Data Science Workflow

There are a lot of articles and blog posts out there from experts and teachers explaining what the data science workflow is, but very little quantitative research on what workflow is the best or what people are actually doing. To inspect the workflow from Jupyter notebooks, I first classified code cells with tasks: imports, loading data, function definitions, data loading and manipulation, machine learning, logic and loops, and visualization. Then I inspected patterns of different tasks across the notebook.

Notebooks Considered

Some of the preprocessing required the use of the Python abstract syntax tree. 13.75% of Python 3 notebooks were not able to be parsed with this package, so these notebooks, and those not written with Python 3, were excluded from this analysis. Further, since we are inspecting the data science workflow, I only consider notebooks actually doing data science. I define a notebook as “doing data science” if it uses at least two of the following data science related packages: Pandas, Numpy, Matplotlib, Seaborn, Scikit-Learn, Keras, and Tensorflow. With this definition, 44.9% of Python 3 notebooks are data science notebooks. There are certainly more visualization and machine learning packages than those listed here, but these are top ones used in Jupyter notebooks. For more detail on this, see the results for Analyzing 4 Million Jupyter Notebooks and Data Science with Jupyter. After removing non-data science notebooks and those that couldn’t be parsed by the abstract syntax tree, there are just over 1.42 million notebooks considered.

Pre-Processing and Tokenization

First, I replaced all import aliases with the packages themselves and removed unnecessary words and symbols. I put flags in place to identify imports, function definitions, machine learning frameworks, visualization packages, logic words and symbols (inequalities, conditionals, booleans, binary operators, etc.), and loop-related words (`for`, `while`, `range`, etc.). When I flagging an import, I replaced the entire import statement with a flag to avoid classifying a cell that imports tensorflow, or another machine learning framework, as a machine learning cell. When flagging a function definition, though, I only replace `def` so the code inside of the definition is not lost. Next, I removed lists, tuples, numbers, strings, and all initializations and uses of user defined variables. Finally, I split on spaces and periods to tokenize each cell.

Original Code:

```
import matplotlib.pyplot as plt
import pandas as pd
```

```
df = pd.read_csv('data.csv')
```

```
def show_plot():
    height = [[1,2,3],[2,3,4],[3,4,5]]
    weight = [[2,3,4],[3,4,5],[4,5,6]]
    c = ['red','orange','yellow']

    fig = plt.figure(figsize = (20,10))
    for i in range(3):
        plt.subplot(1,3,i+1)
        x = height[i]
        y = weight[i]
        plt.plot(x, y, color = c[i])
    plt.tight_layout()
    plt.show()
```

Processed Code:

```
<import>
<import>
```

```
pandas read_csv
```

```
<function_def> show_plot
<visualization> pyplot figure figsize
<loop> <loop> range
<visualization> pyplot subplot
<visualization> pyplot plot color
<visualization> pyplot tight_layout
<visualization> pyplot show
```

Figure 1: Code processing

Classification

I looked into three options to classify cells. Once we know what tasks each cell is used for, we can look at how different tasks are distributed across the lengths of notebooks. To calculate the approximate location of each cell, I divide its place by the total number of cells in its notebook and round to the tenths place. This means that cells at location "0" are those within the first five percent of the notebook ($x < 0.05$ rounds to 0), cells at location "0.1" are between the five percent and fifteen percent mark ($0.05 \le x < 0.15$ rounds to 0.1), etc. Then, I calculate the proportion of all cells at each location that are assigned to each task.

K MEANS CLUSTERING

The first approach was clustering cells with K Means. I first performed TF-IDF of the "documents" and kept 100 tokens as features. To assign meaning to each cluster, I looked at the ten most frequent tokens in the processed cells of each cluster and determined what topic they are associated with.

Experimenting with different numbers of clusters, I found that 6 gave the best separation of tasks. This grouping did not result in a cluster for function definitions and has two clusters relating to data loading and manipulation. However, adding more clusters didn't create a function definition cluster, rather duplicated clusters for other tasks. Further, removing clusters made the groups more convoluted and difficult to assign to a single task.

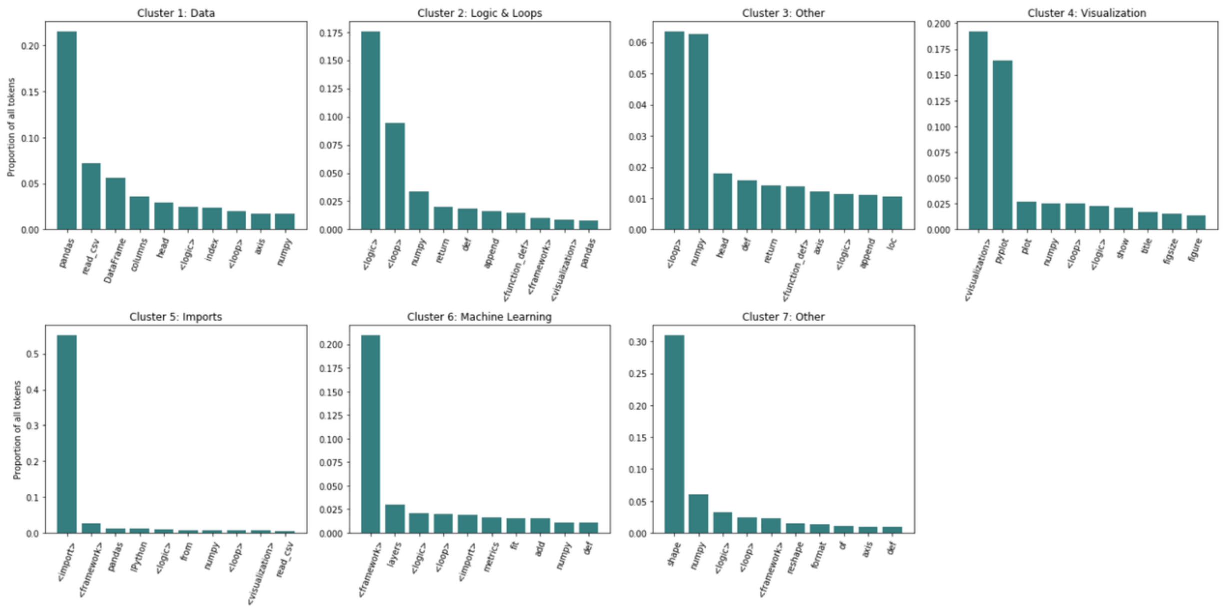


Figure 2: K Means Clustering, top words per cluster

Most of the time, the clusters were easy to distinguish. For example, the top three tokens in cluster 4 are the `<visualization>` tag, `pyplot`, and `plot`. This cluster was assigned the topic "visualization". Two clusters did not relate to a topic, either a mixture of tasks (e.g. cluster 3 is a mixture of logic and data manipulation) or tokens that didn't relate to any task in particular (e.g. cluster 7 doesn't fit any specific topic). These "other" cells did not get assigned a task. Using this strategy, 41% of cells were assigned to a topic.

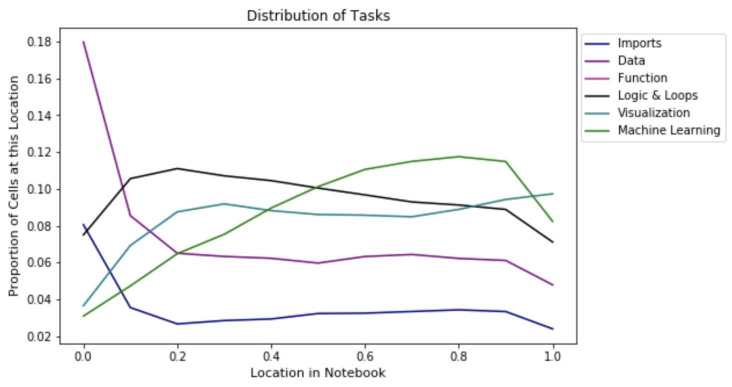


Figure 3: K Means Clustering, distribution of tasks

This plot shows how the frequencies of these tasks are distributed across the notebook. The main standouts are that imports in navy and data loading and manipulation in purple are most frequent at the start of the notebook. After we pass the first 5% of cells (location 0), logic and loops in black takes over. The use of machine learning, in green, increases steadily over the course of the notebook until the very end. And finally, visualizations in teal are unlikely at the start but pretty constant after the first 15% of cells (after location 0.1).

One downside of K means is that it puts cells into discrete boxes, meaning one cell can only be assigned one task. If you look back at the third example cell in Figure 1, it's clear that it defines a function, uses a loop, and creates visualizations. Ideally, this type of cell should be assigned all three tasks.

LATENT DIRICHLET ALLOCATION

The second approach was topic modeling with Latent Dirichlet Allocation. This algorithm identifies topics based on the preconception that each topic follows a distribution over a set of tokens, and each cell follows a distribution over a set of topics. A benefit to this approach over K Means is that it allows cells to be assigned to multiple tasks. When one cell is passed in to the fitted model, what's returned is a probability distribution of topics expressed in that cell.

Experimenting with different numbers of topics and numbers of tokens per topic, I found that 8 topics with 10 tokens each gave the best separation of tasks. This was the minimum number of topics that did not group logic and loops with data loading and manipulation. Adding more tokens per topic didn't change the ways their distributions looked very much, just added more low-frequency, not very meaningful tokens.

To assign meanings, I looked at the tokens distribution for each topic.

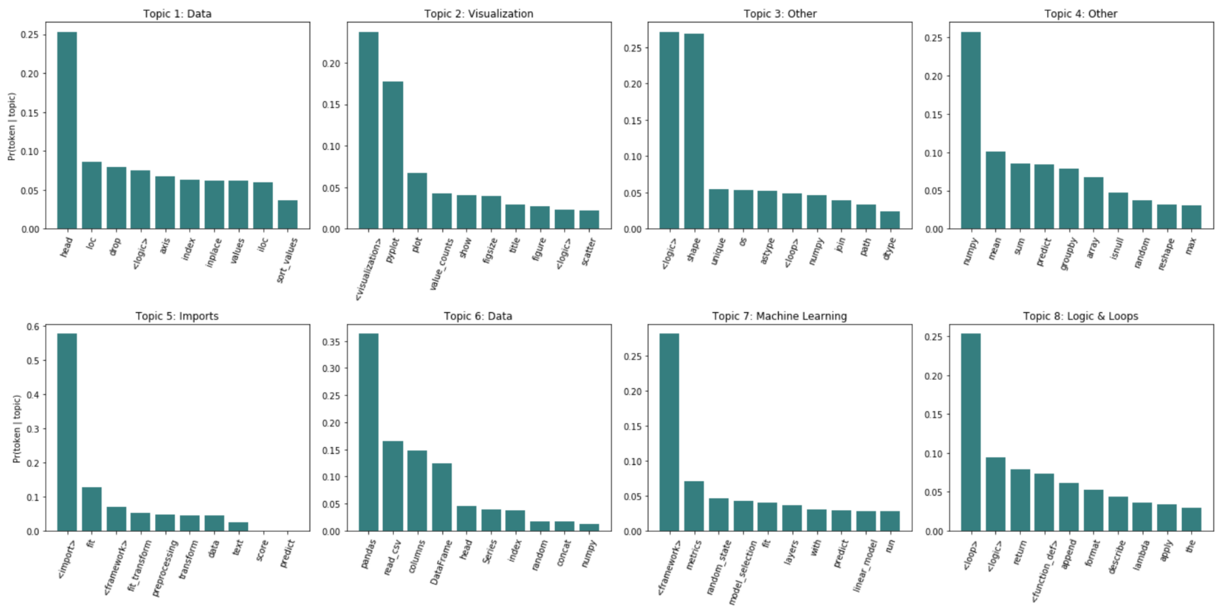


Figure 4: Latent Dirichlet Allocation, token distribution per topic

Just like with K means, some topics make a lot of sense. For example topic 6 has `pandas`, `read_csv`, `columns`, and `dataframe` as its most likely tokens, associating it with data loading and manipulation. Topics 3 and 4 don't make as much sense, combining tokens related to multiple topics or no topics in particular. With this strategy, 77% of cells were assigned at least one task.

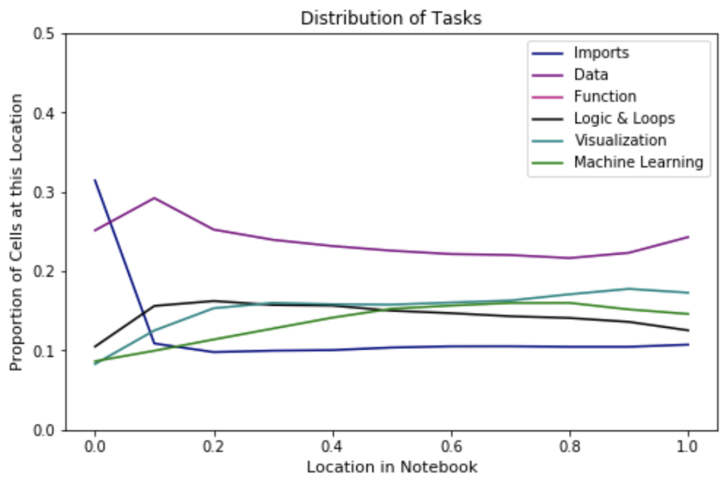


Figure 5: Latent Dirichlet Allocation, distribution of tasks

The main features we saw in the distribution for K Means stayed the same: imports in navy and data loading and manipulation in purple are most frequent at the start of the notebook. Logic and loops become more popular after we pass the first 5% of cells. The use of machine learning, in green, increases steadily over the course of the notebook until the very end. And finally, visualizations are unlikely at the start but fairly constant at after the first 15% of cells.

DETERMINISTIC APPROACH

The final approach was deterministic. Since we have a good idea of what commands and flags will show up in cells of specific tasks, manually assigning classes should be a pretty accurate representation of what's going on in the notebook. With K Means and LDA, we knew exactly what we were looking for in the topic/cluster separation, so it's not entirely necessary to use a probabilistic model. Below is the criteria for a cell to be assigned each task. Like with LDA, a cell can be assigned multiple tasks.

- Imports: contains `<import>`
- Visualization: contains `<visualization>`
- Machine Learning: contains `<framework>` or at least one machine learning API call (at least one of `train_test_split`, `fit`, `fit_generator`, `transform`, `fit_transform`, `predict`, `predict_class`, `predict_generator`, `predict_on_batch`, `evaluate`, or `evalutate_generator`)
- Data Loading / Manipulation: contains `pandas` or `numpy`
- Function Definitions: contains `<function_def>`
- Logic & Loops: contains `<logic>` or `<loop>`

With this approach, 55% of cells fell into at least one category.

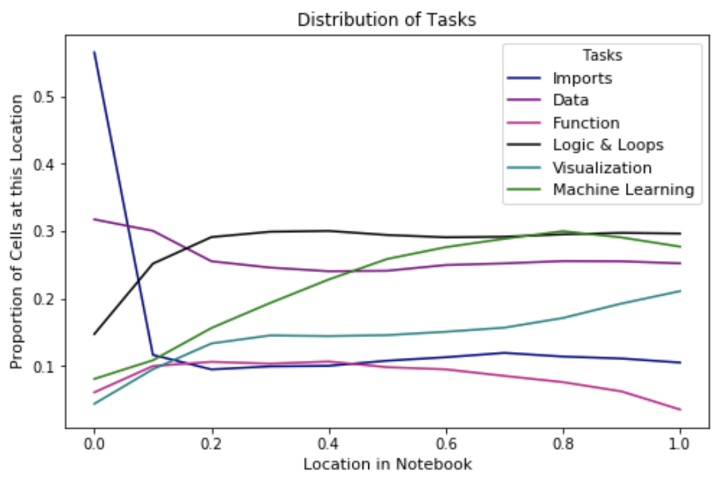


Figure 6: Deterministic Approach, distribution of tasks

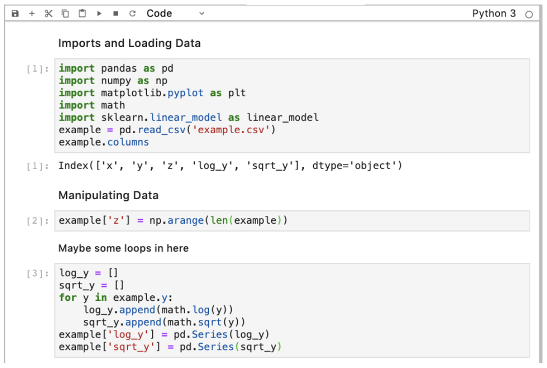
Again, the same patterns pop up. Imports are a large proportion of early cells, and still drops quickly after the first five percent of cells. Data loading and manipulation is also pretty consistent, used in around 28% of cells, but most likely in the first 20% of cells. The use of machine learning increases steadily in the first 80% of cells, then drops slightly in the last 20% of cells. The use of visualizations increases at the beginning and the end, but remains somewhat constant around 15% in the middle 60% of cells. Function definitions are most likely between the 20% and 40% marks and become less likely towards the end. Finally, logic and loops are unlikely at the beginning but are used pretty consistently for the rest of the notebook.

Because cells can be assigned more than one task, we can also use these labels to look at what tasks tend to be in a cell together. There are moderate correlations between tasks are Logic & Loops and Function definitions (r = 0.32) and between Logic & Loops and Data Loading & Manipulation (r = 0.23).

Discussion

Because this is unsupervised, it's hard to say which of the three options is best. Even in the deterministic model, there are data manipulation cells that don't use pandas or numpy that aren't accounted for. While I don't think the exact proportions should be taken as truth, the overall patterns were consistent between the models and can give us a good idea of how the data science workflow progresses through a Jupyter Notebook. First, we have imports and data manipulation. After the first 15% or so of cells have passed by, the use of machine learning and visualization grows. At the very end, machine learning takes a dip and visualizations take one last leap. Logic and loops are used throughout the notebook, but least frequent at the very beginning.

At the most basic level, this is what a Data Science Jupyter Notebook looks like based on these results:



Imports and Loading Data

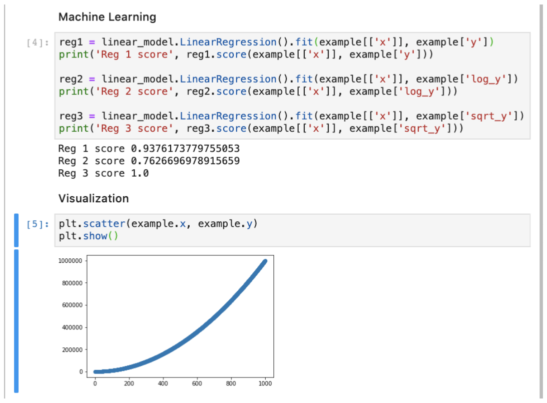
```
[1]: import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
     import math
     import sklearn.linear_model as linear_model
     example = pd.read_csv('example.csv')
     example.columns

[1]: Index(['x', 'y', 'z', 'log_y', 'sqrt_y'], dtype='object')
```

Manipulating Data

```
[2]: example['z'] = np.arange(len(example))
```

Maybe some loops in here

```
[3]: log_y = []
     sqrt_y = []
     for y in example.y:
         log_y.append(math.log(y))
         sqrt_y.append(math.sqrt(y))
     example['log_y'] = pd.Series(log_y)
     example['sqrt_y'] = pd.Series(sqrt_y)
```



Machine Learning

```
[4]: reg1 = linear_model.LinearRegression().fit(example[['x']], example['y'])
     print('Reg 1 score', reg1.score(example[['x']], example['y']))

     reg2 = linear_model.LinearRegression().fit(example[['x']], example['log_y'])
     print('Reg 2 score', reg2.score(example[['x']], example['log_y']))

     reg3 = linear_model.LinearRegression().fit(example[['x']], example['sqrt_y'])
     print('Reg 3 score', reg3.score(example[['x']], example['sqrt_y']))

     Reg 1 score 0.9376173779755053
     Reg 2 score 0.7626696978915659
     Reg 3 score 1.0
```

Visualization

```
[5]: plt.scatter(example.x, example.y)
     plt.show()
```