Yihang Fu
yf329
yihang.fu@yale.edu

Assignment 2
BIS 634 Computational Methods for Informatics

2024/10/16

1 Exercise 1

1.1 1a & 1b

This Bloom Filter spell correction system is initialized by inserting a large dictionary of valid words using multiple hash functions. When a misspelled word is encountered, the system generates possible corrections by simulating all single-character substitutions. Each candidate word is checked against the Bloom filter, and if it is flagged as a possible match, it is added to the list of suggested corrections. Then we evaluate the effectiveness of these suggestions based on two key criteria: the suggestion list should contain no more than three words and must include the correct spelling. Also, the other evaluation metric we need to calculate is called misidentified percentage, which is defined as the percentage of the cases where typed words are in the bloom filter while not correct words.

```
from bitarray import bitarray
from hashlib import sha3_256, sha256, blake2b
import json
import string

# Bloom Filter
class BloomFilter:
    def __init__(self, size, num_hashes):
        self.size = size
        self.num_hashes = num_hashes
        self.bit_array = bitarray(size)
        self.bit_array.setall(0)
        self.hash_functions = [self.my_hash, self.my_hash2, self.my_hash3] #three hashes

    def my_hash(self, s):
        return int(sha256(s.lower().encode()).hexdigest(), 16) % self.size

    def my_hash2(self, s):
        return int(blake2b(s.lower().encode()).hexdigest(), 16) % self.size

    def my_hash3(self, s):
        return int(sha3_256(s.lower().encode()).hexdigest(), 16) % self.size

    def add(self, word):
        for i in range(self.num_hashes):
            index = self.hash_functions[i](word)
            self.bit_array[index] = True

    def might_contain(self, word):
        for i in range(self.num_hashes):
            index = self.hash_functions[i](word)
            if not self.bit_array[index]:
                return False
        return True

# spell check using single-character substitutions
def spell_check(word, bloom_filter):
    alphabet = string.ascii_lowercase
    suggestions = set()

    # Single character substitutions, we regard these substitutions as potential suggestions
    , we will check if they are real suggestions
    for i in range(len(word)):
        for char in alphabet:
            new_word = word[:i] + char + word[i+1:]
            if bloom_filter.might_contain(new_word):
                suggestions.add(new_word)

    return list(suggestions)

# Evaluate results
def evaluate_performance(bloom_filter, dataset):
    typo_total = 0
    good_suggestions = 0
```

```
    misidentified_count = 0

    for typed_word, correct_word in dataset:
        if typed_word != correct_word:  # the typed word is a misspelled one if they are not identical.
            typo_total += 1
            suggestions = spell_check(typed_word, bloom_filter)

            # Good suggestion
            if len(suggestions) <= 3 and correct_word in suggestions:
                good_suggestions += 1

            # Misidentified
            if bloom_filter.might_contain(typed_word):
                misidentified_count += 1

    # Good suggestion
    good_suggestion_percentage = (good_suggestions / typo_total) * 100

    # Misidentified percentage: cases where typed_word is in the Bloom filter but is not correct_word
    misidentified_rate = (misidentified_count / typo_total) * 100

    return good_suggestion_percentage, misidentified_rate

size = int(1e7)
num_hashes = 3
bloom_filter = BloomFilter(size, num_hashes)

with open('words.txt') as f:
    for line in f:
        word = line.strip()
        bloom_filter.add(word)
#Use the given example for experiment
typo = "floeer"
suggestions = spell_check(typo, bloom_filter)
print(f"Suggestions for '{typo}': {suggestions}")

with open('typos.json') as f:
    dataset = json.load(f)

performance = evaluate_performance(bloom_filter, dataset)
print(f"Performance: {performance}%")
```

Listing 1: Bloom Filter for spell check

This 1 is the result when we set the size as 10^7 and number of hash functions =3.

```
Suggestions for 'floeer': ['flower', 'floter']
Performance: (91.74799999999999, 0.20400000000000001)%
```

Figure 1: 1b Experiment result

1.2 1c

Then we can plot the results using the code below.

```
import matplotlib.pyplot as plt

def experiment_and_plot(bloom_filter_sizes, num_hash_functions_list, dataset):
    results = []

    # Experiment with varying filter sizes and number of hash functions
    for size in bloom_filter_sizes:
        for num_hashes in num_hash_functions_list:
            # Initialize a new Bloom filter with the specified size and number of hash functions
            bloom_filter = BloomFilter(size, num_hashes)

            with open('words.txt') as f:
                for line in f:
                    word = line.strip()
                    bloom_filter.add(word)

```

```
            # Evaluate the performance
            good_suggestion_percentage, misidentified_rate = evaluate_performance(bloom_filter, dataset)

            # Store results
            results.append({
                'size': size,
                'hashes': num_hashes,
                'good_suggestions': good_suggestion_percentage,
                'misidentified': misidentified_rate
            })
    plot_combined_results(results)

## Single y-axis for both metrics
def plot_combined_results(results):
    sizes = sorted(list(set([result['size'] for result in results])))
    num_hashes_list = sorted(list(set([result['hashes'] for result in results])))

    plt.figure()

    # Plot both Good Suggestions and Misidentified on the same axis
    #I ask ChatGPT for help and it separates the good_suggestion and misidentified from the results list.
    for num_hashes in num_hashes_list:
        x = [result['size'] for result in results if result['hashes'] == num_hashes]
        y_good = [result['good_suggestions'] for result in results if result['hashes'] == num_hashes]
        y_misidentified = [result['misidentified'] for result in results if result['hashes'] == num_hashes]

        plt.plot(x, y_good, marker='o', linestyle='-', label=f'{num_hashes} Hash Functions - Good Suggestions')
        plt.plot(x, y_misidentified, marker='x', linestyle='--', label=f'{num_hashes} Hash Functions - Misidentified')

    plt.xscale('log')
    plt.xlabel('Bloom Filter Size (log scale)')
    plt.ylabel('Percentage')
    plt.title('Effect of Bloom Filter Size and Hash Functions on Good Suggestions and False Positives')
    plt.grid(True)
    plt.legend(loc='center left', bbox_to_anchor=(0, 0.5))# arrage the position of the legend

    plt.tight_layout()
    plt.show()

# Experiment parameters
bloom_filter_sizes = [int(1e1),int(1e2),int(1e3),int(1e4),int(1e5), int(1e6), int(1e7), int(1e8), int(1e9), int(1e10)]  # Filter sizes
num_hash_functions_list = [1, 2, 3]  # Number of hash functions

# Load the dataset (typos.json)
with open('typos.json') as f:
    dataset = json.load(f)
# Run the experiment and plot results
experiment_and_plot(bloom_filter_sizes, num_hash_functions_list, dataset)
```

Listing 2: plot results of different sizes and number of hash functions

The final result of the implementation of our bloom filter is shown in 2.

1.3 1d

From the figure, we can find that the bloom filter with 1 hash function with a size larger than 10^9 can reach the 85% good suggestion rate. The bloom filter with 2 hash functions with a size larger than 10^8 can reach the 85% good suggestion rate. The bloom filter with 3 hash functions with a size larger than 10^7 can reach the 85% good suggestion rate.

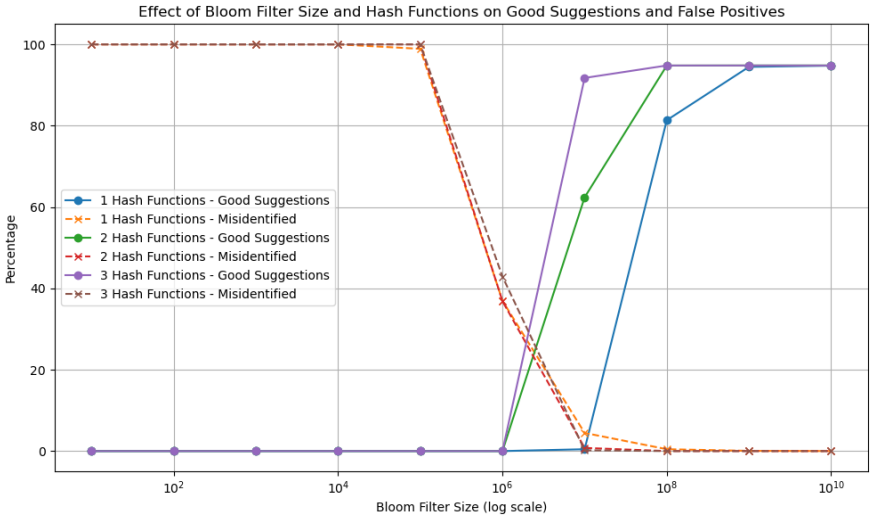

Figure 2: 1c Output

2 Exercise 2

2.1 2a

To make our merge sort algorithm capable of dictionary type data, we add a new parameter called key to search each key of the dictionary iteratively.

```python
def alg2_key(data, key):
    if len(data) <= 1:
        return data
    else:
        split = len(data) // 2
        left = iter(alg2_key(data[:split], key))
        right = iter(alg2_key(data[split:], key))
        result = []

        left_top = next(left)
        right_top = next(right)

        while True:
            if left_top[key] < right_top[key]:
                result.append(left_top)
                try:
                    left_top = next(left)
                except StopIteration:
                    # nothing remains on the left; add the right + return
                    return result + [right_top] + list(right)
            else:
                result.append(right_top)
                try:
                    right_top = next(right)
                except StopIteration:
                    # nothing remains on the right; add the left + return
                    return result + [left_top] + list(left)
```

Listing 3: Modification of merge sort

As shown in figure 3, we first build up a simple dictionary containing the data, and the new merge sort algorithm successfully sorts the data based on the patient ID.

Figure 3: 2a Output

2.2 2b

First of all, we should generate our data. We randomly choose an integer as the patient id and randomly generate a string as the patient data. It is likely to have duplicates, but the probability is low and it does not influence the performance of sorting.

```
def generateNumbers(numElements):
    randomArray = []
    for i in range(numElements):
        patient_id = random.randint(1, 100000000)
        patient_data = ''.join(random.choices(string.ascii_letters + string.digits, k=4))
        randomArray.append({'patient_id': patient_id, 'patient_data': patient_data})
    return randomArray
```

Listing 4: Generate Data

Here is our entire implementation of this task. After generating data, we designed a parallel sorting. The parallel sort function performs a merge sort on an array using parallel processing. Also, if the array size is smaller than a given 'threshold', it switches to a serial sort. For larger arrays, it splits the array into two halves, sorts them in parallel using multiple processes. The *apply async* method is used to asynchronously run the sorting on both halves, allowing them to execute concurrently. Once both halves are sorted, they are merged back together using a standard merge step (alg2 key), which is a serial merge algorithm.

```
from time import perf_counter
import random
import multiprocessing as mp
import matplotlib.pyplot as plt
import numpy as np
import string

# Logging performance data to a file
def logFile(seqtime1, seqtime2, paratime1, paratime2, speedup):
    with open("merge.log", "a") as f:
        f.write("Sequential: \n")
        f.write("Elapsed time: ")
        f.write('{:.20f}'.format(seqtime1 - seqtime2) + "\n")
        f.write("Parallel: \nElapsed time: ")
        f.write('{:.20f}'.format(paratime1 - paratime2) + "\n")
        f.write("Speedup: ")
        f.write('{:.10f}'.format(speedup) + "\n")

# Start Timer
def startTime():
    return perf_counter()

# End Timer
def endTime():
    return perf_counter()

# Custom merge sort for dictionary data based on 'patient_id'
def alg2_key(data, key):
    if len(data) <= 1:
        return data
    else:
        split = len(data) // 2
        left = iter(alg2_key(data[:split], key))
        right = iter(alg2_key(data[split:], key))
        result = []

        left_top = next(left)
        right_top = next(right)

        while True:
            if left_top[key] < right_top[key]:
                result.append(left_top)
                try:
                    left_top = next(left)
                except StopIteration:
                    return result + [right_top] + list(right)
            else:
                result.append(right_top)
                try:
                    right_top = next(right)
                except StopIteration:
                    return result + [left_top] + list(left)

```

```
# Generates randomly a number of elements in dictionary form with 'patient_id' and 'patient_data'
def generateNumbers(numElements):
    randomArray = []
    for i in range(numElements):
        patient_id = random.randint(1, 100000000)
        patient_data = ''.join(random.choices(string.ascii_letters + string.digits, k=4))
        randomArray.append({'patient_id': patient_id, 'patient_data': patient_data})
    return randomArray

def copyArray(arr):
    return arr.copy()

def sumArray(arr):
    return sum(arr)

# Parallel sorting function (using multiprocessing) for dictionary data
def parallel_sort(arr, key, threshold=1000):
    if len(arr) <= threshold:
        return alg2_key(arr, key)  # Switch to serial sort for small arrays

    split = len(arr) // 2
    with mp.Pool(processes=2) as pool:  # Using pool for parallel processing
        left_future = pool.apply_async(parallel_sort, (arr[:split], key))
        right_future = pool.apply_async(parallel_sort, (arr[split:], key))
        left = left_future.get()
        right = right_future.get()

    return alg2_key(left + right, key)

# Plot performance
def plot_performance(sizes, serial_times, parallel_times):
    plt.figure(figsize=(10, 6))
    plt.loglog(sizes, serial_times, label='Serial Merge Sort', marker='o')
    plt.loglog(sizes, parallel_times, label='Parallel Merge Sort', marker='o')
    plt.xlabel('Dataset Size (log scale)')
    plt.ylabel('Time (log scale)')
    plt.title('Performance of Serial vs Parallel Merge Sort')
    plt.legend()
    plt.grid(True)
    plt.show()

if __name__ == '__main__':
    sizes = [10**i for i in range(5,8)]  # Test sizes from 10^5 - 10^7
    sTimeArr = []
    pTimeArr = []

    for size in sizes:
        arr1 = generateNumbers(size)
        arr2 = copyArray(arr1)

        # Sequential merge sort
        start = startTime()
        alg2_key(arr1, 'patient_id')  # Perform serial sort based on 'patient_id'
        end = endTime()
        seq_time = end - start
        sTimeArr.append(seq_time)
        print(f"Sequential Sort - Size {size}: {seq_time:.6f} seconds")

        # Parallel merge sort
        pStart = startTime()
        p = mp.Process(target=parallel_sort, args=(arr2, 'patient_id'))
        p.start()
        p.join()
        pEnd = endTime()
        par_time = pEnd - pStart
        pTimeArr.append(par_time)
        print(f"Parallel Sort - Size {size}: {par_time:.6f} seconds")

        # Check for 70% time reduction
        if par_time <= 0.7 * seq_time:
            print(f"Parallel run for size {size} met the 70% criterion.")
        if par_time <= 0.33 * seq_time:
            print(f"Parallel run for size {size} met the 1/3 criterion for extra credit.")

    # Plot the results
```

```
129     plot_performance(sizes, sTimeArr, pTimeArr)
```

Listing 5: Parallel Merge Sort

The result is shown in figure 4. As we can see, serial merge sort uses less time with a small data size. When the data size increased exponentially, the parallel merge sort consume time much less than the serial merge sort. To be more specific, when the dataset size is 10^5, the time consumed by serial merge sort is 0.122866, and the time consumed by parallel merge sort is 0.685740; When the size is 10^6, the time consumed by serial merge sort is 1.877798, the time consumed by parallel merge sort is 1.212943 (**less than 70% time of the serial merge sort**); When the size is 10^7, the time consumed by serial merge sort is 30.003212, the time consumed by parallel merge sort is 9.570690 (**less than 1/3 time of the serial merge sort**)

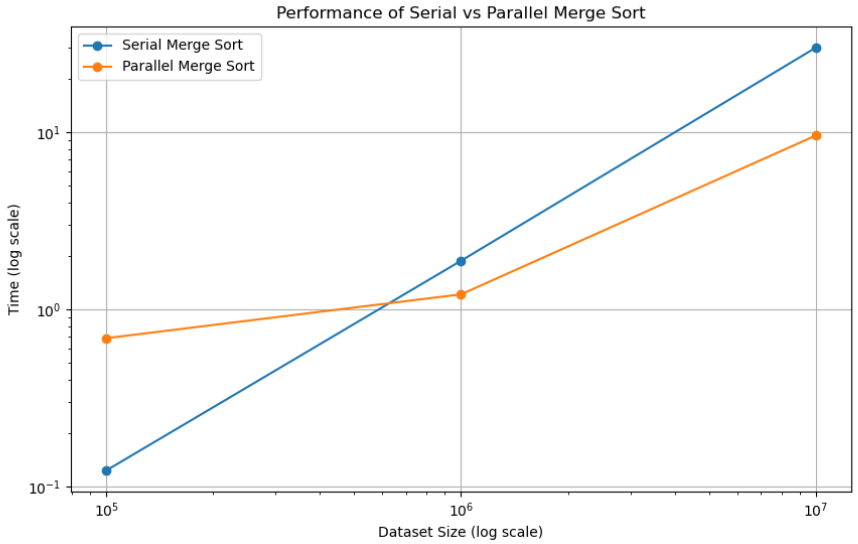

Figure 4: 2b Output

3 Exercise 3

3.1 3a

The ‘get id‘ function queries the PubMed API for research articles related to a specific disease published in 2023. It makes a GET request to retrieve up to 1000 PubMed IDs, parses the XML response, and returns a list of these IDs. If the request fails, it prints an error message and returns an empty list.

```
import requests
import xml.etree.ElementTree as ET
import time

def get_id(disease):
    # Make the request to PubMed API
    r = requests.get(
        "https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?"
        f"db=pubmed&term={disease}+AND+2023[pdat]&retmode=xml&retmax=1000"
    )
    # Respect rate limit as suggested
    time.sleep(1)

    # Check for successful response
    if r.status_code == 200:
        # Parse the XML response
        root = ET.fromstring(r.content)

        # Extract all PubMed IDs
        pubmed_ids = [id_elem.text for id_elem in root.findall(".//Id")]
        return pubmed_ids
    else:
        # Error handling
        print("Failed to fetch IDs:", r.status_code)
        return []
```

Listing 6: get id

3.2 3b, 3c &3d

The ‘get metadata‘ function retrieves metadata (article title and abstract) for a list of PubMed IDs by making a POST request to the PubMed API. It sends the list of IDs, receives the metadata in XML format, and parses it using the ‘ElementTree‘ module. For each article, the function extracts the title and abstract (including handling subsections of abstracts), then stores this information in a dictionary, associating it with the corresponding PubMed ID and the original query term. The helper function ‘get text‘ is used to extract text content from the XML nodes. If no IDs are provided, the function returns an empty dictionary.

```
def get_text(element):
    """Helper function to retrieve text content from an ElementTree node and its children.
    """
    return ''.join(element.itertext())

def get_metadata(id_list, query_term):
    metadata_dict = {}

    if id_list:
        # Make the POST request to fetch metadata
        r = requests.post(
            "https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi",
            data={
                "db": "pubmed",
                "retmode": "xml",
                "id": ",".join(id_list)
            }
        )

        # Respect API rate limiting
        time.sleep(1)

        if r.status_code == 200:
            # Parse the XML response using ElementTree and iterate through PubmedArticle
    nodes
            root = ET.fromstring(r.content)
            articles = root.findall(".//PubmedArticle")

            for article, pubmed_id in zip(articles, id_list):
```

```
                # Extract title using ElementTree
                title_elem = article.find(".//ArticleTitle")
                title = get_text(title_elem) if title_elem is not None else "" #Some
    abstract context is missing

                # Extract abstract using ElementTree, and handle multiple parts of
    structured abstracts
                abstract_elems = article.findall(".//AbstractText")
                abstract = " ".join(get_text(abstract_elem) for abstract_elem in
    abstract_elems) if abstract_elems else ""

                # Store the metadata in the dictionary
                metadata_dict[pubmed_id] = {
                    "ArticleTitle": title,
                    "AbstractText": abstract,
                    "query": query_term
                }
    return metadata_dict
```

Listing 7: get meta data

3.3 Overlap id

We finally retrieve PubMed article IDs for Alzheimer's and cancer, find IDs common to both, and print them. It then fetches metadata for the articles related to each disease and stores all metadata in a JSON file named metadata.json.

```
if __name__ == "__main__":
    Alzheimers_id = get_id("Alzheimers")
    Cancer_id = get_id("cancer")

    common_ids = list(set(Alzheimers_id) & set(Cancer_id))
    print("Common IDs:", common_ids)

    Alzheimers_metadata = get_metadata(Alzheimers_id, "Alzheimers")
    Cancer_metadata = get_metadata(Cancer_id, "cancer")

    all_metadata = {**Alzheimers_metadata, **Cancer_metadata}
    with open('metadata.json', 'w') as f:
        json.dump(all_metadata, f, indent=4)
```

Listing 8: Implementation

4 Exercise 4

We install the pretrained model from huggingface.

```
from transformers import AutoTokenizer, AutoModel

# load model and tokenizer
tokenizer = AutoTokenizer.from_pretrained('allenai/specter')
model = AutoModel.from_pretrained('allenai/specter')
```

Listing 9: Install pretrained model

Then we extract data from the file we generated before and encode the text data using the pretrained model. We can check the dimension of the embedding vectors *embeddings[0].size* and find that the dimension is 768.

```
import json
with open('metadata.json') as f:
    papers = json.load(f)
import tqdm

# we can use a persistent dictionary (via shelve) so we can stop and restart if needed
# alternatively, do the same but with embeddings starting as an empty dictionary
embeddings = {}
for pmid, paper in tqdm.tqdm(papers.items()):
    data = [paper["ArticleTitle"] + tokenizer.sep_token + "".join(paper["AbstractText"])]
    inputs = tokenizer(
        data, padding=True, truncation=True, return_tensors="pt", max_length=512
    )
    result = model(**inputs)
    # take the first token in the batch as the embedding
    embeddings[pmid] = result.last_hidden_state[:, 0, :].detach().numpy()[0]

# turn our dictionary into a list
embeddings = [embeddings[pmid] for pmid in papers.keys()]
```

Listing 10: Encode data

Then, we use PCA to reduce the high-dimension vectors to low-dimension. This helps us better interpret and visualize data. We extract three PCs and make a visualization **??**.

```
from sklearn import decomposition
import pandas as pd

pca = decomposition.PCA(n_components=3)
embeddings_pca = pd.DataFrame(
    pca.fit_transform(embeddings),
    columns=['PC0', 'PC1', 'PC2']
)
embeddings_pca["query"] = [paper["query"] for paper in papers.values()]
```

Listing 11: PCA implementation

One of the ideas of doing this visualization is to see if these vectors can well-represent the articles from different categories. We find that PCA plots reveal some degree of separation between the Alzheimer's (red) and cancer (blue) especially in PC0 and PC1 or PC0 and PC2. However, there is a noticeable overlapping, especially in the PC1 vs PC2 plot, suggesting that while the SPECTER embeddings capture meaningful differences between the two categories. One explanation is that the first two PCs capture the most variance. Variance measures how spread out the data points are from the mean. In high-variance directions, the data points are widely distributed, indicating that these directions capture the underlying differences between data points more effectively. Higher variance therefore often indicates that there are more distinguishing features along that direction.

	PC0	PC1	PC2	query
0	-4.603137	-2.170447	-1.526437	Alzheimers
1	-7.595276	0.435876	-1.645755	Alzheimers
2	-6.625034	-2.008937	-1.253660	Alzheimers
3	-5.550500	2.311249	-4.340761	Alzheimers
4	-5.913295	3.007254	-0.234550	Alzheimers
...	...	...	...	...
1992	0.697405	-3.691629	5.893336	cancer
1993	3.340729	-1.148313	6.953101	cancer
1994	9.356931	-1.196874	-3.162101	cancer
1995	3.036748	-4.447534	6.247646	cancer
1996	6.032124	-1.736871	-1.251254	cancer

1997 rows × 4 columns

Figure 5: PCA results

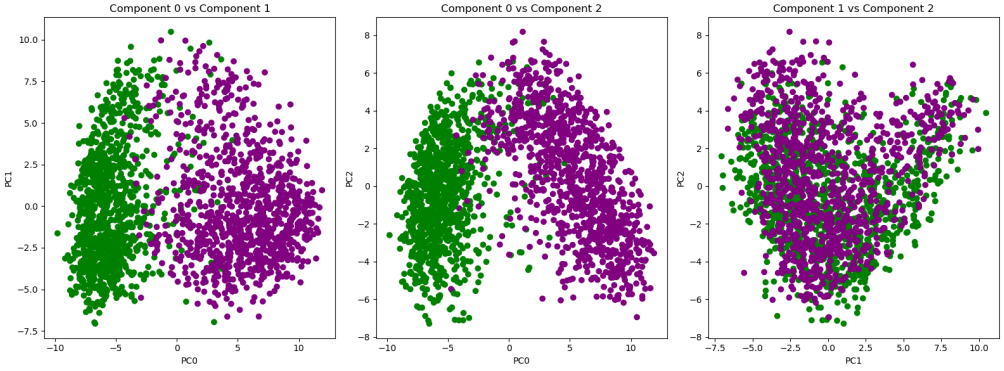

Figure 6: PCA visualization results

References