Wildfires Prediction - Final Exercise

Omer Siton - 316123819
Itay Ottenheimer - 209493519
Shay Cohen - 207283094
Mia Segev Gal - 208730960

1. Exploring and Cleaning the Data

We began the project by reviewing the data, removing features that could cause issues, and dealing with missing data. Our goal was to understand how each feature works for a solid start.

1.1. Leakage features

We started by reviewing the features, and determining which ones introduce leakage and removing them from the dataset:

1.1.1. **NWCGREPORTINGAGENCY** - The unit that wrote the report might be related to the fire cause.

1.1.2. The following features reveal the unit that wrote the report about the fire, except this is a "future" feature, this might be closely related to the fire cause (some units might be reporting for a specific fire cause):

1.1.2.1. **NWCGREPORTINGUNIT_ID**

1.1.2.2. **NWCGREPORTINGUNIT_NAME**

1.1.2.3. **SOURCEREPORTINGUNIT**

1.1.2.4. **SOURCEREPORTINGUNIT_NAME**

1.1.3. **FIRE_CODE** - This can lead to an external database where the fire cause appears.

1.1.4. **FIRE_NAME** - This is a leakage, the fire name could contain indicators to the cause of the fire. For example "LIGHTNING 4-3" from row 1077 directly reveals the cause of the fire.

1.1.5. **COMPLEX_NAME** - This is leakage because the complex that handled the fire eventually might be related to the fire type.

1.1.6. **STATCAUSECODE** - This is what we are trying to predict.

1.1.7. **STATCAUSEDESCR** - This is what we are trying to predict.

1.1.8. **CONT_DATE** - This is a "future" date unknown at the fire's time.

1.1.9. **CONT_DOY** - This is a "future" date unknown at the fire's time.

1.1.10. **CONT_TIME** - This is a "future" date unknown at the fire's time.

1.2. Multiple class incidents

1.2.1. In order to make sure the class labels are as expected, we created a hash for each record according to the following fields: DISCOVERY_DATE, DISCOVERY_TIME, LONGITUDE, LATITUDE and removed hashes that had more than one STAT_CAUSE_DESCR value.

1.2.2. Overall we've removed 163 rows that had more than a single class assignment.

1.3. Exploring features

1.3.1. Dropped features

1.3.1.1. The following do not add much more information about the fire cause, and might make the processing more difficult:

1.3.1.1.1. **ICS_209_NAME** - Almost all of the names are unique which makes it hard to get any valuable information from that feature so we decided to remove it.

1.3.1.1.2. The ID's features are irrelevant for the analysis - **MTBS_ID, LOCAL_INCIDENT_ID, LOCAL_FIRE_REPORT_ID, FOD_ID, FPA_ID, OBJECTID, ICS_209_INCIDENT_NUMBER**

1.3.1.1.3. **Shape** - We tried converting this feature into a geo-object, but after many attempts we've decided to work with coordinates instead.

1.3.1.1.4. **OWNER_DESCR** - We've decided to remove this feature since it has ~53% of missing rows.

1.3.1.1.5. **FIRE_YEAR** - After comparing it to DISCOVERY_YEAR we saw that all the values are the same so we removed it.

1.3.1.1.6. **COUNTY** - After checking county values we observed that there are 2934 unique values with over 35% nulls. After checking for ways to fill the gaps we decided to remove the county and lean on FIPS as our identification for the area.

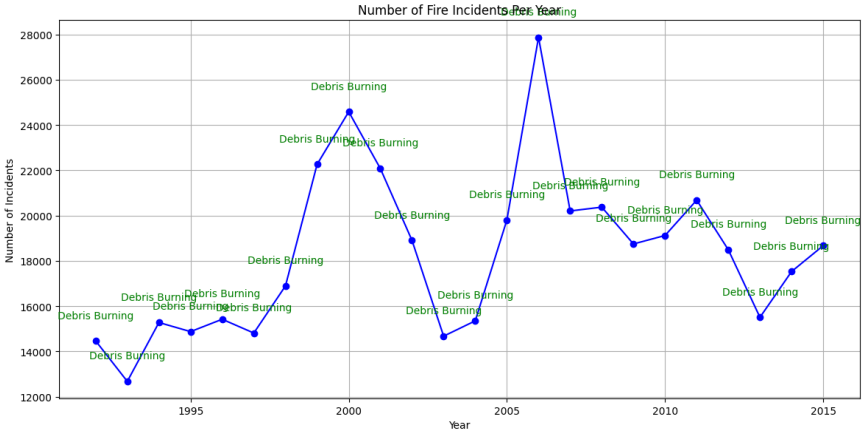
1.3.1.1.7. **FIPS_NAME** - We are using the 'FIPS CODE' so this feature is redundant.

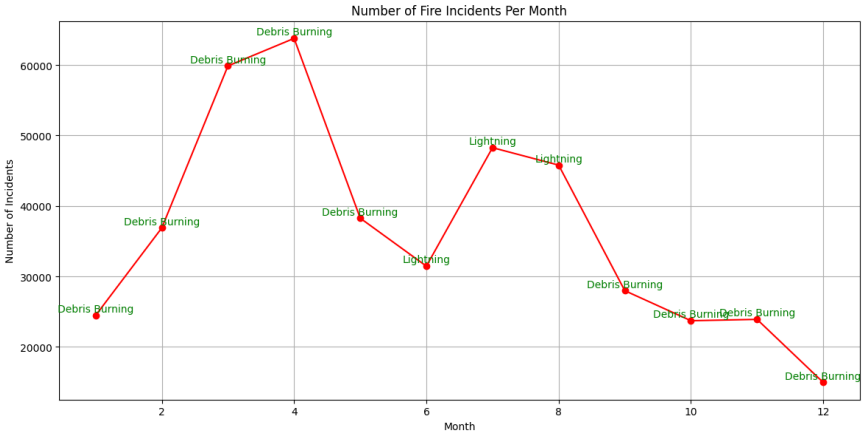
1.3.1.1.8. **MTBS_FIRE_NAME** - Name feature with only 3000± non-null values and with many unique values. We have decided to drop this column.

1.3.2. Used features

1.3.2.1. **DISCOVERY_DATE** - Seems like a very important feature. We decided to break it down to DISCOVERY_YEAR, DISCOVERY_MONTH, DISCOVERY_DAY, and timestamp. Analysis of the timeline per year shows that 2006 was the year with the most fires while Debris burning was the top cause of fire per year. When we plot the data points monthly we can see that most fires occur in April and are

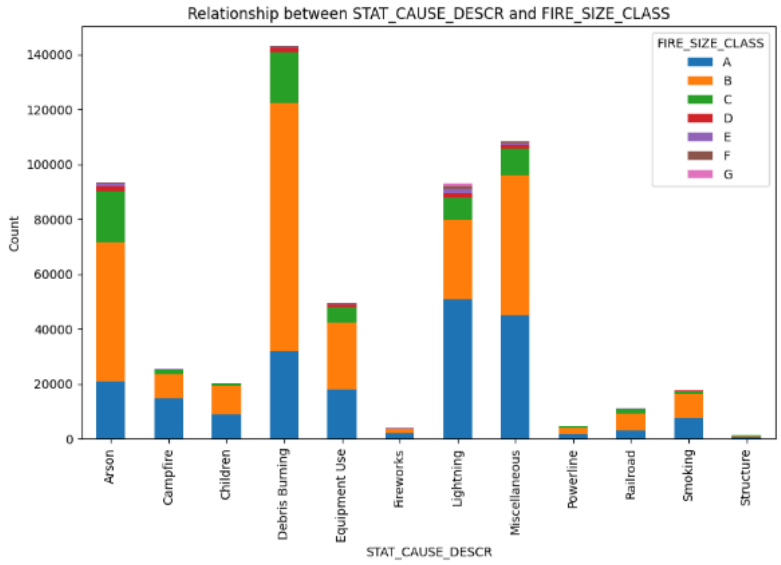
caused by Debris burning.





1.3.2.2. **FIRE_SIZE_CLASS** - observing the correlation between both FIRE_SIZE_CLASS and STAT_CAUSE_DESCR we could see that most of the incidents are B class size. We will

transform this feature using one hot encoding.

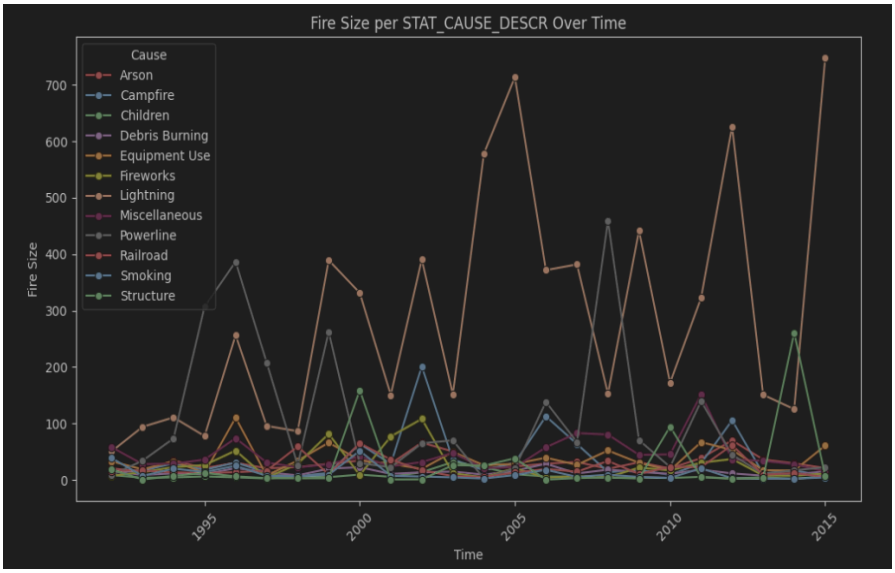


1.3.2.3. We've replaced null values with WKB of the coordinates.

1.3.3. **FIRE_SIZE -**

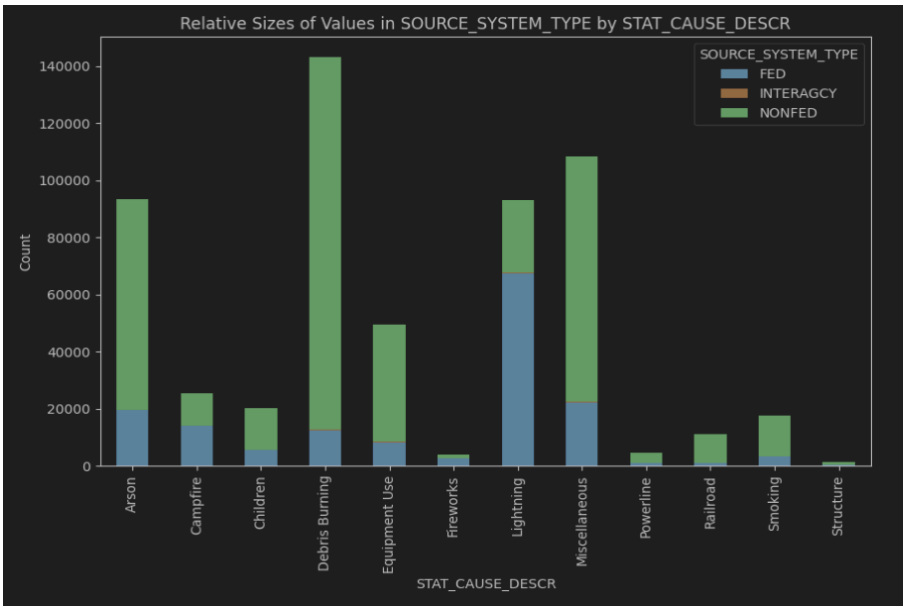
1.3.3.1. We've replaced null values with the median

1.3.3.2. As we can see from the graph, there are different size values for different classes, which means this feature can be highly relevant:

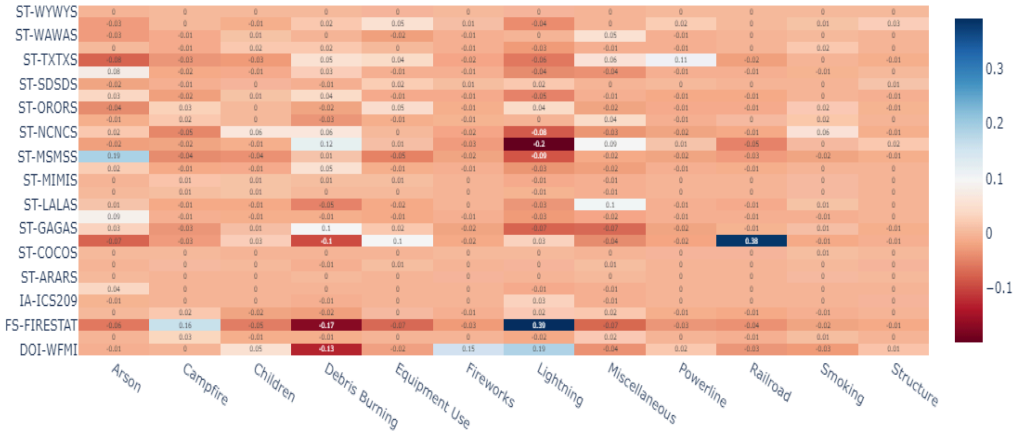


1.3.4. **SOURCE_SYSTEM_TYPE -**

1.3.4.1. As we can see, this feature's values distribution varies over different classes, which means this feature can be useful:



1.3.5. **SOURCE_SYSTEM** - Categorical feature with 32 classes and no null values, correlated to SOURCE_SYSTEM_TYPE but more detailed and broad.

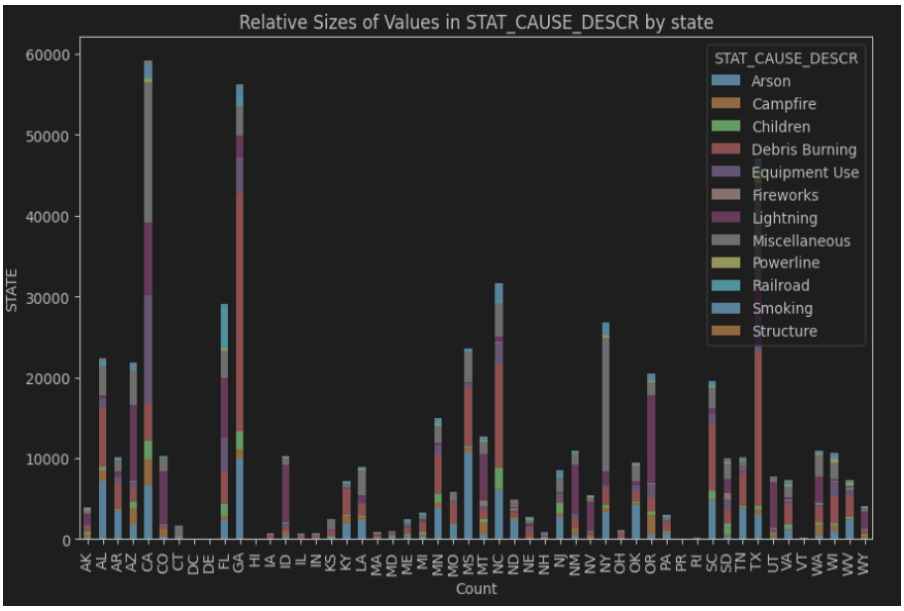


1.3.6. **STATE** -

1.3.6.1. We've replaced nulls with 'UNKNOWN'.

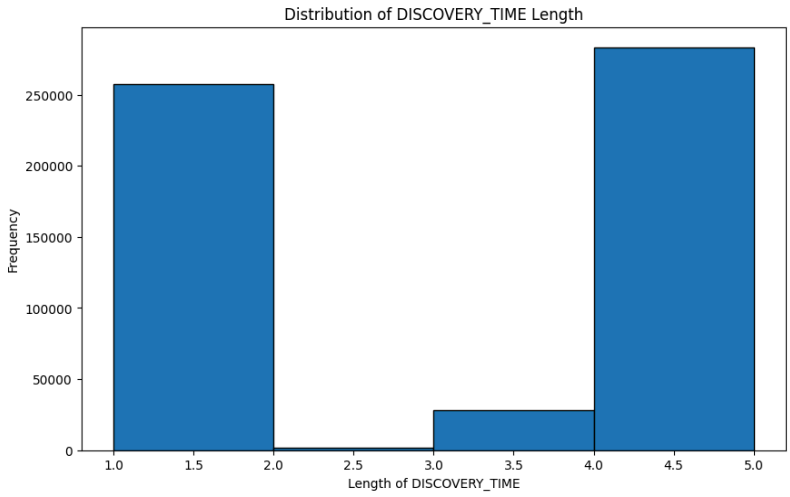
1.3.6.2. Using a similar graph as before, we can see that different states have different state values distribution. This might

indicate the feature can be useful.



1.3.7. **DISCOVERY_TIME** - This feature contains null values and many inaccuracies. We expect this feature to be in a 24-hour time format. The values with a length of 1 are the null values I filled with 0. We will add a leading zero for all the time values with a length of 3. Steps taken:

1.3.7.1. Filling NaN values with 0.
1.3.7.2. Converting the feature to integer and then to string.
1.3.7.3. Padding values of length 3 with a leading zero.
1.3.7.4. Finding the mode of the feature excluding 0.
1.3.7.5. Filling values of length 1 and 2 with the mode.
1.3.7.6. Extracting hour and minute from the feature.
1.3.7.7. Converting hour and minute to cyclic features.
1.3.7.8. Dropping the original DISCOVERY_TIME feature.



1.3.8. **FIPS_CODE** - This feature is a code (number) that contains two parts: the first is the state code from 01 - 56 and the second part is a code for the county inside the state. Reference - https://transition.fcc.gov/oet/info/maps/census/fips/fips.txt

1.3.9. **LONGITUDE**, **LATITUDE** - Those features can be highly relevant for extracting geographic-related info.

1.4. Thinking process

After reviewing the dataset features, we began thinking about new features that can be useful to add.

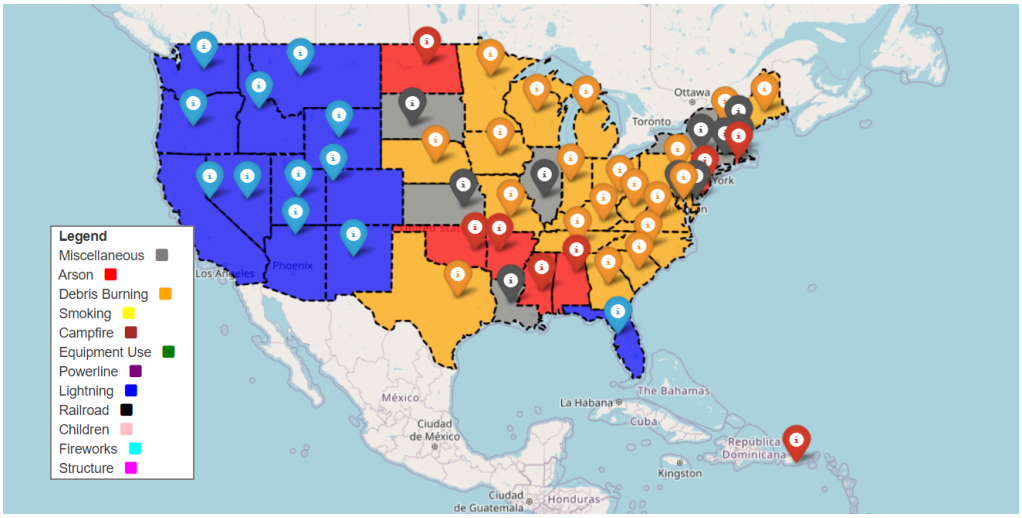
1.4.1. Ideas for features

1.4.1.1. Social economic by location
1.4.1.2. Biome (type of location: forest, city, desert) by location
1.4.1.3. Industrial zones\Urban\Neighborhoods\Nature
1.4.1.4. Population density per area
1.4.1.5. Seasons\Mean
1.4.1.6. Temperature in this day of year for each location
1.4.1.7. BBQ areas
1.4.1.8. Gas stations
1.4.1.9. Height
1.4.1.10. Is near electricity infrastructure

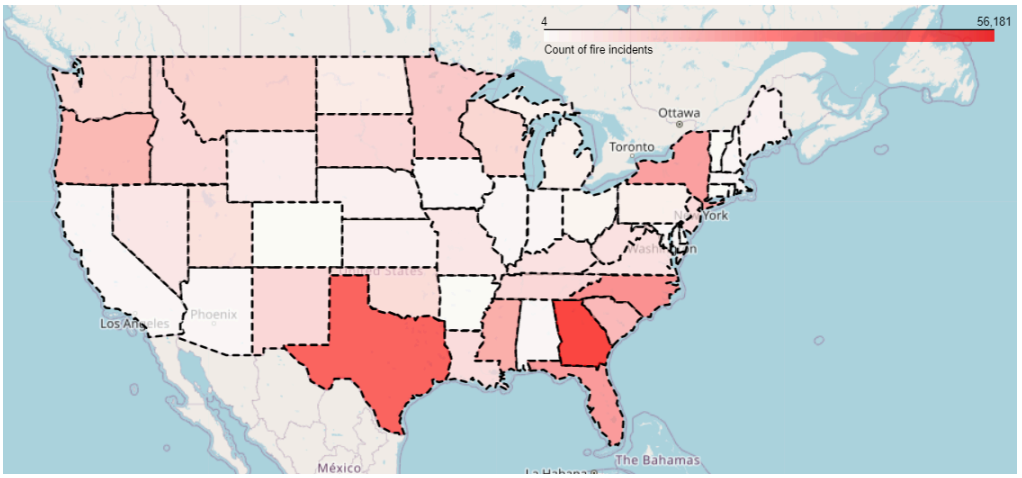
1.4.2. After creating a list of ideas, we began searching for relevant datasets that can be used to create each of them (as detailed in the feature extraction section 2.1).

1.5. Geospatial Data

1.5.1. We attempted to understand the geo attributes of the data better. The next figure shows for each state the main cause of incidents.



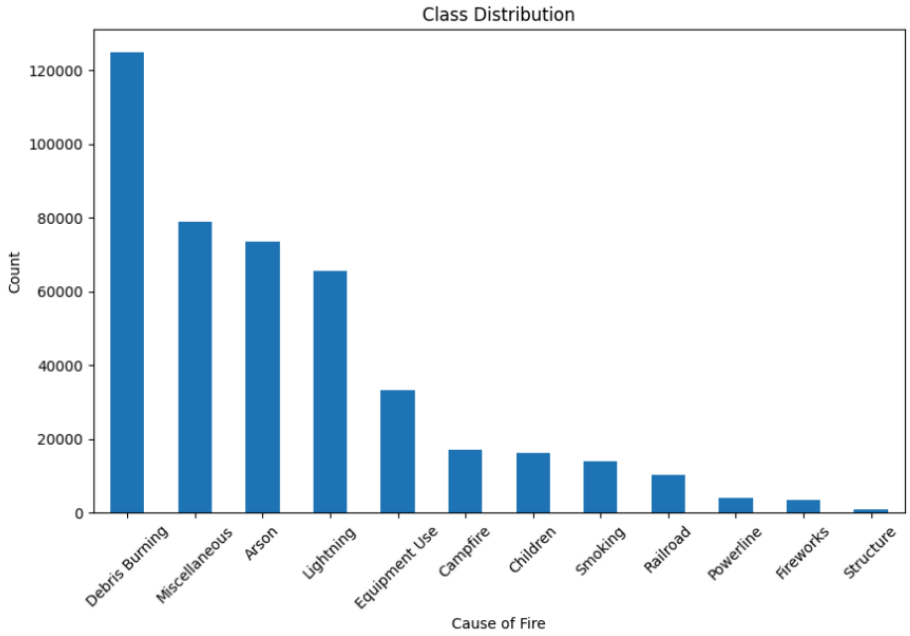
Reviewing the figure we can see that the main cause of fire in the west of the US is lightning.
This information could be useful later when reviewing feature importance. We also plotted the amount of incidents in each state to see if there are places that tend to have more fires than others.



We can see that most of the incidents occur in Texas and Georgia.

1.6. Dealing with Imbalanced data

1.6.1. Upon examining the class distribution, it's evident that the data exhibits a significant imbalance. Notably, over 28% of recorded incidents pertain to debris burning. This suggests that models might lean towards classifying cases as debris burning rather than any other classes.

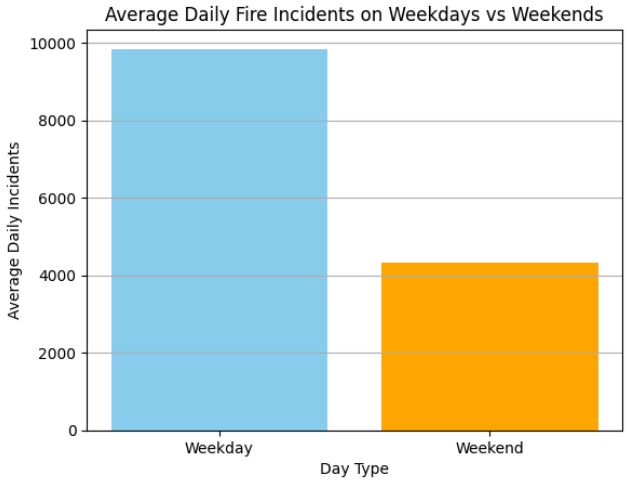


To deal with this, we decided to lean toward decision tree-based algorithms that are known to perform well on imbalanced datasets.

2. Feature Extraction and Engineering

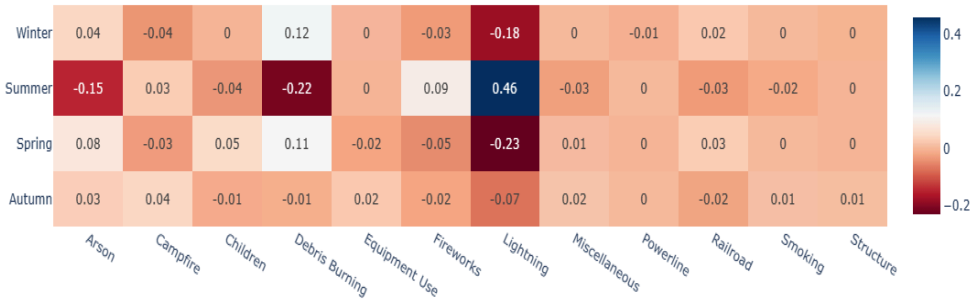
2.1. Feature Extraction

2.1.1. **IS_WEEKDAY** - Boolean value representing whether the fire occurred on a weekend or not. We can see that on average most of the fires occur on the weekdays. That is consistent with our assumption as most people and factories are active on the weekday



2.1.2. **SEASON** - we can use the data feature to extract some more general information about the season of the event.
We can see that some seasons have some correlation with some

causes.



2.1.3. **DISCOVERY_YEAR**, **DISCOVERY_MONTH**, **DISCOVERY_DAY -** break down of DISCOVERY_DATE feature.

2.2. Feature Engineering

2.2.1. Demographic information

One type of information that we thought could yield valuable insights, includes factors such as age, gender, ethnicity, and education, which could be beneficial for some of our predictive classes such as Children.

2.2.1.1. Data source: US County Data

2.2.1.2. GitHub - evangambit/JsonOfCounties: A repo containing various data (demographics, employment, etc.) in JSON form.

2.2.1.3. Analysis: The dataset contains a wide array of information covering various aspects such as demographics, labor, climate, health, education, industry, and economic factors for each county. In this section, we decided to focus on information on labor, demographics, and population for each county. It includes details on the labor force, such as employed and unemployed individuals, as well as demographics like race, gender, and age groups. Additionally, it offers population estimates for 2019. This data helps paint a picture of who lives and works in each county.

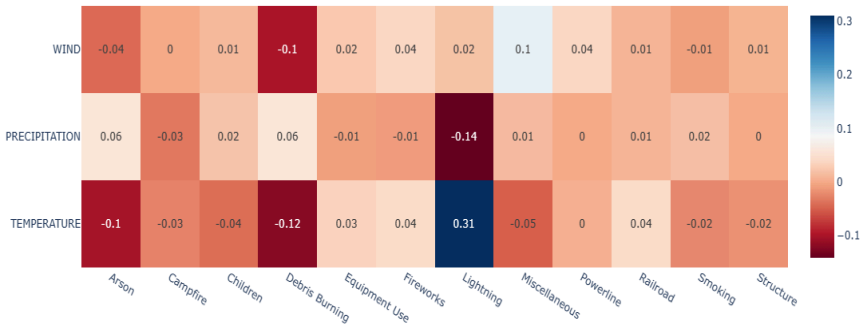
2.2.2. Weather Information

2.2.2.1. Weather information can help us figure out where fires come from because it affects how fires behave. We added information about the temperature, wind speed and precipitation for each fire event based on the date the fire started and its location.

2.2.2.2. Data source: US_counties_weather

2.2.2.3. Data Extraction: Based on the data source mentioned above, we created a new dataset that provides the mean temperature, mean precipitation, and mean wind speed for each FIPS code and month, averaged over 30 years.

2.2.2.4. Analysis: we added 3 features related to weather, temperature, wind, and precipitation. We can see some correlations between the temperature and the Lightning class:



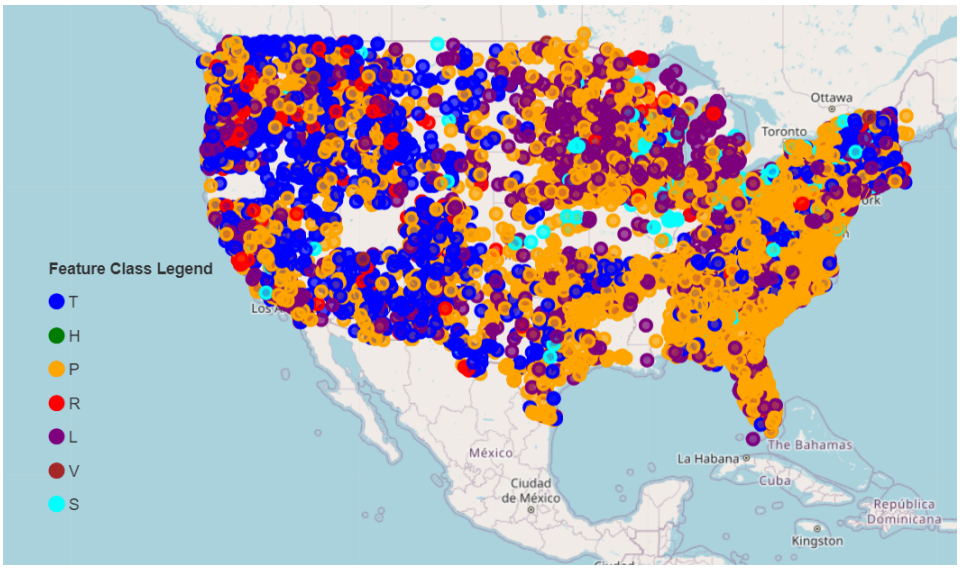
2.2.3. Geo-location information

2.2.4. Another type of information that we thought could yield valuable insights, was geo-location information, we found a huge data set that contains many locations of various landmarks in the US such as parks, lakes, roads, and many more.

2.2.4.1. Data source: [GeoNames database]

2.2.4.2. Analysis & processing: As shown in the map below, this dataset contains points of interest in most of the US. The points are categorized into one out of nine feature classes and further subcategorized into one out of 645 feature codes. We attempted to create a new feature for each class, representing the euclidean distance between each incident and the closest point of that class (for example, distance to nearest forest) as we thought this could be highly relevant for our purpose. After several attempts, we only managed to run this feature on the smallest class, resulting in an unmeaningful feature. We decided to leave this out due to inefficiency issues.

2.2.4.3. Dataset map:

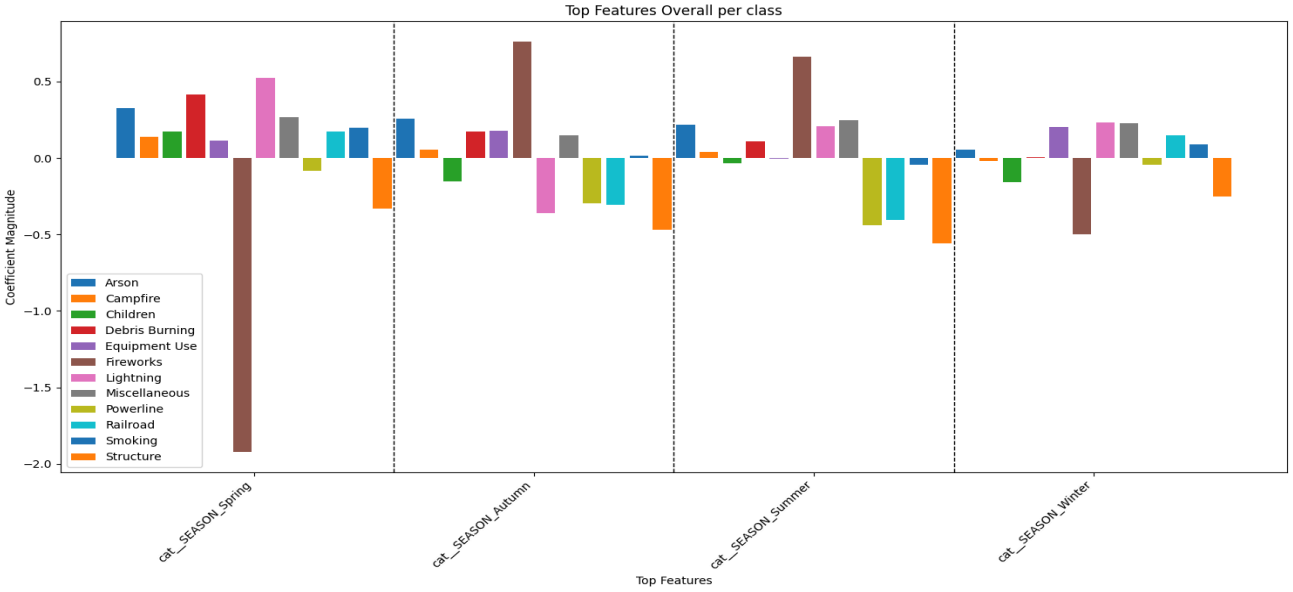


2.3. Feature importance

2.3.1. In this part we added and adjusted features in an attempt to improve the baseline performance and in general the model using outsourced data and features that are present in the given data. **All of the following analysis was made on the Logistic regression baseline.**

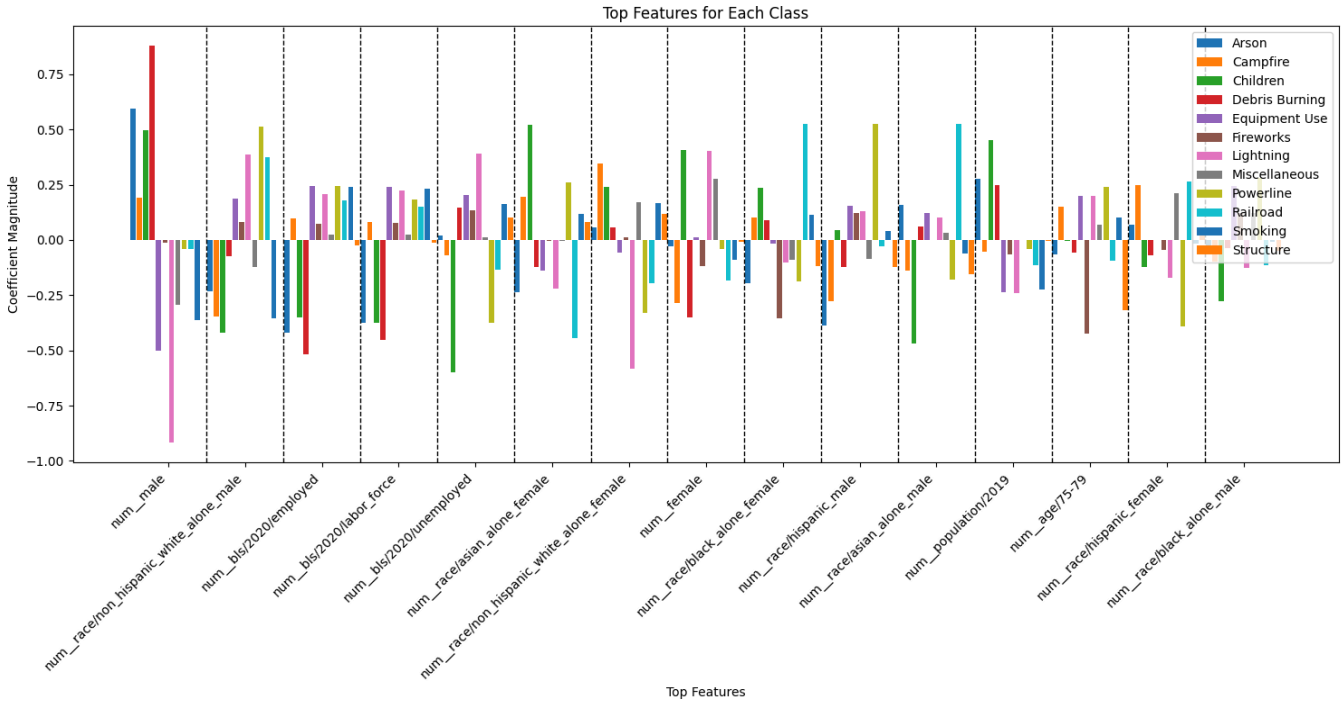
2.3.2. Season features importance

2.3.2.1. As we can see from the graph below, the season features are highly relevant for fireworks and lightning and overall seems relevant.



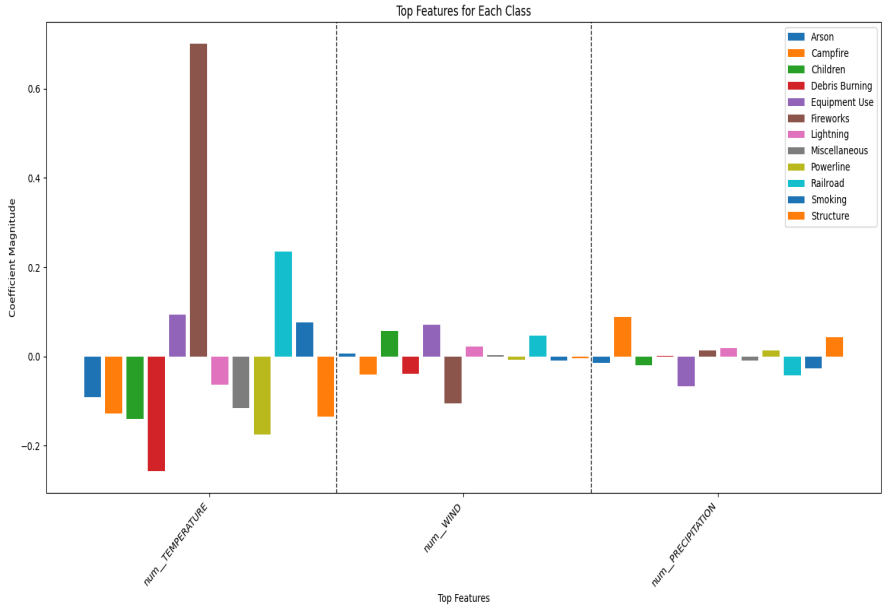
2.3.3. Demographic information importance

2.3.3.1.1. The analysis of the feature importance of the baseline model below shows that some of the new features are highly relevant, for example we can see that the number of males in the state is relatively relevant.



2.3.4. Weather information importance

2.3.4.1. When plotting the importance of the weather features, we can see that the temperature is the most significant one.

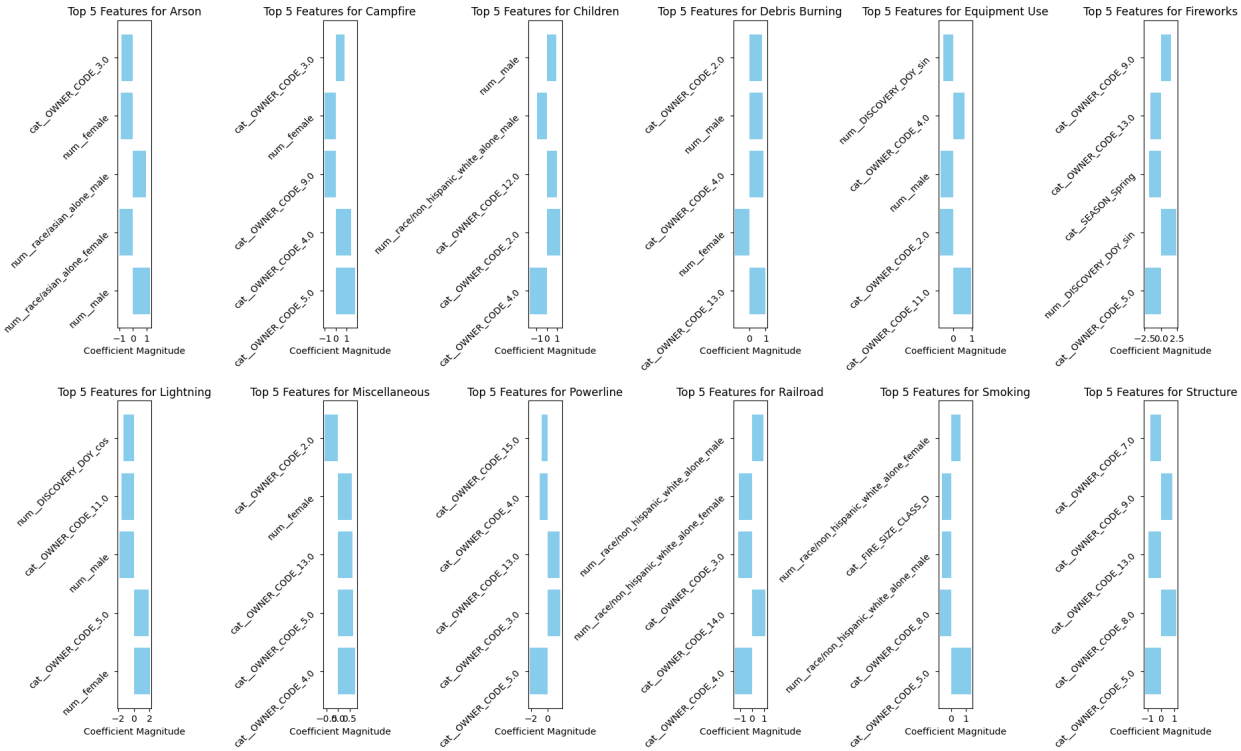


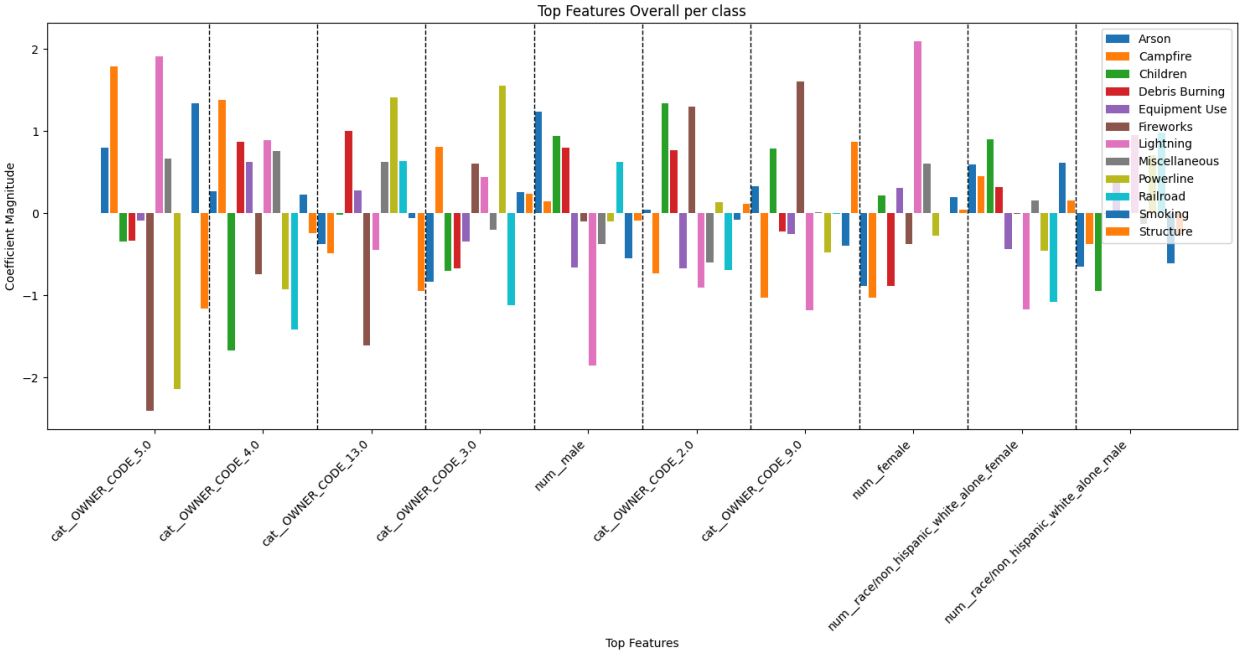
2.3.5. Baseline model features importance

2.3.5.1. At the end of the feature engineering and processing, we wanted to analyze what truly are the features that drive the model. We plotted the top 5 features for each class. We noticed that no matter which features we added, the model mostly relies on the spectral features.

2.3.5.2. In an attempt to better understand the data at hand we decided to train the baseline without ‘STATE’ and ‘FIPS_CODE’ this resulted in a shift in the features' importance. Now we can see that ‘OWNER_CODE’ is at the top of the feature importance for most classes the demographic data is stated to have an effect also.

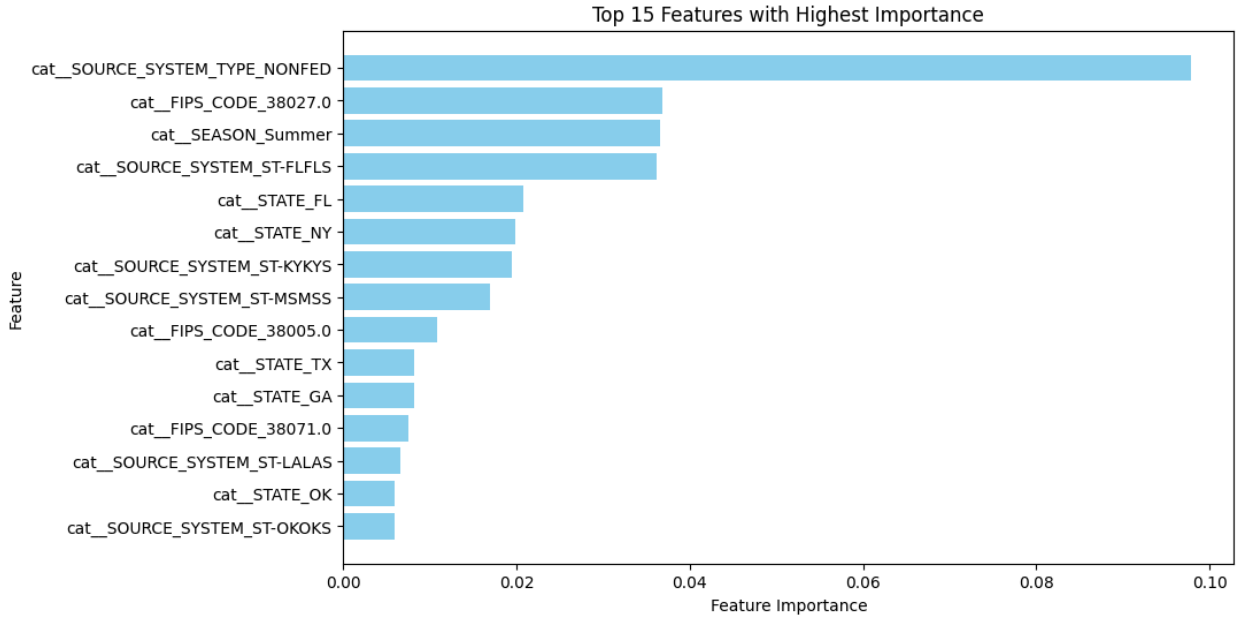
We also plotted the top features for all classes:





2.3.6. Selected model features importance

2.3.7. Analyzing feature importance of XGboost its is surprising to see that this model prefers to use other features to evaluate the class of the incidents. The smart choice by the model is probably the cause for the significant performance improvements



3. Model Selection

3.1. Baselining

When selecting our models, we considered the following aspects:

- Data Size and Quality: Larger datasets might require more complex models to capture the underlying patterns effectively.
- Feature Characteristics: Some models are better suited to handle specific types of features. For example, tree-based models handle non-linear relationships and interactions well.
- Interpretability: Understanding why decisions are made.
- Computation Resources: More complex models typically require more computational power and time. Balance the need for accuracy with available resources.
- Performance Metrics: Choose models based on the metric that best reflects the desired evaluation metric ROC_AUC_ovr_weighted.

On a large dataset where interpretability is a priority alongside performance metrics such as ROC_AUC_ovr_weighted, it's strategic to consider models that balance complexity with the ability to provide insights into their decision-making processes. Here are four models we decided to try:

3.1.1. Logistic Regression

3.1.1.1. Data Size and Quality: Scales well to large datasets and is effective when the relationship between the features and the target is linear or can be linearly separable after transformation.

3.1.1.2. Feature Characteristics: Best suited for problems where features have a linear relationship with the log odds of the target.

3.1.1.3. Interpretability: Highly interpretable, as the coefficients of the model can be directly related to the odds ratio of the corresponding features.

3.1.1.4. Computation Resources: Less demanding in terms of computation compared to tree-based models, which makes it suitable for very large datasets.

3.1.1.5. Performance Metrics: Provides a probabilistic output which can be beneficial for ROC AUC calculations; however, it might underperform if the relationships in the data are highly non-linear without appropriate transformations.

3.1.2. K-Nearest Neighbors (KNN)

3.1.2.1. Data Size and Quality: Works well with a smaller number of instances since it relies on the local neighborhood of each query point, which can become computationally expensive as the dataset size increases. It is sensitive to the scale of the data and irrelevant features, which can adversely affect performance if not preprocessed properly.

3.1.2.2. Feature Characteristics: Ideally suited for instances where similar examples lead to similar outputs, as KNN is a distance-based algorithm. It is non-parametric, making it flexible to fit the structure of the data without assuming a specific form of the function.

3.1.2.3. Interpretability: Moderate to low interpretability. While it is conceptually simple to understand how neighbors vote for class labels, the decisions are not as transparent since they are based on the distribution of neighboring points in a high-dimensional space.

3.1.2.4. Computation Resources: KNN can be quite memory-intensive, as it requires storing the entire dataset for prediction, and the computational cost of finding the nearest neighbors is high for large datasets or datasets with many dimensions (curse of dimensionality).

3.1.2.5. Performance Metrics: KNN can provide useful insights into the data structure through the lens of local similarity. However, its performance can suffer in high-dimensional spaces or when classes are not well separated. It is not inherently probabilistic, which means that it doesn't output probabilities natively, but such probabilities can be inferred from the proportion of neighbors in each class.

3.1.3. Random Forest

3.1.3.1. Data Size and Quality: More robust to overfitting compared to individual decision trees and effective with large datasets.

3.1.3.2. Feature Characteristics: Handles non-linear data well by building multiple trees and averaging their predictions, which also helps in feature selection.

3.1.3.3. Interpretability: Offers moderate interpretability; you can look at feature importance derived across many trees, although the ensemble nature makes it less clear than a single decision tree.

3.1.3.4. Computation Resources: Requires more computational power due to building multiple trees, but training can be easily parallelized.

3.1.3.5. Performance Metrics: Generally performs well on a variety of metrics, including ROC AUC, due to its method of reducing variance and bias.

3.1.4. XGBoost (Extreme Gradient Boosting)

3.1.4.1. Data Size and Quality: Efficient with large datasets, handles overfitting through regularization, and manages missing values well.

3.1.4.2. Feature Characteristics: Captures complex non-linear patterns using gradient boosting, which sequentially optimizes a series of weak prediction models.

3.1.4.3. Interpretability: This provides insights into feature importance, though less transparent than a single decision tree due to its complex model structure.

3.1.4.4. Computation Resources: Computationally intensive but optimized for efficiency, including GPU support.

3.1.4.5. Performance Metrics: Excels in metrics like ROC AUC, precision, and recall, effectively minimizing bias and variance through boosting.

3.2. Results

3.2.1. Random Forest Performance

3.2.1.1. ROC AUC Score: 0.8345

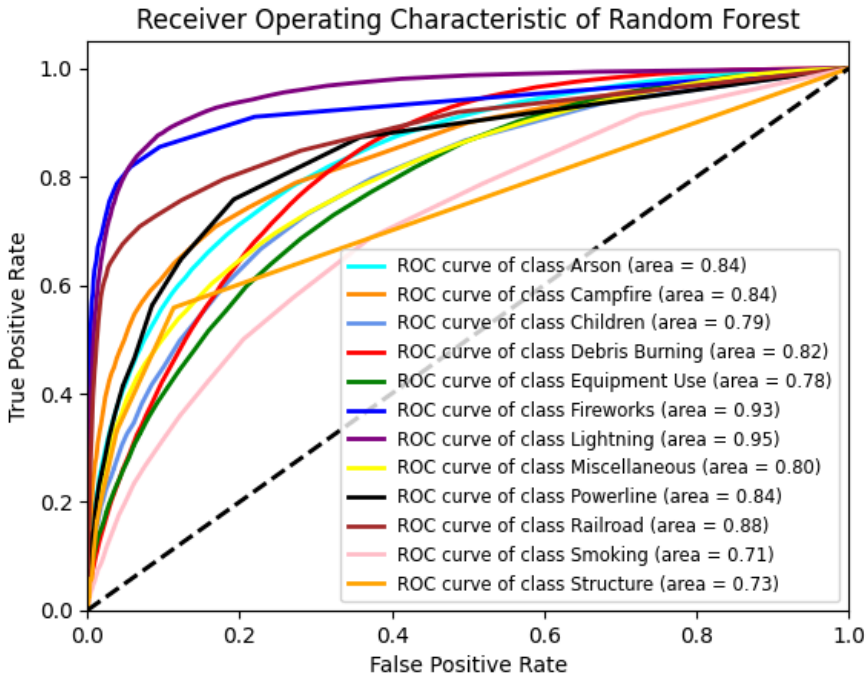
3.2.1.2. Accuracy: 0.55

3.2.1.3. Precision: 0.52

3.2.1.4. Recall: 0.55

3.2.1.5. F1 Score: 0.52

3.2.1.6. The ROC curves for the Random Forest model show a range of area under the curve (AUC) values for different classes, with the highest for class Lightning (AUC = 0.95) and the lowest for class Smoking (AUC = 0.71).



3.2.2. Logistic Regression Performance

3.2.2.1. ROC AUC Score: 0.8153

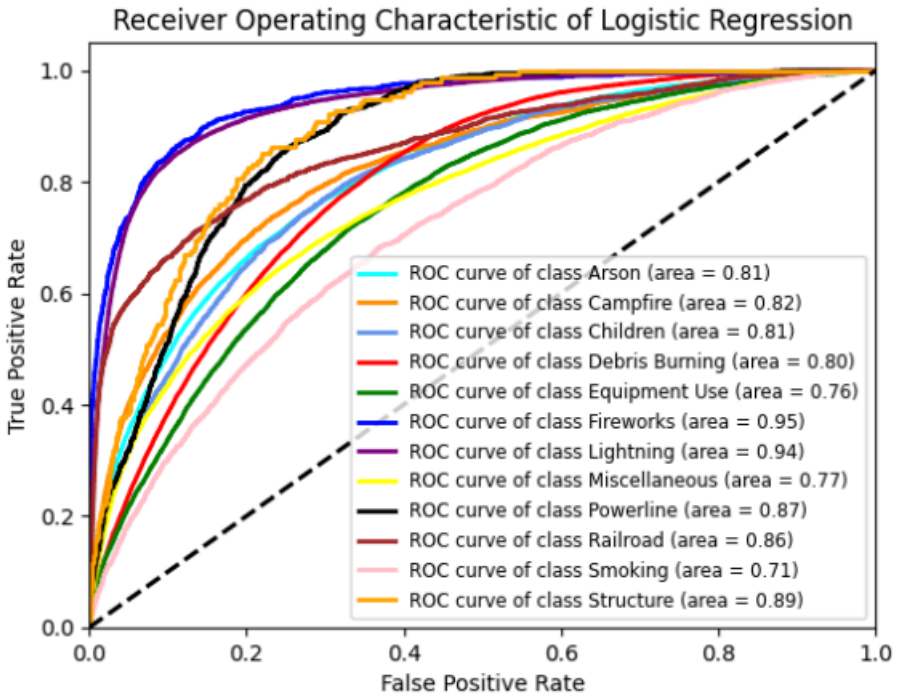
3.2.2.2. Accuracy: 0.51

3.2.2.3. Precision: 0.47

3.2.2.4. Recall: 0.51

3.2.2.5. F1 Score: 0.46

3.2.2.6. The ROC curves for Logistic Regression show class fireworks having the highest AUC (0.95) and class Smoking the lowest noted (AUC = 0.71).



3.2.3. K-Nearest Neighbors Performance

3.2.3.1. ROC AUC Score: 0.7276

3.2.3.2. Accuracy: 0.44

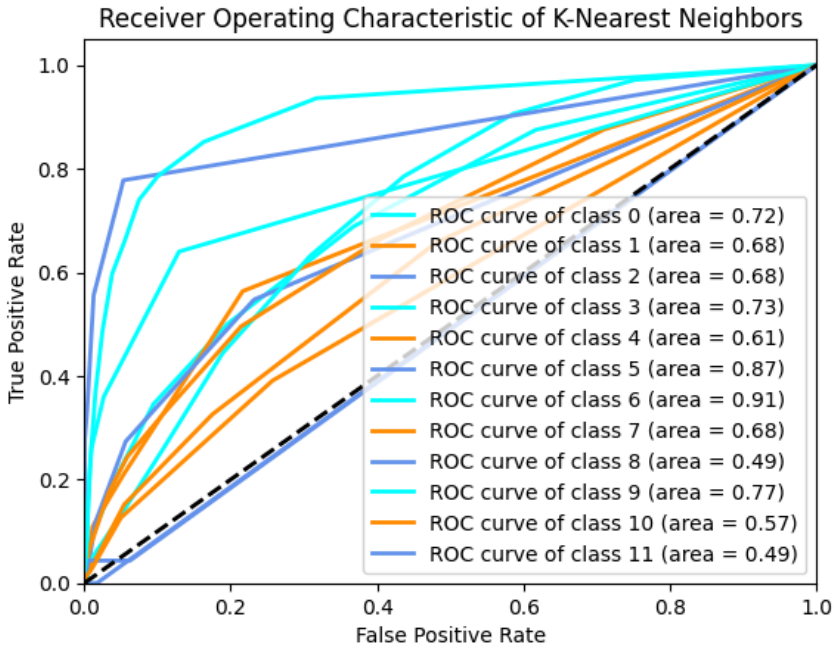
3.2.3.3. Precision: 0.39

3.2.3.4. Recall: 0.44

3.2.3.5. F1 Score: 0.40

3.2.3.6. The AUC values for the K-Nearest Neighbors model are generally lower than the other two models, with the highest for class Lightning (AUC = 0.91) and the lowest for

class Powerline (AUC = 0.49).



3.2.4. XGBoost Performance

3.2.4.1. ROC AUC Score: 0.85

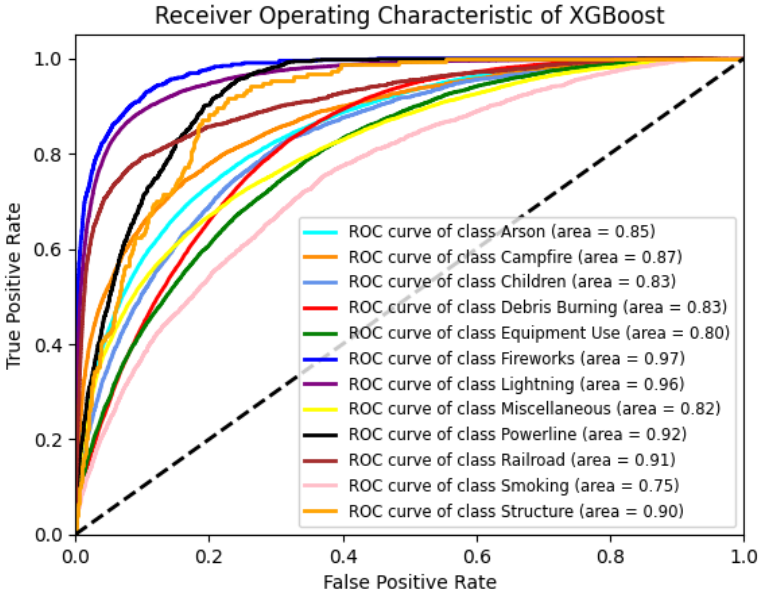
3.2.4.2. Accuracy: 0.55

3.2.4.3. Precision: 0.53

3.2.4.4. Recall: 0.55

3.2.4.5. F1 Score: 0.53

3.2.4.6. The AUC values for the XGBoost model are generally higher than the other models, with the highest for class Fireworks (AUC = 0.97) and the lowest for class Smoking (AUC = 0.75).



3.3. Model Comparison

3.3.1. Overall Performance: XGBoost shows the best overall performance in terms of ROC AUC, which is our evaluation objective. It also has competitive accuracy, recall, and F1 scores compared to the other models.

3.3.2. Class-Specific Performance: The Random Forest and Logistic Regression models have similar strengths in specific classes (e.g., Lightning and Fireworks respectively), but XGBoost consistently outperforms other models across various classes, showing particular strength in the class Fireworks with the highest AUC of 0.97.

3.3.3. Weaknesses: All models struggle with the class Smoking, indicating potential challenges in distinguishing this class from others or insufficient representative features.

3.3.4. Precision and Recall: Precision is relatively low across all models, particularly in KNN, which suggests that all models, to varying degrees, make a significant number of false positive errors. This could be an area for improvement, possibly by refining feature selection or tuning other model-specific parameters.

3.3.5. Suitability: Given its high performance across most metrics and robustness in handling different classes, XGBoost is the most suitable model for scenarios demanding high reliability and accuracy in multiclass classification problems. Random Forest also presents a viable option, offering a balanced mix of accuracy and ROC AUC, but **we will proceed with XGBoost**.

3.4. Hyperparameter Tuning + Cross Validation

We wanted to improve our best performance model from the baseline step XGBoost with a hyperparameter tuning process as we saw in class. These are the features we chose using in the Randomized Search procedure based on our multiclass classification and ROC_AUC_ovr objective.

- model__objective: Including multi:softmax and multi:softprob. multi:softmax is useful for predicting class labels directly, while multi:softprob provides the predicted probability of each class, which is essential for calculating ROC AUC, particularly useful in performance evaluation metrics that require probability outputs (XGBoost Documentation).
- model__booster: The choice among 'gbtree', 'gblinear', and 'dart' boosters allows exploring different boosting algorithms. 'gbtree' uses tree-based models which are powerful for handling varied data types

and interactions. 'gblinear' provides a linear model alternative, which can be faster and more suitable for high-dimensional sparse data. 'dart' includes dropout techniques that can enhance performance by preventing overfitting, particularly in datasets prone to noise (XGBoost Documentation).

- model__eval_metric: 'auc' is specified to focus on the area under the ROC curve, a critical measure for classification tasks. It assesses model ability in distinguishing between classes, which is vital for understanding model efficacy in real-world scenarios (XGBoost Documentation).
- model__learning_rate and model__n_estimators: These control the speed and extent of model learning. A lower learning rate with more estimators can lead to better learning, albeit at a slower pace, ensuring thorough exploration of the solution space. This balance helps in achieving a robust model while avoiding overfitting (XGBoost Documentation).
- model__max_depth: This parameter determines the depth of the trees. Deeper trees can model more complex patterns but are also prone to overfitting. Adjusting this parameter helps in finding an optimal model complexity that captures sufficient data characteristics without excessively fitting to the noise in the training data (XGBoost Documentation).
- model__alpha and model__lambda: These are L1 and L2 regularization parameters, respectively, that help in reducing model complexity and enhancing generalization. By penalizing large weights, these regularization terms ensure that the model remains robust and performs well on unseen data (XGBoost Documentation).

Each parameter is chosen based on its impact on model performance, ability to generalize without overfitting, and computational efficiency. The final selection of these parameters is validated using cross-validation (with 5 folds) to ensure that the model performs optimally across different subsets of the data and various evaluation metrics.

```
param_dist = {
  'model__objective': ['multi:softmax', 'multi:softprob'],
  'model__booster': ['gbtree', 'gblinear', 'dart'],
  'model__eval_metric': ['auc'],
  'model__learning_rate': np.linspace(0.1, 1, 10),
  'model__n_estimators': range(100, 200, 10),
  'model__max_depth': range(5, 15, 1),
  'model__alpha': np.linspace(0.5, 1.5, 10),
  'model__lambda': np.linspace(0, 0.5, 10),
```

}

3.4.1. XGBoost Performance after tune

3.4.1.1. ROC AUC Score: 0.87

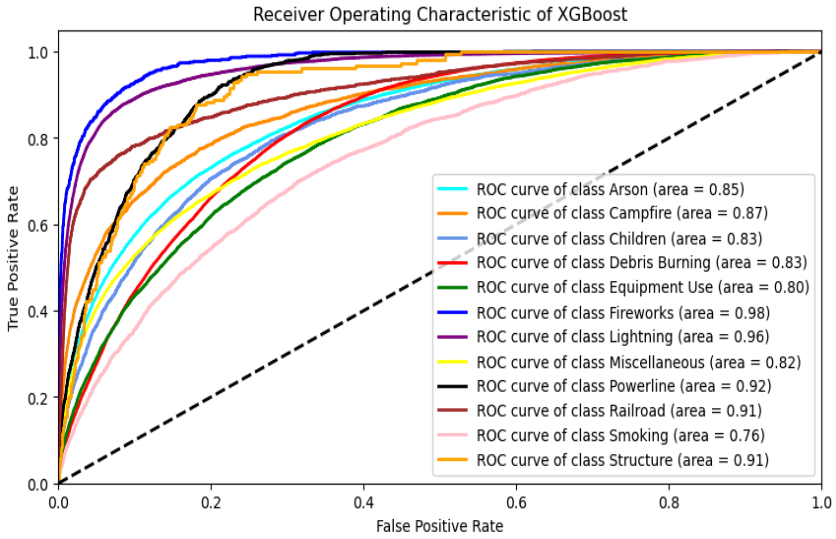
3.4.1.2. Accuracy: 0.5551

3.4.1.3. Precision: 0.4454

3.4.1.4. Recall: 0.3565

3.4.1.5. F1 Score: 0.3741

3.4.1.6. The AUC values for the XGBoost model are generally higher than the other models, with the highest for class 5 (AUC = 0.98) and the lowest for class 10 (AUC = 0.76).



The tuning of hyperparameters resulted in a slight improvement in the model's performance, with the untuned model achieving an AUC of 0.85 and the tuned model reaching an AUC of 0.87. However, it's important to consider the substantial investment of time and resources required for this tuning, approximately four hours, when evaluating the benefits of this improvement.