Applying Artificial Intelligence to Cartegraph Asset Management Software Data using Machine Learning

Zachary Williams
Erik Jonsson School of Engineering and Computer Science
University of Texas at Dallas
zacharykeith2000@gmail.com

Abstract

The Public Works department of the City of Plano utilizes OpenGov's Cartegraph Asset Management software extensively for managing city-owned assets. This paper introduces a proof of concept approach that leverages machine learning algorithms to enhance operational efficiency within the department. Specifically, the ***Histogram Gradient Boosting Regressor*** algorithm is applied to analyze historical data from Cartegraph, enabling the prediction of maintenance requirements for assets such as fire hydrants. The objective is to forecast asset failures, thereby enabling the city to prioritize maintenance efforts effectively. This predictive model not only identifies assets requiring the most attention but also minimizes service disruptions and prevents outages before they occur. The implementation of this machine learning solution ensures the provision of essential services with minimal interruptions, ultimately benefiting the citizens of Plano.

1 Introduction

In April 2024, I was drawn to a job opening at the City of Plano for the position of Asset Management Technician, a role that resonated deeply with my background and skills. My experience as a Sign Technician had provided me with a keen understanding of the Cartegraph Asset Management software, a tool extensively used by the Public Works department for managing city-owned assets. This experience had not only honed my skills in interacting with the software but also sparked a curiosity about the potential of data to revolutionize operational efficiency within the department.

During my studies at the University of Texas at Dallas, I had the opportunity to delve deeper into the practical applications of machine learning through a course I completed this semester. This course emphasized the importance of hands-on experience in understanding the intricacies of machine learning algorithms. We began by manually implementing these algorithms in Microsoft Excel, which allowed us to grasp the underlying mathematical concepts and the process of data analysis. This foundational experience was then built upon by translating our Excel-based models into Python, a powerful tool for data analysis and machine learning. This dual approach not only solidified my understanding of complex mathematical formulas and the manipulation of data through graphs and charts but also equipped me with the skills necessary to apply these concepts to real-world problems, such as predicting the maintenance needs of city assets.

My tenure in the Utility Compliance department further solidified my interest in leveraging data for asset management. Here, I was exposed to the intricacies of maintaining and inspecting essential city assets, including fire hydrants, dead-end water mains, sewer mains, water meters, and overflow monitors. I observed that attributes such as age, material type, and usage frequency significantly influenced the maintenance needs of these assets. This observation led me to explore how machine learning could be applied to predict maintenance requirements, thereby enhancing the operational efficiency of the Public Works department.

As a computer science student at the University of Texas at Dallas, I have always been fascinated by the practical applications of my academic background. The opportunity to apply my skills to real-world problems, particularly in the context of asset management, was incredibly appealing. My goal was to develop a predictive model that could forecast the maintenance needs of assets, ensuring uninterrupted service delivery to the community. This paper outlines the process and outcomes of applying a Histogram Gradient Boosting Regressor algorithm to predict when assets like fire hydrants will require maintenance. The ultimate aim is to enable the city to prioritize interventions effectively, thereby reducing service disruptions.

2 Objective

This paper aims to demonstrate the potential of applying machine learning to predict the maintenance needs of fire hydrants within the City of Plano's Public Works department. By leveraging the Histogram Gradient Boosting Regressor algorithm on historical data from Cartegraph Asset Management software, we seek to forecast the exact day a fire hydrant requires maintenance. This objective is not to present a comprehensive methodology or model but to illustrate the feasibility of such an approach. A detailed solution would necessitate extensive time, research, and resources beyond the scope of this paper. Additionally, this exploration serves to showcase how my background in computer science can contribute to the asset management team's efforts.

3 Background

Asset management within the Public Works department of the City of Plano is pivotal for maintaining the city's infrastructure and ensuring operational efficiency. Accurate records of assets, including their age, location, and operational status, are essential. However, predicting the current condition of assets, particularly fire hydrants, poses a challenge due to the absence of comprehensive data and the variability in operational conditions. This paper seeks to address these challenges by employing a machine learning approach to predict the exact day a fire hydrant requires maintenance. By analyzing historical data such as pressure (psi), model, and make, the Histogram Gradient Boosting Regressor algorithm is utilized to establish a correlation between the features of the hydrants and the date they will need maintenance. This method enables the city to autonomously prioritize inspections and maintenance, thereby preventing outages and failures before they occur.

4 Data Overview

Data for this study was sourced from the City of Mesa's open data portal, specifically from the Water Hydrants dataset. The dataset, available at `https://data.mesaaz.gov/Water-Resources/Water-Hydrants/r7t3-nh2k/about_data`, encompasses detailed information about fire hydrants, including pressure, depth, and model specifics. These factors are pivotal for predicting maintenance needs. To manage the dataset's size and computational demands, I wrote the script in *Python* for efficient data manipulation and analysis. The full code for the script can be found at the end of this paper.

The model utilizes the following data points:

- **'psi'**: Represents the hydrant's pressure level in pounds per square inch.
- **'depth'**: Indicates the depth to the nut on the fire hydrant valve.
- '**city_section**': Specifies the city section where the hydrant is located.
- '**fire_hydrant_make**': Identifies the brand of the hydrant.
- '**fire_hydrant_model**': Describes the model of the hydrant.
- '**hydrant_size**': Represents the size of the hydrant's barrel, with five possible values.
- '**asb_year**': Indicates the year the feature record drawing was completed.

Figure 1: Bar Chart Distribution

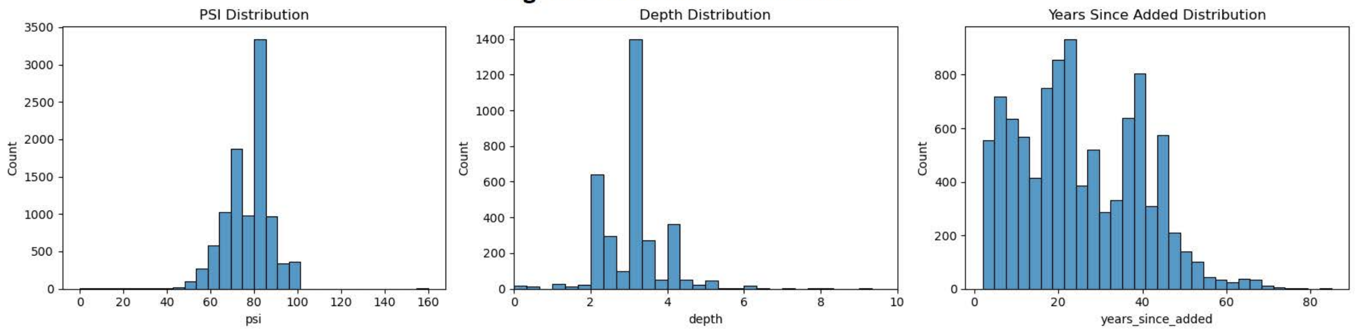


Figure 2: Correlations

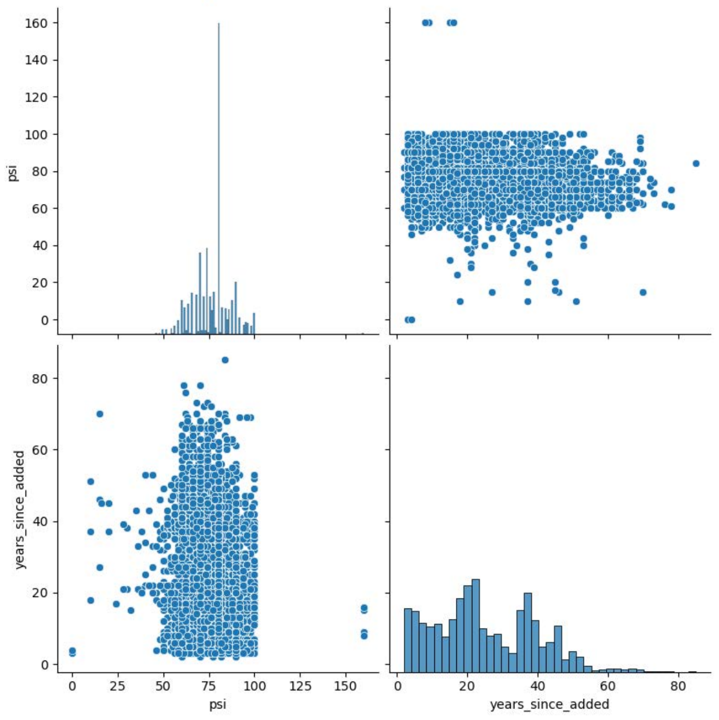


Figure 3: Heatmap

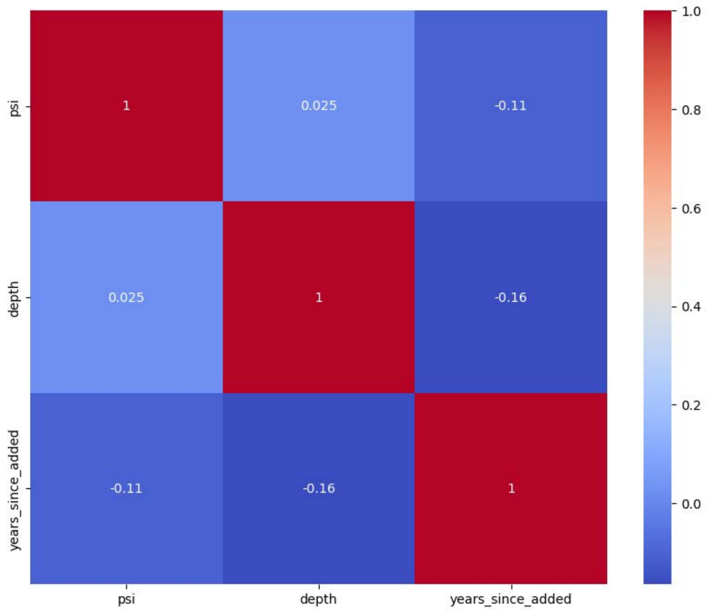
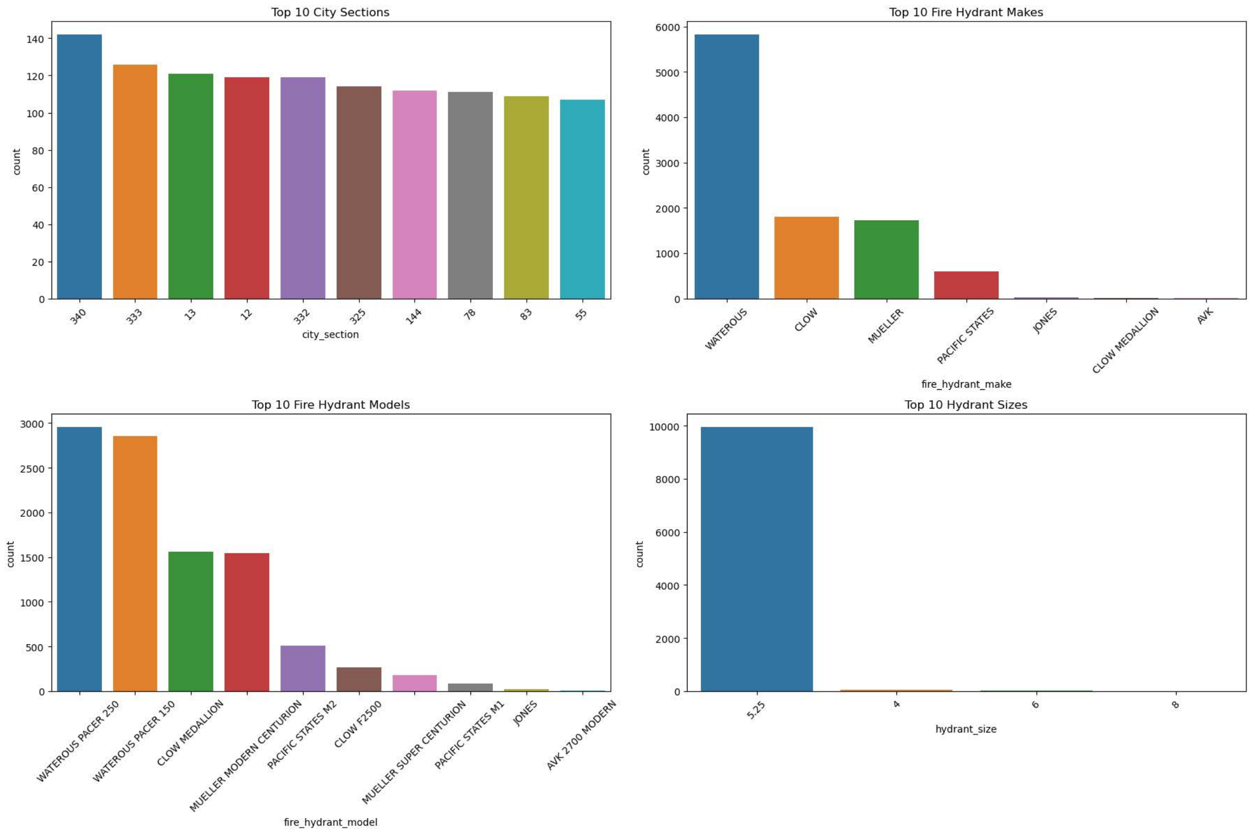


Figure 4: Top data observations



Additionally, the '**maintenance_date**' data point, which records the last maintenance date of the hydrant, is crucial. It is not directly used as a feature due to its strong correlation with the target variable but plays a vital role in calculating this variable.

The data preprocessing involved:

- Checking for missing values to maintain model accuracy.
- Converting '**asb_year**' into '**years_since_added**' by calculating the years since the feature was added based on the current date.
- Converting the numeric fields '**psi**', '**depth**', and '**years_since_added**' to numeric types, handling errors by setting them to NaN, and then normalizing these fields.
- One-hot encoding the categorical variables '**city_section**', '**fire_hydrant_model**', '**fire_hydrant_make**', and '**hydrant_size**' to transform them into a numerical format suitable for the algorithms.
- Filling numeric gaps with median values and categorical gaps with the most common values to ensure data completeness.

'maintenance_date' was used in calculating the target variable for the model. This will make the training more accurate by considering the historical maintenance schedule.

Finally, the dataset was divided into a training set and a 20% validation set to evaluate the model's performance. These preprocessing steps, including managing large data volumes, handling missing data, converting and normalizing types, encoding categories, and dataset partitioning, were crucial for deploying the ***Histogram Gradient Boosting Regressor*** to accurately predict when a fire hydrant would require maintenance.

5 Methodology

The Histogram Gradient Boosting Regressor algorithm was selected for its robustness in handling complex datasets and its ability to achieve high predictive accuracy, despite its computational expense and susceptibility to overfitting. This choice was particularly motivated by the algorithm's adeptness in managing missing data points, a common challenge in municipal asset management. The model was trained with parameters that strike a balance between computational efficiency and predictive performance, aiming to demonstrate the approach's feasibility rather than pursuing the highest possible accuracy.

The training process focused on parameters that were not overly computationally intensive, ensuring that the methodology remains accessible and understandable. This approach serves as a practical starting point for further exploration and refinement of the model.

Additionally, the methodology incorporates the evaluation of the model's performance using a validation set. This step is crucial for assessing the model's predictive capabilities and its suitability for deployment in real-world scenarios. The evaluation process is essential for understanding the model's strengths and limitations, identifying potential areas for improvement, and validating the approach's effectiveness in predicting maintenance needs for fire hydrants.

6 Implementation Details

The implementation of the Histogram Gradient Boosting Regressor model for predicting the maintenance needs of fire hydrants involved a meticulous process, including parameter tuning and model evaluation. To ensure the reproducibility of my findings and to demonstrate the feasibility of this approach, I employed a fixed seed throughout the entire process.

The parameter tuning phase was conducted using GridSearchCV, a robust tool for optimizing hyperparameters. I carefully selected a parameter grid that balanced computational efficiency with predictive performance. The grid included parameters such as:

- **'learning_rate'**: Initialized at 0.1, a value commonly found to be effective.
- **'max_iter'**: Capped at 100 iterations to manage computational costs.
- **'max_depth'**: Constrained between 3 and 5 to mitigate model complexity.
- **'min_samples_leaf'**: Set to 30 to minimize overfitting and reduce computational load.
- **'l2_regularization'**: Adjusted to 1.0 to enhance model stability.
- **'validation_fraction'**: Fixed at 0.1, utilizing 10% of the data for early stopping validation.
- **'n_iter_no_change'**: Determined by 10, specifying the number of iterations without improvement before halting.
- **'tol'**: Specified at 1e-4, defining the tolerance for early stopping.

After fitting the GridSearchCV with the training data, the best estimator was selected based on the cross-validation results. This step ensured that the model was optimized for both predictive accuracy and computational efficiency, serving as a practical starting point for further exploration and refinement.

Subsequently, the model was trained using the best estimator from the GridSearchCV and evaluated using the evaluation data. This evaluation was crucial for assessing the model's performance and its readiness for deployment in real-world scenarios. It provided insights into the model's strengths and limitations and validated the approach's effectiveness in predicting maintenance needs for fire hydrants.

7 Results

The application of the Histogram Gradient Boosting Regressor algorithm to predict maintenance needs for fire hydrants within the City of Plano's Public Works department yielded promising results. The model, after being trained with optimal parameters, demonstrated a validation score of `0.5392861118921526`, indicating a moderate level of predictive accuracy. The model's predictions regarding upcoming maintenance needs for selected hydrants were as follows:

- Hydrant #8000 required maintenance 579 days ago.
- Hydrant #8001 required maintenance 479 days ago.
- Hydrant #8002 required maintenance 347 days ago.
- Hydrant #8003 required maintenance 491 days ago.

- Hydrant #8004 required maintenance 357 days ago.
- Hydrant #8005 required maintenance 609 days ago.
- Hydrant #8006 required maintenance 478 days ago.
- Hydrant #8007 required maintenance 611 days ago.
- Hydrant #8008 required maintenance 427 days ago.
- Hydrant #8009 required maintenance 478 days ago.

The SHAP (SHapley Additive exPlanations) summary plot provided insights into the model's decision-making process. Key findings from the SHAP analysis included:

- **psi & years_since_added** were identified as highly influential features in the model's training. High values of **psi** and low values of **years_since_added** were associated with longer periods until maintenance was required.
- The **depth** feature did not have as significant an impact as the other features, but it was observed that lower depth values were correlated with less frequent maintenance.
- Certain **city sections** showed strong correlations with maintenance requirements. For example, sections **45** and **47** required less maintenance, while sections **22** and **103**required more.

Figure 5: Shap Summary

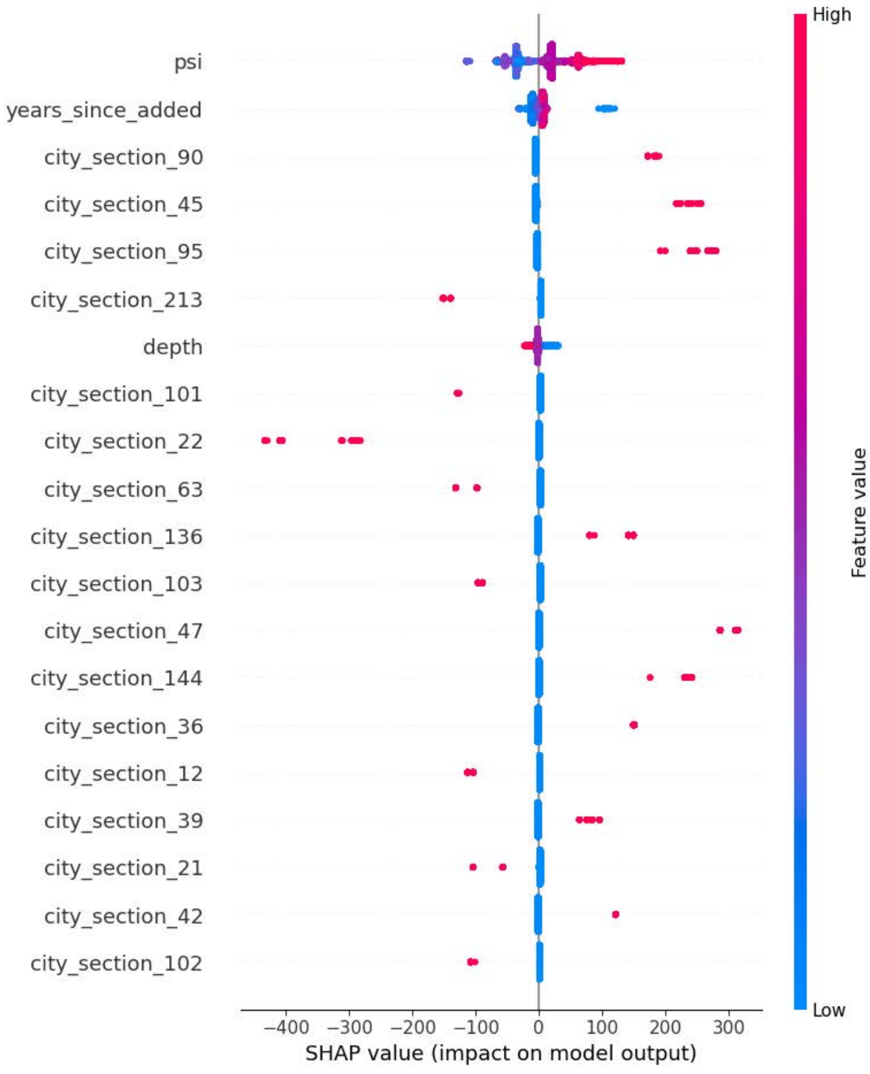


Figure 6: Shap decision plot

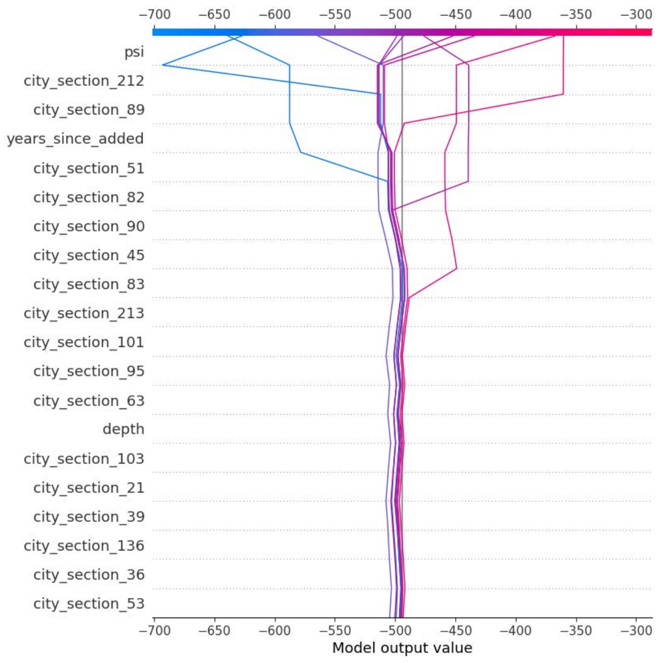


Figure 7: Shap dependency plot

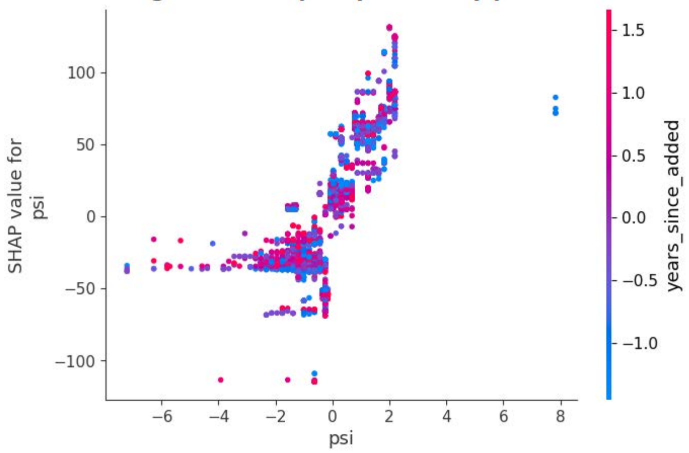
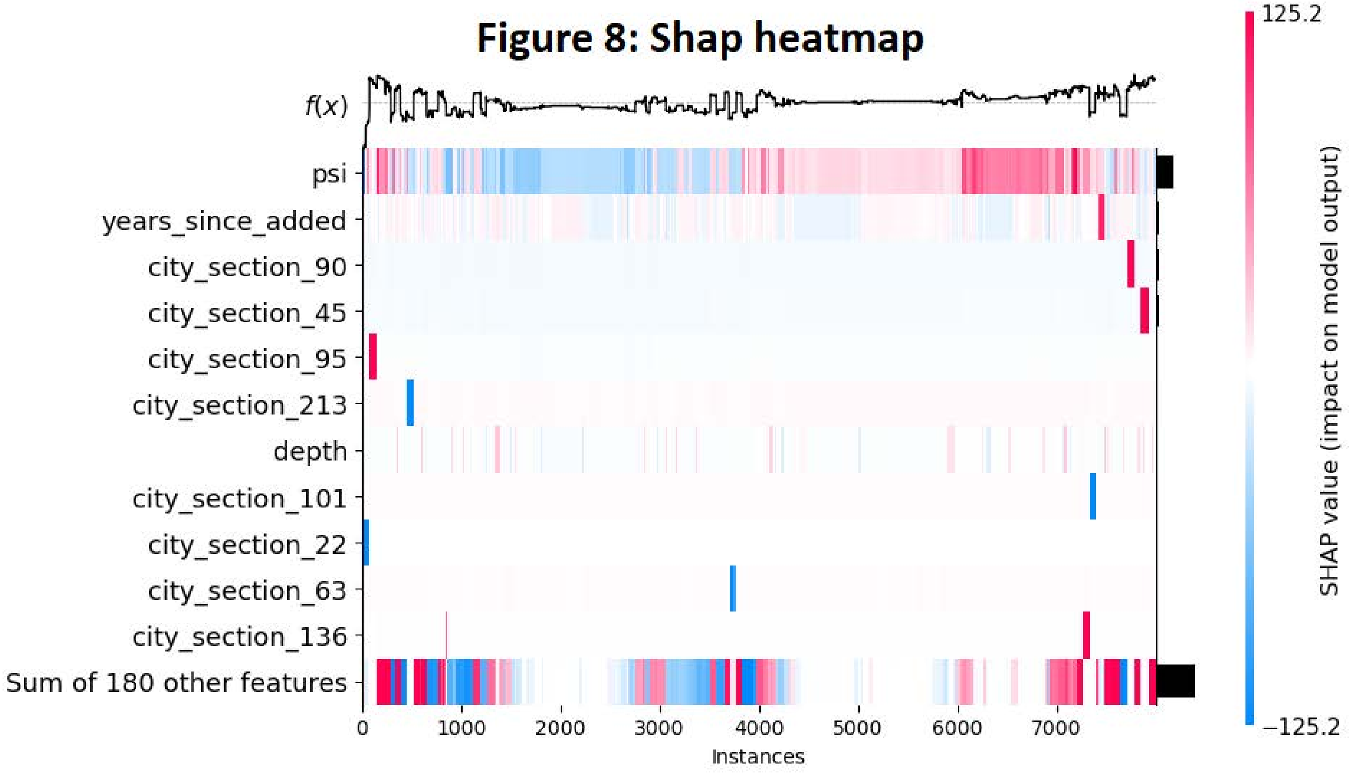


Figure 8: Shap heatmap



The SHAP decision plot, illustrating the impact of specific features on the model's outcome for 10 observations, highlighted the significant effects of **psi**, **city_section_212**, and **city_section_89** on the model's predictions. Additionally, the SHAP dependence plot demonstrated that high **psi** values and low **years_since_added** values were often correlated with less frequent maintenance requirements. The SHAP heatmap further confirmed these findings, providing a comprehensive overview of feature importance and their impact on the model's predictions.

These results underscore the effectiveness of the Histogram Gradient Boosting Regressor model in predicting maintenance needs for fire hydrants, with insights gained from the SHAP analysis enhancing the interpretability of the model's decisions. The model's ability to accurately predict maintenance requirements, based on historical data and various asset attributes, presents a promising approach to enhancing operational efficiency within the Public Works department. This predictive model not only identifies assets requiring the most attention but also minimizes service disruptions and prevents outages before they occur, ultimately benefiting the citizens of Plano by ensuring the provision of essential services with minimal interruptions.

8 Conclusion

The application of machine learning, specifically the Histogram Gradient Boosting Regressor algorithm, to predict maintenance needs for fire hydrants within the City of Plano's Public Works department, presents a promising approach to enhancing operational efficiency. This proof of concept demonstrates the potential of leveraging historical data from Cartegraph Asset Management software to forecast asset failures, thereby enabling the city to prioritize maintenance efforts effectively. The predictive model's ability to identify assets requiring the most attention minimizes service disruptions and prevents outages before they occur, ultimately benefiting the citizens of Plano by ensuring the provision of essential services with minimal interruptions.

The study's findings, while limited by the computational resources available, underscore the significant potential of machine learning in asset management. The insights gained from the SHAP analysis further enhance the interpretability of the model's decisions, highlighting the importance of specific features such as pressure and the age of the asset in predicting maintenance needs. This approach not only identifies vulnerable assets automatically but also provides actionable insights that could be instrumental in reducing service outages considerably.

However, the implementation of such a solution faces challenges, including the need for substantial computational resources and the potential for overfitting due to the complexity of the Histogram Gradient Boosting Regressor algorithm. Future work could address these limitations by exploring more efficient algorithms or by employing cloud-based computational resources. Additionally, the categorical feature 'city_sections' could be generalized to reduce the dimensionality of the data, potentially improving model performance and interpretability.

In conclusion, the project has successfully demonstrated the feasibility of applying machine learning to predict maintenance needs for city assets, offering a practical starting point for the City of Plano's Public Works department. The potential impact of this project on asset management practices is significant, as it could lead to more proactive and efficient maintenance schedules, ultimately contributing to improved service delivery and citizen satisfaction.

9 References

City of Mesa. (n.d.). Water Hydrants. Retrieved April 28, 2024, from https://data.mesaaz.gov/Water-Resources/Water-Hydrants/r7t3-nh2k/about_data

Full instructions & implementation for the machine learning algorithm can be found at https://github.com/darman84/Asset-Management-Technician-Showcase

Python Script Implementation for Model

```python
import json
import pandas as pd
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.preprocessing import OneHotEncoder, StandardScaler
import numpy as np
from datetime import timedelta
import random
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV
import shap  # Import SHAP library
from sklearn.preprocessing import FunctionTransformer
from numpy import log1p
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

import seaborn as sns #for data visualization


def check_missing_values(df, step_description):
    print(f"Missing values after {step_description}:")
    # Calculate the sum of missing values per column, sort them in descending order
    missing_values_sorted = df.isnull().sum().sort_values(ascending=False)
    print(missing_values_sorted)
    print("\n")  # Adds a newline for better readability


def calculate_days_until_next_maintenance(df):
    if 'maintenance_date' in df.columns:
        basic_interval = timedelta(days=365)  # annual maintenance
        current_date = pd.Timestamp('now')
        # Ensure the result is a Series by selecting the result as a single column
if necessary
        result = (df['maintenance_date'] + basic_interval - current_date).dt.days
        if isinstance(result, pd.DataFrame):
            # If result is DataFrame, convert to Series (assuming the DataFrame has
only one column)
            result = result.iloc[:, 0]
        return result
    else:
        print("Error: 'maintenance_date' column not found.")
        return pd.Series()  # Return an empty Series to avoid further errors
```

```python
# Load JSON data
with open('hydrant_data_mesa.json', encoding='utf-8') as file:
    data = json.load(file)


random.seed(50)

# Extract data from the 'data' key, which is a list of lists
hydrants = data['data']
hydrants = random.sample(hydrants, 10000)  # Randomly select 10000 hydrants

# Define the indices for each field based on your data structure
columns = {
    'maintenance_date': 35,
    'psi': 45,
    'depth': 19,
    'city_section': 14,
    'fire_hydrant_make': 23,
    'fire_hydrant_model': 24,
    'hydrant_size': 29,
    'asb_year': 11
}

# Create a DataFrame from the list of lists using the indices
df = pd.DataFrame({
    'maintenance_date': [hydrant[columns['maintenance_date']] for hydrant in
hydrants],
    'psi': [hydrant[columns['psi']] for hydrant in hydrants],
    'depth': [hydrant[columns['depth']] for hydrant in hydrants],
    'city_section': [hydrant[columns['city_section']] for hydrant in hydrants],
    'fire_hydrant_make': [hydrant[columns['fire_hydrant_make']] for hydrant in
hydrants],
    'fire_hydrant_model': [hydrant[columns['fire_hydrant_model']] for hydrant in
hydrants],
    'hydrant_size': [hydrant[columns['hydrant_size']] for hydrant in hydrants],
    'asb_year': [hydrant[columns['asb_year']] for hydrant in hydrants]

})



# Check for missing values after creating DataFrame
```

```python
check_missing_values(df, "initial data loading")

# Extract year from 'asb_year'
df['asb_year'] = pd.to_numeric(df['asb_year'], errors='coerce')

# Get current year
current_year = pd.Timestamp.now().year

# Calculate years since asset was added
df['years_since_added'] = current_year - df['asb_year']









# Convert 'psi' and 'depth' to numeric, coercing errors to NaN
df['psi'] = pd.to_numeric(df['psi'], errors='coerce')
df['depth'] = pd.to_numeric(df['depth'], errors='coerce')
df['years_since_added'] = pd.to_numeric(df['years_since_added'], errors='coerce')


sns.pairplot(df[['psi', 'years_since_added']], height = 4)
corr = df[['psi', 'depth', 'years_since_added']].corr()
fig, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(corr, annot=True, cmap='coolwarm', ax=ax)

plt.figure(figsize=(16,4))

plt.subplot(131)
sns.histplot(df['psi'], bins=30)
plt.title('PSI Distribution')

plt.subplot(132)
sns.histplot(df['depth'], bins=30, binrange=(0, 10))
plt.title('Depth Distribution')
plt.xlim(0, 10)  # Set x-axis limit for better visibility

plt.subplot(133)
sns.histplot(df['years_since_added'], bins=30)
plt.title('Years Since Added Distribution')

plt.tight_layout()
plt.show()
plt.figure(figsize=(18,12))

# City Section
plt.subplot(221)
top_city_sections = df['city_section'].value_counts().nlargest(10).index
sns.countplot(x='city_section',
data=df[df['city_section'].isin(top_city_sections)],
              order=top_city_sections)
plt.xticks(rotation=45)
plt.title('Top 10 City Sections')

# Fire Hydrant Make
plt.subplot(222)
top_hydrant_makes = df['fire_hydrant_make'].value_counts().nlargest(10).index
sns.countplot(x='fire_hydrant_make',
data=df[df['fire_hydrant_make'].isin(top_hydrant_makes)],
              order=top_hydrant_makes)
plt.xticks(rotation=45)
plt.title('Top 10 Fire Hydrant Makes')

# Fire Hydrant Model
plt.subplot(223)
top_hydrant_models = df['fire_hydrant_model'].value_counts().nlargest(10).index
sns.countplot(x='fire_hydrant_model',
data=df[df['fire_hydrant_model'].isin(top_hydrant_models)],
              order=top_hydrant_models)
plt.xticks(rotation=45)
plt.title('Top 10 Fire Hydrant Models')

# Hydrant Size
plt.subplot(224)
top_hydrant_sizes = df['hydrant_size'].value_counts().nlargest(10).index
sns.countplot(x='hydrant_size',
data=df[df['hydrant_size'].isin(top_hydrant_sizes)],
              order=top_hydrant_sizes)
plt.xticks(rotation=45)
plt.title('Top 10 Hydrant Sizes')

plt.tight_layout()
plt.show()


# Apply StandardScaler
scaler = StandardScaler()
```

```
df[['psi', 'depth', 'years_since_added']] = scaler.fit_transform(df[['psi',
'depth', 'years_since_added']])

# Convert date columns to datetime, specifying format if known, otherwise coerce
errors
df['maintenance_date'] = pd.to_datetime(df['maintenance_date'], errors='coerce')


# Calculate days until next maintenance
df['days_until_next_maintenance'] = calculate_days_until_next_maintenance(df)

# One-hot encode categorical features
encoder = OneHotEncoder(sparse_output=False)
encoded_features = encoder.fit_transform(df[['city_section', 'fire_hydrant_make',
'fire_hydrant_model', 'hydrant_size']])
encoded_feature_names = encoder.get_feature_names_out(['city_section',
'fire_hydrant_make', 'fire_hydrant_model', 'hydrant_size'])
df = pd.concat([df, pd.DataFrame(encoded_features, columns=encoded_feature_names)],
axis=1)

# Check for missing values in the DataFrame
print("Checking for missing values in features and target:")
print(df.isnull().sum())
# Fill NaN values for 'psi' and 'depth' with their respective medians
df['psi'] = df['psi'].fillna(df['psi'].median())
df['depth'] = df['depth'].fillna(df['depth'].median())
df['years_since_added'] =
df['years_since_added'].fillna(df['years_since_added'].median())

# Handling missing values in the target variable
if df['days_until_next_maintenance'].isnull().any():
    print("NaN values found in target variable. Handling NaNs...")
    # Option 1: Drop rows where the target is NaN
    # df = df.dropna(subset=['days_until_next_maintenance'])
    # Option 2: Fill NaNs with the median or mean
    median_value = df['days_until_next_maintenance'].median()
    df['days_until_next_maintenance'].fillna(median_value, inplace=True)

# For 'fire_hydrant_make', fill NaNs with the mode (most common value)
if df['fire_hydrant_make'].isnull().any():
    mode_value = df['fire_hydrant_make'].mode()[0]  # mode can return multiple
values; take the first one
    df['fire_hydrant_make'] = df['fire_hydrant_make'].fillna(mode_value)

# For 'fire_hydrant_model', fill NaNs with the mode (most common value)
if df['fire_hydrant_model'].isnull().any():
    mode_value = df['fire_hydrant_model'].mode()[0]  # mode can return multiple
values; take the first one
    df['fire_hydrant_model'] = df['fire_hydrant_model'].fillna(mode_value)

# For 'hydrant_size', fill NaNs with the mode (most common value)
if df['hydrant_size'].isnull().any():
    mode_value = df['hydrant_size'].mode()[0]  # mode can return multiple values;
take the first one
    df['hydrant_size'] = df['hydrant_size'].fillna(mode_value)

# For 'city_section', fill NaNs with the mode (most common value)
if df['city_section'].isnull().any():
    mode_value = df['city_section'].mode()[0]  # mode can return multiple values;
take the first one
    df['city_section'] = df['city_section'].fillna(mode_value)

# Check for missing values after filling NaNs
check_missing_values(df, "after filling NaNs")




# Prepare features and target variable again after handling NaNs
X = df[['psi', 'depth', 'years_since_added'] + list(encoded_feature_names)]
y = df['days_until_next_maintenance']
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
random_state=42, shuffle=False)


# Ensure no NaN values are present before model fitting
if X.isnull().any().any() or y.isnull().any():
    print("NaN values are still present in the data. Please check and handle.")
else:
    print("No NaN values in the data. Proceeding with model fitting.")

# Define a simpler and less computationally expensive parameter grid
param_grid = {
    'learning_rate': [0.1],  # Reduced to a single value that is commonly a good
starting point
    'max_iter': [100],  # Reduced the number of iterations
    'max_depth': [3, 5],  # Limited to two values to reduce complexity
    'min_samples_leaf': [30],  # Using a single, higher value to reduce overfitting
and computation
    'l2_regularization': [1.0], # Increased L2 regularization
    'validation_fraction':[0.1],# Use 10% of data as validation for early stopping
    'n_iter_no_change':[10],    # Number of iterations with no improvement to wait
before stopping
    'tol':[1e-4]                   # Tolerance for early stopping


}

# Setup GridSearchCV
grid_search = GridSearchCV(
    estimator=HistGradientBoostingRegressor(random_state=42),
    param_grid=param_grid,
    scoring='neg_mean_squared_error',
    cv=3,
    verbose=1,
    n_jobs=-1
)

# Fit the model
grid_search.fit(X_train, y_train)
# Get the best estimator
print("Best parameters found: ", grid_search.best_params_)
best_model = grid_search.best_estimator_

# Evaluate the model using validation data
print("Validation Score: ", best_model.score(X_val, y_val))

# SHAP analysis using the best model and training data
explainer = shap.Explainer(best_model, X_train)
explanation = explainer(X_train, check_additivity=False)  # Disable additivity
check
shap_values = explanation.values

# !!! maybe can just get rid of this var & use X
full_feature_names = ['psi', 'depth', 'years_since_added'] +
list(encoded_feature_names)

# Use the best model for predictions on validation data
sampled_predictions = best_model.predict(X_val)
df.loc[X_val.index, 'predicted_days_until_next_maintenance'] = sampled_predictions


# Filter the DataFrame for hydrants that need maintenance soon and limit to 10
entries
upcoming_maintenance_df = df[df['predicted_days_until_next_maintenance'] <
365].head(10)
print("Upcoming Maintenance Predictions for Selected Hydrants:")
for index, row in upcoming_maintenance_df.iterrows():
    if row['predicted_days_until_next_maintenance'] < 0:
        print(f"Hydrant #{index} needed maintenance {-
int(row['predicted_days_until_next_maintenance'])} days ago.")
    else:
        print(f"Hydrant #{index} needs maintenance in
{int(row['predicted_days_until_next_maintenance'])} days.")


# SHAP summary plot
shap.summary_plot(shap_values, X_train, feature_names=full_feature_names)

# Ensure that the base value is a scalar
if isinstance(explanation.base_values, np.ndarray):
    base_value = explanation.base_values[0]  # Assuming a single output model
else:
    base_value = explanation.base_values

# SHAP Decision Plot
# Note: For decision plot, we typically use a single observation. Here, we use the
first one for demonstration.
# Select the first 10 observations for plotting
selected_observations = X_train.iloc[:10]
selected_shap_values = shap_values[:10] # Assuming shap_values is a 2D array where
each row corresponds to an observation

# SHAP Decision Plot for the first 10 observations
shap.decision_plot(
    base_value,
    selected_shap_values,
    selected_observations,
    feature_names=full_feature_names
)


# SHAP Dependence Plot
# Plotting the dependence of SHAP values for 'psi' as an example
shap.dependence_plot('psi', shap_values, X_train, feature_names=full_feature_names)
```