when planned. This inevitably will cause delays to propagate through the production process resulting in loss of time and capital.

While some level of late orders is to be expected and can be dealt with, widespread tardiness can become a source of continuing delays and loss in production. Dealing with late and early orders becomes a necessary concern once operations reach a certain size, and core to dealing with late and early arrivals is gaining some insight into what causes an order to be late or early. Time working with vendors and gaining domain expertise can be valuable tools in addressing why an order might me early or late. However, when a company has millions of detailed and recorded orders, more sophisticated methods must be used to analyze the process.

Invista wished to try these methods to determine how likely an order would be early, late, or on-time. To accomplish this Invista reached out to Utah State University's Analytics Solution Center. The Analytics Solution Center (ASC) is an organization where students at USU are given the opportunity to work with professors and public companies on real-world data intensive projects. The companies gain a solution to a real world problem at a fraction of the cost of a professional consultant, while students gain structured but real-world experience.

The goal of the project was to provide Invista with a model that could take a finished order as input, and then provide as output a classification of on-time, early or late. Our model needed to correctly classify an order as early, late, or on-time correctly less than 90 percent of the time. This rate was a company threshold number to prevent overfitting. This made the goal to obtain a rate between 80 and 85 percent on test data sets for the project.

This requirement was one that was strange to our team as we have often been trained to seek for as close to perfect accuracy as possible. Invista's policy to have a maximum trusted accuracy of 90% ran contrary to perfect accuracy. We had this explained to us that while accuracy over 90% might sound great, Invista had found in their experience such high scores were often the result of over fitting. This justified Invista's confidence in lower accuracy models to be more honest in their limited predictability than highly accurate models. As a

practical note, such high accuracy is challenging to achieve so it was not as limiting as one might expect.

Code developed and tested in the project was required to be usable within the company's code pipeline. The choice for this project was Amazon Web Services (AWS). This was largely a non-issue since the code we wrote used common machine learning libraries available in Python. Implementation into Invista's pipeline was done during the last few days of the internship.

Throughout the internship the team that I was a part of was required to give weekly updates to Invista staff. This staff primarily included people responsible for tracking and recording orders for Invista. The meetings were designed to inform them of interesting results we had found, problems we were encountering, and as an accountability measure. These meetings were to be fifteen minutes with time at the end for questions that the Invista employees had for us.

The analysis and the model fitting were all done in Python using AWS as the computational environment and data repository, and Git as version control. AWS is an on-demand cloud computing platform. Beyond cloud computing we also used AWS cloud storage for our data. Invista used Amazon Lambda, a service providing serverless functions, to deploy our final model. Our involvement in this final part was minimal, though we were walked through how that deployment would go.

The project was split into five parts over a 12-week internship: 1. Data Exploration, 2. Feature Engineering, 3. Model selection, 4. Feature selection, and 5. Validation. The bulk of the time was spent on the first three parts with only 4 weeks spent on the latter two parts. We will spend a section of the paper going over each of these, while a short description to each will be given below.

1. Data Exploration

In section 2 we go over the data that was given to us by Invista. We explain how we had to clean and prepare the data. The main issues we faced were due to the data having been produced over the course of several years. Differing standards of data entry, changing suppliers, and company growth all caused challenges that had to be addressed.

2. Feature Engineering

In section 3 we then go on to show new variables that we created to try and pull more information from the data. There were also attempts to find trends and more useful relationships though not from any machine learning

model. We put this section here to aid in the flow of reading; however, we were always on the lookout for new features or variables to test.

3. Model Selection and the Extreme Gradiant Boosting (XGBoost) Model

In section 3 we go over our process for choosing a model to try to categorize orders as late early or on-time. Two models were tried: a simple Feed Forward Neural Network and the XGBoost model. In the end the XGBoost model was chosen as it was the most accurate in our initial tests and since the model had been used in other Invista projects. We will explain the Xgboost model, how it works, and why it is a good fit for our situation.

4. Feature Selection

After our model was selected, we then went on to decide which variables were the most important to keep. The model proved robust in that its performance was acceptable with very few variables. The model was time consuming to fit so the number of variables tested was limited though not to any serious degree.

5. Validation

Once the final model was finalized, we had to validate the model to show that it was not suffering from overfitting. While there are several ways to address overfitting the time series nature of the data led us to leaving out a validation set. We left out the last few months of data so that 20% of the data was unused during the modeling process.

6. Results

In section 4 we will go over the results of the model. We were able to attain model accuracy of 75 percent using XGBoost. We will discuss which variables were chosen for the final fit, and more about how the model performed while determining which of the categories—late, early, or on-time—the model was predicting.

2. Data

The data we received included all received orders that Invista had received from January 1, 2017 till March 1, 2020. This totaled 1467858 orders received each of these then had an additional 24 features that could be investigated. Among these were company ID's, materials ordered, company location, and most importantly a date of expected arrival and date of actual arrival. From this we were able to determine if an order was on-time, early, or late. The only issue

we ran into was that about a third of the dates were missing a received date so those had to be thrown out of consideration.

A common problem we faced with this dataset involved many missing values for several of the features. We were lucky in that most of these were not considered for ultimate use in the model, but we were forced to remove many of the observations ultimately resulting in a cleaned data set with around 900,000 observation.

Our first action on the clean dataset was to see what the breakdown of late/on-time/early orders was. Our assumption was that most orders would be late then on-time with the fewest being early. This turned out to be incorrect. Most orders were late, but then the second most were early with on-time orders being a surprisingly small number of the totals. Across the whole data set only 14.5% of orders were on-time. Over time this did increase, but this was mostly from a drop in total orders not an increase in on-time orders.

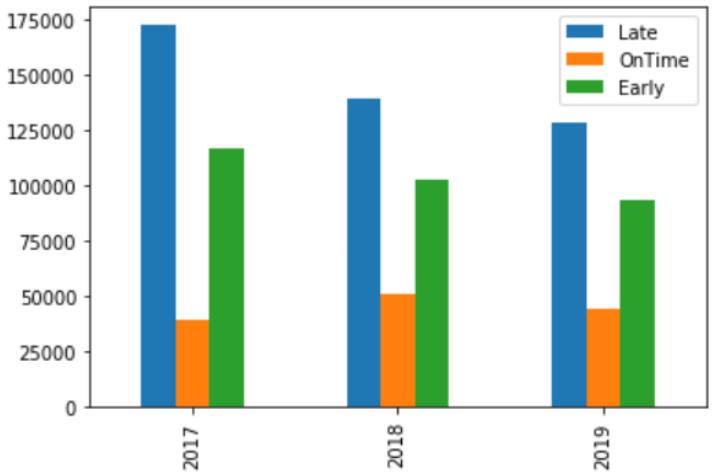

Figure 1 Counts by year of Late, Early, and On-time Orders

This tells us that our target variable is not uniformly distributed. We did not take the possible ramification of this into account which lead to some unutilized information that might have been helpful. This omission and its consequences will be discussed more in section 4.

It was to these cleaned observations that we looked to in our first efforts to understand the problem. Our efforts largely broke down into looking at late/on-time/early distinction order type, who they were ordered from, and general

trends through time. This was done to orient ourselves but was also helpful as an eye forward to later feature engineering.

Order Type	% of Total Orders
X0:	43.4971%
ND:	27.1026%
Y0:	12.7689%
ZB:	9.6199%
Z0:	4.4474%
VB:	1.1638%
PD:	0.009926
V1:	0.003939
RE:	0.000132
V2:	0.000003
RP:	0.000002

Table 1 Percentage of Total Orders by Order Type

Order type is a designation that tells us why the order was made. There are many such distinctions which include orders that are made when predetermined level in inventory is reached, ordered as needed, and other more nuanced situations. As this knowledge is proprietary to the company, I will not explain the labels. As we looked at the order type, we first decided to look at them as independent of their arrival status. This gave us an idea of how to weight these order tpes when we investigated their arrival status. In Figure 1 below we see table that shows the percentage of order type. We see that nearly 93% of orders come from just four of these order types while we can get to 98% with just six. This let us know that a small subset of the order types made up the majority of total orders.

We then wanted to look at the breakdown of arrival status orders by order type to see if the break down followed the same pattern as order type irrespective of arrival status. If this is not the case it would suggest the order type might have predictive power. Arranging them in descending order we can see that these do not exhibit a uniform distribution, particularly across the top four order type categories.

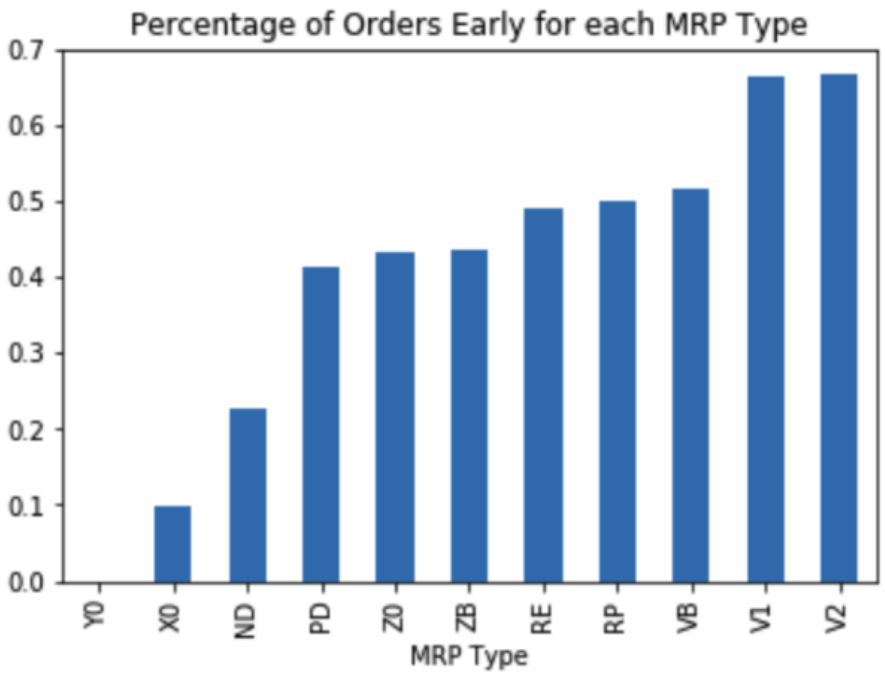

Figure 2 Percentage Early by MRP Type

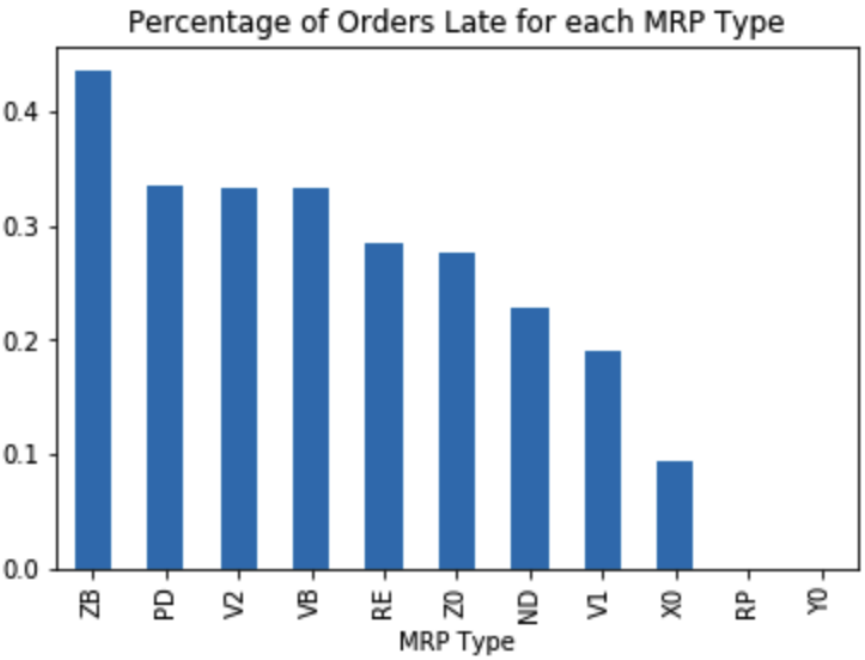

Figure 3 Percentage Late by MRP Type

who they were ordered from, and general trends through time.

We then went about looking at the percentage of orders late by company. What we found so far suggested that most companies should be late a majority of the time. However, when we looked at the distribution, we found

that the most common result was that a company was always late or always early. This result is shown below in a histogram binned by the percent of orders that were late.

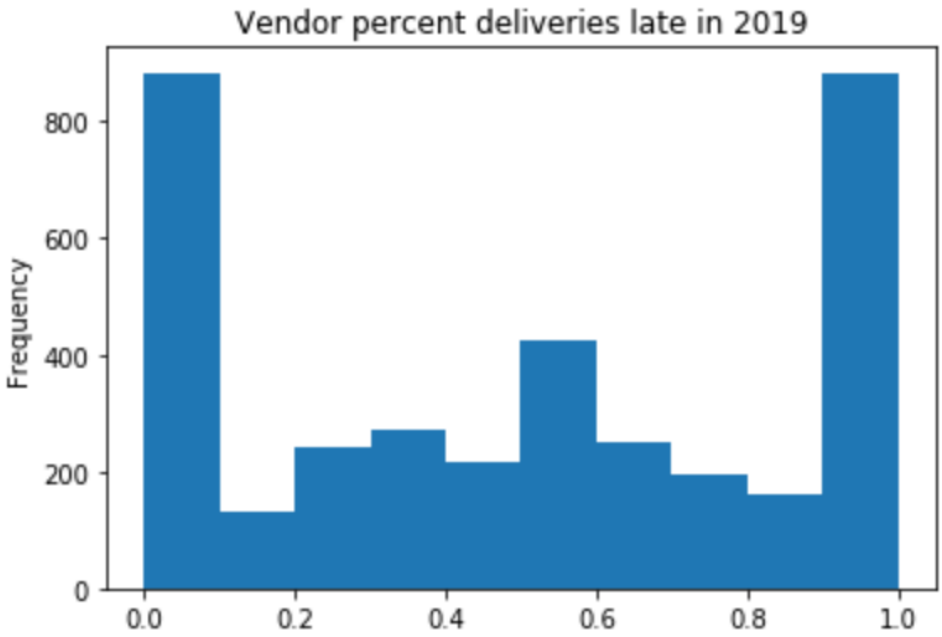

Figure 4 Histogram of Vendors by Late Delivery Rate

We found the reason for this as we looked more closely at the companies order history. There were roughly 5,000 companies that Invista had placed orders with. When we began to look at trends from who the orders where with, we quickly realized that a small group of companies represented the majority of orders. If we were to look at only the top 20 companies, we found that more than 50% of orders came from them. By upping that to the top 100 we had over 90% of orders accounted for.

This made us then investigate the opposite situation. That is how many companies had been ordered from only once. We found that this represented well over a thousand companies, and thus explained the bimodal distribution. When a company is only ordered from once it will be immediately classified into always late or never late.

This became a problem as we began to investigate time trends of orders. It was clear that most companies would not have enough of a history to allow for any meaningful exploration. This meant that we were restricted to look at the total trend or look at trends for the top 100 or so companies that had enough historical data to admit regression techniques.

This problem was further exacerbated when we realized that companies were often dropped even among those that had large order histories prior to being dropped. This became a deciding factor in looking into Invista's order history in its entirety rather than at the companies that Invista ordered from individually.

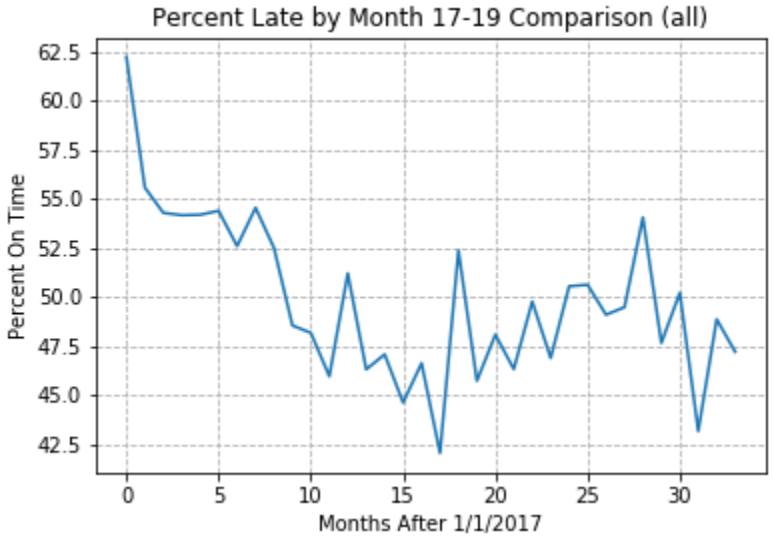

Figure 5 Percent of Orders Late Over Time

We can see from Figure 5 that early in their records there was a drop in the number of late orders going from 62.5% to 48% in 10 months. This then settled into fluctuating around 47.5% for the rest of our available history. The initial drop was not unexpected as it coincided with some changes that occurred in the company at that time. We tried to refine this by looking at bi-monthly late orders, but the results looked much the same. Refining the search into weekly levels was considered but processing the data for that would have taken several hours each time so the refinement was not considered in this project.

3. Feature Engineering

Having done our initial exploration in the data we began to try and pull more information be making new features for the data, analyzing these features, and then ultimately doing cursory model exploration.

Much of the feature creation was to deal with the categorical data we were presented with. Order type, material type, vendor rating, and several other variables needed to be converted into a form that a model could take as input. This was accomplished through one-hot encoding, but to do so we had to create an "other" category to prevent each of these categories from exploding in size.

Since each category was always dominated by three or four types it was straightforward to determine which to group into the “other” category.

We also decided to attach to each company its average percentage of on-time percentage. While this led to many with 100% or 0% we felt like the information was too important to not include. We also included the number of days from when an order was created to when the order was released. Since we believed that if there was a delay here it could lead to trouble down the line.

We then needed to try some models that could categorize the data into late/on-time/early. We decided to pick two models and test them against each other as out of the box models. Whichever did better would be the model that we would use as our final model. We decided to try two since the time constraints did not allow for a broader search. This was particularly true since most models required a fair bit of learning before we could implement them.

The models that we were choosing amongst where Logistic Regression, Feed Forward Neural Network, XGBoost, and Random Forests. Invista had good results with XGBoost before, and the package has a reputation for being excellent in categorization problems which led us to select it as one model. We ruled out Logistic Regression since our problem seemed to be more complicated than it could handle. Between an FFN and a Random Forest we decided to use a FFN since we wanted a non-tree-based method to compare to.

To decide which one to use we would train each on the same training-test data split and see which performed better on the test set. While we did some rough learning rate tuning, we made no other efforts to increase the performance of the models. The results were that the XGBoost model had an accuracy near 65% while the FFN was just above 51%. Since we were classifying into three groups, they both proved better than guessing, but XGBoost was the clear winner.

The last decision that we made at this juncture was whether to change the classification to on-time/not on-time, or to keep the current late/on-time/early distinction. If we change to a binary classification, we were able to get the accuracy of the XGBoost model up to 75% with no other adjustments. Ultimately the choice was presented to the project’s stakeholders, and they made the decision that it was better to have the late/on-time/early distinction than to have an increase in accuracy in a more restricted setting.

4. Gradient Boosting and XGBoost

Extreme Gradient Boosting is an open-source software package that was created a research project by Tianqi Chen at the University of Washington. From the Github repository we have this description of the library and its purpose:

> *"XGBoost is an optimized distributed gradient boosting library designed to be highly **efficient**, **flexible** and **portable**. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Kubernetes, Hadoop, SGE, MPI, Dask) and can solve problems beyond billions of examples."* Chen 2016

The library contains multiple models but for our model we used its implementation of Gradient boosted trees. This was due not only to its popularity but since the Invista team had used it before to good success.

We will now go into the XGBoost model and explain how it functions and some of the things that make it unique among Gradient Boosting algorithms. For readability we will show the results for binary classification and not go into the derivations of the formulas used here, but they will be included in the appendix.

Tree classification works from utilizing the idea that many weak learning models can cumulatively perform quite well. Gradient Boosting tree algorithms construct these trees iteratively, using the previous results to augment the next tree. How these trees are created then is the beginning step for any such algorithm.

For XGBoost the root and splits in the tree are made using a Similarity Score and a Gain factor. The formulas for these at the $j+1$ step in the algorithm are:

$$Similarity\ Score = \frac{\left(\sum_{i=1}^{n} p_{ij} - y_i\right)^2}{\sum_{i=1}^{n} p_{ij}(1-p_{ij}) + \lambda}$$

$$Gain = Left\ Leaf\ Similarity + Right\ Leaf\ Similarity - Root\ Similarity$$

Let p_{ij} be the predicted probability of observing y_i at the j^{th} step in the algorithm. We start with the assumption that our probability of observing y_i is uniform giving us:

$$p_{initial} = \frac{1}{n}, \text{ where } n \text{ is the possible categories of } y_i$$

The algorithm will then start by making the root node by calculating the similarity score for all the data. Each feature in the data is divided into two leaves and the similarity score is calculated for each leaf. The Gain is then calculated for the split and when the gain is maximized the split is kept.

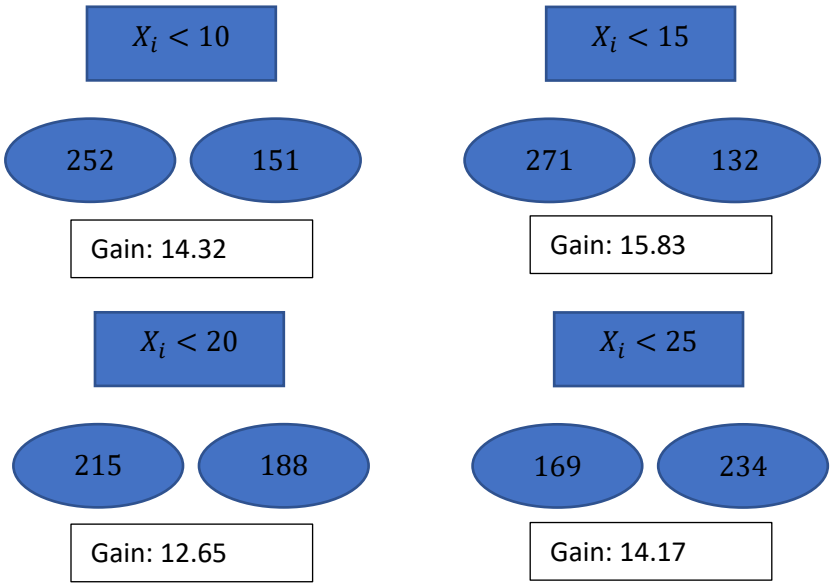

Figure 7 Split Gain Example

When the dataset is small all possible splits will be tried in a Greedy algorithm to find the true maximum value. When the dataset is large this would be time intensive so instead it checks only some divisions. It chooses these by using Sketch algorithms running in parallel to construct a distribution of the data. It then calculates several quantiles and uses these splits as divisions over which to search for the maximum Gain. This is the Approximate greedy algorithm.

This is then repeated until a user chosen number of leaves are reached. This will then be trimmed back according to another user defined parameter γ. A node is kept if the Gain is greater than γ. The value of λ in the Similarity Score affects this process making it more likely to trim a leaf with a larger value since that will shrink the Gain.

Once the trimming is finished an output value is assigned to each leaf using the formula:

$$Leaf\ Output = \frac{\sum_{i=1}^{n} p_{ij} - y_i}{\sum_{i=1}^{n} p_{ij}(1 - p_{ij}) + \lambda}$$

We then will find new predictions using both our new tree and the initial predictions. We do get to choose how much the new tree will affect the predictions using a learning rate η. This formula is then:

$$LogOdds = \log\left(\frac{p_{ij+1}}{1 - p_{ij+1}}\right) = \log\left(\frac{p_{initial}}{1 - p_{intial}}\right) + \eta \sum_{i=1}^{M} Leaf\ Output_{ij}$$

This is then the log-odds estimate which we can then transform into the probability p_{ij+1} using:

$$p_{ij+1} = \frac{e^{LogOdds}}{1 + e^{LogOdds}}$$

We can then use these to make new trees with our data and repeat the process. Iterating through this process we can then build an additive model from all the trees.

There are many other efforts spent to make sure that XGBoost runs as fast as possible while still taking advantage of the performative advantage that Gradient Boosting provides. What has been discussed here is a broad and terse accounting of the mathematical and statistical end of those efforts, a more elaborate definition and explanation will be given in the Appedix, but many more of these efforts are in more economical and particular use of computer architecture and hardware.

5. Feature Selection

Having chosen to use the XGBoost model we now had to decide which features to keep, or to add more, and then work to tune our hyperparameters. This proved to be quite time consuming, so we first fit a model with all features and tuned the hyperparameters. This gave us a benchmark against which to judge other choices. To then speed up the rest of the progress each team member chose to remove or add a set of features and then work to tune the hyperparameters. These results were then compared back to the benchmark.

In each of these cases we also needed to validate the model in each of these steps. We set aside a validation set for this so that each model the team members had would be validated on the same set.

The benchmark that we created was in the range of 74% accurate and had 100 features that it could access. To this end we wanted to test if more variables, particularly time series features, could help or if we could get by on fewer variables.

The time series features that we tested consisted of fitting an AR(p) and an MA(q) model to the ordering history from each company. This gave a series of lag weights, p-values for Dicky-Fuller tests, and MRSE values for each of them. This resulted in an additional 26 features for the dataset and was attempted because it created so many features. The idea was that the model could find a way to use the knowledge provided by these simpler models to make better predictions. While the addition of these features did increase the accuracy, it was only 0.5%. Meanwhile it caused the fitting time to increase by several hours. For this reason, these features were left out.

The attempts made to remove variables led to decreases in accuracy that we felt were too detrimental for the increased efficiency leaving us with a model with 112 features.

6. Validation

After hyper parameter tuning the final accuracy was 75%. It was now that we also wanted to check on the precision and recall capabilities. We wanted to look at this since we knew that our distribution of late/on-time/early observations was not uniform. We were concerned that this would cause it to poorly discern between the categories. We saw that when we grouped late and early into a single “not on time” category that our accuracy improved to 80% and this was without substantial effort in trying hyperparameter tuning.

Delivery Outcome	Precision	Recall
Late	70%	77%
On Time	65%	40%
Early	74%	75%

We can see from our results that our model did struggle correctly categorizing on-time orders more than it did early or late orders, particularly

when it came to recall on on-time orders. While this was unfortunate, we were surprised how high the precision remained for on-time orders. This was heartening and the stakeholders found that this was still acceptable since the main priority was to predict if an order would be early or late. Since those still performed well the model was deemed acceptable.

7. Conclusions

The goal of the project was to build a model that can predict at release if an order will be early late or on-time. We were able to build at XGBoost model that could sort three categories, late/on-time/early with 75% accuracy. This accuracy could be improved by simplifying this to a binary classification of on-time/not-on-time, but the need to differentiate between late or early was deemed to be of more use than an increase in accuracy. The model did show a propensity of struggling to classify an order as on-time as compared to late or early. This likely stems from the dataset having a low number of on-time orders to learn from.

The next steps that were not covered in the project were dealing with how often the model should be refit, and if there were a better way to deal with the disparity in on-time orders to late/early orders. We had not considered this early in the project so by the time we realized the profound effect it had we were out of time to consider remedies more fully for it. If this issue could be addressed more fully it is likely that a better model could be formed.

Appendix

A.1.1 Derivation of Similarity Score

The most common loss function that we used for the XGBoost algorithm is the negative log-loss function.

$$L(y_i, p_{ij}) = -y_i \log(p_{ij}) - (1 - y_i) \log(1 - p_{ij})$$

$$-\sum_{i=1}^{N} y_i \log(p_{y_i}) + (1 - y_i) \log(1 - p_{y_i}) + \gamma T + \lambda \overline{y_i}$$

A.1.2 Derivation of Leaf Output

We start with a regularized loss function.

$$\sum_{i=1}^{n} L(y_i, p_i^0 + O_{value}) + \frac{1}{2} \lambda O_{value}^2$$

We want to minimize this loss function but doing so can be complicated and costly. To sidestep this in the XGBoost algorithm the second order Taylor polynomial is solved instead.

$$L(y, p_i^0 + O_{value}) \approx L(y, p_i) + \left[\frac{d}{dp_i} L(y, p_i)\right] O_{value} + \frac{1}{2}\left[\frac{d^2}{dp_i^2} L(y, p_i)\right] O_{value}^2$$

$$L = \sum_{i=1}^{n} L(y, p_i) + \left[\frac{d}{dp_i} L(y, p_i)\right] O_{value} + \frac{1}{2}\left[\frac{d^2}{dp_i^2} L(y, p_i)\right] O_{value}^2 + \frac{1}{2} \lambda O_{value}^2$$

We can then solve for the output value O_{value} that minimizes this by locating the extreme value points. As this is a quadratic form and the loss function is strictly positive, we know that the resulting extreme value will be a minimum.

$$\frac{dL}{dO_{value}} = \sum_{i=1}^{n} \frac{d}{dp_i} L(y, p_i) + \frac{d^2}{dp_i^2} L(y, p_i) O_{value} + \lambda O_{value} = 0$$

$$\left(\sum_{i=1}^{n}\frac{d^2}{dp_i^2}L(y,p_i)+\lambda\right)O_{value}=\sum_{i=1}^{n}\frac{d}{dp_i}L(y,p_i)$$

In the case of a binary classification this can be solved explicitly. However, for the case of higher classification this results in a matrix problem that, while able to be solved explicitly, is better left to numerical solvers.

B.1 Code for Making Final Features

```
# In[3]:

import sys
import numpy as np
import pandas as pd
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import xgboost as xgb
from sklearn.model_selection import train_test_split

sys.path.append('../')
from util import *

pd.set_option('display.max_columns', 500)
pd.set_option('display.float_format', lambda x: '%.3f' % x)


# In[4]:

def convert_dates(df,
dates=['CREATE_DATE','RELEASE_DATE','FIRST_GR_POSTING_DATE','POR_DELIVERY_DA
TE','REQUESTED_DELIVERY_DATE', 'DELIVERY_DATE'],verbose=False):
    """
    Converts date columns to datetime type.

    Parameters:
    -----------
    df: dataframe on which to perform the conversion
```

```
    dates: columns to convert, default should contain all necessary columns

    Returns:
    ----------
    Panda dataframe
    """
#     dates =
['CREATE_DATE','RELEASE_DATE','FIRST_GR_POSTING_DATE','POR_DELIVERY_DATE','RE
QUESTED_DELIVERY_DATE', 'DELIVERY_DATE']

#     for date in dates:
#         # convert column to datetime, values not converting will become NaN (such as
0.0)
#         df[date] = pd.to_datetime(df[date], format='%Y%m%d', errors='coerce')
#     return df

    # Note: could probably add check for type, will crash if run on datetime
    for date in dates:
        df[date] = df[date].replace([np.inf, -np.inf, np.nan], 0)
        try:
            df[date] = pd.to_datetime(df[date], format='%Y%m%d', errors='coerce')
        except:
            if verbose:
                print(f'exception thrown: {date}')
            df[date] = pd.to_datetime(df[date].astype(int), format='%Y%m%d',
errors='coerce')
        if verbose:
            print(f'CONVERTED {date}')
        df[f'{date}_YEAR'] = df[date].dt.year
        df[f'{date}_MONTH'] = df[date].dt.month
        df[f'{date}_DAY'] = df[date].dt.day
    if verbose:
        print('FINISHED CONVERSION')
    return df


# In[5]:

data2017, data2018, data2019_20, vendor_data = get_procurement_data()

# Combine data2017, data2018, and data2019_20 into one dataframe
```

```
data = pd.concat([data2017, data2018, data2019_20], ignore_index=True)


# In[6]:

convert_dates(data)
'DONE'


# In[7]:

# Creating some features
def get_short_text(df):
   df['Short_Text_Order'] = np.where(df.MATERIAL_ID.isnull() &
df.SHORT_TEXT.notnull(), 1, 0)
   return df

def get_total_orders(df):
   def get_total_orders_sub(df):
      total_vendor_orders = df.groupby('VENDOR_ID').size()
      total_vendor_orders = pd.DataFrame([total_vendor_orders], index=['VENDOR_ID'])
      total_vendor_orders = total_vendor_orders.transpose()
      total_vendor_orders = total_vendor_orders.rename(columns={'VENDOR_ID':
'total_vendor_orders'})
      total_vendor_orders = total_vendor_orders.reset_index()
      total_vendor_orders['total_vendor_orders'] =
total_vendor_orders['total_vendor_orders'].astype(int)

      return total_vendor_orders

   df = pd.merge(df, get_total_orders_sub(df)[['VENDOR_ID','total_vendor_orders']],
left_on='VENDOR_ID', right_on='VENDOR_ID', how='left')
   return df

def get_percent_null_gr(df):
   def get_null_gr_percent_sub(df):
      null_gr_dates =
df[df['FIRST_GR_POSTING_DATE'].isnull()].groupby('VENDOR_ID').size()
      total_vendor_orders = df.groupby('VENDOR_ID').size()
      vendor_nullgr = pd.concat([null_gr_dates, total_vendor_orders], axis=1, sort=True)
```

```
        vendor_nullgr = vendor_nullgr.reset_index()
        vendor_nullgr = vendor_nullgr.rename(columns={'index': 'VENDOR_ID', 0: 'null_gr',
1: 'total_orders'})
        vendor_nullgr['percent_null_gr'] = vendor_nullgr['null_gr'] /
vendor_nullgr['total_orders']
        vendor_nullgr = vendor_nullgr.fillna(0)
        vendor_nullgr['null_gr'] = vendor_nullgr['null_gr'].astype(int)
        return vendor_nullgr

    df = pd.merge(df, get_null_gr_percent_sub(df)[['VENDOR_ID','percent_null_gr']],
left_on='VENDOR_ID', right_on='VENDOR_ID', how='left')
    return df



def get_rating(df, vendor_df):
    df = pd.merge(df, vendor_df[['Vendor ID','Rating']], left_on='VENDOR_ID',
right_on='Vendor ID', how='left')
    df = df.drop(columns='Vendor ID')
    return df

def create_release_diff(df):
    df['Create_to_Release_Diff'] = df['RELEASE_DATE'] - df['CREATE_DATE']
    return df

def material_type_feature(df):
    df['General Operating Supplies'] = np.where(df['MATERIAL_ID'].astype(str).str[0:1] ==
'1', 1, 0)
    df['Semifinished & Finished Materials'] =
np.where(df['MATERIAL_ID'].astype(str).str[0:1] == '2', 1, 0)
    df['Packaging'] = np.where(df['MATERIAL_ID'].astype(str).str[0:1] == '3', 1, 0)
    df['Raw Materials'] = np.where(df['MATERIAL_ID'].astype(str).str[0:1] == '4', 1, 0)

    return df

def subcommodity_type_feature(df):
    df['Custom Manufacturing'] = np.where(df['SUB_COMMODITY_DESC'] == 'Custom
Manufacturing', 1, 0)
    df['Additives, Colorants & Catalysts'] = np.where(df['SUB_COMMODITY_DESC'] ==
'Additives, Colorants & Catalysts', 1, 0)
    df['Tolling'] = np.where(df['SUB_COMMODITY_DESC'] == 'Tolling', 1, 0)
```

```
    df['Subcommodity_Other'] = np.where((df['SUB_COMMODITY_DESC'] != 'Custom
Manufacturing') & (df['SUB_COMMODITY_DESC'] != 'Additives, Colorants & Catalysts') &
(df['SUB_COMMODITY_DESC'] != 'Tolling'), 1, 0)

    return df


def MRP_type_feature(df):
    df['MRP_X0'] = np.where(df['MRP_TYPE_ID'] == 'X0', 1, 0)
    df['MRP_ND'] = np.where(df['MRP_TYPE_ID'] == 'ND', 1, 0)
    df['MRP_Y0'] = np.where(df['MRP_TYPE_ID'] == 'Y0', 1, 0)
    df['MRP_Other'] = np.where((df['MRP_TYPE_ID'] != 'X0') & (df['MRP_TYPE_ID'] !=
'ND') & (df['MRP_TYPE_ID'] != 'Y0'), 1, 0)

    return df


# In[8]:

# Functions that we can perform on the whole dataset
    # 1) Drop null first_gr_posting_dates
    # 2) Filter Vendors
    # 3) get ratings
    # 4) create_release_diff
    # 5) material_type_feature
    # 6) subcommodity_type
    # 7) mrp_type
    # 8) short_text


# In[9]:

# 1) drop null first_gr_posting_dates
# dropping null first_gr_posting dates, 0's are nulls
print("Before: ", data[data['FIRST_GR_POSTING_DATE']==0].shape[0])
print("Before: ", data[data['FIRST_GR_POSTING_DATE'].isna()].shape[0])

data = data[data['FIRST_GR_POSTING_DATE'] != 0]
data = data[data['FIRST_GR_POSTING_DATE'].notnull()]
print("After: ", data[data['FIRST_GR_POSTING_DATE']==0].shape[0])
print("After: ", data[data['FIRST_GR_POSTING_DATE'].isna()].shape[0])
```

```
# In[10]:

# 2) Filter Vendors

# drop all vendors starting with V
# data = data[~data['VENDOR_NM'].astype(str).str.startswith('V')]

print("Before: ", data.shape[0])
data = filterVendors(data)
print("After: ", data.shape[0])


# In[11]:

# 3) Get Ratings
data = vendor_segmentation(data, vendor_data)


# In[12]:

data.columns


# In[13]:

# data.loc[29].Rating


# In[14]:

# def rating_to_number(df):
#     # C: 0, B: 1, A: 2; NaN are set to 0
#     df['RatingNum'] = 0

#     df['RatingNum'] = np.where(df.Rating == 'C', 0, df['RatingNum'])
#     df['RatingNum'] = np.where(df.Rating == 'B', 1, df['RatingNum'])
#     df['RatingNum'] = np.where(df.Rating == 'A', 2, df['RatingNum'])

#     # Create another field to keep track of null ratings
```

```
#     df['NoRating'] = np.where(df.Rating.isnull(),1,0)

#     return df


# In[15]:

# data = rating_to_number(data)


# In[16]:

# data[['Rating', 'RatingNum', 'NoRating']].tail()


# In[17]:

# 4) Create_to_Release Diff
data = create_release_diff(data)


# In[18]:

data['Create_to_Release_Diff'] = data['Create_to_Release_Diff'].dt.days


# In[19]:

data['Create_to_Release_Diff'].tail()


# In[20]:

# 5) material_type_feature
data = material_type_feature(data)


# In[21]:

# 6) subcommodity_type
data = subcommodity_type_feature(data)
```

In[22]:

```
# 7) mrp_type
data = MRP_type_feature(data)
```

In[23]:

```
# 8) short_text
data = get_short_text(data)
```

In[24]:

```
#######################################
# Set the Target and split the data #
#######################################
```

In[25]:

```
set_target(data)
# data = data.drop(columns=['Late', 'OnTime', 'Early', 'FIRST_GR_POSTING_DATE']
```

In[26]:

```
data =
data.drop(columns=['FIRST_GR_POSTING_DATE','FIRST_GR_POSTING_DATE_YEAR','FIRS
T_GR_POSTING_DATE_MONTH','FIRST_GR_POSTING_DATE_DAY'])
```

In[27]:

```
# data = data.drop(columns=['Late','OnTime','Early'])
# data.to_csv('output.csv')
```

In[28]:

```
data_full = data.copy()
# X_data = data_full.drop('Late', axis=1)
X_data = data_full.drop('DeliveryOutcome', axis=1)
y = data_full.DeliveryOutcome


# In[29]:

X_train, X_test, y_train, y_test = get_train_test_data(X_data, y, test_size=.3,
random_state='kyson')
X_train = X_train.copy()

X_test = X_test.copy()
# X_test = X_test.drop(columns=['Late', 'OnTime', 'Early'])

y_train = y_train.copy()
y_test = y_test.copy()


# In[30]:

print(X_train.shape)
print(X_test.shape)


# In[31]:

# functions to be done on training and then imputed to test set
   # 1) get_arrival_percentage
   # 2) get_total_orders
   # 3) group_plants


# In[32]:

# 1) get_arrival_percentage
X_train = X_train.rename(columns={"VendorID": "VENDOR_ID"})
X_test = X_test.rename(columns={'VendorID': 'VENDOR_ID'})
```

In[33]:

```
X_train = get_arrival_percentage_train(X_train,'VENDOR_ID','Vendor')
X_train = get_arrival_percentage_train(X_train,'PLANT_ID','Plant')
X_train = get_arrival_percentage_train(X_train,'MRP_TYPE_ID','MRP_Type')
X_train = get_arrival_percentage_train(X_train, 'POR_DELIVERY_DATE_MONTH',
'POR_Month')
X_train = get_arrival_percentage_train(X_train, 'RELEASE_DATE_MONTH',
'Release_Date_Month')
```

In[34]:

```
X_train.columns
```

In[35]:

```
X_test = get_arrival_percentage_test(X_test, X_train, 'VENDOR_ID', 'Vendor')
X_test = get_arrival_percentage_test(X_test, X_train, 'PLANT_ID', 'Plant')
X_test = get_arrival_percentage_test(X_test, X_train, 'MRP_TYPE_ID', 'MRP_Type')
X_test = get_arrival_percentage_test(X_test, X_train, 'POR_DELIVERY_DATE_MONTH',
'POR_Month')
X_test = get_arrival_percentage_test(X_test, X_train, 'RELEASE_DATE_MONTH',
'Release_Date_Month')
```

In[36]:

```
X_test.columns
```

In[37]:

```
# 3) group_plants
# first on the X_train
X_train = group_plants(X_train, verbose=True, max_buckets=4)
```

In[38]:

```
plants = [4014, 4064, 4050]
X_train.columns


# In[39]:

def group_plants_test(df, plant_list):
   for i in range(len(plant_list)):
#        print(plant_list[i])
       plant_name = "PLANT_" + str(plant_list[i])
       df[plant_name] = np.where(df['PLANT_ID'] == plant_list[i], 1, 0)

#    print(plant_list)
   df['PLANT_Other'] = df.apply(lambda row: 1 if row.PLANT_ID not in plant_list else 0,
axis=1)


# In[40]:

group_plants_test(X_test, plants)


# In[41]:

X_test.isnull().sum()


# In[42]:

###################
# Run the models #
###################


# In[43]:

print(X_train.columns, len(X_train.columns))
print(X_test.columns, len(X_test.columns))
list(X_test.columns)
```

```
# In[44]:

final_columns = ['PURCHASE_DOCUMENT_ID',
#  'CREATE_DATE',
#  'COMPANY_CODE_ID',
#  'COMPANY_CODE_NAME',
#  'VENDOR_ID',
#  'VENDOR_NM',
#  'POSTAL_CD',
#  'RELEASE_DATE',
 'PURCHASE_DOCUMENT_ITEM_ID',
#  'MATERIAL_ID',
#  'SUB_COMMODITY_DESC',
#  'MRP_TYPE_ID',
#  'MRP_TYPE_DESC_E',
#  'SHORT_TEXT',
#  'PLANT_ID',
#  'PLANT_NAME',
#  'POR_DELIVERY_DATE',
#  'DELIVERY_DATE',
#  'REQUESTED_DELIVERY_DATE',
 'DELIVERY_ID',
 'DELIVERY_ITEM_ID',
 'PLANNED_DELIVERY_DAYS',
 'CREATE_DATE_YEAR',
 'CREATE_DATE_MONTH',
 'CREATE_DATE_DAY',
 'RELEASE_DATE_YEAR',
 'RELEASE_DATE_MONTH',
 'RELEASE_DATE_DAY',
#  'FIRST_GR_POSTING_DATE_YEAR',
#  'FIRST_GR_POSTING_DATE_MONTH',
#  'FIRST_GR_POSTING_DATE_DAY',
 'POR_DELIVERY_DATE_YEAR',
 'POR_DELIVERY_DATE_MONTH',
 'POR_DELIVERY_DATE_DAY',
 'REQUESTED_DELIVERY_DATE_YEAR',
 'REQUESTED_DELIVERY_DATE_MONTH',
 'REQUESTED_DELIVERY_DATE_DAY',
 'DELIVERY_DATE_YEAR',
```

```
 'DELIVERY_DATE_MONTH',
 'DELIVERY_DATE_DAY',
#  'VendorID',
#  'Rating',
 'imputeRatingFlag',
#  'VendorGradeA',
#  'VendorGradeB',
#  'VendorGradeC',
 'RatingNum',
#  'Create_to_Release_Diff',
 'General Operating Supplies',
 'Semifinished & Finished Materials',
 'Packaging',
 'Raw Materials',
 'Custom Manufacturing',
 'Additives, Colorants & Catalysts',
 'Tolling',
 'Subcommodity_Other',
#  'MRP_X0',
#  'MRP_ND',
#  'MRP_Y0',
#  'MRP_Other',
 'Short_Text_Order',
#  'Delivery_Difference',
 'Vendor_Percent_Late',
 'Vendor_Percent_OnTime',
 'Vendor_Percent_Early',
 'Plant_Percent_Late',
 'Plant_Percent_OnTime',
 'Plant_Percent_Early',
 'MRP_Type_Percent_Late',
 'MRP_Type_Percent_OnTime',
 'MRP_Type_Percent_Early']
# ,
#  'PLANT_4014',
#  'PLANT_4064',
#  'PLANT_4050',
#  'PLANT_Other']


# In[45]:
```

```
X_train = X_train[final_columns]
X_test = X_test[final_columns]


# In[46]:

print('X_Test', X_test.shape)
print('X_train', X_train.shape)
print('Y_Test', y_test.shape)
print('y_train', y_train.shape)


# In[47]:

#Train the XGboost Model for Classification
model1 = xgb.XGBClassifier()
model2 = xgb.XGBClassifier(n_estimators=100, max_depth=8, learning_rate=0.1,
subsample=0.5)

train_model1 = model1.fit(X_train, y_train)
train_model2 = model2.fit(X_train, y_train)


# In[48]:

#prediction and Classification Report
from sklearn.metrics import classification_report

pred1 = train_model1.predict(X_test)
pred2 = train_model2.predict(X_test)


# In[49]:

print('Model 1 XGboost Report %r\n' % (classification_report(y_test, pred1)))
classification_report(y_test, pred1)
print('Model 2 XGboost Report %r' % (classification_report(y_test, pred2)))


# In[50]:
```

```
#Let's use accuracy score
from sklearn.metrics import accuracy_score

print("Accuracy for model 1: %.2f" % (accuracy_score(y_test, pred1) * 100))
print("Accuracy for model 2: %.2f" % (accuracy_score(y_test, pred2) * 100))

# Accuracy for model 1: 67.25 ~ 70.54
# Accuracy for model 2: 67.08


# In[51]:

tmp = list(train_model1.feature_importances_)
tmp2 = list(X_train.columns)


# In[52]:

fi = {}
for i in range(len(tmp2)):
   fi[tmp2[i]] = tmp[i]


# In[53]:

dfObj = pd.DataFrame(fi.items())
dfObj = dfObj.rename(columns={0: 'Feature', 1: 'feature_importance'})


# In[54]:

dfObj.sort_values('feature_importance', ascending=False)


# In[55]:

X_train[['VENDOR_ID','percent_late','percent_on_time', 'percent_early']]


# In[56 ]:
```

```
#Let's do a little Gridsearch, Hyperparameter Tunning
model3 = xgb.XGBClassifier(
 learning_rate =0.1,
 n_estimators=1000,
 max_depth=5,
 min_child_weight=1,
 gamma=0,
 subsample=0.8,
 colsample_bytree=0.8,
 objective= 'binary:logistic',
 nthread=4,
 scale_pos_weight=1,
 seed=27)


# In[57]:

train_model3 = model3.fit(X_train, y_train)
pred3 = train_model3.predict(X_test)
print("Accuracy for model 3: %.2f" % (accuracy_score(y_test, pred3) * 100))

# model 3 test 1 accuracy: 66.8


# In[58]:

from sklearn.model_selection import GridSearchCV

param_test = {
 'max_depth':[4,5,6],
 'min_child_weight':[4,5,6]
}
gsearch = GridSearchCV(estimator = xgb.XGBClassifier( learning_rate=0.1,
n_estimators=140, max_depth=5,
 min_child_weight=2, gamma=0, subsample=0.8, colsample_bytree=0.8,
 objective= 'binary:logistic', nthread=4, scale_pos_weight=1,seed=27),
 param_grid = param_test, scoring='roc_auc',n_jobs=4,iid=False, cv=5)

train_model4 = gsearch.fit(X_train, y_train)
pred4 = train_model4.predict(X_test)
```

```
print("Accuracy for model 4: %.2f" % (accuracy_score(y_test, pred4) * 100))

# model 4 test 1 accuracy: 75.51


# In[59]:

param_test2b = {
 'min_child_weight':[6,8,10,12]
}
gsearch2b = GridSearchCV(estimator = xgb.XGBClassifier( learning_rate=0.1,
n_estimators=140, max_depth=4,
 min_child_weight=2, gamma=0, subsample=0.8, colsample_bytree=0.8,
 objective= 'binary:logistic', nthread=4, scale_pos_weight=1,seed=27),
 param_grid = param_test2b, scoring='roc_auc',n_jobs=4,iid=False, cv=5)

train_model5 = gsearch2b.fit(X_train, y_train)
pred5 = train_model5.predict(X_test)
print("Accuracy for model 5: %.2f" % (accuracy_score(y_test, pred5) * 100))

# model 5 test 1 accuracy: 72.98


# In[60]:

#Tune Gamma
param_test3 = {
 'gamma':[i/10.0 for i in range(0,5)]
}
gsearch3 = GridSearchCV(estimator = xgb.XGBClassifier( learning_rate =0.1,
n_estimators=140, max_depth=4,
 min_child_weight=6, gamma=0, subsample=0.8, colsample_bytree=0.8,
 objective= 'binary:logistic', nthread=4, scale_pos_weight=1,seed=27),
 param_grid = param_test3, scoring='roc_auc',n_jobs=4,iid=False, cv=5)

train_model6 = gsearch3.fit(X_train, y_train)
pred6 = train_model6.predict(X_test)
print("Accuracy for model 6: %.2f" % (accuracy_score(y_test, pred6) * 100))

# model 6 test 1 accuracy: 73.11
```

```
# In[61]:

xgb2 = xgb.XGBClassifier(
 learning_rate =0.7,
 n_estimators=1000,
 max_depth=4,
 min_child_weight=6,
 gamma=0,
 subsample=0.8,
 colsample_bytree=0.8,
 objective= 'binary:logistic',
 nthread=4,
 scale_pos_weight=1,
 seed=27)

train_model7 = xgb2.fit(X_train, y_train)
pred7 = train_model7.predict(X_test)
# model 7 test 1 accuracy: 81.32
```

References

Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785–794). New York, NY, USA: ACM. https://doi.org/10.1145/2939672.2939785