EPJ Data Science
a SpringerOpen Journal

REGULAR ARTICLE Open Access

CORAL: COde RepresentAtion learning with weakly-supervised transformers for analyzing data analysis

Ge Zhang[1†], Mike A. Merrill[1†], Yang Liu[1], Jeffrey Heer[1] and Tim Althoff[1*]

*Correspondence:
althoff@cs.washington.edu
[1]Paul G. Allen School of Computer Science and Engineering, Seattle, USA
†Equal contributors

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 study of scientific code to date and find that notebooks which devote an higher fraction of code to the typically labor-intensive process of wrangling data in expectation exhibit decreased citation counts for corresponding papers. We also show significant differences between academic and non-academic notebooks, including that academic notebooks devote substantially more code to wrangling and exploring data, and less on modeling.

Keywords: Data science; Meta science; Representation learning

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 [1]. Analytical code is becoming more frequently published in order to improve replication and transparency [2–4]. 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 [6–8]. 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 [11–13]. 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 (Sect. 3.1). We provide an extension to a corpus of 1.23M Jupyter Notebooks (Sect. 3.2): a new dataset of expert annotations of stages in the data analysis process for 1840 code cells in 100 notebooks, which we use exclusively for evaluation and *not* for training (Sect. 3.3).

Next we describe CORAL (Sect. 4): a novel graph neural network model for embedding data science code snippets and classifying them as stages in the data science process. To capture 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 (Sect. 4.2). 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 (Sect. 4.3). To further compensate for limited labels, CORAL combines this weak supervision with unsupervised topic modeling (Sect. 4.4) into a multi-task optimization objective (Sect. 4.5).

We evaluate our model (Sect. 5) by comparing it to baselines including expert heuristics, weakly supervised LDA, and state-of-the-art neural representation techniques such as BERT [18], Word2Vec [19], Tree-LSTM [20] and Tree-Based CNN [21] (Sect. 5.1). 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 unsupervised topic modeling all strictly improve performance, and that including markdown improves performance on cells *without associated markdown* by 13% (Sect. 5.2). This demonstrates that CORAL learns effective joint representations of code and markdown that generalize to settings where only code is available and improve performance even when additional markdown information is unavailable. Further, we explore the impact of maximum input size and dataset size on our model's performance (Sect. 5.3), 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 (Sect. 5.4).

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 (Sect. 6). 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 (Sect. 3).
- A multi-task, weakly supervised transformer architecture for classifying code snippets which jointly models natural language and code (Sect. 4).
- A comprehensive evaluation of code representation learning methods (Sect. 5).
- The largest study of scientific code to date (Sect. 6).

We make all code and data used in this work publicly available at http://bdata.cs.washington.edu/coral/.

2 Related work

2.1 Representation learning for source code

Early methods for code representational learning treated source code as sequence of tokens and built language models on top [22–26]. Later work incorporated additional information specific to source code, such as object-access patterns [27], code comments [28], parse trees [29], serialized Abstract Syntax Trees (ASTs) [30, 31], ASTs as graph structures

[21, 32], associated repository metadata [33], and logs [34]. 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 [26, 35–38]. The learned representations were mostly used for hole completion tasks, including the prediction of self-defined function names [35], API calls [37, 38], and variable names [26, 36]. 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* [39] 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.

2.2 Graph neural networks

GNNs are powerful tools for a variety of tasks, including node classification [40, 41], text classification [42], link prediction [43, 44], graph clustering [45, 46] and graph classification [46–49]. Additional work suggests that feeding underlying graphical syntax to a natural language model can improve generalization and overall performance [20, 50]. Tree structures have been show to help summarize source code [51], and complete code snippets [32, 52] in code representation learning. We build on prior work in attention-based graph neural networks [53] and adopt a self-attention mechanism in our model that jointly learns from ASTs and markdown text.

2.3 Studies of data analysis practices

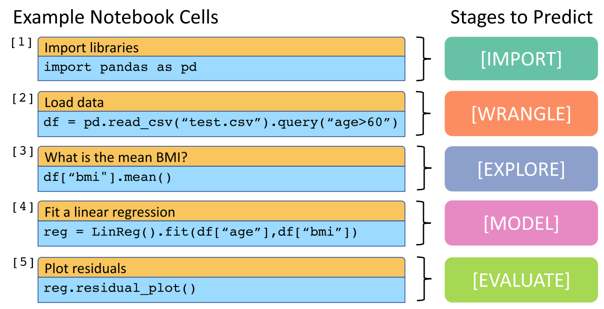
There is significant existing research on understanding data analysis practices (*e.g.*, [5, 10, 11, 13, 15]), mostly using qualitative methods to elicit experiences from analysts. Some interviews focused specifically on Jupyter notebook users [6, 10]. Despite synthesizing rich observations, interview 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.

2.4 The data science process

A related branch of work [11–13] 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 [10, 15], and identify common points of failure [54].

3 Prediction task & datasets

We present a new task for labeling code snippets as stages in the data science process (Fig. 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.



```
import pandas as pd
```

```
df = pd.read_csv("test.csv").query("age>60")
```

```
df["bmi"].mean()
```

```
reg = LinReg().fit(df["age"],df["bmi"])
```

```
reg.residual_plot()
```

Figure 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*

3.1 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` (Sect. 2.4). `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 [55].

In keeping with prior work ([8, 56, 57]) we focus on Jupyter notebook cells as our unit of analysis. While our method could certainly be extended to other snippets like lines, function bodies, or even whole files (given a sufficiently large transformer block), focusing on cells allows us to learn from and exploit authors' tendencies to organically organize their code along these lines.

3.2 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 [58, 59], 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 (Sect. 4).

3.3 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 A.4 (Table A3) [55]. 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", $\kappa = 0.803$) [60]. The annotators resolved the remaining differences in their labels by discussing each disagreement, producing a final dataset of 1840 cells for model evaluation (Sect. 5).

Finally, to verify that the authors' impressions correspond with popular conceptions, three independent data scientists (who had no involvement with this project) were asked to independently annotate five notebooks (comprising 87 cells in total) using the rubric in Table A3. This second group of annotators achieved a mean Cohen's Kappa of 0.64 ("Good" [60]) with the authors and a Krippendorff's Alpha of 0.74 ("Acceptable" [61]) between each other. We note that it took each of these participants approximately one hour to complete their annotation task. At this rate it would take thousands of person-hours to annotate our complete unlabeled notebook corpus, further underscoring the need for semi-supervised models like ours.

Our annotation rubric along with all data and code are available at http://bdata.cs.washington.edu/coral/. Importantly, these expert annotations are never used in training or validation including model selection, but only for the final evaluation (Sect. 5).

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.

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. 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 (Sect. 4.1), while preserving meaningful code structure through a graph-based masked attention mechanism (Sect. 4.2). We show that adding natural language improves performance by 13% on snippets that do not have associated markdown comments (Sect. 5.2).

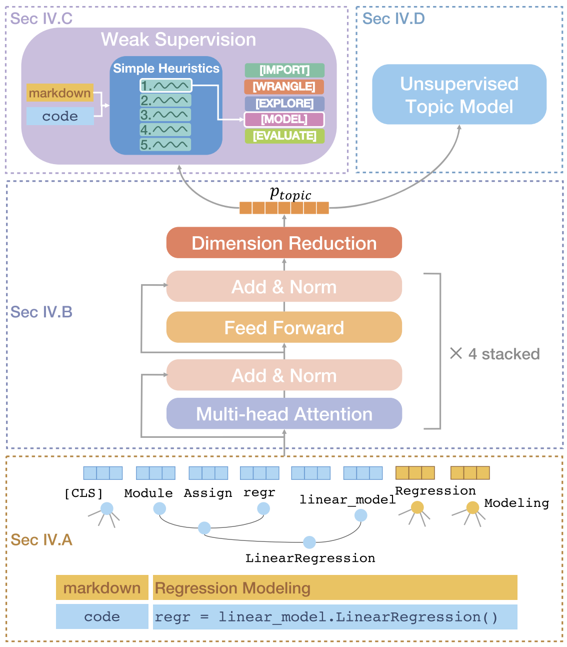


Figure 2 An overview of the architecture of our CORAL model, which combines *weak supervision* and *unsupervised topic modeling* into a multitask objective. For visual clarity, we only show edges from the AST here. In practice, we also use connections between [CLS] and all the others nodes, and between each AST node and markdown node (see Sect. 4.1)

- We address the lack of high-quality training data through an easily extensible weakly supervised objective based on five simple heuristics (Sect. 4.3).
- 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 (Sect. 4.5).

4.1 Input representations

CORAL builds on graph neural networks [62] and masked-attention approaches [53] to encode the AST's graph structure by first serializing the syntax tree in depth-first traversal and then using its adjacency matrix as an attention mask (Sect. 4.2).

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 Fig. 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 [18].

Notation Formally, let $\mathcal{V} = \{u, v, \ldots\}$ 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 Sect. 5.3). 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.

4.2 Encoding code cells with attention

We extend the popular BERT model [18] by adding masked multi-head attention to capture the graphical structure of ASTs. We evaluate the impact of this addition in Sect. 5.1.

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:

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

We adopt the scaling factor $\frac{1}{\sqrt{d_k}}$ from Vaswani *et al.* [17] to mitigate 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:

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

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 Sect. 4.1) and itself.

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

$$\begin{aligned} &MultiHead(Q, K, V) \\ &\quad = Concat(head_1, \ldots, head_h)W_O, && (3) \\ &head_i = Aggregate_k^i\big(XW_Q^i, XW_K^i, XW_V^i\big), && (4) \end{aligned}$$

where $head_i \in \mathbb{R}^{d_v}$ and $W_Q^i \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_K^i \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_V^i \in \mathbb{R}^{d_{\text{model}} \times d_v}$, and $W_O \in \mathbb{R}^{h * d_v \times d_{\text{model}}}$ 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:

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

where

$$W_{\text{FF1}} \in \mathbb{R}^{h * d_{\text{model}} \times d_{\text{model}}},$$

$$W_{\text{FF2}} \in \mathbb{R}^{d_{\text{model}} \times h*d_{\text{model}}}$$

and b_{FF1} and b_{FF2} are parameters learned in model.

Add & norm Each sublayer is followed by layer normalization [63]. The output of each sublayer is:

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

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 $N = 4$ times to allow information to propagate between nodes, the encoder's output contains contextual 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 (Sect. 4.1), denoted as $z \in \mathbb{R}^{d_{\text{model}}}$.

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

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

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

4.3 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 Sect. 3.1:

1. We collect a set of seed functions and assign each to a corresponding stage based on its usage. These functions are among the most commonly used in popular Python data science libraries like matplotlib and sklearn, and were selected by expert Python data scientists. 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.1 [55]. We demonstrate CORAL's ability to correctly classify *unseen* code outside these functions in Sect. 5.4.
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 $n_{\text{stages}} = 6$ stages. We compute the stage distribution p_{stage} as follows, where $W_{\text{stage}} \in \mathbb{R}^{K \times n_{\text{stages}}}$:

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

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

$$L_{\text{weakly\_supervised}} = -\sum_{s} y_{o,s} \log(p_s), \tag{9}$$

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:

$$L_{\text{unique\_stage}} = -\sum_{s} p_s \log(p_s), \tag{10}$$

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

4.4 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}:

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

where $R \in \mathbb{R}^{d_{\text{model}} \times K}$ 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 [64–66]. Thus, in the training process, for each cell, we randomly sample $m = 5$ cells from our dataset as negative samples:

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

[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.

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.* [67] to promote the uniqueness of each topic embedding in R:

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

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 apply this regularization term to encourage orthogonality among the rows of the cell embedding reconstruction matrix R and penalize redundancy between reconstruction vectors. We demonstrate in Sect. 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), and (13) into the final optimization objective:

$$\begin{aligned} L = {} & \lambda_1 L_{\text{weakly\_supervised}} + \lambda_2 L_{\text{unique\_stage}} \\ & + \lambda_3 L_{\text{unsupervised}} + \lambda_4 L_{\text{unique\_topic}}, \end{aligned} \tag{14}$$

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 A.2 [55] 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 (Sect. 3.3), which we exclusively reserve for the final evaluation.

5 Evaluation

CORAL achieves accuracy of over 72% on the stage classification task using an unseen test set (Sect. 3.3), outperforming a range of baseline models and demonstrating that weak supervision, unsupervised topic modeling, and adding markdown information all strictly improve overall classification performance.

5.1 Baseline comparison

In Fig. 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. Unless noted otherwise, all models accept markdown and code using the same input preprocessing (Sect. 4.1).

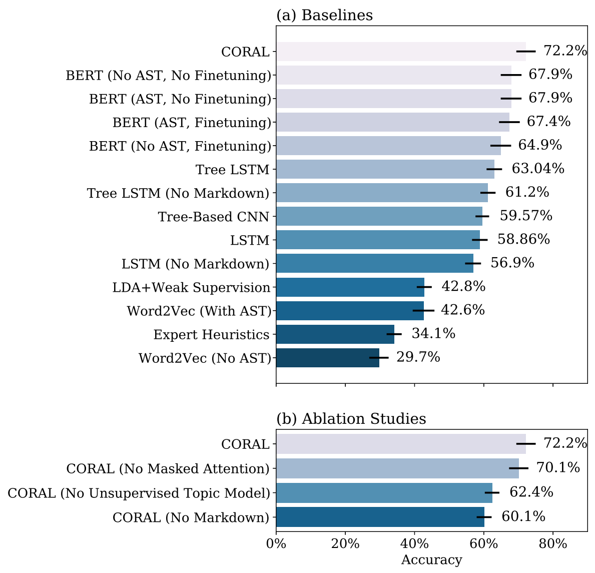


Figure 3 Accuracy on expert-annotated test set for all baselines (**a**) and ablation studies (**b**). Performance improves with neural topic models and weak supervision. CORAL significantly outperforms all baselines (Wilcoxon signed rank, $p < 0.001$) and ablation studies ($p < 0.05$)

How much does our model learn beyond its simple heuristics and weak supervision? To answer this question, we compare CORAL to a baseline that only uses the *Expert Heuristics* described in Sect. 4.3. This set of heuristics considers 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. We show that CORAL outperforms this baseline by a factor of two (Fig. 3(a)), indicating that our model learns significant information beyond its heuristics.

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) [68] topic model, but use the same input data (Sect. 4.1), and the same weak supervision (Sect. 4.3). We denote this model as *LDA+Weak Supervision*. Specifically, we optimize this model with $L_{\text{weak\_supervision}}$ (Eq. (9)) and $L_{\text{unique\_stage}}$ (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}). In order to make this baseline stronger we doubled the number of LDA topics to 100, which did improve performance slightly, but the model still only achieved 42.8% accuracy (Fig. 3(a)). This experiment shows that neural representations lead to significant improvement on our task.

How does CORAL compare to a state-of-the-art transformer-based network? We used the standard *BERT* architecture with the same embedding size as CORAL and masked language model pre-training. After pre-training, we optimized the model against $L_{\text{weak\_supervision}}$ (Eq. (9)) and $L_{\text{unique\_stage}}$ (Eq. (10)) on top of the learned representations of code cells. We evaluate BERT baselines both with and without ASTs and finetuning. With finetuning, we backpropagate the loss from a linear layer trained to predict the snippet's stage from the embedding of the [CLS] virtual node. For experiments conducted with no finetuning, we independently trained a linear layer without backpropagation through the rest of the network. To explore the sensitivity of this model to its input representations, we tried treating code both as sequences of tokens and as serialized ASTs. In all experiments CORAL significantly outperformed BERT (72.2% v.s. at most 67.9%, Fig. 3(a)) indicating that our model improves upon the successful transformer architecture.

How important are contextual embeddings for performance on our task? We trained *Word2Vec* on the dataset, and then took the mean of the embeddings for each token as the embedding for the full sequence. We used a single layer that treated the weak supervision heuristics from CORAL as strong labels to classify these sequences. As with BERT, we run this experiment both with token sequences and serialized ASTs. Word2Vec performs poorly, near the level of LDA + Weak Supervision (42.6%, Fig. 3(a)), indicating that contextual embeddings like those in CORAL are a powerful method for our task.

Is it important to use a transformer? We trained an *LSTM* on depth-first traversal of ASTs from the dataset, with the same embedding size and maximum sequence length as CORAL. Then we took the last hidden state output as the embedding for the full sequence. We used a single layer that treated the weak supervision heuristics from CORAL as strong labels to classify these sequences. CORAL significantly outperforms LSTM (72.2% v.s. 58.86%, Fig. 3(a)), indicating that the attention-centric mechanism of the transformer provides measurable gains in our task.

How does CORAL compare to neural baselines specific to learning code representations? We trained *Tree-Based CNN* (TBCNN) [21] on ASTs from the dataset, with the same embedding size as CORAL. The softmax classifier took the weak supervision heuristics from CORAL as strong labels to classify these cells. As TBCNN does not generalize to cyclic graphs, and markdown nodes are connected to every AST node in our input representation (Sect. 4.1), we cannot evaluate this model with markdown information. CORAL significantly outperforms this baseline (72.2% v.s. 59.6% Fig. 3(a)), indicating that our novel model is a powerful method for learning code representations.

Does a recurrent graph neural network perform well on our task? We trained Dependency *Tree-LSTM* [20] on ASTs from the dataset. At the root node of each AST, we used a softmax classifier to predict the label generated by the weak supervision heuristics from CORAL. The classifier took the hidden state at the root node as the cell embedding. CORAL performs better than Tree-LSTM (72.2% vs 63.0% Fig. 3(a)), indicating that our method learns powerful graph representations.

Summary of baseline comparison. Results from these experiments are available in Fig. 3(a). CORAL learns significantly more than simply memorizing the heuristic rules. CORAL outperforms existing neural models. In particular, we find that it outperforms both the LSTM and Tree-LSTM. CORAL is also 12.6% more accurate than TBCNN, demonstrating the power of its masked self-attention mechanism. CORAL 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, $p < 0.001$).Interestingly, while CORAL's performance is dramatically improved by including markdown (from 60.1% to 72.2%, Sect. 5.2), recurrent models like Tree LSTM and LSTM see only marginal gains. This may indicate that these previous-generation models do not jointly model code and natural language efficiently.

Notably, BERT's performance is not strongly affected by the inclusion of ASTs. Recent work indicates that large transformer-based language models, like BERT, can learn to effectively represent code even without structural information about syntax (i.e. by simply treating code as text) [69, 70]. Our results support this observation.

5.2 Ablation study

We just demonstrated in Sect. 5.1 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 Fig. 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.

How much does jointly learning code and natural language improve performance? For this ablation (denoted as *CORAL No Markdown*), 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 (Sect. 4.1). 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 1). 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.

How well does CORAL perform if fewer weakly supervised labels are available? The weak supervision heuristics described in Sect. 4.3 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 2).

Table 1 Impact of Max Sequence Length on CORAL. Training on markdown data in addition to code significantly increases performance independent of maximum sequence length

Model	Max Sequence Length		
	80	120	160
CORAL	59.9	64.2	72.2
CORAL(No Markdown)	54.4	57.0	60.2

Table 2 CORAL accuracy across weak supervision coverage. Training with more weak supervision significantly improves performances

Model	Weak Supervision Coverage		
	25%	50%	100%
CORAL	41.6	46.4	72.2
CORAL(No Masked Attention)	31.7	46.1	70.4
BERT(AST, No Finetuning)	26.1	47.2	67.9

Table 3 CORAL accuracy across various training dataset sizes. Performance consistently increases with more training data but remains promising even with three orders of magnitude less training data

Model	Number of Cells			
	1k	10k	100k	1M
CORAL	61.9	62.7	63.6	72.2
CORAL (No Masked Attention)	53.6	57.0	59.7	70.4
BERT (AST, No Finetuning)	41.0	52.4	63.2	67.9

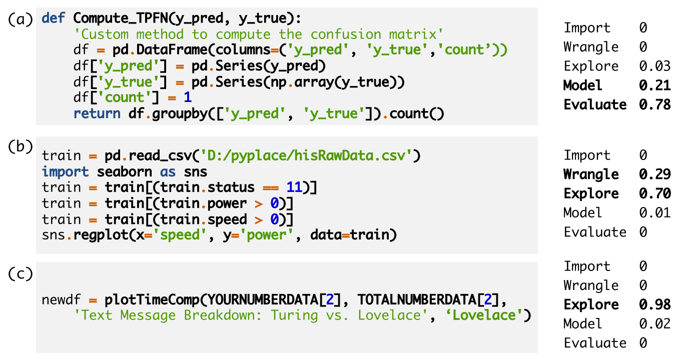
Does CORAL's additional unsupervised topic model objective materially improve performance? The *CORAL(No Unsupervised Topic Model)* ablation addresses this question. Specifically, we remove $L_{\text{unsupervised}}$ (Eq. (12)), and $L_{\text{unique\_topic}}$ (Eq. (13)) from CORAL but keep everything else the same. We show that the unsupervised training objective improves overall accuracy by 10% (Fig. 3(b)). This demonstrates the significant potential of combining limited weak supervision with additional unsupervised training in a multi-task framework.

Does masked attention contribute to CORAL's results? We provide an additional ablation on CORAL in which we replace the model's masked attention mechanism (Sect. 4.1) with standard full attention. As shown in Fig. 3(b), masked attention produces a modest but statistically significant increase in performance from 70.1% to 72.2%.

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 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 3), 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.



```
(a) def Compute_TPFN(y_pred, y_true):
        'Custom method to compute the confusion matrix'
        df = pd.DataFrame(columns=('y_pred', 'y_true','count'))
        df['y_pred'] = pd.Series(y_pred)
        df['y_true'] = pd.Series(np.array(y_true))
        df['count'] = 1
        return df.groupby(['y_pred', 'y_true']).count()
```

```
Import    0
Wrangle   0
Explore   0.03
Model     0.21
Evaluate  0.78
```

```
(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)]
    sns.regplot(x='speed', y='power', data=train)
```

```
Import    0
Wrangle   0.29
Explore   0.70
Model     0.01
Evaluate  0
```

```
(c) newdf = plotTimeComp(YOURNUMBERDATA[2], TOTALNUMBERDATA[2],
        'Text Message Breakdown: Turing vs. Lovelace', 'Lovelace')
```

```
Import    0
Wrangle   0
Explore   0.98
Model     0.02
Evaluate  0
```

Figure 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

5.4 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 A.5 (Fig. A.2) [55]. The most frequent confusion is misclassifying `EXPLORE` as `WRANGLE`. This is in part because `WRANGLE` and `EXPLORE` are the two most common stages 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. For example, in Fig. 4(b) the user applies a trivial transformation by filtering examples where `train.speed` is above zero, the overall intention of the cell is to visualize the relationship between train speed and power. We note that `MODEL` is occasionally confused for `WRANGLE`, perhaps because operations like train-test splits frequently happen in modeling cells (although these operations typically constitute a small fraction of all cells tokens). Furthermore, `EVALUATE` is frequently misclassified as `EXPLORE`, perhaps because model evaluations often involve visualization (e.g. confusion matrices). Finally, we note that predictions on `EVALUATE` cells are wrong more often than not, perhaps because of limited training data for this class. While performance on `IMPORT` cells is higher (despite comparably few training examples), `IMPORT` cells all contain the `import` token and are therefore easier to predict.

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 4). 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 (Sect. 3.3) 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.

Table 4 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

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**

Example predictions We highlight three predictions in Fig. 4 to demonstrate CORAL's ability to capture data analysis semantics and inherent ambiguity. In Fig. 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 Sect. 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 Fig. 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 Fig. 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".

6 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 (Sect. 3.2) 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.

6.1 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 [71]. 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

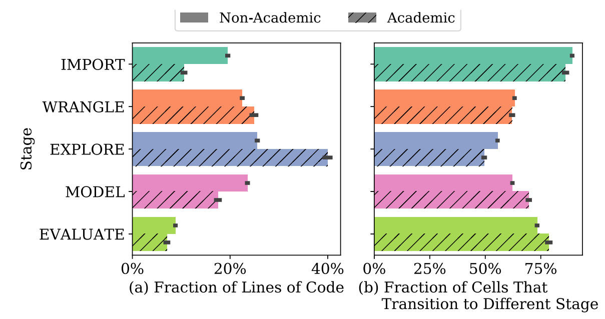


Figure 5 Differences between academic and non-academic notebooks

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 (Fig. 5(a)). Furthermore, we note that analysts on average use only 23% of their code for the traditionally 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.

6.2 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 A.6 [55].

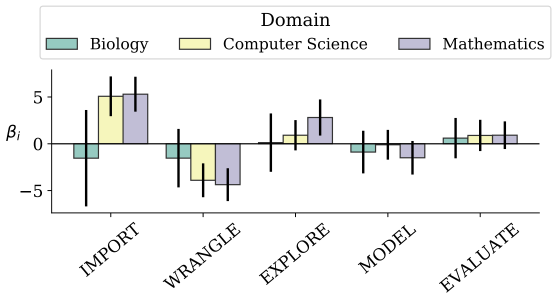


Figure 6 Results from (R2), indicating differences in how paper impact in different domains is related to the content of associated notebooks

Results We find that papers that link to notebooks have $10^{\beta_{\text{hasNotebook}}} = 10^{1.34} \approx 21.88$ 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 $10^{\beta_{\text{stageEntropyZ}}} = 10^{0.33} \approx 2.11$ times increase in citations with an entropy level for each standard deviation above the mean (95% CI: [0.26, 0.39], $p < 0.001$) 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 $10^{\beta_{\text{EXPLORE}}} = 10^{-0.4325} \approx 0.35$ 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` (Fig. 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 in our dataset 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.

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.

Appendix: Reproducability

A.1 Weak supervision seed functions

The seed functions with associated data analysis stages used in weak supervision heuristics are listed in Table A1.

A.2 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 $d_{\text{model}} = 256$. We set the number of topics (Sect. 4.2) to 50. 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. Hyperparameters were selected with a randomized search across $\alpha \in [10^{-6}, 10^{-3}]$, $\beta \in (0, 1]$, $\lambda \in (0, 1]$, $d_{\text{model}} \in \{16, 32, 64, 128, 256, 512\}$, and the number of topics $\in [1, 100]$. Similarly, the choice of appending markdown within the previous three

Table A1 Seed functions with associated data analysis stages used in weak supervision heuristics (Sect. 4.3)

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

```
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: $p_{stage} = Softmax(W_{stage} \cdot p_{topic} + b_{stage})$
```

Figure A.1 CORAL Algorithm

Table A2 Summary statistics for our unlabeled dataset

Number of Notebooks	118,762
Mean Number of Cells per Notebook	19.12
Mean Number of Lines of Code per Cell	3.81
Mean Number of Functions Used per Cell	2.08

cells (as opposed to some other number; Sect. 4.1) was selected with a random search in the range [1, 9].

A.3 Algorithm

The CORAL Algorithm is shown in Fig. A.1.

A.4 Qualitative rubric

The qualitative rubric used for labeling the Expert Annotated Dataset (Sect. 3.3) used for final model evaluation is listed in Table A3.

A.5 Confusion matrix

The confusion matrix for CORAL's predictions on the data analysis stage prediction task is shown in Fig. A.2.

A.6 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 (1308). 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 S2ORC 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.

Table A3 Qualitative rubric used for labeling the Expert Annotated Dataset (Sect. 3.3) 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`

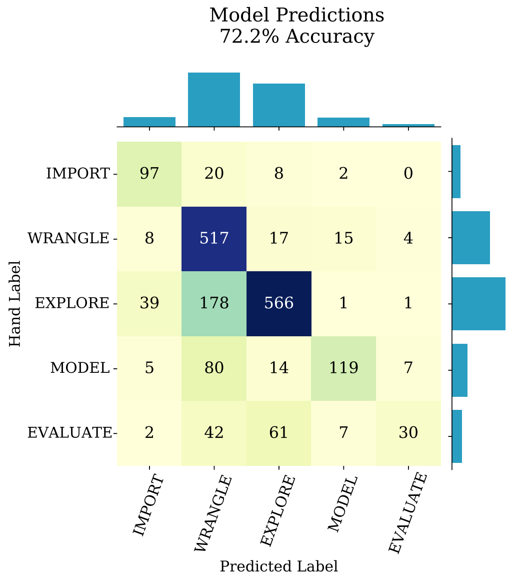
Explore	Interactive explorations of data. These cells tend to yield a result that informs later decisions, or enable the user to draw new conclusions. Explore cells may also transform data, but only for the purpose of exploring relationships and not for further in-depth analysis	Rendering DataFrames, 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)`

Table A3 (*Continued*)

Stage	Definition	When to Use	When Not to Use	Example

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")`



Hand Label \ Predicted Label	IMPORT	WRANGLE	EXPLORE	MODEL	EVALUATE
IMPORT	97	20	8	2	0
WRANGLE	8	517	17	15	4
EXPLORE	39	178	566	1	1
MODEL	5	80	14	119	7
EVALUATE	2	42	61	7	30

Figure A.2 Confusion matrix for CORAL's predictions on the data analysis stage prediction task, with marginal distributions for hand labels and model predictions shown

Funding
The work was supported by The National Science Foundation under NSF-IIS Large #1901386: Analysis Engineering for Robust End-to-End Data Science.

Availability of data and materials
All data and code used in this study are available at http://bdata.cs.washington.edu/coral/.

Declarations

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
Study concept and design: GZ, MAM, TA. Model design: GZ. Statistical analysis: MAM. Interpretation of data: All authors. Drafting of the manuscript: All authors. Critical revision of the manuscript for important intellectual content: All authors. All authors read and approved the final manuscript.

Publisher's Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 9 December 2020 Accepted: 1 March 2022 Published online: 18 March 2022

References
1. Kluyver T, Ragan-Kelley B, Pérez F, Granger BE, Bussonnier M, Frederic J, Kelley K, Hamrick JB, Grout J, Corlay S et al (2016) Jupyter notebooks-a publishing format for reproducible computational workflows. In: Positioning and power in academic publishing: players, agents and agendas
2. Foster ED, Deardorff A (2017) Open science framework (OSF). J. Med. Libr. Assoc. 105(2):203–206
3. Ayers P LibGuides: Citing & publishing software: Publishing research software
4. Pradal C, Varoquaux G, Langtangen HP (2013) Publishing scientific software matters. J Comput Sci 4(5):311–312
5. Liu Y, Althoff T, Heer J (2020) Paths explored, paths omitted, paths obscured: decision points & selective reporting in end-to-end data analysis. In: CHI
6. Rule A, Tabard A, Hollan JD (2018) Exploration and explanation in computational notebooks. In: CHI
7. Rehman MS (2019) Towards understanding data analysis workflows using a large notebook corpus. In: SIGMOD
8. Wang J, Li L, Zeller A (2020) Better code, better sharing: on the need of analyzing Jupyter notebooks. In: ICSE, pp 53–56
9. Wang J, Li L, Zeller A (2020) Better code, better sharing: on the need of analyzing Jupyter notebooks. In: ICSE
10. Kery MB, Radensky M, Arya M, John BE, Myers BA (2018) The story in the notebook: exploratory data science using a literate programming tool. In: CHI
11. Kandel S, Paepcke A, Hellerstein JM, Heer J (2012) Enterprise data analysis and visualization: an interview study. TVCG

12. Wongsuphasawat K, Liu Y, Heer J (2019) Goals, process, and challenges of exploratory data analysis: an interview study. 1911.00568
13. Alspaugh S, Zokaei N, Liu A, Jin C, Hearst MA (2018) Futzing and moseying: interviews with professional data analysts on exploration practices. IEEE Trans Vis Comput Graph 25(1):22–31
14. Johanson A, Hasselbring W (2018) Software engineering for computational science: past, present, future. Comput Sci Eng 20:90–109
15. Kery MB, Horvath A, Myers B (2017) Variolite: supporting exploratory programming by data scientists. In: CHI
16. Hill C, Bellamy R, Erickson T, Burnett M (2016) Trials and tribulations of developers of intelligent systems: A field study. In: VL/HCC. IEEE, Los Alamitos, pp 162–170
17. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: NeurIPS
18. Devlin J, Chang M, Lee K, Toutanova K (2018) BERT: pre-training of deep bidirectional transformers for language understanding. 1810.04805
19. Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J (2013) Distributed representations of words and phrases and their compositionality. In: NeurIPS
20. Tai KS, Socher R, Manning CD (2015) Improved semantic representations from tree-structured long short-term memory networks. CoRR. 1503.00075
21. Mou L, Li G, Jin Z, Zhang L, Wang T (2014) TBCNN: A tree-based convolutional neural network for programming language processing. CoRR. 1409.5718
22. Hindle A, Barr ET, Su Z, Gabel M, Devanbu P (2012) On the naturalness of software. In: ICSE
23. Tu Z, Su Z, Devanbu P (2014) On the localness of software. In: FSE
24. Nguyen TT, Nguyen AT, Nguyen HA, Nguyen TN (2013) A statistical semantic language model for source code. In: ESEC/FSE
25. Allamanis M, Sutton C (2013) Mining source code repositories at massive scale using language modeling. In: 2013 10th working conference on mining software repositories (MSR)
26. Raychev V, Vechev M, Yahav E (2014) Code completion with statistical language models. In: SIGPLAN
27. Kwon T, Su Z (2011) Modeling high-level behavior patterns for precise similarity analysis of software. In: ICDM
28. Movshovitz-Attias D, Cohen W (2013) Natural language models for predicting programming comments. In: ACL
29. Bielik P, Raychev V, Vechev M (2016) Phog: probabilistic model for code. In: ICML
30. Alon U, Zilberstein M, Levy O, Yahav E (2018) A general path-based representation for predicting program properties. In: SIGPLAN, pp 404–419
31. Li J, Wang Y, Lyu MR, King I (2017) Code completion with neural attention and pointer networks. 1711.09573
32. Allamanis M, Brockschmidt M, Khademi M (2017) Learning to represent programs with graphs. 1711.00740
33. Zhang Y, Xu FF, Li S, Meng Y, Wang X, Li Q, Han J (2019) Higitclass: keyword-driven hierarchical classification of github repositories. In: ICDM, pp 876–885
34. Shetty M, Bansal C, Kumar S, Rao N, Nagappan N, Zimmermann T (2020) Neural knowledge extraction from cloud service incidents. 2007.05505
35. Alon U, Zilberstein M, Levy O, Yahav E (2019) code2vec: learning distributed representations of code. POPL 3:40
36. Allamanis M, Barr E, Bird C, Sutton C (2014) Learning natural coding conventions. In: FSE
37. Acharya M, Xie T, Pei J, Xu J (2007) Mining api patterns as partial orders from source code: from usage scenarios to specifications. In: ESEC/FSE
38. Nguyen TD, Nguyen AT, Phan HD, Nguyen TN (2017) Exploring api embedding for api usages and applications. In: ICSE
39. Ratner A, Bach SH, Ehrenberg H, Fries J, Wu S, Ré C (2019) Snorkel: rapid training data creation with weak supervision. VLDB
40. Hamilton W, Ying Z, Leskovec J (2017) Inductive representation learning on large graphs. In: NeurIPS
41. Kipf TN, Welling M (2016) Semi-supervised classification with graph convolutional networks. 1609.02907
42. Wu M, Pan S, Zhu X, Zhou C, Pan L (2019) Domain-adversarial graph neural networks for text classification. In: ICDM
43. Schlichtkrull M, Kipf TN, Bloem P, Van Den Berg R, Titov I, Welling M (2018) Modeling relational data with graph convolutional networks. In: European semantic web conference
44. Zhang M, Chen Y (2018) Link prediction based on graph neural networks. In: NeurIPS
45. Defferrard M, Bresson X, Vandergheynst P (2016) Convolutional neural networks on graphs with fast localized spectral filtering. In: NeurIPS
46. Ying Z, You J, Morris C, Ren X, Hamilton W, Leskovec J (2018) Hierarchical graph representation learning with differentiable pooling. In: NeurIPS
47. Dai H, Dai B, Song L (2016) Discriminative embeddings of latent variable models for structured data. In: ICML
48. Duvenaud DK, Maclaurin D, Iparraguirre J, Bombarell R, Hirzel T, Aspuru-Guzik A, Adams RP (2015) Convolutional networks on graphs for learning molecular fingerprints. In: NeurIPS
49. Scarselli F, Gori M, Tsoi AC, Hagenbuchner M, Monfardini G (2008) The graph neural network model. IEEE Trans Neural Netw 20(1):61–80
50. Battaglia PW, Hamrick JB, Bapst V, Sanchez-Gonzalez A, Zambaldi V, Malinowski M, Tacchetti A, Raposo D, Santoro A, Faulkner R et al (2018) Relational inductive biases, deep learning, and graph networks. 1806.01261
51. Fernandes P, Allamanis M, Brockschmidt M (2018) Structured neural summarization. 1811.01824
52. Brockschmidt M, Allamanis M, Gaunt AL, Polozov O (2018) Generative code modeling with graphs. 1805.08490
53. Veličković P, Cucurull G, Casanova A, Romero A, Lio P, Bengio Y (2017) Graph attention networks. 1710.10903
54. Chen L, Ali Babar M, Nuseibeh B (2013) Characterizing architecturally significant requirements. IEEE Softw 30:38–45
55. Anonymous CORAL: COde RepresentAtion Learning with Weakly-Supervised Transformers for Analyzing Data Analysis. https://bit.ly/3hl1PUX
56. Agashe R, Iyer S, Zettlemoyer L (2019) JuICe: a large scale distantly supervised dataset for open domain context-based code generation. In: Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP). Assoc. Comput. Linguistics, Hong Kong, pp 5435–5445. https://doi.org/10.18653/v1/D19-1546. https://www.aclweb.org/anthology/D19-1546. Accessed 2019-12-03

57. Pimentel JF, Murta L, Braganholo V, Freire J (2019) A large-scale study about quality and reproducibility of Jupyter notebooks. In: 2019 IEEE/ACM 16th international conference on mining software repositories (MSR), pp 507–517. https://doi.org/10.1109/MSR.2019.00077. ISSN: 2574-3864
58. JetBrains Data Science in 2018 (2018). https://www.jetbrains.com/research/data-science-2018/
59. Kelley K, Granger B (2017) Jupyter frontends: from the classic jupyter notebook to jupyterlab, nteract, and beyond. JupyterCon
60. Landis J, Koch G (1977) The measurement of observer agreement for categorical data. Biometrics 33(1):159–174
61. Krippendorff K (2004) Content analysis: an introduction to its methodology, 2nd edn. Sage, Thousand Oaks
62. Gilmer J, Schoenholz SS, Riley PF, Vinyals O, Dahl GE (2017) Neural message passing for quantum chemistry. In: ICML
63. Ba JL, Kiros JR, Hinton GE (2016). Layer normalization. 1607.06450
64. Weston J, Bengio S, Usunier N (2011) Wsabie: scaling up to large vocabulary image annotation. In: IJCAI
65. Socher R, Karpathy A, Le QV, Manning CD, Ng AY (2014) Grounded compositional semantics for finding and describing images with sentences. TACL
66. Iyyer M, Guha A, Chaturvedi S, Boyd-Graber J, Daumé H III (2016) Feuding families and former friends: unsupervised learning for dynamic fictional relationships. In: NAACL-HLT
67. He R, Lee WS, Ng HT, Dahlmeier D (2017) An unsupervised neural attention model for aspect extraction. In: ACL
68. Blei DM, Ng AY, Jordan MI (2003) Latent Dirichlet allocation. J Mach Learn Res 3:993–1022
69. Merrill MA, Zhang G, Althoff T (2021) MULTIVERSE: mining collective data science knowledge from code on the web to suggest alternative analysis approaches. In: Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining. ACM, Singapore, pp 1212–1222. https://doi.org/10.1145/3447548.3467455. Accessed 2021-08-30
70. Chen M, Tworek J, Jun H, Yuan Q, Pinto HPdO, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A, Puri R, Krueger G, Petrov M, Khlaaf H, Sastry G, Mishkin P, Chan B, Gray S, Ryder N, Pavlov M, Power A, Kaiser L, Bavarian M, Winter C, Tillet P, Such FP, Cummings D, Plappert M, Chantzis F, Barnes E, Herbert-Voss A, Guss WH, Nichol A, Paino A, Tezak N, Tang J, Babuschkin I, Balaji S, Jain S, Saunders W, Hesse C, Carr AN, Leike J, Achiam J, Misra V, Morikawa E, Radford A, Knight M, Brundage M, Murati M, Mayer K, Welinder P, McGrew B, Amodei D, McCandlish S, Sutskever I, Zaremba W (2021) Evaluating Large Language Models Trained on Code. 2107.03374 [cs]. Accessed 2021-08-30
71. Lo K, Wang LL, Neumann M, Kinney R, Weld DS (2020) S2ORC: the semantic scholar open research corpus. In: ACL