Gathering Data

Our Story

Your first step at AAVAIL, just like everywhere else, it to look at the data sources. You soon discover that AAVAIL has data everywhere! There is no shortage of data. It looks like they have managed to store every type of transaction with their subscribers.

You will need a smart way of managing all of this data. Let's take a look now at some best practices for managing data in a large enterprise.

Documenting your data

Too often data scientists will find themselves deep in the process of developing a solution, based on the data that was provided to them, before they realize that the data itself is flawed, inaccurate or in some other way non-ideal. Developing the habit of creating a simple document with at least a description of the ideal data needed to test the hypotheses around the business problem may seem like an unnecessary step, but it has potential to both:

1. Streamline the modeling process
2. Help ensure that all future data come in an improved form

ETL

The process of gathering data is often referred to as Extract, Transform, Load (ETL). Data is generally gathered (or extracted) from heterogeneous sources, cleaned (transformed) and loaded into a single place that facilitates analysis. Before the advent of the modern data scientist's toolkit data was often staged in a database,data lake or a data warehouse. Still today data is frequently staged to facilitate collaboration, but there are tools now that enable more possibilities today than ever before. Jupyter Lab has an extension called data grid that allows it to read delimited files with millions or even billions of rows. Then tools like Dask help you scale your analyses. To ensure that projects are completed in a reasonable amount of time the initial pass at ETL should use a simple format like CSV, then a more complex system can be built out once you have accomplished the Minimum Viable Product (MVP).

Common methods of gathering data

Plain text files

Plain text file can come in many forms and generally the open function is used to bring the data into a Python environment. This is a flexible format, but because no structure is imposed, custom scripts are generally needed to parse these files and these scripts do not always generalize to new files.

The large majority of data science projects involve a modeling step that requires input data in a tabular numeric format. In order to extract data from a plain text file you may need to identify patterns in the text and use regular expressions (regex) to pull out the relevant information. Python's built-in regex library is known as re.

On the other hand if the data you are working with consists of natural language, then there are a number of libraries that can work directly with the text files. The two main libraries are:

- spaCy
- NLTK

Also, scikit-learn has become a standard tool in the overall workflow when processing these files.

- scikit-learn's text tutorial

These tools can be applied to unstructured text to generate things like word counts, and word frequencies. We saw an example of this in the Data science workflow combined with design thinking example.

Delimited files

One of the most commonly encountered ways of storing structured data is in delimited files, where rows of tabular data are stored in lines of a text file and the columns within each row are separated by a special "delimiter" character such as a comma or a tab character.

This simple structure helps account for the popularity of these formats, with probably the most widely used being Comma-Separated Values (CSV). CSV files are both human and machine readable, and have minimal overhead in terms of the proportion of the file devoted to defining the structure of the data when compared to most other file formats. As such Pandas comes with methods for both reading and writingCSV files. (Note that these functions can also handle other delimiters like tab or the pipe character "|", but commas are the default.)

Spreadsheet programs like Microsoft Excel that are used for analyzing tabular data also read from and write to files in CSV format. The native Excel file format (often with file extensions .xls or .xlsx) can also be considered a delimited file type. Though these files also contain a significant amount of extra information related to things like styling that are separate from the actual data. Nonetheless, since these files are commonly used to save datasets, Pandas also has a method for reading them: pandas.read_excel.

HINT: A best practice when loading data from plain text or delimited files is to separate the code for parsing into its own script. Because the files are read line by line in a separate Python call, it is more memory efficient and this separation of tasks helps with automation and maintenance.

It is a common mistake to try to read large files into pandas then use the date frame environment to parse. If your parsing (cleaning) task is simple then use a parser. Here is a simple example:

```
#!/usr/bin/evn/python

"""
simple example of a parser
"""

import os
import csv

## specify the files
file_in = os.path.join("..","data","snowfall.csv")
file_out = os.path.join("..","data","snowfall_parsed.csv")

## create an outfile handle (needs to be closed)
fidout = open(file_out,"w")

## use the csv module to read/write
writer = csv.writer(fidout)

## generic parsing function
def parse_line(line):

    if line[3] not in ["HI","NC","OR"]:
        return None
    else:
        return line + ['new_data']

## for each line in the file read in the first file that you need to reference
with open(file_in) as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    header_in = reader.__next__()
    writer.writerow(header_in + ["new_column"])
    for line in reader:
        line = parse_line(line)
        if line:
            writer.writerow(line)

fidout.close()
print("done parsing")
```

The highlighted lines show where this parser changes the original data by filtering and adding an additional column.

JSON files

While delimited files are well suited for housing data in flat tables, datasets with more complex structures require different formats. The JavaScript Object Notation (JSON) file format can accommodate quite complex data hierarchies. Python's built-in library handles reading/writing JSON files.

```
import json
data = json.load(open('some_file.json'))
```

In addition, pandas.read_json is also available for loading JSON files.

At its base a JSON object can be thought of as analogous to a Python dictionary or list of dictionaries. For example a table of data from a JSON file could be read into Python as a list of dictionaries where each dictionary represented a row of the table, and the keys of each dictionary were the column names. This formatting is somewhat inefficient for simple tabular data, with column information explicitly repeated with each row, but is useful when representing more intricate relationships in the data. JSON objects often have a highly nested structure that you can think of as dictionaries within dictionaries within dictionaries.

For example, modern websites track a great deal of information about users' interactions with the site and the varied nature of these interactions make a table structure too rigid for recording them. In practice, most sites send JSON objects back and forth between the user's computer and the website's server. Many data scientists' primary source of data are ultimately these JSON objects.

Relational databases

Relational databases, i.e. those that impose a schema on datasets are a major source of data for data science projects. Database tables can naturally be converted into Python objects like Pandas DataFrames. Reading data into a Python environment requires opening a connection to a database and there are various libraries for managing this connection, depending on the type of database to be accessed. Some Relational DataBase Management System (RDBMS) and their corresponding interface utilities for Python:

RDBMS	Python Connector
MySQL	MySQL Connector
PostgreSQL	Psycopg
SQLite	sqlite3
Microsoft SQL	pyodbc

Each of these tools are designed with maintaining the integrity of the database in mind, including methods for rolling back updates, and ways to safeguard against SQL Injection vulnerabilities. As such, the process of querying the database and ingesting the results can seem fairly involved. For example, here is a basic flow for getting the contents of a table from a PostgreSQL database using psycopg2.

```python
import psycopg2 as pg2
conn = pg2.connect(database='my_db', user='my_username')

# create a cursor to traverse the database
cur = conn.cursor()

# cursor object executes a query, but does not automatically return results
cur.execute("SELECT * FROM my_table")

# Return all query results
results = cur.fetchall()
# WARNING: If the result set is large, it may overwhelm the memory
# resources on your machine.

cur.close()
conn.close()
```

While the steps required to connect, query, and disconnect from a relational database are more involved than when loading in data from a file on your local machine, the table structure from the database schema basically guarantees that the data will fit cleanly into a Pandas DataFrame or NumPy Array.

NoSQL databases

“NoSQL” is a catch-all term referring to “non SQL” or “non relational”, or more recently “not only SQL”. Usually meaning that the method for housing data does not impose a schema on it (or at least not as tightly constrained as in relational databases). This tradeoff gives greater flexibility in what and how data are stored at the cost of increased traversal times when searching the database. This tradeoff is similar to the one we encountered when working with delimited files like CSVs and with JSON files. When loading or dumping data between a file and a database, CSVs are a good match for tables in a relational database, whereas JSONs are more aligned with NoSQL databases.

The are many examples of NoSQL databases, each with different use cases, and most of which can be accessed with Python.

One flexible and popular example is MongoDB. MongoDB is a document-oriented database, where a “document” encapsulates and encodes data in a standard format. In the case of MongoDB, that format is JSON-like. Like the relational databases mentioned above, MongoDB has a client for querying it directly, as well as a connector for querying from within Python. These queries are constructed usingMongoDB’s query syntax.

The Python connector to MongoDB is PyMongo.

```python
from pymongo import MongoClient
# By default a Mongo db running locally is accessible via port 27017
client = MongoClient('localhost', 27017)
db = client['database_name']

# Within a db, documents are grouped into "collections" -- roughly equivalent
# to tables in a relational db.
coll = db['collection_name']

# Return all the documents within the collection
docs = coll.find()
```

NoSQL databases handle messy data better than relational databases. As such, they can be useful as an intermediate step in a data pipeline as the first place to house data after it is collected, but before it can be cleaned and validated. After cleaning, it may be possible to convert the data to a format suitable for storage in a relational database.

Web scraping and APIs

Automating the process of downloading content from websites is known as web scraping.

** IMPORTANT **
Web scraping can be done in legitimate ways, but just as easily web scraping tools do not stop you from violating a websites terms of service. If a website encourages the sharing of their data then they will create a specific API endpoint that you will use. More often than not the API will require to have an identifying key.
Generally websites that encourage data sharing like the National Center for Biotechnology offer usage guidelines
If the usage guidelines are not clear either through specific documentation or within the terms of service, do not attempt to scrape a website without explicit approval from the organization.

Various tools in Python are available for accessing and parsing webpage data.Requests is a user-friendly library for downloading web pages. It can also be used to retrieve files that are exposed through a URL. For a webpage the data returned from a call using Requests is the HyperText Markup Language (HTML) code that instructs a client such as a web browser how to display a page. This HTML code will often (but not always) contain the data you want to collect from the particular webpage.

Modern webpages tend to have a great deal of information in their HTML beyond what is shown to the user, so parsing through it all to collect the relevant data can be a daunting task. Fortunately, if a page is readable in your browser, then its HTML must have a coherent structure. Beautiful Soup is a Python library that provides tools for

walking through that structure in a systematic and repeatable way. Thus, in the context of web scraping Beautiful Soup can be used to extract the relevant data from the *soup* of all the other information contained in an HTML file.

Many websites' contents are dynamically rendered in such a way that the information displayed on a page never makes it directly into the page's HTML. In such cases it may not be possible to download the data of interest with a tool like Requests. One option in this scenario is to move to a tool for browser automation, such as Selenium. Selenium's Python interface is described here.

Another tool, specifically designed for web scraping in Python, is Scrapy.

Depending on your website of interest you may have to try several of these tools to successfully collect the relevant data in a scalable way. But a general rule of thumb is that if you can see what you want to collect in your browser, the the website sent it to you, so it should be retrievable.

Streaming data

In the modern landscape of business data streams are becoming more common. A data stream is a sequence of digitally encoded signals. Data can be streamed for many purposes including storage and further processing (like modeling). Data streams become important when the data of a project or company becomes mature and the AI pipeline is connected to it. As we move into the portions of the AI enterprise workflow that focus on models in production we will be using Apache Spark's streaming to connect deployed models with streaming data. Data collected from sensors or devices connected via the internet of things are oftent setup to produce streaming data. We will work specifically with these types of data in module 5.

Apache Hadoop File Share (HDFS)

Apache Hadoop File Share (HDFS) is the core of Apache Hadoop , an open source system that is designed to use arrays of commodity hardware to store and manage very large datasets.

HDFS is the storage component of the system. Large datasets are divided into blocks, and those blocks are distributed and stored across the nodes in an HDFS cluster. Any code that is created to analyze the datasets stored in a Hadoop cluster is executed locally for each block of data, and in parallel. This parallel analysis of data blocks means that Hadoop can process very large data sets rapidly.

The Hadoop framework itself is written mostly in Java. However, any language, including Python, may be used to analyze the data stored in a Hadoop cluster. The Apache Foundation provides a number of other packages that may be installed alongside Hadoop to add additional relational database functionality and improve scalability.

IMPORTANT: Apache Hadoop is a *de facto* standard in many large enterprises today. It is often used with Apache Spark and a NoSQL database engine to provide data storage and management of data pipelines used by machine learning models.

Other sources of data formats

There are several other formats worth mentioning.

Format	Description
HDF5	There is a is a hierarchical format HDF5 used to store complex scientific data. The format is useful for storing and sharing large amounts of data.
NumPy's *.npy and *.nzp formats	NumPy has its own binary format (NPY) and the NPZ format is an extension of it that allows multiple arrays and compression.