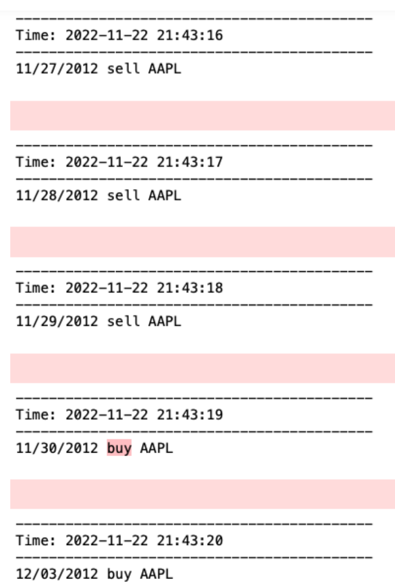
Mattia Danese
CS 119 - Big Data
Professor Singh
Quiz 5: Data Streams

<u>Question 1</u>

- Approach
  For this question, I wrote my code in a Jupyter Notebook. I split my code into three parts: declaring `SparkContext` and `StreamingContext`, manipulating the `DStreams` and doing the necessary computations, and starting the PySpark streaming. I further break down the second step of my code, `setup_stream()`, into two sections: one for calculating the 10 day average and one for calculating the 40 day average. The code for calculating both averages is virtually the same; the only difference is that the DStream windowing for the 10 day average spans 10 AAPL stock prices while that for the 40 day average spans 40 AAPL stock prices. In both cases, I follow these high-level steps: I split the input DStream into one for the date and one for the price and one for the count, I derive the `sum` DStream by adding up consecutive prices, I then key these DStreams, I join `sum_keyed` DStream with the `count_keyed` DStream to get an interim `avg` DStream, I join the `avg` DStream with the `dates_keyed` DStream, and finally unkey the resulting DStream. My last step was to join the final DStreams for 10 day average and 40 day average and map the appropriate buy/sell message based on the values of corresponding 10 day average and 40 day average.

- Results
  To the right is a screenshot of the output I got from running my code. After manually sifting through the output, the 10 day and 40 day moving averages crossover at the dates stated below:
  - 11/30/2012 buy AAPL
  - 12/11/2012 sell AAPL
  - 03/26/2013 buy AAPL
  - 04/03/2013 sell AAPL
  - 05/07/2013 buy AAPL
  - 06/17/2013 sell AAPL
  - 07/24/2013 buy AAPL
  - 09/18/2013 sell AAPL
  - 10/11/2013 buy AAPL
  - 01/10/2014 sell AAPL
  - 02/25/2014 buy AAPL
  - 02/26/2014 sell AAPL
  - 03/12/2014 buy AAPL



- 04/11/2014 sell AAPL
- 04/25/2014 buy AAPL
- 10/07/2014 sell AAPl
- 10/24/2014 buy AAPL

- Jupyter Notebook Code

```
In [ ]: from pyspark import SparkContext
        from pyspark.streaming import StreamingContext

        # sc.stop()
        sc = SparkContext("local[2]", "q1")
        ssc = StreamingContext(sc, 1)
```

```
In [ ]: from pyspark import StorageLevel

        def setup_stream():
            lines = ssc.socketTextStream("localhost", 9999, StorageLevel.MEMORY_AND_DISK)
            date_price_pairs = lines.map(lambda x: x.split(" "))

            # 10 day MA
            count_10day  = date_price_pairs.countByWindow(10, 1)
            dates_10day  = date_price_pairs.map(lambda x: x[0])
            prices_10day = date_price_pairs.map(lambda x: float(x[1][1:]))
            sum_10day    = prices_10day.reduceByWindow(lambda x,y: x + y, lambda x,y: x - y, 10, 1)

            count_10day_keyed = count_10day.map(lambda x: (1, x))
            dates_10day_keyed = dates_10day.map(lambda x: (1, x))
            sum_10day_keyed   = sum_10day.map(lambda x: (1, x))

            join_sum_count_10day = sum_10day_keyed.join(count_10day_keyed)
            avg_10day            = join_sum_count_10day.map(lambda x: (1, x[1][0] / x[1][1]))

            join_date_avg_10day = dates_10day_keyed.join(avg_10day)
            final_10day         = join_date_avg_10day.map(lambda x: (x[1]))

            # 40 day MA
            count_40day  = date_price_pairs.countByWindow(40, 1)
            dates_40day  = date_price_pairs.map(lambda x: x[0])
            prices_40day = date_price_pairs.map(lambda x: float(x[1][1:]))
            sum_40day    = prices_40day.reduceByWindow(lambda x,y: x + y, lambda x,y: x - y, 40, 1)

            count_40day_keyed = count_40day.map(lambda x: (1, x))
            dates_40day_keyed = dates_40day.map(lambda x: (1, x))
            sum_40day_keyed   = sum_40day.map(lambda x: (1, x))

            join_sum_count_40day = sum_40day_keyed.join(count_40day_keyed)
            avg_40day            = join_sum_count_40day.map(lambda x: (1, x[1][0] / x[1][1]))

            join_date_avg_40day = dates_40day_keyed.join(avg_40day)
            final_40day         = join_date_avg_40day.map(lambda x: (x[1]))

            join_10day_40day = final_10day.join(final_40day)
            signals = join_10day_40day.map(lambda x: (x[0] + " buy AAPL" if x[1][0] > x[1][1] else x[0] + " sell AAPL"))
            signals.pprint()

        def launch_stream(w):
            ssc.checkpoint("checkpoint")
            ssc.start()
            ssc.awaitTermination(w)
```

```
In [ ]: setup_stream()
        launch_stream(40)
```

- I would also like to mention that I worked on this question and debugged with other students and TAs during office hours.

Question 2

- Approach
  - For this question, I followed the steps laid out by Professor J in [this](#) slide. I used the implementation of the HyperLogLog algorithm written by Vasily Evseenko which can be found [here](#). All credit for this implementation of the algorithm goes to Vasily Evseenko. Seeing as the implementation of Evseenko is written in a Python2 notebook, I converted it to a Python3 script. This just consisted of syntax fixes (`print()` and conversions to the `long` type) and decomposing the notebook blocks into one continuous script. I then tested this code with small text files of known cardinality and the algorithm performed fairly well. Finally, I fed the algorithm the full `news-feeder.py` stream.

- Testing
  - Text file with a cardinality of 435

```
~/Desktop/CS 119/quizzes/quiz5 🧠 cat words435.txt | python3 unique_word_count.py
estimate cardinality as 440.91377716979883
~/Desktop/CS 119/quizzes/quiz5 🧠
```

  - Text file with a cardinality of 2007

```
~/Desktop/CS 119/quizzes/quiz5 🧠 cat words2007.txt | python3 unique_word_count.py
estimate cardinality as 2208.5891305984464
~/Desktop/CS 119/quizzes/quiz5 🧠
```

- Final Results

```
~/Desktop/CS 119/quizzes/quiz5 🧠python3 news-feeder.py | python3 unique_word_count.py
estimate cardinality as 2430.2024363962787
~/Desktop/CS 119/quizzes/quiz5 🧠
```

- unique_word_count.py

```
import math
import bisect
from hashlib import sha1
import fileinput
import string

'''
DISCLAIMER:
Complete credit is given to Vasily Evseenko for the implementation
below of the HyperLogLog algorithm.
The only changes I made to Evseenko's implementation is making it
compatible with Python3.
'''
```

```
def _get_alpha(b):
   if not (4 <= b <= 16):
       raise ValueError("b=%d should be in range [4 : 16]" % b)

   if b == 4:
       return 0.673
   if b == 5:
       return 0.697
   if b == 6:
       return 0.709
   return 0.7213 / (1.0 + 1.079 / (1 << b))

def estimate_cardinality(alpha, bits, bins):
   # harmonic mean
   E = alpha * float(len(bins) ** 2) / sum(math.pow(2.0, -x) for x in
bins)

   if E <= 2.5 * bits:             # Small range correction
       V = bins.count(0)           #count number or registers equal to
0
       return bits * math.log(bins/ float(V)) if V > 0 else E
   elif E <= float(1 << 160) / 30.0:
       return E
   else:
       return -(1 << 160) * math.log(1.0 - E / (1 << 160))

# 'rho' function to calculate the bit pattern to watch (string of 0s)
# here, 'rho' is the number of 0s to the left of the first 'accuracy'
bits.
def rho(w):
   r = len(bit_bins) - bisect.bisect_right(bit_bins, w)
   return r

# to add a number into the counter:
def add(num):
   # take the hash of 'num'
   num = str(num).encode()
   hash = int(sha1(num).hexdigest(), 16)

   # here, 'bin' is determined by the first 'bits' bits of hash
   bin = hash & ((1 << bits) - 1)

   # now count the number of 0s in the remaining bits
```

```
    remaining_bits = hash >> bits
    count = rho(remaining_bits)

    # take max of currently stored estimation & this one
    estimators[bin] = max(estimators[bin], count)

# choose the precision by choosing how many estimators to track.
bits = 8
alpha = _get_alpha(bits)
num_bins = 1 << bits
bit_bins = [ 1 << i for i in range(160 - bits + 1) ]

# print 'initializing', num_bins, 'estimators'
estimators = [0]*num_bins

for line in fileinput.input():
    for word in line.split():
        word = word.translate(str.maketrans('', '', string.punctuation))
        add(word)

print('estimate cardinality as', estimate_cardinality(alpha, bits,
estimators))
```

Question 3

- Approach
  - For the first part of this question (creating the bloom filter), I consulted the bloom filter implementation and explanation found here. I estimated n (the sum of words to be added and queried to the filter) by taking the cardinality of the headlines "dataset" (found in question 2), adding the number of words with a valence of -4 and -5 (63, based on this GitHub repo), and multiplying this sum by 4 to account for the majority of headlines containing very common words (like "the" or "is"). As a result, I set n to 10,000. After consulting with TA Zhaoqi, I set my false probability rate, p, to $\frac{1}{n} = 0.0001$. I then gave n and p to my bloom filter implementation and it created a bloom filter with size, m, of 191,701 bits and number of hash functions, k, equal to 13. I crossed-referenced the values of m and k with the bloom filter calculator found here, and k matched exactly and m was off by just one bit; thus, I deemed these values reasonable. I added the "bad" words to the bloom filter by copying the list of words in the repo and their respective valence into a text file, looping over each word-valence pair in the text file, and adding only the words with a valence of -4 or -5. Finally, I wrote my bloom filter (the actual bit array) to a text file called `bloom.txt`.
  - For the second part of this question, I first uploaded `bloom.txt` to my cluster. Then, in one terminal, I ran the `pyspark` command and copied the code I wrote in `filter_news_feeder.py` (the script I wrote to filter the incoming headlines) into the pyspark interpreter. In another terminal, I ran the command `python3 news-feeder.py | nc -lk 9999` at the same time in order to send the streams of headlines.

- Results
  - Seeing as the words that I used with a valence of -4 or -5 are truly bad words, it is no surprise that no headlines were flagged for containing such words by my bloom filter (I ran my pyspark code until `news-feeder.py` stopped emitting any headlines). Additionally, my bloom filter did not encounter any false positives so I think this reaffirms that my choices for n and p were good.
  - Note: I understand only flagged headlines with a bad word should be printed out, but, for the sake of having some output, I am printing every headline which is either labeled as a "good" headline or a "bad" headline with the bad word highlighted

- Here is the recorded video session of my bloom filter in action.

- I would also like to mention that I collaborated on this question with classmate Eliza Vardanyan.

- bloom_filter .py (creates `bloom.txt`)

```
import math
import mmh3
from bitarray import bitarray

class BloomFilter(object):
   def __init__(self, items_count, fp_prob):
        self.fp_prob = fp_prob
        self.size = self.get_size(items_count, fp_prob)
        self.hash_count = self.get_hash_count(self.size, items_count)
         self.bit_array = bitarray(self.size)
        self.bit_array.setall(0)
   def add(self, item):
        for i in range(self.hash_count):
            digest = mmh3.hash(item, i) % self.size
            self.bit_array[digest] = True

   def write_to_file(self):
        file = open("./bloom.txt", 'wb')
        self.bit_array.tofile(file)
   @classmethod
   def get_size(self, n, p):
        m = -(n * math.log(p))/(math.log(2)**2)
        return int(m)
   @classmethod
   def get_hash_count(self, m, n):
        k = (m/n) * math.log(2)
        return int(k)


# number of AFINN words to be added to bloom filter
# based on words in
https://github.com/fnielsen/afinn/blob/master/afinn/data/AFINN-en-165.t
xt
n = 10000
false_positive_probability = float(1) / float(n)

bf = BloomFilter(n, false_positive_probability)
print("Size of but array:{}".format(bf.size))
print("Number of hash functions:{}".format(bf.hash_count))

# adding words to bloom filter from afinn file
```

```
with open("afinn.txt") as file:
   for line in file:
       if line:
           data = line.split()
           # only adding words with AFINN of -4 or -5
           if int(data[-1]) >= -4:
               bf.add("".join(data[:-1]))
bf.write_to_file()
```

- filter_news_feeder.py (filters the stream of headlines)

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark import StorageLevel
import math
import hashlib
import mmh3
from bitarray import bitarray

# sc = SparkContext("local[2]", "question3")
ssc = StreamingContext(sc, 1)
global bf

def get_bloom_filter():
   bf = bitarray()
   with open("bloom.txt", 'rb') as file:
       bf.fromfile(file)
   return bf

def is_in_filter(word, bf):
   for i in range(13):
       h = mmh3.hash(word, i) % len(bf)
       if not bf[h]: # == 0
           return False
   return True

def RDD_helper(rdd, bf):
   headlines = rdd.collect()
   output = ""

   if not headlines:
       return
   for headline in headlines:
       output = 'Good headline: "{}"'.format(" ".join(headline))
```

```
       for word in headline:
           if is_in_filter(word, bf):
               output = 'Bad headline: "{}" -> Contains bad word
"{}"'.format(" ".join(headline), word)
               break
       print(output)

def setup_stream():
   bf = get_bloom_filter()
   lines = ssc.socketTextStream("localhost", 9999,
StorageLevel.MEMORY_AND_DISK)
   headline = lines.window(1)
   words = headline.map(lambda x: x.split(" "))
   words.foreachRDD(lambda rdd: RDD_helper(rdd, bf))

def launch_stream(w):
   ssc.checkpoint("checkpoint")
   ssc.start()
   ssc.awaitTermination(w)

setup_stream()
launch_stream(10)
```