CS4234/5234 Project Report

Question 1:

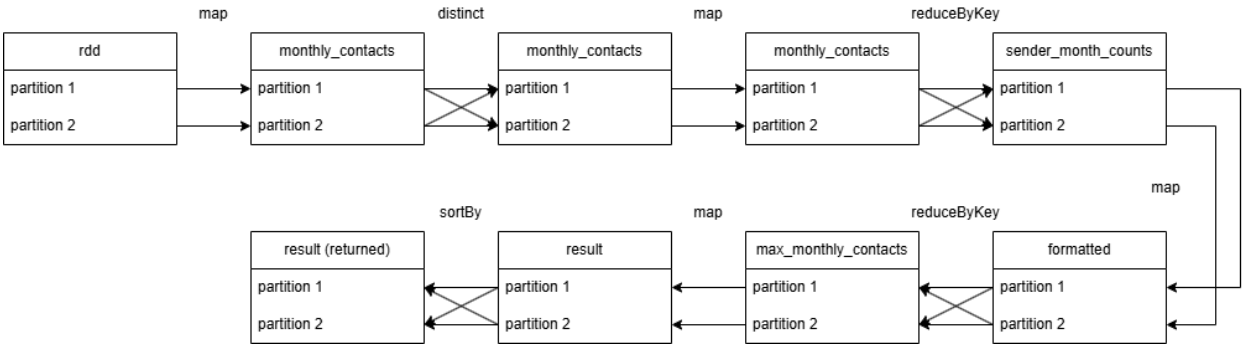
```python
# Q5: replace pass with your code
def get_monthly_contacts(rdd):
    monthly_contacts = rdd.map(lambda x: (x[0], f"{x[2].month}/{x[2].year}", x[1]))
    sender_month_counts = monthly_contacts.distinct().map(lambda x: ((x[0], x[1]), 1))\
                                          .reduceByKey(lambda a, b: a + b)
    formatted = sender_month_counts.map(lambda x: (x[0][0], (x[0][1], x[1])))
    max_monthly_contacts = formatted.reduceByKey(lambda a, b: a if a[1] >= b[1] else b)
    result = max_monthly_contacts.map(lambda x: (x[0], x[1][0], x[1][1]))
    return result.sortBy(lambda x: (-x[2], x[0]))
```

The function *get_monthly_contracts()* takes an RDD as input and computes, for each sender, the month where they contracted the highest number of unique recipients, the function follows these steps.

1. Extract month and year: the timestamp of each email is converted to the MM/YYY format
2. Ensure unique contacts: the function ensures that each recipient is counted only once per sender per month.
3. Count unique contacts per month: using *reduceByKey()* , it calculates the number of unique recipients each sender contacted per month
4. Identify the peak month: For each sender, the function determines the month with the highest number of unique contacts
5. Sorting: the results are sorted by the number of unique contacts (descending) and then by sender email (ascending)

Challenges:

- **Getting the correct date format:** unlike question 1 where it was easy to use the provided *date_to_dt()* function in this question I only need the month and year. If I wanted to use the provided function then I would have to cut it up, instead I decided to get the date different and instead just use the map function instead, and it would create cleaner code.
- **Getting the unique recipient counts per sender per month**: this is something that stumped me for a while, at first, I tried to do it all in one tying to think of different way of extracting the information I wanted. In the end what caused this to be so much of a challenge was not thinking to use *.distinct(),* which I eventually did get after going back to my notes.



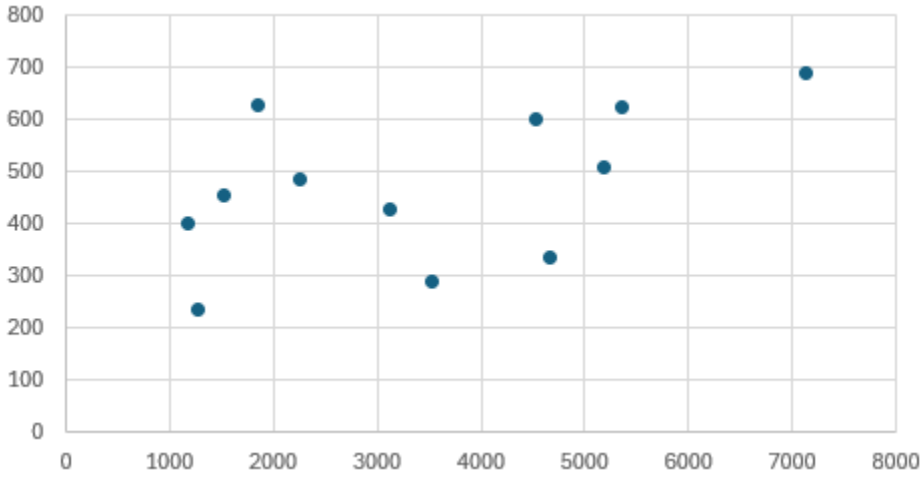
Narrow Dependencies: Map and Distinct

Wide Dependencies: Distinct, reduceByKey, and sortBy

Question 2:

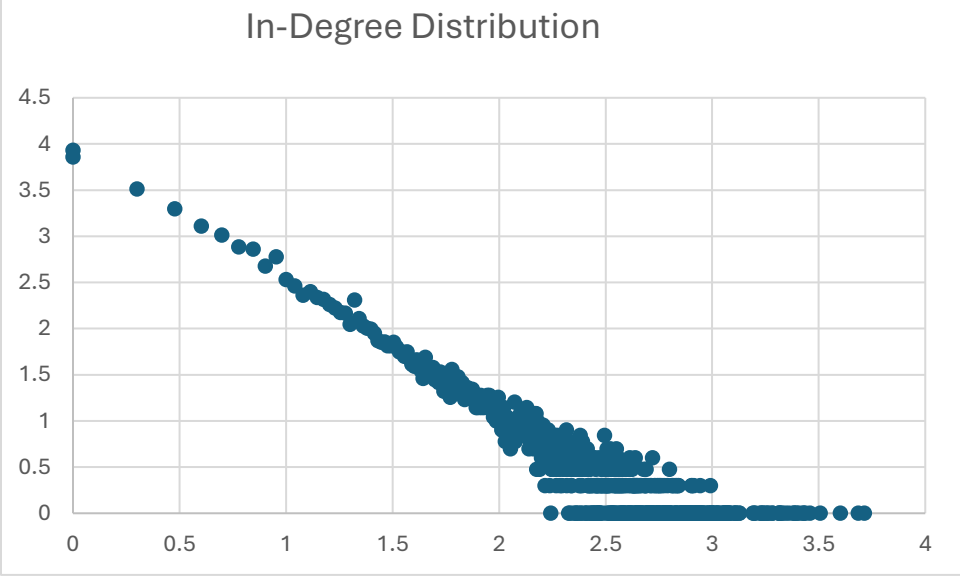
Note The code used to compute these answers is at the bottom of the report

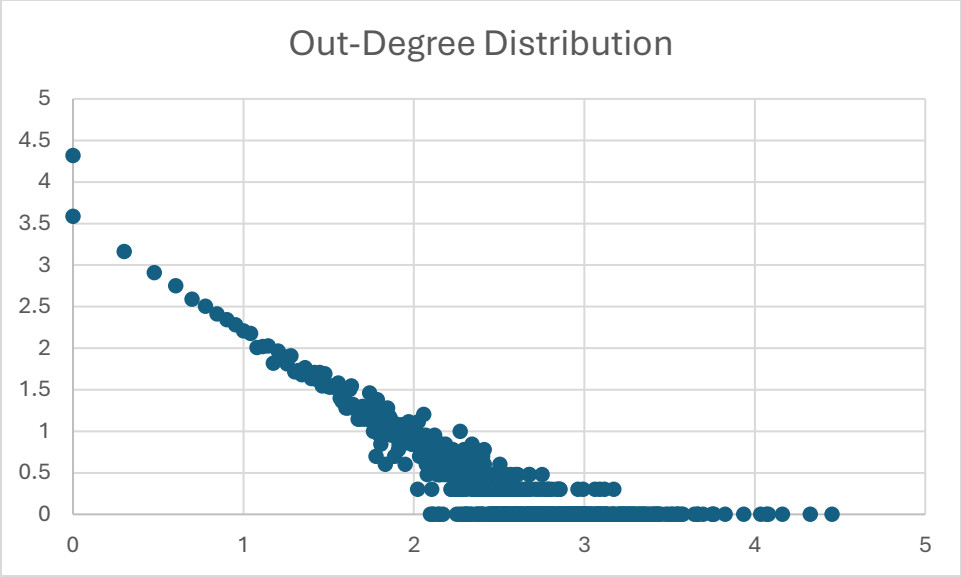
1. To Evaluate the 80/20 rule, I extracted the weighted network for the selected period using *conver_to_weighted_network()*. Then I computed the weighted in-degree and out-degree of each email using my answers from *project.py* . My calculations indicate that these high-degree nodes account for 99.08% of outgoing communications and 92.64% of incoming communications, confirming the presence of a small number of nodes that make up a large amount of the network traffic.
2. To examine the “the rich-get-richer” effect, I tracked the maximum node degree (*k_max*) over times, progressively adding monthly data slices and computing the corresponding number of nodes and edges. Using the functions *get_in_degrees()* and *get_out_degrees()* from *project.py* , I observed that *k_max* increases as more months were included, the results where: (3532, 290), (4660, 335), (5184, 510), (7137, 690), (4523, 600), (1515, 454), (1851, 628), (1274, 236), (1167, 402), (5362, 626), (2266, 487), (3130, 430). When I plotted this in Excel:



The relationship between *k_max* and the number of nodes appears approximately linear, supporting the idea that as the network expands, the most connected nodes continue accumulating an increasing share of traffic.

3. To test the hypothesis, I generated in-degree and out-degree distributions and then printed them into .csv files, these results were then plotted using Excel. I then plotted these on a log-log graph, where a linear trend would indicate power-law behavior. These are the graphs:





The results show a clear linear trend line, this confirms that the Enron email network follows a power-law degree distribution. This finding validates the hypothesis that the Enron email network exhibits the self-organizing properties characteristic of scale-free structures.

Question 2 Code:

```python
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession
from datetime import datetime, timezone
import numpy as np
from project import extract_email_network
import csv

conf = SparkConf().setAppName("Enron Network Analysis")
sc = SparkContext(conf=conf)
spark = SparkSession(sc)

def utf8_decode_and_filter(rdd):
    def utf_decode(s):
        try:
          return str(s, 'utf-8')
        except:
            pass
    return rdd.map(lambda x: utf_decode(x[1])).filter(lambda x: x != None)


def convert_to_weighted_network(rdd, drange=None):
    def filter_by_date(record):
        sender, recipient, timestamp = record
        if drange is None:
            return True
        return drange[0] <= timestamp <= drange[1]

    return (rdd.filter(filter_by_date)
              .map(lambda x: ((x[0], x[1]), 1))
              .reduceByKey(lambda a, b: a + b)
              .map(lambda x: (x[0][0], x[0][1], x[1])))


def get_out_degrees(rdd):
    print("Q3.1")
    all_nodes = rdd.flatMap(lambda x: [x[0], x[1]]).distinct()
    out_degrees = rdd.map(lambda x: (x[0], x[2])).reduceByKey(lambda a, b: a + b)
    final_rdd = all_nodes.map(lambda n: (n, 0)).leftOuterJoin(out_degrees)\
                         .map(lambda x: (x[1][1] if x[1][1] else 0, x[0]))\
                         .sortBy(lambda x: (-x[0], x[1]))

    return final_rdd

def get_in_degrees(rdd):
    all_nodes = rdd.flatMap(lambda x: [x[0], x[1]]).distinct()
    in_degrees = rdd.map(lambda x: (x[1], x[2])).reduceByKey(lambda a, b: a + b)
    final_rdd = all_nodes.map(lambda n: (n, 0)).leftOuterJoin(in_degrees)\
                         .map(lambda x: (x[1][1] if x[1][1] else 0, x[0]))\
                         .sortBy(lambda x: (-x[0], x[1]))
    return final_rdd

rdd = utf8_decode_and_filter(sc.sequenceFile("/user/ufac001/project/enron-full"))

email_rdd = extract_email_network(rdd)

date_range = (datetime(2001, 1, 1, tzinfo=timezone.utc), datetime(2001, 12, 31, tzinfo=timezone.utc))

weighted_rdd = convert_to_weighted_network(email_rdd, date_range)

out_degrees = get_out_degrees(weighted_rdd).collect()
in_degrees = get_in_degrees(weighted_rdd).collect()

total_out_degree = sum(d for d, _ in out_degrees)
total_in_degree = sum(d for d, _ in in_degrees)

top_20_percent = int(len(out_degrees) * 0.2) or 1
top_out_degree = sum(d for d, _ in sorted(out_degrees, reverse=True)[:top_20_percent])
top_in_degree = sum(d for d, _ in sorted(in_degrees, reverse=True)[:top_20_percent])

```

```python
k_max_values = []
node_counts = []
for n in range(1, 13):
    month_range = (datetime(2001, n, 1, tzinfo=timezone.utc), datetime(2001, n, 28, tzinfo=timezone.utc))
    monthly_rdd = convert_to_weighted_network(email_rdd, month_range)
    max_out = max(get_out_degrees(monthly_rdd).collect(), key=lambda x: int(x[0]) if isinstance(x[0], str) else x[0], default=(0, ""))[0]
    max_in = max(get_in_degrees(monthly_rdd).collect(), key=lambda x: int(x[0]) if isinstance(x[0], str) else x[0], default=(0, ""))[0]
    k_max_values.append((max_out, max_in))
    node_counts.append(monthly_rdd.count())

print(f"top_out_degree = {top_out_degree}")
print(f"total out degree = {total_out_degree}")
print(f"top in degree = {top_in_degree}")
print(f"total in degree = {total_in_degree}")
print("80/20 Rule Compliance (Out-degree):", top_out_degree / total_out_degree if total_out_degree else 0)
print("80/20 Rule Compliance (In-degree):", top_in_degree / total_in_degree if total_in_degree else 0)
print("Max Degree (k_max) Over Time:", k_max_values)

def get_out_degree_dist(rdd):
    out_degrees = get_out_degrees(rdd)
    degree_dist = out_degrees.map(lambda x: (x[0], 1)).reduceByKey(lambda a, b: a + b)
    return degree_dist.sortBy(lambda x: x[0])

def get_in_degree_dist(rdd):
    in_degrees = get_in_degrees(rdd)
    degree_dist = in_degrees.map(lambda x: (x[0], 1)).reduceByKey(lambda a, b: a + b)
    return degree_dist.sortBy(lambda x: x[0])

out_degrees = get_out_degrees(weighted_rdd)
in_degrees = get_in_degrees(weighted_rdd)

out_degree_dist = get_out_degree_dist(weighted_rdd).collect()
in_degree_dist = get_in_degree_dist(weighted_rdd).collect()

with open("out_degree_results.csv", mode='a', newline='') as write_file:
    writer = csv.writer(write_file)
    for entry in out_degree_dist:
        writer.writerow(entry)

with open("in_degree_results.csv", mode='a', newline='') as write_file:
    writer = csv.writer(write_file)
    for entry in in_degree_dist:
        writer.writerow(entry)


print(in_degree_dist)

sc.stop()
```

Note: I am happy to provide the original file, or data files for part 3