

**Testing Project**

**April 15, 2021**

**Team #4**

**Jaime Chaire**

**Ethan Dark**

**Jamie Liu**

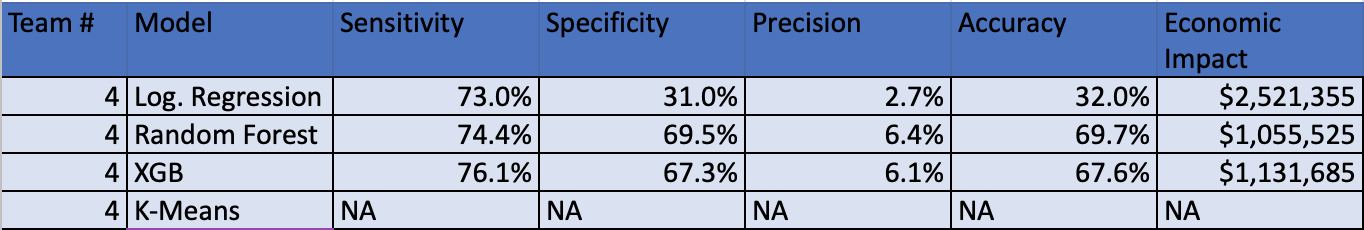
**Huaiqian Yan**

# Summary

This analysis focuses on decreasing Seagate’s costs associated with a hard drive failing within the first 90 days of being in the field with a customer. As Seagate replaces each hard drive that fails within the 90 day time period, accurately predicting a hard drive failure is essential in preventing costs to replace the drive or costs associated with falsely predicting a functional drive as a failure. By predicting the failure of a drive before shipping it to a customer, Seagate will reduce costs of replacement as well as enhance the customer experience by eliminating the risk of a customer receiving a flawed drive. This analysis will ultimately improve Seagate’s manufacturing process by determining the relationships between attributes and drive failures.

To predict drive failures, we tested our cleaned dataset with random forest, extreme gradient boosting, logistic regression, and k-means models to determine the attributes that may have a significant relationship with a drive failing. To build the models, the data set was split into training and test sets. As the data set provided was extremely imbalanced with 97.4% passed drives and 2.6% failed drives, to have a better outcome white testing our models, we undersampled the training data sets for 3 of the models to balance the number of passed and failed drives to give us a better idea of which features can predict failures.

## Model Metrics



## Recommendations

We recommend using our Random Forest model to determine which attributes are the most significant in contributing to the failures of a drive within the first 90 days in the field. The top 10 features rated most important were the following PARAM\_MEDIANs: 2, 25, 26, 27, 28, 229, 54, 112, 114, 116. Although this model is lacking in precision, it resulted in the overall lowest cost for Seagate. We would also suggest looking at the models we ran in PySpark without undersampling to capture the full predictability of our models on the full dataset. Although the PySpark models were not tuned to the fullest extent, we expect them to have the highest performance since they utilize the entire dataset.

A result of our models being optimized for sensitivity in order to capture all possible failures is a significant number of false positives. As falsely predicting successful drives as failures is the most costly at $40 per drive, we suggest implementing methods to retest the predicted failed drives before discarding them. This would ensure that the drives that are functional but are labeled as failures would not be wasted as our models tend to lack in precision due to the limited number of failed drives to train on in the data set.

# Data

## Cleaning approach

To prepare our data for building the models, we began by removing columns that were not expected to have a relationship with failures such as FAD MAGNET LOT TOP, LOT BOT, SHIP CALENDAR, CALENDAR MONTH, and ACTUAL DATE to name a few, ultimately reducing the dataset from it's 220 features. After removing unnecessary columns, we dummy coded each categorical column. The column ‘TARGET’ that contained information on whether the drive passed or failed was also dummy coded. The final ‘TARGET’ column consisted of 1’s if the drive failed, and 0’s if the drive passed. After dummy coding, we ran a correlation analysis on each predictor variable and removed the columns that had high multicollinearity. The final data set used for our models consisted of 126 features. As there were a significant number of nulls throughout the data set, we chose to impute using the median for continuous variables and the mode for categorical variables. This ensured that the imputed values did not significantly skew the data by using values that were likely to be similar to the actual values.

After our presentation, the Seagate team voiced their issues with our undersampling approach. To resolve this, we ensured our models were only undersampled on the training sets. We also ran each model on Spark Apache to leverage Spark’s ability to handle massive data processes. The K-Means Clustering model was already implemented through a Pyspark SparkSession, so we looked into what else was needed for the other models. Since the model needed a VectorAssembler and StandardScaler already, we had the features set up for any type of machine learning model. That being said, all that was left to do was adjust the model parameters and call them on the regular features and normalized features. We were able to train and test on the whole dataset but did not have enough time to fully tune these models.

An aspect of our models we would have liked to explore more if we had the time was how to impute the null values. We used the modes for the categorical columns, but we did not have time to check each imputed mode value to make sure that they made sense in relation to the other categorical columns. It also would have been interesting to test filling in the numerical nulls with the mean values rather than the median values we used to see if they improved our models’ performances. We would have liked to spend more time deciding how to account for the intentionally blank values rather than imputing all of them. This would have been easier to do if we had more information on what the columns represented, but with the limited information we had on the columns and their data, we were unable to decipher which rows and columns had intentional missing values.

# Model 1 - Random Forest

## Description

## The first model that we ran with our dataset was a Random Forest model. The aim to use this model is to observe the bagged decision trees that are built with the data collection of de-correlated trees to further improve our predictive performance. The Random Forest model is built with similar principles as decision trees and bagging trees. This model will be helping reduce tree correlation by introducing more randomness into the tree-growing process. To add more detail while we grow a decision tree during the bagging process, our random forest will be splitting variables randomizing where each time a split is to be performed. Before thoroughly cleaning the dataset our random forest model was not able to predict failures, making this model useless. After deliberating and investigating the reasons why our model was underperforming, we decided to take a closer look at the dataset in hand. We decided to run an important feature function; we managed to obtain a much cleaner dataset, ultimately reducing the uncorrelated features.

## Implementation details

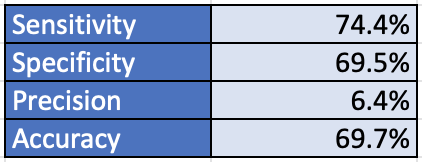
Having a clean dataset, with 214 features subsequently creating our dummy variables for the categorical features we now are in a better state to obtain better results from our model. Another thing that we had to do was to undersample our data. The reason behind undersampling was due to the low ratio of the pass-fail TARGET variable. This was mainly due to encounter better results from our model. Even though undersampling datasets skews the data we were forced to do that in order to obtain better predictions for our confusion matrix.

Our next step was to split out data into training and test data, with a split ratio of 70:30 we trained our dataset and got our first promising results. With our balanced training sample of 5,390 passes and 5,390 fail tests, we were able to train our model to then use its findings and evaluate the entire dataset with 288039 passes and 7802 fails.

## Results

|  |  |  |  |
| --- | --- | --- | --- |
| **True/Predicted** | **Predicted 0** | **Predicted 1** | **Total** |
| Actual 0 | 60,030 | 26,311 | 86,371 |
| Actual 1 | 617 | 1795 | 2,412 |
| **Total** | **60,647** | **28,106** | **88,753** |

## Performance Metrics



## Evaluation

|  |  |  |  |
| --- | --- | --- | --- |
| **True/Predicted** | **Predicted 0** | **Predicted 1** | **Total** |
| Actual 0 | $0 | $1,052,440 | $1,052,440 |
| Actual 1 | $3,085 | $0 | $3,085 |
| **Total** | **$3,085** | **$1,052,440** | **$1,055,525** |

## Conclusion

Looking at our results we can determine that there is room to improve and make higher accurate predictions on hard drives that fail. It is important to highlight that one of the important things that we learned while working on this project can be broken down into three things. One, and probably the most important is how to properly undersample our dataset while always making sure our test data set remains “raw”. Second, depending on what predictions we are trying to make, our dependent variable should be switched to 1. In this case, we made sure our target variable was changed to pass = 0 and fail = 1. Third, is making sure one has the time to better understand the dataset one is working with, this will make it easy when it comes to determining filling null values if it should be filled with the nearest neighbor, mean, mode, or something else.

The outcomes of the models came out with a 69.7% accuracy making this model our best performing model. The goal for the team was to prioritize the capture of all drives that could potentially fail, making our sensitivity score for this model a 74.4%. With more time we would have implemented and tested different approaches with the same model. To start, I would like to train our model with a bigger dataset that way we can potentially increase the accuracy and precision. That being said it's important to spend more time understanding the dataset and the parameters of each variable. This way we could better select what values to input to our Null and missing values in the dataset.

# Model 2 - XGBoost

## Description

The second model we created was an Extreme Gradient Boosting (XGBoost) model. Gradient boosting is a method that creates new models to predict the errors made by previous models. Additional models are added until it is done improving to create a final prediction, therefore, we initially expected our XGBoost model to have the best performance. We split the data into a ratio of 70:30 for the training and test sets. The XGBoost model was trained on data that was undersampled with a ratio of 1:1 for passed and failed drives. Although undersampling limits the model’s overall predictive performance by excluding rows of data that could be useful in training the model, the extreme imbalance in the original data made it difficult to create a model with any predictive ability. By undersampling on the training data, this ensured that the predictability of the model was increased by balancing the proportion of passed drives in comparison to the small number of failed drives.

## Implementation details

To find the optimal hyperparameters for the model, we created a parameter grid to use with a randomized grid search. The range of parameters used resulted in 8,100 total possible combinations for the grid search to test. To ensure efficiency in the grid search process with 10-fold cross-validation, only 50 combinations were tested which resulted in 500 total fits. Each grid search run was scored based on the Area Under the Curve score to determine the best combination of parameters. The final parameters used to build the model were: 'subsample': 1.0, 'n\_estimators': 100, 'min\_child\_weight': 1, 'max\_depth': 9, 'learning\_rate': 0.05, 'gamma': 0.5, 'colsample\_bytree': 0.6.

The model was trained on 5,390 rows of data on drives that passed and 5,390 rows of data on drives that failed. When tested on the full test data set to make predictions, the model resulted in the performance metrics below.

## Results

|  |  |  |  |
| --- | --- | --- | --- |
| **Actual/Predicted** | **Predicted 0** | **Predicted 1** | **Total** |
| Actual 0 | 58,121 | 28,220 | 86,341 |
| Actual 1 | 577 | 1,835 | 2,412 |
| **Total** | **58,698** | **30,055** | **88,753** |

**Performance Metrics**

|  |  |
| --- | --- |
| **Sensitivity** | 76.1% |
| **Specificity** | 67.3% |
| **Precision** | 6.1% |
| **Accuracy** | 67.6% |

The top 20 features with the highest importance scores contributing to the XGBoost model’s prediction of failures were:

PARAM\_192\_MEDIAN (548), PARAM\_028\_MEDIAN (542), PARAM\_029\_MEDIAN (539), PARAM\_116\_MEDIAN (519), PARAM\_002\_MEDIAN (463), PARAM\_130\_MEDIAN (457), PARAM\_027\_MEDIAN (420), PARAM\_025\_MEDIAN (416), PARAM\_026\_MEDIAN (392), PARAM\_144\_MEDIAN (388), PARAM\_112\_MEDIAN (383), PARAM\_201\_MEDIAN (373), PARAM\_054\_MEDIAN (363), PARAM\_059\_MEDIAN (358), PARAM\_043\_MEDIAN (351), FAD\_FBP2 (351), PARAM\_202\_MEDIAN (348), PARAM\_175\_MEDIAN (341), PARAM\_021\_MEDIAN (323), PARAM\_018\_MEDIAN (312).

## Evaluation

|  |  |  |  |
| --- | --- | --- | --- |
| **Actual/Predicted** | **Predicted 0** | **Predicted 1** | **Total** |
| Actual 0 | $0 | $1,128,800 | $1,128,800 |
| Actual 1 | $2,885 | $0 | $2,885 |
| **Total** | **$2,885** | **$1,128,800** | **$1,131,685** |

## Conclusion

A drawback as we have learned from building this model is the effect of undersampling. Because undersampling limits the amount of data used to train the model, the model could potentially lose out on predictability by not using all of the available useful information. As this model was built to optimize the recall score of 76.1% to prioritize capturing all drives that could potentially fail rather than only capturing the drives that actually fail, the precision score was drastically reduced. Looking back at our approach to building this model, if we had more time to optimize our models we would run through more of the 8,100 possible combinations of parameters to find a combination that optimized the precision score seeing as false positives were the most costly at an estimated $40 per drive falsely predicted to fail. Towards the end of our project, we figured out how to run a GBT model in PySpark using the whole dataset but did not have time to fully tune and optimize that model. If we had more time, we would have moved forward with the PySpark models as those models had better performance metrics overall and did not lose predictive power since they did not have to be undersampled.

# Model 3 - Logistic Regression

## Description

Logistic regression is a classification algorithm used to assign observations to a discrete set of classes. It is widely used for predicting binary outcomes and easy to implement. Mathematically, a binary logistic model has a dependent variable with two possible values, such as pass/fail which is represented by an indicator variable, where the two values are labeled "0" and "1". The logistic regression model itself simply models probability of output in terms of input and does not perform statistical classification (it is not a classifier), though it can be used to make a classifier, for instance by choosing a cutoff value and classifying inputs with probability greater than the cutoff as one class, below the cutoff as the other; this is a common way to make a binary classifier.

**Implementation details**

This model was based on the same data prepared for the random forest model. However, it was undersampled randomly by the function “RandomUnderSampler” from the “imblearn” library in python. We let the function choose the number of records to remove automatically. After undersampling, we had 11,008 rows left in the train set instead of 207,088. We built the logistic model through builder from the “sklearn.linear\_model” library with default parameters.

**Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **Actual/Predicted** | **Predicted 0** | **Predicted 1** | **Total** |
| Actual 0 | 26,767 | 59,688 | 86,455 |
| Actual 1 | 620 | 1,678 | 2,298 |
| **Total** | **27,387** | **61,366** | **88,753** |

**Performance Metrics**

|  |  |
| --- | --- |
| **Sensitivity** | 73.0% |
| **Specificity** | 31.0% |
| **Precision** | 2.7% |
| **Accuracy** | 32.0% |

## Evaluation

|  |  |  |  |
| --- | --- | --- | --- |
| **Actual/Predicted** | **Predicted 0** | **Predicted 1** | **Total** |
| Actual 0 | $133,835 | $2,387,520 | $2,521,3555 |
| Actual 1 | $0 | $0 | $0 |
| **Total** | **$**133,835 | **$**2,387,520 | **$**2,521,3555 |

## Conclusion

The logistic model performed not well. Its accuracy was only about 32% which means the probability of its prediction on Pass/Fail to be correct . The worse thing is it predicted only 2.7% of all failed testing successfully. In this situation, the economic impact is terrible which was about $2.5 million. We think we can improve this model from two aspects. The first is dealing with imbalanced data. We need to control the size of undersampling manually to figure out the best number of records we should remove. Also, we can try different undersampling methods to see which one performs better. The undersampling methods can be classified into three categories: methods that select examples to keep, methods that select examples to delete, and combinations of keep and delete methods. The first category includes near miss undersampling and condensed nearest neighbor rule for undersampling. The second category includes Tomek links for undersampling and edited nearest neighbors rule for undersampling. The third category includes one-sided selection for undersampling and neighbourhood cleaning rules for undersampling. We do not talk about the details of these methods, but this is one direction we can work forward. Second, we can tune the parameters of our logistic regression to improve its performance. Obviously, the default parameters did not satisfy the accuracy requirement. We can manually set the weight of each independent variable. In the real world, the weights are absolutely not equal. We always have some elements affecting more. We can also try different solvers which are the algorithms to use in the optimization. In a word, there are tons of things we can try to improve this model and we believe the performance will be better as long as we keep working on this.

# Model 4 - K-Means Clustering

## Description

The K-Means Clustering model was created to group all observed features into clusters. The objective of the model is to evaluate interactions between features and use that information to find patterns of potential problem areas. The model does not predict anything itself, but serves as an explanation for some predictive outputs of the other three models. The model split the features into two clusters, placing each instance into one or the other. The model can be evaluated with a silhouette score which explains how separated each cluster is. The closer the score is to one, the more separated and defined the clusters are.

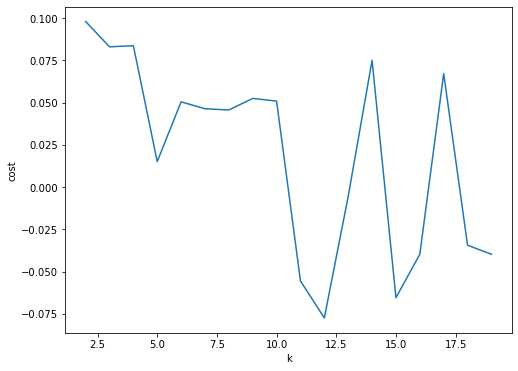
## Implementation details

As stated in the data cleaning approach, the model had to be prepared through the PySpark machine learning steps. To begin the preprocessing, we had to construct a features vector placing all attributes into one column. Since the dummy coding was done in the data cleaning, there was no need for string indexing or one hot encoding. We used the VectorAssembler function to group each feature. To normalize the features, we also ran the features through the StandardScalar function which brought all values down to scale to avoid skew.

With the features prepared, it was now time to tune the model parameters to find the best performing K-Means model. Since we were looking for the best silhouette score, we ran the model through a for loop. The for loop ran the KMeans function with k values ranging from two to 20. After each iteration the for loop would print the silhouette score. This allowed us to see which k value returned a score closest to one. Seeing that two performed the best, we were able to run the function and return cluster placements.

## Results

Since this model only served to group each instance into clusters, there is no confusion matrix associated. However, we were able to measure the silhouette score using squared Euclidean as the distance measurement parameter. The score was 0.1733. Pictured below is the graph showing the iterations through k values from two to 20.



## Evaluation

Since there was no prediction involved in this model, there is no associated economic evaluation. However, as stated above, we hope this model serves as insight for variable association when evaluating the other models.

## Conclusion

Given more time, we would have liked to tune this further. Ideally we would have made more clusters given how many columns were included in the dataset. The issues came with interpretation of each variable. It may have been smarter to separate out the parameter columns for this model to cluster those on their own. With a lack of understanding of what a lot of the columns were, we struggled to decipher how to handle each. It might have been better to look at similar categorical columns to cluster each at a factor level. The silhouette score was lower than we had hoped, but given more time and more information we could have made this model more informative. However, we do believe that this serves as a great ground level model for K-Means clustering of the dataset.