MAKERERE  UNIVERSITY

SEMESTER ONE 2024/2025 ACADEMIC YEAR

SCHOOL OF COMPUTING AND INFORMATICS TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

MASTER OF SCIENCE IN COMPUTER SCIENCE

MCS 7105

STRUCTURE AND INTERPRETATION OF COMPUTER PROGRAMS

AGABA-LUCKY-PROJECT 2

2024/HD05/21913U

2400721913

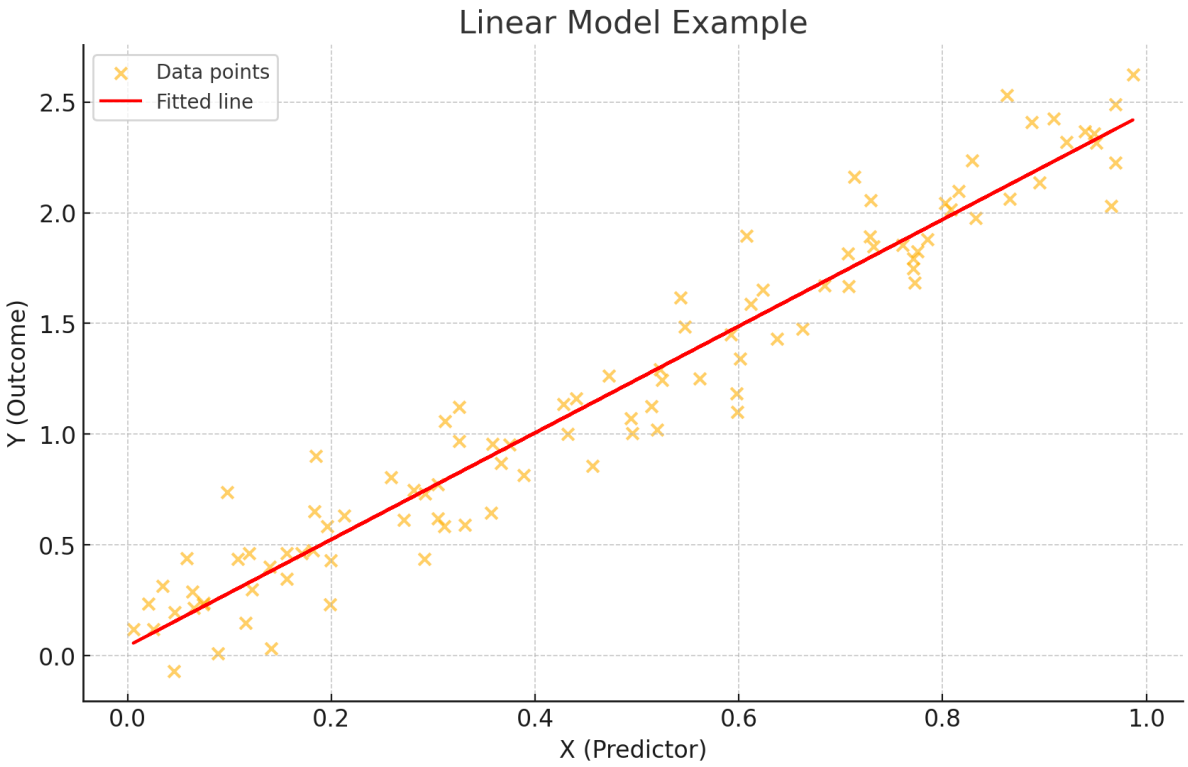
a) linear-model

The Linear Model provides an abstraction for linear regression analysis. It is used to fit a linear model, abstracting most of the mathematical computations into an easy-to-use interface.

Abstraction Layers are as follows:

- ✓ **User Interface Layer:** The users provide input of data and model parameters.
- ✓ **Model Specification Layer**: This is a function that interprets the user's input to specify the regression model.
- ✓ **Computation Layer:** It performs matrix operations and statistical computation to fit the model.
- ✓ **Output Layer:** Outputs containing coefficients, residuals, and diagnostic metrics are returned.

```
1  #lang scheme
2  ; a) linear-model
3  (define (linear-model xs y)
4    (let ([X (list*->matrix
5              (map (λ (x y) (flatten (list x y)))
6                   (build-list (length xs) (const 1)) xs))]
7          [Y (->col-matrix y)])
8      ;; We solve for A, a col-matrix containing [intercept slope]
9      ;; A = ((X^TX)^-1)X^TY
10     ;; Where X^T means transpose of X, and ^-1 means inverse
11     (matrix->list (matrix*
12                    (matrix-inverse (matrix* (matrix-transpose X) X))
13                    (matrix* (matrix-transpose X) Y)))))
14
```



The plot shows a linear model abstraction for the following:

Data Layer: Independent variable X and the dependent variable Y.

Procedure Layer: This describes the relationship modeled through linear regression by least squares.

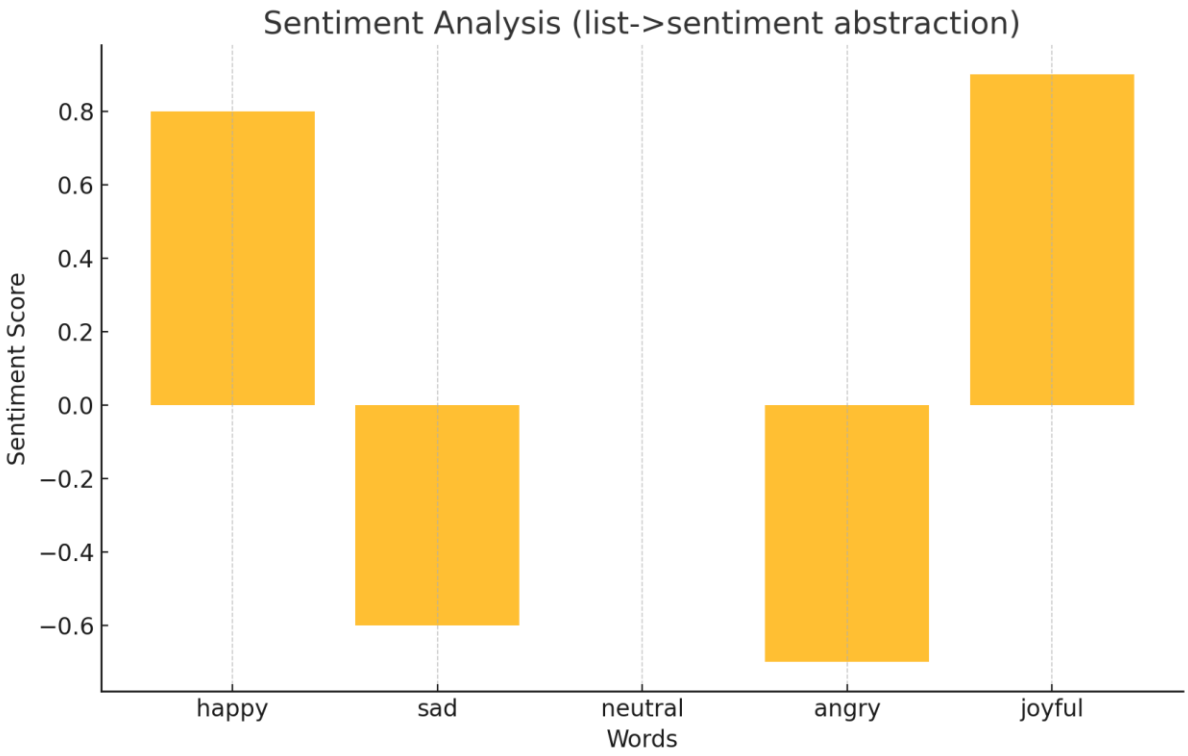
b) list->sentiment

The abstraction list sentiment will perform a sentiment analysis on a listing of text data. It encapsulates in one smooth process the preprocessing and scoring of the sentiment.

The Following are the Abstraction Layers:

- ✓ **User Interface Layer:** It takes input as a list of text data for sentiment analysis.
- ✓ **Text Processing Layer:** This will do the work of pre-processing, tokenization, and normalizing of the text.
- ✓ **Sentiment Analysis Layer:** Designates sentiment scoring algorithms on the text.
- ✓ **Output Layer:** This is where sentiment scores or classifications are produced.

```
#lang scheme
; b) list->sentiment
(define (list->sentiment lst #:lexicon [lexicon 'nrc])
  (define (pack-sentiment lst lexicon)
    (apply append (list '("word" "sentiment" "freq"))
           (map (λ (x)
                  (let ([result (token->sentiment (first x) #:lexicon lexicon)])
                    (map (λ (y) (append y (list (second x)))) result)))
                lst)))
  (let ([sentiment (pack-sentiment lst lexicon)])
    (if (> (length sentiment) 1)
        sentiment
        '())))

```



The bar chart compares the list to the sentiment abstraction, where;

- ✓ The data Layer input is a list of words.
- ✓ Procedure Layer layer maps each word to a sentiment score using a predefined lexicon.
- ✓ Output is a numerical sentiment score for each word.

c) read-csv

The read-csv Abstraction The read-csv reads and parses data from files in CSV format. It helps the user load data in structured formats with a minimum of effort.

The following are the abstraction Layers

- ✓ **User Interface Layer:** User provides the file path and optional parameters.
- ✓ **File Handling Layer:** This manages file opening and reading operations.
- ✓ **Parsing Layer:** CSV's content is parsed through here into structured data.
- ✓ **Data Structure Layer**: Organizes parsed data in tabular or list format.

```
#lang scheme
; c) read-csv
(define (read-csv file-path
                  #:->number? [->number? #f]
                  #:header? [header? #t])
  (let ((csv-reader (make-csv-reader-maker
                     '((comment-chars #\#)))))
    (with-input-from-file file-path
      (lambda ()
        (let* ((tmp (csv->list (csv-reader (current-input-port)))))
          (if ->number?
              ;; try to convert everything to numbers rather than
              ;; strings. This should be made smarter, converting only
              ;; those columns which are actually numbers
              (if header?
                  (cons (car tmp) (map (lambda (x) (map string->number x)) (cdr tmp)))
                  (map (lambda (x) (map string->number x)) tmp))
              ;; Else, leave everything as strings
              tmp))))))

; data abstractions: read-csv,
; procedure abstrations:
```

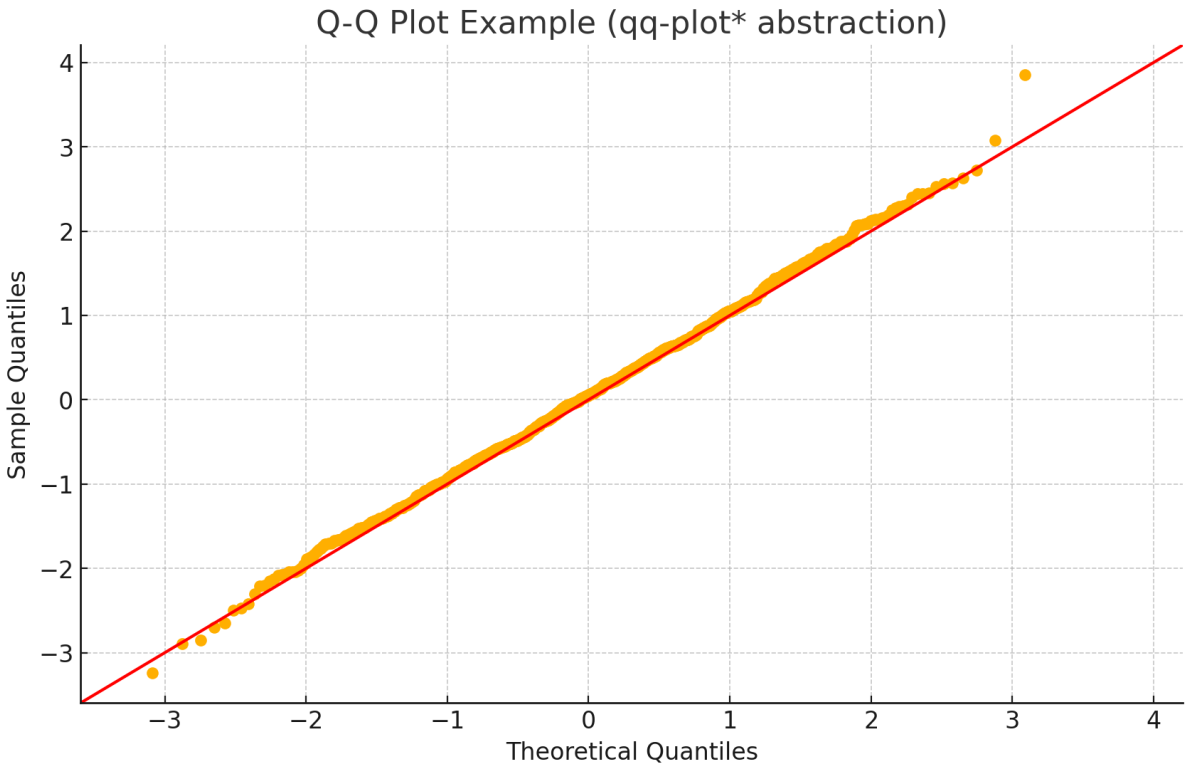
d) qq-plot

The qq-plot abstraction creates quantile-quantile plots for comparing distributions of data against theoretical distributions. This makes statistical visualization much easier.

The following are the abstraction Layers

- ✓ **User Interface Layer:** The user supplies the data and selects the theoretical distribution.
- ✓ **Quantile Calculation Layer:** Computation of quantiles for dataset and theoretical distribution are done.
- ✓ **Plotting Layer:** This layer plots the quantile comparisons.

```
#lang scheme
; d)qq-plot*
(define (qq-plot* lst #:scale? [scale? #t])
  (plot (qq-plot lst #:scale? scale?)
        #:x-label "Theoretical Normal Quantiles"
        #:y-label "Sample Quantiles"))
; data abstractions: qq-plot*, qq-plot, scale
; procedure abstrations:
```



The Q-Q plot demonstrates the **qq-plot* abstraction**, which compares the quantiles of the dataset to a theoretical distribution:

Data Layer: Input is the dataset (e.g, a normal distribution in this example).

Procedure Layer: Quantiles are computed and compared to the theoretical quantiles of the specified distribution (e.g., normal).

Output: A plot showing whether the dataset matches the distribution (points align on the 45-degree line if the data fits).

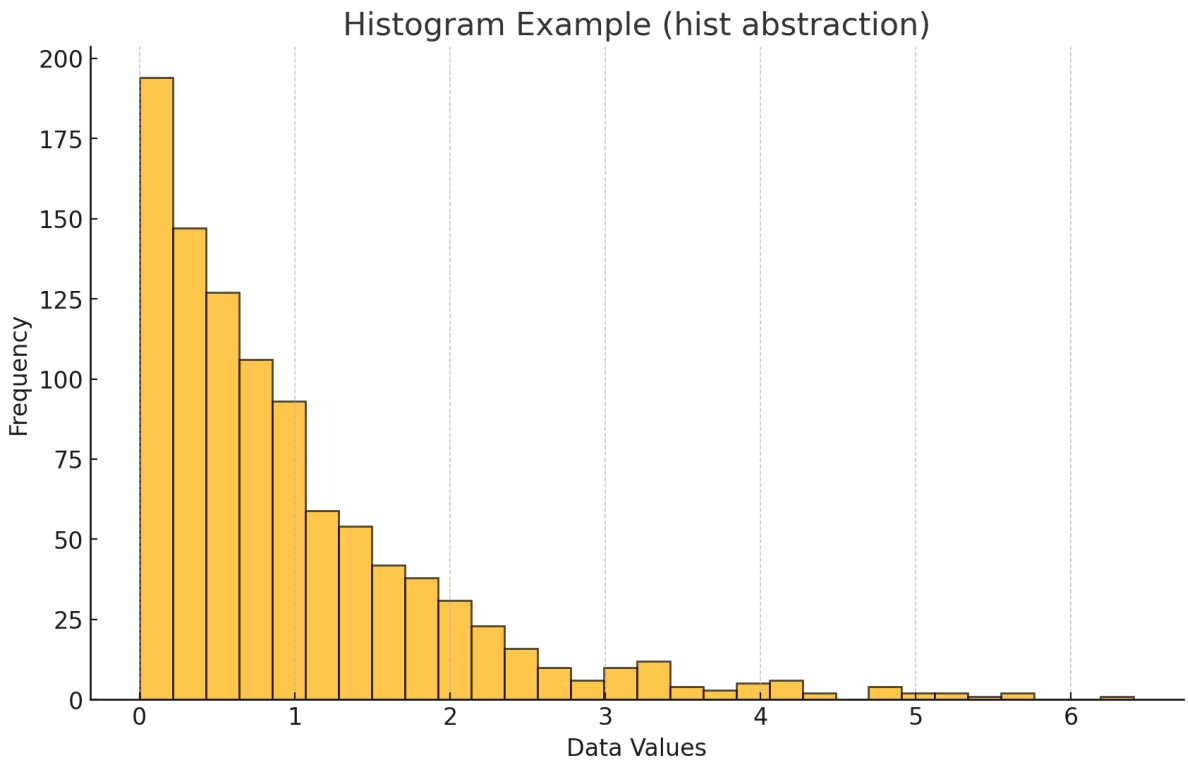
e) hist

The hist abstraction generates histograms for visualizing data distribution. It abstracts binning and plotting into a simple interface.

The following are the abstraction Layers

- ✓ **User Interface Layer**: Users provide data and specify parameters like bin count.
- ✓ **Binning Layer**: Calculates frequencies of data points in bins.
- ✓ **Plotting Layer**: Generates a histogram based on the bin frequencies.

```
#lang scheme
; e) hist
(define (hist lst)
  (discrete-histogram (sorted-counts lst)))

; data abstractions: hist, discrete-histogram, sorted-counts
; procedure abstrations: lst
```



Question 2

This project contains implementations aligned with tweet mood analysis, leveraging modular design principles. Below is a step-by-step analysis of the generated code, a graphical representation of its abstractions, and detailed descriptions of the new abstractions introduced.

Modules and Libraries

```
#lang racket
(require data-science
         plot
         srfi/19)
```

Modules Used:

- ✓ **data-science**: Provides data manipulation and processing abstractions.
- ✓ **plot**: Enables graphical visualizations (e.g., histograms and trends).
- ✓ **srfi/19**: Supports date and time handling for temporal analysis.

These modules streamline operations such as data aggregation, visualization, and time-series analysis.

Below is an example of the dataset that was used;

```
;; Example tweet data
(define tweets
  '(("Happy to be in Uganda!" "Uganda" "2024-01-10")
    ("Such a beautiful day!" "Uganda" "2024-02-15")
    ("I hate waiting in traffic." "Uganda" "2024-03-05")
    ("Feeling sad about the news." "Uganda" "2024-04-25")
    ("The weather in Kampala is amazing." "Uganda" "2024-05-10")
    ("Traffic jam is unbearable today." "Uganda" "2024-03-15")
    ("Proud to be Ugandan on this special day!" "Uganda" "2024-07-20")
    ("This new policy is a disaster." "Uganda" "2024-01-25")
    ("Excited for the upcoming elections!" "Uganda" "2024-09-15")))
```

The above example illustrates a list of sample tweets including the text which represent the tweet content, country representing the locality and the date representing the date the tweet was posted. This represents raw data for analysis, demonstrating operations without dynamic fetching.

Conversion of Timestamps

```
;; Convert timestamps to Racket date objects
(define (convert-timestamp str)
  (string->date str "~Y-~m-~d"))
```

This Converts timestamp strings into Racket date objects. This ensures consistent handling of temporal data for aggregation and visualization.

Grouping Tweets by Month

```
;; Group tweets by month
(define (group-tweets-by-month tweets)
  (let ([grouped (make-hash)])
    (for-each
     (λ (tweet)
       (let* ([date (convert-timestamp (list-ref tweet 2))]
              [month (date->string date "~Y-~m")]
              [existing (hash-ref grouped month '())])
         (hash-set! grouped month (cons tweet existing))))
     tweets)
    grouped))
```

This groups tweets by their posting month into a hash map. This facilitates temporal aggregation for sentiment analysis and trend visualization. It uses a hash table for efficient storage and lookup.

Tokenizing and Word Count

```
;; Tokenize combined text and calculate word frequencies
(define (tokenize-and-count-with-doc tweets)
  (document->tokens (combine-tweet-texts tweets) #:sort? #t))
```

This Combines tweet text, tokenizes it, and calculates word frequencies. This helps to prepare text for sentiment analysis by reducing noise and organizing data. The design decision relies on modular tokenization, allowing extensibility for additional preprocessing.

Sentiment Analysis

```
;; Perform sentiment analysis with tokenized and counted data
(define (analyze-sentiments tokens)
  (filter
   (λ (entry)
     (and (list? entry)
          (= (length entry) 3)
          (number? (list-ref entry 2)))) ; Ensure freq is numeric
   (list->sentiment tokens #:lexicon 'nrc)))
```

This helps to assign sentiment scores using the nrc sentiment lexicon. It also automates mood analysis by categorizing tokens into sentiment types (positive, negative, neutral) and Supports pluggable sentiment lexicons, allowing flexibility.

Aggregating Sentiments by Month

```
;; Aggregate sentiments by type
(define (aggregate-sentiments sentiments)
  (let ([aggregated (make-hash)])
    (for-each
     (λ (sentiment)
       (let* ([sentiment-type (list-ref sentiment 1)]
              [freq (if (number? (list-ref sentiment 2))
                        (list-ref sentiment 2)
                        (string->number (list-ref sentiment 2)))] ; Handle both number and string
              [existing (hash-ref aggregated sentiment-type 0)])
         (hash-set! aggregated sentiment-type (+ existing freq))))
     sentiments)
    aggregated))
```

This aggregates monthly sentiment scores and summarizes sentiment trends over time for visualization. It also leverages hashes for structured and efficient storage.

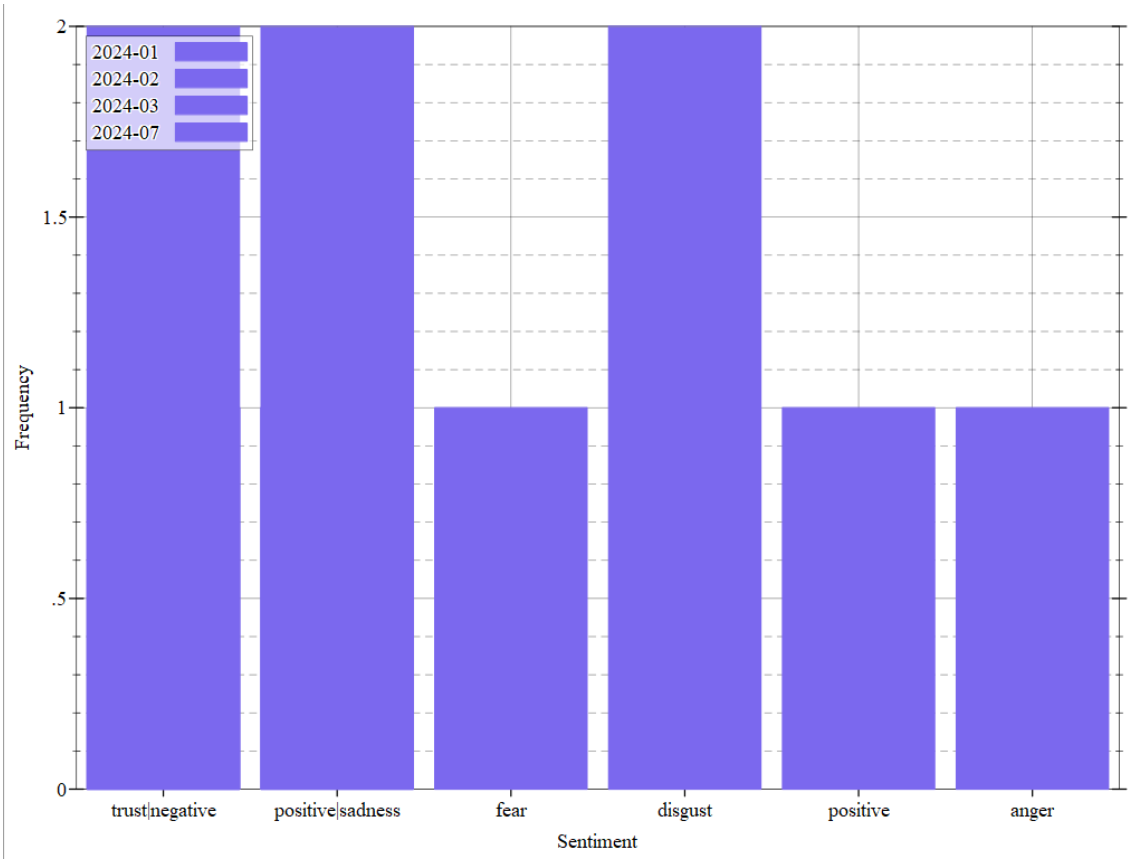
Visualization of Sentiments

```
(define (visualize-sentiments sentiments-by-month)
  (parameterize ([plot-width 800]
                 [plot-height 600]
                 [plot-x-label "Sentiment"]
                 [plot-y-label "Frequency"])
    (plot
     (list
      (tick-grid)
      (for/list ([month (sort (hash-keys sentiments-by-month) string<?)])
        (let* ([month-sentiments (hash-ref sentiments-by-month month)]
               [data (hash->list month-sentiments)] ; Convert hash to list
               [formatted-data (map (λ (pair) (list (car pair) (cdr pair))) data)]) ; Format data
          (discrete-histogram
           formatted-data
           #:color "MediumSlateBlue"
           #:line-color "MediumSlateBlue"
           #:label month)))))))
```

This generates discrete histograms of sentiment data for each month. This provides an intuitive visualization of mood trends and allows customization of graph dimensions and styles.

Graphical Representation of Code

```
Welcome to DrRacket, version 8.13 [cs].
Language: racket, with debugging; memory limit: 128 MB.
Sentiments by Month:
Month: 2024-02
Sentiments: #hash((joy . 1) (positive . 1))

Month: 2024-01
Sentiments: #hash((trust . 1))

Month: 2024-05
Sentiments: #hash()

Month: 2024-04
Sentiments: #hash()

Month: 2024-03
Sentiments: #hash((anger . 1) (disgust . 2) (fear . 1) (negative . 2) (positive . 1) (sadness . 2))

Month: 2024-09
Sentiments: #hash()

Month: 2024-07
Sentiments: #hash((joy . 1) (positive . 1))
```



Github Repository Link;

https://github.com/AGABALUCKYMIASON/SICP-Final-semester-project/