Data Privacy and Ethics

Philipp Beer

Dipl. Wirtschaftsinformatiker

January 30, 2021

Semester: winter 2020/21
Advisor: Thomas Liebig

University of Nicosia
School of Sciences and Engineering
Department of Computer Science

1 Introduction

The goal of this assignment is to implement the Flajolet-Martin algorithm to estimate cardinalities of multisets of words for a variety of entries in the online encyclopedia Wikipedia; specifically count the number of unique words in a Wikipedia page.

1.1 Motivation

Counting unique elements is one of the fundamental activities in most computer applications. In their original paper [FM85] mentioned computational constraints as a reason to search for alternative ways of achieving a reasonably accurate estimate of cardinalities of multisets. This is still true today, given the increasing amounts of data that are being generated. Aside from this, today additional use cases have arisen, that make the utilization of the below introduced of *Flajolet-Martin algorithm* interesting.

As will be shown the algorithm is capable of handling stream data while estimating the cardinalities of the transmitted dataset in a single pass. This can be used in areas related to data privacy as the basis for the counting procedure can be computed in a distributed manner and only an anonymous array of bits is aggregated centrally to compute the unique number of elements in a given data stream.

2 Resources

2.1 Programming Language

For the implementation Python is chosen as programming language. Aside from its ubiquity it also offers a number of packages that can be utilized for the realization of this task. In particular the wikpedia package [Py$_{W}$*iki*] can be used to read arbitrary wikipedia articles as it layed out in 2.2.

2.2 Data set

The resources for which unique items (n) are to be counted are the unique words occcuring in *Wikipedia* entries. Numerous options for processing this data are available that fall into these main categories:

- Scraping the website and processing the read html pages for their content
- Downloading Wikipedia as a database as described in [WikiDB]
- Utilizing a package that encapsulates the aforementioned activities and provides the data to the user

Downloading *Wikipedia* as database is possible but requires large storage space (~11GB) and for the most part would not be used. The second option is also a larger challenge as it requires signficant preprocessing steps before the actual activities in the assignment can be carried out. Therefore the third option of a package that provides the page content as a data object appears to be the most straight forward approach. Additional pre-processing steps were applied to the page content. The stream elements were:

- modified to lowercase only

- non-word characters were exchanged with whitespace character
- all digits were exchanged with whitespace character
- multi-whitespace characters replaced with a single whitespace

To get an accurate word count the above steps appear to be sensible to avoid words that can be considered unique and may only be different because of that fact that they appear at the beginning of a sentence or not. Digits in this work are not considered words and hence removed. Removing the whitespace characters ensures a clean dataset so that situations where elements are exchanged with whitespace are not counted multiple times.

In order to have access to arbitrary Wikipedia entries a readily available python package *wikipedia* is utilized and implemented as class that reads and digests *Wikipedia* pages based on the search term provided by the class user.

3 General Solution Approach

3.1 Data Ingestion

In order to get be able to count the number of unique words in a *Wikipedia* entry the *WikiText* class is instanciated and provided a search term for which an entry is queried by the wikipedia-Python package and if found returns an object with the respective page content. In case a search term is provided that can not found or retrieved a *PageError* is returned to the use.

Subsequently, the pre-processing activities are carried out and the data object is ready for analysis.

3.2 Algorithm Execution

After the data from the *Wikipedia* page has been ingested and pre-processed a second class (*FlajoletMartin*) is utilized on order to execute the counting of the number of the unique words within that article. The data stream is transferred from the wikipedia page content provided to the newly created object.

The code of these classes can be seen in refsec:class

3.3 Verification

In order to validate the veracity of the Flajolet-Martin algorithm implementation a second method for counting the unique number of words is implemented inside the *WikiText* class. It is based on the "classical" approach of creating a set after the page content has been pre-processed and reading the length of the remaining set. Hence, this approach relies soly on Python "on-board" functionality and is used for the verification of the Flajolet-Martin implementation.

4 Flajolet-Martin Algorithm

4.1 Introduction

The *Flajolet-Martin algorithm* builds on probabilities encountered in the use of hashing functions can provide reasonably accurate estimates of cardinalities in large datasets. It is built on the assumption that the records to be estimated can be hashed in a fitting pseudo-random manner.

4.2 Basic Estimation Approach

The paper by Flajolet and Martin lays out the following required elements for the estimation process:
- A single word is denoted as $x = (x_0, x_1, \dots, x_p)$ is hashed via

$$hash\ function(x) = (M + N \sum_{j=0}^{p} ord(x_j) 128^j)\ mod\ 2^L \tag{1}$$

which transforms words into integers (y) with a uniform distribution. These integers are considered in the bit form via:

$$y = \sum_{k \geq 0} bit(y, k) 2^k. \tag{2}$$

- The counting mechanism relies on p(y) which represents the position of the least significant 1-bit in the binary representation of y. The results are ranked starting from zero. The length of the bitmap vector is set as:

$$> log_2(n/nmap) + 4. \tag{3}$$

It is expected that for the value at bitmap[0] is set in n/2 times, bitmap[1] approximately n/4 times .... Therefore, the bit in the bitmap at position i is almost certainly zero if $i \gg \log_2 n$ and *1* if $i \ll \log_2 n$. The nmap value determines the number of bitmaps calculated for each word which are combined via a bitwise 'OR' statement to treat the standard deviation of R, which is leftmost zero position in the bitmap and usually has $\sigma(R) \approx 1.12$ This dispersion results in an error roughly 1 binary order of magnitude.

4.3 Results using the basic estimation approach

With the implementation of the basic algorithm described in 4.2 and the utilization of the correction method of averaging multiple bitmaps yielded fluctuating results. The results still flucuated roughly one binary order of magnitude around the correct results despite the use of nmap = 64.

4.4 Probabilistic Counting with Stochastic Averaging

In their paper [FM85] Flajolet and Martin point out an additional approach *Probabilistics Counting with Stochastic Averaging* (PCSA) to improve the performance of the algorithmic result. In this modification the hashing function $\alpha = h(x)\ mod\ m$ is utilized to determine which of the nmap bitmaps is updated. The corresponding information is stored at h(x) div m $\equiv \lfloor h(x)/m \rfloor$. In the final step the average between the different bitmaps is calculated using

$$A = \frac{R^{<1>} + R^{<2>} + \cdots + R^{<m>}}{m}. \tag{4}$$

Under the assumption that the distribution of the hashed words into the different lots is even, it can be expected that n/m elements fall into each lot so that $(1/\varphi)2^A$ can be a reasonable approximation of n/m. Flajolet and Martin define $\varphi = 0.77351 \cdots$.

5 Validation

5.1 Basic Setup

To validate the different aspects of the *Flajolet-Martin* algorithm, 3 categories of *Wikpedia* entries were defined according to the length of their unique words:

Name	Range
Small	0 - 1049
Medium	1050 - 2549
Large	n > 2550

With this setup it is validated how the performance of the basic *Flajolet-Martin* approach compared to the *PCSA* approach differs among different ranges of n.

5.2 Wikipedia Entries

The search queries chosen are a random list of *Wikipedia* entries that fall into in 5.1 mentioned categories. The list is as follows: To validate the different aspects of the *Flajolet-Martin* algorithm, 3 categories of *Wikpedia* were defined:

Search Term	True Unique Values
List of fatal dog attacks in the United States (2010s)	54
Weisswurst	265
university of nicosia	1035
data privacy	1049
Timeline of the Israeli–Palestinian conflict 2015	1406
covid	1657
List of Crusades to Europe and the Holy Land	2464
michael jordan	2529
List of University of Pennsylvania people	2928
Donald Trump	4633
2020 Nagorno-Karabakh war	4643
List of association football families	5883

5.3 Results

For each search term the two estimation algorithms were run 1.000 teams each to retrieve a sufficient sample to analyze the behavior of each estimation method.

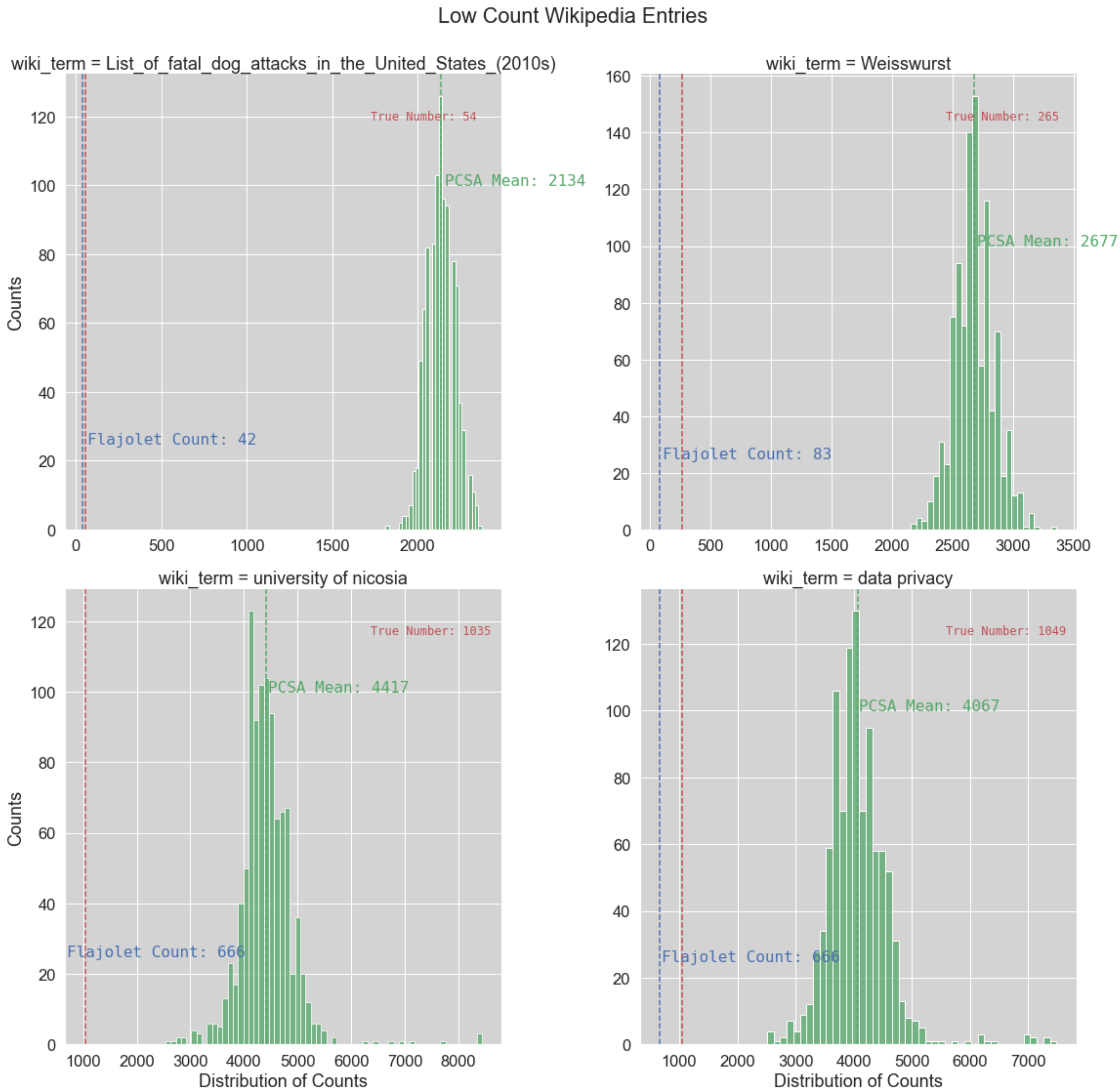

Figure 1: Low Count - Distribution of Estimations

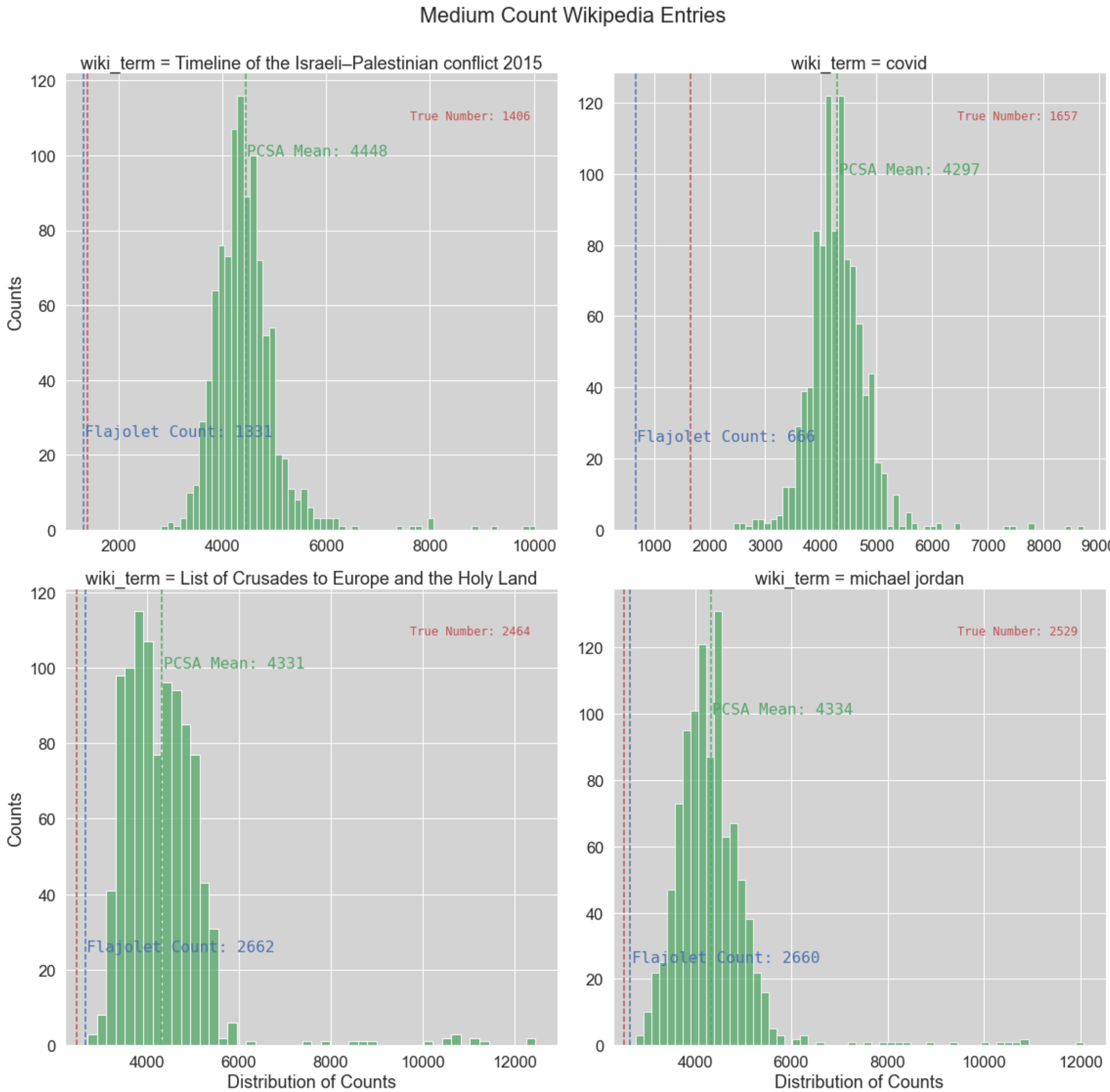

Figure 2: Medium Count - Distribuion of Estimations

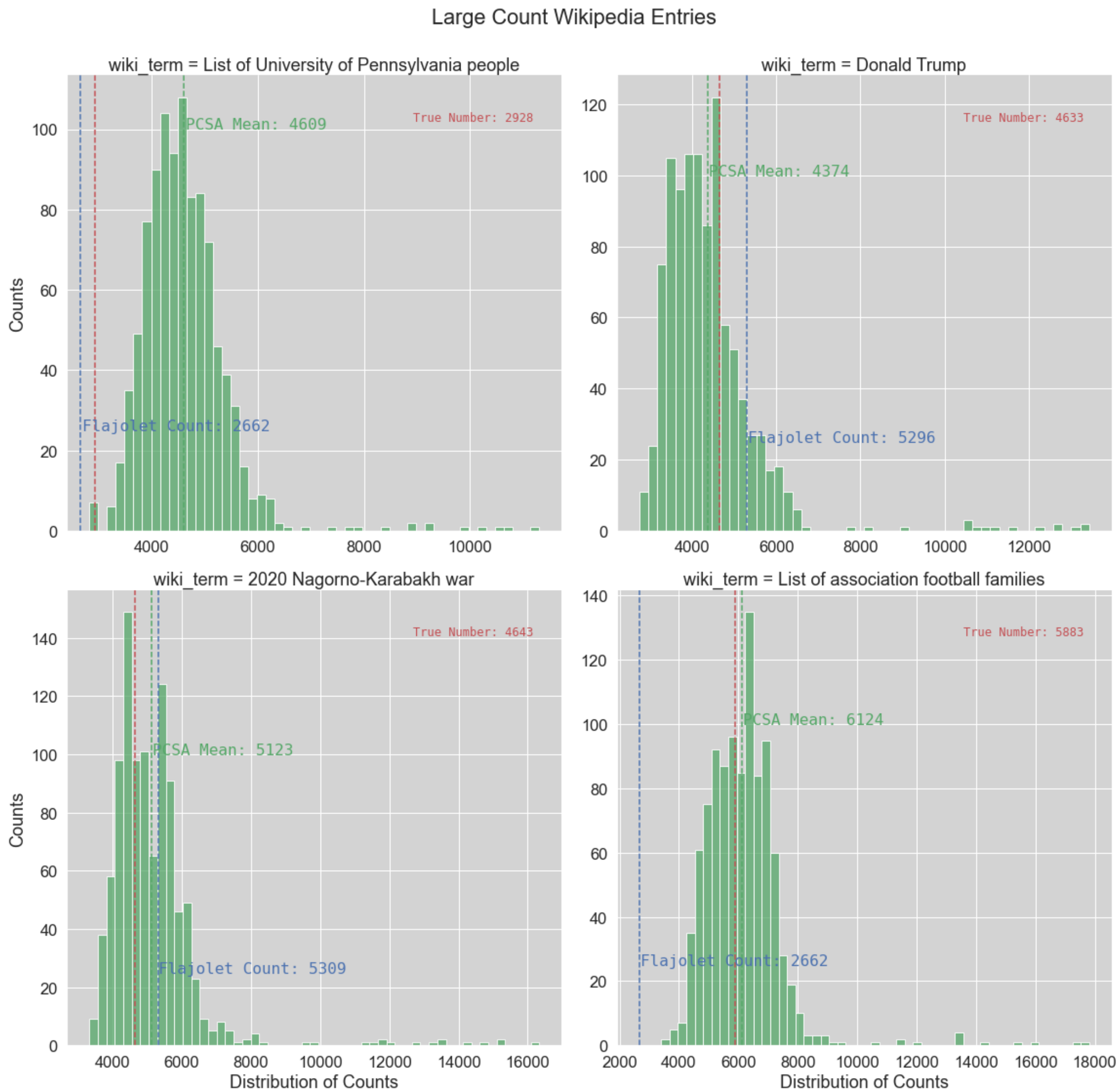

Figure 3: Medium Count - Distribuion of Estimations

5.4 Discussion

Several interesting observations can be made:

- The basic estimation method is much more stable in its behavior compared to the *PCSA* implementation. The results are very consistent and in most cases do not deviate in their results over the 1.000 executions. The only variable factor during those executions are the chosen factors for the hash function 1.
- The *PCSA* has a large distribution and tends to overestimate results in some individual cases very significantly.

- The *PCSA* method performed worst within the low count category of the Wikipedia entries. In the worst case the algorithm was off by factor of 51.
- As the number of unique entries increases the performance of the *PCSA* method improves but still lags the basic estimation method. Only in the case of the query with the largest unique entries in the sample set (List of assocation football families) the *PCSA* outperforms the basic estimation method in terms of accuracy.
- In the area of performance the *PCSA* approach is significantly more performant compared to the basic estimation method as it executes the has function per entry only 1 compared to nmap times for the basic estimation approach.
- Running the entire test scenario with 12 search terms, nmap = 64 and 1.000 executions per search term took roughly 180 minutes on a 4-core system with Hyper-Threading active and a very naive multi-threading implementation.

This implementation of the basic *Flajolet-Martin* algorithm achieves mediocre results that are generally ok but in some instances (e.g. search terms covid, university of nicosia, data privacy) show to much deviation for most real-world applications. The *PCSA* implementation performs significantly worse compared to the basic estimation. In general its performance improves as the number of unique items in the stream increases. In the range of 4500 unique items it starts to enter the range predicted by Flajolet and Martin in /citefm85. It is assumed that the result of this is so poor, due to an improper choice for the hashing function.

The area of improvement appears to be a better choice of the hash function for the *PCSA* method which does not produce reasonable results in its current state.

6 Summary

In this paper we have focused on implementing the Flajolet-Martin algorithm in two flavors, to estimate the unique in elements in a stream. In our case, unique words from *Wikipedia* entries.

The algorithm was implemented in Python and tested on 12 different search terms of varying true unique values with 1000 executions per method. The basic estimation method performed more accurately but also required significantly more compute time for a reasonable accuracy. The *PCSA* method offered much better compute performance but in its current implementation does not provide the accuracy (deviation of up to a factor of 51) described by Flajolet-Martin.

In the future, we will modify the hash function for the *PCSA* method to achieve better estimation results.

6.1 Flajolet-Martin and Wikipedia Processing

The implementation of both classes as well as the test run described above is as follows:

Listing 1: *Flajolet-Martin* Implementation

```
from typing import List, Set
import wikipedia
import re
import numpy as np
```

```python
from typing import Union
import hashlib
import sympy
import math
import random
import pandas as pd
import time
import multiprocessing


class WikiText:
    """loads pages from wikipedia and preprocesses them"""

    def __init__(self, term: str) -> None:
        self.term = term
        self.wiki_page: wikipedia.wikipedia.WikipediaPage = self.
            get_wiki_page(
             self.term)
        #self.content: List[str] = self.preprocess_page_content(self.
            wiki_page)
        self.content: np.ndarray = np.array(
             self.preprocess_page_content(self.wiki_page))

    def get_wiki_page(self, search_term: str) -> wikipedia.wikipedia.
        WikipediaPage:
        """
        returns wikipedia page from wiki wrapper API

        params:
        :search_term: (string) search term for which page from
        wikipedia is to be returned

        returns: (WikipediaPage obj) page from wikipedia
        """
        try:
            return wikipedia.page(search_term)
        except PageError:
            print(f'Page for search term: {search_term} could not be
                found')

    def get_count_distinct_words(self) -> int:
        """
        expects a list of strings to be processed and returns number of
             unique words
        """
        assert type(self.content) == np.ndarray, 'requires a numpy
            array'
        return len(set(self.content))
```

```
    def preprocess_page_content(self,
                                page: wikipedia.wikipedia.WikipediaPage
                                ) -> List[str]:
        """
        preprocess page content (remove non-word characters)

        params:
        :content: (string) content to cleaned

        returns:
        :list_of_content_words: (list of strings) corpus split into
            list
        """
        assert type(
            page) is wikipedia.wikipedia.WikipediaPage, 'requires␣
                WikipediaPage␣obj'
        assert len(page.content) != 0, 'requires␣non-empty␣corpus'
        l = page.content.lower().split("␣")
        for i in range(len(l)):
            l[i] = re.sub("\W", "␣", l[i])
            l[i] = re.sub("\d", "␣", l[i])
            l[i] = re.sub("\s+", "␣", l[i])
        return l


class FlajoletMartin:
    """
    FlajoletMartin algorithm to estimate length of passed stream
    """

    def __init__(self, data_stream: np.ndarray,
                 nmap: int = 1, L: int = 22,
                 optimization: str = 'reduce',
                 prime_hash: bool = True) -> None:
        """
        prepares Flajolet-Martin distinct count on provided data stream
        params:
        :data_stream: (numpy array) data stream for which to count
            unique elements
        :copies: (integer) number of copies of the hash function
        :L: (integer) power by which 2 will be raised to define length
            of bitmap
        """
        assert type(
            data_stream) == np.ndarray, 'data␣stream␣must␣be␣numpy␣
                array'
        assert data_stream.shape[0] > 0, 'data␣stream␣must␣be␣set'
```

```
        assert len(data_stream.shape) == 1, 'data stream must be flat'
        self.data_stream = data_stream
        assert type(nmap) == int, 'number of copies must be integer'
        assert nmap > 0, 'number of bitmaps must be larger than zero'
        self.nmap: int = nmap
        opt_options = ['reduce', 'mean_r']
        assert optimization in opt_options, 'valid optimization
            strategy must be chosen'
        self.optimization = optimization

        self.C: float = 1.3  # bias correction factor
        self.phi: float = 0.77351  # correction factor
        self.L = self.get_L(len(self.data_stream))

        # bit vector initialized to zeros
        self.bitmaps = np.zeros((self.nmap, 2**self.L), dtype=int)
        self.vs: list = self.generate_hash_factors(range_end=25,
                                                   number=self.nmap,
                                                   prime=prime_hash)  #
                                                       initialize
                                                      number of v-
                                                      factors for hash
        self.ws: list = self.generate_hash_factors(range_end=25,
                                                   number=self.nmap,
                                                   prime=prime_hash)  #
                                                       initialize
                                                      number of w-
                                                      factors for hash

        # PCSA Counting
        self.m = sympy.randprime(1, self.nmap)
        self.n = sympy.randprime(1, self.nmap)

    def get_L(self, n: int) -> int:
        """
        generates L based on Probabilistics Counting
        Algorithms for Data Base Applications by Philippe Flajolet
        params:
        :n: (integer) length of data stream for which unique count to
            be executed
        returns:
        :L: (integer) L
        """
        assert type(n) == int, 'length of data stream must be integer'
        assert n > 0, 'length of data stream must be positive'
        return math.floor(math.log2(n/self.nmap)+4)

    def generate_hash_factors(self, range_end: int = 10,
```

```python
                            prime: bool = False,
                            number: int = 1) -> np.ndarray:
    """
    generates list of hash factors v, w and p based on set number
        of copies
    """
    if prime:
        l_prime = list(sympy.sieve.primerange(1, range_end))
        l_prime.sort()
        while len(l_prime) < number:
            l_prime.append(sympy.nextprime(l_prime[-1]))
        random.shuffle(l_prime)  # send random shuffle
        return l_prime
    else:
        val = 0
        vals = list()
        for i in range(self.nmap):
            val = 2
            while (val % 2 == 0):
                val = random.randint(1, 2*self.nmap)
            vals.append(val)
        assert len(vals) == self.nmap, 'to few hash values
            generated'
        return vals

def hash_val(self, word: str, v: int, w: int) -> int:
    """
    execute hashing of passed value via function h(a) = ((va+w) mod
        p)
    params:
    :a: (word) value to be hashed
    :v: (integer) multiplication factor
    :w: (integer) addend
    :p: (integer) modulo value (ideally prime)
    """
    # turn word into list of characters
    l = list(word)
    term1: int = 0
    for i in range(len(l)):
        term1 += ord(l[i])*128**i
    return int((v*term1 + w) % 2**self.L)

def update_bitmap(self, word: str) -> None:
    """
    hash function that hashes the given value
    params:
    :e: (int) integer values to be hashed
    returns:
```

```python
        :result: (int) hash result
        """
        # calculate hash value
        for i in range(self.nmap):
            # calculate hash with current set of values
            hash_val = self.hash_val(word=word,
                                     v=self.vs[i],
                                     w=self.ws[i])
            # find rightmost set bit in hash value
            r = self.rightmost_set_bit(hash_val)
            if r == None:  # cases need to be ignored as element value
                is 0
                continue
            assert type(r) == int, 'r must be int'
            if self.bitmaps[i, r] == 0:
                self.bitmaps[i, r] = 1

    def rightmost_set_bit(self, v: int) -> int:
        """
        calculates the position of the rightmost set bit
        params:
        :v: (integer) value for which to obtain rightmost set bit
        returns:
        :rightmost_position_set:
        """
        # using bit operations to identify position
        # of least significant set bit
        if v == 0:
            return None
        return int(math.log2(v & (~v + 1)))


    def leftmost_zero(self, bitmap: np.ndarray) -> int:
        """
        identifies position of leftmost bit set to zero
        params:
        :b: (Numpy Array) to be searched for rightmost zero
        returns
        :leftmost_zero: (integer) returns rightmost zero index position
                        counting from the right starting with index
                        0
        """
        res = np.where(bitmap == 0)[0]  # finds all zeros in bitmap
        return res[0]

    def reduce_bitmaps(self, bitmap: np.ndarray) -> np.ndarray:
        """
        reduces the bitmaps filled by random hash functions
```

```
        to single bitmap via element wise or
        returns:
        :reduced_bitmap: bitmap joined by component-wise OR on all
            bitmaps
        """
        # set inital bitmap for elementwise OR
        if bitmap.shape[0] == 1:
            return bitmap[0]
        else:
            reduced_bitmap = bitmap[0, :]
            for i in range(1, bitmap.shape[0]):
                assert bitmap[i, :].shape == reduced_bitmap.shape
                comp_bitmap = bitmap[i, :]
                reduced_bitmap = np.bitwise_or(reduced_bitmap,
                    comp_bitmap)
            return reduced_bitmap

    def fm(self) -> int:
        """
        applies hash function to each value in stream
        params:
        :stream: (Numpy Array) to which hash function needs to be
            applied elementwise
        returns:
        :hashed_values: (Numpy Array) hashed values stream
        """
        # allowing for hashing of entire stream
        vbitmap_update = np.vectorize(self.update_bitmap)
        # contains hashed values for each element in stream
        vbitmap_update(self.data_stream)

        if self.optimization == 'reduce':
            # reduce bitmap
            red_bitmap = self.reduce_bitmaps(self.bitmaps)
            R = self.leftmost_zero(red_bitmap)
            return self.C*2**R
        elif self.optimization == 'mean_r':
            R = np.zeros((self.nmap,))
            for i in range(self.nmap):
                R[i] = self.leftmost_zero(self.bitmaps[i, :])
            mean_R = np.mean(R)
            return self.C*2**mean_R

    def pcsa_bitmap(self, word: str) -> None:
        """
        pcsa bitmap
        params:
        :e: (int) integer values to be hashed
```

```python
        returns:
        :result: (int) hash result
        """
        hashedx = self.hash_val(word=word,
                                v=self.m,
                                w=self.n)
        alpha = hashedx % self.nmap
        beta = math.floor(hashedx/self.nmap)
        assert isinstance(beta, int), "index is integer"
        idx = self.rightmost_set_bit(beta)
        self.bitmaps[alpha, idx] = 1

    def fm_pcsa(self) -> int:
        # allowing for hashing of entire stream
        vbitmap_update = np.vectorize(self.pcsa_bitmap)
        # contains hashed values for each element in stream
        vbitmap_update(self.data_stream)
        S = 0
        for i in range(self.nmap):
            R = 0
            while (self.bitmaps[i, R] == 1) and (R < self.L):
                R += 1
            S += R
        return math.floor(self.nmap/self.phi*2**(S/self.nmap))


def calc_sample(term: int, rounds: int, ret_wiki_term: List[str],
                ret_true_cnt: List[int],
                ret_fm_cnt: list,
                ret_pcsa_cnt: list) -> None:
    assert isinstance(term, str), "term is a string"
    print(f'started processing of {term}')
    stream = WikiText(term)
    distinct_count = stream.get_count_distinct_words()
    # lists to capture results
    wiki_term = list()
    true_count = list()
    fm_count = list()
    fm_pcsa_count = list()
    for i in range(rounds):
        wiki_term.append(term)
        true_count.append(distinct_count)
        fma = FlajoletMartin(stream.content, nmap=64)
        fm_count.append(fma.fm())
        fm_pcsa_count.append(fma.fm_pcsa())

    ret_wiki_term.extend(wiki_term)
    ret_true_cnt.extend(true_count)
```

```
    ret_fm_cnt.extend(fm_count)
    ret_pcsa_cnt.extend(fm_pcsa_count)
    print(f'term {term} attached to mgr lists')


if __name__ == "__main__":
    search_terms = ['michael jordan',
                    'covid',
                    '2020 Nagorno-Karabakh war',
                    'List of association football families',
                    'Weisswurst',
                    'List of Crusades to Europe and the Holy Land',
                    'List_of_fatal_dog_attacks_in_the_United_States_
                       (2010s)',
                    'Donald Trump',
                    'Timeline of the Israeli Palestinian conflict 
                       2015',
                    'List of University of Pennsylvania people',
                    'university of nicosia',
                    'data privacy'
                    ]

    procs = multiprocessing.cpu_count() - 1  # number of processes
    rounds = 2
    jobs = []
    df_all = pd.DataFrame()

    manager = multiprocessing.Manager()
    ret_wiki_term = manager.list()
    ret_true_cnt = manager.list()
    ret_fm_cnt = manager.list()
    ret_pcsa_cnt = manager.list()


    #########
    for elm in search_terms:
        p = multiprocessing.Process(target=calc_sample,
                                    args=(elm,
                                          rounds,
                                          ret_wiki_term,
                                          ret_true_cnt,
                                          ret_fm_cnt,
                                          ret_pcsa_cnt))
        jobs.append(p)
        p.start()

    # checking they are done
    for j in jobs:
```

```
        j.join()
    print('done')
    data = {'wiki_term': ret_wiki_term[:],
            'true_count': ret_true_cnt[:],
            'fm_count': ret_fm_cnt[:],
            'pcsa_count': ret_pcsa_cnt[:]}

    df = pd.DataFrame.from_dict(data)
    df.to_csv('./fm_analysis_'+str(rounds)+'.csv', index=False)
```

6.2 Visualization Code

The visualization were done with slight variation of the following code:

Listing 2: Visualization Code

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv('./fm_analysis_1000.csv').sort_values(by='true_count')

sns.set(font_scale=1.5,
        rc={'axes.facecolor': 'lightgrey'})

g = sns.FacetGrid(df, sharex=False, sharey=False, col='wiki_term',
                  height=8, aspect=1, col_wrap=2, margin_titles=True)
g.map(sns.histplot, 'pcsa_count', color='g').set(yscale='log');

def vertical_mean_line(x, **kwargs):

    if x.name == 'true_count':
        name = "True Value"
        c = 'r'
        plt.axvline(x.mean(), linestyle="--", color=c)
    elif x.name == 'fm_count':
        name = 'Flajolet Count'
        c = 'b'
        plt.axvline(x.mean(), linestyle="--", color=c)
        txkw = dict(size=16, color=c, fontfamily='monospace')
        tx = name+": {:.0f}".format(x.mean())
        plt.text(x.mean()+25, 25, tx, **txkw)
    else:
        c = 'g'
        name= 'PCSA Mean'
        plt.axvline(x.mean(), linestyle="--", color=c)
        txkw = dict(size=16, color=c, fontfamily='monospace')
        tx = name+": {:.0f}".format(x.mean())
        plt.text(x.mean()+25, 100, tx, **txkw)
```

```python
def text_box(x, **kwargs):
    txkw = dict(size=12, color = 'r', fontfamily='monospace')
    mean = x.mean()
    tx = "True Number: {:.0f}".format(mean)
    ax = plt.gca()
    ax.text(0.7, 0.9, tx, **txkw,
            transform=ax.transAxes);


g.map(vertical_mean_line, 'true_count')
g.map(vertical_mean_line, 'fm_count')
g.map(vertical_mean_line, 'pcsa_count')
g.map(text_box, 'true_count')
g.fig.subplots_adjust(top=0.92)
g.fig.suptitle('Length Estimation Wikipedia Entries')
g.set_axis_labels( "Distribution of Counts" , "Counts")
g.savefig('distribution.png')
```

References

[FM85] Flajolet M. and Martin G. N., "Probabalistic Counting for Data Base Applications", *Journal of Compyter and System Sciences*, 1985

[PPMM] Kamp, Michael, et. al., "Privacy-Preserving Mobility Monitoring using Sketches of Stationary Sensor Readings", 2013

[Py$_W$*iki*] Goldsmith, J. "Wikipedia", `https://github.com/goldsmith/Wikipedia`

[WikiDB] Wikimedia Downloads *https://dumps.wikimedia.org*, Wikimedia! The Wikimedia Foundation, Inc., 2021