Metadata Workshop

Release 1.0.0

Mark Hall

Aug 24, 2020

CONTENTS

This tutorial was developed for the DHd Working Group on Newspapers and Periodicals' Metadata Workshop 2020 and is also available as a `PDF`.

It will introduce you to the basics of loading metadata, filtering out unwanted metadata, transforming the metadata into the required structure, and then analysing and visualising the final metadata. You will also be introduced to the (very) basics of programming with Python.

The tutorial does not have any pre-requisites, except that you must be prepared to get your hands a little bit dirty with a bit of Python code. The tutorial makes heavy use of the Polymatheia library to hide some of the complexity involved in the load, filter, transform stages. It then uses the Pandas and Seaborn libraries for the data analysis and visualisation steps.

CHAPTER

ONE

LOADING DATA

Loading data is (obviously) the first step in doing any kind of work with metadata.

Note: The tutorial is focused on supporting research using metadata as the primary research object. For simplicity sake, the tutorial will thus refer to the metadata as data.

In this part of the tutorial we will look at fetching data from a number of sources:

- OAI-PMH endpoints,
- the Europeana API,
- the local filesystem.

We will also look at storing the retrieved data locally, which is necessary for performance and reproducibility reasons.

However, first you will briefly be introduced to the tutorial interface on the next page.

1.1 Interface

To get this far in the tutorial, you have already used some parts of the interface, so this will introduce you to the remaining elements and in particular the Jupyter notebook interface that you will use for writing your code.

The interface as a whole is split into two main sections. On the left is the tutorial text that will guide you through the tutorial itself. On the right you have the Jupyter notebook in which you will write and then run your code.

1.1.1 Tutorial Text

To get back to the start of the tutorial, you can always click on the “Metadata Workshop” or “Home” links. Next to the “Home” link, you can see the list of blocks that make up the tutorial’s main structure.

Below the block list and at the bottom of each tutorial page are the navigation elements that allow you to move within a block. You can use the “Next” and “Previous” links to move through the tutorial page-by-page. If you wish to jump to another page within the block, click on the link in the middle of the navigation bar. This will always be the title of the current page and a little structure icon. Clicking on this opens an overlay that shows you all the pages within the current block and you can navigate to a page by clicking on it.

1.1.2 Jupyter Notebook

On the right you see the Jupyter notebook for the current tutorial page or pages. At various points in the tutorial as you move between pages, the Jupyter notebook will change. However, any changes you have made will automatically be saved and will still be there when you come back.

Each Jupyter notebook consists of a list of cells. Each cell contains either code or text. To add another cell to the Jupyter notebook, you can either click on the plus icon in the toolbar or double-click below any cell. You can then use the dropdown in the toolbar to specify whether the cell should be a code cell or a text cell.

To delete a cell, simply click on the "cut" icon in the toolbar (the small scissors) or select the cell and then press the "d" key twice.

Code cells

When you add a cell, it is always a code cell. A code cell always consists of one or more lines of code to be run and the output of running the code below.

The Jupyter notebook on the right initially contains a single code cell. Into that type the following code:

```
hello = 'Metadata rule the world!'
hello
```

Then press `Ctrl + Enter` on your keyboard to run the code in the cell (you can also click on the "Run" button in the toolbar). You will see that in the output area below the code it will say `'Metadata rule the world!'`.

Text cells

Text cells are mainly there for adding notes to your code. To get a new text cell, first add a cell, then, in the toolbar dropdown that says "Code", select "Markdown". That will convert the cell to a text cell and you can then simply type whatever you want to write down. If you want to format the text, you can use Markdown to do so.

Try it out now by adding a cell to the notebook, switching that to a text cell, and then adding a quick note.

When you create a new text cell, then it is in editing mode. If you want to switch it to display mode, press `Ctrl + Enter` on your keyboard. To switch a text cell from display to edit mode, double-click on the cell.

1.2 Loading data via OAI-PMH

The first metadata format we will look at loading is OAI-PMH. It is a generic protocol designed for harvesting large amounts of metadata from an archive and was initially intended primarily as an archive-to-archive metadata exchange protocol. As a result of this focus, the OAI-PMH protocol provides only very limited facilities for filtering data on the server side. Instead the standard pattern is to download complete data-sets from the server and then filter and transform locally (which is what we are going to do here).

The OAI-PMH protocol provides a number of functions for accessing aspects of the archive, but for our purposes we will only look at the following:

- Formats: What metadata formats are provided by the archive;
- Records: Loading the actual metadata records;
- Sets: What sets of records are available for filtering the metadata records.

For the tutorial we will use the OAI-PMH endpoint of the DigiZeitschriften project, which has its OAI-PMH endpoint at http://www.digizeitschriften.de/oai2/. If you click on the OAI-PMH endpoint link, you will see a human-readable version of the OAI-PMH responses and you can explore the data a bit through that interface as well, which is useful when you are starting out with a new archive.

1.2.1 Available Metadata Formats

The first step when accessing an OAI-PMH archive is to determine which formats you can request the metadata in, when you later fetch the actual metadata records. The first step to doing this is to import the `OAIMetadataFormatReader` class, provided by the Polymatheia library, into our notebook. To do this, add the following code into the first cell of the notebook:

```
from polymatheia.data.reader import OAIMetadataFormatReader
```

Then run the cell. If you do not get an error, then this has worked perfectly. Otherwise check for typos and then re-run the cell.

Note: It is generally good practice to put all your imports together at the top of your notebook. That means that later in the tutorial, when we import additional classes, simply add the import statements into this same cell and re-run the cell.

Now that we have our reader class for the metadata we can use this to find out which metadata formats the archive supports. Into a new cell in the notebook add the following:

```
1 reader = OAIMetadataFormatReader('http://www.digizeitschriften.de/oai2/')
2 for format in reader:
3     print(format)
```

Now run the cell and the output should look something like this:

```
{
  "schema": "http://www.openarchives.org/OAI/2.0/dc.xsd",
  "metadataPrefix": "oai_dc",
  "metadataNamespace": "http://purl.org/dc/elements/1.1/"
}
{
  "schema": "http://www.loc.gov/mets/mets.xsd http://www.loc.gov/standards/mods/v3/
↪mods-3-2.xsd",
  "metadataPrefix": "mets",
  "metadataNamespace": "http://www.loc.gov/METS/ http://www.loc.gov/mods/v3"
}
```

Before we look at the output in a bit more detail, let us have a quick look at the code we use:

- In line #1 we first create a new `OAIMetadataFormatReader` object with a single parameter, the base URL of the OAI-PMH endpoint. This new object is then assigned to the `reader` variable. At this point no data has been fetched from the OAI-PMH server. That will only happen in line #2.
- In line #2 we use a "for" loop to loop over the format objects provided by the `reader` object we created in the previous line. It is at this point that the `OAIMetadataFormatReader` fetches the list of available metadata formats from the OAI-PMH server and makes them available to the `for` loop.

  The "for" loop is one of the core concepts of programming. It allows us to apply a series of instructions to a list of things. In this case the things are the format objects provided by the `reader`. In each iteration of the loop one format object is made available via the `format` variable. Then the so-called body of the loop, which are those instructions that follow the for line and are indented by at least 4 spaces, is run. In our case there is only one instruction in the body of the loop, which is line #3. This line will be run once for each format object provided by the reader.

- In line #3 we simply print out the content of the format variable. In each iteration of the loop the format will have different content, but the same code will be applied, here printing out the format object.

If you look at the output, you can see that two metadata formats are provided by the OAI-PMH server. This means that the body of the for loop was run two times, each time printing out the content of one format object. If you look at the output of the individual format objects, you can see that they have three properties:

- *schema*: The schema definition for the metadata format;
- *metadataPrefix*: The prefix used to identify this metadata format, which we will later on use to fetch records in a specific format;
- *metadataNamespace*: The XML namespace the schema uses.

In practice we will only use the *metadataPrefix*, so let us try to output only this in the for loop. Add the following code into a new cell:

```
for format in reader:
    print(format.metadataPrefix)
```

If you run the cell, the output will look as follows:

```
oai_dc
mets
```

There are two differences between the two cells. First, in the second cell we don't create the reader object. Instead, we will simply re-use the reader object created in the previous cell. This illustrates an important aspect of the notebook, namely that while there are individual cells, all cells in a notebook share the same execution environment, thus anything defined in a cell that has been run is available to all other cells.

The second difference is line #2. In the second example we use "dot-notation" to access a specific field within the format object (format.metadataPrefix). You can try replacing the "metadataPrefix" with one of the other two properties in the format object ("schema" or "metadataNamespace"). Then re-run the cell to see what the output looks like.

Now that we know which metadata formats are available we can move on to loading some actual metadata records.

1.2.2 Fetching Records

Fetching the actual records follows the same pattern as fetching the metadata formats. First, we need to import the reader class. Add the following into the first cell of the notebook and then re-run that cell:

```
from polymatheia.data.reader import OAIRecordReader
```

Next, in a new cell at the bottom of the notebook add the following code:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=1)
for record in reader:
    print(record)
```

- Line #1 creates a new OAIRecordReader, again passing the OAI-PMH endpoint URL as the first parameter. By default the OAIRecordReader loads all records from the OAI-PMH server, which can be many thousand. We thus provide a second parameter max_records=1, which tells the OAIRecordReader to return at most 1 records.

- Line #2 sets up a for loop to loop over all records provided by the `OAIRecordReader`. In this case, this will only be one record, but you can change that parameter in line #1 to return multiple records.
- Line #3 prints the record. In practice this is where we would do more complex processing of each record.

By default the `OAIRecordReader` fetches the data in "oai_dc" format, which looks like this:

```
{
  "header": {
    "identifier": {
      "_text": "oai:www.digizeitschriften.de:PPN720885019_3_0114"
    },
    "datestamp": {
      "_text": "2020-06-30T08:31:20Z"
    },
    "setSpec": {
    }
  },
  "metadata": {
    "{http://www.openarchives.org/OAI/2.0/oai_dc/}dc": {
      "_attrib": {
        "xsi_schemaLocation": "http://www.openarchives.org/OAI/2.0/oai_dc/ http://www.
↪openarchives.org/OAI/2.0/oai_dc.xsd"
      },
      "dc_title": {
        "_text": "Bl\u00e4tter f\u00fcr w\u00fcrttembergische Kirchengeschichte 114"
      },
      "dc_subject": [
        {
          "_text": "200.religion"
        },
        {
          "_text": "PeriodicalVolume"
        }
      ],
      "dc_language": {
        "_text": "ger"
      },
      "dc_publisher": {
        "_text": "Scheufele"
      },
      "dc_date": {
        "_text": "2014"
      },
      "dc_type": {
        "_text": "Text"
      },
      "dc_format": [
        {
          "_text": "image/jpeg"
        },
        {
          "_text": "application/pdf"
        }
      ],
      "dc_identifier": [
        {
          "_text": "http://resolver.sub.uni-goettingen.de/purl?PPN720885019_3_0114"
        },
```

(continues on next page)

(continued from previous page)

```
        {
          "_text": "DigiZeit PPN720885019_3_0114"
        }
      ],
      "dc_source": {
        "_text": "Scheufele: Bl\u00e4tter f\u00fcr w\u00fcrttembergische␣
↪Kirchengeschichte. Stuttgart 2014"
      },
      "dc_rights": [
        {
          "_text": "DigiZeitschriften Abo"
        },
        {
          "_text": "VereinWKG"
        },
        {
          "_text": "Religion"
        }
      ]
    }
  }
}
```

To switch to using METS/MODS, we add an extra parameter to when we create the new `OAIRecordReader`, specifying the `metadata_prefix` of the format you want to use. You can use any of the prefixes that the `OAIMetadataReader` returns. Update the cell to look like this and then re-run it:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=1,␣
↪metadata_prefix='mets')
for record in reader:
    print(record.header.identifier._text)
```

The METS/MODS output is too large to be included here, but you can see that it follows the same basic structure as for the "oai_dc" metadata.

As with the `OAIMetadataReader`, you can navigate the records using "dot-notation". We can start by limiting our output to the "metadata" key of the record:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=1,␣
↪metadata_prefix='mets')
for record in reader:
    print(record.metadata)
```

If you look at the output, you will see that the first key is "{http://www.loc.gov/METS/}mets". Because this includes a ".", we cannot use dot-notation on its own to access it. Instead we need to use square-bracket notation, so update the code to look like this and then re-run the cell:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=1,␣
↪metadata_prefix='mets')
for record in reader:
    print(record.metadata['{http://www.loc.gov/METS/}mets'].mets_dmdSec)
```

Looking at the output here, we can see that it starts with a square bracket "[". This means that the "mets_dmdSec" element is a list. To access a specific element within the list we again use the square-bracket notation. The difference is that now we simply put the index of the element we wish to access between the square brackets:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=1,␣
↪metadata_prefix='mets')
for record in reader:
    print(record.metadata['{http://www.loc.gov/METS/}mets'].mets_dmdSec[0])
```

We are now digging deep into the METS/MODS metadata, so let's update the code a bit more to pick out the title. You can either update the cell or add a new cell, if you want to be able to compare the dot-notation path with the original metadata:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=1,␣
↪metadata_prefix='mets')
for record in reader:
    print(record.metadata['{http://www.loc.gov/METS/}mets'].mets_dmdSec[0].mets_
↪mdWrap.mets_xmlData.mods_mods.mods_titleInfo.mods_title._text)
```

We can now increase the number of records to show, by increasing the `max_records` parameter:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=10,␣
↪metadata_prefix='mets')
for record in reader:
    print(record.metadata['{http://www.loc.gov/METS/}mets'].mets_dmdSec[0].mets_
↪mdWrap.mets_xmlData.mods_mods.mods_titleInfo.mods_title._text)
```

If we remove the `max_records` parameter, then the code would print out the titles of all records provided by the archive (which in this case would be about 1 million records).

As most archives provide large numbers of records, in the ideal case we would fetch only a sub-set. While OAI-PMH does not provide fine-grained filtering methods, it does allow requesting sub-sets of the data and the next step is to look using those.

1.2.3 Filtering by Set

OAI-PMH provides only one way of filtering on the server-side and that is the "Set". Sets are constructed by the archive administrators, by attaching one or more set specs (essentially unique identifiers) to each metadata record. Then, when fetching records, we can specify one set spec and it will return only those records that have been annotated with that spec.

The first step when using sets is always to fetch the list of all available sets, to see what options are available. You should by now be familiar (and hopefully comfortable) with the pattern for this. First, add the following line to the first cell in the notebook and then re-run the cell:

```
from polymatheia.data.reader import OAISetReader
```

Next, add a new cell to the end of the notebook and add the following code:

```
reader = OAISetReader('http://www.digizeitschriften.de/oai2/')
for setSpec in reader:
    print(setSpec)
```

As you can see from the output, each set spec consists of a "setSpec", which is the unique identifier, and a "setName", which is a human-readable version. We use the human-readable version to decide which set to use and then we use the unique identifier for filtering. In this example we will use the "Europeana" set, which has the set spec "EU". To filter, add the following code to a new cell at the end of the notebook:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=10,␣
↪metadata_prefix='mets', set_spec='EU')
for record in reader:
    print(record.metadata['{http://www.loc.gov/METS/}mets'].mets_dmdSec[0].mets_
↪mdWrap.mets_xmlData.mods_mods.mods_titleInfo.mods_title._text)
```

When you run this code, you are most likely going to get a `KeyError`. This is because sometimes the `mets_dmdSec` is a single object and sometimes it is a list of objects. To make the code work we need to distinguish these two cases. The way to do that in code is the `if` statement. Update the cell so that it looks like this:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=10,␣
↪metadata_prefix='mets', set_spec='EU')
for record in reader:
    if isinstance(record.metadata['{http://www.loc.gov/METS/}mets'].mets_dmdSec,␣
↪list):
        print(record.metadata['{http://www.loc.gov/METS/}mets'].mets_dmdSec[0].mets_
↪mdWrap.mets_xmlData.mods_mods.mods_titleInfo.mods_title._text)
    else:
        print(record.metadata['{http://www.loc.gov/METS/}mets'].mets_dmdSec.mets_
↪mdWrap.mets_xmlData.mods_mods.mods_titleInfo.mods_title._text)
```

When you run the cell, you will see that now your code outputs ten title.

The polymatheia library provides an alternative function for getting values from a record. Add a new cell to the end of the notebook with this code:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=10,␣
↪metadata_prefix='mets', set_spec='EU')
for record in reader:
    print(record.get(['metadata', '{http://www.loc.gov/METS/}mets', 'mets_dmdSec',
↪'mets_mdWrap', 'mets_xmlData', 'mods_mods', 'mods_titleInfo', 'mods_title', '_text
↪']))
```

Each record comes with a `get` function, which we can either pass a path as a single dotted string, or if, as in this case, one of the path elements contains a full-stop, a list of path elements. The `get` function does one thing differently to just using the dot-notation. If a path element identifies a list (such as the `'mets_dmdSec'` element) and the next element is not a list index (`'mets_mdWrap`), then the `get` function will apply the remainder of the path to each element in the list and return a list of values. If you run this cell, you will see that you now get 9 lists of titles and one single title.

Which one of the two approaches you use depends on the specific scenario and whether you need to control whether a certain element is a list value or not.

As OAI-PMH is not really meant for repeatedly fetching small structures, but rather at fetching everything and then working locally, the next step is to look at how to store things locally.

1.3 Storing data locally

Fetching data remotely is a comparatively slow process, so in practice it is better to fetch the data once and store it locally.

To do that we first need to import the `LocalWriter` class into the first cell of the notebook and re-run the cell:

```
from polymatheia.data.writer import LocalWriter
```

The `LocalWriter` requires a unique identifier for each record, which it then uses to structure how the data is stored locally. Since we are dealing with OAI-PMH data at the moment, this information is best found in the header. Add and run the following code in a new cell:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=10,␣
↪metadata_prefix='mets', set_spec='EU')
for record in reader:
    print(record.header)
```

The code will print out 10 headers in a row. You can see in the output that each record's header has an "identifier" entry, which contains the unique identifier for that record. This is something we need, so update the code so that it looks as follows:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=10,␣
↪metadata_prefix='mets', set_spec='EU')
for record in reader:
    print(record.header.identifier._text)
```

Re-run the cell and you will see 10 identifiers printed out. Now that we know where to get the unique identifier for a record, add a cell with the following code:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=10,␣
↪metadata_prefix='mets', set_spec='EU')
writer = LocalWriter('digiz_eu', 'header.identifier._text')
writer.write(reader)
```

Lets have a look at that new elements in the code:

- Line #2 creates a new `LocalWriter` in the `writer` variable, which takes two parameters. The first is the name of the directory into which to store the data. The second is the dotted path used to access the identifier value.
- Line #3 instructs the `writer` to write all the records returned by the `reader`.

If you run the cell, this will take a few seconds and if everything worked, it will produce no visible output. You can, however, see that it is working by the "[*]" indicator on the left side of the cell. The `LocalWriter` saves each record as a JSON (JavaScript Object Notation) file on the local filesystem and on the next page of the tutorial we will look at loading this data gain.

Let us now fetch a slightly larger dataset that we can then work with. Update the cell, increasing the `max_records` to 1000 and re-run the cell:

```
reader = OAIRecordReader('http://www.digizeitschriften.de/oai2/', max_records=1000,␣
↪metadata_prefix='mets', set_spec='EU')
writer = LocalWriter('digiz_eu', 'header.identifier._text')
writer.write(reader)
```

This will take a while, so this is probably a good time to take a short break.

Note: The code here writes OAI-PMH records to the local disk. However, the `LocalWriter` does not actually care about the type of records it writes and will work just as well with data read from any of the other data-sources that Polymatheia provides.

1.4 Loading data from disk

After storing data, we obviously need to load it again, in order to make use of it. Polymatheia provides two methods for reading data from disk. The `LocalReader` is used to load JSON data, and the `CSVReader` for loading data from CSV files.

1.4.1 Loading JSON data

As stated earleir, the `LocalWriter` stores records as JSON files. In order to load these back into the notebook, add the following into the cell at the top and re-run the cell:

```
from polymatheia.data.reader import LocalReader
```

Then add and run a cell at the end of the notebook with the following code:

```
reader = LocalReader('digiz_eu')
count = 0
for record in reader:
    count = count + 1
print(count)
```

Unlike the other readers, the `LocalReader` only takes a single parameter, which is the directory the `LocalWriter` stored the data in. The rest of the code example shows another pattern that is often useful. We first define a `counter` and set it to 0 (line #2). Then we iterate over all records and increment the value of the `counter` each time (line #3-4). Then, after the `for` loop completes, we print out the value of `counter`.

The result should be that "1000" is output, showing that the `LocalWriter` did retrieve and store 1000 records.

Note: While in this tutorial we used the `LocalReader` to load data saved with the `LocalWriter`, the `LocalReader` will load any kind of JSON data. The only constraint is that each JSON file **must** contain exactly one record.

1.4.2 Loading CSV data

Polymatheia also provides functionality for loading data from CSV files. We won't be using it in this tutorial, but if you need to load data from a CSV file, then use the following code:

```
from polymatheia.data.reader import CSVReader
reader = CSVReader('filename.csv')
```

The `CSVReader` object can then be used wherever you have used any other readers.

1.5 Loading data from Europeana

We will now look at a second remote data-source: Europeana. Europeana is the European aggregator for cultural heritage data and comes with its own metadata schema and a custom API (application programming interface) for accessing the data. In order to use the API you need to have an API Key, which you need to apply for here before we continue.

When you have your key, we can start with acessing the Europeana API. First, we need to import the `EuropeanaSearchReader` in the first cell, by adding the following code and then re-running the cell:

```
from polymatheia.data.reader import EuropeanaSearchReader
```

Next, add a cell at the end of the notebook and add the following code:

```
EUROPEANA_API_KEY = ''
reader = EuropeanaSearchReader(EUROPEANA_API_KEY, 'Python', max_records=1)
for record in reader:
    print(record)
```

Copy your Europeana API Key between the `''` string markers in the first line. Then run the cell. If everything works, you will see a single record output. We don't have time to go through all the fields in detail here, but you can find a description of all the fields in the Europeana Search API documentation.

Creating a new `EuropeanaSearchReader` always requires at least two parameters. The first is the Europeana API key and the second is the search query. In our first example we simply search for the string "Python". If you want, you can try replacing that with a different value and seeing what the results are like.

We can interact with a Europeana record in the same way as with an OAI-PMH one, using dot-notation. Update the code in the cell to fetch 10 records and output only the country value:

```
EUROPEANA_API_KEY = ''
reader = EuropeanaSearchReader(EUROPEANA_API_KEY, 'Python', max_records=10)
for record in reader:
    print(record.country)
```

1.5.1 Complex Queries

Europeana provides full documentation of the Europeana query language. In this tutorial we will just look at some of the basics. First, add a new cell with the following content:

```
reader = EuropeanaSearchReader(EUROPEANA_API_KEY, '"Karl Gutzkow"', max_records=10)
for record in reader:
    print(record.title)
```

Using the double quotes ensures that the results contain the text as it is. If you want to have all the keywords in your results, but are prepared to allow for extra text between them, try the following code in a new cell:

```
reader = EuropeanaSearchReader(EUROPEANA_API_KEY, 'Karl AND Gutzkow', max_records=10)
for record in reader:
    print(record.title)
```

By default the search will run across all fields. To search only within a single field, you can specify the field in the search query. Try this code in a new cell:

```
reader = EuropeanaSearchReader(EUROPEANA_API_KEY, 'title:Karl AND title:Gutzkow', max_
↪records=10)
for record in reader:
    print(record.title)
```

We cannot cover all the available fields here, instead look at the Europeana documentation.

1.5.2 Result Profiles

The Europeana Search API provides different levels of detail in the results. By default it provides the "standard" result profile and for determining what query to use to find the data-set one is interested in, it is ideal. However, when downloading the data for further processing, the "rich" profile is generally better, as it provides the maximum level of detail. Run the following in a new cell:

```
reader = EuropeanaSearchReader(EUROPEANA_API_KEY, 'title:Karl AND title:Gutzkow',␣
↪profile='rich', max_records=1)
for record in reader:
    print(record)
```

If you compare the output to the output from the previous cell, you will see the additional data that is provided.

Note: It is important to note that the profile level specifies the maximum level of detail. If a record does not have any values for a field, then that field will not be contained within the response.

1.5.3 Reusability

All records in the Europeana archive are provided with rights information, specifying what use is allowed. To restrict the results to, for example, those where any kind of re-use is possible, we use the `reusability` parameter. Run the following code in a new cell:

```
reader = EuropeanaSearchReader(EUROPEANA_API_KEY, 'title:Karl AND title:Gutzkow',␣
↪reusability='open', max_records=1)
for record in reader:
    print(record)
```

Running this code will only return those records that are freely re-usable. This includes public domain works, and CreativeCommons Attribution and Attribution-ShareAlike works. If you want to narrow it down more specifically, you need to filter in the query:

```
reader = EuropeanaSearchReader(EUROPEANA_API_KEY, 'title:Karl AND title:Gutzkow AND␣
↪RIGHTS:"http://creativecommons.org/publicdomain/mark/1.0/"', reusability='open',␣
↪max_records=1)
for record in reader:
    print(record)
```

1.5.4 Bigger Dataset

Before we move on, we will just fetch a second larger data-set to use for the rest of the tutorial. Add a new cell with the following code and run it:

```
reader = EuropeanaSearchReader(EUROPEANA_API_KEY, 'Gutzkow OR Zäunemann OR Heyse',␣
↪profile='rich')
writer = LocalWriter('europeana_test', 'guid')
writer.write(reader)
```

As you can see the code is very similar to the code we used for saving the OAI-PMH data. The difference is only the name of the directory to store the data in and the path for the unique identifier which for Europeana is the "guid" field. This will take a while to download all the data. When it has completed, you can move on to the next step in the tutorial.

CHAPTER TWO

FILTERING & TRANSFORMATION

Whether you fetch the data from an OAI-PMH server, a specific API like Europeana, or have the data available locally, the next step in working with the data is to filter out those records that are of no interest and transform the interesting records into the required format for analysis and visualisation.

While the tutorial will first look at filtering and then at transforming, in practice it is more common that these two steps are performed alternatingly and repeatedly in order to produce the final data-set for analysis.

2.1 Filtering

We shall start by looking at the filtering possibilities provided by Polymatheia. Let's get started by adding a new cell to the new notebook on the right and running the following code:

```python
from polymatheia.data.reader import LocalReader
from polymatheia.filter import RecordsFilter
```

Then add and run another cell to read in the data we will use for the filtering and transformation:

```python
reader = LocalReader('europeana_test')
```

Now that we have the basic code in place, add a new cell with the following code and then run it:

```python
1  fltr = ('true',)
2  filtered = RecordsFilter(reader, fltr)
3  count = 0
4  for record in filtered:
5      count = count + 1
6  print(count)
```

You will see that the output is 1009, the same number of records that the original Europeana query returned and that we stored locally.

The only real difference to the code we have written in previous parts of the tutorial are lines #2 and #3. In line #2 we define a new variable `fltr` (the mis-spelling is on purpose, as `filter` is a reserved keyword in Python) and assign it the value `('true',)`. The round brackets of the filter expression define what in Python is known as a "tuple". A tuple is an ordered list of values that *cannot* be changed after being created. In polymatheia all filters are defined as tuples and the simplest filter is the `'true'` filter we use here, which simply lets any record pass.

In line #3 we create a new `RecordsFilter`, passing it the `reader` and a `fltr`, storing the new `RecordsFilter` in the `filtered` variable. The `RecordsFilter` instance can then be used like any other reader, thus we can use it in the `for` loop. The `record` variable in the `for` loop (line #4) is then assigned each of the records provided by the original `reader` that match the filter expression. Because we use the `'true'` filter, all 1009 records are passed through by the filter.

The `'true'` filter is not that interesting and is mainly used internally by Polymatheia for performance reasons. Polymatheia implements two groups of filters that we will now look at: *Basic Filters* and *Compound Filters*.

2.1.1 Basic Filters

The basic filters provided by Polymatheia allow you to compare a value in a record to a fixed value. We have already seen the simplest basic filter, the `'true'` filter. Polymatheia provides the following basic filters:

- `('true',)`: Lets any record pass.
- `('false',)`: Lets no record pass.
- `('eq', a, b)`: Lets the record pass if the value of `a` is equal to the value of `b`.
- `('neq', a, b)`: Lets the record pass if the value of `a` is not equal to the value of `b`.
- `('gt', a, b)`: Lets the record pass if the value of `a` is greater than the value of `b`.
- `('gte', a, b)`: Lets the record pass if the value of `a` is greater than or equal to the value of `b`.
- `('lt', a, b)`: Lets the record pass if the value of `a` is less than the value of `b`.
- `('lte', a, b)`: Lets the record pass if the value of `a` is less than or equal to the value of `b`.
- `('contains', a, b)`: Lets the record pass if the value of `a` is contains the value of `b`.
- `('exists', a)`: Lets the record pass if the value of `a` is not `None`.

Where the filter expression contains `a` and `b`, either of these can be one of:

- A dotted string: in this case the value to be compared is taken from the record using the dotted string to identify the value to compare.
- A list: the value to be compared is taken from the record using the list to identify the value to compare.
- Anything else: the value is compared as is.

We can now try out a few of the basic filters.

Filtering by type

The first thing we will try out is filtering records so that only the image records are returned. Add a new cell at the bottom of the notebook and add the following code:

```
fltr = ('eq', ['type'], 'IMAGE')
images = RecordsFilter(reader, fltr)
count = 0
for record in images:
    count = count + 1
    print(record.type)
print(count)
```

If you look at the code, the only interesting change is the definition of the `fltr` variable, which here is defined as an equal filter. We use the list format to specify the field to fetch from the record for comparison, because if we just used the string notation, there would be no "." in the string, and it would not be recognised as a dotted path.

If you run the cell, you will see that we now get 101 image records. We could also flip this around to show anything but the image records, by replacing the `'eq'` with `'neq'` in the filter. Try it out and see what the result looks like.

Filtering by completeness

Now let us try out the numerical comparison filters. Each Europeana record comes with a field "europeanaCompleteness", which specifies how complete that metadata record is. We can now use that to filter our records to high quality items. Add a new cell at the bottom of the notebook and add the following code:

```
fltr = ('gt', ['europeanaCompleteness'], 9)
complete = RecordsFilter(reader, fltr)
count = 0
for record in complete:
    count = count + 1
print(count)
```

The completeness value is in the range 1 to 10, so testing if the value is greater than 9 is essentially the same as testing for equality with 10. If you run the cell, you will see that there are 159 records with a europeanaCompleteness of 10.

We can now play with the comparison operator. First, let us say that we are happy with a value of 9 as well in our records. Change the `'gt'` to `'gte'` to allow values of 9 or 10 and re-run the cell. You will see that now there are 258 records.

We can do the opposite as well. By replacing `'gte'` with `'lt'` we can find those records that have a completeness value smaller than 9, which produces 751 records. This also validates that the filters are doing what we expected, as 751 records plus the 258 greater-than-or-equal records results in the correct total of 1009 records.

Filtering by language

Finally we will look at filtering by language. Add the following code into a new cell at the bottom of the notebook:

```
fltr = ('contains', ['dcLanguage'], 'ger')
german = RecordsFilter(reader, fltr)
count = 0
for record in german:
    count = count + 1
print(count)
```

If you run the cell, you will see that a total of 103 records are returned. However, as we will see in a moment, when we move on to complex filters, these are not all the German language records and we need to do a bit more work to see all of them.

Sometimes at the filtering step we don't care what the value is, just that it is there. For that we can use the `'exists'` filter. Add and run a new cell with the following code:

```
with_language = RecordsFilter(reader, ('exists', ['dcLanguage']))
count = 0
for record in with_language:
    count = count + 1
print(count)
```

You will see that it shows that 923 of the records have the dcLanguage attribute set.

The advantage of the `'exists'` filter is that for any further processing it is guaranteed that the "dcLanguage" value exists, making further processing much simpler.

2.1.2 Compound Filters

Very often you will want to filter the records based on more than a single filter expression. To do this Polymatheia provides three compound filters:

- `('not', filter_expression)`: Lets the record pass if the `filter_expression` is not `True`.
- `('or', filter_expression_1, ..., filter_expression_n)`: Lets the record pass if one or more of the `filter_expression_1` to `filter_expression_n` is `True`.
- `('and', filter_expression_1, ..., filter_expression_n)`: Lets the record pass only if all `filter_expression_1` to `filter_expression_n` are `True`.

Negation

For most of the basic filters, the `'not'` filter is not needed, as they come with their own negation equivalents. The exception is the `'contains'` filter. For example if we want to find all records that are not in German, we would run the following code in a new cell:

```
fltr = ('not', ('contains', ['dcLanguage'], 'de'))
not_german = RecordsFilter(reader, fltr)
count = 0
for record in not_german:
    count = count + 1
print(count)
```

If you run the cell, you will see that there are 414 records that do not have a “dcLanguage” of “de”.

OR filtering

When we looked at filtering with the basic `'contain'` filter, we noted that filtering by `'de'` does not produce all German language records. This is because some are annotated with the three-letter language code `'ger'`. Because the `'contains'` filter only supports a single value, we need to use multiple `'contains'` filters and then combine them using the `'or'` filter. Add a new code cell with the following code:

```
fltr = ('or', ('contains', ['dcLanguage'], 'de'), ('contains', ['dcLanguage'], 'ger'))
full_german = RecordsFilter(reader, fltr)
count = 0
for record in full_german:
    count = count + 1
print(count)
```

You will see that the filtered data now contains 698 records. If you go back to the basic `'contains'` filter code and try it with the two values, you will see that `'de'` produces 595 records and `'ger'` 103, which in sum is exactly the 698 records produced here. This might not always be the case, as it is possible that a record in your data matches more than one of the nested filter expressions, in which case the combined total will be less than the individual counts.

AND filtering

Similar to the scenario with the `'or'` filter, you sometimes want to require that multiple filter expressions are `True` on a record. For example, we might want to get all German language images. Create a new cell and add the following code:

```
fltr = ('and', ('contains', ['dcLanguage'], 'de'), ('eq', ['type'], 'IMAGE'))
german_images = RecordsFilter(reader, fltr)
count = 0
for record in german_images:
    count = count + 1
print(count)
```

As you can see when you run the cell, combining filter expressions can quickly reduce a large data-set to something small and more manageable.

Nesting all the way down

Obviously in the last code sample we did not include all German language images, as we were missing the records with the “dcLanguage” “ger”. To include all of them, we can create complex filter expressions that combine multiple operators. Create a new cell and add the following code:

```
fltr = ('and',
             ('or',
                 ('contains', ['dcLanguage'], 'de'),
                 ('contains', ['dcLanguage'], 'ger')),
             ('eq', ['type'], 'IMAGE'))
full_german_images = RecordsFilter(reader, fltr)
count = 0
for record in full_german_images:
    count = count + 1
print(count)
```

As you can see when you run the cell, this only includes a single extra image, but that image might be very important, so good to have found that too.

When we write more complex filters, such as this one, it is generally a good idea to use indentation to indicate how the filter expressions are nested. Compare the following line of code, which does **exactly** the same:

```
fltr = ('and', ('or', ('contains', ['dcLanguage'], 'de'), ('contains', ['dcLanguage'],
↪ 'ger')), ('eq', ['type'], 'IMAGE'))
```

The difference in readability is clear and applies to all other code as well. Remember to also add notes to your notebooks to document what you have done.

Important: Experience from software development shows that code you wrote yourselv, but haven’t looked at for two weeks is as readable and understandable as code that somebody else wrote. Thus it is imperative to cleanly structure your code and provide comments as well, so that you can remember what you were thinking.

2.2 Transformation

For the analysis and visualisation the data generally need to be in a tabular format. As we have seen, this is generally not the case for the source metadata. However, even if the source metadata is already tabular, it is often useful to transform it into a simpler form for the analysis.

We will first look at some *Basic Transformations* and then *Complex Transformations*.

Let's get started by adding a new cell to the new notebook on the right and running the following code:

```
from polymatheia.data.reader import LocalReader
from polymatheia.filter import RecordsFilter
from polymatheia.transform import RecordsTransform
```

Then add and run another cell to read in the data we will use for the filtering and transformation:

```
reader = LocalReader('europeana_test')
```

2.2.1 Basic Transformations

The basic transformations are all designed for taking one or more values from the source record and converting those values into a single value in the output record. Polymatheia supports the following basic transformations:

- `('copy', target, source)`: copy the value identified by the dotted path `source` to the location specified by the dotted path `target`.
- `('static', target, value)`: set the static `value` at the location specified by the dotted path `target`.
- `('fill', target, value)`: set the static `value` at the location specified by the dotted path `target` **IF** there is no pre-existing value at the dotted path `target` or if that value is `None`.
- `('join', target, joiner, source1, source2, ...)`: join together the values identified by the dotted paths `source1`, `source2`, `...` using the string `joiner` and set it at the location specified by the dotted path `target`.
- `('custom', target, function)`: call the function `function` with the record as the parameter and store the returned value at the location specified by the dotted path `target`.

Single value transformations

The most commonly used transformation is `'copy'`. Add a new cell with the following code:

```
mapping = ('copy', 'lang', 'dcLanguage[0]')
transformed = RecordsTransform(reader, mapping)
for record in transformed:
    print(record)
```

As you can see, the basic pattern is the same we saw when filtering records. The transformation `mapping` is specified as a tuple in line #1. Here we specify a transformation that will extract the language specified on the record. Because the "dcLanguage" field is always a list, we use the `[0]` subscript to fetch the first value in the list. The result is stored under the key "lang". Then in the next line (#2) we create a new `RecordsTransform` with the `reader` providing the source data and the `mapping` to apply.

When you run the code, you will see that it produces a long list of outputs that look like this (the language values will vary):

```
{
    "lang": "de"
}
```

Next, let us look at the `'static'` transformation. Add a new cell with the following code:

```
mapping = ('static', 'source', 'europeana')
filtered =
transformed = RecordsTransform(reader, mapping)
for record in transformed:
    print(record)
```

If you run the code, you will see that, as promised by the name of the transformation, it simply sets the static value. At first look there may not seem to be much point to this, but it can often be useful to set specific static data that indicates either where the data came from or how it has been processed. In particular if you later merge the data with other data, this static value can then be used to distinguish things in the analysis.

We will skip the `'fill'` transformation for the moment. When used on its own, it works exactly like the `'static'` transformation. Only when used together with another transformation can it be used to fill in a default value.

Multiple value transformations

Polymatheia actually supports a range of multi-value transformations, but here we will only look at one: `'join'`. Create a new cell and add the following code:

```
filtered = RecordsFilter(reader, ('and', ('exists', ['edmPlaceLatitude']), ('exists',␣
↪['edmPlaceLongitude'])))
mapping = ('join', 'lat_lon', ',', 'edmPlaceLatitude[0]', 'edmPlaceLongitude[0]')
transformed = RecordsTransform(filtered, mapping)
for record in transformed:
    print(record)
```

Here you can see a pattern that you will frequently see in your own code. First, we apply a filter to the data-set, then we transform those records that pass through the filter. Here the filter is simply used to ensure we transform only those records that have co-ordinates.

In the transform, we take the first “edmPlaceLatitude” value and the first “edmPlaceLongitude” value and combine them using a “,”, storing the result in a field called “lat_lon”. We may need a transformation such as this one because we are planning to plot the data on a map and that requires the co-ordinates to be in this format. Run the cell to see how the data is transformed.

Custom transformations

The final transformation we will look at, very briefly, is the `'custom'` transformation. The `'custom'` transformation allows us to apply our own transformation function to each record. The function is given the record and it is expected that it returns a single value, which is stored as the result. Create a new cell and add the following code:

```
mapping = ('custom', 'title_tokens', lambda record: len(record.title[0].split()))
transformed = RecordsTransform(reader, mapping)
for record in transformed:
    print(record)
```

If you run it, you will see that this outputs the number of white-space split tokens in the title. The important bit of code here is this:

```
lambda record: len(record.title[0].split())
```

The `lambda` defines what is known as a “lambda function”, which is basically a very simple function that can be defined in-line, as is the case here. After the `lambda` keyword, the next element(s) are the parameters to the lambda function. Because the `'custom'` transform calls the lambda function with a single parameter, we only define a single parameter here (`record`). The “:” indicates that what follows is the function’s body code, which is run every time the function is called. In this case we first get the first title value `record.title[0]` and then `split()` that. By default the `split()` function splits on white-space, which is exactly what we want here. The `split()` returns a list of split tokens, so we then pass that to the `len()` function, which counts the number of tokens and returns that as a value.

Run the code and you will see that the result contains the number of white-space-split tokens for each title.

The function passed to the `'custom'` transform does not have to be a lambda function, it can also be a full function, but looking at writing those is outside the scope for this tutorial.

2.2.2 Complex Transformations

The basic transformations are useful, but to actually construct a more complete transformed record, we generally need to apply multiple transformations to the same record, and then possibly further transform the result.

Parallel transformation

First we will look at the `'parallel'` transformation, which allows us to apply multiple transformations to the same record and combine their outputs into a single resulting record. Create a new cell and add the following code:

```
filtered = RecordsFilter(reader, ('and', ('exists', ['edmPlaceLatitude']), ('exists',␣
↪['edmPlaceLongitude'])))
mapping = ('parallel', ('copy', 'id', 'id'),
                       ('copy', 'lang', 'dcLanguage[0]'),
                       ('join', 'lat_lon', ',', 'edmPlaceLatitude[0]',
↪'edmPlaceLongitude[0]'),
                       ('custom', 'title_tokens', lambda record: len(record.title[0].
↪split())))
transformed = RecordsTransform(filtered, mapping)
for record in transformed:
    print(record)
```

As you can see, the `'parallel'` transformation contains four nested basic transformations, two copy, one join, and a custom transformation. As you can see, at this point the number of nested round brackets is quite large and if you wanted to add something, it can be tricky to see where to do so. However, the notebook provides some help. If you place the cursor just behind a bracket, then it will highlight both the opening and closing bracket.

If you run the cell, you will see that each result record now has four values.

Sequential transformation

If you go back to the output of the basic `('copy', 'lang', 'dcLanguage[0]')` transform, you will see that some of the values are `null`. We briefly touched on the `'fill'` transformation as a way of filling those empty values, but that needs to be run in sequence after the initial `'copy'` transformation. Create a new cell and add the following code:

```
filtered = RecordsFilter(reader, ('and', ('exists', ['edmPlaceLatitude']), ('exists',␣
↪['edmPlaceLongitude'])))
mapping = ('sequence', ('copy', 'lang', 'dcLanguage[0]'),
                       ('fill', 'lang', 'NA'))
transformed = RecordsTransform(filtered, mapping)
for record in transformed:
    print(record)
```

You can see that the `'sequence'` filter is defined like the `'parallel'` filters. The difference is that the output of the first transformation is passed as the input to the second transformation (and so on, if you have more transformations in sequence). Here we use the value `'NA'` as the default value to use where there is no `'lang'` value. If you run the cell, you will see that it now contains a mix of language values and `'NA'` values.

Complex transformations

In practice most transformations will combine `'parallel'` and `'sequence'` transformations. We can combine the last two examples into a complete example:

```
filtered = RecordsFilter(reader, ('and', ('exists', ['edmPlaceLatitude']), ('exists',␣
↪['edmPlaceLongitude'])))
mapping = ('parallel', ('copy', 'id', 'id'),
                       ('sequence', ('copy', 'lang', 'dcLanguage[0]'),
                                    ('fill', 'lang', 'NA')),
                       ('join', 'lat_lon', ',', 'edmPlaceLatitude[0]',
↪'edmPlaceLongitude[0]'),
                       ('custom', 'title_tokens', lambda record: len(record.title[0].
↪split())))
transformed = RecordsTransform(filtered, mapping)
for record in transformed:
    print(record)
```

Here you can see that the `'sequence'` transformation is nested inside the `'parallel'` transformation. If you run the cell, you will see that each record now contains four values and that all the `null` values have been replaced with `'NA'`.

CHAPTER

THREE

ANALYSIS & VISUALISATION

The previous blocks have covered loading and then filtering/transforming the data in preparation for analysis and visualisation. While the tutorial up to this point has been relatively linear, this is not the case for the analysis and visualisation sections. You are free to do them in any order you wish, although the recommendation in general is to analyse first and then visualise what the analysis shows to be interesting. At the same time, using a bit of visualisation can be helpful in order to determine where to apply the statistical analysis.

3.1 Basic Statistical Analysis

The number of potential analysis methods that can be applied to your data is vast and exceeds what can be covered here, we will thus focus on basic descriptive and comparative statistics. Even in these two areas there is more than can be covered here, so this should really only be treated as a first analysis step.

For the actual analysis we will use Pandas to store the data and SciPy for the statistical analysis. This is not something that is that important at this point, but both libraries provide a lot more functionality that can potentially be applied to your data, so you might want to explore their documentation.

3.1.1 Descriptive Statistics

When analysing a data-set the first step should always be to generate some basic statistics that describe the data-set you are working with. The first step towards doing that is to make the data available to Pandas for the basic statistics. Create a new cell with the following content:

```
from collections import Counter
from polymatheia.data.reader import LocalReader
from polymatheia.data.writer import PandasDFWriter
from polymatheia.filter import RecordsFilter
from polymatheia.transform import RecordsTransform

reader = LocalReader('europeana_test')
```

Two additions here that we have not used previously are the `Counter` and the `PandasDFWriter`. The `PandasDFWriter` is used first to transform the source metadata records into the Pandas `DataFrame` structure needed for all further analysis. The `Counter` will be used later to determine frequencies.

First add the following code into a new cell and run it:

```
mapping = ('parallel', ('copy', 'id', 'id'),
                       ('copy', 'lang', 'dcLanguage[0]'),
                       ('custom', 'title_tokens', lambda record: len(record.title[0].
↪split())),
```

(continues on next page)

(continued from previous page)

```
                              ('copy', 'completeness', 'europeanaCompleteness'),
                              ('copy', 'type', 'type'))
transformed = RecordsTransform(reader, mapping)
df = PandasDFWriter().write(transformed)
```

The addition to what we have covered previously is line #6, where we create a new `PandasDFWriter` and then use its `write` method to convert the `transformed` records into a Pandas `DataFrame`, which we store in the `df` variable.

This will not have output anything, unless there is an error somewhere. Add and run the following code in a new cell:

```
df
```

The result you will see is an overview over the `DataFrame` we created. You can see that each field created in the transformed data has been converted into a column. If you look at the bottom of the output, you will see that it is 1009 rows and 5 columns, which is exactly the same as the input data.

Categorical data

We will first generate some summary information about categorical data, looking at the "lang" column. Add a new cell with the following content:

```
Counter(df['lang']).most_common()
```

When you run it, you will see that it prints out a long list of all "lang" values, together with how often they occur. Looking at the data, there are some interesting things you can see immediately. First of all, while we have previously treated "de" and "ger" as German-language codes, there is also "deu" and "Deutsch". Second, the second-most-frequent category is "mul", which stands for "multiple", indicating that there are multiple languages, but not specifying which ones they are.

Looking at the data, we can see a few more filters that we could apply. We probably want to filter those where the language is set to multiple and those that have no language at all. We would probably also want to combine the various language values that all refer to the same language. Finally we would like to filter out those languages that have less than 10 occurrences. However, before we modify the data any more, it is worth looking at the other columns as well, before we make any decisions.

Numeric data

Our dataframe has two numeric columns. The number of "title_tokens" and the "completeness"" score. Let us look at the "completeness" first. To generate summary statistics for a numeric column, add a new cell with the following code:

```
df['completeness'].describe()
```

The first row shows the number of values (which might be less than the number of rows, if some rows are empty).

The second and third row show the mean and standard deviation (std). The mean is what is commonly known as the average. It is calculated by summing up all values and dividing by the number of values. The standard deviation is a measure that indicates how much the actual values differ from the mean value. A higher standard deviation indicates that many values are further away from the mean, while a lower value indicates that the values cluster closely to the mean. The standard deviation is essentially the average of the difference between the mean and each value.

The next five rows are the most common "percentiles". The percentiles are calculated by sorting the values lowest to highest. The "min" value (also the 0th percentile) is the first value in the ordered list. Likewise the "max" is the last value in the ordered list. The other three are the value at specific points in that ordered list. The 25th percentile is the value $\frac{1}{4}$ of the way through the list, the 50th percentile half way, and the 75th percentile $\frac{3}{4}$ of the way. The 50th percentile is also called the "median" and the difference between the 75th and 25th percentiles is the so-called "inter-quartile range". Median and inter-quartile range have the same role as mean and standard deviation.

The question is do we use mean/standard deviation or median/inter-quartile range to interpret the data. The fundamental difference between the two is that the mean is much more sensitive to variation in the data. For example if the values we were looking at were `[1, 1, 1, 1, 1, 100]`, then the mean is 17.5 (std 36.9), while the median is 1 (iqr 0). If you know that you don't have any extreme outliers and fractional values in the result make sense, then the mean is the way to go. If neither of these are true, then the median is better.

For an example with outliers, we can look at the lengths of the titles. Add and run a new cell with the following code:

```
df['title_tokens'].describe()
```

If you look at the output, you will see that the mean is about 2 words longer than the median. You will also see that the maximum title length is 249. Clearly our mean is being skewed and we need to filter out some outliers. The question is where to we draw the line? One way is to look at the 95th percentile. Update the code cell to look like this:

```
df['title_tokens'].describe(percentiles=[0.25, 0.5, 0.75, 0.95])
```

You can now see that 95% of all titles have 29 or less words. If we filter out anything with more than 29 words, we will loose 5% of the data, but at the same time any analysis is less influenced by the outliers.

3.1.2 Further Filtering

In the basic statistical analysis we identified a number of additional filters and transforms we wish to apply to our data-set:

- Remove all rows with multiple languages or no language
- Merge language codes
- Remove all titles with 30 or more tokens
- Remove languages that have less than 10 entries

Start by adding a new cell importing everything we will need:

```
from collections import Counter
from pandas import DataFrame
from polymatheia.data.reader import LocalReader
from polymatheia.data.writer import PandasDFWriter
from polymatheia.filter import RecordsFilter
from polymatheia.transform import RecordsTransform
from scipy import stats
```

Then add and run a cell with the following code:

```
reader = LocalReader('europeana_test')
mapping = ('parallel', ('copy', 'id', 'id'),
                       ('copy', 'lang', 'dcLanguage[0]'),
                       ('custom', 'title_tokens', lambda record: len(record.title[0].
↪split())),
                       ('copy', 'completeness', 'europeanaCompleteness'),
                       ('copy', 'type', 'type'))
transformed = RecordsTransform(reader, mapping)
fltr = ('and', ('exists', ['lang']),
```

(continues on next page)

(continued from previous page)

```
               ('neq', ['lang'], 'mul'),
               ('lt', ['title_tokens'], 30))
filtered = RecordsFilter(transformed, fltr)
df = PandasDFWriter().write(filtered)
```

As you can see, we first re-use the transformation we previously developed (lines #2-#7). Next, we pass the transformation into a filter that requires that each record has a language, the language is not "multiple languages", and the number of title tokens is less than 30. The transformed and filtered data is then stored in a `DataFrame` via the `PandasDFWriter`. Let us see what effect this has by adding and running another cell with the following code:

```
df
```

You will see that this filters the data-set down to 767 rows (which is about 75% of the original data-set). The next step is to merge the languages. To do that, we need a full list of languages used. Add and run a new cell with the following code:

```
Counter(df['lang']).most_common()
```

We will use the two-letter codes, meaning we need to find mappings for "ger", "und", "deu", "fre", "Deutsch", "pol", "cat", "swe", "lat", "hun", "zxx", and "ita". Of these "zxx" actually means that the record has no linguistic content, so we don't need to map that, but we do need to filter it. Similarly "und" means "undefined", which we should also get rid of. Update the code in the read/transform/filter cell to look like this:

```
def map_language(record):
    if record.lang == 'ger' or record.lang == 'deu' or record.lang == 'Deutsch':
        return 'de'
    elif record.lang == 'hun':
        return 'hu'
    elif record.lang == 'swe':
        return 'sv'
    elif record.lang == 'pol':
        return 'pl'
    elif record.lang == 'fre':
        return 'fr'
    elif record.lang == 'cat':
        return 'ca'
    elif record.lang == 'lat':
        return 'la'
    elif record.lang == 'ita':
        return 'it'
    return record.lang

reader = LocalReader('europeana_test')
mapping = ('parallel', ('copy', 'id', 'id'),
                       ('sequence', ('copy', 'lang', 'dcLanguage[0]'),
                                    ('custom', 'lang', map_language)),
                       ('custom', 'title_tokens', lambda record: len(record.title[0].
↪split())),
                       ('copy', 'completeness', 'europeanaCompleteness'),
                       ('copy', 'type', 'type'))
transformed = RecordsTransform(reader, mapping)
fltr = ('and', ('exists', ['lang']),
               ('neq', ['lang'], 'mul'),
               ('neq', ['lang'], 'zxx'),
               ('neq', ['lang'], 'und'),
               ('lt', ['title_tokens'], 30))
```

(continues on next page)

(continued from previous page)

```
33 filtered = RecordsFilter(transformed, fltr)
34 df = PandasDFWriter().write(filtered)
```

The big change is the new function we have defined in lines #1-#18. When we first looked at custom transforms, we used lambda functions, but it is also possible to use a full function in a custom transform. Just as with the lambda function, the full function takes a single parameter, which is the record to transform. Inside our function we have a series of `if` statements. An `if` statement is a control structure that tells the computer that if a given condition is `True`, then run the code that is in the `if` body (in Python indicated through indentation). If the condition is not `True`, skip the body. The `elif` is an extension of that which you should read as “if the previous `if` condition was not `True` and this `if` statement’s condition is `True`, then run the nested block”. If the language does not match any of the specific language codes we check, then we simply return the existing language value.

We use our `map_language` function in the `mapping`, running the language `'copy'` and then our `'custom'` transform in sequence. Additionally in the `fltr` we have filtered out the “zxx” and “und” language codes.

Run the cell and then run the `df` cell as well. You will see that now our dataframe has 754 rows, indicating that the additional unneeded language codes have been filtered out. However, we will still have some language codes that occur only very infrequently.

To filter those out, first run the `Counter(df['lang']).most_common()` cell again and look at the output. The languages “sv”, “la”, “da”, “ca”, “nl”, “es”, “it”, “en”, and “et” all have less than 10 occurences, so should be filtered. Update the load/transform/filter cell to look like this:

```
def map_language(record):
    if record.lang == 'ger' or record.lang == 'deu' or record.lang == 'Deutsch':
        return 'de'
    elif record.lang == 'hun':
        return 'hu'
    elif record.lang == 'swe':
        return 'sv'
    elif record.lang == 'pol':
        return 'pl'
    elif record.lang == 'fre':
        return 'fr'
    elif record.lang == 'cat':
        return 'ca'
    elif record.lang == 'lat':
        return 'la'
    elif record.lang == 'ita':
        return 'it'
    return record.lang

reader = LocalReader('europeana_test')
mapping = ('parallel', ('copy', 'id', 'id'),
                       ('sequence', ('copy', 'lang', 'dcLanguage[0]'), ('custom',
↪'lang', map_language)),
                       ('custom', 'title_tokens', lambda record: len(record.title[0].
↪split())),
                       ('copy', 'completeness', 'europeanaCompleteness'),
                       ('copy', 'type', 'type'))
transformed = RecordsTransform(reader, mapping)
fltr = ('and', ('exists', ['lang']),
               ('neq', ['lang'], 'mul'),
               ('neq', ['lang'], 'zxx'),
               ('neq', ['lang'], 'und'),
               ('neq', ['lang'], 'la'),
               ('neq', ['lang'], 'sv'),
```

(continues on next page)

(continued from previous page)

```
                ('neq', ['lang'], 'es'),
                ('neq', ['lang'], 'da'),
                ('neq', ['lang'], 'nl'),
                ('neq', ['lang'], 'ca'),
                ('neq', ['lang'], 'it'),
                ('neq', ['lang'], 'et'),
                ('neq', ['lang'], 'en'),
                ('lt', ['title_tokens'], 30))
filtered = RecordsFilter(transformed, fltr)
df = PandasDFWriter().write(filtered)
```

If you run the cell again and also re-run the `df` cell, then you will see that we have now reduced the size of our analysis data-set to 721 rows (about 71% of the original data-set). We can also see the effect this filtering has had on the number of tokens in the tiles by adding a new cell with the following code:

```
df['title_tokens'].describe()
```

As you can see in the output, the mean length has reduced by 2 and the median by 1, bringing them much closer together, indicating that the data-set is now more cohesive. We can now move on to applying some comparative statistics to our cleaned data-set.

Important: One of the advantages of Jupyter Notebooks is that you can trace the steps of your analysis. However, you should also make notes of the reasoning for the various changes, as it is important to be able to trace your analysis, otherwise it is hard to put any trust in the results (as the filtering will affect and may bias results).

3.1.3 Comparative Statistics

Now that we have a filtered-down and cleaned data-set, we can look at doing some comparative statistical analysis on the data-set. We will look at two tests in this section: Mann-Whitney-U and χ^2 (chi-square).

Before we look at the tests in detail, we need to have a quick look at some basic concepts in this area of statistics. The fundamental assumption in all these statistical tests is that you do **not** have access to a complete view of the world, instead you are relying on a sample of the data. This aligns nicely with any archival work, because archiving is always a sampling process. At the same time there is an important caveat here. The statistical tests assume that there is no explicit or implicit bias in the archival sampling process. That means that statistical tests may report that there are significant effects, when in reality those are caused by an explicit or implicit bias in how the digital archive was created. It is thus important when reporting statistical results, to check for and discuss the effects of any potential biases.

Here are some of the most important concepts in this area of statistics:

- *population*: the full set of all values, which you generally do not have access to.
- *sample*: a sub-set of the full set of values in the *population*. The *sample* should be selected from the *population* using an unbiased method, based upon one or more criteria.
- *independent samples*: two or more *samples* that have been acquired in such a way that acquiring the values in one did not influence the values in the other(s). Two values calculated using the same formula from two separate texts are *independent*. Two values calculated using different formulas from the same text are *dependent*.
- *null-hypothesis*: the default assumption when comparing the *samples*. Generally the *null-hypothesis* is that the *samples* have been taken from the same population and do not show any differences.
- *alternative hypothesis*: our hypothesis as to what the difference between the *samples* will be. The *alternative hypothesis* is what we actually believe the relationship between the *samples* to be. For example we may believe

that the values of the first sample are smaller than the values of the second sample and that is the *alternative hypoethisis*, while the *null hypoethesis* is that there is no difference.

- *statistical test*: a mathematical model that calculates how likely it is that the *null-hypothesis* is wrong and that the *alternative hypoethesis* is correct.
- *p-value*: the probability that any differences between the *samples* are due to bad luck in the sampling process and are not actually indicative of any real difference between the two samples. For example a *p-value* of 0.05 means that there is a 1 in 20 chance of seeing the observed *samples* just by chance and not due to any actual differences between the samples.
- *statistical significance*: is used to indicate when a *p-value* implies that there is an actual difference between the *samples*, meaning that there are two distinct groups within the *population*. Generally for *statistical significance* the *p-value* is expected to be smaller than 0.05 (1 in 20), 0.01 (1 in 100), or 0.001 (1 in 1000). Which boundary you choose depends upon how definite you require your result to be. For example a *p-value* of 0.05 means that if you sampled 20 times, you would see one set of samples where there is *statistical significance* even though there is no actual difference in the underlying *populations*. The less you know about how your samples were generated, the lower you should set your boundary.

Mann-Whitney-U

The Mann-Whitney-U test is used to compare two independent samples of ordinal values (ordinal meaning that for any two values you can determine which one is greater than the other or equal to each other). We will use this to investigate whether the language of the record influences the length of title. First, let us quickly remind ourselves of which languages are still in our dataset by running the following code in a new cell:

```
Counter(df['lang'])
```

You will see that there are four languages: German (680 records), Hungarian (19 records), Polish (12 records), and French (10 records). To apply the test, we need to split the data for each language out of the DataFrame. Add a new cell with the following code:

```
de_title_lengths = df[df['lang'] == 'de']['title_tokens']
hu_title_lengths = df[df['lang'] == 'hu']['title_tokens']
pl_title_lengths = df[df['lang'] == 'pl']['title_tokens']
fr_title_lengths = df[df['lang'] == 'fr']['title_tokens']
```

You can see that the four expressions are all the same, except for the varying language codes. They all create Series of "title_tokens", but each filtered to include only one of the four languages. We will now look at the first expression in detail:

- First the following part of the expression creates a filter mask that defines which records we want to have included in our sub-set:

  ```
  df['lang'] == 'de'
  ```

  (You can actually run this in a separate cell and you will see that you get a long list of `True` and `False` values).
- Next, we apply the filter mask to our DataFrame:

  ```
  df[df['lang'] == 'de']
  ```

  This will create a new DataFrame, which only includes those rows where the filter mask is `True`, but still includes all columns.
- Finally, we select only the "title_tokens" column from our filtered DataFrame:

```
df[df['lang] == 'de']['title_tokens']
```

Now that we have our Series of “title_tokens” values for the four languages, we can run the Mann-Whitney-U test. Run the following code in a new cell:

```
stats.mannwhitneyu(de_title_lengths, hu_title_lengths, alternative='two-sided')
```

The `stats.mannwhitneyu` function takes three parameters. The first two are the samples that we wish to compare. Here we are comparing the lengths of the German and Hungarian titles. The third specifies the *alternative hypothesis* that we want the test to consider. This can be either `'two-sided'` meaning that we believe the two samples to have different values, but we don’t know whether larger or smaller, `'less'` meaning that we believe the values from the first sample to be smaller than the values from the second, or `'greater'` meaning that we believe the values from the first sample to be greater than the values from the second. Initially it is often good to use `'two-sided'`, which makes less assumptions about the data.

The result of running the test should look like this:

```
MannwhitneyuResult(statistic=4545.0, pvalue=0.026990873270466658)
```

The `statistic` is a mathematical value that the test calculates based on the input samples. The `pvalue` indicates how likely it is that the *null-hypothesis* is correct and that there is no significant difference between the two samples. Here we have a value of 0.027, indicating that the difference is statistically significant at $p < 0.05$. So it is likely that there is some kind of effect here that we can investigate further.

The next step is to look at the median values for the two samples. Run the following code in a new cell:

```
print(de_title_lengths.median(), hu_title_lengths.median())
```

You will see that the German titles have a median length of 8 and the Hungarian ones a median length of 13. We can thus adapt our *alternative hypothesis* to indicate that we believe the German titles to be shorter. Run the following code in a new cell:

```
stats.mannwhitneyu(de_title_lengths, hu_title_lengths, alternative='less')
```

which produces the following result:

```
MannwhitneyuResult(statistic=4545.0, pvalue=0.013495436635233329)
```

The *p-value* here is lower than in the previous test, indicating that the it is likely that the German titles are shorter than the Hungarian ones.

We would now need to apply the pair-wise test to all combinations of two languages. For example to see a pair with very significant differences run the following:

```
stats.mannwhitneyu(de_title_lengths, fr_title_lengths, alternative='less')
```

This will result in:

```
MannwhitneyuResult(statistic=1177.5, pvalue=0.00018441519206957308)
```

Indicating a highly significant difference between the two (again German being shorter than French).

When reporting results it is important to always report the sample size, median values of the two samples, the test `statistic` and the `pvalue`. In this example, while we are seeing statistically significant differences, the number of observations in the non-German samples are much smaller than for German, which has to be taken into consideration when reporting and discussing the results.

Chi-square

The second test we will look at is the χ^2 (chi-square) test. This test is used when we want to compare two or more samples of categorical data. As it is impossible to order categorical data, we cannot apply Mann-Whitney-U. Instead we will count how often the different values appear and place those values into what is known as a contingency table. We will use this to test whether the language of the record influences the type of record. To do that, we first need to count how often each type of record appears in each language. Run the following code in a new cell:

```
1 for lang in ['de', 'hu', 'pl', 'fr']:
2     print(lang)
3     print(Counter(df[df['lang'] == lang]['type']).most_common())
```

You are already familiar with most of the concepts here. In line #1 we loop through a list with the language codes. In each iteration the `lang` variable will hold one language code, which we print out in line #2. In line #3 we first create a filter mask comparing the "lang" column to our `lang` variable, then we apply that filter to the DataFrame, select the "type" column, pass that into a `Counter` and print out the `most_common()` values. The result will look like this:

```
de
[('TEXT', 663), ('IMAGE', 12), ('SOUND', 5)]
hu
[('TEXT', 19)]
pl
[('TEXT', 12)]
fr
[('TEXT', 10)]
```

Based on these we can now create a new DataFrame to hold our contingency table. Run the following code in a new cell:

```
df2 = DataFrame([{'text': 663, 'image': 12, 'sound': 5},
                 {'text': 19, 'image': 0, 'sound': 0},
                 {'text': 12, 'image': 0, 'sound': 0},
                 {'text': 10, 'image': 0, 'sound': 0}])
```

When creating a new `DataFrame`, we pass a list to the `DataFrame`. Each element in the list represents one row in the resulting `DataFrame`. Inside each element, we specify a dictionary, mapping keys to frequencies. It is important to explicitly provide the `0` values, as otherwise those would be seen as missing values by the `DataFrame` and the χ^2 test does not work with missing values. If you run the following code in a new cell, you will see our contingency table:

```
df2
```

This kind of table is known as a "contingency table", because it tests whether the column categories (here the record type) are contingent (meaning they depend on) with the row categories (the record language). To apply the χ^2 test, run the following code in a new cell:

```
stats.chi2_contingency(df2)
```

The result will look something like this:

```
(1.049751420454546,
 0.983655785107175,
 6,
 array([[6.63966713e+02, 1.13176144e+01, 4.71567268e+00],
        [1.85520111e+01, 3.16227462e-01, 1.31761442e-01],
        [1.17170596e+01, 1.99722607e-01, 8.32177531e-02],
        [9.76421637e+00, 1.66435506e-01, 6.93481276e-02]]))
```

The first value is the χ^2 statistic, the second value the *p-value*, the third the degrees of freedom, and the fourth a representation of the expected frequencies that the test used to determine whether there was any significant differences. A *p-value* of 0.98 indicates that there is no significant link between the type and language of the record.

To see how the χ^2 behaves with different values, try changing the values and re-running the cell where we defined our new `DataFrame`. Then re-run the test. See how much you need to change the values to make the differences statistically significant.

The nice thing about the χ^2 test is that you can feed in multiple values in both the rows and columns. However, the resulting *p-value* will only tell you whether there is any significant difference somewhere in the contingency table, not where it is. To determine that, you then need to create multiple contingency tables for each pair of two rows and run the χ^2 test on each one. That will tell you where significant differences exist.

3.2 Visualisation

Visualisations are a useful method for making information and results more accessible. However, it is important to note that visualisations are also **always** reductions, simplifications, and distortions of the underlying data and should only be used as illustration, not as proof of anything.

To get started, add the following code into the notebook and run it:

```
import seaborn
from polymatheia.data.reader import LocalReader
from polymatheia.data.writer import PandasDFWriter
from polymatheia.filter import RecordsFilter
from polymatheia.transform import RecordsTransform

def map_language(record):
    if record.lang == 'ger' or record.lang == 'deu' or record.lang == 'Deutsch':
        return 'de'
    elif record.lang == 'hun':
        return 'hu'
    elif record.lang == 'swe':
        return 'sv'
    elif record.lang == 'pol':
        return 'pl'
    elif record.lang == 'fre':
        return 'fr'
    elif record.lang == 'cat':
        return 'ca'
    elif record.lang == 'lat':
        return 'la'
    elif record.lang == 'ita':
        return 'it'
    return record.lang

reader = LocalReader('europeana_test')
mapping = ('parallel', ('copy', 'id', 'id'),
                       ('sequence', ('copy', 'lang', 'dcLanguage[0]'), ('custom',
↪'lang', map_language)),
                       ('custom', 'title_tokens', lambda record: len(record.title[0].
↪split())),
                       ('copy', 'completeness', 'europeanaCompleteness'),
                       ('copy', 'type', 'type'))
transformed = RecordsTransform(reader, mapping)
fltr = ('and', ('exists', ['lang']),
```

(continues on next page)

(continued from previous page)

```
                ('neq', ['lang'], 'mul'),
                ('neq', ['lang'], 'zxx'),
                ('neq', ['lang'], 'und'),
                ('neq', ['lang'], 'la'),
                ('neq', ['lang'], 'sv'),
                ('neq', ['lang'], 'es'),
                ('neq', ['lang'], 'da'),
                ('neq', ['lang'], 'nl'),
                ('neq', ['lang'], 'ca'),
                ('neq', ['lang'], 'it'),
                ('neq', ['lang'], 'et'),
                ('neq', ['lang'], 'en'),
                ('lt', ['title_tokens'], 30))
filtered = RecordsFilter(transformed, fltr)
df = PandasDFWriter().write(filtered)
```

With our data now in place, we will look at visualising *Numerical Data* and *Categorical Data*.

3.2.1 Numerical Data

Distributionplots

We will start with numerical data. For numerical columns, we can use the `distplot` to plot the distribution of the values in the "title_tokens". Run the following code in a new cell:

```
seaborn.distplot(df['title_tokens'], bins=range(0, max(df['title_tokens']), 1))
```

The `distplot` merges two plots into a single one. The bars are a histogram plot and the line is a density plot. A histogram plot is created by splitting the range of values into a number of `bins` and then assigning each value to the appropriate bin. The size of each bar then represents the fraction of values that are in that bin. By default the number of bins is estimated from the data, but we can also specify what bins we want:

```
seaborn.distplot(df['title_tokens'], bins=range(0, max(df['title_tokens']), 1))
```

Here we use the `range` function to create a list of values from 0 to the maximum value in "title_tokens", with a step size of 1 (`[0, 1, 2, 3, ... n]`). Try running the code to see how the plot compares. Then in a new cell, run the same code, but with a step size of 2 and compare the resulting diagrams. You can see that by increasing the step size, we are essentially smoothing the individual values.

The second part of the diagram is the line which represents a density estimate. Essentially it shows the same as the histogram, but smoothened out even more. The density estimate is calculated as a weighted sum of the values within a certain distance from the current value. That means that values that are further away from the current value are down-weighted, while closer values are given a higher weight. This is also the reason that the curve continues below 0 (even though that makes no sense), as there are still values within the density function's range whose influence slowly reduces until the density estimate hits 0.

Looking at this kind of diagram is a good way to develop an understanding of how the data looks, but any kind of interpretation is exactly that. To draw conclusions you should go back to the statistical analysis.

Scatterplots

Sometimes you will want to see how two variables are distributed at the same time. For this we use the `scatterplot`. Run the following code in a new cell:

```
seaborn.scatterplot(x='title_tokens', y='completeness', data=df)
```

This code-example also shows how most plots are generated in Seaborn. You specify the DataFrame to visualise via the `data` parameter and then the columns to plot along the `x` and `y` axes using the respective parameters to specify the column names.

One of the limitations of the scatterplot is that where multiple records have the same pair of values, a single point is shown, giving no indication of how many values might be there. To get a bit of an insight into this, we can add histograms on the outside of the plot with the following code:

```
seaborn.jointplot(x='title_tokens', y='completeness', data=df)
```

By taking into account both plots at the same time, we can see where there are more or less values. We can also visualise this another way, using a two-dimensional density estimate plot. Run the following in a new cell:

```
seaborn.kdeplot(df['title_tokens'], df['completeness'], shade=True)
```

In the density plot, the more values there are at a certain point, the darker the shade of the area. The plot shows a peak around 8 “completeness” and 10 “title_tokens”, but it also shows that there is a separate area of interest with 0 “completeness” and between 3 and 8 “title_tokens”.

Just as with the scatterplot, we can add the density estimates for the individual values on the outside with the following code in a new cell:

```
seaborn.jointplot(x='title_tokens', y='completeness', data=df, kind='kde')
```

This kind of plot also shows that second peak in the “completeness” very cleanly.

Boxplots

A different way of visualising the distribution of values is the `boxplot`. Run the following code in a new cell:

```
seaborn.boxplot(y='title_tokens', data=df)
```

In the `boxplot` the thick horizontal line in the middle is the median value, while the top and bottom ends of the main box are the 25 and 75 percentiles. The whiskers indicate either the maximum or minimum value. The exception to this is if there are values that are more than 1.5 times the inter-quartile-range (IQR, the difference between the 75 and 25 percentiles) higher or lower than the 25 and 75 percentiles. In this case, the whisker is drawn at the value of the 75 percentile plus 1.5 times the IQR and any values greater than that are marked out as dots, indicating outliers. The same is done at the lower end.

One thing we can do with the `boxplot` is to use it to visualise differences in the distributions for a second categorical variable. For example to see the distribution of the “title_tokens” split by language, run the following in a new cell:

```
seaborn.boxplot(y='title_tokens', x='lang', data=df)
```

Violinplots

One thing the `boxplot` does not show in detail is how the values are distributed. To also see this we use the `violinplot`. Run the following in a new cell:

```
seaborn.violinplot(y='title_tokens', x='lang', data=df)
```

The `violinplot` basically combine a boxplot (the white dot is the median, the thick bar indicates the 25 and 75 percentiles, the line the min/max, and dots for outliers) with a density estimate.

3.2.2 Categorical Data

In addition to numeric data it is also possible to visualise categorical data. This works similarly to the χ^2 test in that we will visualise counts of values, rather than the actual values. Run the following code in a new cell:

```
seaborn.countplot(x='type', data=df)
```

As you can see in the resulting plot, the type of record is primarily "TEXT", with a few "IMAGE" and "SOUND" records.

We can also visualise a categorical numeric value, such as the "completeness", by running the following code in a new cell:

```
seaborn.countplot(x='completeness', data=df)
```

As we have already seen in other plots, the most common "completeness" value is 8.

Finally, it is also possible to further split each categorical value via another categorical column. Run the following code in a new cell:

```
seaborn.countplot(x='completeness', hue='type', data=df)
```

While this visualisation is not that easy to read, we can see that the "IMAGE" records tend to have more values around 6 and 0. This also shows one of the weaknesses of any kind of visualisation. If your data is very skewed in a value that you want to visualise (as is the "type"), then differences in values can often be hard to see.

CHAPTER

FOUR

NEXT STEPS

You have reached the end of the tutorial. This section will briefly cover installing and running the various libraries outside of the tutorial, so that you can use what you have learned here in a full research setting.

4.1 Running via Docker

The complete tutorial environment is available as a Docker container. The Docker Engine is available for Windows, Mac, and Linux.

4.2 Running fully locally

To run the various libraries locally, it is recommended to install everything into a virtual environment, which separates all the Python libraries from anything else you have installed on your machine. If you have no preferences regarding how to manage your virtual environments, then Poetry is the recommended way to create and manage your virtual environments.

After installing Poetry, follow these steps to install all required dependencies:

```
poetry add polymatheia
poetry add scipy
poetry add seaborn
```

After that you can run `python` inside the virtual environment and all the required libraries will be available.

4.2.1 Running inside a local Jupyter Notebook Server

If you want to run a local Jupyter Notebook Server first run this command:

```
poetry add jupyter
```

Then run the following to start the server:

```
jupyter notebook
```

This will automatically load the Single-user Notebook page in your browser.