Distributed Graph Techniques to Quantify Social Media Engagement of Covid-19 Scientific Literature through Incremental Tweet Chain Measurements

Distributed Graph Techniques to Quantify Social Media Engagement of Covid-19 Scientific Literature through Incremental Tweet Chain Measurements

Michael L. Johns II

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2021

Abstract

This research combined existing distributed data and graph techniques with a novel approach, refined over many experiments, to quantify social media engagement. The developed methodology scores engagement through measures including unique users reached, number of coverage items about which the user posted, and number and length of social post chains. In calculating scores, the use of quality measures goes beyond page rank's more limited concern with the centrality of a user within the social reference graph structure. The experiments used data comprised of scientific literature from the CORD-19 dataset, Scopus citation graphs relating to CORD-19, and PlumX altmetrics with corresponding Twitter posts intersecting with papers in the CORD-19 and Scopus citation graphs. The most significant outcomes of the research are two scoring algorithms, Social Media Engagement (SME) and Social Media Noise (SMN), which quantify complexity of engagement, or lack thereof, by users within social data. The incremental design allows new data to be added to a cumulative score without recalculation. SME and SMN algorithms have wide applicability for all social data at any scale or velocity and could become the basis for a new class of altmetrics.

Acknowledgements

I would like to gratefully acknowledge several people who went on this adventure alongside me. First, I am eternally thankful for my wife Mary for taking on thousands of big and small extra tasks to allow me the freedom to conduct thesis research throughout 2020, a year marked nonetheless by a global pandemic, social distancing, and creative school options for Sophia, Annabelle, and Liliana. My family has been a boundless reservoir of strength and joy, and only heaven knows all the ways they have sacrificed so that I could achieve this goal after many years of study. I would also like to express my deepest appreciation to Harvard Extension School Senior Research Advisor Hongming Wang for skillfully guiding me through the twists and turns of the proposal and then graciously offering to see me all the way through execution as my Thesis Director. She worked tirelessly, even at off hours, to support me. Many thanks also to Director and Research Advisor Sylvain Jaume for his support and making this research possible. I would also like to extend my sincere thanks to Elsevier Labs Director Ronald Daniel for generously serving as my Industry Advisor throughout the thesis execution, helping me navigate a series of pivots to refocus research towards Covid-19. Many thanks to Elsevier VP of Research Collaborations Anita de Waard as well as Collaboration Manager Jennifer Pamphile for making themselves available to check-in with my research and finding ways to help remove any obstacles. I am very appreciative of support from Elsevier Technical Research Director Helena Deus for helping shape my research in the earliest

months of execution. I would also like to offer my special thanks to ICSR Associate Kristy James for cheerfully helping me with Scopus table extracts, gathering Twitter identifiers, and configuring Databricks.

Contents

Table of Contents **vi**

List of Figures **ix**

List of Tables **xii**

List of Equations **xiv**

List of Code **xv**

1 Introduction **1**

1.1 Graphs 1

1.2 Distributed Computation with Apache Spark™ 4

1.2.1 Spark DataFrames 5

1.2.2 Graph with Spark 6

1.3 Delta Lake 7

1.4 Databricks 8

1.5 Scopus 9

1.6 PlumX and Altmetrics 9

1.7 CORD-19 Dataset 11

2 Methodology **12**

2.1 ICSR Research Environment and Datasets 13
2.2 CORD-19 Scientific Literature 14
2.2.1 Combine Scopus and CORD-19 15
2.3 Data Modeling: CORD-19 Twitter 16
2.3.1 Tweet Data Engineering 16
2.3.2 Reference Tweets and Orphans 19
2.3.3 Identify Missing Reference Tweets 20
2.3.4 Tweet Blasters and Spammers 21
2.3.5 Graph Remodeling: Reference Tweets 22
2.3.6 Graph Measures and Algorithms 23
2.4 Graph Modeling: Social Media Trees and Chains 27
2.4.1 Calculate Roots 29
2.4.2 Derive Chains from Reference Tweets 31
2.4.3 Add User Information to Tweet Chains 35
2.4.4 Broken Social Media Chains 35
2.4.5 All Relative Chain Paths 36
2.5 Distributed Scoring Algorithms: Social Media Measures 40
2.5.1 Twitter Nomenclature 41
2.5.2 Social Media Engagement (SME) Formula and Implementation 41
2.5.3 Social Media Noise (SMN) Formula and Implementation . . . 44
2.5.4 Bounded SME and SMN Scoring 45
2.5.5 Synthetic Data 45
2.5.6 Per Category SME and SMN Scoring 52
2.5.7 Temporal Filters for SME and SMN Scoring 53
2.5.8 Incremental SME and SMN Scores 55

3 Experiments and Results **57**

3.1 CORD-19 and Scopus Information Propagation 57

3.2 CORD-19 and Twitter Data Interactions 63

3.2.1 CORD-19 1-Hop Tweet Counts 63

3.2.2 Compare CORD-19 Publish and Tweet Dates 64

3.2.3 Tweet Velocity 66

3.2.4 Tweet Blasts 67

3.2.5 Tweet Chains 67

3.3 Twitter CORD-19 Social Media Measures 70

3.3.1 SM Measures for Twitter CORD-19 70

3.3.2 SM Measures for ASJC Twitter CORD-19 75

3.3.3 SM Measures for Incremental Twitter CORD-19 76

3.4 Twitter CORD-19 Social Media Scores 86

3.4.1 SM Scores for Available Twitter CORD-19 86

3.4.2 SM Scores for ASJC Twitter CORD-19 90

3.4.3 SM Scores for Incremental Twitter CORD-19 91

4 Discussion **99**

5 Conclusions **103**

References **106**

List of Figures

1.1 Scopus data assets (Elsevier, 2020). 9

1.2 Elsevier PlumX metrics example Journal of Clinical Virology top social media articles. 10

1.3 Four ways to measure impact (Priem et al., 2010b). 11

2.1 Counts of new and removed papers over CORD-19 for 2020 (01 March - 19 Dec). 14

2.2 Resulting CORD-19 metadata schema nested and clean. 15

2.3 Tweet and paper venn diagrams (Scopus v.003 – 07-07-2020). 18

2.4 Non-orphan tweets by number of tweet reference field values populated. 20

2.5 Example tweet tree. 28

2.6 Example tree transformed into user graph. 29

2.7 Absolute and relative tweet roots. 30

2.8 Absolute and relative chains from tweet graph. 32

2.9 Counts for relative tweet chain lengths. 39

2.10 Synthetic SME ascending measure scores. 47

2.11 Synthetic SME inverse measure line chart. 48

2.12 Synthetic SME inverse measure scores. 49

2.13 Synthetic SME random measure scores. 50

2.14 Synthetic SMN and SME bar plot. 51

2.15 Scatterplot of random synthetic SME, SMN, and unfixed measures. . 52

3.1 Velocity chart showing week by week increases and decreases in added, removed, and published CORD-19 papers. 58
3.2 Relative velocity chart for publish and removal from CORD-19 in 2020. 58
3.3 Percentage of detected overall gender in CORD-19 Scopus 1-hop corpus. 59
3.4 2020 CORD-19 Scopus paper country counts. 60
3.5 2020 CORD-19 1-hop Scopus degree and page rank scatterplot. . . . 62
3.6 2020 CORD-19 Scopus component and page rank. 63
3.7 2020 1-hop CORD-19 PlumX tweet counts. 64
3.8 2020 CORD-19 paper dates and tweet dates (through July). 65
3.9 2020 Scopus publish dates and tweet dates (through July). 66
3.10 2020 Top-10 tweet velocity graphs (through July). 66
3.11 DOI tweet blasts and spam from top 25 most prolific users. 67
3.12 Tweet chain root, orphan, leaf, and non-leaf counts. 68
3.13 Tweet chain lengths, with percentage of sequential width and depth. . 69
3.14 Tweet chain length outliers. 70
3.15 Standard user SME scores bucketed by 10. 88
3.16 Compare number of posts against SME and SMN scores for top 1,000 users. 88
3.17 Compare SME and SMN scores for top 1,000 page rank users. 90
3.18 Calculated per ASJC SME and SMN scores. 91
3.19 Compare $H1_{w2020}$ incremental SME score with all available. 94
3.20 Compare rolling weekly chains sizes over 2020. 95
3.21 Compare SME scores applying each rolling weekly chain measure. . . 96

3.22 Compare SME score outliers for rolling lag4 and blended rolling lag4 measures. 96

List of Tables

1.1 Filter DataFrame to generate GraphFrame. 7

2.1 Join CORD-19 with ICSR Lab Scopus (v.005 – 12-12-2020). 16
2.2 Summary of Figure 2.3. 18
2.3 Reference tweet table. 19
2.4 In-degree top 5 for reference graph. 24
2.5 Out-degree top 5 for reference graph. 24
2.6 Page rank top 5 for reference graph. 25
2.7 Connected components output for reference graph. 26
2.8 Connected components top 5 counts for reference graph. 26
2.9 Combined measures top 5 counts for reference graph. 27
2.10 First 10 edges for example tweet tree. 29
2.11 Chain length 2 processing results. 33
2.12 Summary of reference tweet chain processing. 34
2.13 Calculation of lagging weeks. 55
2.14 Calculation of blended lagging weeks. 56

3.1 Top 25 2020 CORD-19 Scopus paper countries. 61
3.2 Scopus 1-hop citation graph measures. 61
3.3 First 10 SME scores, bucketed by 10. 87

3.4 Top 10 ASJC SME scores. 91
3.5 Top 5 incremental $H1_{w2020}$ SME scores. 93
3.6 Chain week comparison of single outlier user-1. 97
3.7 Chain week comparison of single outlier user-2. 98

List of Equations

2.1 Per user Social Media Engagement (SME) score formula. 42

2.2 Per user Social Media Noise (SMN) score formula. 44

2.3 Per ASJC Social Media Engagement (SME) score formula. 53

2.4 Per ASJC Social Media Noise (SMN) score formula. 53

List of Code

2.1 Hydrate tweets. 17

2.2 Identify missing reference tweets. 20

2.3 Most prolific tweet originators. 21

2.4 Group tweets about DOIs by top 25 most prolific users. 22

2.5 Graph edge DataFrame from reference tweets. 22

2.6 Graph vertex DataFrame from reference tweets. 22

2.7 Tweet roots from reference edges. 30

2.8 Function to find all chains of a specified length. 31

2.9 Transformation of tweet chains to users chains. 34

2.10 Transformation of tweet chains to users chains. 35

2.11 Explode tweet chains. 37

2.12 Derive relative user chains. 37

2.13 Python implementation of SME score algorithm. 43

2.14 Python implementation of SMN score algorithm. 44

2.15 Ascending synthetic data generation. 46

2.16 Descending and ascending synthetic data generation. 47

2.17 Random synthetic data generation. 49

3.1 Generate DataFrame for user posts measure. 71

3.2 Generate DataFrame for tweet coverage measure. 71

3.3 Generate DataFrame for user coverage measure. 72

3.4 Generate DataFrame for unique number of users reached. 72
3.5 Generate DataFrame with number of chains for each user. 73
3.6 Add columns for various calculated statistics. 74
3.7 Filter to ASJC and calculate per user posts. 75
3.8 Generate DataFrame for standard incremental weekly chain sizes. . . 78
3.9 Calculate SME and SMN scoring for measures. 86
3.10 Calculate ASCJ SME and SMN scoring from user scores. 90
3.11 Calculate user incremental SME and SMN scoring over $Q1_w$, $Q2_w$, and $H1_w$. 92

Chapter I.

Introduction

The goal of this research was to develop a methodology which accounts for the quality of social media engagement, or lack thereof, with scientific literature. The experiments were run on Twitter data filtered to tweets containing Digital Object Identifiers (DOI) affiliated with the CORD-19 (§1.7) dataset, expanded with Scopus (§1.5) citation graphs and PlumX (§1.6) altmetrics data. The methodology implementation combined existing distributed data and graph techniques with a novel approach to establish quality measures of tweet chain complexity, distinct users reached, and unique DOIs engaged. Further, the design is incremental, allowing cumulative weekly scoring for quarter, semi-annual, and annual without recalculation. This research methodology is distinct from existing altmetrics, and can blaze a path for future altmetrics which quantify complexity of social engagement.

1.1. Graphs

Graphs are essentially pair relationships between two vertices connected by edges. The very nature of graphs are so prevalent that most any discipline has at least one area where graph theory is applicable (Bruyr, 1965). Computational approaches based on graph theory can facilitate k-parallel search graph techniques such as k-depth, breadth depth, and breadth-first (Arjomandi & Cornell, 1975). In

scientific literature, the citation data represents a relation between two affiliations (Skulimowski, 2019). However, citation data alone lacks any objective meaning and internal structure (Skulimowski, 2019) and may have a noisy or incomplete context (Farber & Sampath, 2019). But when placed in context with other bibliometric measures, the various interactions among collaborators, and their affiliations, can be readily modeled and studied using graph structures (Montoya et al., 2018), even ones that account for complexity within the model (Nanumyan et al., 2020). In social networks graphs can be generated from relations between posts or users.

i. Degree: The number of vertices to which a given vertex v is linked by an edge becomes the degree of v. For directional edges, in-degree is the count of edges where a given vertex v is the destination in graph edges, such as edge $e_{ba} = v_b \rightarrow v_a$ where v_a in-degree would be incremented by 1. In the same edge, vertex v_b in-degree would remain constant. Conversely, out-degree is the count of edges where a given vertex v is the source in graph edges. For edge e_{ba} vertex v_a out-degree would remain constant and vertex v_b out-degree would be incremented by 1.

ii. Aggregate Message Passing: Send messages between vertices, and aggregate messages for each vertex.

iii. Pregel Algorithm: Pregel is a bulk-synchronous message-passing API for implementing custom iterative graph algorithms (Malewicz et al., 2010).

iv. Page Rank: Introduced by Google (and borrowed from academic citation literature) to provide an objective measure of the importance of a web page, allowing web search engines make use of the citation graph of the web in a manner that people would subjectively rank pages (Brin & Page, 1998). The

algorithm counts in-links to a node as a measure of its importance. It is often implemented within a Pregel API. Page rank can serve as a proxy for quantitative measures of academic reputation with the rationale being that citations are, to some extent, driven by the reputation, or popularity, of the referenced scientific literature (Massucci & Docampo, 2019). In social networks in links to a node are the posts and repost engagement of other users. Page rank is limited to edge-based, or first-order, relations between two connected nodes; therefore page rank will not be able to identify higher-order relations between nodes, when present (Zhao et al., 2019).

v. Belief and Label Propagation: Pass local messages for community detection. This class of algorithms has utility in solving inference problems with many applications including those found within AI, computer vision, and error-correcting coding theory (Yedidia et al., 2003). Label propagation algorithm (LPA) can be less computationally expensive than connected components with tradeoffs. First, the message passing may not converge in dense networks leading to incomplete components identified or may end up with trivial results such as 1 community (Raghavan et al., 2007). Second, component roots get a separate label from the vertices within the component, so extra steps are required to fully tie the community together.

vi. Connected Components: Community detection iterative algorithm to efficiently partition a graph into connected components, biconnected components, and simple paths where each iteration identifies a new path between two vertices already on known paths (Hopcroft & Tarjan, 1973). The component id is set to the component's lowest numbered vertex. For temporally ordered data, such as social media, the component id will also be the earliest vertex. For social media

data, the earliest vertex in a component is also the root of the post tree.

vii. Strongly Connected Components (SCC): Like connected components except requires a path for each pair of vertices assigned to a SCC where connected components only requires a minimum of 1 path between any vertex and any other vertex in the component.

1.2. Distributed Computation with Apache Spark™

Distributed computation systems are able to process and analyze large-scale data through batch and streaming interfaces using multiple machines, called nodes, to perform massively parallel processing (MPP) over a set of coordinated computations. There have been a number of distributed programming models which target types of compute workloads: Google first introduced MapReduce for distributed batch, Dremel for distributed, interactive SQL, and Pregel for distributed, iterative graph (Zaharia et al., 2016). Many open source alternatives have also been introduced to enable specialized distributed computation: there is the Apache Hadoop stack and ecosystem which includes disparate specialized systems such as Storm for streaming, Impala for SQL, Giraph for graph, and Mahout for machine learning. Given that big data is, by nature, quite diverse and often messy, the reality is that big data applications often combine many different processing types: a typical pipeline will need MapReduce-like code for data loading, SQL-like queries, and iterative machine learning (Zaharia et al., 2016). The use of specialized engines, over generalized ones, can increase overall complexity and inefficiency, forcing users to integrate multiple specialized systems, with still a number of applications which cannot be expressed efficiently in any engine (Zaharia et al., 2016).

The Apache Spark™ project grew out of University of California, Berkeley in

2009 with the goal of developing a unified distributed data processing engine (Zaharia et al., 2016). Spark extended the MapReduce programming model with a shared data abstraction called Resilient Distributed Datasets (RDD) (Zaharia et al., 2010), making it a generalizable engine to a wide range of workloads to include batch, streaming, SQL, graph, and machine learning (Zaharia et al., 2016). Spark allows various libraries to run over a common engine using a programming abstraction without performance loss, which makes applications straightforward and efficient to compose (Zaharia et al., 2016). Spark's memory-primary architecture significantly reduces I/O round trips to underlying storage. It can outperform Hadoop by 10x or more in machine learning applications, which are iterative in nature, and can be used to interactively query large-scale datasets with sub-second response time (Zaharia et al., 2010). Spark is written in Scala but has language bindings for Java, Python, R, and SQL. Spark has become one of the most popular open source projects for big data analytics with over 1,000 contributors and support from virtually every major commercial vendor (Armbrust et al., 2016). Additionally, since Spark separates compute from storage, it is well-suited for ephemeral, elastic compute infrastructure such as provided by Cloud Service Providers (Armbrust et al., 2010; Islam et al., 2020).

1.2.1 Spark DataFrames

The Spark DataFrames package builds on the popular Data Science abstractions used in Python (pandas) and R languages with labeled rows and columns and a syntax for querying and transforming the structure. DataFrames build on Spark foundational APIs to allow interfaces which are more relational and structural, based on SQL concepts and syntax (Armbrust et al., 2016). Whereas with the lower-level RDD API programmers fully express how to accomplish an intention through directly executed code, with DataFrames programmers express what they want to accom-

plish and allow the Catalyst relational optimizer to plan appropriate execution and then perform whole-stage code generation based on the final, optimized physical plan (Armbrust et al., 2016). Further, user-defined functions (UDF) allow programmers to express custom or third-party code through supported Spark language bindings, providing a balance between generated code and language-specific libraries.

1.2.2 Graph with Spark

Spark core includes GraphX as part of its unified APIs, a lower-level, RDD implementation for graph-parallel operations with Scala-only APIs. GraphX offers iterative algorithms for page rank, connected components, label propagation, matrix factorization, and triangle counting.

GraphFrames is a Spark package for DataFrame-based graph parallel operations with high-level APIs in Scala, Java, and Python. GraphFrames capabilities combine iterative algorithms, such as found in GraphX, with pattern matching, graph algorithms, and relational queries using joins, and optimizes the work across them, which enables Spark parallel execution and integration with custom code (Dave et al., 2016). Pattern matching in GraphFrames is called motif finding.

The GraphFrames package provides distributed algorithm implementations for breadth first search (BFS), connected components, strongly connected components, label propagation algorithm (LPA), page rank, shortest paths, and triangle counting. It also offers aggregate message passing to include customized algorithms based on pregel and offers a customizable belief propagation implementation.

All the power of DataFrames is available when working with GraphFrames. A common pattern is to filter a full data set prior to generating a graph. In the Table 1.1 example, a call like `filtered_vertex_df = vertex_df.filter("category < 20")` can then be used to generate a GraphFrame. Due to the filter condition, the row with

"XYZ Journal" is not included in the results since its `category` is `>= 20`.

Table 1.1: Filter DataFrame to generate GraphFrame.

	id	**journal**	**category**	**additional_props**
01	DOI_1	ACME Journal	1	...
02	DOI_2	PQR Journal	10	...
03	DOI_3	XYZ Journal	20	...
04	DOI_4	PQR Journal	15	...

If connecting papers to papers in a directed graph, the vertices in the graph would be DOI_1, DOI_2, DOI_3, DOI_4. For edges, if relating papers based on presence in the same journal, then edges would be $DOI_2 \rightarrow DOI_4$ and $DOI_4 \rightarrow DOI_2$ since they both share "PQR Journal".

1.3. Delta Lake

Delta Lake is an open source ACID table storage layer over cloud object stores (Armbrust et al., 2020), which adds a transaction log, or journal, to Parquet columnar data. The log file additionally maintains statistics of the data for performant query planning and allowing for data skipping (Armbrust et al., 2020). Additionally, the underlying files of a Delta Lake table can be compacted to a configured size, such as 256MB, to avoid the proliferation of small files as often encountered in low-latency streaming applications, resulting in inefficiencies listing data on cloud storage, I/O bottlenecks, and under-utilization of compute resources during read operations from inefficient task parallelism (Armbrust et al., 2020).

1.4. Databricks

Databricks founders are among the original inventors of the previous Spark project conceived out of UC Berkeley's AMPLab in 2009, which was open sourced in 2010, and moved to the Apache Software Foundation in 2013 where it has resided since as Apache Spark™ (§1.2). Databricks employees comprise 26 of Spark's current 83 project management committee (PMC) members who drive a significant number of major features within the project. Other popular open source sponsored by Databricks includes Delta Lake (§1.3) for reliability and performance over cloud storage, mlflow to manage the machine learning lifecycle, Koalas for pandas syntax over Spark, and Redash for SQL analytics over big data. In total, Databricks sponsored open source generates 8M downloads per month from around the world. Databricks has developed a number of enhanced packages for Spark and the big data and AI ecosystem around it.

Beyond sponsoring open source, Databricks offers an enterprise unified data analytics platform which integrates natively with cloud infrastructure and services. It is currently available on Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). The Databricks platform offers performance and reliability advantages over open source. For example and relevant to this research, Databricks built a transparent data and journal caching feature over Delta Lake tables which significantly increases data and metadata query performance (Armbrust et al., 2020). Also relevant to this research, GraphFrames has been further optimized within the Databricks Runtime for Machine Learning to improve connected components and motif finding (§1.1). Databricks offers integrated tooling for various data team personas. Data engineers can generate and execute automate reliable jobs over sophisticated multi-hop data pipelines powered by Delta Lake optimizations, data scientists can

use the Jupyter compatible notebook workspace to run interactive advanced analytics over CPU or GPU compute clusters, and data analysts can use the SQL workspace or connect through business intelligence (BI) tools for low or no code tooling.

1.5. Scopus

Scopus is an abstract and citation database of peer-reviewed literature by Elsevier for scientific journals, books and conference proceedings in the fields of science, technology, medicine, social sciences, and arts and humanities. It contains 1.4 billion cited references dating back to 1970, 70 million items, 16 million author profiles, 150,000 books, 70,000 main institution profiles, 22,000 serial titles, and 5,000 publisher profiles. Scopus also enables researchers to discover and interact with bibliometric data. Scopus is among the largest curated abstract and citation databases, while ensuring only the highest quality data are indexed through rigorous content selection and re-evaluation by an independent Content Selection and Advisory Board (Baas et al., 2020).

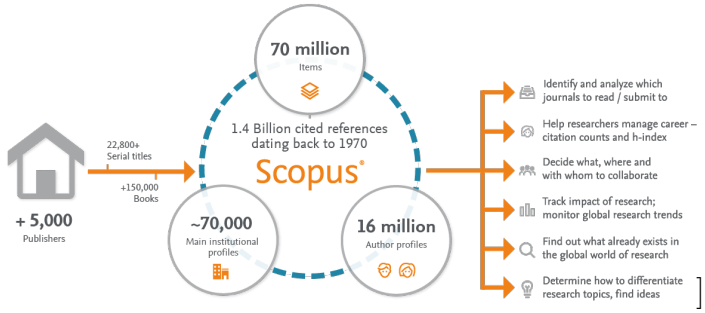

Figure 1.1: Scopus data assets (Elsevier, 2020).

1.6. PlumX and Altmetrics

PlumX is one of a number of analysis tools which uses a relatively newer category of alternative metrics, called altmetrics (Priem et al., 2010a), alongside tra-

ditional measures of research impact. PlumX metrics are available within Scopus through the company Plum Analytics which was acquired by Elsevier in 2017. Altmetrics go beyond journal citations and journal impact metrics. Altmetrics measure research impact in the short term, before the work is cited; they also identify uses of scientific literature beyond citations (Swoger, 2013). Additionally, altmetrics can be implemented to measure the impact of work which are beyond traditional peer reviewed articles to include datasets, software, and presentation slides (Swoger, 2013). Didegah et al. study showed that measures such as journal impact factor, individual collaboration, international collaboration, institution prestige, country prestige, research funding, abstract readability, abstract length, title length, number of cited references, field size, and field type can be modeled and correlated in association with citation counts and altmetrics such as Twitter posts, Facebook posts, blog posts, news posts, and Mendeley readers (Didegah et al., 2018). Altmetrics can also be used to improve search engine results and recommender systems which are domain-aware for the researcher, making the process of conducting research more efficient and contextualized (Magara et al., 2017).

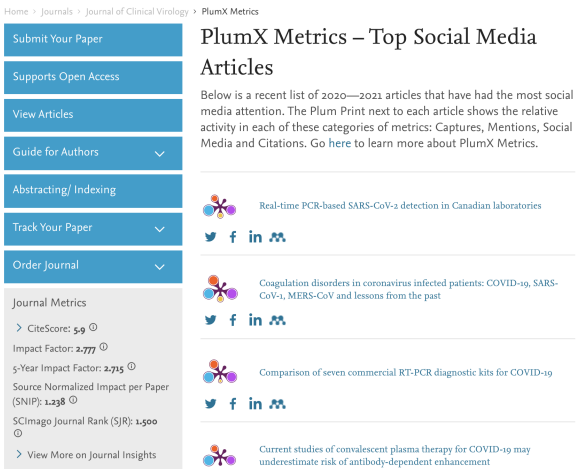

Figure 1.2: Elsevier PlumX metrics example Journal of Clinical Virology top social media articles.

1.7. CORD-19 Dataset

The CORD-19 dataset is a large and growing collection of publications and preprints on Covid-19 and related historical coronaviruses such as SARS and MERS initially released on March 16, 2020 (Wang et al., 2020). It has a wide number of partners including the Allen Institute for AI (AI2), the White House Office of Science and Technology Policy (OSTP), the National Library of Medicine (NLM), the Chan Zuckerburg Initiative (CZI), Microsoft Research, and Kaggle, and is also organized by Georgetown University's Center for Security and Emerging Technology (CSET) (Wang et al., 2020). The initial release contained around 28K papers and grew considerably over to around 400K by the end of 2020.

Figure 1.3: Four ways to measure impact (Priem et al., 2010b).

Chapter II.

Methodology

The research was conducted with three components: (i) for Covid-19, (ii) for information propagation, and (iii) for identifying novel techniques.

i. Given the unprecedented urgency of a global pandemic and the rapidly growing scale of information transfer, identify and apply distributed data and graph techniques to datasets affiliated with Covid-19. Leverage traditional techniques and existing processed datasets as needed.

ii. Study information propagation properties with the affiliated Covid-19 datasets. Consider propagation of scientific literature, their references, and contextualizing social media data. Specially assess graph chains and clustering with time-bound intervals and categorical restrictions. The research design will amplify software engineering elements such as remodeling data through data engineering to support many different experiments. Further, the research will seek to apply the established techniques to incomplete and dynamically changing data.

iii. Potentially identify novel and formalized uses of existing data or graph techniques and apply them in a new way relating to the primary datasets. Depending on the outcomes, run any improvements on common benchmark datasets to position the research within literature.

2.1. ICSR Research Environment and Datasets

The research was conducted within Elsevier's International Center for the Study of Research (ICSR) Lab. The ICSR Lab is a cloud-based computational platform, powered by Databricks, which enables researchers to analyze large structured and unstructured datasets, including those that power Elsevier solutions such as Scopus and PlumX. The datasets used within ICSR Lab relevant to the focus of the research are as follows:

i. CORD-19 (§1.7) – Publications and preprints on Covid-19 and related historical coronaviruses.

ii. SCOPUS (§1.5) – Citation database for papers referenced by CORD-19 dataset as well as the papers they reference, referred to as the 1-hop corpus. There were 2 releases used in the experiments, v.003 from July and v.005 from December 2020.

iii. PlumX (§1.6) – Altmetrics on CORD-19 paper impact provided by Plum Analytics and available within Scopus.

iv. Twitter – Set of tweets, delivered through coordination with Plum Analytics, filtered to DOIs found within the super set of CORD-19 and Scopus 1-hop scientific literature.

Experiments were conducted within the ICSR Lab's AWS Databricks environment. The cluster made available for experiments was able to scale up to 5 i3.xlarge worker nodes where each node had 30.5GiB Memory, 4 vCPUs, and 0.950 TB NVMe SSD. ICSR made available table extracts from Scopus and PlumX from the 1-hop corpus protected by IAM roles to isolate the research from other projects within the

environment, enforced through cluster policies. The clusters used Databricks Runtime for Machine Learning (§1.4) version 7.3 which uses an optimized Spark version 3.0 as well as GraphFrames version 0.8.0 in addition to other libraries for data science. Where possible, experiments were executed in single-node cluster mode for more efficient use of compute resources.

2.2. CORD-19 Scientific Literature

In order for the CORD-19 (§1.7) data to be ready for the experiments, it was processed in multiple phases and stored in a Delta Lake table which included all changelogs from March through 19 December 2020. This table captured aggregate paper count changes, schema changes throughout the year, and dates for each new metdata release. Figure 2.1 give the dates and counts for paper additions and removals over 2020 with an exceptionally large spike of papers being added in mid-May as well as a couple of increases in paper removals in mid-May, early June, and mid-August.

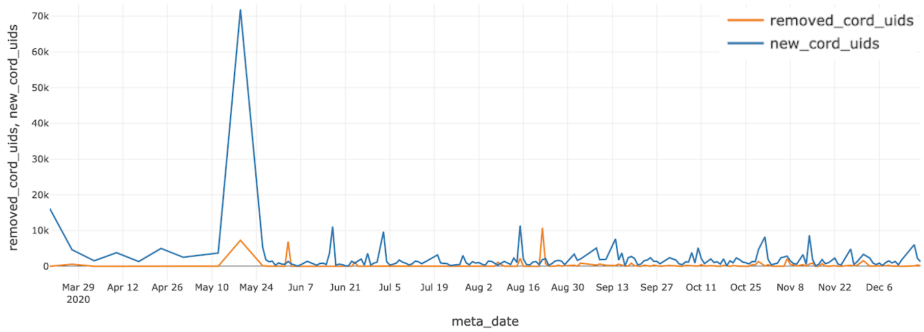

Figure 2.1: Counts of new and removed papers over CORD-19 for 2020 (01 March - 19 Dec).

The changelog table was used to generate the CORD-19 metadata Delta Lake table, produced by standardizing each metadata release which was on weekly frequency up to May 26, 2020 and then daily for the remainder of the year. The

changelog identified multiple schema changes and captured date of first availability of a paper within CORD-19 and, if it was removed, the date of removal. Figure 2.2 shows the schema of the resulting Delta Lake table used in the experiments.

Figure 2.2: Resulting CORD-19 metadata schema nested and clean.

2.2.1 Combine Scopus and CORD-19

In preparation for experiments, the CORD-19 data was combined with Scopus (§1.5) within ICSR Lab to produce the following Scopus table extracts from the intersection of CORD-19 papers found within Scopus and any papers they reference, referred to as the 1-hop corpus within this research:

Table 2.1: Join CORD-19 with ICSR Lab Scopus (v.005 – 12-12-2020).

Delta Table Name	**Scopus Version**
ani_cord19_1hop.delta	20201201
apr_cord19_1hop.delta	20201201
gender_inference_cord19_1hop.delta	20200304
ipr_cord19_1hop.delta	20201201
plumx_cord19_1hop.delta	20201201
sources_cord19_1hop.delta	20201113
top_topic_eid_cord19_1hop.delta	20201130
top_topic_keywords_cord19_1hop.delta	20200722
top_topic_prominence_cord19_1hop.delta	20200722
top_topic_topiccluster_cord19_1hop.delta	20200722

2.3. Data Modeling: CORD-19 Twitter

This section highlights some of the work required to prepare the Twitter data for research experiments. Tweets of interest were hydrated from the CORD-19 1-hop Scopus DOIs (§2.3.1), processed reference tweet data (§2.3.2), handled missing reference tweets (§2.3.3), identified tweet blasters and spammers (§2.3.4), and remodeled the tabular data into graph form (§2.3.5) for uses discussed in the social media tree and chain section (§2.4). Additionally, degree, page rank, and connected components were applied to the reference graph model (§2.3.6).

2.3.1 Tweet Data Engineering

In coordination with Elsevier ICSR Lab, a total of 1.8M DOIs were identified from which PlumX generated the extract of Twitter results through an offline custom operation. These DOIs, stored in `df_master_dois`, were the superset of CORD-19 metadata unioned with Scopus v.003 as of July 07, 2020 as well as the union of papers referenced by the intersection of the CORD-19 metadata with Scopus.

```
df_master_dois = (
  df_scopus.select("doi")
    .union(df_plumX.select("doi"))
    .union(df_cord.select("doi"))
  .distinct()
)
```

The resulting DOIs had approximately 82K in common with the CORD-19 data, with 1.2M only in Scopus from the additional 1-hop references and another 496K which were only in PlumX from all the additional altmetrics information they provide. Due to the normal lag time in loading Scopus extracts in ICSR as well as wider coverage of CORD-19 papers than what Scopus provided. From the DOIs in CORD-19, 68K were not available in Scopus at the time of the experiments.

Python package twarc was used to hydrate the tweets using a Twitter developer account. The Twitter API was limited to 100 `tweet_ids` per request and rate limited to 900 requests per 15-minute window, allowing a maximum of 90K tweets to be hydrated every 15 minutes. It only took around 10.25 minutes to hydrate 90K `tweet_ids` from within the ICSR Databricks environment, so each iteration ended up sleeping approximately 4.75 minutes. At approximately 16M initial rows to hydrate, this operation took 1.86 days to complete yielding 179 files.

Listing 2.1: Hydrate tweets.

```
for i in range(0, 180):
    part_n = str(i).zfill(5)
    path_src = f'/dbfs/mnt/proj_045/twitter/twarc_ids/part-{part_n}.csv'
    path_dst = f'/dbfs/mnt/proj_045/twitter/twarc_out/part-{part_n}.json'
    if os.path.isfile(path_dst):
        continue
    with open(path_src) as f:
        ids = f.read().splitlines()
    start_time = time.localtime()
    tweets = []
    for tweet in twarc_client.hydrate(ids):
        tweets.append(json.dumps(tweet))
    with open(path_dst, 'w') as f:
        f.write('\n'.join(tweets))
    end_time = time.localtime()
```

After processing, analysis showed there were 12.4M distinct `doi` and `tweet_id` rows, but really only 8.6M distinct `tweet_ids` since a number of tweets include more than a single DOI and there were 392K deleted tweets. Once the tweets were hydrated, they were standardized through multiple operations which produced 132 fields.

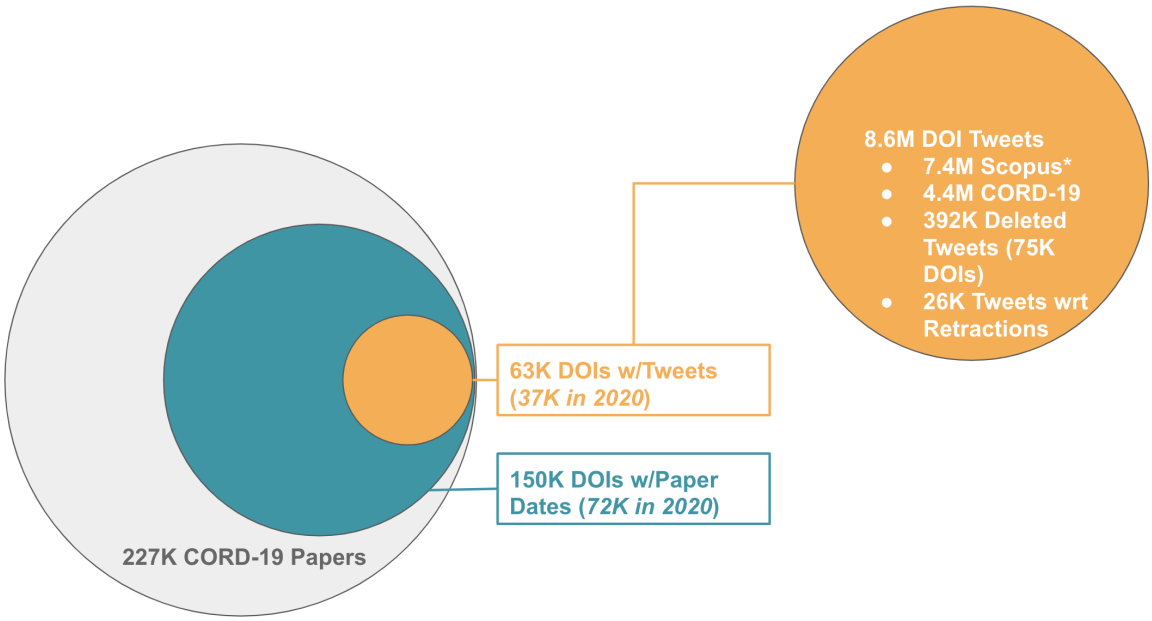

Figure 2.3: Tweet and paper venn diagrams (Scopus v.003 – 07-07-2020).

Table 2.2: Summary of Figure 2.3.

Papers in JUL CORD-19	227,102
DOIs having tweets	63,268 total (37,180 in 2020)
Hydrated tweets	11,927,267 (duplicates from scraping)
Unique tweets	8,679,169
Tweets related to Scopus DOIs	7,400,002 (includes 1-hop reference papers)
Tweets related to CORD-19 DOIs	4,401,015
Deleted tweets not hydrated	391,704

2.3.2 Reference Tweets and Orphans

Each tweet may reference other tweets by having values in any of the following fields:

```
tweet_in_reply_to_status_id, tweet_quoted_status_id, tweet_quoted_status--
    in_reply_to_status_id, tweet_quoted_status--quoted_status_id, retweet_id,
    retweet_in_reply_to_status_id, retweet_quoted_status_id, retweet_quoted_status
    --in_reply_to_status_id, retweet_quoted_status--quoted_status_id.
```

A tweet will be an orphan when it does not have values in any of the reference fields and it cannot be found in the reference fields of other tweets. Essentially an orphan can be described as a tweet which never engages and was never engaged by other tweets. Field `orphan_tweet` was added to `df_tweet`, resulting in 21% of tweets having `orphan_tweet = True`. Data processing filtered out all orphan tweets to produce the reference tweet DataFrame with the structure listed in Table 2.3.

Table 2.3: Reference tweet table.

	ref_tweet_id	**tweet_id**	**ref_tweet_type**	**is_tweet_id**
1	60915661970477057	254623451028017152	tweet_in_reply_to_status_id	false
2	147031327860985856	147213236314443776	retweet_id	false
3	147815834377650176	147860604613431296	retweet_id	true
4	147815834377650176	148082729253212161	retweet_id	true
5	147815834377650176	148589079544610816	retweet_id	true

From Figure 2.4 23% of the reference tweets have values populated in more than 1 of the reference fields, such as rows 3 to 5 in Table 2.3. The other 77% have values populated in only 1 of the reference fields.

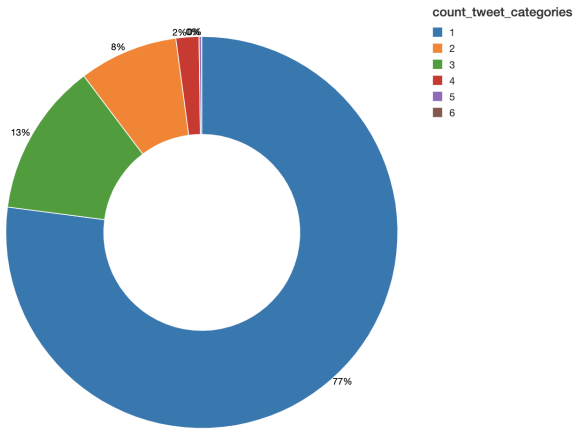

Figure 2.4: Non-orphan tweets by number of tweet reference field values populated.

2.3.3 Identify Missing Reference Tweets

The reference tweet DataFrame from Table 2.3 was joined with the original tweet DataFrame which added field `is_tweet_id` to identify missing data. Field `is_tweet_id` was set to `True` when `ref_tweet_id` was found in the actual data. When `ref_tweet_id` was missing from the actual data, `is_tweet_id = False`. Excluding orphans, 80% of the reference tweets were found in the actual data and 20% were not. The identification of missing reference tweets was used further to process tweet chains in §2.4.

Listing 2.2: Identify missing reference tweets.

```
df_ref_tweet_lookup = (
    df_tweet_category_lookup
        .join(
            df_ref_tweet_ids.selectExpr("tweet_id as ref_tweet_id")
                .withColumn("is_tweet_id", F.lit(True)),
            ["ref_tweet_id"],
            "left"
        )
        .fillna(False)
)
```

2.3.4 Tweet Blasters and Spammers

For various reasons, there were a number of user accounts which post about the same DOI many times over. As the number of repeated posts from the same user increase about a given DOI, the blast activity also becomes a candidate for spam behavior. In Listing 2.3 the users who originate tweet trees – those generating absolute roots – were ranked using Spark `Window` function `rank()` from most prolific (rank 1) to least prolific. An originating tweet does not reference any other tweet when posted, either through widening (repost) or deepening (reply) operations. The schema after the prolific ranking was the following:

```
tweet_user--id (long), tweet_user--screen_name (string),
    user_originator_tweet_rank (int), user_originator_tweet_count (long).
```

Listing 2.3: Most prolific tweet originators.

```
df_user_originator_rank = (
  df
    .filter("`tweet_created_at^^year` = 2020")
    .select(
      "tweet_id","tweet_user--id","tweet_user--screen_name","
    tweet_in_reply_to_user_id","tweet_quoted_status--in_reply_to_user_id","
    retweet_in_reply_to_user_id",
      "retweet_quoted_status--in_reply_to_user_id","tweet_quoted_status_user--id",
    "retweet_user--id","retweet_quoted_status_user--id"
    )
      .distinct() # important for accuracy
      .filter(col("tweet_in_reply_to_user_id").isNull())
      .filter(col("tweet_quoted_status--in_reply_to_user_id").isNull())
      .filter(col("retweet_in_reply_to_user_id").isNull())
      .filter(col("retweet_quoted_status--in_reply_to_user_id").isNull())
      .filter(col("tweet_quoted_status_user--id").isNull())
      .filter(col("retweet_user--id").isNull())
      .filter(col("retweet_quoted_status_user--id").isNull())
    .groupBy("tweet_user--id","tweet_user--screen_name")
      .count()
    .withColumn("user_originator_tweet_rank", F.rank().over(Window.orderBy(F.desc(
    'count'))))
    .select("tweet_user--id", "tweet_user--screen_name", "
    user_originator_tweet_rank", col("count").alias("user_originator_tweet_count")
    )
)
```

In Listing 2.4 the top 25 most prolific users for available data in 2020 were assessed for any blast activity to count each user's tweets containing a given DOI. The results will be discussed in §3.2.4.

Listing 2.4: Group tweets about DOIs by top 25 most prolific users.

```
df
  .filter("`tweet_created_at^^year` = 2020")
  .join(
    df_user_originator_rank
      .filter("user_originator_tweet_rank <= 25")
      .drop("tweet_user--screen_name"),
    ["tweet_user--id"]
  )
.groupBy("tweet_user--id","tweet_user--screen_name","user_originator_tweet_rank","
    user_originator_tweet_count","tweet_doi")
.count()
.filter("count > 10")
.orderBy(F.desc("count"))
```

2.3.5 Graph Remodeling: Reference Tweets

The edges for the reference tweet graph were produced from the same DataFrame listed in Table 2.3 and Listing 2.2 such that `tweet_id` becomes `src`, `ref_tweet_id` becomes `dst`, `ref_tweet_type` becomes `relation`, and `is_tweet_id` becomes `dst_data`.

Listing 2.5: Graph edge DataFrame from reference tweets.

```
df_e = (
    df_ref_tweet_lookup
        .selectExpr("tweet_id as src", "ref_tweet_id as dst", "ref_tweet_type as
    relation", "is_tweet_id as dst_data")
        .distinct()
)
```

The vertices for the reference tweet graph were generated from the distinct union of `src` and `dst` which becomes field `id`. All of the `src` values were found in the data represented by `id_data = True`. The `dst` values were found in the data only when `dst_data = True`.

Listing 2.6: Graph vertex DataFrame from reference tweets.

```
df_v = (
    df_e
        .selectExpr("src as id") # tweet_ids
            .withColumn("id_data", F.lit(True))
    .union(
        df_e
        .selectExpr("dst as id", "dst_data as id_data") # ref_tweet_ids
    )
    .distinct()
)
```

The edge and vertices DataFrames were the parameters for the Spark GraphFrame used in follow on graph operations: `g = GraphFrame(df_v, df_e)`.

2.3.6 Graph Measures and Algorithms

Graph Caching. The reference graph contained 9.5M distinct vertices and 10.2M distinct edges. This size was larger than the initial Twitter hydrated data due to the presence of reference tweets which were not actually in the available data. For processing degree, page rank, and connected components the graph could easily be cached and repartitioned in-memory, given the relatively small graph size, within the scalable 5 worker i3.xlarge cluster available for the research within the ICSR Lab Databricks environment. Repartitioning was accomplished by `repartitionByRange(...)` called on long field `id` for vertices and on long fields `src` and `dst` for edges. Spark GraphFrame `g_cache` was constructed from the vertex and edge DataFrames which were each cached with the repartitioning through a call to `cache()`.

```
df_v_cache = spark.read.load(v_delta_path).repartitionByRange("id").cache()
df_e_cache = spark.read.load(e_delta_path).repartitionByRange("src","dst").cache()
g_cache = GraphFrame(df_v_cache, df_e_cache)
```

Degree. In-degree was calculated on the graph through a call to GraphFrames `inDegrees`, with the top 5 tweet `ids` shown in Table 2.4. The highest `id` was 1251538798292467712 with `inDegree` of 115K.

```
df_in_degrees = g_cache.inDegrees.orderBy(F.desc("inDegree"))
```

Table 2.4: In-degree top 5 for reference graph.

	id	**inDegree**
1	1251538798292467712	115,964
2	1268613313702891523	52,418
3	1285207186591887360	40,249
4	1234320094018228228	39,661
5	1245547129831067649	24,504

Out-degree was similarly calculated on the graph through a call to `outDegrees`. The top 5 tweet `ids`, shown in Table 2.5, all have value 6 due to a single tweet only being able to have out-degrees to a global maximum of 9 reference fields. It will be described in §2.3.2, with 6 fields being the local maximum for this data.

```
df_out_degrees = g_cache.outDegrees.orderBy(F.desc("outDegree"))
```

Table 2.5: Out-degree top 5 for reference graph.

	id	**outDegree**
1	1093874535928135686	6
2	1255316853880168449	6
3	1127383338803589120	6
4	1246778087414603788	6
5	1254908977076293632	6

Page Rank. Page rank was called on the graph with the random reset parameter `resetProbability` set to 0.01 and the maximum iterations parameter `maxIter` set to 20. Only values for vertices were maintained, with results in Table 2.6. The highest `id` was 1251538798292467712 with `pagerank` of 35K. The same `ids` were the top ones for out-degree.

```
df_page_rank = (g_cache
    .pageRank(resetProbability=0.01, maxIter=20)
    .vertices)
```

Table 2.6: Page rank top 5 for reference graph.

	id	pagerank
1	1251538798292467712	35175.99937736224
2	1268613313702891523	15907.213275846976
3	1285207186591887360	13229.11068847198
4	1234320094018228228	11117.542217267122
5	1245547129831067649	9925.058023775598

Connected Components. Connected components experiments identified groups of vertices connected together. Using the cached and repartitioned graph, the algorithm's `broadcastThreshold` was set to 5M and the algorithm was set to `graphx` mode, the combination of which reduced processing time to 2.5 minutes with 100% accuracy for identifying all 235K chains. Also, while no orphans were expected within the reference graph, `dropIsolatedVertices()` ensured they were not included in the processing. Table 2.7 gives unordered results with fields `id`, `id_data`, and `component`. The `component` field was set to the lowest `id` within each detected component. For example in row 1 the `component` and `id` were both 635, meaning that `id` would become the component identifier. This has utility with tweet trees and chains as the lowest numbered tweet `id` of a given component would also be the root. Experiments with LPA (§1.1, not shown) for clustering through message passing did not fully converge and did not have the same properties of lowest identifier becoming the component, so the methodology was standardized to use connected components.

```
df_cc = (g_cache
    .dropIsolatedVertices()
    .connectedComponents(algorithm='graphx',broadcastThreshold=5000000))
```

Table 2.7: Connected components output for reference graph.

	id	id_data	component
1	635	false	635
2	9387802947	false	9387802947
3	12191663170	false	12191663170
4	20395980281	false	20395980281
5	27000723399	false	27000723399

From the components with top 5 membership listed in Table 2.8, two very large components `756710850694639617` with 482K members and `1238615797028679681` with 68K members dominate the top 2 rankings.

Table 2.8: Connected components top 5 counts for reference graph.

	component	count
1	756710850694639617	482,786
2	1238615797028679681	67,961
3	1185575289385959425	25,730
4	1231737928600244231	20,489
5	1087479339782877184	13,574

Combined Graph Measures. The graph measures for degree, page rank, and connected component were joined together and ranked through Spark `Window rank()`. The resulting fields were as follows:

```
id (long), component (long),  outDegree (int),  inDegree (int),  id_data (bool),
    pagerank (double), and id_component_rank (int).
```

```
df_combine = (
    df_cc.drop("id_data")
        .join(df_out_degrees, ["id"], "left")
        .join(df_in_degrees, ["id"], "left")
        .join(df_page_rank, ["id"], "left")
    .withColumn(
        "id_component_rank",
        F.rank().over(Window.partitionBy("component").orderBy(F.desc("pagerank")))
```

```
    )
)
```

Additionally, field `component_id1` was added having a values set to the tweet `id` with the highest page rank per component. New field `pagerank_id1` was set to the same `id`. With these additions, the original field `component` was left unchanged. Table 2.9 give the top 5 counts for the combined graph measures when grouped by `component`, `component_id1`, and `pagerank_id1`. As anticipated from earlier outputs, see a massive spike for component 756710850694639617 having page rank 35K.

```
df_all = (
    df_combine
        .join(
            df_component_rank.filter("id_component_rank = 1").selectExpr("
    component", "id as component_id1", "pagerank as pagerank_id1"),
            ["component"],
            "left"
        )
)
```

Table 2.9: Combined measures top 5 counts for reference graph.

	component	**component_id1**	**pagerank_id1**	**count**
1	756710850694639617	1251538798292467712	35175.99937736224	482,786
2	1238615797028679681	1285207186591887360	13229.11068847198	67,961
3	1185575289385959425	1234320094018228228	11117.542217267122	25,730
4	1231737928600244231	1245547129831067649	9925.058023775598	20,489
5	1087479339782877184	1090328713814831105	6863.841785902526	13,574

2.4. Graph Modeling: Social Media Trees and Chains

Trees encompass the combination of Social Media (SM) posts and the posts they reference. Trees form from an absolute root, the originating post to which all other posts in a tree ultimately can be traced as descendants. This section discusses the development of formalized graph techniques to model social data chains. Chains are composed of enumerated paths from nodes in a tree to an identified target node

through intermediary nodes. Chains become a measure required for SME and SMN algorithms discussed in the next section (§2.5) and also become the primary measure of interest for incremental scoring (§2.5.8).

An originating post not referenced by any other post becomes an orphan. A post which references no other post and has a reference by only one other post is the most simplified form of a tree, having a single chain length of 1. Comparable to a tree in nature having the potential for many branches, a single tree root has the potential for many chains, formed from within graph structures through various combinations of widening (reposts) and deepening (reply) engagements. Figure 2.5 plots the tweet tree for one of the more engaged roots. The 75 vertices were ordered by tweet date, which then makes vertex 0 the tweet root. There were 143 edges in this tree.

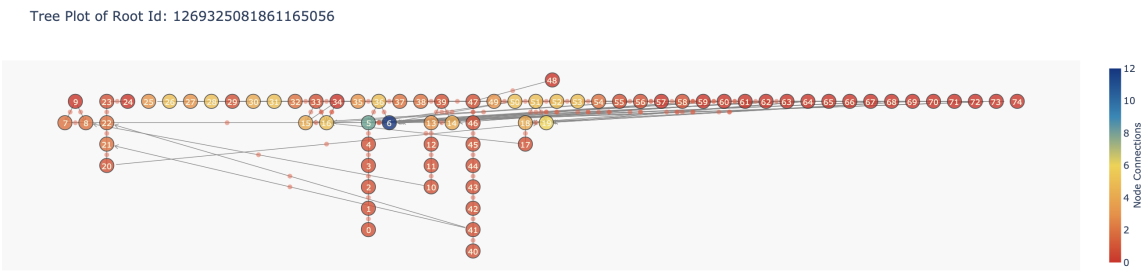

Figure 2.5: Example tweet tree.

From Table 2.10 the first 8 edges were all deepening engagements with relationship `tweet_in_reply_to_status_id`. Then edge 09 $v_9 \rightarrow v_8$ with relationship `retweet_id` becomes the first widening engagement. Edge 10 $v_9 \rightarrow v_7$ provides the additional information that the retweet of v_8 was replying to v_7. Edge 10 could be left out of the graph if adhering to a tree structure schema only linking children to their immediate parents.

Table 2.10: First 10 edges for example tweet tree.

	src	dst	src_id	dst_id	relation
01	1	0	1269328504908156928	1269325081861165056	tweet_in_reply_...
02	2	1	1269330167966560258	1269328504908156928	tweet_in_reply_...
03	3	2	1269333839131717632	1269330167966560258	tweet_in_reply_...
04	4	3	1269335308090277888	1269333839131717632	tweet_in_reply_...
05	5	4	1269338104948682752	1269335308090277888	tweet_in_reply_...
06	6	5	1269340826678652928	1269338104948682752	tweet_in_reply_...
07	7	6	1269343443311947776	1269340826678652928	tweet_in_reply_...
08	8	7	1269345168374652929	1269343443311947776	tweet_in_reply_...
09	9	8	1269345179040808960	1269345168374652929	retweet_id
10	9	7	1269345179040808960	1269343443311947776	retweet_in_reply_...

Each tweet was from a user account. A graph of unique users can be produced by the transformation of the tweet tree as in Figure 2.6 where the relationships among the resulting 6 user vertices yield 7 unique edges.

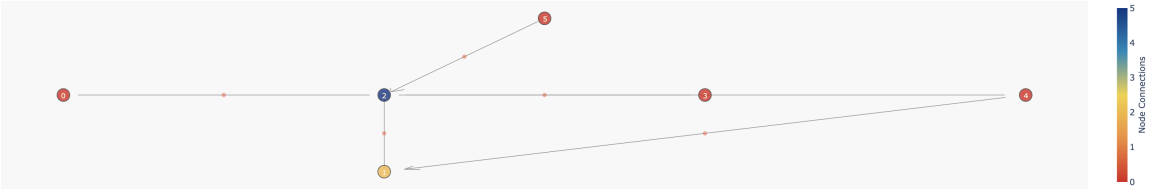

Figure 2.6: Example tree transformed into user graph.

2.4.1 Calculate Roots

Roots may be relative or absolute, depending on data completeness. An absolute root can only be found in `dst` field, never in `src` field of edges. The roots can be distinguished in Listing 2.7 by subtracting out all `src` values found in `dst` field, with both field names being transformed to `id`. Then field `id_data`, transformed from `dst_data`, was maintained in the derived DataFrame. When `id_data = True` the root becomes absolute since it was actually present in the data; conversely, for

`False` the root becomes relative since the true, or absolute, root exists outside the available data. In generating chains and roots, `relationship` field in graph edges described in §2.3.5 was dropped to eliminate the duplication of edge and vertex rows. If `relationship` was not dropped, the calculated size of chains such as in §2.4.2 would be artificially inflated for the 23% of reference tweets which include more than one reference tweet target as in §2.3.2.

Listing 2.7: Tweet roots from reference edges.

```
df_roots = (
  g_cache.edges
    .selectExpr("dst as id", "dst_data as id_data")
    .subtract(
      g_cache.edges
        .selectExpr("src as id")
        .withColumn("id_data", F.lit(True)) # src is always in the data
    )
    .distinct()
)
```

The data used in this research contains 1.4M roots within the original 8.6M unique DOI tweets. Figure 2.7 plots the split between true roots, comprising 46% of the 1.4M roots, and relative roots, comprising 54%.

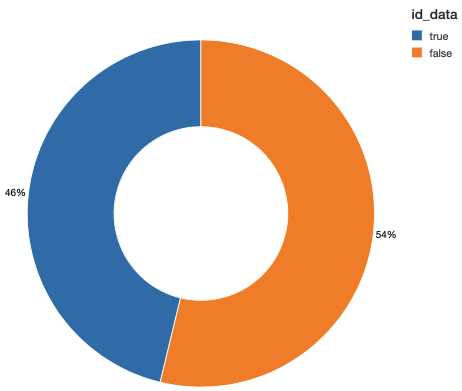

Figure 2.7: Absolute and relative tweet roots.

2.4.2 Derive Chains from Reference Tweets

The reference tweet graph generated in §2.3.5 was passed to Listing 2.8 in a loop that increased the value of `chain_len`, starting at 2, until the processing yielded 0 results. The terminating condition was reached at `chain_length = 30`, meaning `chain_length = 29` was the max chain length for the available CORD-19 tweet data used in the research.

Listing 2.8: Function to find all chains of a specified length.

```
def gen_chain(g: GraphFrame, chain_len: int, root_id_col: str = "root_id",
    doi_col:str = "tweet_doi", add_doi_col:str = True, debug_mode:bool = False) ->
     DataFrame:
    """
    Ignoring all but the last edge.
    - chain_len 2 requires 3 letters
    """
    vs = gen_chain_vs(chain_len, debug_mode=debug_mode)
    v_root = vs[chain_len]
    motifs = []
    # chain_len vs chain_len + 1
    for i in range(chain_len):
        c = vs[i]
        c_next = vs[i+1]
        motifs.append(f"({c})-[e{i}]->({c_next})")
    motif_str = "; ".join(motifs)
    df_result = g.find(motif_str)
    cols = []
    for i,c in enumerate(vs):
        v_id = f"{c}_id"
        df_result = df_result.withColumn(v_id, col(f"`{c}`.id"))
        cols.append(v_id)
        if i < chain_len:
            e_rel = f"e{i}_rel"
            df_result = df_result.withColumn(e_rel, col(f"`e{i}`.relation"))
            cols.append(e_rel)
    if add_doi_col:
        df_result = (
            df_result
                .select(
                    f"e0.{doi_col}",
                    *cols
                )
        )
    else:
        df_result = df_result.select(*cols)
```

```
return (
    df_result
        .withColumnRenamed(f"{v_root}_id", root_id_col)
        .distinct()
)
```

A root may be involved in many chains such as in Figure 2.8. This was the same tree as Figure 2.5 but now additionally annotating chains within the tree. There were multiple chains terminating at the absolute root v_0, starting with initial chain $v_2 \rightarrow v_1 \rightarrow v_0$ then chain $v_3 \rightarrow v_2 \rightarrow v_1 \rightarrow v_0$ and so on until all unique chains have been identified, for this tree the max chain was length 29. In addition to calculating the chains involving the absolute root, relative roots can also be calculated.The earliest relative root was v_1 with initial chain $v_3 \rightarrow v_2 \rightarrow v_1$ then chain $v_4 \rightarrow v_3 \rightarrow v_2 \rightarrow v_1$, following the same pattern until all unique chains have been identified.

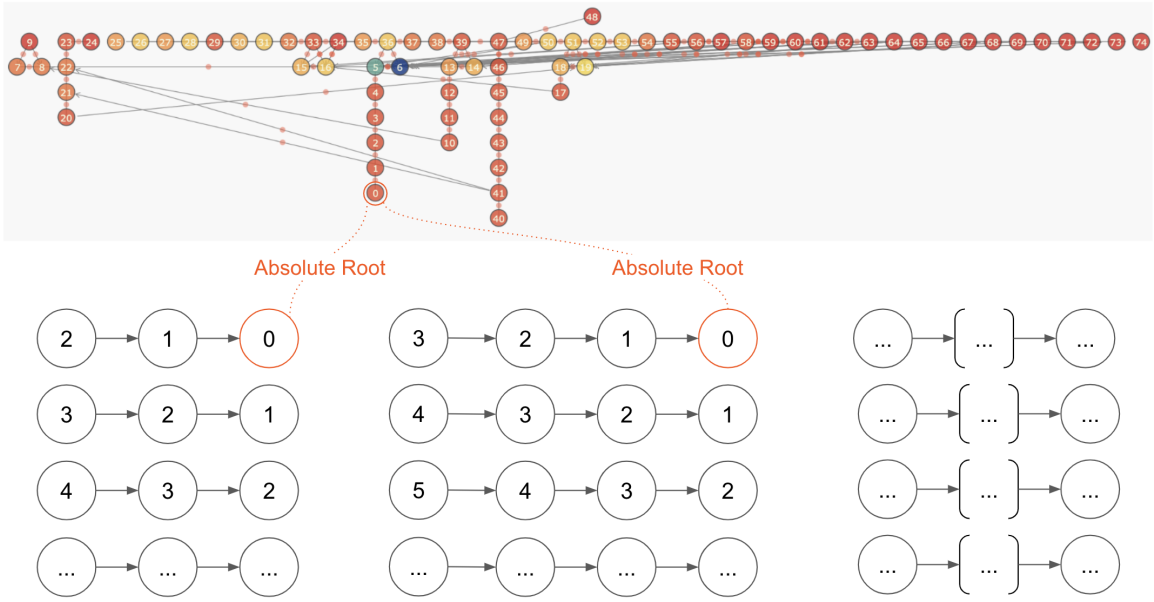

Figure 2.8: Absolute and relative chains from tweet graph.

Example output for chain length 2+ in Table 2.11. The nomenclature will be 2+ because a chain must be at least length 2 to be also detected as length 3, and a chain must be at least length 3 to be detected as length 4 (and so on). There were

2.2M chains of 2+ length and 259K roots. The call to function `gen_chain_vs(...)` in Listing 2.8 generates the GraphFrames motif values for the provided chain length using alpha characters of indefinite size. For chain length 2, the motif became `g.find("(a)-[e0]->(b); (b)->[e1]->(c)")` which effectively would find all 2 length chains. In the case of chain length 2+ the alpha vertex fields from motif finding were `a_id` and `b_id` in addition to `root_id` and the relationship fields. Chain length 26 used `a-z` and chain length 29 additionally used `aa-cc`.

Table 2.11: Chain length 2 processing results.

	a_id	**e0_rel**	**b_id**	**e1_rel**	**root_id**
01	...7745	...	162204041772924929	...	162203374383665153
02	...9280	...	258569114044477440	...	258565518250893312
03	...6944	...	281212060984041472	...	281211072034263040
04	...0608	...	314293156638244864	...	314292065141919744
05	...6560	...	340062744336269312	...	340061507477962752

From 2.2M chains of length 2+ in Table 2.12, find only 211K chains of length 3+, 45K chains of length 4+, and 22K chains of 5+. This decrease generally continues to a single chain of length 29. Chain sizes 10 to 15 differ from the general linear decrease due to additionally discovered vertices by motif finding after chain size 9 due to widening followed by deepening operations within the trees. Of the 259K roots 91%, or 237K, have a maximum chain length of 2 and 7% have a maximum chain length of 3, leaving only 2% for all the other lengths.

Table 2.12: Summary of reference tweet chain processing.

Chain Depth	Chains	Roots	Chain Depth	Chains	Roots
02+	2,257,970	259,045	17+	2,175	55
03+	211,235	22,040	18+	958	45
04+	45,196	3,683	19+	458	38
05+	22,179	1,544	20+	230	30
06+	15,557	858	21+	135	25
07+	12,161	551	22+	101	21
08+	9,274	388	23+	82	18
09+	8,012	285	24+	69	15
10+	8,984	209	25+	52	12
11+	11,574	162	26+	30	9
12+	14,610	134	27+	15	6
13+	15,969	115	28+	5	3
14+	14,007	98	29+	1	1
15+	9,416	83	30+	0	0
16+	4,911	67			

Experiments to add fields for widening (repost) and deepening (reply) operations were conducted during the research. Those fields ended up introducing duplication of chain data, so instead chain logic was focused on the transformations in Listing 2.9 to the chains DataFrame which added chain vertices in field `chain_nodes_from_root` and size of chain in field `num_chains`.

Listing 2.9: Transformation of tweet chains to users chains.

```
df_chains_annotated_delta = (
    df_chains
        .withColumn(
            "chain_nodes_from_root",
            F.array([col(f"{v}_id") for v in vs[:-1]][::-1]) # reversed
        )
        .withColumn("num_chains", F.size("chain_link_types_from_root"))
)
```

2.4.3 Add User Information to Tweet Chains

Due to some roots missing in the available data, find there were 78K null root users. The DataFrame in Listing 2.10 joined user information on the existing tweet chain as `root_user_id` and `chain_node_user_id`.

Listing 2.10: Transformation of tweet chains to users chains.

```
df_user_chains_explode = (
    df_chains_explode
        .join(df_tweet_user_lookup.selectExpr("id as root_id", "user_id as
    root_user_id"),["root_id"],"left")
        .join(df_tweet_user_lookup.selectExpr("id as chain_node", "user_id as
    chain_node_user"),["chain_node"],"left")
        .join(df_all_aug.selectExpr("id as chain_node", "id_data as
    chain_node_id_data"), ["chain_node"], "left") # id_data
        .join(df_all_aug.selectExpr("id as root_id", "id_data as root_id_data"), [
    "root_id"], "left") # root id_data
)
```

Additionally field `chain_hash` was derived from the new Spark 3.0 hash function `F.xxhash64("root_id", "chain_nodes_from_root"))` which generates a 64-bit hash code of `root_id` and `chain_nodes_from_root` and returns the result as a long column. The final schema for the tweet chains DataFrame stores the following field information for each full chain:

```
root_id (long), chain_size (int), chain_hash (long), root_user_id (long),
    root_id_data (bool), all_chain_nodes (array [long]), all_chain_node_users (
    array [long]), all_distinct_chain_node_users (array [long]),
    all_num_distinct_chain_node_users (int).
```

2.4.4 Broken Social Media Chains

Broken chains may be the result of posts deleted either by the user or removed from the SM site. Also, broken chains may be posts that were only missing in the available data as a result of incomplete collection or filter criteria which produced a subset of data. An example of a broken chain follows:

i. `https://twitter.com/JAMA_current/status/971143392041340928` can be identified as the true root of the tree (14 retweets, 6 quoted tweets, and 23 likes), meaning it has no references but has been referenced by other tweets.

ii. `https://twitter.com/schin_au/status/1095795357622173696` was ahead of the next (2 retweets, 1 quoted retweet, 1 like). It was also referenced by some deleted tweets.

iii. `https://twitter.com/ShaunaRadOnc/status/1173688604993957888` was identified as a retweet of the above (3 retweets, 5 likes).

Of the tweets listed above, the reference tweet DataFrame only contained the tweet from (iii) in the available data. Therefore the chain categorically becomes a broken chain since the tree's true root (i) as well as (ii) were missing in the available data. However, the tree's chains can still be be processed by maintaining the broken references but only scoring the tweets actually present. In this way, the first available tweet at (iii) becomes the base relative root of the tree.

2.4.5 All Relative Chain Paths

In addition to the DataFrame storing all roots, in order to calculate chain measures as defined in §2.4.2 there were benefits to identifying all relative paths within a chain from the view of each vertex along a path. The intuition being that while each chain has an absolute root, the actual vertices which extend from a root would be both temporally after the presence of the root and also would contribute to their own engagement through subsequent reposts and replies. In this manner each vertex in a chain should be treated as its own relative root in order to isolate temporal as well as per user engagement, even down to the week as in §2.5.8. To accomplish

deriving all relative chain paths, several operations were called, starting with Spark `posexplode(chain_nodes_col)` in Listing 2.11.

Listing 2.11: Explode tweet chains.

```
def explode_chains(df, root_id_col="root_id",
    chain_nodes_col="chain_nodes_from_root", chain_hash_col="chain_hash",
    new_rel_chain_size_col="rel_chain_size", new_chain_size_col="chain_size",
    new_chain_node_col="chain_node", new_chain_pos_col="pos_from_root"):
    """
    Handle exploding out chains.
    """
    return (
        df
            .select(root_id_col, chain_nodes_col, chain_hash_col)
                .distinct()
                .withColumn(new_chain_size_col, F.size(chain_nodes_col))
            .select("*", F.posexplode(chain_nodes_col))
                .withColumnRenamed("col", new_chain_node_col)
                .withColumnRenamed("pos", new_chain_pos_col)
                .withColumn(new_rel_chain_size_col, col(new_chain_size_col) - col(
    new_chain_pos_col) - 1)
            .select(root_id_col, new_chain_node_col, new_chain_pos_col,
    new_chain_size_col, new_rel_chain_size_col, chain_hash_col)
            .orderBy(chain_hash_col, new_chain_pos_col)
    )
```

After exploding the chains, Spark `Window` operations `collect_list()` and `collect_set()` were called to derive the relative chains in Listing 2.12. This resulted in 6.5M rows of relative chains, of which 287K chains were greater than chain length 2. The goal was to have all relative paths from a per tweet perspective, regardless of whether tweet was a root or not. This technique has importance for per user scoring (§2.5) as well as additional importance for temporal incremental scoring introduced in §2.5.8.

Listing 2.12: Derive relative user chains.

```
def augment_users_explode(df_user_explode: DataFrame, is_rel: bool,
    chain_hash_col="chain_hash", pos_from_root_col="pos_from_root",
    chain_node_col="chain_node", chain_node_user_col="chain_node_user",
    root_id_col="root_id", root_user_id_col="root_user_id"):
    """
    Add columns related to users to an exploded DataFrame.
    - This may be relative or include root information
```

```
"""
chain_window = None
if is_rel:
    chain_window = (
        Window.partitionBy(chain_hash_col).orderBy(pos_from_root_col)
            .rowsBetween(Window.currentRow + 1, Window.unboundedFollowing)
    )
else:
    chain_window = (
        Window.partitionBy(chain_hash_col).orderBy(pos_from_root_col)
            .rowsBetween(Window.currentRow, Window.unboundedFollowing)
    )
dict_cols = None
start_id_col = None
start_user_id_col = None
if is_rel:
    dict_cols = {
        "chain_nodes": "rel_chain_nodes",
        "chain_node_users": "rel_chain_node_users",
        "distinct_chain_node_users": "distinct_rel_chain_node_users",
        "num_distinct_chain_node_users": "num_distinct_rel_chain_node_users"
    }
    start_id_col = chain_node_col
    start_user_id_col = chain_node_user_col
else:
    dict_cols = {
        "chain_nodes": "all_chain_nodes",
        "chain_node_users": "all_chain_node_users",
        "distinct_chain_node_users": "all_distinct_chain_node_users",
        "num_distinct_chain_node_users": "all_num_distinct_chain_node_users"
    }
    start_id_col = root_id_col
    start_user_id_col = root_user_id_col
return (
    df_user_explode
        .withColumn(
            dict_cols["chain_nodes"],
            F.concat(F.array(start_id_col), F.collect_list(chain_node_col).
over(chain_window))
        )
        .withColumn(
            dict_cols["chain_node_users"],
            F.concat(F.array(start_user_id_col), F.collect_list(
chain_node_user_col).over(chain_window))
        )
        .withColumn(
            dict_cols["distinct_chain_node_users"],
            F.array_distinct(F.concat(F.array(start_user_id_col), F.
collect_set(chain_node_user_col).over(chain_window)))
        )
        .withColumn(
```

```
                dict_cols["num_distinct_chain_node_users"],
                F.size(dict_cols["distinct_chain_node_users"])
            )
        )
```

The count of relative chains of length 3 was 188K, length 4 was 145K, length 5 was 124K, length 6 was 110K, continuing in a linear decay as rendered Figure 2.9.

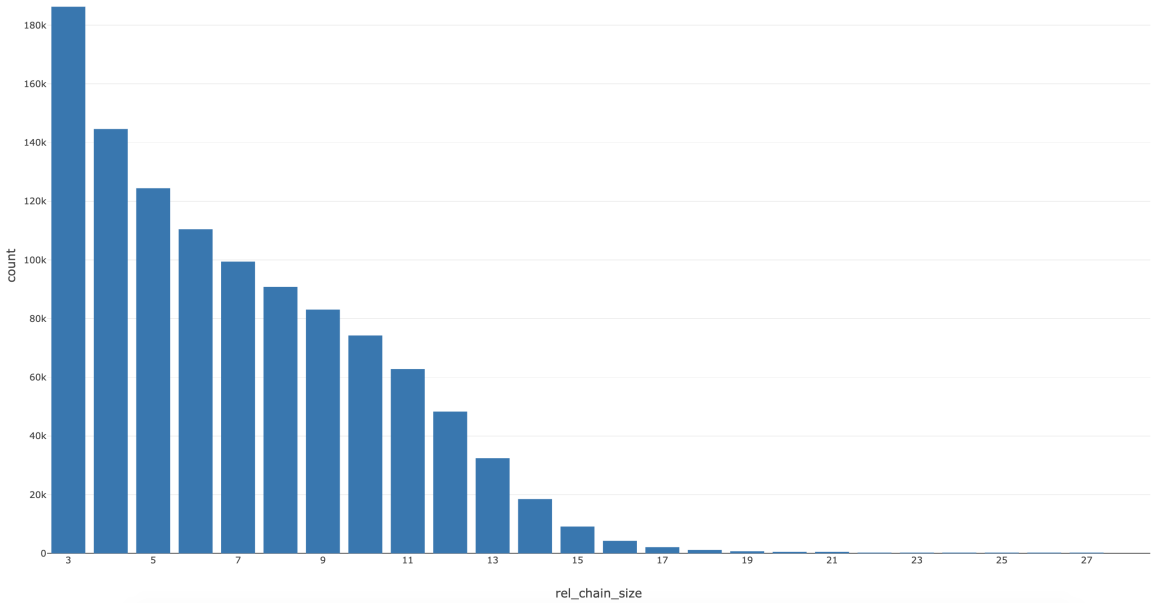

Figure 2.9: Counts for relative tweet chain lengths.

The final schema for the relative tweet chains DataFrame shares schema similarities with the schema generated for actual tweet chain roots. There was the addition of `chain_node` which becomes the relative root of the chain along with the addition of `pos_from_root` to store its position in the full chain. Then the addition of `rel_chain_size` identified the chain length from the relative node position in the full chain. Also, there was the addition of `chain_node_user` and boolean `chain_node_id_data` to identify whether the chain node was in the data. Then the addition of lists `rel_chains_nodes` and `distinct_rel_chain_node_users` along with `num_distinct_rel_chain_node_users`. The relative schema follows:

```
root_id: (long), chain_node (long), pos_from_root (int), chain_size (int),
    rel_chain_size (int), chain_hash (long), root_user_id (long), chain_node_user
    (long), chain_node_id_data (bool), root_id_data (bool), rel_chain_nodes (array
     [long]), rel_chain_node_users (array [long]), distinct_rel_chain_node_users (
    array [long]), num_distinct_rel_chain_node_users (int).
```

2.5. Distributed Scoring Algorithms: Social Media Measures

This research has identified and formalized two Social Media (SM) scoring algorithms, Social Media Engagement (SME) and Social Media Noise (SMN), which quantify complexity of engagement, or lack thereof, by a user within social data. Given that only users – person or organization accounts – communicate on social sites, the developed technique calculated all scores based on users first, then assigned all coverage and category specific scores as derived from filtering of per user scores. SME and SMN scores have wide applicability for quantifying all social data at any scale or velocity.

Supporting incremental scoring was a key goal of the research leading to establishing SM scores. The most fundamental temporal unit would be weekly where users would be scored on number of posts `n_posts` such as tweets, number of coverage items `n_coverage` such as DOIs, number of unique users reached `n_reach`, and on number of chains `n_quality_indicators` as well as the overall median chain lengths `median_quality_indicator_value`.

All of the measures can be isolated strictly to the given week with the exception of overall median chain lengths `median_quality_indicator_value` which has proven to be better calculated as a blended four week lagging score as described in §2.5.8. The calculated weekly user SM scores can be simply added together without any recalculation to generate weekly aggregate scoring to include quarterly, half year, and annual scores.

2.5.1 Twitter Nomenclature

The nomenclature for SM scores can be easily adapted: Facebook (SME_{Fuser}), Reddit (SME_{Ruser}), or an aggregate spanning multiple social sites ($SME_{FRTuser}$). User will be calculated first always within bounds of temporal and categorical filtering as applicable, then the categorical score will be calculated from specific rules from across the users scores. For Twitter SM scores will be referenced as follows:

i. SME_{Xx} for SME Twitter will reference user as SME_{Tuser} and ASCJ as SME_{Tasjc}.

ii. SMN_{Xx} for SMN Twitter will reference user as SMN_{Tuser} and ASCJ as SMN_{Tasjc}.

2.5.2 Social Media Engagement (SME) Formula and Implementation

The SME score was calculated based on posts `p`, coverage `c`, reach `r`, and quality `q`. Equation 2.1 provides a representation of the user score formula. Engagement can be quantified through the number of unique users who reply or repost, through the unique number of coverage items that were engaged such as DOIs, and through the number and length of chains generated through post engagement. The SME formula design mitigates against contrived user behaviors which result in low engagement such as tweet blasts in §2.3.4. When both engagement and the number of posts were high, the SME score would be high as well. When posts were high but engagement was low, the SME score would be lower and the SMN (noise) score would be higher. The SME formula design does not allow a high number of user posts, irregardless of engagement, to dominate the score. In this way, the SME score can be compared to page rank (§1.1) which was established to mitigate against self-boosting behaviors of websites linking out to numerous other sites, referred to as out-degree, in order to improve their search engine ranking. Page rank shifted the focus to sites linking to a target site, referred to as in-degree, as a key measure for elevating search rankings. Page rank's focus on in-degree for scoring would be a simplified form of accounting

for engagement to no longer reward self-boosting behaviors. By incorporating users, coverage, and chain quality, the SME formula design goes beyond structural connections used in page rank. Thus, relating to social data engagement the SME formula can generate more useful rankings than page rank as discussed further in Chapter 3.

Equation 2.1 Per user Social Media Engagement (SME) score formula.

$$SME_{Xuser} = \left(\ln\left(1 + |P_{Xuser}|\right) * |C_{Xuser}|\right) + \left(|R_{Xuser}| * \sqrt{1 + (\tilde{q}_{Xuser*} \mid Q_{Xuser} \mid)}\right.$$
$$\left. * \left(\ln\left(\mathbf{e} - 1 + |C_{Xuser}|\right) - \ln\left(\mathbf{e} - 1\right)\right) * \ln\left(1 + \tilde{q}_{Xuser}\right)\right) \tag{2.1}$$

i. Number of posts in set P_{Xuser} will be represented by $\left(|P_{Xuser}|\right)$. For Twitter this becomes tweets authored by a target user, optionally bounded by a corpus such as relating to CORD-19 tweets.

ii. Size of engagement coverage in set C_{Xuser} will be represented by $\left(|C_{Xuser}|\right)$. For Twitter CORD-19 corpus, this becomes unique DOIs included across all tweets by a user. Coverage can be ignored by setting $C_{Xuser} = 1$ as the equation has been designed to be unaffected by coverage of 1 which will be handled by the two $\ln(\mathbf{e} - 1)$ expressions.

iii. Size of unique users reach based on engagement in set R_{Xuser} will be represented by $\left(|R_{Xuser}|\right)$. For Twitter, this becomes the unique Twitter users engaging a target user's tweets.

iv. Engagement quality in set Q_{Xuser}, with count represented by $\left(|Q_{Xuser}|\right)$ and median of set represented by $\left(\tilde{q_{Xuser}}\right)$. For Twitter $\left(|Q_{Tuser}|\right)$ would be the

count of tweet chains engaged by the target user and $\left(q_{T\tilde{user}}\right)$ would be the median chain length of all tweet chains having the target user as a chain actual or relative root. $q_{T\tilde{user}}$ was used twice in the formula. The first use can be found in the expression $\sqrt{1 + (\tilde{q}_{Xuser*} \mid Q_{Xuser} \mid)}$ which will resolve to 1 if either expression variable equals zero. The expression $\ln\left(1 + \tilde{q}_{Xuser}\right)$ includes 1+ to ensure that the formula remains valid even for $\tilde{q}_{Xuser} = 0$.

v. SME score can be zero but not less than zero.

vi. SME scores were not normalized. There will not be an upper limit on the engagement of a user except the limits of data being scored, which may be bounded or otherwise incomplete. In the synthetic data from §2.5.5 used in establishing the methodology and designing the resulting SM formulas, the higher SME scores were above 1M.

The data used in research experiments was Twitter, so $_{Xuser}$ can be represented as $_{Tuser}$; however, the nomenclature for the scoring can be customized to a given SM source providing its social data minimally includes posts, users, and engagement chains. Listing 2.13 gives the python code implementation of the SME algorithm for a single user as applied in this research.

Listing 2.13: Python implementation of SME score algorithm.

```
def calc_sme_score(n_posts: int, n_coverage: int, n_reach: int,
    n_quality_indicators: int, median_quality_indicator_value: int) -> float:
    """
    Generate SME Score.
    - math.log is ln, i.e. base e, by default
    """
    return (
        (math.log(1 + n_posts) * n_coverage)
        + (
            n_reach
            * math.sqrt(
                1 + (median_quality_indicator_value * n_quality_indicators)
            )
```

```
            * (math.log(math.e - 1 + n_coverage) - math.log(math.e - 1))
            * math.log(1 + median_quality_indicator_value)
        )
    )
```

2.5.3 Social Media Noise (SMN) Formula and Implementation

SMN scores were calculated based on the total posts `p` using the same post value applied to the SME formula in Equation 2.1. For scores where $SMN_{Xx} \approxeq 1$ the assessed target can be considered not noisy. As the SMN score increases from 1, the more noisy the target. Just like SME scores, SMN scores will be generated per user.

Equation 2.2 Per user Social Media Noise (SMN) score formula.

$$SMN_{Xuser} = \frac{total\_posts_{Xuser}}{\left(1 + SME_{Xuser}\right)} \tag{2.2}$$

i. $_{X}user$ will be the same user and measures scored as SME_{Xuser}.

ii. The same number of posts used in the SME score for the user, represented by $total\_posts_{Xuser}$.

iii. To avoid a potential divide by zero error, the denominator adds one to the SME score in the expression $\left(1 + SME_{Xuser}\right)$.

iv. SMN score can be zero but not less than zero.

Python code implementation of the SMN algorithm for a single user as applied in this research follows:

Listing 2.14: Python implementation of SMN score algorithm.

```
def calc_smn_score(n_posts: int, sme_score: float) -> float:
```

```
    """
    Generate SMN Score.
    """
    return (n_posts / (1 + sme_score))
```

2.5.4 Bounded SME and SMN Scoring

Scoring $_{Xuser}$ may be limited to measures for a user within any combination of imposed constraints. Constraints may be temporal, categorical, or filtered by one or more conditions that must be met within the data. PlumX provided Twitter data for this research, filtered to tweets containing DOIs from the 1-hop CORD-19 corpus (§2.1) prior to delivery for the research. The Twitter data provided ended at week 36 (end of August to early September) of 2020 and was based on DOIs in the CORD-19 corpus up to early July, 2020. Any references to *unbounded* in the experiments were still constrained to the data available. Given the above caveats to the use of the term *unbounded*, this research evaluated both unbounded and bounded temporal scoring per Twitter user, with the temporally bounded experiments constrained to the first 26 weeks of 2020. Also, categorical constraints focused on per user ASJC engagement and noise scoring, meaning users were scored based on each ASJC category related to their tweets. Scopus tables provided the ASJC categories for each DOI.

SME and SMN scores can be adapted to meet the needs of a researcher within the limits of the available data. Given their flexibility, any formalized scoring as an altmetric would require clear definitions of how the data was bounded and data completeness for the given SM source.

2.5.5 Synthetic Data

Synthetic data was initially used to validate the final SM formulas. The first variation discussed below applied a mix of fixed and ascending measures. The sec-

ond variation applied a mix of fixed, ascending and descending measures. The final variation generated and applied a random mix of measures.

i. Variable `d100_asc` stored generated values in ascending order 0 to 100.

ii. Variable `d100_desc` stored generated values in descending order 100 to 0.

iii. Variable `d100_1` stored generated value *1* repeated 100 times.

iv. Variable `d100_5` stored generated value *5* repeated 100 times.

v. Variable `d100_10` stored generated value *10* repeated 100 times.

vi. Variable `d100_100` stored generated value *100* repeated 100 times.

Synthetic Ascending Data. The first variation applied ascending values from `d100_asc` for `n_reach` measure, with all other measures applying fixed values.

Listing 2.15: Ascending synthetic data generation.

```
df_sme = sme_measures_to_df(
    d100_100, # posts
    d100_1,   # coverage
    d100_asc, # reaches <-- going up
    d100_10,  # quality_indicators
    d100_10   # median_quality_indicator_values
)
```

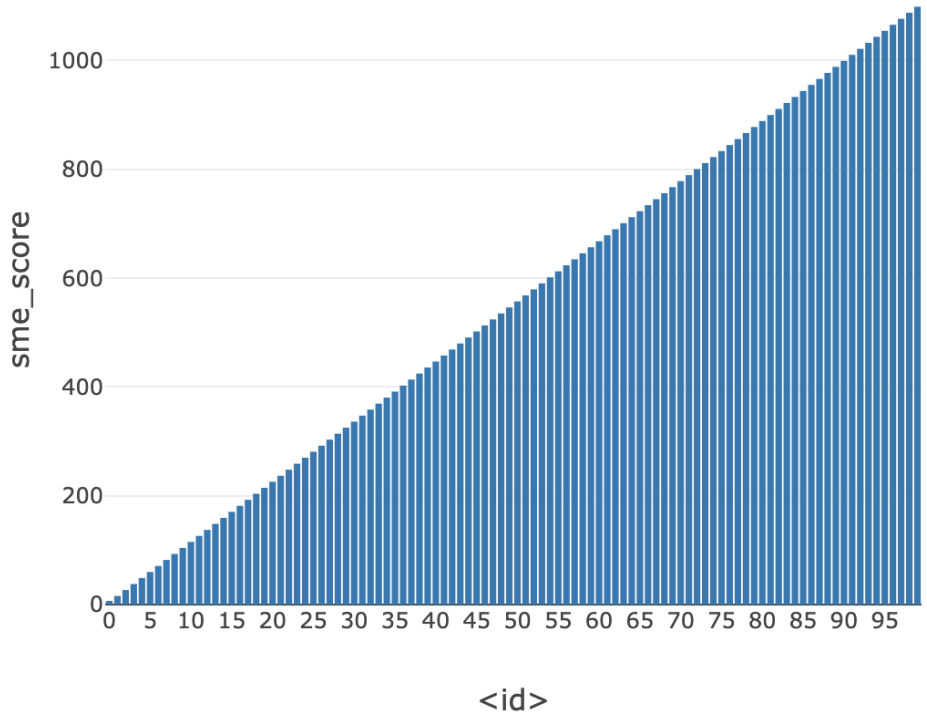

Figure 2.10: Synthetic SME ascending measure scores.

The resulting SME score likewise ascended linearly, growing from 0 to over 1,000.

Synthetic Inverse Data. In the inverse synthetic variation used to validate the final SM formulas as part of establishing the methodology, `n_reach` ascended according to `d100_asc` as in the previous ascending variation, while `n_quality_indicators` descended according to `d100_desc` instead of remaining fixed.

Listing 2.16: Descending and ascending synthetic data generation.

```
df_sme = sme_measures_to_df(
    d100_100, # posts
    d100_10,   # coverage
    d100_asc,  # reaches <-- going up
    d100_desc, # quality_indicators <-- going down
    d100_10    # median_quality_indicator_values
)
```

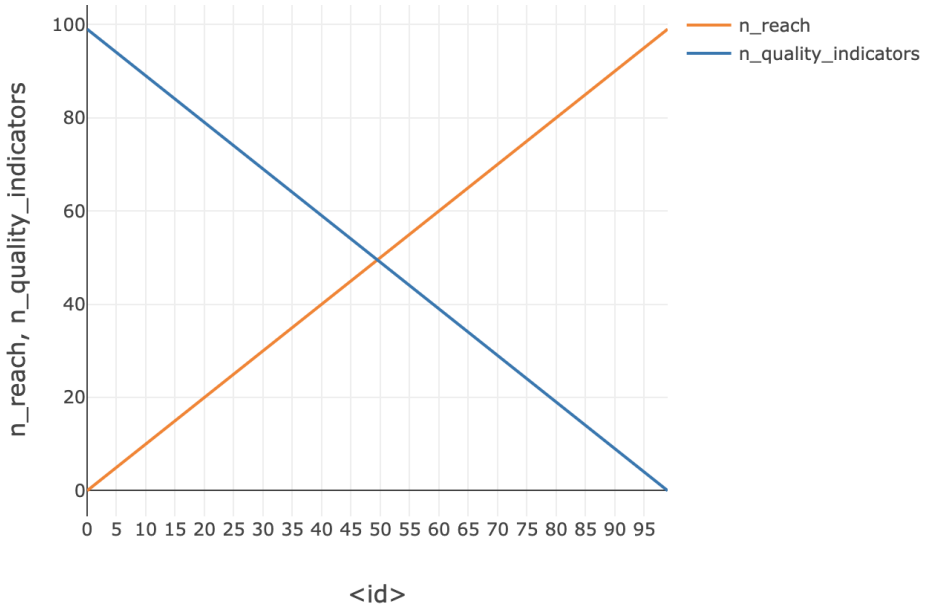

Figure 2.11: Synthetic SME inverse measure line chart.

Measure `n_reach` increased up from 0 to 100 as `n_quality_indicators` decreased from 100 to 0. The two measures crossed over at 50. SME score climbed in Figure 2.12 as `n_reach` went up even as `n_quality_indicators` went down until a cross-over point was reached where even as `n_reach` continued to go up, the SME score declined as the formula began to fully penalize for decreases in quality indicators.

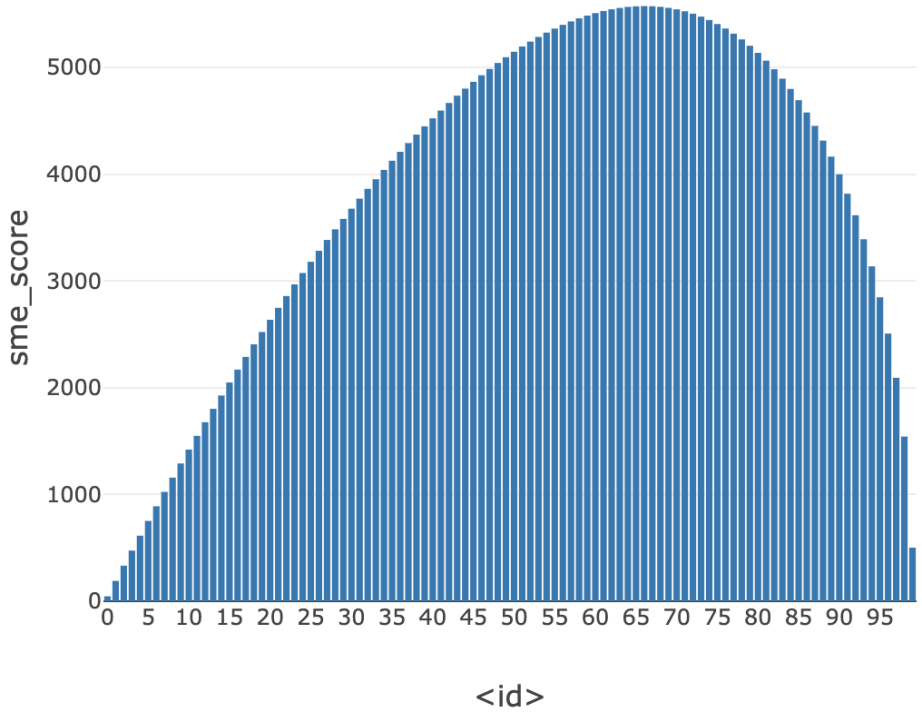

Figure 2.12: Synthetic SME inverse measure scores.

Synthetic Random Data. The random data generated produces 1K rows, with `n_post`, `n_reach`, and `n_quality_indicators` randomized but capped to 1K as well, `n_coverage` randomized but capped to 100, and `median_quality_indicator_value` randomized but capped to 25. This synthetic data does not represent the sparseness found in most actual social media data, so it will be anticipated to have a higher density of SME scores, compared to real world data which has an exponential decay with most SME scores quite low, followed by a long tail of increasingly higher scores.

Listing 2.17: Random synthetic data generation.

```
df_sme_random = sme_measures_to_df(
    create_data(
        1000 * 1000, cap=1000, sample= 1000, sort=False
        ).tolist(), # posts (random within 1K)
    create_data(
        1000 * 1000, cap=100,  sample= 1000, sort=False
        ).tolist(), # coverage (random within 100)
    create_data(
```

```
        1000 * 1000, cap=1000, sample= 1000, sort=False
        ).tolist(), # reaches (random within 1K)
    create_data(
        1000 * 1000, cap=1000, sample= 1000, sort=False
        ).tolist(), # quality_indicators (random within 1K)
    create_data(
        1000 * 1000, cap=25,   sample= 1000, sort=False
        ).tolist()  # median_quality_indicator_values (random within 25)
)
```

The random score experiment confirmed the design of the SME formula as shown in Figure 2.13 ordered by SME score ascending. In a similar manner as page rank, `n_posts` (in green) do not overly influence the engagement score as they were sometimes higher and sometimes lower throughout the bucketed SME scores, shown on the x-axis, which range from 0 to 1,700 at a multiple of 1,000 per bucket, yielding actual score ranging up to 1.7M. This was explored further in results for the real Twitter data in Chapter 3.

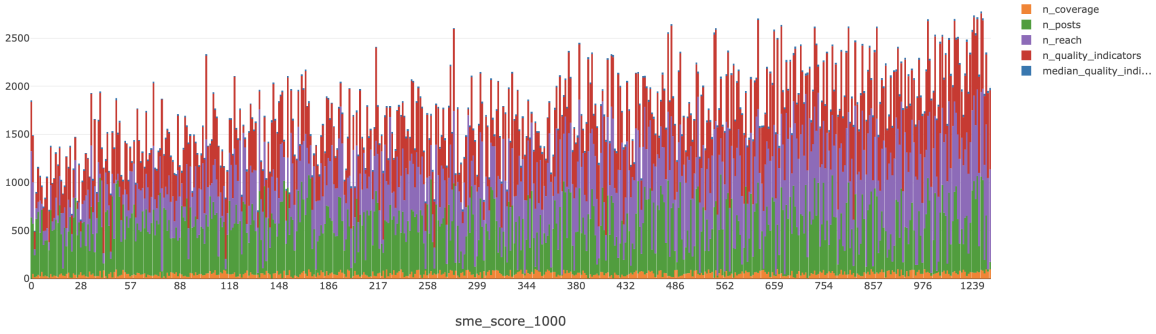

Figure 2.13: Synthetic SME random measure scores.

Given the relationship between `sme_score` and `smn_score`, $SMN \geq 1$ becomes the approximate threshold for identifying more noisy targets. A meaningful threshold will offer some separation between SMN and SME as shown in Figure 2.14 which plots only those rows in the random synthetic data with $SMN \geq 1$. The resulting 40 rows out of 1M, ordered by SMN descending, can be identified as outliers. The rows on the left return the very highest SMN scores, peaking at 907, with gen-

erally lower SME scores, meaning they were the noisiest. The further right, the less extreme the difference, with the rightmost having a SME score of 558 and an SMN score just above 1.0, making that value the least noisy of the outliers. The threshold for filtering SMN scores could be adjusted above 1.0 for data more stratified than the synthetic.

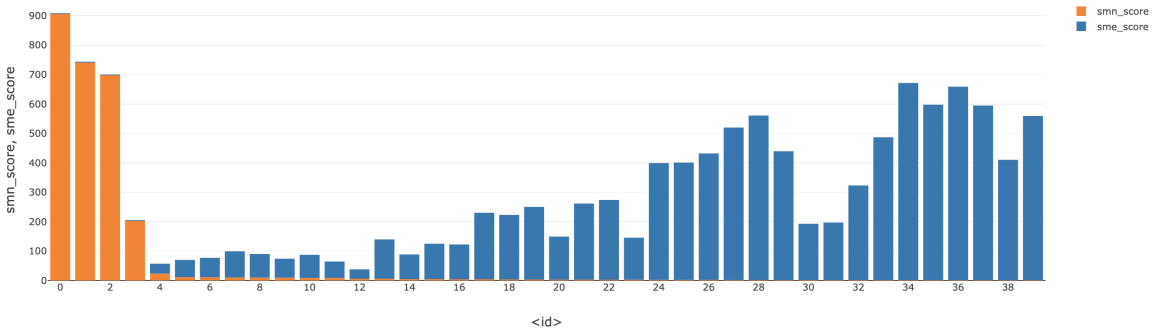

Figure 2.14: Synthetic SMN and SME bar plot.

Figure 2.15 gives a scatterplot of SME and SMN scores as well as measures `n_post`, `n_reach`, and `n_quality_indicators`. Randomness of the synthetic data comes through with the measures as values were distributed fairly evenly with no apparent correlation among them. Plotting the SME scores against each measure shows a consistent distribution which fills approximately half the area. Plotting the SME scores against SMN scores easily identifies both outliers as well as the inverse relationship such that higher SME scores yield lower SMN scores.

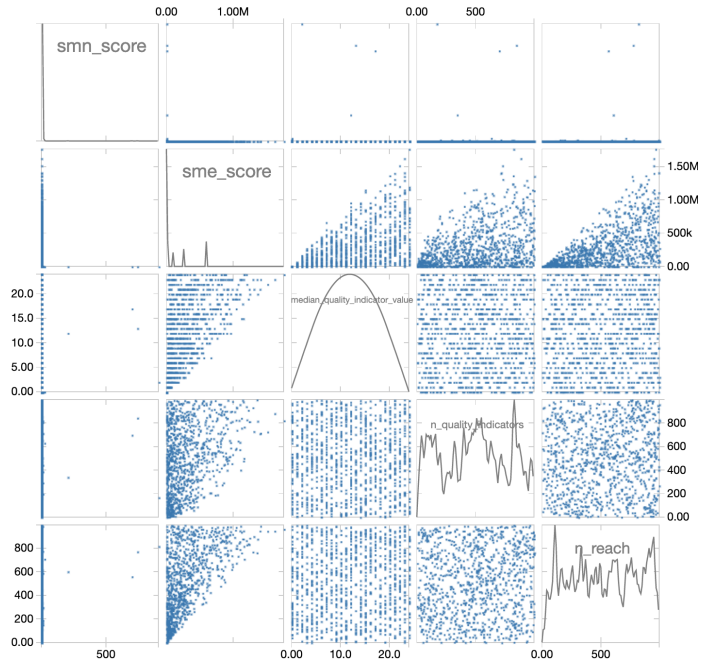

Figure 2.15: Scatterplot of random synthetic SME, SMN, and unfixed measures.

2.5.6 Per Category SME and SMN Scoring

Where available categorical restriction can be applied to SM scoring. This research experimented with ASJC categorical restrictions in §3.4.2, represented by ${}_{Xasjc}\sigma$. The overall scoring for a category becomes the sample standard deviation of user scores within that category. This was accomplished by first calculating each ${}_{Xuser}$ SME score bounded by each ASJC engagement. Then within each ASJC, take the sample standard deviation over ${}_{Xuser}$ which effectively becomes a category's score, discussed further in Chapter 3.

Equation 2.3 Per ASJC Social Media Engagement (SME) score formula.

$$SME_{Xasjc}\sigma = \left(\sqrt{\frac{1}{N_{|XuserSME-asjc|} - 1} \sum_{i=1}^{N_{|XuserSME-asjc|}} (x_i - \bar{x})^2} \right) \tag{2.3}$$

ASJC, or any categorical restriction, for SMN would be scored similarly to SME by calculating each $_{Xuser}$ SMN score bounded by each ASJC engagement. The sample standard deviation over $_{Xuser}$ within each ASJC effectively becomes a category's score.

Equation 2.4 Per ASJC Social Media Noise (SMN) score formula.

$$SMN_{Xasjc}\sigma = \left(\sqrt{\frac{1}{N_{|XuserSMN-asjc|} - 1} \sum_{i=1}^{N_{|XuserSMN-asjc|}} (x_i - \bar{x})^2} \right) \tag{2.4}$$

2.5.7 Temporal Filters for SME and SMN Scoring

Scoring can be temporally bound. This research experimented with weekly SM scores for Twitter CORD-19 data over the first 26 weeks of 2020 in §3.3.3 and §3.4.3.

i. Weekly will be any or all of weeks 1-52 in the year. Week 53, if available, will not be used as it will overlap into the following year. For 2020 week 53 starts on December 28, 2020 and ends on January 03, 2021. Similarly, week 1 of a year might include dates overlapping from the prior year. For 2020 week 1 starts on December 30, 2019 and ends on January 05, 2020. Experiments for this research with temporal boundaries have been standardized to weekly.

ii. Quarter-weeks were broken into exactly 13 weeks, resulting in $Q1_w$, $Q2_w$, $Q3_w$, and $Q4_w$. This definition for quarter-weeks differs from the calendar year definition of quarterly as being at the exact start and end day of every three months; therefore calendar quarter will not be standardized to week boundaries.

iii. Semi-annual-weeks were broken into exactly 26 weeks, resulting in $H1_w$ and $H2_w$. This definition for semi-annual-weeks will differ from the calendar year definition of semi-annual in the same manner as quarter-weeks differ.

iv. The temporal nomenclature might be represented with the year included, such as $Q2_{2020w}$ to indicate the second quarter-week of 2020. The year annotation was left off the formal definitions given that temporal restrictions would be optional.

v. Month-weeks were not included due to standardization to weekly versus exact start and end days per calendar month. If it were used, there would be 13 month-weeks in a year, instead of 12, assuming each month-week were standardized to 4 weeks. This could be partially mitigated by allowing overlapping weeks at the month boundaries to include all days of a month. For example, January 2020 includes weeks 1 to 5 with both weeks 1 and 5 also having days from a surrounding month. Then February 2020 includes weeks 5 to 9 with weeks 5 and 9 having days from a surrounding month. If overlapping weeks were included, month-weeks would vary from year to year such as February in 2021 has 4 weeks instead of 5 in 2020. Overlapping month-week nomenclature *could* then be further specified as JAN_{2020w} and FEB_{2020w} ending in 2020 at DEC_{2020w}, instead of the non-overlapping nomenclatures of $M1_{2020w}$ and $M2_{2020w}$ ending with $M13_{2020w}$.

vi. The use of the word *month* should be avoided to define non-overlapping 4 week intervals within a year as practitioners would anticipate 12 month-weeks following the calendar year instead of the resulting 13. Instead of using $M1_{w2020}$ for month-week 1, a more explicit nomenclature would be $W1\text{–}4_{2020}$ for weeks 1 to 4. Thus, $W49\text{–}52_{2020}$ would be immediately understandable as compared

to $M13_w2020$.

2.5.8 Incremental SME and SMN Scores

Incremental SM scoring will be the application of aggregated weekly temporal calculations as defined in §2.5.7. Per user SM engagement remains the center of scoring as defined in §2.5.2. Incremental when combined with categorical also uses sample standard deviation as defined in §2.5.6.

Rolling Incremental SME and SMN Scores. For standard incremental SM scoring, all measures were limited to the week of occurrence. Rolling incremental SM scoring differs in the calculation of `median_quality_indicator_value` by applying an arithmetic mean over n lagging weeks to calculate instead of applying only its weekly value. This can be useful since SM chains (§2.4) will often span multiple weeks and standard incremental calculations were restricted to a single week only, thus not adequately capture the overall complexity of chains. Lagging introduces value smoothing by having an average of the current week and the previous n lagging weeks. For experiments in this research, lagging week 1 to lagging week 4 were calculated.

Table 2.13: Calculation of lagging weeks.

	Week	**Lag1**	**Lag2**	**Lag3**	**Lag4**
w1	w1	...	...	...	...
w2	w2	avg(2,1)	...	...	...
w3	w3	avg(3,2)	avg(3,2,1)	...	...
w4	w4	avg(4,3)	avg(4,3,2)	avg(4,3,2,1)	...
w5	w5	avg(5,4)	avg(5,4,3)	avg(5,4,3,2)	avg(5,4,3,2,1)
w_n	w_n	...	...	...	...

Blended Rolling Incremental SME and SMN Scores. Blended rolling incremental of lagging weekly `median_quality_indicator_value` calculations has been identified as the preferred approach for this research. Blended allows additional value smoothing over included lagging weeks. It has the benefit of increasing the weight of the current week which will be calculated in each lagging average as well as again in each blended average.

Table 2.14: Calculation of blended lagging weeks.

	Week	**Blended1**	**Blended2**	**Blended3**	**Blended4**
w1	w1	...	...	...	...
w2	w2	avg(2,$l1$)	...	...	...
w3	w3	avg(3,$l1$)	avg(3,$l2$,$l1$)	...	...
w4	w4	avg(4,$l1$)	avg(4,$l2$,$l1$)	avg(4,$l3$,$l2$,$l1$)	...
w5	w5	avg(5,$l1$)	avg(5,$l2$,$l1$)	avg(5,$l3$,$l2$,$l1$)	avg(5,$l4$,$l3$,$l2$,$l1$)
w_n	w_n	...	...	...	...

Chapter III.

Experiments and Results

3.1. CORD-19 and Scopus Information Propagation

CORD-19 Velocity. As part of a series of experiments to understand information propagation of CORD-19 scientific literature, velocity charts were generated from the total additions, removals, and published papers per week. Velocity charts are able to visually indicate spikes of activity and sustained increases and decreases over time, this case week over week. While the velocity for the first week of data or a week after no activity is effectively that series' actual value, for other weeks where non-zero data preceded, the velocity is the change from the previous week rather than the actual of the current week.

In Figure 3.1 series velocity data for `add` (blue), `remove` (green), and `publish` (gray) is plotted by week of data (§2.2). The plot reveals that weeks 13 and 21 experienced significant spikes in papers added to the CORD-19 dataset of around 50K and 70K respectively. The plot also reveals cleanup of the dataset through removals around weeks 20 and 34, both on the order of 10K. Publishing activities did not reveal a spike in this plot.

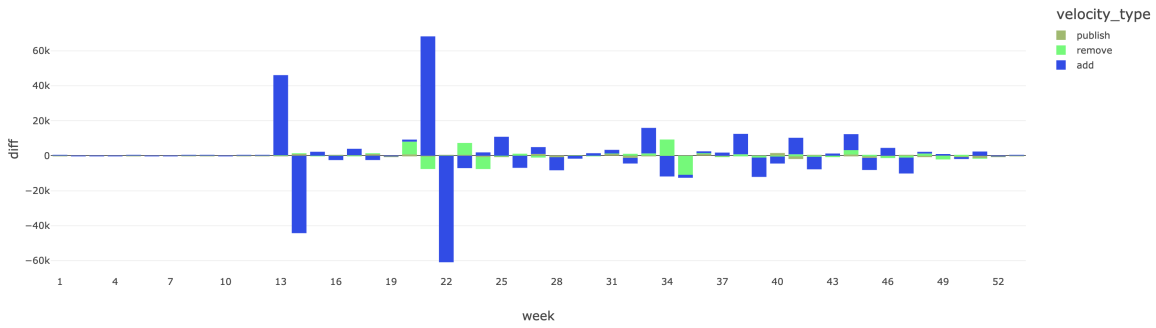

Figure 3.1: Velocity chart showing week by week increases and decreases in added, removed, and published CORD-19 papers.

In Figure 3.2 series velocity data for `remove` (orange) and `publish` (blue) is plotted by relative week of data instead of actual week where (relative) week 0 is the date the paper was added to CORD-19. The plot reveals that the vast majority of published papers were added to CORD-19 right around the week they were published, with ~30K the exact week, ~40K one week after, and ~5K each of the 4 four weeks prior to publish date. Removals show a clear but smaller ~10K spike of occurring 1 week after being initially added to CORD-19. In total 12% of 2020 papers were removed from the CORD-19 corpus and 54% papers were published from the CORD-19 corpus over the period of March 01 to December 19, 2020.

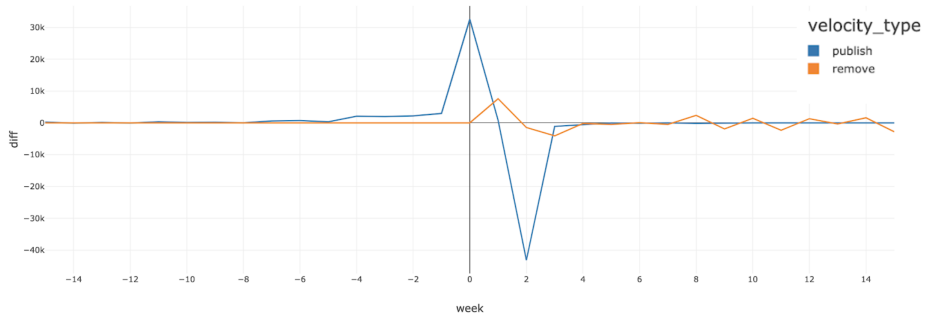

Figure 3.2: Relative velocity chart for publish and removal from CORD-19 in 2020.

CORD-19 Scopus Author Gender Inference. To understand information propagation of CORD-19 scientific literature by author, charts were generated identifying probable gender using Scopus table extracts with fields from a new gender inference implementation covering 16 countries and regions (Jayabalasingham, 2020). The plots reveal a 70% male and 30% female split overall for the CORD-19 1-hop Scopus data (§2.2). The split goes down 3% female to 27% when limited to only CORD-19 DOIs and goes up 1% female to 31% when looking at only non CORD-19 DOIs within the 1-hop data. The comparison of this gender split reflects more historic averages than current trends. According to the Elsevier Jayabalasingham study, from 1999 to 2003 female authors wrote 29% of papers, while over 2014 to 2018 female authorship had grown 9% with 38% female authorship. This alignment of CORD-19 authorship with earlier publishing trends instead of more recent trends is anticipated to be the outcome of two factors. First, the CORD-19 corpus comprises papers published during and after historical coronaviruses such as SARS and MERS (§1.7). Second, the CORD-19 corpus is contained to more limited subject areas which may reflect proportionally larger male authorship. More research would be needed to better understand the differences.

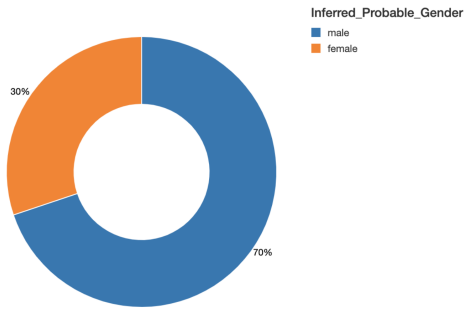

Figure 3.3: Percentage of detected overall gender in CORD-19 Scopus 1-hop corpus.

CORD-19 Scopus Country. To understand information propagation of CORD-19 places, charts were generated identifying papers' `meta_source_country` using Sco-

pus table extracts. Figure 3.4 reveals that 18.3K were published in the United States, followed by 9.7K in China, 6.2K in the United Kingdom, 5.9K in Italy, and then numerous other countries. There were also 4.3K papers with undefined countries due to irregularities existing in publishing standards.

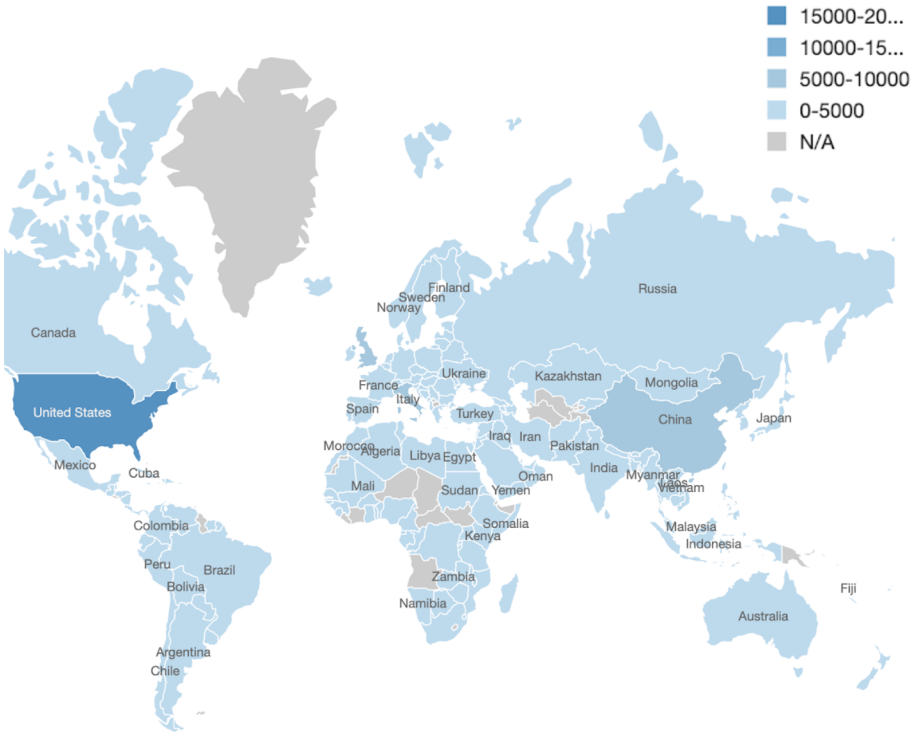

Figure 3.4: 2020 CORD-19 Scopus paper country counts.

Table 3.1: Top 25 2020 CORD-19 Scopus paper countries.

Rank	Country	Counts	Rank	Country	Counts
01	United States	18,308	14	Iran	1,388
02	China	9,275	15	Netherlands	1,106
03	United Kingdom	6,216	16	Turkey	1,084
04	Italy	5,920	17	South Korea	1,056
05	undefined	4,396	18	Switzerland	832
06	India	3,481	19	Singapore	714
07	Germany	2,929	20	Poland	661
08	France	2,663	21	Belgium	622
09	Spain	2,634	22	Taiwan	593
10	Canada	2,461	23	Saudi Arabia	547
11	Australia	2,178	24	Israel	507
12	Brazil	1,897	25	Greece	499
13	Japan	1,540			

Scopus Citation Graph. To understand information propagation of CORD-19 paper references, experiments were run which modeled citation graphs from Scopus 1-hop data (§2.2). From Table 3.2 there were a total of 103M paper edges within this graph with other measures listed.

Table 3.2: Scopus 1-hop citation graph measures.

Measure	Value
Edges ($Paper \rightarrow References$)	103M
Vertices (All)	24M
Vertices (Edge Source)	2M
Vertices (Edge Destination)	24M
In-Degree (Citations)	24M
In-Degree (Citations) Null	64K
Out-Degree (References)	2M
Out-Degree (References) Null	22M

A set of experiments executed graph algorithms (§1.1) out-degree, in-degree, page rank, and connected components over the 1-hop Scopus data. As anticipated, in

the sampled scatterplot in Figure 3.5 there is a linear correlation between in-degree and page rank due to the algorithmic weighting page rank will apply to vertex in-degree.

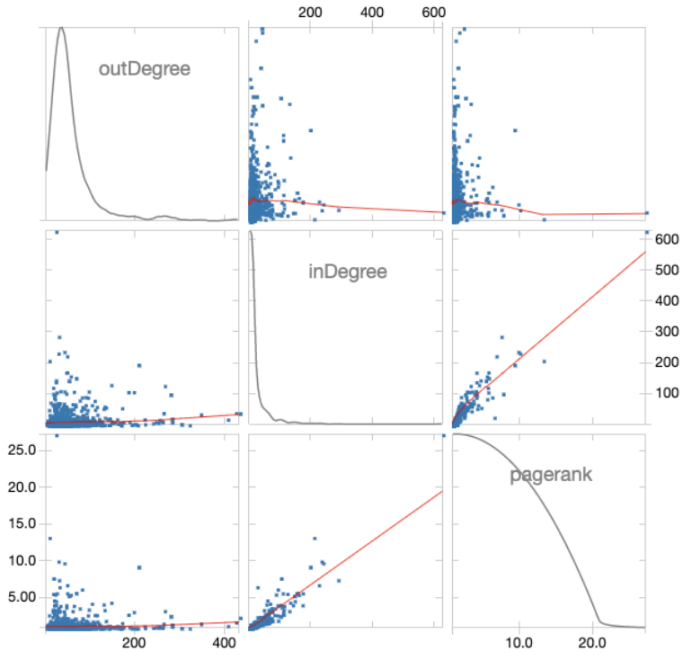

Figure 3.5: 2020 CORD-19 1-hop Scopus degree and page rank scatterplot.

Figure 3.6 plots a logarithmic scaled component versus `page_rank_id1` of the 572 connected components identified by the graph experiments. The top component has a page rank of 820 and has 24M members, meaning the vast majority of papers are interrelated by at least one reference. The size tails off significantly to the second highest component with a page rank of ~5 and only 109 members. This research ultimately focused the most effort around social media engagement and noise (§2.5), but it would be interesting for follow-up research to identify the distinctives of the smaller components and why they are not part of the massive top component.

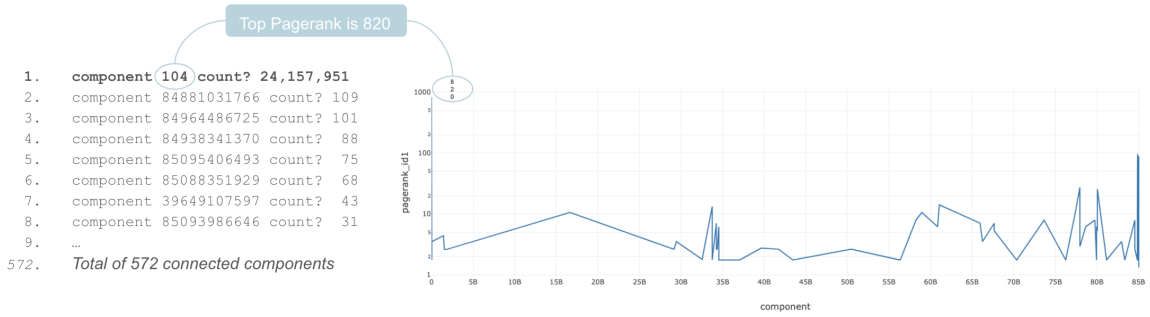

Figure 3.6: 2020 CORD-19 Scopus component and page rank.

3.2. CORD-19 and Twitter Data Interactions

3.2.1 CORD-19 1-Hop Tweet Counts

Experiments to assess information propagation of tweets per scientific literature DOI in Scopus identified through PlumX found the vast majority of papers received a low number of tweets. Figure 3.7 plots `tweet_total` in bins of size 100 on the x-axis against counts of papers falling into a given bin on the y-axis in a logarithmic scale. Bin 1, on the left, contains ~1M papers with less than 100 tweets and bin 2 contains ~50K papers with 101 to 200 tweets, bin 3 contains ~20K papers with 201 to 300 tweets, and so on until the scale moves to only 930 papers with 901-1K tweets at bin 10. The long tail of decay continues all the way out to bin 805 containing 2 papers with 80,401 to 80,500 tweets.

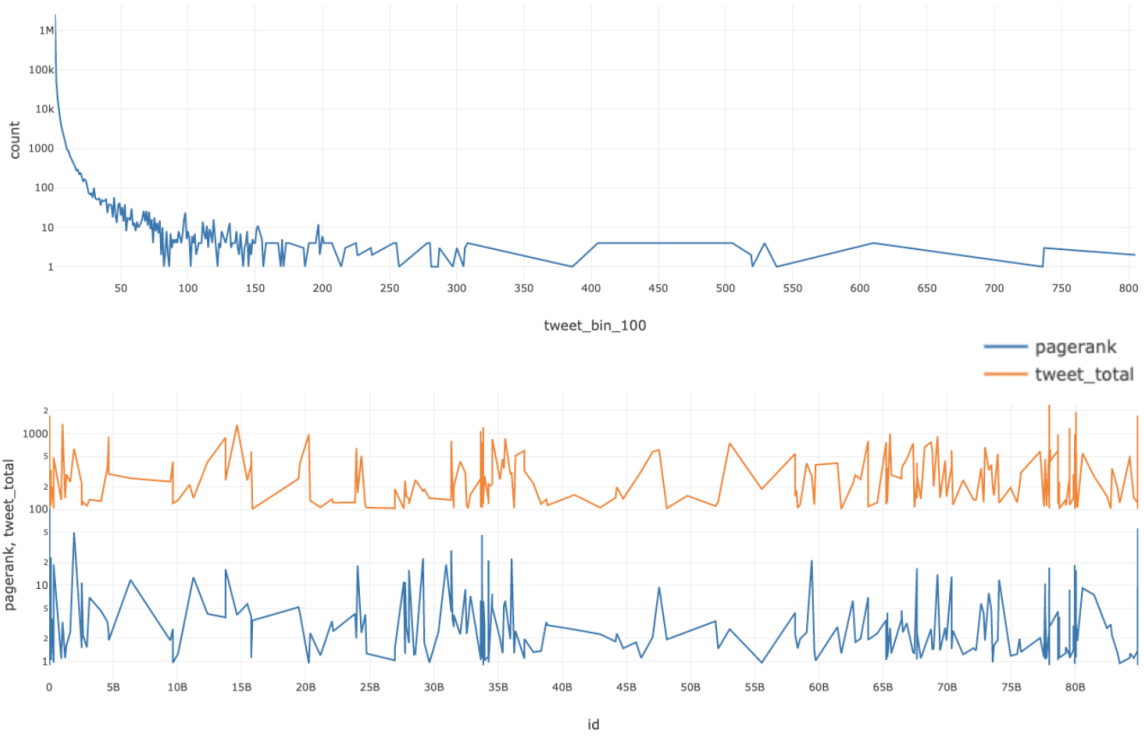

Figure 3.7: 2020 1-hop CORD-19 PlumX tweet counts.

In the bottom of Figure 3.7 a sample of PlumX published `tweet_total` values for a given DOI were compared with calculated page rank of the same paper from the citation graph on a logarithmic scale. Visually there are some general parallels in movement between page rank and `tweet_total` and could be a promising direction for follow-on research.

3.2.2 Compare CORD-19 Publish and Tweet Dates

From experiments to assess information propagation of tweets about papers on Twitter relative to each paper's publish date scaled to month, find that in 2020 CORD-19 paper dates follow a sharp saw-tooth pattern of spiking, visually indicated from a highest peak in April and a dip in June.

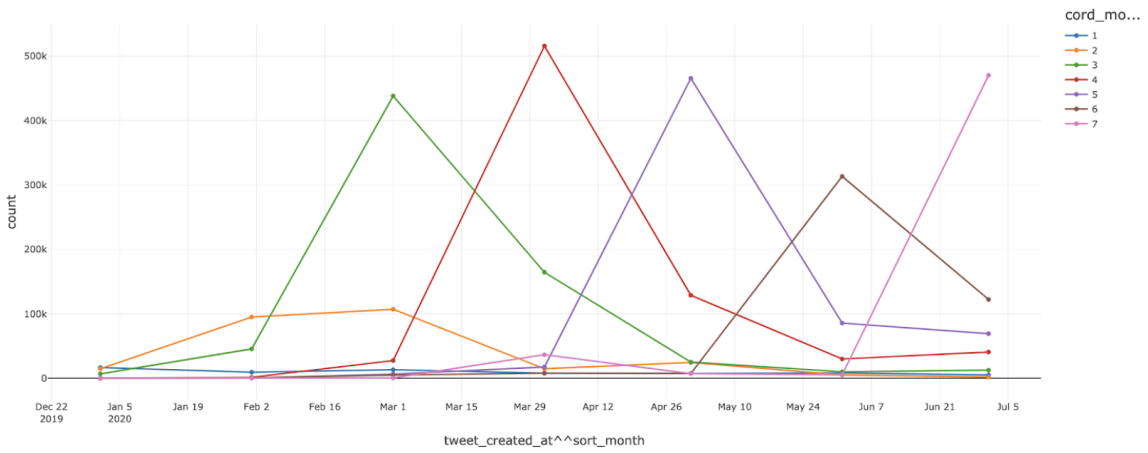

Figure 3.8: 2020 CORD-19 paper dates and tweet dates (through July).

Experiments constraining the CORD-19 corpus to only Scopus, found publish dates generally follow a gentle saw-tooth pattern of spiking, trending towards longer engagement on Twitter than the broader CORD-19 papers. This is immediately visually noticeable in comparing the plots in Figures 3.8 and 3.9. One partial explanatory factor for the longer engagement of Scopus papers would be quality thresholds for which papers make it into Scopus versus the wider CORD-19 corpus. Another partial explanatory factor might include that Scopus contains a higher percentage of more prominent papers relating to Covid-19, to include those published by Elsevier holdings such as The Lancet which drove international attention at various periods of time. Additional research would be needed to understand the reasons for the longer engagement by Scopus papers on Twitter.

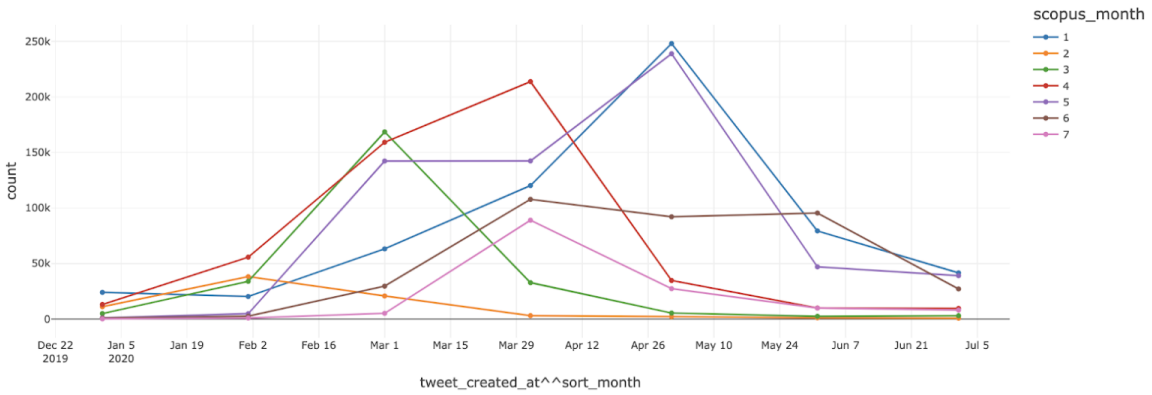

Figure 3.9: 2020 Scopus publish dates and tweet dates (through July).

3.2.3 Tweet Velocity

Figure 3.10 is a velocity plot of the 10 most tweeted DOIs up to July 2020. Visually, there is a distinct repeated pattern of a DOI hitting its high-water mark of tweet volume in first week and then very sharp deceleration. This pattern is not always observed as shown in the circled area including `https://jamanetwork.com/journals/jama/fullarticle/2763983` which had a multi-week run.

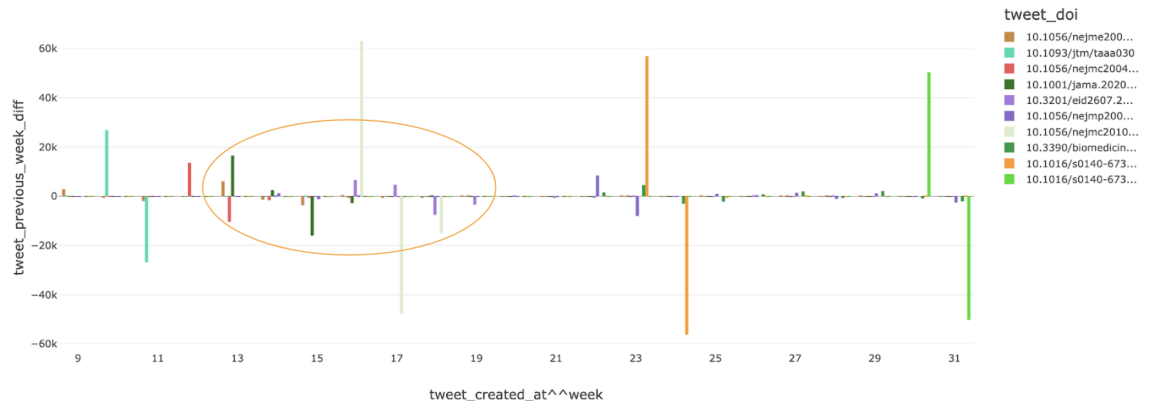

Figure 3.10: 2020 Top-10 tweet velocity graphs (through July).

3.2.4 Tweet Blasts

Tweet boosting behavior methodology was discussed in §2.3.4. The results, given in Figure 3.11, yield up to 332 repeats of the same DOI as the most prolific blast activity among those users within the top 25 `user_originated_tweet_rank`. This is displayed with actual user identities redacted in accordance with Twitter guidelines. The implication is that boosting behavior can skew altmetrics and potentially graph measures as well if not addressed. More mitigation would be possible, to include using §2.5.3 to identify users who are prolific in posting but drive lower engagement.

	tweet_user--id	tweet_user--screen_name	user_originator_tweet_rank	user_originator_tweet_count	tweet_doi	count
1	Redacted		12	654	10.1016/j.jinorgbio.2009.05.019	332
2			12	654	10.1542/peds.2012-1928	281
3			20	457	10.1370/afm.1713	233
4			25	394	10.1007/s00284-012-0277-2	174
5			11	847	10.1016/s0140-6736(03)13771-3	119
6			11	847	10.1016/s0163-4453(84)93192-x	119
7			11	847	10.1016/j.ebiom.2017.01.041	119
8			20	457	10.1056/nejmsa073350	111
9			11	847	10.1093/cid/cis307	105
10			20	457	10.3390/ijms19082410	97
11			20	457	10.1016/j.cell.2016.07.026	70
12			20	457	10.2174/1570159043359477	68
13			11	847	10.1016/j.acap.2010.08.011	66

Figure 3.11: DOI tweet blasts and spam from top 25 most prolific users.

3.2.5 Tweet Chains

Experiments with tweet chains from the reference tweet graph established in §2.4 identified roots, orphans, leaf nodes, and intermediate non-leafs. The plots from Figure 3.12 reveal a dominance of orphan and leaf tweets, both a reflection of low to no engagement. The pie chart shows 75% of tweets are leafs, meaning they terminate the engagement of the chain. 20% are orphans, meaning they are fully isolated with no other engagement. Only 5% of tweets remain then for raising engagement levels. On the right, DOIs having the highest levels of non-leaf (> 5 tweets) and roots (> 35 tweets) are plotted. Even after applying filtering criteria to include only those DOI

chains which were more highly engaged, leaf nodes (blue) and orphan nodes (green) dominate.

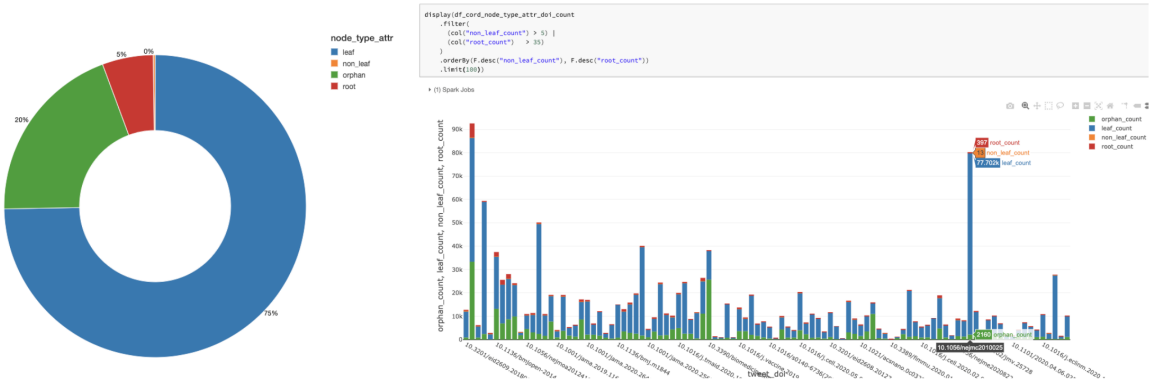

Figure 3.12: Tweet chain root, orphan, leaf, and non-leaf counts.

Experiments to identify depth and width chains are shown in Figure 3.13. Depth chains are uninterrupted sequences of replies, and width chains are are uninterrupted sequences of reposts. The plots reveal the same overall trend towards shorter chains within DOI tweets. In the percentage plots at the top of the figure, find the depth chains growing from 20% to around 45% by chain length 10 and beyond, while the width chains sharply decline from 5% at length 2 and disappear by chain length 5. Also, find mixed chains from 75% to 55% of all chain lengths. In the counts plot at the bottom of the figure, see the sharp tail off after lengths of 6 for mixed and depth chains and see the width sharply decline after length 2 and disappear by length 5. The overall trend for chains being shorter is most likely the outcome of real world sparse data (§2.5.5) more so than being specific to scientific literature; however, additional research could perform more baseline comparisons to validate this assertion.

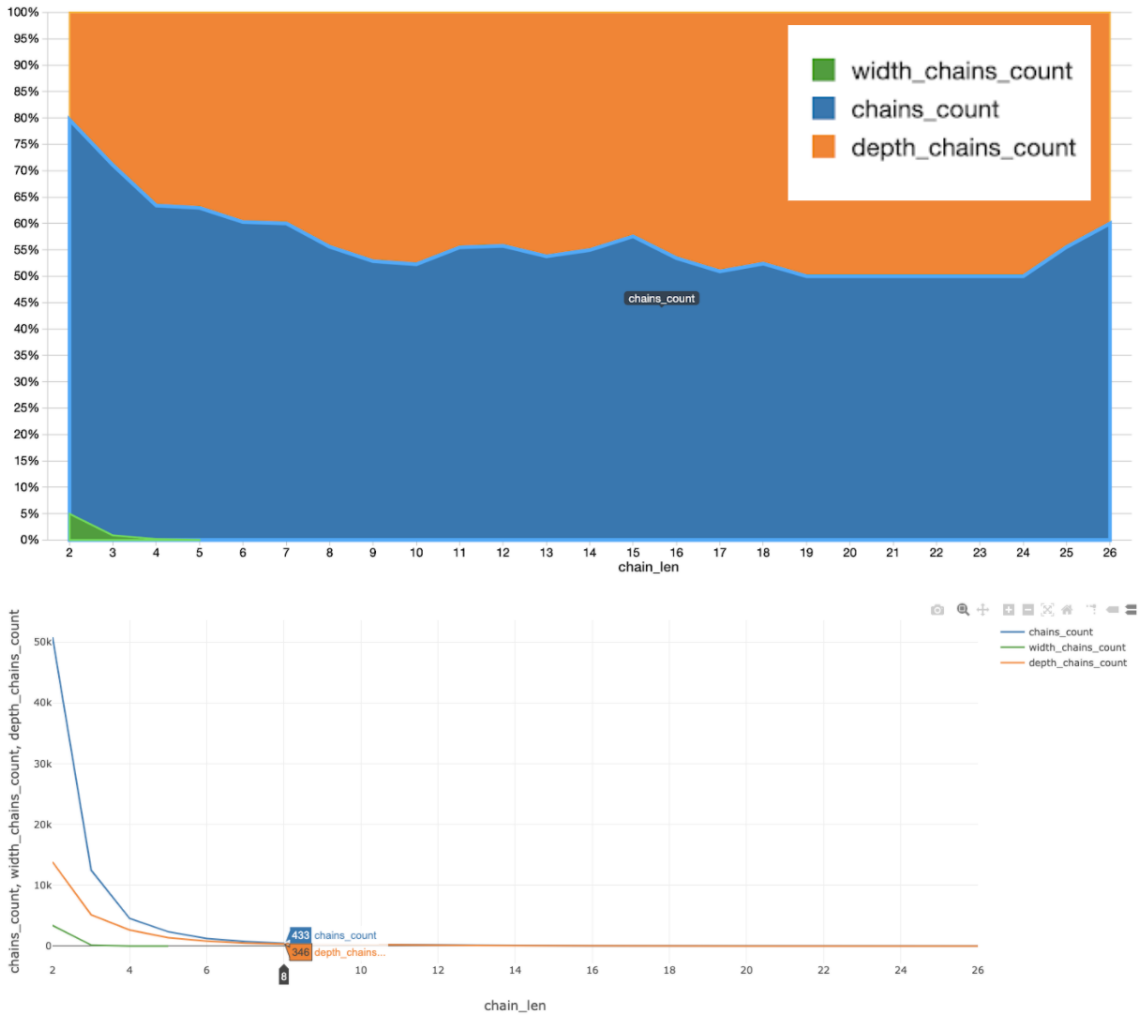

Figure 3.13: Tweet chain lengths, with percentage of sequential width and depth.

An experiment to compare the overall lengths of tweet chains yields Figure 3.14 which plots a Q-Q exponential distribution with 5 very distinct outliers having chain member counts above 700. The "wildest" outlier root is `https://thelancet.com/journals/lancet/article/PIIS0140-6736(20)31483-5/fulltext`. That outlier tweet when viewed directly on Twitter has 1.1K retweets and 134 quote tweets. These are all its immediate children but not the totality of engagement from the tweet, when accounting for all descendants in the resulting tweet tree (§2.4). From the data pro-

vided by PlumX (up to July 2020), that "wild" tweet has 2.3K total descendant tweets which would potentially raise the DOI's altmetric impact. The Social Media Engagement (SME) scoring formalized within this research not only counts each tweet for impact but also changes its scoring based on the number of tweet chains with which a paper is involved and the median length of its chains (§2.5).

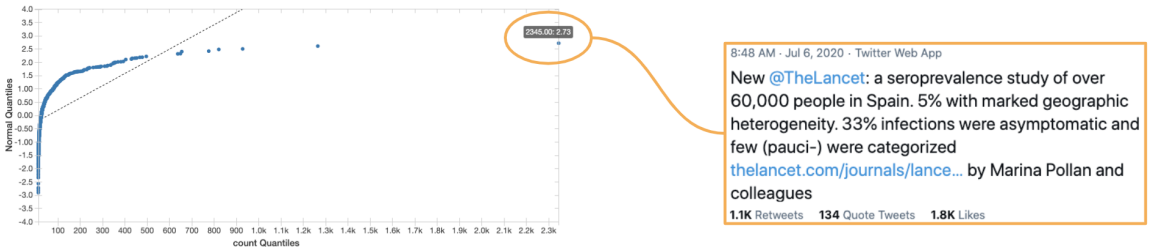

Figure 3.14: Tweet chain length outliers.

3.3. Twitter CORD-19 Social Media Measures

This research limited experiments to Twitter data for social media (SM) engagement (SME) and noise (SMN) scoring (§2.5). The research was specifically constrained to tweets having DOIs found in the CORD-19 dataset up to early July 2020 (§2.1). This section describes all the experiments and results for establishing the measures required for SM score calculations over all available data (§3.3.1), ASJC categorical (§3.3.2), and incremental (§3.3.3). The next section (§3.4) describes the actual SM scoring experiments and results.

3.3.1 SM Measures for Twitter CORD-19

The Twitter data was transformed into multiple DataFrames to generate the following per `user_id` measures based on the methodology established in §2.5:

`n_posts (int), n_coverage (int), n_reach (int), n_quality_indicators (long), median_quality_indicator_value (decimal).`

Function `gen_df_user_posts(...)` in Listing 3.1 was used to populate measure `n_posts` for number of tweets posted by a user. The function uses Spark `Window` function `collect_set` to only store and count unique tweets.

Listing 3.1: Generate DataFrame for user posts measure.

```
def gen_df_user_posts(df_tweet_user_lookup: DataFrame, tweet_id_col: str = "
    tweet_id", new_tweet_ids_col: str = "tweet_ids", new_tweet_ids_size_col: str =
     "tweet_ids_size", partition_cols: List[str] = ["user_id"]) -> DataFrame:
    """
    - Example value for df_tweet_user_lookup is df_tweet_user_date_lookup
    """
    df_result = df_tweet_user_lookup
    for c in partition_cols:
        df_result = df_result.filter(col(c).isNotNull())
    return (
      df_result
          .withColumn(new_tweet_ids_col, F.collect_set(tweet_id_col).over(Window.
    partitionBy(*partition_cols)))
          .withColumn(new_tweet_ids_size_col, F.size(new_tweet_ids_col))
              .select(*partition_cols, new_tweet_ids_col, new_tweet_ids_size_col)
          .distinct()
    )
```

Function `gen_df_tweet_dois(...)` in Listing 3.2 gathers coverage per tweet, followed by function `gen_df_user_dois(...)` in Listing 3.3 used to populate measure `n_coverage` which, for this experiment, was the set of DOIs about which a user tweets. The functions also use Spark `Window` function `collect_set` to only store and count unique DOIs.

Listing 3.2: Generate DataFrame for tweet coverage measure.

```
def gen_df_tweet_dois(df_doi_tweets:DataFrame, tweet_doi_col: str = "tweet_doi",
    new_tweet_dois_col: str = "tweet_dois", new_tweet_dois_size_col: str = "
    tweet_dois_size", partition_cols: List[str] = "tweet_id") -> DataFrame:
    """
    - Example value for df_doi_tweets is df_tweets
    """
    return (
        df_doi_tweets
            .withColumn(new_tweet_dois_col, F.collect_set(tweet_doi_col).over(
    Window.partitionBy(*partition_cols)))
            .withColumn(new_tweet_dois_size_col, F.size(new_tweet_dois_col))
                .select(*partition_cols, new_tweet_dois_col,
    new_tweet_dois_size_col)
```

```
            .distinct()
    )
```

Listing 3.3: Generate DataFrame for user coverage measure.

```
def gen_df_user_dois(df_tweet_dois:DataFrame, df_tweet_user_lookup: DataFrame,
    user_id_col: str = "user_id", tweet_id_col: str = "tweet_id", tweet_dois_col:
    str = "tweet_dois", tmp_tweet_doi_col: str = "tweet_doi",
    new_tweet_dois_size_col: str = "tweet_dois_size") -> DataFrame:
    """
    - Example value for df_doi_tweets is a dataframe returned from `
    gen_df_tweet_dois()`
    - Example value for df_tweet_user_lookup is df_tweet_user_date_lookup for user
     to tweet ids
    """
    return (
        df_tweet_dois.select(tweet_id_col, tweet_dois_col)
            .join(
                df_tweet_user_lookup
                    .selectExpr(tweet_id_col, user_id_col),
                [tweet_id_col]
            )
            .filter(col(user_id_col).isNotNull())
            .select(user_id_col, tweet_dois_col, F.explode(tweet_dois_col).alias(
    tmp_tweet_doi_col))
            .drop(tweet_dois_col)
            .distinct()
            .withColumn(tweet_dois_col, F.collect_set(tmp_tweet_doi_col).over(
    Window.partitionBy(user_id_col)))
            .withColumn(new_tweet_dois_size_col, F.size(tweet_dois_col))
            .drop(tmp_tweet_doi_col)
            .distinct()
    )
```

Function `gen_df_chains_unique_user_reach(...)` in Listing 3.4 was used to populate measure `n_reach` for number of unique users engaged by a user's posts. The function also uses Spark `Window` function `collect_set` to only store and count unique tweets.

Listing 3.4: Generate DataFrame for unique number of users reached.

```
def gen_df_chains_unique_user_reach(df_chains: DataFrame, chain_node_user_col: str
     = "chain_node_user", new_all_reach_col: str = "all_reach",
    new_all_reach_size_col: str = "all_reach_size", partition_cols: List[str] = "
    user_id") -> DataFrame:
    """
    - Example value for df_chains is a dataframe returned from `
    unique_true_rel_roots()`
```

```
    """
    return (
        df_chains
            .withColumn(new_all_reach_col, F.collect_set(chain_node_user_col).over
    (Window.partitionBy(*partition_cols)))
            .withColumn(new_all_reach_size_col, F.size(new_all_reach_col))
                .select(*partition_cols, new_all_reach_col, new_all_reach_size_col
    )
            .distinct()
    )
```

Function `gen_df_chains_count(...)` in Listing 3.5 was used to populate measure `n_quality_indicators` for count of chains generated by a user's posts. Only users who are either true or relative chain roots are considered as described in §2.4.

Listing 3.5: Generate DataFrame with number of chains for each user.

```
def gen_df_chains_count(df_union_chain_sizes: DataFrame, group_by_cols: List[str]
    = ["user_id"], new_chain_counts_col: str = "all_chain_counts") -> DataFrame:
    """
    - Example of df_union_chain_sizes is the result of `
    union_true_rel_root_chain_sizes(df_root_user_chains_annotated,
    df_user_chains_aug)`
    """
    return (
        df_union_chain_sizes
            .groupBy(*group_by_cols)
                .count()
            .selectExpr(*group_by_cols, f"count as {new_chain_counts_col}")
    )
```

Function `add_avg_median_stddev_skewness_variance(...)` in Listing 3.6 was used to populate measure `median_quality_indicator_value` for median length over all chains generated by a user's posts. Only users who are either true or relative chain roots are considered as described in §2.4. The use of average was evaluated over median. The use of average would have been easier for self-boosting behaviors to target in order to achieve artificially inflated scores. Given chain lengths 1, 2, 3, 4, and 25 the average would be $\frac{(1+2+3+4+25)}{5} = 7$ which means a user with self-boosting intentions could focus on just one engagement chain length to raise their

quality measure. Median of the same would be 3. Therefore, since median is harder to manipulate, it was chosen to be the basis of the quality indicator value.

Listing 3.6: Add columns for various calculated statistics.

```
def add_avg_median_stddev_skewness_variance(df: DataFrame, val_col: str,
    partition_cols: List[str], new_col_prefix: str, filter_vals_gt0: bool = True,
    filter_vals_null: bool = True) -> DataFrame:
    """
    - Adding orderBy to make sure all calculations are accurate
    - pass None for partition_cols if not used
    """
    window_op = None
    if partition_cols is not None:
        window_op = Window.partitionBy(*partition_cols).orderBy(val_col).
    rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)
    else:
        window_op = Window.orderBy(val_col).rowsBetween(Window.unboundedPreceding,
     Window.unboundedFollowing)
    df_return = df
    if filter_vals_null:
        df_return = df_return.filter(col(val_col).isNotNull())
    if filter_vals_gt0:
        df_return = df_return.filter(f"{val_col} > 0")
    df_return = (
        df_return
            .withColumn(
                f"avg_{new_col_prefix}_{val_col}",
                F.expr(f'avg({val_col})').over(window_op).cast(DecimalType(scale
    =2))
            )
            .withColumn(
                f"median_{new_col_prefix}_{val_col}",
                F.expr(f'percentile_approx({val_col}, 0.5)').over(window_op).cast(
    DecimalType(scale=2))
            )
            .withColumn(
                f"stddev_{new_col_prefix}_{val_col}",
                F.expr(f'stddev({val_col})').over(window_op).cast(DecimalType(
    scale=2))
            )
            .withColumn(
                f"skewness_{new_col_prefix}_{val_col}",
                F.expr(f'skewness({val_col})').over(window_op).cast(DecimalType(
    scale=2))
            )
            .withColumn(
                f"variance_{new_col_prefix}_{val_col}",
                F.expr(f'variance({val_col})').over(window_op).cast(DecimalType(
    scale=2))
            )
```

```
            .select(*partition_cols, f"avg_{new_col_prefix}_{val_col}", f"median_{
    new_col_prefix}_{val_col}",
                    f"stddev_{new_col_prefix}_{val_col}", f"skewness_{
    new_col_prefix}_{val_col}", f"variance_{new_col_prefix}_{val_col}")
            .distinct()
    )
    if partition_cols is not None:
        df_return = df_return.orderBy(*partition_cols)
    return df_return
```

These are the measures needed to execute the experiment to score SME and SMN for all available Twitter data in §3.4.1.

3.3.2 SM Measures for ASJC Twitter CORD-19

Similar to §3.3.1 Twitter data was transformed into multiple DataFrames to generate the per `user_id` measures based on the methodology established in §2.5. In this experiment, the CORD-19 DOIs available in Scopus were exclusively considered. Within that subset, all the Twitter data was isolated to each ASJC categorized for the set of DOIs. Function `gen_df_user_posts(...)` as introduced in Listing 3.1 was invoked over a loop which pre-filters for each ASJC to limit the calculation of measure `n_posts`, shown in Listing 3.7.

Listing 3.7: Filter to ASJC and calculate per user posts.

```
for i,a in enumerate(asjc_list):
    dict_df_user_asjc_posts[str(a)] = (
        gen_df_user_posts(df_master_asjc_lookup_delta.filter(f"asjc = {a}"))
            .withColumn("asjc", F.lit(a))
    )
```

For measure `n_coverage`, the same pattern of filtering by ASJC was followed as in Listing 3.7, first calling `gen_df_tweet_dois(...)`, then calling:

```
gen_df_user_dois(
    dict_df_asjc_tweet_coverage[str(a)],
    df_master_asjc_lookup_delta.filter(f"asjc = {a}")
)
```

For measure `n_reach`, the same pattern of filtering by ASJC was followed as in Listing 3.7, calling:

```
gen_df_chains_unique_user_reach(
    unique_true_rel_roots(
        df_true_root_asjc_chains.filter(f"asjc = '{a}'"),
        df_rel_root_asjc_chains.filter(f"asjc = '{a}'")
    )
)
```

For measure `n_quality_indicators`, the same pattern of filtering by ASJC was followed as in Listing 3.7, calling:

```
gen_df_chains_count(
    unique_true_rel_roots(
        df_true_root_asjc_chains.filter(f"asjc = '{a}'"),
        df_rel_root_asjc_chains.filter(f"asjc = '{a}'")
    )
)
```

Measure `n_quality_indicators` was calculated the same as in Listing 3.5 but restricted to AJSC, then `median_quality_indicator_value` was calculated by calling:

```
add_avg_median_stddev_skewness_variance(
    df_asjc_true_rel_root_chains,
    "chain_size", ["user_id","asjc"], "asjc")
```

These are the measures needed to execute the experiment to score SME and SMN for all available ASJC Twitter data in §3.4.2.

3.3.3 SM Measures for Incremental Twitter CORD-19

Similar to §3.3.1 Twitter data was transformed into multiple DataFrames to generate per `user_id` measures based on the methodology established in §2.5. The available CORD-19 Twitter data was restricted to the first 26 weeks of 2020 for these experiments which include standard incremental as well as rolling blended in §2.5.8. In standard incremental, only user activities within the same week are scored, with

no exceptions.

Function `gen_df_user_posts(...)` in Listing 3.1 was used to populate measure `n_posts` for number of tweets posted by a user. To restrict to a week, argument `partition_cols=["year","week","user_id"])` was provided to drive Spark `Window` operations as follows:

```
gen_df_user_posts(df_tweet_user_date_lookup.filter("year = 2020"), partition_cols
    =["year","week","user_id"])
```

For measure `n_coverage`, the same pattern of overriding partition columns to include week was passed as in Listing 3.2, calling:

```
gen_df_tweet_dois(
        df_tweet_user_date_lookup.filter("year = 2020").join(df_tweets.select("
    tweet_id","tweet_doi"), ["tweet_id"]),
        partition_cols=["year","week","user_id"]
    )
```

For measure `n_reach`, the same pattern of overriding partition columns to include week was passed as in Listing 3.4, calling:

```
gen_df_chains_unique_user_reach(
    df_weekly_chains.selectExpr("root_year as year", "root_week as week", "
    root_user_id as user_id", "chain_user_id"),
    chain_node_user_col="chain_user_id",
    partition_cols=["year", "week", "user_id"]
)
```

For the standard incremental experiment, chain sizes were restricted to only the same week as in Listing 3.8 by expression `filter("root_week = chain_week")` to only evaluate chains where engagement occurs within the same week. Since the methodology includes relative roots along the chain, a root will be any user post within a chain which was engaged by a user through reply or repost. The standard incremental technique only considers data within each week's boundary since posts later in the week have less opportunity to be engaged as those posted earlier in the week, making standard incremental vulnerable to losing chain fidelity from week to

week.

Listing 3.8: Generate DataFrame for standard incremental weekly chain sizes.

```
df_weekly_chain_sizes = (
    df_weekly_chains
        .filter("root_year = chain_year") # only chains in same year and week
        .filter("root_week = chain_week")
        .selectExpr("root_year as year", "root_week as week","root_id", "
    root_user_id as user_id", "chain_id", "chain_pos")
    .withColumn(
        "weekly_chain_nodes",
        F.expr("collect_list(chain_id)").over(
            Window.partitionBy("year", "week", "user_id", "root_id")
                .orderBy("chain_pos")
                .rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)
        )
    )
    .drop("chain_id","chain_pos")
    .distinct()
    .withColumn(
        "weekly_chain_size",
        F.size("weekly_chain_nodes")
    )
)
```

Measure `n_quality_indicators` was calculated the same as in Listing 3.5 but restricted to week, then `median_quality_indicator_value` was calculated by calling:

```
add_avg_median_stddev_skewness_variance(
    df_weekly_chain_sizes, "weekly_chain_size",
    ["year","week","user_id"], "user"
)
```

These are the measures needed to execute the experiment to score SME and SMN for standard incremental Twitter data in §3.4.3.

SM Measures for Rolling Incremental Twitter CORD-19. Rolling incremental differs from standard incremental which strictly constrains chains to those acted upon within the same week by expression `filter("root_week = chain_week")` as in Listing 3.8. Rolling incremental chains can be lagging by n weeks or blended lagging by n weeks as described in §2.5. For the available CORD-19 Twitter data rolling incre-

mental measures are calculated over a series of operations to follow.

i. The first operation prepared rolling incremental chain measures. It grouped relative and actual roots by the root user and the chain year through Spark function `pivot("chain_week")` which translated rows to columns. This pivot operation acted upon weekly counts of chains for each root in the Twitter 2020 data:

```
pivot_group_cols = ["root_id", "root_user_id", "chain_year"]
df_weekly_chain_sizes = (
    df_weekly_chains
        .groupBy(*pivot_group_cols)
        .pivot("chain_week")
        .count()
        .fillna(0)
)
```

ii. The next operation stored all pivot week columns in list `week_n_cols` and created a more friendly column name stored in list `weekly_cols`:

```
# sort weeks
week_n_cols = []
for c in df_weekly_chain_sizes.columns:
    if c not in pivot_group_cols:
        week_n_cols.append(int(c)) # need numbers
week_n_cols.sort()
# add week size cols per root_id
weekly_cols = []
for c in week_n_cols:
    week_col = f"week_{c}_size"
    weekly_cols.append(week_col)
    df_weekly_chain_sizes = df_weekly_chain_sizes.withColumnRenamed(str(c),
    week_col)
```

iii. Lag by 1 week column `week_{c}_lag1_size` was populated for each week c in `week_n_cols`. The size value was supplied from the value of the DataFrame field of `week_cols` at the same index position i. Week 1 must be handled specially since it does not have a lagging week in 2020, handled for all weeks in the following loop:

```
week_lag1_cols = []
for i,c in enumerate(week_n_cols):
    week_col = f"week_{c}_lag1_size"
    week_lag1_cols.append(week_col)
    if c == 1:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]))
    else:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]) + col(weekly_cols[i-1]))
```

iv. Lag by 2 weeks column `week_{c}_lag2_size` was then populated for each week c in 2020. Weeks 1 and 2 must be handled specially since week 1 does not have any lagging weeks and week 2 only has one lagging week:

```
week_lag2_cols = []
for i,c in enumerate(week_n_cols):
    week_col = f"week_{c}_lag2_size"
    week_lag2_cols.append(week_col)
    if c == 1:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]))
    elif c == 2:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]) + col(weekly_cols[i-1]))
    else:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]) + col(weekly_cols[i-1]) + col(weekly_cols[i-2]))
```

v. Lag by 3 weeks column `week_{c}_lag3_size` was then populated for each week c in 2020. Weeks 1 to 3 must be handled specially since week 1 does not have any lagging weeks, week 2 only has one lagging week, and week 3 only has two lagging weeks:

```
week_lag3_cols = []
for i,c in enumerate(week_n_cols):
    week_col = f"week_{c}_lag3_size"
    week_lag3_cols.append(week_col)
    if c == 1:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]))
    elif c == 2:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]) + col(weekly_cols[i-1]))
```

```
    elif c == 3:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]) + col(weekly_cols[i-1]) + col(weekly_cols[i-2]))
    else:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]) + col(weekly_cols[i-1]) + col(weekly_cols[i-2]) + col
    (weekly_cols[i-3]))
```

vi. Lag by 4 weeks column `week_{c}_lag4_size` was then populated for each week c in 2020. Weeks 1 to 4 must be handled specially since week 1 does not have any lagging weeks, week 2 only has one lagging week, week 3 only has two lagging weeks, and week 4 only has three lagging weeks:

```
week_lag4_cols = []
for i,c in enumerate(week_n_cols):
    week_col = f"week_{c}_lag4_size"
    week_lag4_cols.append(week_col)
    if c == 1:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]))
    elif c == 2:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]) + col(weekly_cols[i-1]))
    elif c == 3:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]) + col(weekly_cols[i-1]) + col(weekly_cols[i-2]))
    elif c == 4:
        df_weekly_chain_sizes = df_weekly_chain_sizes.withColumn(week_col,
    col(weekly_cols[i]) + col(weekly_cols[i-1]) + col(weekly_cols[i-2]) + col
    (weekly_cols[i-3]))
    else:
        df_weekly_chain_sizes = (
            df_weekly_chain_sizes.withColumn(
                week_col,
                col(weekly_cols[i]) + col(weekly_cols[i-1]) + col(
    weekly_cols[i-2]) + col(weekly_cols[i-3]) + col(weekly_cols[i-4]))
            )
```

vii. All the DataFrame column names generated after the initial pivot were flattened into a single list:

```
all_week_cols = sum([list(z) for z in zip(weekly_cols, week_lag1_cols,
    week_lag2_cols, week_lag3_cols, week_lag4_cols)], [])
```

viii. The command `stack` was used to effectively un-pivot the DataFrame such that its columns become rows. The additional operations use Spark `Window` functions to populate the lagging week columns in the transformed DataFrame:

```
l_stack=[]
separator = ', '
k_tmp=separator.join(all_week_cols).split(",")
n_stack=len(all_week_cols)
for c in range(n_stack):
    l_stack.append(f"'{all_week_cols[c]}',{k_tmp[c]}")
k_stack = separator.join(l_stack)
df_weekly_chain_sizes = (
    df_weekly_chain_sizes
        .join(
            df_weekly_chains
                .groupBy(*pivot_group_cols)
                .count()
                .withColumnRenamed("count", "sum_chain_sizes"),
            pivot_group_cols
        )
        .select(*pivot_group_cols, "sum_chain_sizes",  F.expr(f"""stack({
    n_stack},{k_stack}) as (week_cat, week_cat_size)"""))
        .filter("week_cat_size > 0") # don't want 0s
        .withColumn(
            "week",
            (F.split("week_cat", "_")[1]).cast(IntegerType()) # parse out
    the week num
        )
        .withColumn(
            "week_cat_rank",
            F.when(F.split("week_cat", "_")[2] == F.lit("size"), F.lit(1))
            .when(F.split("week_cat", "_")[2] == F.lit("lag1"), F.lit(2))
            .when(F.split("week_cat", "_")[2] == F.lit("lag2"), F.lit(3))
            .when(F.split("week_cat", "_")[2] == F.lit("lag3"), F.lit(4))
            .when(F.split("week_cat", "_")[2] == F.lit("lag4"), F.lit(5))
        )
        .withColumn(
            "week_lag1_size",
            F.when(
                F.split("week_cat", "_")[2] == F.lit("size"),
                F.lead("week_cat_size", offset=1).over(Window.partitionBy(*
    pivot_group_cols,"week").orderBy("week_cat_rank"))
            )
        )
        .withColumn(
            "week_lag2_size",
            F.when(
                F.split("week_cat", "_")[2] == F.lit("size"), # only want to
     calc from the week, not the lags
```

```
            F.lead("week_cat_size", offset=2).over(Window.partitionBy(*
    pivot_group_cols,"week").orderBy("week_cat_rank"))
        )
    )
    .withColumn(
        "week_lag3_size",
        F.when(
            F.split("week_cat", "_")[2] == F.lit("size"), # only want to
     calc from the week, not the lags
            F.lead("week_cat_size", offset=3).over(Window.partitionBy(*
    pivot_group_cols,"week").orderBy("week_cat_rank"))
        )
    )
    .withColumn(
        "week_lag4_size",
        F.when(
            F.split("week_cat", "_")[2] == F.lit("size"), # only want to
     calc from the week, not the lags
            F.lead("week_cat_size", offset=4).over(Window.partitionBy(*
    pivot_group_cols,"week").orderBy("week_cat_rank"))
        )
    )
    .filter(F.split("week_cat", "_")[2] == F.lit("size")) # only want
    week_cat rows, not the lag rows
    .fillna(0)
    .selectExpr(*pivot_group_cols, "sum_chain_sizes", "week", "
    week_cat_size as week_size", "week_lag1_size", "week_lag2_size","
    week_lag3_size", "week_lag4_size")
)
```

ix. Function `add_avg_median_stddev_skewness_variance` was the next operation called for the current week, as was done in standard incremental, which generated `median_quality_indicators_value`. Additionally, the same function was called on each of the four lagging weeks which were joined back together:

```
stats_partition_cols = ["year","week","user_id"]
df_chains_stats = (
    add_avg_median_stddev_skewness_variance(
        df_weekly_chain_sizes_delta, "week_size", stats_partition_cols, "
    user"
    )
    .drop("stddev_user_week_size", "skewness_user_week_size", "
    variance_user_week_size")
    .join(
        add_avg_median_stddev_skewness_variance(
```

```
            df_weekly_chain_sizes_delta, "week_lag1_size",
    stats_partition_cols, "user"
        )
        .drop("stddev_user_week_lag1_size", "skewness_user_week_lag1_size",
    "variance_user_week_lag1_size"),
        stats_partition_cols
    )
    .join(
        add_avg_median_stddev_skewness_variance(
            df_weekly_chain_sizes_delta, "week_lag2_size",
    stats_partition_cols, "user"
        )
        .drop("stddev_user_week_lag2_size", "skewness_user_week_lag2_size",
    "variance_user_week_lag2_size"),
        stats_partition_cols
    )
    .join(
        add_avg_median_stddev_skewness_variance(
            df_weekly_chain_sizes_delta, "week_lag3_size",
    stats_partition_cols, "user"
        )
        .drop("stddev_user_week_lag3_size", "skewness_user_week_lag3_size",
    "variance_user_week_lag3_size"),
        stats_partition_cols
    )
    .join(
        add_avg_median_stddev_skewness_variance(
            df_weekly_chain_sizes_delta, "week_lag4_size",
    stats_partition_cols, "user"
        )
        .drop("stddev_user_week_lag4_size", "skewness_user_week_lag4_size",
    "variance_user_week_lag4_size"),
        stats_partition_cols
    )
)
```

x. Calculation of the blended lagging median chain values builds on the previous operations by taking the arithmetic average for each week and its lagging week, or weeks, which were joined back together:

```
df_sm_measures = (
    gen_df_sm_measures(df_user_posts_delta, df_user_coverage_delta,
    df_chains_reach_delta, df_chains_median_delta, join_cols=["year","week","
    user_id"])
        .join(
            df_chains_median_delta
                .selectExpr(
                    "year","week","user_id",
```

```
                "median_lag1_chain_size as
lag1_median_quality_indicator_value",
                "median_lag2_chain_size as
lag2_median_quality_indicator_value",
                "median_lag3_chain_size as
lag3_median_quality_indicator_value",
                "median_lag4_chain_size as
lag4_median_quality_indicator_value"
            ),
        ["year","week","user_id"]
    )
    .withColumn(
        "blended_lag1_median_quality_indicator_value",
        sum(
            x for x in [
                col("median_quality_indicator_value"), col("
lag1_median_quality_indicator_value")
            ]
        )/2
    )
    .withColumn(
        "blended_lag2_median_quality_indicator_value",
        sum(
            x for x in [
                col("median_quality_indicator_value"), col("
lag1_median_quality_indicator_value"), col("
lag2_median_quality_indicator_value")
            ]
        )/3
    )
    .withColumn(
        "blended_lag3_median_quality_indicator_value",
        sum(
            x for x in [
                col("median_quality_indicator_value"), col("
lag1_median_quality_indicator_value"), col("
lag2_median_quality_indicator_value"),
                col("lag3_median_quality_indicator_value")
            ]
        )/4
    )
    .withColumn(
        "blended_lag4_median_quality_indicator_value",
        sum(
            x for x in [
                col("median_quality_indicator_value"), col("
lag1_median_quality_indicator_value"), col("
lag2_median_quality_indicator_value"),
                col("lag3_median_quality_indicator_value"), col("
lag4_median_quality_indicator_value")
            ]
```

```
                )/5
            )
    )
```

Measure `median_quality_indicator_value` was affected by the calculation changes in rolling incremental experiments which in turn will influence SME and SMN scores. The standard incremental technique will be vulnerable at each week's boundary since posts later in the week have less opportunity to be engaged as those posted earlier in the week and chain fidelity would be lost week to week. This vulnerability was sufficiently addressed by the rolling incremental modifications.

3.4. Twitter CORD-19 Social Media Scores

The previous section (§3.3) described all the experiments and results to establish the measures required for SM score calculations of the Twitter DOI data. This section describes the actual SM scoring experiments and results over all available data (§3.4.1), ASJC categorical (§3.4.2), and incremental (§3.4.3).

3.4.1 SM Scores for Available Twitter CORD-19

The final count of unrestricted measures generated in §3.3.1 for available Twitter data was 872K, `df_user_posts` count was 872K, `df_user_coverage` count was 845K, `df_chains_reach` count was 185K, and `df_chains_median` count was 872K. Listing 3.9 is the standardized method called to score SM measures:

Listing 3.9: Calculate SME and SMN scoring for measures.

```
def df_sm_scores_ranked(df_sm_measures: DataFrame) -> DataFrame:
    """
    Combines calculating sme and smn scores.
    - See those signatures for more (not showing all the named params)
    """
    df_sme_scores = df_sme_scores_ranked(df_sm_measures)
    return df_smn_scores_ranked(df_sme_scores)
```

This will ultimately only score the `user_ids` with data across all fields as users with missing fields will end up with a score 0. The resulting columns are as follows:

```
user_id (long), n_posts (int), n_coverage (int), n_reach (int),
    n_quality_indicators (long), median_quality_indicator_value (decimal),
    sme_score (float), sme_score_rank (int), sme_score_row_num (int), smn_score (
    float), smn_score_rank (int), smn_score_row_num (int).
```

SM Score Results for Available Twitter. The results from scoring users was first looked at in aggregate, shown in Table 3.3 where the SME scores were bucketed into scores of 10 with bucket 0 for users with a SME score of zero, bucket 1 for SME scores between 0 and 10, and so on. The first four buckets are rather large with ~26K users in bucket 0, ~734K in bucket 1, ~37K in bucket 2, and ~158K in bucket 3.

Table 3.3: First 10 SME scores, bucketed by 10.

SME Bucket	**Count**	**SME Bucket**	**Count**
0	26,524	5	5,925
1	734,769	6	4,390
2	37,298	7	3,265
3	158,86	8	2,611
4	9,163	9	2,187

Given the larger sizes overall of the first four buckets (0 to 3), Figure 3.15 plotted the buckets after 3 using a logarithmic scale to render with the exponential decay. Only ~100 users are in bucket 100 (SME scores from 991 to 1,000) which further declines up through bucket 1,111 (SME scores from 11,101 to 11,110). Only the first 1,000 rows are plotted here, so the long tail of decay will go on further.

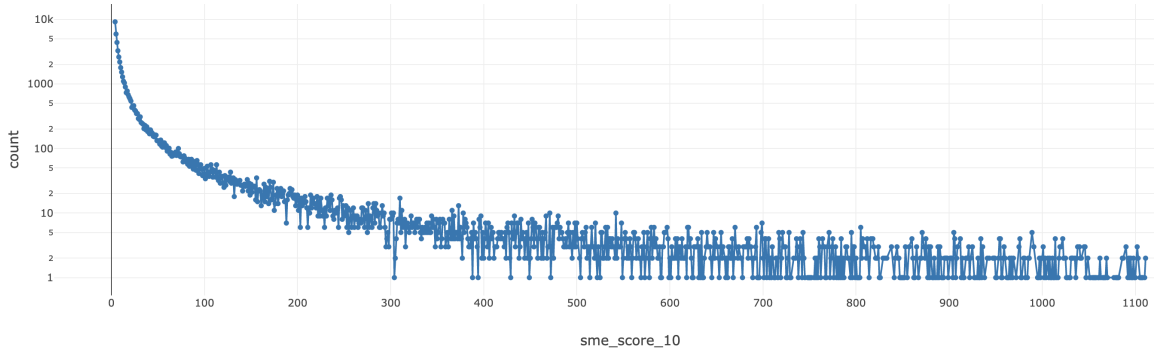

Figure 3.15: Standard user SME scores bucketed by 10.

In Figure 3.15 the top 1,000 users ordered by number of posts were plotted on the x-axis with their SME score and SMN scores plotted on the y-axis. The results show first that for these users, who are the most prolific in tweeting, the SME scores are generally in the 10K to 10M range, placing nearly all of them in the outlier range of where Figure 3.15 ended above. Also, SMN scores are generally below 1, which is the threshold for considering noisiness, meaning these are not generally noisy. Additionally, find there is a lack of straight correlation between number of user posts and SME and SMN scores. This characteristic is part of the design of the SM scores to ensure posts alone do not drive engagement.

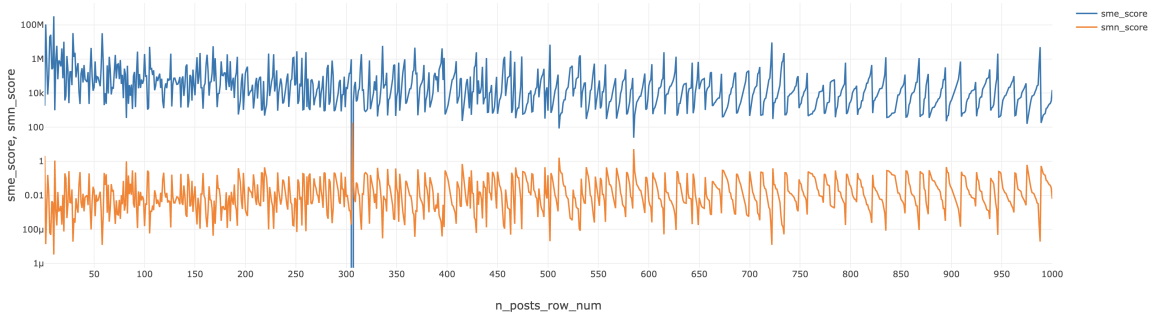

Figure 3.16: Compare number of posts against SME and SMN scores for top 1,000 users.

One clear anomaly from the other posts is at user rank 306 (on number of

posts) where the SME score drops to zero and the SMN score spikes to 183, making that user a very strong candidate for one who is prolific in posting but not driving engagement. Given that Twitter specifies rules for identifying users in a negative context within the research, the user identity will not be revealed. However, under anonymity, it is safe to mention this user was deleted at the time of this research making that account more probable as a spammer or bot of sorts, and that account ended up having the highest overall SMN score.

Additionally, further analysis based on calculating rank differences from number of posts ranking (used above) and ranking based on SME and SMN scores (not shown), revealed more anomalous activity. User at rank 415 with 159 posts and user at rank 975 with 96 posts were using the exact same profile image and were most likely also spammers or bots. This is really promising to find several anomalous accounts readily based on SM scores.

Compare SM Scores with Page Rank. Experiments comparing SME and SMN scores against page rank are plotted in Figure 3.17, showing a distinct lack of correlation with page rank (smooth blue line), SME score in orange, and SMN score in green). The plot is ordered by users with the highest page rank calculated from the Twitter reference graph. This result is similar to 3.15 which demonstrated that number of posts did not directly correlate either. This characteristic is part of the design of the SM scores to ensure quality of engagement such as unique users reached, number of coverage items (DOIs for this research), and tweet chain complexity drives engagement over centrality within the graph structure. Also, observe very noticeable drops in SME scores and spikes in SMN scores which would be candidates for uncovering additional outliers from normal patterns of engagement.

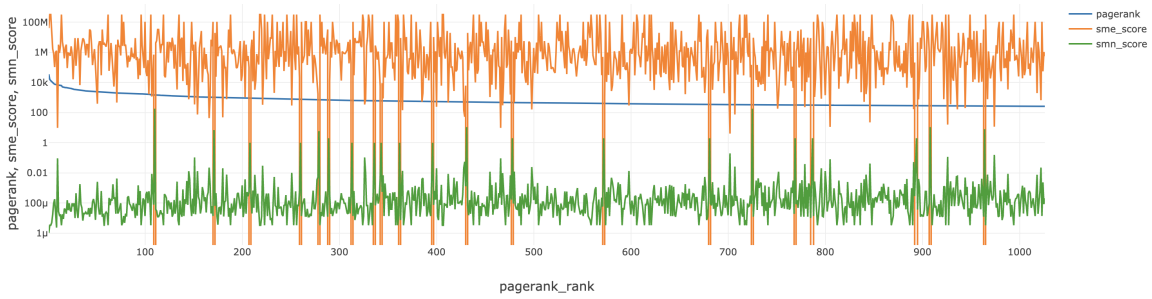

Figure 3.17: Compare SME and SMN scores for top 1,000 page rank users.

3.4.2 SM Scores for ASJC Twitter CORD-19

All user posts in the Twitter CORD-19 data were scored per Scopus ASJC engagement as described in §3.3.2. The measures themselves were passed to Listing 3.9 as done in §3.4.1. This resulted in 1.6M rows involving 317 ASJC categories. From those scores, the 317 ASJC categories themselves were scored by taking the sample standard deviation as described in §2.5.6.

Listing 3.10: Calculate ASCJ SME and SMN scoring from user scores.

```
df_asjc_sm_score_stats = (
    add_avg_median_stddev_skewness_variance(
        df_asjc_sm_scores_delta,
        "sme_score", ["asjc"], "asjc"
    )
    .join(
        add_avg_median_stddev_skewness_variance(
            df_asjc_sm_scores,
            "smn_score", ["asjc"], "asjc"
        ),
        ["asjc"]
    )
)
```

SM Score Results for ASJC Twitter. The resulting per ASJC scores were plotted in Figure 3.18, revealing that SMN scores all stayed below the 1.0 threshold meaning that while there were some noisy individual users, the overall engagement at the category level can be considered non-noisy. Also, find that SME scores identify most

engaged categories. Table 3.4 rank orders the top 10 for easier reference, showing that by a very large margin, General Medicine is the top ASJC from the CORD-19 Twitter corpus with a SME score of ~46K, followed by Infectious Diseases with a SME score of ~11K, then Epidemiology with a SME score of ~10K.

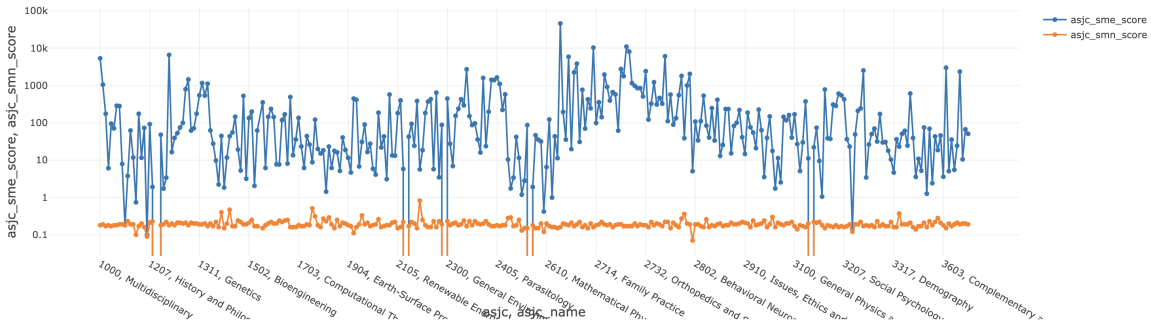

Figure 3.18: Calculated per ASJC SME and SMN scores.

Table 3.4: Top 10 ASJC SME scores.

	SME	**ASJC**	**Name**
1	46,058	2700	General Medicine
2	11,036	2725	Infectious Diseases
3	10,377	2713	Epidemiology
4	8,091	2726	Microbiology (medical)
5	6,613	1300	General Biochemistry, Genetics & Molecular Biology
6	6,077	2739	Public Health, Environmental & Occupational Health
7	5,904	2703	Anesthesiology & Pain Medicine
8	5,350	1000	Multidisciplinary
9	3,849	2706	Critical Care & Intensive Care Medicine
10	3,003	3605	Health Information Management

3.4.3 SM Scores for Incremental Twitter CORD-19

All user posts in the Twitter CORD-19 data were scored over the first 26 weeks of 2020 as described in §3.3.3. The measures themselves were passed to Listing

3.9 as done in §3.4.1. Weekly standard deviation scoring could also have been run as described in §2.5.6 and undertaken in experiment §3.4.2. Instead, this experiment calculated per user incremental scores over $Q1_{2020w}$, $Q2_{2020w}$, and $H1_{2020w}$ as described in §2.5.7. Per user weeks 1 to 13 scores were added together to generate $Q1_w$, weeks 14 to 26 scores were added together to generate $Q2_w$, and $Q1_w$ and $Q2_w$ were added together to generate $H1_w$. This was accomplished through Spark `Window` function `sum` partitioned by `user_id`.

Listing 3.11: Calculate user incremental SME and SMN scoring over $Q1_w$, $Q2_w$, and $H1_w$.

```
df_inc_sm_scores = (
    df_sm_scores_delta
        .filter("year = 2020")
        .filter(col("week").isin(list(range(1,14))))
        .withColumn(
            "q1_inc_sme_score",
            F.sum("sme_score").over(Window.partitionBy("user_id")).cast(
    DecimalType(scale=2))
        )
        .withColumn(
            "q1_inc_smn_score",
            F.sum("smn_score").over(Window.partitionBy("user_id")).cast(
    DecimalType(scale=2))
        )
        .select("year","user_id","q1_inc_sme_score","q1_inc_smn_score")
        .distinct()
        .join(
            df_sm_scores_delta
                .filter("year = 2020")
                .filter(col("week").isin(list(range(14,27))))
                .withColumn(
                    "q2_inc_sme_score",
                    F.sum("sme_score").over(Window.partitionBy("user_id")).cast(
    DecimalType(scale=2))
                )
                .withColumn(
                    "q2_inc_smn_score",
                    F.sum("smn_score").over(Window.partitionBy("user_id")).cast(
    DecimalType(scale=2))
                )
            .select("year","user_id","q2_inc_sme_score","q2_inc_smn_score")
            .distinct(),
            ["year","user_id"]
        )
```

```
        .withColumn(
            "h1_inc_sme_score",
            col("q1_inc_sme_score") + col("q2_inc_sme_score")
        )
        .withColumn(
            "h1_inc_smn_score",
            col("q1_inc_smn_score") + col("q2_inc_smn_score")
        )
)
```

SM Score Results for Incremental Twitter. Experiments to calculate standard weekly incremental user SME and SMN scores resulted in a top semi-annual $H1_{w2020}$ SME score of ~37.6M with the next highest SME score at ~5.5M. The top 5 user SME scores are listed in Table 3.5. Actual user details are not shown in keeping with Twitter rules. The SMN scores were all below the noise threshold of 1 for these top scores, so they are not shown. Interestingly the top scoring incremental user had a $Q1_{w2020}$ SME of only 172 and the third highest scoring incremental user had a $Q1_{w2020}$ SME of 0, meaning both of those had unbalanced engagement on Twitter, heavily weighted in $Q2_{w2020}$.

Table 3.5: Top 5 incremental $H1_{w2020}$ SME scores.

	H1 SME	**Q1 SME**	**Q2 SME**
1	37,611,098	172	37,610,925
2	5,592,897	1,188,381	4,404,515
3	4,946,562	0	4,946,562
4	4,903,144	789,858	4,113,285
4	3,380,492	65,227	3,315,264
5	1,579,842	318,116	1,261,725

Figure 3.19 plots the incremental $H1_{w2020}$ SME user scores (orange) against all available data (blue) over 2020 in buckets of 1000, with sum count shown. Find that there is a good deal of separation between the overall score and the incremental score but that they generally follow a similar trend. This is directly due to standard

incremental only accounting for same week engagement of chains where overall has no constraints. Rolling incremental (next) assesses the level of difference when chains are given more time to grow. Also, while DOI collection was right at week 26 (same as $H1_{w2020}$), the PlumX tweet harvesting for all available data extends up to week 36 which will also contribute to higher SME scores for some DOIs.

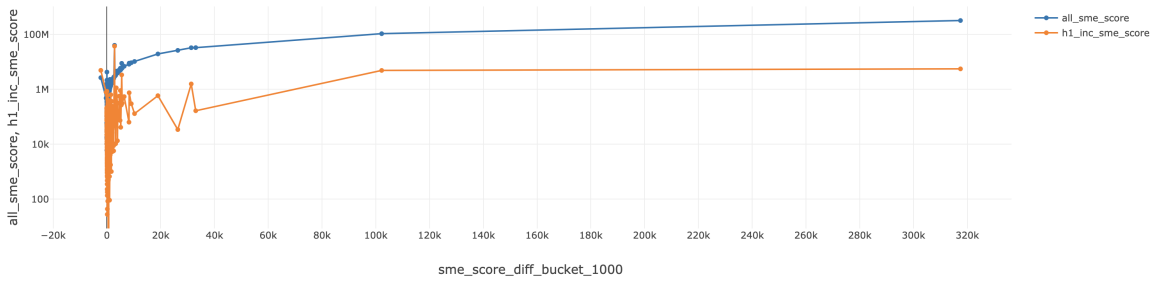

Figure 3.19: Compare $H1_{w2020}$ incremental SME score with all available.

SM Scores for Rolling Incremental Twitter CORD-19. In a separate experiment all user posts in the Twitter CORD-19 data were scored over the first 26 weeks of 2020 for rolling incremental measures generated under §2.5.8. The experiment calculated per user SME and SMN scores over $Q1_w$, $Q2_w$, and $H1_w$ using standard incremental techniques. Additionally, it calculated rolling incremental SME and SMN scores per user over the same intervals for each of the following lagging and blended columns (listing only SME columns):

```
q1_inc_sme_score, q2_inc_sme_score, h1_inc_sme_score, lag1_q1_inc_sme_score,
    lag1_q2_inc_sme_score, lag1_h1_inc_sme_score,  lag2_q1_inc_sme_score,
    lag2_q2_inc_sme_score, lag2_h1_inc_sme_score,  lag3_q1_inc_sme_score,
    lag3_q2_inc_sme_score, lag3_h1_inc_sme_score,  lag4_q1_inc_sme_score,
    lag4_q2_inc_sme_score, lag4_h1_inc_sme_score, blended_lag1_q1_inc_sme_score,
    blended_lag1_q2_inc_sme_score, blended_lag1_h1_inc_sme_score,
    blended_lag2_q1_inc_sme_score, blended_lag2_q2_inc_sme_score,
    blended_lag2_h1_inc_sme_score,  blended_lag3_q1_inc_sme_score,
    blended_lag3_q2_inc_sme_score, blended_lag3_h1_inc_sme_score,
    blended_lag4_q1_inc_sme_score, blended_lag4_q2_inc_sme_score,
    blended_lag4_h1_inc_sme_score.
```

SM Score Results for Rolling Incremental Twitter. The chain size differences are plotted for actual (blue), standard incremental weekly (brown), and weekly rolling lagging 1 to 4 (orange, green, red, purple) series in Figure 3.20. Each point represents the sum of all chain sizes for that measure at that week. This includes true as well as relative roots as do all the chain measures used in SME scoring. The standard week size aggregates are consistently lower (as expected), followed by lag 1 being separated by 50K in aggregate from lag 2. However, lag 2 to lag 4 track more closely. When compared with the actual chain size, see that lag 4 (as expected) tracks closer but still in aggregate lower by about 75K. Also, all chain sizes go up to week 36 which is important when comparing with $H1_{w2020}$ as previously discussed. Blended lagging, the ultimately chosen measure, is not shown in this plot.

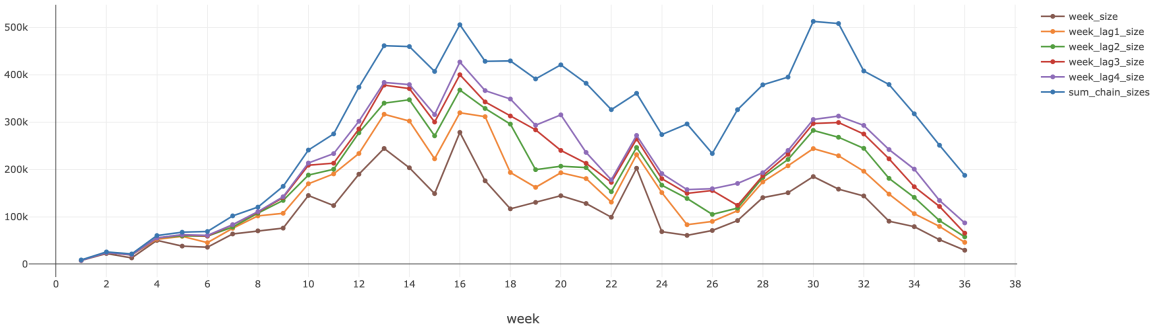

Figure 3.20: Compare rolling weekly chains sizes over 2020.

All of weekly and lagging values are available as measures for the rolling approach. In aggregate there is not much of a difference between any of the median chains sizes when considering which lagging, especially when working inside the outliers. Figure 3.21 filter to summed SME scores for number of posts >1 and <40 for all of standard incremental, the lag scores, and the blended lag scores. The SME score was summed within each series at the given number of posts. With the outliers filtered, all the series were very close in their approximation of the non-incremental

score. This is anticipated because users with lower engagement (versus outliers) will also have fewer chains with lower complexity, so the rolling calculations will not be distinguished. With that understanding, the standard incremental weekly (gold) series is observed to be consistently lower than the other rolling incremental series.

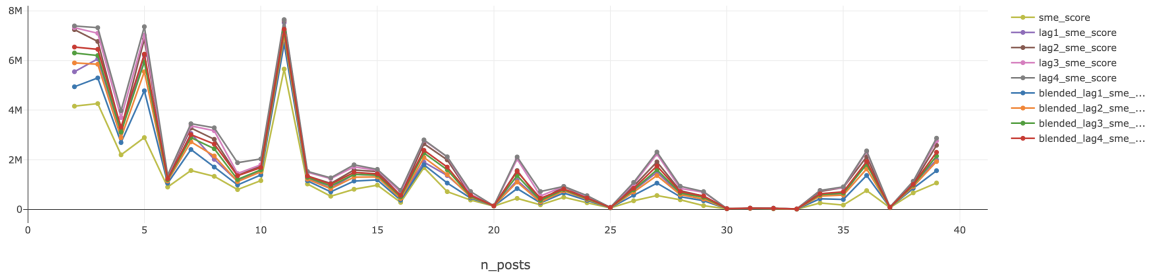

Figure 3.21: Compare SME scores applying each rolling weekly chain measure.

Figure 3.22 filters to just outliers where the SME score difference between incremental and full value is >15K. This plot is rendered in logarithmic scale with the SME difference values plotted on the x-axis in buckets of size 1,000 and the summed SME scores within each bucket for each of the series shown plotted on the y-axis. The two Tables to follow (3.6 and 3.7) use outliers from this plot with the actual user names anonymized to comply with Twitter policies.

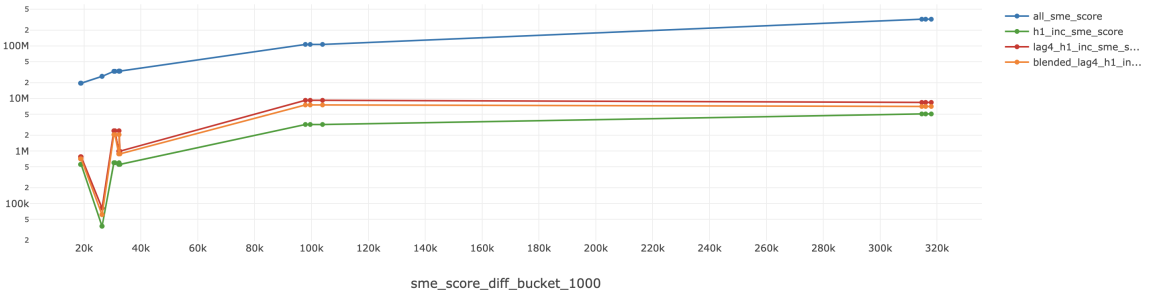

Figure 3.22: Compare SME score outliers for rolling lag4 and blended rolling lag4 measures.

Table 3.6 gives the first 7 weeks of a representative outlier user, referenced

as `user-1`, for standard incremental `w` and rolling lagging incremental `l1` to `l4` and rolling blended lagging incremental `b1` to `b4`. Assessing a single user overcomes the tendency towards the mean which occurs when analyzing aggregates. What comes through more clearly at the user level, e.g. in week 3 (also 11, 34, 35 not shown), is that the current week can have a lower median chain value while each of the lagging weeks can have the same higher median value, uninfluenced by the current week. For week 3 the median chain length is 6 but for `l1` to `l4` (lagging 1 to 4) it is 14 whereas `b1` to `b4` (blended lagging 1 to 4) move closer to the week 3 value at 10 to 12. This begins to demonstrate that `l4` (lag 4) in these columns is more unaffected by the current week it is meant to influence while `b4` (blended lag 4) is affected. Blended becomes a better approach for incorporating each of the lagged valued and the standard value.

Table 3.6: Chain week comparison of single outlier user-1.

	w	**l1**	**l2**	**l3**	**l4**	**b1**	**b2**	**b3**	**b4**
1	12	12	12	12	12	12	12	12	12
2	3	5	5	5	5	4	4	4	4
3	6	14	14	14	14	10	11	12	12
4	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1	1
7	2	2	2	2	2	2	2	2	2

To further solidify the rationale for rolling blended lagging 4, Table 3.7 gives the first 7 weeks of another representative outlier user, referenced as `user-2`, again for standard incremental `w` and rolling lagging incremental `l1` to `l4` and rolling blended lagging incremental `b1` to `b4`. What comes through again more clearly now in weeks 2, 4, 5, 6, 7 (as well as many others not listed) is that the current week has a lower median chain and then each of the lags has the same higher median value. In week 2 the current week median chain length value is 7, while `l1` to `l4` values are all 43,

while `b1` to `b4` values are 25 to 35. This means that `l4` (lag 4) in these columns is more unaffected by the current week while blended is influenced yet able to generate a smoother average over the previous 4 weeks which gives additional weighting to the current week as designed. Blended is confirmed again as the better option to incorporate each of the lagged valued and the standard value.

Table 3.7: Chain week comparison of single outlier user-2.

	w	**l1**	**l2**	**l3**	**l4**	**b1**	**b2**	**b3**	**b4**
1	3	3	3	3	3	3	3	3	3
2	7	43	43	43	43	25	31	34	35
3	24	24	24	24	24	24	24	24	24
4	4	38	59	59	59	21	33	40	43
5	15	39	40	40	40	27	31	33	34
6	19	42	101	128	128	30	54	72	83
7	14	21	40	42	42	17	25	29	31

While there is overall modest differences between use of rolling incremental lagging 4 and rolling incremental blended lagging 4 when considering aggregates, blended lagging 4 smooths better over higher SME scores and therefore is recommended for rolling incremental over weekly standard incremental technique, the rolling incremental lagging technique, and the other blended techniques (for lagging weeks 1 to 3). Weekly standard deviation scoring could also have been run as described in §2.5.6 and undertaken as done in experiment §3.4.2.

Chapter IV.

Discussion

This research laid out a unique methodology to quantify complexity of social engagement upon which other research can build. Within the interests of library sciences, the formalized methodology opens up a new area for future altmetrics development. Discussion follows on the *three research components* identified at the beginning of Chapter 2: (i) for Covid-19, (ii) for information propagation, and (iii) for identifying novel techniques.

i. *Identify and apply distributed data and graph techniques to datasets affiliated with Covid-19. Leverage traditional techniques and existing processed datasets as needed.* The research benefited from the high quality data from ICSR Lab and PlumX, both Scopus citation graphs and the cohesive social media corpus consisting of 8.6M DOI tweets. The experiments were run on Databricks using Apache Spark™ DataFrame APIs over AWS cloud infrastructure. All experiments were designed for distributed execution. The data was stored in the Delta Lake format, The graph processing was run using GraphFrames with graphs edges reaching in the 100M+ range in experiments over the full CORD-19 Scopus 1-hop data.

ii. *Study information propagation properties with the affiliated Covid-19 datasets. Consider propagation of scientific literature, their references, and contextualiz-*

ing social media data. Specially assess graph chains and clustering with time-bound intervals and categorical restrictions. The research design amplified software engineering elements such as remodeling data through data engineering to support many different experiments. The research established quality engagement measures such as number of users reached, number of distinct DOIs, and number and length of generated tweet chains. The design is incremental, allowing cumulative weekly scoring for quarter, semi-annual, and annual without recalculation.

iii. *Potentially identify novel and formalized uses of existing data or graph techniques and apply them in a new way relating to the primary datasets.* The most significant outcomes were the algorithms to generate the Social Media Engagement (SME) and Social Media Noise (SMN) scores (§2.5). The formalized graph modeling of the social data chains (§2.4) for incremental scoring was also an important outcome of this research.

Although the experiments centered around scientific literature, the techniques and algorithms are readily applicable to any social data. Instead of filtering to DOIs, the tweets could be unfiltered or could be filtered on a particular set of hashtags or keywords. Areas for additional study are discussed below.

While per user and per ASJC categorical SM scores were assessed, per DOI scoring was not assessed in the experiments. Per DOI scoring, following the methodology in §2.5, could be the basis for a new class of altmetrics.

Tweet chains for DOIs were short overall (§3.2.5), with a long tail of more highly engaged outliers. While the distribution of chain sizes is most likely the outcome of real world sparse data more so than being specific to scientific literature, additional research could perform more baseline comparisons to validate or invalidate

this assertion. Further, likes and favorites were not part of the quality measures implemented in this research, so follow-on research might evaluate the strengths and weaknesses of incorporating them into the scoring equations.

In the course of analyzing experiment results (§3.4.1), SME and SMN formulas readily identified anomalous accounts with behaviors highly correlated to those of spammers, bots, and self-boosting. The implication is that boosting behavior will throw off altmetrics and potentially graph measures as well if not mitigated. More research to flag anomalous accounts exhibiting noisy and low-quality engagement on social media is a promising future direction.

Figure 3.6 plots a logarithmic scaled component versus `page_rank_id1` of the 572 connected components identified by the graph experiments. The top component has a page rank of 820 and has 24M members, meaning the vast majority of papers are interrelated by at least one reference. The size tails of significantly to the second highest component with a page rank ~5 and only 109 members. This research ultimately focused the most effort around social media engagement and noise (§2.5), but it would be interesting for follow-op research to identify the distinctives of the smaller components and why they are not part of the massive top component.

In the bottom of Figure 3.7 a sample of PlumX published `tweet_total` values for a given DOI were compared with calculated page rank of the same paper from the citation graph on a logarithmic scale. Visually there are some general parallels in movement between page rank and `tweet_total` and could be a promising direction for follow-on research.

Experiments constraining the CORD-19 corpus to only those DOIs found within Scopus, found publish dates generally follow a "gentle" saw-tooth pattern of spiking, trending towards longer engagement on Twitter than the broader CORD-19 papers. This is immediately visually noticeable in comparing the plots in Figures

3.8 and 3.9. One explanation for the longer engagement of Scopus papers would be quality thresholds for which papers make it into Scopus versus the wider CORD-19 corpus. Another explanation is that there were a number of prominent papers relating to Covid-19 which were published by Elsevier holdings such as The Lancet which drove international attention at various periods of time. Additional research would be needed to understand the reasons for the longer engagement by Scopus papers on Twitter.

The alignment of gender authorship splits with earlier publishing trends (§3.1) is anticipated to be the outcome of two factors. First, the CORD-19 corpus comprises papers published during and after historical coronaviruses such as SARS and MERS (1.7). Second, the CORD-19 corpus is contained to more limited subject areas which may reflect proportionally larger male authors. More study would be needed to better understand the differences.

Chapter V.

Conclusions

This research combined existing distributed data and graph techniques with a novel and formalized methodology for measuring and scoring social media engagement. The experiments were run on Twitter data filtered to tweets containing Digital Object Identifiers (DOI) affiliated with the CORD-19 (§1.7) dataset, expanded with Scopus (§1.5) citation graphs and PlumX (§1.6) altmetrics data. The most significant outcomes of the research were two scoring algorithms, Social Media Engagement (SME) and Social Media Noise (SMN), which quantify complexity of engagement, or lack thereof, by a user within social data at any scale or velocity. The guiding intuition was that user accounts are the communicators on social sites, so calculate scores based on users first. Then assign coverage and category specific scores derived from per user scores.

The SME score is calculated based on posts `p`, coverage `c`, reach `r`, and quality `q`. The developed methodology (§2.5) scores the quality of engagement through measures including unique users reached, number of coverage items about which the user posted, and engagement complexity measures based on number and length of tweet chains (§2.4). In calculating scores, the use of quality measures goes beyond page rank's (§1.1) more limited concern with the centrality of a user within the social graph structure. The SME formula design mitigates against contrived user behaviors

which result in low engagement such as tweet blasts in §2.3.4. When both engagement and the number of posts are high, the SME score is going to be high as well. When posts are high but engagement is low, the SME score will be lower and the SMN (noise) score will be higher. The SME formula design does not allow a high number of user posts, irregardless of engagement, to dominate the score.

Equation 5.1 Per user Social Media Engagement (SME) formula.

$$SME_{Xuser} = \left(\ln\left(1 + |P_{Xuser}|\right) * |C_{Xuser}| \right) + \left(|R_{Xuser}| * \sqrt{1 + (\tilde{q}_{Xuser} * \mid Q_{Xuser} \mid)} * \left(\ln\left(\mathbf{e} - 1 + |C_{Xuser}|\right) - \ln\left(\mathbf{e} - 1\right) \right) * \ln\left(1 + \tilde{q}_{Xuser}\right) \right) \tag{5.1}$$

SMN scores are calculated based on the total posts `p` using the same post value applied to the SME formula.

Equation 5.2 Per user Social Media Noise (SMN) score formula.

$$SMN_{Xuser} = \frac{total\_posts_{Xuser}}{\left(1 + SME_{Xuser}\right)} \tag{5.2}$$

SME and SMN formulas offer several benefits to any research involving social media. They can be applied to individual social data providers, can be aggregated over any combination of providers, and can be constrained to only score social posts with coverage items. The established approach is adaptable for categorical and can be fully unbounded up to the limits of the data. The incremental design allows cumulative weekly scoring for quarter, semi-annual, and annual while avoiding any recalculation.

This methodology shows promise to be the baseline for a new class of altmetrics standardized to score quality and complexity of scientific literature engagement within social media.

References

Arjomandi, E. & Cornell, D. G. (1975). Parallel computations in graph theory BT - 16th Annual Symposium on Foundations of Computer Science, SFCS 1975, October 13, 1975 - October 15, 1975. volume 1975-Octob of *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS* (pp. 13–18). Department of Computer Science, University of Toronto, Toronto; ON, Canada: IEEE Computer Society.

Armbrust, M., Bateman, D., Xin, R., & Zaharia, M. (2016). Introduction to Spark 2.0 for database researchers BT - 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD 2016, June 26, 2016 - July 1, 2016. volume 26-June-20 of *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 2193–2194). Databricks, United StatesMIT, CSAIL, United States: Association for Computing Machinery.

Armbrust, M., Das, T., Paranjpye, S., Xin, R., Zhu, S., Ghodsi, A., Yavuz, B., Murthy, M., Torres, J., Sun, L., Boncz, P., Mokhtar, M., Hovell, H. V., Ionescu, A., Luszczak, A., Switakowski, M., Ueshin, T., Li, X., Szafranski, M., Senster, P., & Zaharia, M. (2020). Delta lake: High-performance acid table storage over cloud object stores. *Proc. VLDB Endow.*, 13, 3411–3424.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee,

G., Patterson, D., Rabkin, A., Stoica, I., & Zaharia, M. (2010). A view of cloud computing. *Communications of the ACM*, 53(4), 50–58.

Baas, J., Schotten, M., Plume, A., Côté, G., & Karimi, R. (2020). Scopus as a curated, high-quality bibliometric data source for academic research in quantitative science studies. *Quantitative Science Studies*, 1(1), 377–386.

Brin, S. & Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7), 107–117.

Bruyr, D. L. (1965). *AN INTRODUCTION TO GRAPH THEORY*. PhD thesis, Oklahoma State University.

Dave, A., Jindal, A., Li, E. L., Xin, R., Gonzalez, J., & Zaharia, M. (2016). GraphFrames: An integrated API for mixing graph and relational queries BT - 4th International Workshop on Graph Data Management Experiences and Systems, GRADES 2016, June 24, 2016. volume 24-June-20 of *ACM International Conference Proceeding Series* University of California, Berkeley, United StatesMIT, United StatesUber Technologies, United StatesDatabricks, United StatesMicrosoft, United States: Association for Computing Machinery.

Didegah, F., Bowman, T. D., & Holmberg, K. (2018). On the differences between citations and altmetrics: An investigation of factors driving altmetrics versus citations for finnish articles. *Journal of the Association for Information Science and Technology*, 69(6), 832–843.

Elsevier (2020). Scopus Data Assets [Photograph]. `https://www.elsevier.com/__data/assets/image/0003/482421/content_NumbersImage_V2.png`. Retrieved 2020-05-10.

Farber, M. & Sampath, A. (2019). Determining How Citations Are Used in Citation Contexts. In *23rd International Conference on Theory and Practice of Digital Libraries, TPDL 2019, September 9, 2019 - September 12, 2019*, volume 11799 LNCS (pp. 380–383). Oslo, Norway: Springer Verlag.

Hopcroft, J. & Tarjan, R. (1973). Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6), 372–378.

Islam, M. T., Srirama, S. N., Karunasekera, S., & Buyya, R. (2020). Cost-efficient dynamic scheduling of big data applications in apache spark on cloud. *Journal of Systems and Software*, 162.

Jayabalasingham, B. (2020). "The researcher journey through a gender lens".

Magara, M. B., Ojo, S., & Zuva, T. (2017). Toward altmetric-driven research-paper recommender system framework. In *13th International Conference on Signal-Image Technology and Internet-Based Systems, SITIS 2017, December 4, 2017 - December 7, 2017*, volume 2018-Janua (pp. 63–68). Jaipur, India: Institute of Electrical and Electronics Engineers Inc.

Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10 (pp. 135–146). New York, NY, USA: Association for Computing Machinery.

Massucci, F. A. & Docampo, D. (2019). Measuring the academic reputation through citation networks via PageRank. *Journal of Informetrics*, 13(1), 185–201.

Montoya, F. G., Alcayde, A., Banos, R., & Manzano-Agugliaro, F. (2018). A

fast method for identifying worldwide scientific collaborations using the Scopus database. *Telematics and Informatics*, 35(1), 168–185.

Nanumyan, V., Gote, C., & Schweitzer, F. (2020). A multi-layer network approach to modelling authorship influence on citation dynamics in physics journals.

Priem, J., Taraborelli, D., Groth, P., & Neylon, C. (2010a). Altmetrics: A manifesto, 26 October 2010. `http://altmetrics.org/manifesto/`. Retrieved 2020-05-10.

Priem, J., Taraborelli, D., Groth, P., & Neylon, C. (2010b). Four Ways to Measure Impact [Photograph]. `http://altmetrics.org/wp-content/uploads/2010/10/four-ways-to-measure-impact-copy.png`. Retrieved 2020-05-10.

Raghavan, U. N., Albert, R., & Kumara, S. (2007). Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3).

Skulimowski, M. (2019). A Semantic Representation of the Citation Structure. In *13th International Conference on Metadata and Semantic Research, MTSR 2019, October 28, 2019 - October 31, 2019*, volume 1057 CCIS (pp. 298–303). Rome, Italy: Springer.

Swoger, B. J. M. (2013). PlumX.(plumanalytics.com)(Website overview). *Library Journal*, 138(13), 124.

Wang, L. L., Lo, K., Chandrasekhar, Y., Reas, R., Yang, J., Burdick, D., Eide, D., Funk, K., Katsis, Y., Kinney, R., Li, Y., Liu, Z., Merrill, W., Mooney, P., Murdick, D., Rishi, D., Sheehan, J., Shen, Z., Stilson, B., Wade, A., Wang, K., Wang, N. X. R., Wilhelm, C., Xie, B., Raymond, D., Weld, D. S., Etzioni, O., & Kohlmeier, S. (2020). CORD-19: The COVID-19 Open Research Dataset.

Yedidia, J. S., Freeman, W. T., & Weiss, Y. (2003). Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium*, 8, 236–239.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10 (pp.1̃0). USA: USENIX Association.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., & Stoica, I. (2016). Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11), 56–65.

Zhao, H., Xu, X., Song, Y., Lee, D. L., Chen, Z., & Gao, H. (2019). Ranking users in social networks with motif-based pagerank. *IEEE Transactions on Knowledge and Data Engineering*, (pp. 1–1).