De-Identification Tool for Clinical Reports

Marc Zimmermann[1], Katie Kalt[2], Patrick Hirschi[2], and Gunnar Rätsch[1,2]

[1] Biomedical Informatics Group, Department of Computer Science, ETH Zurich, Zurich, Switzerland

[2] Department Research and Teaching, University Hospital Zurich, Zurich, Switzerland

October 2021

Abstract

This document outlines the development and implementation of a robust software tool designed to de-identify clinical reports to address privacy regulations and policies. The tool employs advanced natural language processing techniques to accurately identify and anonymize personal health information (PHI) from clinical texts, making them suitable for further research and analysis without compromising patient confidentiality. Key components of the software include a customizable annotation pipeline, extensive lexica for precise entity recognition, and sophisticated algorithms for sensitive data detection and substitution. The document also details the software's architecture, setup requirements, usage guidelines, and performance metrics, supported by a case study on its application in a real-world healthcare setting. This comprehensive approach not only supports using the tool to meet regulatory requirements but also adapts efficiently to varied data formats and clinical environments.

Contents

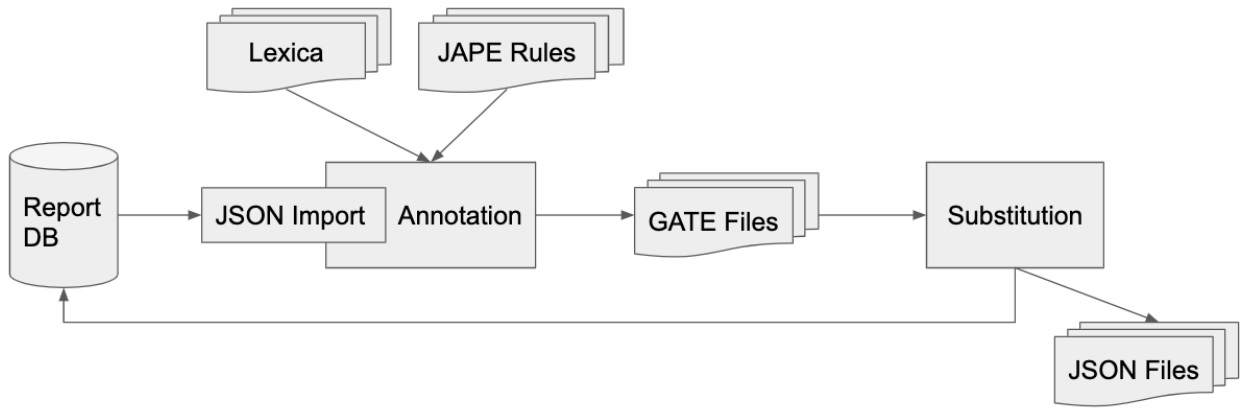


Figure 1: System overview

1 Introduction

The deidentification tool consists of a collection of command line applications written in Java. The applications is based on the GATE framework

Here, a short overview over each command line application. Each square box represents a different command described in the following sections.

1.1 Document Import

The deidentifier tool imports JSON reports typically from a database (json reports on the filesystem are also supported) and converts them to a GATE compatible representation. The `annotate` command can read directly from the appropriate source. The `import` command only does the import and conversion step and stores a batch of documents into a GATE corpus (which is a directory on a filesystem).

It is assumed that the documents are stored in a database table or view, one row per document. The actual report content is encoded as JSON string in one of the columns. Other required columns denote the document type (`FCODE`) as well as the report id.

The tree like structure of the JSON documents is preserved during the conversion to the GATE compatible representation. This can be exploited during the annotation.

1.2 annotate

The `annotate` command takes documents from a GATE corpus and runs an annotation pipeline over the reports, i.e. annotates portions of the text which contain entities to be deidentified. The output of this process is again a GATE corpus, which can be examined e.g. using the GATE developer tool.

An *annotation* simply denotes a span of text with some properties associated, for example: “Mr. Muster, born 01.01.1964 in Aarau” could have 3 annotations related to deidentification: one for ‘Muster’ (Name), another for ‘01.01.1964’ (Date, with the additional information that it is a birthdate) and ‘Aarau’ (Location).

Currently, the following entities are annotated: * Age * Contact (distinguishing phone numbers, email and websites) * Date (if possible determining birth date, admission date, discharge date) * ID (patient or case ID, social security or insurance numbers) * Name (if possible distinguishing patient from staff) * Location (broad category containing geographical locations as well as organizations) * Occupation

The annotation pipeline consists basically of the following consecutive steps:

1. **Tokenization**: splitting the text into units of characters (‘words’), for example ‘Mr. Muster’ would be split into 3 tokens ‘Mr’, ‘.’ and ‘Muster’

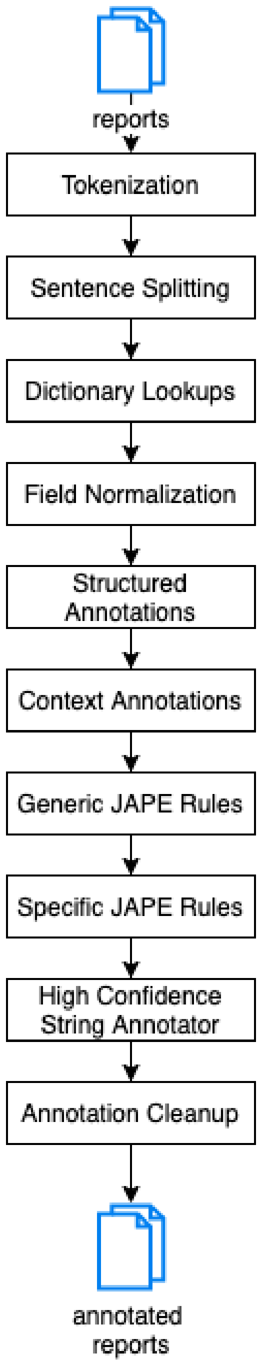


Figure 2: Pipeline steps

2. **Sentence Splitting**: Grouping tokens together into sentences
3. **Lookups**: Annotating tokens using dictionaries/lexica (called "Gazetteers" in GATE). For instance, the tokens 'Universitätsspital Zürich' could be annotated as "organsation" and the token "Zürich" as "location"
4. **Field Normalization**: Renaming some fields in the report to a common name to simplify downstream steps. E.g. `AddrLine`, `Address`, `Adresse` could all be renamed to `AddressField`.
5. **Structured Annotation**: annotate entire fields known to contain information to be deidentified. For example, the content of a field `Tel` denoting a phone number could immediately be annotated without having to apply any further processing.
6. **Context Annotation**: annotate a window (of tokens) around some trigger token with a context annotation. like `NameContext` or `OccupationContext` to help JAPE rules to disambiguate between e.g. surnames and professions.
7. **JAPE rules**: annotate tokens based on a regular expression-like language. This is based on the structure or content of tokens (e.g. a specific trigger word such as "Dr" or being a number) as well as previous annotations from dictionaries from the previous step. Note, that rules can also exploit the tree-like structure of the document, for example a rule may only apply if the token in a field is part of the section of the document related to patient information.
8. **High Confidence String Annotator**: Some JAPE rules in the previous step can be marked as `high confidence`, that is, whatever these rules annotate, we have high confidence that it is correct. The `HighConfidenceStringAnnotator` then checks what tokens were annotated by a high confidence rule and then annotates the same tokens in the remaining document.
9. **Clean up**: resolve overlapping or conflicting annotations using heuristics.

1.2.1 JAPE Example

Here an example of a JAPE rule. We are interested in recognizing ages in a very specific pattern, namely the age followed by "jährige", "jähriger" or "jährigen", e.g. "59-jähriger Patient":

```
// 0-119 (including decimals)
Macro: POSSIBLE_AGE
(
    ({Token.string =~ "[1-9]*[0-9]"} | {Token.string =~ "1[0-1][0-9]"})
    ({Token.string == "."} {Token.string =~ "[0-9]+"})?
)


Rule: AgeRightContextTrigger
(
   (POSSIBLE_AGE):age
   ({Token.string == "-"})?
   ({Token.string =~ "jährige[rn]?"})
)
-->
:age.Age = {rule = "AgeRightContextTrigger"}
```

Rules describe a sequence of tokens on the "left hand side", i.e. before "–>". If such a sequence is recognized in a text a rule is triggered and the "right hand side" is applied. In the above case, the right hand side adds an annotation of type `Age` having as a property the name of the rule triggered (this helps debugging).

A token can be described exactly, like {`Token.string == "-"`} where the token should consist exactly of `-` or using regular expressions as in {`Token.string =~ "1[0-1][0-9]"`} describing the numbers from 100 to 119. A sequence of tokens may contain optional elements denoted with `?`. In the above example the (`{Token.string == "-"})?` signifies, that there may or may not be a dash.

More details regarding JAPE can be found in the JAPE Grammar Tutorial

Note, that the above example is not robust against typos, e.g. the rule would fail to annotate the age in "59-järiger Patient".

1.3 substitute

Takes an annotated GATE corpus and generates a JSON representation of the content with to be deidentified tokens replaced. The JSON version of these reports can then be saved back to a database table or to JSON files on a drive one per report.

1.3.1 Substitution Policies

There are several policies implemented on how annotated tokens should be replaced:

- `ScrubberSubstitution`: entities are replaced by a fixed string depending on the annotation type, for example 'am 01.02.2003' would be replaced as "am DATE" and 'Dr. Muster' by 'Dr. NAME'
- `DateShift`: same as `ScrubberSubstitution`, but all dates of a report are shifted by a random amount of days into the future or past.
- `ReplacementTags`: In this policy information are passed along to a downstream application which takes care of the actual deidentification. For that purpose entities are replaced by 'tags' which contain as much information as possible from the annotation pipeline. For example a text like "Dr. P. Muster empfiehlt" could be replaced by `Dr. [[[Name;P. Muster;firstname=P;lastname=Muster;type=medical staff]]] empfiehlt` that is, the original value is preserved and the downstream application can decide how to replace the name most appropriately.

There exist also the `--fields-blacklist` option, where a list of field names can be provided which are completely erased from the document. This can be useful for fields with are notoriously hard to deidentify, but contain no relevant information for a downstream application.

2 System Setup and Requirements

2.1 Installation

2.1.1 Prerequisites

- at least Java 8
- appropriate JDBC driver, if reports are loaded from database (PostgreSQL drivers are already included)

2.1.2 Getting the Software

Download the latest release from the releases page, that is, the jar file in the `Assets` section.

Alternatively (more advanced), you can download a recent zip archive generated everytime the github action workflows are triggered. You can look for workflow runs of the branch "main" and look for the "Artifacts" section of a specific run.

Building Yourself You need'll need Maven to build from source code. Once `maven` is set up, you can run the following in the root directory of the repository

```
mvn package --file deidentifier-pipeline/pom.xml
```

You'll then find the `jar` file containing the pipeline code as well as its dependencies in `deidentifier-pipeline/target`

2.1.3 Basic Usage Example

In the following, a small example to deidentify a few JSON files.

First, create a directory `orig_reports` and populate it with an example file and/or create your own JSON files to put there.

The basic invocation of the deidentification tool is

```
java -jar [path to deidentifier-VERSION.jar]
```

In the following we abbreviate this by `DEID_CMD` (on a shell you could e.g. run `DEID_CMD="java -jar deidentifier-1.0.jar"`).

To annotate terms which need to be deidentified in the documents, run

```
$DEID_CMD annotate -i orig_reports --json-input -o annotated_reports -c configs/kisim-usz/kisim_usz.con
```

You may have to adapt the path to the kisim_usz.conf file. The output in `annotated_reports` can be opened and inspected using GATE Developer, the graphical user interface of the GATE framework.

The annotations can now be replaced and written out to disk again.

```
$DEID_CMD substitute -o substituted_reports --method Scrubber annotated_reps
```

The substituted reports can now be found in `substituted_reports`, where annotated terms are replaced by a fixed string (more about this in the Section 1.3.1.

Adapt Database Configuration In case reports should be read from a database instead of a directory, a configuration file needs to be created specifying host, username, password table etc. You can find more details in Section 4.2.1.

In the following we'll assume the file is called `db_conf.txt`.

Here an example, how the file could look like:

```
jdbc_url=jdbc:sqlserver://myhost:2345;databaseName=MyDB;
user=deid_poc
password=1asdffea
query=SELECT DAT,FALLNR,CONTENT,FCODE,REPORTNR FROM MyDB.KISIM_KIS_T_REPORT_JSON.KIS_T_REPORT_JSON
json_field_name=CONTENT
reportid_field_name=REPORTNR
report_type_id_name=FCODE
date_field_name=DAT

# for writing back
dest_table=subst_test
dest_columns=CONTENT,REPORTNR,FCODE,DAT,FALLNR
```

Reports are read from the `KISIM_KIS_T_REPORT_JSON` table of the `MyDB` mssql database. Substituted reports are written back into the table `subst_test` into the column `CONTENT` and the columns `REPORTNR,FCODE,DAT,FALLNR` are just copied over as is.

For a postgres DB the JDBC URL would start with `jdbc:postgresql://...`. Other databases are supported in principle, but the corresponding JDBC driver needs to be made available on the java classpath.

To annotate reports from a database table, the command would become:

```
$DEID_CMD annotate -d db_conf.txt -o annotated_reports -c configs/kisim-usz/kisim_usz.conf
```

The `annotate` command provides some basic mean to select appropriate reports via the options `--max-docs`, `--skip-docs`, `--doc-id-filter` and `--doc-type-filter` (see `$DEID_CMD annotate --help` for more details) More complex filtering could be done by tweaking the `query` field in `db_conf.txt`.

To substitute the annotated reports and write them back into another database table, the command would be:

```
$DEID_CMD substitute -d db_conf.txt --method Scrubber annotated_reps
```

2.1.4 Further Tips for Running the Deidentifier Tool

File Encoding If you run into encoding related issues, try adding `-Dfile.encoding=UTF-8` into your `DEID_CMD`, e.g

```
java -Dfile.encoding=UTF-8 -jar deidentifier-*.jar
```

Increasing Memory If the tool crashes with e.g. an `OutOfMemoryError` or the processing is very slow, try increasing the memory the Java virtual machine (jvm) is allowed to use using the `-Xmx` option, e.g.

```
java -Xmx4g -jar deidentifier-*.jar
```

Here, in total at most 4g of RAM would be used.

Customize Logging We use `log4j2` to manage logs. You can provide a custom `log4j` config by passing `-Dlog4j.configurationFile=[path to log config]` to the java command. The default logging configuration can be found here. See the log4j documentation for more infos.

To debug the logging setup, you can add the `-Dlog4j.debug` flag to the java command.

Adding JDBC Drivers To connect to a database other than Postgres, you need to download an appropriate JDBC driver (make sure it is compatible with both the java and database version you are using).

The command line invocation (the `DEID_CMD`) becomes then

```
java -cp "deidentifier-pipeline/target/deidentifier-0.99.jar;[path to jdbc jar]" org.ratschlab.deidenti
```

3 Architectural Overview

A brief overview over the software architecture and design of the tool, s.t. it becomes easier to navigate the code base and understand how the different parts work together.

The tool is a command line application written in Java 8. Its most important dependency is the GATE NLP framework.

3.1 Package Overview

Largest part of the Java code are in the package `org.ratschlab.deidentifier`: * `annotation`: Language analyzers (steps in the annnotation pipeline) * `dev`: Small tools for sanity checks during development. Not used in production * `pipelines`: Constructing and testing GATE pipelines for deidentifiation * `sources`: Handling sources of reports. Currently only reports from KISIM in JSON format * `substitution`: implementation of different strategies to substitute annotated tokens * `utils`: common utility functions * `workflows`:

3.2 NLP Pipeline Implementation

Annotating tokens to be deidentified happens over several steps forming a pipeline. A GATE annotation pipeline (`SerialAnalyserController`) consists of sequence of `LanguageAnalysers`. A document is passed along every analyser and modified accordingly (e.g. by adding certain annotations). Note, that many concepts in GATE are represented by annotations, including tokens and sentences.

The pipelines are constructed in a `PipelineFactory` in the `org.ratschlab.deidentifier.pipelines` package. It adds analyser steps based on the pipeline configuration. Many analysers could be reused from GATE or GATE plugins, notably tokenization, sentence splitting and lexica (gazeteer) annotations and JAPE rule engine.

3.3 Aspects

3.3.1 Configuration

Pipeline configurations are managed using a configuration file in HOCON format. A pipeline configuration file include paths to relevant files containing lexica, specific JAPE rules, field lists for structured annotation etc

3.3.2 Parallelization

Documents can be processed in parallel by having several instances of the annotation (or substitution) pipeline running in parallel.

This is managed by an Akka Stream compute flow/graph, which consist of the following parts: * reading a document from a source (file or DB) and convert it to a GATE document structure * distributing the document to one of the annotation pipeline instance and annotate it * post process single doc: e.g. write to database or file * post process corpus: e.g. write doc stats to a single file, evaluate corpus

All these steps happen in a concurrent and asynchronic manner.

3.3.3 Testing

Code is tested using tests written in JUnit. Additionally, there is a testing framework to test annotations. See `Testcases` section in the components.md.

4 Key Components and Configurations

4.1 Annotation Pipeline Components and theirs Configurations

The deidentification tool was designed to be relatively flexible to accommodate various needs.

4.2 Data Input

More details on the configuration of the report input data source.

4.2.1 DB Configuration

Configuration parameters to load reports from a database are stored in a text file (properties file) with the following attributes (example config in Section 2.1.3):

- `jdbc_url`: the URL to connect to the database, see also
- `user`: the database user name
- `password`: the password
- `query`: a “SELECT” SQL query. Can be anything (e.g. joining data from several tables, a view . . .) as long as certain columns are present

- `json_field_name`: the column name in the above SQL query denoting the JSON content of a report
- `reportid_field_name`: the column name denoting the reportid of a report in the SQL query
- `report_type_id`: the column name in the above query denoting the report type ("FCODE") of a report. This is mainly used to select certain types of reports.
- `date_field_name`: the column name in the above query denoting the creation date of a report (optional).

In case reports should be written back to a database after substitution: * `dest_table`: table name to write into * `dest_columns`: the names of the columns to write back. These should be a subset of the columns in the above SELECT SQL query (could also be all of them)

Report Filtering

Document Type Filter Depending on the project, only certain document types ("FCODE") might be relevant. These could be filtered out in the SQL query or also using a simple text file which can be passed to the `annotate` or `import` command using the `--doc-type-filter` option.

The file contains one row per document type and at least one column (seperated by ','), where the first column denotes the document type name. There can be more columns (for example human readable description), which are ignored by the application.

Document ID Filter Similar to document type filters, one can specify to load only documents having a specific ID. This can be done by passing a file path using the `--doc-id-filter` option.

Columns: * report id

4.3 Annotation Pipeline

Quite a few aspects of the annotation pipeline can be parameterized. In this section, more details about various annotation steps and their configuration.

4.3.1 Pipeline Configuration File

Many pipeline steps can be parametrized by specific configuration files or other parameters. The parametrization happens via a configuration file setting all relevant parameters for the annotation pipeline. You can pass the path of the file to the `annotate` command using `-c`. The syntax of the file follows the HOCON format, see the configuration of the USZ pipeline as example

Configurations relevant to the pipeline are grouped together into the `pipeline` 'section'.

4.3.2 Lexica (Dictionaries, Gazetteers)

The `pipeline.gazetteer` option should point to a GATE gazetteer file definition (`*.def`). This text file contains an entry for each dictionary file with the (relative) path and the annotation type. A dictionary file is simply a text file with one or more token per line. More details in the GATE Documentation

The annotation pipeline also uses a second category of gazetteers specified in `pipeline.suffixGazeteer` not matching entire tokens but suffixes. This is useful for rules based on word endings, for example to recognize surnames ("-mann", "-oulos", "-elli") and medical terms ("-karzinom", "-suffizienz", "-beschwerden").

4.3.3 Specific JAPE Rules

There is a generic set of JAPE rules shipped with the application. Typically, these rules cannot cover special cases appearing in a given organization. This can be done using a separate rule set.

Specific rules can be added via the `pipeline.specificTransducer` option pointing to a `*.jape` file. This file would contain a list of different phases, where every phase is a separate `*.jape` file. These files would then contain the actual JAPE rules. See also the Gate Documentation.

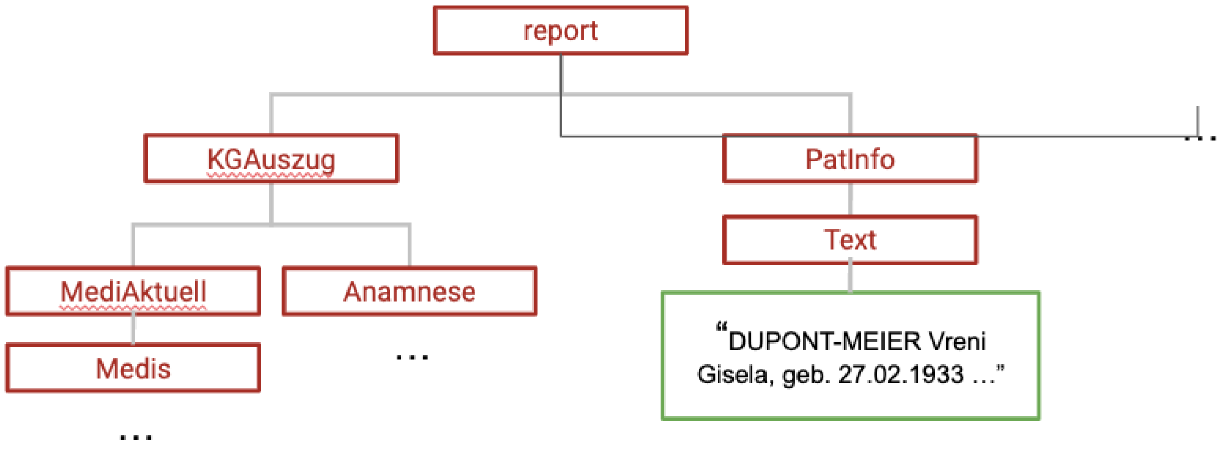


Figure 3: simple report structure

4.3.4 Report Structure

If available, the structure of input documents can be exploited during the annotation (and in principle also during substitution). That is, prior knowledge about the document can be added.

In case of JSON, the structure elements would be field names and nested objects. A JSON document can then be seen as a tree where the leaves contain the actual report text fragments.

Paths in Field Tree In the various configuration files related to annotations, paths in the "field tree" can be used to denote certain parts of the document (similar to XPath for XML documents).

A **path** can consist of the following elements: * field name: can be a regular expression accepted by Java * '/' denoting that the nodes must appear consecutively, e.g. `field_a/field_c` matches only "field_a/field_c" but not "field_a/field_b/field_c" . * '//' denoting that the nodes don't need to be necessarily consecutive, e.g. `field_a/field_c` would match both "field_a/field_c" and "field_a/field_b/field_c"

Note, that paths are case-sensitive.

Examples: Simple field names which can appear anywhere in the tree: * `Id` * `ID` * some regular expression `[\\p{IsAlphabetic}]*VisDat` ("
p" is used to match unicode characters). * Field names with some constraints regarding the parent. For instance, to match `Text` fields only when it is a child of `PatInfo`, you can use `//PatInfo//Text`, which would match e.g. "/report/PatInfo/some_element/Text" * Field constraints "anchored" from the top: `/NOTE` would match a field "NOTE" directly under the root of a tree, but not "/PatientInfo/NOTE".

Structured Annotations The "structured annotation step" allows annotating entire text fields. For instance, if you know that a certain field contains the name of a person, then an annotation can be performed at that level.

This step can be parametrized with a configuration file passed in `pipeline.structuredFieldMapping` in the pipeline configuration.

Columns in the config file (separated by ";"): * Path * Annotation type (e.g. `Name`, `Date` etc) * Features (properties) of the annotation . Features are separated by "," where key and values are separated by "="

Example: `//Personalien//Name; Name; type=patient`

All leaves with `Name` having `Personalien` as a parent somewhere are annotated with `Name` and having the `type` property set to `patient`.

Field Normalization (Field Annotation Mapping) Sometimes, we can not blindly annotate an entire field, but need to apply a JAPE rule on it. For example, signatures could have a structure like "ABCD, 20.10.2018" where "ABCD" is the shorthand for a doctor. Since there are many fields with similar or identical structure, but different paths the fields can get renamed to a common name. A JAPE rule processing the pattern would then refer to that common name.

This step can be parametrized with a configuration file passed in `pipeline.annotationMapping` in the pipeline configuration. Columns (separated by ";"): * Path * New field name

Example: `//Patient/Alter/Val; AgeField`

A field `Val` with immediate ancestors `Alter` and `Patient` gets named an `AgeField`. Now a JAPE rule only working on `AgeField`s, could for example annotate any number in there as an age.

Annotation Blacklist For some fields we can exclude a priori certain annotations. An example could be a field containing computer generated identifiers like `9b02d92c-c16e-4d71-2019-280237bb8cb5` where a JAPE rule may erroneously pick up a date (for example "2019" in the example). A blacklisting step would remove such annotations.

This step can be parametrized with a configuration file passed in `pipeline.annotationBlacklist` in the pipeline configuration.

Columns (seperated by ";"): * Path * Comma separated annotation types which should *not* appear within elements denoted in path

Example: `//DiagnList//CodeList//Version; Date`

The `Version` field having `CodeList` and `DiagnList` as parents should not contain `Date` annotations.

4.3.5 Context Annotations

There are some JAPE rules which only get triggered if tokens appear in a specific language context. This can be useful to disambiguate between e.g. surnames and professions or between the profession of a patient vs the role of staff.

The context can be given by a field (using Annotation Mappings) or by using **context annotations**. They can be added around trigger tokens. For instance, text in the vicinity of `Sohn` (son) may contain his name or information about his profession. Therefore, `NameContext` and `OccupationContext` context annotations are added to the document, spanning the e.g. 5 tokens before `Sohn` and 5 tokens after. Later on, if within these context annotations e.g. an isolated first name appears, it can be annotated as `Name` since we assume it is a context where names can occur (otherwise we wouldn't annotate it, as there is not enough evidence). Context annotations are performed in early stages of the pipeline, s.t. they can be referred to in JAPE rules later on.

These context triggers can be configured in a config file whose path has to be provided in `pipeline.contextTriggerConfig` in the pipeline configuration.

Columns (separated by ";"): * Context token (no spaces) * Name of the context (e.g. `NameContext`) * start of the context annotation in number of tokens before the trigger token * end of the context annotation in number of tokens after the trigger token

Examples: * `Sohn;NameContext;5;5` * `Partner;OccupationContext;5;5`

4.4 Test Suite

A small testing framework was developed to test the annotation behavior of the pipeline in a fast and isolated way. That is, small test cases can be defined consisting of a phrase and the annotations the pipeline is expected to produce. This allows for test driven development/tuning of the annotation pipeline.

4.4.1 Test Cases Specification

The testcases are described in a textfile. The first line of the text file contains the annotation types the pipeline is tested against as well as the context fields. The context fields annotations are used to test rules based on the document structure. The lists for annotation types and context fields are seperated by ; and the entries in lists by ,.

Then testcases follow, one per line. Manual annotations and fields are added using XML-tags. Comments using '#' are allowed either to comment entire lines are the remainder of a line. Commented parts are ignored. There may be empty lines for making the file a bit more readable. If new lines are needed to test a specific situation, this can simply be done using \n.

Following an example with 3 test cases:

```
Name; FieldWithSignature

Der Patient <Name>Luigi D'Ambrosio</Name> wurde...

<FieldWithSignature>20.01.2018 / <Name>AMUSTER</Name> </FieldWithSignature>
20.01.2018 / AMUSTER # don't expect name annotation in arbitrary fields
```

In this example, the annotation of `Name` is tested. The <`Name`> tags are removed before the test case is passed through the pipeline. Then, the `Name` annotations of the pipeline output are checked whether they indeed contain `Name` annotations at the same place, and only there. If this is the case, the test passes, otherwise it fails with an appropriate message.

The second test case tests a context specific rule, i.e. the rule is only applied within fields `FieldWithSignature` (In the USZ pipeline, `FieldWithSignature` annotation is added the annotation mapping step, see above) The third test case is just to see, if the previous rule is not triggered outside the required context. Or said differently, we expect `AMUSTER` not to be annotated, i.e. annotating it would be wrong.

In some existing tests there is also the `OUTOFVOC` token. It stands for "out of vocabulary" and makes it explicit, that the rule should rely exclusively on structure, and not be based on entries in the dictionary.

Running Tests A test suite can be run using the `test` command from the `DeidMain` entry point:

```
$DEID_CMD test [pipeline configuration file] [testcase directory]
```

It commands needs a path to a pipeline configuration file (e.g. `configs/kisim-usz/kisim_usz.conf`) and a directory with testcases (e.g. `configs/kisim-usz/testcases/`). Every `*.txt` in that directory is assumed to contain test cases.

The generic rules shipped with the tool are tested using the same mechanism. They are run as unit tests for the tool itself. The test cases can be found in the directory `deidentifier-pipeline/src/test/resources/pipeline_tes` You normally don't need to modify these tests while tuning the pipeline, but you may consider them a source of useful examples.

5 Dictionary and Lexicon Assets

5.1 Lexica

In GATE, a lexicon (or gazetteer) consists of a text file with one term per line. A term may contain spaces. The terms are treated as case-sensitive.

The lexica compiled for the USZ pipeline have many different sources. Below tables describing the different lexica along with their provenance. Note, that some lexica cannot be published, as they contain hospital internal data. Note, that `ANNIE` refers to a GATE plugin, `GeoNames` to the geographical database which can be found here `https://www.geonames.org/`.

5.1.1 General

File Name	Source	Description
abbreviations_stop.lst	ANNIE German	Abbreviations like `z.B`
general_wordlist_with_uppercased.lst	Aspell dictionary	General wordlist
stop.lst	ANNIE German	Stopwords

5.1.2 Locations

Geographical

File Name	Source	Description
additional_locations.lst	Manually edited	
canton_names.lst	GeoNames and manually added	"Uri"
cantons_abbrevs.lst	GeoNames and manually added	"ZH", "BE"
citizenships.lst	Wikipedia and manually added	"Schweizer", "Deutsche"
city.lst	ANNIE	various cities (worldwide)
city_ambiguous_manual.lst	Manually edited	Cities which typically have another meaning in the context of medical reports, e.g. `Wangen`, `Füssen`. No Location annotations are performed.
city_derived.lst	ANNIE	
city_german.lst	ANNIE	
city_switzerland.lst	Swisstopo Ortschaften-verzeichnis	
country.lst	ANNIE	
country_adjectives_german.lst	Wikipedia and manually added	"Italienisch", "Italienischer"
country_german.lst	ANNIE	
country_german_wiki.lst	Wikipedia	Country Names
country_iso_codes.lst	Manual	
country_manual.lst	Manual	Contains countries not existing anymore
country_regions_german_wiki.lst	Wikipedia	"Norditalien"
languages_manual.lst	Manual	
larger_cities.lst	GeoNames	Larger international cities, "Tripoli", "Hannover"
location_false_positives.lst	Manual	Typically medical terms containing location as a part (not annotated)
province.lst	ANNIE	
regions.lst	Manual	
streetnames.lst	Manual	Streets or similar not matching a typical pattern
toponyms_switzerland.lst	GeoNames	All sorts of topopynms. Extensive blacklist was needed for ambiguous locations
toponyms_switzerland_manual.lst	Manual	More Swiss Toponyms

Organisational

File Name	Source	Description
buildings_usz.lst	USZ (KISIM)	Building abbreviations at USZ
hospitals.lst	Manual	Hospital names
institutions.lst	Manual	Institutions related to USZ
organisational_units_usz.lst	USZ (KISIM)	Abbreviations of organisational units
related_organisations_usz.lst	USZ (KISIM)	Institutions related to USZ. Internal only.

5.1.3 Medical

Including information about medical terms into the pipeline is mainly to avoid an annotation on it, typically for surnames.

File Name	Source	Description
drugs_usz.lst	USZ (KISIM)	
medical_mesh_terms.lst	MESH 2019 German Translation	https://www.dimdi.de/dynamic/en/classifications/further-classifications-and-standards/mesh/
medical_terms_de.lst	Wikipedia	
medical_terms_manual.lst	Manual	

5.1.4 Occupations

Professions and companies a patient might work for.

File Name	Source	Description
company_list_ch.lst	Wikipedia	
generic_occupations.lst	Manual	More for testing purposes
occupations_usz.lst	USZ Kisim	Processed list of professions entered in KISIM. Internal only.

5.1.5 Person Names

A few lexica are partitioned into two parts with "frequent" and "seldom" names. The distinction is used by some rules as indication whether a token might likely be a name or not.

File Name	Source	Description
firstnames_switzerland_frequent.lst	Bundesamt für Statistik: Vornamen in der Schweiz 2017	
firstnames_switzerland_seldom.lst	Bundesamt für Statistik: Vornamen in der Schweiz 2017	
firstnames_usz_frequent.lst	USZ (KISIM)	
name_false_positives.lst	Manual	Often medical terms
surnames_usz_frequent.lst	USZ (KISIM)	
firstnames_usz_seldom.lst	USZ (KISIM)	Internal only
surnames_staff_usz.lst	USZ (KISIM)	Internal only
surnames_usz_seldom.lst	USZ (KISIM)	Internal only

5.1.6 Suffix Lists

Instead of using lexica annotating terms 1:1 in the text, a part of the pipeline annotates tokens based on suffixes. This is useful for missing terms in the other dictionaries. For example `Nasenerkrankung` would still be recognized as medical term, even though it might not be in any medical dictionary.

File Name	Source	Description
medical_suffixes.lst	Manual	`erkrankung`,`geräusch`
surname_suffixes.lst	Manual	`mann`,`oulos`

6 Software Development and Updates

For more details on the code structure here

6.1 Releases

Releases can be done via github actions and the built jar and users can download it on the github release page

1. Adapt the version in the <`version`> section of maven config
2. merge this change into main
3. create a tag locally on your computer using `git tag v[version]`, e.g. `git tag v1.0.0`
4. push tag using `git push origin v1.0.0` (adapt to the version)
5. If the github actions workflow ran successfully, you can edit the release as necessary on the github release page and switch it from Pre-release to release (remove the flag `This is a pre-release`)

Note, that there is a danger that the version in `pom.xml` and in the git tag don't match. This could/should be streamlined in the future.

7 Data Processing and Workflow Structuring

7.1 Diagnosis Extraction Pipeline

This document briefly describes a pipeline to extract diagnosis out of a medical report along with a (rough) estimate on its reliability (confirmed, suspected, excluded).

7.1.1 Approach

The current implementation is mainly based on keyword matching and is hence very basic. An initial evaluation shows that the diagnosis are extracted well, however, the reliability of the diagnosis are not detected very reliably as the language around suspected or excluded cases can be involved. Because of way the pipeline is currently implemented, there is a bias towards 'confirmed' diagnosis, i.e the pipeline may label a diagnosis as `confirmed` whereas the actual diagnosis was merely a suspection or even an exclusion.

7.1.2 Pipeline Overview

The pipeline is based on the same framework as the deidentification pipeline and hence shares many commonalities.

The input is the same as for the `annotation` tool i.e. reports out of KISIM either directly from a database or already imported as GATE documents.

The pipeline generates for every diagnosis the following output:

- document ID/report ID
- annotation text found related to diagnosis (mainly for debugging)
- code (ICD-10)
- reliability of diagnosis: confirmed, suspected, excluded

Note, that per report several diagnosis may be extracted. Furthermore, the reliabilities may be conflicting, that is, in a report there may be the same diagnosis twice with different reliabilities. Depending on the use case, the downstream system need to resolve this "conflict".

These fields can be written back to a database table and/or written to a text file for further analysis in pandas/excel.

7.1.3 Configuration

Database Configuraiton The configuration related for reading and writing into the database are the same as with the deidentification pipeline. There are just a few more configuration keys related to the naming of the fields generated by the diagnosis extraction pipeline:

```
annotationtext_field_name=annotationtext
reliability_field_name=reliability
code_field_name=code
```

These properties can be changed to match the destination table schema. The same keys should then be used in the `dest_columns` property.

For example:

```
dest_columns=reportnr,fcode,dat,fallnr,annotationtext,code,reliability
```

Pipeline Configuration Many aspects of the pipeline can be tweaked by editing text files.

Keywords Configuration For every ICD-10 code to be extracted there needs to be a few keywords/names for the condition. The keyword configuration file is a text file where a row corresponds to one ICD-10 code. The fields are separated by `;` and constituting the following:

- ICD-10 code, for example `G35`;
- keywords separated by `,`, for example `Multiple Sklerosis,Encephalitis disseminata`
- blacklist paths: paths in the document structure which should not be considered to search for the keyword: e.g. `Header,Fragestellung` (see also `Paths in Field Tree` in `components.md`)

Reliability Context Configuration In order to assess the reliability of a diagnosis the 'reliability context' of a diagnosis is determined. This is done in a very crude way by looking for keywords within the neighborhood of a diagnosis keyword. For instance `ausgeschlossen` somewhere close after a diagnosis term like `Multiple Sklerose` could mean that the diagnosis is excluded.

These keywords can be configured the in reliability context configuration file. This is a textfile with one keyword a line, where the fields are separated by `;`. The fields are as follows

- context keyword, e.g. `ausgeschlossen`
- context name: `ExclusionContext` or `SuspectionContext`
- left extend: number of tokens from the context keyword to the left, the context is valid
- right extend: number of tokens from the context keyword to the right

7.1.4 Invocation

The entry point of the pipeline is the `org.ratschlab.structuring.DiagnosisExtractionCmd` class. It has the following usage (note, that currently not all options are implemented).

```
Usage: <main class> [--json-input] [--xml-input]
                    [--doc-id-filter=<docIdFilterPath>]
                    [--doc-type-filter=<docTypeFilterPath>]
                    [--max-docs=<maxDocs>]
                    [--output-corpus-dir=<outputCorpusDir>]
                    [--skip-docs=<skipDocs>] -c=<pipelineConfigFile>
                    [-d=<databaseConfigPath>] [-i=<corpusInputDir>]
                    [-o=<outputFile>] [-t=<threads>]
      --doc-id-filter=<docIdFilterPath>
                             Path to file id list to consider
      --doc-type-filter=<docTypeFilterPath>
                             Path to file type list to consider
      --json-input           Assumes input dir consists of json files, one per
                               report (testing purposes)
      --max-docs=<maxDocs>   Maximum number of docs to process
      --output-corpus-dir=<outputCorpusDir>
                             Output GATE Corpus Dir
      --skip-docs=<skipDocs> Skipping number of docs (useful to just work on a slice
                               of the corpus)
      --xml-input            Assumes input dir consists of xml files, one per report
                               (testing purposes)
  -c=<pipelineConfigFile>    Config file
  -d=<databaseConfigPath>    DB config path
  -i=<corpusInputDir>        Input corpus dir
  -o=<outputFile>            Output Txt File
  -t=<threads>               Number of threads
```

7.1.5 Pipeline Description

The pipeline executes the following steps for every document:

1. tokenization of the import report
2. diagnosis keywords are annotated
3. reliability contexts are annotated
4. JAPE rules are run to
   - determine the reliability of a diagnosis by either checking whether it is within some reliability context or whether it matches a certain language pattern (e.g. `Verdacht auf` ...)
   - Removing some false positives like `some_diagnosis Sprechstunde` or `some_diagnosis Abklaerung`
5. Consolidate diagnosis anntoations: * adding `confirmed` reliability as default, if no other reliability could be determined * removing duplicates
6. Writing to file and/or database

8 Annotation Rules and Optimization Strategies

8.1 Rules Guide and Tuning

This section describes how the rules are set up as well as their current limitations. Also, some typical tuning scenarios are described. We assume the reader is familiar with the annotation pipeline components.

8.1.1 General Considerations

In general, we need to balance between annotating as many relevant tokens as possible not annotating "too much". That is, we'd like to have a high recall (few false negatives), but still maintain a reasonable precision (few false positives), otherwise the data quality of downstream applications may suffer.

Typically, annotations carry attributes to be able to trace back to the rule generating it. This is very useful for debugging.

There are also a few "negative" rules, i.e. rules which "consume" tokens but don't trigger annotations. This is typically to avoid false positives. An example are citations like `Meier et al.`, where `Meier` should not be annotated as surname to be deidentified, since it is a citation.

8.1.2 Date Annotation

Dates are a fairly closed category, that is, as long as most patterns are captured in the rules how dates can be written, the annotation works well.

There is one slightly more challenging pattern where the year as well as the last `.` is missing like in `10.1` meaning 10th of January. Some care needs to be taken to correctly distinguish such dates from decimal numbers (which are e.g. followed by some unit).

Remaining issues are mainly due to misspellings of dates such as `16.11.2918` or `21.111.2018` or `20.102015`. Although to a human reader it is clear what date is actually meant, it is not straightforward to capture these in patterns. This could be addressed in future extensions.

Information Extraction for Dates Additionally to recognizing dates in text, the annotation pipeline also attempts to infer the structure of dates. This includes determining the date format, e.g. "dd.MM.yyyy" as well as to extract day, month and year components. This is helpful later on, when substitution is done with the `ReplacementTags` strategy.

The formats extracted are compatible with the SimpleDataFormat in Java. Note, that the information extraction is not guaranteed to succeed, i.e. fields may be missing/empty.

8.1.3 Name Annotation

Name annotations are driven by triggers such as titles like `Dr.` or names from lexika. That is, tokens like `Dr.`, `Frau`, `Prof` etc are most of the time followed by a name. Some rules exploit this fact.

Names not preceded by such trigger tokens are recognized using lexica. Note, that blindly annotating tokens appearing in name lexica will lead to many false positives, e.g. `Iris` may be annotated as name although the part of the eye is meant. Hence, some more "evidence" is needed, that a name candidate is indeed a name. This includes: * token is followed/preceded by another token appearing in a name lexicon * token is a frequent name and the token doesn't appear in any medical or general lexica

This approach requires lexica of good quality, i.e. they should be reasonably complete and not contain tokens which are not actually names (may be problematic if lexica directly compiled from a hospital database system)

Special care is also taken to not annotate citations such as `Meier et al.`

Shorthands/Abbreviations Shorthands or abbreviations of hospital staff also fall in the name category. At USZ staff shorthands are typically 5 characters long (although the length ranges from 3 to 8 characters) and most of the time spelled in upper case, such as `ABCDE`.

Since shorter abbreviations of 3 or 4 characters may also simply be a (medical) acronym, the annotation of such strings is only triggered in certain fields. Some report specific rules were needed for some report types where the shorthands are spelled in small case.

Information Extraction for Names Similar to the `Date` annotation, the structure of a `Name` annotation is also extracted by the pipeline, e.g. the firstname and lastname. If a salutation could be recognized, it is also included (this can be used during a substitution procedure to determine the gender of the involved name).

The following fields are extracted:

Field	Example	Description
firstname	Hans	
lastname	Meier-Müller	
signature	ABCDE	internal abbreviation/shorthand
salutation	Frau Dr.	complete salutation (usually preceding an annotation)
format	ff ll	structure of name

The format field is composed of following tokens:

Letters	Meaning
f	firstname short (typically 1 letter)
ff	full firstname
ll	lastname
LL	lastname all upper case: `MEIER`
s	signature
S	signature all upper case

Note, that fields may be empty. Furthermore, if an annotation can be called by various rules, the extracted information can be contradictory (contradictory information is separated by `,`). For instance, the text `Peter Simon` may be called by 2 rules, one for double lastnames and one for double first names leading to contradictory values for `format` (and also `firstname` and `lastname`). These conflicts are currently not resolved and need to be handled by the downstream pipeline.

8.1.4 Location Annotation

This is a fairly broad category encompassing physical locations such as places and countries (by extension also languages) as well as organizations such as hospitals, departments within hospitals, medical practices etc.

`Location` annotations heavily depend on good lexica. There are a few rules to recognize street names and zip codes. Also many health care organizations can be recognized by certain triggers like `Spital`, `Altersheim` etc. These triggers are also important to recognize organizations which may be in some lexica, but are referred to differently by the medical staff (e.g. `Altersheim XY` instead of the official `Pflegezentrum XY`).

Challenges arise with ambiguous locations such as `Wangen` (place and "cheeks") as well as with misspellings `Grüningen` vs `Grünigen`, `*thal` vs `*tal`.

8.1.5 Occupation Annotation

Both patterns and lexica are important to recognize occupations. Similarly with names, we cannot blindly annotate tokens which appear in a occupation dictionary, since the token may designate for example a surname or the profession or role of somebody involved with the patient, but not the patient him/herself (e.g. `beim Augenarzt`, `Polizist`).

8.1.6 Tuning Guidelines

In case a token should have been annotated by the pipeline and it wasn't, the following steps could be taken: * first (!) add a test case to the test suite. Perhaps don't take the original data for privacy reasons but slightly alter it to reflect the situation. Verify the test case fails (`make test`). * if appropriate, add more tokens to corresponding lexicon * if appropriate, tweak existing rule or add a new rule * verify the test(s) passes now.

The same procedure can roughly be followed if a token was wrongly annotated. In this case, perhaps entries need to be removed from a lexicon. Or in some added to a "false positive" or "ambiguous" dictionary. Refer to the lexica overview to pick an appropriate lexicon to edit.

When editing lexica try to limit the editing to lexica marked as "manual" in lexica overview. This way, if other types of lexica get regenerated by a script, your changes don't get overwritten.

9 Guidelines for System Tuning

9.1 Tuning Tutorial

Here few hands-on "exercises" on how to tune the pipeline on a heavily simplified version of the USZ deidentification pipeline. The goal is to get familiar with the various components in a simplified setting without being overwhelmed by all the details of a full-fledged pipeline.

9.1.1 Setup

To test whether modification of the pipeline lead to the desired behavior, we are going to use the testing framework included in the deidentification tool. Tests for many of the exercises below are already prepared in the `configs/tutorial/testcases`, they just need to be uncommented (i.e. removing the `#`)

The test suite can be run using the following command:

```
java -jar [path to jar file] test [pipeline config file] [test cases directory]
```

where `[path to jar file]` should point to a current `jar` file of the pipeline (typically `deidentifier-*.jar`), `[pipeline config file]` to some path ending with `configs/tutorial/tutorial.conf` and `[test cases directory]` a path ending on `configs/tutorial/testcases`. Tip: put the resulting long command into a `.bat` or `.sh` file which you then execute.

When running the test suite, if everything goes well, you should see lines containing `Reading testcases from` at the end. If a testcase should fail, a clear error message is displayed with some more details what went wrong.

9.1.2 Exercices

Add test case for dates In the file `date.txt` there is already one test case defined to check whether a date is indeed recognized. Uncomment that line and run the test suite. You should see something like

```
2019-12-06 13:27:40.891 INFO  org.ratschlab.deidentifier.pipelines.testing.PipelineTestSuite - Reading 
2019-12-06 13:27:40.895 INFO  org.ratschlab.deidentifier.pipelines.testing.PipelineTester - Running tes
```

On a new line add another date without the <`Date`> tags. Run the test suite again and see how it fails. Add the tags, run again and this time the suite should pass.

Add missing location The location `Oberikon` is not recognized in a document. Add a test case for it in the `locations.txt` file and run the test suite. It should fail on that test. Then, add the place to some appropriate lexikon (e.g. in the already existing `locations/additional_locations.lst`). After that, the test suite

Internal phone number format Assume internal phone numbers consist of two blocks of 3 digits, e.g.: `123 456`. Write a JAPE rule which recognizes these numbers. There is already some test case in `contact.txt`

Hints: * add the rule in `specific-rules/contacts.jape`. There is already some rule recognizing some Swiss phone numbers (copied from the generic JAPE rule set). * See https://gate.ac.uk/sale/thakker-jape-tutorial/GATE%20JAPE%20manual.pdf if you'd like to know more about how JAPE rules work. * In practice, you would probably add a "trigger" on the left side, i.e. fire the rule only if the two blocks are preceded by a "Tel" token.

10 Performance Evaluation and Metrics

10.1 Evaluation Guide

We sketch how to evaluate the performance of the pipeline on your reports.

10.1.1 Create Goldstandard Set

One approach is to annotate relevant reports using the existing pipeline (using the `annotate` command) and let experts check/complement the resulting annotations. Alternatively, you can simply convert the reports to the GATE internal format using the `import` command. In this case, the annotator would have to create all annotations manually, which can be very tedious.

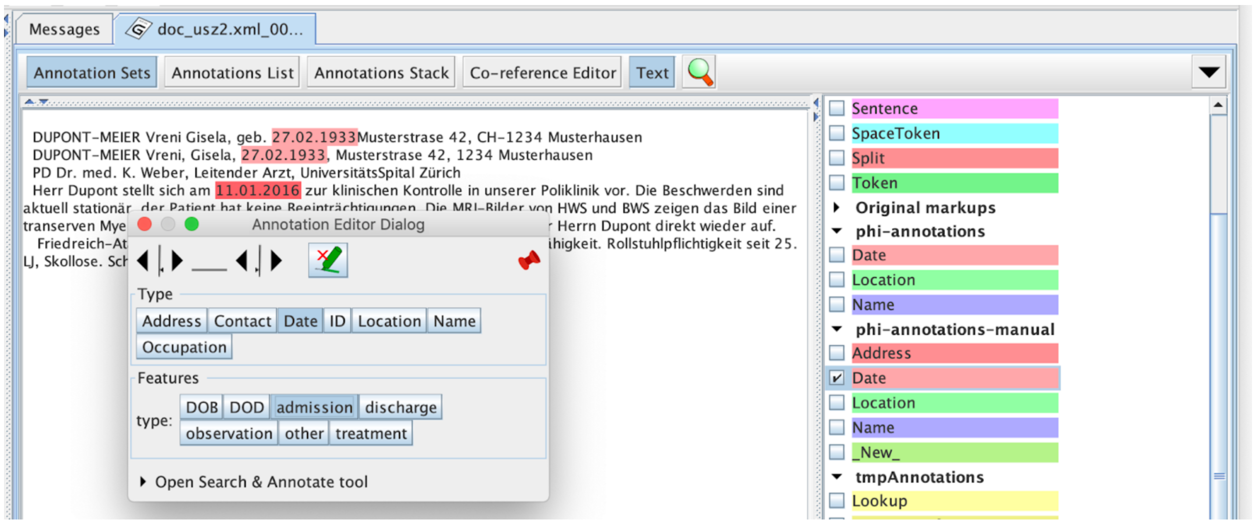
Annotation Guide First, install GATE Developer.

If you haven't already, load the `Schema Annotation Editor` via `File` -> `Manage CREOLE Plugins`. Tick `Load now` and `Load always` next to `Schema Annotation Editor`. Restart GATE

Load a schema into GATE: right click on `Language Resources` in the explorer view, choose `New` -> `Annotation Schema`. Click on the briefcase symbol on look for the master.xml file on your filesytem . Open a GATE document, activate `Annotation Sets` view and select the phi-annotations-manual annotation set on the right.

Make sure, schemas are loaded in GATE before you open a corpus to annotate. Load the corpus via `File` -> `Datastores` -> `Open Datastore`. Load document and add necessary annotations by marking some tokens and pressing `Ctrl+E` When creating a new annotation, make sure `phi-annotations` is selected on the right hand side. Regularly do Right-click on document then `Save to its datastore` (not done automatically!)

Also make sure reports are read-only to not accidentally edit the document:



DUPONT-MEIER Vreni Gisela, geb. 27.02.1933Musterstrase 42, CH-1234 Musterhausen
DUPONT-MEIER Vreni, Gisela, 27.02.1933, Musterstrase 42, 1234 Musterhausen
PD Dr. med. K. Weber, Leitender Arzt, UniversitätsSpital Zürich
Herr Dupont stellt sich am 11.01.2016 zur klinischen Kontrolle in unserer Poliklinik vor. Die Beschwerden sind aktuell stationär, der Patient hat keine Beeinträchtigungen. Die MRI-Bilder von HWS und BWS zeigen das Bild einer transerven Mye … r Herrn Dupont direkt wieder auf.
Friedreich-At … ihigkeit. Rollstuhlpflichtigkeit seit 25. LJ, Skollose. Sch

Figure 4: Annotation View

10.1.2 Evaluate with Respect to Goldstandard Set

To compare the pipeline output to the goldstandard corpus, annotate the same reports using the `annotate` comman, but this time add the `-m` option with the path to the goldstandard corpus (`m` for "marked"). You can also add the option `--diagnostics-dir` with path where diagnostics information should be output. It will contain a performance summary in `corpus-stats.html` as well detailed output for every report.

Analyse Results in More Details With the `--diagnostics-dir` option, various "features" of every token are extracted and stored in json files in the `ml-features.json`. These features include the annotations of the pipeline and the rule generating the annotation as well as the annotations from the goldstandard (if available). They also include in what lexica a token was found, in what field the token appeared, the position within the sentence.

These json file(s) can be converted to parquet using a python script and then explored in a jupyter notebook. This is very helpful to get an overview over problems in one or several corpora.

11 Case Study: Pipeline Performance on Report from University Hospital Zurich

11.1 Pipeline Evaluation

11.1.1 Method and Corpus

400 reports from the University Hospital Zurich were picked at random across around 30 document types (mainly discharge reports) giving rise to around 2.7 Mio tokens. These reports then got annotated using a prelimninary version of the pipeline. A medical student then went through these annotations in GATE Developer and complemented/corrected the existing ones. In a last step, the annotations got checked by a member of the USZ staff. Then, the annotations of the mature version of the pipeline was compared to the verified annotations ("goldstandard") and evaluated.

The corpus was split into two parts with 200 reports each. The first part was heavily used for development and tuning ("training set") whereas the second was only used for evaluation ("validation set"). No evaluation on a strictly unseen test set was performed.

A bit more details can be found in Section 10.1.

11.1.2 Results

The following tables contain Precision/Recall by annotation types comparing the pipeline output with the annotations in the goldstandard. The `Recall` and `Precision` columns refer to exact matches of annotations, the columns `Recall Lenient` and `Precision Lenient` also include partial matches, for instance, for the name `Hanna Meier Huber`, the pipeline only annotates `Hanna Meier`.

Entire Goldstandard Corpus (Parts I + II):

Type	Matches	Recall	Recall Lenient	Precision	Precision Lenient
Age	716	94.584	94.584	95.086	95.086
Contact	4805	99.154	99.505	99.154	99.505
Date	47223	99.145	99.397	98.574	98.825
ID	187	79.915	85.897	3.166	3.403
Location	31427	85.300	91.415	84.309	90.353
Name	17422	98.787	99.484	94.561	95.229
Occupation	290	55.238	63.619	60.291	69.439

Goldstandard Part I

Type	Matches	Recall	Recall Lenient	Precision	Precision Lenient
Age	361	96.524	96.524	93.282	93.282
Contact	2384	99.375	99.625	98.962	99.211
Date	24686	99.252	99.481	98.673	98.901
ID	82	80.392	87.255	2.495	2.708
Location	16220	87.576	93.391	84.020	89.599
Name	9195	98.712	99.377	93.445	94.075
Occupation	157	60.153	68.966	62.800	72.000

Goldstandard Part II

Type	Matches	Recall	Recall Lenient	Precision	Precision Lenient
Age	355	92.689	92.689	96.995	96.995
Contact	2421	98.937	99.387	99.343	99.795
Date	22537	99.029	99.306	98.466	98.742
ID	105	79.545	84.848	4.009	4.276
Location	15207	82.999	89.417	84.620	91.164
Name	8227	98.870	99.603	95.841	96.552
Occupation	133	50.379	58.333	57.576	66.667

11.1.3 Annotation Issues Observed

Age Issues with more complex sentence structures, such as

- `Brüder verstorben mit 77, 71 und 78 Jahren`
- `Sie sei eigentlich 49 und nicht 42 Jahre alt`

Contact Some hospital internal phone numbers were not recognized.

Date Some scores are erroneously recognized as dates.

- `Nutritional Assessment: 7/15, Faszikulationen an 10/10 Stellen`
- `Beginn: 1.2, Albuminquotient 13.4`

ID The extremely low precision comes from the fact, that what is considered as IDs was changed. That is, some entities now annotated by the pipeline are not annotated in the goldstandard. For the recall, some model numbers were erroneously annotated in the gold standard

Location The somewhat medium performance in the location category originates in the broad definition including place names and terms referring to names of organisations or organisational units. It is also not always clear whether terms like `Physiotherapie`, `Unfallchirurgie`, `Innere Medizin` and the like are part of an organization name or are just generic medical terms not carrying any identifying meaning. This sort of ambiguous case make up the largest part.

A more detailed look reveals, that the pipeline missed very few patient related location information, that is a 7 locations outside Switzerland in Goldstandard Part I. About 10 organisations were missed.

Name Great care was taken to not miss names of patients or staff, i.e. the pipeline is tuned for a high recall. The comparatively low precision is due to the fact, that some staff abbreviations were not annotated in the goldstandard corpus. Another problem are that medical terms like `M. Scheuermann`, `Spina`, `Carina`, `Karina`, `B. Fieber` get annotated as names.

Occupation This is a difficult category to annotate, since the way professions or occupations can be expressed are quite variable. They typically occur in certain fields related to anamnesis. These can be excluded using the `--fields-blacklist` option in the substitute command (Section 1.3.1) if the risk is too high.

12 Discussion and Summary

This document has detailed the development and deployment of a sophisticated de-identification tool for clinical reports, designed to ensure privacy and compliance with health data regulations. Through rigorous testing, including a focused case study at the University Hospital Zurich, the tool has demonstrated high effectiveness in recognizing and anonymizing personal health information (PHI). Notably, the tool achieves high precision and recall across various data types, with particularly robust performance in handling names and dates, which are common identifiers in clinical data.

Challenges remain in accurately identifying and processing less structured data and nuanced PHI elements, which sometimes lead to inconsistencies, particularly in free-text fields. Future improvements will focus on enhancing the tool's machine learning models to better understand contextual nuances and reduce false positives, thereby increasing the reliability of the de-identification process.

Overall, the tool stands as a critical asset in the realm of medical data processing, offering robust privacy safeguards without compromising the utility of the data for research and clinical review.

13 Discussion and Summary

This document has detailed the development and deployment of a sophisticated de-identification tool for clinical reports, designed to ensure privacy and compliance with health data regulations. Through rigorous testing, including a focused case study at the University Hospital Zurich, the tool has demonstrated high effectiveness in recognizing and anonymizing personal health information (PHI).

Notably, the tool achieves over 99% recall in identifying and anonymizing 'Contact', 'Date', and 'Name' entities, and approximately 95% for 'Age', with 'Location' entities recognized

with about 91% recall. These results underscore the tool's robust capability to safeguard sensitive information. Challenges remain in the lower recall rates for 'ID' and 'Occupation', which are 85% and 64%, respectively. However, these entities are considered less critical from a privacy standpoint as 'ID' numbers appear infrequently and occupations are generally not too specific. **If a recall threshold of at least 90% is set as a benchmark for privacy concerns, this tool reliably removes critical PHI categories such as Age, Contact, Date, Location, and Names.**

Future improvements will focus on enhancing the tool's machine learning models to better understand contextual nuances and reduce false positives, thereby increasing the reliability of the de-identification process. This ongoing enhancement will ensure that the tool not only meets the regulatory requirements but also adapts efficiently to varied data formats and clinical environments, maintaining high standards of patient privacy without compromising the utility of the data for research and clinical review.