main

December 10, 2024

Import all required imports and define some common types.

[1076]:
```python
from typing import Union, Dict, Any, Tuple, Dict, List, Callable
import matplotlib.pyplot as plt
import numpy as np
import geopandas as gpd
import requests
import random
import folium
from shapely.geometry import Point, Polygon, MultiPolygon
from pathlib import Path
from folium.plugins import HeatMap
from branca.colormap import LinearColormap
from branca.colormap import linear
import seaborn as sns
import pandas as pd
import sklearn.metrics
import statsmodels.api as sm
from sklearn import linear_model
from sklearn.model_selection import cross_val_score
from itertools import combinations
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
import matplotlib.cm as cm
import matplotlib.colors
import warnings
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
from rich.console import Console
from rich.panel import Panel
from rich.columns import Columns


BoundingBox = Tuple[float, float, float, float]
```

```
# Suppress warnings
warnings.filterwarnings("ignore")
```

1 Constants & Hyperparameters

[1077]:
```
class Config:
    _config_data: Dict[str, Any] = {
        # hyperparameters
        "hp": {
            "test_size": 0.2,  # the ratio of the dataset to use for testing
            "subsection_count": 40,  # Number of subsections to divide the city␣
 ↪into
            "subsection_count_model2": 22,  # Number of subsections to divide␣
 ↪the city into
            "dataset_bounds_reference": "sidewalks",
            "dataset_bounds_reference_model2": "education",
            "cross_val_folds": {
                "linear": 5,
                "best_subset": 5,
                "lasso": 5,
                "ridge": 5,
                "svm": 3
            },
            "scoring_metric": {  # the scoring metric to use for each model␣
 ↪during cross-validation
                "best_subset": "r2",
                "lasso": "r2",
                "ridge": "r2",
                "svm" : "accuracy"
            },
            "distance_to_nearest_infrastructure": [  # the sparse␣
 ↪infrastructure types to calculate distance-to for
                "fire_stations",
                "landfills",
                "streetcar_stops",
                "streetcar_routes",
            ],
            "count_instances_infrastructure": [
                "bicycle_boulevards",
                "bridges",
                "crosswalks",
                "major_streets",
                "sidewalks",
                "streetlights",
                "suntran_stops",
            ],
            "geometry_contains_method": {  # the method to use to determine␣
↪whether an infrastructure type is considered "nearby"
                "arrests": "within",
                "bicycle_boulevards": "intersects",
                "bridges": "within",
                "crosswalks": "intersects",
                "fire_stations": "within",
                "landfills": "within",
                "major_streets": "intersects",
                "streetcar_routes": "intersects",
                "streetcar_stops": "within",
                "scenic_routes": "intersects",
                "sidewalks": "intersects",
                "streetlights": "within",
                "suntran_stops": "within",
            },
        },
        "feature_names": {
            "model1": [
                "bicycle_boulevards",
                "bridges",
                "crosswalks",
                "fire_stations",
                "landfills",
                "major_streets",
                "streetcar_routes",
                "streetcar_stops",
                "scenic_routes",
                "sidewalks",
                "streetlights",
                "suntran_stops",
            ],
        },
        "tucson_center_coordinates": [32.22174, -110.92648],
        "tucson_bounds": [-111, 31.042601, -110.7347, 32.2226],
        "data_sources": {
            "arrests": "crime/Tucson_Police_Arrests_-_2017_-_Open_Data.geojson",
            "bicycle_boulevards": "infrastructure/Bicycle_Boulevards.geojson",
            "bridges": "infrastructure/Bridges_-_Open_Data.geojson",
            "business_licenses": "infrastructure/Business_Licenses_(Open_Data).
↪geojson",
            "crosswalks": "infrastructure/Crosswalks_-_Open_Data.geojson",
            "education": "socio-economic/Neighborhood_Educational_Attainment.
↪geojson",
            "fire_stations": "infrastructure/Fire_Stations.geojson",
            "income": "socio-economic/Neighborhood_Income.geojson",
            "landfills": "infrastructure/Landfills_(Eastern_Pima_County).
↪geojson",
            "major_streets": "infrastructure/
↪Major_Streets_and_Routes_-_Open_Data.geojson",
            "streetcar_routes": "infrastructure/
↪Modern_Streetcar_Route_(Polygon)_-_Open_Data.geojson",
            "streetcar_stops": "infrastructure/
↪Modern_Streetcar_Stops_-_Open_Data.geojson",
            "scenic_routes": "infrastructure/Scenic_Routes_-_Open_Data.geojson",
            "sidewalks": "infrastructure/Sidewalks.geojson",
            "streetlights": "infrastructure/
↪Streetlights_-_City_of_Tucson_-_Open_Data.geojson",
            "suntran_stops": "infrastructure/Sun_Tran_Bus_Stops_-_Open_Data.
↪geojson",
        },
    }

    def __getitem__(self, key: str) -> Any:
        """Get a configuration value by key, with KeyError if missing."""
        if key not in Config._config_data:
            msg = (
                f"Could not find key '{key}' in config."
                + f"Available keys: {list(Config._config_data.keys())}"
            )
            raise KeyError(msg)
        return Config._config_data[key]

    def __setitem__(self, key: str, value: Union[str, Path]) -> None:
        """Set a configuration value by key, converting Path to string if␣
↪necessary."""
        if isinstance(value, Path):
            value = str(value.resolve())
        Config._config_data[key] = value


config = Config()
```

2 Model 1—Subsection-Level Crime Frequency Prediction from Infrastructure Features

2.1 Data Collection

2.1.1 Fetch and Load Datasets

[1078]:
```
_cache: Dict[str, gpd.GeoDataFrame] = {}


def load_dataset(dataset_name: str):
    if dataset_name in _cache:  # Check if the dataset is already cached
        return _cache[dataset_name]

    # Load the dataset from Config if not cached
    filepath = f"https://raw.githubusercontent.com/christian-byrne/
↪tucson-crime-models/main/data/{config['data_sources'][dataset_name]}"

    if filepath.startswith("http"):
        response = requests.get(filepath)
        response.raise_for_status()
        with open(f"/tmp/{dataset_name}.geojson", "wb") as f:
            f.write(response.content)
        dataframe = gpd.read_file(f"/tmp/{dataset_name}.geojson")
    else:
        dataframe = gpd.read_file(filepath)

    _cache[dataset_name] = dataframe  # Cache the loaded dataset

    return dataframe
```

2.2 Data Processing and Cleaning

2.2.1 Define the Area of Interest

Since we are using a large number of datasets, there is some variance in the bounds across each dataset. We attempt to resolve this by choosing a single dataset to use as the reference for the bounds of the area of interest.

The chosen dataset is treated hyperparameter.

In initial testing, we started by using the Arrests dataset (target variable). However, it performed poorly perhaps because it circumscribed far too large of an area. Additionally, it naturally didn't allow us to capture areas of zero-crime, which potentially inhibited the model's ability to generalize.

The best-performing bounds were those from the `sidewalks` dataset, which sets a relatively tight boundary around the city center.

[1079]:
```
# Get the reference dataset name from the config object
ref_dataset = config["hp"]["dataset_bounds_reference"]
```

```
# Get the bounds of the reference dataset
min_x, min_y, max_x, max_y = load_dataset(ref_dataset).total_bounds
print(
    "Setting the bounds of the area of interest based on the\n",
    f"{ref_dataset} dataset's bounds ({min_x, min_y, max_x, max_y})",
)

# Update the bounds in the config object
config["tucson_bounds"] = [min_x, min_y, max_x, max_y]
```

```
Setting the bounds of the area of interest based on the
 sidewalks dataset's bounds ((np.float64(-111.05254574930352), np.float64(32.
 ↪06233602492571),
np.float64(-110.72263755643428), np.float64(32.32006865632446)))
```

2.2.2 Partition Area of Interest into Subsections (Items)

The items/rows for both models are subsections of the area of interest.

We create a function to partition the area of interest into subsections using a simple grid approach.

Note on the Sampling Methodology:

In the early stages of developing the model, we created a function that simply partitioned the area of interest into a grid with equally-sized subsections equally spaced apart. However, we discovered that the model performed better the more randomness we introduced into the subsection creation process. That is, when we created items by randomly sampling subsections on the map (of random location and random size), the model performed significantly better (from 0.7 to 0.96 R^2).

```
[1080]: def gen_ran_subsection_sample(
            bounds_box: BoundingBox, target_num: int, load_from_file: bool = False
        ) -> List[BoundingBox]:
            """
            Generate a semi-random sample of subsections from a geographical region␣
         ↪described by the bounds_box argument.

            Sizes are adjusted dynamically based on the target number and remaining␣
         ↪space. The target_num is only a target,
            since the random nature of sampling won't be able to gaurentee the exact␣
         ↪number of subsections.

            Params:
                domain: BoundingBox - (min_x, min_y, max_x, max_y) - bounding box of␣
         ↪the area to be divided.
                target_num: int - Target number of subsections.

            Returns:
```

```
        List[BoundingBox] - List of bounding boxes representing the subsections.
    """
    # If option is specified, load the pre-computed best-performing subsection␣
↪sample computed during hyperparam tuning
    if load_from_file:
        subsections = pd.read_csv("good-subsection-splits.csv")
        return [
            (row["min_x"], row["min_y"], row["max_x"], row["max_y"])
            for _, row in subsections.iterrows()
        ]

    min_x, min_y, max_x, max_y = bounds_box
    total_width = max_x - min_x
    total_height = max_y - min_y

    # Calculate the approximate size of each subsection
    approx_size = (total_width * total_height) / target_num
    approx_width = approx_size**0.5  # Assume square-like subsections
    approx_height = approx_size**0.5

    subsections = []
    current_y = min_y

    # Partition rows
    while total_height > 0:
        # Determine row height based on approximation, with a random adjustment
        row_height = min(
            random.uniform(0.8 * approx_height, 1.2 * approx_height),␣
↪total_height
        )
        total_height -= row_height
        current_x = min_x
        remaining_width = total_width

        # Partition columns within the row
        while remaining_width > 0:
            # Determine column width based on approximation, with a random␣
↪adjustment
            col_width = min(
                random.uniform(0.8 * approx_width, 1.2 * approx_width),␣
↪remaining_width
            )
            remaining_width -= col_width

            # Create the bounding box for this subsection
            subsections.append(
                (current_x, current_y, current_x + col_width, current_y +␣
↪row_height)
            )
            current_x += col_width

        current_y += row_height
    return subsections
```

The subsection sampling method is "semi-random" in the sense that it still accepts a target number of subsections and adjusts the paramters of the random number generation to ensure that the number of subsections created is close to the target number.

Note on the Subsection Count Hyperparameter:

K-fold cross-validation was used to determine the optimal number of subsections. On average, the model started to perform worse after increasing beyond 25 subsections or decreasing below 16 subsections. This may be due to the fact that too many subsections creates small subsections that will naturally be more variable in feature values and therefore cause overfitting on the training data; while too few subsections may not capture the underlying patterns in the data and cause underfitting.

[1081]:
```
print(f"Partitioning Tucson into {config['hp']['subsection_count']}␣
↪subsections")
subsections = gen_ran_subsection_sample(
    config["tucson_bounds"], config["hp"]["subsection_count"],␣
↪load_from_file=True
)
```

```
Partitioning Tucson into 40 subsections
```

2.3 Exploratory Data Analysis and Data Visualization

2.3.1 Visualize the Subsection Sample on a Geographic Map

We begin by visualizing the boundaries of the sampled subsections interpolated onto the map of the area of interest.

This process assists in

- Verifying the subsections (mostly) span the area of interest.
- Identifying any potential issues with the subsections (e.g., overlapping subsections, overly sparse subsections).
- Ensuring an adequate amount of noise is present in the subsections' locations and sizes.

[1082]:
```
def visualize_subsections_on_map(
    subsections: List[BoundingBox],
    center_coordinates: Tuple[float, float],
    zoom_start=11,
    fill_opacity=0.16,
    color="blue",
):
    """
    Visualize the geographical subsections on a Folium map.

    Parameters:
        subsections: list[tuple[float]] - List of bounding boxes for␣
 ↪subsections in (min_x, min_y, max_x, max_y) format.
        center_coordinates: tuple[float] - Latitude and longitude to center the␣
 ↪map on.
        zoom_start: int - Initial zoom level for the map.

    Returns:
    folium.Map: The Folium map with subsections visualized.
    """
    # Create a Folium map centered at the given location
    folium_map = folium.Map(location=center_coordinates, zoom_start=zoom_start)

    # Add rectangles for each subsection
    for subsection in subsections:
        bounds = [[subsection[1], subsection[0]], [subsection[3],␣
 ↪subsection[2]]]
        folium.Rectangle(
            bounds, color=color, fill=True, fill_opacity=fill_opacity
        ).add_to(folium_map)

    return folium_map


visualize_subsections_on_map(subsections, config["tucson_center_coordinates"])
```

```
[1082]: <folium.folium.Map at 0x7de7311d3bc0>
```

2.3.2 Explore Infrastructure Datasets

Our goal for the first model is to predict crime frequency from infrastructure features. Therefore, we need to explore the infrastructure datasets to understand the features available and their distributions.

We start by visualizing various infrastructure datasets on the map of the area of interest, also including the subsections.

From this, we get a better idea of how each dataset may be processed, transformed and/or used in the model.

If we are to consider many of the infrastructure features in terms of density, it becomes necessary to use a heatmap, as a scatterplot would be too cluttered.

We will also include the target variable `Arrests` as an additional heatmap layer to get an initial idea of how the infrastructure features correlate with crime frequency.

Since creating the `folium` maps is computationally expensive and adds a lot of overhead to the notebook, we find it best to use the layer control feature of `folium` to allow the user to toggle the visibility of each dataset.

```
[1083]: def visualize_feature_heatmaps(
            datasets: dict[str, gpd.GeoDataFrame],
            subsections: list[tuple[float, float, float, float]],
            zoom_start=12,
            fill_opacity=0.2,
            heatmap_radius=10,
        ):
            """
            Visualizes multiple heatmaps for different datasets on the same map with␣
          ↪toggle buttons.

            Params:
            datasets: dict[str, GeoDataFrame] - Dictionary where keys are category␣
          ↪names and values are GeoDataFrames for each category.
            subsections: list[tuple[float]] - List of bounding boxes for subsections in␣
          ↪(min_x, min_y, max_x, max_y) format.
            zoom_start: int - Initial zoom level of the map.
            fill_opacity: float - Opacity of the subsection rectangles.
            heatmap_radius: int - Radius of the heatmap points.

            Returns:
            folium.Map: The Folium map with multiple heatmaps and toggle buttons.
            """
            # Create a Folium map centered on Tucson
            folium_map = folium.Map(
                location=config["tucson_center_coordinates"],
                zoom_start=zoom_start,
            )

            # Add rectangles for each subsection
            for subsection in subsections:
                bounds = [[subsection[1], subsection[0]], [subsection[3],␣
          ↪subsection[2]]]
                folium.Rectangle(
                    bounds, color="blue", fill=True, fill_opacity=fill_opacity
                ).add_to(folium_map)

            # Create a colormap for categories
            colormap = LinearColormap(
                colors=[
                    "red",
                    "green",
                    "blue",
                    "purple",
                    "orange",
                    "yellow",
                    "cyan",
                    "magenta",
                    "lime",
                    "pink",
                    "teal",
                    "lavender",
                ],
                index=range(len(datasets)),
                vmin=0,
                vmax=len(datasets) - 1,
            )

            # Add heatmaps and category groups
            for i, (category, data) in enumerate(datasets.items()):
                data = data[data.geometry.notnull()]  # Filter out missing or non-Point␣
        ↪geometry
                category_group = folium.FeatureGroup(
                    name=f"{category} Heatmap",
                    show=category == "arrests" or category == "bicycle_boulevards",
                )
                heatmap_data = []

                # Extract heatmap data from GeoDataFrame
                for _, obj in data.iterrows():
                    if obj.geometry.geom_type == "Point":
                        heatmap_data.append([obj.geometry.y, obj.geometry.x])
                    elif obj.geometry.geom_type == "LineString":
                        heatmap_data.extend(
                            [[point[1], point[0]] for point in obj.geometry.coords]
                        )

                # Add heatmap with specific color
                if heatmap_data:
                    color = colormap.rgb_hex_str(i / (len(datasets) - 1))  # Use hex␣
        ↪color
                    HeatMap(
                        heatmap_data,
                        radius=heatmap_radius,
                        gradient={0: color},  # Gradient for the category
                        blur=15,
                    ).add_to(category_group)
                category_group.add_to(folium_map)  # Add the category group to the map
            folium.LayerControl().add_to(folium_map)  # Add LayerControl for toggling␣
        ↪layers
```

```
    # Add the legend to the map
    legend_html = """
    <div style="
        position: fixed;
        bottom: 50px;
        left: 50px;
        width: 150px;
        height: auto;
        padding: 10px;
        background-color: white;
        z-index: 9999;
        font-size: 14px;
        border: 1px solid black;
        border-radius: 5px;
    ">
    <b>Legend</b> - Use the layer button on the top right of the map to toggle␣
  ↪feature heatmap layers<br>
    """
    for i, category in enumerate(datasets.keys()):
        color = colormap.rgb_hex_str(i / (len(datasets) - 1))
        legend_html += f"""
        <i style="background: {color}; width: 10px; height: 10px; display:␣
  ↪inline-block;"></i>
        {category}<br>
        """
    legend_html += "</div>"
    folium_map.get_root().html.add_child(folium.Element(legend_html))

    return folium_map
```

[1084]:
```
# Load each infrastructure dataset and arrests dataset and map them to a name
infra_data = {
    dataset_name: load_dataset(dataset_name)
    for dataset_name in config["feature_names"]["model1"] + ["arrests"]
}
visualize_feature_heatmaps(infra_data, subsections)
```

[1084]:
```
<folium.folium.Map at 0x7de72ff3f500>
```

Two concerns/observations are immediately clear:

1. Some of the infrastrcture features are very sparse, indicating that we should not represent them as densities.
2. Some of the pairs of features also seem to have very strong inter-feature correlations, which may indicate that they are measuring the same underlying phenomenon and also that we should account for this in the model.

We will address both of these concerns in the following sections.

Transform Sparse Features from *Incidence Count* to *Distance to Nearest* To address point (1) above, we identify the sparse features using the heatmap visualization and transform them from incidence count to distance to the nearest feature.

This way, we can still capture the information that the feature provides without inappropriately inferring a linear relationship with the feature's nominal density.

Display the features identified as sparse:

[1085]:
```
print(
    "Identified the following features as having sparse data and requiring␣
 ↪transformation to distance metrics:\n",
    f"{config['hp']['distance_to_nearest_infrastructure']}",
)
```

```
Identified the following features as having sparse data and requiring␣
 ↪transformation to distance metrics:
 ['fire_stations', 'landfills', 'streetcar_stops', 'streetcar_routes']
```

2.3.3 Define Functions Process the two Types of Features

Process Density Features For the density features (non-sparse data), we will use the nominal incidence count as the feature value.

To do so, define a function that extracts the feature values from the datasets and assigns them to the subsections.

Note on the Feature Value Extraction Methodology:

To determine what qualifies an incidence of a feature being "inside" of a subection (e.g., what constitutes a sidewalk being inside of a subsection), we must choose one of a few possible methods:

- `within`: The feature instance is entirely conatined within the subsection.
- `intersects`: The feature instance shares any part of its space with the subsection.
- `contains`: The subsection and feature instance overlap, but exclude cases where one geometry entirely contains or is entirely contained by the other.

We tried treating this as a hyperparam and performing tuning on it, but found ultimately that using a heuristic approach yielded the best results. That is, the method for each feature is determined in an intuitive way. The exact choices are displayed below

[1086]:
```
# Report the geometry method for each infrastructure type
for feature_, geo_method in config["hp"]["geometry_contains_method"].items():
    print(f"{feature_:<20} {geo_method}")
```

```
arrests              within


bicycle_boulevards   intersects
```

```
bridges             within

crosswalks          intersects

fire_stations       within

landfills           within

major_streets       intersects

streetcar_routes    intersects

streetcar_stops     within

scenic_routes       intersects

sidewalks           intersects

streetlights        within

suntran_stops       within
```

[1087]:

```
# def count_objects_in_subsection(
#     objects: gpd.GeoDataFrame, subsection: BoundingBox, object_name: str
# ):
#     """
#     Accept list of locations of objects (sidewalk, landfill, crosswalk, etc.)␣
↪and a subsection area, return the number of objects in the subsection.

#     Params:
#     objects: GeoDataFrame - locations of objects
#     subsection: tuple[Float] - (min_x, min_y, max_x, max_y) - bounding box of␣
↪the subsection
#     """
#     subsection_box = box(*subsection)

#     # Determine the method to use for checking if the object is within the␣
↪subsection (can be tuned)
#     method = config["hp"]["geometry_contains_method"][object_name]

#     # Filter objects within subsection
#     if method == "within":
#         objects_in_subsection = objects[objects.geometry.
↪within(subsection_box)]
#     elif method == "intersects":
#         objects_in_subsection = objects[objects.geometry.
↪intersects(subsection_box)]
#     elif method == "overlaps":
#         objects_in_subsection = objects[objects.geometry.
↪overlaps(subsection_box)]
#     else:
#         raise ValueError(f"Invalid geometry method: {method}")

#     return len(objects_in_subsection)


# def count_objects_in_subsection(
#     objects: gpd.GeoDataFrame,
#     subsection: Union[tuple[float, float, float, float], Polygon,␣
↪MultiPolygon],
#     object_name: str,
#     use_polygon: bool = False,
# ) -> int:
#     """
#     Count objects (sidewalk, landfill, crosswalk, etc.) within a given␣
↪subsection or neighborhood.

#     Params:
#     objects: GeoDataFrame - Locations of objects.
#     subsection: Union[BoundingBox, Polygon, MultiPolygon] - Bounding box␣
↪(min_x, min_y, max_x, max_y)
#                 or geometry of the subsection/neighborhood.
#     object_name: str - Name of the object type, used for determining the␣
↪geometry matching method.
#     use_polygon: bool - Whether to treat the subsection as a Polygon/
↪MultiPolygon. Defaults to False.

#     Returns:
#     int - Number of objects in the subsection.
#     """
#     # If using a bounding box, create the subsection as a box
#     if not use_polygon and isinstance(subsection, tuple):
#         subsection_geom = box(*subsection)
#     elif use_polygon and isinstance(subsection, (Polygon, MultiPolygon)):
#         subsection_geom = subsection

#     # Determine the method to use for spatial filtering
#     method = config["hp"]["geometry_contains_method"][object_name]

#     # Filter objects based on spatial relationship
#     if method == "within":
#         objects_in_subsection = objects[objects.geometry.
 ↪within(subsection_geom)]
#     elif method == "intersects":
#         objects_in_subsection = objects[objects.geometry.
 ↪intersects(subsection_geom)]
#     elif method == "overlaps":
#         objects_in_subsection = objects[objects.geometry.
 ↪overlaps(subsection_geom)]
#     else:
#         raise ValueError(f"Invalid geometry method: {method}")

#     return len(objects_in_subsection)

def count_objects_in_subsection(
    objects: gpd.GeoDataFrame,
    subsection: Union[tuple[float, float, float, float], Polygon, MultiPolygon],
    object_name: str,
    use_polygon: bool = False,
    condition: Callable[[gpd.GeoSeries], bool] = None,
) -> int:
    """
    Count objects (sidewalk, landfill, crosswalk, etc.) within a given␣
 ↪subsection or neighborhood,
    optionally filtering objects based on a condition.

    Params:
    objects: GeoDataFrame - Locations of objects.
    subsection: Union[BoundingBox, Polygon, MultiPolygon] - Bounding box␣
 ↪(min_x, min_y, max_x, max_y)
                or geometry of the subsection/neighborhood.
    object_name: str - Name of the object type, used for determining the␣
 ↪geometry matching method.
    use_polygon: bool - Whether to treat the subsection as a Polygon/
 ↪MultiPolygon. Defaults to False.
    condition: Callable[[gpd.GeoSeries], bool] - Optional function to filter␣
 ↪objects based on a condition.

    Returns:
      int - Number of objects in the subsection meeting the condition (if␣
 ↪provided).
    """
    # If using a bounding box, create the subsection as a box
    if not use_polygon and isinstance(subsection, tuple):
        subsection_geom = box(*subsection)
    elif use_polygon and isinstance(subsection, (Polygon, MultiPolygon)):
        subsection_geom = subsection
    else:
        raise ValueError(
            "Invalid subsection type. Provide a bounding box (tuple) or a␣
↪Polygon/MultiPolygon."
        )

    # Determine the method to use for spatial filtering
    method = config["hp"]["geometry_contains_method"][object_name]

    # Filter objects based on spatial relationship
    if method == "within":
        objects_in_subsection = objects[objects.geometry.
↪within(subsection_geom)]
    elif method == "intersects":
        objects_in_subsection = objects[objects.geometry.
↪intersects(subsection_geom)]
    elif method == "overlaps":
        objects_in_subsection = objects[objects.geometry.
↪overlaps(subsection_geom)]
    else:
        raise ValueError(f"Invalid geometry method: {method}")

    # Apply the condition if provided
    if condition:
        objects_in_subsection = objects_in_subsection[objects_in_subsection.
↪apply(condition, axis=1)]

    return len(objects_in_subsection)
```

Process Distance Features For the distance features (sparse data), we will use the distance to the nearest feature as the feature value.

Define a function that calculates the distance from the center point of each subsection to the nearest feature of the given type. If the feature is inside of the subsection, the distance will be zero.

[1088]:
```
def distance_to_nearest(objects: gpd.GeoDataFrame, subsection: BoundingBox):
    # Create a point at the center of the subsection
    center_x = (subsection[0] + subsection[2]) / 2
    center_y = (subsection[1] + subsection[3]) / 2
    center_point = gpd.GeoSeries([Point(center_x, center_y)], crs="EPSG:4326")

    # Reproject to a projected CRS for distance calculations (e.g., UTM or a␣
↪global CRS like EPSG:3857)
```

```
    projected_crs = "EPSG:3857"
    objects_projected = objects.to_crs(projected_crs)
    center_point_projected = center_point.to_crs(projected_crs)

    # If one of the objects is INSIDE of the subsection, the distance will be␣
↪0, do this separately
    if objects_projected.within(center_point_projected.iloc[0]).any():
        return 0.0

    # Calculate the distance to the nearest object
    return objects_projected.distance(center_point_projected.iloc[0]).min()
```

2.3.4 Create the Input Features

Collect the density and distance features for each subsection into a single dataframe.

Begin with the distance features:

[1089]:
```
col_names = []
cols = []
for feature in config["hp"]["distance_to_nearest_infrastructure"]:
    feature_col = []
    dataset = load_dataset(feature)
    for subsection in subsections:
        feature_col.append(distance_to_nearest(dataset, subsection))

    cols.append(feature_col)
    col_names.append(f"distance_to_nearest_{feature}")

df = pd.DataFrame(cols).T  # Create a DataFrame
df.columns = col_names
df.head()  # Preview the DataFrame
```

[1089]:
```
   distance_to_nearest_fire_stations  distance_to_nearest_landfills  \
0                        3570.684876                    8181.369052
1                        3567.317034                    3671.235203
2                         901.712818                     718.522916
3                        4208.127744                    3871.341136
4                        2906.090212                    3294.865540

   distance_to_nearest_streetcar_stops  distance_to_nearest_streetcar_routes
0                         18084.590003                          18065.251999
1                         17271.049392                          17262.114571
2                         17569.097586                          17522.332029
3                         18875.510098                          18822.484825
4                         20722.911756                          20662.432774
```

Then add the density features:

```
[1090]: for feature in config["hp"]["count_instances_infrastructure"]:
            dataset = load_dataset(feature)
            df[f"count_{feature}"] = [
                count_objects_in_subsection(dataset, subsection, feature)
                for subsection in subsections
            ]

        df.head()  # Preview the DataFrame
```

```
[1090]:    distance_to_nearest_fire_stations  distance_to_nearest_landfills  \
        0                        3570.684876                    8181.369052
        1                        3567.317034                    3671.235203
        2                         901.712818                     718.522916
        3                        4208.127744                    3871.341136
        4                        2906.090212                    3294.865540

           distance_to_nearest_streetcar_stops  distance_to_nearest_streetcar_routes  \
        0                         18084.590003                          18065.251999
        1                         17271.049392                          17262.114571
        2                         17569.097586                          17522.332029
        3                         18875.510098                          18822.484825
        4                         20722.911756                          20662.432774

           count_bicycle_boulevards  count_bridges  count_crosswalks  \
        0                         0              0                 0
        1                         0              0                 0
        2                         0              0                 0
        3                         0              0                 0
        4                         0              0                 0

           count_major_streets  count_sidewalks  count_streetlights  \
        0                    0                0                   0
        1                    2                0                   0
        2                    6                0                   0
        3                   11                0                   0
        4                   18                0                   0

           count_suntran_stops
        0                    2
        1                    4
        2                    6
        3                    0
        4                    0
```

Visualize Feature Sparsity after Transformation Visualize the sparsity by measuring the proportion of 0-values in order to verify that the sparse features have been adequately handled (point 1 from above).

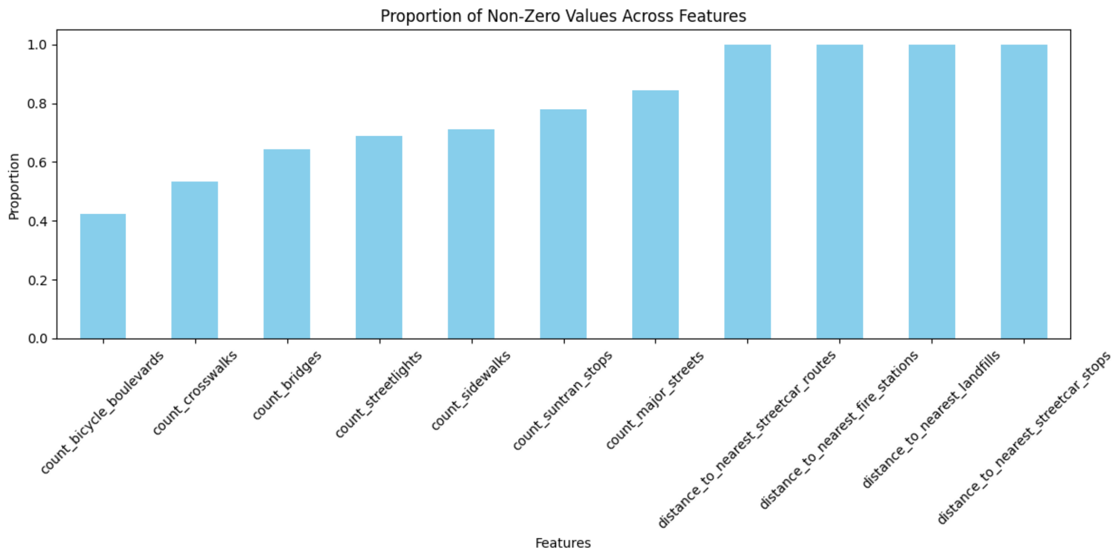
This will also allow us to start addressing point (2) — the potential multicollinearity between features.

Define function to visualize feature sparsity:

[1091]:
```
def visualize_feature_nonzero_proportion(dataframe: pd.DataFrame):
    """
    Create a bar plot showing the proportion of non-zero values for each␣
 ↪feature.

    Params:
    dataframe: pd.DataFrame - Input DataFrame to analyze.

    Returns:
    None
    """
    nonzero_proportion = (dataframe != 0).sum(axis=0) / len(dataframe)
    plt.figure(figsize=(12, 6))
    nonzero_proportion.sort_values().plot(kind="bar", color="skyblue")
    plt.title("Proportion of Non-Zero Values Across Features")
    plt.ylabel("Proportion")
    plt.xlabel("Features")
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()
```

Plot the sparsity of each feature side-by-side:

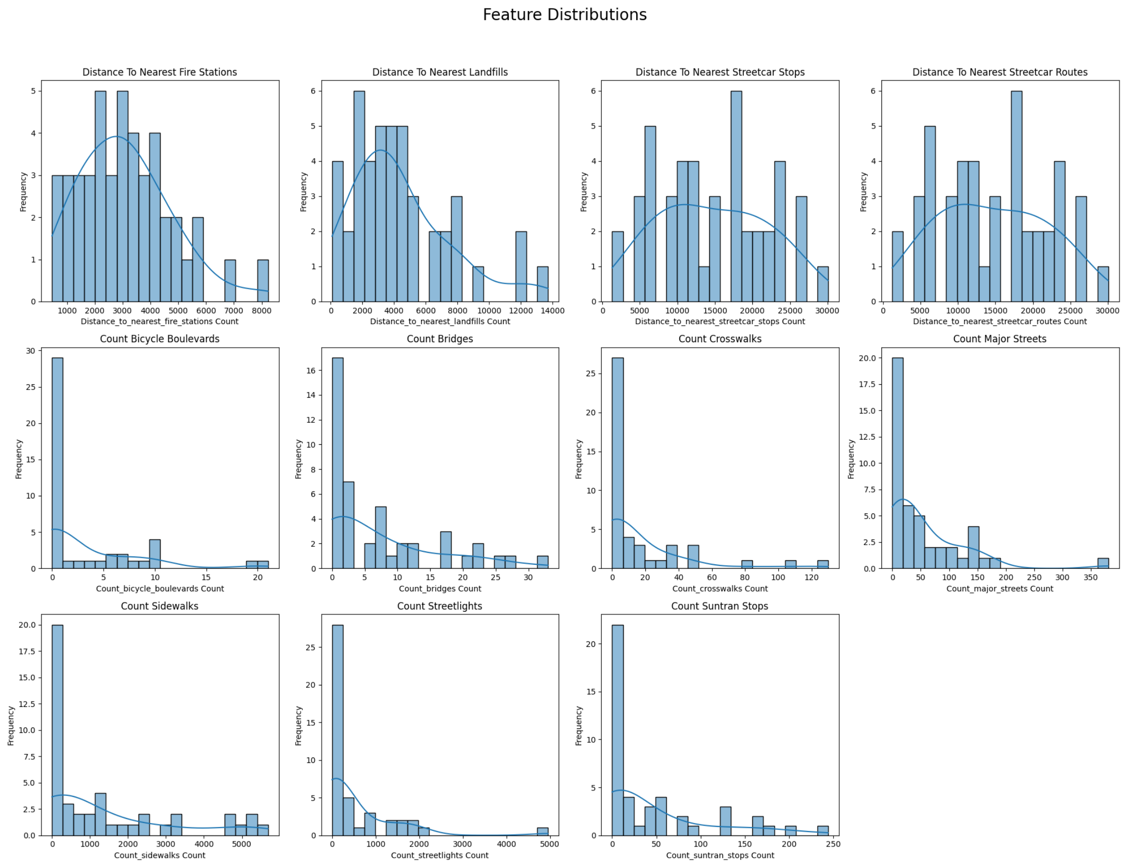
[1092]:
```
visualize_feature_nonzero_proportion(df)
```



Visualize Feature Distributions As a final measure, we verify that each feature is appropriately processed and transformed/scaled by visualizing the distributions of the features.

Define function to visualize feature distributions:

[1093]:
```
def visualize_feature_densities(dataframe: pd.DataFrame):
    """Create a single plot with subplots for each feature in the input␣
 ↪DataFrame.
    Show the distribution of each feature in order to determine if␣
 ↪transformations, projections, or re-engineering are needed.
    """
    # Create a figure with a subplot for each feature
    fig, axs = plt.subplots(len(dataframe.columns) // 4 + 1, 4, figsize=(20,␣
 ↪16))
    fig.suptitle("Feature Distributions", fontsize=20)

    # Iterate over each feature and create a histogram
    for i, feature in enumerate(dataframe.columns):
        sns.histplot(dataframe[feature], kde=True, bins=20, ax=axs[i // 4, i %␣
 ↪4])
        axs[i // 4, i % 4].set_xlabel(f"{feature.capitalize()} Count")

        # Only need to label y if it's the first column
        axs[i // 4, i % 4].set_ylabel("Frequency")
        axs[i // 4, i % 4].set_title(f"{feature.replace('_', ' ').title()}")

    # Hide any unused subplots
    for i in range(len(dataframe.columns), len(axs.flat)):
        axs.flat[i].set_visible(False)

    # Adjust layout and display the plot
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()


visualize_feature_densities(df)
```



2.3.5 Identify Multicollinearity within Input Features

We now attempt to address concern/point (2): the potential multicollinearity between features.

Here's why this step is important:

- Highly correlated features (multicollinearity) can make coefficient estimates unreliable by unduly weighting certain features who have unaccounted contributions from other features in the loss calculation
- Removing or consolidating correlated features helps reduce redundancy, making the model simpler and easier to interpret while maintaining performance

By attempting to understand and interpret highly correlated features manually, we may be able to combine collinear feature combinations into a single, more informative feature by using practical knowledge (as opposed to leaving it all to the regularization and best-subset selection algorithms).

Since all of our features in this model are continuous, we will use the Pearson method to construct a correlation matrix and visualize it using a heatmap.

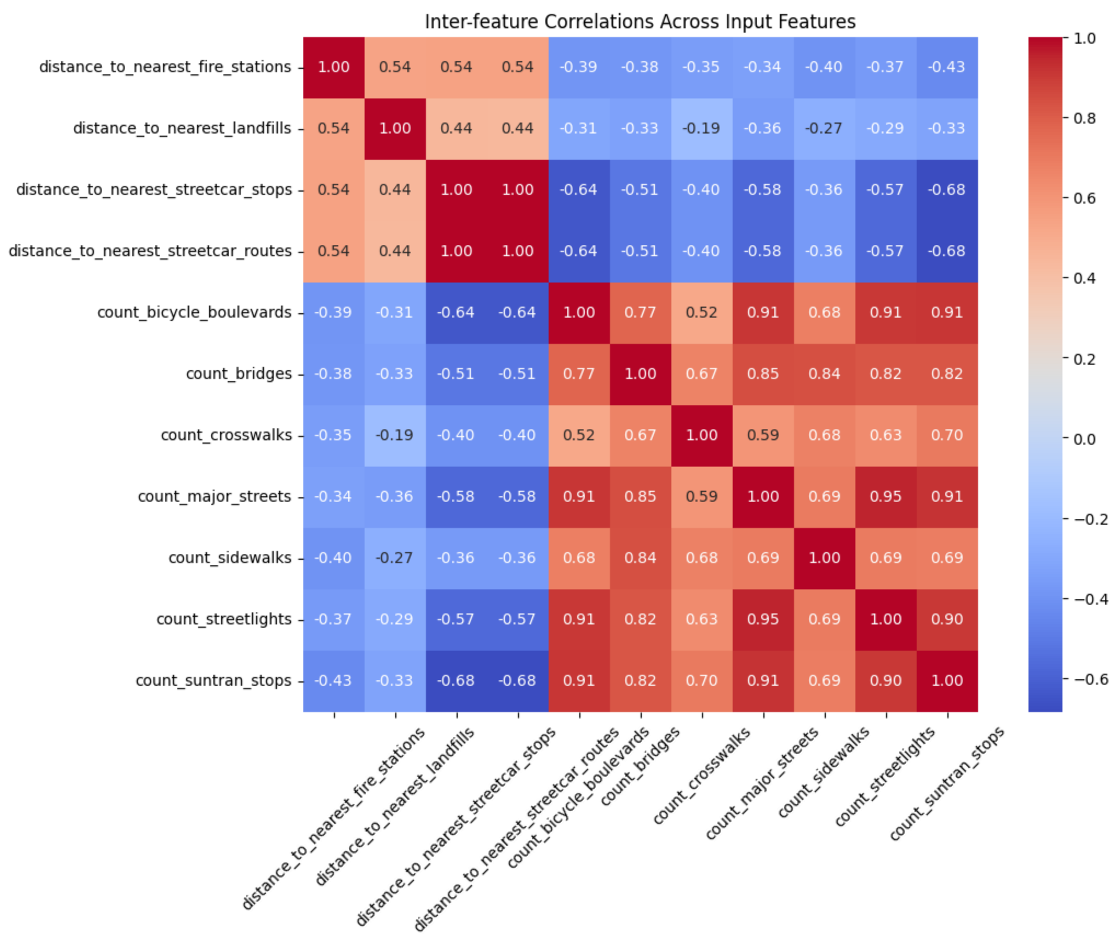
Caveats

- Correlation is only a linear relationship. Features with non-linear dependencies may still cause issues and may need separate analysis (e.g., mutual information).

- In some cases, correlated features may hold independent predictive value for non-linear models (e.g., gradient boosting), so removing them might not always be necessary.

Define a function to visualize the correlation matrix:

```
[1094]: def correlation_heatmap(dataframe: pd.DataFrame):
            """
            Generate a heatmap of correlation coefficients for all numerical features␣
         ↪in the DataFrame.
            """
            corr_matrix = dataframe.corr(method="pearson")  # Compute correlation matrix
            plt.figure(figsize=(12, 8))
            sns.heatmap(
                corr_matrix, annot=True, fmt=".2f", cmap="coolwarm", cbar=True,␣
         ↪square=True
            )
            plt.xticks(rotation=45)  # Rotate the x-tick labels for better readability
            plt.title("Inter-feature Correlations Across Input Features")
            plt.show()


        correlation_heatmap(df)
```



	distance_to_nearest_fire_stations	distance_to_nearest_landfills	distance_to_nearest_streetcar_stops	distance_to_nearest_streetcar_routes	count_bicycle_boulevards	count_bridges	count_crosswalks	count_major_streets	count_sidewalks	count_streetlights	count_suntran_stops
distance_to_nearest_fire_stations	1.00	0.54	0.54	0.54	-0.39	-0.38	-0.35	-0.34	-0.40	-0.37	-0.43
distance_to_nearest_landfills	0.54	1.00	0.44	0.44	-0.31	-0.33	-0.19	-0.36	-0.27	-0.29	-0.33
distance_to_nearest_streetcar_stops	0.54	0.44	1.00	1.00	-0.64	-0.51	-0.40	-0.58	-0.36	-0.57	-0.68
distance_to_nearest_streetcar_routes	0.54	0.44	1.00	1.00	-0.64	-0.51	-0.40	-0.58	-0.36	-0.57	-0.68
count_bicycle_boulevards	-0.39	-0.31	-0.64	-0.64	1.00	0.77	0.52	0.91	0.68	0.91	0.91
count_bridges	-0.38	-0.33	-0.51	-0.51	0.77	1.00	0.67	0.85	0.84	0.82	0.82
count_crosswalks	-0.35	-0.19	-0.40	-0.40	0.52	0.67	1.00	0.59	0.68	0.63	0.70
count_major_streets	-0.34	-0.36	-0.58	-0.58	0.91	0.85	0.59	1.00	0.69	0.95	0.91
count_sidewalks	-0.40	-0.27	-0.36	-0.36	0.68	0.84	0.68	0.69	1.00	0.69	0.69
count_streetlights	-0.37	-0.29	-0.57	-0.57	0.91	0.82	0.63	0.95	0.69	1.00	0.90
count_suntran_stops	-0.43	-0.33	-0.68	-0.68	0.91	0.82	0.70	0.91	0.69	0.90	1.00

We will set a threshold for correlation strength and report any pairs of features that exceed this threshold.

```
[1095]: MULTICOLLINEARITY_THRESHOLD = 0.93

correlation_matrix = df.corr() # Compute the correlation matrix

# Find pairs with correlation above the threshold
high_correlation = (correlation_matrix > MULTICOLLINEARITY_THRESHOLD) &␣
 ↪(correlation_matrix < 1)

# Stack the correlation matrix and filter based on the high_correlation mask
high_correlation_pairs = (
    correlation_matrix
    .where(high_correlation)  # Mask correlations not above the threshold
    .stack()  # Convert to Series for filtering
    .reset_index()  # Convert to DataFrame for better readability
)
```

```
high_correlation_pairs.columns = ["Feature 1", "Feature 2", "Correlation"] #␣
  ↪Rename columns for clarity
high_correlation_pairs # Display the pairs of features with high correlation
```

[1095]:
```
                              Feature 1                           Feature 2  \
0   distance_to_nearest_streetcar_stops  distance_to_nearest_streetcar_routes
1  distance_to_nearest_streetcar_routes   distance_to_nearest_streetcar_stops
2                    count_major_streets                  count_streetlights
3                     count_streetlights                 count_major_streets

   Correlation
0     0.999993
1     0.999993
2     0.950527
3     0.950527
```

Combine/Drop Collinear Features From this analysis, it's clear that some features should either be dropped or combined.

Some of the features were dropped during initial testing (e.g., "business_licenses" and "scenic_routes")

For the highly-correlated feature pairs that weren't dropped in previous iterations of the model, we had better results combining them using PCA (perhaps because they share a very similar scale but were relatively sparse, so combining them into a single feature helped distribute the variance more evenly while not overfitting on the training data).

Use PCA to Combine Collinear Features with Same Scales

[1096]:
```
# Keep track of features that have already been processed
processed_features = set()

# Iterate through high correlation pairs
for _, row in high_correlation_pairs.iterrows():
    feature_1, feature_2 = row["Feature 1"], row["Feature 2"]

    # Skip if either feature has already been processed
    if feature_1 in processed_features or feature_2 in processed_features:
        continue

    # Perform PCA on the pair of features
    pca = PCA(n_components=1)
    new_feature_name = f"{feature_1}_and_{feature_2}_PCA"
    df[new_feature_name] = pca.fit_transform(df[[feature_1, feature_2]])

    # Drop the original features
    df.drop([feature_1, feature_2], axis=1, inplace=True)
    # Mark these features as processed
    processed_features.update([feature_1, feature_2])

# Display the updated DataFrame
df.head()
```

[1096]:

```
   distance_to_nearest_fire_stations  distance_to_nearest_landfills  \
0                        3570.684876                    8181.369052
1                        3567.317034                    3671.235203
2                         901.712818                     718.522916
3                        4208.127744                    3871.341136
4                        2906.090212                    3294.865540

   count_bicycle_boulevards  count_bridges  count_crosswalks  count_sidewalks  \
0                         0              0                 0                0
1                         0              0                 0                0
2                         0              0                 0                0
3                         0              0                 0                0
4                         0              0                 0                0

   count_suntran_stops  \
0                    2
1                    4
2                    6
3                    0
4                    0

distance_to_nearest_streetcar_stops_and_distance_to_nearest_streetcar_routes_PCA
\
0                                        4867.399364
1                                        3724.230406
2                                        4119.002802
3                                        5962.125494
4                                        8569.478500

   count_major_streets_and_count_streetlights_PCA
0                                     -508.595178
1                                     -508.448187
2                                     -508.154205
3                                     -507.786728
4                                     -507.272259
```

2.3.6 Create Target Feature

For the first model, the target feature will be the crime frequency in each subsection.

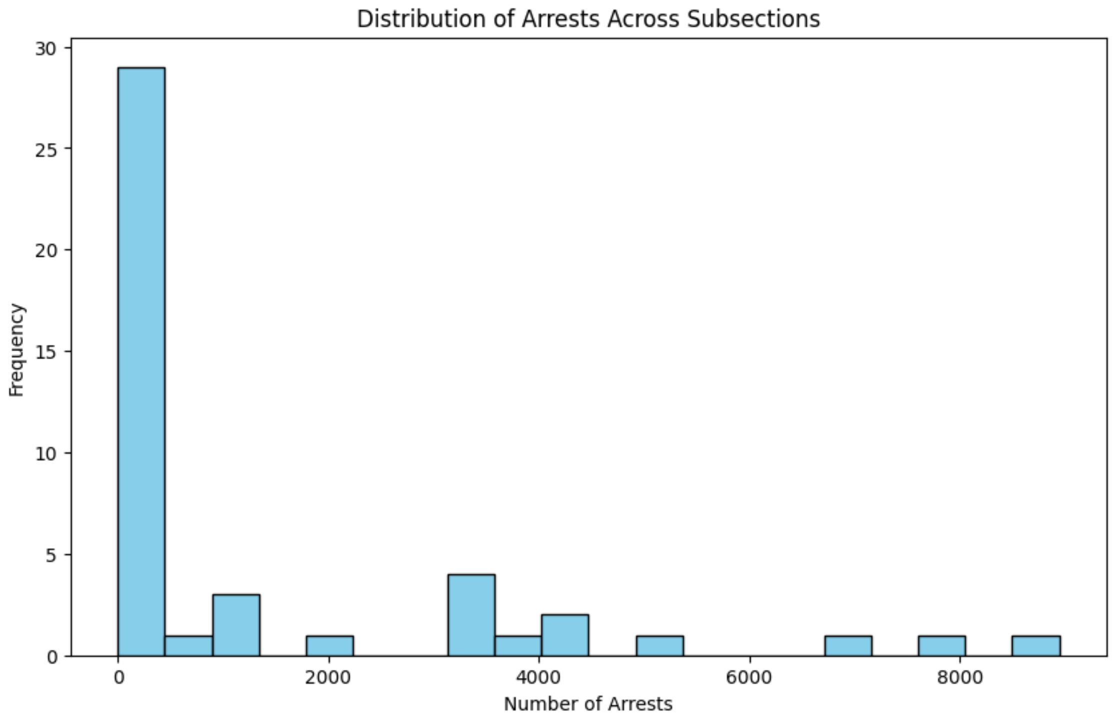
The process of creating this feature will follow the same steps as the input features that are based on density.

Load the arrests data and count the number of arrests in each subsection

```
[1097]: df["arrests"] = [
            count_objects_in_subsection(load_dataset("arrests"), subsection, "arrests")
            for subsection in subsections
        ]
```

Visualize the Target Variable's Distribution across Subsections

```
[1098]: plt.figure(figsize=(10, 6))
        plt.hist(df["arrests"], bins=20, color="skyblue", edgecolor="black")
        plt.xlabel("Number of Arrests")
        plt.ylabel("Frequency")
        plt.title("Distribution of Arrests Across Subsections")
        plt.show()
```

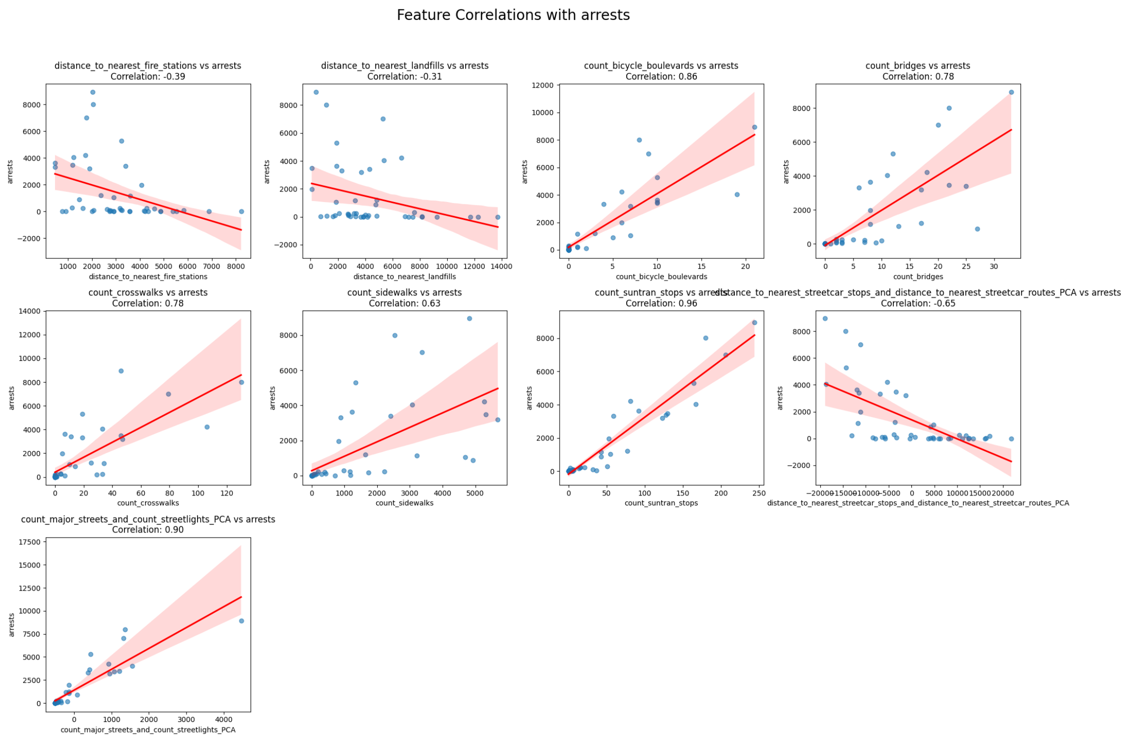


Visualize the Target Variable's Correlation with Each Feature Individually To get an idea of how the target variable correlates with each feature, we will visualize the target variable's distribution across the feature values.

This will help us understand the relationship between the target variable and each feature and identify any potential outliers.

This can also help us identify any potential non-linear relationships between the target variable and the features which might require additional feature engineering.

[1145]:
```
def visualize_feature_correlations(dataframe: pd.DataFrame, target_feature:␣
↪str):
    """
    Create subplots for each feature in the DataFrame, showing the relationship
    between each feature and the target variable, and print correlation␣
↪coefficients.
    """
    features = [col for col in dataframe.columns if col != target_feature]
    num_features = len(features)
    num_cols = 4
    num_rows = (num_features + num_cols - 1) // num_cols  # Compute rows␣
↪dynamically

    fig, axs = plt.subplots(
        num_rows, num_cols, figsize=(20, 5 * num_rows)
    )  # Create a figure with subplots
    fig.suptitle(f"Feature Correlations with {target_feature}", fontsize=20)
    axs = axs.flatten()  # Flatten axes for easier indexing

    # Iterate over features and plot
    for i, feature in enumerate(features):
        ax = axs[i]

        # Compute correlation coefficient
        corr_coef = (
            dataframe[feature].corr(dataframe[target_feature])
            if pd.api.types.is_numeric_dtype(dataframe[feature])
            else None
        )

        # Numerical features
        if pd.api.types.is_numeric_dtype(dataframe[feature]):
            sns.regplot(
                x=dataframe[feature],
                y=dataframe[target_feature],
                ax=ax,
                scatter_kws={"alpha": 0.6},
                line_kws={"color": "red"},
            )
            ax.set_title(f"{feature} vs {target_feature}\nCorrelation:␣
↪{corr_coef:.2f}")

        # Categorical features
        else:
            sns.boxplot(x=dataframe[feature], y=dataframe[target_feature],␣
↪ax=ax)
            ax.set_title(f"{feature} vs {target_feature}")
        ax.set_xlabel(feature)
        ax.set_ylabel(target_feature)

    # Hide unused subplots
    for j in range(i + 1, len(axs)):
        axs[j].set_visible(False)

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()


visualize_feature_correlations(df, "arrests")
```



2.4 Model Training

2.4.1 Split Data into Training and Testing Sets

We treat the split ratio as a hyperparameter subject to tuning.

[1100]:
```
train, test = train_test_split(df, test_size=config["hp"]["test_size"],␣
  ↪random_state=42)
```

2.4.2 Normalize Data

Scale (normalize) the data

[1101]:
```
# Ensure the target variable is indeed the last column
assert df.columns[-1] == "arrests"

X_features = df.columns[:-1]  # X = All columns except the last one
features = df.columns

# Initialize the StandardScaler
scaler = StandardScaler().fit(train[features])  # Scale feature columns

# Scaling is done after train/test split to avoid data leakage
train[features] = scaler.transform(train[features])
test[features] = scaler.transform(test[features])
```

[1102]:
```
print("Training data after scaling:")
train.head()
```

```
Training data after scaling:
```

[1102]:
```
    distance_to_nearest_fire_stations  distance_to_nearest_landfills  \
3                            0.688015                      -0.051772
6                           -0.297862                      -0.419722
24                          -1.157949                       0.450926
32                           0.606057                      -1.314827
19                          -0.751549                      -0.102081

    count_bicycle_boulevards  count_bridges  count_crosswalks  \
3                  -0.642138      -0.861412         -0.568302
6                  -0.642138       0.287137          0.451126
24                  2.987799       0.401992          0.591737
32                  0.504158       0.057427         -0.392538
19                  0.695207       1.091121          1.083875

    count_sidewalks  count_suntran_stops  \
3         -0.822299            -0.755807
6          0.177580            -0.529200
24         0.953785             1.947294
32        -0.344900             0.102063
19         2.464584             1.235100

distance_to_nearest_streetcar_stops_and_distance_to_nearest_streetcar_routes_PCA
\
3                                            0.564853
6                                            1.588206
```

```
24                                          -1.699279
32                                          -1.009773
19                                          -0.099773

    count_major_streets_and_count_streetlights_PCA   arrests
3                                        -0.556260 -0.635006
6                                        -0.217019 -0.554514
24                                        1.608218  1.145058
32                                       -0.159756  0.231494
19                                        0.978791  0.771188
```

[1103]:
```python
print("Testing data after scaling:")
test.head()
```

```
Testing data after scaling:
```

[1103]:
```
    distance_to_nearest_fire_stations  distance_to_nearest_landfills  \
39                          -0.227268                      -0.048572
25                          -0.832542                       0.421314
26                          -1.194060                      -1.305953
43                           2.347151                       2.753896
35                          -0.675493                       1.374155

    count_bicycle_boulevards  count_bridges  count_crosswalks  \
39                 -0.642138      -0.861412         -0.568302
25                  1.077306       1.435686          2.208761
26                  1.268355       1.665396          1.048722
43                 -0.642138      -0.861412         -0.568302
35                 -0.642138      -0.861412         -0.568302

    count_sidewalks  count_suntran_stops  \
39        -0.803804            -0.739621
25         1.120817             2.578557
26         2.251893             1.348403
43        -0.822299            -0.755807
35        -0.822299            -0.755807

distance_to_nearest_streetcar_stops_and_distance_to_nearest_streetcar_routes_PCA
\
39                                          -0.702999
25                                          -1.009440
26                                          -0.288384
43                                           1.166521
35                                           0.759384

    count_major_streets_and_count_streetlights_PCA   arrests
39                                       -0.556955 -0.631047
```

```
25                                            1.359451  2.449207
26                                            1.255042  0.892586
43                                           -0.557109 -0.635006
35                                           -0.556800 -0.632367
```

Split Data into Input (X) and Output (y) Data

[1104]:
```
target_feature_name = "arrests"
other = [col for col in train.columns if col != target_feature_name]
X_train, y_train = train[other], train[target_feature_name]
X_test, y_test = test[other], test[target_feature_name]
```

2.4.3 Hyperparameter Tuning and Model Selection

Initially, we train three models: Linear Regression using Best Subset, Ridge Regression, and Lasso Regression.

The three models will be evaluated and the best model will be selected based on the R^2 score.

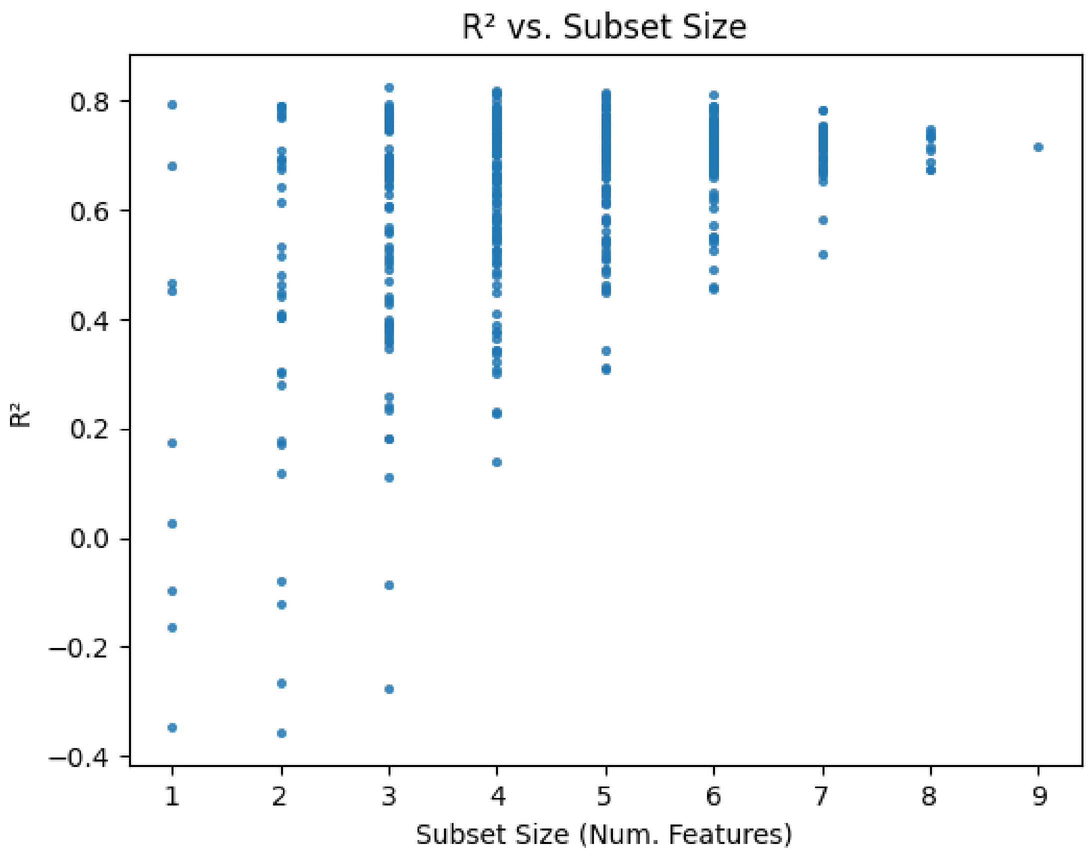
Best Subsets

[1105]:
```
# Initialize the linear regression model
best_subsets_lr = LinearRegression()
subset_sizes = []
r2_means_subsets = []
for subset_size in range(1, len(X_features) + 1):
    subsets = set(
        combinations(X_features, subset_size)
    )  # Find all possible subsets for current size

    # Calculate the mean R^2 score for each subset
    for subset in subsets:
        mean_r2 = cross_val_score(
            best_subsets_lr,
            X_train[list(subset)],
            y_train,
            cv=config["hp"]["cross_val_folds"]["best_subset"],
            scoring=config["hp"]["scoring_metric"]["best_subset"],
        ).mean()
        subset_sizes.append(subset_size)
        r2_means_subsets.append((mean_r2, subset))
```

Visualize the general performance of each subset size

[1106]:
```
plt.scatter(subset_sizes, [r2 for r2, _ in r2_means_subsets], alpha=0.8, s=7)
plt.xlabel("Subset Size (Num. Features)")
plt.ylabel("R²")
plt.title("R² vs. Subset Size")
```

[1106]:
```
Text(0.5, 1.0, 'R² vs. Subset Size')
```



Determine the best-performing subset size

```
[1107]: best_r2_subset_2, best_feature_combo = sorted(r2_means_subsets, key=lambda x:␣
          ↪-x[0])[0]
        print(
            f"Best-performing set of features:\n\t{best_feature_combo}\n\tR² =␣
          ↪{best_r2_subset_2}"
        )
```

```
Best-performing set of features:
        ('count_crosswalks', 'count_sidewalks', 'count_suntran_stops')
        R² = 0.8261252575190225
```

Ridge

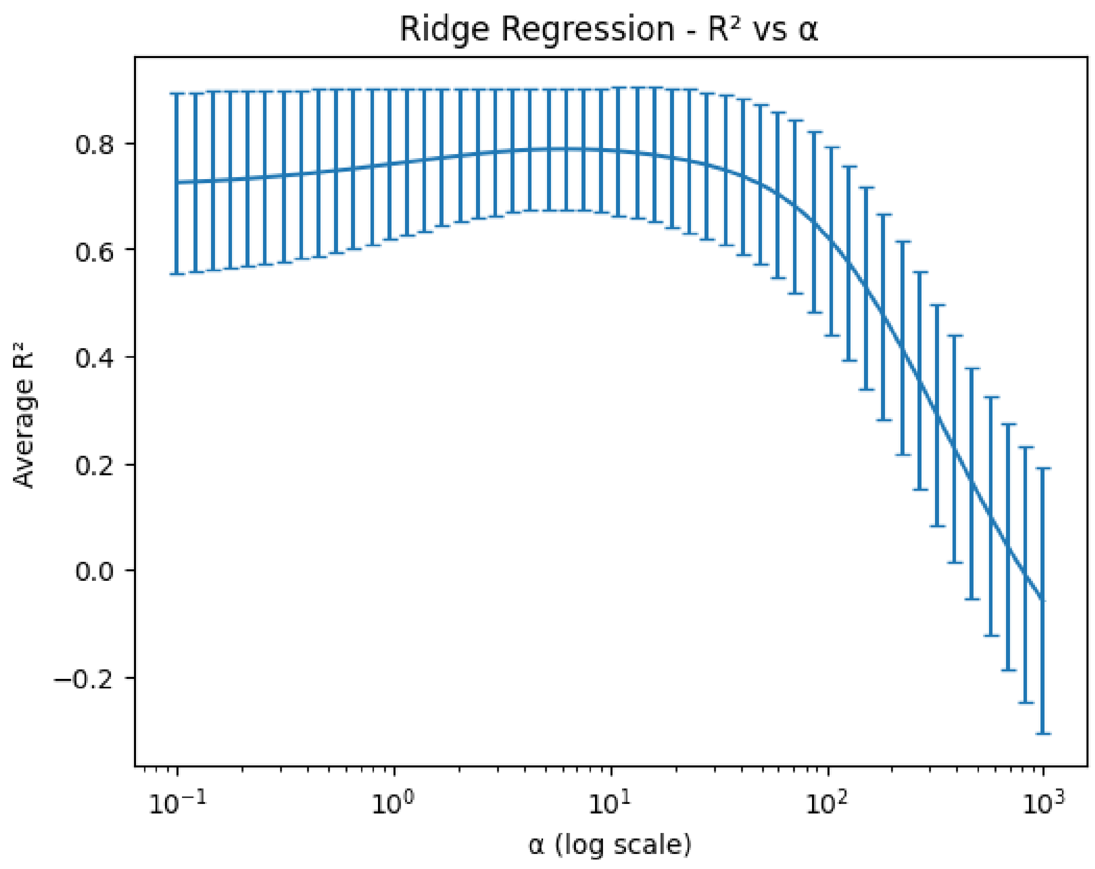
```
[1108]: # Create a range of 50 alpha values spaced logarithmically in the range [10^-1,␣
          ↪10^3]
        bounds_box = np.logspace(-1, 3, 50)

        # Find the best alpha value across domain
```

```python
r2_means_ridge, r2_stds_ridge = [], []
for alpha in bounds_box:
    reg = linear_model.Ridge(alpha=alpha)
    r2_vals = cross_val_score(
        reg,
        X_train,
        y_train,
        cv=config["hp"]["cross_val_folds"]["ridge"],
        scoring=config["hp"]["scoring_metric"]["ridge"],
    )
    r2_means_ridge.append(r2_vals.mean())
    r2_stds_ridge.append(r2_vals.std())
```

Visualize the performance of each alpha value

[1109]:
```python
plt.errorbar(bounds_box, r2_means_ridge, yerr=r2_stds_ridge, fmt="-", capsize=3)
plt.xscale("log")
plt.title("Ridge Regression - R² vs  ")
plt.xlabel("  (log scale)")
plt.ylabel("Average R²")
```

[1109]:
```
Text(0, 0.5, 'Average R²')
```



Determine the best-performing alpha value

```
[1110]: best_r2_ridge = np.max(r2_means_ridge)
        best_alpha_ridge = bounds_box[np.argmax(r2_means_ridge)]
        print(f"Best R²: {best_r2_ridge:.3f}\nBest  : {best_alpha_ridge:.3f}")
```

```
Best R²: 0.788
Best  : 6.251
```

Lasso

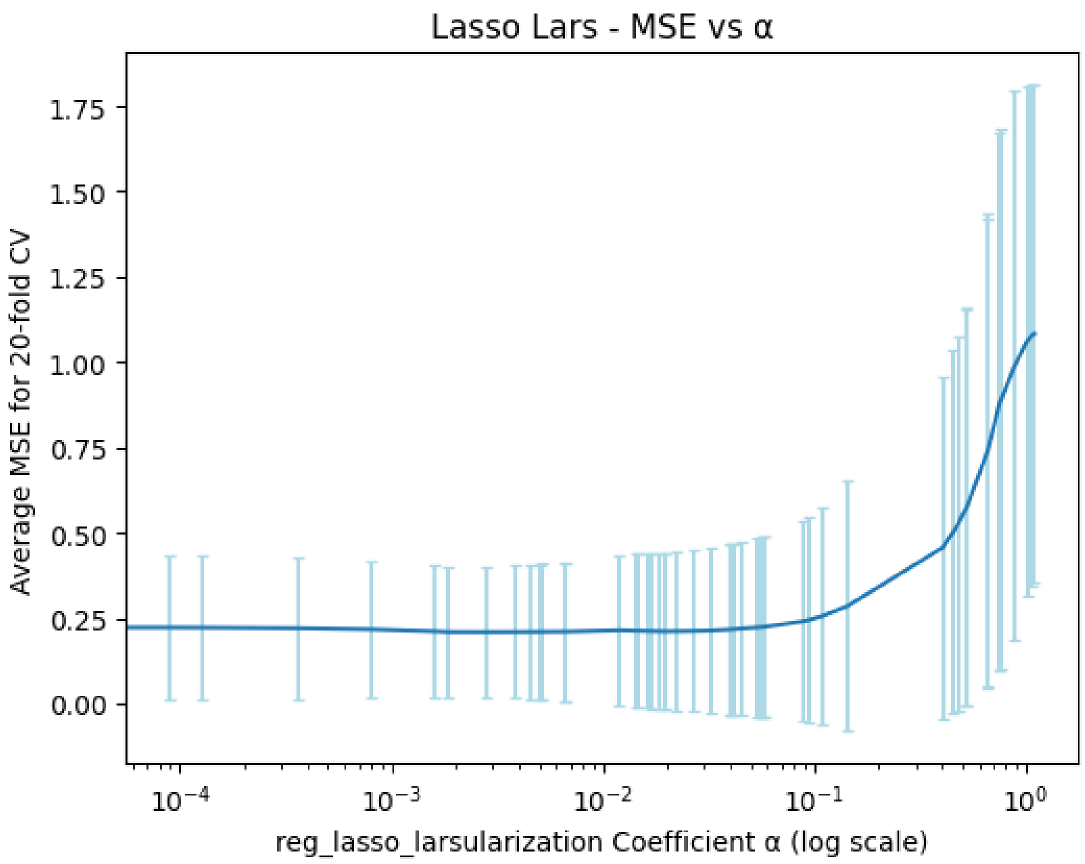
```
[1111]: reg_lasso_lars = linear_model.
          ↪LassoLarsCV(cv=config["hp"]["cross_val_folds"]["lasso"])
        reg_lasso_lars.fit(X_train, y_train)
        reg_lasso_lars.coef_
```

```
[1111]: array([ 0.01649396, -0.03661831,  0.09055428,  0.        ,  0.30460641,
               -0.17729898,  0.53646454, -0.00649058,  0.26606934])
```

Visualize the performance of each regularization coefficient

```
[1112]: # Plot the MSE for each   value
        plt.errorbar(
            reg_lasso_lars.cv_alphas_,  # =   values
            reg_lasso_lars.mse_path_.mean(axis=1),  # = mean of MSE for each   across␣
          ↪20 folds
            yerr=reg_lasso_lars.mse_path_.std(
                axis=1
            ),  # = standard deviation of MSE for each   across 20 folds
            fmt="-",
            capsize=2,
            ecolor="lightblue",
        )
        plt.xscale("log")
        plt.title("Lasso Lars - MSE vs  ")
        plt.xlabel("reg_lasso_larsularization Coefficient   (log scale)")
        plt.ylabel("Average MSE for 20-fold CV")
```

```
[1112]: Text(0, 0.5, 'Average MSE for 20-fold CV')
```



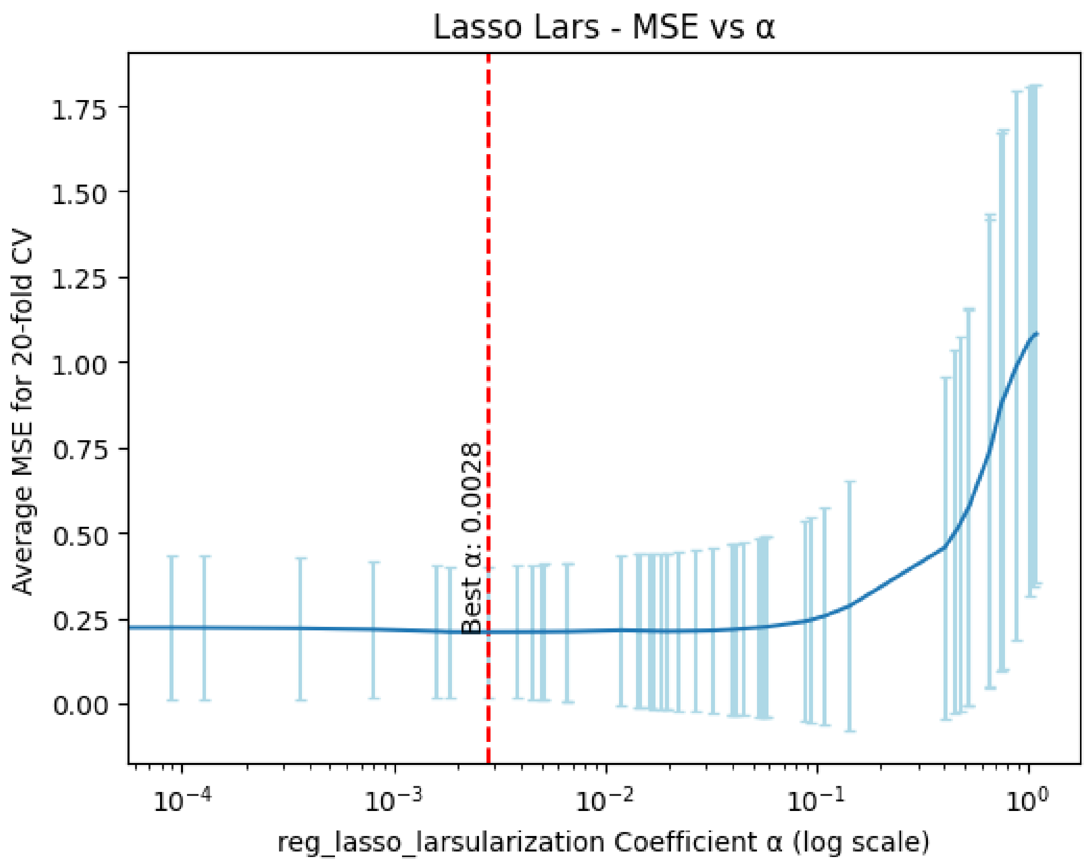
Determine the best-performing regularization coefficient value

```
[1113]: # report the best alpha and the corresponding MSE
best_alpha_lasso = reg_lasso_lars.alpha_
best_mse_lasso = reg_lasso_lars.mse_path_.mean(axis=1)[
    np.where(reg_lasso_lars.cv_alphas_ == best_alpha_lasso)
][0]
print(f"Best  : {best_alpha_lasso:.4f}\nMSE: {best_mse_lasso:.4f}")
```

```
Best  : 0.0028
MSE: 0.2089
```

Add a bar to visualize where the best-performing regularization coefficient is on the graph

```
[1114]: # Plot the MSE for each  value
plt.errorbar(
    reg_lasso_lars.cv_alphas_,  # =  values
    reg_lasso_lars.mse_path_.mean(axis=1),  # = mean of MSE for each  across␣
↪20 folds
    yerr=reg_lasso_lars.mse_path_.std(
        axis=1
    ),  # = standard deviation of MSE for each   across 20 folds
    fmt="-",
    capsize=2,
    ecolor="lightblue",
)
plt.xscale("log")
plt.title("Lasso Lars - MSE vs  ")
plt.xlabel("reg_lasso_larsularization Coefficient   (log scale)")
plt.ylabel("Average MSE for 20-fold CV")

# Highlight best alpha on plot
plt.axvline(best_alpha_lasso, color="red", linestyle="--")
plt.text(
    best_alpha_lasso,
    best_mse_lasso,
    f"Best  : {best_alpha_lasso:.4f}",
    rotation=90,
    va="bottom",
    ha="right",
);
```



2.5 Model Evaluation

Once all 3 models have been trained and tuned, we evaluate them using the test data.

2.5.1 Best Subsets

[1115]:
```python
# use best combination of features from previous step to fit the model
best_subsets_lr = linear_model.LinearRegression()
best_subsets_lr.fit(X_train[list(best_feature_combo)], y_train)

# score the model's accuracy on the test data
r2_best_subsets = best_subsets_lr.score(X_test[list(best_feature_combo)],␣
 ↪y_test)
preds_best_subsets = best_subsets_lr.predict(X_test[list(best_feature_combo)])
mse_best_subsets = sklearn.metrics.mean_squared_error(y_test,␣
 ↪preds_best_subsets)

print(
    "Best Subsets:",
    f"Features Used: {best_feature_combo}",
    f"Coefficients: {best_subsets_lr.coef_}",
    f"MSE: {mse_best_subsets:.3f}",
    f"R²: {r2_best_subsets:.3f}\n",
    sep="\n\t",
)

# show an additional summary of the model
X_train_sm = sm.add_constant(X_train[list(best_feature_combo)])
model = sm.OLS(y_train, X_train_sm).fit()
print(model.summary())
```

```
Best Subsets:
        Features Used: ('count_crosswalks', 'count_sidewalks',␣
 ↪'count_suntran_stops')
        Coefficients: [ 0.26944407 -0.13910593  0.87068815]
        MSE: 0.042
        R²: 0.959


                            OLS Regression Results
==============================================================================
Dep. Variable:                arrests   R-squared:                       0.941
Model:                            OLS   Adj. R-squared:                  0.936
Method:                 Least Squares   F-statistic:                     171.1
Date:                Tue, 10 Dec 2024   Prob (F-statistic):           8.88e-20
```

```
Time:                        16:46:23   Log-Likelihood:              -0.040984
No. Observations:                  36   AIC:                             8.082
Df Residuals:                      32   BIC:                             14.42
Df Model:                           3
Covariance Type:            nonrobust
=========================================================================================
                            coef    std err          t      P>|t|      [0.025      ␣
↪0.975]
-----------------------------------------------------------------------------------------
const                  2.776e-17      0.043   6.48e-16      1.000      -0.087      ␣
↪ 0.087
count_crosswalks          0.2694      0.060      4.517      0.000       0.148      ␣
↪ 0.391
count_sidewalks          -0.1391      0.060     -2.304      0.028      -0.262      ␣
↪-0.016
count_suntran_stops       0.8707      0.060     14.579      0.000       0.749      ␣
↪ 0.992
==============================================================================
Omnibus:                        3.206   Durbin-Watson:                   1.935
Prob(Omnibus):                  0.201   Jarque-Bera (JB):                2.088
Skew:                          -0.282   Prob(JB):                        0.352
Kurtosis:                       4.036   Cond. No.                         2.50
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

2.5.2 Ridge

[1116]:
```python
# Use best alpha value from previous step to fit the model
reg_ridge = linear_model.Ridge(alpha=best_alpha_ridge)
reg_ridge.fit(X_train, y_train)

# Score the model's accuracy on the test data
r2_ridge = reg_ridge.score(X_test, y_test)
preds_ridge = reg_ridge.predict(X_test)
mse_ridge = sklearn.metrics.mean_squared_error(y_test, preds_ridge)

print(
    "Ridge Regression:",
    f"Coefficients:{reg_ridge.coef_}",
    f"alpha = {best_alpha_ridge:.3f}",
    f"MSE = {mse_ridge:.3f}",
    f"R² = {r2_ridge:.3f}",
    sep="\n\t",
)
```

```
Ridge Regression:
        Coefficients:[ 0.02419366 -0.04124107  0.17618552  0.05782215  0.
↪27203466 -0.1039813
 0.3078223  -0.07213951  0.24599121]
        alpha = 6.251
        MSE = 0.044
        R² = 0.958
```

2.5.3 Lasso

[1117]:
```
# Use the optimal regularization coefficient from the previous step to fit the␣
↪model
reg_lasso = linear_model.Lasso(alpha=best_alpha_lasso)
reg_lasso.fit(X_train, y_train)

# Score the model's accuracy on the test data
r2_lasso = reg_lasso.score(X_test, y_test)
preds_lasso = reg_lasso.predict(X_test)
mse_lasso = sklearn.metrics.mean_squared_error(preds_lasso, y_test)

print(
    "Lasso Regression:",
    f"Coefficients:{reg_lasso.coef_}",
    f"alpha = {reg_lasso.alpha}",
    f"MSE = {mse_lasso:.3f}",
    f"R² = {r2_lasso:.3f}",
    sep="\n\t",
)
```

```
Lasso Regression:
        Coefficients:[ 0.01643655 -0.03655038  0.08999424  0.          0.3044009␣
↪ -0.17726365
 0.53693242 -0.00649942  0.26623392]
        alpha = 0.002827020252684362
        MSE = 0.015
        R² = 0.986
```

2.5.4 Compare Feature Weights for each Model

[1118]:
```
feature_names = df.columns[:-1]
model_names = ["Best Subset Regression", "Ridge Regression", "Lasso Regression"]
results_df = pd.DataFrame(columns=model_names, index=feature_names)
for i, feature in enumerate(feature_names):
    results_df.loc[feature] = [
        (
            best_subsets_lr.coef_[list(best_feature_combo).index(feature)]
            if feature in best_feature_combo
            else 0
        ),
        reg_ridge.coef_[i],
        reg_lasso.coef_[i],
    ]

# Create another table comparing metrics
metrics_df = pd.DataFrame(columns=model_names, index=["MSE", "R²"])
metrics_df.loc["MSE"] = [mse_best_subsets, mse_ridge, mse_lasso]
metrics_df.loc["R²"] = [r2_best_subsets, r2_ridge, r2_lasso]
print("Model performance metrics:\n")
print(metrics_df)

print("\n\nModel Feature weights:")
results_df
```

```
Model performance metrics:


    Best Subset Regression Ridge Regression Lasso Regression
MSE               0.041975         0.043641         0.014971
R²                0.959451         0.957841         0.985537



Model Feature weights:
```

[1118]:

```
                                                   Best Subset Regression  \
distance_to_nearest_fire_stations                                       0
distance_to_nearest_landfills                                           0
count_bicycle_boulevards                                                0
count_bridges                                                           0
count_crosswalks                                                 0.269444
count_sidewalks                                                 -0.139106
count_suntran_stops                                              0.870688
distance_to_nearest_streetcar_stops_and_distanc…                      0
count_major_streets_and_count_streetlights_PCA                          0

                                                   Ridge Regression  \
distance_to_nearest_fire_stations                          0.024194
distance_to_nearest_landfills                             -0.041241
count_bicycle_boulevards                                   0.176186
```

```
count_bridges                                           0.057822
count_crosswalks                                        0.272035
count_sidewalks                                        -0.103981
count_suntran_stops                                     0.307822
distance_to_nearest_streetcar_stops_and_distanc…      -0.07214
count_major_streets_and_count_streetlights_PCA          0.245991

                                                Lasso Regression
distance_to_nearest_fire_stations                       0.016437
distance_to_nearest_landfills                           -0.03655
count_bicycle_boulevards                                0.089994
count_bridges                                                0.0
count_crosswalks                                        0.304401
count_sidewalks                                        -0.177264
count_suntran_stops                                     0.536932
distance_to_nearest_streetcar_stops_and_distanc…     -0.006499
count_major_streets_and_count_streetlights_PCA          0.266234
```

2.6 7—Discussion and Visualization

3 Model 2—Subsection-Level Sociodemographic Features Prediction from Crime Density Features

3.1 Exploratory Data Analysis and Data Visualization

Start by collecting the education dataset. The data is based on neighborhood partitions, which are relatively sparse and do not cleanly span some grid from which we can sample. It possible to randomly sample subsections from the area defined by the union of all neighborhoods, as this may lead to better generalization.

For now, we will simply use the neighborhoods as the subsections (data items/rows) and see how the model performs.

3.1.1 Load the Dataset

[1119]:
```
education_data = load_dataset("education")
education_data.head()  # Preview the data
```

[1119]:
```
   OBJECTID              NAME  WARD     DATASOURCE ID sourceCountry  \
0         1        A Mountain     1  NEIGHBORHOODS  0            US
1         2          Adelanto     3  NEIGHBORHOODS  1            US
2         3  Alvernon Heights     5  NEIGHBORHOODS  2            US
3         4             Amphi     3  NEIGHBORHOODS  3            US
4         5       Armory Park     6  NEIGHBORHOODS  4            US

   ENRICH_FID                 aggregationMethod  \
0           1  BlockApportionment:US.BlockGroups
1           2  BlockApportionment:US.BlockGroups
```

```
2           3  BlockApportionment:US.BlockGroups
3           4  BlockApportionment:US.BlockGroups
4           5  BlockApportionment:US.BlockGroups

   populationToPolygonSizeRating  apportionmentConfidence  …  NOHS_CY  \
0                          2.191                    2.576  …      382
1                          2.191                    2.576  …       54
2                          2.191                    2.576  …        3
3                          2.191                    2.576  …      405
4                          2.191                    2.576  …       65

   SOMEHS_CY  HSGRAD_CY  GED_CY  SMCOLL_CY  ASSCDEG_CY  BACHDEG_CY  \
0        291        323     187        498         131          88
1         34         55      17          9          10          10
2         21         34      17         69          12          26
3        697        954     232       1260         395         432
4         70        138      73        287          71         404

   GRADDEG_CY  EDUCBASECY                                           geometry
0          13        1913  POLYGON ((-111.0076 32.20691, -111.00673 32.20…
1          11         200  POLYGON ((-110.98426 32.24578, -110.98222 32.2…
2           7         189  POLYGON ((-110.90819 32.20294, -110.90818 32.2…
3         213        4588  POLYGON ((-110.97768 32.27921, -110.97731 32.2…
4         449        1557  POLYGON ((-110.97102 32.22046, -110.97112 32.2…

[5 rows x 21 columns]
```

3.1.2 Clean the Data

Remove any neighborhoods with missing data (from columnds relevant to the model)

[1120]:
```
# Drop any rows with null values in the relevant education-level columns
relevant_cols = [
    "NOHS_CY",
    "SOMEHS_CY",
    "HSGRAD_CY",
    "GED_CY",
    "SMCOLL_CY",
    "ASSCDEG_CY",
    "BACHDEG_CY",
    "GRADDEG_CY",
]
education_data = education_data.dropna(subset=relevant_cols)

# Drop any rows with null values in the geometry (location) column
education_data = education_data.dropna(subset=["geometry"])
```

3.1.3 Explore the Dataset

To get an idea of how best to construct features for the purpose of predicting sociodemographic features from crime density, we will visualize the dataset.

Visualize the Neighborhoods on the Map Since our data items/rows will (tentatively) be neighborhoods, we start by defining a function to visualize the neighborhoods on the map.

[1121]:
```python
def visualize_neighborhood_boundaries(
    gdf: gpd.GeoDataFrame, folium_map=None, fill_opacity=None
):
    """
    Visualizes neighborhood boundaries as polygons on a Folium map.

    Params:
    geojson_data: dict - GeoJSON data with neighborhood boundaries.
    folium_map: folium.Map - The Folium map to add boundaries to.
    fill_opacity: float - Opacity of the filled neighborhood polygons.
    line_color: str - Color of the polygon borders.

    Returns:
    folium.Map: The updated Folium map with neighborhood boundaries.
    """
    if folium_map is None:
        folium_map = folium.Map(
            location=config["tucson_center_coordinates"], zoom_start=12
        )

    # Reproject to WGS84 if not already
    if gdf.crs != "EPSG:4326":
        gdf = gdf.to_crs("EPSG:4326")

    # Add each neighborhood boundary as a polygon without fill
    for _, row in gdf.iterrows():
        geometry = row.geometry
        if geometry.geom_type == "Polygon":
            # Add a single polygon
            folium.PolyLine(
                locations=[[y, x] for x, y in geometry.exterior.coords],
                color="#9D00FF",
                fill=fill_opacity is not None,
                fill_opacity=fill_opacity,
            ).add_to(folium_map)
        elif geometry.geom_type == "MultiPolygon":
            # Add each part of a MultiPolygon
            for polygon in geometry.geoms:
                folium.PolyLine(
                    locations=[[y, x] for x, y in polygon.exterior.coords],
                    color="#9D00FF",
                    fill=fill_opacity is not None,
                    fill_opacity=fill_opacity,
                ).add_to(folium_map)

    return folium_map
```

[1122]:
```
visualize_neighborhood_boundaries(education_data, fill_opacity=0.2)
```

[1122]:
```
<folium.folium.Map at 0x7de76079e8a0>
```

Visualize the Distribution of the Target Variable (Education Level) across Neighborhoods In the dataset, the education levels per neighborhood are represented by 6 categories:

Column Name	Description
GRADDEG_CY	Graduate degree
BACHDEG_CY	Bachelor's degree
ASSCDEG_CY	Associate degree
SMCOLL_CY	Some college
GED_CY	GED
HSGRAD_CY	High school graduate
SOMEHS_CY	Some high school
NOHS_CY	No high school

The values in these columns are nominal counts of the number of people in each neighborhood that fall into each category.

To visualize the data, we want to first combine the columnds into a single continuous variable that represents the education level of each neighborhood. To accomplish this, we assign a weight to each category and sum the weighted counts. The weight will represent the ordinal value of the education level such that higher weights correspond to higher education levels:

Column Name	Weight	Description
GRADDEG_CY	6	Graduate degree
BACHDEG_CY	5	Bachelor's degree
ASSCDEG_CY	4	Associate degree
SMCOLL_CY	3	Some college
GED_CY	2	GED
HSGRAD_CY	1.5	High school graduate
SOMEHS_CY	1	Some high school
NOHS_CY	0.5	No high school

Define a function that sums the weighted counts to create a single continuous variable and then visualize the distribution of this variable across the neighborhoods:

[1123]:
```
# Define weights for each education level
edu_lvl_weights = {
    "GRADDEG_CY": 6,  # Graduate degree
    "BACHDEG_CY": 5,  # Bachelor's degree
    "ASSCDEG_CY": 4,  # Associate degree
    "SMCOLL_CY": 3,  # Some college
    "GED_CY": 2,  # GED
    "HSGRAD_CY": 1.5,  # High school graduate
    "SOMEHS_CY": 1,  # Some high school
    "NOHS_CY": 0.5,  # No high school
}
```

Create a column representing the total education counts weighted by the education level.

[1124]:
```
education_data["weighted_education"] = sum(
    education_data[column] * weight for column, weight in edu_lvl_weights.
 ↪items()
)
```

Now, visualize the distribution of the total education level across the neighborhoods.

To do so, define a function that visualizes the distribution of the target variable across the neighborhoods:

[1125]:
```
def visualize_neighborhood_feature(
    gdf: gpd.GeoDataFrame, feature, zoom_start=12, color_scale="Blues"
):
    """
    Visualizes neighborhoods with a color gradient representing weighted␣
 ↪education levels.

    Params:
    gdf: GeoDataFrame - GeoDataFrame with features including education levels.
    zoom_start: int - Initial zoom level of the map.
    color_scale: str - Name of a Matplotlib colormap for the gradient.

    Returns:
    folium.Map: The Folium map with filled neighborhoods based on weighted␣
 ↪education levels.
    """
    # Normalize weighted values for color mapping
    min_val = gdf[feature].min()
    max_val = gdf[feature].max()
    norm = matplotlib.colors.Normalize(
        vmin=gdf[feature].min(), vmax=gdf[feature].max()
    )
    colormap = cm.ScalarMappable(norm=norm, cmap=color_scale)
    # Create a Folium map
    center_coords = [gdf.geometry.centroid.y.mean(), gdf.geometry.centroid.x.
↪mean()]
    folium_map = folium.Map(location=center_coords, zoom_start=zoom_start)

    # Add polygons with color fill
    for _, row in gdf.iterrows():
        geometry = row.geometry
        weighted_value = row[feature]
        fill_color = matplotlib.colors.to_hex(colormap.to_rgba(weighted_value))

        if geometry.geom_type == "Polygon":
            # Add single polygon
            folium.Polygon(
                locations=[[y, x] for x, y in geometry.exterior.coords],
                color="black",
                fill=True,
                fill_color=fill_color,
                fill_opacity=0.7,
            ).add_to(folium_map)
        elif geometry.geom_type == "MultiPolygon":
            # Add each part of a MultiPolygon
            for polygon in geometry.geoms:
                folium.Polygon(
                    locations=[[y, x] for x, y in polygon.exterior.coords],
                    color="black",
                    fill=True,
                    fill_color=fill_color,
                    fill_opacity=0.7,
                ).add_to(folium_map)

    # Add a color scale legend
    colormap = linear.Blues_09.scale(min_val, max_val)  # Create a colormap␣
↪using branca
    colormap.caption = f"{feature.title()} Level"
    colormap.add_to(folium_map)

    return folium_map
```

[1126]:
```
visualize_neighborhood_boundaries(
    education_data, visualize_neighborhood_feature(education_data,␣
↪"weighted_education")
)
```

[1126]:
```
<folium.folium.Map at 0x7de75e7d1370>
```

3.1.4 Normalize Education Level by Population Size

Upon inspection of the heatmap, we notice that the lack of normalization by population size may be causing some neighborhoods to appear more educated than they actually are and generally creating too wide of a spread in the education level values.

To resolve this, we will normalize the education level by the population size of each neighborhood:

[1127]:
```
# Normalize the weighted education scores by population size
education_data["normalized_weighted_education"] = education_data[
    "weighted_education"
] / education_data[edu_lvl_weights.keys()].sum(axis=1)

# Preview the new columns
education_data[["weighted_education", "normalized_weighted_education"]].head()
```

[1127]:
```
   weighted_education  normalized_weighted_education
0              3876.5                       2.026398
1               360.5                       1.802500
2               534.5                       2.828042
3             11592.5                       2.526700
4              6314.5                       4.055556
```

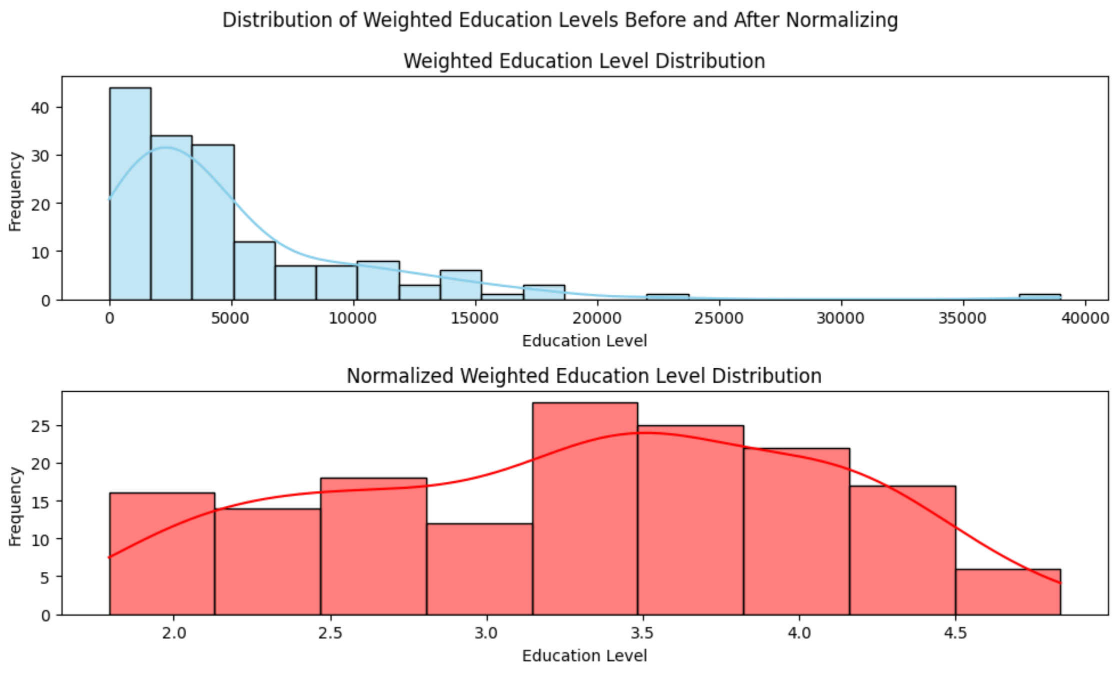
Visualize the Normalized Education Levels

[1128]:
```
visualize_neighborhood_boundaries(
    education_data,
    visualize_neighborhood_feature(education_data,␣
 ↪"normalized_weighted_education"),
)
```

[1128]:
```
<folium.folium.Map at 0x7de792d2a480>
```

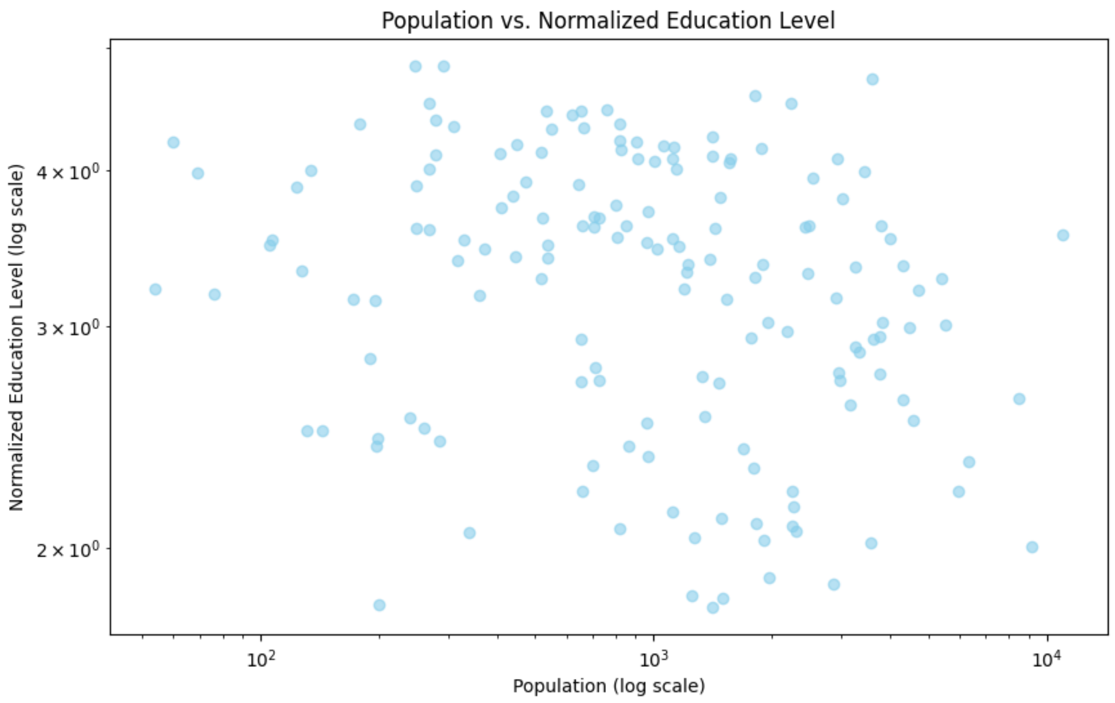
It appears that normalizing did not change the heatmap at least by visual inspection. To verify, plot the distribution of the normalized education levels and non-normalized education levels side-by-side.

[1129]:
```
# plot the distribution of the normalized education levels and non-normalized␣
 ↪education levels side-by-side.
fig, ax = plt.subplots(2, 1, figsize=(10, 6))
fig.suptitle("Distribution of Weighted Education Levels Before and After␣
 ↪Normalizing")

sns.histplot(education_data["weighted_education"], kde=True, ax=ax[0],␣
 ↪color="skyblue")
ax[0].set_title("Weighted Education Level Distribution")
ax[0].set_xlabel("Education Level")
ax[0].set_ylabel("Frequency")

sns.histplot(
    education_data["normalized_weighted_education"], kde=True, ax=ax[1],␣
  ↪color="red"
)
ax[1].set_title("Normalized Weighted Education Level Distribution")
ax[1].set_xlabel("Education Level")
ax[1].set_ylabel("Frequency")

plt.tight_layout()
```



Plot Log-Log Distribution of the Normalized Education Levels

[1130]:
```
# Plot the log-log relationship between the population size of neighborhoods␣
  ↪and their normalized education levels
plt.figure(figsize=(10, 6))
plt.scatter(
    education_data[edu_lvl_weights.keys()].sum(axis=1), # As before, sum the␣
  ↪nominal education counts to infer population
    education_data["normalized_weighted_education"],
    alpha=0.6,
    color="skyblue",
)

# Add line of best fit, but can't use `linregress` module since this is a power␣
  ↪law relationship
plt.plot(
    np.unique(education_data[edu_lvl_weights.keys()].sum(axis=1)),
    np.poly1d(
        np.polyfit(
            education_data[edu_lvl_weights.keys()].sum(axis=1),
            education_data["normalized_weighted_education"],
            1,
        )
    )(np.unique(education_data[edu_lvl_weights.keys()].sum(axis=1))),
    color="red",
)


plt.xscale("log")
plt.yscale("log")
plt.title("Population vs. Normalized Education Level")
plt.xlabel("Population (log scale)")
plt.ylabel("Normalized Education Level (log scale)")
plt.show()
```



The plot confirms that the distribution is not a power law distribution, which is a good sign for the model.

3.1.5 Transform Education Level into a Categorical Variable

We explore two possible ways of casting the education level as a categorical variable:

1. Taking the most common education level in each neighborhood
2. Averaging the weighted education levels and choosest the closest weight (from the weights defined above that map education levels to ordinal values)

This corresponds to the mode or mean of the education level in each neighborhood, respectively.

Create both options and compare:

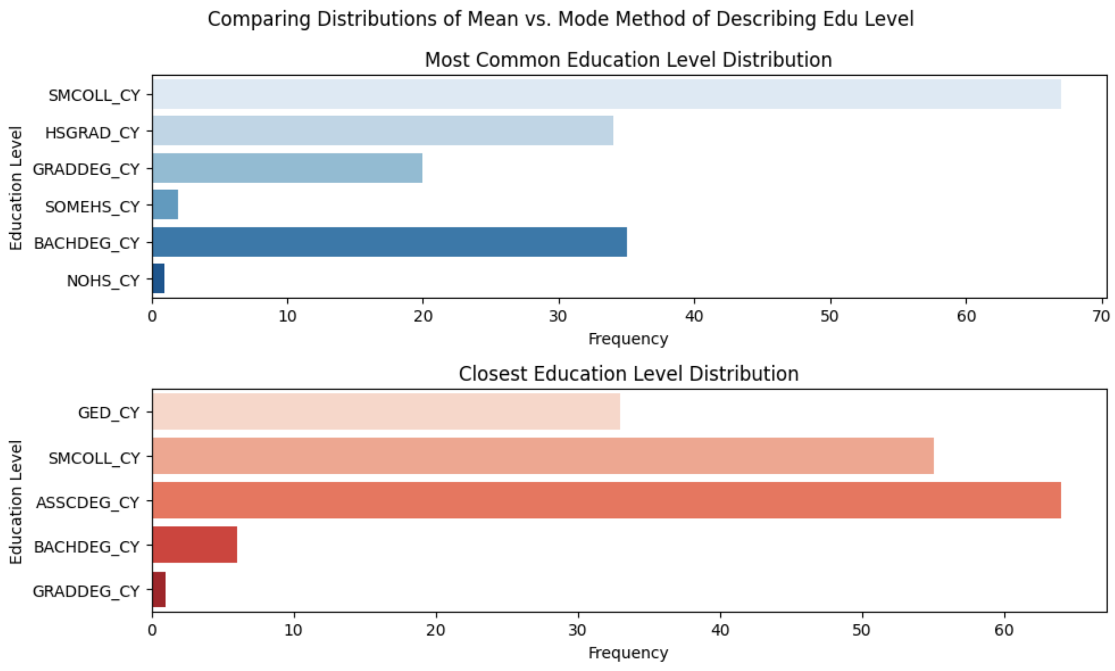
[1131]:
```
# Create a column representing the most common education level in each␣
↪neighborhood
education_data["most_common_education"] = education_data[edu_lvl_weights.
↪keys()].idxmax(
    axis=1
)

# Extract weights and their corresponding labels
weight_to_label = {v: k for k, v in edu_lvl_weights.items()}
weights = list(weight_to_label.keys())

# Create column representing closest ordinal value to the normalized education␣
↪level
education_data["closest_education_level"] = [
    weight_to_label[min(weights, key=lambda w: abs(w - value))]
    for value in education_data["normalized_weighted_education"]
]

# Preview the newly added columns
education_data[["most_common_education", "closest_education_level"]].head()

print(education_data["normalized_weighted_education"].describe())
```

```
count    158.000000
mean       3.291363
std        0.781929
min        1.792761
25%        2.646656
50%        3.373859
75%        3.967775
max        4.837329
Name: normalized_weighted_education, dtype: float64
```

The preview indicates that using the most common value creates a wider range of values. Verify by plotting and comparing the distributions of the two methods:

[1132]:
```
# Plot and compare side-by-side the distributions of most_common_education vs␣
↪closest_education_level
fig, ax = plt.subplots(2, 1, figsize=(10, 6))
```

```
fig.suptitle("Comparing Distributions of Mean vs. Mode Method of Describing Edu␣
↪Level")

sns.countplot(education_data["most_common_education"], ax=ax[0],␣
↪palette="Blues")
ax[0].set_title("Most Common Education Level Distribution")
ax[0].set_xlabel("Frequency")
ax[0].set_ylabel("Education Level")

sns.countplot(education_data["closest_education_level"], ax=ax[1],␣
↪palette="Reds")
ax[1].set_title("Closest Education Level Distribution")
ax[1].set_xlabel("Frequency")
ax[1].set_ylabel("Education Level")

plt.tight_layout()
```



Since both distributions are manageable, we will use the one corresponding to mean education level for the model as it is more informative.

3.1.6 Create Crime Density Features per Neighborhood

To create the input features for the model, we will use the crime density features from the first model. However, we will have to re-traverse the `arrests` dataset and sum up the number of arrests in each neighborhood.

Count the Number of Arrests in Each Neighborhood

[1133]:
```
education_data["arrests"] = [
    count_objects_in_subsection(
        load_dataset("arrests"), subsection, "arrests", use_polygon=True
    )
    for subsection in education_data.geometry
]

education_data["arrests"]  # Preview the new column
```

[1133]:
```
0       230
1        21
2        44
3      1994
4       278
       …
154      27
155      16
156    2559
157      65
158     192
Name: arrests, Length: 159, dtype: int64
```

Calculate the Proportion of Felonies vs. Misdemeanors in Each Neighborhood The proportion of felonies vs. misdemeanors in each neighborhood can represent the severity of the crimes in each neighborhood.

[1134]:
```
# Add felony counts
education_data["felony_count"] = [
    count_objects_in_subsection(
        load_dataset("arrests"),
        subsection,
        "arrests",
        use_polygon=True,
        condition=lambda row: row["fel_misd"] == "F",
    )
    for subsection in education_data.geometry
]

# Add misdemeanor counts
education_data["misdemeanor_count"] = [
    count_objects_in_subsection(
        load_dataset("arrests"),
        subsection,
        "arrests",
        use_polygon=True,
        condition=lambda row: row["fel_misd"] == "M",
    )
    for subsection in education_data.geometry
]

# Calculate the proportion of felonies
education_data["prop_felonies"] = education_data["felony_count"] / (
    education_data["felony_count"] + education_data["misdemeanor_count"]
)

# If any of the proportion values are NaN (as a result of felony_count = 0),␣
 ↪replace with 0
education_data["prop_felonies"] = education_data["prop_felonies"].fillna(0)

# Preview the updated DataFrame
education_data[["felony_count", "misdemeanor_count", "prop_felonies"]]

# print(education_data[['prop_felonies', 'arrests']].describe())
print(education_data[['prop_felonies', 'arrests']])
```

```
     prop_felonies  arrests
0         0.376682      230
1         0.050000       21
2         0.105263       44
3         0.256052     1994
4         0.137931      278
..             ...      ...
154       0.458333       27
155       0.375000       16
156       0.160622     2559
157       0.066667       65
158       0.299435      192

[159 rows x 2 columns]
```

Adjust Crime Density Per-Capita

[1135]:
```
# # Adjust crime density per-capita
education_data["arrests"] = education_data["arrests"] / education_data[
    edu_lvl_weights.keys()
].sum(axis=1)

# # Report if any NaN or infinite values were added as a result of adjustment
# print(
#     "NaN or infinite values in arrests column:",
#     education_data["arrests"].isnull().sum(),
#     education_data["arrests"].isna().sum(),
# )
# # Preview the updated DataFrame
education_data["arrests"]
```

[1135]:
```
0        0.120230
1        0.105000
2        0.232804
3        0.434612
4        0.178548
           …
154      0.103448
155      0.043011
156     20.804878
157      0.451389
158      0.050646
Name: arrests, Length: 159, dtype: float64
```

Test for Extreme Outliers During EDA we noticed some neighborhoods having extreme outlier values for the crime density features (even after controlling for population size). We will test for these outliers and remove them if necessary.

[1136]:
```
# Identify outliers in the arrests column
Q1 = education_data["arrests"].quantile(0.25)
Q3 = education_data["arrests"].quantile(0.75)
IQR = Q3 - Q1

# Report the number of outliers and details about each one
outliers = education_data[
    (education_data["arrests"] < Q1 - 1.5 * IQR)
    | (education_data["arrests"] > Q3 + 1.5 * IQR)
]

education_data[education_data["NAME"] == "San Ignacio Yaqui"]
print("Outliers:")
outliers[
    [
        "NAME",
        "arrests",
        "prop_felonies",
        "closest_education_level",
        "most_common_education",
    ]
].head()
```

```
Outliers:
```

[1136]:
```
                            NAME   arrests  prop_felonies  \
52                    Iron Horse  0.764516       0.174157
80                     Pie Allen  2.763052       0.099822
100  Santa Rita Park - West Ochoa  0.557143       0.154993
115              West University  0.680102       0.113636
121                    Millville  1.215385       0.136150

    closest_education_level most_common_education
52               ASSCDEG_CY            BACHDEG_CY
80               ASSCDEG_CY            BACHDEG_CY
100               SMCOLL_CY             HSGRAD_CY
115              ASSCDEG_CY            BACHDEG_CY
121               SMCOLL_CY             SMCOLL_CY
```

Remove any extreme outliers from the dataset:

[1137]:
```
#  Filter out outliers
education_data = education_data[
    (education_data["arrests"] >= Q1 - 1.5 * IQR)
    & (education_data["arrests"] <= Q3 + 1.5 * IQR)
]
```

3.1.7 Normalize the Data

Since we are using SVM, we should normalize the continuous features.

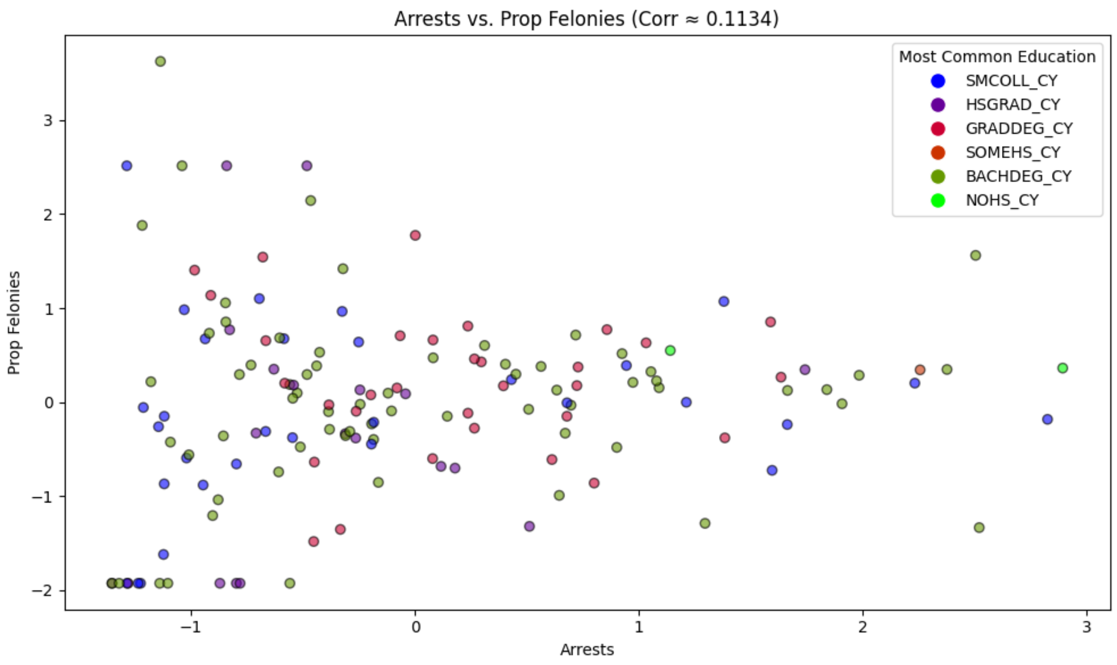
[1138]:
```
scaler = StandardScaler()

# Scale the relevant columns
education_data[["arrests", "prop_felonies"]] = scaler.fit_transform(
    education_data[["arrests", "prop_felonies"]]
)

# Preview the scaled columns
education_data[["arrests", "prop_felonies"]].head()
```

[1138]:
```
    arrests  prop_felonies
0 -0.320805       1.418329
1 -0.451504      -1.482012
2  0.645274      -0.991375
3  2.377128       0.347353
4  0.179667      -0.701344
```

3.1.8 Visualize Separation into Education Level Categories from Crime Density

Get an initial idea of how well we can separate the neighborhoods into the education level categories using the crime-related features.

```
[1139]: import matplotlib.pyplot as plt
import numpy as np

# Define input features and target class
feature_x = "arrests"  # Total arrests
feature_y = "prop_felonies"  # Proportion of felonies
target_class = "most_common_education"

# Create a scatter plot
fig, ax = plt.subplots(figsize=(10, 6))

# Scatter plot with color encoding by target class
scatter = ax.scatter(
    education_data[feature_x],
    education_data[feature_y],
    c=education_data[target_class]
    .astype("category")
    .cat.codes,  # Encode classes as integers
    cmap="brg",
    alpha=0.6,
    edgecolor="k",
)

# Calculate correlation coefficient
corr = np.corrcoef(education_data[feature_x], education_data[feature_y])[0, 1]

# Set axis labels, title, and colorbar
ax.set_xlabel(feature_x.replace("_", " ").title())
ax.set_ylabel(feature_y.replace("_", " ").title())
ax.set_title(
    f"{feature_x.replace('_', ' ').title()} vs. {feature_y.replace('_', ' ').
 ↪title()} (Corr  {corr:.4f})"
)

# Create a legend for the target classes
unique_classes = education_data[target_class].unique()
colors = scatter.cmap(np.linspace(0, 1, len(unique_classes)))

handles = [
    plt.Line2D(
        [0],
        [0],
        marker="o",
        color="w",
        label=cls,
        markersize=10,
        markerfacecolor=color,
    )
    for cls, color in zip(unique_classes, colors)
]
ax.legend(handles=handles, title="Most Common Education", loc="best")

# Show plot
plt.tight_layout()
plt.show()
```



3.1.9 Split Data into Training and Testing Sets

[1140]:
```
X_train, X_test = train_test_split(
    education_data[["arrests", "prop_felonies"]],
    test_size=config["hp"]["test_size"],
    random_state=42,
)

y_train, y_test = train_test_split(
    education_data["closest_education_level"],
    test_size=config["hp"]["test_size"],
    random_state=42,
)

X_train.head() # Preview the training inputs
```

[1140]:

```
      arrests  prop_felonies
15  -1.119285      -0.868994
131 -0.481344       0.293626
11  -0.119376       0.095452
133 -0.383589      -0.028510
51  -1.285011      -1.925921
```

3.2 Train the Model

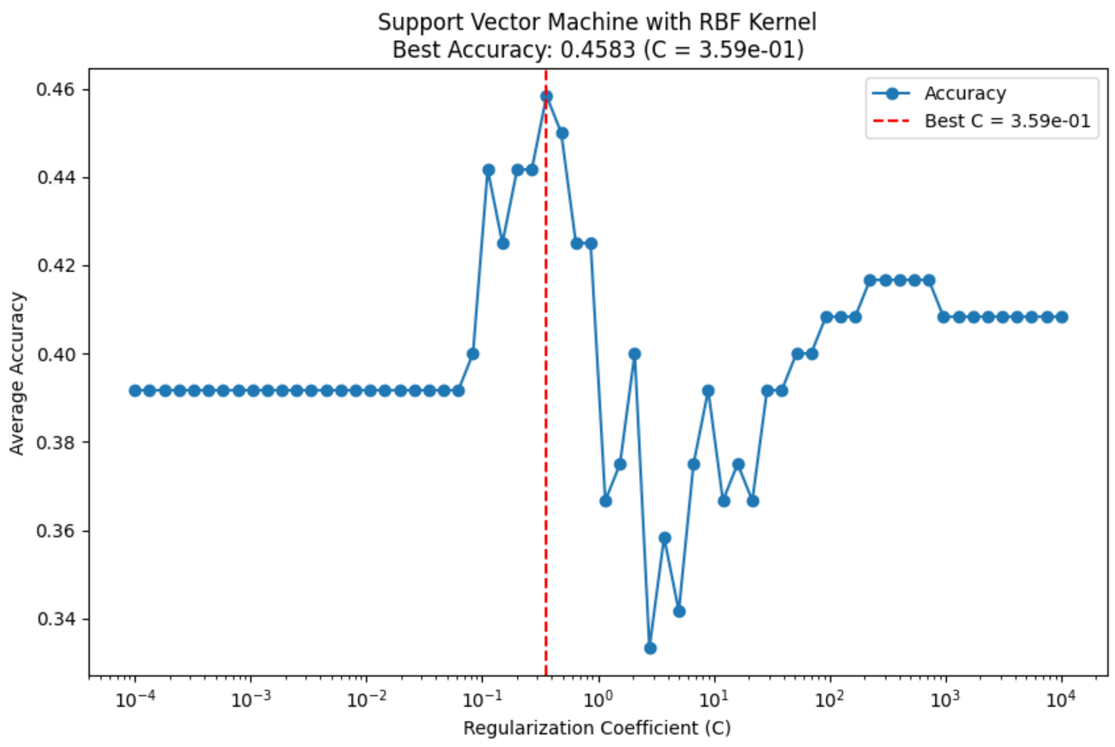
3.2.1 Hyperparameter Tuning

Choose the best regularization parameter for the SVM model.

[1141]:

```python
reg_coeff_domain = np.logspace(-4, 4, 64)  # Create a range of values

svm_scores_per_coeff = np.zeros(
    len(reg_coeff_domain)
)  # Initialize an array to store scores

for idx, coeff in enumerate(reg_coeff_domain):
    # Initialize the SVM classifier with the current C value
    svm = SVC(
        C=coeff,
        kernel="sigmoid",  # TODO: test other kernels
    )

    # Fit the classifier and calculate the score
    svm_scores_per_coeff[idx] = cross_val_score(
        svm,
        X_train,
        y_train,
        cv=config["hp"]["cross_val_folds"]["svm"],
        scoring=config["hp"]["scoring_metric"]["svm"],
    ).mean()

# Find the best accuracy and its corresponding C value
best_accuracy_idx = np.argmax(svm_scores_per_coeff)
best_score_svm = svm_scores_per_coeff[best_accuracy_idx]
best_coeff_svm = reg_coeff_domain[best_accuracy_idx]

# Plot average accuracy versus regularization coefficient
plt.figure(figsize=(10, 6))
plt.plot(
    reg_coeff_domain, svm_scores_per_coeff, marker="o", linestyle="-",␣
 ↪label="Accuracy"
)
plt.xscale("log")
plt.xlabel("Regularization Coefficient (C)")
plt.ylabel("Average Accuracy")
plt.title(
    f"Support Vector Machine with RBF Kernel\nBest Accuracy: {best_score_svm:.
 ↪4f} (C = {best_coeff_svm:.2e})"
)
plt.axvline(
    x=best_coeff_svm,
    color="red",
    linestyle="--",
    label=f"Best C = {best_coeff_svm:.2e}",
)
plt.legend()

# Report the maximum accuracy and best coefficient
print(
    f"Best Accuracy:                    {best_score_svm:.4f}",
    f"Using Regularization Coefficient: {best_coeff_svm:.2e}",
    sep="\n",
)
```

```
Best Accuracy:                    0.4583
Using Regularization Coefficient: 3.59e-01
```



3.3 Evaluate Model

3.3.1 Determine Baseline Accuracy

Set the baseline for our dataset using a Majority Classifier.

Determine the proportion of the majority class in the dataset:

[1142]:
```
prop_majority_class = y_train.value_counts(normalize=True).max()
print(f"Proportion of Majority Class: {prop_majority_class:.4f}")
```

```
Proportion of Majority Class: 0.3917
```

Define a function to visualize the evaluation metrics:

[1143]:
```
def display_model_evaluation(
    y_true, y_pred, features, model_type, training_accuracy,␣
 ↪regularization_coeff
):
    """
    Displays a consolidated panel with model summary and confusion matrix/
 ↪classification report side-by-side using rich.
    """
    console = Console()
    class_names = [str(cls) for cls in np.unique(y_true)]
    feature_list = ", ".join(features)
    model_summary = (
        f"[bold]Model Type:[/bold]                    {model_type}\n"
        f"[bold]Features Used:[/bold]                 {feature_list}\n"
        f"[bold]Regularization Coefficient:[/bold]    {regularization_coeff:.
 ↪2e}\n"
        f"[bold]Accuracy on Training Data:[/bold]     {training_accuracy:.3f}\n"
        f"[bold]Accuracy of Baseline Model:[/bold]    {prop_majority_class:.3f}"
        f"\n[bold yellow]Note:[/bold yellow] Above metrics are based on CV on␣
 ↪training data. Below metrics are based on evaluation on test data."
    )

    # Classification Report
    class_report = classification_report(
        y_true, y_pred, target_names=class_names, digits=2
    )
    classification_report_section = Panel(
        f"[bold green]Classification Report[/bold green]\n\n{class_report}",
        title="Classification Report",
        expand=True,
    )

    # Confusion Matrix as Formatted String
    conf_matrix = confusion_matrix(y_true, y_pred)
    column_width = max(len(cls) for cls in class_names) + 2
    header = "".join(f"{cls:<{column_width}}" for cls in [""] + class_names)
    rows = "\n".join(
        f"{class_names[i]:<{column_width}}"
        + "".join(f"{val:<{column_width}}" for val in row)
        for i, row in enumerate(conf_matrix)
    )
    conf_matrix_section = Panel(
        f"[bold cyan]Confusion Matrix[/bold cyan]\n\n{header}\n{rows}",
        title="Confusion Matrix",
        expand=True,
    )

    # Combine confusion matrix and classification report side-by-side
    side_by_side_content = Columns([conf_matrix_section,␣
↪classification_report_section])

    # Combine everything into a single panel
    console.print(Panel(model_summary, title="Model Summary", expand=True))
    console.print(side_by_side_content)
```

Visualize the evaluation metrics of the classifier model:

[1144]:
```
best_svm = SVC(
    C=best_coeff_svm,
    kernel="linear",
)
best_svm.fit(X_train, y_train)

# Predict on the test data
y_pred_svm = best_svm.predict(X_test)

display_model_evaluation(
    y_test,
    y_pred_svm,
    ["arrests", "prop_felonies"],
    "Support Vector Machine (Linear Kernel)",
    best_score_svm,
    best_coeff_svm,
)
```

```
                            Model Summary
 Model Type:                  Support Vector Machine (Linear Kernel)          ␣
↪
 Features Used:               arrests, prop_felonies                          ␣
↪
 Regularization Coefficient:  3.59e-01                                        ␣
↪
```

```
Accuracy on Training Data:   0.458                                                    ␣
↪
Accuracy of Baseline Model:  0.392                                                    ␣
↪
Note: Above metrics are based on CV on training data. Below metrics are based␣
↪on evaluation on test data.



            Confusion Matrix
Confusion Matrix

            ASSCDEG_CY  BACHDEG_CY  GED_CY      SMCOLL_CY
ASSCDEG_CY  7           0           0           6
BACHDEG_CY  1           0           0           0
GED_CY      2           0           0           2
SMCOLL_CY   5           0           0           8

          Classification Report
Classification Report

              precision    recall  f1-score   support

  ASSCDEG_CY       0.47      0.54      0.50        13
  BACHDEG_CY       0.00      0.00      0.00         1
      GED_CY       0.00      0.00      0.00         4
   SMCOLL_CY       0.50      0.62      0.55        13

    accuracy                           0.48        31
   macro avg       0.24      0.29      0.26        31
weighted avg       0.41      0.48      0.44        31
```