AALBORG UNIVERSITET
STUDENTERRAPPORT

6th Semester
Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg
http://www.cs.aau.dk

Title:

A Tool for Data Wrangling Geographical Data from Open Data Portals

Project Theme:

Bachelor Project

Project Period:

Spring Semester 2022

Project Group:

cs-22-dat-6-01

Participants:

Emil Gybel Henriksen
Lasse Enggaard Rasmussen
Niels Ulrik Gajhede
Theodor Risager

Supervisor:

Olivier Paul Pelgrin

Number of Pages:

11

Date of Completion:

May 24, 2022

Abstract:

This paper introduces a new tool called TWISTER. The tool specializes in handling data from open data portals such as `Opendata.dk`. The goal of the tool is to parse any dataset containing geographical data into a common format, which only contains information that is relevant to datasets of the same type. Throughout the paper the tool and its individual components have been described to give a sense of how the tool works. The tool has been tested to showcase correct and incorrect behavior in its current implementation. The tests which consist of a small number of datasets show a good result in a few cases, and highlight some cases which could be improved. The current implementation is discussed to provide suggestions on how some of the current disadvantages could be handled, and in general improve the tool.

Resume

This paper describes a tool called TWISTER that improves the accessibility of the data managed by open data portals. In particular the danish open data portal `Opendata.dk` has been used for analysis and experiments in this paper. The tool can be found at `https://github.com/theodor349/P6-Making-Sense-of-OpenData.dk-`.

Open data portals are websites that link up public data and information in a central place. From the portals data can be searched and accessed in a straightforward manner. The purpose of open data portals is to make public data more accessible for interested parties. By making data more accessible the hope is that the public data can be used widely to benefit the future of society. [1]

Many data portals allow data providers to share a large variety of data types and formats. This approach makes it easy for the providers to share their data, but can make data processing with data from the portals more difficult.

Data can be stored in a variety of ways and in different formats, however most data processing programs require data to be stored in a specific and uniform manner. The reason most data processing programs have restrictions on datasets is because it can be difficult to predict and support all current and future dataset formats.

As with many other open data portals, `Opendata.dk` contains a lot of different types of data and formats. One of the problems with having many different data formats is that it is difficult to unify them into one uniform format, while retaining the same expressive power. The reason why it is so difficult, is that some data formats have a structure that provides the data with a specific meaning. When a conversion to a uniform format is applied, the defining structure will be transformed and the information encoded in the original data structure is lost.

In order to combat this TWISTER uses two types of objects throughout the pipeline to represent the original dataset and its transformations. The first type of object is the intermediate dataset and intermediate objects which represent the source dataset. This is done by storing information about the source dataset in the e.g. file name and type. Structural information can be added to either the intermediate object if it is object specific, or to the dataset if it defines the dataset as a howl. Because TWISTER needs to be able to handle many different formats, a similar structure to JSON was chosen for the intermediate object. As that format is very flexible and able to e.g. represent a CSV row as an object with as many attributes as there are columns in the CSV. And the objects stored in tables e.g. a relational database could be represented using nested attributes.

The other type is the output dataset and intermediate output objects which represents the resulting objects which are to be printed to a target source. The intermediate output objects are generated by TWISTER as an internal step in the tool, before it is printed to the desired format. This data structure is the final intermediate representation of the data and is the structure used by the interchangeable printer component to generate the final output of the tool. The object consists of a key-value pair for each custom property. The properties of an output object are specified in a specialization.

TWISTER also has an output log object, which is populated with information about the classification phase. Its intended purpose is to enable better debugging and optimization of the classification component.

TWISTER is split into seven components each with its own responsibility. Each component is dependent on the work of the previous component. The first component `Input Parser` is responsible for reading the source file and converting it to an intermediate dataset with a collection of intermediate objects. This dataset is then passed along to the `Pre Processor` which will do any pre-processing of the data, before it is sent to the `Labeler`. The `Labeler` will assign labels to all the attributes on each intermediate object. These labels will be used in the following components. First of which is the `Post Processor` which will process the data before it is sent on. An example is adding the Coordinate Reference System to the dataset if it was not found by the `Input Parser` component. When all the data has been found and the `ObjectAttribute`s labeled, then the `Classifier` will use the labels and data about the dataset to decide what type of dataset it is. After it has been classified the `Output Generator` will select the correct specification and convert the intermediate dataset into an output dataset. This output dataset is then handed over to the last component, the `Printer`, which will print the output dataset to a specified output format e.g. GeoJSON or a relational database.

TWISTER is implemented using Dependency Injection, which means that all the component's dependencies are requested using an interface and injected by the system. This enables easy substitution of components.

The results presented in the results section, show that the tool is indeed able to improve the accessibility for a number of cases. However it is also shown that TWISTER has a number of cases where it is unable to provide valuable results. Some of the cases that lead to incorrect results can be corrected by introducing more features and improving already existing features. However it has also been found that datasets with sparse information in some cases simply cannot be classified and parsed correctly. A dataset with this sparse property could be one that only contains coordinates, where there is no information about the Coordinate Reference System anywhere in the data. This is problematic as there exists many different Coordinate Reference Systems that potentially could be the right one, and picking the wrong one will result in a completely different and incorrect translation.

A Tool for Data Wrangling Geographical Data from Open Data Portals

Emil Gybel Henriksen[1], Lasse Enggaard Rasmussen[2],
Niels Ulrik Gajhede[3], and Theodor Risager[4]

Department of Computer Science, University of Aalborg, Denmark

[1]eghe19@student.aau.dk, [2]lrasm19@student.aau.dk, [3]ngajhe11@student.aau.dk, [4]trisag19@student.aau.dk

This paper introduces a new tool called TWISTER. The tool specializes in handling data from open data portals such as `Opendata.dk`. The goal of the tool is to parse any dataset containing geographical data into a common format, which only contains information that is relevant to datasets of the same type. Throughout the paper the tool and its individual components have been described to give a sense of how the tool works. The tool has been tested to showcase correct and incorrect behavior in its current implementation. The tests which consist of a small number of datasets show a good result in a few cases, and highlight some cases which could be improved. The current implementation is discussed to provide suggestions on how some of the current disadvantages could be handled, and in general improve the tool.

1 Introduction

This paper describes a tool called TWISTER that improves the accessibility of the data managed by open data portals. In particular the danish open data portal `Opendata.dk` has been used for analysis and experiments in this paper. The tool can be found at `https://github.com/theodor349/P6-Making-Sense-of-OpenData.dk-`.

Open data portals are websites that link up public data and information in a central place. From the portals data can be searched and accessed in a straightforward manner. The purpose of open data portals is to make public data more accessible for interested parties. By making data more accessible the hope is that the public data can be used widely to benefit the future of society. [1]

Many data portals allow data providers to share a large variety of data types and formats. This approach makes it easy for the providers to share their data, but can make data processing with data from the portals more difficult.

Data can be stored in a variety of ways and in different formats, however most data processing programs require data to be stored in a specific and uniform manner. The reason most data processing programs have restrictions on datasets is because it can be difficult to predict and support all current and future dataset formats.

As with many other open data portals, `Opendata.dk` contains a lot of different types of data and formats. One of the problems with having many different data formats is that it is difficult to unify them into one uniform format, while retaining the same expressive power. The reason why it is so difficult, is that some data formats have a structure that provides the data with a specific meaning. When a conversion to a uniform format is applied, the defining structure will be transformed and the information encoded in the original data structure is lost.

This issue has been resolved by storing and maintaining this information independently from the data itself in a new data structure called an intermediate dataset. The intermediate dataset contains all the information that can be obtained from the source dataset. By storing the information about the dataset separately from the objects the tool is able to use the information about the dataset throughout the process of parsing the data.

Another issue with mapping all data into the same uniform format is that the uniform format is not necessarily supported by the targeted processing programs. If the output format was GeoJSON, some programs might be able to process it, however many other programs will not be able to process it. Therefore the component responsible for generating the output format, is made in such a way that it can be substituted for a new one. This means that if a specific program needs the data to be stored in e.g. a relational database using a specific schema, the component can be substituted with a component that is able to output in that format. Due to the time constraints for this paper, only one version of the component was made. This component is able to generate GeoJSON as output.

The tool developed for this paper is called TWISTER and is a pipeline split into seven components, where two of those components can be dynamically changed. These dynamic components makes it possible to define specific descriptive patterns in the datasets without recompiling the code of TWISTER. And allows the user to customise the classification to the specification of the datasets they are working with and thereby improve its accuracy on those datasets.

2 Related Works

This paper presents a tool for assisting in what is commonly known as "data wrangling" or "data munging". Data wrangling is the process of cleaning, structuring, enriching and validating data in order for the data to be useful for some application.

One already existing tool for aiding the process of data wrangling is ir_datasets [2], specifically to be used in information retrieval research. ir_datasets allows the user to download and use datasets through either a Python API, or a command line interface, and have them be represented in data structures. However, ir_datasets only supports a number of already implemented information retrieval related datasets. These datasets consist of documents, and can be used, for example, in the testing of a search engine. ir_dataset cannot process datasets that are not implemented [2].

A research paper by Martin Koehler et al. [3] proposes a different approach to data wrangling which would make use of the so-called "data context", in order to inform the wrangling process and thereby increase precision. This data context could be in the form of metadata, reference data or other types of contextual data. An increase in precision was seen when data context was incorporated into the wrangling process, alongside a little outside intervention from a data scientist specifying the target schema, based on said context data. A different paper by Ignacio Terrizzano et al. [4] on the challenges of using data wrangling to fill a "data lake". The paper describes how the process of "grooming data", which is described as making data consumable by analytical applications i.e. homogenizing data, would be an important step in filling a data lake, as the format of raw data can vary wildly, and therefore is not necessarily usable in its raw form. The paper also mentions how the use of metadata would be important for this step.

3 Objects

TWISTER uses two types of objects through out the pipeline to represent the original dataset and its transformations. The first type of object is the intermediate dataset and intermediate objects which represent the source dataset. The other type is the output dataset and intermediate output objects which represents the resulting objects which are to be printed to a target source.

Besides these two object types, this section will also describe the output log object, which is populated with information about the classification phase. And will enable better debugging and optimization of the classification component.

3.1 Intermediate Dataset

The intermediate dataset contains information about the original dataset and a list of all the objects in the source dataset. An example can bee seen in Listing 1.

The *original name* and *source type* is always stored, as that can be used to determine what kind of operations should be executed on the dataset. For example the original source type, is used to determine what input parser to use, as different input sources require different parsers. And the original name is part of the process to determine what kind of data the dataset contains.

Other information that is found to be useful for the entire dataset is stored as *key-value pairs*. An example is the Coordinate Reference System (CRS) which is used to convert coordinates to a standard for representing coordinates called WGS 84, this is then stored on the dataset instead of on all intermediate objects.

```
"Intermediate Dataset": {
   "Original File Name": "filename",
   "Original File Extension": "csv",
   "Properties": {
      "Key 1": "Value 1",
      "Key N": "Value N",
   }
   "Objects": {
      "IO 1": {... },
      "IO N": {... },
   }
}
```

Listing 1: Example showcasing the intermediate dataset

All objects in the dataset are stored as a list and populated by the input parser component, which converts the source data into intermediate objects.

3.2 Intermediate Object

The intermediate object's purpose is to represent all *objects* in any given dataset. This means that it should be able to represent rows from CSV, objects from JSON or xml, or database tables.

In order to represent all these different formats, a similar structure to JSON was chosen. As that format is very flexible and able to e.g. represent a CSV row as an object with as many attributes as there are columns in the CSV. And the objects stored in tables e.g. a relational database could be represented using nested attributes.

3.2.1 Object Attribute

An intermediate object contains a list of `ObjectAttributes`. An `ObjectAttribute` contains a *name*, a *value*, and a *list of labels*. In most cases the name will be the original name of the source attribute e.g. column name for CSV. And the value will be set to the original value of the source attribute e.g. the value of the cell in a CSV.

Labels are a central part of TWISTER as it is used to classify the dataset and retrieve relevant information for intermediate output objects. Each `ObjectAttribute` is assigned one or more labels in the labeling component.

A *label* has two properties a *label name* and a *probability*. The label name is essentially the key, which is used to search for the label. And the probability is used to indicate how confident TWISTER is that the `ObjectAttribute` is what the label resembles. For example a point is an `ObjectAttribute` with only two children which both have to be numbers of the type double. The probability of a point label is given by the average of the two children's probability.

In order to better represent different data types the `ObjectAttribute` is implemented as an abstract class, which means it cannot be instanced on its own. Instead it's inherited into multiple classes, one for handling lists of `ObjectAttributes`, another for dates, a third for null values and then one for text, longs, doubles, and Booleans. And the value property in the `ObjectAttribute` is implemented as an object, as it enables TWISTER to store all kinds of data types e.g. lists and booleans, while still being able to easily traverse the tree of `ObjectAttributes`.

3.3 Output Dataset

The output dataset object contains information about the original dataset and a list of the resulting *intermediate output objects*. The information it contains about the dataset is the *original filename* and its *file extension*. This information is used as part of the printing component to save the resulting GeoJSON file with the same name as the original source file.

3.4 Intermediate Output

The intermediate output objects are generated by TWISTER as an internal step in the tool, before it is printed to the desired format. This data structure is the final intermediate representation of the data and is the structure used by the interchangeable printer component to generate the final output of the tool.

The object consists of a *key-value pair* for each *custom property*. The properties of an output object are specified in a *specialization*.

All properties are represented using the list of key-value pairs, where the *key* is the name of the property and the *value* its value. This approach has been chosen because it is able to accommodate a large variety of specializations. As it is possible to create specializations with zero or many properties. One of the shortcomings of this approach is that it is not trivially extended to composite properties, like collections and nested objects.

A property is specified using a *name* and a list of *target labels*. The target labels are used to find the attribute on the intermediate object which is most likely to contain the value for the given property. So for the case of a parking spot a property to represent the number of spots, can be created with the key being "NumSpots" and the target labels being "Parking" and "Amount".

```
"Parking Spot Specilization": {
    "GeoFeature": "Polygon",
    "Properties": {
        "NumSpots": ["Parking", "Amount"]
    }
}
```

Listing 2: Example parking spot specification

Because the focus of this paper is geographical data, a new object which inherits from the intermediate object, called *Geo Data Output* has been created. This object also contains a *geometric feature*, which can also be specified in the specification. A geometric feature could for the case of a parking spot be a Polygon or a MultiPolygon. As each parking spot is defined as an area, which a Polygon represents. The list of possible geometric features are the same as outlined in the GeoJSON standard, so it is possible to choose between *Point*, *Polygon*, *LineString*, and collections of them called *MultiType* (e.g. MultiPolygon).

An example of a parking spot specification can be seen in Listing 2.

3.5 Output log object

This object is designed to store information gathered during the classification of datasets. The information is printed to a log file after classification is completed and can be used to analyze and improve the classification process.

The output log object is the following set of relevant attributes. Firstly it contains the *file name* so it can be identified. Then it has a Boolean attribute that indicates if the classification was successful, along with chosen *classification* and final score. Next it has the classifications that were not chosen with their individual scores. Then it contains some general information about *labels*. For each label in the dataset it provides information about how many times the label was found, the label name and the average confidence for the correctness of the label. The object also has information about the *total amount of objects* in the dataset, how many of the objects were *labeled* and how many were given a *custom label*. Lastly it contains two derived attributes, one for the percentage of objects with a custom label, and one for the average confidence of labels across all the different labels, both of which are calculated on the basis of the other data.

4 Components

TWISTER is split into seven components each with its own responsibility. Each component is dependent on the work of the previous component as can be seen in Figure 1.

The first component `Input Parser` is responsible for reading the source file and converting it to an intermediate dataset with a collection of intermediate objects. This dataset is then passed along to the `Pre Processor` which will do any pre-processing of the data, before it is sent to the `Labeler`. The `Labeler` will assign labels to all the attributes on each intermediate object. These labels will be used in the following components. First of which is the `Post Processor` which will process the data before it is sent on. An example is adding the Coordinate Reference System to the dataset if it was not found by the `Input Parser` component. When all the data has been found and the `ObjectAttributes` labeled, then the `Classifier` will use the labels and data about the dataset to decide what type of dataset it is. After it has been classified the `Output Generator` will select the correct specification and convert the intermediate dataset into an output dataset. This output dataset is then handed over to the last component, the `Printer`, which will print the output dataset to a specified output format e.g. GeoJSON or a relational database.

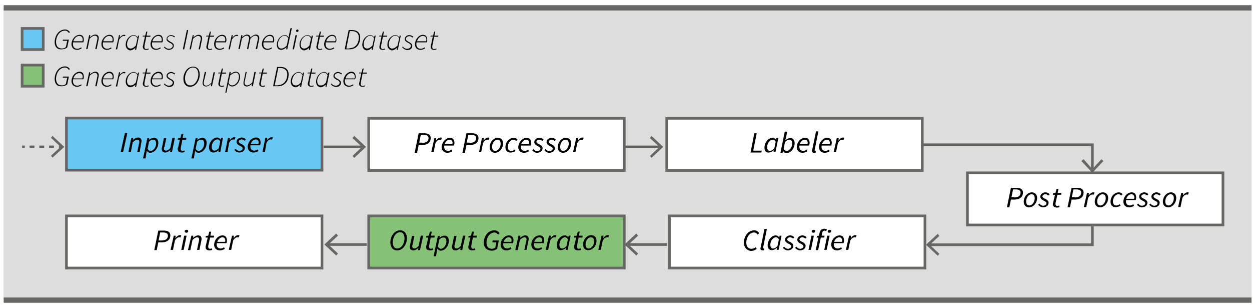

Figure 1: The pipeline architecture

TWISTER is implemented using Dependency Injection, which means that all the component's dependencies are requested using an interface and injected by the system. This enables easy substitution of components.

An example could be if the user wanted the final output from TWISTER to be a not yet supported format. In this case the user could make do with writing a printer that implements the `IPrinter` interface and outputs the given format, and then replace the implementation of the interface in the Dependency Injection system.

4.1 Input Parser

The input parser is the first part of the pipeline in TWISTER. This component's responsibility is to generate the *intermediate dataset* that is used as the data structure for the pipeline until the input data has been classified and is ready to be parsed to a *specialized object*.

The component is split into different sub components that each handle a different type of file conversion. Then whenever the component is called it uses the path to the original data to find the file extension of the file. With this information the input parser chooses a sub component fit for the format conversion which then carries out the conversion to the intermediate dataset.

For this paper two different sub components have been developed. These components handle the two completely different file formats JSON and CSV.

4.1.1 JSON

Parsing a JSON or GeoJSON file to an intermediate dataset is handled by the JSON Parser. A `JsonTextReader` from the library Newtonsoft is used to read the original data.

The JSON Parser uses the reader to read through the original dataset one token at a time and parses the tokens to their corresponding attributes of an intermediate dataset.

After the intermediate dataset has been constructed from the reading process, the intermediate dataset is handed to the intermediate dataset splitter for analysis. The analysis is carried out to remove any data in the dataset that does not correspond to a dataset entry.

The splitter first traverses the entire dataset and registers patterns. A pattern is a class that represents a data node in the intermediate dataset. It contains the attributes needed to evaluate whether two data nodes have the same structure, which are the following:

- Depth
- Count
- Signature

Where depth is the distance from the root object. Count is the amount of times the pattern has been located and the signature contains information about the data nodes children and their data types. After the traversal is completed the registered patterns are sorted based on their depth. Then the patterns are searched for the first entry that has a count of more than one. When such a pattern is found a new intermediate dataset is constructed. This new intermediate dataset is based on the old but only data that fits the pattern is retained.

The reason why this splitter has been conceived is that in some datasets data about the dataset is stored at the root level together with a collection of entries. And it is these entries that should be converted to intermediate objects and not the entire dataset. Had this not been done, then each dataset would be converted to a single intermediate object instead of a collection of many.

4.1.2 CSV

The CSV Parser starts by reading the first row, as it contains the name of each column and in turn the name of each attribute on the resulting intermediate object. Following this it iterates through each line one by one and retrieves the values corresponding to each column.

Each value is then parsed by trying to parse to different data types, to make sure that types like integers are not represented as text. This is done for primitive data-types and for geometric data-types. The primitive datatype is fairly straight forward as C# has the ability to `TryParse` a value to a data-type [5]. And if the parsing was successful then the value is perceived as being of that type. The order in which this is done matters, as an integer can always be parsed as a double number, and therefore the integer should be `TryParsed` first.

As the focus of the paper is on geometric data a special parsing method has also been implemented. This method can identify a column as being the column containing the geometric data. This is done by looking at the attribute name, as it was found that CSVs from `Opendata.dk` contain a column named "wkb_geometry" or "Geometry_SPA" if it contains geometric data. When the values have been identified to contain geometric data it is then parsed as such. This is done using regular expressions as polygons, linestrings and points all have the same structure across datasets.

The resulting intermediate `ObjectAttribute`s are all except for the geometric data attribute not nested objects. And the geometric attribute is a nested object with different levels of nesting, depending on the type of geometric form. An example object can be seen on Figure 2.

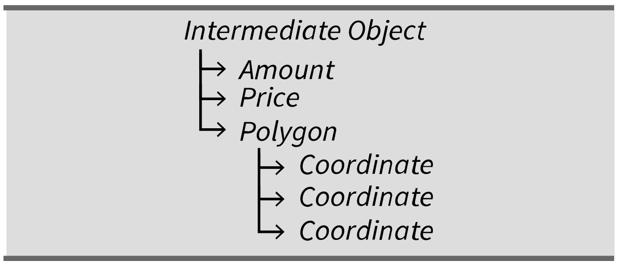

Figure 2: Example of an intermediate object

4.2 Pre Processor

The goal of the pre processor is to transform the intermediate objects before they are handed to the labeler. An example of a transformation could be an `ObjectAttribute` which contains an address with country, city, street name, and street number. It would be ideal to have that `ObjectAttribute` transformed into a `ListAttribute` with four child attributes where all, but the street number, are either a `TextAttribute` or a `LongAttribute`.

At the time of writing this functionality has not been implemented, however the part of the input parser which handles CSV contains similar functionality, to convert the `ObjectAttribute` that contains the geometric data into the correct tree structure, which in future work could be moved to the pre processor as this functionality can be useful for other input formats.

4.3 Labeler

In this component all attributes contained in the Intermediate objects of the intermediate dataset are traversed and labeled. A label is an object that contains a *name* and a *probability*. The probability describes with how much confidence the label was assigned. The labels assigned to attributes in this component are later used as part of the classification of the dataset.

The labeling component is split into three different parts, that each handles a unique set of labels.

The sets developed for this paper are the following:

- Data types
- Geographical objects
- Custom labels

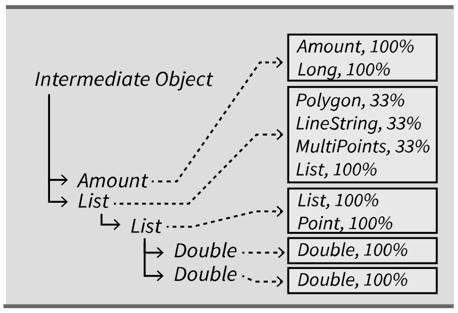

Figure 3: An intermediate object with labels illustrated by dotted lines.

This approach has been chosen to make it easy to expand, maintain and remove different sets of labels. This approach also makes it possible to make sets of labels that are dependent on other labels. As an example a point label requires two nested double attributes, so labeling data types before looking for coordinates simplifies the labeling process of coordinates. This is also illustrated in Figure 3.

Data type labels describes the data type of the attribute and is identified by analyzing the original data. All attributes in the entire dataset will have at least one of such labels after the traversal. Before attaching data type labels and probabilities all data about a given attribute is collected across all intermediate objects. When the data has been gathered an analysis is carried out, which looks at the present data types and sees if they can fit into a single data type category, to see if it is possible to determine with certainty what data type the attribute should have. If it is not possible to figure out with certainty, the attribute is given all data types that has been found across the intermediate objects and the confidence is divided between the labels and calculated based on how often the data type was found.

Geographical objects are identified by analyzing the nested attributes of an attribute. If a pattern that fits a description of a geographical object is discovered a label with the name of the geographical object is added. In the case where the nested structure matches more than one geographical object, the attribute receives all fitting geographical labels. As with the data type labels probability is divided evenly between potential objects. The geographical object *coordinate* is a special case and the probability for this object is calculated based on the average probability that the nested attributes are double attributes. The labeler is not designed to make the choices between potential geographical labels as the correct choice of attribute classification depends entirely on the dataset classification, which is determined in a later part of the pipeline.

Custom labels are a dynamic part of the pipeline that can be changed without recompiling the program. These labels are defined in a JSON file, where it is possible to add and remove labels according to different use cases. Custom labels are defined as a *target* that is searched for in the attributes. If the target is found either as part of the attribute name or in the attribute value, the attribute is given the custom label. In the case where the target could not be found a look up is performed in a thesaurus to see if a synonym can be found within the attribute and if this is the case the custom label is given to the attribute. The custom labels play a central role in improving and specializing the dataset classification because they make it possible to alter what the dataset will be classified as.

The probability calculations for custom labels behave differently from the previously described label sets. The main reason is that relations between custom labels are unknown and can change as they can be manipulated dynamically. Therefore each custom label is categorized as an independent set and will be labeled with certainty if the target is found directly and with a probability of 50 % if it is found by using the thesaurus.

4.4 Post Processor

The goal of the post processor is to add any information that might be needed in the later components, but cannot be added before the intermediate objects have been labeled. As of writing it only entails finding the CRS as it is needed to convert coordinates from their old system to the WGS 84 CRS. This cannot be done before the intermediate objects have been labeled, as it needs a reference to a coordinate to estimate the correct CRS.

The way the CRS is estimated is by converting a coordinate using different Coordinate Reference Systems and checking whether the converted coordinate is in a target area. An example can be seen in Figure 4.

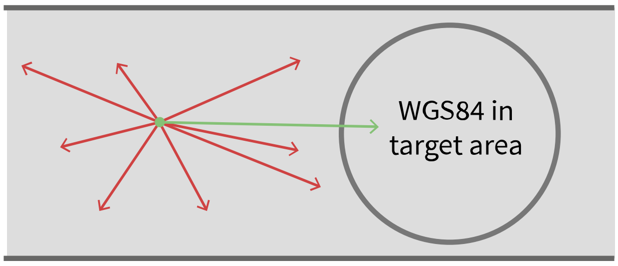

Figure 4: Example of coordinate translation using different CRSses, where the green CRS translation moves it to the target area

Unfortunately no other way has been found. The target area can be specified using the *app-settings* and for this project the area has been set to a square around Denmark excluding Bornholm. Because `Opendata.dk` is only concerned with data from Denmark. The reason why Bornholm has been excluded is that in the Universal Transverse Mercator system [6] Denmark has split into two squares 32N[7] and 33N[8], but in regular use the 32N square has been extended to cover the rest of Denmark except for Bornholm which is still in 33N. [9] And when converting a 32N coordinate to 33N it can be moved to the same longitude as Bornholm, which would result in a false positive, if Bornhold was included in the target area.

4.5 Classifier

The task of the classifier is to use the previously obtained information about the dataset to classify the original data, such that TWISTER can generate the correct output. The classifier component is dynamic which means the definition of the classification requirements can be changed without recompiling the code of TWISTER.

While the classifier is running it saves the analysis information and when the classification has been completed the component prints this information to a log file. The information in the log file can then be used to correct and improve the classification process of TWISTER.

The classification process is based on *specification requirements* and *scores*, where the specification that achieves the highest total score during the analysis is chosen as the dataset's classification.

Each classification consists of two sub classifications. A sub classification called *title specification* based on the name of the original dataset and a sub classification called *content specification* based on the content of the dataset. An example of a content specification can be seen in listing 3. The scores of both sub classifications are added together for the total score. By default TWISTER will give a score of 1 for a matching title specification but only a score of 0.6 for a matching content specification. These scores can however be changed. This has been chosen because file naming often is used to describe datasets in a general sense to make them easier to identify. The analysis of the content of the dataset can be ambiguous as the content in its entirety might match several classifications.

```
    "ContentSpecification": [
        {
            "DatasetClassification": "Parking",
            "Score": 0.6,
            "Requirements": [
                "Parking",
                "Point"
            ]
        }
    ]
```

Listing 3: Example of a content specification that contains a Parking specification – where Point is the label for coordinates

The first thing the classifier does is to search and add all the identified labels in the dataset to a list needed for the content specification calculations.

Then the classifier goes through each classification and calculates its total score. First the title score is calculated by searching the original file name for each of the defined requirement strings, if a match is found the title specification score is achieved.

After the title classification has been calculated the content calculation is performed. The content calculation is different from to the title calculation. In this classification the requirements represents labels and all the required labels have to be located to achieve the full content classification score. When both scores have been calculated the scores are added together.

After all classification scores have been calculated the one with the highest score is picked and the dataset is given that classification.

4.6 Output Generator

The goal of this component is to convert the intermediate dataset into an output dataset, by extracting specific information out of the intermediate objects and generating intermediate output objects. E.g. if an intermediate object that represents a parking spot has 10 attributes, but the specialization only specifies "NumSpots" as a property of a parking spot. Then the output generator's goal is to extract that one property from the intermediate object.

The extraction of information happens in two parts, the first of which copies the general information about the dataset from the intermediate dataset to the newly created output dataset. Following that the appropriate specialization is selected for converting the intermediate objects. The specialization is chosen based on the classification from the classifier. When it is selected, each intermediate object is converted one by one in isolation, which makes it possible to run in parallel and thereby speed up the process.

As outlined in Section 3.4 the intermediate output specialization contains a list of properties, with a set of target labels. These target labels are used to find all the `ObjectAttribute`s that could contain the value for those properties. Retrieving these matches are describe in Subsection 4.6.1. When the matches have been retrieved they are sorted by best match, which is described in Subsection 4.6.2. After they have been sorted the value of the `ObjectAttribute` with the best match is retrieved and assigned to the property.

The *geometric feature* is special as it is a composite property, which means that it cannot be found on a single `ObjectAttribute`. Therefore it is handled separately. First the root `ObjectAttribute` is found using the same approach as for the properties. When it is found it is traversed as either a Polygon, Linestring or what else is specified in the specialization. When traversing the `ObjectAttribute`s the geometric feature is constructed, and at the end it is added to the intermediate output object. Because TWISTER uses the same Coordinate Reference System as GeoJSON, all points must be converted to the World Geodetic System 1984 [10]. Therefore before a new point is added, it is converted to WGS 84. To do this the original CRS must be known, and this can be retrieved from the dataset, as it has been added in either the input parser or the pre processor component.

Most geometric features can be retrieved one-to-one but polygon must adhere to the right-hand rule which states that all points in the polygon must be ordered clockwise. Therefore when the output generator is working with polygons, they are all checked for this rule. Every polygon with points that are not clockwise have their points reversed.

4.6.1 Finding Matches

This method is given a list of labels to look for and an intermediate object. It then traverses the intermediate objects `ObjectAttribute`'s to find matches for each of the labels. Whenever it finds an `ObjectAttribute` that contains one of the labels, it is added to the result for the given label. At the end, a list of key-value pairs are returned. Where the key is the label and the value is a list of `ObjectAttribute`s that matches.

4.6.2 Sorting by Best Match

Each match has two properties; probability and count. Probability represents the combined confidence of all the matched labels on the `ObjectAttribute`. Count represents how many target labels where found. They are sorted first by count and then by probability, as if there are many matched labels it is more likely that it is the correct attribute.

4.7 Printer

This component is responsible for printing the output dataset to a target format. The printer is implemented using Dependency injection, and it can therefore easily be substituted with a new by changing the implementation of the `IPrinter` interface.

The `IPrinter` interface contains a single method named `Print` which returns a task and takes two parameters an integer resembling the *file index* and the *output dataset*. The integer is used to make sure that even if multiple dataset sources have the same name, the resulting files will have different names as the integer is appended to the filename. The output dataset contains all the output objects which has to be printed.

4.7.1 GeoJSON

The format which has been implemented at the time of writing is GeoJSON. However substituting the printing component with a new one, can enable printing to other formats like a relational database, CSV, or other custom formats.

The GeoJSON standard used for the printer implementation is the one outlined by the Internet Engineering Task Force (IETF) in 2016 [11]. From this specification a *feature collection* is used to store one or more objects. And the specification specifies that the JSON must include the *type* which in this case is "FeatureCollection" and an array of *features*, which corresponds to the output objects generated by the output generator. Each feature in the array must contain three attributes *type*, *geometry*, and *properties*. The type must be set to "feature" as the object is referred to as a feature of the collection. Geometry contains the type of geometry e.g. Polygon and an array named coordinates which contain the geometric data. The last attribute properties contain all the properties found in the list of key-value pairs in the intermediate output, which for a parking spot could be "NumSpots" and "Address".

5 Results

In this section the results of running TWISTER on ten datasets that have been pulled from `Opendata.dk` are shown. Five of the datasets contain parking spot data, and the other five contain data from different kinds of routes. The datasets are either in GeoJSON or CSV format. TWISTER was then run on these ten datasets. In the following subsections the result of running the tool on these ten datasets, will be analyzed. This will highlight some of the advantages and disadvantages of the tool.

5.1 Classification Accuracy

Table 1 shows the results of running TWISTER on ten datasets. The *Set* column is an identifier for each dataset. *Form* shows what format each dataset was before running the tool. *Coords* indicates whether or not TWISTER was able to extract the geographical data from the source dataset. *Type* indicates if the dataset is of type route (R) or Parking (P). The *Class.* column is a boolean that shows if a dataset has been successfully classified or not. The *Avg. Confi.* column indicates the average confidence over all the labels given to the different properties in a dataset. *Score* represents the score each dataset has been classified with as explained in Subsection 4.5. As can be seen in Table 1 the dataset with the lowest average confidence is dataset 10 with 72.63% and the highest is dataset 1 which has 100%. The average across all datasets is 84.48%.

Set	Form	Coords	Type	Class.	Avg. Confi.	Score
1	Geo	False	R	Sucss	100.00%	1.0
2	CSV	False	R	Sucss	91.64%	1.0
3	Geo	True	P	Sucss	84.93%	1.0
4	Geo	True	R	Sucss	79.09%	0.6
5	CSV	False	R	Sucss	90.15%	1.0
6	CSV	True	R	Sucss	83.82%	1.6
7	Geo	True	P	Fail	82.61%	0.0
8	Geo	True	P	Sucss	76.60%	1.0
9	CSV	False	P	Sucss	83.33%	1.0
10	Geo	True	P	Sucss	72.63%	0.6

Table 1: List of the ten datasets used in the experiment

The average *confidence* for a dataset shows the average confidence overall labels in that dataset. The number therefore indicates how well the data in the dataset has been classified. An average of 84.48% suggests that most of the data have been labeled fairly confidently.

As indicated in Table 1, dataset 1 has an average confidence of 100.00%. Although this is a case in which the dataset has been wrongfully parsed by the input parser. Further detail of this example will be shown in Subsection 5.3.

As Table 1 shows, all datasets except for one were classified successfully. Most of these classifications were made due to the naming of the dataset file itself, containing keywords that helped classify the dataset. Table 2 shows the result of running the same ten datasets again, however this time their filename was changed, to not include anything meaningful. As this makes sure that the classifier would not use the filename to classify the dataset. This yields a result in which the majority of datasets were not classified.

Dataset	Classification	Score
1	Fail	0
2	Fail	0
3	Fail	0
4	Success	0.6
5	Fail	0
6	Success	0.6
7	Fail	0
8	Success	0.6
9	Fail	0
10	Success	0.6

Table 2: List of the ten datasets classifications, where the name of the dataset does not contain information

When comparing the results from Table 1 and Table 2 it can be seen that the name of the dataset contributes a lot to how a dataset is classified. The *scores* from Table 1, shows that all except for one dataset has received a score. The datasets with a score of 1, suggest that the classifier only uses the name of the dataset to determine the type. And the ones where the score is 0.6, suggest that it only uses the content of the dataset.

Comparing the two tables shows that most of the datasets were not classified when the name was changed. It also shows that dataset 8 gets a score for parking of 1 in Table 1 because of the file name, and in Table 2 route scores 0.6 because of the content present in the dataset.

5.2 Example of Correct Behavior

In this section we look at an example in which TWISTER , did the intended job. The dataset that will be examined is dataset 6. Listing 4 shows one row from the original CSV file of dataset 6. Note that Listing 4 is not in CSV format as the format has been changed to JSON to make it easier to read in the paper. On line 18 of Listing 4 the coordinates have been omitted, to save some space, although in the original file the coordinates would have been placed inside the parenthesis after `LINESTRING`.

```
1  FID: CYKELRUTER.31
2  temakode: 0
3  temanavn:
4  systid_fra: 03/04/2012
5  systid_til:
6  oprettet: 03/04/2012
7  cvr_kode: 29189640
8  cvr_navn: Silkeborg Kommune
9  bruger_id: skc
10 oprindkode: 4
11 oprindelse: FOT/Teknisk kort
12 statuskode: 3
13 status: Gaeldende/Vedtaget
14 link:
15 type: Silkeborg, Hjoellund og Vrads
16 navn: Silkeborg, Hjoellund og Vrads
17 laengde: 47km
18 Geometry_SPA: LINESTRING()
19 Geometry:
```

Listing 4: Reformatted input of dataset 6

After running TWISTER on the input dataset shown in Listing 4 the output shown in Listing 5 is produced. Listing 5 consists of a `featurecollection`, the featurecollection has a number of features each representing one route. Each feature has a geometry object which contains a list of coordinates. Most of the coordinates which make up the route have been omitted from the example to save space. Furthermore a feature has a properties object which has different values depending on the specialization as described in Subsection 3.4, in the case of a route the values are Name and Length. The index value is added mostly for debugging purposes.

```json
{
  "type": "Feature",
  "geometry": {
    "type": "LineString",
    "coordinates": [
      [
        9.470058512082682,
        56.147751784554345
      ],
      ...
      [
        9.474243370866773,
        56.1479137059335
      ]
    ]
  },
  "properties": {
    "Name": "Silkeborg, Hjoellund og Vrads",
    "Length": "47 km",
    "Index": 29
  }
}
```

Listing 5: Part of the output for dataset 6

This example shows a case in which TWISTER performs as intended, and therefore pulls relevant data into a format that represents a route. Although there exist cases in which the current approach will not function as intended. An example of such a case will be given in Subsection 5.3.

5.3 Example of incorrect behavior

In this section some of the disadvantages of TWISTER, will be highlighted. In Listing 6 an example of dataset 1 is shown. The reason TWISTER fails to handle this file correctly is that the source dataset only has one feature in its featurecollection. This leads to the input parser being unable to find any meaningful pattern. Because the only pattern it finds is one for each coordinate of its geometric feature. Therefore each coordinate is parsed into an intermediate object. And therefore TWISTER is unable to find any of the target properties such as Name or Length.

Other examples in which TWISTER does not work as intended are datasets containing parking spot information, that have chosen to represent their geographical data as points instead of polygons. As TWISTER expects parking spot datasets to consist of polygons/multipolygons.

In datasets that have more than one column which contains for example Name or Amount, the tool chooses the last one encountered in the order they are presented in the source dataset, as the one to put into the output object. An example is dataset 3 which contains a column named ANTAL_PLADSER (AMOUNT_SPOTS) and a column named ANTAL_LYSMASTER (AMOUNT_LAMPPOSTS) and due to ANTAL_LYSMASTER being placed further down in the dataset, the non intended value corresponding to ANTAL_LYSMASTER, is the one put into the output file.

```json
{
  "type": "Feature",
  "geometry": {
    "type": "LineString",
    "coordinates": []
  },
  "properties": {
    "Index": 0
  }
}
```

Listing 6: Part of the output for dataset 1

6 Discussion

This section will discuss the findings from the results section and the usability of TWISTER.

As was found in the results Section 5 TWISTER is able to successfully classify datasets that contain information about the dataset in its name. However if this information is removed, it struggles to correctly classify the datasets. Therefore improving the labeling component would increase the accuracy of the classifier and thereby enhance TWISTER . This change is discussed in Subsection 6.1.2.

As the name of the file suggests, metadata can have a large impact on the accuracy of the classification. Therefore adding more metadata to each dataset could improve TWISTER . An idea of how to get more metadata is presented in Subsection 6.1.1

When the dataset has been classified, the Output Generator retrieves the relevant information from the original intermediate object. As was pointed out in Subsection 5.3, if multiple attributes are assigned the same label with the same probability, the generator will select the last occurrence. This might not always be what is intended, as "Count" might be preferred over "Count_SomethingExtra". Therefore an improved approach for assigning probability to a label is needed, to improve the retrieval of information. Improvements for this are also discussed in Subsection 6.1.2.

6.1 Future Works

The following subsections will cover some suggestions for improvements for TWISTER.

6.1.1 Web Scraper

Creating a web scraper that would be able to scrape open data sites for datasets, would make it easier to convert large amounts of datasets. Furthermore it would enable

TWISTER to gather more metadata about each dataset, as there would most likely be more data about the dataset on the website.

`Opendata.dk` for example includes a description of the dataset alongside keywords and information about the author.

6.1.2 Labeling

The current labeling procedure for custom labels, assigns a confidence of 100 % to attributes if a target label is found directly in the name. If a synonym of the target label is found it is assigned a confidence of 50 %. A way to improve this method is to use a thesaurus that has a distance function in terms of word similarity, and use the distance to get a more accurate confidence in the label. This means the distance from two words that are synonyms becomes part of the outcome and can be used in the classification component of TWISTER.

Another interesting extension to the labeling process would be to include more information in the labeling process. By taking data types into account for custom labels the labeler could potentially rule out some labels if the attributes are not of the correct type. Another type of information that could benefit the labeling process for custom labels is information obtained from nested `ObjectAttributes`. The idea would then be to make requirements to what `ObjectAttributes` are needed to assign the labels. Just as with data types, this new information could also potentially prevent incorrect assigned labels.

In the current implementation TWISTER does not distinguish between two `ObjectAttributes` that both contain the same requirement for a given label. An example would be "Count" and "Count_SomethingElse", when the target is "Count". A part of this problem could be solved by analyzing the string in more detail and adjusting the confidence for the label based on the analysis.

6.1.3 Transformation of Input

The labeler uses the structure of each intermediate object's attributes to decide what geographical label should be added to a given attribute. The same approach can be used to label attributes like addresses. In order to do this, it will be useful to split the attributes into as many sub attributes as possible.

For instance if an attribute contains an address it should be changed to `ListAttribute` with a sub attribute for country, city, street name, and street number.

6.1.4 Nested Attributes in Output

The implementation of the output object is not able to represent objects where the attributes represent collections or nested objects. For an example if a building were to be represented as an object, that building could have rooms as attributes which would include their own attributes, and so on. Extending this to be able to represent these types of attributes, would improve the flexibility of the pipeline and enable more complex data retrieval operations.

This however would also require an extension to the output generator, as it is currently only able to retrieve single attributes. It would need to be extended to be able to retrieve collections and nested objects.

This could be implemented in the same way as the specialized sub-component which is in charge of retrieving the geometric feature from the intermediate object. In this way the generator would switch on the type of attribute and then choose the appropriate sub-component to handle the retrieval of the data.

6.1.5 Inferring Coordinate Reference System

When working with geographical data, making sure that the coordinates that make up the geometry is correctly read is very important. For this reason the old version of GeoJSON specification specifies that this information must be present, and the newest version specifies that all coordinates must be encoded using WGS 84[11].

Working with data formats where this information is not present, as with CSV where this information is only present if the data provider added it to the source. This means that it might not be present in all formats or might be difficult to find. Therefore being able to infer the CRS from only the coordinates, would make the pipeline more resilient and result in better quality data.

7 Conclusion

The tool TWISTER that was developed for this paper was meant to improve the accessibility of data on open data portals. In section 6 it is discussed that the results presented in section 5, show that the tool is indeed able to improve the accessibility for a number of cases. However it is also shown that TWISTER has a number of cases where it is unable to provide valuable results. Some of the cases that lead to incorrect results can be corrected by introducing more features and improving already existing features. However it has also been found that datasets with sparse information in some cases simply cannot be classified and parsed correctly. A dataset with this sparse property could be one that only contains coordinates, where there is no information about the Coordinate Reference System anywhere in the data. This is problematic as there exists many different CRSs that potentially could be the right one, and picking the wrong one will result in a completely different and incorrect translation.

References

[1] European Commission. *Open data portals | Shaping Europe's digital future*. https://digital-strategy.ec.europa.eu/en/policies/open-data-portals, 2022. Accessed: 10-06-2022.

[2] Sean MacAvaney, Andrew Yates, Sergey Feldman, Doug Downey, Arman Cohan, and Nazli Goharian. Simplified data wrangling with ir_datasets. In Fernando Diaz, Chirag Shah, Torsten Suel, Pablo Castells, Rosie Jones, and Tetsuya Sakai, editors, *SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021*, pages 2429–2436. ACM, 2021. doi:10.1145/3404835.3463254. URL https://doi.org/10.1145/3404835.3463254.

[3] Martin Koehler, Alex Bogatu, Cristina Civili, Nikolaos Konstantinou, Edward Abel, Alvaro A. A. Fernandes, John A. Keane, Leonid Libkin, and Norman W. Paton. Data context informed data wrangling. In Jian-Yun Nie, Zoran Obradovic, Toyotaro Suzumura, Rumi Ghosh, Raghunath Nambiar, Chonggang Wang, Hui Zang, Ricardo Baeza-Yates, Xiaohua Hu, Jeremy Kepner, Alfredo Cuzzocrea, Jian Tang, and Masashi Toyoda, editors, *2017 IEEE International Conference on Big Data (IEEE BigData 2017), Boston, MA, USA, December 11-14, 2017*, pages 956–963. IEEE Computer Society, 2017. doi:10.1109/BigData.2017.8258015. URL https://doi.org/10.1109/BigData.2017.8258015.

[4] Ignacio G. Terrizzano, Peter M. Schwarz, Mary Roth, and John E. Colino. Data wrangling: The challenging yourney from the wild to the lake. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015. URL http://cidrdb.org/cidr2015/Papers/CIDR15_Paper2.pdf.

[5] Microsoft. *Double.TryParse Method*. https://docs.microsoft.com/en-us/dotnet/api/system.double.tryparse?view=net-6.0. Visited: 20/05/2022).

[6] U.S. Geological Survey. The universal transverse mercator (utm) grid. 077-01:2, 2001.

[7] epsg.io. *EPSG:23032*. https://epsg.io/23032-8653, . Visited: 17/05/2022).

[8] epsg.io. *EPSG:32633*. https://epsg.io/32633, . Visited: 17/05/2022).

[9] Agency for Data Supply and Efficiency (SDFE). Utm/etrs89: Den primære kortprojektion i danmark. First:6, 2017.

[10] Defense Technical Information Center. Department of defense world geodetic system 1984: Its definition and relationships with local geodetic systems. second edition. Second Edition:169, 1991.

[11] H. Butler, M. Daly, A. Doyle, Sean Gillies, T. Schaub, and Stefan Hagen. The GeoJSON Format. RFC 7946, August 2016. URL https://www.rfc-editor.org/info/rfc7946.