In this post I will talk about the Cancer Imaging Archive, a massive collection of freely available medical image data, and demonstrate how to programmatically access and retrieve that data through a Python client tapping into the provided REST API.

Introduction

Background

The Cancer Imaging Archive (TCIA)

A while back, I mentioned in my previous post about DICOM file IO in Python, that DICOM databases are usually restricting the re-distribution of their data. Well, that was before I discovered the Cancer Imaging Archive (TCIA).

Per their 'about page', TCIA is a *"large archive of medical images of cancer accessible for public download. Registering is free. All images are stored in DICOM file format. The images are organized as "Collections", typically patients related by a common disease (e.g. lung cancer), image modality (MRI, CT, etc) or research focus."*. In a nutshell, this is a **fantastic resource** for those who delve into medical image processing and I thought it would be criminal not to promote it :).

A summary of the available image 'collections' can be seen on the TCIA homepage, while a more comprehensive overview can be seen here.

Image acquisition interface

The image data is all accessible through the primary TCIA 'search' interface and general instructions on its usage can be found here. In addition, many of the collections come with auxiliary data, more information on which can be found under this link.

This 'primary' interface allows the user to filter the existing datasets by selecting the modality, anatomical region, and collection name. A series of 'results' is then presented where the user can click an 'add to cart' icon, thus collecting all series one wants to download. This interface can be seen in the next figure.

Primary image acquisition interface for the TCIA.

Once done, one can 'checkout', i.e., download, all series by clicking the 'Manage Data Basket' button and using the 'Download Manager' button to download a little Java executable which can download all the series into a folder on your computer.

REST API interface

Now I don’t want to come across like an ungrateful jackass, the TCIA people provide an invaluable service and enough image data to spend 3 lifetimes segmenting. However, I found the aforementioned interface to be slow, unresponsive, unintuitive, prone to timeouts, and overall archaic. It really overshadows the quality of the archive.

Thankfully, someone in the TCIA team probably shared my distaste for the conventional interface and decided to provide a REST API for this service which allows a user with basic development skills to access and retrieve all available image datasets programmatically. Per the API documentation the *“API is a RESTful interface, accessed through web URLs. There is no software that an application developer needs to download in order to use the API. The application developer can build their own access routines using just the API documentation provided. The interface employs a set of predefined query functions (see REST API Directory) that access TCIA databases”*.

Summary

In this post I’ll show you how to access the TCIA databases through a Python client written to take advantage of their REST API.

The process will include configuring and creating a ‘client’ to the API, performing basic queries to retrieve the examined anatomical sites, imaging modalities, and different sets of medical image data, as well as downloading such data.

API

API Key

As clearly stated in the TCIA API documentation, one needs an API key to access the interface. Unfortunately, there’s no automatized way to acquire such a key and one would have to email `help@cancerimagingarchive.net` and kindly ask them for one. However, I’ve done so already so feel free to use mine within reason to try out the interface.

Should you want to use the API extensively please acquire your own key cause I’d hate for them to ban me :D.

My API key, which is included in today’s notebook, is `16ade9bc-f2fa-4a37-b357-36466a0020fc` which can be used directly with the API client, which is discussed below.

Python Client

The TCIA API is a typical RESTful API which works pretty straightforwardly with HTTP requests. However, should you be using Python, or Java, there’s no need for you to put together your own API client code.

The TCIA folk were kind enough to put together such a client for you which is hosted under Bitbucket in the unfortunately named `TCIA-REST-API-Example` repo. A cleaned-up Python-only subset of that repo can be found in my fork here.

Python REST API access to TCIA

What you need to do in order to try out the presented code is download the `tciaclient.py` file, either from my repo fork, from the PyScience repo, or from the TCIA repo, and place it alongside today's notebook. With that in place you're ready to try it out :).

Imports

As always, let us start with the imports:

```
import tciaclient
import pandas
```

As you can see, we start by importing `tciaclient` which contains the simple client class through which we'll access the TCIA servers.

Next, we import `pandas`, which we'll use to very easily and cleanly read in the `JSON` responses as they're returned from the TCIA servers. Now if you don't know what `pandas` is/does then I can't help you, this tutorial won't focus on that but feel free to visit the official `pandas` website (do it! it will change the way you use Python).

Helper-Functions

For the purposes of this tutorial, we're defining a single `helper-function` at the beginning of today's notebook and use it throughout.

This function is `getResponseString(response)` and its code can be seen below:

```
def getResponseString(response):
    if response.getcode() is not 200:
        raise ValueError("Server returned an error")
    else:
        return response.read()
```

What this simple function does is take the server 'response', as returned from the different client methods which we'll see later, ensure that the process was successful, and read whatever data that response included as a string which it then returns.

This is super-basic Pythonic web-dev stuff but if you don't understand it don't fret, you don't really need to. Just read on :).

API Settings

Now, when we create a new client-object we need to provide some configuration options, which I'll define a-priori so as to make their purpose clear. Take a look at the code below:

```
# CAUTION: You can use my API key for your experiments but
# please try not to get me banned :). Thanks!
api_key = "16ade9bc-f2fa-4a37-b357-36466a0020fc"
baseUrl="https://services.cancerimagingarchive.net/services/v3"
resource = "TCIA"
```

Firstly, we need to specify the API key, which will give us access to the server API. As I mentioned prior, I've acquired such a key and I'm sharing it with you so that you can test this code out. However, please do not abuse it and should you need to access TCIA consistently please do email help@cancerimagingarchive.net and request one of your own.

Next, we define the base URL through which we'll be querying the server. Please refer to the API Usage Guide for details on this one. One think to note is the `v3` at the end of the URL. TCIA has been constantly updating their API (it went from v1 to v3 in a couple of months) so if you use this service a lot you might want to keep up with changes and updates. Lastly, we define the 'resource' which I don't even know what it does, I'm just following the usage guide's instructions.

Accessing Basic Information

Now we're ready to tap into the amazing resource that is TCIA! Let's first create the client object:

```
client = tciaclient.TCIAClient(api_key, baseUrl, resource)
```

Yep, that's all it takes. We simply create a new `TCIAClient` object (nestled within the `tciaclient` module imported in the beginning) and store it as `client`. Note how we pass the 'API Settings' we defined prior. Now let's finally start querying!

Query Imaging Modalities

One of the methods offered by the `TCIAClient` class is `get_modality_values`. While the class isn't documented/commented, much the methods are pretty self-explanatory. Let's see this method's signature through `help(client.get_modality_values)` which returns the following:

```
Help on method get_modality_values in module tciaclient:

get_modality_values(self, collection=None, bodyPartExamined=None, modality=None, outputFormat=
```

As you can see, we can query for all medical imaging modalities utilized in the different collections and series of medical image data. We can further filter the returned results by specifying the examined anatomical site `bodyPartExamined` or particular `collection` as a string. We can also specify the file-format of the server response which defaults to JSON but can return several other formats such as CSV, HTML, or XML. However, lets just retrieve all utilized modalities included in the TCIA. This is done as follows:

```
response = client.get_modality_values()
strRespModalities = getResponseString(response)
```

```
pandas.io.json.read_json(strRespModalities)
```

As the different arguments of the `get_modality_values` method are all optional we don't need to 'filter' the results so we call it without any arguments. We then merely 'read' the data stored in the `response` using the `getResponseString` helper-function and store the results, as a string, under `strRespModalities`.

Now the contents of the `strRespModalities` string are in a JSON format, which we can read in a myriad of ways. However, I chose to use `pandas` which has built-in JSON support and which returns a very powerful `pandas.DataFrame` object, which as we'll see later on is a godsend. In addition, IPython Notebook has built-in support for `pandas.DataFrame` objects and displays a beautiful table which, for the above code, can be seen in the next figure.

Table of the imaging modalities currently available in TCIA.

Query Anatomical Sites

Another method offered by the `TCIAClient` class is `get_body_part_values` which allows us to query all anatomical sites examined in the different image data series available on TCIA. Let's see this method's signature through `help(client.get_body_part_values)` which returns the following:

```
Help on method get_body_part_values in module tciaclient:

get_body_part_values(self, collection=None, bodyPartExamined=None, modality=None, outputFormat
```

Similarly to the case of the 'get_modality_values' method, you can see its possible to filter the returned results by specifying the imaging `modality` or particular `collection` etc. However, lets once more retrieve **all** anatomical sites by using the defaults arguments:

```
response = client.get_body_part_values()
strRespBodyParts = getResponseString(response)

pandas.io.json.read_json(strRespBodyParts)
```

You can obviously see that the code is nearly identical to the one we used to retrieve the imaging modalities, this ain't rocket science :). The above code will display a nice table with all anatomical sites, the first 10 of which you can see in the next figure.

Table of the first 10 anatomical sites of which image data is currently available in TCIA.

Query Collections

Another method I should mention is `get_collection_values`. As mentioned in the introduction, *"images are organized as "Collections", typically patients related by a common disease (e.g. lung cancer), image modality (MRI, CT, etc) or research focus."*. Querying the server for these collections is, once more, embarrassingly simple. Let's quickly take a look at the method's signature through `help(client.get_collection_values)` which returns:

```
Help on method get_collection_values in module tciaclient:

get_collection_values(self, outputFormat='json') method of tciaclient.TCIAClient instance
```

As you can see, this method doesn't really take any parameters, apart from the format the data will be returned in. The call is as simple as follows:

```
response = client.get_collection_values()
strRespCollections = getResponseString(response)

pandas.io.json.read_json(strRespCollections)
```

While you might be tempted to ignore the collection-based filtering, and instead filter your series based on more 'tangible' parameters like the imaging modality and anatomical site, generally I wouldn't recommend it. TCIA has tons upon tons of images and if your 'series' query, which we will see below, is too generic, you will end up with tens of thousands of series in the response which the TCIA server might actually refuse to send you in a glorious timeout. Keep it simple, TCIA ain't Amazon Web Services and what they're providing is free :D.

Generally, I would recommend you go through the different collections listed on the TCIA homepage so you can get a rough idea of where you want your data to come from :).

Query & Filter Series

Finally, let's get to the interesting part :). Retrieving information on 'series' of image data! The method of interest here is `get_series` whose signature we can see through `help(client.get_series)`:

```
Help on method get_series in module tciaclient:

get_series(self, collection=None, modality=None, studyInstanceUid=None, outputFormat='json') m
```

As we can see above, we can only filter series based on 'colllection', 'modality', and UID. The latter is a unique identification number (UID) separating each series from each other. You may well argue here that filtering by anatomical site should've been allowed/included but, at least at the time of writing this, it is not. Hence, until TCIA updates their API yet again and includes it, you should filter your query by collection which is loosely associated with a given anatomical site as previously outlined :).

Now let's see the query we're using in today's notebook:

```
response = client.get_series(modality="CT", collection="QIN-HEADNECK")
strRespSeries = getResponseString(response)

pdfSeries = pandas.io.json.read_json(strRespSeries)
```

As you can see above, we want all series under the 'QIN-HEADNECK' collection that were acquired through 'CT'. We used the same approach to get and read the server response but this time we're storing the results of the JSON data conversion into a `pandas.DataFrame` object by the name of `pdfSeries`.

If you display `pdfSeries` in the IPython Notebook, you will see something like 500 series that matched the set criteria! Way more than we can quickly evaluate. Hence, let's whittle these down:

```
pdfSeries[(pdfSeries.BodyPartExamined == "HEADNECK") &
          (pdfSeries.Modality=="CT") &
          (pdfSeries.ImageCount>50) &
          (pdfSeries.ImageCount<200)]
```

All I'm doing above is using the fantastic indexing provided with `pandas.DataFrame` objects to return a manageable subset of the `pdfSeries` object. Again, should the above syntax seem weird to you, I sincerely suggest you look into `pandas`, it'll change the way you think about data :).

What I'm doing above is returning a new `pandas.DataFrame` containing the series where the anatomical site was 'HEADNECK', the used imaging modality was 'CT', and where the series contains between 50 and 200 images. The selection criteria here are completely arbitrary but it shows you how to further filter the series with criteria not offered by the TCIA API.

As you can see in today's notebook, only a handful of series fit the above criteria. A part of the displayed table can be seen in the next figure.

Partial table of the returned series fitting the selection criteria.

Download Series

Now its finally time to download a series of image data. Let's first randomly choose such a series from the above table, e.g., the one on row '461', and retrieve its UID:

```
strSeriesUID = pdfSeries.ix[461].SeriesInstanceUID
```

Then let's use the `get_series_size` method to see the size of this series in bytes. Using `help(client.get_series_size)` we can see this method's signature:

```
Help on method get_series_size in module tciaclient:

get_series_size(self, SeriesInstanceUID=None, outputFormat='json') method of tciaclient.TCIACl
```

As we can see, this method needs a series UID so lets call it using the UID we just chose:

```
response = client.get_series_size(SeriesInstanceUID=strSeriesUID)
pandas.io.json.read_json(getResponseString(response))
```

which will return a nice representation of the series' size in bytes and image count as shown in the next figure.

The size in bytes and image count of the selected series.

> Note: The size in bytes you see above is slightly misleading. While that is indeed the size of the image data, the series will be provided from the server as a single `.zip` archive. After personal correspondence with the TCIA admins I learned that upon request of a series, the TCIA server will start compressing the image data and streaming the `.zip` file in real-time so as to reduce transfer time. Thus, the size of the received file will naturally be smaller than the indicated series size.

Lastly, let us use the `get_image` method to finally download the image data. Through `help(client.get_image)` we can see this method's signature:

```
Help on method get_image in module tciaclient:

get_image(self, seriesInstanceUid) method of tciaclient.TCIAClient instance
```

As we can see, we only need the series' UID to download the image data. All it takes to do so is the following:

```
response = client.get_image(strSeriesUID)
strResponseImage = getResponseString(response)
```

> Note: The process above may take a fair while depending on your bandwidth and the load the TCIA servers are under. The particular series clocks at ca. 100MB while uncompressed, and ca. 50MB when compressed so be prepared to wait a tad. Keep that in mind and don't freak out if the above takes forever to return.

Finally, having the `strResponseImage` in place we can merely save it as a `.zip` file as such:

```
with open("images.zip","wb") as fid:
    fid.write(strResponseImage)
    fid.close()
```

which results in a nice `images.zip` file being stored in our current working directory.

Finally, we can use the `zipfile` package to extract the contents of our newly download `.zip` file into a folder `images` as such:

```
import zipfile
fid = zipfile.ZipFile("images.zip")
fid.extractall("images")
```

At this point, there's obviously tons of stuff we can do with this type of data. Check back to previous posts about reading DICOM data, segmentation, surface extraction, volume rendering etc.

I've only demonstrated the basic functionality of the TCIA API, which is under active development. New types of queries are added between versions, and the community behind it is very much alive. So go ahead, give this amazing service a try, and happy medical image processing :).

Links & Resources

Material

Here's the material used in this post:

- IPython Notebook with the entire process.
- `tciaclient.py`: The module containing the TCIA API Python client.

TCIA

A few helpful links on TCIA:

- TCIA Homepage
- TCIA Wiki
- TCIA 'search' interface
- TCIA API Usage Guide

See also

Check out these past posts which were used and referenced today or are relevant to this post:

- Anaconda: The crème de la crème of Python distros
- DICOM in Python: Importing medical image data into NumPy with PyDICOM and VTK
- Surface Extraction: Creating a mesh from pixel-data using Python and VTK
- Image Segmentation with Python and SimpleITK
- Multi-Modal Image Segmentation with Python & SimpleITK
- Volume Rendering with Python and VTK

> Don't forget: all material I'm presenting in this blog can be found under the PyScience BitBucket repository.