Guided Data Science

Project Number: 18-1-1-1638

Project Report

By:

GUR YANIV 203347968

TAMIR HUBER 204252506

Advisor:

AMIT SOMECH TEL AVIV UNIVERSITY

Contents

Figures

ABSTRACT

We present a recommendation system for Data Scientists that given a user cell of code[1] will recommend what the next line of code should be.

The recommendation system is built of three main parts (that will be thoroughly explained later):

Data-set Builder[2]: Collects the necessary data to build our system.

Workflow-Stage Classifier[3]: Classifies the code to the relevant Data Science workflow stage and provides context to the code.

Recommendation Engine[4]: Generates the next-line recommendation.

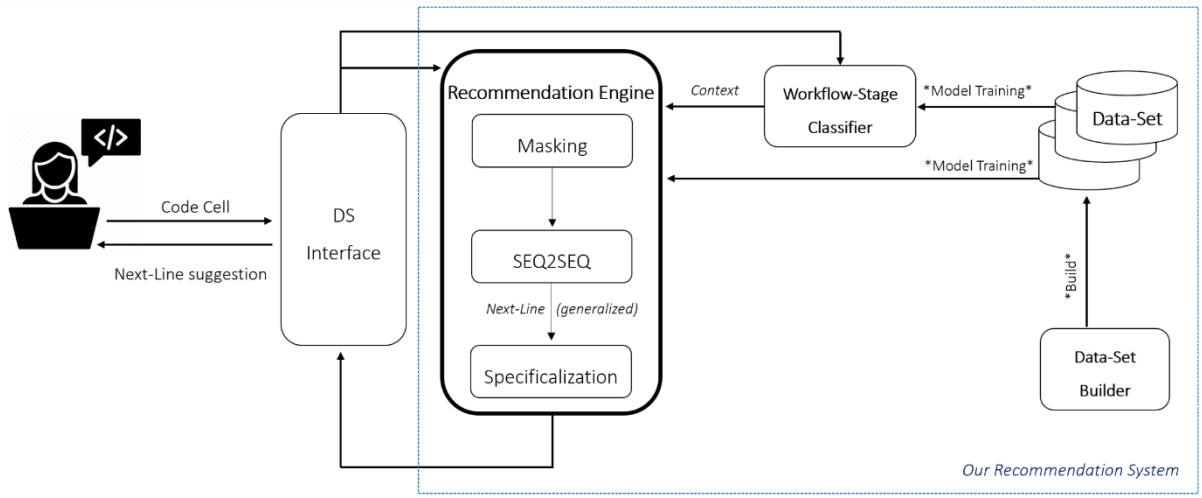


Figure 1. The system architecture scheme

[1] See section 2.1 for more details

[2] See section 3.1 for more details

[3] See section 3.2 for more details

[4] See section 3.3 for more details

1 INTRODUCTION

Data Science is an interdisciplinary field about scientific methods, processes and systems to extract knowledge or insights from data.
It is now increasingly common for many aspects of business decisions to be guided in some form by data.

The data scientists' workflow[5] is consisted of data gathering, data exploration, data preparation and cleaning, feature engineering, training a model and tuning its parameters, evaluating the model etc. until they achieve the business goals and the model is deployed.

It is a difficult task for several reasons:

- Data scientists explore Big Data:
  Volume, Velocity, Variety, but also: Veracity and Volatility. There is an enormous amount of data, from different sources, with different reliability and accuracy.
- The data sometimes could be "dirty" or missing.
- Lack of understanding of the investigated domain may lead to poor results, wrong/irrelevant insights.
- Best or common practices are not easily accessible or shared.

Although the field is advancing rapidly, there are still large inefficiencies in the process. Even though there are many existing platforms to process big data or develop models, the knowledge sharing efforts aren't adequate for different problems.

We want to help, by creating a recommendation system for data scientists, utilizing existing knowledge sharing platforms, that will:

1. Hook in the data science UI
2. "Understand" the dataset and task
3. Leverage other people's relevant knowledge
4. Automatically adapt it to fit in the current user's context.
5. Suggest adequate, context-sensitive, code-snippets.

To design such a framework, we need:

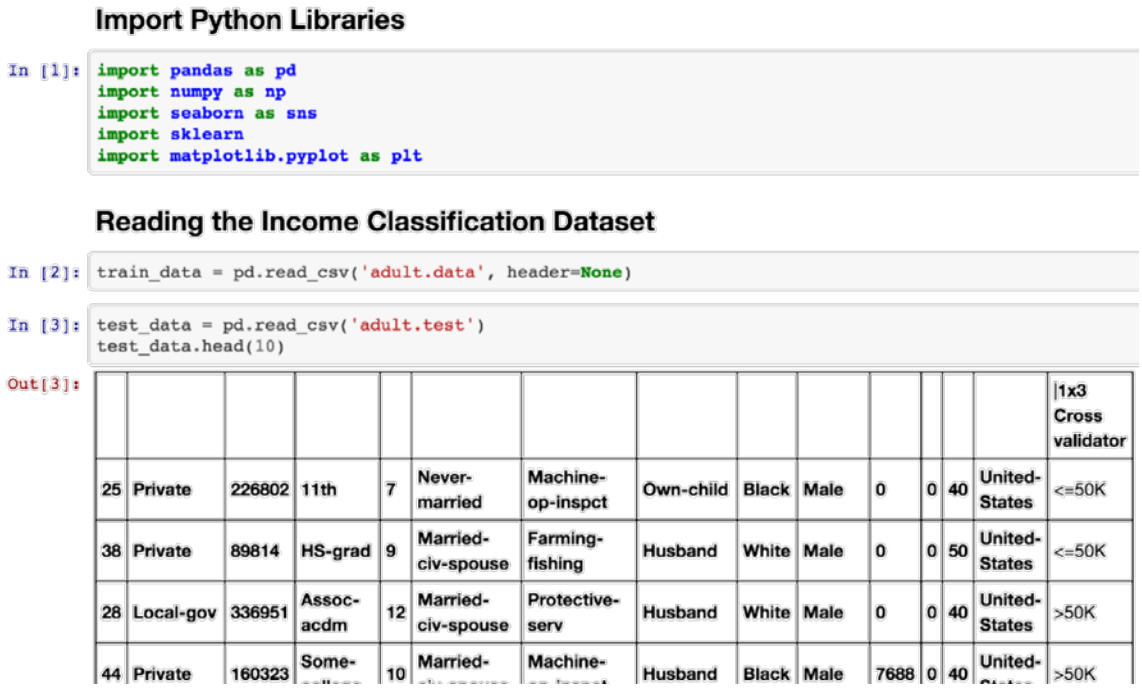
- A repository of reproducible data science **code-snippets**.
- A method for analyzing the user's **context.**
- A similarity notion and an efficient algorithm for **identifying similar code-snippets**
- A procedure that takes a code-snippet and **adapt it to the user's context**

[5] See section 2.3 for more details

2 THEORETICAL BACKGROUND

2.1 JUPYTER NOTEBOOKS

Jupyter notebooks are web based interactive scripting environments. They contain computer code (e.g. python) and rich text elements (paragraph, equations, figures, links, etc.).
Notebook documents are both human-readable documents containing the analysis description and the results (figures, tables, etc..) as well as executable documents which can be run to perform data analysis.
The notebooks are constructed of markdown (text) cells, code **cells** and their run output.
They are extremely popular among data scientists, as they tie together both the code snippets and the results, and even better- they are reproducible!



Import Python Libraries

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import sklearn
import matplotlib.pyplot as plt
```

Reading the Income Classification Dataset

In [2]:
```
train_data = pd.read_csv('adult.data', header=None)
```

In [3]:
```
test_data = pd.read_csv('adult.test')
test_data.head(10)
```

Out[3]:

														1x3 Cross validator
25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K
38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K
28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K
44	Private	160323	Some-	10	Married-	Machine-	Husband	Black	Male	7688	0	40	United-	>50K

Figure 2. A Jupyter Notebook

2.2 KAGGLE

Kaggle is an online community of data scientists and machine learners, owned by Google LLC. Kaggle allows users to find and publish data sets, explore and build models in a web-based data-science environment, work with other data scientists and machine learning engineers, and enter competitions to solve data science challenges. Kaggle got its start by offering machine learning competitions and now also offers a public data platform, a cloud-based workbench for data science, and short form AI education.
It is a publicly available source for datasets and Jupyter notebooks.

2.3 DATA SCIENCE WORKFLOW

Cross-industry standard process for data mining, known as CRISP-DM, is an open standard process model that describes common approaches used by data mining experts. It is the most widely-used analytics model. CRISP-DM breaks the process of data mining into six major phases:

- Business Understanding:
  What is Goal of the project? Why is it necessary? What has already been done? Is it a competition? Is there a timeline?
- Data Understanding:
  What can be achieved? What are the low hanging fruits? How can the problem be addressed? What is the cost/effort of dealing with or getting more data?
- Data Preparation:
  Data Cleaning (missing values, outliers), Data discretization, Data normalization, Dimensionality reduction (Feature selection).
- Modeling:
  Build Model, Train Model, Parameter Tuning.
- Evaluation:
  Evaluate the model's performance, does it meet the goals?

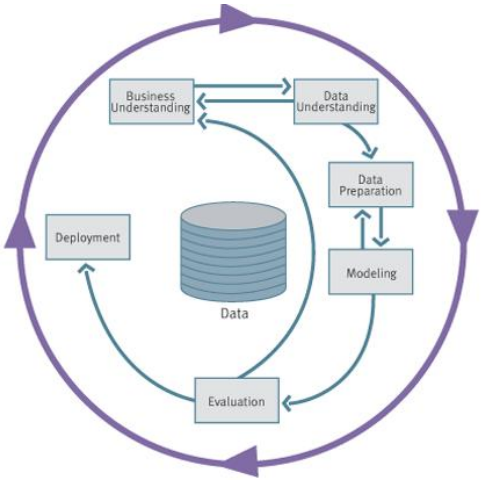


Figure 3. CRISP-DM workflow

3 IMPLEMENTATION

As mentioned, our recommendation system is built of three main parts. In this section we will explain those parts' implementation thoroughly, providing additional background other than what was previously explained, if necessary.

3.1 DATASET BUILDER

As mentioned in the introduction, we need to create a repository of reproducible data science code-snippets, so that we will be able to utilize it to understand the context of a given code cell and generate next-line recommendations.

In the "heart" of our system are machine learning models, that require a big amount of data, so we need this repository to contain as much data as possible.

In order to create such a repository, we used Kaggle[6].

Kaggle provides its own API, that helps to interact with its data and features. But using the API by itself is not enough to collect data extensively, since Kaggle API allows you to download a notebook only if you have a direct URL link to it.

We used a web-crawling algorithm in order to collect those links.

Each notebook in Kaggle "belongs" to a certain dataset, meaning the notebook contains an exploration of the data from that dataset or uses the data to create predictive models or solve some problem. The first step in our algorithm was to collect Kaggle's datasets and competition which has a minimum number of notebooks (we choose 20 to be the minimum amount). We used Kaggle API, which allows us to use the search feature in order to search for datasets by keywords, and for each dataset get the kernels (notebooks) amount.

The second step was to get the kernels (notebooks) URL links for each of the datasets. Since Kaggle API doesn't allow this (probably in order to prevent massive data gathering from it, which is what we are trying to achieve), we used scraping methods, using Selenium framework and WebDriver API. The WebDriver browses into each of the datasets kernels pages (a list of all the notebooks that belong to a certain dataset), fetches all the URL links of the kernels (notebooks), and saves them into a configuration text file.

The WebDriver also collects important metadata regarding those datasets while fetching the links (datasets overview, input, tags, etc.).

Next, we used Kaggle API again to physically download the kernels (as we already got the URL links by that point). Since the API prevents using multiple requests in a small time margin, the algorithm sleeps between requests and also updates two different configurations- one keeps track of which kernel has already been downloaded, and the second one keeps track of which kernel was not

[6] See section 2.2 for more details

downloaded successfully (there is a variety of possible reasons to that), so we were able to split this step over several runs.

The last step was to execute each one of the downloaded notebooks, so they would contain the cells' outputs. Again, by using WebDriver we developed a python script which accesses each of the Jupyter notebooks and simulates a user clicking on "Run all cells".

Using the dataset Builder, we've created a dataset of 146 Datasets, 19,081 Jupyter notebooks, 296,281 Cells of code.

3.2 WORKFLOW STAGE CLASSIFIER

Now that we have a dataset (built using the dataset builder), we want to utilize it to create a method for analyzing the context of a given user code.

We decided to create a classifier that given a user cell of code will classify it to the relevant Data Science workflow stage[7].

We narrowed the stages into six classes:

- Imports: cells that contain only imports and have no other purpose but import modules.
- Load Data: cells that load data from the dataset.
- Data Exploration: cells that their purpose is to explore and understand the data.
- Data Preparation: cells that manipulate the data, cleaning it and performing different actions.
- Model Training and Parameter Tuning: cells where models are created and trained, or their parameters and definitions are tuned.
- Model Evaluation: cells for evaluating an existing model and its outputs.

Today's state-of-the-art machine learning models require massive labeled data to train. Since our collected data is unlabeled (the class is unknown for each cell to begin with), we decided to use a new paradigm called weak supervision, that uses “data programming” to create large training sets quickly[1]. This paradigm is implemented by Stanford’s HazyResearch group, led by Prof. Chris Ré, in Snorkel[2] . In Snorkel, the developer focuses on defining a set of labeling functions, which are just scripts that programmatically label data in a binary manner (is/isn’t in this class). The resulting labels are noisy, as functions aren’t always correct and there are a lot of “collisions” between different labeling functions, but Snorkel automatically models this process, using its generative model-learning, essentially, which labeling functions are more accurate than others, and assigning weights accordingly. Snorkel provides a general framework for many weak supervision techniques, and as defining a new programming model for weakly-supervised machine learning systems.

We defined 41 different labeling functions to tag the unlabeled data with the relevant workflow stage, as mentioned before. In addition, we tagged ~1000 cells manually.

Snorkel automatically learned a generative model over the labeling functions, estimated their accuracies and correlations, learning from the agreements and disagreements of the labeling functions (Snorkel also provides the user with actionable feedback about labelling function quality, which allowed us to improve the functions accordingly).

Snorkel generative model[3] outputs the “marginals”- the probability that the cell belongs to each of the classes (stages). We than label the data according to the highest probability, getting a labeled dataset for our “end-model” which is the classifier itself.

---

[7] See section 2.3 for more details

Data pre-processing:

For each cell source code, we removed all comments, as comments may refer to actions that weren't really done or to what was done previously to the current cell, so that it just interferes in our task to classify the current cell correctly.

The data (source code) was normalized and tokenized- we turned the code to-lower, converted special chars and dots to spaces, than split what's left to tokens by spaces.

Data Vector Representation:

Each cell code (after being normalized and tokenized) is represented as a vector of integers that represent the tokens, when only the 8,000 most common tokens gets to be represented, as the less common ones are probably useless for our purpose (meaningless names etc.).

The vectors are than padded to a fixed max_len of 120 (we looked at the common code length and set the max_length accordingly, so that only outliers will be cut).

There is a further embedding of these vectors in the model itself (in the embedding layer).

End Model – LSTM:

For our task we need persistence – to consider not only the latest input but also the previous inputs when tagging, we cannot only process single data points, but entire sequences of data. Thus, we need a recurrent neural network.

There is also a long-term dependency between "words" within a cell's source code and between different cells – we need more context. That's why LSTM is the preferred model for our purpose (simple RNN is uncapable of learning long-term dependencies).

LSTMs[4] are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior.

Our LSTM model was built using Keras Deep Learning library.

The model layers are as follows:

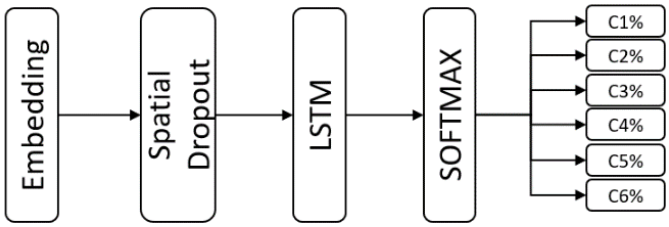


Figure 4. Classifier model layers

As mentioned, the training set contains vectors of integers that represent the code and one-hot encoding of the label.

The Embedding layer[5] is used as the first layer. It turns positive integers into dense vectors of fixed size. It resembles the word2vec operation by creating word embeddings. Next layers use the embedding matrix to keep the size of each vector much smaller.

The second layer is a Spatial Dropout layer. It drops entire 1D feature maps (instead of individual elements in a regular dropout layer). If adjacent frames within feature maps are strongly correlated (as is normally the case in early convolution layers) then regular dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease. In this case, SpatialDropout1D will help promote independence between feature maps and should be used instead. This layer helps preventing overfitting.

The third layer is an LSTM layer, as described.

The fourth layer is a Softmax layer (implemented by keras dense layer). It outputs the Softmax transformation results which are the outputs of our model – probability of the cell to belong to each of the classes.

Our loss function is Categorical Cross-Entropy. The model is optimized according to this loss function. In addition, when we train the model, we set an EarlyStopping parameter that stops the training if the validation loss stops improving. Cross-Entropy is the better option for classification models with a Softmax output node[6].

After Training, we consider the model's prediction for a given input cell as the class with the highest probability.

3.3 RECOMMENDATION ENGINE

Now, that we have a repository of reproducible data science code-snippets (cells) built using the dataset builder, and we also have a method for analyzing the context of a given cell (classifying it to the relevant workflow stage, using the workflow stage classifier), we want to create a recommendation system, that given a user's cell of code will output adequate, context-sensitive, next-line recommendations.

In the heart of our recommendation engine is a sequence-to-sequence (chatbot) model (more accurately- 6 different models, as we'll explain).

Sequence-to- Sequence

Conversational models are a hot topic in artificial intelligence research. Chatbots can be found in a variety of settings, including customer service applications and online helpdesks. These bots are often powered by retrieval-based models, which output predefined responses to questions of certain forms. Recently, the deep learning boom has allowed for powerful generative models like Google's Neural Conversational Model[7], which marks a large step towards multi-domain generative conversational models. The goal of a sequence-to-sequence model is to take a variable-length sequence as an input and return a variable-length sequence as an output using a fixed-sized model.

Sutskever et al. discovered[8] that by using two separate recurrent neural networks together, we can accomplish this task. One RNN acts as an encoder, which encodes a variable length input sequence to a fixed-length context vector. In theory, this context vector (the final hidden layer of the RNN) will contain semantic information about the query sentence that is input to the bot. The second RNN is a decoder, which takes an input word and the context vector, and returns a guess for the next word in the sequence and a hidden state to use in the next iteration.

We will implement this kind of model in PyTorch. In our case instead of spoken language we'll use a programming language, specifically python Jupyter notebook cells.

The model input is a code of cell and its output is a next-line recommendation (first it was meant to be a next cell recommendation, as we'll describe next).

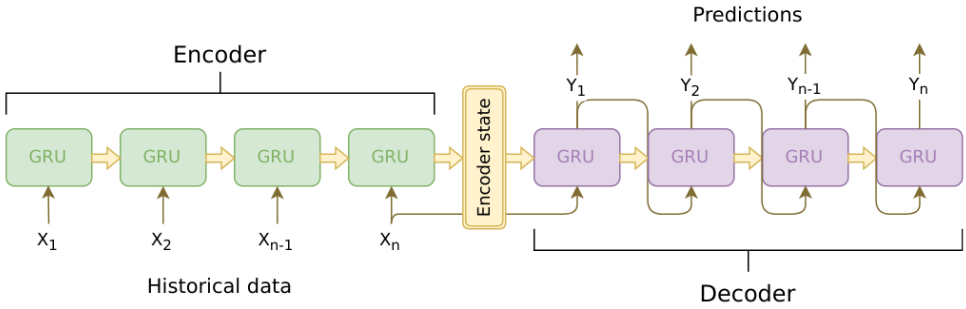


Figure 5. sequence-to-sequence model

Encoder

The encoder RNN iterates through the input sentence one token (e.g. word) at a time, at each time step outputting an “output” vector and a “hidden state” vector. The hidden state vector is then passed to the next time step, while the output vector is recorded. The encoder transforms the context it saw at each point in the sequence into a set of points in a high-dimensional space, which the decoder will use to generate a meaningful output for the given task.

At the heart of our encoder is a multi-layered Gated Recurrent Unit, invented by Cho et al. in 2014[9]. We will use a bidirectional variant of the GRU, meaning that there are essentially two independent RNNs: one that is fed the input sequence in normal sequential order, and one that is fed the input sequence in reverse order. The outputs of each network are summed at each time step. Using a bidirectional GRU will give us the advantage of encoding both past and future context.

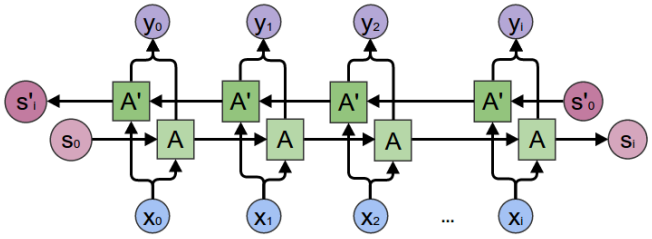


Figure 6. bidirectional RNN

Computation Graph:

1. Convert word indexes to embeddings.
2. Pack padded batch of sequences for RNN module.
3. Forward pass through GRU.
4. Unpack padding.
5. Sum bidirectional GRU outputs.
6. Return output and final hidden state.

Decoder

The decoder RNN generates the response sentence in a token-by-token fashion. It uses the encoder’s context vectors, and internal hidden states to generate the next word in the sequence. It continues generating words until it outputs an EOS_token, representing the end of the sentence. A common problem with a vanilla seq2seq decoder is that if we rely solely on the context vector to encode the entire input sequence’s meaning, it is likely that we will have information loss. This is especially the case when dealing with long input sequences, greatly limiting the capability of our decoder.

To combat this, Bahdanau et al. created an “attention mechanism”[10] that allows the decoder to pay attention to certain parts of the input sequence, rather than using the entire fixed context at every step.

At a high level, attention is calculated using the decoder's current hidden state and the encoder's outputs. The output attention weights have the same shape as the input sequence, allowing us to multiply them by the encoder outputs, giving us a weighted sum, which indicates the parts of encoder output to pay attention to. Sean Robertson's figure describes this very well:

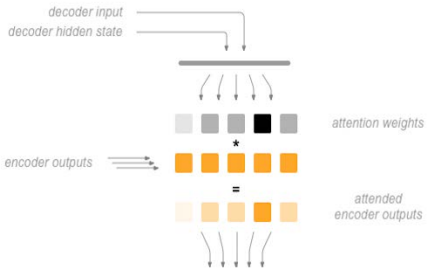


Figure 7. Attention mechanism

Luong et al. improved upon Bahdanau et al.'s groundwork by creating "Global attention" [11]. The key difference is that with "Global attention", we consider all of the encoder's hidden states, as opposed to Bahdanau et al.'s "Local attention", which only considers the encoder's hidden state from the current time step. Another difference is that with "Global attention", we calculate attention weights, or energies, using the hidden state of the decoder from the current time step only. Bahdanau et al.'s attention calculation requires knowledge of the decoder's state from the previous time step. Also, Luong et al. provides various methods to calculate the attention energies between the encoder output and decoder output which are called "score functions":

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W}_a \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W}_a [\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

where h_t = current target decoder state and $\bar{h}_s$ = all encoder states.

Overall, the Global attention mechanism can be summarized by the following figure:

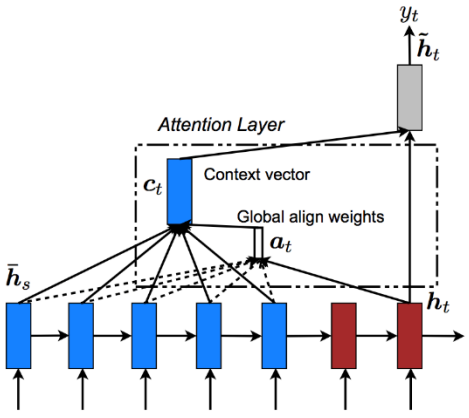


Figure 8. Global Attention Mechanism

Loss

Since we are dealing with batches of padded sequences, we cannot simply consider all elements of the tensor when calculating loss. We define a mask loss to calculate our loss based on our decoder's output tensor, the target tensor, and a binary mask tensor describing the padding of the target tensor. This loss function calculates the average negative log likelihood of the elements that correspond to a 1 in the mask tensor. Essentially, we want to minimize the cross-entropy one again, as the cross-entropy lowers when the model's output is more similar to the label (real next cell or line).

Training

Training is done in iterations (each iteration gets a single batch of inputs).

We use a couple of clever tricks to aid in convergence:

- Teacher Forcing:

  This means that at some probability, set by us, we use the current target word as the decoder's next input rather than using the decoder's current guess. This technique acts as "training wheels" for the decoder, aiding in more efficient training. However, teacher forcing can lead to model instability during inference, as the decoder may not have a sufficient chance to truly craft its own output sequences during training. Thus, we must be mindful of how we are setting the teacher forcing ratio, and not be fooled by fast convergence.

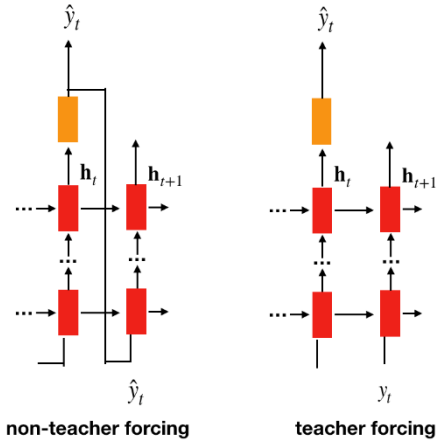


Figure 9. Teacher Forcing

- Gradient Clipping:
  This is a commonly used technique for countering the “exploding gradient” problem. In essence, by clipping or thresholding gradients to a maximum value, we prevent the gradients from growing exponentially and either overflow (NaN), or overshoot steep cliffs in the cost function.

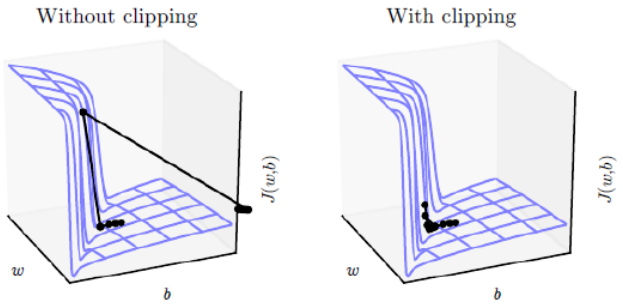


Figure 10. Gradient Clipping

Training Sequence of Operations:

1. Forward pass entire input batch through encoder.
2. Initialize decoder inputs as SOS_token, and hidden state as the encoder’s final hidden state.
3. Forward input batch sequence through decoder one time step at a time.
4. If teacher forcing: set next decoder input as the current target; else: set next decoder input as current decoder output.
5. Calculate and accumulate loss.
6. Perform backpropagation.
7. Clip gradients.
8. Update encoder and decoder model parameters.

Data pre-processing

To Train our model we create a nicely formatted data file, from our dataset, in which each line contains a tab-separated query and a response pair. To begin with:

***CELL* /t *NEXTCELL* /n**

Next we decided to use different representations of the code (not the original source code as is), as we'll explain below. We also decided to focus on next lines recommendations, so we instead of cell pairs, in the final pair files the structure is as follows:

***3-LINES Masked Representation* /t *NEXT-LINE Masked Representation * /n**

In the final models, such a file was created for each workflow stage of the input cell (6 different files), when we also kept only the 3 most common outputs for each input.

After creating the pairs files, our next order of business is to create a vocabulary from the file and load query/response pairs into memory.

Note that we are dealing with sequences of code words, which do not have an implicit mapping to a discrete numerical space. Thus, we must create one by mapping each unique word that we encounter in our dataset to an index value.

For this we defined a Voc class, which keeps a mapping from words to indexes, a reverse mapping of indexes to words, a count of each word and a total word count. The class provides methods for adding a word to the vocabulary (addWord), adding all words in a sentence (addSentence) and trimming infrequently seen words (trim).

Using this class, we assemble our vocabulary and query/response sentence pairs. Before we are ready to use this data, we must perform some pre-processing.

First, we must convert the Unicode strings to ASCII using unicodeToAscii. Next, we should convert all letters to lowercase and trim all non-letter characters except for basic punctuation. Finally, to aid in training convergence, we will filter out sentences with length greater than 20 tokens. Another tactic that is beneficial to achieving faster convergence during training is trimming rarely used words out of our vocabulary. Decreasing the feature space will also soften the difficulty of the function that the model must learn to approximate. We will do this as a two-step process:

1. Trim tokens that appear less than three times.
2. Filter out pairs with trimmed words.

Finally, we need to convert the cells or lines to numerical torch tensors, to be used as inputs to our encoder, in a similar manner as described before, padding them to a fixed max length of 20 tokens.

Code Masking

Our first tries to rain the model, using the entire source code as input, were futile. We got unstructured recommendations with a lot of repeating tokens and irrelevant data.

In order to provide some structure, we tried to use the code's **AST**. **A**bstract **S**yntax **T**ree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Using the AST we can understand the structure of a code snippet and also understand which part is more relevant. Using Python's ast module we converted the cells' source code into AST pairs and trained the model, but then the model learned the structure and not the recommendation.

So, we needed a representation of the code that will keep only the relevant data and provide us with some structure, That's the Masking- using the AST we extract only necessary information- libraries and functions and which variables they are operating on.

There is a delicate tradeoff:

- Not enough data – too many identical cells and different outputs for same input, can't "reconstruct" the code line to give a working recommendation (for example keeping only function names without any parameters or context). Recommendations aren't helpful.
- Too much data – model learns irrelevant stuff or doesn't learn well at all, bad for prediction (like using the entire AST, the model learns the structure but misses the recommendations

In addition, In the masking process we want to keep track of original variable names, so we can tailor our recommendation to the user (specificalization: output a ready-to-execute recommendation), so when converting the code into the summarized representation we keep a variable dictionary.

Recommendation Engine

We wanted to use the context provided by the workflow stage classifier to reduce loss and achieve better recommendations. In order to do so we classified all our dataset using the classifier and trained a different model for each stage of the input cell (3-lines → next-line pairs of masked representation). The model to use is chosen according to the input.

The Recommendation Engine Flow:

OFFLINE:

1. Cells were classified using the workflow stage classifier
2. Different model was trained for each stage (of the input cell)

ONLINE:

1. The user's code is turned into a summed representation using its AST, Variable names are kept in a dictionary
2. The user's code is classified and its representation is passed to the relevant model
3. The model outputs a next-cell recommendation
4. Specificalization: The recommendation is personalized using the Variables dictionary
5. Output: a ready-to-execute next cell recommendation

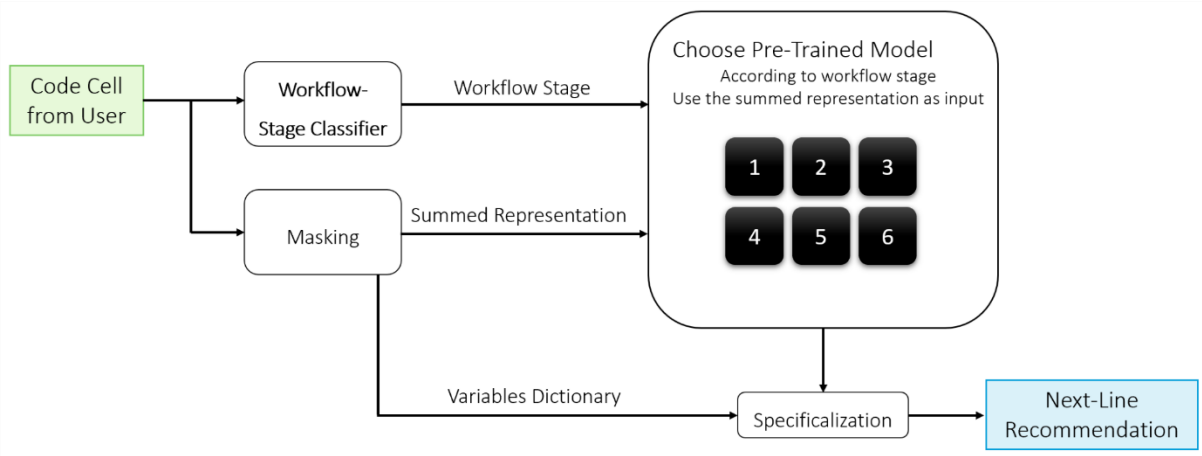


Figure 11. Recommendation Engine

4 EVALUATION

4.1 WORKFLOW STAGE CLASSIFIER

Weak Supervision evaluation:

We check the Categorical Accuracy over the test set – how many cells where tagged right (according to the hand-labeled data). We achieved an accuracy of 82.3%.

The classification report:

		precision	recall	f1-score	support
Load	1.0	0.71	1.00	0.83	29
Prep	2.0	0.96	0.69	0.80	96
Train	3.0	0.94	0.89	0.92	55
Eval	4.0	0.79	0.70	0.75	44
Exp	5.0	0.72	0.91	0.81	89
Import	6.0	1.00	1.00	1.00	10
micro avg		0.82	0.82	0.82	323
macro avg		0.85	0.87	0.85	323
weighted avg		0.85	0.82	0.82	323

We can see that we label some of the data preparation cells as data exploration, as data preparation recall is lower and data exploration precision is lower, but that's a hard task. Overall the results are pretty good.

End Model evaluation:

- Again, we look at the Categorical Accuracy over the test set (it's a different test set than before of course, we train-test-split the tagged data). We achieved an accuracy of 86.7%.

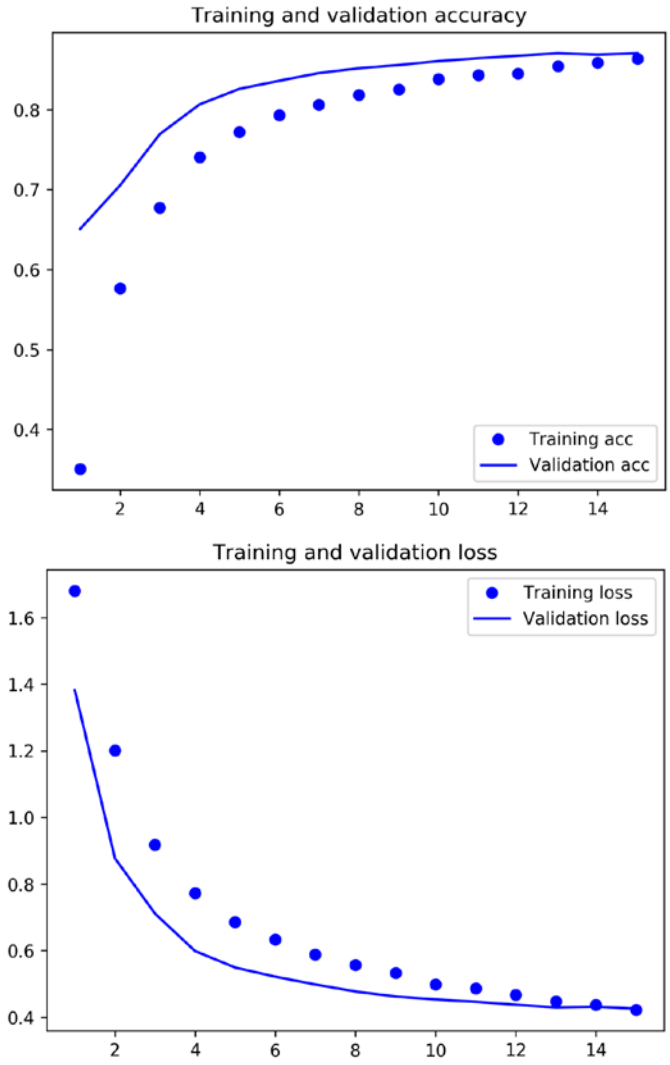
  The classification report:

		precision	recall	f1-score	support
Load	0	0.94	0.92	0.93	1247
Prep	1	0.82	0.84	0.83	1213
Train	2	0.86	0.89	0.87	1207
Eval	3	0.87	0.82	0.85	1257
Exp	4	0.83	0.83	0.83	1286
Import	5	0.89	0.90	0.89	1246
micro avg		0.87	0.87	0.87	7456
macro avg		0.87	0.87	0.87	7456
weighted avg		0.87	0.87	0.87	7456

  We can see the model performs well for all labels.

- Our loss function is Categorical Cross-Entropy. The model is optimized according to this loss function. In addition, when we train the model, we set an EarlyStopping parameter that stops the training if the validation loss stops improving.

We also looked at the loss and accuracy on the train set and the validation set, when the loss is much lower and the accuracy is much higher for the train set, that means we overfitted, so we took that into account when setting the dropout and the number of epochs.
The model convergence graphs:



We stop at 15 epochs to not overfit the model.
We also look at the Mean squared error. We can see that the MSE is low and it is similar between the test set and the validation set.

- Finally, we evaluated “by-hand”. Just ran some examples and see that the models classify correctly.

4.2 RECOMMENDATION ENGINE

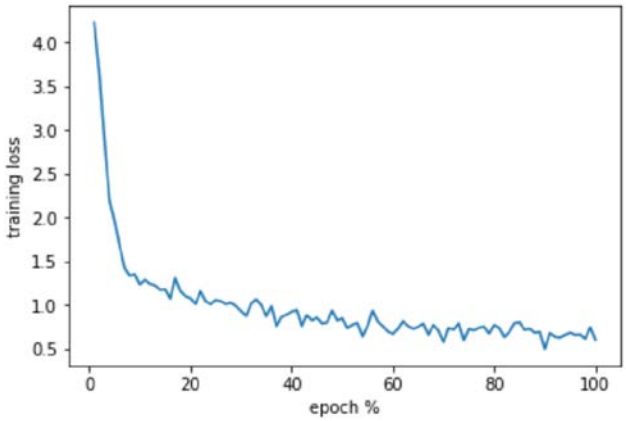
As mentioned in section 3.3 we used Masked cross-Entropy loss optimization to train our sequence-to-sequence models.
In addition to the loss value we compared between the different models using BLUE Score[12].
The **Bil**ingual **E**valuation **U**nderstudy is a score for comparing a candidate text to one or more reference texts. Although developed for translation, it's widely used to evaluate text generated for a suite of NLP tasks.
However, the most important evaluation is manual evaluation – "chatting" with the model and examining its recommendations for our inputs.

First Try

First, we used pairs of the cells source code "as is" as input.

Results: We achieved a low loss of 0.73 but a BLEU score of 0, not even close.

Model convergence graph:



Recommendation Example:

input: df_train = pd.read_csv('../input/20-newsgroups-ciphertext-challenge/train.csv') df_train.head()
output: loadintraining loadintraining imagens imagens fillinh fillinh fillinh faces finance imagens imagens fillinh ysize quarters quarters leftjoin .bandgap datasetexporter krr krr allimage allimage allimage floatcols encipher rod earlystop rod .lotsizesquarefeet bincolumns bincolumns airport startweights startweights hiddenunit atomic imagens imagens imagens .cvr fillinh fillinh faces finance officer cd loadintraining loadintraining titlecat development filtereddf imagens imagens imagens .cvr mhm applicaton loadintraining imagens imagens titlecat imagens .cvr applicaton titlecat filtereddf imagens imagens imagens .cvr applicaton titlecat filtereddf imagens imagens imagens .cvr applicaton titlecat filtereddf imagens imagens imagens .cvr applicaton titlecat filtereddf imagens imagens imagens .cvr applicaton titlecat filtereddf imagens imagens imagens .cvr applicaton titlecat
real next: df_test = pd.read_csv('../input/20-newsgroups-ciphertext-challenge/test.csv') df_test.head()

Problems:

- After debugging we figured that teacher forcing ratio[8] was 100%, that means the "training wheels" didn't allow the model to learn at all, the loss and convergence are deceiving in that case.
- We could see that there is too much irrelevant data, unnecessary for our purpose of generating recommendations, and it makes it harder for the model to train.

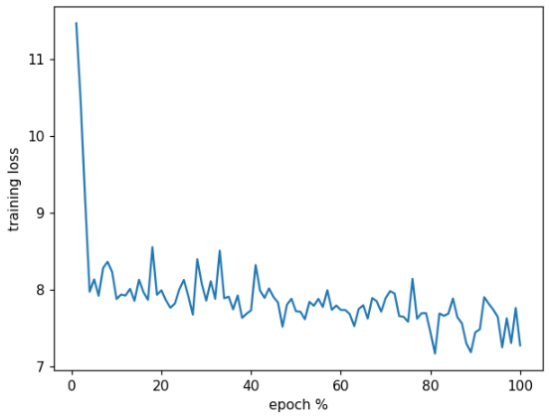
[8] See training section in 3.3 for more details about teacher forcing

Second Try

Again, we used the cells source code as input, but now we lowered the teacher forcing ratio and trimmed rare tokens.

Results: High loss of 7.27 (but now the model actually trains), BLEU score of 5.76e-232 (better than 0). More like it… still bad

Model convergence graph:



We can see the convergence is still odd, quick but to a high loss.

Recommendations examples (chatting with the model):

```
> import numpy as np
Bot: train pd .read csv . . input train .tsv sep t test pd .read csv . . input test .tsv sep t np .nan test id
> read_csv(data.csv)
Bot: import matplotlib .pyplot as plt plt .style .use seaborn plt .ylabel frequency . . . . . .
> df.head()
Bot: plotpercolumndistribution df np .sum np .sum np .sum np .sum np .mean np .array df .columns df .columns df .columns df
.columns df .columns
```

Problems:

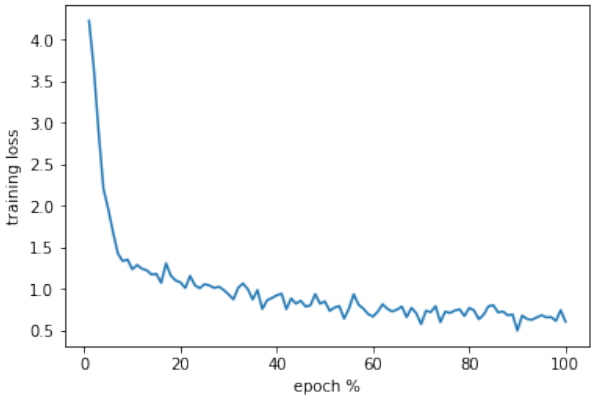
- Model gets fixated on certain tokens and returns them again and again
- Still there is irrelevant data, even after trimming rare tokens.
- The recommendations are out of structure (we want a ready to run recommendation)

Third Try

In order to provide structure to the recommendations we tried to use the cells AST[9] as input.

Results: Low loss of 0.64, BLEU score of 1.17e-231

Model Convergence graph:



Recommendation example:

input: Module(body=[Assign(targets=[Name(id='news_list', ctx=Store())],
value=Call(func=Attribute(value=Name(id='pd', ctx=Load()), attr='read_csv', ctx=Load()), args=[Str(s='../input/20-newsgroups/list.csv')], keywords=[])),
Expr(value=Attribute(value=Name(id='news_list', ctx=Load()), attr='shape', ctx=Load()))])

output: module body expr value call func attribute value name id learner ctx load attr fit ctx load args name id lr ctx load num n keywords EOS EOS expr value call func attribute value name id learner ctx load attr fit ctx load args name id lr ctx load num n keywords EOS EOS expr value call func attribute value name id learner ctx load attr fit ctx load args name id lr ctx load num n keywords EOS EOS EOS EOS EOS EOS EOS EOS EOS EOS EOS EOS EOS expr value call func attribute value name id learner

real next: Module(body=[Expr(value=Call(func=Name(id='print', ctx=Load()), args=[Attribute(value=Name(id='df_train', ctx=Load()),

Problems:

- The model learns the structure tokens instead of the recommendations, therefore is useless for our purpose.

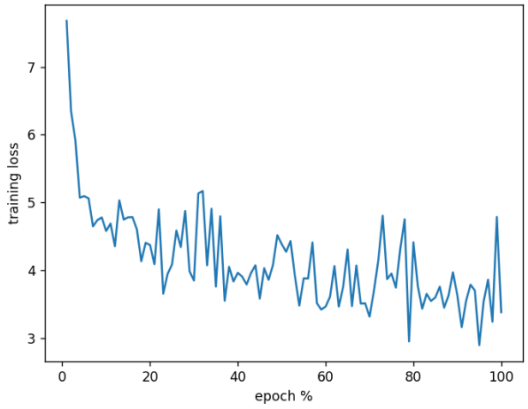
[9] See Data Pre-Processing section in 3.3 for more details about ASTs

Fourth Try

In order to provide the model with some structure while still allowing it to learn relevant data, we created a masked[10] representation of the code using its AST.
Our pairs are now cell pairs of masked representation.

Results: Loss of 3.41, BLEU score of 5.45e-3 (magnitudes of order better).
Not too bad, some correct recommendations

Model Convergence graph:



Recommendation Examples:

input: import_datetime import_numpy import_os import_pandas **output:** var0=pandas.read_csv **real next:** var0=pandas.read_csv var0.head
input: var0=pandas.read_csv var0.head **output:** var1=pandas.read_csv var1.head **real next:** var1=pandas.read_csv var1.head
input: var1=pandas.read_csv var1.head **output:** var2=pandas.series matplotlib.pyplot.figure matplotlib.pyplot.hist matplotlib.pyplot.yscale matplotlib.pyplot.title matplotlib.pyplot.xlabel matplotlib.pyplot.ylabel matplotlib.pyplot.ylabel matplotlib.pyplot.ylabel matplotlib.pyplot.ylabel **real next:** var2=pandas.read_csv
input: var5.fit **output:** nb_end **real next:** var6=var3.transform

Problems:

- The model is" eager" to end the notebook (outputs NB_END).
- We get Long recommendations. The model doesn't learn well enough where to stop, there are still repeating tokens.
- It Gets the "easy" ones (the most frequent), what about the rest?

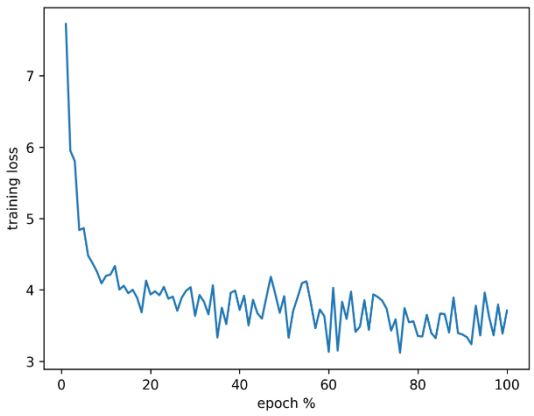
[10] See Masking section in 3.3 for more details about the masked representation

Fifth try

To prevent the model from ending the notebook we removed all empty output cell pairs, since they are irrelevant for our recommendation.
To prevent long and repeating recommendations and get more accurate recommendations we decided to focus on next-line recommendations instead of next-cell recommendations.
Our input is now pairs of 3-lines and next-line of masked representation.

Results: Loss of 3.71, BLEU score of 4.03e-232 (BLEU score is less relevant in that case, since there are no n-grams, but a single token)
Looks better…

Convergence graph:



Recommendation Examples:

input: var0=pandas.read_csv var0.head var1=pandas.read_csv **output:** var1.head **real next:** var1.head
input: var1.head var2=pandas.read_csv matplotlib.pyplot.figure **output:** seaborn.barplot **real next:** seaborn.barplot
input: var23=lightgbm.Dataset var24=lightgbm.Dataset var25=lightgbm.train **output:** matplotlib.pyplot.figure **real next:** var26=var25.predict

Problems:

- The model still gets the “easy” (most frequent) ones and has trouble to output more complicated recommendations.

Final Recommendation Engine Evaluation

In our final recommendation engine[11] we used the context provided by the workflow stage classifier to train six different models (one for each workflow stage of the input cell). And combined them with our classifier, masking and specificalization to create a recommendation system.

Compared to the single model (the fifth try) that converged at 3.71, when we used different models according to context, we were able to reduce the loss for each model.

Models Convergence (single model vs. using context):

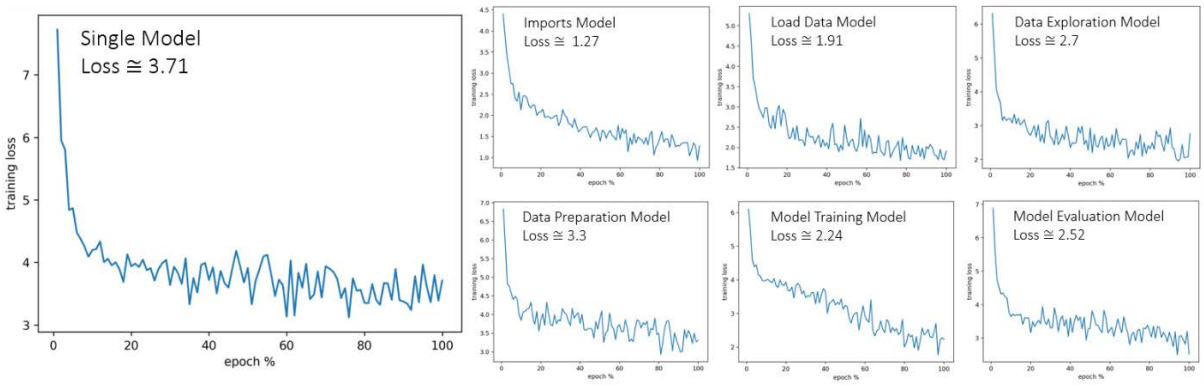


Figure 12. Model convergence context-less vs. context

Recommendations Examples (Demo Run):

```
Input Cell:
***
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import string
import nltk
import os
import re
***
Getting workflow stage for input cell...
Worklfow stage is:Import
Setting model accordingly...
Encoder, Decoder, Voc set to Import.
Getting Masked summed representation...
Masked cell: import_matplotlib.pyplot import_numpy import_pandas import_string impo
rt_nltk import_os import_re
Getting next-line recommendation from model...
Generalized Recommendation: var0=pandas.read_csv
Specificalizing...
Bot:  var0 = pd.read_csv
------------------------
```

[11] See Recommendation Engine section in 3.3 for more details

```
Input Cell:
***
detailed_class_info = pd.read_csv('../input/stage_1_detailed_class_info.csv')
train_labels = pd.read_csv('../input/stage_1_train_labels.csv')
df = pd.merge(left = detailed_class_info, right = train_labels, how = 'left', on =
'patientId')
***
Getting workflow stage for input cell...
Worklfow stage is:Load
Setting model accordingly...
Encoder, Decoder, Voc set to Load.
Getting Masked summed representation...
Masked cell: var0=pandas.read_csv var1=pandas.read_csv var2=pandas.merge
Getting next-line recommendation from model...
Generalized Recommendation: var0.head
Specificalizing...
Bot:  detailed_class_info.head
------------------------
Input Cell:
***
detailed_class_info.head()
***
Getting workflow stage for input cell...
Worklfow stage is:Explore
Setting model accordingly...
Encoder, Decoder, Voc set to Explore.
Getting Masked summed representation...
Masked cell: var0.head
Getting next-line recommendation from model...
Generalized Recommendation: var0.info
Specificalizing...
Bot:  detailed_class_info.info
------------------------
Input Cell:
***
df = df.drop_duplicates()
***
Getting workflow stage for input cell...
Worklfow stage is:Prep
Setting model accordingly...
Encoder, Decoder, Voc set to Prep.
Getting Masked summed representation...
Masked cell: var2=var2.drop_duplicates
Getting next-line recommendation from model...
Generalized Recommendation: shape
Specificalizing...
Bot:  df.shape
------------------------
Input Cell:
***
#Build LSTM Network model
max_features = 50000
model_lstm = Sequential()
model_lstm.add(Embedding(max_features, 300, input_length=max_len))
model_lstm.add(Dropout(0.2))
***
Getting workflow stage for input cell...
Worklfow stage is:Train
Setting model accordingly...
Encoder, Decoder, Voc set to Train.
Getting Masked summed representation...
Masked cell: var4.add_embedding var4.add_dropout
Getting next-line recommendation from model...
Generalized Recommendation: var4.add_dense
Specificalizing...
Bot:  model_lstm.add_dense
------------------------
```

```
Input Cell:
***
model_lstm.add(Dense(1024, activation='softsign'))
model_lstm.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc
uracy'])
model_lstm.fit(X_train,y_train_1h, batch_size=30, epochs=20, verbose=2, callbacks=[
EarlyStopping(monitor='val_loss', patience=1)], validation_data=(X_pred, y_true_1h)
, shuffle=True)
***
Getting workflow stage for input cell...
Worklfow stage is:Train
Setting model accordingly...
Encoder, Decoder, Voc set to Train.
Getting Masked summed representation...
Masked cell: var4.add_dense var4.compile var4.fit
Getting next-line recommendation from model...
Generalized Recommendation: var5=var4.predict
Specificalizing...
Bot:  var5 = model_lstm.predict
------------------------
Input Cell:
***
prediction = model_lstm.predict(detailed_class_info)
***
Getting workflow stage for input cell...
Worklfow stage is:Eval
Setting model accordingly...
Encoder, Decoder, Voc set to Eval.
Getting Masked summed representation...
Masked cell: var5=var4.predict
Getting next-line recommendation from model...
Generalized Recommendation: matplotlib.pyplot.figure
Specificalizing...
Bot:  matplotlib.pyplot.figure
------------------------
Input Cell:
***
acc = prediction.history['acc']
val_acc = prediction.history['val_acc']
loss = prediction.history['loss']
val_loss = prediction.history['val_loss']
plt.figure()
***
Getting workflow stage for input cell...
Worklfow stage is:Eval
Setting model accordingly...
Encoder, Decoder, Voc set to Eval.
Getting Masked summed representation...
Masked cell: history history history history matplotlib.pyplot.figure
Getting next-line recommendation from model...
Generalized Recommendation: matplotlib.pyplot.plot
Specificalizing...
Bot:  matplotlib.pyplot.plot
------------------------
```

```
Input Cell:
***
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.ylabel('Training and validation loss')
plt.xlabel('epochs')
plt.legend()
***
Getting workflow stage for input cell...
Worklfow stage is:Eval
Setting model accordingly...
Encoder, Decoder, Voc set to Eval.
Getting Masked summed representation...
Masked cell: matplotlib.pyplot.plot matplotlib.pyplot.plot matplotlib.pyplot.ylabel
matplotlib.pyplot.xlabel matplotlib.pyplot.legend
Getting next-line recommendation from model...
Generalized Recommendation: matplotlib.pyplot.show
Specificalizing...
Bot:  matplotlib.pyplot.show
------------------------
```

5 CONCLUSIONS AND FUTURE WORK

First, we created a working dataset-builder, creating a repository of 146 Datasets, 19,081 Jupyter notebooks, 296,281 Cells of code.

Such a repository of reproducible data science code-snippets didn't exist prior to our work and could be useful for many research purposes.

Our workflow stage classifier provided decent results[12], Meaning that for a given data science notebook cell we can know its general purpose with high probability. This could also be useful for purposes other than our own. For example, to "understand code" - interpret a notebook automatically, or try getting context by this sort of classification in other fields.

Our recommendation system achieved its goals by generating adequate next-line recommendations based on the current code and some extra context provided by the classifier.
With that said, our recommendation engine is merely a proof-of-concept of what could be achieved in that field of using machine learning sequence-to-sequence models to generate code recommendations.
In addition, we demonstrated that using the context of a given cell we can lower the loss and generate better recommendations, as can be seen in figure 12.

[12] See workflow stage classifier evaluation in 4.1 for more details

Future work

Our model itself could be improved and the research could be expanded in several ways:

- Modify the current summarized (Masked) representation, and add more data to it, to get more useful and meaningful recommendations
- Handle loops in next-line recommendations caused by the line cuts
- Usage of more focused models- instead of considering only the workflow stage of the input cell, consider also the stage of the output cell. Creating a model for each Input-stage → Output-stage pair.
- Such models will also unable to give recommendation options based on the desired workflow stage of the next cell, instead of just one recommendation.
- Expand the system to next cell recommendations (instead of next-line recommendation)
    - Add manual cell structure constraints to prevent non-compiling suggestions.
    - Handle repeating tokens
- Use more features in the model training, such as the cells outputs or execution count
- Collect more data from different sources. Create a measure for grading notebooks (or take existing grades for competitions), take only high graded notebooks into account.
- Identify forks of same notebook and remove duplicates
- Explore different models as encoder and decoder and their influence on the recommendations
- UI integration
- Compare to existing code-completion efforts (for python for example).

REFERENCES AND RELATED WORK

[1] Alexander Ratner, Christopher De Sa, Sen Wu, Daniel Selsam, Christopher Ré; Stanford University, "Data Programming: Creating Large Training Sets, Quickly", 2016 https://arxiv.org/pdf/1605.07723.pdf

[2] "Snorkel: A System for Fast Training Data Creation", https://hazyresearch.github.io/snorkel/

[3] Alexander Ratner, Christopher De Sa, Sen Wu, Daniel Selsam, Christopher Ré; Stanford University, "Snorkel: rapid training data creation with weak supervision", 2017 https://arxiv.org/pdf/1711.10160.pdf

[4] Sepp Hochreiter and Jürgen Schmidhuber, "Long Short Term Memory", http://www.bioinf.jku.at/publications/older/2604.pdf

[5] Rutger Ruizendaal - "Why You Need to Start Using Embedding Layers", https://towardsdatascience.com/deep-learning-4-embedding-layers-f9a02d55ac12

[6] James D. McCaffrey - "Why You Should Use Cross-Entropy Error Instead Of Classification Error Or Mean Squared Error For Neural Network Classifier Training" https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/

[7] Oriol Vinyals, Quoc Le, google, "A Neural Conversational Model", 2015 https://arxiv.org/pdf/1506.05869.pdf

[8] Ilya Sutskever, Oriol Vinyals, Quoc V. Le; google, "Sequence to Sequence Learning with Neural Networks", https://arxiv.org/pdf/1409.3215.pdf

[9] Kyunghyun Cho, Fethi Bougares, Holger Schwenk, Dzmitry Bahdanau, Yoshua Bengio, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation" https://arxiv.org/pdf/1406.1078v3.pdf

[10] Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate" https://arxiv.org/pdf/1409.0473.pdf

[11] Minh-Thang Luong, Hieu Pham, Christopher D. Manning, "Effective Approaches to Attention-based Neural Machine Translation" https://arxiv.org/pdf/1508.04025.pdf

[12] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu; IBM, "BLEU: a Method for Automatic Evaluation of Machine Translation" https://www.aclweb.org/anthology/P02-1040.pdf

[13] Tova Milo, Amit Somech; Tel Aviv University, "REACT: Context-Sensitive Recommendations for Data Analysis", 2016 https://dl.acm.org/citation.cfm?doid=2882903.2899392

[14] Tova Milo, Amit Somech; Tel Aviv University, "Next-Step Suggestions for Modern Interactive Data Analysis Platforms" http://cs.tau.ac.il/~amitsome/pdf/react.pdf

[15] Rogers Jeffrey, Leo John, Navneet Potti, Jignesh M. Patel; University of Wisconsin-Madison "Ava: Fron Data to Insights Through Conversation" http://pages.cs.wisc.edu/~jignesh/publ/Ava.pdf

[16] Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav; Technion
"code2vec: Learning Distributed Representations of Code"
https://urialon.cswp.cs.technion.ac.il/wp-content/uploads/sites/83/2018/12/code2vec-popl19.pdf

[17] Avishkar Bhoopchand, Tim Rocktaschel, Earl Barr & Sebastian Riedel; University College London,
"LEARNING PYTHON CODE SUGGESTION WITH A SPARSE POINTER NETWORK"
https://arxiv.org/pdf/1611.08307.pdf

[18] Matthew Inkawhich – "CHATBOT TUTORIAL"
https://pytorch.org/tutorials/beginner/chatbot_tutorial.html

[19] Yuan-Kuei Wu – "pytorch-chatbot"
https://github.com/ywk991112/pytorch-chatbot

[20] Sean Robertson – "practical-pytorch"
https://github.com/spro/practical-pytorch/tree/master/seq2seq-translation