CORAL: COde RepresentAtion Learning with Weakly-Supervised Transformers for Analyzing Data Analysis

Abstract—Large scale analysis of source code, and in particular scientific source code, holds the promise of better understanding the data science process, identifying analytical best practices, and providing insights to the builders of scientific toolkits. However, large corpora have remained unanalyzed in depth, as descriptive labels are absent and require expert domain knowledge to generate. We propose a novel weakly supervised transformer-based architecture for computing joint representations of code from both abstract syntax trees and surrounding natural language comments. We then evaluate the model on a new classification task for labeling computational notebook cells as stages in the data analysis process from data import to wrangling, exploration, modeling, and evaluation. We show that our model, leveraging only easily-available weak supervision, achieves a 38% increase in accuracy over expert-supplied heuristics and outperforms a suite of baselines. Our model enables us to examine a set of 118,000 Jupyter Notebooks to uncover common data analysis patterns. Focusing on notebooks with relationships to academic articles, we conduct the largest ever study of scientific code and find that notebook composition correlates with the citation count of corresponding papers.

I. INTRODUCTION

Data analysis is central to the scientific process. Increasingly, analytical results are derived from code, often in the form of computational notebooks, such as Jupyter notebooks [1]. Analytical code is becoming more frequently published in order to improve replication and transparency [2], [3], [4]. However, as of yet no tools exist to study unlabeled source code both at scale and in depth. Previous in-depth analyses of scientific code heavily rely on expert annotations, limiting the scale of these studies to the order of a hundred examples [5], [6]. Large-scale studies across thousands of examples have been limited to simple summaries such as the number or nature of imported libraries, total line counts, or the fraction of lines that are used for comments [6], [7], [8]. The software engineering community has emphasized the inadequacy of these analyses, noting that "there is a strong need to programmatically analyze Jupyter notebooks" [9], while HCI researchers have observed that studying the data science process through notebooks may play a role in addressing the scientific reproducability crisis [5], [10].

Automated annotation tools could enable researchers to answer important questions about the scientific process across millions of code artifacts. Do analysts share common sequential patterns or processes in their code? Do different scientific domains have different standards or best practices for data analysis? How does the content of scientific code relate to the impact of corresponding publications? To draw insights on the data science process, previous work has conceptualized the analysis pipeline as a sequence of discrete stages starting from importing libraries and wrangling data to evaluation [11], [12], [13]. Building on this conceptual model, our goal is to develop a tool that can automatically annotate code blocks with the analysis stage they support, enabling large-scale studies of scientific data analysis to answer the questions above.

Analyzing scientific code is particularly difficult because as a "means to an end" [14], scientific code is often messy and poorly documented. Researchers engage in an iterative process as they transition between tasks and update their code to reflect new insights [15], [16]. As such, a computational notebook may interleave snippets for importing libraries, wrangling data, exploring patterns, building statistical models, and evaluating analytical results, thereby building a complex and frequently non-linear sequence of tasks [5], [11]. While some analysts use markdown annotations, README's, or code comments to express the intended purpose of their code, these pieces of documentation are often sparse and rarely document the full analysis pipeline [6]. Domain-specific best practices, techniques, and libraries may additionally obfuscate the intent of any particular code snippet. As a result, interpreting scientific code typically requires significant expertise and effort, making it prohibitively expensive to obtain ground truth labels on a large corpus, and therefore infeasible to build annotation tools which require anything more than minimal supervision.

In this paper, we present COde RepresentAtion Learning with weakly-supervised transformers (CORAL) to classify scientific code into stages in the data analysis process. Importantly, the model requires only easily available weak supervision in the form of five simple heuristics, and does not rely on any manual annotations. We show that CORAL learns new relationships beyond the information provided by these heuristics, indicating that currently popular transformer architectures [17] can be extended to weakly supervised tasks with the addition of a small amount of expert guidance. Our model achieves high agreement with human expert annotators and can be scaled to analyze millions of code artifacts, uniquely enabling large-scale studies of scientific data analysis.

We describe a new task for classifying code snippets as stages in the data analysis process (§III-A). We provide an extension to a corpus of 1.23M Jupyter Notebooks (§III-B): a new dataset of expert annotations of stages in the data analysis process for 1,840 code cells in 100 notebooks, which we use exclusively for evaluation and *not* for training (§III-C).

Next we describe CORAL (§IV): a novel graph neural network model for embedding data science code snippets and classifying them as stages in the data science process. To cap-

ture semantic clues about the analyst's intention, CORAL uses a novel masked attention mechanism to jointly model natural language context (such as markdown comments) with structured source code (§IV-B). We implement a weakly supervised architecture with five simple heuristics to compensate for the absence of labels, as labeling code requires domain expertise and is therefore expensive and infeasible at massive scale (§IV-C). To further compensate for limited labels, CORAL combines this weak supervision with unsupervised topic modeling into a multi-task optimization objective (§IV-E).

We evaluate our model (§V) by comparing it to baselines including expert heuristics, weakly supervised LDA, and state-of-the-art neural representation techniques (§V-A). We demonstrate that CORAL, using both code and surrounding natural language annotations, outperforms expert heuristics by 36% and significantly outperforms all other baselines. Through an ablation study we demonstrate that increased maximum sequence length M, weak supervision and unsupervied topic modeling all strictly improve performance, and that including markdown improves performance on cells *without associated markdown* by 13% (§V-B). Further, we explore the impact of maximum input size and dataset size on our model's performance (§V-C), showing that CORAL significantly outperforms all baselines even when trained on only 1k examples. In a comprehensive error analysis, we demonstrate that previously unseen data science functions are correctly labeled with appropriate analysis stages (§V-D).

We then deploy our model to resolve previously unanswered questions about data analysis by linking academic notebooks and associated publications to conduct the largest ever study of scientific code (§VI). We find that (1) there are significant differences between academic and non-academic papers, (2) that papers which include references to notebooks receive on average 22 times the number of citations as papers that do not, and (3) that papers linked to notebooks that more evenly capture the full data science process in expectation receive twice the number of citations for every one standard deviation increase in entropy between stages.

In summary, the contributions of this paper are:

- A new task and public dataset for classifying Jupyter cells as stages in the data science process (§III).
- A multi-task, weakly supervised transformer architecture for classifying code snippets which jointly models natural language and code (§IV).
- A comprehensive evaluation of code representation learning methods (§V).
- The largest ever study of scientific code (§VI).

We make all code and data used in this work publicly available at https://anonymous.

II. RELATED WORK

A. Representation Learning for Source Code

Early methods for code representational learning treated source code as sequence of tokens and built language models on top [18], [19], [20], [21], [22]. Later work incorporated additional information specific to source code, such as object-access patterns [23], code comments [24], parse trees [25], serialized Abstract Syntax Trees (ASTs) [26], [27], ASTs as graph structures [28], and associated repository metadata [29]. As documentation (in markdown format) is prevalent in Jupyter notebooks [6], our model incorporates both markdown text and graph-structured ASTs, taking advantage of both semantic and structural information.

Due to the scarcity of labeled examples, most previous work learned code representations without supervision [30], [31], [32], [33], [22]. The learned representations were mostly used for hole completion tasks, including the prediction of self-defined function names [30], API calls [32], [33], and variable names [31], [22]. In contrast, our task – classifying code cells as analysis stages – arguably requires a higher level understanding of the intention of code. To overcome the bottleneck of manual labeling, we turn to weak supervision. *Snorkel* [34] combined labels from multiple weak supervision sources, denoised them, and used the resulting probabilistic labels to train discriminative models. Building on this idea, we introduce weak supervision to code representation learning by leveraging a small number of expert-supplied heuristics.

B. Graph Neural Networks

GNNs are powerful tools for a variety of tasks, including node classification [35], [36], text classification [37], link prediction [38], [39], graph clustering [40], [41] and graph classification [41], [42], [43], [44]. Additional work suggests that feeding underlying graphical syntax to a natural language model can improve generalization [45]. Tree structures have been show to help summarize source code [46], and complete code snippets [28], [47] in code representation learning. We build on prior work in attention-based graph neural networks [48] and adopt a self-attention mechanism in our model that jointly learns from ASTs and markdown text.

C. Studies of Data Analysis Practices

There is significant existing research on understanding data analysis practices (*e.g.,* [11], [13], [10], [15], [5]), mostly using qualitative methods to elicit experiences from analysts. Some interviews focused specifically on Jupyter notebook users [10], [6]. Despite synthesizing rich observations, interview studies were limited to dozens of participants. A few studies conducted large-scale analysis of Jupyter notebooks, but were limited to simple summary statistics [6], a single library [7], or code quality [8]. Our model enables the analysis of data science both at scale and in depth, which may validate and complement findings from previous qualitative studies.

D. The Data Science Process

A related branch of work [11], [13], [12] modeled the data analysis process as a sequence of iteratively visited stages. Other authors have noted that a better understanding of this process could improve scientific reproducability [5], aid in the development of new analysis tools [15], [10], and identify common points of failure [49].
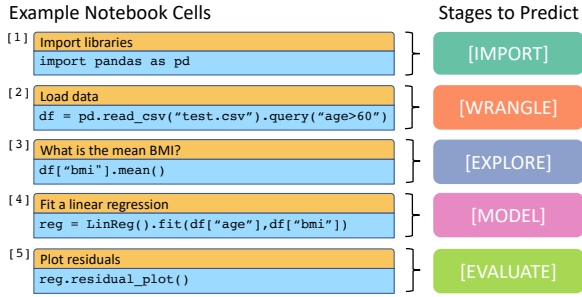
Fig. 1. Examples of our proposed task of automatically labeling code snippets and accompanying natural language annotations as stages in the data science process, with code in blue and markdown in yellow.

III. PREDICTION TASK & DATASETS

We present a new task for labeling code snippets as stages in the data science process (Figure 1), identify a corpus of computational notebooks for large-scale training, and provide a new dataset of expert annotations that are used exclusively in the final evaluation.

A. Prediction Task

In order to automatically learn useful data science constructs from code, we propose a new task and accompanying dataset for classifying code snippets as stages in the data science process. Figure 1 shows five mock examples from this task. We task models with associating a snippet with one of five labels, which are drawn from and motivated by previous work: IMPORT, WRANGLE, EXPLORE, MODEL, and EVALUATE (§II-D). IMPORT cells primarily load external libraries and set environment variables, while WRANGLE cells load data and perform simple transformations. EXPLORE cells are used to visualize data, or calculate simple statistics. MODEL cells define and fit statistical models to the data, and finally EVALUATE cells measure the explanatory power and/or significance of models. Additional details on these stages is available in the Online Reproducability Appendix [50].

B. Jupyter Notebook Corpus for Training

We curate a training set for this task by building upon the UCSD Jupyter notebook corpus, which contains all 1.23M publicly available Jupyter notebooks on Github [6]. Jupyter is the most popular IDE among data scientists, with more than 8M users [51], [52], at least in part because it enables users to combine code with informative natural language markdown documentation. As noted by the corpus' authors, the dataset contains many examples of the myriad uses for notebooks, including completing homework assignments, demonstrating concepts, training lab members, and more [6]. For the purposes of this paper we filtered the corpus to those notebooks that transform, model, or otherwise manipulate data by limiting our analysis to notebooks that import *pandas*, *statsmodels*, *gensim*, *keras*, *scikit-klearn*, *xgboost* or *scipy*. This leaves us with a total of 118k Jupyter notebooks, which we randomly split into training (90%) and validation sets (10%). These notebooks are not annotated with any ground truth labels of data science stages. Thus, we propose a combination of

unsupervised representation learning and weak supervision to study them at scale (§IV).

C. Expert Annotated Notebooks (Only Used for Evaluation)

Annotation. We randomly sampled 100 notebooks containing 1840 individual cells from the filtered dataset for hand-labeling. The first two authors, who have significant familiarity with the Python data science ecosystem, independently annotated the cells with one of the five data science stages. The annotators performed a preliminary round of coding, discussed their results, and produced a standardized rubric for qualitative coding, which is available in the extended paper's Online Reproducability Appendix D (Table VI) [50]. The rubric clearly defines each data analysis stage and provides guidelines for when a label should and should not be used. Using this rubric, the annotators each made a second independent coding pass. We evaluate inter-rater reliability with Cohen's kappa statistic, which corrects for agreement by chance, and find the highest level of correspondence ("substantial agreement", $\kappa = 0.803$) [53]. Finally, the annotators resolved the remaining differences in their labels by discussing each disagreement, producing a final dataset of 1840 cells for model evaluation (§V). Our annotation rubric along with all data and code are available at https://anonymous. Importantly, these expert annotations are never used in training or validation including model selection, but only for the final evaluation (§V).

Multi-Class v.s. Multi-Label. Both annotators paid close attention to potentially ambiguous cells while labeling, observing that it was quite rare for a single cell to be used for multiple stages of the data science process (less than 5% of the time). Furthermore, the median cell in the dataset had two lines of code, making it difficult for a cell to sufficiently express more than one stage. Low label ambiguity at the cell level and high inter-rater reliability support the formulation of this task as multi-class (i.e., five mutually exclusive labels) rather than multi-label (i.e., a cell may have one or more labels), and the selection of cells as the unit of analysis.

IV. THE CORAL MODEL

COde RepresentAtion Learning with weakly-supervised transformers (CORAL) is a model for learning neural representations of data science code snippets and classifying them as stages in the data analysis process. CORAL leverages both source code abstract syntax trees (ASTs) and associated natural language annotations in markdown text (see Fig. 2).

Model Contributions. CORAL contributes the following:

- CORAL jointly learns from code and surrounding natural language (§IV-A), while preserving meaningful code structure through a graph-based masked attention mechanism (§IV-B). We show that adding natural language improves performance by 13% on snippets that do not have associated markdown comments (§V-B).
- We address the lack of high-quality training data through an easily extensible weakly supervised objective based on five simple heuristics (§IV-C).

Fig. 2. An overview of the architecture of our CORAL model, which combines **weak supervision** and **unsupervised topic modeling** into a multitask objective. For visual clarity, we only show edges from the AST here. In practice, we also use connections between [CLS] and all the others nodes, and between each AST node and markdown node (see Section IV-A).

- CORAL combines this weak supervision with an additional unsupervised training objective (again to avoid costly ground truth labels) based on topic modeling, which we combine with other objectives in a multi-task learning framework (§IV-E).

A. Input Representations

CORAL builds on graph neural networks [54] and masked-attention approaches [48] to encode the AST's graph structure by first serializing the tree and then using its adjacency matrix as an attention mask (§IV-B).

We add additional nodes to the AST to capture surrounding natural language. For each code cell, we concatenate its most recent prior markdown as a token sequence to the AST graph sequence (yellow in Figure 2), so long as the markdown is no more than three cells away. Concretely, we create a node for each markdown token and then connect each markdown node with each AST node. Finally, we add a virtual node [CLS] (for *classification*) at the head of every input sequence and connect all the other nodes to it. Similar to BERT, we take this node's embedding as the representation of the cell [55].

Notation. Formally, let $\mathcal{V} = \{u, v, ...\}$ be the set of nodes in the input, where each node v is either an AST node or markdown token. For any input sequence that has more than M nodes, we truncate it and keep only the first M nodes (a modeling choice which we evaluate in §V-C). We use A to represent the graph adjacency matrix that encodes the relationship between nodes as described above. All input nodes are converted to embedding vectors of dimension d_{model}. We assemble these embeddings into a matrix X.

B. Encoding Code Cells with Attention

We extend the popular BERT model [55] by adding masked multi-head attention to capture the graphical structure of ASTs. We evaluate the impact of this addition in §V-A.

CORAL feeds the input code and natural language representations to an encoder, which is composed of a stack of $N = 4$ identical layers (Fig. 2). Similar to Transformers [17], we equip each layer with a multi-head self-attention sublayer and a feed-forward sublayer. The graph structure is captured through *masked* attention (Eq. 2 below).

Masked Multi-Head Attention. We use $Aggregate_k^i$ to represent the self-attention function of $head_i$ in $layer_k$. Let (q, k, v) be the query, key, and value decomposition of the input to $Aggregate_k^i$. Queries and keys are vectors of dimension d_k, and values are vectors of dimension d_v. For a given node u, let (q_u, k_u, v_u) be the triple of query, key and value, and let $N(u)$ be the set of all its neighbours. Formally, the parameters q_u, k_u, v_u vary across each $head_i$ and $layer_k$, but we drop additional notation for simplicity here. Then we compute aggregate results as:

$$Aggregate_k^i(u) = \Sigma_{v \in N(u)} Softmax(\frac{q_u \cdot k_v}{\sqrt{d_k}}) \cdot v_u \quad (1)$$

We adopt the scaling factor $\frac{1}{\sqrt{d_k}}$ from Vaswani *et al.* [17] to mitigate the the dot product's growth in magnitude with d_k. In practice, the queries, keys, and values are assembled into matrices Q, K, V. We compute the output in matrix form as:

$$Aggregate_k^i(Q, K, V) = Softmax(\frac{\tilde{A} \odot QK^T}{\sqrt{d_k}})V \quad (2)$$

where $\tilde{A} = A + I$ is the adjacency matrix with self-loops added to implement the masked attention approach, where each node only attends to its neighbours (described in §IV-A) and itself.

Since we adopt multi-head attention, we concatenate h heads within the same layer:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W_O \quad (3)$$

$$head_i = Aggregate_k^i(XW_Q^i, XW_K^i, XW_V^i) \quad (4)$$

where $head_i \in \mathbb{R}^{d_v}$ and $W_Q^i \in \mathbb{R}^{d_{model} \times d_k}$, $W_K^i \in \mathbb{R}^{d_{model} \times d_k}$, $W_V^i \in \mathbb{R}^{d_{model} \times d_v}$, and $W_O \in \mathbb{R}^{h*d_v \times d_{model}}$ are projection matrices that map the node embeddings X to queries, keys, values, and multi-head output, respectively.

Feed Forward. In each layer, we additionally apply a fully connected feed-forward sublayer. This is composed of two linear transformations with ReLU activation in between:

$$FFN(x) = W_{FF2} \cdot max(0, W_{FF1} \cdot x + b_{FF1}) + b_{FF2} \quad (5)$$

where $W_{FF1} \in \mathbb{R}^{h*d_{model} \times d_{model}}$, $W_{FF2} \in \mathbb{R}^{d_{model} \times h*d_{model}}$, b_{FF1} and b_{FF2} are parameters learned in model.

Add & Norm. Each sublayer is followed by layer normalization [56]. The output of each sublayer is:

$$LayerNorm(x + Sublayer(x)) \quad (6)$$

where $Sublayer(x)$ is multi-head attention or feed forward.

Output. The multi-head attention sublayer and feed-forward sublayer are stacked and make up one "layer". After stacking this layer four times, the encoder's output contains representations of all the nodes in the input sequence. We take the embedding of the [CLS] node as the representation of the each notebook cell's graph (Section IV-A), denoted as $z \in \mathbb{R}^{d_{model}}$.

We compress this cell representation z into a lower-dimensional distribution over K "topics" to capture information about the data analysis stages. Concretely:

$$p_{topic} = Softmax(W_{topic} \cdot z + b) \qquad (7)$$

where $W_{topic} \in \mathbb{R}^{K \times d_{model}}$ is the weighted matrix parameter and b is the bias vector.

C. Weak Supervision

It is prohibitively expensive to obtain manual annotations of data analysis stages at scale, as doing so would require thousands of person-hours of work by domain experts. Therefore, we use five simple heuristics to tailor CORAL to the prediction task described in §III-A:

1) We collect a set of seed functions and assign each to a corresponding stage based on its usage. Any cell that uses a seed is weakly labeled as the corresponding stage. For example, any cell that calls "sklearn.linear_model.LinearRegression" is weakly labeled MODEL. The full set of 39 seed functions is in Online Reproducability Appendix A [50]. We demonstrate CORAL's ability to correctly classify *unseen* code outside these functions in §V-D.

2) A cell with one line of code that does not create a new variable is weakly labeled EXPLORE. This rule leverages a common pattern in Jupyter notebooks where users often use single line expressions to examine a variable, such as a DataFrame.

3) A cell with more than 30% import statements is labeled IMPORT.

4) A cell whose corresponding markdown is less than four words and contains {'logistic regression', 'machine learning', 'random forest'} is weakly labeled MODEL.

5) A cell whose corresponding markdown is less than four words and contains 'cross validation' is weakly labeled EVALUATE.

Note that there may be conflicts between these rules. We observe that less than one percent of cells in our corpus comply with more than one of these heuristics, further supporting our decision to formulate labels as mutually exclusive. We resolve any such conflicts by assigning priority in the following order: IMPORT, MODEL, EVALUATE, EXPLORE, WRANGLE.[1] In this layer, we aim to compute p_{stage} - a probability distribution over these six stages - from the topic distribution computed in Eq. 7. We implement this by mapping the topic distribution p_{topic} to a probability distribution p_{stage} over the $n_{stages} = 6$

stages. We compute the stage distribution p_{stage} as follows, where $W_{stage} \in \mathbb{R}^{K \times n_{stages}}$:

$$p_{stage} = softmax(W_{stage} \cdot p_{topic} + b_{stage}) \qquad (8)$$

We adopt cross entropy loss to minimize classification error on weak labels. For each p_{topic}, loss is computed as:

$$L_{weakly\_supervised} = -\Sigma_s y_{o,s} log(p_s) \qquad (9)$$

where $y_{o,s}$ is a binary indicator (0 or 1) if stage label s is the correct classification for observation o and p_s is the predicted probability p_{stage} is of stage s.

The five weak supervision heuristics cover about 20% of notebook cells in the training data. To minimize the model's ambiguity on the remaining 80% of unlabeled data, and encourage it to choose a stage for each topic, we add an additional loss function. Concretely, we add an entropy term to p_{stage} to encourage uniqueness by forcing the topic distribution to map to as few stages as possible:

$$L_{unique\_stage} = -\Sigma_s p_s log(p_s) \qquad (10)$$

where p_s is the predicted probability $p_{stage}[s]$ for stage s. This entropy objective is minimized when $p_s = 1$ for some s and $p_{s'} = 0$ all other s'.

D. Unsupervised Learning Through Reconstruction

As the weak supervision heuristics only cover about 20% of the cells, we enrich the model with additional training through an unsupervised topic model. Here, the goal is to optimize the topic representation p_{topic} such that we can reconstruct the intermediate cell representation z. We reconstruct z from a linear combination of its topic embeddings p_{topic}:

$$r = R \cdot p_{topic} \qquad (11)$$

where $R \in \mathbb{R}^{d_{model} \times K}$ is the learned cell embedding reconstruction matrix. This unsupervised topic model is trained to minimize the reconstruction error. We adopt the contrastive max-margin objective function using a Hinge loss formulation [57], [58], [59]. Thus, in the training process, for each cell, we randomly sample $m = 5$ cells from our dataset as negative samples:

$$L_{unsupervised} = \Sigma_{c \in D} \Sigma_{i=1}^{m=5} max(0, 1 - r_c z_c + r_c n_i) \quad (12)$$

where D is the training data set, r_c is reconstructed vector of cell c, z_c is intermediate representation of cell c, and n_i is the reconstructed vector of each negative sample. This objective function seeks to minimize the inner product between r_c and n_i, while simultaneously maximizing the inner product between r_c and z_c.

We also employ a regularization term from He *et al.* [60] to promote the uniqueness of each topic embedding in T:

$$L_{unique\_topic} = \left\| R_{norm} \cdot R_{norm}^T - I \right\| \qquad (13)$$

where I is the identity matrix and R_{norm} is the result of L2-row-normalization of R. This objective function reaches its minimum when the inner product of two topic embeddings is 0. We demonstrate in §V-B that this additional unsupervised training improves overall classification performance.

---

[1] We also include a dummy sixth stage to represent cells that are empty or not covered by one of these heuristics. To reflect the uncertainty of these stages they are not included in the model's loss function.
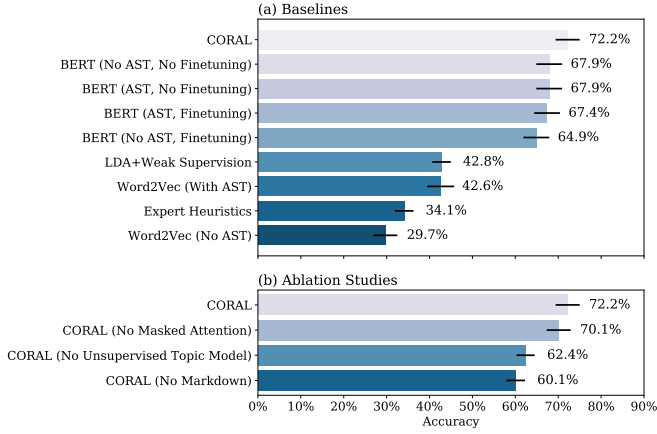
Fig. 3. Accuracy on expert-annotated test set for all baselines (a) and ablation studies (b). Performance improves with neural topic models and weak supervision. CORAL significantly outperforms all baselines (Wilcoxon signed rank, $p < 0.001$) and ablation studies ($p < 0.05$)

E. Final Optimization Objective

We combine the loss functions of Equations (9),(10),(12), and (13) into the final optimization objective:

$$L = \lambda_1 L_{weakly\_supervised} + \lambda_2 L_{unique\_stage}$$
$$+ \lambda_3 L_{unsupervised} + \lambda_4 L_{unique\_topic} \quad (14)$$

where λ_1, λ_2, λ_3 and λ_4 are hyperparameters that control the weights of optimization objectives.

We experiment with various training curricula and find that CORAL with the hyperparameters in Online Reproducability Appendix B [50] achieve the best loss (Eq. 14) on the validation set. Importantly, this optimization and model training is based on solely on the labels from weak supervision heuristics. We do not use expert annotations (§III-C), which we exclusively reserve for the final evaluation.

V. Evaluation

CORAL achieves accuracy of over 72% on the stage classification task using an unseen test set (Section III-C), outperforming a range of baseline models and demonstrating that weak supervision, unsupervised topic modeling, and adding markdown information all strictly improve overall classification performance.

A. Baseline Comparison

In Figure 3(a) we compare CORAL's performance to eight baselines, which we describe below. Importantly, the lack of ground truth labels in our training set makes it impossible to evaluate a model that does not use some amount of weak supervision, as without these heuristics we cannot map between learned topics and data science stages.

Expert Heuristics (Weak Supervision Only). How well does a simple baseline perform that considers only library information? For example, *pandas* is commonly used to wrangle data, and *scikit-learn* is common in modeling. We compare against an improved version of this baseline, where we include *all* expert heuristics described in §IV-C. This set of heuristics consider function-level and markdown information in addition to library information. This is a natural comparison since

this is the exact weak supervision used in CORAL. These heuristics cover only 20.38% of the test examples, so we choose one stage uniformly at random otherwise.

LDA Representation + Weak Supervision. How important is it to use a deep neural encoder for our task? To address this question, we replace CORAL's encoder with a Latent Dirichlet Allocation (LDA) [61] topic model, but use the same input data (§IV-A), and the same weak supervision (Section IV-C). Specifically, we optimize this model with $L_{weak\_supervision}$ (Eq. (9)) and L_{unique_stage} (Eq. (10)) on top of the unsupervised LDA representation. We first used the same number of LDA topics (50) as we use in CORAL (i.e. the size of the cell representation p_{topic}). However, this baseline only performed at the level of the Expert Heuristics Only baseline. In order to make this baseline stronger we doubled the number of LDA topics to 100, which did improve performance.

Neural Baselines. How well does a noncontextual neural model perform on our task? What are the benefits of using the graphical structure of ASTs instead of treating code cells as sequences of tokens in deep neural networks? How important is the multitask objective that combines weak supervision heuristics and an unsupervised topic model? To address these questions, we compare CORAL against the noncontextual Word2Vec [62] model and the state-of-the-art language model, BERT [55], which have both been previously applied to source code representation learning [63], [64]. We trained all neural baselines with both markdown and code using the same pre-training corpus as CORAL. To explore the sensitivity of these models to their input representations, we tried both treating the code as sequences of tokens and as serialized ASTs. For the BERT baselines we used the standard architecture with the same embedding size as CORAL and masked language model pretraining. Predictions are made with a single layer using the same weak supervision heuristics as CORAL. We evaluated BERT baselines both with and without ASTs and finetuning. When finetuning, we backpropagated the single layer's loss through the encoder. After pre-training, we optimized the model with $L_{weak\_supervision}$ (Eq. (9)) and L_{unique_stage} (Eq. (10)) on top of the learned representations of code cells.

Results. Results from these experiments are available in Figure 3(a). The Expert Heuristics baseline achieves 34.1% accuracy on the unseen expert annotations described in §III-C. Even though it uses the same amount of supervision, CORAL is 38% more accurate than this baseline, demonstrating that CORAL learns significantly more than simply memorizing the heuristic rules. CORAL also favorably compares to state-of-the-art neural language models, beating the highest performing BERT baseline by 4.3%. We observe that while popular deep learning techniques like finetuning produce only a marginal difference in model performance, CORAL significantly outperforms all other baselines (Wilcoxon signed rank, $p < 0.001$).

B. Ablation Study

We just demonstrated in §V-A that CORAL improves significantly over expert heuristics, representations that do

not leverage graphical structure, and state-of-the-art neural models. Here we show that (1) adding markdown information, (2) weak supervision, and (3) additional unsupervised training all independently improve the performance of CORAL, as shown in Figure 3(b). Across all experiments we use maximum sequence length of $M = 160$ and train on the maximum 1M code cells, based on the best performing model overall.

CORAL without Markdown. For this ablation, we remove any markdown information from the input sequence, while keeping all other aspects of CORAL the same. We compare maximum sequence length of 80, 120 and 160 since the maximum sequence length M may interact with markdown information due to truncation (§IV-A). We find that including markdown information consistently and significantly improves performance 12% at $M = 160$, even though less that 9% of cells are directly preceded by markdown (Table I). Furthermore, these comparatively rare comments significantly improve performance even on cells that *do not have corresponding markdown information* from 59.6% to 72.6%, suggesting that markdown cells help CORAL better represent source code independent of these comments.

CORAL with Less Weak Supervision. The weak supervision heuristics described in §IV-C cover about 20% of the training examples. We simulate lower coverage by randomly subsampling 50% and 25% of these weakly labeled examples (*i.e.,* 10% and 5% of all examples). Higher weak supervision coverage dramatically increases performance, but even at 25% of examples CORAL still outperforms CORAL (No Masked Attention) by 10% and BERT by 15% (Table II).

CORAL without Unsupervised Topic Model. This baseline evaluates the marginal benefit of CORAL's unsupervised topic model. Specifically, we remove $L_{unsupervised}$ (Eq. 12), and

Fig. 4. Example predictions. Probability distributions over stages from CORAL's SoftMax output (Eq. (7)) are listed on the right side. In (a), CORAL correctly identifies the cell as EVALUATE rather than WRANGLE, likely by interpreting "confusion matrix", perhaps based on previously seen markdown. In (b), the model identifies the use of `sns.regplot`, an unseen statistical visualization function, as an example of EXPLORE. In (c), CORAL correctly interprets a user-defined function.

L_{unique_topic} (Eq. 13) from CORAL but keep everything else the same. We show that the unsupervised training objective improves overall accuracy by 10% (Figure 3(b)). This demonstrates the significant potential of combining limited weak supervision with additional unsupervised training in a multi-task framework.

C. Impact of Input Length & Training Set Size

Maximum Sequence Length. We investigate how model performance changes with the maximum input sequence length M (see Table I). For CORAL models with and without markdown, a larger maximum sequence length consistently improves accuracy. Longer sequence lengths may include more markdown information and limit truncation of larger cells. Only 6% of the training examples have more than 160 nodes, and increases in M also increase training time and memory requirements. Therefore, we did not consider models beyond $M = 160$ and use this setting for all other experiments.

Training Dataset Size. We evaluate the accuracy of CORAL and two other high-performing models with different training dataset sizes to gauge how sensitive our model is to training data size. We fix M to 160 and train with a maximum of 1M notebook cells. In all other experiments, we use the maximum 1M notebook cells for training. While performance consistently decreases with smaller training data (Table III), CORAL achieves an accuracy of 61.85% with only 1k examples and outperforms baselines by a large margin. This demonstrates that the CORAL architecture is effective at learning useful code representations even in smaller-data scenarios, such as on the order of magnitude of a typical GitHub repository.

D. Error Analysis

Confusion Matrix. We include a confusion matrix of CORAL's predictions from the best performing model ($M = 160$ trained on 1M examples) in the Online Reproducability Appendix E (Figure 8) [50]. The most frequent confusion is misclassifying EXPLORE as WRANGLE. This is in part because WRANGLE and EXPLORE are the two most common stages

TABLE IV
FRACTION OF PREDICTED STAGES FOR CELLS THAT CONTAIN PREVIOUSLY UNSEEN FUNCTIONS

Function	Expectation	IMPORT	WRANGLE	EXPLORE	MODEL	EVALUATE
pandas.DataFrame.dropna	Wrangle	0	**0.93**	0.07	0	0
pandas.DataFrame.groupby	Wrangle	0	**0.52**	0.12	0.02	0.34
seaborn.jointplot	Explore	0	0.00	**0.98**	0.00	0.02
seaborn.countplot	Explore	0	0.01	**0.98**	0.00	0.01
sklearn.linear_model.SGDClassifier	Model	0	0	0	**0.67**	0.32
sklearn.linear_model.PassiveAggressiveClassifier	Model	0	0.06	0	**0.61**	0.39
sklearn.metrics.f1_score	Evaluate	0	0	0.01	0.05	**0.94**
sklearn.metrics.log_loss	Evaluate	0.02	0.01	0.02	0.26	**0.70**

CORAL accurately categorizes common data analysis functions as frequently belonging to their expected stage.

in the hand labeled corpus, but also possibly because analysts may apply simple transformations while primarily using a cell to visualize or otherwise explore data.

Unseen Functions. To evaluate how well CORAL can learn beyond memorizing examples from weak supervision, we select eight common data analysis function and compare the labels of cells that contain them (Table IV). Importantly, these functions were not used in weak supervision and thus were never directly associated with any label in the model. Many functions demonstrate clear stage membership in line with our expectations (*e.g., pandas.DataFrame.groupby*, *seaborn.countplot*), demonstrating that CORAL can assign cells including these functions to likely correct stages. Other functions exhibit a more even distribution across stages. For example, *sklearn.linear_model.PassiveAggressiveClassifier*, a simple linear classifier, appears in both MODEL and EVALUATE cells. While ambiguity between stages is rare overall (§III-C) we hypothesize that this confusion may be the result of the scikit-learn use pattern where users specify and evaluate their models in the same cell.

Example Predictions. We highlight three predictions in Figure 4 to demonstrate CORAL's ability to capture data analysis semantics and inherent ambiguity. In Figure 4(a), the user transforms a *pandas* DataFrame and calls *pandas.DataFrame.groupby*, a function typically used to aggregate data. While a naive method (*e.g.,* the expert heuristic baseline in §V-A) might label the cell as WRANGLE, CORAL infers that the analyst's intention is to use this user-defined function to evaluate a classifier with a confusion matrix, likely making use of the information in the comment and function parameters, and appropriately labels the cell as EVALUATE.

In Figure 4(b), the analyst loads data, selects a subset, creates a plot, and fits a linear regression. CORAL correctly identifies this example as serving to both modify data and look for patterns, but assigns a higher probability to EXPLORE, demonstrating its ability to capture the significance of previously unseen statistical visualization methods like *seaborn.regplot*.

In Figure 4(c), the analyst calls a user-defined function. While CORAL has never seen this function or notebook, it still correctly identifies the intent of the cell as EXPLORE likely by attending to tokens like "plot" and "breakdown".
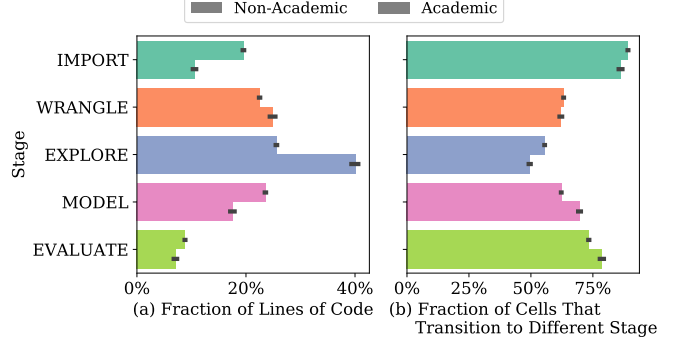

Fig. 5. Differences between academic and non-academic notebooks.

(a) Fraction of Lines of Code
(b) Fraction of Cells That Transition to Different Stage

VI. LARGE SCALE STUDIES OF SCIENTIFIC DATA ANALYSIS

Our model and datasets provide an opportunity to pose and answer previously unaddressable questions about the data analysis process, the role of scientific analysis in academic publishing, and differences between scientific domains. We note that our corpus (§III-B) is limited to the most recent (potentially partial) snapshot of the user's analysis and that the observational nature of this data prohibits any causal claims.

A. Are There Differences Between Academic Notebooks and Non-Academic Notebooks?

Differences between academic and non-academic notebooks could identify how practices vary across these communities.

Method. The Semantic Scholar Open Research Corpus (S2ORC) is a publicly available dataset containing 8.1M full-text academic articles [65]. In order to relate these papers to relevant source code, we performed a regular expression search across the corpus for any reference to a GitHub repository, returning associations between 2.0k papers and 7.1k notebooks from the UCSD corpus. We use this dataset to resolve previously unanswerable questions about the role of analysis code in the scientific process. Although there is no strict guarantee that a linked notebook contains the data analysis that was used to create the paper, the median notebook is linked to exactly one paper, indicating some degree of injectivity from notebooks to papers. Furthermore, manual inspection of our dataset and prior work indicate that researchers often break their analysis up across many notebooks, which may explain why papers link to multiple notebooks. So as not to bias our analysis against how a scientist decides to structure their code, we compute statistics for each paper by concatenating

all associated notebooks. We compute the fraction of code devoted to each data analysis stage and the fraction of cells that are followed by a cell of a different stage and examine differences between academic and non-academic notebooks.

Results. Academic notebooks devote 56% more code to exploring data and 26% less code to developing models than non-academic notebooks (Figure 5(a)). Furthermore, we note that analysts on average use only 23% of their code for the traditionally boring and laborious process of wrangling data. While the relative size of the stage likely does not accurately reflect the relative *effort* of data wrangling, it is perhaps surprising that such a maligned stage of the process [11] is represented by a comparatively low fraction of all code. We also find significant differences in the fraction of cells that are followed by a cell of a different stage (Figure 5(b)). Most interestingly, cells are in general more likely than not to transition to a different stage. This result supports the hypotheses that notebooks follow a transitory process through the data science process to complete an analysis rather than dwelling on any particular stage.

B. Is the Content of Notebooks Related to the Impact of Associated Publications?

Evidence of a relationship between scientific notebooks and publication impact may encourage researchers to publish their code, and could reveal differences between the priorities placed on scientific data analysis by different domains.

Method. We employ a negative binomial regression to estimate the impact of notebook stage distribution on the number of citations their associated papers receive. We hypothesize that notebooks which evenly and comprehensively document their analysis (rather than focusing on just one part) may receive more citations. In our first regression R1, we therefore regress citation count on the Stage Entropy $= -\sum_k p_k \log p_k$, where p_k is the fraction of the notebook that is devoted to stage k. This captures the uniformity of the distribution of stages across a paper's associated notebooks. Here, we normalized this quantity across all publications by taking the Z-score. We controlled for a paper's year of publication and domain. To reveal differences between disciplines, we build upon this experiment with a second regression R2, which includes all terms from R1 except for the entropy term, but adds interaction variables between the Z-scores of the fraction of each paper's notebook devoted to each data analysis stage and paper domains to capture differences between disciplines. additional details for these regression models are available in the Online Reproducability Appendix F [50].

Results. We find that papers that link to notebooks have $10^{\beta_{hasNotebook}} = 10^{1.34} \approx 21.88$ times more citations than papers that do not reference a notebook (95% CI: [1.29, 1.41], $p < 0.001$). From R1 we note that Stage Entropy is strongly related to the number of citations a publication receives, as those publications can expect a $10^{\beta_{stageEntropyZ}} = 10^{0.33} \approx 2.11$ times increase in citations with an entropy level for each standard deviation above the mean (95% CI: [0.26, 0.39],
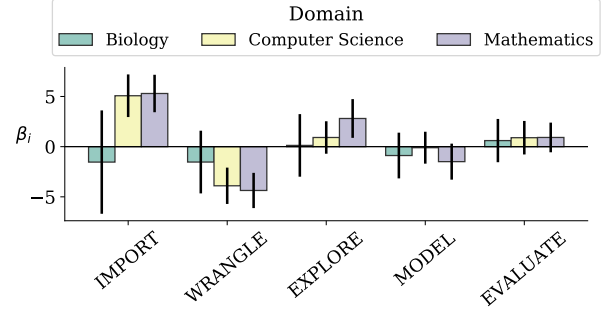

Fig. 6. Results from (R2), indicating differences in how paper impact in different domains is related to the content of associated notebooks.

$p < 0.001$) This result suggests that researchers may value notebooks which evenly document the whole data science process, rather than highlighting just one part of analysis. These results also indicate that a notebook with one standard deviation more than the average EXPLORE code would expect $10^{\beta_{EXPLORE}} = 10^{-0.4325} \approx 0.35$ times the citations in its associated paper than a notebook with an average quantity of all stages (95% CI: [-0.64,-0.22], $p < 0.001$). One possible explanation for this effect is that notebooks which feature a high volume of code for exploring data are associated with generating hypotheses, and may therefore be associated with incomplete or exploratory publications that are less likely to attract references.

The results from R2 (Figure 6) indicate significant differences between domains. Most notably, we find that in computer science and mathematics an increase in the portion of code devoted to wrangling data decreases the citation count in expectation, while no such interaction is present for papers from biological sciences. We hypothesize that the most popular cited notebooks in computer science and mathematics may cleanly demonstrate new techniques and models, rather than documenting an extensive data wrangling pipeline.

We note that although these effect sizes may seem large, we need to consider that the median citation count for papers is only two. This implies that even with a high citation multiplier, papers with just a few citations would expect a rather moderate increase in citations.

VII. CONCLUSION

We presented CORAL, a novel weakly supervised neural architecture for generating representations of code snippets and classifying them as stages in the analysis pipeline. We showed that this model outperforms a suite of baselines on this new classification task. Further, we introduced and made public the largest dataset of code with associated publications for scientific data analysis, and employed CORAL to answer open questions about the data analysis process.

REFERENCES

[1] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay *et al.*, "Jupyter notebooks-a publishing format for reproducible computational workflows." in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 2016.

[2] E. Foster and A. Deardorff, "Open Science Framework (OSF)," *Journal of the Medical Library Association : JMLA*, 2017.

[3] P. Ayers, "LibGuides: Citing & publishing software: Publishing research software."

[4] C. Pradal, G. Varoquaux, and H. Langtangen, "Publishing scientific software matters," *Journal of Computational Science*, pp. 311–312, 2013.

[5] Y. Liu, T. Althoff, and J. Heer, "Paths explored, paths omitted, paths obscured: Decision points & selective reporting in end-to-end data analysis," in *CHI*, 2020.

[6] A. Rule, A. Tabard, and J. Hollan, "Exploration and explanation in computational notebooks," in *CHI*, 2018.

[7] M. Rehman, "Towards understanding data analysis workflows using a large notebook corpus," in *SIGMOD*, 2019.

[8] J. Wang, L. Li, and A. Zeller, "Better code, better sharing: On the need of analyzing Jupyter notebooks," *ICSE*, 2020.

[9] J. Wang, L. Li, and A. Zeller, "Better code, better sharing:on the need of analyzing jupyter notebooks," in *ICSE*, 2020.

[10] M. Kery, M. Radensky, M. Arya, B. John, and B. Myers, "The story in the notebook: Exploratory data science using a literate programming tool," in *CHI*, 2018.

[11] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer, "Enterprise data analysis and visualization: An interview study," *TVCG*, 2012.

[12] K. Wongsuphasawat, Y. Liu, and J. Heer, "Goals, process, and challenges of exploratory data analysis: An interview study," *arXiv:1911.00568*, 2019.

[13] S. Alspaugh, N. Zokaei, A. Liu, C. Jin, and M. Hearst, "Futzing and moseying: Interviews with professional data analysts on exploration practices," *TVCG*, 2018.

[14] A. Johanson and W. Hasselbring, "Software engineering for computational science: Past, present, future," *Computing in Science Engineering*, 2018.

[15] M. Kery, A. Horvath, and B. Myers, "Variolite: Supporting exploratory programming by data scientists," in *CHI*, 2017.

[16] C. Hill, R. Bellamy, T. Erickson, and M. Burnett, "Trials and tribulations of developers of intelligent systems: A field study," in *(VL/HCC)*, 2016.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *NeurIPS*, 2017.

[18] A. Hindle, E. T. B., Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *ICSE*, 2012.

[19] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *FSE*, 2014.

[20] T. Nguyen, A. Nguyen, H. Nguyen, and T. Nguyen, "A statistical semantic language model for source code," in *ESEC/FSE*, 2013.

[21] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[22] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *SIGPLAN*, 2014.

[23] T. Kwon and Z. Su, "Modeling high-level behavior patterns for precise similarity analysis of software," in *ICDM*, 2011.

[24] D. Movshovitz-Attias and W. Cohen, "Natural language models for predicting programming comments," in *ACL*, 2013.

[25] P. Bielik, V. Raychev, and M. Vechev, "Phog: probabilistic model for code," in *ICML*, 2016.

[26] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *SIGPLAN*, 2018.

[27] J. Li, Y. Wang, M. Lyu, and I. King, "Code completion with neural attention and pointer networks," *arXiv:1711.09573*, 2017.

[28] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv:1711.00740*, 2017.

[29] Y. Zhang, F. F. Xu, S. Li, Y. Meng, X. Wang, Q. Li, and J. Han, "Higitclass: Keyword-driven hierarchical classification of github repositories," in *ICDM*, 2019, pp. 876–885.

[30] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *POPL*, 2019.

[31] M. Allamanis, E. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *FSE*, 2014.

[32] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: from usage scenarios to specifications," in *ESEC/FSE*, 2007.

[33] T. Nguyen, A. Nguyen, H. Phan, and T. Nguyen, "Exploring api embedding for api usages and applications," in *ICSE*, 2017.

[34] A. Ratner, S. Bach, H. Ehrenberg, J. Fries, S. Wu, and C. Ré, "Snorkel: Rapid training data creation with weak supervision," *VLDB*, 2019.

[35] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NeurIPS*, 2017.

[36] T. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv:1609.02907*, 2016.

[37] M. Wu, S. Pan, X. Zhu, C. Zhou, and L. Pan, "Domain-adversarial graph neural networks for text classification," in *ICDM*, 2019.

[38] M. Schlichtkrull, T. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European Semantic Web Conference*, 2018.

[39] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *NeurIPS*, 2018.

[40] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *NeurIPS*, 2016.

[41] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," in *NeurIPS*, 2018.

[42] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *ICML*, 2016.

[43] D. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *NeurIPS*, 2015.

[44] F. Scarselli, M. Gori, A. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, 2008.

[45] P. Battaglia, J. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv:1806.01261*, 2018.

[46] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," *arXiv:1811.01824*, 2018.

[47] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," *arXiv:1805.08490*, 2018.

[48] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv:1710.10903*, 2017.

[49] L. Chen, M. Ali Babar, and B. Nuseibeh, "Characterizing architecturally significant requirements," *IEEE Software*, 2013.

[50] Anonymous, "CORAL: COde RepresentAtion Learning with Weakly-Supervised Transformers for Analyzing Data Analysis," https://bit.ly/3hl1PUX.

[51] "Jetbrains data science in 2018." [Online]. Available: https://www.jetbrains.com/research/data-science-2018/

[52] K. Kelley and B. Granger, "Jupyter frontends: From the classic jupyter notebook to jupyterlab, nteract, and beyond." *JupyterCon*, 2017.

[53] J. Landis and G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, 1977.

[54] J. Gilmer, S. Schoenholz, P. Riley, O. Vinyals, and G. Dahl, "Neural message passing for quantum chemistry," in *ICML*, 2017.

[55] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv:1810.04805*, 2018.

[56] J. Ba, J. Kiros, and G. Hinton, "Layer normalization," *arXiv:1607.06450*, 2016.

[57] J. Weston, S. Bengio, and N. Usunier, "Wsabie: Scaling up to large vocabulary image annotation," in *IJCAI*, 2011.

[58] R. Socher, A. Karpathy, Q. Le, C. Manning, and A. Ng, "Grounded compositional semantics for finding and describing images with sentences," *TACL*, 2014.

[59] M. Iyyer, A. Guha, S. Chaturvedi, J. Boyd-Graber, and H. Daumé III, "Feuding families and former friends: Unsupervised learning for dynamic fictional relationships," in *NAACL-HLT*, 2016.

[60] R. He, W. S. Lee, H. T. Ng, and D. Dahlmeier, "An unsupervised neural attention model for aspect extraction," in *ACL*, 2017.

[61] D. Blei, A. Ng, and M. Jordan, "Latent dirichlet allocation," *JMLR*, 2003.

[62] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NeurIPS*, 2013.

[63] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[64] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Pre-trained contextual embedding of source code," 2019.

[65] K. Lo, L. L. Wang, M. Neumann, R. Kinney, and D. S. Weld, "S2ORC: The Semantic Scholar Open Research Corpus," in *ACL*, 2020.

A. Weak Supervision Seed Functions

The seed functions with associated data analysis stages used in weak supervision heuristics are listed in Table V.

TABLE V
SEED FUNCTIONS WITH ASSOCIATED DATA ANALYSIS STAGES USED IN WEAK SUPERVISION HEURISTICS (SECTION IV-C).

Stage	Seed Functions
Wrangle	pandas.read_csv
	pandas.read_csv.dropna
	pandas.read_csv.fillna
	pandas.DataFrame.fillna
	sklearn.datasets.load_iris
	scipy.misc.imread
	scipy.io.loadmat
	sklearn.preprocessing.LabelEncoder
	scipy.interpolate.interp1d
Explore	matplotlib.pyplot.show
	matplotlib.pyplot.plot
	matplotlib.pyplot.figure
	seaborn.pairplot
	seaborn.heatmap
	seaborn.lmplot
	pandas.read_csv.describe
	pandas.DataFrame.describe
Model	sklearn.decomposition.PCA
	sklearn.naive_bayes.GaussianNB
	sklearn.ensemble.RandomForestClassifier
	sklearn.linear_model.LinearRegression
	sklearn.linear_model.LogisticRegression
	sklearn.tree.DecisionTreeRegressor
	sklearn.ensemble.BaggingRegressor
	sklearn.neighbors.KNeighborsClassifier
	sklearn.naive_bayes.MultinomialNB
	sklearn.svm.SVC
	sklearn.tree.DecisionTreeClassifier
	tensorflow.Session
	sklearn.linear_model.Ridge
	sklearn.linear_model.Lasso
Evaluate	sklearn.cross_validation.cross_val_score
	sklearn.metrics.mean_squared_error
	sklearn.model_selection.cross_val_score
	scipy.stats.ttest_ind
	sklearn.metrics.accuracy_score

B. Experiment Setting

We train CORAL with 1M cells on a single GeForce RTX 2080 Ti GPU. The model has four attention heads and four layers of dimension d_{model} = 256. We set the number of topics (§IV-B) to 50 and maximum sequence length (M) to 160. We set $\lambda_1 = 0.1$, $\lambda_2 = 0.3$, $\lambda_3 = 1$ and $\lambda_4 = 1$. We train the model by minimizing L in Equation (14), using the SGD optimizer with a learning rate $\alpha = 1 \times 10^{-5}$, $\beta = 0.9$. Training is done on mini-batches of size 16, for up to 8 epochs with an early stopping criteria if validation error had not improved for 3 epochs. Each epoch takes about 2.5 hours to train.

C. Algorithm

The CORAL Algorithm is shown in Figure 7.

Algorithm 1 CORAL

    Input: Set of nodes V; adjacency matrix A
    Output: Cell embedding z; reconstruted embedding r;
    probability distribution over stages p_{stage}

1: $X = Embedding(V)$
2: **for** $i = 1$ to 4 **do**
3:     $M = MultiHeadAttention(X, A)$
4:     $X = LayerNorm(X + M)$
5:     $F = FeedForward(X)$
6:     $X = LayerNorm'(X + F)$
7: $z = X['[CLS]']$
8: $p_{topic} = Softmax(W_{topic} \cdot z + b)$
9: $r = R \cdot p_{topic}$
10: $p_{stage} = Softmax(W_{stage} \cdot p_{topic} + b_{stage})$

Fig. 7. CORAL Algorithm.

D. Qualitative Rubric

The qualitative rubric used for labeling the Expert Annotated Dataset (Section III-C) used for final model evaluation is listed in Table VI.

TABLE VI
QUALITATIVE RUBRIC USED FOR LABELING THE EXPERT ANNOTATED DATASET (SECTION III-C) USED FOR FINAL MODEL EVALUATION.

Stage	Definition	When to Use	When Not to Use	Example

Import	These cells are used primarily to import libraries into the Python environment. Although they may serve other functions, like defining constants or initializing helper objects, the majority of the code in these cells sets up analytical tools for use later in the notebook.	Loading libraries, defining constants, initializing environments, connecting to databases	A cell has one or more import statements, but most of the cell serves another purpose	`%load_ext autoreload`
				`%autoreload 2`
				`import pandas as pd`
				`import numpy as np`
				`from matplotlib import rcParams`
				`rcParams['figure.figsize'] = 20,10`

Wrangle	Wrangle cells clean, filter, summarize, and/or integrate data. These cells often permute data for use in later cells.	Cleaning data, feature processing, data transformations, augmenting an existing dataset, loading and/or saving data, splitting data into train and test sets	Transformations are applied, but the result is simply examined (See: Explore)	`from sklearn.datasets import load_iris`
				`data = load_iris()`
				`df = pd.DataFrame(data.data)`
				`IN_PER_CM = 0.393701`
				`df = df/IN_PER_CM`

Explore	Interactive explorations of data. These cells tend to yield a result that informs later decisions, or enable the user to draw new conclusions. Explore cells may also transform data, but only for the purpose of exploring relationships and not for further in-depth analysis	Rendering DataFrames, visualizing relationships, printing summaries of data, calculating simple statistics, examining the output of functions	Visualizations are used to evaluate the performance of a model (See: Evaluate)	`df.explore()`

Model	Define and fit models of relationships to data. These cells may include some data transformations, but the primary purpose is to create a model to describe or predict some facet of the dataset	Statistical modeling, fitting and/or specifying machine learning models, simulation, defining loss functions	Significance testing and calculating feature importance (See: Evaluate)	`from sklearn.neighbors import KNNeighbors`
				`knn = KNNeighbors.Classifier()`
				`knn.fit(iris_x_train, iris_y_train)`
				`knn.predict(iris_x_test)`

Evaluate	Measure the explanatory power or predictive accuracy of model using appropriate statistical techniques. These cells sometimes employ visualizations to explore analytical results (e.g. plotting regression residuals)	Cross validation, significance testing, inspecting model output, plotting feature significance.	If a cell both evaluates and defines a machine learning model (a common pattern), default to "Model"	`plot_confusion_matrix(knn,`
				`    iris_x_test, iris_y_test)`
				`plt.set_title("Confusion Matrix")`

E. Confusion Matrix

The confusion matrix for CORAL's predictions on the data analysis stage prediction task is shown in Figure 8.
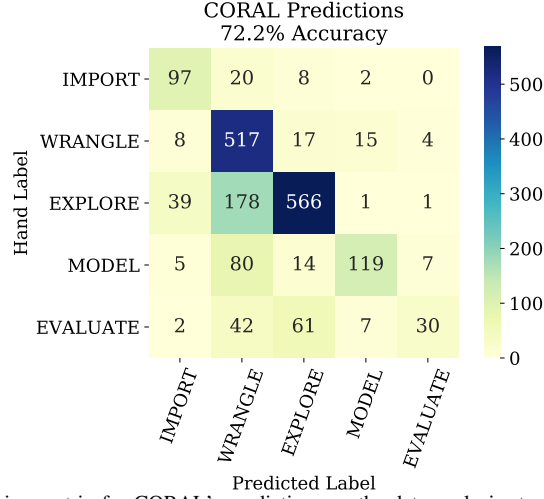

Fig. 8. Confusion matrix for CORAL's predictions on the data analysis stage prediction task.

F. Regression Details

The following details apply to both regression (R1) and regression (R2). We chose to use a negative binomial for zero-inflated counts regression because we observed that the mean number of citations (8.52) was substantially less than the variance (1,308). We expect that a paper's year of publication will influence its citation count, and therefore we control for this variable. We also expect each paper's domain to be related to notebook characteristics, so we limit our analysis to the three most common domains in GORC and control for this factor using indicator variables. We note that our analysis does not substantially change with the inclusion of the top five, 10, or 20 domains. If a paper is linked to more than one notebook, for the purpose of these regressions, we concatenate the notebooks and calculate statistics across this concatenation.