New York Dog Bite Analysis

Why Interest

My growing interest in dog ownership led me to explore the New York Dog Bites dataset. I believe that responsible dog ownership involves not only caring for the well-being of the pet but also minimizing any negative impact on the community. This dataset raises important questions about responsibility in dog bite incidents, whether involving owned or stray dogs and highlights the broader societal issue of balancing pet welfare with public safety. It underscores the need for a harmonious coexistence between pets and people that requires the attention of both residents and governments.

Project Purpose

The primary objective of this project is to analyze dog bite incidents in New York using a dataset. The analysis focuses on understanding the factors that contribute to these incidents by examining geographic distribution, temporal trends, breed and age distribution, and the impact of gender and spay/neuter status. The findings are intended to provide actionable insights for public safety and promoting responsible dog ownership.

Data Scope and Analysis Dimensions

1. Dataset Source: Kaggle – "Predict New York Dog Bites" (https://www.kaggle.com/datasets/stealthtechnologies/predict-new-york-dog-bites)
2. Data Collector: Umair Zia
3. Original Source: data.world

4. Author: Andy

Variables Included:

1. Unique ID (Integer): A unique identifier for each dog bite record.
2. Date of Bite (String): The date when the dog bite occurred.
3. Breed (String): The breed of the dog involved. Some records are marked as "UNKNOWN" or values like "Mixed/Other".
4. Age (Float, may contain NaN): The age of the dog (in years). Some records have missing values.
5. Gender (String): The gender of the dog, where "M" stands for Male, "F" for Female, and "U" for Unknown.
6. SpayNeuter (Boolean): Indicates whether the dog was spayed or neutered at the time of the bite.
7. ZipCode (String, may contain NaN): The zip code where the dog bite occurred.
8. Sample Size: 22664

The dataset is of a reasonable size, given the considerable quantity of data and the numerous categoricals that warrant analysis and exploration.

Analysis Method

The project uses the dog bites CSV dataset and is implemented in Rust. The code and dataset will be uploaded to GitHub for transparency and reproducibility.

1. Data Preprocessing: Rows with missing or empty ZipCode values were removed to ensure accurate geographical analysis.
2. Breed Normalization:
    a. Rows with "UNKNOWN" breeds were excluded from breed analysis.
    b. Breed names were standardized to ensure consistency. For example:
        i. All variations of "Pit Bull" (e.g., American Pit Bull, Pit Bull Mix) were grouped under "Pit Bull".
        ii. Breeds containing "Mixed" or "Mix Pit Bull" were grouped into the category "Mixed".

Analysis Techniques

1. Frequency Analysis: Used to analyze the frequency of dog bite incidents based on ZipCode, Breed, Gender, and Time (month/season).
2. Grouped Average Calculation:
    a. Dog bite incidents were grouped by breed.
    b. The average age of dogs involved in bite incidents was calculated for each breed.
3. Cross-Tabulation Analysis
    a. Performed a cross-analysis of Gender and Spay/Neuter status to examine the relationship between these factors and the frequency of dog bite incidents.
    b. For example, the frequency of incidents was analyzed for combinations like:

i. Male, Spayed/Neutered

ii. Male, Not Spayed/Neutered

iii. Female, Spayed/Neutered

Data Interpretation & Finding

1. Geographical distribution: Analysis of the distribution of dog bite incidents by postcode. The analysis identifies the top 10 ZipCodes with the highest frequency of incidents and the corresponding number of incidents.

```
--- ZipCode Frequency Analysis ---
Total distinct zip codes: 520
Top 10 ZipCodes with highest dog bite frequencies:
  10029: 369
  11208: 261
  11368: 226
  10065: 221
  10128: 211
  10314: 207
  10467: 198
  11234: 196
  10456: 188
  11377: 188
```

As illustrated in the accompanying figure, the areas with the highest frequency of dog bites are 10029 (369 incidents), 11208 (261 incidents), and 11368 (226 incidents). These areas may require targeted public safety interventions, such as responsible dog ownership education programmes and stricter pet management practices.

2. Temporal trends: Examines the frequency of dog bite incidents by month and season. The analysis highlights periods of high risk, for example, whether there are more

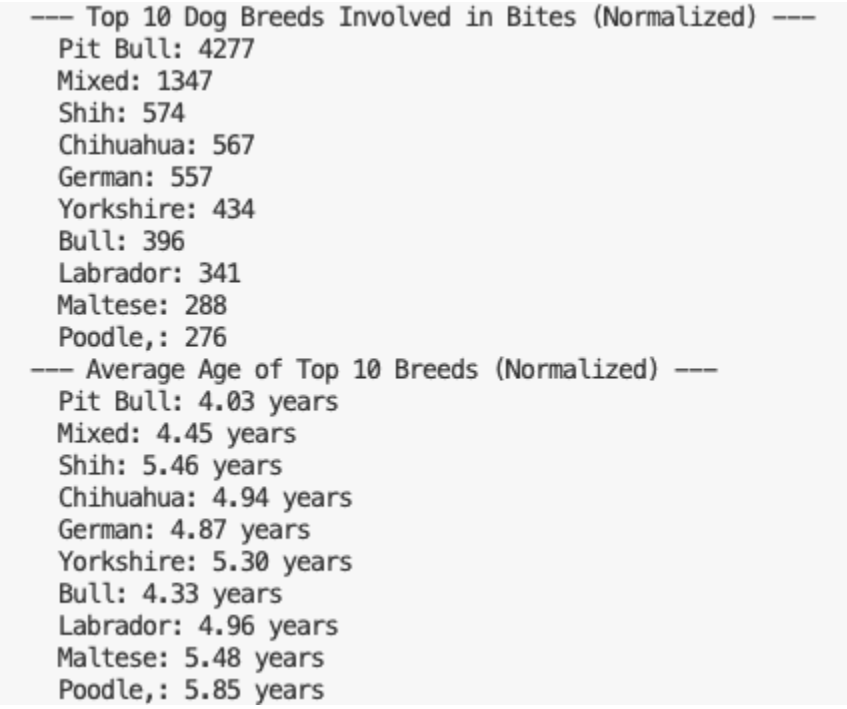
incidents in the summer months compared to other seasons.

```
--- Monthly Distribution ---
  Month 1: 1115
  Month 2: 1174
  Month 3: 1238
  Month 4: 1394
  Month 5: 1550
  Month 6: 1702
  Month 7: 1706
  Month 8: 1658
  Month 9: 1441
  Month 10: 1354
  Month 11: 1306
  Month 12: 1171
--- Seasonal Distribution ---
  Winter: 3460
  Spring: 4182
  Summer: 5066
  Autumn: 4101
```

The data presented in this figure allows us to identify the monthly trends. The highest number of incidents was recorded in July, with 1706 cases, followed closely by August (1658) and June (1702). Dog bites is highest during the summer months, which may be associated with increased levels of outdoor activity and interactions between dogs and humans.

3. Breed and age analysis: The breeds most frequently involved in bite incidents were analyzed. For high-risk breeds, the average age of the dogs was calculated. A table is

provided with the top 10 breeds and their corresponding average age.



The data presented in this image indicates that the Pit Bull is the most frequently involved breed, with 4277 incidents, followed by Mixed breeds with 1347 incidents and smaller breeds such as the Shih Tzu (574 incidents) and the Chihuahua (567 incidents). The average age of a Pit Bull is 4.03 years, which is relatively young in comparison to other breeds. For example, the average age of a Poodle is 5.85 years and a Maltese is 5.48 years. It is notable that a considerable number of dog bite incidents are attributed to Pit Bulls and mixed-breed dogs. Furthermore, age may be a contributing factor, as younger dogs may display more aggressive or unpredictable behavior.

4. Gender and effects of neutering: To compare the frequency of dog bite incidents between different genders (male, female, and unknown). In addition, the relationship between a dog's spay/neuter status and the likelihood of being involved in a bite incident was analyzed. The results include a statistical breakdown of dog bite incidents by sex and

spay/neuter combinations.

```
--- Dog Bite Incidents by Gender ---
  U: 6275
  M: 7589
  F: 2945
--- Dog Bite Incidents by Spay/Neuter Status ---
  true: 5018
  false: 11791
--- Dog Bite Incidents by Gender and Spay/Neuter Status ---
  Gender: M, Spay/Neuter: false -> 4087
  Gender: F, Spay/Neuter: true -> 1463
  Gender: U, Spay/Neuter: true -> 53
  Gender: U, Spay/Neuter: false -> 6222
  Gender: F, Spay/Neuter: false -> 1482
  Gender: M, Spay/Neuter: true -> 3502
```

As illustrated in the accompanying illustration, male dogs are involved in a greater number of bites than female dogs. In total, 7589 incidents involved male dogs, while 2945 incidents involved female dogs. The majority of incidents involved non-spayed/neutered dogs (11,791 incidents), compared to spayed/neutered dogs (5,018 incidents). The highest frequency was observed in non-neutered male dogs (4087 incidents). There is a discernible correlation between the presence of non-neutered male dogs and an elevated probability of bite incidents.

How to run the code

First, open a new terminal, then use "cd project" to make sure the position is correct. Then type "cargo test" in the terminal to test our code. After testing, there is no error, so we can type "cargo run" to get all the output.

```
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.80s
     Running unittests src/main.rs (target/debug/deps/ny_dog_bite_analysis-53b0a8644e8a73a3)

running 3 tests
test utils::tests::test_extract_month_and_season ... ok
test analysis::tests::test_normalize_breed ... ok
test utils::tests::test_parse_age ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

```
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.65s
     Running `target/debug/ny_dog_bite_analysis`
Total records loaded after filtering: 16809
--- ZipCode Frequency Analysis ---
Total distinct zip codes: 520
Top 10 ZipCodes with highest dog bite frequencies:
  10029: 369
  11208: 261
  11368: 226
  10065: 221
  10128: 211
  10314: 207
  10467: 198
  11234: 196
  11377: 188
  10456: 188
--- Monthly Distribution ---
  Month 1: 1115
  Month 2: 1174
  Month 3: 1238
  Month 4: 1394
  Month 5: 1550
  Month 6: 1702
  Month 7: 1706
  Month 8: 1658
  Month 9: 1441
  Month 10: 1354
  Month 11: 1306
  Month 12: 1171
--- Seasonal Distribution ---
  Winter: 3460
  Spring: 4182
  Summer: 5066
  Autumn: 4101
--- Top 10 Dog Breeds Involved in Bites (Normalized) ---
  Pit Bull: 4277
  Mixed: 1347
  Shih: 574
  Chihuahua: 567
  German: 557
  Yorkshire: 434
  Bull: 396
  Labrador: 341
  Maltese: 288
  Poodle,: 276
--- Average Age of Top 10 Breeds (Normalized) ---
  Pit Bull: 4.03 years
  Mixed: 4.45 years
  Shih: 5.46 years
  Chihuahua: 4.94 years
  German: 4.87 years
  Yorkshire: 5.30 years
  Bull: 4.33 years
  Labrador: 4.96 years
  Maltese: 5.48 years
  Poodle,: 5.85 years
--- Dog Bite Incidents by Gender ---
  F: 2945
  U: 6275
  M: 7589
--- Dog Bite Incidents by Spay/Neuter Status ---
  true: 5018
  false: 11791
--- Dog Bite Incidents by Gender and Spay/Neuter Status ---
  Gender: F, Spay/Neuter: true -> 1463
```

Code Explanation

My Rust code is in five files: analysis.rs, main.rs, models.rs, test.rs and utils.rs.

This is the code in analysis.rs.

project > src > analysis.rs

```rust
use crate::models::DogBiteRecord;
use crate::utils::{extract_month_and_season, parse_age};
use std::collections::HashMap;

pub fn normalize_breed(breed: &str) -> String {
    let breed_keywords = vec![
        ("Pit Bull", "Pit Bull"),
        ("American Pit Bull Mix", "Pit Bull"),
        ("American Pit Bull Terrier", "Pit Bull"),
        ("Mixed/Other", "Mixed"),
        ("MIXED BREED", "Mixed"),
        ("MIXED", "Mixed"),
    ];

    let breed_upper = breed.to_uppercase();

    for (keyword, normalized) in breed_keywords.iter() {
        if breed_upper.contains(&keyword.to_uppercase()) {
            return normalized.to_string();
        }
    }

    if let Some(first_word) = breed.split_whitespace().next() {
        return first_word.to_string();
    }

    "Unknown".to_string()
}
```

The normalise_breed function standardises dog breed names for consistency in data analysis. It takes one input, the breed, as a string, and returns the normalised breed name as a string. The function starts by defining a vector of tuples where each tuple maps a possible breed name to a single standardised name. This mapping groups similar breed names together under a common label. For example, "American Pit Bull Mix" and "American Pit Bull Terrier" are normalised to

"Pit Bull", while "Mixed/Other", "MIXED BREED" and "MIXED" are grouped under "Mixed". The input breed name is converted to uppercase to handle case insensitivity. The function then checks if the input breed name contains any predefined keywords using the contains method. If there's a match, the normalised breed name is returned as a string. This approach allows partial matches in longer breed names, ensuring flexibility in normalisation. If there is no match, the function splits the input name into words. It gets the first word of the breed name and gives it as the result. So, if the breed name is "Golden Retriever", it will give "Golden" as the normalised output. If the input breed name is empty or cannot be split into words, the function gives "Unknown". This makes sure that if the input is wrong, it will still give the right result.

```rust
pub fn analyze_breed_and_age_with_normalization(records: &[DogBiteRecord]) {
    let mut breed_count: HashMap<String, usize> = HashMap::new();
    let mut age_sum: HashMap<String, (f32, usize)> = HashMap::new();

    for r in records {
        if let Some(breed) = &r.breed {
            let normalized_breed = normalize_breed(breed);

            if normalized_breed.to_uppercase() == "UNKNOWN" {
                continue;
            }

            *breed_count.entry(normalized_breed.clone()).or_insert(0) += 1;

            if let Some(age) = parse_age(&r.age) {
                let entry = age_sum.entry(normalized_breed.clone()).or_insert((0.0, 0));
                entry.0 += age;
                entry.1 += 1;
            }
        }
    }

    println!("--- Top 10 Dog Breeds Involved in Bites (Normalized) ---");
    let mut sorted_breeds: Vec<_> = breed_count.into_iter().collect();
    sorted_breeds.sort_by_key(|&(_, count)| std::cmp::Reverse(count));
    for (breed, count) in sorted_breeds.iter().take(10) {
        println!("  {}: {}", breed, count);
    }

    println!("--- Average Age of Top 10 Breeds (Normalized) ---");
    for (breed, _) in sorted_breeds.iter().take(10) {
        if let Some((age_sum, count)) = age_sum.get(breed) {
            let avg_age = age_sum / (*count as f32);
            println!("  {}: {:.2} years", breed, avg_age);
        } else {
            println!("  {}: No age data available", breed);
        }
    }
}
```

The analyse_breed_and_age_with_normalisation function looks at dog bite records to find the most common breeds and how old they are. It processes the dataset to ensure consistent breed names. Two hash maps are created at the start: breed_count to track each breed's frequency and age_sum to store the age and count of dogs for each breed. The function goes through the dataset, normalising each record's breed. This groups similar breeds together, for example, "American Pit Bull Terrier" and "Pit Bull". Unknown breeds are skipped to maintain data

accuracy. The function increases the count for valid breeds. If there is a valid age in the record, the age_sum map is updated with the dog's age and the count for that breed is increased. The function then sorts the breeds in order of how often they occur. The 10 most common breeds involved in dog bites are displayed. The function then calculates the average age for the top 10 breeds. The average age is found by dividing the total age by the total number of dogs. The results are printed with two decimal places, showing the average age of dogs for the most common breeds. If there is no age data for a breed, a message is displayed.

```
70  pub fn analyze_zipcode_frequency(records: &[DogBiteRecord]) {
71      let mut zip_count: HashMap<String, usize> = HashMap::new();
72
73      for r in records {
74          if let Some(zc) = &r.zip_code {
75              *zip_count.entry(zc.clone()).or_insert(0) += 1;
76          }
77      }
78
79      println!("--- ZipCode Frequency Analysis ---");
80      println!("Total distinct zip codes: {}", zip_count.len());
81
82      let mut zip_vec: Vec<_> = zip_count.into_iter().collect();
83      zip_vec.sort_by_key(|&(_, c)| std::cmp::Reverse(c));
84
85      println!("Top 10 ZipCodes with highest dog bite frequencies:");
86      for (zc, count) in zip_vec.iter().take(10) {
87          println!("  {}: {}", zc, count);
88      }
89  }
```

The `analyze_zipcode_frequency` function looks at where dog bites happen most often. It starts by creating a `HashMap` called `zip_count`, which maps each ZipCode to the number of times it appears in the dataset. This makes counting and looking up data faster. The function then goes through each record in the input dataset. For each record, it checks if the zip code is valid and updates the count if it is. The `or_insert(0)` method sets a new ZipCode's default count to 0 and increments it by 1 for each occurrence. The function prints a summary of the total number of distinct zip codes after processing all the records. This shows the dataset's geographic diversity.

To find the top ZipCodes, the HashMap is converted into a list of pairs using collect(). The vector is then sorted by frequency. This is done with the `sort_by_key` method and `std::cmp::Reverse`, which puts the highest counts first.

```
91  pub fn analyze_temporal_distribution(records: &[DogBiteRecord]) {
92      let mut month_count: HashMap<u32, usize> = HashMap::new();
93      let mut season_count: HashMap<u32, usize> = HashMap::new();
94
95      for record in records {
96          if let Some((month, season)) = extract_month_and_season(&record.date_of_bite) {
97              *month_count.entry(month).or_insert(0) += 1;
98              *season_count.entry(season).or_insert(0) += 1;
99          }
100     }
101
102     println!("--- Monthly Distribution ---");
103     let mut sorted_months: Vec<_> = month_count.into_iter().collect();
104     sorted_months.sort_by_key(|&(month, _)| month);
105     for (month, count) in sorted_months {
106         println!("  Month {}: {}", month, count);
107     }
108
109     println!("--- Seasonal Distribution ---");
110     let seasons = ["Winter", "Spring", "Summer", "Autumn"];
111     let mut sorted_seasons: Vec<_> = season_count.into_iter().collect();
112     sorted_seasons.sort_by_key(|&(season, _)| season);
113     for (season, count) in sorted_seasons {
114         if season > 0 && (season as usize) <= seasons.len() {
115             println!("  {}: {}", seasons[(season - 1) as usize], count);
116         }
117     }
118 }
```

The analyse_temporal_distribution function groups dog bite incidents by month and season. It starts by creating two maps: one for each month and season. The function then looks at the dog bite records. It gets the month and season from the date_of_bite field with the extract_month_and_season helper function. If the extraction is successful, the count for that month and season is increased in the hash maps. The function then sorts and displays the monthly distribution. The month_count map is converted into a list and sorted by month. The results are printed, showing the month and incident count. The function then looks at seasonal

trends. An array maps numeric season keys to names, allowing the function to display season names. The season_count map is sorted, and the number of incidents for each season is printed.

```rust
pub fn analyze_gender_and_spayneuter(records: &[DogBiteRecord]) {
    let mut gender_count: HashMap<String, usize> = HashMap::new();
    let mut spay_neuter_count: HashMap<String, usize> = HashMap::new();
    let mut combined_stats: HashMap<(String, String), usize> = HashMap::new();

    for r in records {
        if let Some(gender) = &r.gender {
            *gender_count.entry(gender.clone()).or_insert(0) += 1;

            if let Some(spay_neuter) = &r.spay_neuter {
                *spay_neuter_count.entry(spay_neuter.clone()).or_insert(0) += 1;
                let key = (gender.clone(), spay_neuter.clone());
                *combined_stats.entry(key).or_insert(0) += 1;
            }
        }
    }

    println!("--- Dog Bite Incidents by Gender ---");
    for (gender, count) in gender_count.iter() {
        println!("  {}: {}", gender, count);
    }

    println!("--- Dog Bite Incidents by Spay/Neuter Status ---");
    for (status, count) in spay_neuter_count.iter() {
        println!("  {}: {}", status, count);
    }

    println!("--- Dog Bite Incidents by Gender and Spay/Neuter Status ---");
    for ((gender, status), count) in combined_stats.iter() {
        println!("  Gender: {}, Spay/Neuter: {} -> {}", gender, status, count);
    }
}
```

The analyse_gender_and_spayneuter function looks at dog bite incidents based on gender and whether the dogs were spayed or not. It also looks at how these two variables work together to find patterns in dog bites. The function initialises three hash maps: gender_count to store the frequency of incidents by gender, spay_neuter_count to count incidents based on spay/neuter status, and combined_stats to track incidents for each combination of gender and spay/neuter status. The function checks if the dog's gender is valid and, if so, increments the count for that

gender in gender_count. If the spay/neuter status is valid, it updates the spay_neuter_count and increments the count in the combined_stats.

This is the code in models.rs

project > src > models.rs

```rust
use serde::Deserialize;

#[derive(Debug, Deserialize)]
pub struct DogBiteRecord {
    #[serde(rename = "UniqueID")]
    pub unique_id: Option<String>,
    #[serde(rename = "DateOfBite")]
    pub date_of_bite: Option<String>,
    #[serde(rename = "Species")]
    pub species: Option<String>,
    #[serde(rename = "Breed")]
    pub breed: Option<String>,
    #[serde(rename = "Age")]
    pub age: Option<String>,
    #[serde(rename = "Gender")]
    pub gender: Option<String>,
    #[serde(rename = "SpayNeuter")]
    pub spay_neuter: Option<String>,
    #[serde(rename = "Borough")]
    pub borough: Option<String>,
    #[serde(rename = "ZipCode")]
    pub zip_code: Option<String>,
}
```

The DogBiteRecord struct in models.rs stores dog bite records from a dataset. It uses the serde crate to automatically turn raw data like a CSV file into structured Rust objects. The struct is annotated with #[derive(Debug, Deserialize)] to allow the program to convert serialised data into instances of DogBiteRecord and print the structure for debugging. Each field in the struct matches a column in the dataset and is annotated with #[serde(rename = "...")] to map the field names to the column headers in the input file. This ensures compatibility even when the dataset's

column names are different. The struct includes fields for unique ID, date of bite, species, breed, age, gender, spay/neuter, borough and zip code. All fields are of type Option<String>, which allows them to handle missing or incomplete data without causing runtime errors. For example, breed captures the dog's breed, while spay_neuter indicates whether the dog was spayed or neutered at the time of the bite.

This is the code in utils.rs.

project > src > utils.rs

```rust
use crate::models::DogBiteRecord;
use csv::ReaderBuilder;
use std::error::Error;
use std::fs::File;
use chrono::{NaiveDate, Datelike};

pub fn load_data(file_path: &str) -> Result<Vec<DogBiteRecord>, Box<dyn Error>> {
    let file = File::open(file_path)?;
    let mut rdr = ReaderBuilder::new()
        .has_headers(true)
        .from_reader(file);

    let mut records = Vec::new();
    for result in rdr.deserialize() {
        let record: DogBiteRecord = result?;
        if let Some(zc) = &record.zip_code {
            if !zc.trim().is_empty() {
                records.push(record);
            }
        }
    }

    Ok(records)
}

pub fn extract_month_and_season(date_str: &Option<String>) -> Option<(u32, u32)> {
    if let Some(date_str) = date_str {
        if let Ok(date) = NaiveDate::parse_from_str(date_str, "%B %d %Y") {
            let month = date.month();
            let season = match month {
                12 | 1 | 2 => 1, // Winter
                3 | 4 | 5 => 2, // Spring
                6 | 7 | 8 => 3, // Summer
                9 | 10 | 11 => 4, // Fall
                _ => 0,
            };
            return Some((month, season));
        }
    }
    None
}

pub fn parse_age(age_str: &Option<String>) -> Option<f32> {
    if let Some(age_str) = age_str {
        if let Some(age_num) = age_str.trim_end_matches('Y').parse::<f32>().ok() {
            return Some(age_num);
        }
    }
    None
}

```

The utils.rs file prepares the dog bite dataset for analysis. It has three main functions: load_data, extract_month_and_season, and parse_age. The load_data function reads the dataset from a CSV file, turns each row into a structured DogBiteRecord, and removes records with missing or empty ZipCode values. The function uses the CSV crate and serde for deserialization, ensuring only valid records are retained and returned as a vector. This makes the data ready for further processing. The extract_month_and_season function gets the month and season from a date. It parses the date using the format "%B %d %Y" (e.g., "January 01 2020") and determines the season based on the month. Winter (Dec-Feb), spring (Mar-May), summer (Jun-Aug), and fall (Sep-Nov). If the date is invalid or missing, the function returns None. This lets the program analyse trends in a structured way. The last function, parse_age, processes the dog's age. It removes the 'Y' from the age string (e.g. "5Y" becomes "5") and tries to parse it into a number. If the age can be parsed, it is returned; otherwise, None is returned. This stops inconsistent or invalid age data affecting the analysis.

This is the code in test.rs

```rust
#[cfg(test)]
mod tests {
    use crate::analysis::normalize_breed;
    use crate::utils::{parse_age, extract_month_and_season};

    #[test]
    fn test_normalize_breed() {
        assert_eq!(normalize_breed("American Pit Bull Mix / Pit Bull Mix"), "Pit Bull");
        assert_eq!(normalize_breed("Mixed/Other"), "Mixed");
        assert_eq!(normalize_breed("MIXED BREED"), "Mixed");
        assert_eq!(normalize_breed("German Shepherd"), "German");
        assert_eq!(normalize_breed("Unknown"), "Unknown");
    }

    #[test]
    fn test_parse_age() {
        assert_eq!(parse_age(&Some("4Y".to_string())), Some(4.0));
        assert_eq!(parse_age(&Some("5".to_string())), Some(5.0));
        assert_eq!(parse_age(&Some("".to_string())), None);
        assert_eq!(parse_age(&None), None);
    }

    #[test]
    fn test_extract_month_and_season() {
        assert_eq!(
            extract_month_and_season(&Some("January 15 2022".to_string())),
            Some((1, 1)) // 1月
        );
        assert_eq!(
            extract_month_and_season(&Some("July 04 2022".to_string())),
            Some((7, 3)) // 7月
        );
        assert_eq!(
            extract_month_and_season(&Some("October 20 2022".to_string())),
            Some((10, 4)) // 10月
        );
        assert_eq!(
            extract_month_and_season(&Some("Invalid Date".to_string())),
            None
        );
    }
}
```

This code tests the core utility and data normalisation functions used in the dog bite analysis project. It is only enabled during testing. The module has three key tests: test_normalize_breed, test_parse_age, and test_extract_month_and_season. Each test validates a specific function with different inputs. The test_normalize_breed function checks the normalize_breed function, which standardises breed names. It ensures similar breeds are correctly normalised. It also checks that

"German Shepherd" returns "German" and that "Unknown" is handled. The test_parse_age function tests the parse_age utility, which converts a dog's age from a string to a number. It shows that inputs like "4Y" and "5" are correctly parsed, while invalid inputs return None. This makes sure the data is correct. The test_extract_month_and_season function tests the extract_month_and_season function, which parses a date string to extract the month and season. Dates like "January 15 2022," "July 04 2022," and "October 20 2022" return the correct month and season, such as (1, 1) for January and (7, 3) for July. Invalid dates like "Invalid Date" return None.

This is the code in main.rs.

project > src > main.rs

```rust
mod models;
mod utils;
mod analysis;
mod test;

use analysis::{
    analyze_zipcode_frequency,
    analyze_temporal_distribution,
    analyze_breed_and_age_with_normalization,
    analyze_gender_and_spayneuter,
};
use utils::load_data;

fn main() {
    let file_path = "/opt/app-root/src/project/Dog_Bites_Data.csv";
    let records = match load_data(file_path) {
        Ok(r) => r,
        Err(e) => {
            eprintln!("Error loading data: {}", e);
            return;
        }
    };

    println!("Total records loaded after filtering: {}", records.len());

    analyze_zipcode_frequency(&records);

    analyze_temporal_distribution(&records);

    analyze_breed_and_age_with_normalization(&records);

    analyze_gender_and_spayneuter(&records);
}

```

The main.rs file is the start of the dog bite analysis project. It imports the necessary modules. `models` for data structure, `utils` for data preparation, `analysis` for analysis, and `test` for testing. The program defines the file path to the dataset in the main function. "/opt/app-root/src/project/Dog_Bites_Data.csv" The `utils` module's `load_data` function loads and deserializes the CSV file into `DogBiteRecord` objects. A 'match' statement checks if the

data is loaded successfully. If there's an error, the program prints a message and stops. Once the dataset is loaded, the program prints the number of valid records after filtering. The program then calls a series of analysis functions from the `analysis` module. The `analyze_zipcode_frequency` function shows where dog bites happen most often. The next step is to look at how incidents change over time. The `analyze_breed_and_age_with_normalization` function looks at dog breeds, normalising names to make them consistent, calculating how often different breeds occur, and working out the average age of dogs for the most common breeds. Finally, the `analyze_gender_and_spayneuter` function looks at how gender and spay/neuter status affect dog bites.