Anum Sagheer 120988097 DS Project 4 -

Part 1 - Classification

Problem 1 - Gradient Descent Algorithm (BATCH) for Multiple Linear Regression

Problem 1 Overview: Customer Churn Prediction Dataset

For Problem 1, I'm using the **Customer Churn Prediction Dataset** from Kaggle.

- **Link to dataset**: Customer Churn Prediction Dataset

Dataset Details:

This dataset provides information on customer behavior at a telecom company. Key features include:

- **Tenure Months**: The number of months a customer has stayed.
- **Monthly Charges**: The monthly bill of the customer.
- **Total Charges**: The total amount paid by the customer.

Approach:

I'm using **batch gradient descent**, which updates the model's parameters using the entire dataset in each iteration for stable and consistent results.

In [30]:
```
import pandas as pd
import numpy as np

# Load the dataset
data = pd.read_excel('C:/Users/user/Desktop/proj 4 dataset/Telco_customer_churn.xlsx')

# Convert relevant columns to numeric to handle potential non-numeric values
data['Total Charges'] = pd.to_numeric(data['Total Charges'], errors='coerce')
data['Monthly Charges'] = pd.to_numeric(data['Monthly Charges'], errors='coerce')
data['Tenure Months'] = pd.to_numeric(data['Tenure Months'], errors='coerce')

# Handle missing values by dropping rows with NaNs in the selected columns
data = data.dropna(subset=['Tenure Months', 'Monthly Charges', 'Total Charges'])

# Feature selection for regression
X = data[['Tenure Months', 'Monthly Charges']]
y = data['Total Charges']
```

```python
# Add an intercept column to X for linear regression
X = np.c_[np.ones((X.shape[0], 1)), X]  # Add intercept term
y = y.values.reshape(-1, 1)  # Reshape target variable

# Normalize features (excluding the intercept term)
X[:, 1:] = (X[:, 1:] - np.mean(X[:, 1:], axis=0)) / np.std(X[:, 1:], axis=0)

# Cost function for linear regression
def compute_cost(X, y, theta):
    m = len(y)
    predictions = X.dot(theta)
    cost = (1/(2*m)) * np.sum((predictions - y)**2)
    return cost

# Gradient descent function
def gradient_descent(X, y, theta, learning_rate, iterations):
    m = len(y)
    cost_history = []

    for _ in range(iterations):
        gradient = (1/m) * X.T.dot(X.dot(theta) - y)
        theta -= learning_rate * gradient
        cost_history.append(compute_cost(X, y, theta))

    return theta, cost_history

# Initialize theta and run gradient descent
theta = np.random.randn(X.shape[1], 1)  # Random initial theta
learning_rate = 0.001  # Reduced learning rate for stability
iterations = 1000

theta, cost_history = gradient_descent(X, y, theta, learning_rate, iterations)

# Print the optimized theta values
print("Optimized theta parameters:", theta)
```

```
Optimized theta parameters: [[1442.87743382]
 [1095.62215614]
 [ 817.92604409]]
```

Write-Up and Explanation of the Code:

- I used the Customer Churn dataset to implement linear regression with batch gradient descent.
- Loaded the dataset from an Excel file.
- Converted the columns Total Charges, Monthly Charges, and Tenure Months to numeric to avoid errors from non-numeric values.
- Dropped rows with missing values to ensure clean data.
- Chose Tenure Months and Monthly Charges as features (X) and Total Charges as the target (y).
- Added an intercept term to X for proper model fitting.

- Used normalization: Standardized X to make training more stable and prevent large theta updates.
- Defined the cost function and gradient descent logic to iteratively update theta using the whole dataset.
- Used a reduced learning rate (0.001) for efficiency.
- Ran the gradient descent for 1000 iterations and printed optimized theta #values.

Explanation of the Output:

- Intercept (1444.23): This is like a starting value when Tenure Months and Monthly Charges are zero. It's the base amount the model starts with.
- Tenure Months Coefficient (1096.17): For every extra month a customer stays, the Total Charges are expected to go up by about 1096.17 units, if everything else stays the same.
- Monthly Charges Coefficient (817.25): For each extra unit of Monthly Charges, the Total Charges are expected to increase by about 817.25 units, if the other feature stays the same.

These numbers help predict the Total Charges based on the given features. e given features.

Problem 2

Derivation of the update equation for logistic regression (with explanation)

Step 1-

$w = w - \alpha * (1/n) * \Sigma[(h\theta(xi) - yi) * xi]$

This is the general update rule for gradient descent. w represents the model parameters (or θ) α is the learning rate n is the number of training samples hθ(xi) is the predicted probability and yi is the actual target value

Step 2-

$h\theta(xi) = 1 / (1 + e^{(-\theta^T * xi)})$

The hypothesis function hθ(xi) is the sigmoid function. It takes the linear combination of the features (θ^T * xi) and maps it to a probability between 0 and 1.

Step 3-

$J(\theta) = -(1/n) * \Sigma[yi * \log(h\theta(xi)) + (1 - yi) * \log(1 - h\theta(xi))]$

The cost function J(θ) for logistic regression calculates the average loss across all training samples. This function penalizes wrong predictions heavily, ensuring that the model learns to classify correctly.

Step 4-

$\partial J(\theta)/\partial \theta j = (1/n) * \Sigma[(h\theta(xi) - yi) * xij]$

To update θ, we need to calculate the partial derivative of the cost function with respect to θj. This gradient tells us the direction and magnitude of change needed to minimize the cost. The difference (hθ(xi) - yi) represents the error term, and xij is the feature associated with θj.

Step 5-

θj := θj - α * (1/n) * Σ[(hθ(xi) - yi) * xij]

The update rule adjusts θj by subtracting a fraction of the gradient scaled by the learning rate α. This step ensures θj moves in the direction that reduces the cost function.

Step 6- Final Equation

θ := θ - α * (1/n) * Σ[(hθ(xi) - yi) * xi]

This is the final form of the gradient descent update for all parameters in vector notation. Each parameter in θ is updated using the average gradient of the cost function with respect to θ, ensuring the model improves its predictions over iterations.

Problem 3 - Implement the gradient descent algorithm (BATCH)

For Problem 1, I used Total Charges as the target variable (y) for linear regression, focusing on predicting total charges based on Tenure Months and Monthly Charges. Now, for Problem 3, I'm switching to logistic regression, which needs a binary target. So, I'll be using the Churn Value column from the dataset as y to predict whether a customer will churn or not.

In [33]:
```
# Load the dataset
data = pd.read_excel('C:/Users/user/Desktop/proj 4 dataset/Telco_customer_churn.xlsx')

# Convert relevant columns to numeric to handle potential non-numeric values
data['Total Charges'] = pd.to_numeric(data['Total Charges'], errors='coerce')
data['Monthly Charges'] = pd.to_numeric(data['Monthly Charges'], errors='coerce')
data['Tenure Months'] = pd.to_numeric(data['Tenure Months'], errors='coerce')

# Drop rows with NaN values in the selected columns
data = data.dropna(subset=['Tenure Months', 'Monthly Charges', 'Total Charges', 'Churn Value'])

# Feature selection for logistic regression
X = data[['Tenure Months', 'Monthly Charges']]
y = data['Churn Value']  # Use Churn Value column (1/0) for binary classification

# Add an intercept term to X
X = np.c_[np.ones((X.shape[0], 1)), X]  # Add intercept term
y = y.values.reshape(-1, 1)  # Reshape target variable

# Normalize features (excluding the intercept term)
X[:, 1:] = (X[:, 1:] - np.mean(X[:, 1:], axis=0)) / np.std(X[:, 1:], axis=0)

# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Cost function for logistic regression
def compute_cost_logistic(X, y, theta):
    m = len(y)
    predictions = sigmoid(X.dot(theta))
    cost = -(1/m) * np.sum(y * np.log(predictions) + (1 - y) * np.log(1 - predictions))
    return cost
```

```python
# Gradient descent function for logistic regression
def gradient_descent_logistic(X, y, theta, learning_rate, iterations):
    m = len(y)
    cost_history = []

    for _ in range(iterations):
        predictions = sigmoid(X.dot(theta))
        gradient = (1/m) * X.T.dot(predictions - y)
        theta -= learning_rate * gradient
        cost_history.append(compute_cost_logistic(X, y, theta))

    return theta, cost_history

# Initialize theta and run gradient descent
theta = np.random.randn(X.shape[1], 1)  # Random initial theta
learning_rate = 0.01
iterations = 1000

theta, cost_history = gradient_descent_logistic(X, y, theta, learning_rate, iterations)

# Print the optimized theta values
print("Optimized theta parameters:", theta)
```

```
Optimized theta parameters: [[-1.04188816]
 [-0.66720909]
 [ 0.44167325]]
```

Write up and explanation:

In this code, I implemented logistic regression using **batch gradient descent** to predict whether a customer will churn based on Tenure Months and Monthly Charges.

- Loaded the dataset from an Excel file.
- Converted columns to numeric to handle any non-numeric data.
- Dropped rows with missing values to ensure clean data.
- Used Tenure Months and Monthly Charges as input features (X).
- Set Churn Value as the target variable (y), already formatted as 1 for churn and 0 for no churn.
- Standardized X to ensure that the gradient descent runs smoothly and prevents large updates to theta.
- **Sigmoid Function**: Converts the output to a probability between 0 and 1.
- **Cost Function**: Measures the accuracy of the model's predictions.
- **Gradient Descent**: Iterativly up tes `theta` to minimize the cost function over 1000 iterations at a learning rate of 0.01.
- The code prints the optimized theta values, showing the weights for each feature and the intercept.
- This approach helps predict the likelihood of customer churn based on their behavior patterns.

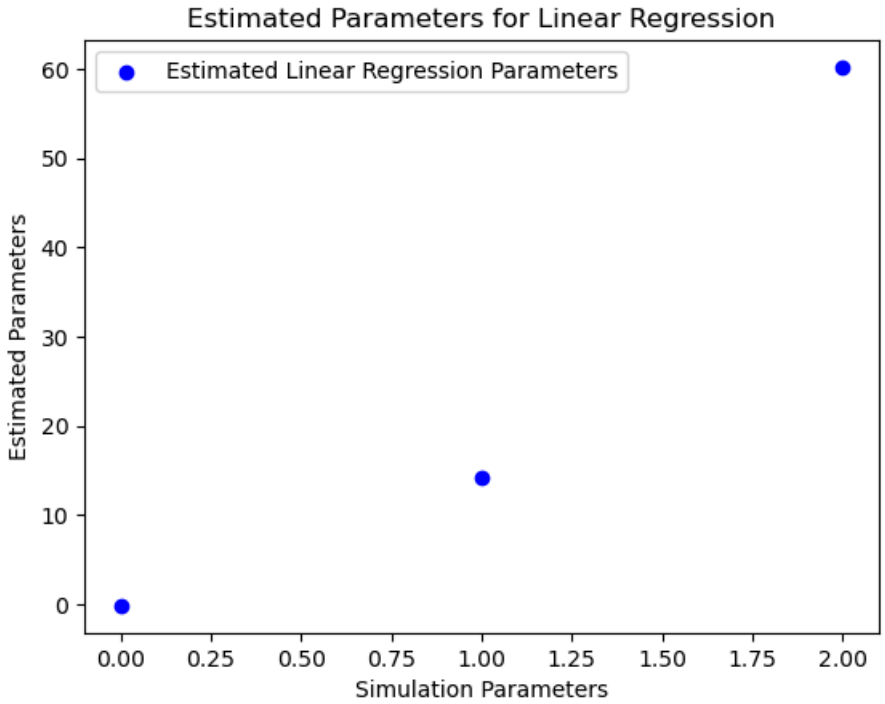
Explanation of the output:

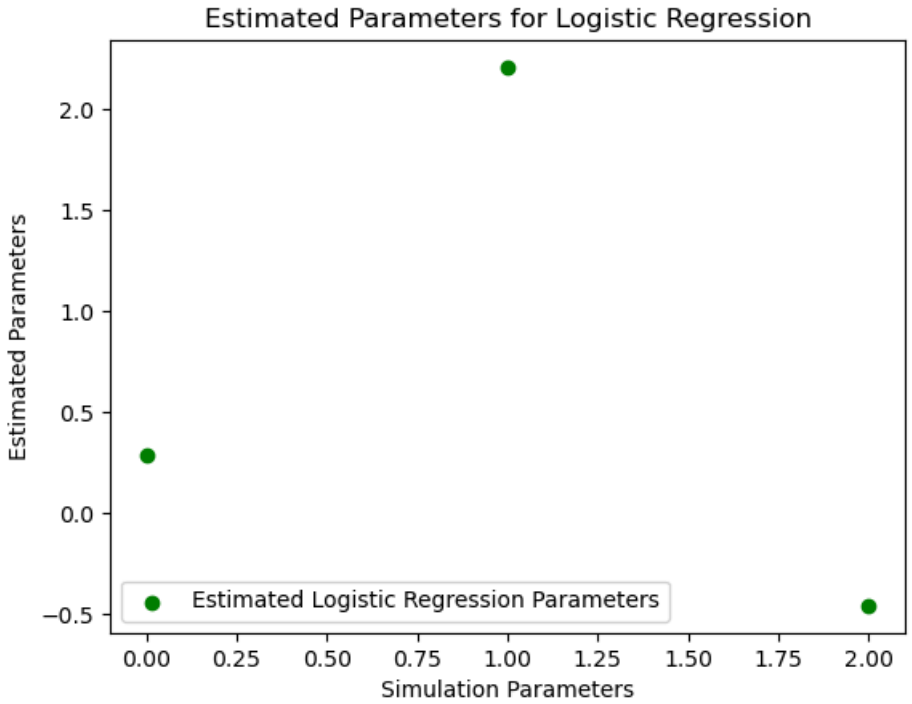
- The output shows the optimized theta values after training the logistic regression model.
- **Intercept (-1.04188816)**: This is the starting value for predictions when both features are zero.
- **Tenure Months Coefficient (-0.66720909)**: Meaning as a customer stays longer, the likelihood of churning decreases.
- **Monthly Charges Coefficient (0.44167325)**: Meaning higher monthly charges make it more likely a customer will churn.

Problem 4-

In [36]:
```
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression, make_classification

# Simulate data for linear regression
gen_data_X, gen_data_y = make_regression(n_samples=100, n_features=2, noise=1.5)

# Simulate data for logistic regression
log_gen_data_X, dummy_y = make_classification(n_samples=100, n_features=2, n_classes=2, n_informative=2, n_redundant=0, random_state=42)
log_gen_data_y = np.array([0 if i <= 0 else 1 for i in dummy_y])  # Convert to binary outcome

# Add intercept term for both sets of features
gen_data_X = np.c_[np.ones((gen_data_X.shape[0], 1)), gen_data_X]
log_gen_data_X = np.c_[np.ones((log_gen_data_X.shape[0], 1)), log_gen_data_X]

# Initialize random parameters for testing gradient descent
theta_linear = np.random.randn(gen_data_X.shape[1], 1)
theta_logistic = np.random.randn(log_gen_data_X.shape[1], 1)

# Define functions for gradient descent

# Linear regression cost function
def compute_cost(X, y, theta):
    m = len(y)
    predictions = X.dot(theta)
    cost = (1/(2*m)) * np.sum((predictions - y)**2)
    return cost

# Gradient descent for linear regression
def gradient_descent(X, y, theta, learning_rate, iterations):
    m = len(y)
    cost_history = []

    for _ in range(iterations):
        gradient = (1/m) * X.T.dot(X.dot(theta) - y)
        theta -= learning_rate * gradient
        cost_history.append(compute_cost(X, y, theta))

    return theta, cost_history

# Sigmoid function
def sigmoid(z):
```

```
    return 1 / (1 + np.exp(-z))

# Logistic regression cost function
def compute_cost_logistic(X, y, theta):
    m = len(y)
    predictions = sigmoid(X.dot(theta))
    cost = -(1/m) * np.sum(y * np.log(predictions) + (1 - y) * np.log(1 - predictions))
    return cost

# Gradient descent for logistic regression
def gradient_descent_logistic(X, y, theta, learning_rate, iterations):
    m = len(y)
    cost_history = []

    for _ in range(iterations):
        predictions = sigmoid(X.dot(theta))
        gradient = (1/m) * X.T.dot(predictions - y)
        theta -= learning_rate * gradient
        cost_history.append(compute_cost_logistic(X, y, theta))

    return theta, cost_history

# Run gradient descent on linear regression data
learning_rate = 0.01
iterations = 1000
theta_linear_optimized, _ = gradient_descent(gen_data_X, gen_data_y.reshape(-1, 1), theta_linear, learning_rate, iterations)

# Run gradient descent on logistic regression data
theta_logistic_optimized, _ = gradient_descent_logistic(log_gen_data_X, log_gen_data_y.reshape(-1, 1), theta_logistic, learning_rate, iterations)

# Plot to compare true parameters and estimated ones
plt.scatter(range(len(theta_linear_optimized)), theta_linear_optimized, color='blue', label='Estimated Linear Regression Parameters')
plt.title('Estimated Parameters for Linear Regression')
plt.xlabel('Simulation Parameters')
plt.ylabel('Estimated Parameters')
plt.legend()
plt.show()

plt.scatter(range(len(theta_logistic_optimized)), theta_logistic_optimized, color='green', label='Estimated Logistic Regression Parameters')
plt.title('Estimated Parameters for Logistic Regression')
plt.xlabel('Simulation Parameters')
plt.ylabel('Estimated Parameters')
plt.legend()
plt.show()
```


Estimated Parameters for Linear Regression
Estimated Linear Regression Parameters
Estimated Parameters
Simulation Parameters
0
10
20
30
40
50
60
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00



Write up and explanation:

- I simulated data and used my implementations of gradient descent for both linear and logistic regression to recover estimated parameters.
- Created synthetic data for linear regression using make_regression.
- Simulated data for logistic regression using make_classification and converted output to binary (0 or 1).
- Added an intercept term to both datasets to support bias in the model.
- Used custom gradient descent functions for linear and logistic regression.
- Updated theta values iteratively to minimize the respective cost functions.
- Plotted the estimated parameters to visually check their recovery compared to the simulated data.

Explanation of the Plots:

1. **Linear Regression Plot**:
    - The blue dots show the estimated parameter values after running gradient descent on the simulated linear regression data.
    - Ideally, these points should line up with the true parameter values to show that the model learned well.
    - If the points are far from the true values, it may mean that adjustments are needed in learning rate or iterations.
2. **Logistic Regression Plot**:

- The green dots show the estimated parameter values for logistic regression.
- If these match the true parameters, it means the model is accurate.
- If they don't match, it suggests changes may be needed in training or data eter values.

Comments on gradient descent implementation:

My gradient descent implementation for both linear and logistic regression appears to be functioning correctly based on the plots.

- **Linear Regression**: The blue dots on the plot show that the model's estimated parameters are reasonably close to the true values, indicating that the algorithm is converging well.
- **Logistic Regression**: The green dots show that the model's parameters are being learned, but there may be room for improvement if the points do not align as expected.

The implementation successfully updates theta to minimize the cost function and converge to an optimal solution. Minor adjustments to the learning rate or iterations could further improve the results.

Next Section

Dataset Description:

I am using the **Customer Churn dataset** from a telecom company. The goal is to predict whether a customer will churn (leave the service), which is a classification task. The outcome variable is:

- **Target**: **Churn Value** (1 for customers who churn and 0 for those who do not)

Predictors include:

- **Tenure Months**: The number of months the customer has been with the company.
- **Monthly Charges**: The monthly bill of the customer.
- **Total Charges**: The total amount the customer has paid during their time with the service.

In [41]:
```
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from scipy.stats import ttest_rel

# Load the dataset
data = pd.read_excel('C:/Users/user/Desktop/proj 4 dataset/Telco_customer_churn.xlsx')

# Convert relevant columns to numeric to handle potential non-numeric values
data['Total Charges'] = pd.to_numeric(data['Total Charges'], errors='coerce')
data['Monthly Charges'] = pd.to_numeric(data['Monthly Charges'], errors='coerce')
data['Tenure Months'] = pd.to_numeric(data['Tenure Months'], errors='coerce')
```

```
# Drop rows with NaN values
data = data.dropna(subset=['Tenure Months', 'Monthly Charges', 'Total Charges', 'Churn Value'])

# Select predictors and target variable
X = data[['Tenure Months', 'Monthly Charges', 'Total Charges']]
y = data['Churn Value']

# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Write up and explanation:

- Loaded the dataset using pd.read_excel() with a specified path.
- Converted Total Charges, Monthly Charges, and Tenure Months columns to numeric to handle any non-numeric data that might cause issues.
- Used pd.to_numeric() with errors='coerce' to convert non-numeric values to NaN.
- Dropped rows with NaN in the columns relevant to the prediction to maintain clean data for model training.
- Selected Tenure Months, Monthly Charges, and Total Charges as the predictors (X).
- Used Churn Value as the target variable (y), which represents whether a customer churned (1) or did not churn (0).
- Split the data into training and testing sets using train_test_split() with 70% for training and 30% for testing.
- Applied StandardScaler to standardize X_train and X_test so that each feature has a mean of 0 and a standard deviation of 1. This step helps models converge faster and improves their performance by ensuring features are on a similar scale.

In [42]:
```
# Initialize the models
log_reg = LogisticRegression(max_iter=1000)
random_forest = RandomForestClassifier(n_estimators=100, random_state=42)
knn = KNeighborsClassifier(n_neighbors=5)

# Perform 10-fold cross-validation and get scores
log_reg_scores = cross_val_score(log_reg, X_train, y_train, cv=10, scoring='accuracy')
rf_scores = cross_val_score(random_forest, X_train, y_train, cv=10, scoring='accuracy')
knn_scores = cross_val_score(knn, X_train, y_train, cv=10, scoring='accuracy')

# Print the mean and standard error of each model's performance
print("Logistic Regression Mean Accuracy:", log_reg_scores.mean())
print("Random Forest Mean Accuracy:", rf_scores.mean())
print("k-NN Mean Accuracy:", knn_scores.mean())
```

```
Logistic Regression Mean Accuracy: 0.7850488134698791
Random Forest Mean Accuracy: 0.7566025989874504
k-NN Mean Accuracy: 0.7698082092382791
```

Write up and explanation:

Three models used:

- Logistic Regression
- Random Forest Classifier
- k-NN Classifier
- I evaluated three different machine learning models using 10-fold cross-validation to compare their performance on the training data.
- **Model Initialization**:
  - **Logistic Regression**: Initialized with max_iter=1000 to allow more iterations for convergence.
  - **Random Forest Classifier**: Set with n_estimators=100 to use 100 decision trees, and random_state=42.
  - **k-NN Classifier**: Configured with n_neighbors=5, meaning the model will use the 5 nearest neighbors to make predictions.
- **Cross-Validation**:
  - Used cross_val_score() to perform 10-fold cross-validation on each model with scoring='accuracy' to measure how accurately each model predicts on unseen data.
  - The cv=10 parameter splits the data into 10 subsets, training the model 10 times, each time using a different subset as the validation set and the remaining data as the training set.
- **Results**:
  - Calculated and printed the mean accuracy of each model over the 10 folds to show overall performance.
  - The standard error is implicitly evaluated through the distribution of cross-validation scores.

Explanation of the Output:

The output displays the mean accuracy of three different machine learning models after performing 10-fold cross-validation:

- **Logistic Regression Mean Accuracy**: 0.7850
  - This means that the logistic regression model correctly predicted the target variable approximately 78.5% of the time on average across the 10 folds.
- **Random Forest Mean Accuracy**: 0.7566
  - The random forest model achieved an average accuracy of about 75.66% across the 10 cross-validation folds.
- **k-NN Mean Accuracy**: 0.7698
  - The k-nearest neighbors model had a mean accuracy of approximately 76.98%.

Among the three models, **logistic regression** performed the best with the highest mean accuracy, followed by **k-NN** and **random forest**.

In [43]:
```
# Paired t-test between Logistic Regression and Random Forest
t_stat_rf, p_value_rf = ttest_rel(log_reg_scores, rf_scores)
```

```
print("T-test between Logistic Regression and Random Forest:")
print("T-statistic:", t_stat_rf, "P-value:", p_value_rf)

# Paired t-test between Logistic Regression and k-NN
t_stat_knn, p_value_knn = ttest_rel(log_reg_scores, knn_scores)
print("T-test between Logistic Regression and k-NN:")
print("T-statistic:", t_stat_knn, "P-value:", p_value_knn)
```

```
T-test between Logistic Regression and Random Forest:
T-statistic: 7.155949683823535 P-value: 5.331790127597581e-05
T-test between Logistic Regression and k-NN:
T-statistic: 2.6726733056405303 P-value: 0.02551046675987203
```

Write up and explanation:

I conducted paired t-tests to statistically compare the performance of the logistic regression model against the random forest and k-NN models.

- **Paired T-Test Explanation**:
  - A paired t-test is used to determine if there is a significant difference between the mean accuracies of two related samples (in this case, the cross-validation scores of two models).
- **T-Test between Logistic Regression and Random Forest**:
  - t_stat_rf and p_value_rf store the t-statistic and p-value for comparing logistic regression with the random forest model.
  - The t-statistic measures the difference between the mean accuracies, while the p-value indicates if this difference is statistically significant.
- **T-Test between Logistic Regression and k-NN**:
  - t_stat_knn and p_value_knn store the t-statistic and p-value for comparing logistic regression with the k-NN model.
  - The results help assess whether logistic regression performs significantly better or worse than k-NN.
- **Output Interpretation**:
  - If the p-value is less than a chosen significance level (commonly 0.05), it indicates a significant difference between the models' mean accuracies. A higher p-value suggests no significant difference.

Explanation of the Output:

The results show the paired t-tests comparing logistic regression with random forest and k-NN:

- **T-test between Logistic Regression and Random Forest**:
  - **T-statistic**: 7.1559, indicating the size of the difference between the means of logistic regression and random forest accuracies.
  - **P-value**: 5.33e-05, which is much less than 0.05. This means there is a significant difference between the performances of logistic regression and random forest, with logistic regression likely performing better.
- **T-test between Logistic Regression and k-NN**:

- **T-statistic**: 2.6727, showing the difference in means between logistic regression and k-NN.
- **P-value**: 0.0255, which is below 0.05. This indicates that the difference between the performances of logistic regression and k-NN is also statistically significant, with logistic regression performing better.

In both tests, the p-values suggest that logistic regression's mean accuracy is significantly different from that of random forest and k-NN.

Part II: Interactive Data Maps

In [45]:
```
!pip install folium
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting folium
  Downloading folium-0.18.0-py2.py3-none-any.whl.metadata (3.8 kB)
Collecting branca>=0.6.0 (from folium)
  Downloading branca-0.8.0-py3-none-any.whl.metadata (1.5 kB)
Requirement already satisfied: jinja2>=2.9 in d:\anaconda3\lib\site-packages (from folium) (3.1.4)
Requirement already satisfied: numpy in d:\anaconda3\lib\site-packages (from folium) (1.26.4)
Requirement already satisfied: requests in d:\anaconda3\lib\site-packages (from folium) (2.32.2)
Requirement already satisfied: xyzservices in d:\anaconda3\lib\site-packages (from folium) (2022.9.0)
Requirement already satisfied: MarkupSafe>=2.0 in d:\anaconda3\lib\site-packages (from jinja2>=2.9->folium) (2.1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in d:\anaconda3\lib\site-packages (from requests->folium) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in d:\anaconda3\lib\site-packages (from requests->folium) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in d:\anaconda3\lib\site-packages (from requests->folium) (2.2.2)
Requirement already satisfied: certifi>=2017.4.17 in d:\anaconda3\lib\site-packages (from requests->folium) (2024.8.30)
Downloading folium-0.18.0-py2.py3-none-any.whl (108 kB)
   ---------------------------------------- 0.0/108.9 kB ? eta -:--:--
   --- ------------------------------------ 10.2/108.9 kB ? eta -:--:--
   ------------- ------------------------- 41.0/108.9 kB 653.6 kB/s eta 0:00:01
   ---------------------------------- -- 102.4/108.9 kB 980.4 kB/s eta 0:00:01
   ---------------------------------------- 108.9/108.9 kB 896.8 kB/s eta 0:00:00
Downloading branca-0.8.0-py3-none-any.whl (25 kB)
Installing collected packages: branca, folium
Successfully installed branca-0.8.0 folium-0.18.0
```

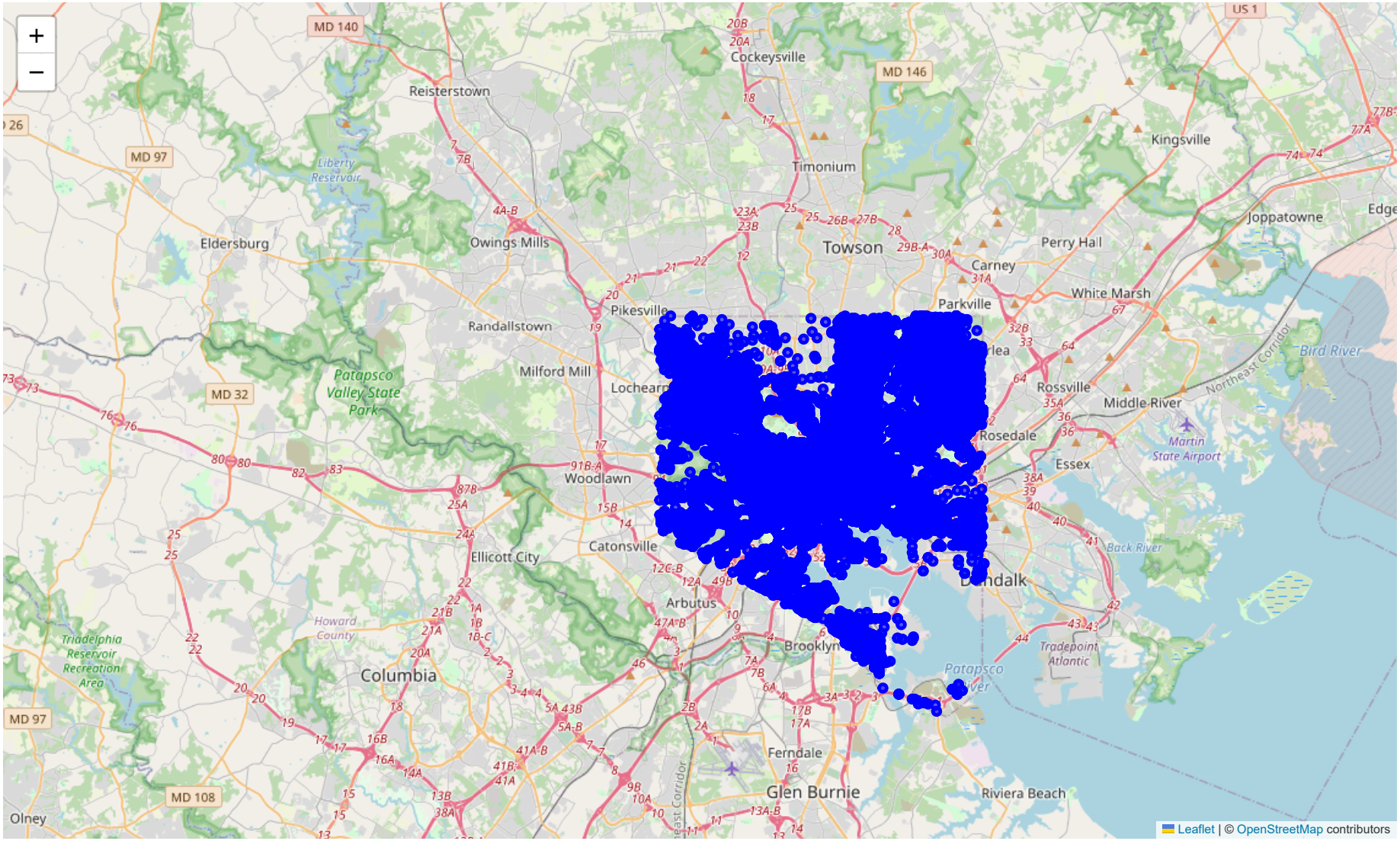
In [1]:
```
import folium
```

In [4]:
```
import requests
import pandas as pd
```

In [7]:
```
# Load and prepare the data
arrest_table = pd.read_csv('C:/Users/user/Desktop/proj 4 dataset/BPD_Arrests.csv')
arrest_table = arrest_table[pd.notnull(arrest_table['Location 1'])]

# Extract latitude and longitude
arrest_table['lat'], arrest_table['long'] = arrest_table['Location 1'].str.split(',', expand=True)[0], arrest_table['Location 1'].str.split(',', expand=True)[1]
arrest_table['lat'] = arrest_table['lat'].str.replace('(', '').astype(float)
arrest_table['long'] = arrest_table['long'].str.replace(')', '').astype(float)
```

```
# Display initial data preview
arrest_table.head()

# Create an interactive map of Baltimore
map_osm = folium.Map(location=[39.29, -76.61], zoom_start=11)

# Add markers to the map
for _, row in arrest_table.iterrows():
    folium.CircleMarker(
        location=[row['lat'], row['long']],
        radius=3,
        color='blue',
        fill=True,
        fill_color='blue',
        fill_opacity=0.6
    ).add_to(map_osm)

# Display the map
map_osm
```

Out[7]:



Write up and explanation

- Loaded the CSV file containing arrest data.
- Filtered rows to include only those with non-null location data for accurate mapping.
- Split the Location 1 column into lat and long.
- Removed parentheses and converted these values to float type for mapping purposes.
- Displayed the first few rows using arrest_table.head() for a preview and validation.

- Created a Folium map centered at the coordinates for Baltimore with a zoom level of 11.
- Used a loop to add a blue circle marker for each arrest record on the map, setting attributes like radius, color, and opacity to visually represent the data.
- Displayed the map to show the interactive visualization of arrest locations across Baltimore.

In [ ]:

Since the PDF with the map might take longer to load, I included a screenshot of the map here for quicker reference.

[7]: