Wildfires Big Data Analysis

Team composition:

- Alvaro Tapia
- Sanidhya Singh
- Chinmay Ghaskadbi
- Jorge R Meza Cabrera
- Ahmed Mosaad
- Nakshatra Sharma
- Evan Dougherty

Credit Statement:

The whole team planned to collaborate effectively by constantly communicating our findings, results, and improvements in the project along with being present in the group meetings. In most cases, the work to be done was split among the members of the group to ensure efficiency and feasibility. A description of the contributions from each member will be stated in the following paragraphs:

- Alvaro Tapia: Helped with the initial organization of the group and project. The main tasks I performed were the abstract, introduction, data description, and the creation of the representative schematic with the inclusion of all the important information from the methodologies. Helped with the methods part of the PowerPoint slides. Assisted in the integration of ICDS parallelism, and also gave potential ideas for investigations at the beginning of the project. Supported in this document's design.

- Chinmay Ghaskadbi: Data Collection: Found the initial dataset used in the project. Dove into the data and found the appropriate target variable. Researched methodologies such as PCA and K-means and their implication in our project. Worked on multiple sections of the report, and try to summarize key findings.

- Sanidhya Singh: Wrote part of code for loops determining PCA variance as well as Kmeans loop to determine optimal number of clusters as well as generate cluster visuals for features following PCA. Modified decision tree code for regression implementation and ran decision tree code within cluster environment to observe runtime performance, recorded and documented results.

- Jorge R Meza Cabrera: Wrote code to find the optimal K value and implemented PCA. Also helped with preliminary Kmeans implementation. Modified Lasso Regression to improve the RMSE result.

- Nakshatra Sharma: Researched various methodologies learnt in class and previous lab to help with their integration and documentation for this project. Assisted with summarizing the data science techniques used for this report.

- Ahmed Mosaad: Researched the data set to understand the context of our project. Ran the k-means in cluster mode. Researched some of the methods such as k-means and decision trees to decipher results. Assisted in the documentation.

- Evan Dougherty: Conducted research on persist() as well as on random forests, decision trees, and other machine learning models to help guide decisions on what we were going to do. Helped create the decision tree model. Assisted with documentation.

Abstract - Wildfires are currently a major environmental issue that countries all around the world are facing. Places such as the United States, Brazil, and Australia, are dealing with the loss of wildlife, human life, and billions of dollars in forests, homes, and structures due to wildfires spreading out of control. Much research has been done in the past to develop potential models, analyses, and programs in order to prevent the spread of wildfires by detecting their potential direction or predicting their final location. However, in many cases, the models have had poor results. Lack of big data and substandard modeling have been the main culprits for this failure. Therefore, the purpose of this project is to utilize the official biggest open-source wildfire dataset for large-scale processes in order to develop machine learning models for wildfire prediction occurrence seeking higher accuracy than past papers with the usage of the parallel clustering process. Through thorough experimentation, we decided that a decision tree model would be the best for this project. We performed PCA analysis to first reduce the dimensionality of the data and using a decision tree, we were able to predict FRP scores with a root mean squared error of 47-53. Given frp scores at no more than a few hundred, our model still has a ways to go regarding accuracy.

I. INTRODUCTION & BACKGROUND

Wildfire prediction occurrence is an ongoing investigation by a plethora of researchers due to the impact the consequences of this disaster event has not only on the ecosystem but also on the population in terms of health and wealth. It is argued by the National Oceanic and Atmospheric Administration that hundreds of houses, buildings, and homes are constantly being destroyed by these fires every year. This gains more importance as the quality of air humans need is profoundly affected and is a major challenge to the United States and many other countries to control this and recover from the air quality negative impact. For this purpose, many different investigations were developed that only take into consideration single features for wildfire prediction, such as by uniquely analyzing satellite images, some investigate historical data, others focus on the climate change data for their prediction, and some integrate social media information by gathering fire incidents. Nevertheless, the unique implementation of these parameters as single research papers significantly reduces the efficiency of the prediction as well as the potential accuracy that it may achieve if the majority of these features are combined. However, the integration of these parameters in many cases represents a huge challenge to many researchers due to the incapability of integrating big data as their main dataset of investigation as sometimes run time errors may happen, or just for feasibility they choose to avoid these processes.

In this sense, this project aims to predict the Fire Radiative Power (FRP) to determine the intensity and occurrence of fire emissions, by implementing machine learning algorithms such as K-means and Decision Trees, and dimensionality optimizer tools such as PCA to develop accurate models. It will be utilized as the official biggest large-scale open-source wildfire dataset that has all previously mentioned features joined. It is aimed to reach the attention of potential investigators or investors who may want to continue improving the project through the implementation of other machine learning models and different analytical

algorithms. As well as, make public and utilize this investigation for the benefit of assessing the risk of wildfires by strategically and efficiently preventing wildfires and taking action when needed to mitigate them.

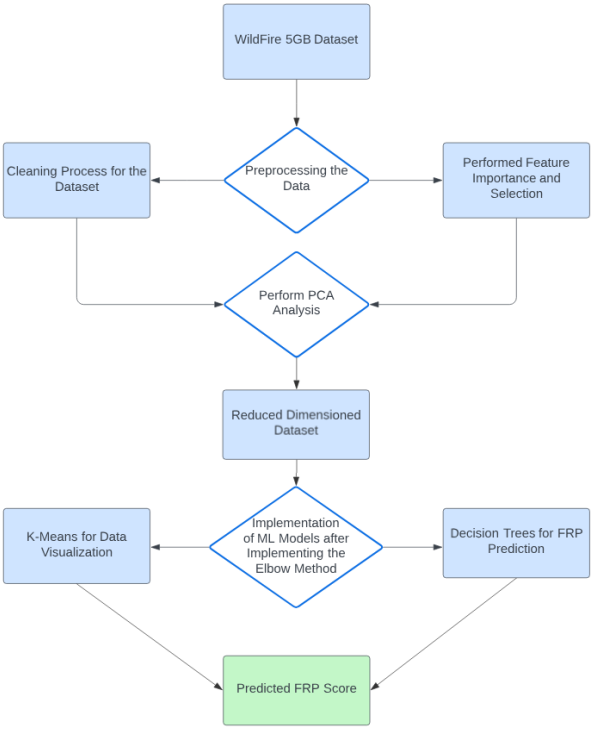


Fig 1. Representative Schematic

II. DATASET DESCRIPTION

For this investigation, we selected the existing largest official WildFireDB dataset as the data source. This dataset is significant as it utilizes previous historical wildfire occurrences with relevant features extracted from satellite imagery to form an open-source wildfire dataset. This is the first open-source wildfire dataset that contains information about past wildfire occurrences and also contains data about features extracted from satellite imagery. It contains 17 million data points collected from 2012 to 2018. There is a small version of this dataset with a size of 200 Mb and also a full version with a size of 5.3 GB. This dataset contains cumulative information from different websites and sources such as Landfire for topography, Visible Infrared Imaging Radiometer Suite (VIIRS) for thermal anomalies, Metostat for weather data, and other relevant raw data from the National Oceanic and Atmospheric Administration (NOAA). Characteristically, each data point from the dataset is a polygon cell on fire that occurs in a single day, each neighbor from these polygons is denominated as Fire Radiative Power (FRP). According to the authors, every FRP cell has different conditions such as “the maximum, minimum, median, mode, sum, mode, and count values of canopy base density, height, cover, height, existing vegetation cover, height, type. Each instance also consists of the following weather data – the average, minimum, and maximum temperature, total precipitation, average atmospheric pressure, and relative wind speed.” (Singla, Diao, Mukhopadhyay, Eldawy, Shachte, & Kochenderfer, 2020)

Link to Dataset:

Github: https://wildfire-modeling.github.io/

Direct link to data:
https://drive.google.com/file/d/1B582y8_cPWxNuevpm3ZM-SZf_23HRUAQ/view

III. DATA PREPROCESSING

The data quality was already quite high to begin with, requiring only minor adjustments on our end. Additionally, we partition our 5GB dataset into a more manageable 200MB subset. This allows for rapid prototyping and testing of code functionality, without the need to utilize resources on processing the entire 5GB with each iteration of our code.

We carefully examined the data types and distribution of values for each feature to ensure suitability for modeling, which led to dropping a couple of unnecessary columns such as polygon IDs and acquisition dates that would not contribute predictive value. We also handled a small number of null values by removal rather than imputation given their rarity.

Overall, the data was in good shape for modeling with no major preprocessing needed - just some minor tweaks to optimize input features and remove unnecessary complexity. Our goal was to avoid significantly altering the underlying data distribution so that our models would more accurately reflect the relationships present within the dataset itself. By focusing our efforts on feature suitability and data integrity over extensive preprocessing, we have retained maximum fidelity to the source data.

We first tested the preprocessing, modeling, and dimensionality technique process in the smaller version dataset to make sure that it worked before later scaling it to run efficiently using the official larger dataset with the implementation of cluster mode.

IV. METHODOLOGIES

A) Big Data Handling:

Principal Component Analysis (PCA) in Big Data Handling

In big data handling, Principal Component Analysis (PCA) stands out as a powerful tool for managing high-dimensional datasets. Traditional visualization methods struggle with data that has numerous dimensions, making PCA a valuable technique for condensing data while retaining its essential features.

PCA is particularly useful in applications such as data visualization, dimensionality reduction, and image recognition. By transforming a dataset from, say, 20 dimensions to a more manageable 3 through PCA, the data becomes more interpretable without sacrificing critical information. The process involves creating new characteristics, called principal components, which capture the underlying structure of the original variables.

The first step involves employing PCA to address the challenges posed by the high dimensionality of the WildFireDB dataset. PCA will linearly transform the original variables,

creating principal components ordered by variance. This process condenses the data, allowing for a more interpretable representation while preserving critical information.

Unlike some methods that merely select features, PCA linearly transforms the original variables. The resulting principal components are ordered by the amount of variance they explain. The objective is to maximize information in each component, allowing less informative components to be discarded. This results in a more interpretable dataset that retains the most crucial information.

Unsupervised Machine Learning Techniques: K-means Clustering and Decision Trees

Unsupervised learning plays a pivotal role in extracting insights from data without predefined outcomes. Two notable techniques in this domain are K-means clustering and Decision Trees.

K-means clustering is a versatile algorithm for identifying groups or clusters within a dataset. It iteratively assigns data points to clusters based on features and outputs centroids, representing cluster centers. This technique is particularly useful when exploring patterns in unlabeled data. For example, in the context of wildfire prediction, K-means clustering can help identify spatial patterns and distinguish areas with varying levels of fire intensity.

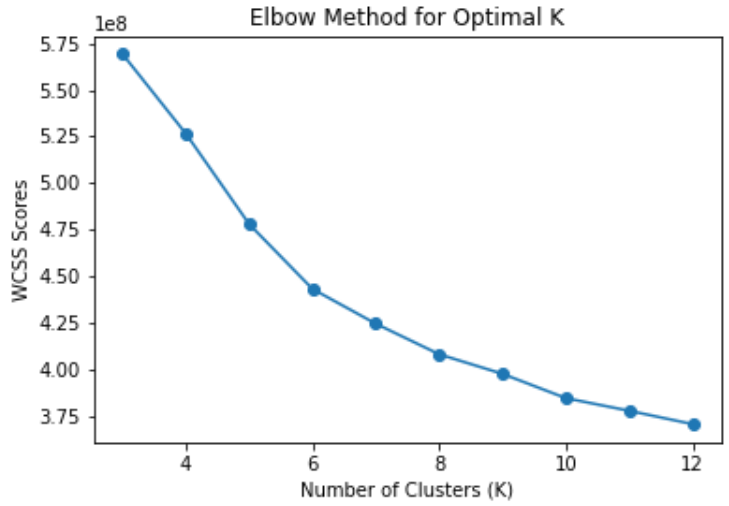


Fig. 1 Elbow Method Graph

Using the elbow method, we determined the optimal number of clusters to be 6.

With the dimensionality-reduced dataset, the focus shifts to uncovering spatial patterns and identifying clusters within the wildfire occurrences. K-means clustering was applied to categorize data points into distinct groups based on common features, revealing regions with similar fire intensity levels. The clustering results will enhance our understanding of geographical patterns in wildfire occurrences, aiding in the identification of high-risk areas.

Decision Trees aim to build predictive models by learning decision rules from training data. In this method, a tree structure is created, where internal nodes correspond to attributes and leaf nodes correspond to class labels. Decision Trees are beneficial for making predictions about target variables and are widely used for their interpretability. In the context of wildfire prediction, Decision Trees can be employed to understand the factors influencing fire occurrence and intensity, such as weather conditions, topography, and historical data.

Building on the insights gained from clustering, Decision Trees will be employed to construct predictive models for Fire Radiative Power (FRP) intensity. The tree structure will be generated, associating attributes like weather conditions, topography, and historical data with FRP levels. This facilitates the development of interpretable models. The Decision Trees will provide actionable predictions, allowing for a better understanding of the factors influencing fire occurrence and assisting in formulating targeted preventive measures.

Integration with WildFireDB Dataset

For this investigation, the WildFireDB dataset serves as a comprehensive data source. This dataset, spanning from 2012 to 2018 and comprising 17 million data points, incorporates information on historical wildfire occurrences and features extracted from satellite imagery. It includes various sources such as Landfire for topography, VIIRS for thermal anomalies, Metostat for weather data, and raw data from NOAA.

The dataset's complexity necessitates the application of dimensionality reduction techniques and unsupervised learning methods. PCA is employed to simplify the dataset, making it more manageable without compromising essential information. K-means clustering and Decision Trees are then applied to explore patterns, identify clusters, and build predictive models for Fire Radiative Power (FRP) intensity.

The integration of PCA, K-means clustering, and Decision Trees ensures a comprehensive analytical approach. PCA streamlines the dataset, K-means identifies spatial patterns, and Decision Trees build predictive models.

B) Cluster Mode: (Chinmay)

Done using 6 cores and 20 GB memory, what we changed was the number of Nodes/groups of cores running as showcased in the results below are the results of the decision tree in cluster mode

Nodes	Time	Best max_depth	Best minInstancesPerNode	Testing RMSE
1	17m 41.085s	10	6	47.596169961797739
2	22m 19.451s	10	7	47.863746126935375
3	53m 43.144s	9	7	47.868732317160806
4	58m 21.306s	10	7	47.599112762846175
5	70m 40.384s	10	7	47.765204639822353

Table 2. Final Results table

V. RESULTS DISCUSSION

We developed a bar graph in order to represent the selected principal components after performing PCA based on their importance from 0 to 0.05. These were the main selected features for the next testing process and development of models for FRP prediction

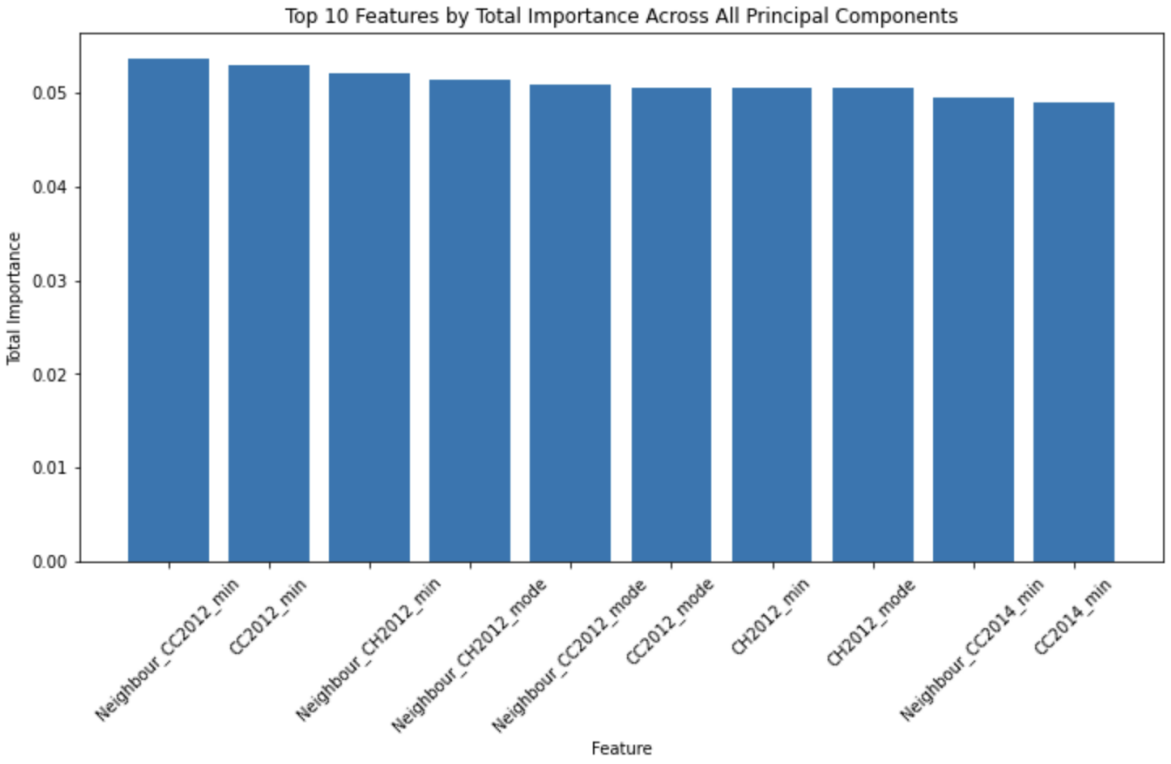


Fig. 3 PCA Bar Graph

One notable result we found was that increasing the number of worker nodes did not show a decrease in runtime across multiple experiments as well as in the final results. We believe this exists due to a potential deficiency or inefficiency that exists within our code that results in a poorly optimized synchronization between those workers' nodes when they share data with one another. This communication between the workers is what we hypothesize is leading to an increase in time in various observations while running/modeling our decision trees. Another reason we believe that could be impacting our run times was that we did not rename folders or file paths so the workers first had to overwrite old data/files on top of processing the existing code. As for the reasoning why we chose to focus on the decision tree modeling along with its hyperparameter tuning, it was that we were planning on utlizing the whole 22GBs dataset so we thought that having in the Kmeans and it's generated visuals along with the variance search would be too much. And then due to time constraints we were unable to properly shift our cluster mode process to adequately fit/match just the 5GBs of data.

1 Node : 17m and 41.085s
The best max_depth is 10 , best minInstancesPerNode = 6 , testing rmse = 47.59616996179739
2 Nodes: 22m and 19.451s
The best max_depth is 10 , best minInstancesPerNode = 7 , testing rmse = 47.863746126935375
3 Nodes: 53m and 43.144s
The best max_depth is 9 , best minInstancesPerNode = 7 , testing rmse = 47.868732317160806

4 Nodes: 58m and 21.306s
The best max_depth is 10 , best minInstancesPerNode = 7 , testing rmse = 47.599112762846175
5 Nodes: 70m and 40.384s
The best max_depth is 10 , best minInstancesPerNode = 7 , testing rmse = 47.76520463982353

The times below showcase our efforts and tests to measure the impact that the change of removing the need for the workers to overwrite pre-existing files has on the runtimes.

5 Nodes: 47m and 1.534s
The best max_depth is 10 , best minInstancesPerNode = 6 , testing rmse = 47.70474452853713
4 Nodes: 52m and 17.048s
The best max_depth is 10 , best minInstancesPerNode = 7 , testing rmse = 47.968543189861954
3 Nodes: 54m and 18.899s
The best max_depth is 10 , best minInstancesPerNode = 6 , testing rmse = 47.56668966516222
2 Nodes: 22m and 23.703s
The best max_depth is 10 , best minInstancesPerNode = 7 , testing rmse = 47.77839694759787
1 Node: 19m and 26.519s
The best max_depth is 10 , best minInstancesPerNode = 5 , testing rmse = 47.39343954313842

And as we can see, some runs did become faster, whether or not it was attributed to that cause we can't say with 100% certainty. However, we do believe that if we were to utilize the resources effectively, either including the full variance search as well as the kmeans cluster loop and its visuals with the decision tree model tuning, or simply running the decision tree code on the full 22GBs, that we would see that given more nodes/workers the runtime would decrease and we would see the results that we first expected towards the beginning of this project.
Another point of discussion was the evaluation we received from the decision trees performed on the 5 GB (in cluster mode) and 200MB (in Roar Collab) datasets which on average provided us around a 47 to 53 RMSE with the trees modeled in cluster mode closer to 47 RMSE. This showcases that the model isn't very accurate and leaves quite a bit of room for error.
But as some of the FRP, fire predictability values, go up to or even past the 100s and 200s, we believe that this type of model would at least be able to give some level of insight regarding areas that have a higher likelihood of wildfires occurring. This would allow us to combine that information with our results from the K-means clustering and take a deeper look at the features with more insight. However, due to time constraints and other hurdles in the project that is not something we have done.

VI. CONCLUSIONS & FUTURE WORK

Overall while our project fell short of our expectations in a few areas, we believe that we had quite a substantial learning experience in regards to the topics discussed in this course such as working with substantially large amounts of data. The largest road bump or pitfall that we had was moving from just 5gbs of data to the full 22 GBS of data, we ran into various memory and other issues in that shift which resulted in us sticking to just 5gbs for our PCA, Kmeans, and decision trees. As we discussed in the results we believe that those issues arose from inefficiencies in our code and we further discuss how we would combat and resolve this issue in our future work section.
Another hurdle our group faced during this project was with our expectations for runtimes during the use of the cluster mode on the decision tree models. As per discussions with the professor, the lecture on parallelism, etc. we expected our runtimes for the models to decrease

to an extent but our runtimes instead increased as we brought in more nodes for execution as mentioned in our results section our assumption is that since our code and amount of resources wasn't as fine-tuned to our work which led to the increased runtimes as we added in more workers. The logs for those results and times, as shown in the table above, are within the log files in our Git Hub (specifically in the folder labeled full_environment), each log file has a number at the beginning pertaining to the number of large nodes being used in cluster mode on the decision trees. These log files were the results of using the interactive mode within the cluster environment, we did have multiple attempts for using sbatch as well as the sh script that was provided in canvas but due to those issues and time constraints we executed them while manually changing the number of nodes (-N) to use/allocate as discussed in the results section.

One major objective in terms of future work we wanted to focus on is to be able to properly/efficiently work with the full 22gb dataset as we originally planned, due to time constraints and other issues we encountered, most of our analysis and work in this project was focused on just 5gbs of the data. Another aspect of the project that we want to improve on is the proper feature analysis through the use of lasso regression and Kmeans. Initially, we had the idea of using these algorithms to help us with a preliminary form of feature selection that we would apply during the data processing/cleaning steps. Aside from new implementations into our project method, we also would want to spend more time fine-tuning the performance of our code itself. In its current state, while our code runs fine within the ICDS environment, for the most part, it is still quite inefficient with many potential changes that could boost the execution and running times.

Another thing that we could add to this project in the future is to get it to work with an even bigger dataset in order to get more accurate data. We could also apply it to more states than just California, as this dataset only pertains to wildfires in California. Finally, we could look to use other more advanced forms of machine learning such as a random forest or a neural network.

VII. REFERENCES

1. Scikit-learn Developers. (n.d.). Decision Trees. Retrieved December 3, 2023, from https://scikit-learn.org/stable/modules/tree.html.
2. Singla, S., Diao, T., Mukhopadhyay, A., Eldawy, A., Shachter, R., & Kochenderfer, M. (2020). WildfireDB: A Spatio-Temporal Dataset ComnbiningWildfire Occurrence with Relevant Covariates, AI for Earth Sciences Workshop at Neurips 2020.

APPENDIX

```
In [1]: import pyspark
        import pandas as pd
        import numpy as np
        import math

        from pyspark import SparkContext
        from pyspark.sql import SparkSession
        from pyspark.sql.types import StructField, StructType, StringType, LongType, IntegerTy
        from pyspark.ml.feature import VectorAssembler, StandardScaler
        from pyspark.ml.feature import PCA
        import matplotlib.pyplot as plt

        from pyspark.sql.functions import col, mean, column
        import matplotlib.pyplot as plt
        from pyspark.sql.functions import expr
        from pyspark.sql.functions import split
        from pyspark.sql import Row
        #from pyspark.mllib.recommendation import ALS

        from pyspark.sql import SparkSession
        from pyspark.ml.feature import VectorAssembler, StandardScaler, MinMaxScaler
        from pyspark.ml.feature import StringIndexer, OneHotEncoder
        from pyspark.ml.clustering import KMeans
        from pyspark.ml.evaluation import ClusteringEvaluator
        import matplotlib.pyplot as plt
        import pandas as pd
        import os

        from pyspark.ml import Pipeline
        from pyspark.ml.regression import DecisionTreeRegressor
        from pyspark.ml.feature import VectorIndexer
        from pyspark.ml.evaluation import RegressionEvaluator
        from pyspark.mllib.util import MLUtils

        from decision_tree_plot.decision_tree_parser import decision_tree_parse
        from decision_tree_plot.decision_tree_plot import plot_trees
```

```
In [2]: #ss = SparkSession.builder.master("local").config("spark.driver.memory", "16g").appNa
        #ss = SparkSession.builder.config("spark.driver.memory", "5g").master("local").appName
        ss = SparkSession.builder.master("local").appName("PCAExample1").getOrCreate()
```

```
In [3]: ss.sparkContext.setCheckpointDir("/storage/home/sxs6549/work/Project/scratch")
```

```
In [4]: %%time
        df_raw = ss.read.csv("wildfiredb.csv", header=True, inferSchema=True)
        #df_raw = spark.read.csv("wildfire100.csv" , header = True, inferSchema = True)
        column_names = df_raw.columns

        #df_raw = df_raw.drop("acq_date")
        df_raw = df_raw.dropna()
```

```
CPU times: user 45.6 ms, sys: 8.82 ms, total: 54.4 ms
Wall time: 2min 31s
```

```
In [5]: #feature_columns = df_raw.columns
        col_list = list(df_raw.columns)
        feature_columns = list(set(col_list) - set(['_c0', 'Polygon_ID', 'acq_date']))

        assembler = VectorAssembler(inputCols = feature_columns, outputCol =  "features")
        assembled_data = assembler.transform(df_raw)
        scaler = StandardScaler(inputCol="features", outputCol="scaled_features", withStd=True
        scaler_model = scaler.fit(assembled_data)

        scaled_data = scaler_model.transform(assembled_data)
```

```
In [6]: # Let us perform a hyperparameter sweep
        # using 'features'
        def variance_search():
            k_values = range(13, 20)  # Example: Try K from 2 to 10


            for k_val in k_values:
                #?
                #k = 2

                pca = PCA(k=k_val, inputCol = "scaled_features", outputCol ="pcaFeatures")
                model = pca.fit(scaled_data)
                result = model.transform(scaled_data)

                explained_variance = model.explainedVariance
                print("Explained Variance: ", sum(explained_variance), "K: Value: ", k_val)
        #variance_search()
```

```
In [6]: pca = PCA(k=36, inputCol="scaled_features", outputCol="pcaFeatures")
        model = pca.fit(scaled_data)
        result = model.transform(scaled_data)
```

```
In [7]: def fit_kmeans(df_input,column_name='pcaFeatures',num_cluster_centers=3):
            '''
            Requires
            df_input: spark dataframe with column name "features" having vector of real-valued
            num_cluster_centers: integer that tells the algorithm the value of k
            column_name: column name (string) that indicates which vector of features to choos
            Returns
            cluster_data: spark dataframe with predictions
            silhouette_score: float with silhouette score
            wcss: float within cluster sum of squares
            '''
            # Create a K-Means instance
            kmeans = KMeans(featuresCol=column_name).setK(num_cluster_centers).setSeed(1)

            # Fit the K-Means model to the data
            model = kmeans.fit(df_input)

            # Get the cluster assignments for each data point
            clustered_data = model.transform(df_input)

            # Evaluate the clustering using the ClusteringEvaluator
            evaluator = ClusteringEvaluator()
            silhouette_score = evaluator.evaluate(clustered_data)

            # Print the WCSS (Within-Cluster Sum of Squares)
            print(f"Silhouette Score: {silhouette_score}")
            wcss = model.summary.trainingCost
            print(f"Within cluster sum of squares: {wcss}")

            # Get the cluster sizes and centers
            cluster_sizes = clustered_data.groupBy("prediction").count()
            cluster_sizes.show()

            # Return the clustered data
            return clustered_data, silhouette_score, wcss
```

```
In [8]: # Define a function to perform visualization
        def visualize_clusters_2D(clustered_data, feature_1, feature_2,  num_cluster_centers=
            '''
            Requires:
            clustered_data: dataframe returned from Kmeans fit
            num_cluster_centers: integer number of clusters
            feature_1: string with identifiers for first column name
            feature_2: string with identifiers for second column name
            '''

            # Convert the DataFrame to a Pandas DataFrame for visualization
            pandas_df = clustered_data.select(feature_1, feature_2, "prediction").toPandas()

            # Extract the cluster assignments
            cluster_assignments = pandas_df["prediction"]

            # Extract the indices for the cluster center
            p1 = col_list.index(feature_1)
            p2 = col_list.index(feature_2)

            # Create a scatter plot for each cluster
            for cluster_id in range(num_cluster_centers):
                # Generate a random color
                random_color = generate_random_color()
                cluster_data = pandas_df[pandas_df["prediction"] == cluster_id]
                plt.scatter(cluster_data[feature_1], cluster_data[feature_2], color=[random_co
                plt.scatter(cluster_data[feature_1].mean(),cluster_data[feature_2].mean(), col

            # Set labels and title
            plt.xlabel(feature_1)
            plt.ylabel(feature_2)
            plt.title("K-Means Clustering")
            #Save figure to folder location

            mypath = "/storage/home/sxs6549/work/Project/graphs/"
            if not os.path.isdir(mypath):
               os.makedirs(mypath)
            #fig_name = "figure_frp_" + feature_1
            path = "/storage/home/sxs6549/work/Project/graphs/figure_frp_" + feature_1
            plt.savefig(path)

            # Show the legend
            plt.legend()

            # Display the plot
            plt.show()
```

```
In [9]: import random
        def generate_random_color():
            r = random.random()
            g = random.random()
            b = random.random()
            return (r, g, b)
```

```
In [ ]:
```

```
In [10]: #def kmeans_loop():

         #    k_values = range(3, 13)  # Example: Try K from 2 to 10

         #    pca_silhouette_scores = []
         #    pca_wcss_scores = []

         #    for k in k_values:
         #        clustered_data, sil, wcss = fit_kmeans(result,column_name='pcaFeatures',num_c
         #        pca_silhouette_scores.append(sil)
         #        pca_wcss_scores.append(wcss)
         #    return pca_silhouette_scores, pca_wcss_scores
         #kmeans_loop()
```

```
Silhouette Score: -0.0020533317479589216
Within cluster sum of squares: 473731605.89298344
+----------+------+
|prediction| count|
+----------+------+
|         1|470353|
|         2|980568|
|         0|916259|
+----------+------+

Silhouette Score: -0.15963946098487028
Within cluster sum of squares: 424046470.96653455
+----------+------+
|prediction| count|
+----------+------+
|         1|230626|
|         3|919490|
|         2|977131|
|         0|239933|
+----------+------+

Silhouette Score: -0.1720033456268473
Within cluster sum of squares: 385580682.76924366
+----------+------+
|prediction| count|
+----------+------+
|         1|225594|
|         3|634558|
|         4|227494|
|         2|589534|
|         0|690000|
+----------+------+

Silhouette Score: -0.18409268517246977
Within cluster sum of squares: 351140058.148487
+----------+------+
|prediction| count|
+----------+------+
|         1|208488|
|         3|427178|
|         5|734049|
|         4|503581|
|         2|212931|
|         0|280953|
+----------+------+

Silhouette Score: -0.18862840978343925
Within cluster sum of squares: 337817574.6024609
+----------+------+
|prediction| count|
+----------+------+
|         1|598220|
|         6|355481|
|         3|209386|
|         5|265720|
|         4|372777|
|         2|205559|
|         0|360037|
+----------+------+

Silhouette Score: -0.191617109190983
Within cluster sum of squares: 330928657.1099426
+----------+------+
|prediction| count|
+----------+------+
|         1|455198|
|         6|207055|
|         3|347679|
|         5|356708|
|         4|177030|
|         7|361438|
|         2|203660|
|         0|258412|
+----------+------+

Silhouette Score: -0.3446858379460037
Within cluster sum of squares: 309872289.8296143
+----------+------+
|prediction| count|
+----------+------+
|         1|153169|
|         6|261576|
|         3| 86114|
|         5|345507|
|         4|362384|
|         8|146260|
|         7|536411|
|         2|263647|
|         0|212112|
+----------+------+

Silhouette Score: -0.31886102692248797
Within cluster sum of squares: 297124919.2819469
+----------+------+
|prediction| count|
+----------+------+
|         1|261904|
|         6|476083|
|         3|196763|
|         5| 73644|
|         9|147426|
|         4|155281|
|         8|149047|
|         7|378960|
|         2|361241|
|         0|166831|
+----------+------+

Silhouette Score: -0.3550195236080584
Within cluster sum of squares: 287772185.23022616
+----------+------+
|prediction| count|
+----------+------+
|         1| 73434|
|         6|138623|
|         3|250408|
|         5|152932|
|         9|150533|
|         4|139300|
|         8|154002|
|         7|280700|
|        10|453640|
|         2|323126|
|         0|250482|
+----------+------+

Silhouette Score: -0.34053972383375697
Within cluster sum of squares: 283587878.3289545
+----------+------+
|prediction| count|
+----------+------+
|         1| 82102|
|         6|150116|
|         3|138264|
|         5|351194|
|         9|170206|
|         4|144927|
|         8|184944|
|         7|250899|
|        10|260545|
|        11|288255|
|         2|219268|
|         0|126460|
+----------+------+
```

```
Out[10]: ([-0.0020533317479589216,
           -0.15963946098487028,
           -0.1720033456268473,
           -0.18409268517246977,
           -0.18862840978343925,
           -0.191617109190983,
           -0.3446858379460037,
           -0.31886102692248797,
           -0.3550195236080584,
           -0.34053972383375697],
          [473731605.89298344,
           424046470.96653455,
           385580682.76924366,
           351140058.148487,
           337817574.6024609,
           330928657.1099426,
           309872289.8296143,
           297124919.2819469,
           287772185.23022616,
           283587878.3289545])
```

```
In [ ]:
```

```
In [ ]:
```

```
In [10]: def kmeans_loop():
             #result.persist()
             k_values = range(3, 13)  # Example: Try K from 2 to 10

             pca_silhouette_scores = []
             pca_wcss_scores = []

             for k in k_values:
                 clustered_data, sil, wcss = fit_kmeans(result,column_name='pcaFeatures',num_cl
                 pca_silhouette_scores.append(sil)
                 pca_wcss_scores.append(wcss)
             return pca_silhouette_scores, pca_wcss_scores

         #store_km_vals = kmeans_loop()
```

```
Silhouette Score: -0.002051685603785867
Within cluster sum of squares: 473768564.47975487
+----------+------+
|prediction| count|
+----------+------+
|         1|470364|
|         2|980310|
|         0|916506|
+----------+------+
```

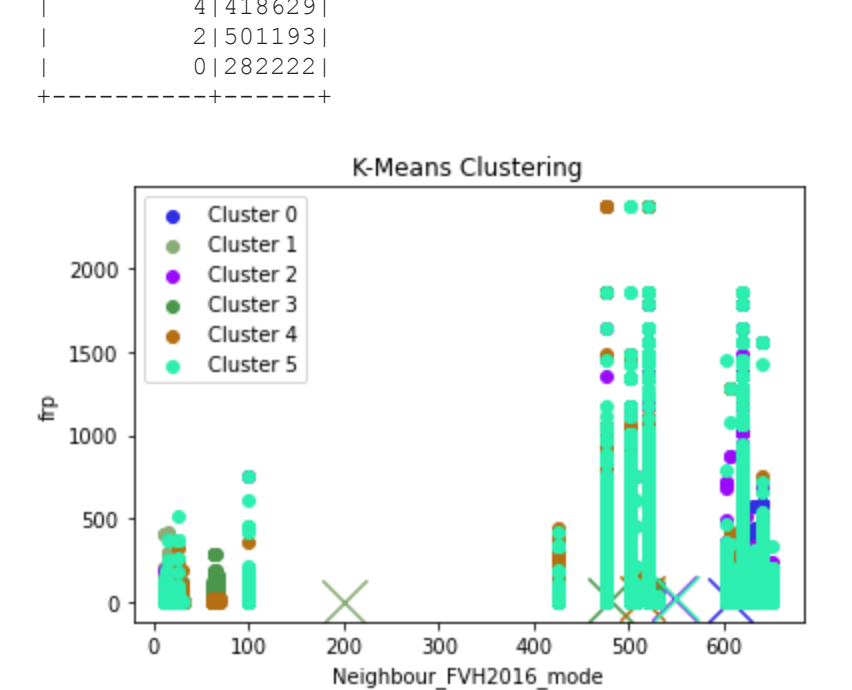
```
ERROR:root:KeyboardInterrupt while sending command.
Traceback (most recent call last):
  File "/storage/home/sxs6549/.local/lib/python3.8/site-packages/py4j/java_gateway.py",
    response = connection.send_command(command)
  File "/storage/home/sxs6549/.local/lib/python3.8/site-packages/py4j/clientserver.py",
    answer = smart_decode(self.stream.readline()[:-1])
  File "/swst/apps/anaconda3/2021.05_gcc-8.5.0/lib/python3.8/socket.py", line 669, in
    return self._sock.recv_into(b)
KeyboardInterrupt
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-10-1b30203386d8> in <module>
     12     return pca_silhouette_scores, pca_wcss_scores
     13
---> 14 store_km_vals = kmeans_loop()

<ipython-input-10-1b30203386d8> in kmeans_loop()
      7
      8     for k in k_values:
----> 9         clustered_data, sil, wcss = fit_kmeans(result,column_name="pcaFeatures
     10         pca_silhouette_scores.append(sil)
     11         pca_wcss_scores.append(wcss)

<ipython-input-7-119ff3aba002> in fit_kmeans(df_input, column_name, num_cluster_center
     14
     15     # Fit the K-Means model to the data
---> 16     model = kmeans.fit(df_input)
     17
     18     # Get the cluster assignments for each data point

~/.local/lib/python3.8/site-packages/pyspark/ml/base.py in fit(self, dataset, params)
    203                 return self.copy(params)._fit(dataset)
    204             else:
--> 205                 return self._fit(dataset)
    206         else:
    207             raise TypeError(

~/.local/lib/python3.8/site-packages/pyspark/ml/wrapper.py in _fit(self, dataset)
    379
    380     def _fit(self, dataset: DataFrame) -> JM:
--> 381         java_model = self._fit_java(dataset)
    382         model = self._create_model(java_model)
    383         return self._copyValues(model)

~/.local/lib/python3.8/site-packages/pyspark/ml/wrapper.py in _fit_java(self, dataset)
    376
    377         self._transfer_params_to_java()
--> 378         return self._java_obj.fit(dataset._jdf)
    379
    380     def _fit(self, dataset: DataFrame) -> JM:

~/.local/lib/python3.8/site-packages/py4j/java_gateway.py in __call__(self, *args)
   1319             proto.END_COMMAND_PART
   1320
-> 1321         answer = self.gateway_client.send_command(command)
   1322         return_value = get_return_value(
   1323             answer, self.gateway_client, self.target_id, self.name)

~/.local/lib/python3.8/site-packages/py4j/java_gateway.py in send_command(self, comman
   1036         connection = self._get_connection()
   1037         try:
-> 1038             response = connection.send_command(command)
   1039             if binary:
   1040                 return response, self._create_connection_guard(connection)

~/.local/lib/python3.8/site-packages/py4j/clientserver.py in send_command(self, comman
    509         try:
    510             while True:
--> 511                 answer = smart_decode(self.stream.readline()[:-1])
    512                 logger.debug("Answer received: {0}".format(answer))
    513                 # Happens when a the other end is dead. There might be an empt

/swst/apps/anaconda3/2021.05_gcc-8.5.0/lib/python3.8/socket.py in readinto(self, b)
    667         while True:
    668             try:
--> 669                 return self._sock.recv_into(b)
    670             except timeout:
    671                 self._timeout_occurred = True

KeyboardInterrupt:
```

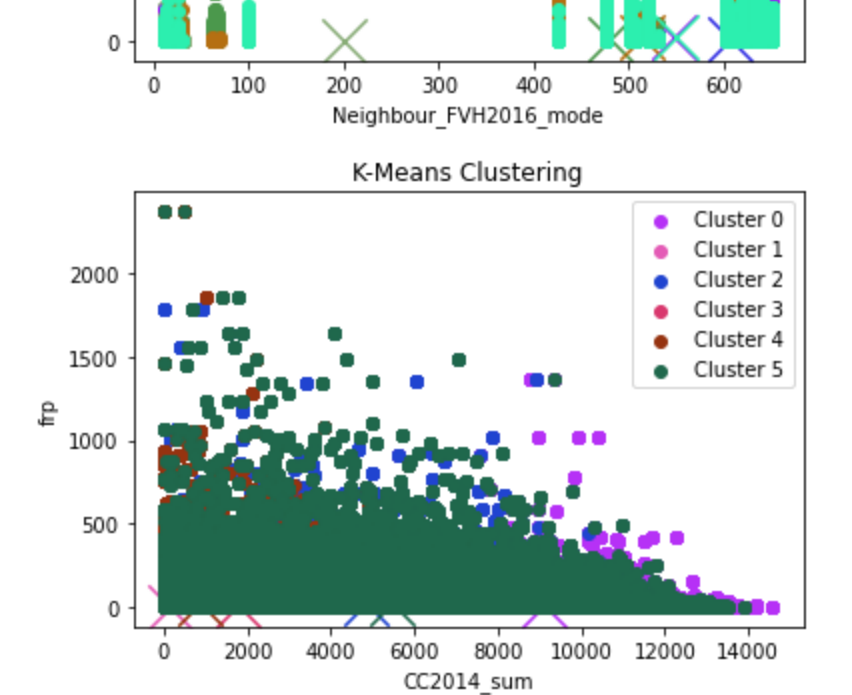
```
In [ ]: wcss_scores = store_km_vals[1]
```

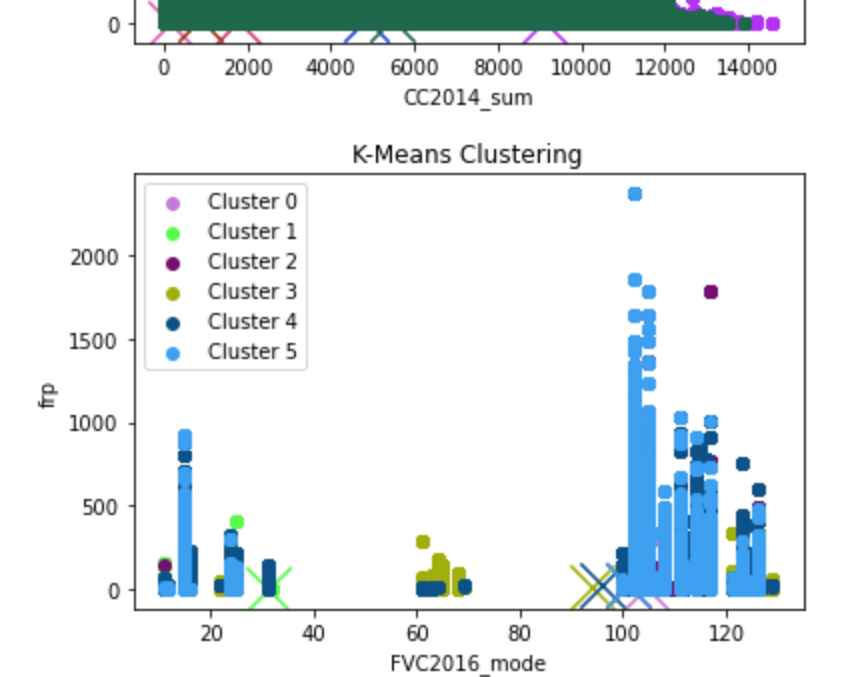
```
In [ ]: # Plot the Elbow Method curve
        k_values = range(3, 13)
        plt.plot(k_values, wcss_scores, marker='o')
        plt.title("Elbow Method for Optimal K")
        plt.xlabel("Number of Clusters (K)")
        plt.ylabel("WCSS Scores")
        plt.savefig('elbow_chart')
        plt.show()
```

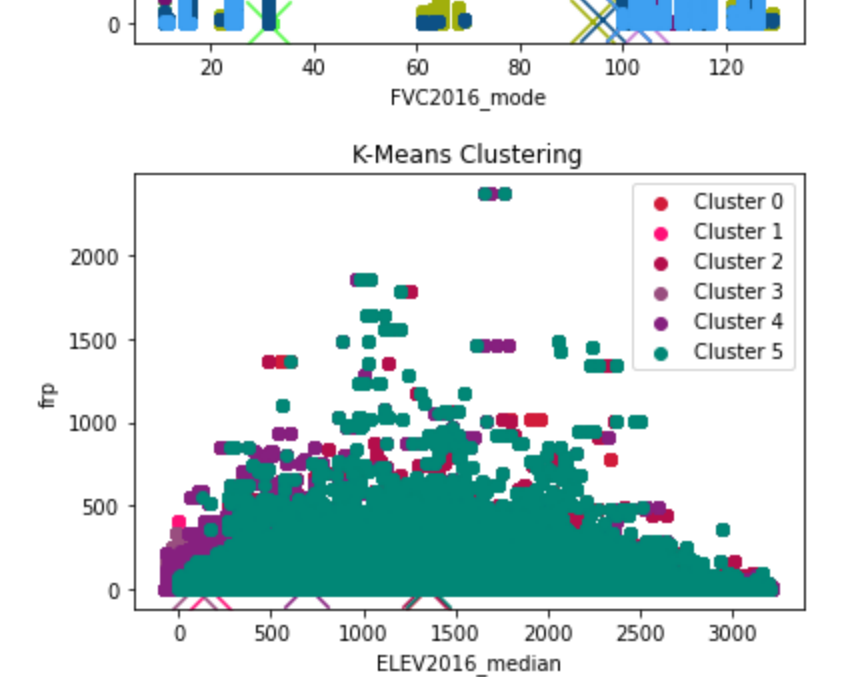
```
In [ ]: # Let us perform a hyperparameter sweep
        # using 'features'
        def kmeans_vis_loop():

            #k_values = range(3, 13)  # Example: Try K from 2 to 10
            k_value = 6
            pca_silhouette_scores = []
            pca_wcss_scores = []

            #for k in k_values:
            clustered_data, sil, wcss = fit_kmeans(result,column_name='pcaFeatures',num_cluste


            for feature in feature_columns:
                #?
                #k = 2
                # Convert the DataFrame to a Pandas DataFrame for visualization
                feature_1 = feature
                feature_2 = "frp"
                #print("X")
                visualize_clusters_2D(clustered_data, feature_1, feature_2, k_value)
        kmeans_vis_loop()
```

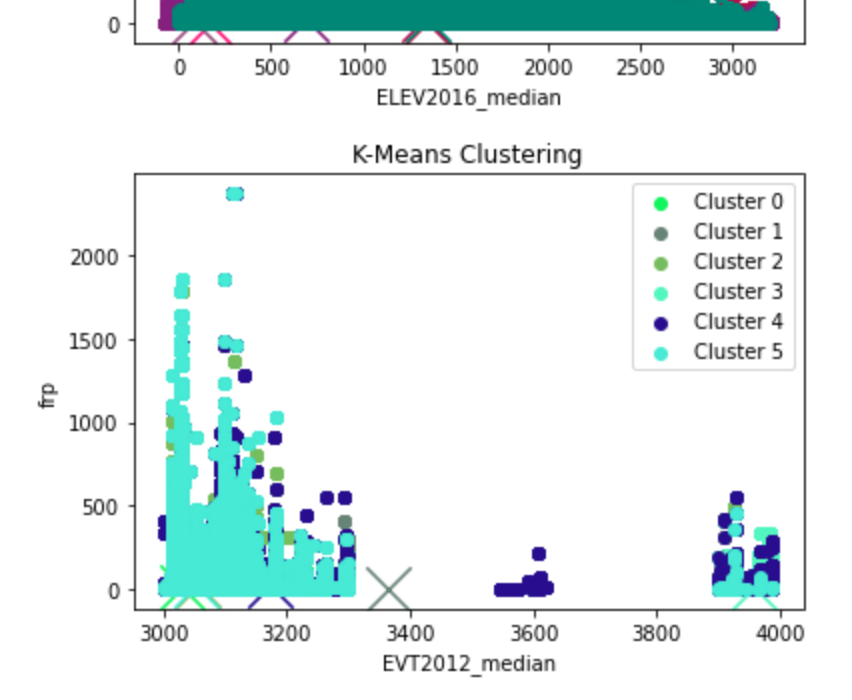
```
Silhouette Score: -0.18517583088434597
Within cluster sum of squares: 443100587.72990435
+----------+------+
|prediction| count|
+----------+------+
|         1|207611|
|         3|212671|
|         5|744854|
|         4|418629|
|         2|501193|
|         0|282222|
+----------+------+
```

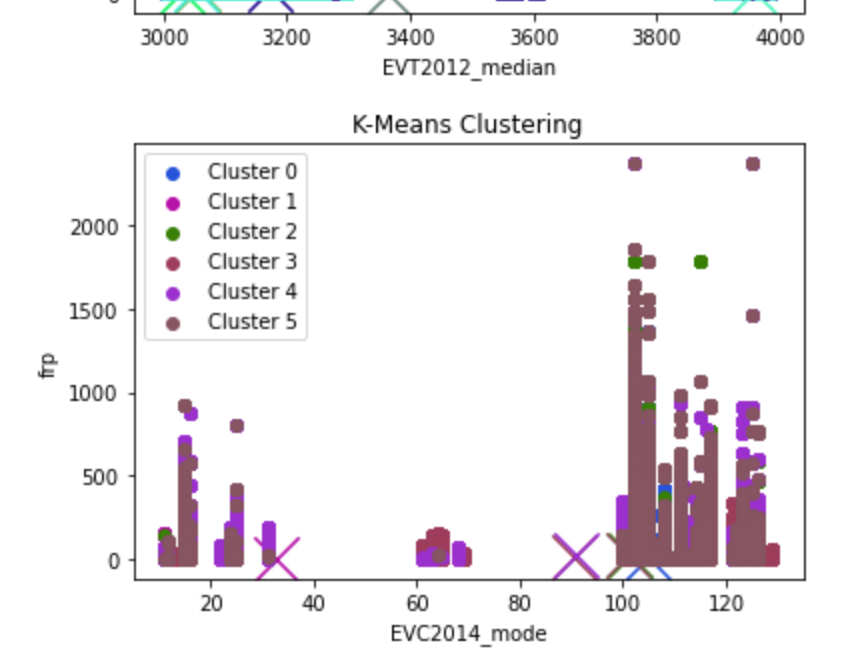


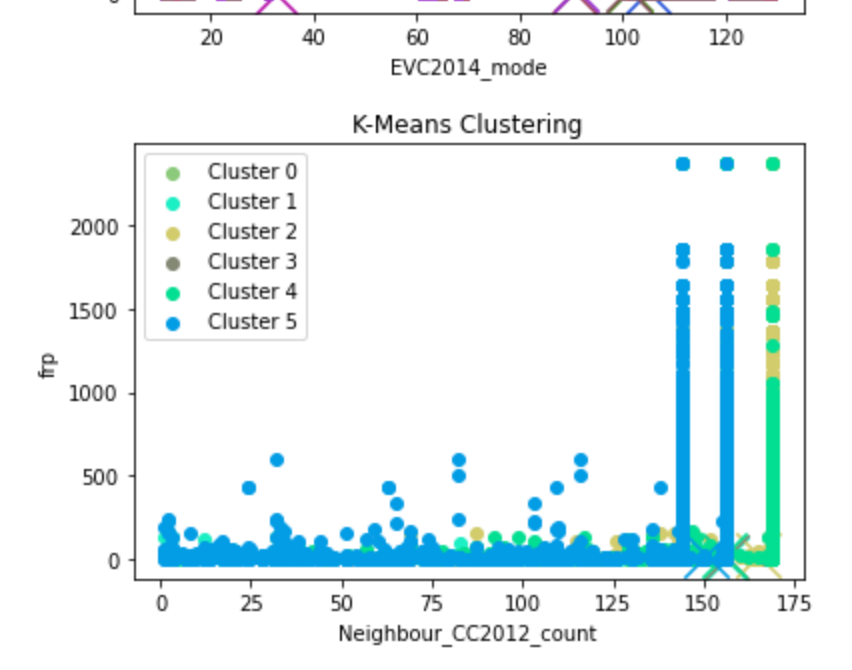


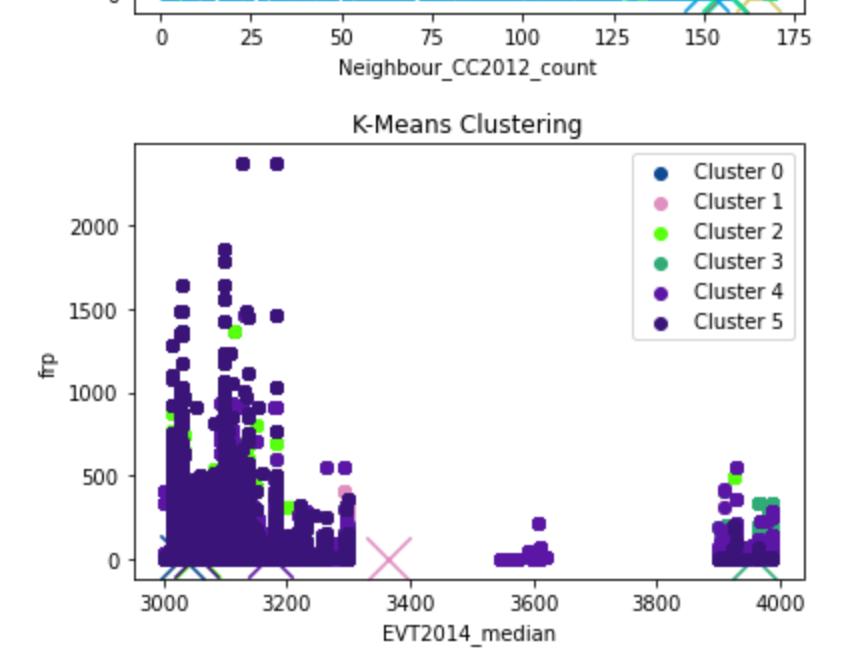


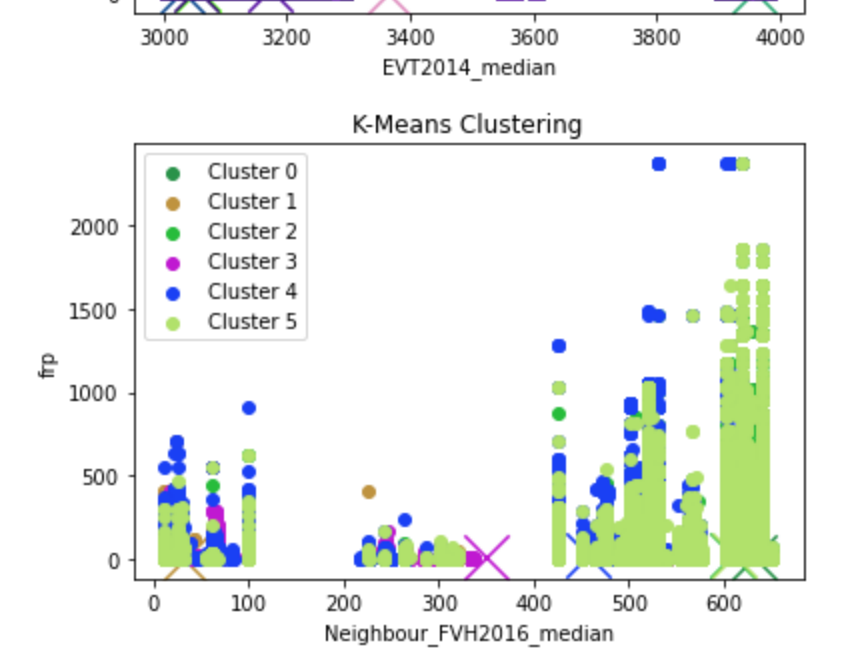


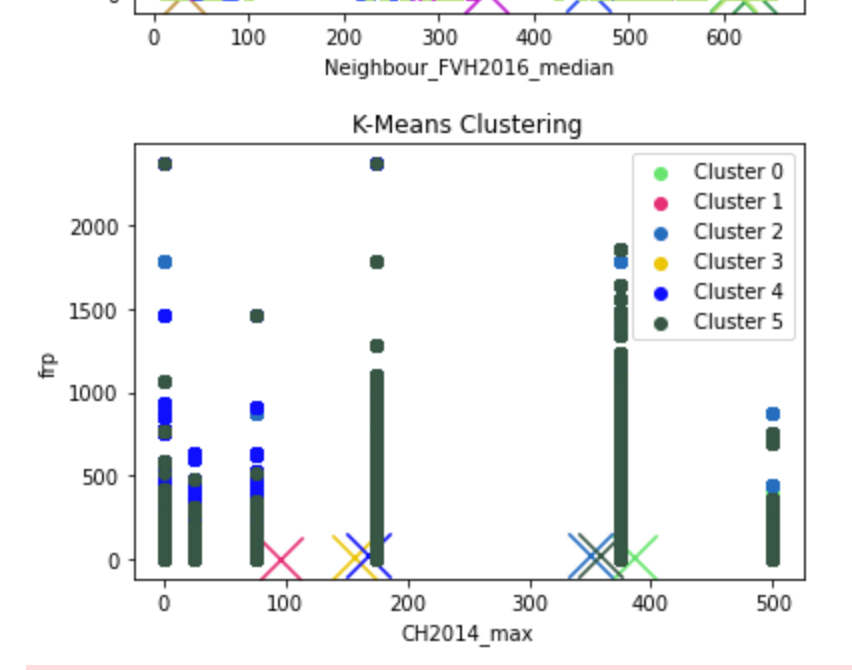




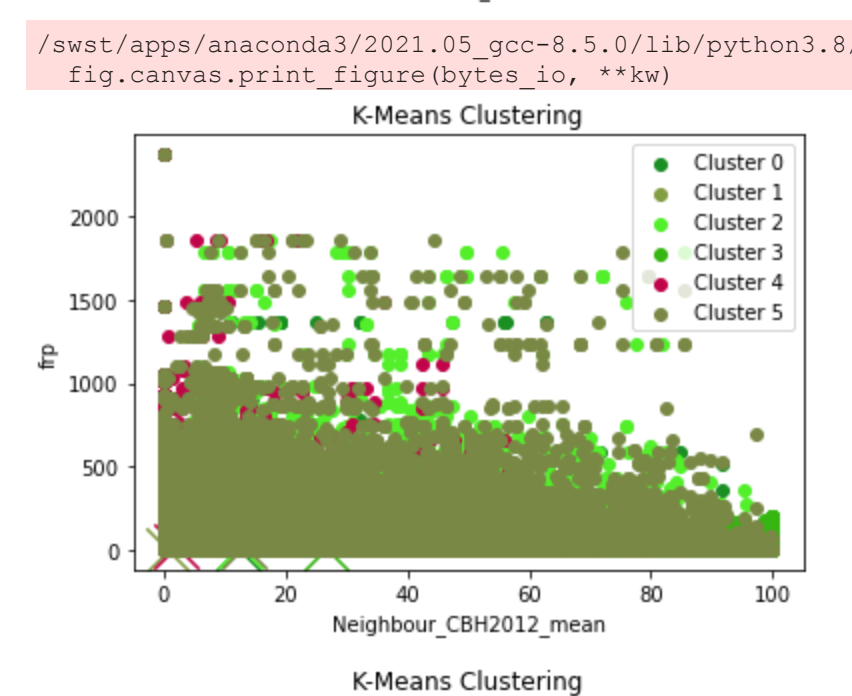


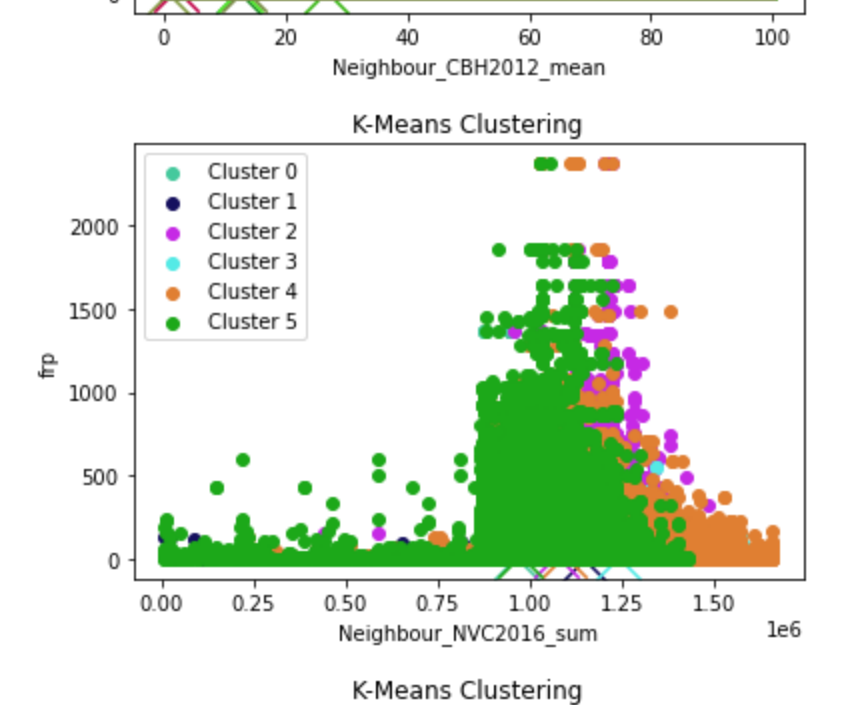


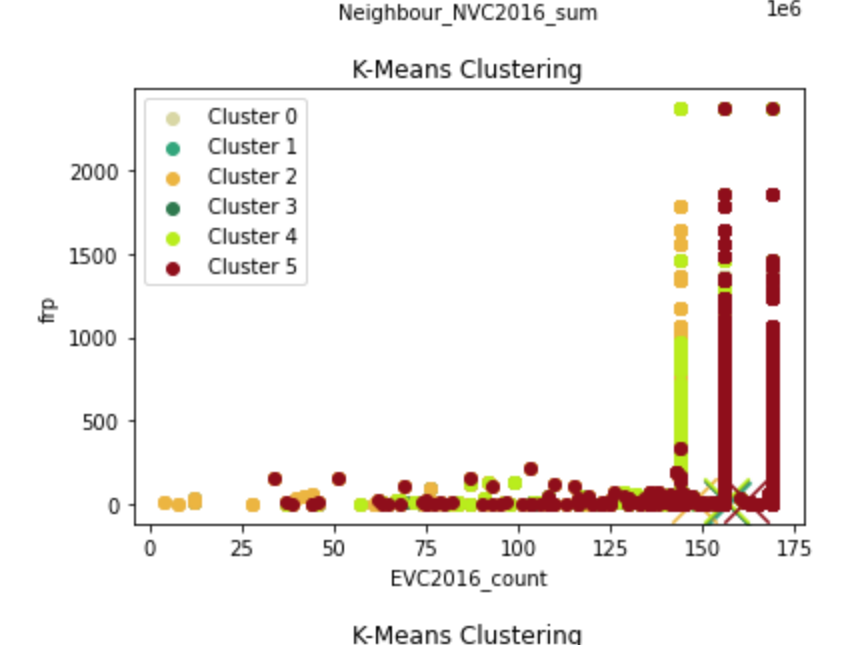


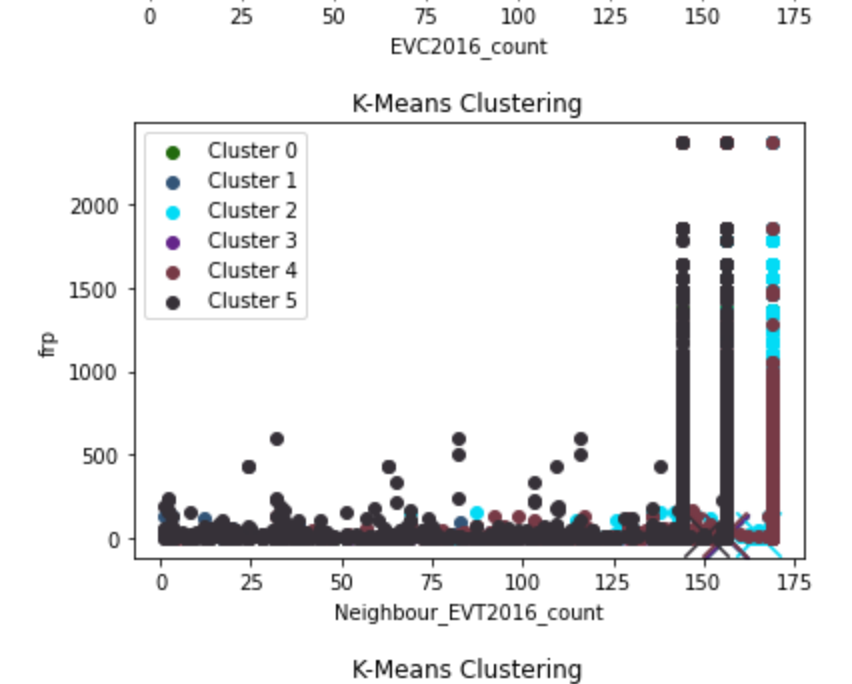


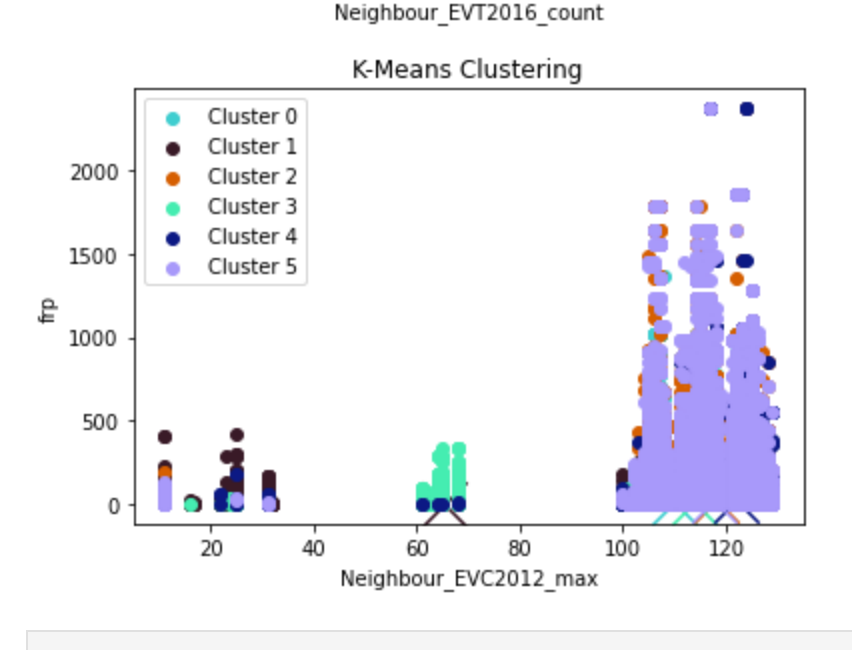

```
/swst/apps/anaconda3/2021.05_gcc-8.5.0/lib/python3.8/site-packages/IPython/core/pylabt
  fig.canvas.print_figure(bytes_io, **kw)
```












```
In [ ]:
```

```
In [ ]: #feature_columns = df_raw.columns
        #col_list = list(df_raw.columns)
        #col_list = list(df_raw_trial.columns)
        feature_inputs = list(set(col_list) - set(['_c0', 'Polygon_ID', 'acq_date', 'frp']))

        assembler_tree = VectorAssembler(inputCols = feature_inputs, outputCol =  "features")
        #assembled_data_tree = assembler_tree.transform(df_raw_trial)
        assembled_data_tree = assembler_tree.transform(df_raw)
        #scaler = StandardScaler(inputCol="features", outputCol="scaled_features", withStd=Tr
        #scaler_model = scaler.fit(assembled_data)

        #scaled_data = scaler_model.transform(assembled_data)
```

```
In [ ]: def tree_hyperparam_loop(data, model_path):

            (trainingData, testingData) = data.randomSplit([0.8, 0.2], seed=1237)


            ## Initialize a Pandas DataFrame to store evaluation results of all combination o
            hyperparams_eval_df = pd.DataFrame( columns = ['max_depth', 'minInstancesPerNode',
            # initialize index to the hyperparam_eval_df to 0
            index =0
            # initialize lowest_error
            lowest_testing_rmse = 100000
            # Set up the possible hyperparameter values to be evaluated
            max_depth_list = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
            minInstancesPerNode_list = [2, 3, 4, 5, 6, 7]
            #max_depth_list = [2]
            #minInstancesPerNode_list = [9]
            #labelIndexer = StringIndexer(inputCol="class", outputCol="indexedLabel").fit(data
            #feature_inputs = list(set(col_list) - set(['_c0', 'Polygon_ID', 'acq_date', 'frp
            #assembler = VectorAssembler( inputCols=feature_inputs, outputCol="features")
            #labelConverter = IndexToString(inputCol = "prediction", outputCol="predictedClas
            #model_path="/storage/home/sxs6549/work/Project/DTmodel_vis"
            trainingData.persist()
            testingData.persist()


            for max_depth in max_depth_list:
                for minInsPN in minInstancesPerNode_list:
                    trainingData.persist()
                    testingData.persist()

                    seed = 37
                    # Construct a DT model using a set of hyper-parameter values and training
                    #dt= DecisionTreeClassifier(labelCol="indexedLabel", featuresCol="feature
                    dt = DecisionTreeRegressor(labelCol="frp", featuresCol="pcaFeatures", maxD
                    #pipeline = Pipeline(stages=[labelIndexer, assembler, dt, predictionConve
                    model = dt.fit(trainingData)
                    training_predictions = model.transform(trainingData)
                    testing_predictions = model.transform(testingData)
                    #evaluator = MulticlassClassificationEvaluator(labelCol="indexedLabel", p
                    evaluator = RegressionEvaluator(labelCol="frp", predictionCol="prediction"
                    training_rmse = evaluator.evaluate(training_predictions)
                    testing_rmse = evaluator.evaluate(testing_predictions)
                    # We use 0 as default value of the 'Best Model' column in the Pandas Data
                    # The best model will have a value 1000
                    hyperparams_eval_df.loc[index] = [ max_depth, minInsPN, training_rmse, tes
                    index = index +1
                    if testing_rmse < lowest_testing_rmse :
                        best_max_depth = max_depth
                        best_minInsPN = minInsPN
                        best_index = index -1
                        best_parameters_training_rmse = training_rmse
                        best_DTmodel= model
                        best_tree = decision_tree_parse(best_DTmodel, ss, model_path)
                        column = dict( [ (str(idx), i) for idx, i in enumerate(feature_inputs)])
                        lowest_testing_rmse = testing_rmse
            print('The best max_depth is ', best_max_depth, ', best minInstancesPerNode = ', '
                  best_minInsPN, ', testing rmse = ', lowest_testing_rmse)
            column = dict([(str(idx), i) for idx, i in enumerate(feature_inputs)])


            best_model_path_part7="/storage/home/sxs6549/work/Project/DT_HPT_cluster"
            best_tree=decision_tree_parse(best_DTmodel, ss, best_model_path_part7)
            column = dict([(str(idx), i) for idx, i in enumerate(feature_inputs)])
            plot_trees(best_tree, column = column, output_path = '/storage/home/sxs6549/work/P
            hyperparams_eval_df.loc[best_index]=[best_max_depth, best_minInsPN, best_paramete
            output_path = "/storage/home/sxs6549/work/Project/HPT_cluster.csv"
            hyperparams_eval_df.to_csv(output_path)

        tree_hyperparam_loop(result_tree, model_path="/storage/home/sxs6549/work/Project/DTmo
```

```
In [ ]: import pyspark
        import pandas as pd
        import numpy as np
        import math

        from pyspark import SparkContext
        from pyspark.sql import SparkSession
        from pyspark.sql.types import StructField, StructType, StringType, LongType, IntegerTy
        from pyspark.ml.feature import VectorAssembler, StandardScaler
        from pyspark.ml.feature import PCA
        import matplotlib.pyplot as plt

        from pyspark.sql.functions import col, mean, column
        import matplotlib.pyplot as plt
        from pyspark.sql.functions import expr
        from pyspark.sql.functions import split
        from pyspark.sql import Row
        #from pyspark.mllib.recommendation import ALS

        from pyspark.sql import SparkSession
        from pyspark.ml.feature import VectorAssembler, StandardScaler, MinMaxScaler
        from pyspark.ml.feature import StringIndexer, OneHotEncoder
        from pyspark.ml.clustering import KMeans
        from pyspark.ml.evaluation import ClusteringEvaluator
        import matplotlib.pyplot as plt
        import pandas as pd

        from pyspark.ml import Pipeline
        from pyspark.ml.regression import DecisionTreeRegressor
        from pyspark.ml.feature import VectorIndexer
        from pyspark.ml.evaluation import RegressionEvaluator
        from pyspark.mllib.util import MLUtils

        from decision_tree_plot.decision_tree_parser import decision_tree_parse
        from decision_tree_plot.decision_tree_plot import plot_trees
```

```
In [ ]: #The code from here below is what was used within cluster mode to find the best hyperp
        #Right now the outputs only show us testing the code on the small dataset, but in clu
```

In [1]:
```python
from IPython.core.display import HTML
display(HTML("<style>pre { white-space: pre !important; }</style>"))
```

In [3]:
```python
ss = SparkSession.builder.config("spark.driver.memory", "16g").appName("ProjectTree1")
#ss = SparkSession.builder.config("spark.driver.memory", "5g").master("local").appName
#ss = SparkSession.builder.master("local").appName("PCAExample1").getOrCreate()
```

In [4]:
```python
ss.sparkContext.setCheckpointDir("/storage/home/sxs6549/work/Project/scratch")
```

In [5]:
```python
#%%time
df_raw = ss.read.csv("wildfiredb.csv", header=True, inferSchema=True)
#df_raw = spark.read.csv("wildfire100.csv" , header = True, inferSchema = True)
#column_names = df_raw.columns

#df_raw = df_raw.drop("acq_date")
df_raw = df_raw.dropna()
```

```
CPU times: user 43.9 ms, sys: 12.3 ms, total: 56.2 ms
Wall time: 2min 40s
```

In [4]:
```python
#%%time
#df_raw_trial = ss.read.csv("fire_small.csv", header=True, inferSchema=True)
#df_raw = spark.read.csv("wildfire100.csv" , header = True, inferSchema = True)
#column_names = df_raw_trial.columns

#df_raw = df_raw.drop("acq_date")
#df_raw_trial = df_raw_trial.dropna()
```

```
CPU times: user 25.6 ms, sys: 3.78 ms, total: 29.4 ms
Wall time: 11.5 s
```

In [4]:
```python
#col_list = list(df_raw.columns)
#col_list
```

In [5]:
```python
#col_list_new = list(set(col_list) - set(['_c0', 'Polygon_ID', 'acq_date', 'frp']))
#col_list_new
```

In [6]:
```python
#feature_columns = df_raw.columns
#col_list = list(df_raw.columns)
col_list = list(df_raw.columns)
feature_inputs = list(set(col_list) - set(['_c0', 'Polygon_ID', 'acq_date', 'frp']))

assembler_tree = VectorAssembler(inputCols = feature_inputs, outputCol =  "features")
assembled_data_tree = assembler_tree.transform(df_raw)
#scaler = StandardScaler(inputCol="features", outputCol="scaled_features", withStd=Tr
#scaler_model = scaler.fit(assembled_data)

#scaled_data = scaler_model.transform(assembled_data)
```

In [7]:
```python
pca_tree = PCA(k=36, inputCol="features", outputCol="pcaFeatures")
model_tree = pca_tree.fit(assembled_data_tree)
result_tree = model_tree.transform(assembled_data_tree)
```

In [19]:
```python
from pyspark.ml import Pipeline
from pyspark.ml.regression import DecisionTreeRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.mllib.util import MLUtils

from decision_tree_plot.decision_tree_parser import decision_tree_parse
from decision_tree_plot.decision_tree_plot import plot_trees
# Split the data into training and test sets (20% held out for testing)
(trainingData, testingData) = result_tree.randomSplit([0.8, 0.2], seed=1237)

#Code cell for Part 7
## Initialize a Pandas DataFrame to store evaluation results of all combination of hy
hyperparams_eval_df = pd.DataFrame( columns = ['max_depth', 'minInstancesPerNode', 't
# initialize index to the hyperparam_eval_df to 0
index =0
# initialize lowest_error
lowest_testing_rmse = 100000
# Set up the possible hyperparameter values to be evaluated
max_depth_list = [2, 3, 4, 5, 6, 7, 8, 9, 10]
minInstancesPerNode_list = [2, 3, 4, 5, 6, 7]
#max_depth_list = [2]
#minInstancesPerNode_list = [9]
#labelIndexer = StringIndexer(inputCol="class", outputCol="indexedLabel").fit(data2)
#feature_inputs = list(set(col_list) - set(['_c0', 'Polygon_ID', 'acq_date', 'frp']))
#assembler = VectorAssembler( inputCols=feature_inputs, outputCol="features")
#labelConverter = IndexToString(inputCol = "prediction", outputCol="predictedClass", 
model_path="/storage/home/sxs6549/work/Project/fire_DTmodel_vis"
```

Out[19]:
```
DataFrame[_c0: int, Polygon_ID: int, acq_date: date, frp: double, Neighbour: int, Neig
```

In [20]:
```python
#%%time
for max_depth in max_depth_list:
    for minInsPN in minInstancesPerNode_list:
        trainingData.persist()
        testingData.persist()

        seed = 37
        # Construct a DT model using a set of hyper-parameter values and training data
        #dt= DecisionTreeClassifier(labelCol="indexedLabel", featuresCol="features", 
        dt = DecisionTreeRegressor(labelCol="frp", featuresCol="pcaFeatures", maxDept
        #pipeline = Pipeline(stages=[labelIndexer, assembler, dt, predictionConverter
        model = dt.fit(trainingData)
        training_predictions = model.transform(trainingData)
        testing_predictions = model.transform(testingData)
        #evaluator = MulticlassClassificationEvaluator(labelCol="indexedLabel", predi
        evaluator = RegressionEvaluator(labelCol="frp", predictionCol="prediction", me
        training_rmse = evaluator.evaluate(training_predictions)
        testing_rmse = evaluator.evaluate(testing_predictions)
        # We use 0 as default value of the 'Best Model' column in the Pandas DataFram
        # The best model will have a value 1000
        hyperparams_eval_df.loc[index] = [ max_depth, minInsPN, training_rmse, testin
        index = index +1
        if testing_rmse < lowest_testing_rmse :
            best_max_depth = max_depth
            best_minInsPN = minInsPN
            best_index = index -1
            best_parameters_training_rmse = training_rmse
            best_DTmodel= model
            best_tree = decision_tree_parse(best_DTmodel, ss, model_path)
            column = dict( [ (str(idx), i) for idx, i in enumerate(feature_inputs) ])
            lowest_testing_rmse = testing_rmse
print('The best max_depth is ', best_max_depth, ', best minInstancesPerNode = ', \
      best_minInsPN, ', testing rmse = ', lowest_testing_rmse)
column = dict([(str(idx), i) for idx, i in enumerate(feature_inputs)])
```

```
The best max_depth is  8 , best minInstancesPerNode =  4 , testing rmse =  51.34257362
CPU times: user 1.41 s, sys: 313 ms, total: 1.72 s
Wall time: 4min 12s
```

In [28]:
```python
#training_predictions.show(100)
```

In [15]:
```python
#Code cell for Part 7
best_model_path_part7="/storage/home/sxs6549/work/Project/fire_DT_HPT_cluster"
```

In [16]:
```python
#Code cell for Part 7
best_tree=decision_tree_parse(best_DTmodel, ss, best_model_path_part7)
column = dict([(str(idx), i) for idx, i in enumerate(feature_inputs)])
plot_trees(best_tree, column = column, output_path = '/storage/home/sxs6549/work/Proj
```

In [17]:
```python
#Code cell for Part 7
# Store the Testing RMS in the DataFrame
hyperparams_eval_df.loc[best_index]=[best_max_depth, best_minInsPN, best_parameters_t
```

In [18]:
```python
output_path = "/storage/home/sxs6549/work/Project/fire_HPT_cluster.csv"
hyperparams_eval_df.to_csv(output_path)
```

In [ ]:

In [ ]:

In [1]:
```python
import pyspark
import pandas as pd
import numpy as np
import math

from pyspark import SparkContext
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, StringType, LongType, IntegerT
from pyspark.ml.feature import VectorAssembler, StandardScaler
from pyspark.ml.feature import PCA
import matplotlib.pyplot as plt

from pyspark.sql.functions import col, mean, column
import matplotlib.pyplot as plt
from pyspark.sql.functions import expr
from pyspark.sql.functions import split
from pyspark.sql import Row
from pyspark.mllib.recommendation import ALS

from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler, StandardScaler, MinMaxScaler
from pyspark.ml.feature import StringIndexer, OneHotEncoder
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator
import matplotlib.pyplot as plt
import pandas as pd
```

In [ ]:
```python
#The code from here and below represents the work for the PCA visualization and other
```

In [2]:
```python
spark = SparkSession.builder.appName("PCAExample1").getOrCreate()
```

In [3]:
```python
df_raw = spark.read.csv("wildfire100.csv" , header = True, inferSchema = True)
column_names = df_raw.columns

df_raw = df_raw.drop("acq_date")
df_raw = df_raw.dropna()
```

In [4]:
```python
feature_columns = [col for col in df_raw.columns if col != 'frp']

assembler = VectorAssembler(inputCols = feature_columns, outputCol =  "features")
assembled_data = assembler.transform(df_raw)
scaler = StandardScaler(inputCol="features", outputCol="scaled_features", withStd=True
scaler_model = scaler.fit(assembled_data)

scaled_data = scaler_model.transform(assembled_data)
```

In [5]:
```python
pca = PCA(k=36, inputCol="scaled_features", outputCol="pcaFeatues")
model = pca.fit(scaled_data)
result = model.transform(scaled_data)
```

In [6]:
```python
explained_variance = model.explainedVariance
print ("Explained Variance:", sum(explained_variance))
```

```
Explained Variance: 0.7321926605768483
```

In [15]:
```python
# Extract PCA loadings
# Assuming pca_model is the name of your PCA model instance
loadings = model.pc.toArray()

# For simplicity, let's analyze the loadings for the first principal component

pc1_loadings = loadings[:, 0]

# Pair the loadings with feature names and sort
feature_importance = sorted(list(zip(feature_columns, pc1_loadings)), key=lambda x: x

#for feature, loading in feature_importance:
 # print(f"{feature}: {loading}")
```

In [20]:
```python
# Original Feature Variance impact

total_importance = np.sum(loadings**2, axis=1)

# Pair the aggregated importance with feature names
feature_importance = list(zip(feature_columns, total_importance))

# Sort the features by their importance
sorted_features = sorted(feature_importance, key=lambda x: x[1], reverse=True)

# Select the top 10 features
top_10_features = sorted_features[:10]

# Unzip the feature names and their corresponding importances
features, importances = zip(*top_10_features)

# Create a bar graph
plt.figure(figsize=(12, 6))
plt.bar(features, importances)
plt.xlabel('Feature')
plt.ylabel('Total Importance')
plt.title('Top 10 Features by Total Importance Across All Principal Components')
plt.xticks(rotation=45)
plt.show()
```

In [16]:
```python
num_cluster_centers = 10

kmean = KMeans(featuresCol= "scaled_features").setK(num_cluster_centers).setSeed(1)

model= kmean.fit(scaled_data)

cluster_centers = model.clusterCenters()
clustered_data = model.transform(scaled_data)

evaluator = ClusteringEvaluator()
silhouetter_score = evaluator.evaluate(clustered_data)

wcss = model.summary.trainingCost

cluster_sizes = clustered_data.groupBy("prediction").count()
cluster_sizes.show()
```

```
+----------+-----+
|prediction|count|
+----------+-----+
|         1| 2265|
|         6| 6183|
|         3| 1288|
|         5| 4116|
|         9| 4276|
|         4| 6900|
|         8| 2172|
|         7| 3125|
|         2| 2550|
|         0| 6561|
+----------+-----+
```

In [17]:
```python
cluster_profiles = clustered_data.groupBy('prediction').mean().collect()
#
#for row in cluster_profiles:
#    print(f"Cluster {row['prediction']}:")
#    for feature in feature_columns:
#        print(f"{feature}: {row['avg(' + feature + ')']}")
 #   print("\n")
```

In [50]:
```python
# First, apply KMeans clustering on the PCA results
kmean = KMeans(featuresCol="pcaFeatues", k=num_cluster_centers).setSeed(1)
kmeans_model = kmean.fit(result)
clustered_data = kmeans_model.transform(result)

# Extract PCA values and predictions
pca_values = clustered_data.select("pcaFeatues", "prediction").collect()
x = [row['pcaFeatues'][0] for row in pca_values]
y = [row['pcaFeatues'][1] for row in pca_values]
colors = [row['prediction'] for row in pca_values]

# Plotting
plt.scatter(x, y, c=colors, cmap='tab10')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar()
plt.show()
```