Protocol: Investigating DOIs classes of errors V.4

Cristian Santini[1], Arcangelo Massari[1], Ricarda Boente[1], Deniz Tural[1]
[1]University of Bologna

Version 4
May 03, 2021

1 Works for me dx.doi.org/10.17504/protocols.io.bunnnvde

Open Science 2020/2021

Arcangelo Massari

ABSTRACT

The purpose of this protocol is to provide an automated process to repair invalid DOIs that have been collected by the OpenCitations Index Of Crossref Open DOI-To-DOI References (COCI) while processing data provided by Crossref.
The data needed for this work is provided by COCI as a CSV containing pairs of valid citing DOIs and invalid cited DOIs. With the goal to determine an automated process, we first classified the errors that characterize the wrong DOIs in the list. The starting hypothesis is that there are two main classes of errors: factual errors, such as wrong characters, and DOIs that are not yet valid at the time of processing. The first class can be furtherly divided into three classes: errors due to irrelevant strings added to the beginning (prefix-type errors) or at the end (suffix-type errors) of the correct DOI, and errors due to unwanted characters in the middle (other-type errors). Once the classes of errors are addressed, we propose automatic processes to obtain correct DOIs from wrong ones. These processes involve the use of the information returned from DOI API, the January 2021 Public Data File from Crossref, as well as rule-based methods, including regular expressions to correct invalid DOIs.
The application of this methodology produced a CSV dataset containing all the pairs of citing and cited DOIs in the original dataset, each one enriched by 5 fields: "Already_Valid", which tells if the cited DOI was already valid before cleaning, "New_DOI", which contain a clean, valid DOI (if our procedure was able to produce one), and "prefix_error", "suffix_error" and "other-type_error" fields, which contain, for each cleaned DOI the number of errors that were cleaned.

DOI

dx.doi.org/10.17504/protocols.io.bunnnvde

PROTOCOL CITATION

Cristian Santini, Arcangelo Massari, Ricarda Boente, Deniz Tural 2021. Protocol: Investigating DOIs classes of errors . **protocols.io**
https://dx.doi.org/10.17504/protocols.io.bunnnvde
Version created by Arcangelo Massari

KEYWORDS

Open Access, I4OC, OpenCitations, citations, DOI, Crossref, COCI, data quality

LICENSE

CREATED

Apr 30, 2021

LAST MODIFIED

May 03, 2021

PROTOCOL INTEGER ID

49582

MATERIALS TEXT

COCI, the OpenCitations Index of Crossref open DOI-to-DOI citations (https://w3id.org/oc/index/coci), is a collection of open citations in which citations are exposed as first-class data entities with accompanying properties. All the data available in COCI are derived from those accessible through Crossref (https://www.crossref.org/). Since Crossref does not double-check their data, many of the DOIs that they publicly provide as citations data are not valid. The data upon which our project is build is provided by COCI as result of a procedure of collection of errors and mistakes of DOIs available on Crossref and is publicly available at http://doi.org/10.5281/zenodo.4625300.

SAFETY WARNINGS

The procedure includes some optional steps (1.1, 1.2, 3.1), which aims to dramatically reduce execution times by sacrificing RAM. It is recommended to run them only having at least 64GB of RAM.

DISCLAIMER:

This research has been realized during the University of Bologna course in Open Science held by Professor Silvio Peroni in the y.a. 2020-2021.

BEFORE STARTING

This methodology takes for granted some basic knowledge regarding the scholarly publishing domain, the nature of identifiers (e.g. DOI), the open citation movement and Python (variable, classes, methods and libraries).

Before starting, you need to make sure you have Python3.x installed on your computer, in addition, in order to correctly execute the Python-based scripts indicated in the methodology, you must install the required libraries defined in requirements.txt. Please follow the official Python guidelines at https://wiki.python.org/moin/BeginnersGuide/ to check and eventually install python and the required libraries locally on your machine.

In the rest of this protocol we will use some notation to describe the data and the processes involved in our protocol:
<var>: variables are mentioned inside angle brackets and are written in bold.
function() is a block of code to perform a specific operation. They also occur in bold.
class Object(): describe a class, a specific object which includes different functions as its methods. They occur in bold and are preceded by "class" prefix.

DATA IMPORT 10h

1 We import a CSV containing invalid DOI-to-DOI citations. The CSV is provided on public license by Peroni, Silvio. (2021). Citations to invalid DOI-identified entities obtained from processing DOI-to-DOI citations to add in COCI (Version 1.0) [Data set]. *Zenodo*. http://doi.org/10.5281/zenodo.4625300.

Citations to invalid DOIs obtained from Crossref

The CSV is organized as follows: the first column contains valid citing DOIs and the second contains invalid cited DOIs.

Valid_citing_DOI	Invalid_Cited_DOI
10.14778/1920841.1920954	10.5555/646836.708343
10.5406/ethnomusicology.59.2.0202	10.2307/20184517
10.1161/01.cir.63.6.1391	10.1161/circ.37.4.509

Extract of the CSV file containing the citations to invalid DOIs obtained from Crossref.

1.1 In order to make the DOI validity check procedure dramatically more efficient, Crossref's public data file containing over 120 million metadata records can be used. The reason is that the DOIs reported by Crossref at the work level are assigned by Crossref itself and not by publishers and, therefore, they are necessarily correct. The records are provided on public licence by Crossref. (2021). January 2021 Public Data File from Crossref. https://doi.org/10.13003/GU3DQMJVG4. 2h

January 2021 Public Data File from Crossref

According to the statistics reported by Crossref at the indicated address, the compressed file weighs 102.51 GB and the download time at an average speed of 12.73 MB/s is 2 hours.

This sub-step and the following one are completely optional and it is recommended to perform them only having at least 64 GB of RAM available, since over 120 million DOIs must be stored in a set and kept in RAM during the entire process shown at step 3.

1.2 Once downloaded, the file consists of a folder containing 40'228 JSON files. The function **get_all_crossref_dois()** is used to extract the DOI names from each work. 8h

```
import ijson, os, tqdm as tqdm


def get_all_crossref_dois(folder_path:str) -> list:
        json_files = [pos_json for pos_json in os.listdir(folder_path) if
pos_json.endswith('.json')]
        dois = list()
        pbar = tqdm(total=len(json_files))
        for json_file in json_files:
            with open(os.path.join(folder_path, json_file)) as json_file:
                parser = ijson.parse(json_file)
                for prefix, event, value in parser:
                    if (prefix, event) == ('items.item.DOI', 'string'):
                        dois.append({"crossref_doi": value})
            pbar.update(1)
        pbar.close()
        return dois
```

The function cycles through all the JSON files in a folder, opens them, and parses them looking for works' DOIs. For this purpose the ijson library was used, that instead of loading the whole file into memory and parsing everything at once, lazily streams the data only responsing to specific events, similar to what SAX does for XML.

Finally, the list thus created is saved in a CSV file. The CSV file is preferred over JSON for this purpose because, as there is no hierarchy in the data, CSV takes up less space as it uses fewer characters than JSON. The CSV thus obtained consists of a list of all the DOIs of all the works found in Crossref, as shown in the table below:

A
10.1001/.389
10.1001/.391
10.1001/.405
10.1001/.411

Extract of the CSV file containing the DOIs of the works obtained from Crossref.

ERROR ANALYSIS

2 Taking as a reference the DOI error's taxonomy proposed in *Errors in DOI indexing by bibliometric databases* (Franceschini et al. 2015), there are two main classes of errors: author errors, made by authors when creating the list of cited articles for their publication, and database mapping errors, related to a data-entry error. This protocol deals only with the second kind of error, which can be further divided between prefix errors, suffix errors, and other-type errors, as proposed in *DOI errors and possible solutions for Web of Science* (Xu et al., 2019). In order to solve our problem, we manually isolated from a subset of 100 DOIs recurrent strings at the beginning and at the end with corrupted DOI prefix and suffix respectively. In addition, we found other types of errors, like double underscores, double periods, XML tags, spaces, and forward slashes that could be removed at the end of the data cleaning process.

2.1 For cleaning **prefix-type errors**, we took a modified version of the regular expression from (Xu et al., 2019) in order to remove unwanted characters at the beginning of a DOI. The regular expression for matching prefix-type errors is the following:

```
"(.*?)(?:\.)?(?:HTTP:\/\/DX\.D[0|O]I\.[0|O]RG\/|HTTPS:\/\/D[0|O]I\.[0|O]RG\/)(.*)"
```

It is worth noting that in this as in the following regular expressions the character "O" and the number "0" both count as a match, since, as demonstrated in *DOI errors and possible solutions for Web of Science* (Zhu et al., 2018), the two characters are often confused.

2.2 For capturing **suffix-type errors**, we enriched the approach provided by (Xu et al., 2019) and we devised a set of 17 alternative suffix regular expressions. The regular expression are the following:

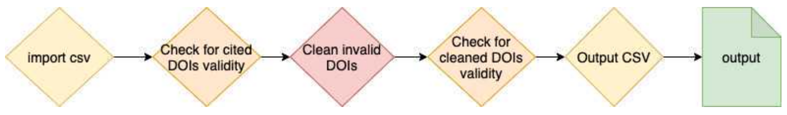
```
1) "(.*?)(?:\/-\/DCSUPPLEMENTAL)$"
2) "(.*?)(?:SUPPINF[0|O](\.)?)$"
3) "(.*?)(?:[\.|\(|,|;]?PMID:\d+.*?)$"
4) "(.*?)(?:[\.|\(|,|;]?PMCID:PMC\d+.*?)$"
5) "(.*?)(?:[\(|\[]EPUBAHEADOFPRINT[\)\]])$"
6) "(.*?)(?:[\.|\(|,|;]?ARTICLEPUBLISHEDONLINE.*?\d{4})$"
7) "(.*?)(?:[\.|\(|,|;]*HTTP:\/\/.*?)$"
8) "(.*?)(?:\/(META|ABSTRACT|FULL|EPDF|PDF|SUMMARY)([>|\)](LAST)?ACCESSED\d+)?)$"
9) "(.*?)(?:[>|)](LAST)?ACCESSED\d+)$"
10) "(.*?)(?:[\.|(|,|;]?[A-Z]*\.?SAGEPUB.*)$?"
11) "(.*?)(?:\.{5}.*?)$"
12) "(.*?)(?:[\.|,|<|&|(|;])$"
13) "(.*?)(?:[\.|(|,|;]10.\d{4}\/.*?)$"
14) "(.*?)(?:\[DOI\].*?)$"
15) "(.*?)(?:\(\d{4}\)?)$"
16) "(.*?)(?:\?.*?=.*?)$"
17) "(.*?)(?:#.*?)$"
```

All the suffix regular expressions can be combined in a single regular expression, by using alternatives within a non-capturing group. This is how the regular expression for suffix-type errors looks like:

```
"(.*?)(?:\/-\/DCSUPPLEMENTAL|SUPPINF[0|O](\.)?|[\.|\(|,|;]?PMID:\d+.*?|[\.|\(|,|;]?
PMCID:PMC\d+.*?|[\(|\[]EPUBAHEADOFPRINT[\)\]]|[\.|\(|,|;]?ARTICLEPUBLISHEDONLINE.*?\d{4}|
[\.|\(|,|;]*HTTP:\/\/.*?|\/(META|ABSTRACT|FULL|EPDF|PDF|SUMMARY)([>|\)](LAST)?
ACCESSED\d+)?|[>|\)](LAST)?ACCESSED\d+|[\.|\(|,|;]?[A-Z]*\.?SAGEPUB.*?|\.{5}.*?|[\.|,|<|&|\(|;]+|
[\.|\(|,|;]10.\d{4}\/.*?|\[DOI\].*?|\(\d{4}\)?|\?.*?=.*?|#.*?)$"
```

DATA CLEANING PROCEDURE

3 Here we describe the steps through which we carried out the cleaning procedure of factually invalid DOIs. The workflow of the procedure is organized as follows:


import csv
Check for cited DOIs validity
Clean invalid DOIs
Check for cleaned DOIs validity
Output CSV
output

Workflow of the data cleaning procedure.

We created a Python software for handling the whole procedure. The source code is publicly available on ISC license at http://doi.org/10.5281/zenodo.4734013.
For our data cleaning system, we devised two class objects: one called **class Clean_DOIs()**, with all the methods required to carry our cleaning/validation procedure, and **class Support()**, with all the methods required to process and dump CSV as well as carry HTTP requests. In this protocol, we describe the use of the different methods that can be imported from our classes. If you want to reproduce the procedure described in the next step, once you have verified

to have Python 3.x installed and the required libraries, you can clone our Github repository and launch the *tutorial.py* file in a terminal inside the folder or you can execute the commands listed in the following sub-sections.

Launch tutorial.py

```
python tutorial.py
```

3.1 First, with the function **process_csv_input()** from our **class Support()** we import our data in CSV format and we store it in a *DictReader*, an object that maps the information in each row to a dictionary whose keys are given by the optional fieldnames parameter.

```
@staticmethod
def process_csv_input(path:str) -> list:
    print(f"[Support:INFO Proccessing csv at path {path}]")
    with open(path, 'r', encoding='utf-8') as csvfile:
        reader = csv.DictReader(csvfile)
        return list(reader)
```

On our procedure, we perform this method on our input csv and we store it as a **<data>** variable. Optionally, it is also possible to import in the **<crossref_dois>** variable the Crossref DOIs previously stored in a CSV file as shown in step 1.2. It is recommended to perform this step having at least 64GB of RAM, as over 120 million DOIs need to be stored in a set and kept on RAM throughout the entire process. You can reproduce the experiment by launching:

```
from support import Support
from clean_dois import Clean_DOIs

data = Support.process_csv_input("./dataset/invalid_dois.csv")
crossref_dois = Support.process_csv_input("./dataset/crossref_dois.csv")
```

After this procedure, the **<data>** variable (and, if included, the **<crossref_dois>** variable) will consist of a list of dictionaries, as shown below:

```
[{"Valid_citing_DOI": "10.14778/1920841.1920954", "Invalid_cited_DOI":
"10.5555/646836.708343"}, {"Valid_citing_DOI": "10.5406/ethnomusicology.59.2.0202",
"Invalid_cited_DOI": "10.2307/20184517"},
...]
```

3.2 The procedure first checks for each cited DOI if it is factually invalid or it has become valid in the meanwhile. For this purpose, we use the function **check_dois_validity()**, from our **class Clean_DOIs()**, which takes in input a list of dictionaries, like the one described above, and does a GET request to the DOI Proxy with the cited DOI (https://www.doi.org/factsheets/DOIProxy.html) of each row of the CSV: if the status code corresponding to that specific DOI is different from 1, it means that it is not valid; otherwise, that DOI has become valid in the meanwhile. The function outputs an enriched version of the list of dictionaries in input, with two additional fields: *"Valid_DOI"*, which stores a validated DOI, and *"Already_valid"*, which stores 1 if it has become valid in the meanwhile, 0 otherwise. 1w

In case the optional **<crossref_dois>** argument has been passed, a set containing all DOIs in the corresponding dictionaries list is generated. The set was chosen and not the list because, having all the values hashed, it makes the content check instantaneous, at the expense of more RAM. Therefore, the request to the DOI Proxy Server is made only if the current DOI is not found in the set, in which case it is registered as valid.

```
def check_dois_validity(self, data:list) -> list:
        checked_dois = list()
        pbar = tqdm(total=len(data))
        for row in data:
            valid_citing_doi = row["Valid_citing_DOI"]
            invalid_cited_doi = row["Invalid_cited_DOI"]
            invalid_dictionary = {
                "Valid_citing_DOI": valid_citing_doi,
                "Invalid_cited_DOI": invalid_cited_doi,
                "Valid_DOI": "",
                "Already_valid": 0
            }
            if invalid_cited_doi in self.crossref_dois:
                handle = {"responseCode": 1}
            else:
                handle =
Support().handle_request(url=f"https://doi.org/api/handles/{invalid_cited_doi}"
cache_path=self.cache_path, error_log_dict=self.logs)
            if handle is not None:
                if handle["responseCode"] == 1:
                    checked_dois.append(
                        {"Valid_citing_DOI": valid_citing_doi,
                        "Invalid_cited_DOI": invalid_cited_doi,
                        "Valid_DOI": invalid_cited_doi,
                        "Already_valid": 1
                        })
                else:
                    checked_dois.append(invalid_dictionary)
            else:
                checked_dois.append(invalid_dictionary)
            pbar.update(1)
        pbar.close()
        return checked_dois
```

For performing the HTTP request, the function uses the **handle_request()** method of the **class Support()**.

```
def _requests_retry_session(
        self,
        tries=2,
        status_forcelist=(500, 502, 504, 520, 521),
        session=None
    ) -> Session:
        session = session or requests.Session()
        retry = Retry(
            total=tries,
            read=tries,
            connect=tries,
            status_forcelist=status_forcelist,
        )
        adapter = HTTPAdapter(max_retries=retry)
        session.mount('http://', adapter)
        session.mount('https://', adapter)
        return session

    def handle_request(self, url:str, cache_path:str="",
error_log_dict:dict=dict()):
        if cache_path != "":
            requests_cache.install_cache(cache_path)
        try:
            data = self._requests_retry_session().get(url, timeout=10)
            if data.status_code == 200:
                return data.json()
            else:
                error_log_dict[url] = data.status_code
        except Exception as e:
            error_log_dict[url] = str(e)
```

We can now instantiate the **Clean_DOIs()** class, passing none or more of the optional parameters **<crossref_dois>**, **<cache_path>** and **<logs>**,. Then, we run the **check_dois_validity()** function passing the **<data>** and storing the results in a **<checked_dois>** variable:

```
clean_dois = Clean_DOIs(crossref_dois=crossref_dois,
cache_path="./cache/doi_cache", logs=doi_logs)

checked_dois=clean_dois.check_dois_validity(data=data)
```

Depending on the dataset size, on the bandwidth and whether or not the **<crossref_dois>** parameter has been used, this step can take a long time, up to a week in the case of the over one million DOIs of the input dataset mentioned at step 1 at an average bandwidth of 100 Mb/s.

Once the execution is complete, this is how the **<checked_dois>** variable appears:

```
[{"Valid_citing_DOI": "10.1007/s11771-020-4410-2", "Invalid_cited_DOI":
"10.13745/j.esf.2016.02.011", "Valid_DOI": "10.13745/j.esf.2016.02.011", Already_valid": 1},
...]
```

3.3 Afterwards, we pass the **<checked_dois>** variable obtained in the previous step to another function called **procedure(),** aimed to clean the DOIs with the regular expressions described above, and to verify the validity of the cleaned DOIs. 5d

```
def procedure(self, data:list) -> list:
        output = list()
        pbar = tqdm(total=len(data))
        for row in data:
            valid_citing_doi = row["Valid_citing_DOI"]
            invalid_cited_doi = row["Invalid_cited_DOI"]
            unclean_dictionary = {
                "Valid_citing_DOI": valid_citing_doi,
                "Invalid_cited_DOI": invalid_cited_doi,
                "Valid_DOI": row["Valid_DOI"],
                "Already_valid": row["Already_valid"],
                "Prefix_error": 0,
                "Suffix_error": 0,
                "Other-type_error": 0
            }
            if row["Already_valid"] != 1:
                new_doi, classes_of_errors = self.clean_doi(invalid_cited_doi)
                clean_dictionary = {
                    "Valid_citing_DOI": valid_citing_doi,
                    "Invalid_cited_DOI": invalid_cited_doi,
                    "Valid_DOI": new_doi,
                    "Already_valid": row["Already_valid"],
                    "Prefix_error": classes_of_errors["prefix"],
                    "Suffix_error": classes_of_errors["suffix"],
                    "Other-type_error": classes_of_errors["other-type"]
                }
                if new_doi != invalid_cited_doi:
                    if new_doi in self.crossref_dois:
                        handle = {"responseCode": 1}
                    else:
                        handle =
Support().handle_request(url=f"https://doi.org/api/handles/{new_doi}",
cache_path=self.cache_path, error_log_dict=self.logs)
                    if handle is not None:
                        if handle["responseCode"] == 1:
                            output.append(clean_dictionary)
                        else:
                            output.append(unclean_dictionary)
                    else:
                        output.append(unclean_dictionary)
                else:
                    output.append(unclean_dictionary)
            else:
                output.append(unclean_dictionary)
            pbar.update(1)
        pbar.close()
        return output
```

In our data cleaning procedure we store the output of the **procedure()** function in the **<output>** variable:

```
output = clean_dois.procedure(data=checked_dois)
```

For each cited DOI, the function **procedure()** performs five actions:

1. it removes white spaces in the string;
2. it applies the prefix regular expression to catch eventual unwanted characters that precede a DOI. If there is a match, the regular expression removes the unwanted characters and the DOI is replaced with the cleaned one;

3. it applies the suffix regular expressions as a list of alternative unwanted patterns that may occur at the end of the string. If there is a match, the regular expression removes the unwanted characters and the DOI is replaced with the cleaned one;
4. it removes eventual unwanted characters, like double slashes, double points, xml tags and backward slashes;
5. finally, the algorithm checks if the cited DOI after the cleaning procedure is different from the one in **<checked_dois>**, if it is contained in the **<crossref_dois>** set or returned as valid by the DOI Handle API. If it is different and is validated, the algorithm stores the cleaned DOI in the "*Valid_DOI" field* of the dictionary; if it is not different, the dictionary will store no new DOI.

By applying **procedure()** to our **<checked_dois>** we obtain a list of dictionaries containing 7 fields: the four already present in the input list ("*Valid_citing_DOI", "Invalid_cited_DOI", "Valid_DOI",* "*Already_Valid"*) and, in addition, *"prefix_error", "suffix_error", "other-type*" fields, which contains, for each cleaned DOI, the number of errors that were cleaned. This is how the **<output>** variable looks like:

```
[{"Valid_citing_DOI": "10.1007/s40617-018-00299-1", "Invalid_cited_DOI":
"10.1901/jaba.2012.45-657", "Valid_DOI": "10.1901/JABA.2012.45-657", Already_valid": 1,
"prefix_error": 0, "suffix_error" 0:, "other-type": 0 },
{"Valid_citing_DOI": "10.1074/jbc.m508416200", "Invalid_cited_DOI": "10.1059/0003-4819-
100-4-483", "Valid_DOI": "", Already_valid": 0, "prefix_error": 0, "suffix_error" 0:, "other-type": 0 },
{"Valid_citing_DOI": "10.1177/2054358119836124", "Invalid_cited_DOI":
"10.1016/j.amepre.2015.07.017.", "Valid_DOI": "10.1016/J.AMEPRE.2015.07.017",
Already_valid": 0, "prefix_error": 0, "suffix_error" 1:, "other-type": 0 },
...]
```

3.4 At the end of our procedure we dump our **<output>** in a csv file through the **dump_csv()** method of the class **Support()**:

```
@staticmethod
def dump_csv(data:list, path:str):
    print(f"[Support:INFO Writing csv at path {path}]")
    with open(path, 'w', newline='', encoding='utf8')  as output_file:
        keys = data[0].keys()
        dict_writer = csv.DictWriter(output_file, keys)
        dict_writer.writeheader()
        dict_writer.writerows(data)
```

This is how the method is used in our data cleaning procedure:

```
Support.dump_csv(data=output, path="./output.csv")
```

The output of the function is then stored in a CSV file, organized in 7 fields as the dictionary mentioned above:

Valid_citing_DOI	Invalid_cited_DOI	Valid_DOI	Already_valid	Prefix_error	Suffix_error	Other-type_error
10.1007/s40617-018-00299-1	10.1901/jaba.2012.45-657	10.1901/JABA.2012.45-657	1	0	0	0
10.1074/jbc.m508416200	10.1059/0003-4819-100-4-483		0	0	0	0
10.1177/2054358119836124	10.1016/j.amepre.2015.07.017.	10.1016/J.AMEPRE.2015.07.017	0	0	1	0

The output dataset is available at http://doi.org/10.5281/zenodo.4733647 (Boente, R. et al., 2021)
Further information about the possible uses of research data generated by this project, and the support provided by the authors for reuse, are available in the Data Management Plan provided by the authors at https://doi.org/10.5281/zenodo.4733920.

STATISTICS AND VISUALIZATION

4 Finally, two visualizations were produced from the output file. For this purpose, d3.js v6.7.0, an open source Javascript library for manipulating data-driven documents, was used. In particular, a barchart and a treemap have been created, which compare the number of DOIs still not valid after the cleaning procedure with those that have been corrected, and the number of those belonging to the four classes of errors considered, i.e. DOIs already valid and DOIs with prefix, suffix, or other-type errors. The bar chart can be sorted in ascending or descending order and it is possible to hover the mouse over both visualizations to highlight the represented value and the percentage of the total. It is possible to interact with the visualization at the following address: https://open-sci.github.io/2020-2021-grasshoppers-code/.