

Practical Machine Learning and Data Exploration

Project 1: GitHub Repository Metrics Data Exploration

Student Name: Ibrahim Mohammed Alsidrani

Student ID: 262000987

Project Objectives

The main objectives of this project are:

1. Handle missing values in the GitHub repository metrics dataset.
2. Detect and handle outliers in continuous repository metrics.
3. Perform data visualization to explore programming languages, repository popularity, activity, and feature relationships.

Dataset File: github_repos.csv

Dataset Source: Kaggle - GitHub Repository Metrics Dataset (5K+ Repos)

1. Importing Required Libraries

The required Python libraries are imported for data manipulation, numerical operations, and visualization.

```
In [3]: # Import NumPy for numerical operations.
import numpy as np

# Import Pandas for loading, cleaning, and analyzing tabular data.
import pandas as pd

# Import Matplotlib for data visualization.
import matplotlib.pyplot as plt

# Import warnings to hide non-critical warning messages.
import warnings
warnings.filterwarnings("ignore")

# Display all DataFrame columns.
pd.set_option("display.max_columns", None)

print("Libraries imported successfully.")
```

Libraries imported successfully.

2. Loading the Dataset

The GitHub repository metrics dataset is loaded from the local file github_repos.csv .

```
In [5]: # Store the local dataset filename.
data_file = "github_repos.csv"

# Load the CSV file into a Pandas DataFrame.
df = pd.read_csv(data_file)

# Confirm that the dataset was loaded successfully.
print("Dataset loaded successfully.")
print("Dataset shape:", df.shape)
```

Dataset loaded successfully.

Dataset shape: (5500, 29)

3. Dataset Understanding

This section examines the first records, dataset dimensions, column names, data types, and duplicated rows.

```
In [7]: # Display the first five rows.
df.head()
```

Out[7]:

	repo_name	language	description	stars	forks	watchers	open_issues	closed_issues	open_pull_requests	merged_pull_requests	contributors	commits	releases	license	topics	created_date	last_commit_date	readme
0	serve-ml	Ruby	server framework for building authentication	13	3	0	12	99	3	36	5	200	2	GPL-3.0	api automation devops rails deep-learning	2024-09-14	2024-12-19	
1	easy-link	TypeScript	Scalable pipeline for database management	32	5	2	2	19	0	10	6	1167	24	BSD-3	angular react web api	2022-12-18	2024-12-26	
2	go-graph-agent	Kotlin	lightweight dashboard built with Kotlin	2	0	0	0	10	0	4	2	10	0	NaN	mobile desktop	2024-12-13	2024-12-15	
3	turbo-core-forge	C++	engine framework for building database management	109	17	6	0	1	0	0	4	195	10	MIT	learning tool graphql serverless	2023-12-09	2024-12-22	
4	deep-graph-dash	Python	dashboard framework for building database mana...	21	10	0	13	31	4	15	9	2187	14	MIT	automation data-science api rag	2018-01-04	2024-11-20	

```
In [8]: # Display the first five rows in transposed format for easier reading.
df.head().T
```

Out[8]:

	0	1	2	3	4
repo_name	serve-ml	easy-link	go-graph-agent	turbo-core-forge	deep-graph-dash
language	Ruby	TypeScript	Kotlin	C++	Python
description	server framework for building authentication	Scalable pipeline for database management	lightweight dashboard built with Kotlin	engine framework for building database management	dashboard framework for building database mana...
stars	13	32	2	109	21
forks	3	5	0	17	10
watchers	0	2	0	6	0
open_issues	12	2	0	0	13
closed_issues	99	19	10	1	31
open_pull_requests	3	0	0	0	4
merged_pull_requests	36	10	4	0	15
contributors	5	6	2	4	9
commits	200	1167	10	195	2187
releases	2	24	0	10	14
license	GPL-3.0	BSD-3	NaN	MIT	MIT
topics	api automation devops rails deep-learning	angular react web api	mobile desktop	systems deep-learning tool graphql serverless	automation data-science api rag
created_date	2024-09-14	2022-12-18	2024-12-13	2023-12-09	2018-01-04
last_commit_date	2024-12-19	2024-12-26	2024-12-15	2024-12-22	2024-11-20
readme_length	1282	1902	1251	155	3860
has_ci	0	0	0	1	0
test_coverage	NaN	NaN	NaN	73.5	NaN
has_code_of_conduct	0	0	0	0	0
has_contributing_guide	0	0	0	0	1
has_wiki	0	0	1	0	0
has_pages	0	0	0	0	0
has_discussions	1	0	0	0	0
default_branch	main	master	develop	main	develop
is_archived	0	0	0	0	0
is_fork	1	1	0	0	1
size_kb	497	4622	2148	812	5932

In [9]:

```
# Display the number of rows and columns.
print("Number of rows:", df.shape[0])
print("Number of columns:", df.shape[1])

# Display all column names.
print("\nColumn names:")
print(df.columns.tolist())

Number of rows: 5500
Number of columns: 29

Column names:
['repo_name', 'language', 'description', 'stars', 'forks', 'watchers', 'open_issues', 'closed_issues', 'open_pull_requests', 'merged_pull_requests', 'contributors', 'commits', 'releases', 'license', 'topics', 'created_date',
'last_commit_date', 'readme_length', 'has_ci', 'test_coverage', 'has_code_of_conduct', 'has_contributing_guide', 'has_wiki', 'has_pages', 'has_discussions', 'default_branch', 'is_archived', 'is_fork', 'size_kb']

# Display information about column types and non-null values.
df.info()
```

In [10]:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5500 entries, 0 to 5499
Data columns (total 29 columns):
 #   Column              Non-Null Count  Dtype
---  -
 0   repo_name           5500 non-null   object
 1   language            5500 non-null   object
 2   description         5500 non-null   object
 3   stars               5500 non-null   int64
 4   forks               5500 non-null   int64
 5   watchers            5500 non-null   int64
 6   open_issues         5500 non-null   int64
 7   closed_issues       5500 non-null   int64
 8   open_pull_requests  5500 non-null   int64
 9   merged_pull_requests 5500 non-null   int64
10   contributors        5500 non-null   int64
11   commits             5500 non-null   int64
12   releases            5500 non-null   int64
13   license             4760 non-null   object
14   topics              5500 non-null   object
15   created_date        5500 non-null   object
16   last_commit_date    5500 non-null   object
17   readme_length       5500 non-null   int64
18   has_ci              5500 non-null   int64
19   test_coverage       1702 non-null   float64
20   has_code_of_conduct 5500 non-null   int64
21   has_contributing_guide 5500 non-null   int64
22   has_wiki            5500 non-null   int64
23   has_pages           5500 non-null   int64
24   has_discussions     5500 non-null   int64
25   default_branch      5500 non-null   object
26   is_archived         5500 non-null   int64
27   is_fork             5500 non-null   int64
28   size_kb             5500 non-null   int64
dtypes: float64(1), int64(20), object(8)
memory usage: 1.2+ MB
```

```
In [11]: # Count duplicated rows.
duplicate_count = df.duplicated().sum()

print("Number of duplicated rows:", duplicate_count)
```

Number of duplicated rows: 0

4. Initial Data Cleaning

Column names and text values are cleaned before handling missing values and outliers. Blank and invalid text values are converted into `NaN`.

```
In [13]: # Remove unnecessary spaces from column names.
df.columns = df.columns.str.strip()

# Identify text columns.
object_columns = df.select_dtypes(include="object").columns

# Remove Leading and trailing spaces from text values.
for column in object_columns:
    df[column] = df[column].apply(
        lambda value: value.strip()
        if isinstance(value, str)
        else value
    )

# Replace invalid or empty text values with NumPy NaN.
df.replace(
    ["", "?", "nan", "NaN", "None", "null", "NULL"],
    np.nan,
    inplace=True
)

print("Initial data cleaning completed.")
```

Initial data cleaning completed.

5. Descriptive Analysis

Descriptive statistics summarize the main numerical characteristics of the repository metrics.

```
In [15]: # Display descriptive statistics for numerical columns.
df.describe().T.round(2)
```

Out[15]:

	count	mean	std	min	25%	50%	75%	max
stars	5500.0	45.18	419.63	0.0	2.00	7.0	22.00	20816.0
forks	5500.0	9.74	71.15	0.0	0.00	2.0	7.00	2727.0
watchers	5500.0	3.93	32.85	0.0	0.00	1.0	3.00	2002.0
open_issues	5500.0	5.37	11.67	0.0	1.00	3.0	7.00	470.0
closed_issues	5500.0	34.40	48.41	0.0	13.00	25.0	44.00	1984.0
open_pull_requests	5500.0	1.19	3.88	0.0	0.00	0.0	2.00	188.0
merged_pull_requests	5500.0	18.27	26.53	0.0	6.00	13.0	24.00	840.0
contributors	5500.0	3.61	2.98	1.0	1.00	3.0	5.00	24.0
commits	5500.0	517.68	684.20	1.0	108.75	262.5	666.25	7933.0
releases	5500.0	13.14	16.57	0.0	2.00	7.0	18.00	119.0
readme_length	5500.0	2448.41	4065.35	100.0	526.75	1163.5	2661.25	55493.0
has_ci	5500.0	0.31	0.46	0.0	0.00	0.0	1.00	1.0
test_coverage	1702.0	59.60	20.02	4.8	44.62	61.5	75.07	98.1
has_code_of_conduct	5500.0	0.12	0.32	0.0	0.00	0.0	0.00	1.0
has_contributing_guide	5500.0	0.18	0.38	0.0	0.00	0.0	0.00	1.0
has_wiki	5500.0	0.40	0.49	0.0	0.00	0.0	1.00	1.0
has_pages	5500.0	0.15	0.35	0.0	0.00	0.0	0.00	1.0
has_discussions	5500.0	0.10	0.31	0.0	0.00	0.0	0.00	1.0
is_archived	5500.0	0.02	0.15	0.0	0.00	0.0	0.00	1.0
is_fork	5500.0	0.15	0.35	0.0	0.00	0.0	0.00	1.0
size_kb	5500.0	20795.02	99851.34	24.0	1849.00	4810.5	13041.25	4405170.0

Part One: Handling Missing Values

Missing values are detected, summarized, visualized, and treated according to the feature type.

```
In [17]: # Count missing values in every column.
missing_before = df.isnull().sum()

# Keep only columns that contain missing values.
missing_before = missing_before[
    missing_before > 0
].sort_values(ascending=False)

missing_before
```

Out[17]: test_coverage 3798
license 740
dtype: int64

```
In [18]: # Create a summary table containing missing-value counts and percentages.
missing_summary = pd.DataFrame({
    "Missing Values": df.isnull().sum(),
    "Missing Percentage (%)": df.isnull().mean() * 100
})

# Keep only columns that contain missing values.
missing_summary = missing_summary[
    missing_summary["Missing Values"] > 0
].sort_values(
    by="Missing Values",
    ascending=False
)

missing_summary.round(2)
```

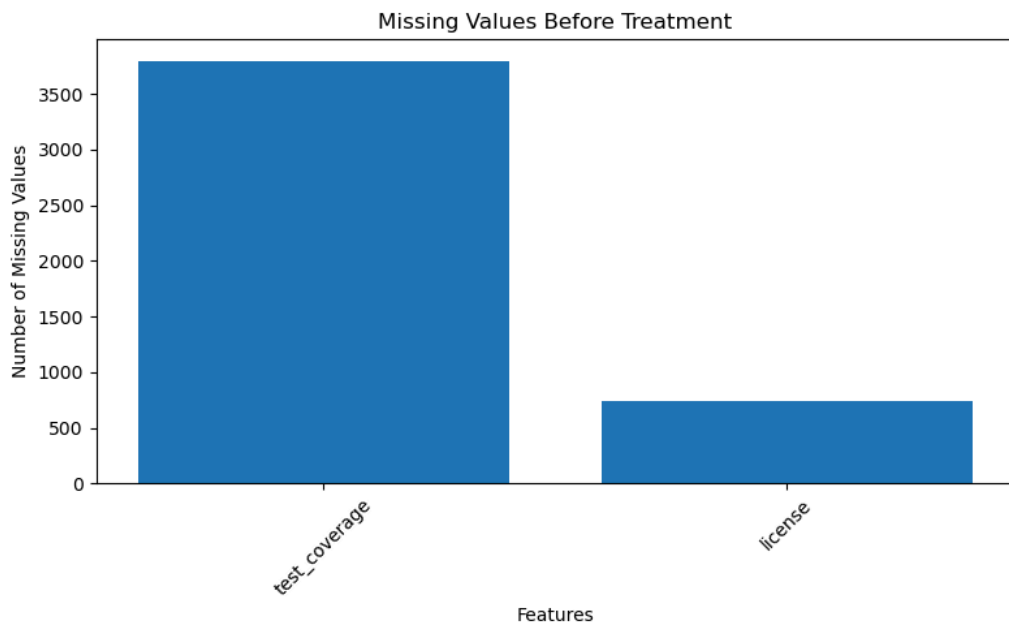
```
Out[18]:
```

	Missing Values	Missing Percentage (%)
test_coverage	3798	69.05
license	740	13.45

```
In [19]: # Visualize missing values before treatment.
plt.figure(figsize=(8, 5))

plt.bar(
    missing_before.index,
    missing_before.values
)

plt.title("Missing Values Before Treatment")
plt.xlabel("Features")
plt.ylabel("Number of Missing Values")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Missing-Value Treatment Strategy

- Missing numerical values are replaced with the **median** because the median is less affected by extreme values.

- Missing categorical values are replaced with the **mode** because the mode represents the most frequent category.

```
In [21]: # Identify numerical columns.
numerical_columns = df.select_dtypes(
    include=["number"]
).columns.tolist()

# Identify categorical columns.
categorical_columns = df.select_dtypes(
    include=["object", "category", "bool"]
).columns.tolist()

print("Numerical Columns:")
print(numerical_columns)

print("\nCategorical Columns:")
print(categorical_columns)
```

Numerical Columns:

```
['stars', 'forks', 'watchers', 'open_issues', 'closed_issues', 'open_pull_requests', 'merged_pull_requests', 'contributors', 'commits', 'releases', 'readme_length', 'has_ci', 'test_coverage', 'has_code_of_conduct', 'has_contributing_guide', 'has_wiki', 'has_pages', 'has_discussions', 'is_archived', 'is_fork', 'size_kb']
```

Categorical Columns:

```
['repo_name', 'language', 'description', 'license', 'topics', 'created_date', 'last_commit_date', 'default_branch']
```

```
In [22]: # Fill missing numerical values using the median.
for column in numerical_columns:
    if df[column].isnull().any():
        median_value = df[column].median()
        df[column] = df[column].fillna(median_value)

# Fill missing categorical values using the mode.
for column in categorical_columns:
    if df[column].isnull().any():
        mode_value = df[column].mode()[0]
        df[column] = df[column].fillna(mode_value)

# Verify that all missing values have been treated.
print(
    "Total missing values after treatment:",
    df.isnull().sum().sum()
)
```

Total missing values after treatment: 0

```
In [23]: # Compare missing values before and after treatment.
missing_comparison = pd.DataFrame({
    "Before Treatment": missing_before,
    "After Treatment": df[
        missing_before.index
    ].isnull().sum()
})

missing_comparison
```

```
Out[23]:
```

	Before Treatment	After Treatment
test_coverage	3798	0
license	740	0

```
In [24]: # Visualize missing values before and after treatment.
x_positions = np.arange(len(missing_comparison.index))
bar_width = 0.35

plt.figure(figsize=(9, 5))

plt.bar(
    x_positions - bar_width / 2,
    missing_comparison["Before Treatment"],
    width=bar_width,
```

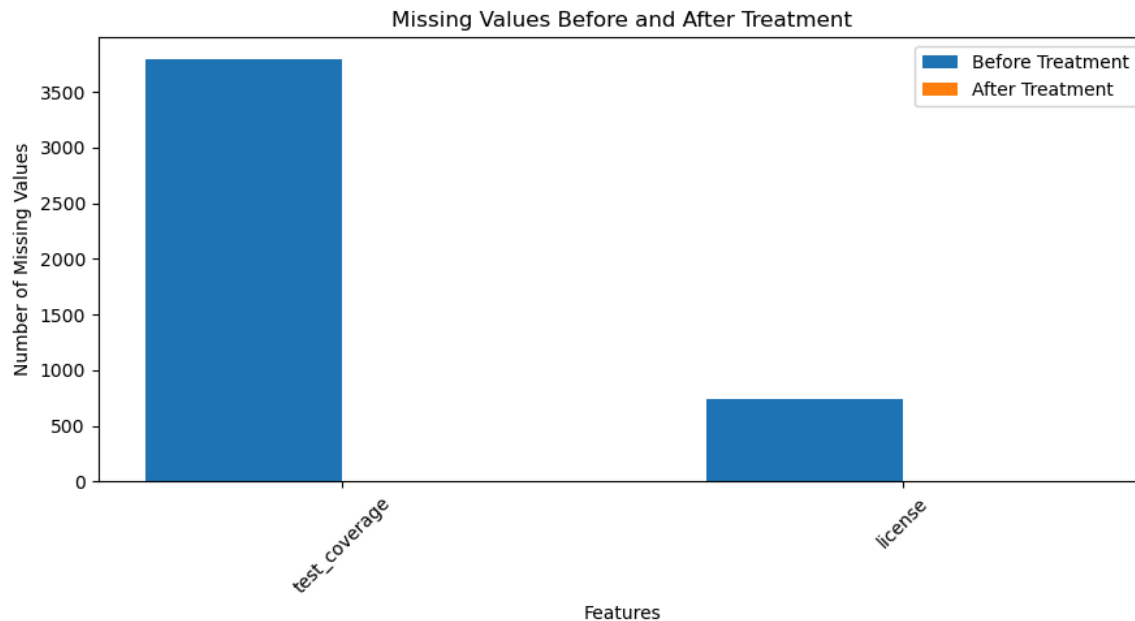
```

    label="Before Treatment"
)

plt.bar(
    x_positions + bar_width / 2,
    missing_comparison["After Treatment"],
    width=bar_width,
    label="After Treatment"
)

plt.title("Missing Values Before and After Treatment")
plt.xlabel("Features")
plt.ylabel("Number of Missing Values")
plt.xticks(
    x_positions,
    missing_comparison.index,
    rotation=45
)
plt.legend()
plt.tight_layout()
plt.show()

```



Part Two: Detecting and Handling Outliers

Potential outliers are detected using box plots and the **Interquartile Range (IQR)** method.

Binary columns are excluded because their values represent categories rather than continuous measurements.

The `test_coverage` feature is also excluded from IQR capping because approximately 69% of its values were missing and replaced using the median. Applying IQR after this imputation would make the quartile range collapse and could incorrectly replace most of the original observed values.

```

In [26]: # Select continuous numerical repository metrics.
continuous_columns = [
    "stars",
    "forks",
    "watchers",
    "open_issues",
    "closed_issues",
    "open_pull_requests",

```

```

        "merged_pull_requests",
        "contributors",
        "commits",
        "releases",
        "readme_length",
        "size_kb"
    ]

    # Keep only columns that exist in the dataset.
    continuous_columns = [
        column for column in continuous_columns
        if column in df.columns
    ]

    print("Continuous numerical columns:")
    print(continuous_columns)

```

Continuous numerical columns:

```
['stars', 'forks', 'watchers', 'open_issues', 'closed_issues', 'open_pull_requests', 'merged_pull_requests', 'contributors', 'commits', 'releases', 'readme_length', 'size_kb']
```

Box Plots Before Outlier Treatment

The following box plots show the distributions before applying IQR capping.

```

In [28]: # Create box plots before outlier treatment.
fig, axes = plt.subplots(
    5,
    3,
    figsize=(15, 20)
)

axes = axes.flatten()

for index, column in enumerate(continuous_columns):
    axes[index].boxplot(
        df[column],
        vert=False
    )

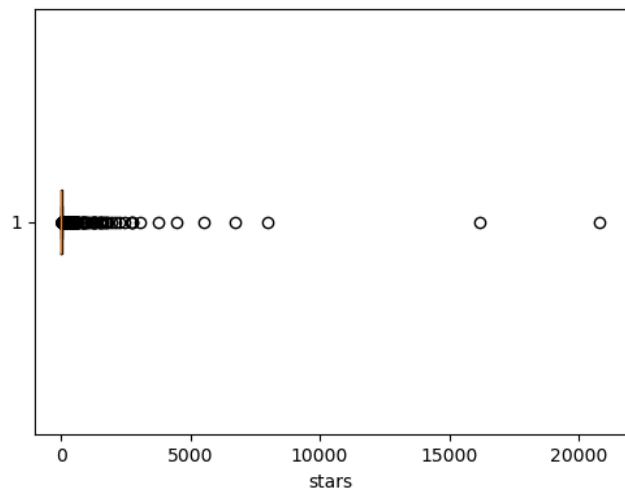
    axes[index].set_title(
        f"{column} Before Treatment"
    )
    axes[index].set_xlabel(column)

# Hide unused chart positions.
for index in range(
    len(continuous_columns),
    len(axes)
):
    axes[index].axis("off")

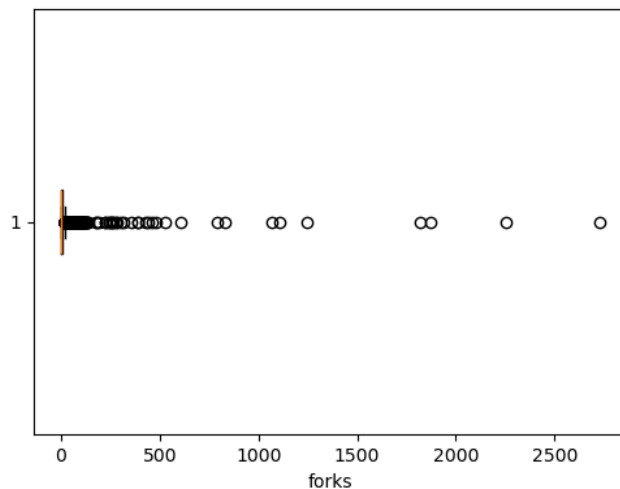
plt.tight_layout()
plt.show()

```

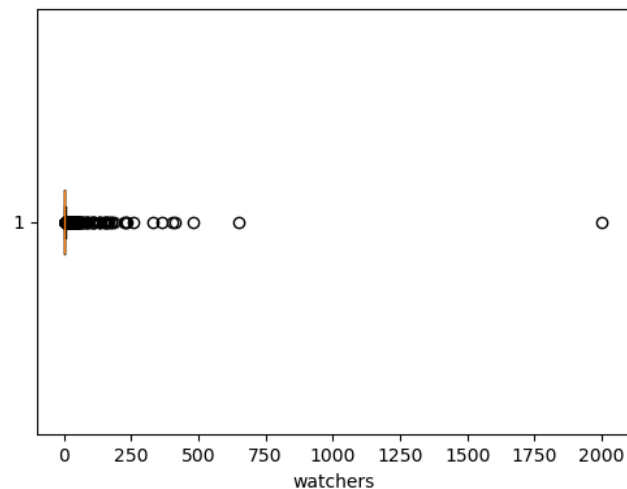
stars Before Treatment



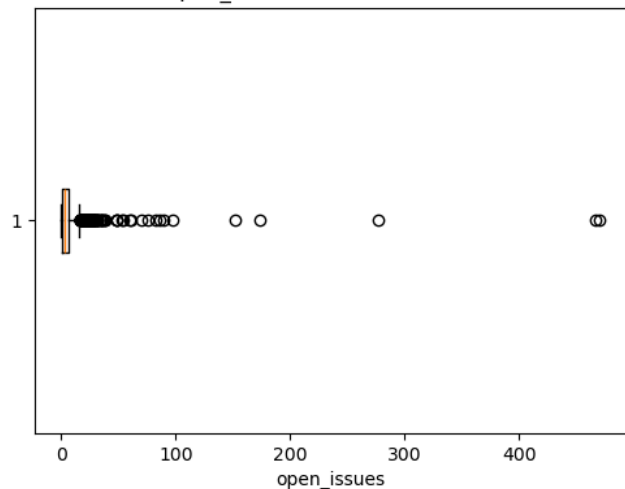
forks Before Treatment



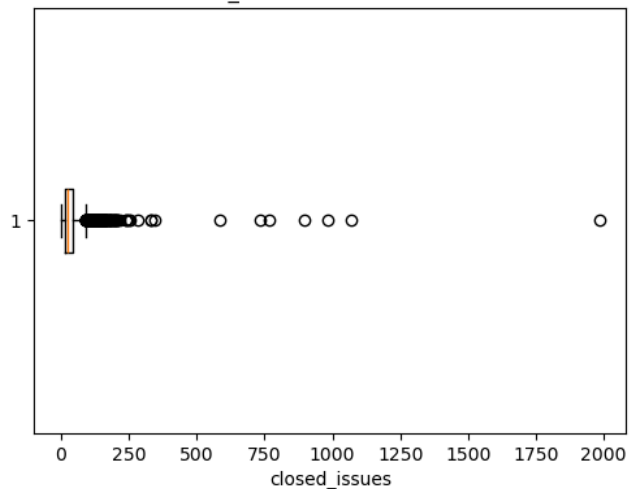
watchers Before Treatment



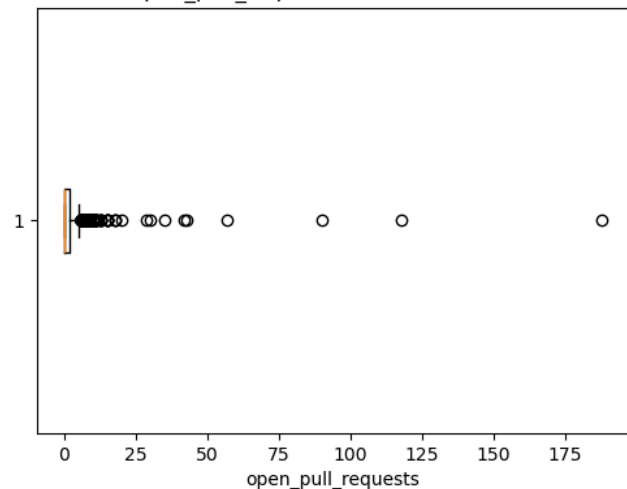
open_issues Before Treatment



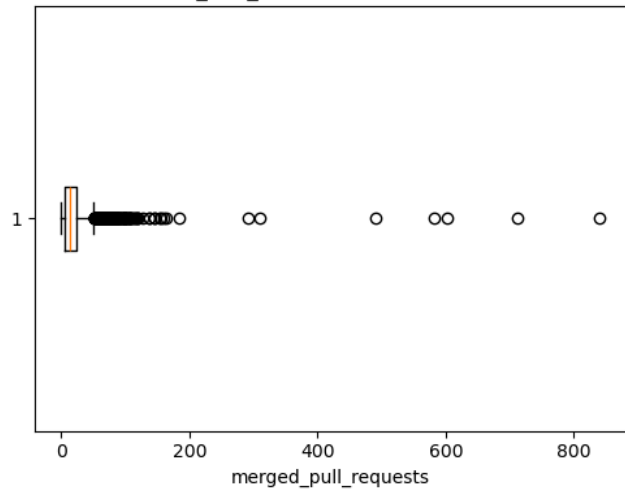
closed_issues Before Treatment



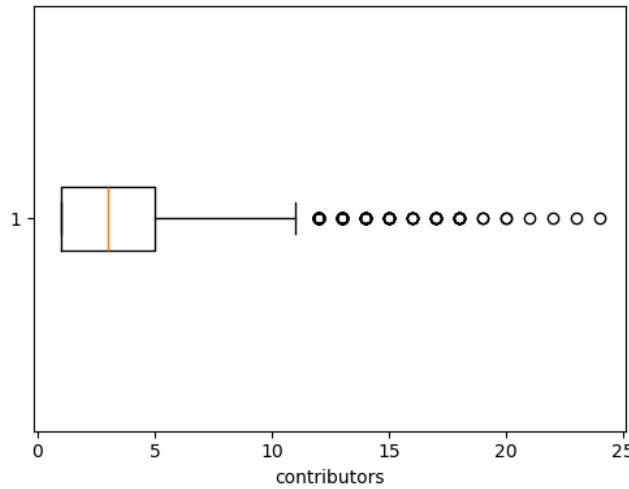
open_pull_requests Before Treatment



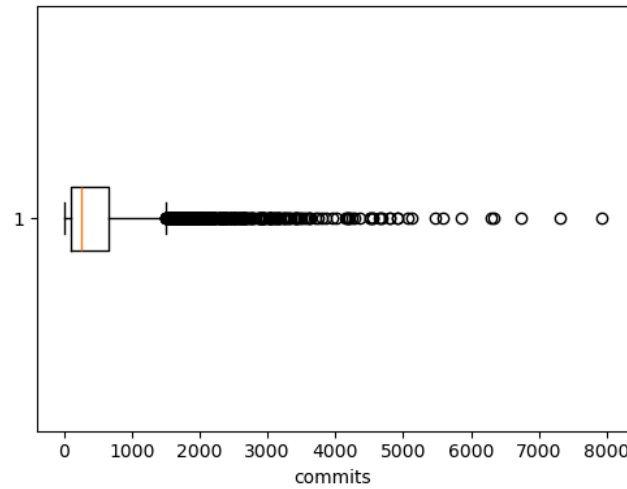
merged_pull_requests Before Treatment

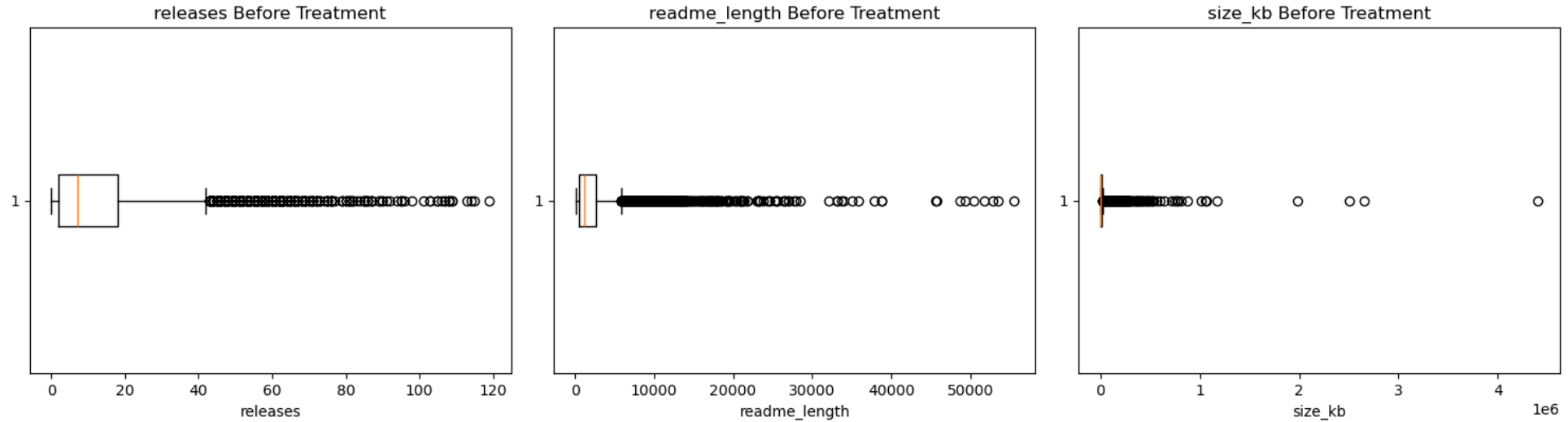


contributors Before Treatment



commits Before Treatment





Calculating Original IQR Boundaries

For each continuous feature:

- $IQR = Q3 - Q1$
- Lower Bound = $Q1 - 1.5 \times IQR$
- Upper Bound = $Q3 + 1.5 \times IQR$

The original boundaries are stored and reused for both treatment and verification.

```
In [30]: # Save a copy of the dataset before outlier treatment.
df_before_outlier_treatment = df.copy()

# Create structures to store original boundaries and counts.
outlier_bounds = {}
outliers_before = []

for column in continuous_columns:
    q1 = df_before_outlier_treatment[
        column
    ].quantile(0.25)

    q3 = df_before_outlier_treatment[
```

```

column
].quantile(0.75)

iqr = q3 - q1

lower_bound = q1 - 1.5 * iqr
upper_bound = q3 + 1.5 * iqr

# Store original boundaries.
outlier_bounds[column] = (
    lower_bound,
    upper_bound
)

# Count values outside the original boundaries.
outlier_count = (
    (
        df_before_outlier_treatment[column]
        < lower_bound
    )
    |
    (
        df_before_outlier_treatment[column]
        > upper_bound
    )
).sum()

outliers_before.append({
    "Feature": column,
    "Lower Bound": lower_bound,
    "Upper Bound": upper_bound,
    "Outliers Before Treatment": int(outlier_count)
})

# Convert the summary into a DataFrame.
outliers_before_df = pd.DataFrame(
    outliers_before
)

outliers_before_df.round(2)

```

Out[30]:

	Feature	Lower Bound	Upper Bound	Outliers Before Treatment
0	stars	-28.00	52.00	615
1	forks	-10.50	17.50	398
2	watchers	-4.50	7.50	409
3	open_issues	-8.00	16.00	215
4	closed_issues	-33.50	90.50	259
5	open_pull_requests	-3.00	5.00	183
6	merged_pull_requests	-21.00	51.00	274
7	contributors	-5.00	11.00	122
8	commits	-727.50	1502.50	419
9	releases	-22.00	42.00	351
10	readme_length	-2675.00	5863.00	516
11	size_kb	-14939.38	29829.62	696

In [31]:

```

# Visualize the number of outliers before treatment.
plt.figure(figsize=(13, 6))

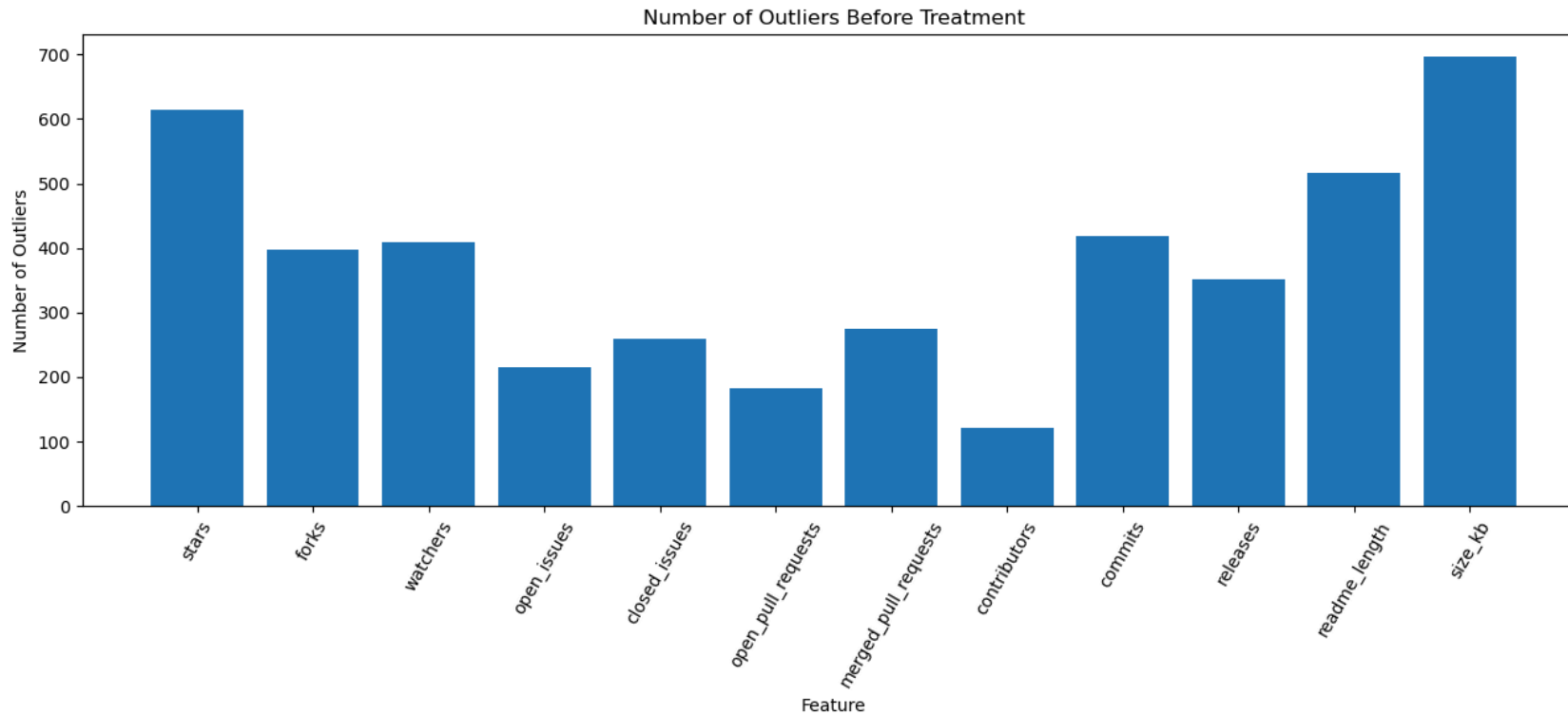
plt.bar(
    outliers_before_df["Feature"],
    outliers_before_df["Outliers Before Treatment"]
)

```

```

)
plt.title("Number of Outliers Before Treatment")
plt.xlabel("Feature")
plt.ylabel("Number of Outliers")
plt.xticks(rotation=60)
plt.tight_layout()
plt.show()

```



Outlier Treatment Using IQR Capping

Outliers are treated using the `clip()` function:

- Values below the lower boundary are replaced with the lower boundary.
- Values above the upper boundary are replaced with the upper boundary.
- No repository rows are deleted.

```

In [33]: # Apply IQR capping using the original stored boundaries.
for column in continuous_columns:
    lower_bound, upper_bound = outlier_bounds[column]

    df[column] = df[column].clip(
        lower=lower_bound,
        upper=upper_bound
    )

print("Outlier capping completed.")

```

Outlier capping completed.

Verifying Outlier Treatment

The same original boundaries are used to verify that all detected outliers were capped successfully.

```
In [35]: # Count outliers after treatment.
outliers_after = []

for column in continuous_columns:
    lower_bound, upper_bound = outlier_bounds[column]

    outlier_count_after = (
        (df[column] < lower_bound)
        |
        (df[column] > upper_bound)
    ).sum()

    outliers_after.append({
        "Feature": column,
        "Outliers After Treatment": int(
            outlier_count_after
        )
    })

outliers_after_df = pd.DataFrame(
    outliers_after
)

# Merge before-and-after counts.
outlier_comparison = outliers_before_df[
    ["Feature", "Outliers Before Treatment"]
].merge(
    outliers_after_df,
    on="Feature"
)

outlier_comparison
```

Out[35]:

	Feature	Outliers Before Treatment	Outliers After Treatment
0	stars	615	0
1	forks	398	0
2	watchers	409	0
3	open_issues	215	0
4	closed_issues	259	0
5	open_pull_requests	183	0
6	merged_pull_requests	274	0
7	contributors	122	0
8	commits	419	0
9	releases	351	0
10	readme_length	516	0
11	size_kb	696	0

```
In [36]: # Confirm that all detected outliers were capped.
all_outliers_capped = (
    outlier_comparison[
        "Outliers After Treatment"
    ] == 0
).all()

print(
    "All detected outliers were capped successfully:",
    all_outliers_capped
)
```

All detected outliers were capped successfully: True

```
In [37]: # Visualize outlier counts before and after treatment.
x_positions = np.arange(
    len(outlier_comparison["Feature"])
)

bar_width = 0.35

plt.figure(figsize=(14, 6))

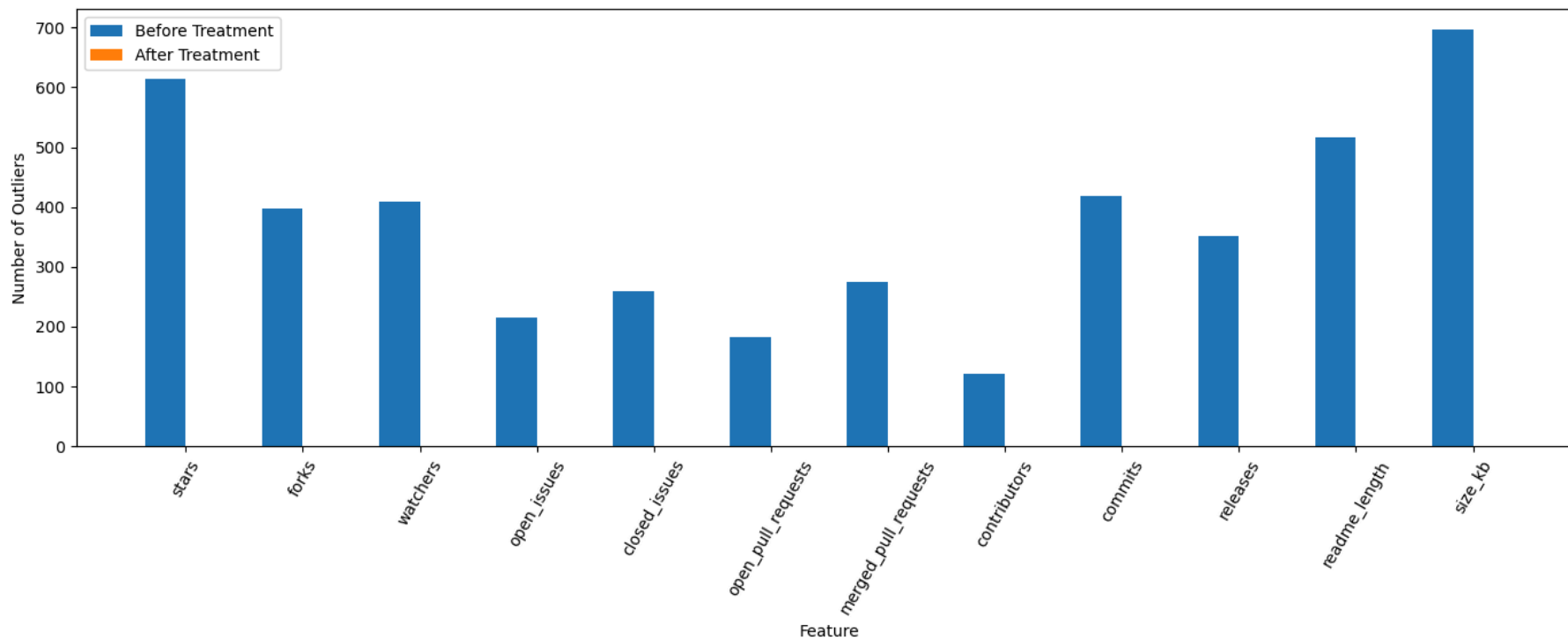
plt.bar(
    x_positions - bar_width / 2,
    outlier_comparison[
        "Outliers Before Treatment"
    ],
    width=bar_width,
    label="Before Treatment"
)

plt.bar(
    x_positions + bar_width / 2,
    outlier_comparison[
        "Outliers After Treatment"
    ],
    width=bar_width,
    label="After Treatment"
)

plt.title("Outliers Before and After Treatment")
plt.xlabel("Feature")
plt.ylabel("Number of Outliers")
plt.xticks(
    x_positions,
    outlier_comparison["Feature"],
    rotation=60
)

plt.legend()
plt.tight_layout()
plt.show()
```

Outliers Before and After Treatment



Box Plots After Outlier Treatment

The following box plots show the distributions after applying IQR capping.

```
In [39]: # Create box plots after outlier treatment.
fig, axes = plt.subplots(
    5,
    3,
    figsize=(15, 20)
)

axes = axes.flatten()

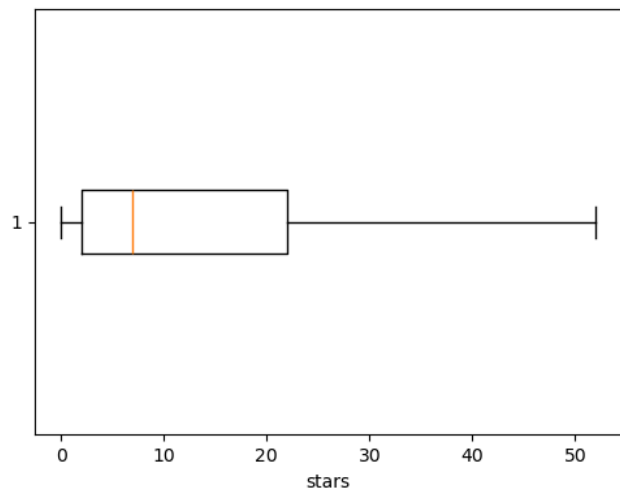
for index, column in enumerate(continuous_columns):
    axes[index].boxplot(
        df[column],
        vert=False
    )

    axes[index].set_title(
        f"{column} After Treatment"
    )
    axes[index].set_xlabel(column)

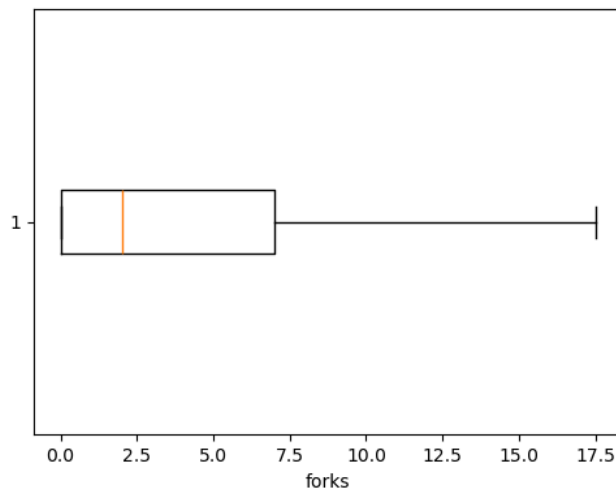
# Hide unused chart positions.
for index in range(
    len(continuous_columns),
    len(axes)
):
    axes[index].axis("off")

plt.tight_layout()
plt.show()
```

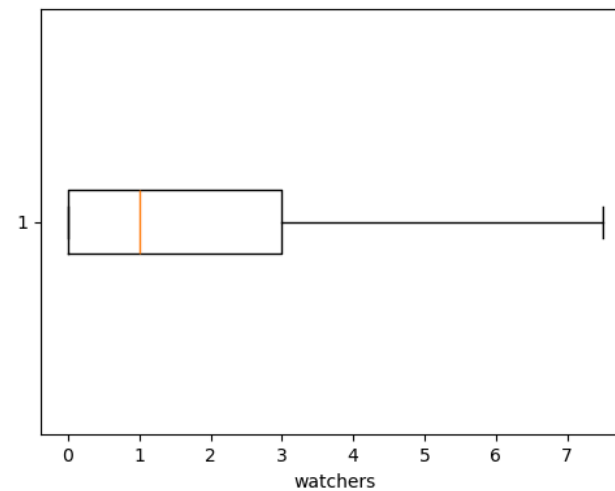
stars After Treatment



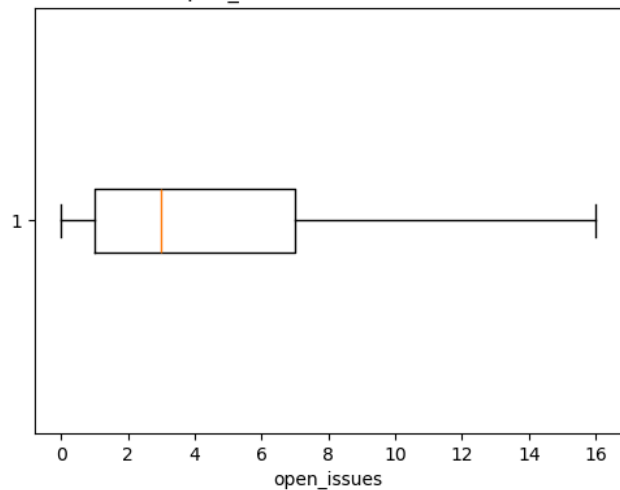
forks After Treatment



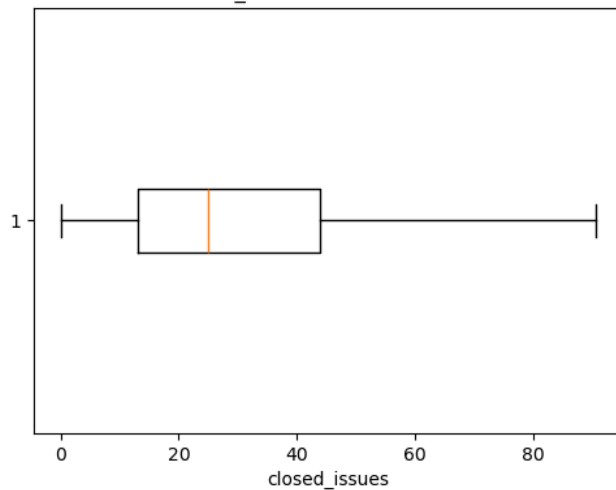
watchers After Treatment



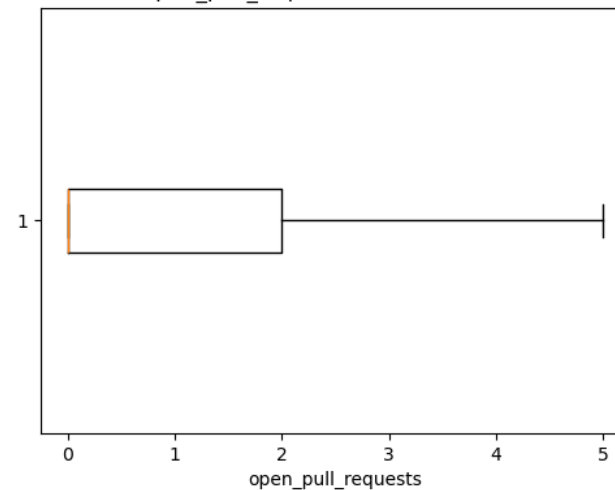
open_issues After Treatment



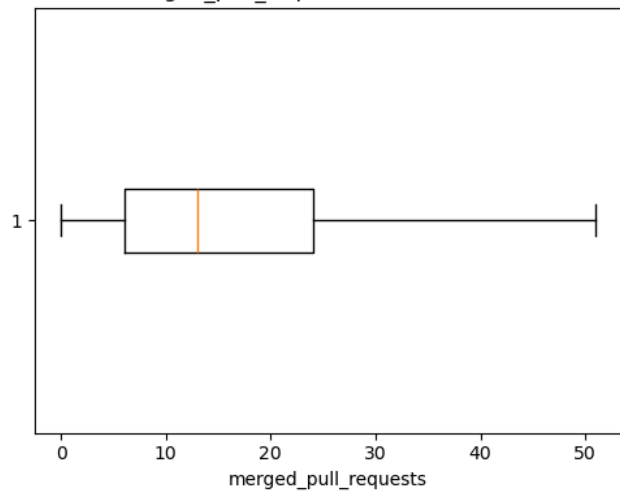
closed_issues After Treatment



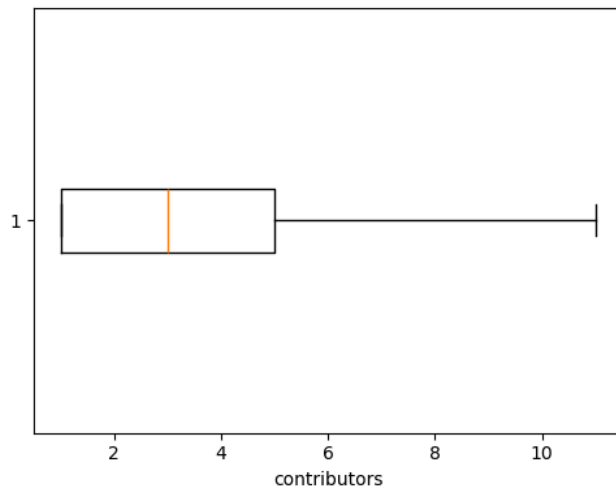
open_pull_requests After Treatment



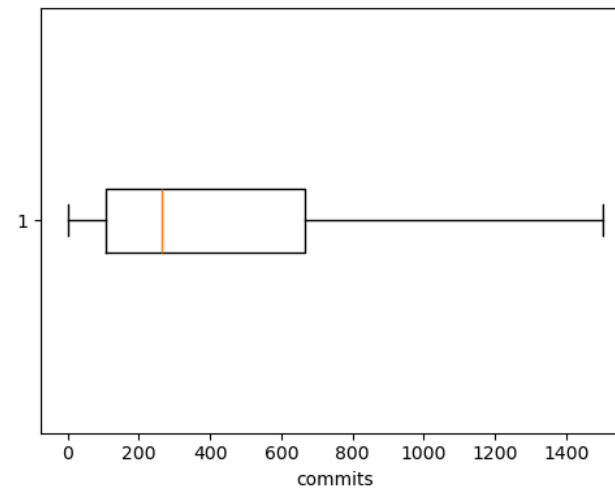
merged_pull_requests After Treatment

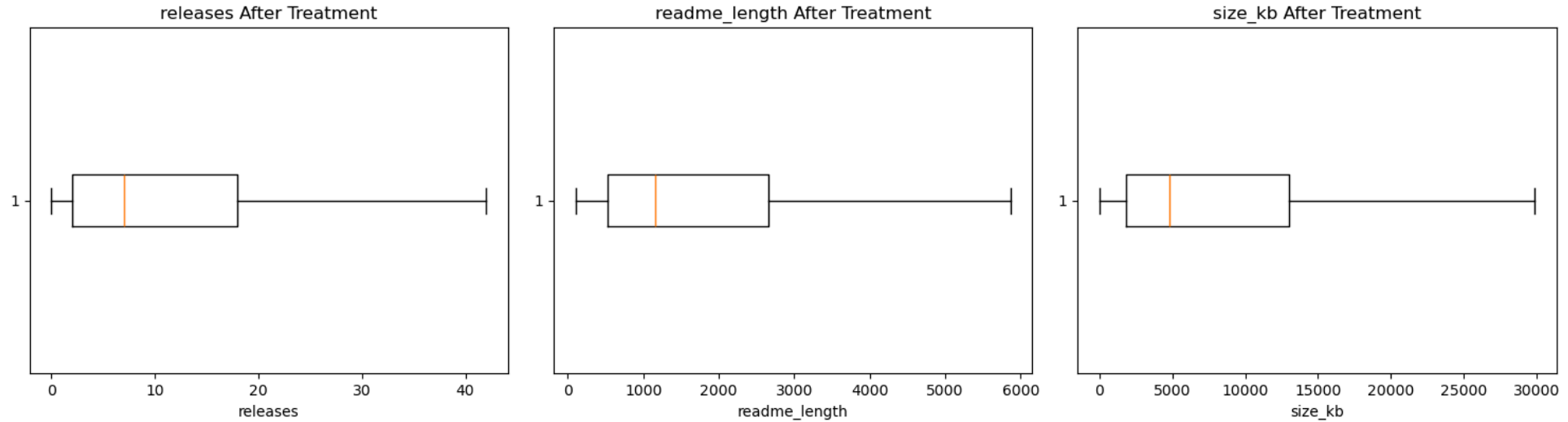


contributors After Treatment



commits After Treatment





Part Three: Data Visualization

This section explores programming languages, repository popularity, activity, and relationships between repository metrics.

Top Programming Languages

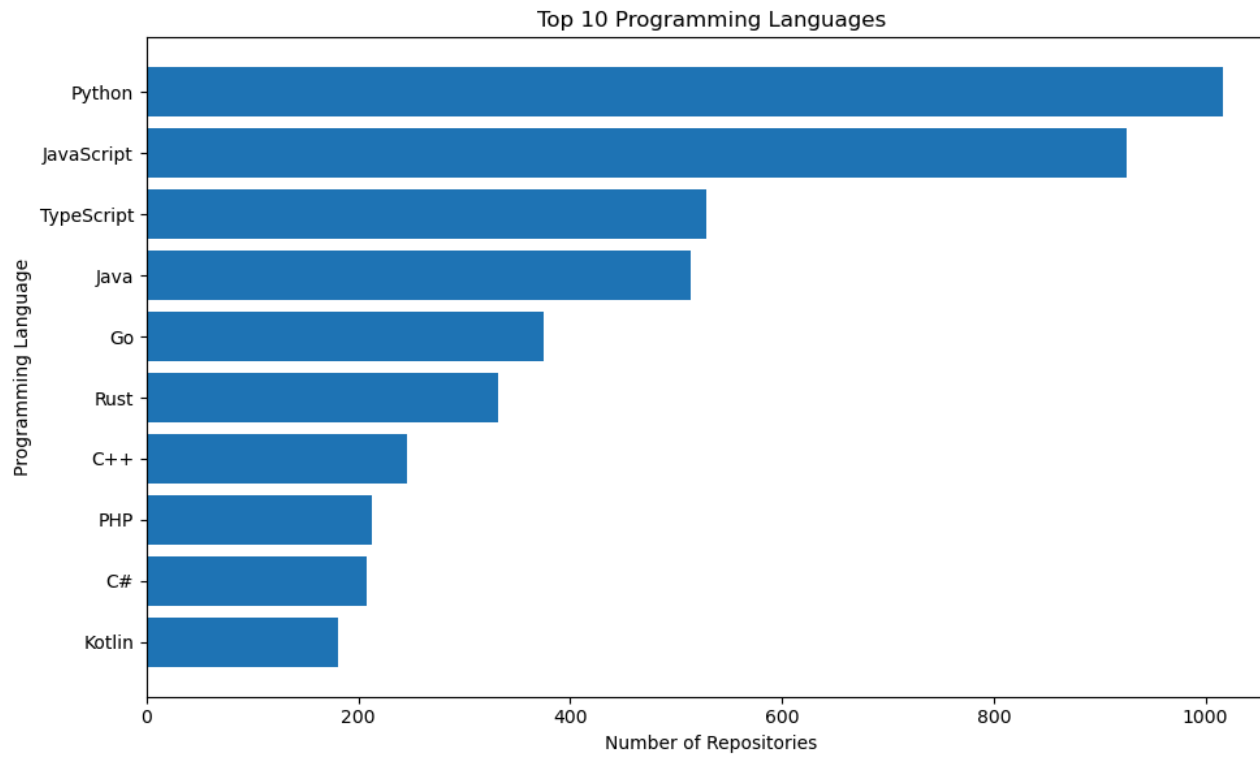
```
In [42]: # Count the ten most common programming Languages.
top_languages = (
    df["language"]
    .value_counts()
    .head(10)
)

plt.figure(figsize=(10, 6))

plt.barh(
    top_languages.index,
    top_languages.values
)

plt.title("Top 10 Programming Languages")
plt.xlabel("Number of Repositories")
```

```
plt.ylabel("Programming Language")
plt.gca().invert_yaxis()
plt.tight_layout()
plt.show()
```



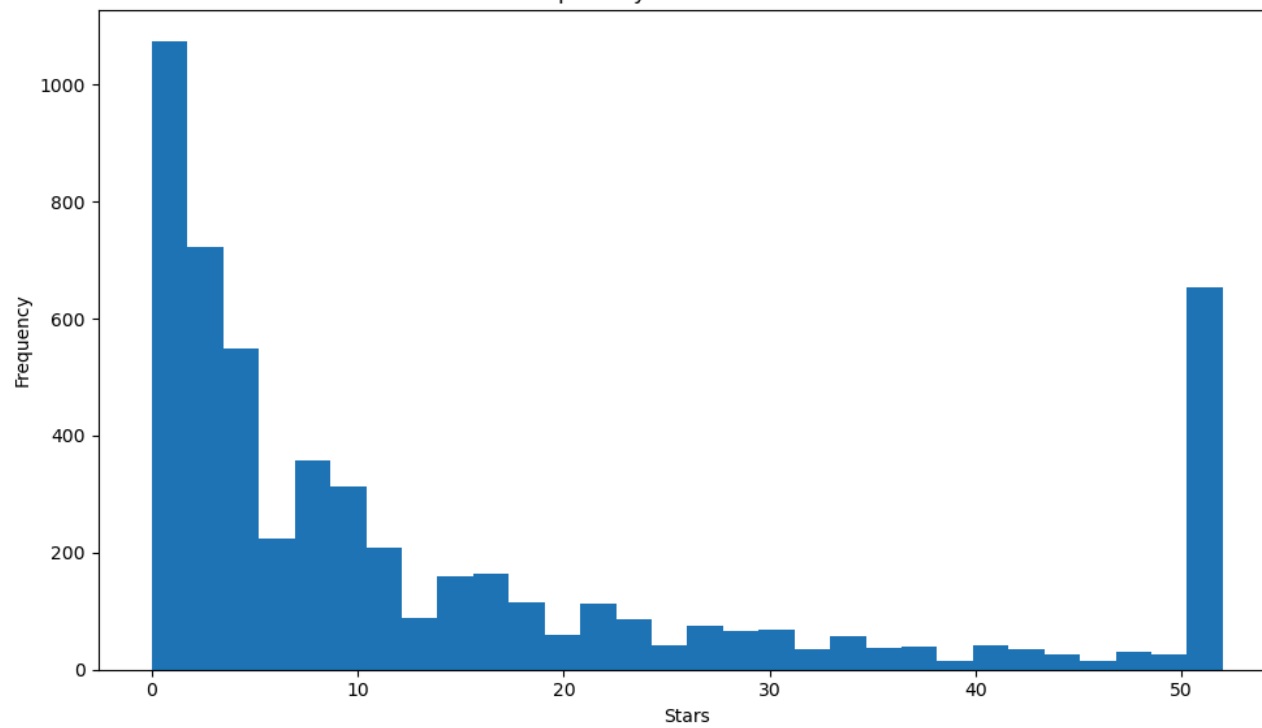
Repository Stars Distribution

```
In [44]: # Display the distribution of repository stars.
plt.figure(figsize=(10, 6))

plt.hist(
    df["stars"],
    bins=30
)

plt.title("Repository Stars Distribution")
plt.xlabel("Stars")
plt.ylabel("Frequency")
plt.tight_layout()
plt.show()
```

Repository Stars Distribution



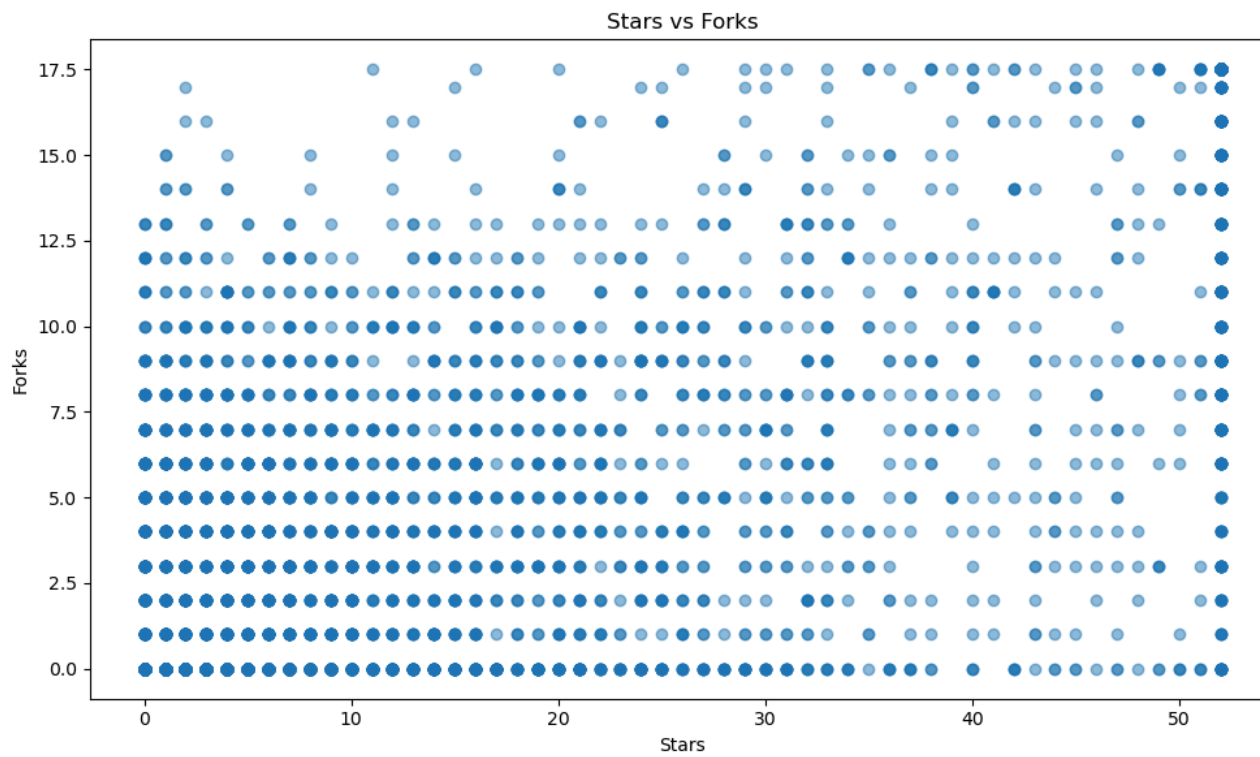
Stars and Forks Relationship

In [46]: *# Display the relationship between stars and forks.*

```
plt.figure(figsize=(10, 6))

plt.scatter(
    df["stars"],
    df["forks"],
    alpha=0.5
)

plt.title("Stars vs Forks")
plt.xlabel("Stars")
plt.ylabel("Forks")
plt.tight_layout()
plt.show()
```



Average Stars by Programming Language

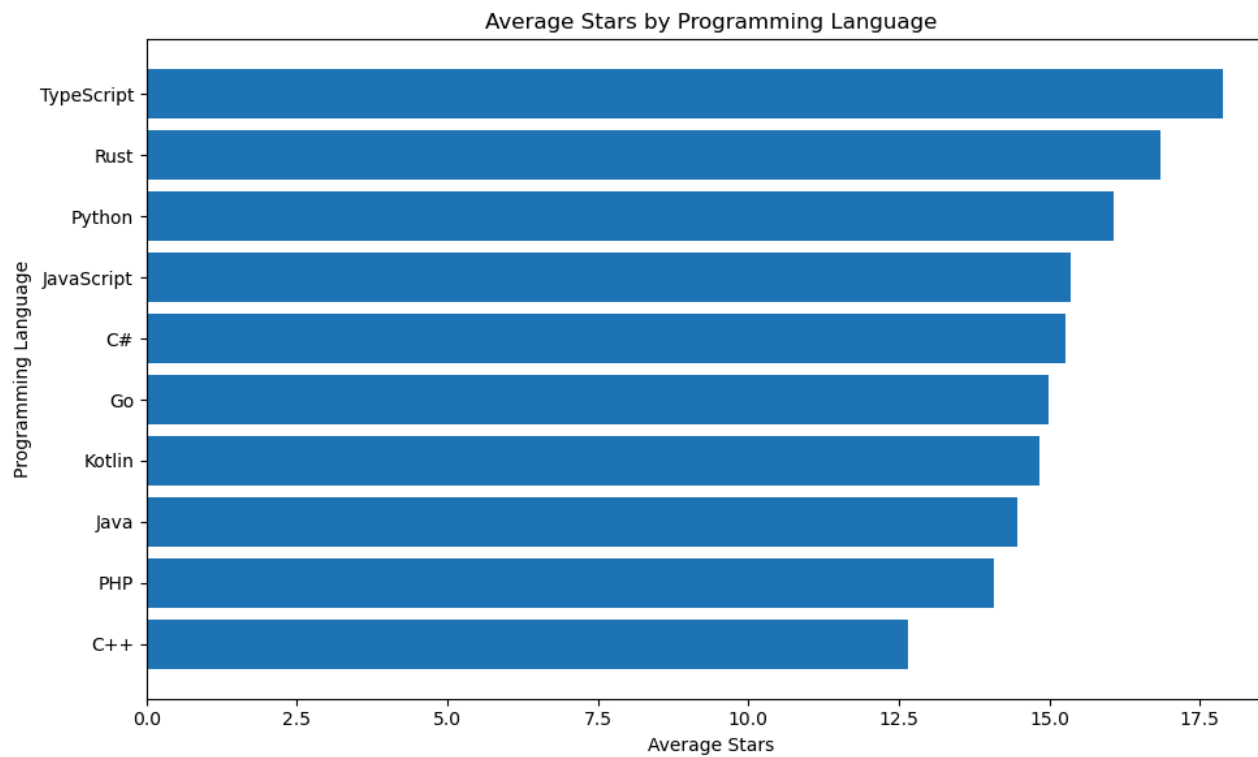
In [48]: `# Calculate average stars for the ten most common languages.
common_languages = top_languages.index`

```
average_stars = (
    df[df["language"].isin(common_languages)]
      .groupby("language")["stars"]
      .mean()
      .sort_values(ascending=False)
)

plt.figure(figsize=(10, 6))

plt.barh(
    average_stars.index,
    average_stars.values
)

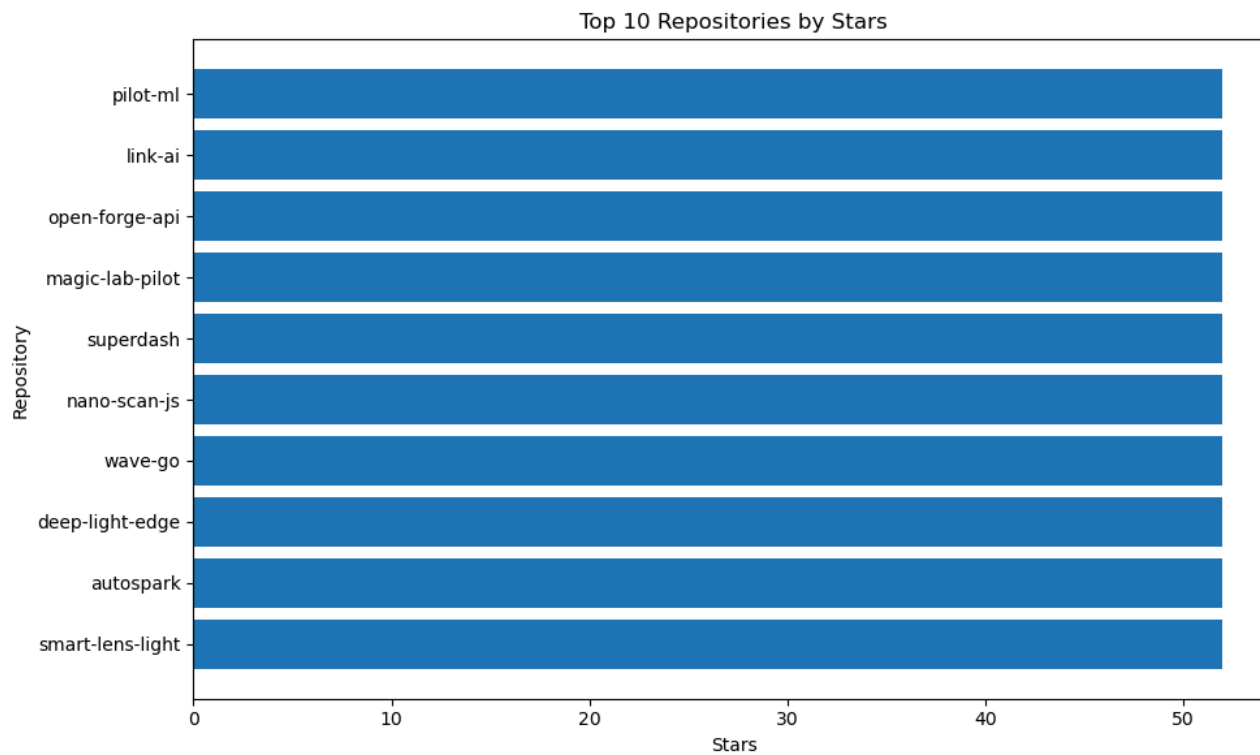
plt.title("Average Stars by Programming Language")
plt.xlabel("Average Stars")
plt.ylabel("Programming Language")
plt.gca().invert_yaxis()
plt.tight_layout()
plt.show()
```



Top Repositories by Stars

In [50]: *# Select the ten repositories with the highest star counts.*

```
top_repositories = (  
    df[  
        ["repo_name", "stars"]  
    ]  
    .sort_values(  
        "stars",  
        ascending=False  
    )  
    .head(10)  
)  
  
plt.figure(figsize=(10, 6))  
  
plt.barh(  
    top_repositories["repo_name"],  
    top_repositories["stars"]  
)  
  
plt.title("Top 10 Repositories by Stars")  
plt.xlabel("Stars")  
plt.ylabel("Repository")  
plt.gca().invert_yaxis()  
plt.tight_layout()  
plt.show()
```



Correlation Heatmap

The heatmap shows correlations among selected numerical repository metrics.

```
In [52]: # Select numerical metrics for correlation analysis.
correlation_columns = [
    "stars",
    "forks",
    "watchers",
    "open_issues",
    "closed_issues",
    "contributors",
    "commits",
    "releases",
    "readme_length",
    "test_coverage",
    "size_kb"
]

# Calculate the correlation matrix.
correlation_matrix = df[
    correlation_columns
].corr()

# Create a heatmap using Matplotlib.
fig, ax = plt.subplots(
    figsize=(12, 10)
)

image = ax.imshow(
    correlation_matrix,
    aspect="auto"
)
```

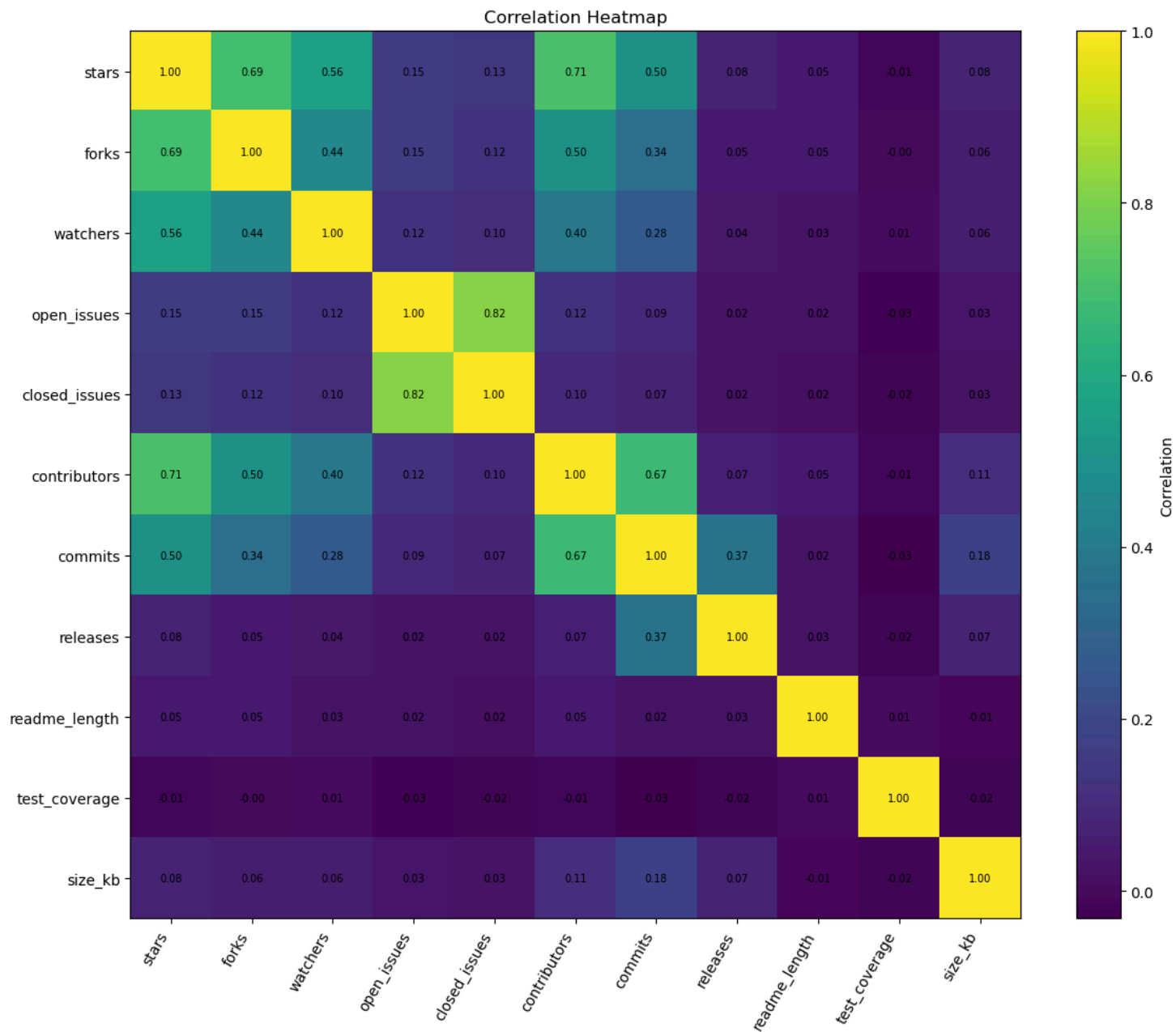
```
ax.set_xticks(
    range(len(correlation_columns))
)
ax.set_yticks(
    range(len(correlation_columns))
)

ax.set_xticklabels(
    correlation_columns,
    rotation=60,
    ha="right"
)
ax.set_yticklabels(
    correlation_columns
)

# Display correlation values inside the heatmap cells.
for row in range(len(correlation_columns)):
    for column in range(len(correlation_columns)):
        ax.text(
            column,
            row,
            f"{correlation_matrix.iloc[row, column]:.2f}",
            ha="center",
            va="center",
            fontsize=7
        )

plt.colorbar(
    image,
    ax=ax,
    label="Correlation"
)

plt.title("Correlation Heatmap")
plt.tight_layout()
plt.show()
```



Final Verification

The project is considered complete when:

- No missing values remain.
- All detected continuous-feature outliers have been capped.
- The dataset retains the same number of rows.
- Required visualizations are displayed successfully.

```
In [54]: # Print the final verification results.
print(
    "Original number of rows:",
    df_before_outlier_treatment.shape[0]
)

print(
    "Final number of rows:",
    df.shape[0]
)

print(
    "Final number of columns:",
    df.shape[1]
)

print(
    "Total missing values:",
    df.isnull().sum().sum()
)

print(
    "Total outliers after treatment:",
    outlier_comparison[
        "Outliers After Treatment"
    ].sum()
)

print(
    "Duplicated rows:",
    df.duplicated().sum()
)
```

Original number of rows: 5500
Final number of rows: 5500
Final number of columns: 29
Total missing values: 0
Total outliers after treatment: 0
Duplicated rows: 0

Conclusion

This project successfully completed the required exploratory data analysis tasks:

1. Missing values were identified and treated using median and mode imputation.
2. Outliers were detected using the IQR method and treated using capping.
3. Multiple visualizations were created to describe repository popularity, programming languages, activity metrics, and correlations.