Growing a FLOWER: Building a Diagram Unifying Flow and ER Notation for Data Science

Carlos Ordonez
University of Houston
Houston, USA

Robin Varghese
University of Houston
Houston, USA

Nguyen Phan
University of Houston
Houston, USA

Wojciech Macyna
Wroclaw University of Technology
Wroclaw, Poland

ABSTRACT

An ER diagram is a fundamental visual abstraction to design a database. Modern ER notation has evolved with UML symbols to represent both entities (logical level) and relational tables (physical level). On the other hand, flow diagrams (flowcharts, process flow) remain an important mechanism to visualize the main steps of a data processing pipeline. However, in modern data science projects there is a significant fraction of data that does not come from databases or data that is exported outside the database system, being processed by Python code, without any data model whatsoever. In this paper, we present a novel diagram which is built from source code and its associated browser-based GUI for collaborating on data integration and data preprocessing, mixing diverse data sources and diverse programming languages (mainly Python and SQL). Specifically, our targets are data integration, data cleaning and data transformation, which are needed to derive data sets that can be used as input for a machine learning model. We present a couple of target applications and a preliminary GUI, which partially automates diagram creation. We show our diagram has promise understanding, extending and reusing both data preparation source code and data sets.

KEYWORDS

Diagram, Python, ER, database model, SQL, source code

ACM Reference Format:
Carlos Ordonez, Robin Varghese, Nguyen Phan, and Wojciech Macyna. 2024. Growing a FLOWER: Building a Diagram Unifying Flow and ER Notation for Data Science. In *Workshop on Human-In-the-Loop Data Analytics (HILDA 24), June 14, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3665939.3665958

1 INTRODUCTION

In most data science projects most code development time is spent integrating and pre-processing data because data sets come from diverse sources, they have different structure, they come in different file formats and they are not integrated. Hence data data scientists need to collect, integrate, merge, aggregate, clean, and transform data before they can perform analysis. This is achieved by using powerful libraries (mainly from Python these days) or query languages (SQL, SPARQL), but the programming effort remains significant. Such data transformations create many intermediate files, tables, even matrices, in a disorganized manner. This problem becomes more difficult when pre-processing big data beyond alphanumeric data sets (i.e. traditional statistical or data mining analysis). Today the AI analytic challenge generally involves a combination of plain data files, databases, text, images and even video.

ER diagrams have a proven track record in modeling, designing data structure and relationships, extending their use beyond databases. Their strengths lie in their key coherence, generality, flexibility, and intuitive visual representation. On the other hand, despite being older, flow diagrams remain the primary mechanism for visualizing major components or main processing steps of a software system, although they are less useful for understanding complex algorithms. In a traditional database project, an ER diagram is iteratively refined, then this ER diagram is mapped to relational tables, which are normalized for efficient transaction processing, in general with SQL. Therefore, the database structure is designed first and then tables can populated with data. In this paper, we reverse this process, where the input is source code and files with non-relational data and the goal is create a data-centric diagram, combining flow and ER elements. While we are not the first to propose diagrams for understanding data pre-processing, to the best of our knowledge, we are the first to extend modern UML diagram notation for ER with a minimal change (instead of a more disruptive change): an arrow symbol on top of the relationship.

Previous work [11] proposed a framework (analytic component architecture) and a method (steps) to generate diagrams from source working on databases and plain big data files (logs, csv, SQL files). In closely related work on ER diagrams, other researchers have proposed entities for workflows [6], conceptual modeling for data processing [4], automatically drawing flow diagrams [3], [16], and learning data transformations [13], [12]. On the other hand, numerous methods and tools have been proposed to enhance the collaboration between users, data, and processes in data science projects. Such tools can improve the understanding of code executions [17], simplify the annotation process [8], visualize data transformation [14], [5], and accelerate the lifecycle process [2], [7].

HILDA 24 , June 14, 2024, Santiago, AA, Chile
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0693-6/24/06…$15.00
https://doi.org/10.1145/3665939.3665958

1.1 Example

Consider an AI project aiming to predict which products are likely to increase in sales based on historical sales data, web interaction, product features, customer demographics and customer reviews. This is, arguably, the most common AI project in the retail industry. In this case data sources are relational databases, web server logs, text written in natural language and even images. Data scientists start gathering and integrating diverse files with SQL or Python, helped by relational and NoSQL database systems. It is reasonable to assume most processing is happening in Python programs, which in turn connect to diverse systems, which can send queries both in SQL and NoSQL, which can process files with Py libraries (e.g. pytorch, Pandas) and which can read files extracting information in common formats (CSV, JSON). Then a team of analysts starts computing "merge" (join) operations between sales tables and related sales information (customer, product based on keys when feasible), computing sales aggregations by product id and some appropriate time unit (weeks, months). Then depending on model requirements, the analyst performs feature engineering (creates variables in a statistical sense, create target/prediction variables) to perform regression, time series analysis or classification. During the project it will be necessary to apply diverse data transformations to deal with missing values, outliers, spikes, numeric singularities (undefined matrix factorizations, unstable gradient) and so on. Needless to say, the Python code behind doing all this data manipulation can reach hundreds and even thousands of lines. maintained by multiple people. Considering this challenging data analysis environment, our diagram has the following major goals: (1) identifying keys embedded in the code which are used to access each record, row or vector or which key attributes are used to merge (join, link) files with diverse information. Since files may not come databases we assume there exists at least a subscript to access each row, each line, each section, each page and so on, captured by some Python variable. We should emphasize that an ER diagram does not make sense without keys because are the mechanism to identify instances (objects, records) and link entities (via foreign keys). (2) identifying non-key attributes (features, variables) which depend on the previously identified keys. In Section 4 we showcase this representative project.

1.2 Contributions

Our contributions can be summarized as follows, comparing with closely related work. The ideas in this paper have roots in an extended ER diagram to understand complex SQL queries doing data pre-processing in a data mining project, working exclusively with relational tables inside a DBMS [10] (i.e., the formerly popular data warehouse). Intuitively, the result of each query is a "transformation" entity and raw input data represents 'source" entities. Given the migration of analytic processing outside the DBMS realm pushed by data science, we were motivated to revisit this idea in a much broader context [9, 15]. This paper represents a major technical improvement over [9] and [15], where an initial prototype was proposed. The prototype [9] introduced a primitive Python parser to identify variables in Python code using the Pandas library, referring to data frames and matrices and proposed introducing a flow symbol. The integration with the Python ecosystem is important given the significant research in the field of data science today. In [15], we explored how our hybrid diagram (FLOWER=FLOW+ER) can be exploited in an IT organization following Business Administration practices, to assist data scientists in pre-processing raw data. In this paper, we provide a formal statement of the diagram building problem and we refine and extend the FLOWER diagram to represent data flow more accurately, considering relationship cadinality, flow direction and we added identifying processing function (from source code). In addition, we present two case studies on Natural Language Processing (NLP) and biomedical nerve signals, where we show we can partially automate identification of entities and their keys in analytic data objects like data frames, tables and matrices. From a practical perspective, we now offer an interactive GUI designed to facilitate collaboration among data scientists. This GUI enables team members to share insights, update, and refine programs in a shared cloud environment.

2 BACKGROUND

2.1 Definitions

Our proposed diagram is a minimal, but useful, extension of an entity-relationship (ER) diagram. In an ER diagram the basic unit of information is an entity, where each element (also called instance) can be identified by a key. Entities are linked with each other via relationships, which get captured via foreign keys. Relationships have cardinalities on each end, based on the participating entities (1:1, 1:N, M:N). We treat entities as objects (with physical storage implications), abusing a little traditional conceptual modeling, which treats entities as language concepts.

Given the evolution of ER and Object-Oriented Design (OOD) tools, we adhere to modern UML notation which is more concise and more elegant than old ER notation. UML notation scales well to many entities, having many attributes producing a more elegant diagram with less elements and lines than old ER notation where each attribute is placed in an ellipse. In this notation, entities are depicted by rectangles, and relationships are illustrated by lines, with crowfeet on the "many" side (also called :N side). It is assumed that each entity has an identifying set of attributes (e.g. a table key, a matrix entry subscripts), such as a primary key, a file name, a variable name, subscript(s) or an object id. We assume that, in general, each entity ends up associated with one data file highlighting the importance of specific data objects stored by the analyst. For now, considering all potential transformations in main memory is out of scope because Python is a dynamically typed language and the transformations may be scattered in the source code.

The ER diagram can be understood as a graph. Let $G = (E, R)$ be a directed graph where E represents entities (as vertices) and R represents relationships between entities (as edges) of the graph. G must be connected, which in practical terms means any file can be integrated with any other file with a chain of merge (join) operations.

2.2 Cross Industry Standard Process for Data Mining (CRISP-DM)

CRISP-DM (Cross-Industry Standard Process for Data Mining) is a widely used software development standard born during the data mining era for guiding and managing the process of extracting useful knowledge from data. CRISP-DM remains in wide use today in big data analytics. This is a robust framework that consists of six main stages (phases):

(1) **Business Understanding:** In this initial stage, the project objectives and requirements are defined from a business perspective. Understanding the business goals and criteria for success is crucial.
(2) **Data Understanding:** Here, data collection and exploration take place. The data is gathered, its quality assessed, and initial insights are gained to inform subsequent steps.
(3) **Data Preparation:** Data preprocessing occurs in this stage. Tasks such as cleaning, transforming, and integrating data are undertaken to ensure it's suitable for analysis.
(4) **Modeling:** This stage involves the selection and application of various modeling techniques to build predictive or descriptive models based on the prepared data.
(5) **Evaluation:** Models are evaluated based on predefined criteria to determine their effectiveness in meeting the business objectives set in the first stage.
(6) **Deployment:** Finally, successful models are deployed into production, and plans are made for monitoring and maintaining their performance over time.

Our diagram main goal is to understand data integration and data preparation source code from a data-centric perspective. We provide a framework that assists users in better understanding data preprocessing in machine learning and AI projects. Moreover, we broaden the perspective to a "data-centric" approach, which offers intuitive visualization to project managers involved in challenging Big Data projects.

3 AUTOMATICALLY BUILDING A DIAGRAM UNIFYING DATA FLOW AND ER: FLOWER

3.1 Representing Flow with an Arrow

To capture data flow in visual form, we propose to extend ER relationships with an arrow indicating the direction of data flow. To bind flow to source code, the arrow is labeled with a function name or with a mathematical expression parsed and isolated (scoped) from the source code, instead of traditional verbs (e.g., "sold by," "works for," "is part of," etc.), as used in traditional ER diagrams. The arrow direction is fundamental as it signifies input and output in functions or expressions in a data science program (most commonly Python these days), originating from "source" entities and moving towards "destination" entities computing data transformation, data derivation or data integration. Then the challenges are identifying entities, understanding which variables in the program can act as keys and then linking entities via such keys.

We believe our arrow symbol is a minor, yet powerful change, that facilitates navigation through interconnected data objects (files, tables, data frames, matrices), providing a data-centric processing flow (instead of function-centric).

3.2 Problem Statement

In this section we atempt to formalize our problem and its solution, based on database systems principles, extended with programming languages concepts.

Goal: The goal is to identify n entities embedded in source code, where each entity is either a source (raw data) entity or a transformation (used as a generic term to capture data integration, data derivation, data cleaning, data aggregation).

Input: The input is source code written in some programming language (e.g. Python) and data files (perhaps exported from a SQL or NoSQL system). We assume external data sources like SQL databases, key-value stores get exported and converted to a file format that can read in Python (e.g. the most common today are CSV and JSON). The input source code is parsed to identify variable names and function names with standard compiler techniques (context-free grammars, parser, scoping).

Solution: In the initial solution explored in this paper, we identify entities and their keys in the source code, helped by human interaction. Keys are variables which can be used to uniquely identify one piece of information. Therefore, a subscript or position can act as a key. The remaining attributes in an entity are functionally dependent on the key. Looking at data values to discover functional dependendies is left as future work because most dependencies can be identified from source code and looking at data values requires executing (running) code.

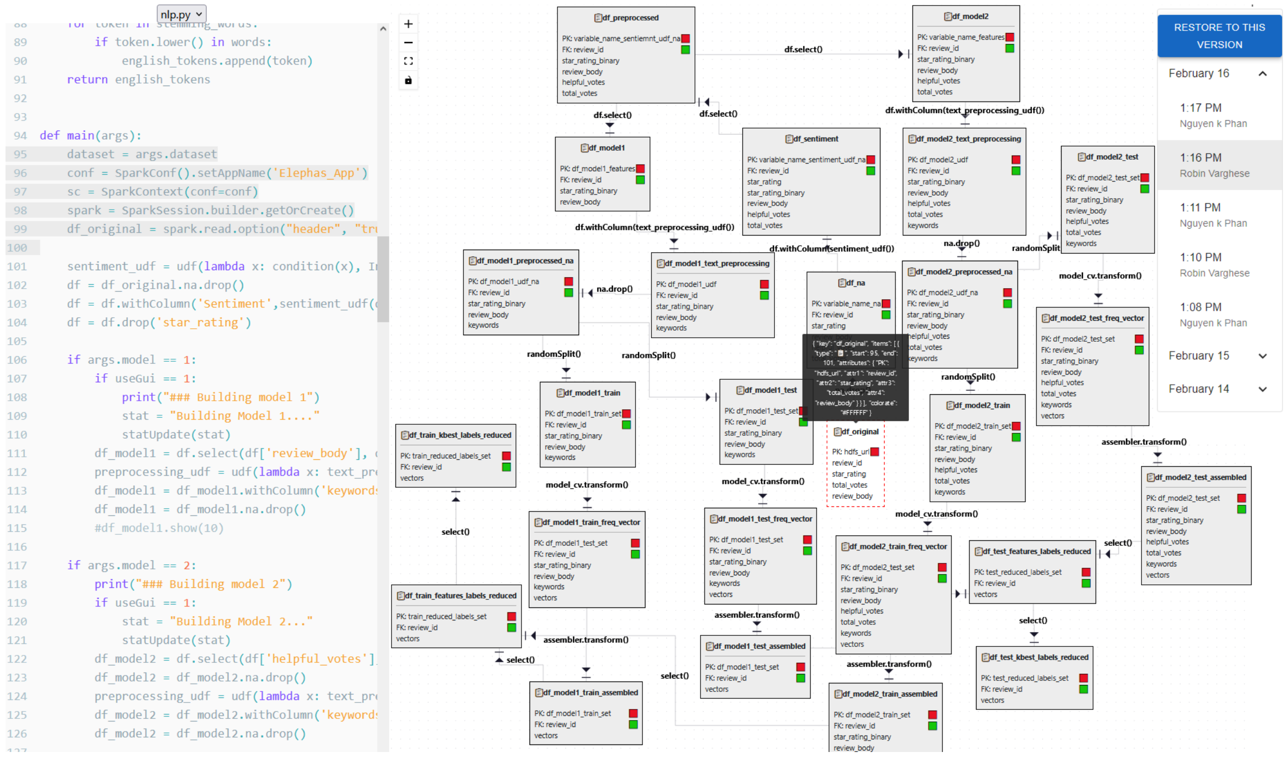
Output: we propose to represent FLOWER metadata with JSON object-oriented notation, given its well-defined syntax, flexibility to capture diverse structure, text-based storage and interoperability. Entities are stored in a list of {entity,attribute,keyflag} triples and relationships are {entity,key,entity} triples. The key is the "glue" that links to entities together, where key can either be primnary key on one entity and foreign key on the other entity (1:N), or primary key and foreign keys on both (1:1).

Examples: Each element in a Python list is functionally dependent on its position (automatically determined). Each row in a matrix, each record in a data frame, each coordinate in a vector, can be uniquely identified with one subscript (e.g. $[i]$). It is straightforward all entries $a_{i1}, a_{i2}, ..$ in a matrix row are functionally dependent on i: $i \rightarrow a_{i1}, i \rightarrow a_{i2}, \ldots$. Each entry in a sparse matrix, stored in "triple" (i, j, v) form, can be uniquely identified with the row/column i, j pair (noice this is a traditional database storage in SQL): $i, j \rightarrow v$. Each line in a text file can be uniquely identified with a line number, like a code development editor. We acknowledge our keys come from code, not from a design process. Therefore, they abuse established database concepts.

3.3 Building Diagrams

Our solution generates a preliminary diagram as follows. This diagram can be polished and manually customized by the data scientist. First, we show a preliminary example of the generated diagram using the corresponding JSON files in Fig 3.

(1) First, we incorporate the Human In The Loop (HIL) aspect to manually annotate major transformations in the code, providing initial starting points for diagram generation. A purely automated approach based on every data transformation in a Python script, would produce a convoluted diagram with excessive entities. With some initial insights provided by the user, annotated by "#start - transformation()" and "#end - transformation()" comments in the Python script, we


nlp.py
df_preprocessed
df_model2
df.select()
df.withColumn(text_preprocessing_udf())
df_model1
df_sentiment
df_model2_text_preprocessing
df_model2_test
df.withColumn(text_preprocessing_udf())
df.withColumn(sentiment_udf())
na.drop()
randomSplit()
df_model1_preprocessed_na
df_model1_text_preprocessing
df_na
df_model2_preprocessed_na
model_cv.transform()
df_model2_test_freq_vector
randomSplit()
df_model1_train
df_model1_test
df_original
df_model2_train
df_train_kbest_labels_reduced
select()
assembler.transform()
df_model1_train_freq_vector
df_model1_test_freq_vector
df_model2_train_freq_vector
df_test_features_labels_reduced
df_model2_test_assembled
df_train_features_labels_reduced
df_model1_test_assembled
df_test_kbest_labels_reduced
df_model1_train_assembled
df_model2_train_assembled
RESTORE TO THIS VERSION
February 16
1:17 PM
Nguyen k Phan
1:16 PM
Robin Varghese
1:11 PM
Nguyen k Phan
1:10 PM
Robin Varghese
1:08 PM
Nguyen k Phan
February 15
February 14

```
    for token in stemming_words:
        if token.lower() in words:
            english_tokens.append(token)
    return english_tokens


def main(args):
    dataset = args.dataset
    conf = SparkConf().setAppName('Elephas_App')
    sc = SparkContext(conf=conf)
    spark = SparkSession.builder.getOrCreate()
    df_original = spark.read.option("header", "tru

    sentiment_udf = udf(lambda x: condition(x), Ir
    df = df_original.na.drop()
    df = df.withColumn('Sentiment',sentiment_udf(c
    df = df.drop('star_rating')

    if args.model == 1:
        if useGui == 1:
            print("### Building model 1")
            stat = "Building Model 1...."
            statUpdate(stat)
        df_model1 = df.select(df['review_body'], c
        preprocessing_udf = udf(lambda x: text_pre
        df_model1 = df_model1.withColumn('keywords
        df_model1 = df_model1.na.drop()
        #df_model1.show(10)

    if args.model == 2:
        print("### Building model 2")
        if useGui == 1:
            stat = "Building Model 2..."
            statUpdate(stat)
        df_model2 = df.select(df['helpful_votes']
        df_model2 = df_model2.na.drop()
        preprocessing_udf = udf(lambda x: text_pre
        df_model2 = df_model2.withColumn('keywords
        df_model2 = df_model2.na.drop()
```

Figure 1: FLOWER diagram for NLP pre-processing (dataframe level data unit). Highlighting code, data flow visualization through FLOWER, and team collaboration of each user's customized version history.

can have an overall idea of the major transformation for a code block.

(2) Using the Python built-in standard libraries `tokenize` and `ast`, we are able to extract the annotated code blocks into a structured view of the code. Our proposed preliminary approach identifies data-preprocessing transformations by targeting the basic dataframe manipulation functions in `pyspark` like `select()`, `drop()` and `transform()` for the NLP case study and functions in numpy like `dot()`, `append()`, `transpose()` for the biomedical case study. Using these identify fundamental transformations, we can automatically provide suggestions expanding the initial proposed major transformation. In our interactive GUI, a user will be able to update the diagram by editing the corresponding json files.

(3) Next, we make an initial pass through the input Python program determining whether the basic data unit for interpretation (entity), should be represented as files (higher level of abstraction) or dataframe variables (lower level of abstraction). Our proposed approach uses typical semantics in Python where we can identify whether the computation involves writing to a file (to_csv()), or is an in-RAM process and provide a recommended representation in the diagram. Furthermore, we differentiate this in entity notation in the diagram by a folder icon for files Figure. 3, and a clipboard icon for dataframes Figure. 1.

Metadata for our FLOWER diagram is stored as follows. There are two JSON files, where the first file contains the entities (attributes, keys) and the second one contains relationships (connecting keys together). In the data transformation module, we define the transformation type and create new transformed entities. Data scientists may perform several transformations discussed above in the source code that generates a temporary entity. In the case of "Merge", the entity structure may change but the attribute values remain the same, and the "Aggregation" may use one or more grouping attributes along with or without aggregations (sum, count, avg). In general, aggregations will return numbers, but using only "Group by" will return the attribute values as their types. Mathematical transformations will mostly return derived attributes. Now, the new transformed entities are linked with the original entities using an arrow. After each valid transformation step, we can store the newly generated ER-Flow diagram in JSON files. This ER-Flow diagram can help data scientists to have data-oriented view of the program, navigate source code, reuse functions, and avoid creating redundant data sets.

These are the main steps towards creating a data set for ML analysis:

(1) **Data Source Entities:** Identified via a primary key. Using a filename as a primary key is valid because filenames are unique identifiers for files within a folder in the file system. This ensures that each record associated with a particular filename remains distinct and easily retrievable.

(2) **Data Transformation Entities:** Our case studies examine applications in AI. AI is full of data transformations and often inherit attributes from the data source.
(3) **Relationships:** We link two entities E_i, E_j i with relationships and are described by the Python functions that perform the transformations.
(4) **Drawing Diagram:** Given the Data Source and Data Transformation entities and their respective relationships, we can automatically generate our FLOWER diagram (hybrid mixing data flow and ER=FLOW+ER).
(5) **Finalized Data Set:** AI/ML data sets are often written as one final data set and fed to iterative training. We offer two case studies involving one or multiple finalized data sets.

Human In The Loop: While certain steps in the process can be automated, some substeps require human intervention. This is particularly true in Natural Language Processing (NLP) and biomedical signal processing, where the selection of specific features often determines the desired outcome of the target variable. However, the context of the original data sources frequently becomes obscured or entirely lost after several layers of pre-processing and transformation, reducing the information to mere numerical matrices. To address this issue, we aim to demonstrate the value of interactive visualizations of data transformations. Our proposed FLOWER diagram seeks to restore context to the transformed data, providing valuable insights in the process. Moreover, we utilize modern web application architecture on cloud platforms to offer interactive functionalities for users, enhancing the overall experience and facilitating deeper data analytics and understanding.

3.4 Diagram Symbols

Our entity concept is broad: entities can represent a file, a matrix, a relational (SQL) table, or a dataframe (Python, R). A file can contain an image, a document or data records (i.e. CSV or JSON). From a programming language perspective, objects in main memory can be atomic data types (integers, reals, string, dates), or composite (lists, multidimensional arrays and data frames). We want to stress our data scope goes beyond previous database-centric approaches focusing only on records with alphanumeric attributes (SQL tables, plain strongly-structured CSV files).

Entities are classified as source entities, representing raw data and Data Transformation (integration, cleaning, derivation) entities being the output of some tool or program written in a typical data sience programming language (Python, R, SQL).

In our generalized diagram entities represent a set of data elements. In turn, data elements can be records, SQL rows, image pixels, text keywords, video frames. Moreover, data elements can be aggregated bottom up to obtain more abstract entities.

Furthermore, we propose a tweak to the usual UML notation used in ER diagrams. We want to add in function calls that lead to significant changes, next to the usual notation that show relationships between entities. This idea is what led us to create the FLOWER diagrams. By visualizaing data flow and transformations, we make it easier to see not just how data is structured, but also how it's being changed as it moves through processing pipelines. This approach makes ER diagrams not just a static snapshot of data relationships but also a map showing data in motion, providing a clearer picture/interpretability of what's happening under the hood.

3.5 Data Integration and Data Transformation

We focus on representing data integration and transformation for analytics, including machine learning, graphs, and even text files (documents). However, our proposed diagram does not consider the "analytic output" such as the parameters of the ML model, model accuracy, language embedding, IR metrics like precision/recall, graph metrics. We consider simple filter transformations as an implicit property of entities and providing representation in the diagram hinders data flow interpretability as it will convolute with excessive entities. Similar to the σ operator in relational algebra, the result is simply reduced to a fewer number of rows or the source entity.

Furthermore, we propose these major categories of data transformations:

(1) Merge, which splices (joins) multiple entities. We can think of it as a generalized relational join operator ($\bowtie$). We assume Merge is a generic operator to integrate diverse data sources.
(2) Aggregation, which partitions data elements and computes some aggregate function.
(3) Mathematical, which represent derived attributes coming from a combination of functions and value-level operators (e.g. equations, arithmetic expression, nested function calls).

3.6 Collaboration: Cloud Environment

To enhance the usability of our proposed FLOWER diagram, we plan to enable collaborative viewing and editing in a cloud environment using the *socketio* networking library. This setup will allow data scientists to interact with the diagram simultaneously, promoting teamwork and knowledge sharing. Using a peer-to-peer approach with *socketio*, we can synchronize the application's state across users without a complex server-client infrastructure, ensuring all collaborators work with the latest diagram version in real-time. Additionally, a collaborative history list view will track each modification by date and user, improving transparency and teamwork. This cloud-based setup is designed to streamline data pre-processing and team communication.

3.7 Tool: GUI

To make our FLOWER diagram more accessible and user-friendly, we have developed an interactive web application that allows users to generate and manipulate the diagram with ease. The web application consists of a split-pane layout, with the interactive UML diagram displayed on the middle panel and a code preview on the left panel (figure 1). Users can select an entity or relationship link in the diagram, which will highlight the corresponding part in the code or show a preview of the associated data file on the left panel. This interactive feature helps users navigate between the visual representation and the underlying code and data, facilitating a deeper understanding of the data pre-processing process. The right panel features a collaborative history list view, detailing the diagram's modifications by various team members, organized by date. While currently a mockup, the list view is a chronological interface that offers snapshots of the UML diagram at various stages of its development. We are actively working to evolve this into a fully functional feature with visual previews and a robust version control mechanism that allows reverting the diagram to any specific timestamp.

Our framework is built using *Node.js*, and *React.js*, a front-end library for building user interfaces with components. The interactivity within the web application is powered by *ReactFlow*, a comprehensive library for creating draggable, zoomable, and customizable node-based graphs. We've also integrated the *react-syntax-highlighter* library to improve the readability of Python code snippets.

4 CASE STUDIES

We feature two AI/ML examples with FLOWER diagramming. For credibility, we opted for these case studies because they have publicly available source code (APPENDIX) and corresponding published research. In the presented case studies, the source entities are colored white and the transformed entities are colored grey. The data flow of the transformed entities are linked with an arrows and the corresponding Python functions performing the transformations.

4.1 Diagram Qualitative Properties

To evaluate the visual quality and effectiveness of the FLOWER diagram, we list a set of properties commonly used in visualization research that consider various aspects of the diagram's design and presentation. We aim to ensure that the diagram is clear, readable, and informative for its intended audience.

(1) **Visual Clarity:** The diagram should prioritize legibility and readability by using consistent font sizes, appropriate edge line widths, and sufficient background color contrast. This ensures that the information presented is easily discernible and visually appealing.
(2) **Layout and Organization:** The placement of entities should be logical and balanced, minimizing overlaps and ensuring a clear flow of relationships. Relationship lines should be routed cleanly, minimizing crossings, and related entities should be grouped together with adequate spacing to avoid clutter and improve visual coherence.
(3) **Information Density:** The ratio of entities to relationships should be balanced to convey necessary information without overwhelming the viewer. The diagram should be concise and focused, avoiding the inclusion of extraneous or redundant information, while maintaining the ability to effectively communicate the key aspects of the data model.

4.2 Natural Language Processing

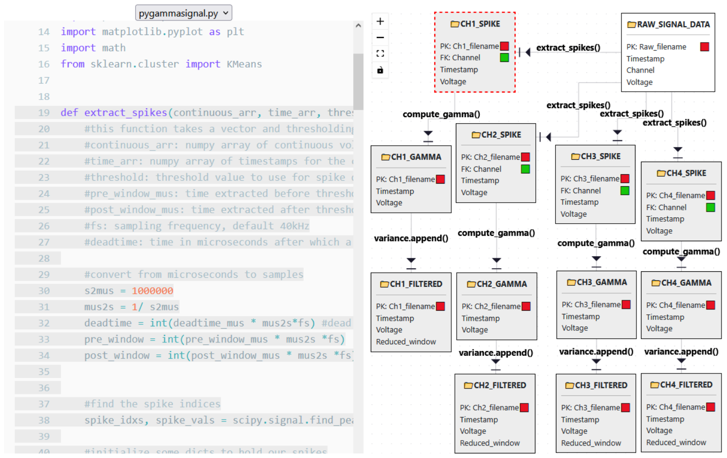
We provide a typical data preparation pipeline for NLP applied to analyzing Amazon reviews. Pre-processing data is a crucial step towards providing stable mode performance especially in predictive deep neural network models. The main steps are as follows:

- Tokenizing the text to break down the review into manageable pieces.
- Removing non-English words to maintain language consistency.
- Eliminating stop words that add little to no value to the analysis.
- Applying lemmatization to condense words to their base or dictionary form.
- Stripping punctuation to reduce noise and focus on textual content.
- Conducting Part-Of-Speech (POS) tagging to understand grammatical structures.
- Selecting relevant features that contribute most significantly to model performance.

In general, this structured approach strives to ensure text data is prepared for the further complexities of NLP during model training. More importantly using FLOWER, this process becomes far more understandable and intuitive. Additionally, a team is able to collaborate together simultaneously and have deeper insights of the overall data processing beginning from the data source, using an interactive webapp. Users are able to login securely using private credentials and have their own local representation of the production data pipeline. We provide an example in Figure 1 how an NLP pipeline is derived from a python script, producing our intuitive FLOWER diagram, with interactivity to the code and the json representation of the entity for possible modifications.

As demonstrated in Figure 1, we believe FLOWER successfully maintains visual clarity, clean layout and intituitive organization when handling 26 entities. By employing consistent fonts, colors, and spacing, along with adjustable and scalable components, FLOWER proviers intuitive visualization that remains effective even with high information density.

4.3 Biomedical Signals



```
import matplotlib.pyplot as plt
import math
from sklearn.cluster import KMeans


def extract_spikes(continuous_arr, time_arr, thres
    #this function takes a vector and thresholdin
    #continuous_arr: numpy array of continuous vo
    #time_arr: numpy array of timestamps for the 
    #threshold: threshold value to use for spike 
    #pre_window_mus: time extracted before thresh
    #post_window_mus: time extracted after thresh
    #fs: sampling frequency, default 40kHz
    #deadtime: time in microseconds after which a

    #convert from microseconds to samples
    s2mus = 1000000
    mus2s = 1/ s2mus
    deadtime = int(deadtime_mus * mus2s*fs) #dead
    pre_window = int(pre_window_mus * mus2s *fs)
    post_window = int(post_window_mus * mus2s *fs


    #find the spike indices
    spike_idxs, spike_vals = scipy.signal.find_pea

    #initialize some dicts to hold our spikes
```

Figure 2: Interactive FLOWER webapp for biomedical signal pre-processing.

In this section we present a practical example illustrating how to pre-process a set of biomedical signal data for clustering [1]. This paper delves into the challenges and advancements in identifying similar patterns in physiological nerve signals collected from micro electrical sensors in animal organs. The primary challenge is discerning these patterns, which appear as spikes within millisecond time-windows amidst high-dimensional data sets, especially with the interference of background electrical noise.

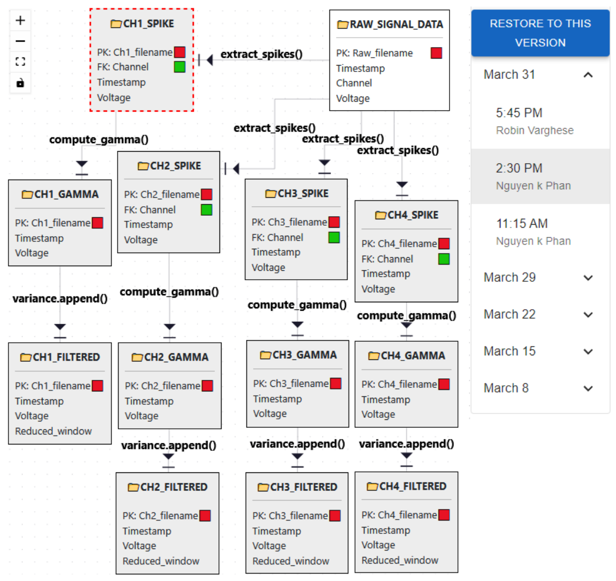


Figure 3: FLOWER diagram for biomedical signal data pre-processing (file level data unit).

- **Objective:** The main aim is to detect similar patterns in high throughput nerve signal data.
- **Previous Systems:** Earlier methods combined PCA (Principal Component Analysis) and K-means clustering but were slow and required multiple tools.
- **Proposed System:** The paper introduces an integrated system that combines signal filtering, feature engineering, and multidimensional data summarization for a more effective integration of PCA and K-means clustering.
- **Applications:** The ultimate goal is to associate signal patterns with specific physiological functions, potentially leading to innovative medical treatments via nerve stimulation.
- **Contribution:** The research offers an efficient method to analyze multiple continuous signals over time, detecting signal patterns across them using machine learning techniques.
- **Implementation:** The entire system is implemented in Python, a popular language in Big Data and Data Science.

The transformations applied to the source (raw) signals are as follows:

1. **Computing Correlations Between Channels:**
   - Correlations are computed based on split raw data sets ({D1, D2, ..., DM}).
   - An incremental algorithm computes the correlation matrix using a summarization matrix formed by multiplying the combined raw data set with its transpose (e.g. $D_1 D_1^T$, ..., $D_M D_M^T$).
2. **Filter Noise And Detect Spikes:** Noise is filtered and spikes are detected from the correlated channels.
3. **Reduce Dimensions to $\hat{d}$ Dimensions:** Original variable values are retained instead of using principal components.

Contrary to the NLP use case, we also demonstrate FLOWER's ability to maintain high-quality diagram visualization with fewer source entities in Figure 2. Even with lower information density, clarity and organization are effectively upheld. With fewer entities, the diagram becomes more digestible at an initial glance, enhancing ease of understanding. Moreover, despite the reduced number of entities, the transformations depicted remain concise and accurately represent the essential pre-processing steps. This adaptability highlights FLOWER's effectiveness across varying levels of complexity.

5 CONCLUSIONS

We have shown our proposed FLOWER (FLOW+ER) diagram and interactive GUI have promise enhancing collaboration and automation aspects of typical analytic pipelines and workflows. Our idea is basically reverse software engineering, but producing a data-centric diagram instead of code. Users can examine and collaborate on their own customized FLOWER diagrams, they can browse source code following a data-centric representation, they can understand how to reuse and extend existing data sets instead of creating new ones. Even though we cannot claim code development can be automated, we believe a data-centric representation could partially automate code maintenance.

We have identified functional dependency discovery in source code as a central problem towards automatic diagram construction, but this aspect requires further research. We have currently explored building the diagram from source code, without looking at data, which means our entire FLOWER computation is $O(1)$. However, we anticipate it will be necessary to execute code and inspect values, at least on data samples to infer data types and evaluate code correctness. Our future efforts will enhance our tool by considering additional DevOps software development aspects such as Continuous Integration (CI) using popular source code control like `GitHub`, and Continuous Delivery (CD) through cloud services (e.g. AWS CodePipeline and Azure Pipelines). This DevOps angle will allow for deeper code development insights and more effective code reuse and maintenance.

Table 1: Overview of DevOps Pillars Addressed by our FLOWER Solution (human participation is required in all pillars)

DevOps Pillar	Enhanced by FLOWER
Collective Responsibility	Partially
Collaboration and Integration	Yes
Pragmatic Implementation	Yes
Bridging Compliance and Development	No
Automation	Partially
Measure, Report and Action	Partially

ACKNOWLEDGMENTS

We would like to acknowledge Wellington Cabrera, David Matusevich, Sofian Maabout, Sikder Tahsin Al-Amin, Elijah Mitchell, Ladjel Bellatreche and Nabila Berkani for their contributions in preliminary efforts and research that laid the groundwork for FLOWER. We also thank Oliver Kennedy for his suggestions to improve the formalization of our problem, as well as clarifying the motivation and potential application scope of our work.

REFERENCES

[1] Sikder Tahsin Al-Amin, Robin Varghese, David Lloyd, Maria A. Gonzalez-Gonzalez, Mario I. Romero-Ortega, and Carlos Ordonez. 2022. Discovering Similar Spike Patterns in High Dimensional Biomedical Signals. In *2022 IEEE International Conference on Big Data (Big Data)*. 4337–4345. https://doi.org/10.1109/BigData55660.2022.10021088

[2] Jing Ao and Rada Chirkova. 2019. Effective and Efficient Data Cleaning for Entity Matching. In *Proc. of the Workshop on Human-In-the-Loop Data Analytics (HILDA)*. ACM, 1–7.

[3] Carlo Batini, Enrico Nardelli, and Roberto Tamassia. 1986. A Layout Algorithm for Data Flow Diagrams. *IEEE Trans. Software Eng.* 12, 4 (1986), 538–546.

[4] Carlo Combi, Barbara Oliboni, Mathias Weske, and Francesca Zerbato. 2018. Conceptual Modeling of Processes and Data: Connecting Different Perspectives. In *Proc. of Conceptual Modeling Conference ER (Lecture Notes in Computer Science)*, Vol. 11157. Springer, 236–250. https://doi.org/10.1007/978-3-030-00847-5_18

[5] Mike Dreves, Gene Huang, Zhuo Peng, Neoklis Polyzotis, Evan Rosen, and Paul Suganthan G. C. 2020. From Data to Models and Back. In *Proceedings of the Fourth Workshop on Data Management for End-To-End Machine Learning, In conjunction with the 2020 ACM SIGMOD/PODS Conference, DEEM@SIGMOD 2020, Portland, OR, USA, June 14, 2020*. ACM, 1:1–1:4. https://doi.org/10.1145/3399579.3399868

[6] Gaoyang Guo. 2018. An Active Workflow Method for Entity-Oriented Data Collection. In *Advances in Conceptual Modeling - ER Workshops*.

[7] Benjamin Hilprecht, Christian Hammacher, Eduardo Souza dos Reis, Mohamed Abdelaal, and Carsten Binnig. 2023. DiffML: End-to-end Differentiable ML Pipelines. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning, DEEM 2023, Seattle, WA, USA, 18 June 2023*. ACM, 7:1–7:7. https://doi.org/10.1145/3595360.3595857

[8] Md. Fazle Elahi Khan, Renran Tian, and Xiao Luo. 2022. Flexible and scalable annotation tool to develop scene understanding datasets. In *HILDA@SIGMOD 2022: Proceedings of the Workshop on Human-In-the-Loop Data Analytics, Philadelphia, Pennsylvania, 12 June 2022*. ACM, 9:1–9:7. https://doi.org/10.1145/3546930.3547499

[9] Elijah Mitchell, Nabila Berkani, Ladjel Bellatreche, and Carlos Ordonez. 2023. FLOWER: Viewing Data Flow In ER Diagrams. In *Proc. of DaWaK Conference* (Penang, Malaysia). Springer-Verlag, Berlin, Heidelberg, 356âĂŞ371. https://doi.org/10.1007/978-3-031-39831-5_32

[10] Carlos Ordonez, Sofian Maabout, David Sergio Matusevich, and Wellington Cabrera. 2013. Extending ER models to capture database transformations to build data sets for data mining. *Data & Knowledge Engineering* 89 (2013), 38–54.

[11] Carlos Ordonez, Sikder Tahsin Al-Amin, and Ladjel Bellatreche. 2020. An ER-Flow Diagram for Big Data. In *2020 IEEE International Conference on Big Data (Big Data)*. 5795–5797. https://doi.org/10.1109/BigData50022.2020.9378088

[12] Minh Pham, Craig A. Knoblock, and Jay Pujara. 2019. Learning Data Transformations with Minimal User Effort. In *IEEE International Conference on Big Data (BigData)*. 657–664.

[13] Merlijn Sebrechts, Sander Borny, Thomas Vanhove, Gregory van Seghbroeck, Tim Wauters, Bruno Volckaert, and Filip De Turck. 2016. Model-driven deployment and management of workflows on analytics frameworks. In *IEEE International Conference on Big Data*. 2819–2826.

[14] William Spoth, Poonam Kumari, Oliver Kennedy, and Fatemeh Nargesian. 2020. Loki: Streamlining integration and enrichment. In *Proc. Human in the Loop Data Analytics (HILDA)*. ACM.

[15] Robin Varghese and Carlos Ordonez. 2023. Understanding Data Pre-processing with a Hybrid Diagram Integrating ER and Data Flow Notation. In *IEEE International Conference on Big Data, BigData 2023, Sorrento, Italy, December 15-18, 2023*. IEEE, 2450–2455. https://doi.org/10.1109/BIGDATA59044.2023.10386722

[16] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell. 2005. Pattern-Based Analysis of the Control-Flow Perspective of UML Activity Diagrams. In *Proc. Conceptual Modeling Conference ER*, Vol. 3716. 63–78.

[17] Jinjin Zhao, Avigdor Gal, and Sanjay Krishnan. 2023. Data Makes Better Data Scientists. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA 2023, Seattle, WA, USA, 18 June 2023*. ACM, 12:1–12:3. https://doi.org/10.1145/3597465.3605228

A GITHUB REPOSITORIES

The source code for each case study can be found at the following github repositories:

- NLP
  https://github.com/RobinVar/NLP
- Biomedical Signals
  https://github.com/RobinVar/pygammasignal