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

Anonymous Author(s)

ABSTRACT

Large scale analysis of computational notebooks holds the promise of better understanding the data science process, quantifying differences between scientific domains, and providing insights to the builders of scientific toolkits. However, such notebooks have remained largely unanalyzed at scale, as labels are absent and require expert domain knowledge to generate. We present a new classification task for labeling computational notebook cells as stages in the data analysis process (i.e., data import, wrangling, exploration, modeling, and evaluation). For this task, we propose a novel weakly supervised transformer architecture for computing joint representations of data science code from both abstract syntax trees and natural language annotations. We show that our model, leveraging only easily-available weak supervision, achieves a 35% increase in accuracy over expert-supplied heuristics. Our model enables us to examine a set of 118,000 Jupyter Notebooks to uncover common data analysis patterns. In the largest analysis of scientific code to date, we relate our public dataset of notebooks to a large corpus of academic articles, finding that notebook characteristics significantly correlate with the citation count of corresponding papers.

1 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 [32]. Further, analytical code is becoming more frequently published in order to improve replication and transparency [8, 19, 40]. However, as of yet no tools exist to study these computational artifacts both at scale and in depth. Previous in-depth analyses of scientific code heavily rely on expert annotations, limiting the scale of these projects to the order of a hundred examples [35, 44]. Large-scale studies across thousands of examples have been limited to superficial analyses and summaries such as the number or nature of imported libraries, total line counts, or the fraction of lines that are used for comments [43, 44, 53].

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 [7, 28, 55]. Informed by 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" [27], 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 [24, 29]. As such, a notebook may interleave snippets for importing libraries, wrangling data, exploring patterns, building statistical models, and evaluating analytical results [28, 35]. 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 [44]. 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.

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 expert annotations. We demonstrate that our model achieves high agreement with human expert annotators and that it can be scaled to analyze millions of code artifacts, uniquely enabling large-scale studies of scientific data analysis. The contributions of this work are as follows:

We describe a new task for classifying code snippets as stages in the data analysis process (Section 3.2). We provide two significant extensions to a corpus of 1.23M Jupyter Notebooks (Section 3.1): a new dataset of expert annotations of stages in the data analysis process for 1,784 code cells in 100 notebooks, which we use exclusively for evaluation and *not* for training (Section 3.2), and a subset of 7.8k notebooks with explicit links to academic publications, forming the largest corpus of scientific analysis code (Section 3.3).

Next we describe CORAL (Section 4): a novel graph neural network for embedding data science code snippets and classifying them as stages in the data science process. Our model builds upon state-of-the-art attention mechanisms from Transformers, which have been shown to effectively capture graph structures [52] (Section 4.2). We implement a weakly-supervised architecture of just five simple heuristics to compensate for sparse labels, as labeling requires domain expertise and is therefore expensive (Section 4.3).

We evaluate our model (Section 5) by comparing it to baselines including expert heuristics, weakly supervised LDA (Section 5.1), and a model without natural language embeddings (Section 5.2). We demonstrate that CORAL, using both code and surrounding natural language annotations, outperforms expert heuristics by 31%. Further, we explore the impact of maximum input size and dataset size on our model's performance (Section 5.3). We discuss specific examples of model predictions that highlight its abilities and calibration even in ambiguous situations in which one code block addresses multiple analysis stages (Section 5.4). In a comprehensive error analysis, we demonstrate that previously unseen data science functions are correctly labeled with appropriate analysis stages (Section 5.4).

We then deploy our model to resolve previously unanswered questions about data analysis (Section 6). Based on a corpus of

118,000 Jupyter Notebooks, we show that analyses tend to follow a common sequential process and find significant differences between academic and non-academic notebooks. We also link academic notebooks and associated publications to find that (1) papers that include references to notebooks receive on average 21.9 times the number of citations as papers that do not, and (2) papers linked to notebooks that more evenly capture the full data science process (with entropy between stages increased by one standard deviation relative to the mean) in expectation receive 2.11 times more citations.

We make all code and data used in this work publicly available at REDACTED FOR ANONYMITY.

2 RELATED WORK

Representation Learning for Source Code. Early methods for code representational learning treated source code as a sequence of tokens and built language models on top [4, 25, 37, 38, 42, 50]. Later work incorporated additional structural information specific to source code, such as parse trees [11, 37], serialized Abstract Syntax Trees (ASTs) [5, 34], and ASTs in graph structures [3]. Besides source code, prior work also demonstrated that concatenating preceding natural language in code documentation with code snippets improved model performance. As documentation (in markdown format) is prevalent in Jupyter notebooks [44], 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 [1, 2, 6, 39, 42]. The learned representations were mostly used for hole completion tasks, including the prediction of self-defined function names [6], API calls [1, 39], and variable names [2, 42]. In contrast, our task – classifying code cells as analysis stages – is arguably more abstract. To overcome the bottleneck of manual labeling, we turn to weak supervision. *Snorkel* [41] 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, leveraging seed functions and heuristic rules supplied by experts.

Graph Neural Networks. GNNs are powerful tools for a variety of tasks, including node classification [22, 31], link prediction [46, 57], graph clustering [15, 56] and graph classification [14, 17, 56]. Battaglia *et al.* [10] suggest that feeding the underlying graphical syntax to a natural language model can improve generalization. Graphs have also been shown to better represent the compositional aspect of natural language [48]. Learning from graphs also improves learning of long-ranging dependencies within a document [49]. Fernandes *et al.* [18] show inspiring results with tree structure in the domain of source code. Two studies [3, 13] apply graph neural networks on programming code for code completion tasks. Another two studies [21, 45] create models that are able to learn vector representations of nodes and graphs in an end-to-end fashion. Veličković *et al.* [52] highlight the potential of attention-based graph neural networks on node classification tasks. We build on this work and adopt a self-attention mechanism in our model and apply it on ASTs and markdown text simultaneously.

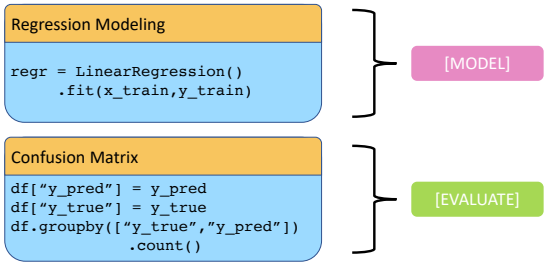

Figure 1: An example of our proposed task for labeling code snippets and accompanying natural language annotations as stages in the data science process.

Analyzing Data Analysis. There is significant existing research on understanding data analysis practices (*e.g.*, [7, 28–30, 35]), mostly using qualitative methods to elicit experiences from analysts. Some interviews focused specifically on Jupyter notebook users [30, 44]. A related branch of work [7, 28, 55] sought to model the data analysis process, conceptualizing the pipeline as a sequence of (iteratively visited) stages. 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 [44], a single library [43], or code quality [53]. Our model enables the analysis of data science both at scale and in depth, which may validate and complement findings from previous qualitative studies.

3 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, and provide two significant extensions to the corpus.

3.1 Jupyter Notebook Corpus

We build upon the UCSD Jupyter notebook corpus, which contains all 1.23M publicly available Jupyter notebooks on Github as of July 2017 [44]. Jupyter notebooks are among the most popular form of computational notebooks, and provide a platform for combining code with natural language markdown annotations. As noted by the original authors, the dataset contains many examples of the myriad uses for notebooks, including completing homework assignments, demonstrating concepts, training lab members, and more [44]. 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 used *pandas*, *statsmodels*, *gensim*, *keras*, *scikit-klearn*, *xgboost* or *scipy*. This leaves us with a total of 118k Jupyter notebooks, from which we randomly sample (10%) for validation and reserve the rest for training.

3.2 Task and Expert-Annotated Notebooks

In order to validate our model's performance, we propose a new task and accompanying dataset for classifying code snippets as stages in the data science process. Figure 1 shows two mock examples from this task. We randomly sampled 100 notebooks containing 1.7k 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

five open-ended labels drawn from prior work: IMPORT, WRANGLE, EXPLORE, MODEL, and EVALUATE. The annotators performed a preliminary round of coding, discussed their results, and produced a standardized rubric for qualitative coding (Table 5). 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, achieving a substantial level of agreement ($\kappa = 0.803$) [33]. Finally, the annotators resolved the remaining differences in their labels by discussing each disagreement, producing a final dataset of 1745 cells for model evaluation (Section 5). Our annotation rubric along with all data and code are available at REDACTED FOR ANONYMITY. These are the largest public datasets of labeled data science code and repositories attached to scientific publications to date. Importantly, these expert annotations are never used in training or validation, but only for the final evaluation (Section 5).

3.3 Massive Corpus of Academic Articles

The Semantic Scholar Graph of References in Context (GORC) is a publicly available dataset containing 8.1M full-text academic articles [36]. 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 7,080 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 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.

4 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 (see Figure 2). CORAL leverages both source code abstract syntax trees (ASTs) and associated natural language annotations in markdown text (Section 4.1). It then learns a representation of code and markdown using the masked self-attention mechanism of Transformers [51] (Section 4.2). To compensate for sparse labels, we combine weak supervision and unsupervised representation learning. For weak supervision, we use only five simple heuristics supplied by researchers, which provide weak labels to about 20% of the cells in the training corpus (Section 4.3). We also apply an unsupervised training objective akin to topic modeling [23], which seeks to reconstruct the cell embedding from the lower-dimensional cell topic distribution (Section 4.4).

4.1 Input Representations

CORAL represents each code cell's AST as a graph and builds on graph neural networks [20] and masked-attention approaches [52] for encoding this graph structure (see Figure 2). We serialize this AST graph into a sequence through depth first search starting at node "Module"[34]. However, we will still leverage this graph

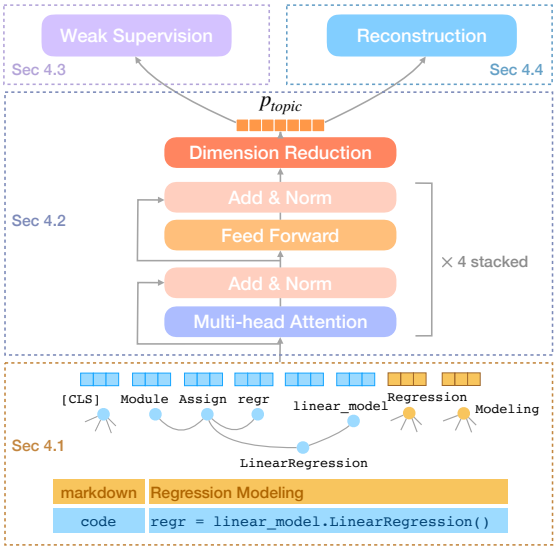

Figure 2: An overview of the neural architecture of our CORAL model. For visual clarity, we only show edges from the AST here. In practice, we also use connections between [CLS] and all the others nodes, and between each AST node and markdown node (see Section 4.1).

structure through a masked self-attention approach using the graph adjacency matrix described in Section 4.2. We add additional nodes and edges to the AST graph structure to capture natural language information. For each code cell, we concatenate its most recent prior markdown as a token sequence to the AST graph sequence (yellow in Figure 2). Concatenating both sequences allows the self-attention mechanism to jointly leverage both AST and markdown information. Formally, we implement this by creating a graph node for each markdown token, and then connecting 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 virtual node's embedding as the representation of the cell [16].

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. We use M to represent maximum sequence length of input sequences to the model. For any input sequence that has more than M nodes, we truncate it and keep only the first M nodes. We use A to represent the graph adjacency matrix that encodes the relationship between nodes as described above. All input tokens are converted to embedding vectors of dimension d_{model}. We assemble all these embeddings into a matrix X.

4.2 Encoding Code Cells with Attention

CORAL feeds the input code and natural language representations to an encoder, which is composed of a stack of N = 4 identical layers (Figure 2). Similar to Transformers [51], we equip each layer with a multi-head self-attention sublayer and a feed-forward sublayer. The graph structure will be leveraged by using masked attention (Equation 2 below).

Masked Multi-Head Attention. We use $Aggregate_k^i$ to represent the self-attention function of $head_i$ in $layer_k$, passing messages from neighbors for nodes. 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:

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

We adopt the scaling factor $\frac{1}{\sqrt{d_k}}$ from Vaswani *et al.* [51]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:

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

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 Section 4.1) and itself.

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

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

$$head_i = Aggregate_k^i(XW_Q^i, XW_K^i, XW_V^i) \tag{4}$$

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 $W_O \in \mathbb{R}^{hd_v \times d_{model}}$ 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:

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

where $W_{FF1} \in \mathbb{R}^{4d_{model} \times d_{model}}$, $W_{FF2} \in \mathbb{R}^{d_{model} \times 4d_{model}}$, b_{FF1} and b_{FF2} are parameters learned in model.

Add & Norm. Each sublayer is followed by layer normalization [9]. The output of each sublayer is:

$$LayerNorm(x + Sublayer(x)) \tag{6}$$

where $Sublayer(x)$ is either 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 outputs of the encoder are the representations of all the tokens in the input sequence. We take the embedding of the [CLS] node as the intermediate representation of the graph associated with each notebook cell (Section 4.1), denoted as $z \in \mathbb{R}^{d_{model}}$.

We now compress this cell representation z into a lower-dimens-ional probability distribution over K "topics". The intuition behind this step is to learn a set of topics that will capture all relevant information to classify snippets into data analysis stages. Concretely:

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

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

4.3 Weak Supervision

In our model, we use the five data analysis stages described in Section 3.1 : `IMPORT`, `WRANGLE`, `EXPLORE`, `MODEL`, and `EVALUATE`. In addition, we use a sixth dummy stage, `STAGE_PAD` for empty cells. As described in the introduction, it is prohibitively expensive to obtain manual annotations of data analysis stages at scale. Therefore, CORAL leverages weak supervision signals based on five simple heuristics:

(1) We collect a set of seed functions based on their usage, with each assigned to a corresponding stage. 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 Appendix A.2 . We demonstrate CORAL's ability to correctly classify *unseen* code outside these seed functions in Section 5 .
(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 resolve any such conflicts by assigning priority in the following order: `IMPORT`, `MODEL`, `EVALUATE`, `EXPLORE`, `WRANGLE`.

In this layer, we aim to compute p_{stage} - a probability distribution over these six stages - from the topic distribution computed in Equation (7). We implement this by mapping the topic distribution p_{topic} to a probability distribution p_{stage} over the six stages. We compute the stage distribution p_{stage} as follows, where $W_{stage} \in \mathbb{R}^{K \times 6}$:

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

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

$$L_{weakly\_supervised} = -\Sigma_{c=1}^{6} y_{o,s} log(p_s) \tag{9}$$

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:

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

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

4.4 Unsupervised Topic Model

As the weak supervision heuristics only cover 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 approximately reconstruct the intermediate cell representation z. We reconstruct z from a linear combination of its topic embeddings p_{topic}:

$$r = R \cdot p_{topic} \tag{11}$$

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 [26, 47, 54]. Thus, in the training process, for each cell, we randomly sample $m = 5$ cells from our dataset as negative samples:

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

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.

Additionally, we employ a regularization term used by He *et al.* [23] to promote the uniqueness of each topic embedding in T:

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

where I is the identity matrix and R_{norm} is the result of row-normalizing(L2) R. This objective function reaches its minimum when the inner product of two different topic embeddings is 0. We demonstrate in Section 5.2 that this additional unsupervised training improves overall classification performance.

4.5 Final Optimization Objective

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

$$\begin{aligned} L = & L_{weakly\_supervised} + L_{unique\_stage} \\ & + L_{unsupervised} + L_{unique\_topic} \end{aligned} \tag{14}$$

We experiment with various training curricula and find that CORAL with hypterparameters in Appendix A.3 achieve the best loss (Equation (14)) on the validation set. Importantly, this optimization and model training is based on solely on the labels from weak supervision heuristics. We did not use expert annotations (Section 3.2) , which we exclusively reserved for the final evaluation.

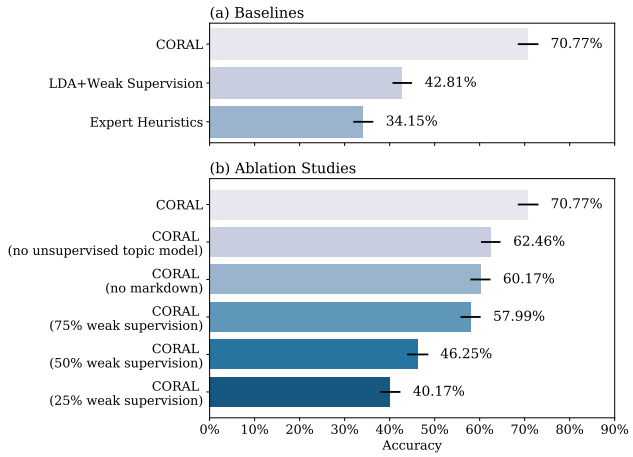

Figure 3: Accuracy on expert-annotated test set for all baselines. Performance improves with neural topic models and weak supervision.

5 EVALUATION

CORAL achieves accuracy of over 70% on the stage classification task using an unseen test set (Section 3.2) , outperforming a range of baseline models and demonstrating that weak supervision, unsupervised learning, and adding markdown information all strictly improve the overall classification performance.

5.1 Baseline Comparison

In Figure 3(a) we compare CORAL's performance to two baselines, which we describe below.

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 Section 4.3. This set of heuristics consider more than library information including function-level and markdown information. This is a natural comparison since this is the exact weak supervision used in CORAL. These heuristics cover only 20.53% of the test examples, so we choose uniformly at random otherwise.

LDA Representation + Weak Supervision. How important is the masked attention-based encoding mechanism leveraging AST graph information described in Section 4.2? To address this question, we replace CORAL's encoder with a Latent Dirichlet Allocation (LDA) [12] topic model, but use the same input data (Section 4.1), and the same weak supervision (Section 4.3). Specifically, we optimize this model with $L_{weak\_supervision}$ (Eq. (9)) and L_{unique_stage} (Eq. (10)) on top of the unsupervised LDA representation. In contrast to CORAL, LDA ignores the graphical structure of ASTs and treats code cells as sequences of tokens. 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, performance of this baseline was only on the level of the Weak Supervision baseline. In order to make this baseline stronger, we doubled the number of LDA topics to 100, which did improve performance.

Max Sequence Length	Accuracy(%)	
	CORAL (No Markdown)	CORAL
160	60.17	70.77
120	56.96	65.62
80	54.04	59.48

Table 1: The impact of including markdown information in CORAL. Training on markdown data in addition to code significantly increases performance independent of maximum sequence length.

Number of Cells in Training	Accuracy(%)
1M	70.77
100k	64.47
10k	62.64
1k	60.80

Table 2: CORAL accuracy across various training dataset sizes. Performance consistently increases with more training data but remains promising even with very little data.

Results. The Weak Supervision Only baseline achieves 34.15% accuracy on the unseen expert annotations described in Section 3.2 (or 16.14% without the uniform guess addition; Figure 3). Even though it uses the same amount of supervision, CORAL is 35% more accurate than this baseline, demonstrating that CORAL learns significantly more than simply memorizing the heuristic rules.

The weakly supervised LDA model outperforms Expert Heuristics by 8%, but is 28% worse than CORAL. This shows that graph neural networks leveraging the graph-structured information in code offer better performance for code representation learning than NLP-inspired token-based methods on our task.

5.2 Ablation Study

We just demonstrated in Section 5.1 that CORAL improves significantly over weak supervision without representation learning and over representations that do not leverage graphical structure in code with attention. 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 may interact with markdown information due to truncation. We find that including markdown information consistently and significantly improves performance up to 10% at $M = 160$ (Table 1). This demonstrates the utility in including natural language markdown information in code representation learning.

CORAL with Less Weak Supervision. The weak supervision heuristics described in Section 4.3 cover about 20% of the training examples. Here, we simulate lower coverage by randomly subsampling to only 75%, 50%, and 25% of these weakly labeled examples (*i.e.,* , 15%, 10% and 5% of all training examples). We find that higher weak supervision coverage dramatically increases performance, but

(a)
```
def Compute_TPFN(y_pred, y_true):
    'Custom method to compute the confusion matrix'
    df = pd.DataFrame(columns=('y_pred', 'y_true')
    df['y_pred'] = pd.Series(y_pred)
    df['y_true'] = pd.Series(np.array(y_true))
    return df.groupby(['y_pred', 'y_true']).count()
```
```
Import    0
Wrangle   0.01
Explore   0
Model     0
Evaluate  0.99
```

(b)
```
train = pd.read_csv('D:/pyplace/hisRawData.csv')
import seaborn as sns
train = train[(train.status == 11)]
train = train[(train.power > 0)]
train = train[(train.speed > 0)]
get_ipython().run_line_magic('matplotlib','inline')
sns.regplot(x='speed', y='power', data=train)
```
```
Import    0
Wrangle   0.29
Explore   0.70
Model     0.01
Evaluate  0
```

(c)
```
eclf4 = VotingClassifier(estimators=[('lr', alg),
     ('DT', alg6), ('Ada', alg4)], voting='soft')
eclf4.fit(X_train, y_train)
probs = eclf4.predict_proba(X_test)
score = log_loss(y_test, probs)
print score
```
```
Import    0
Wrangle   0
Explore   0
Model     0.54
Evaluate  0.46
```

Figure 4: Example predictions. Probability distributions over stages from CORAL's SoftMax output (Eq. (7)) are listed on the right side. We show that CORAL leverages semantics from natural language and captures ambiguity between stages.

that even at 25% of examples, CORAL still outperforms the Expert Heuristics baseline, a shown in Figure 3(b).

CORAL without Unsupervised Training. This baseline evaluates the marginal benefit of CORAL's unsupervised topic model. Specifically, we remove $L_{unsupervised}$ (Eq. 12), and L_{unique_topic} (Eq. 13) from CORAL but keep everything else the same. We show that unsupervised training objectives improve overall accuracy by 8% (Figure 3(b)). This demonstrates the significant potential of combining limited weak supervision with additional unsupervised training to learn better performing code representations.

5.3 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 1). For both models with and without markdown, a larger maximum sequence length consistently improves accuracy. Longer sequence lengths tend to include more markdown information in training and prevents the truncation of some larger cells. Only 6% of the training examples have more than 160 nodes, and increases in maximum sequence length 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 with different training dataset sizes to gauge how sensitive our model is to the amount of training data. We fix M to 160 and train with a maximum of 1M notebook cells. While performance consistently decreases with smaller training data (Table 2), it is promising to see that even with only 1k examples, CORAL achieves an accuracy of 60.80% 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. In all other experiments, we use the maximum 1M notebooks cells for training.

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**

Table 3: Fraction of predicted stages for cells that contain previously unseen functions. CORAL accurately categorizes common data analysis functions as frequently belonging to their expected stage.

5.4 Error Analysis

Confusion Matrix. Figure 8 shows the confusion matrix of CORAL predictions for the best performing model ($M = 160$ trained on 1M examples). The largest confusion is mispredicting WRANGLE as EXPLORE. Possibly, this is due to many analysts also applying simple statistical models, such as moving averages, while wrangling data in a single cell.

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 3). Importantly, these functions were not used in weak supervision and 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. Based on manual inspection, we hypothesize that this ambiguity may be the result of the common 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 Section 5.1) 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 ambiguously serving to both modify data and look for patterns in data, 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 initializes a classifier, trains it, and then calculates its log loss on test data. The user is clearly both initializing a model and evaluating its performance, which is captured by CORAL's confusion between the two classes.

6 LARGE SCALE STUDIES OF SCIENTIFIC DATA ANALYSIS

Our model and datasets provide an opportunity to pose and answer previously unsolvable questions about the data analysis process, the role of scientific analysis in academic publishing, and differences between scientific domains. However, we note that as the UCSD Jupyter Notebook Corpus (Section 3.1) does not contain the full history of each notebook, our experiments are limited to the most recent (potentially partial) snapshot of the user's analysis. In addition, the observational nature of this data prohibits any causal claims.

6.1 Are There Common Sequential Patterns in Data Science Notebooks?

A deeper understanding of patterns within the data analysis process could help developers design tools that better meet the needs of their users. For example, if analyses tend to be messy, highly iterative and non-sequential, tools supporting analysts may need to take this into account.

Method. We represent each notebook as a series of cells $cell_0, cell_1 \ldots cell_\ell$, where ℓ is the total number of cells in the notebook. CORAL assigns a label $y_i \in \{\texttt{IMPORT}, \texttt{WRANGLE}, \texttt{EXPLORE}, \texttt{MODEL}, \texttt{EVALUATE}\}$ to each cell in the notebook. We compute a transition matrix between stages by estimating $P(cell_{i+1} = y_{i+1} | cell_i = y_i)$. We additionally compute the normalized cell index $\frac{i}{\ell}$ for each label, and take the median across the whole dataset as a representation of each label's expected location in a notebook.

Results. We find that EXPLORE, and MODEL are all most likely to transition to cells of the same stage, whereas IMPORT cells are most likely to transition to WRANGLE and EXPLORE (Figure 5). EVALUATE cells are frequently followed by MODEL cells, perhaps indicating an iterative process as users produce models, evaluate them, and transfer the insights from the evaluation to a new model. Furthermore, we find that IMPORT, WRANGLE, EXPLORE, MODEL, and EVALUATE have median normalized cell indexes of 0.07, 0.41, 0.50, 0.54, and 0.63, respectively. This indicates that the *average* notebook follows a predictable sequence from importing libraries to evaluating models, while *individual* notebooks tend to follow a more iterative, non-sequential path.

6.2 Are There Differences Between Academic Notebooks and Non-Academic Notebooks?

Differences between academic and non-academic notebooks could be used to identify how best practices vary across these disciplines.

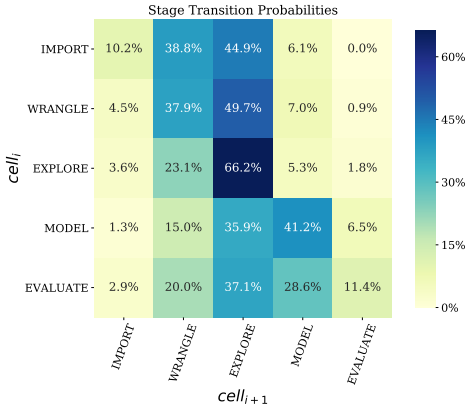

Figure 5: Transitions between data science stages in a corpus of 118k Jupyter Notebooks.

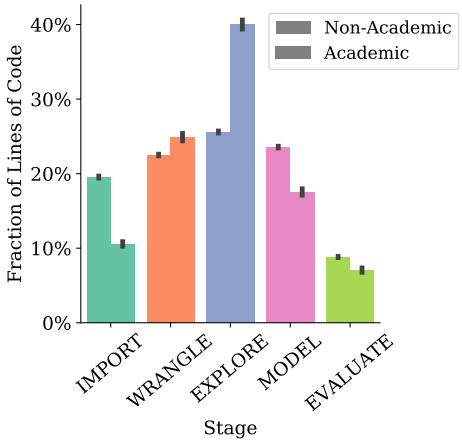

Figure 6: The fraction of code devoted to each stage.

Method. As detailed in Section 3.3, we use a corpus of 8.1M full-text academic articles to identify associations between 2.0k papers and 7.8k Jupyter Notebooks. We compute the fraction of code devoted to each data analysis 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 6). 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 [28] is represented by a comparatively low fraction of all code.

6.3 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 aid in quantifying differences between the priorities placed on scientific data analysis across 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. We build upon this experiment with a second regression R2, which include all terms from R1 except for the entropy term, but add 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. More details on these regression models are available in Appendix A.1.

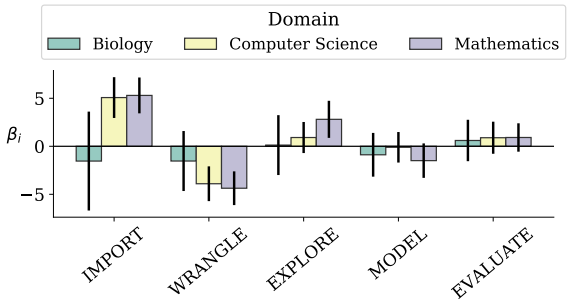

Figure 7: Results from (R2), indicating differences in how paper impact in different domains is related to the content of associated notebooks.

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 (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 $10^{\beta_{stageEntropyZ}} = 10^{0.325} \approx 2.11$ times increase in citations with an entropy level for each standard deviation above the mean. 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 mean 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. 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 7) indicate significant differences between domains. Most notably, we find that only computer scientists and mathematicians care about the amount of code used for imports and that there is not a single significant interaction for biologists, perhaps indicating that these fields assign less importance to the publication of scientific source code.

We note that although these effect sizes may seem large, we need to consider that the median citation count for papers is only 2 (mean 12.73). This implies that even with a high citation multiplier, papers with just a few citations would expect a rather moderate increase in citations.

7 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] M. Acharya, T. Xie, J. Pei, and J. Xu. 2007. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC/FSE*. 25–34.

[2] M. Allamanis, E. Barr, C. Bird, and C. Sutton. 2014. Learning natural coding conventions. In *FSE*. 281–293.

[3] M. Allamanis, M. Brockschmidt, and M. Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[4] M. Allamanis and C. Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 207–216.

[5] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. 2018. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices* 53, 4 (2018), 404–419.

[6] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. 2019. code2vec: Learning distributed representations of code. *popl* 3, POPL (2019), 1–29.

[7] S. Alspaugh, N. Zokaei, A. Liu, C. Jin, and M.A. Hearst. 2018. Futzing and moseying: Interviews with professional data analysts on exploration practices. *TVCG* 25, 1 (2018), 22–31.

[8] P. Ayers. [n.d.]. LibGuides: Citing & publishing software: Publishing research software. https://libguides.mit.edu/c.php?g=551454&p=3786120

[9] J.L. Ba, J.R. Kiros, and G.E. Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

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

[11] P. Bielik, V. Raychev, and M. Vechev. 2016. PHOG: probabilistic model for code. In *ICML*. 2933–2942.

[12] D.M. Blei, A.Y. Ng, and M.I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[13] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O Polozov. 2018. Generative code modeling with graphs. *ArXiv* abs/1805.08490 (2018).

[14] H. Dai, B. Dai, and L. Song. 2016. Discriminative embeddings of latent variable models for structured data. In *ICML*. 2702–2711.

[15] M. Defferrard, X. Bresson, and P. Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *NeurIPS*. 3844–3852.

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

[17] D.K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R.P. Adams. 2015. Convolutional networks on graphs for learning molecular fingerprints. In *NeurIPS*. 2224–2232.

[18] P. Fernandes, M. Allamanis, and M. Brockschmidt. 2018. Structured neural summarization. *arXiv preprint arXiv:1811.01824* (2018).

[19] E.D. Foster and A. Deardorff. 2017. Open Science Framework (OSF). *Journal of the Medical Library Association : JMLA* 105, 2 (April 2017), 203–206.

[20] J. Gilmer, S.S. Schoenholz, P.F. Riley, O. Vinyals, and G.E. Dahl. 2017. Neural message passing for quantum chemistry. In *ICML*. 1263–1272.

[21] M. Gori, G. Monfardini, and F. Scarselli. 2005. A new model for learning in graph domains. In *IJCNN*, Vol. 2.

[22] W. Hamilton, Z. Ying, and J. Leskovec. 2017. Inductive representation learning on large graphs. In *NeurIPS*. 1024–1034.

[23] R. He, W. Sun Lee, H. Tou Ng, and D. Dahlmeier. 2017. An unsupervised neural attention model for aspect extraction. In *ACL*. 388–397.

[24] C. Hill, R. Bellamy, T. Erickson, and M. Burnett. 2016. Trials and tribulations of developers of intelligent systems: A field study. In *(VL/HCC)*. 162–170.

[25] A. Hindle, Earl T B., Z. Su, M. Gabel, and P. Devanbu. 2012. On the naturalness of software. In *ICSE*. 837–847.

[26] M. Iyyer, A. Guha, S. Chaturvedi, J. Boyd-Graber, and H. Daumé III. 2016. Feuding families and former friends: Unsupervised learning for dynamic fictional relationships. In *NAACL-HLT*. 1534–1544.

[27] A. Johanson and W. Hasselbring. 2018. Software engineering for computational science: Past, present, future. *Computing in Science Engineering* (2018), 1–1.

[28] S. Kandel, A. Paepcke, J.M. Hellerstein, and J. Heer. 2012. Enterprise data analysis and visualization: An interview study. *TVCG* 18, 12 (2012).

[29] M.B. Kery, A. Horvath, and B. Myers. 2017. Variolite: Supporting exploratory programming by data scientists. In *CHI*. 1265–1276.

[30] M.B. Kery, M. Radensky, M. Arya, B.E. John, and B.A. Myers. 2018. The story in the notebook: Exploratory data science using a literate programming tool. In *CHI*. 1–11.

[31] T.N. Kipf and M. Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[32] T. Kluyver, B. Ragan-Kelley, F. Pérez, B.E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J.B. Hamrick, J. Grout, S. Corlay, et al. 2016. Jupyter Notebooks-a publishing format for reproducible computational workflows.. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. 87–90.

[33] JR Landis and GG Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* (1977), 159–174.

[34] J. Li, Y. Wang, M.R. Lyu, and I. King. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).

[35] Y. Liu, T. Althoff, and J. Heer. 2020. Paths explored, paths omitted, paths obscured: Decision points & selective reporting in end-to-end data analysis. In *CHI*. 406:1–406:14.

[36] K. Lo, L.L. Wang, M. Neumann, R. Kinney, and D.S. Weld. 2019. GORC: A large contextual citation graph of academic papers. *arXiv preprint arXiv:1911.02782* (2019).

[37] C. Maddison and D. Tarlow. 2014. Structured generative models of natural source code. In *ICML*. 649–657.

[38] T.T. Nguyen, A.T. Nguyen, H.A. Nguyen, and T.N. Nguyen. 2013. A statistical semantic language model for source code. In *ESEC/FSE*. 532–542.

[39] T.D. Nguyen, A.T. Nguyen, H.D. Phan, and T.N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *ICSE*. 438–449.

[40] C. Pradal, G. Varoquaux, and H.P. Langtangen. 2013. Publishing scientific software matters. *Journal of Computational Science* 4, 5 (2013), 311–312.

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

[42] V. Raychev, M. Vechev, and E. Yahav. 2014. Code completion with statistical language models. In *SIGPLAN*. 419–428.

[43] M.S. Rehman. 2019. Towards understanding data analysis workflows using a large notebook corpus. In *SIGMOD*. 1841–1843.

[44] A. Rule, A. Tabard, and J.D. Hollan. 2018. Exploration and explanation in computational notebooks. In *CHI*. 32:1–32:12.

[45] F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, and G. Monfardini. 2008. The graph neural network model. *IEEE Transactions on Neural Networks* 20, 1 (2008).

[46] M. Schlichtkrull, T.N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling. 2018. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*. 593–607.

[47] R. Socher, A. Karpathy, Q.V. Le, C.D. Manning, and A.Y. Ng. 2014. Grounded compositional semantics for finding and describing images with sentences. *TACL* 2 (2014), 207–218.

[48] R. Socher, A. Perelygin, J. Wu, J. Chuang, C.D. Manning, A.Y. Ng, and C. Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*. 1631–1642.

[49] K.S. Tai, R. Socher, and C.D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[50] Z. Tu, Z. Su, and P. Devanbu. 2014. On the localness of software. In *FSE*. 269–280.

[51] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, and I. Polosukhin. 2017. Attention is all you need. In *NeurIPS*. 5998–6008.

[52] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[53] J. Wang, L. Li, and A. Zeller. 2019. Better code, better sharing: On the need of analyzing Jupyter notebooks. *arXiv preprint arXiv:1906.05234* (2019).

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

[55] K. Wongsuphasawat, Y. Liu, and J. Heer. 2019. Goals, process, and challenges of exploratory data analysis: An interview study. *arXiv preprint arXiv:1911.00568* (2019). https://arxiv.org/abs/1911.00568

[56] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec. 2018. Hierarchical graph representation learning with differentiable pooling. In *NeurIPS*. 4800–4810.

[57] M. Zhang and Y. Chen. 2018. Link prediction based on graph neural networks. In *NeurIPS*. 5165–5175.

A REPRODUCIBILITY

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

Table 4: Seed functions with associated data analysis stages used in weak supervision heuristics (Section 4.3).

A.1 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 20 domains. If a paper is linked to more than one notebook,

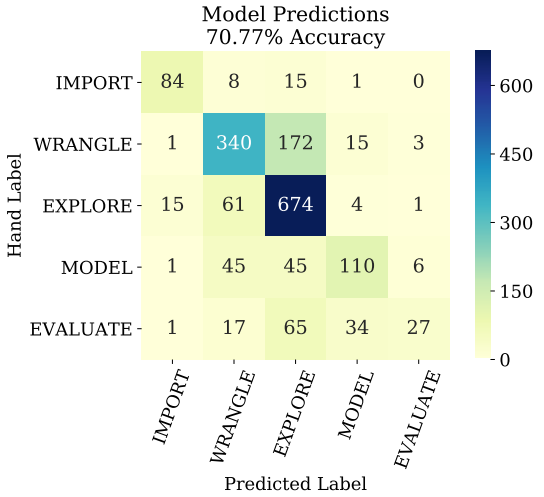

Figure 8: Confusion matrix for CORAL's predictions on the data analysis stage prediction task.

Algorithm 1 CORAL

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

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

Figure 9: CORAL Algotithm.

for the purpose of these regressions, we concatenate the notebooks and calculate statistics across the this concatenation.

A.2 Weak Supervision Seed Functions

The seed functions with associated data analysis stages used in weak supervision heuristics are listed in Table 4.

A.3 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 (Section 4.2) to 50 and maximum sequence length (M) to 160. 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 15 epochs with an early stopping criteria if validation error had not improved for 3 epochs. Each epoch takes about 2.5 hours to train.

A.4 Algorithm

The CORAL Algorithm is shown in Figure 9.

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")`

Table 5: Qualitative rubric used for labeling the Expert Annotated Dataset (Section 3.2) used for final model evaluation.