Data Processing Infrastructure (DPI)

Fatema Bhayani
6-1-2020

1 Table of Contents

2 Introduction

Given that the scope of the project is social network algorithms, we want a datastore that contains social network data we can input into various algorithms. This is accomplished through Twitter data preprocessing functions that take in raw Twitter data and output information such as Tweets in between a certain time period. This data is expected to be stored inside a database and accessible to future algorithms that need this data for input.

Moreover, we want a system where we can add new classes or algorithms and processes without the risk of breaking what has been implemented. Moreover, we want to optimize our code design so that users of any process are offered as much simplicity as possible when it comes to determining what database they want to use.

3 Project Goal

One of the primary goals of this project is to create professional code that is scalable and maintainable. We want a system where we can add new classes or algorithms and processes without the risk of breaking what has been implemented. Moreover, we want to optimize our code design so that users of any Process (e.g. stemming or Twitter download) are offered as much simplicity as possible when it comes to determining what database they want to use.

4 Solution Approach/Philosophy

In order to make it easier for future developers to add to our code, we have a Process abstract class that future classes of algorithms can extend. These new Process subclasses automatically gain useful features such as the ability to access various datastores, without us having to write additional code.

4.1 Process and Data Store Object (DAO) Code

At the core of this design are three abstract classes,

1) InputDAO: An abstract interface that is able to read from some datastore.
2) OutputDAO: An abstract interface that is able to write to some datastore.
3) Process: Contains inputs and outputs fields, which are respectively lists of InputDAO and OutputDAO objects. There are process method(s) that process data obtained from the InputDAOs and output the data via the OutputDAOs.

Addressing the issue of breaking what has been implemented, the design decouples processes and the data upon which they operate on, in accordance with the Dependency Inversion Principle, which emphasizes that one should program with abstraction in mind. Programming with abstraction in mind means that our code does not care about the implementation details of a given object. Rather, we only care that the object gives us the operations that we need. As a result, whenever operation details need to be changed for that object, we do not have to change and risk breaking client code.

To accomplish this, the idea of abstract interfaces is used, as well as the Data Access Object Design Pattern, which abstracts away datastore operations such as “read” and “create” with their concrete implementations (as a result, we have InputDAOs and OutputDAOs). Consequently, Process objects no longer needs to worry about the implementation details of operations such as inputting or outputting data to a datastore. Thus, Process can use different types of datastore as input or output without ever having to change its code, resulting in code that is more scalable since we no longer risk breaking the Process code whenever new datastores are introduced.

4.2 Datastore Configuration File (DSConfig)

To ensure that users of any Process (e.g. stemming or Twitter download) are offered as much simplicity as possible when it comes to determining what database they want to use, we create a datastore configuration file. With a Datastore Configuration File, the user needs to give the details for input and output datastores, and the InputDAOs and OutputDAOs are automatically generated and input into the Process object. This automatic generation is done via a Factory object, in accordance with the Factory Design Pattern, which abstracts away the construction of new objects. The Factory Design Pattern helps us create a standardized architectural model for a

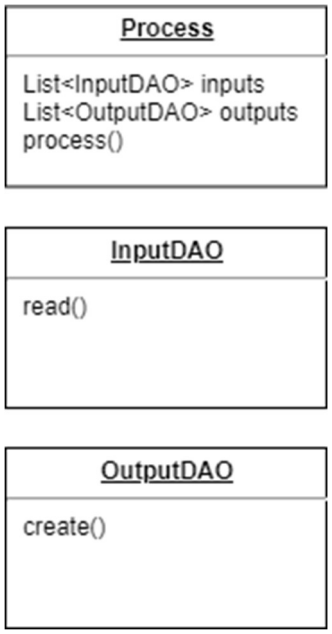
range of applications, algorithms, and processes, but allow individual applications to define their own domain object and instantiate them. By using the Factory Design Pattern to create the Datastore Configuration File, we are able to specify the details of the database and delegate the implementation details of loading and storing. As a result, the user does not have to create Data Store Objects manually (i.e. without the use of a factory), which removes unnecessary overhead to using Process and decouples Process and the Input and Output DAOs.

5 Building Blocks

Our Data Processing Infrastructure has two key parts: The Process abstract class and the Datastore Configuration File. The Process abstract class allows future classes of processes and algorithms to extend it. These new Process subclasses automatically gain useful features such as the ability to access various datastores, without us having to write additional code. The Datastore Configuration File allows the user to give the details for input and output datastores, and the InputDAOs and OutputDAOs are automatically generated and input into the Process object.

5.1 Process and Data Store Object (DAO) Implementation

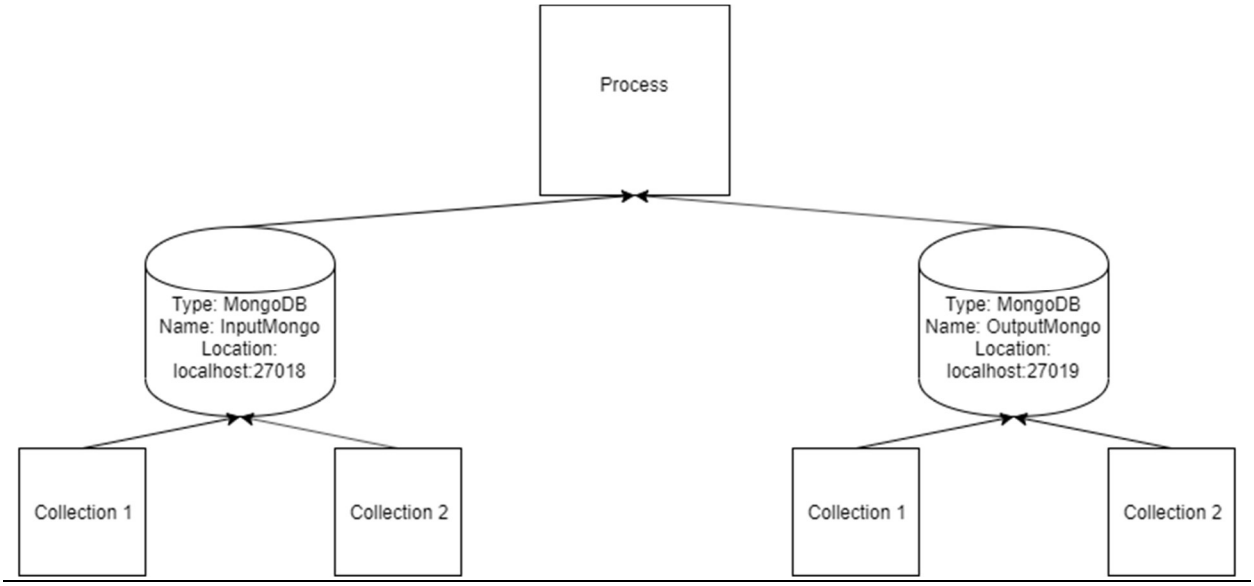
Process, InputDAO, and OutputDAO, are abstract classes with the following form,



Process, InputDAO, and OutputDAO are abstract classes. Concrete implementations simply need to extend these classes. For example, if I want to implement a Datastore Object for storing the output of an algorithm to a Mongo database, I can extend OutputDAO and take advantage of its functionality without writing duplicate code.

5.2 Datastore Configuration File (DSConfig)

We can imagine a given Process as taking various collections of data from various datastores as input, then outputting the results of the process into various collections in various datastores.



Taking inspiration from this idea, we just need to specify the identity and corresponding information for each collection of input and output data in the datastore configuration file.

The format of the config file, is as follows,

```
input-datastores:
  input-ds1:
    project-name: "Process"
    datastore-name: "InputMongo"
    collection-name: "Collection 1"
    type: "MongoDB"
    location: "mongodb://localhost:27018"
  ...


output-datastores:
  output-ds1:
    project-name: "Process"
```

```
  datastore-name: "OutputMongo"

  collection-name: "Collection 1"

  type: "MongoDB"

  location: "mongodb://localhost:27019"

...
```

The path for a datastore configuration file can then be passed into a Process, whereupon the specified Data Store Objects are generated in order to interact with the corresponding datastores.

6 Twitter API Interface

The Twitter API allows developers to follow, search and get users and tweets. More specifically, the Twitter API makes it easier to get a list of followers and friends of a user. The friends of a user are the Twitter users that a specific user follows. On the other hand, the followers of a user refers to the Twitter users that follow a specific user. Twitter API can also be used to get tweets of a specific user.

The Twitter API exposes dozens of HTTP endpoints that can be used to retrieve, create and delete tweets, retweets and likes. It provides direct access to rich and real-time tweet data, but requires having to deal with a lot of low level details. Tweepy is an open source python library package that allows you to bypass a lot of those low level details. We use Tweepy to access the Twitter API since it allows us to interact with the Twitter API easily.

6.1 Connect with the Twitter API

To connect with the Twitter API and get the authentication code, a developer account is required. Once you have the authentication code, you can put the details in Tweepy. For more details, refer to the Tutorials below.

Twitter Downloader has been split up into two classes, one for a specific domain of Twitter download. We can imagine adding new classes of Twitter download as the need arises.

```
"""

Download Tweets for use in future algorithms.

"""

class TwitterTweetDownloader(Process):
```

```
"""

Download Twitter Friends for use in future algorithms.

"""

class TwitterFriendDownloader(Process):
```

Each Twitter download class implements the Process abstract class, so that they may enjoy the benefits offered by using abstract Input and Output DAOs, as well as initializing the DAOs via the Datastore initial configuration file.

7 MongoDB Interface

MongoDB is a database which stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time.

For example, if we are downloading tweets from Twitter, the following key-value structure will be used:

Project Name: “TwitterDownload”

Datastore Name: “TwitterDownload”

Collection Name: “Tweets”

As a result, the tweets will be stored in a collection “Tweets” inside a datastore “TwitterDownload” in a project “TwitterDownload”.

Inside this collection, the key of this collection is the string of the user id while the values are a dictionary. The keys to this dictionary are the strings which represent the different details about the tweet like “created at” which gives the date on which the Tweet was posted, “id_str” which gives the string of the id of the tweet, “text” which gives the text contents of the tweet, “truncated” which returns true if the tweet text was truncated and false otherwise, “entities” which is a dictionary containing hashtags, symbols, and user mentions.

For example, "tweets" : [{ **"created_at"** : "Tue May 05 19:04:48 +0000 2020", **"id"** : NumberLong("1257748095808528385"), **"id_str"** : "1257748095808528385", **"text"** : "RT @igilitschenski: I'm thrilled to announce…." All of thi…", **"truncated"** : false, **"entities"** : { *"hashtags"* : [ ], *"symbols"* : [ ], *"user_mentions"* : [{ "screen_name" : "igilitschenski", "name" : "Igor Gilitschenski", "id" : 19208872, "id_str" : "19208872", "indices" : [3, 18] }]

For MongoDB paths, we access a given database by specifying an IP and port, as opposed to giving an actual physical path.

7.1 MongoDB Support

In order for our system to support MongoDB, we need concrete implementations of the Input and Output DAOs that wrap around code for Python’s MongoDB API (Pymongo).

```
"""

Concrete implementation of InputDAO that supports MongoDB.

"""

class MongoInputDAO(InputDAO, MongoDAO):OutputDAO
```

```
"""

Concrete implementation of OutputDAO that supports MongoDB.

"""

class MongoOutputDAO(OutputDAO, MongoDAO):
```

Both implement the respective methods of read and create using Pymongo.

7.2Mongo Daemon Generator

In addition to implementing Input and Output DAO for MongoDB, a tool was developed to allow for the easy generation of new MongoDB daemons on a machine for different projects.

There are two aspects to this tool.

1) Mongo Project Configuration YAML
2) Mongo Daemon Generator Python script

Users simply provide the names of the project and location of where they want the project files to be stored in the configuration file. The Python script is subsequently run on the configuration file to generate new MongoDB daemons for the projects. The configuration is then updated so that users know which ports to access the MongoDB services.

8 Tutorials and Examples

8.1 Getting Twitter API Authentication

Apply for a developer account on https://developer.twitter.com/en/apply-for-access. It will take a couple of days for your application to be approved. Once your application is approved, you will get a consumer key, consumer secret, access token, and access token secret.

Run the code below to verify your authentication.

```
import tweepy

# Authenticate to Twitter
auth = tweepy.OAuthHandler("CONSUMER_KEY", "CONSUMER_SECRET")
auth.set_access_token("ACCESS_TOKEN","ACCESS_TOKEN_SECRET")api =
tweepy.API(auth)# test authentication
try:
    api.verify_credentials()
    print("Authentication OK")
except:
    print("Error during authentication")
```

The code is currently hosted on GitHub as an organization public repository. The following is a link to the repo: https://github.com/Social-Network-Algorithms/

Clone the repo onto your local device. In the main directory, make a new file “credentials.py”. In it, add your credentials for accessing the Tweepy API in the following format:

```
ACCESS_TOKEN = <stuff here>
ACCESS_TOKEN_SECRET = <stuff here>
CONSUMER_KEY = <stuff here>
CONSUMER_SECRET = <stuff here>
```

8.2 Download Data from Twitter

To download data from Twitter, we first need a Datastore Configuration File (DSConfig). To create a datastore configuration file for download Tweets, the file would look something like this,

```
input-datastores:

  input-ds1:
```

```
    project-name: "TwitterDownload"
    datastore-name: ""
    collection-name: ""
    type: "Tweepy"
    location: ""



output-datastores:
  output-ds1:
    project-name: "TwitterDownload"
    datastore-name: "TwitterDownload"
    collection-name: "Tweets"
    type: "MongoDB"
    location: "mongodb://localhost:27018"
```

We are downloading data from Twitter, so our **input** datastore is Tweepy. We are using MongoDB to store our downloaded data, which in this case are Tweets. Thus, we want to **output** the downloaded data to the Tweets collection inside the TwitterDownload database, whose location can be accessed at the specified path. Note that, for MongoDB paths, we access a given database by specifying an IP and port, as opposed to giving an actual physical path.

We have at this point a datastore initial configuration file for downloading Tweets. The next step is to instantiate a TwitterTweetDownloader object.

```
twitterDownloader = TwitterTweetDownloader("ds-init-config.yaml")
```

And just like that, we are able to download Tweets.

```
twitterDownloader.get_tweets(user_name, start_date, end_date, num_tweets)
```

The downloaded data is stored in the output collection and database, as specified by “ds-init-config.yaml.”

If we want to download a list of a user's friends, then we could first modify the output datastore section for our datastore config file.

```
output-datastores:

  output-ds1:

    project-name: "TwitterDownload"

    datastore-name: "TwitterDownload"

    collection-name: "Friends"

    type: "MongoDB"

    location: "mongodb://localhost:27018"
```

Notice here that we changed the collection name to Friends, since we want different types of downloaded data to be stored in different collections.

Then, just like before, we instantiate a Downloader object and call the respective method to download.

```
twitterDownloader = TwitterFriendsDownloader("ds-init-config.yaml")

twitterDownloader.get_friends_by_screen_name(user_name, num_friends_to_get)
```

8.2.1 Download Random Tweets

In order to automate the daily download of raw tweets, we have created a script called raw_tweet_scheduler_daemon.

To use this script, simply call it like so,

```
python3 raw_tweet_scheduler_daemon.py --num-tweets-to-download <num_tweets> --
time-to-download <time> --options <option-value>
```

Notice that we provide two arguments,

1) number of tweets to download

2) time each day we want to download our tweets

"options" currently serves as a placeholder for additional configuration options that this program will provide that are not yet implemented (such as how a user would like to receive monitoring information). Furthermore, if num-tweets-to-download is not specified, the program will assume

that the user wants to download at the full rate limit. If time-to-download is not specified, the program will download at a default set time.

There are three aspects of this script.

1) Schedule daily.
2) Download raw tweets.
3) Run as a daemon in the background.

To address the issue of scheduling, we use a Python library called schedule. This allows us to run the raw Tweet download function at a specified time daily.

To address the issue of running in the background, we use another Python library to run the script as a daemon. This library handles many issues associated with running a daemon, such as not printing to stdout.

Addressing the issue of downloading raw tweets, we are using the random Tweet download method,

```
get_random_tweets(self, num_tweets: int, output_collection_name: str)
```

The way this works is that we have the following function,

```
def download_daily_tweets():
```

Our scheduler calls this function daily. In download_daily_tweets, we generate a collection name, which is based on the date. We pass in the given collection date to get_random_tweets.

In turn, get_random_tweets creates a new collection with the specified name and where documents (elements) in this collection are Tweet objects (contain information like text, tweeter, location).

8.3MongoDB Support

Before we talk about MongoOutputDAO and MongoInputDAO, we need to first understand how type conversion works between Python and MongoDB.

Elements inside a MongoDB collection are called documents. These roughly correspond to Python dictionaries, and Pymongo(the library we're using to interact with MongoDB) is responsible for type conversion.

For example, say that we want to insert the following item into a collection.

```
item = {
```

```
            "user_name": user_name, # type: string

            "id": id, # type: int

            "start_date": start_date, # type: date

            "end_date": end_date, # type: date

            "tweets": tweets # type: list of dictionaries

        }
```

To insert into a collection, we run,

```
collection.insert_one(item)
```

Where collection refers to a Pymongo Collection object.

Pymongo is responsible for converting our dictionary into a BSON, which is the MongoDB data type and very similar to JSON except that it supports additional data types such as dates.

Notice how our keys must be strings. These are, in turn, converted into the fields of the BSON. The values are then converted to their MongoDB counterpart. For example, Python ints are converted into MongoDB ints, dictionaries are converted into BSON objects, Python dates are converted into MongoDB dates, and so on. As long as our Python values have a corresponding MongoDB type, Pymongo is able to do the conversion.

8.3.1MongoDAO

MongoOutputDAO implements the create method,

```
def create(self, items)
```

Here, **item** refers to a Python dictionary that contains data that is to be output to OutputDAO.

For example, if we want to store in *tweet_collection* inside *download_database* all tweets from user *id* in between *start_date* and *end_date*, we would first have to create a MongoOutputDAO object that is able to interact with the correct datastore. Note that, in general, users of any Process object would not do this; the Process object has a DAOFactory that is responsible for instantiating DAO objects.

```
output = MongoOutputDAO(location, download_database, tweet_collection)
```

Then,

```
items = {
```

```
            "id": id,

            "start_date": start_date,

            "end_date": end_date,

             "tweets": tweets

}

output.create(items)
```

The steps are very similar for MongoInputDAO. MongoInputDAO implements the read method,

```
def read(self, query):
```

Here, **query** refers to a set of attributes we want to search in the collection the MongoInputDAO refers to. The result is a list of MongoDB objects that satisfy the query.

For example, building on the previous section, say that we want to get from our MongoDB all tweets from user *id* in between dates *start_date* and *end_date*.

We would first have to create a MongoInputDAO object that is able to interact with the correct datastore.

```
input = MongoInputDAO(location, download_database, tweet_collection)
```

Then, our query would look something like,

```
query = {

            "id": id,

            "start_date": start_date,

            "end_date": end_date,

}

input.read(query)
```

The result would be a list of all documents inside the collection that have the same matching attributes.

8.3.2Mongo Daemon Generator

To use this tool, a user creates a configuration file with the following format,

```
project-name:

  location: <location of where we want the MongoDB database files to be stored>

  name: <database name>

  port: <-1 if we trying to set up a new daemon>
```

The user then passes in this configuration file to the mongo_daemon_generator Python script using administrative privileges.

```
sudo python3 mongo_daemon_generator.py mongo-project-config.yaml
```

This creates a new MongoDB daemon, where new database files are stored in the specified location. The Python script also updates the configuration file with the port used by the new MongoDB daemon.

The file is updated and looks like this,

```
project-name:

  location: <location>

  name: <name>

  port: 27017
```

Notice the port is no longer -1. When creating a datastore initial configuration file, if users want to use MongoDB, they can simply get the IP of where the daemon service is being run, as well as the updated port number from “mongo-project-config.yaml”

Users can add new entries to this file, being careful not to modify the contents of previous projects, to create new daemons,

```
project-name:

  location: <location>

  name: <name>

  port: 27017
```

```
project-name2:

  location: <location2>

  name: <name2>

  port: -1
```

Whereupon we can run mongo_daemon_generator.py again.