Industry Practicum Final Report
CDL IoT Use Cases

By Riu Sakaguchi, Henry Liang, Ruben Nakano, Sharika Mahadevan, Yi Chen, Yumeng Zhang, Samuel Swain

Table of Contents

Introduction ***2***

Use Case 1: Hard Drive ***3***

Data Description **3**
Exploratory Data Analysis 4
Data Cleaning 5
Data Transformation (Feature Engineering) 5

Modeling **6**

Analysis **15**
Model comparison 15

Use Case 2: Divvy Bikes ***17***

Data Description **17**
Exploratory Data Analysis 17
Data Cleaning 18
Data Transformation (Feature Engineering) 19

Modeling **19**

Analysis **21**
Model comparison 21
Feature importance & Interpretation 22

Use Case 3: MotionSense ***24***

Data Description **24**
Exploratory Data Analysis 24
Data Cleaning 25
Data Transformation 25

Modeling **26**

Analysis **27**
Model comparison 27
Feature importance & Interpretation 27

AWS Implementation ***29***

Solution Architecture **29**

Solution Cost Estimation **30**

Performance Measurements **30**

Future Considerations **31**

Bibliography ***32***

Appendix ***33***

Introduction

Project Overview:

REFiT is a one-of-a-kind cloud service that allows businesses to harness the power of the Internet of Things (IoT). It captures streaming data from these devices in real-time, augments it with static information to make it meaningful, and generates business values by making predictions that could address the woes of the industry. Hence, it provides an integration layer between the services that produce events and systems that consume these events including Grafana or Apache Cassandra.

REFIT differentiates itself from the other available options in the market by deploying Northwestern's ML research with flexibility, building a foundation on latest open-source tools, and most importantly, through the ease of scalability of ML models.

In the past, REFIT has implemented multiple use cases across numerous industries including the following:

- Agriculture: Based on sensors used in the field, REFIT can enable study of preventive maintenance or controller release of pesticides/fertilizers needed for growing crops.
- Healthcare: Based on real-time monitoring of vital signs of patients, REFIT model has predictions of development of certain medical conditions.
- Manufacturing: REFIT can be used to monitor the status of shipments, equipment, and market conditions taking preemptive actions based on predictive solutions.

Project Scope:

- Develop and implement three IoT use cases based on public data.
- Build and end-to-end solution for each use case on AWS, mimicking the general architecture leveraged in REFIT.
- Assess the potential pros and cons of implementing a streaming-based solution in AWS versus REFIT.

Use Case 1: Hard Drive

Data Description

Servers comprise of hard drive disks aggregated together to form a storage pod. BackBlaze monitored these sensors using S.M.A.R.T (Self-monitoring, Analysis, and Reporting Technology) attributes. The dataset can be retrieved from the following link; a smaller version of the same data can be obtained directly from Kaggle. The dataset collects information each day for the various hard drive disks and the data from the corresponding sensors over a long period. For the purpose of analysis, the data are analyzed over a shorter term of four months (from January 2016 to April 2016) which consists of approximately three million observations. During this period, roughly 205 unique hard drive disks have failed. The leading objective revolves around helping people at server centers identify hard drives that are close to failure to mitigate the risk of servers malfunctioning by taking corrective action promptly.

Based on *(Klein, 2016),* some of the most important S.M.A.R.T attributes that were monitored by BackBlaze for predictive maintenance are the following:

1. SMART 5 – Reallocated Sectors Count
2. SMART 187 – Reported Uncorrectable Error
3. SMART 188 – Command Timeout
4. SMART 197 – Current Pending Sector Count
5. SMART 198 – Uncorrectable Sector Count

Dataset Description:

Variable name	**Type of feature**	**Description**
date	Datetime (It is initially string but converted to datetime)	The date variable corresponds to the date.
serial_number	String	The serial number corresponds to each unique hard drive disk of each model
model	String	The model corresponds to each unique model of the model
capacity_bytes	float	The capacity_bytes correspond to capacity of hard drives in bytes.
failure	binary	The failure variables indicate whether a hard drive failed or not.
smart_raw/smart_normalized	integer	There are 90 columns of data, that are the Raw and Normalized values for 45 different SMART stats as reported by the given drive. Each value is the number reported by the drive.

S.M.A.R.T stands for Self-Monitoring, Analysis, and Reporting Technology. The smart_raw and smart_normalized are generated by sensors that are monitoring each individual hard drive. During the data analysis, most of the S.M.A.R.T attributes were null values. Such features were dropped from the analysis as these sensor readings are independent of previous readings and other hard drives.

Exploratory Data Analysis

The initial exploratory data analysis depicted strong class imbalance, with only 0.0068% failures out of all the observations.

Figure 1: Class Imbalance

Given that the likelihood of the hard drive failing increases with time, handling the class imbalance through upsampling may not be the most ideal for modelling the data. Hence, the problem was framed as a pure regression problem instead.

To find an appropriate regression problem, several discussions were held within the team and advisor in addition to referencing preexisting literature in the domain. The initial proposal was to model the failure rates (hourly or daily) over a period; however, this would require more computational resources to handle larger data (over a longer period). Hence, further research was done to identify a suitable problem. The paper *(Interpretable predictive maintenance for hard drives, 2021)* discusses modelling the data with useful lifetime as response variable, where useful lifetime is defined as time remaining before a failure (the calculation is described under Data Transformation section).[1]

Figure 2: There are a maximum of 2 failures during this period and all others have at least one failure. Hence, predictions will not be skewed.

[1] Further analysis done is attached in Appendix. It is not included here to maintain continuity. This analysis is related to identification of suitable machine learning problem to solve.

Data Cleaning

There was a myriad of rows with null values across numerous columns and rows. Thus, all columns with only null values are dropped first. Subsequently, rows were inspected for potential null values with the results indicating the absence of such rows.

Previously, the dataset would have labels of whether the hard drive would fail or not. Each hard drive moved closer towards its failure with time. Since the problem was translated to predicting the useful lifetime of each hard drive, we eliminated the time dependency between each observation. Thus, the columns with date, end_date, model name, serial number and failure (label) were dropped. Since the maximum number of failures observed per serial number during the period is 2 (*Figure 2*), the predictions will not be impacted.

Data Transformation (Feature Engineering)

The relevant response variable identified was *useful lifetime.* To find this, first the subset of all hard drive disks that had failed are tagged. Then, the *end date* is calculated for each hard drive disk. More specifically, the useful lifetime is calculated as the time between the current date and the end date of each hard drive disk.

Aside from dropping columns with only null values in the dataset and columns with date, end_date, model name, serial number and failure (label), no other transformations were applied to the dataset: a means of facilitating model interpretability. In addition, we also built a regression tree to identify the least useful predictors in the feature set.

For the feature selection, we started with a full dataset; upon building regression tree however, the feature importance metrics spotlighted multiple columns with zero importance, including SMART 198 and SMART 242 as illustrated below:

```
smart_9_raw             0.251
smart_9_normalized      0.148
smart_7_normalized      0.116
smart_194_normalized    0.087
smart_197_raw           0.084
smart_12_raw            0.082
smart_194_raw           0.058
smart_4_raw             0.053
smart_3_normalized      0.049
smart_5_raw             0.031
smart_3_raw             0.022
smart_199_raw           0.014
smart_1_normalized      0.003
smart_1_raw             0.002
smart_5_normalized      0.000
smart_198_raw           0.000
smart_198_normalized    0.000
capacity_bytes          0.000
smart_199_normalized    0.000
smart_241_raw           0.000
smart_240_raw           0.000
smart_10_raw            0.000
smart_197_normalized    0.000
smart_188_raw           0.000
smart_12_normalized     0.000
smart_10_normalized     0.000
smart_7_raw             0.000
smart_4_normalized      0.000
smart_242_raw           0.000
dtype: float64
```

The above feature importance was cross-checked with the plot of the regression tree to ensure that features with zero importance scores did not appear in the regression tree output. The features which did not appear in the regression tree were also dropped. This includes: 'smart_5_normalized', 'smart_198_raw', 'smart_198_normalized', 'smart_199_normalized', 'smart_241_raw','smart_240_raw','smart_10_raw', 'smart_197_normalized', 'smart_188_raw',

'smart_12_normalized','smart_10_normalized','smart_7_raw','smart_4_normalized', 'smart_242_raw'. It is also important to note that based on modeling in paper *(Klein, 2016),* , these features were also removed to avoid autocorrelation and focused the problem as a regression problem. Thus, the models were built using only the remaining features.

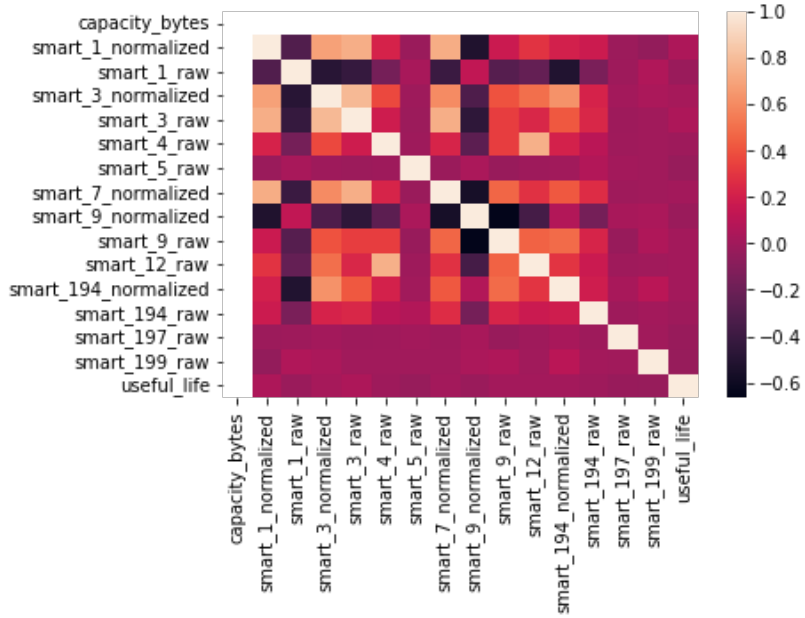


Figure 3: Correlation Heatmap of all features with response

Based on dataset description, it was concerning to maintain both raw as well as normalized data for a smart attribute. For example, both smart_9_raw and smart_9_normalized are included in the predictor. However, it was seen that there is no strong correlation between such pairs. In case of smart_9_raw and smart_9_normalized, the correlation is around 0.6, which is not strong. To avoid the risk of multicollinearity, we thus began modelling with Ridge Regression as benchmark, rather than a traditional linear regression.

Modeling

The best models identified are XGBoost, Decision Tree and CatBoost. The 'Modeling' section outlines all the models that are built. While model interpretation through ALE Plots and Feature Importance is studied for all models, it is not placed in this section. The section focuses on Model Interpretation only for the top 3 models.

It is worth noting that based on the understanding of the dataset, each S.M.A.R.T attribute does not influence the other nor does it impact the influence of another on the response. Hence, the second order interaction ALE Plots are not mentioned in the modeling section. This understanding is also corroborated by the low correlations seen in *Figure 3*. ALE Plot based on non-standardized data is used to study the influence of each predictor. Since the values are small in standardized data, it is difficult to visualize the differences.

Ridge Regression

Ridge regression is the first model we tried since it comes from simple linear regression with penalty on model complexity. As previously stated, we saw few features with correlation of 0.6. We opted to use ridge regression over traditional linear regression to reduce the risk of multicollinearity. With L2 penalty, we can start with this simple model and avoid overfitting problems. The ridge regression model is also our baseline model.

When building ridge regression model, the only hyper-parameter we tuned was the penalty factor alpha. The range for alpha was from 1×10^{-5} to 100. After tuning through cross-validation, we found that the ridge regression with 100 penalty factor was the best model.

With our best hyper-parameters, we then fitted it with all our training dataset and reached test R^2 = 0.0077, which is low. It showed that there were still a lot of nonlinearities not captured by the model.

K-Nearest Neighbors

Why KNN: We also tried K-Nearest Neighbors which was a simple and intuitive model. This model will be really effective when the dataset has a large number of rows and small number of columns. And there is no training step for this model. So, the running KNN will be pretty efficient.

Hyperparameter: When building K-Nearest Neighbors, the only hyper-parameter that matters was the number of neighbors. When the number of neighbors is small, we may obtain model with high accuracy but may also encounter overfitting problems. But if the number of neighbors is large, then the model's accuracy will be low. So, the number of neighbors influences is really significant.

We also did cross-validation to tune the number of neighbors. The result showed that KNN with number of neighbors = 14 outperformed other models with cross validation R^2 = 0.1133. Then we fitted the model with the best parameters using the whole training dataset and obtained a test R^2 = 0.9501. Due to the very low CV score, we are suspicious whether the test data has very similar samples as training data. Hence, it is also not recommended.

Local Linear Regression

Why local linear regression: With important features obtained in tree model, SMART_9_RAW, SMART_9_NORMALIZED, SAMRT_7_NORMALIZED, SMART_194_RAW, we also wanted to try the local kernel regression on our dataset. In class, we learned that local kernel regression can let us estimate response variable with its neighbor points better than KNN as it avoids boundary bias. We initially planned to run local linear regression in Python. But the "rpy2" package required could not run successfully on our computer. So, we just run local kernel regression model in R with loess function.

Hyperparameters: The hyperparameters required for tuning are degree: control for specific kernel we used, and span: related to the smoothening parameters. After tuning with cross-validation, we found that model with degree = 1 and span = 0.3 outperformed other models with $R^2 = 12.07\%$.

We also fitted the best model on test dataset and obtained 14.89% test R^2 which is worse than the results of XGBoost. So, we're not going to analyze the main effects of local kernel regression in our report.

Generalized Additive Model

Why GAM: GAM was explored because the Local Regression model, only allows for 4 individual predictors. GAM allows to use all. The dataset after feature selection has only continuous data. So, it is suitable for GAM. The generalized additive model models the response as the sum of functions of one or two predictors at the same time.

$$\text{GAM: } Y(\mathbf{x}) = \alpha + f_1(x_1) + f_2(x_2) + \ldots + f_k(x_k) + \varepsilon$$

$$\text{or sometimes } + \sum_{j=1}^{k} \sum_{l=j+1}^{k} f_{j,l}(x_j, x_l)$$

When running GAM model, there were no hyper-parameters we needed to tune explicitly. So, we only used cross-validation here to ensure model's stability and to have a comparable CV Model Performance R^2 score.

The cross-validation R^2 is –0.8662 which is negative, showing that this model is even worse than null model. So, we will not display the ALEPlot for this model in the report.

Projection Pursuit Regression

Why PPR: Although by definition of features, that they do not influence the impact of one on the response, we still tried to capture interaction through PPR and confirm if our understanding is correct. Though GAM can capture them, it might not be so efficient to include all the different interaction terms in GAM. So, we also tried Projection Pursuit Regression which will use 1-D function of linear combination of all predictors rather than only one predictor.

$$\text{PPR model: } Y(\mathbf{x}) = \alpha + f_1(\boldsymbol{\beta}_1^T\mathbf{x}) + f_2(\boldsymbol{\beta}_2^T\mathbf{x}) + \ldots + f_M(\boldsymbol{\beta}_M^T\mathbf{x}) + \varepsilon$$

$$= \alpha + f_1(v_1) + f_2(v_2) + \ldots + f_M(v_M) + \varepsilon$$

$$\text{where } v_j = \boldsymbol{\beta}_j^T\mathbf{x} = \beta_{j1}x_1 + \beta_{j2}x_2 + \ldots + \beta_{jk}x_k \text{ (linear combo of } x\text{'s)}$$

Considering the format of PPR model, we could easily know that M, the number of terms is one of the important hyperparameters we should care about. So, we tuned this hyperparamter in our cross-validation process. Since running PPR is pretty computationally expensive in our local machine, we only chose two values for the number of terms, 10 and 15. After tuning, the best number of terms was 10 but with a negative Cross-validation R^2. It showed that our best PPR model was even worse than null model. This could be primarily due to insignificance of interactions or because we needed to test more values for hyperparameters.

Regression Tree

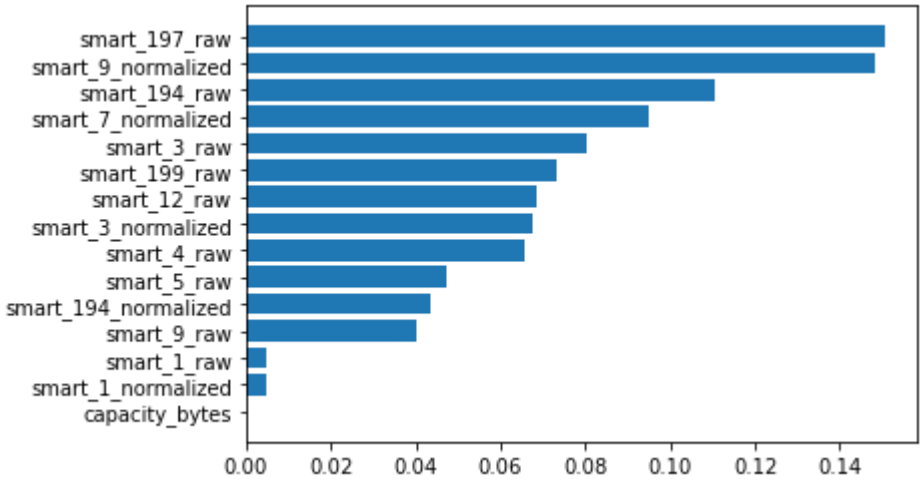
Why Regression Tree: Regression Trees are one of the simplest non-linear models that can be used. With each split in the tree, the loss function (in this case, SSE) is optimized. Hence, it follows a greedy approach. In the case of the hard drive dataset, it was interesting to use the regression tree, since there are few unique values of each S.M.A.R.T attribute.

Hyperparameter: The most important hyperparameter is ccp_alpha.

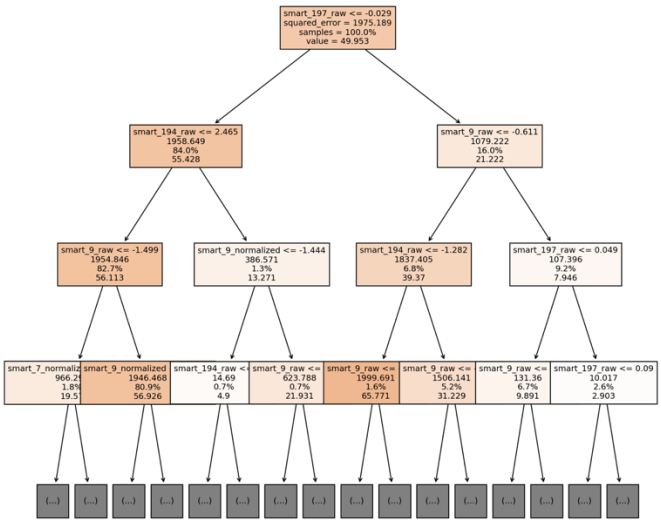
Based on CV, the optimal ccp_alpha was chosen. The model performance in terms of R^2 score for CV is 88.2% and for test is 89.6%.

Model Interpretation:

The feature importance plot is made to understand the tree better.



The most important features for predicting response with regression tree are smart_197_raw, smart_9_normalized, smart_194_raw and smart_7_normalized. Below visualization depicts is tree and its splits.



Random Forest

Why Random Forest: While we did receive good performance on Decision Tree, we decided to implement the Random Forest algorithm, which in most cases has better predictive power than Decision Tree. This is because it adds randomness by

- *Building each individual tree over a bootstrap of the original dataset*
- *Selecting only a subset of individual features at each split*

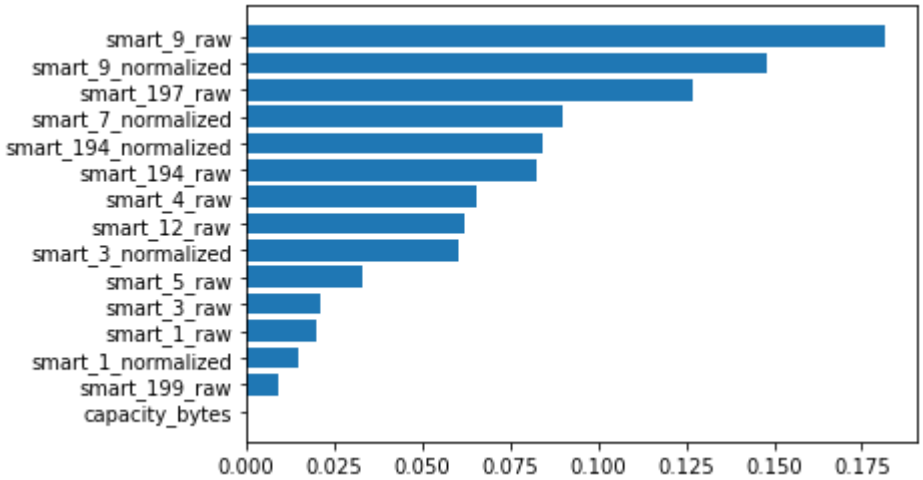
Hyperparameters: The most important hyperparameters for the random forest algorithm are:

- The depth of each individual tree (min_samples_leaf, max_depth) - If each tree is grown to large, it could lead to overfit.
- The maximum number of features to be selected at each split (max_features) - By selecting a smaller subset of features, there is more randomness introduced in each tree.
- The number of individual trees in each random forest (n_estimators) – This is not very important. We only need to ensure that there are sufficient number of trees to make meaningful predictions. (More number of trees were tested; however it was taking a long time to finish running. Hence, only n_estimators = 500 is tested.

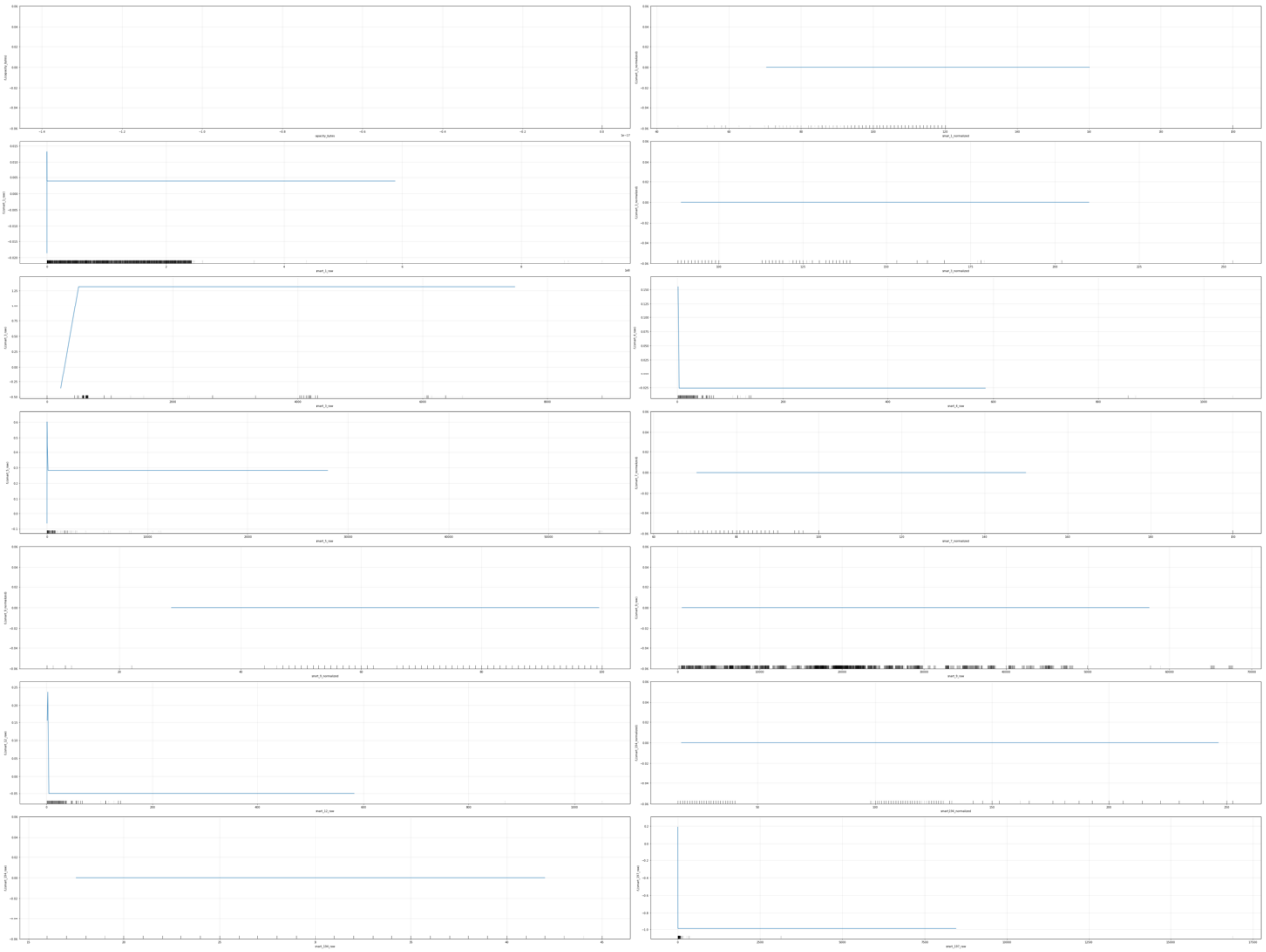
Based on cross-validation, the optimal hyperparameters were chosen. The model performance in terms of R^2 score for CV is 73.5% and for test is 73.51%. This was surprising to see. We concluded that this could be happening because there are only few important predictors and by random subsetting of variables, there are multiple weak splits as well as poor trees developed. If the ALE plot is studied carefully, then many of the predictors did not have strong influence on response. The next step would be to increase the number of trees to overcome this. However, this would come at a computational cost. Hence, we decided to explore other algorithms.

Model interpretation:

Please find below the feature importance plot.



Based on the Feature Importance plot, the most important predictors are smart_9_raw, smart_9_normalized, smart_197_raw and smart_7_normalized. The ALEPlot is also done to understand impact on useful_life.



In the ALE Plot, most of the important features appear to have minimal influence. This can be because each individual tree in the random forest is building over a bootstrap sample and each split chooses a random subset of features. Hence, it is not possible to ascertain the influence of one predictor.

XGBoost

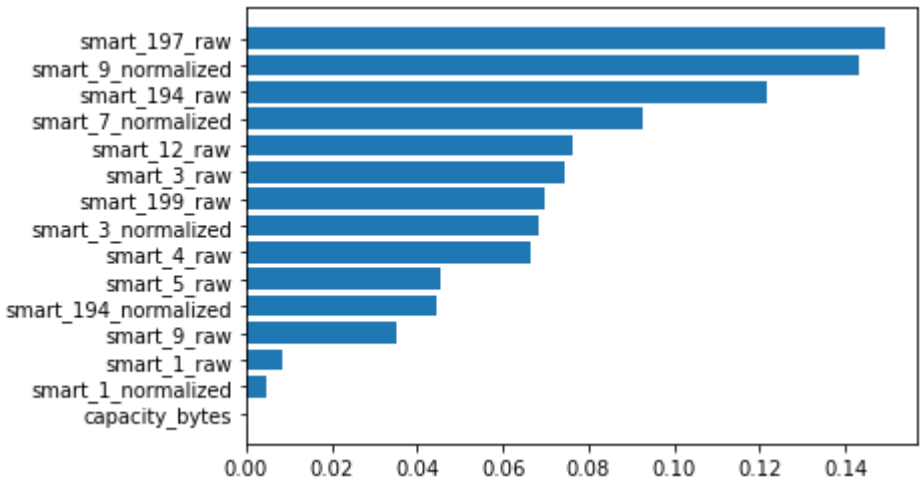
Why XGBoost: Boosting algorithm builds predictions sequentially by trying to account for remaining error in the previous prediction built. Although Gradient Boosting was tested on the dataset, it is seen to perform poorly on the CV and Test set. Gradient Boosting trees, unlike random forest, are subject to overfitting. XGBoost overcomes this by using regularization parameter.

Hyperparameters: The most important hyperparameters for XGBoost algorithm are

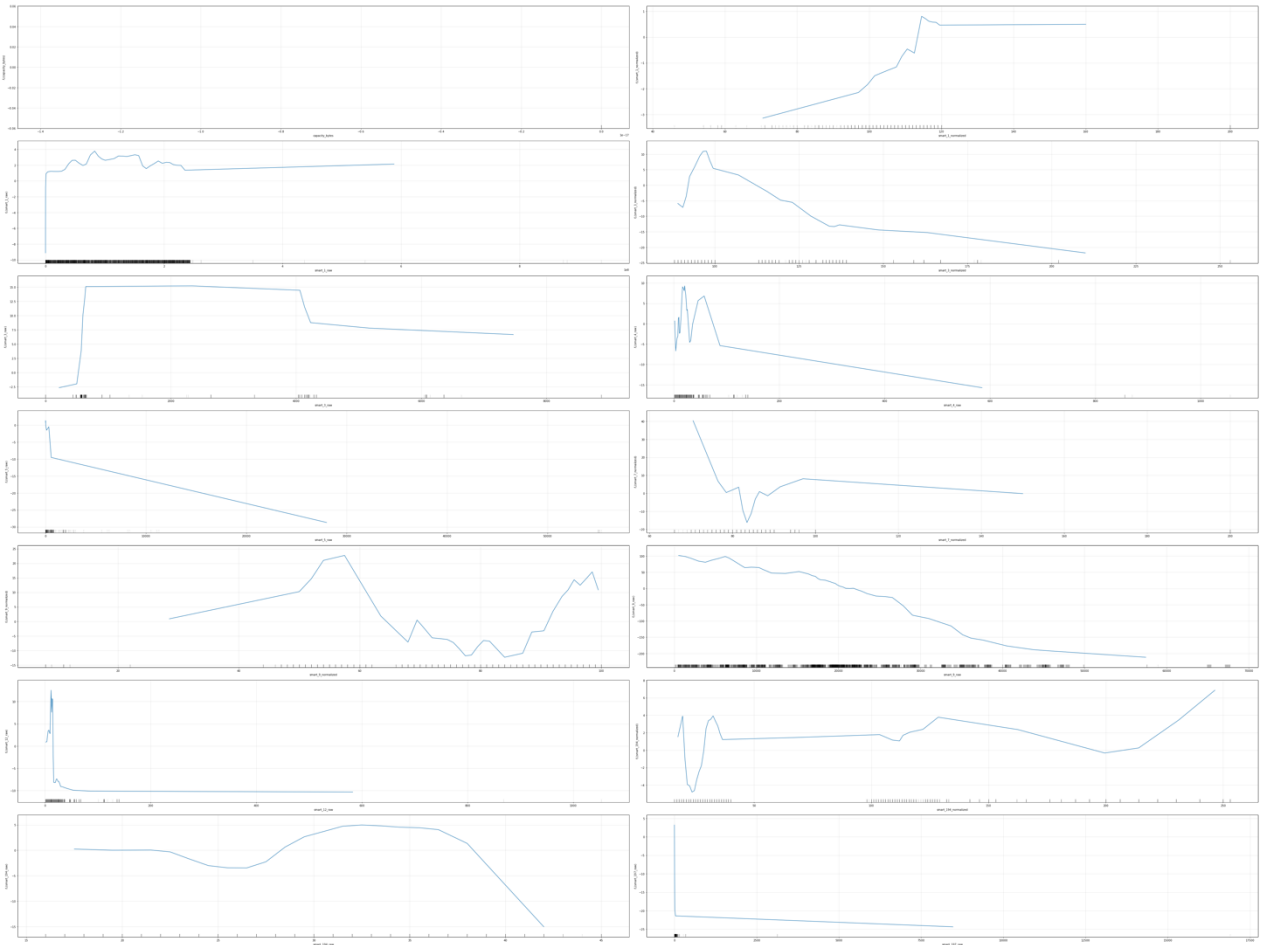
1. learning_rate- The Learning rate helps to reduce the complexity.
2. max_depth – Gradient Boosting algorithm is built on proving predictions of previous poor predictions. Hence, the max_depth of boosted trees are usually limited between 4 and 8.
3. 'max_leaves' - The max_leaves also controls for the depth of each individual tree.
4. n_estimators – This controls the number of trees to be included in the XGBoost algorithm.

Model Interpretation:

The XGBoost model outperformed all other models in terms of accuracy (measured by R2 score). It had R^2 score of 95.51% on CV and 95.81% on the test set. Below is the feature importance plot:



Based on the above feature important plot, it is visible that the top 4 most important predictors are smart_197_raw, smart_9_normalized, smart_194_raw and smart_7_normalized. Now, the Main effects ALEPlot is used to study how it is affecting the response.



ALE Plot based on non-standardized data is used to study the influence of each predictor. Since the values are very small in standardized, it is difficult to visualize the differences.

Based on the ALE plot, the following summarizes the impact of the most important features:

1. smart_197_raw: There are a lot of zeros in the dataset. It is seen that the response (that is, useful life) is maximum at this point. This implies that 0 indicates that the hard drive is functioning well. As soon as the value starts to increase, useful life starts to decrease. So, any values of smart_197_raw beyond 0 means that the hard drive is moving closer to failure.
2. smart_9_normalized: It is seen that smart_9_normalized is indicative of good useful life around ~55 and ~95. But it is declining for other values.
3. smart_194_raw: smart_194_raw is seen to be indicative of the most useful life around ~35. For low values lesser than this, it is likely to move towards failure. So, decline in smart_194_raw is indicative of failure.
4. smart_7_normalized: This feature is indicative of failure when the values start to increase (similar to smart_197_raw).

CatBoost

Why CatBoost: CatBoost algorithm is chosen to be tested for the section of model not taught in class. If each S.M.A.R.T attribute is studied, there is a limited number of unique data points. So, we wanted to check if CatBoost will perform well with this dataset. Apart from this, the key advantages of CatBoost are that it reduces overfitting through gradient-based regularization and per-iteration learning rate (meaning it uses a different learning rate at each iteration). It does not take a long time to run the code.

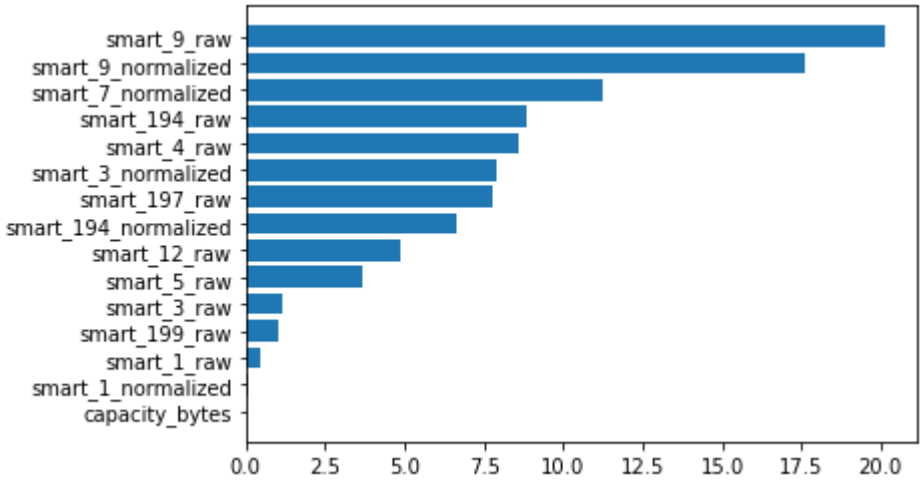
Hyperparameters: The most important hyperparameters that are tuned in cross validation are

- max_depth: The max_depth controls the depth of each tree
- n_estimators: This controls the number of trees.

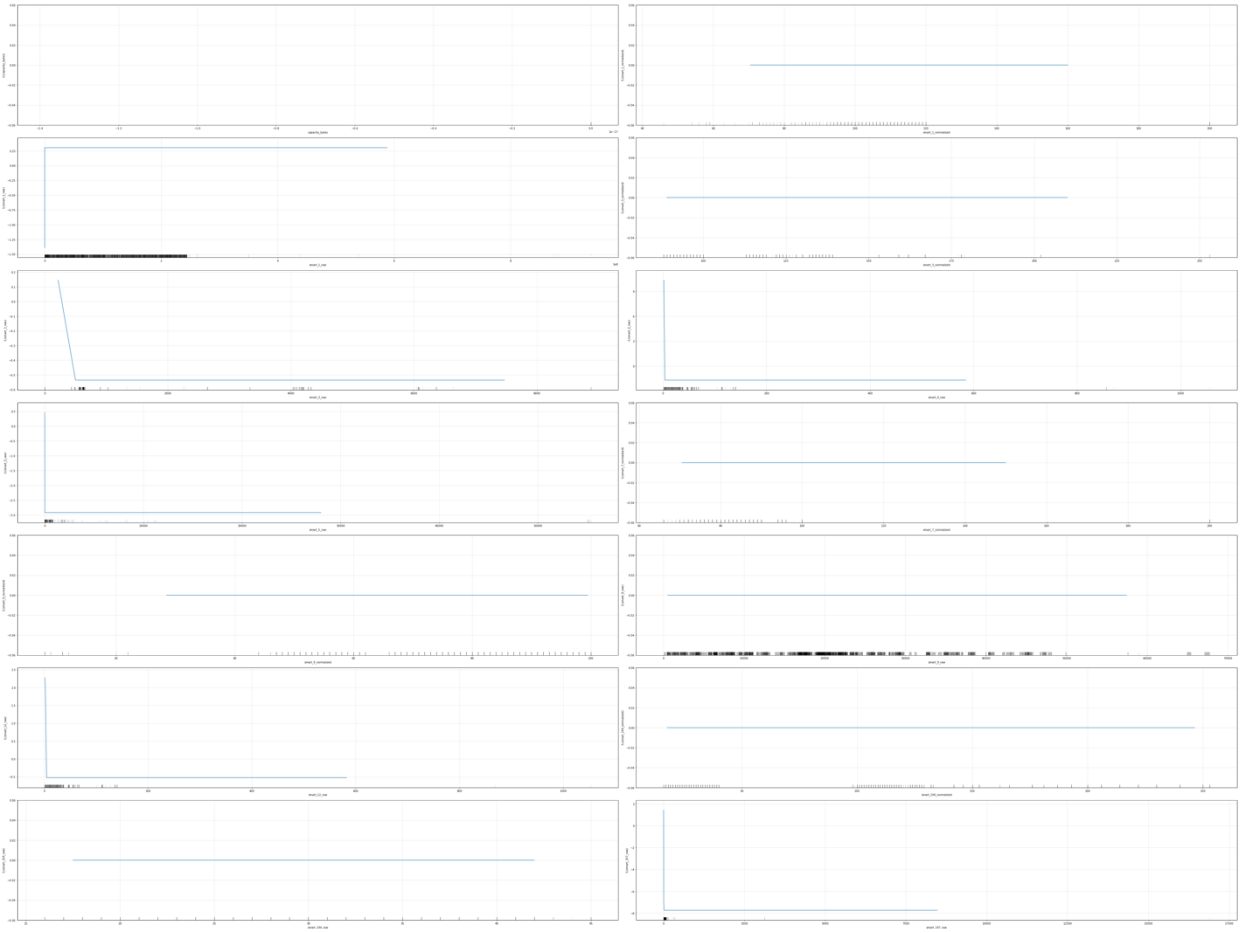
CatBoost algorithm was one of the top 3 models in terms of performance. It had R^2 score of 86.27% on CV and 85.88% on Test set.

Model Interpretation:

Below is the plot of feature importance:



From the feature importance plot, the most important features are smart_9_raw, smart_9_normalized, smart_7_normalized and smart_194_raw. The ALE plot is studied for these predictors. Please find below the corresponding Main Effects ALE plot.



ALE Plot based on non-standardized data is used to study the influence of each predictor. Since the values are small in standardized data, it is difficult to visualize the differences.

In the ALE Plot, it is seen that most of the key features (smart_9_raw, smart_9_normalized, smart_7_normalized) have minimal influence. This can be because each individual tree in the CatBoost has a different penalty term due to its implementation of per-iteration learning rate. Further, the Feature Importance plot is based on the loss function metric being optimized. Thus, this can be one of the reasons the ALE plot does not agree with Feature Importance plot. When studying smart_197_normalized, it is seen that apart from zero, all values are indicative of failure.

Neural network

Why Neural Network: Due to the nonlinearities in our dataset, besides tree models, we also thought about using neural network to make predictions. As we have learned in class, neural networks can handle multiple types of nonlinearities with different number of layers, the number of nodes in one layer and then provide high prediction accuracy.

Hyperparameters: Due to the complexity of neural network, the hyperparameters we tuned here including:

- Hidden_layer_sizes: decide the number of neurons in the hidden layer
- Solver: decide the weight optimization method
- Alpha: strength of L2 regularization term

After tuning with cross-validation, model with alpha = 0.05, hidden_layer_sizes = 28, solver = 'sgd' outperformed others with CV R^2 = 0.5766. We also fitted the best model with test dataset and obtained test R^2 = 0.5350, which was worse than XGBoost.

Analysis

Model comparison

Model	**Best Hyperparameters**	**CV R^2 Score (%)**	**Test MSE**	**Test R^2 Score (%)**	**Time (ms)**
Ridge Regression	'alpha': 100	-1.08	1927.25	0.76	307
Regression Tree	'ccp_alpha': 0.00275 'min_samples_leaf':5	88.2	200.67	89.6	25.6**
Random Forest	'max_depth': 12, 'max_features': 3, 'min_samples_leaf': 2	73.50	514.43	73.51	1850
Boosted Tree	'learning_rate': 0.04, 'max_depth': 4, 'min_samples_leaf': 2, 'n_estimators': 150	53.63	965.65	50.27	1060
XGBoost	'learning_rate': 0.1, 'max_depth': 8, 'max_leaves': 4, 'n_estimators': 200	95.51	81.244	95.81	21500
Neural Network	'activation': 'logistic', 'alpha': 0.01, 'hidden_layer_sizes': 28, 'max_iter': 1000, 'solver': 'sgd'	57.37	828.09	57.36	274000 (4min 34s)
KNN	'n_neighbors': 14	11.32	96.22	95.04	11700

GAM	Hyperparameters are tuned internally	-86.62	1548.87	20.25	8610
PPR	'r': 15	-977365734	2298.85	-18.36 *(negative implies that null model is better)*	2118000 (35min 18s)
LOESS	'span': 0.3; 'degree': 1	12.07	1695.41	14.89	600
CatBoost	'max_depth': 5, 'n_estimators': 300	86.27	274.065	85.88	3680

** - *Decision Tree time taken is computed as time to overgrow the initial tree and then prune back later*

Model Performance is evaluated in terms of R^2 score over Cross Validation and Test set. MSE over test set is also reported. Model Performance also needs to account for time taken to train the model. In this case, the time taken to build the final best model is evaluated.

From the above table, the best performance in terms of R2 score (which is a metric that captures how well the model explain the response) is:

- XGBoost
- Decision Tree
- CatBoost

Only these 3 models will be evaluated when comparing the execution time. Decision Tree is much faster by an order of 10^{-2}, followed by CatBoost and XGBoost. The latency requirement for this use would be daily. Hence, it would be fine to use the XGBoost model. Further analysis is described in the next steps.

Caveats of the model

- The dataset is limited to January 2016 to April 2016. This itself has over 3 million observations. Hence, it was computationally expensive to include more. Due to this choice made, it may not give a comprehensive view of the lifetime of all models.
- During this time frame, there are limited number of unique values for some features. Once the data is ready to be evaluated on cloud platforms (like AWS or REFiT), this will be checked.
- Capacity bytes has almost no impact on the response variable. Going forward, it will be dropped.

Use Case 2: Divvy Bikes

Data Description

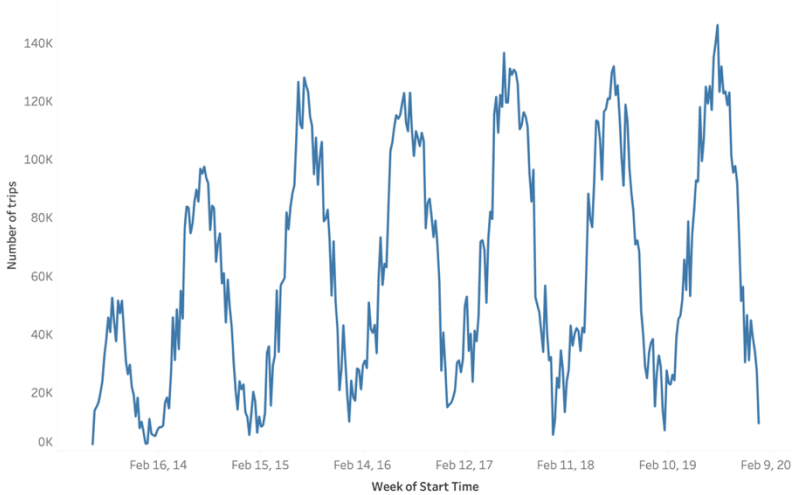
The dataset for this particular use case was downloaded from the Divvy bikes website and comprises data from August 2013 to December 2019. Each row in the dataset contains information about a specific trip of each individual user – each trip has a specific start and end time, a start and end station, the bike id, the type of user, the gender of the user, and the birth year of the user. It is worth pointing out that some of the data on the users stopped being collected throughout the years (this was not an issue for our analysis since the columns used for modeling remained consistent during the period).

The final data array used to fit the models was built by aggregating the total number of unique trips in daily intervals. Therefore, the dataset only has 2 columns: one is the date and the other is the total number of trips on the whole Divvy system for that day. Since it is a time series model, both the response and the predictor are the number of trips. Note that some models required the creation of different features – this will be thoroughly explained in the section 'Data Transformation' below.

Exploratory Data Analysis

Since the goal of our models will be to predict daily trip rides, we focused our exploratory data analysis on understanding the seasonal patterns in the data.

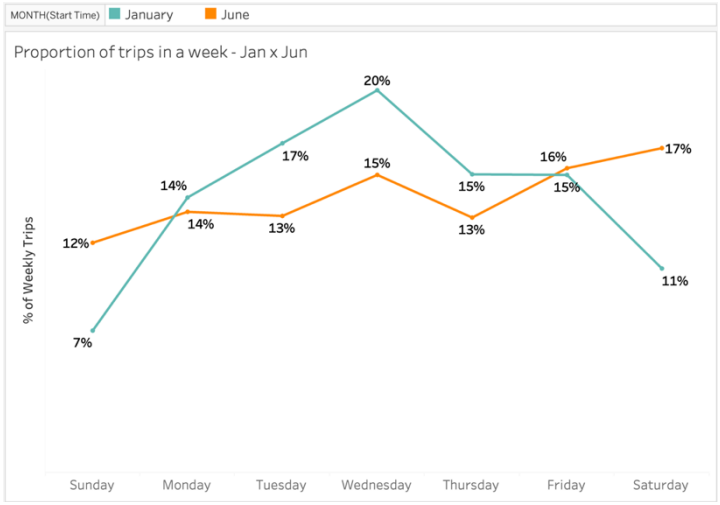
Initially, we investigated the yearly seasonality to explore how the number of trips might change for each month. To do this we plotted weekly total trips as it's easier to see a pattern with more aggregated data. The figure below clearly shows there is seasonal variation: most of the trips happen during the summer months, when temperatures are higher. Moreover, we also notice an upward trend from 2013 to 2015, after which the number of trips seems to plateau.



Yearly seasonality

We then proceeded to investigate intraweek seasonality and discovered that it also changes throughout the year. During colder months, most trips happen between Monday-Friday. This is most likely the case because users are renting bikes to commute to and from work, and there are fewer people that rent bikes for leisure purposes during the weekend.

During warmer months, on the other hand, the distribution of trips in a week is a lot flatter and Fridays and Saturdays stand out as the weekdays with most trips. This probably happens because there is a much larger number of people that use the system for leisure when temperatures are higher.



Comparison between trip distributions across the week in January and June

Hence, we see that our data contains multiple seasonal components (and interactions between them) that will pose challenges to the models we fit. This will be further discussed in the modeling sections of this report.

In addition, we also conducted outlier analysis that showed how holidays can have an impact on demand for bikes. For the purposes of this project, however, we chose not to remove them and see how the different models deal with them.

Data Cleaning

The original dataset downloaded from the Divvy website was reasonably clean from the start, however, getting the data ready for modeling still presented some challenges. First, the data was split into several csv files – each file usually contained data for a particular month, but for some years only aggregated files for a quarter were available. Second, as mentioned above, data on users (gender, birthyear) stopped being collected after a couple of years. This meant that the number of columns could change in each of the csv files, thus making joining them a harder task. In addition, we noticed that station ID numbers had minor inconsistencies. However, since our goal is to predict the total number of trips regardless of stations, this was not a major issue.

Data Transformation (Feature Engineering)

Since we are dealing with time series data that clearly has seasonality, we chose the two types of models covered in class that can handle this type of data: Holt-Winter's and SARIMA. In addition, we also modeled the data using a type of neural network called Long Short-Term Memory (LSTM).

No feature engineering was required for Holt-Winter's and SARIMA models apart from aggregating the data to daily trip numbers. For LSTM, however, the final dataset used to fit the model was built through the following procedure:

1. Creating a column that computes how many days have passed since the start of the Divvy bike system ('days_since_start'). This was done by subtracting every single date by the initial date in the dataset.
2. Turn the column 'days_since_start' into sine and cosine signals.
3. Dropping the column 'date' and 'days_since_start' from the dataset.

Thus, the final data frame used to fit LSTM contained 3 columns: the sine and cosine signals of days since start for a particular date, and the total number of trips associated to that date.

Modeling

Holt-Winter's

The first method in predicting the number of Bike rides at the daily level in Chicago involved Holt-Winter's forecasting, incorporating the full stack of time series components: level, trend, and seasonality. With the cyclical nature of the data indicating strong seasonality a given, the gradual increase in the overall level corroborates the importance of accounting for trend. The initial exploratory data analysis also helps inform the model specification: an additive trend and seasonality combination. As with the earlier SARIMA model, the number of seasonal periods was defined as 12 to reflect the monthly fluctuations in the number of Divvy Bike rides in Chicago.

To ensure meaningful comparisons across different methods, the same training set, defined as data before January 16, 2018, was employed to fit the optimal Holt-Winter's model. Notably, cross-validation and other forms of deliberate hyperparameter tuning was unnecessary as the algorithm internally estimates the optimal parameters. Implementation of the final Holt-Winter's model found a smoothing coefficient of 0.359, a trend coefficient of 0.001, and a seasonal parameter value of 0.028. In addition, individual seasonal coefficients were also extracted corresponding to the 12 months defined in line with the seasonality present. Future forecasts for the next two years were produced, with performance tested directly against the test set. An incredibly high test MAPE value of 365.95% sheds light upon the inability of the Holt-Winter's model to accurately forecast the daily number of trips.

SARIMA

Why SARIMA: A divergent approach considered for the Divvy Bikes use case in modeling bike trips involved SARIMA: an extension of base ARIMA model incorporating seasonality effects. Such distinction is particularly crucial given the clear seasonality trend dictating the number of rides. The data was first separated into a training and test set with a specified split date of January

16, 2018; the former contains data prior to the split date while the latter comprises data subsequent the partition. The nature of the separation ensures that the temporal relationship between the observations is preserved: a fundamental assumption in time series analysis. The training and test set comprises of data between 2013 and 2018, and 2018 to 2020, respectively.

Hyperparameters: The SARIMA algorithm features three distinct hyperparameters that control the number of lag variables, size of the moving window average, and the stationarity of the data. With traditional *k-fold* cross-validation not a possibility due to the randomization of rows, a further training and validation split was constructed from the initial training set for hyperparameter tuning. More specifically, a grid of *p, d,* and *q* values were first defined together with a *seasonality order* of 12 to represent the monthly fluctuations. The MAPE was then calculated on the validation set for SARIMA models fit on each of the 72 possible combinations. The optimal model employed an order of *(2, 1, 4),* simultaneously producing the lowest MAPE value of 69.84%.

Although a differencing parameter of $d = 1$ produced the lowest MAPE, the initial exploratory data analysis and stationary nature of the data suggests $d = 0$ as a potentially better alternative. The notion of SARIMA models often suffering significantly in forecasting performance combined with the simplicity of the tuning process hints towards the latter as the more well-informed choice. Hence, SARIMA (2, 0, 4) was selected as the preferred model and fitted on the entire training data. Finally, future forecasts were generated from the optimal model and juxtaposed with the true number of trips in the test set, yielding a test MAPE of 71.83%.

LSTM

The third method considered for the Divvy Bikes demand forecasting problem was LSTM. LSTM stands for Long Short-Term memory and is an extension of recurrent neural networks (RNN). RNN is incredibly useful in dealing with sequential data as it allows previous outputs of nodes to affect subsequent inputs to that node. However, the downside of RNN is that it fails to "remember" parameters learned in previous inputs due to the exploding and / or vanishing gradient problem. LSTM improves upon the vanilla RNN by implementing more sophisticated input, output, and forget gates as well as solving the exploding and / or vanishing gradient problem. By doing so, LSTM can learn long-term dependencies. Long-term dependencies are important because of the seasonality in the Divvy Bikes demand, and the model should "remember" previous inputs.

The LSTM model features seven hyperparameters that control the length of the input window, the offset, and the length of the prediction window. The input window length d is the number of days of trip data that the model will consider, the offset o is the number of days into the future that the model will predict, and the prediction window length p is the number of predictions that the model will generate. This is essentially the lag variable. That is, the model takes d number of days as input, predict p labels o days into the future. Then, the input window of size d slides to include the entire training dataset. The remaining three hyperparameters are related to the number of layers, nodes, dropout probabilities, and learning rate. Research has shown that one LSTM layer is sufficient to learn most complex features, and dropout layers are used to prevent overfitting. Since the data is sequential and the sequence is important, there is no real way to do cross validation for grid search for the optimal number of nodes per layer and the learning rate. They were thus chosen through trial and error. The downside of this method is that the total window length, $d + o$, must be no greater than the number of rows of the test set to generate any type of forecasting and

evaluation using the test set, which is 238. Feature engineering was conducted in order to take advantage of the periodicity of the demand. The number of days since the start was broken down into its respective sine and cosine components.

Because of the row-wise correlation present in time series, traditional cross validation is not possible. Thus, a 70/20/10 train/validation/test split was conducted prior to model training. The specific dates for splitting are discussed above. To speed up training, a batch size of 32 was used. The size of the input window was continuously adjusted such that the model is able to learn the trends in the demand of Divvy Bikes, and each window size was evaluated on the validation set. The prediction window and offset were both set at seven days (i.e., the model predicts demand for each day for seven days into the future). The lowest test MAPE was 22.86% using an input size of 231 days. There was one layer with 32 LSTM nodes, and one dropout layer with probability of 0.2, and a learning rate of 0.005.

Analysis

Model comparison

Model	**Test MAPE**	**Time**
Holt-Winter's	365.95%	0.3 secs
SARIMA	71.83%	40.3 secs
LSTM (GPU Boosted with 1080ti)	22.86% (see below for caveat)	1.0 sec

Three approaches were considered for the Divvy Bikes time series analysis: to predict the number of daily bike rides in Chicago. The first and perhaps most naïve method, Holt-Winter's forecasting, produced dramatically unfavorable future predictions. Specifically, an overestimation of a positive trend around the final observations in the training set engenders a tremendous degree of extrapolation: the forecasts are likewise significantly overestimated. In addition, the cyclical waves representing the projected seasonality appear far smaller than the monthly fluctuations apparent in the original data. Thus, the Holt-Winter's model likely learned a smaller seasonality trend, failing to capture the larger monthly seasonality present.

The seasonal ARIMA method produced a test MAPE far more tolerable from a forecasting perspective than the Holt-Winter's model alongside reasonable computational speed. However, the final SARIMA model fails to predict a high number of bike rides throughout the entirety of future dates. Consequently, forecasts are underestimated overall besides the months with a drastically lower number of bike rides as a natural exhibition of seasonality effects. While the differencing parameter is likely not the issue given the considerably stationary nature of the data, higher order lag terms are perhaps needed to accurately forecast the high spikes in bike rides. A final mention regarding SARIMA surrounds the granularity of seasonality in the data. Provided that weekly and even daily fluctuations exist in the number of bike rides, the incapacity of the model to capture multiple dimensions of seasonality reveals a patent downside.

The LSTM model achieved a test MAPE of 22.86%. However, the validation MAPE was 129.35%. This large disparity is a direct result of the limitations of the sliding window technique. The length

of the test set was 238 data points, so the total length of the window (input + prediction) must be less than or equal to 238. Another problem arises because the input window must be sufficiently large such that the model is able to learn those long-term dependencies present in the data. Thus, there exists a dilemma between choosing a wide enough input window and choosing a prediction window that makes business sense. The final input window was chosen at 231 days, and the prediction window seven days. Because of such a short prediction window, the MAPE might not be a reliable estimate for forecasts beyond the period of the test set. The seven-day forecast can be seen in the appendix as Plot 1.

The model fails to predict the larger variations in the demand by day, and it suffers from the same issue as SARIMA of being unable to predict trends at a more granular level. Despite this, the model seems to do fairly well in terms of capturing overall trend. To do this, the input window was set at 365 days to capture yearly trends, and the prediction window was also set to 365 days (into the future). An example of the predictions versus labels can be seen in Plot 1 in the appendix.

There is room for improvement, however. Because bike rides are heavily dependent on the weather (e.g., poor weather would lead to lower rides), including weather data such as wind speed, precipitation, and weather type would be useful in helping the model learn the large fluctuations in demand day over day and the more granular trends in the demand. Moreover, because many Divvy Bike stations are close together, there exists the possibility of other stations cannibalizing the demand of a particular station. Thus, demand of stations within a one-mile radius of a given station can be added as additional regressors.

Considering all factors, the overall best model for Divvy Bikes demand forecasting is LSTM. The MAPE was by far the lowest out of the three models at 22.86%, though the seven-day forecast may not be reliable beyond the test set time period as explained above. However, comparing Plot 1, which was the forecast by LSTM, and Plot 2, which includes the forecasts of SARIMA and Holt-Winters, the LSTM model is much better at capturing overall trends than the other models. Both plots can be seen in the appendix. Note that the running time of the LSTM model cannot be compared with the running time of SARIMA and Holt-Winters due to using GPU boosting for LSTM training.

Feature importance & Interpretation

The SARIMA model lacks traditional features leveraged in typical supervised machine learning models; rather, the algorithm considers lag coefficients and lagged forecasts errors to render predictions on the response variable. Hence, feature importance has little relevance in such context, particularly since all lags prior to the hyperparameter specified in the model are necessarily incorporated in the model. Nevertheless, coefficient values alongside the corresponding p-values for the preferred SARIMA model are displayed in the Appendix. The multiplicative effect of several auto regressive and moving average coefficients combined with the seasonality component appear significant at the 5% alpha level, suggesting a sense of rationality in its inclusion. However, a detailed interpretation becomes particularly involved given the context of the variables and the multiplicative nature.

Tantamount to SARIMA, the Holt-Winter's algorithm similarly diverges from the general domain of feature importance interpretation given that only the response and appropriate lag terms are utilized for model fitting: a typical property of time series analysis. However, the parameter and seasonal coefficients facilitate a level of interpretation of the model. An alpha coefficient of 0.359 suggests that a moderate-sized window is considered for estimating the level; a smaller value indicates a larger window and more smoothing while a larger value simulates greater decay. Especially low values of smoothing trend and seasonal coefficients spotlight the selection of a notably large window. In the context of the seasonal values, positive coefficients capture high volume month; negatives, on the other hand, highlight periods that observe a general drop in the number of bike rides. A value of 534.497 for May and –755.389 for November spotlights the hottest and driest months in the year for Divvy bikes, respectively.

On the other hand, LSTM is a highly sophisticated neural network model with recurrent nodes and input, output, and forget gates. Because of this, it is a fully black box model with no real interpretability, despite being able to extract the values of the parameters.

Use Case 3: MotionSense

Data Description

This dataset includes time-series data generated by accelerometer and gyroscope sensors (attitude, gravity, userAcceleration, and rotationRate). It is collected with an iPhone 6s kept in the participant's front pocket using SensingKit which collects information from Core Motion framework on iOS devices. All data collected in 50Hz sample rate. A total of 24 participants in a range of gender, age, weight, and height performed 6 activities in 15 trials in the same environment and conditions: downstairs, upstairs, walking, jogging, sitting, and standing. The major objective of the Motion Sense Case is to predict the type of activities (downstairs, upstairs, walking, jogging, sitting, and standing) using time-series model.

The dataset consists of accelerometer and gyroscope measurements from a group of subjects performing different activities. The data is presented in a comma-separated values format with a header row indicating the column names. Each row of data corresponds to a measurement taken at a specific time point during the activity. The dataset includes 25 columns of information for each measurement, including the tick number, attitude, gravity, rotation rate, user acceleration, and subject demographics.

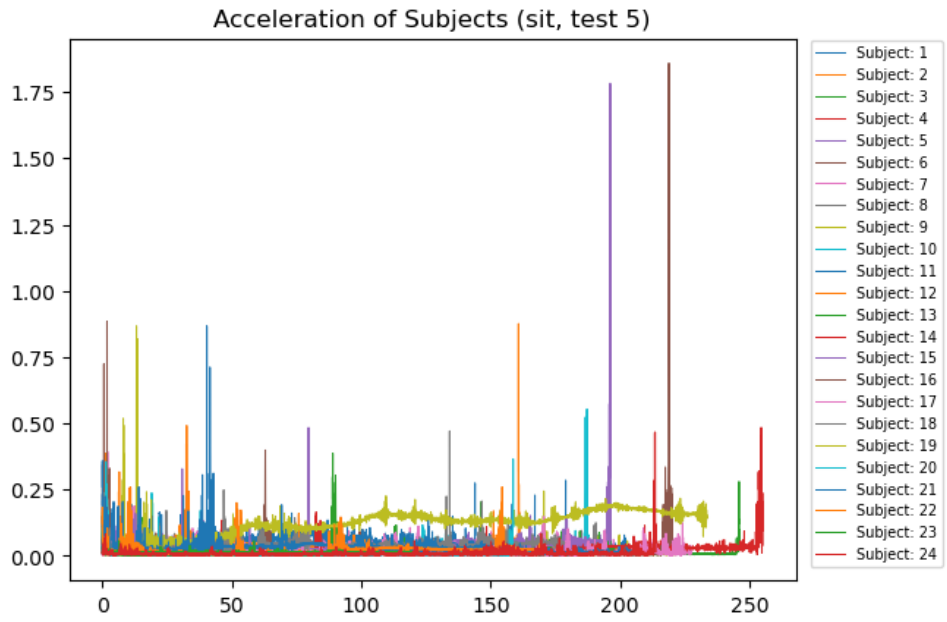
The tick number column indicates the time of each measurement in terms of the number of ticks since the beginning of the recording. The attitude columns represent the orientation of the device in space, with roll, pitch, and yaw angles measured in radians. The gravity columns represent the force of gravity in the x, y, and z directions, and the rotation rate columns represent the rate of change of the orientation angles in the x, y, and z directions.

The user acceleration columns represent the acceleration of the device in the x, y, and z directions, adjusted for the force of gravity. The test type column indicates the type of activity performed by the subject, and the subject ID and test trial number columns identify the subject and trial number, respectively. The time since the start column indicates the time in seconds since the beginning of the activity, and the time series data column provides the date and time stamp of each measurement.

Finally, the weight, height, age, and gender columns provide demographic information about the subject. The dataset is intended for use in the development of activity recognition algorithms and may be used to train machine learning models to classify different types of physical activities based on the accelerometer and gyroscope data.

Exploratory Data Analysis

Upon examining the data, it appears to be rather chaotic as it consists of gyroscopic data. To provide an overview of the data, let's look at the acceleration of everyone during the fifth sit test:



The data for the other recorded gyroscope attributes appears to follow a similar pattern, thus, we do not see a reason to include further EDA in this section. Other plots of the data are included in the submission file.

Data Cleaning

We begin with 361 files: one file containing the attributes of the participants and 360 others containing the gyroscope information for each participant during each test. As there was no missing data in any file, we were able to head straight to stitching the files together.

Here is a brief overview of the code used to combine all 361 files into a single data frame. First, we use multiple for loops to iterate through the many files. We iterate over trial_id, subject_id, and test_numbers. For each loop, we read in the subject's file for one test and trial type, add the subject data to the gyroscope data, and append it to the main data frame. We will talk about feature engineering next.

Data Transformation

During the data cleaning section, we spoke about the algorithm we used to read and combine all test files. During this script, we did a small amount of feature engineering. First, we wanted to add synthetic timestamps to the data frame. To do this, I included the index as a column in the data frame. After this, knowing the observations occurred at 50Hz, I was able to multiply each index by 0.02 and get the time of each observation since the trial started. After this, I converted the timestamps to datetimes. After adding timestamps, I added a column for the magnitude for each gyroscope sensor. This entailed calculating sqrt(x^2+y^2+z^2) for each axis. This resulted in four additional features.

An important task in machine learning is to train, test, and validation splitting your data to ensure you are not overfitting. To do this, we first specify which trials will be used for training, and which will be used for testing and validation. The training trials are numbered 1 through 9 while the

testing and validation trials are numbered 11 through 16. The order of trials is maintained so as to ensure that we are using past data to predict the future. We then create three empty data frames for the training, testing, and validation sets. Next, the code selects a subset of subjects to be used for testing and validation. This is done by randomly selecting half of the subject IDs in the dataset to be used for testing and then assigning the remaining subjects to the validation set. Finally, the code loops through each trial in the dataset and assigns it to the appropriate data frame based on its trial number. If the trial number is in the list of training trials, the data for that trial is added to the training data frame. If the trial number is in the list of testing and validation trials, the data is added to either the testing or validation data frame depending on whether the subject ID for that data is in the testing subset or the validation subset.

Modeling

Random Forest (RF)

Given the limitations of decision trees in achieving optimal classification performance, we sought to evaluate the suitability of Random Forest, a popular ensemble learning method that provides higher predictive power. To optimize the Random Forest model, we also conducted a grid search cross validation with 5 folds of hyperparameters, including 'criterion', 'max_depth', and 'max_features'. The reason we did not use 10-folds here is because the hyperparameter tuning of random forest model takes a long time to run in our cases, so we reduce the number to five.

Our cross-validation accuracy yielded approximately 70%, while the corresponding test accuracy was 82.5%. It is noteworthy that despite the similarity in cross-validation accuracy, we observed a notable improvement of 5% in the test accuracy. While these findings are encouraging, we remain committed to exploring other viable methods that may yield more optimal model performance.

Convolutional Neural Network (CNN)

Moving away from more well-known supervised learning methods, we arrive at convolutional neural networks. After doing some research, we realized these models can be applied to more complex problems such as computer vision. We thought we would try applying this model to our problem as it contains highly nonlinear data.

Getting into our model, we decided to go with 4 dense layers and an input dimension of 7. The dense layer sizes start from 80, go to 40 to 20 to 6. After compiling the mode, we determined that 10 epochs with a batch size of 200 gave the best performance.

Long Short-Term Memory Networks (LSTM)

After trying many models, we decided none of them performed as well as hoped. To fix this, we researched models designed to more specifically tackle these kinds of problems. This is how we came across an LSTM model. Below is a high-level overview of how the model works.

LSTM stands for Long Short-Term Memory. It is a type of recurrent neural network (RNN) that is particularly well-suited for processing sequential data. The basic idea behind LSTMs is to use memory cells that can maintain their state over time, allowing the network to selectively remember or forget information as it processes a sequence.

Since our data is sequential in nature, we thought this model would be perfect for our data. To implement the model, we had to convert our data frames to 3D NumPy arrays that were timesteps * features * rows. The optimal arrays ended up being 50 timesteps back and 12 features wide. For our problem, we determined that one dense layer provided sufficient predictive power. For our model settings, we used adam as the optimizer, categorical cross entropy for the loss function, and accuracy for the measurement metric. We determined through running the model a few times that a batch size of 64 was sufficient along with 5 epochs. We were not able to run more epochs due to computational limitations. However, by the 5th epoch our training accuracy was at about 99% so there was no need to run more.

The main reason that this model seems to perform particularly well is its ability to "look" back into the past to determine the future.

Analysis

Model comparison

Models	**CV Accuracy**	**Test Accuracy**
Random Forest	70.34%	82.49%
CNN	78.60% (Train not CV)	78.98%
LSTM	**99.11% (Train not CV)**	**95.20%**

Note: *Why the test accuracy might be higher than the cv accuracy.*
As the data is time series, we manually selected the test set. While creating the train, test, and validation sets, we chose certain trials for each data frame. This could have impacted cross-validation accuracy, as the test data might have been easier to predict. One alternative approach could have been to use lag features to remove autocorrelation and enable cross-validation. However, this was computationally expensive, and we already had a good model (LSTM). Although we tried training with lag features and saw a slight improvement in model performance, the increase in computational expenses was disproportionate. Therefore, we chose not to include lag features to save on run time.

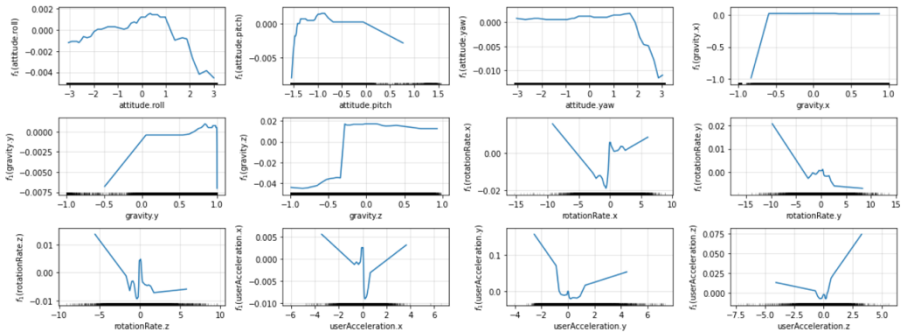
The results provided show the performance of various machine learning models on a classification task. The models considered are Random Forest, CNN, and LSTM. The performance is measured by cross-validation accuracy (CV accuracy) and test accuracy. The Random Forest model has a CV accuracy of 70.34%. It's interesting that the test accuracy is much higher at 82.49%. The CNN model has a high train accuracy of 78.60%, but it is not cross validated. Its test accuracy is 78.98%. Finally, the LSTM model has the highest train accuracy of 99.11%, but it is not cross validated. Its test accuracy is 95.20%. Overall, the Random Forest and CNN models seem to perform well on the test set. The LSTM model has the highest train accuracy as it can "look" into the past to gain more insights about the dependent variable.

Feature importance & Interpretation

As we can see in the ALE plots, gravity.y, attitude.pitch, and userAcceleration.y vary the y axis the most. We can see this reflected in the feature importance plot for the GBM model in appendix, Motionsense GBM Feature Importance.

Random Forest (RF)

To interpret the random forest model, aleplots are employed, although the computational expense associated with the large size of the dataset necessitates the use of a random subsample. Specifically, a random subsample of fraction 0.2 is selected without replacement for plotting the graphs, since the entire dataset cannot be utilized. The obtained results are presented below.



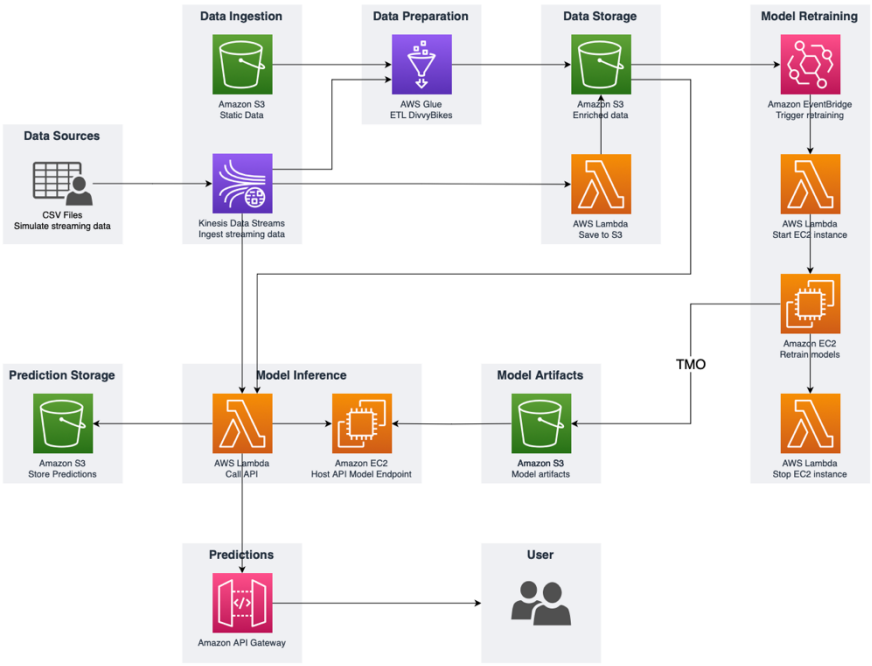
As we can see from the plots, gravity.x, userAcceleration.y and rotationRate.z are the top three important features with a larger range.

Long Short-Term Memory Networks (LSTM)

Finally, we have LSTM. LSTM is a highly sophisticated neural network model with recurrent nodes and input, output, and forget gates. Because of this, it is a fully black box model with no real interpretability, despite being able to extract the values of the parameters.

AWS Implementation

Solution Architecture



The end-to-end solution implemented in AWS as depicted above predominantly mimics the general architecture leveraged by REFIT. A high-level overview of the pipeline is provided below:

1. Streaming data is ingested through Kinesis Data Streams. The ingestion automatically triggers three separate lambda functions for each of the three use cases, respectively.
2. The lambda function *"Save to S3"* retrieves the streamed data and subsequently stores it in an S3 bucket later leveraged for model retraining purposes.
3. The lambda *"Call API"* is depicted as a single function in the architecture for clarity; however, its functions are twofold:
    a. The first calls the API to retrieve the prediction and stores it in an S3 bucket.
    b. The latter calls the API to get the prediction and then displays it to the end user.
4. The predictions are depicted through Amazon API Gateway employing the WebSocket protocol to ensure real-time results.

Retraining Pipelines:

- Initialization: Importation of dependencies and the configuration of logging for process tracking and troubleshooting.
- Function definitions: Numerous functions are defined for interfacing with the S3 platform and creating datasets.
- Path definitions: Appropriate path definitions ensure the successful execution of retraining scripts.

- Data handling: Previously defined functions are employed to handle data; in particular, the acquiring of old and new data, transferring new data from *"kinesis_data"* to *"kinesis_data_old"*, setting data windows, and producing windowed data. The output is ultimately saved to an S3 bucket.
- Data transformation: The data are transformed in line with the needs of each use case as a preparation step for subsequent retraining.
- Model development: The machine learning models are defined, fitted, and ultimately stored; this final step concludes the retraining pipeline, ensuring the model is ready for future use.

Solution Cost Estimation

Given the nature of AWS as a paid service, an accurate cost estimation ensures that the end-to-end solution remains within the allocated budget and further promotes overall transparency. The combined cost of the final solution architecture for the three use cases is estimated to reach an annual total of $3,207.22 USD or an equivalent monthly total of $267.67 USD. The configurations are chosen based on the most intensive use case. Bearing in mind the time scope of the practicum project, the latter estimate paints a more accurate picture of the true relevant cost.

Amazon API Gateway and AWS Glue are two of most costly services incorporated as part of the solution with monthly estimates of $131.40 USD and $24.21 USD, respectively. However, several AWS services being employed are considerably more modest in expenses, balancing out the overarching monthly expenses. In addition, notable cost reductions were considered as part of the final architecture, avoiding particularly price-intensive services such as Amazon SageMaker and instead opting for a cheaper alternative in the form of Amazon EC2. Finally, it is worth noting that the end-to-end solution was only run several times for final testing and performance measurement purposes, with the actual project spendings amounting to a total far less than monthly estimate.

Performance Measurements

1. Throughput
   - The more quantitative approach of the two metrics considered, throughput measures the quantity of data transferred during a specific time interval. For the end-to-end AWS solution, throughput is evaluated in the context from streaming to prediction.
   - The average throughput metrics for each of the three uses cases are as follows:
     - Divvy Bikes: 5-6 data points/sec or 300-360 data points/min
     - Hard Drives: 40-55 data points/sec or 2400-3300 data points/min
     - Motionsense: 12-17 data points/sec or 720-1020 data points/min
   - The variation in throughput between the use cases stems from their respective differences in the overall implementation and design. However, the metrics indicate the success of the implemented solution with respect to real-time streaming.

2. Ease of Use
   - The second performance measurement rests on the qualitative end of the spectrum: the ease of understanding and implementing the full end-to-end solution on AWS.
   - The entire AWS architecture, from the planning phase to completing the entire solution, took roughly 2 months with a team of seven members.

    - The team members had no prior experience with AWS but concurrent course work in cloud engineering proved particularly helpful.
    - Completing distinct sections of the architecture proved less involved than connecting the full solution all together.
- The end-to-end solution on AWS is extremely scalable even upon full implementation; however, one potential flaw is the lack of flexibility in the sense that individual AWS services require catering to a specific use case without the ability for generalization.

Future Considerations

The final scope and objectives of the project have transitioned slightly from the original proposal including the implementation of the three use cases on REFIT and designing a model agnostic feature selection algorithm for time series data. In particular, corresponding implementation leveraging REFIT capabilities will facilitate relevant comparisons between the former with cloud services. These works could serve as potential avenues for consideration for future projects with CDL.

Finally, future work could also serve to improve the current end-to-end AWS implementation across two notable areas. First, the machine learning predictions for the use cases are currently only provided as raw values. Hence, augmenting an additional service to visualize past and current predictions could help further improve the solution. Furthermore, the current throughput metrics appear to decrease inversely proportional to the stream size: the lambda function sending records to EC2 was identified as the likely culprit limiting the maximum potential throughput. Increasing the allocated compute power and memory, or employing other relevant solutions are further considerations to build upon the CDL IoT Use Cases practicum project.

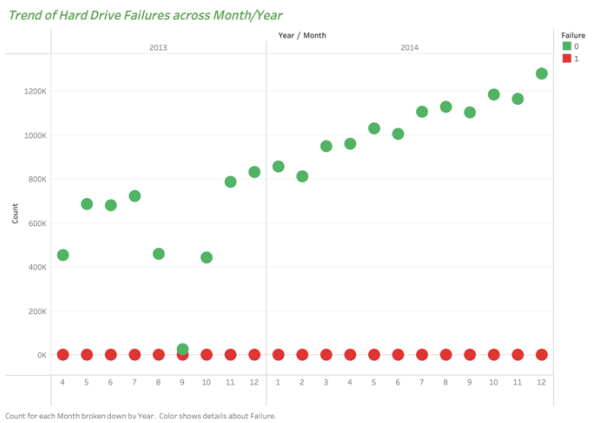
Bibliography

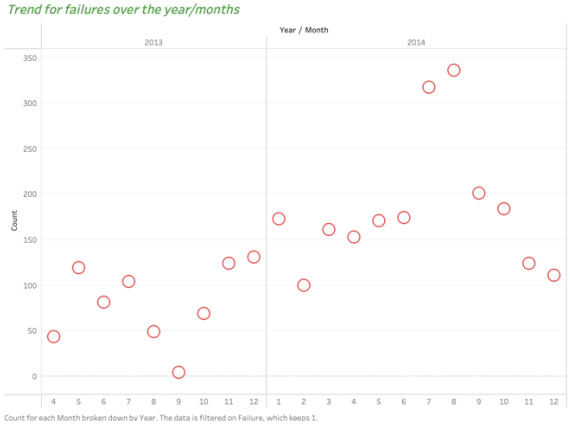
1. *Interpretable predictive maintenance for hard drives.* **Maxime Amram, Jack Dunn, Jeremy J. Toledano, Ying Daisy Zhuo. 2021.** 2021, Machine Learning with Applications.
2. **Klein, Andy. 2016.** What SMART Stats Tell Us About Hard Drives. *Backblaze.* [Online] October 6, 2016. https://www.backblaze.com/blog/what-smart-stats-indicate-hard-drive-failures/.

Appendix

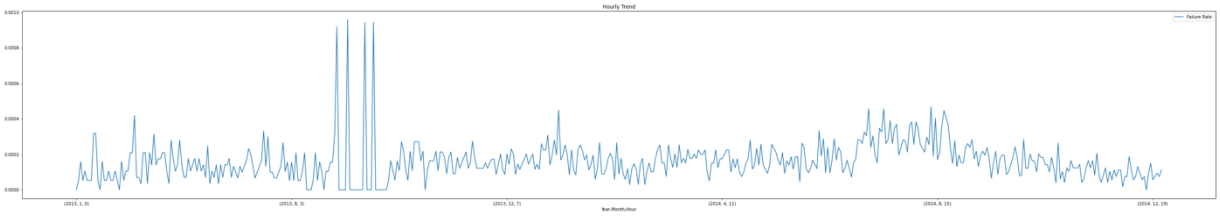
Use Case 1: Hard Drive

Data Analysis for Hard Drive dataset (The data is pertaining to 2013-2014 (wider range) to check whether failure rate might be good for modelling)



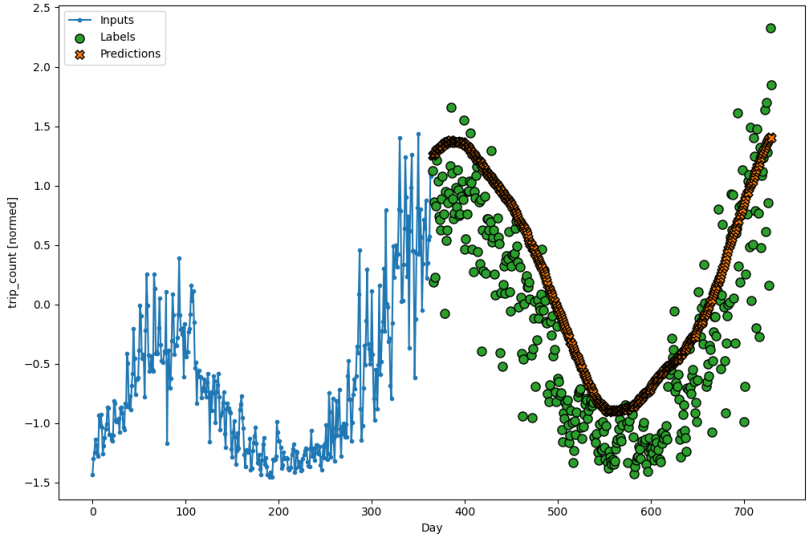


Hourly failure rate trend from 2013 - 2014

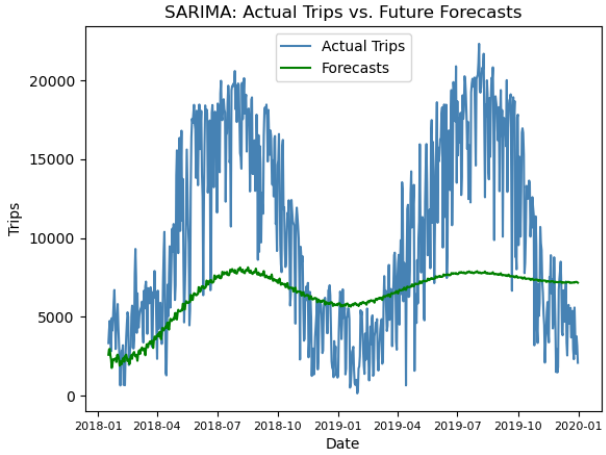


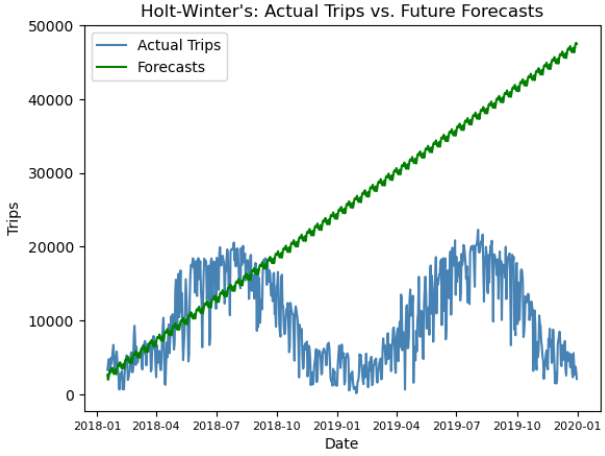
Use Case 2: Divvy Bikes

Plot 1: LSTM: Input window 365 days, prediction of 365 days.



Plot 2: Holt-Winter's and SARIMA Forecasts





SARIMA model summary

	coef	std err	z	P>|z|	[0.025	0.975]
const	8336.1974	4306.834	1.936	0.053	-105.042	1.68e+04
ar.L1	0.3233	0.503	0.643	0.520	-0.662	1.309
ar.L2	0.6735	0.502	1.342	0.180	-0.310	1.657
ma.L1	0.1707	0.503	0.339	0.734	-0.816	1.157
ma.L2	-0.6100	0.258	-2.367	0.018	-1.115	-0.105
ma.L3	-0.2025	0.146	-1.385	0.166	-0.489	0.084
ma.L4	-0.0347	0.021	-1.689	0.091	-0.075	0.006
ar.S.L12	1.8809	0.038	49.178	0.000	1.806	1.956
ar.S.L24	-0.9274	0.037	-25.088	0.000	-1.000	-0.855
ma.S.L12	-1.9379	0.047	-41.217	0.000	-2.030	-1.846
ma.S.L24	1.0398	0.068	15.345	0.000	0.907	1.173
ma.S.L36	-0.0171	0.053	-0.320	0.749	-0.122	0.087
ma.S.L48	-0.0389	0.026	-1.496	0.135	-0.090	0.012

Holt-Winter's model summary

	coeff
smoothing_level	0.3585714
smoothing_trend	0.0001
smoothing_seasonal	0.0278882
initial_level	1679.2194
initial_trend	73.904040
initial_seasons.0	-340.17014
initial_seasons.1	139.32986
initial_seasons.2	394.25694
initial_seasons.3	411.59028
initial_seasons.4	534.49653
initial_seasons.5	-474.34722
initial_seasons.6	-295.99306
...	
initial_seasons.9	-69.576389
initial_seasons.10	-755.38889
initial_seasons.11	-269.80556

Use Case 3: MotionSense
Motionsense GBM Feature Importance