Summative Assessment

Section 1: Theory supported by code samples (50%, 1400 words plus code samples)

Evidence for learning outcome: Demonstrate critical understanding of the theory and application of advanced programming techniques; Design and implement programs for real world problems.

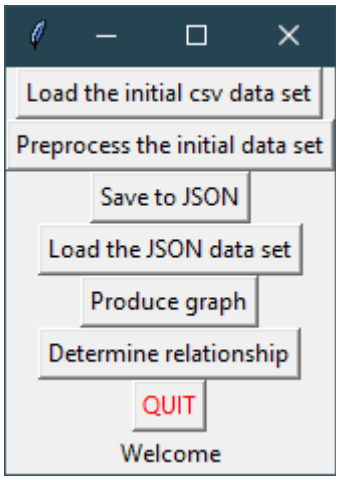
From the project specifications under technical requirements: "The application or its parts do not run concurrently, do NOT use python threads." In following with the client specifications, no concurrent mechanisms have been selected for the purpose of creating a concurrent program. Concurrent mechanisms are present in the program, run locally, as a natural consequence of its functionality and features. Atomicity is present as processes are run sequentially on the database. When buttons in the UI are pressed, to call a function, the program will queue the command while it finishes the current task. This prevents concurrent tasks within the program from being done on data that hasn't finished being handled. No parallel programs are present that would access the same database leading to errors.

In the case of making this program concurrent, some features could be added to minimize issues for a shared database accessed by multiple programs. Critical sections are sections of code in which data is read from and written to files, that should run with no interference [2]. While a json file is being saved to with the current program's dataframe, that same file could be receiving additional data from a concurrent program leading to errors. This is prevented by having a single instance of that code allowed to run at a time. During these critical sections, the program could be specified to exclude any other processes until the current process has finished, requiring locks to be placed on certain data items [10] or on certain sections of code.

```
def save_dataframe_to_json(self):
    if self.loaded:
        self.Inspections.to_json(r'Inspections.json')
        self.Inventroy.to_json(r'Inventroy.json')
        self.violations.to_json(r'violations.json')
        self.text.set("Saved to JSON")
    else:
        self.text.set("Error. Please load data from files")
```

Semaphores could be used for this purpose. A "guard"[2] is a device that allows a set of code to be executed when a specified condition is true; in this case, whether the database is being accessed by an instance of the program. A simple implementation of this could be a conditional that checks the value of a boolean that is switched when a program begins or ends its task on the shared database. When a concurrent program attempts to access the database while the guard is active, it is blocked. The situation of being blocked could be handled by returning an error, or for scalabilty the task

could be queued until the gate is opened and the task is next to be run. To protect critical sections from errors more thoroughly, a monitor could be implemented such that interference is prevented. There is a rare case in which a process begins its task after a concurrent program has checked the semaphore and before it changes the semaphore. The guarantee of a monitor is desirable for implementing a concurrent program at a large scale with unexpected circumstances that could lead to data corruption [2].



The prototype UI is contained in a simple window with buttons that execute functions for the required specifications. Each task is modularized to be run by the user when required. The program's text below the buttons communicate's the program's current status and gives regular feedback to the user. The program's exit button is highlighted in red for a quick identification and easy escape. Design complexity is minimal, allowing for an intuitive and comprehensive UX.

Each button is appropriately labelled, leaving no ambiguity on the overall task while hiding small tasks that would only confuse or slow down the user. Whenever a task is complete, the status text changes to inform the user. Data visualizations are produced in the console. While a task is running, the button remains "pressed" to indicate so. Clicking another button whilst a function is running doesn't produce a response indicating to the user that the program is busy. Any errors caused by invalid or absent data lead to a status message indicating that there is an error and the appropriate corrective response.

```
    def load_json_data(self):
        try:
            self.Inspections = pd.read_json(r'Inspections.json')
            self.Inventroy = pd.read_json(r'Inventroy.json')
            self.violations = pd.read_json(r'violations.json')
            self.text.set("Loaded JSON")
            self.loaded = True
        except:
            self.text.set("Error loading data. Please check that files are
```

The first functional requirement is to be able to load the initial data set, fulfilled by the first button. Saving the data to a JSON is provided by the third button. This button comes after the preprocessing function so that the data being saved is likely to have been preprocessed beforehand. The most desireable or efficient sequence of commands follows the order going down the list of buttons. The user is compelled to execute commands in the order presented in the UI naturally. Thus, the data saved to the JSON is in the desired, more compact preprocessed format rather than directly from the csv, which would require an additional override for the data to be in the desired format.

Preprocessing steps are bundled into a function executed by the user. Individual steps aren't shown as this would add unnecessary complexity to the UI. Data is loaded by both a csv and a json file. Buttons for each of these loads inform the user of what task is being executed. This is also communicated through the program's status message, providing visibility of the program's status. The tasks for producing a suitable graph for violations and determining a correlation between violations and zip codes are each given a UI button that provides user control and freedom on what task to run next after each stable output. The design choices and their justifications listed are in line with UI heuristics commonly used for effective UI design [3].

Between Java and Python, Python is more appropriate for the data manipulation and processing required for the project specifications. Python comes with useful data libraries such as pandas that allow for efficient and simple code to perform merges, joins, dropping na values, and loading/saving to a file. Weka is a Java tool with similar functionality for processing data, creating visualizions, finding patterns and applying machine learning models. Weka supports only a sequential node execution and thus is limited in scalability. DistributedWekaSpark is a distributed framework built on Spark that allows Weka to handle larger datasets at scale [9].

Java is appropriate for defining classes and creating a functional UI in a way similar to python. In this regard, Java is comparable. Having a modular ide for testing each function is made simple with Jupyter notebook. Errors are displayed in Jupyter logs clearly for each section of code. Python includes machine learning libraries such as scikit learn that work easily with dataframes in pandas and can create visualizations for logistic regression in a few lines of code. Java similarly has access to tools such as JSAT, a library for machine learning compared with scikitlearn that is easy for developers and researchers to use [8]. Java's performance is generally faster than that of Python, but it has been found through trials, generally speaking, that writing code in Python takes no more than half the time of writing it in lower level languages such as Java, with the resulting program also being half as long [4]. For presentable code that demonstrates an implementation and understanding of advanced programming principles without unnecessary complexity, Python is thus more suitable.

In [9]:
```
import csv
import json
import pandas as pd
import seaborn as sns
import tkinter as tk
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression
```

Section 2: Design decisions supported by code samples (40% 1200 words plus code samples)

Evidence for learning outcome: Communicate design decisions for the selection, storage and manipulation of data; Design and implement programs for real world problems.

This program follows the specifications outlined in the list of requirements. Loading the data into our program is a reasonable first step. This allows the developer to test later functions and understand the data. The first step is to import the csv python library, a set of methods and tools for reading and writing csv files using python. This program must parse the csv input into usable XML or JSON. This program uses JSON.

In [9]:
```
import csv
import json
import pandas as pd
import seaborn as sns
import tkinter as tk
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression
```

JSON data can be worked on simply through methods in the python json library. Json format makes translation into a dictionary or collection of objects straightforward. The csv data is loaded as a pandas dataframe, then the dataframe function saves it to a json file. This file is used for the second specification of preserving the current state of the data. Loading and saving data will be designated to functions that return the data in dataframe format and update the json files with the current dataframes.

```
        self.save = tk.Button(self)
        self.save["text"] = "Save to JSON"
        self.save["command"] = self.save_dataframe_to_json
        self.save.pack(side="top")
```

```
    def save_dataframe_to_json(self):
        if self.loaded:
            self.Inspections.to_json(r'Inspections.json')
            self.Inventroy.to_json(r'Inventroy.json')
            self.violations.to_json(r'violations.json')
            self.text.set("Saved to JSON")
        else:
            self.text.set("Error. Please load data from files")
```

```
def load_json_data(self):
    try:
        self.Inspections = pd.read_json(r'Inspections.json')
        self.Inventroy = pd.read_json(r'Inventroy.json')
        self.violations = pd.read_json(r'violations.json')
        self.text.set("Loaded JSON")
        self.loaded = True
    except:
        self.text.set("Error loading data. Please check that files are
```

The pandas library comes packaged with data manipulation methods that this program uses extensively, so pandas is imported as pd. Pandas dataframes also make a straightforward medium for csv and json data. With one line of code a pandas dataframe can be built with csv data or converted to JSON. The specifications for data formatting is dependent on client requirements. These processes are batched together in one function that is started through the UI. Methods and variables are contained within the Application UI class constructed and displayed with the tkinter library. The program status is displayed through simple text at the bottom of the UI window. A string variable widget is added to the application that is updated during each method call executed with labeled buttons appearing in the UI window.

```
def create_widgets(self):
    self.text = tk.StringVar()
    self.text.set("Welcome")
    self.label = tk.Label(textvariable=self.text)
    self.label.pack(side="top")

    self.load_initial = tk.Button(self)
    self.load_initial["text"] = "Load the initial csv data set"
    self.load_initial["command"] = self.load_csv_data
    self.load_initial.pack(side="top")

    self.clean = tk.Button(self)
    self.clean["text"] = "Preprocess the initial data set"
    self.clean["command"] = self.preprocess
    self.clean.pack(side="top")

    self.save = tk.Button(self)
    self.save["text"] = "Save to JSON"
    self.save["command"] = self.save_dataframe_to_json
    self.save.pack(side="top")

    self.load = tk.Button(self)
    self.load["text"] = "Load the JSON data set"
    self.load["command"] = self.load_json_data
    self.load.pack(side="top")

    self.visual = tk.Button(self)
    self.visual["text"] = "Produce graph"
    self.visual["command"] = self.visualize
    self.visual.pack(side="top")

    self.determine = tk.Button(self)
    self.determine["text"] = "Determine relationship"
    self.determine["command"] = self.correlate
    self.determine.pack(side="top")

    self.quit = tk.Button(self, text="QUIT", fg="red",
                          command=self.master.destroy)
    self.quit.pack(side="bottom")
```

Error handling for loading csv and json files is implemented through a try clause. The class contains a boolean that indicates whether data is successfully loaded into the program and is updated if so. Otherwise, an exception is raised and the user interface updates the user. When data is saved, the boolean for whether data is loaded is checked and the program will run if True else the user will be informed of an error and no action will be taken. Conditionals are used for the rest of the functions in order to prevent errors on nonexistent data.

```
def load_csv_data(self):
    try:
        self.Inspections = pd.read_csv(r'Inspections.csv')
        self.Inventroy = pd.read_csv(r'Inventroy.csv', nrows=100)
        self.violations = pd.read_csv(r'violations.csv')
        self.text.set("Loaded CSV")
        self.loaded = True
    except:
        self.text.set("Error loading data. Please check that files are
```

Preprocess removes data from vendors with 'PROGRAM STATUS' of INACTIVE as per the client specifications. The number and type of seating from the 'PE DESCRIPTION' column is extracted into a new column. This prepares the data for dataframe operations. To extract the substrings within parenthesis, the python find method in a for loop grabs the indexes of '(' and ')' , using the list expression ':' to create a substring that is added to the 'SEATING' column.

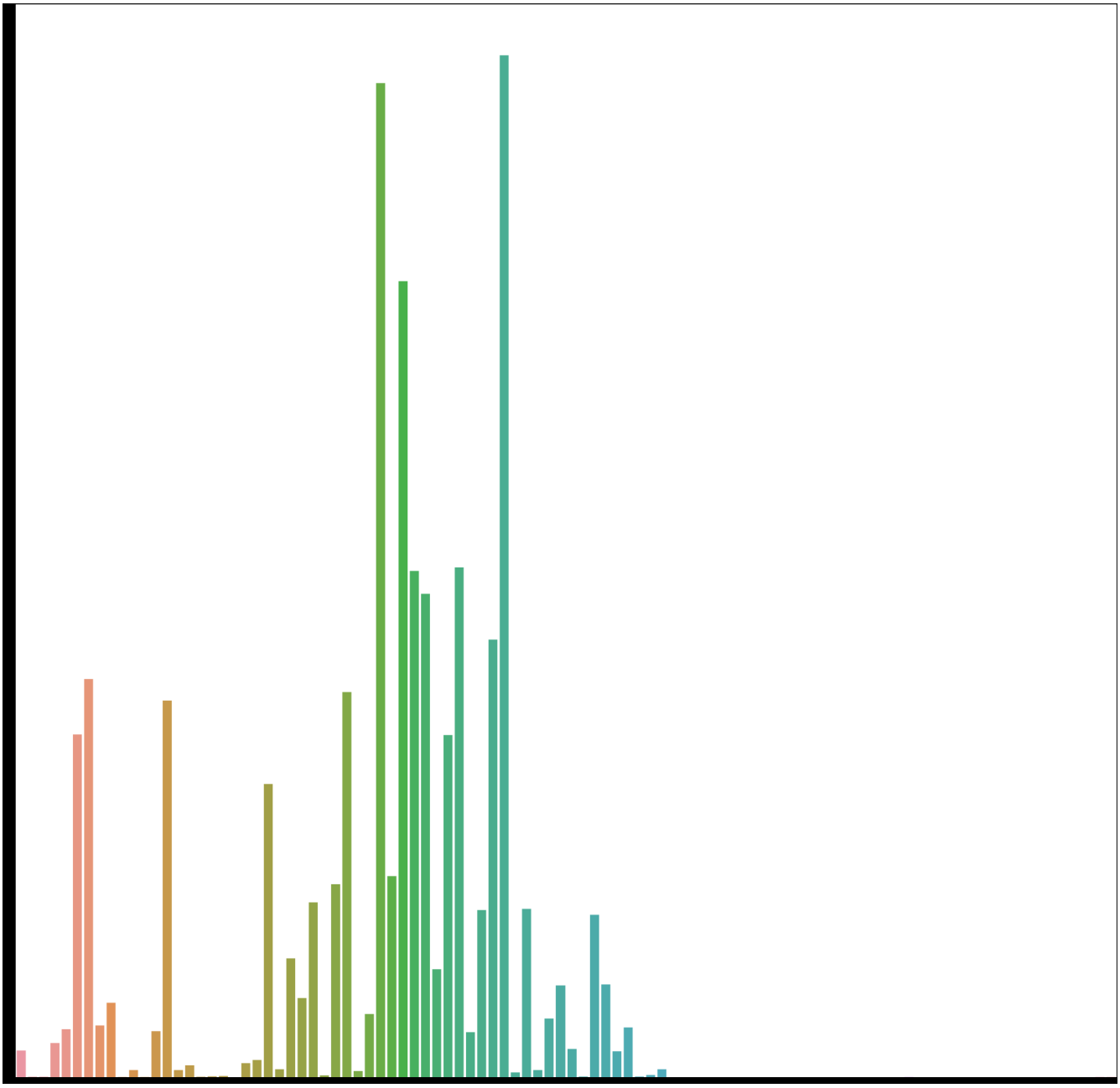
```
def preprocess(self):
    if not self.loaded:
        self.text.set("Error. Please load data from files")
        return

    self.Inspections = self.Inspections[self.Inspections['PROGRAM STATUS'] != 'INACTIVE']
    seating = []
    for description in self.Inspections['PE DESCRIPTION']:
        start = description.find('(')+1
        end = description.find(')')
        seating.append(description[start:end])

    self.Inspections['SEATING'] = seating
    self.Inspections['PE DESCRIPTION'] = self.Inspections['PE DESCRIPTION'].str.replace(r"\(.*\)","")

    seating_mean = self.Inspections.groupby('PE DESCRIPTION').mean()
    seating_median = self.Inspections.groupby('PE DESCRIPTION').median()
    seating_mode = self.Inspections.groupby('PE DESCRIPTION').agg(pd.Series.mode)

    zip_mean = self.Inspections.groupby('Zip Codes').mean()
    zip_median = self.Inspections.groupby('Zip Codes').median()
    zip_mode = self.Inspections.groupby('Zip Codes').agg(pd.Series.mode)

    self.text.set("Data preprocessed")
```

Producing the mean, mode, and median for inspection scores for each type of seating and zip code is done through merge operations with dataframes. The groupby method is called with the values "mean", "median", and "mode" to create new dataframes with the calculated values. The necessary columns are included. These dataframes can be joined for use in data visualizations. Results are printed in logs.

The data visualization function produces a graph of the number of establishments that commited each type of violation. A histogram is most appropriate with each category being represented by a bar of height equal to the number of establishments. The data count is discrete; a continuous plot wouldn't represent any significant information. A line graph or plot is thus inappropriate. The number of violations per category don't represent parts of one whole, nor would proportional representation give any valuable info. Thus, a pie graph is not a good fit.

```
    def visualize(self):
        if not self.loaded:
            self.text.set("Error. Please load data from files")
            return

        violations_count = self.violations.groupby('VIOLATION CODE').count()
        violations_count.reset_index(level=0, inplace=True)
        ax = sns.barplot(x='VIOLATION CODE', y='POINTS',
                    data = violations_count)
        #violations_count.index,y=violations_count["POINTS"])
        #plt.rcParams['figure.figsize'] = (100.0,100.0)
        plt.show()
        self.text.set("Bar Graph generated")
```

The violations dataframe is processed using the groupby method on the count method. Seaborn is imported and defined as sns to create the bar plot. The x axis is defined as the violation types and the y axis is defined as the number of occurences of each violation type. Colors are kept as the default. The initial size of the plot was too small to fit the x axis labels appropriately; they were overcrowded and overlapping. The size is adjusted with the matplotlib python library pyplot module. Pyplot is imported as plt. The method rcParams from plt is used to alter the figure.figsize parameter if needed so that the width of the plot is set to producing a readable plot in the output called through plt.show().



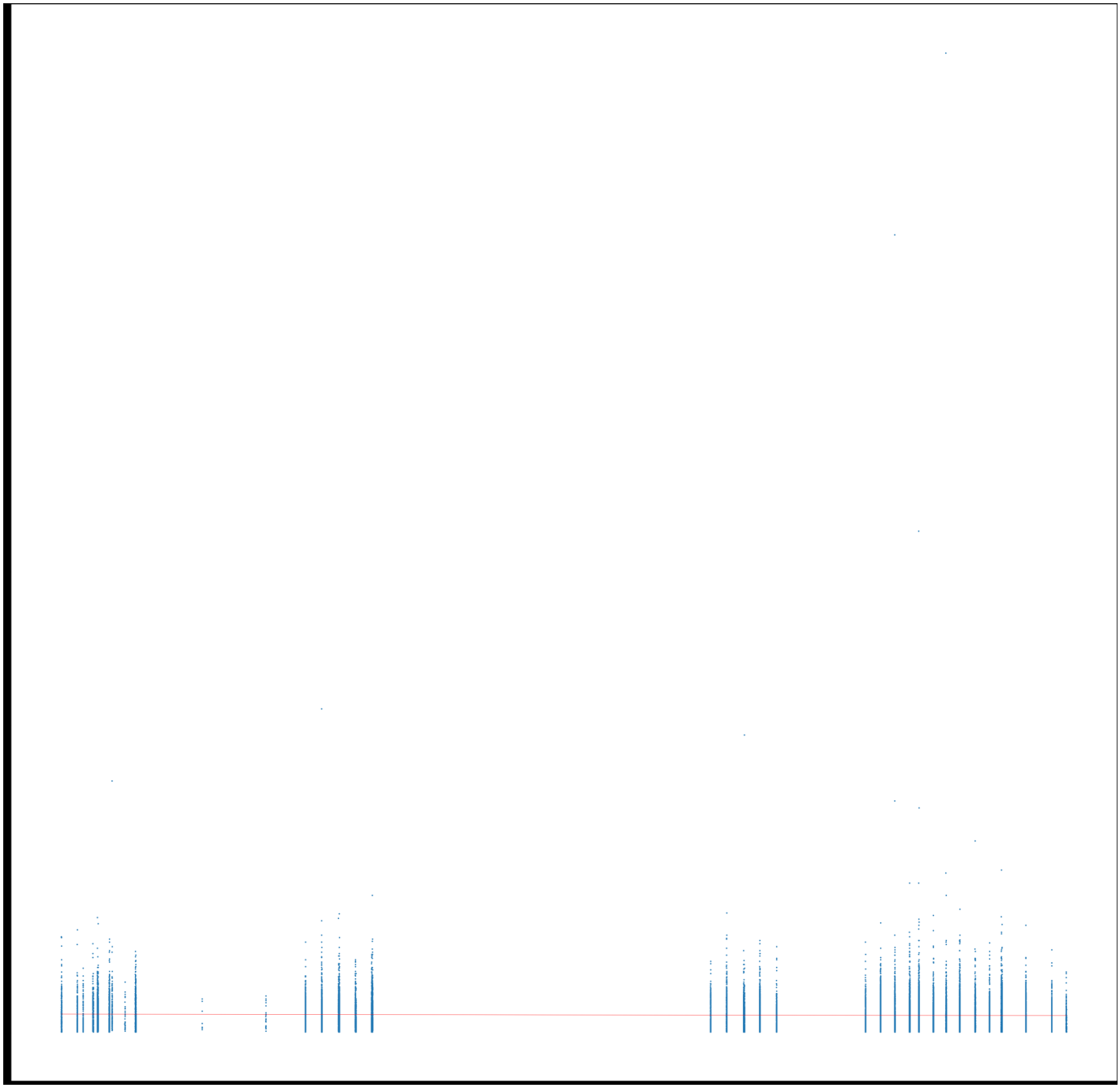
For the next specification, the program is asked whether there is a correlation between the number of violations commited per vendor and their zip code. A 2d scatterplot could be used. This program could make the assumption that zip codes close together in value are also close together geographically and have a significant relationship to each other. Therefore, by visualizing the data with a scatterplot and performing linear regression with a machine learning model, the user and or model can find any patterns if they are present.

Similarly to finding the number of vendors per violation code, a function is added to the class that counts the number of violations per vendor. The common feature of these tables that identifies inspections is the SERIAL NUMBER. The SERIAL NUMBER column joins dataframes in the first step. First, a dataframe of the number of violations per SERIAL NUMBER is constructed from the violations data set. A similar method to counting the number of violations per VIOLATION CODE is used. A dataframe is constructed with the groupby method with the count method on the POINTS column. The index is reset to make join operations simpler. Then a join on the SERIAL NUMBER of the Inspections dataframe including the FACILITY ID and Zip Code is used to find the number of violations per inspection.

```
def correlate(self):
    if not self.loaded:
        self.text.set("Error. Please load data from files")
        return

    violations_count = self.violations[['SERIAL NUMBER', 'POINTS']].groupby('SERIAL NUMBER').count()
    violations_count.reset_index(level=0, inplace=True)
    df = self.Inspections[['SERIAL NUMBER','Zip Codes','FACILITY ID']].set_index('SERIAL NUMBER').join(
        violations_count.set_index('SERIAL NUMBER'))
    df = df.dropna()
    df.reset_index(level=0, inplace=True)
    df = df.groupby(['Zip Codes','FACILITY ID']).sum().reset_index()
    df = df[['Zip Codes','POINTS']]

    #Linear Regression
    X = df.iloc[:, 0].values.reshape(-1, 1)
    Y = df.iloc[:, 1].values.reshape(-1, 1)
    linear_regressor = LinearRegression()
    linear_regressor.fit(X, Y)
    Y_pred = linear_regressor.predict(X)

    #Plot the linear regression
    plt.scatter(X, Y)
    plt.plot(X, Y_pred, color='red')
    plt.show()

    self.text.set("Scatter Plot generated")
```

The result of this join is a dataframe containing a column for zip codes for each inspection and the number of violations per inspection. The inspections must be grouped with the vendors so an additional groupby is used that calculates the sum of the POINTS column. To clean the data, dropna is used for empty values (inpections with no violations or bad data). The dataframe now contains a column for zip codes beside a column for number of violations with each row representing a vendor

A machine learning linear regression model is used through scikitlearn. For this purpose, the program imports numpy as np and LinearRegression from sklearn.linear_model. Dataframe series are extracted for the X variable to be used for input and the Y variable to learn predictions. The iloc method serves this purpose by taking all row values with ":" and the first or second column with "0" or "1". The LinearRegression class is called to create a linear regressor object. The X and Y data is fitted, then predictions are made for the regressor and stored in Y_pred. The program constructs a scatter plot of points representing each establishment. The X values represent zip codes and the Y values represent the number of violations. The linear regression predictions are represented by a red line. The regression line is flat across the graph, indicating no continuous linear patterns between zip codes and number of violations.



Section 3: Reflection on the ethics, moral and legal aspects (10% 400 words)

Evidence for learning outcome: Critically evaluate the legal and ethical impact of software developments within real world contexts.

Decisions by humans about computers affect people's lives [1]. Properly implementing ethical evaluations to a proper standard has the ability to greatly affect users or even the lives of people in society at large. An article by Burt et al explains the impact of food safety: "government bears a huge responsibility to ensure proper regulation of the safety practices of... vendors" [5]. With new information provided by computer programs, policies or decisions must be rethought ethically and addressed early to minimize the amount of people negatively affected. In a study by Jones et al [6], restaurant inspection scores in the state of Tennessee and data patterns drawn from these scores provide evidence for their conclusion about restaurant safety in their area. For example, mean scores of restaurants experiencing foodborne disease outbreaks did not differ from restaurants with no reported outbreaks. Using software, numbers of violations can be narrowed down to common patterns within a particular area.

In a study by Schomberg et al [7], data collected from social media, Yelp, is fed into a predictive algorithm that is able to predict health code violations in 78% of restaurants that received serious citations. Certainly, with health and safety being a priority, any reliable source of information should be utilized to prevent illness or death if such an accurate predictive model is available. The issue of user privacy could be addressed with an agreement from reviewers to have their data mined before creating an account or posting reviews, in accordance with legislation such as the UK's Data Protection Act [12]. Yelp reviews hold influence over customer flow to establishments and can help or damage businesses, with each ratings star translating to anywhere between a 5-9% effect on revenues [11]. Extending Yelp's influence to determine administrative action and law enforcement would give Yelp political power with its decisions affecting widespread health standards. Similar issues are already present with services such as Twitter holding significant sway on policy and shaping information for decision making at a large scale [13]. If powerful tools for collecting data can be used for improving and saving lives, it is ethical to use these tools to maximum effect. Fair ownership for these tools cannot be easily solved and will be approached differently based an local laws until global societies are able to come to a consensus on the distribution of wealth and influence across a worldwide network.

Citations:

[1] Gotterbarn, Donald. "Software engineering ethics." *Encyclopedia of Software Engineering* (2002).

[2] Sebesta, Robert W. *Concepts of programming languages*. Pearson Education, Inc, 2012.

[3] Nielsen, Jakob. "Ten usability heuristics." (2005)

[4] Prechelt, Lutz. "An empirical comparison of seven programming languages." *Computer* 33.10 (2000): 23-29.

[5] G. D. Service, “Data protection,” *GOV.UK*, 16-Sep-2015. [Online]. Available: https://www.gov.uk/data-protection. [Accessed: 11-Dec-2020].

[6] Jones, Timothy F., et al. "Restaurant inspection scores and foodborne disease." *Emerging Infectious Diseases* 10.4 (2004): 688.

[7] Schomberg, John P., et al. "Supplementing public health inspection via social media." *PloS one* 11.3 (2016): e0152117.

[8] Raff, Edward. "JSAT: Java statistical analysis tool, a library for machine learning." *The Journal of Machine Learning Research* 18.1 (2017): 792-796.

[9] Koliopoulos, Aris-Kyriakos, et al. "A parallel distributed weka framework for big data mining using spark." *2015 IEEE international congress on big data*. IEEE, 2015.

[10] Roscoe, Bill. "The theory and practice of concurrency." (1998).

[11] Blanding, Michael. "The Yelp factor: Are consumer reviews good for business." *Harvard School of Business* (2011).

[12] Service, Government Digital. “Data Protection.” *GOV.UK*, GOV.UK, 16 Sept. 2015, www.gov.uk/data-protection.

[13] Puschmann, Cornelius, and Jean Burgess. "The politics of Twitter data." (2013).