DsgvoAssistant - Removing Dsgvo Content From HTML

Erik Jonathan Schmidt
Otto-Friedrich University of Bamberg
96049 Bamberg, Germany
erik.jonathan-schmidt@stud.uni-bamberg.de

Projekt: MfI-IR-Proj 15 ECTs
Degree: M.Sc. AI
Matriculation #: 1867139

Abstract

The content of DSGVO elements in websites is suspected to blur search results. To test and address this issue a dataset of 428 websites is created and manually labeled. Based on this dataset it is shown that the text related to DSGVO can dominate the text extracted for a whole website. The goal is to resolve this problem by removing such text prior to indexing. At first a proof of concept for identifying DSGVO text and elements is created. Achieving 94% accuracy in this classification task, the proof of concept is developed further into the Kotlin module *DsgvoAssistant* that removes DSGVO elements from the DOM tree of an HTML automatically.

1 Introduction

Search engines help users to navigate the world wide web. They provide a way for finding websites that are relevant or interesting to the user without the need to know the exact web address. Under the hood, those search engines employ crawlers - programs that permanently traverse the web and process the content they encounter while accessing each web address. This continuous traversal is called **crawling** and is followed by **indexing** the accumulated content. Indexing refers to creating registers or similar structures that later help to quickly associate an incoming user query with a website that contains content that is somehow similar or relevant to the query text. The process of indexing involves many steps of parsing and preprocessing the HTML and plain text of a website into formats that are suitable for later search. For example one might store all named entities that can be detected within the content into a list that is stored separate from the rest of the text. Then, when a user query contains a named entity, the search engine can quickly check if the named entity is mentioned on a site by only looking up this list, not considering the rest of content.

An other action that takes place after the raw website content was crawled and before data is eventually indexed is filtering the text with respect to relevance. That is the step where this project comes into play. In 2016 the EU introduced the data privacy law that made it mandatory for hosts of websites to inform visitors about what personal information is collected and to offer an opt out (eud). In most cases this is implemented as a pop up window that appears when entering a site as shown in fig. 1. The German adoption of this EU wide law is called DSGVO, which stands for "Datenschutz-Grundverordnung". In the following the English term GDPR is used when speaking about websites of arbitrary language, while DSGVO is used only when describing German websites.

These pop ups and similar user interface elements contain text to inform users about data usage and often allow to opt in and out of several options. They do not contain information that corresponds to the website's original topic. For example, on a web developer's homepage you would not expect to find her CV or contact information within the DSGVO pop up. The content within these pop ups

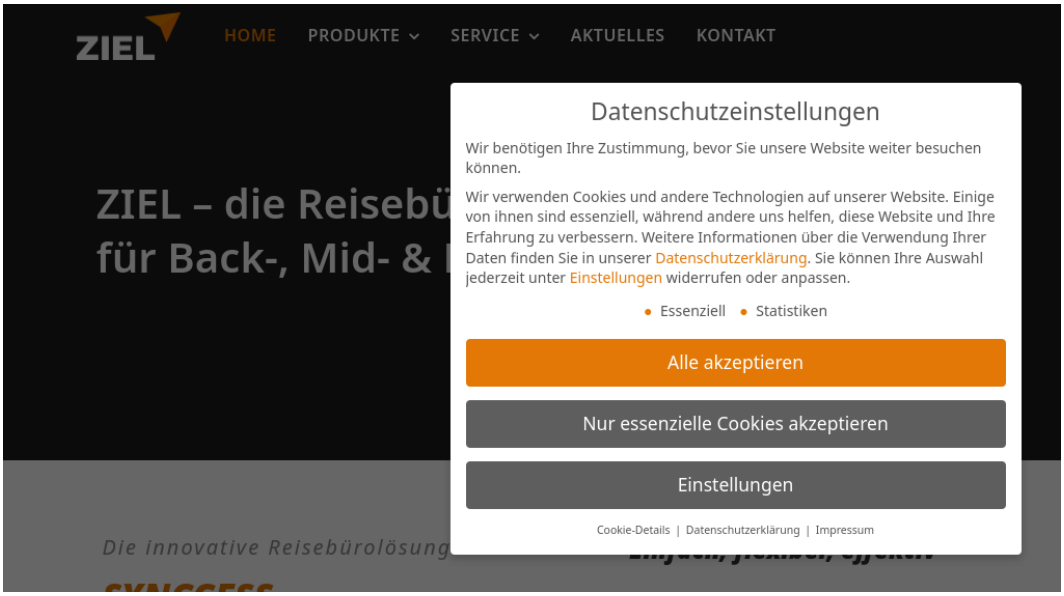

Figure 1: A screenshot of an exemplary DSGVO pop up.

always and only concerns data privacy, irrespective of what the website is about. It is not part of what the host of this website intended users to find when they visit this particular page. Consequently, storing the text found within DSGVO related elements does not bear any value when crawling and indexing websites for a search engine. Storing these redundant paragraphs for each website could probably even pollute search results. This seems likely, especially if the text of the DSGVO element makes up for a big share of all text found on a website. In that case a website would appear as mainly covering data privacy questions, when it might actually is a homepage of some company concerned with a totally different domain that just included a long extensive data privacy pop up on their page.

The aim of this project is to create a module for handling the DSGVO content of websites before they get stored and indexed.

Originally, the idea for such a DSGVO assistant emerged from issues that were observed while developing the IT-Atals-Oberfranken. This IT-Atlas is a vertical search machine for job offers in the area of Oberfranken Germany. Observations from the development of this IT-Atlas suggested that indexing DSGVO texts hinders search performance and hence led to this project.

2 Background

With the goal of developing a DSGVO assistant at hand, this section explains some concepts and tools that are used throughout the project. Following this, related work is reviewed and discussed.

2.1 Content Management Providers

Content Management Providers (CMPs) are platforms that help website owners manage user consent for data collection. They enable the display of privacy pop-ups that inform users about cookies and tracking, allowing them to choose preferences. These pop-ups comply with privacy regulations like DSGVO, giving users control over their online privacy. Using a pop up from a CMP instead of implementing it from scratch allows website hosts to outsource coding and maintenance. Furthermore CMPs provide legal expertise in the field of data privacy that the host might not have.

2.2 Chrome DevTools

Chrome DevTools is a suite of web development and debugging tools integrated directly into the Google Chrome browser. It offers a comprehensive set of features that aid developers in inspecting,

debugging, and optimizing web applications. In this project DevTools are used to inspect and copy the HTML of websites.

2.3 Shadow Root

In HTML, the shadow-root comment is a visible placeholder comment that developers often use to indicate the location where a Shadow DOM (Document Object Model) is attached. Shadow DOM allows encapsulation of markup, styles, and behavior in a closed container, separate from the main document DOM. When a website host includes third party code for a DSGVO pop up, encapsulating it as Shadow DOM ensures that the third party code does not interfere with the code of the website. The <!– shadow-root –> comment acts as a marker to indicate the boundary where the Shadow DOM starts within an element. It doesn't contain any content or style by itself but serves as a visual cue for developers working with Shadow DOM to identify the presence and placement of the encapsulated content.

2.4 Consent-O-Matic FireFox Plugin

Developed as Aarhus University, Consent-O-Matic is an opensource FireFox plugin that automatically enters user preferences into data privacy pop ups. It was developed in the work of Nouwens et al. (2020) and maintains a list of the most common CMPs and how to handle their pop ups. Consequently this implementation is not able to cope with pop ups that were not already added to the list. The plugin enters user preferences and interacts with the pop up. This ensures that the content on the website is displayed exactly as it would be if the user clicked herself. Bollinger (2021) followed a similar approach when analysing such pop ups.

2.5 Cleaning Pure HTML

Inspection of websites showed that just focusing on pop ups from CMPs is not sufficient for this project, because less then half of the inspected websites contain a pop up from a CMP and the rest employ custom solutions.

Therefore this project does not build on a comprehensive list of DSGVO pop up providers. Instead a solution is built, that detects such pop ups based on the text that is visible to users. The proposed solution operates directly on the HTML DOM tree, which is assumed to be faster then relying on a web driver to click the right buttons.

3 Project Overview

To get started with the topic of identifying and cleaning HTML of DSGVO text the first part of this project is dedicated to developing a proof of concept in Python. The code produced in this phase is described in section 4, section 5 and section 6 and is located at *dsgvo_handler/POC/*. Within the *POC/* directory code is grouped into subdirectories that represent different steps of the proof of concept phase. The subdirectories are listed in chronological order below, together with a short summary of what they contain:

1. *dsgvo_handler/POC/data/* - here all the raw data is stored
2. *dsgvo_handler/POC/dev_dataset_creation/* - code for creating the raw dataset all subsequent steps are based on
3. *dsgvo_handler/POC/dev_dataset_refinement/* - code for ensuring data quality and removing unwanted samples
4. *dsgvo_handler/POC/dev_dataset_stats/* - code for analysing the created datasets
5. *dsgvo_handler/POC/svm_trainingset_creation/* - code for constructing a training dataset and for training a Support Vector Machine
6. *dsgvo_handler/POC/my_utils* - code for loading the dataset, text preprocessing and implementations of the two proof of concept approaches
7. *dsgvo_handler/POC/dev_dataset_for_jvm_impl/* - code for preparing the data for the Koltin DsgvoAssistant

8. *dsgvo_handler/POC/data_for_jvm/* - datasets that are used in the Kotlin DsgvoAssistant in JSON format

Next to these subdirectories the proof of concept directory contains two Streamlit applications and a Jupyter Notebook for testing the two proof of concept approaches:

- *dsgvo_handler/POC/dev_dataset_browser_application.py* - Streamlit web application two review the created datasets
- *dsgvo_handler/POC/test_subtree_scorer_application.py* - Streamlit web application that presents the two approaches from *my_utils*
- *dsgvo_handler/svm_poc_cross_validation.ipynb* - Jupyter Notebook for finding the best parameters of the two approaches

For each script *sample_script.py* that needs to load datasets or similar a configuration JSON file *sample_script_config.json* is given, that specifies the paths. To make this code executable these absolute paths need to be adjusted according to ones machine.

The second part of this project is to use the datasets and approaches, that were developed in the proof of concept phase, to implement the DsgvoAssistant module in Kotlin. This is described in section 7 and section 8. The according code is located at *dsgvo_handler/IR_Kotlin_Proj/* and *dsgvo_handler/IR_Kotlin_Proj_evaluation/*.

4 Dataset

For the goal of developing the DSGVO Assistant, the first step is to get familiar with the websites it is meant to assist with. The IT-Atlas crawl is based on a list of 748 seed URLs corresponding to IT companies located in the area of Oberfranken. Such IT companies are the target of the vertical search engine the assistant is built to improve. Therefore a subset of 428 of these websites were inspected manually and compiled into the *dev_dataset_refined* consisting of 300 German websites labeled as containing some sort of DSGVO element or not.

4.1 Dataset Creation

dsgvo_handler/POC/dev_dataset_creation/

In the first iteration of dataset creation the 428 websites from the seed URL list are viewed. While the website inspection is done manually, Python scripts are used to speed up the process. At first the Python Jupyter Notebook *1_curated_dataset_creation* loads all seed URLs from the Excel file *companies_enc*. Then the *seed_website_index* is set to zero by hand. With *seed_website_index*=0 a subfolder *dev_dataset/0* is created and the 0th seed URL is opened with the help of a Selenium web driver. Selenium stores the websites source HTML to *dev_dataset/0/page_source.html*. In case selenium could not extract the page source HTML a "-1" is stored inside the respective file, this occurs for example when the website is not accessible. With the website open in the browser, the developer decides if there is any DSGVO element visible. If not, the *seed_website_index* is increased and the next website is examined and added to the *dev_dataset*. If one is found, the execution of the next code cell creates the empty files *dev_dataset/0/page_source_removed.html* and *dev_dataset/0/page_source_cleaned.html*. Now the developer tools tab of the browser is used to view the source HTML. By hand the div that wraps all DSGVO content is selected and cut from the DOM tree. The cut DSGVO element HTML is stored to *dev_dataset/0/page_source_removed.html* and the HTML of the website with the DSGVO content now cut out is copied over to *dev_dataset/0/page_source_cleaned.html*. This process is iterated until the development dataset folder is filled with subfolders of 428 websites.

Then the next Jupyter Notebook *2_automated_dataset_completion* is used to augment the existing dataset. This notebooks performs three actions on all 428 website subfolders. At first the plain text that users can read on a website is extracted from *dev_dataset/0/page_source.html* and *dev_dataset/0/page_source_removed.html* and stored to *dev_dataset/0/content.txt* and *dev_dataset/0/content_removed.txt* respectively. Secondly, a pretrained model with the fastText architecture proposed by Joulin et al. (2016) is used for language detection. For each website the readable text stored in *dev_dataset/0/content.txt* is fed to the model to obtain a language label, e.g.

"__label__de" if the text is classified as German language. The resulting map, index of website to language label, is stored as *dev_dataset/dev_dataset_languages.json*. The last step to complete the development dataset is to associate each website subfolder with its origin URL. To do so, the according URL for each index of a website is looked up from the initial list of seed URLs and the mapping is stored to *dev_dataset/dev_dataset_urls.json*.

Figure 2: A screenshot of of the development dataset folder structure after creation.

This concludes the creation of the base development dataset that will be filtered and refined in the following. All websites collected and inspected at this point serve as the starting point for examining DSGVO content on pages. The folder structure of the dataset at this stage is shown in fig. 2. Subfolder *dev_dataset/0* corresponds to a website where no DSGVO element is present, so the plain text and the HTML are all that is stored. The data stored for a website with DSGVO related content can be seen in subfolder *dev_dataset/3*, where the removed content, the cleaned HTML and the removed HTML are stored in addition to the two files that are always populated. The two files that contain the mapping of development dataset index to language and URL of each website reside at the end of the *dev_dataset* folder.

4.2 Dataset Browser App

To explore the websites of the dataset a web app is built with the open source framework Streamlit. The code for running the app resides in *dev_dataset_browser_application* and makes use of the *GDPRDataset* class. This class implements a wrapper for the before described data structure. Given the path to the root of the dataset folder structure *GDPRDataset* reads in all files into one big table and adds a "is_GDPR" column to state whether a sample contains DSGVO content. If a sample subfolder has no files *content_removed*, *page_source_cleaned_html* and *page_source_removed_html* than "is_GDPR" is set to false. If they all exist it is set to true. The data format is summarized in table 1.

While the folder structure is more convenient for creating and manually extending the dataset, having all data in one table with one row per website is the preferred way for handling the data in subsequent steps.

With the dataset prepared, the application allows to go through the entries one by one as shown in fig. 3. For the currently selected website it displays a rendering of the website HTML next to the manually extracted GDPR related text. Furthermore the user can set filters to only inspect websites with German text or only those, where GDPR content was found. This app helps to validate the correctness or sensibleness of the dataset. During the phase of dataset creation it is a valuable tool for exploring the dataset. As the dataset can be reviewed faster with the app than by navigating through the underlying folder structure, it helps to identify and revise mistakes introduced through the manual data acquisition. Problems and inconsistencies discovered with the help of the app are addressed in the following dataset refinement step.

4.3 Dataset Refinement

dsgvo_handler/POC/dev_dataset_refinement/

In this section the pool of all inspected website is screened for completeness of data and relevance to the goal of removing DSGVO text from crawled German language websites.

The very first step is to remove samples that were assigned a language different to German. Note that the reduction to only German websites is due to the nature of the overall project, which is a

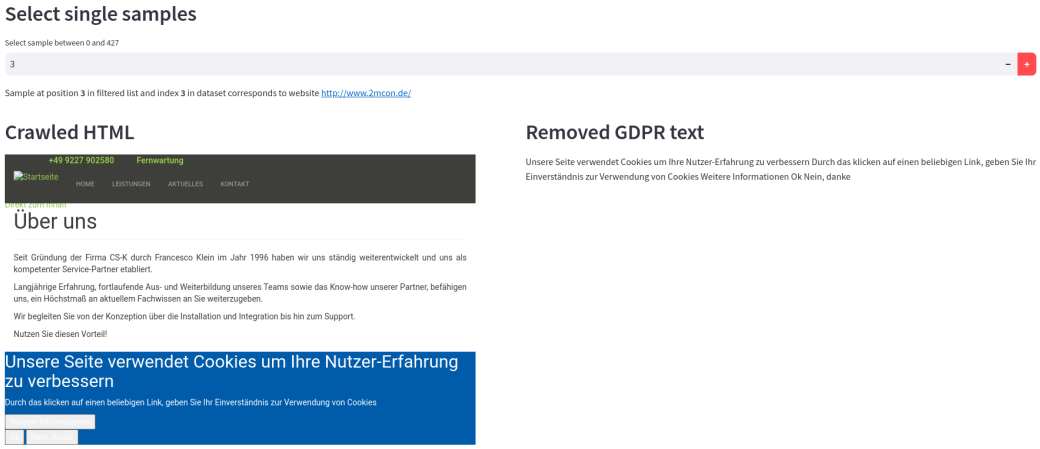

Figure 3: Screenshot of the dataset browser app. The website with index 3 is displayed. To the left the HTML extracted with Selenium is rendered. To the right the raw text of the website that was manually classified as DSGVO content is shown.

search machine limited to job offerings within "Oberfranken" Germany. Consequently websites with German language are the focus of this work. While not tested it seems likely that the following approach would work for other languages as well.

At next, samples get removed if Selenium could not downloaded the source HTML and a "-1" was stored to *page_source.html*. The resulting dataset is stored to *dev_dataset_refined_intermediate*.

As previously done with the *dev_dataset*, the *dev_dataset_refined_intermediate* is inspected with the dataset browser app. This reveals samples with a common inconsistency. The rendering of the downloaded HTML does not show any DSGVO element, but on creation a DSGVO element was extracted manually with the browser development tools. Based on this observation a consistency check is done. For each website the number of words that are only present in the removed DSGVO text, but not in the overall website content is obtained. This yields ten samples with DSGVO content that is not contained within the extracted page HTML. On seven of these websites the DSGVO element is located inside a **shadow root** element which hides its descendants.

Consequently Selenium is not able to capture the DSGVO content below the shadow root element, while inspecting the website with the development tools makes descendants visible. The other three samples exhibit similar inconsistencies, originating from content being hidden in JavaScript scripts embedded in the HTML. At this point the original dataset of 428 samples is condensed down to 300 samples which are considered complete and consistent. This final dataset is saved at *dev_dataset_refined*.

4.4 Dataset Format and Statistics

dsgvo_handler/POC/dev_dataset_stats/

The schema of the refined dataset is displayed in table 1. Note that the value of "language" is "__label__de" for all 300 entries as the data was filtered for websites with German content. For samples where "is_GDPR" is true the value of "content_removed", "page_source_cleaned_html" and "page_source_removed_html" is an empty string.

Overall 151 of 300 samples correspond to websites with DSGVO element. This equal distribution is a beneficial property of the dataset for later use as training data. Summarizing statistics on the developed datasets are given in table 2. The share of GDPR containing or German websites can be retrieved from the datasets directly.

Index	Index from 0 to 299
language	Label stored as string
url	Website address as string
content	Visible text as string
page_source_html	Website HTML as string
content_removed	DSGVO text as string
page_source_cleaned_html	Cleaned website HTML as string
page_source_removed_html	Removed HTML as string
is_GDPR	Boolean

Table 1: Description of the data format of the refined dataset. Each sample consists of the eight fields shown.

	dev_dataset	*dev_dataset_refined*
samples	428	300
samples with GDPR	188	151
German samples	310	300

Table 2: Statistics on *dev_dataset* and *dev_dataset_refined*

.

A deeper analysis is done on the data to find out how much text on websites is DSGVO related and to determine how widespread the use of pop ups from CPMs is.

$$\frac{r_0 + ... + r_{299}}{c_0 + ... + c_{299}} = 0.000095 \quad (1)$$

Figure 4: Calculation of the overall share of DSGVO text in the dataset.

For each sample with index i we obtain the length of its content c_i as the number of words found by a tokenizer and the length r_i of the removed DSGVO content similarly. Summing them up and getting their ratio reveals that 0.0095% of the overall stored text is DSGVO related (See fig. 4). Consequently relevant savings in storage can not be expected as a result of cleaning text of DSGVO content.

$$\left(\frac{r_0}{c_0} + ... + \frac{r_{299}}{c_{299}}\right)/300 = 0.131546 \quad (2)$$

Figure 5: Calculation of the mean share of DSGVO text per sample.

Looking at the ratio for each sample individually gives some further insides. The mean calculated in fig. 5 shows that on average over an eighth of each website content corresponds to the content of a DSGVO element. This speaks for the hypothesis of search results being blurred by websites containing irrelevant DSGVO related text. Among the 151 samples that contain DSGVO text the average mean is 0.262621. So on average a website with some DSGVO element contains 26% text that does not help identifying what the website is about. This suggests that cleaning website contents of DSGVO text before storing and indexing them could enhance search results. Experiments that verify this hypothesis are not part of this project and remain for future investigation. Fig 6 shows the distribution of the DSGVO to content ratio within the positive labeled samples. While over fifty of the 151 samples contain less then 10% DSGVO related content, a considerable amount of websites display over 50% DSGVO related text, that is considered irrelevant or even detrimental for search. The text originating from DSGVO elements makes up to 88% of the content of a website. These findings underline the hypothesis that cleaning HTML of DSGVO text before indexing could improve search.

The datasets undergo further investigation to estimate how many of DSGVO pop ups are provided by a CMP. A CMP provides an implementation for data privacy clarifications that can be used by hosts instead of a custom built solution. Therefore the same or similar DSGVO pop up of a single CMP is found on multiple sites. To find out how widespread such solutions are the "page_source_removed_html" of each sample in a dataset is inspected. All tag ids found in an HTML are collected. For each id it is counted on how many different websites it occurs. This reveals that "BorlabsCookieBox" is the most common id in GDPR related HTML subtrees. Consequently it is assumed that 30 of the GDPR websites contain a pop up form the CMP "Borlabs". At this point all websites that contain the id "BorlabsCookieBox" are removed and the most common id in the remaining samples is detected. With this approach we identify pop ups from six different companies

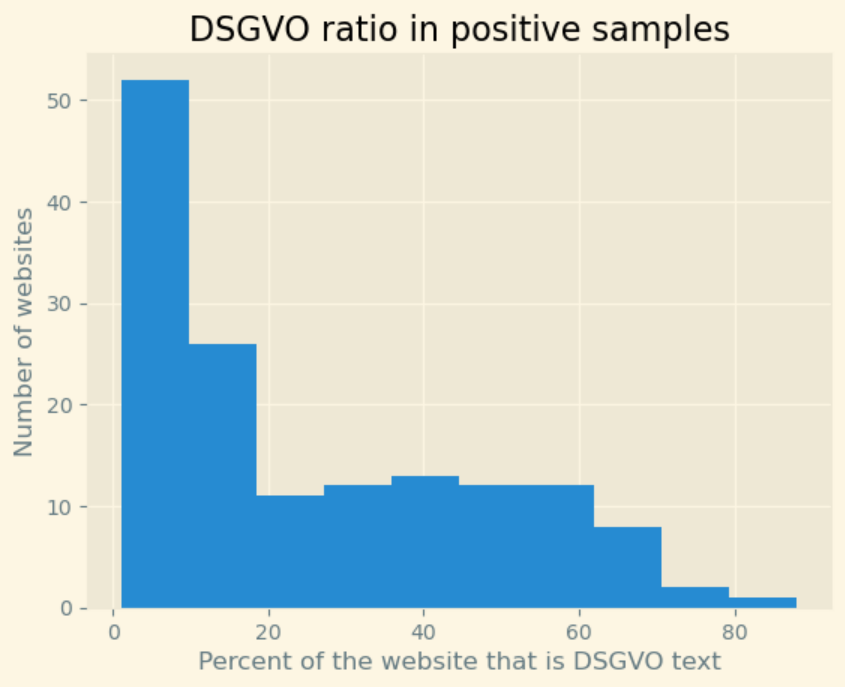

Figure 6: DSGVO content ratio among websites that contain DSGVO elements in the *dev_dataset_refined*. For each sample ratio $\frac{r_i}{c_i}$ is calculated.

CMS name	*Occurences*
Borlabs	30
Cookiebot	16
Usercentricts	19
Complianz/CMPLZ	10
OneTrust	4
reDim GmbH	4

Table 3: Company names of CMPs and number of websites in *dev_dataset* that use their GDPR elements.

that are used in several sample websites. The companies and the number of websites using them in *dev_dataset* are listed in 3. Overall 83 of the 188 GDPR containing websites in *dev_dataset* contain a pop up from one of these companies. That means that the remaining 105 websites either use a CMP that was not detected in the analysis or they use a custom built element to information about data privacy. So the approach of blocking common CMP pop ups like the Firefox plugin Consent-O-Matic2.4 would fail to clean over half of the websites.

5 Identifying DSGVO Text with SVM

dsgvo_handler/POC/svm_training_set_creation

At this point data is collected and labeled. The final goal is to create a module that takes in the source HTML of a website, decides if there is DSGVO text displayed and removes the according DOM tree elements if so. The idea is to build this assistant based on text classification. The text displayed in DSGVO elements is similar in wording among websites, so it is suggested that a classification algorithm can learn what differentiates DSGVO content from other text. Before classifying a whole website as containing DSGVO text or not, a Support Vector Machine is trained to distinguish between texts extracted from DSGVO elements and texts from other areas within the websites. This classifier will serve as the core component of the later described *DsgvoAssistant*(section 7).

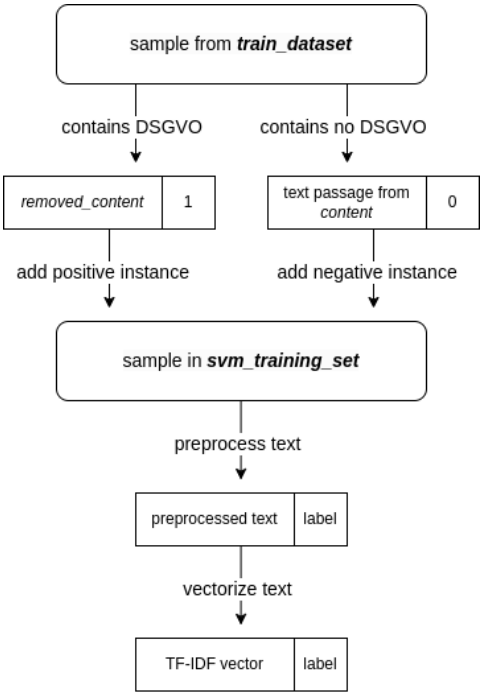

Figure 7: Shows how a sample in *train_dataset*, which has the format described in table 1, is turned into a text-to-label sample for *svm_training_set* and eventually into a vector-to-label sample.

In *1_create_svm_training_set_from_dev_dataset* the 300 samples of *dev_dataset_refined* are randomly split into 250 samples *train_dataset* and 50 samples *test_dataset*. The SVM will eventually be trained on document labeled vectors, so each training sample needs to consist of a text and a label. For each DSGVO website in *train_dataset* the text of the DSGVO element in *content_removed* is used as a positive instance. For each website that has no DSGVO element a text is extracted from *content* as a negative instance. The texts of the negative instances are cut to have similar lengths as the texts of positive instances. The resulting training dataset of labeled texts is stored to *svm_training_set*. The pipeline from *train_dataset* to *svm_training_set* is depicted in fig. 7.

To train the SVM, these short texts are first preprocessed and then used to obtain TF-IDF vectors that can be used to train the classifier on. TF-IDF vectors are document vectors, that is they are a numerical representation of a passage of text. For preprocessing German stop words are removed and words are stemmed to tokens with the SnowballStemmer by Porter (2001). TF-IDF vector values are calculated from word counts, so stemming the words is a crucial step for obtaining meaningful vectors. Without stemming the occurrences of "run" and "running" are counted separately and are associated with different values of the vector, which leads to a big vocabulary that lacks generalization.

The TF-IDF vectorizer is fit on all preprocessed texts of *train_dataset* in order to obtain the inverse document frequencies. For each preprocessed text from *svm_training_set* the term frequencies are calculated and multiplied by the inverse document frequencies to get the TFIDF score of each token in the vocabulary. The order of tokens in the dictionary is fixed, so that the list of TF-IDF scores can be used as vector, where the score of token X is always found at the same index position. The fitted vectorizer is stored at *svm_models/vectorizer.joblib*.

First experiments reveal that the SVM struggles to learn on those vectors. In many of the training runs the SVM does not adjust to the training data at all and predicts the same label for all instances. Normalizing the vectors helps overcome this problem. The RobustScaler is fit on 200 of the 250 samples in *svm_training_set* while the other 50 are set aside for validation of SVM hyperparameters. The fitted RobustScaler scales and centers the values per position. So after this standardization all the first values of all the training vectors are closer to forming a normal distribution with mean zero. This helps the classifier because now different features (TF-IDF score of different tokens) will take values of similar magnitude while relative differences within one feature survive.

These standardized vectors are then, together with their labels, used to fit a SVM. A Support Vector Machine is an algorithm for regression or classification on vectors. Here it is used as a support vector classifier. Given the vector of a text, the fitted SVM predicts two scores between 0 and 1. The first score describes how much the SVM tends towards identifying the sample as a no DSGVO text, the second value is interpreted as "is DSGVO" score. Typically, for classification one determines which of the scores is higher and returns the according class as prediction. Later in this project the second is DSGVO score is interpreted as telling how certain a given text belongs to DSGVO elements.

Support vector machines are parameterized by a regularization term C. To find the best value for this hyperparameter C in this use case different SVMs are trained on the 200 samples drawn from *svm_training_set* and evaluated on the remaining 50 validation samples.

The trained models are stored at *svm_model/*. This shows that high regularization parameter C leads to better training and validation accuracy as displayed in fig. 8. The regularization term is scaled by the inverse of C, so the best performing SVMs almost ignore the regularization term. The validation accuracy plateaus at 94% for C bigger or equal 3000, so the SVM trained with $C = 3000$ is picked for further use and this section concludes with a fitted SVM that helps distinguishing DSGVO text from normal text.

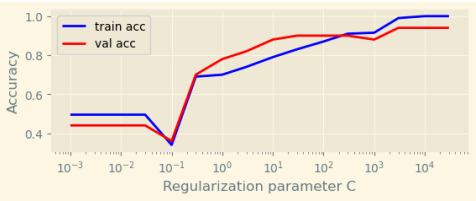

Figure 8: Plot of training and test (validation on 50 of the training samples) accuracy of SVMs trained with different regularization terms C.

Experiments in this section were conducted with the SVM, RobustScaler and vectorizers from sicit-learn (Pedregosa et al., 2011) and can be found in *2_test_svm*.

6 Identifying Websites Containing DSGVO Text

Based on the described process for text vectorization and the trained SVM, two approaches for detecting DSGVO websites are tested. While the SVM described above classifies or scores a single text, these approaches predict if an HTML contains an element with DSGVO related text somewhere. The second of the two approaches servers as a proof of concept for the later implemented Kotlin module presented insection 7. To determine whether an HTML contains a DSGVO pop up it is not sufficient to only examine the visible text of the HTML as a whole. Imagine the visible text of an HTML as a whole is thousands of words long, but the DSGVO pop up consists of just two sentences and an accept button as in fig. 9. In this case it is unlikely that the whole text can be identified as DSGVO related. The few DSGVO typical words from the pop up will have almost no effect on the calculated TF-IDF vector. Therefore subsets of the whole text are classified individually.

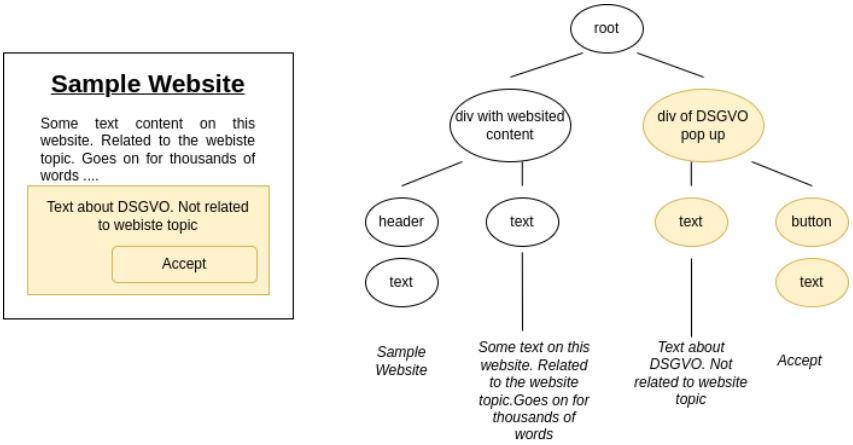

Figure 9: **Left**: An example website with text related to the domain and a DSGVO pop up that contains unrelated text. **Right**: A sketch of the corresponding DOM tree. DSGVO elements are marked in yellow.

Given an HTML its DOM tree is used to find subsets of its total visible text. The Document Object Model (DOM) tree is a hierarchical representation of an HTML document. It defines the logical structure of the document where each element, attribute, and piece of text is represented as a node in a tree-like structure. Take the DSGVO pop up with two sentences and a button as an example.

The element with the DSGVO sentences and the element with the accept button will share the same parent node (div of DSGVO pop up in fig. 9). This parent node with all its descendants (all marked yellow) represents the DSGVO pop up as whole. So the DOM subtree with this parent as root contains all the DSGVO related text. The DOM tree is traversed with BeautifulSoup (Richardson, 2007) in pre-order. For each "div" node the visible text of the corresponding subtree is extracted and added to the list of text passages, that need to be scored. A list of all nine possible subtree root nodes of the website depicted in fig. 9 and imaginary DSGVO scores for the text contained within each subtree are displayed in table 4.

root node	*DSGVO score*
root	0.3
div with website content	0.1
header	0.2
text1	0.2
text2	0.05
div with DSGVO pop up	0.99
text3	0.95
button	0.98
text4	0.98

Table 4: Subtree root nodes according to the DOM tree of fig. 9 and exemplary DSGVO scores. Cutting every subtree with a score higher than 0.9 would clean this website HTML as wanted.

The first **TFIDF+cosine** approach is based on cosine similarity and serves as a baseline. The implementation is found in *score_subtrees.TFIDF_ref_doc_sim_subtree_scorer*. At first all DSGVO texts (content_removed) in the train dataset get concatenated to one long text. The TF-IDF vector of this text serves as representation of the average DSGVO text. Given a website to classify, the HTML DOM tree is traversed and for each node the text of the according subtree is vectorized with the before fitted TF-IDF vectorizer. Then the vector is compared to the reference vector. This results in a score for each subtree of the HTML as depicted in table 4. The highest reference vector similarity score is used to make the prediction. If it lies above a certain threshold the website is classified as containing a DSGVO element. With this **TFIDF+cosine** approach the highest accuracy on the *test_dataset* is 90% with a cosine similarity threshold of 0.2.

The second **TFIDF+SVM** approach traverses the DOM tree in similar fashion to score subtree texts. The implementation is found in *score_subtrees.TFIDF_SVM_subtree_scorer*. Instead of the cosine similarity the is-DSGVO-score (the score the SVM output for how much the sample belongs to class 1, the group of DSGVO texts) of each subtree is predicted by the fitted SVM. Note that the scores are calculated differently in both approaches and are not comparable. Again a threshold needs to be picked that needs to be surpassed by one of the scored subtree texts in order for the whole website to by classified as containing a DSGVO element. For determining the best threshold the approach is tested on the whole *train_dataset* with thresholds between 0.5 and 0.95. With the best performing threshold of 0.9 this approach classifies 82% of the websites in the *train_dataset* correctly. In addition to the training accuracy this threshold is used to classify all websites in *text_dataset* an yields a test accuracy of 92% with three false positives and one false negative.

In order to configure the thresholds and to observe how the two approaches perform on the test websites the Streamlit application *test_subtree_scorer_application* can load the two different approaches. It allows to go through the test set and shows the highest scoring subtree within each sample website. Furthermore the threshold can be configured which needs to be surpassed by a subtree text to classify the website as containing DSGVO text. After selecting a threshold the resulting test accuracy is given. That is, how many of the 50 sample websites are classified correctly by comparing the score of their highest scoring subtree with the selected threshold. Sample groundtruth labels and predicted labels are listed to help find edge cases. This reveals that for all three websites, that are falsely classified as positive by the **TFIDF+SVM** approach, the subtree text that was identified as DSGVO text has less then 13 tokens (See fig. 10). This observation will help to improve accuracy on the final module7 by setting a minimum subtree text length and ignoring smaller subtrees. At this point the SVM based approach beats the 90% accuracy of the cosine similarity based baseline approach by 2%. However it is suspected that the accuracy can be lifted via configurations like the minimum subtree length.

This concludes the proof of concept for this project: An approach for identifying websites with DSGVO elements. So its has been proven, that the first main task of the final module can be accomplished. The second step after identifying a DSGVO containing website is to remove the DSGVO content from the HTML. Whether this works is not tested in the proof of concept. However it seems plausible that checking the subtree scores around the highest scoring subtree provides enough clues for this task as well.

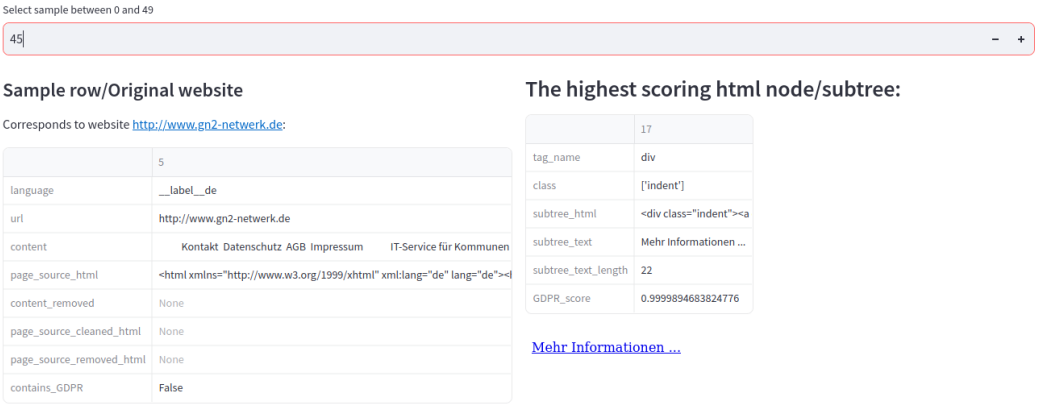

Figure 10: Screenshot from the *test_subtree_scorer_application*. The highest scoring subtree of sample 45 of the *text_dataset* receives a DSGVO score of over 99% from the SVM. Therefore the website is classified as positive instance although the website does not contain any DSGVO element (contains_GDPR=False). The app reveals that the highest scoring subtree only displays a text of three words: "Mehr Informationen ..."

The implementation details of the DOM traversal and scoring process can be found in *score_subtree*.

7 Kotlin Implementation of DsgvoAssisstant

In this section the **TFIDF+SVM** approach is implemented in Kotlin and extended to also remove DSGVO elements from HTML. As Python is the most common programming language for data science the project was done with Python up until this point. Python Jupyter Notebooks offer a fast iterative way for manipulating and analysing data. This makes it a popular choice for tasks like creating a dataset or training a machine learning model. Python is the leading language for data science and consequently there are many libraries for machine learning and natural language processing available. This is why Python was picked for the proof of concept phase of this project.

However the IT-Atlasref search machine and its crawler are written in Kotlin. The goal of this work is to provide a module that can be incorporated in the IT-Atlas crawler, cleans HTML of DSGVO elements and that way improves the search machine. Therefore the final module is required to also be written in Kotlin, in order to make maintenance and readability of the whole project easier.

7.1 Assets of the DsgvoAssistant

dsgvo_handler/POC/data_for_jvm/

Similar to the proof of concept, datasets are needed to fit a text vectorizer and a SVM. Therefore the already assembled data is prepared for usage in the DsgvoAssitant in *dev_dataset_for_jvm_impl*:

- The training set of text passages and DSGVO labels *svm_training_set* is stored to *data_for_jvm/fit_svm_training_data.json*
- All the extracted texts of *train_dataset* are stored to *data_for_jvm/fit_tfidf_documents.json*
- The *test_dataset* is stored to *data_for_jvm/test_dataset.json*

With the data ready a Kotlin project is set up at *dsgvo_handler/IR_Kotlin_Proj*. Then the datasets *fit_svm_training_data.json* and *fit_tfidf_documents.json* are copied over to the resources directory *Kotlin_Proj/src/main/resources/data/*. The testset *test_dataset.json* is stored at *Kotlin_Proj/src/test/resources/data/*.

Data classes that reflect the structure of all data used in the project are created in the *datamodel* directory inside the Kotlin project. For example the class *FitTfIdfDocuments* has just one value "documents" which is a list of Strings for loading *fit_tfidf_documents.json*. The "Serializable"

decorator from kotlinx is used for all data classes that correspond to a JSON file in the resources. That way the JSON can easily be parsed to an instance of the data class using the "Json.decodeFromString" function. To load these datasets and other assets (like pretrained models) the *DataProvider* class is created. It exhibits one get function per asset. For example, "getDocumentsForTfIdfFitting" returns an instance of class *FitTfIdfDocuments*.

7.2 Handling Text in the DsgvoAssistant

dsgvo_handler/IR_Kotlin_Proj/src/main/kotlin/architecture/document/

In the proof of concept it has been explained how and why text passages, from now on called documents, need to be preprocessed, vectorized and classified in order to assemble an HTML level classifier. Three interfaces and implementations of these document level actions can be found in the *document* directory.

document/preprocessor:

The *TextPreprocessor* interface contains one function "preprocessDocument" that takes a "Document", which is a type alias for a String, and returns a "TokenizedDocument", which is a type alias for a list of Strings. Its implementation *TextPreprocessorGermanBase* takes two constructor parameters. A pretrained model of "TokenizerME" from the Apache Opennlp (Apache, 2010) library and a list of German stopwords. Both assets can be found in the resources at *resources/models/opennlp-de-ud-gsd-tokens-1.0-1.9.3.bin* and *resources/data/german_stopwords.json* and are accessible via the *DataProvider* methods "getTokenizerModel" and "getGermanStopwords". This list of stopwords is downloaded from a GitHub repository and the tokenizer model from the Opennlp website. Once instantiated, the "preprocessDocument" method of *TextPreprocessorGermanBase* handles a document as follows. First the String is split into a list of individual tokens with help of the loaded tokenizer model. Then all tokens that cannot be encoded in UTF-8 are removed. Next all tokens that appear in the list of German stopwords get dropped. Finally the SnowballStemmer from the Apache Opennlp library is used to stem the tokens. The resulting list of Strings is returned as the "TokenizedDocument".

document/vectorizer:

The *DocumentVectorizer* interface provides three methods and is implemented by the class *TfIdfVectorizerCustom*. This class takes a *TextPreprocessor* as constructor parameter alongside an instance of the data class *FitTfIdfDocuments*, which holds a list of documents to calculate the inverse document frequencies from. Alternatively a map of tokens and their precalculated inverse document frequency can be handed instead of the *FitTfIdfDocuments* instance. The first method, "preprocessAndVectorizeDocument" takes a document and returns its "DocumentVector", which is a type alias for the underlying DoubleArray. To do so the *DocumentVectorizer* turns the document into a list of tokens and an instance of *TFIDF*, that either calculated the inverse document frequencies on instantiation or was given precalcuated ones, is used to obtain the DoubleArray that is the "DocumentVector" representation of the input text. Note that the implementation of TF-IDF developed in this project does not add new documents to its calculation once it was initialized. The second method "getVocabulary" returns a list of all tokens that are considered when calculating the vector of a text. The third method "getTextSimilarity" takes two documents, calculates their vectors with the method "preprocessAndVectorizeDocument" and returns the cosine similarity of the two vectors. The later two methods are only used for testing, while "preprocessAndVectorizeDocument" is a core function and mandatory for the DsgvoAssitant to work.

document/classifier:

The *DocumentDsgvoClassifier* interface specifies classifiers that were trained to classify a vector that represents the plain text of some document as either DSGVO related or not. Therefore it exposes the "getDsgvoPredictionForDocumentVector" function, which takes a "DocumentVector" and returns an instance of the *Prediction* data class. That data class contains the Double value "isDsgvoScore" and the Boolean "isDsgvoPrediction". For the classification three Java SVM libraries are considered. For each library an implementation of the *DocumentDsgvoClassifier* interface is created. The SVM from the Liblinear (Fan et al., 2008) library is chosen over the others without dedicated testing or comparing. The test classes show that the other two implementations are at least capable of classifying two test samples correctly as well. The *DsgvoAssitant* uses the

DocumentDsgvoClassifierLiblinearSVM implementation of the *DocumentDsgvoClassifier* interface. The classifier takes a *DocumentVectorizer* as constructor parameter. Furthermore it either needs training data and SVM hyperparameters or a pretrained SVM model on instantiation. If no pretrained model is provided the "train" function of Liblinear is used to fit a *Model*, which is then assigned to the "fittedSVM" field of the classifier. Otherwise the "fittedSVM" is directly set to the pretrained model. Calling "getDsgvoPredictionForDocumentVector" just passes the vector through the "fittedSVM", which returns a single Double value. A negative value denotes that the input vector belongs to class "-1", which in this case means no DSGVO text. A positive value indicates that the vector represents a DSGVO related text. Note that the Liblinear SVM behaves different to the SVM used in the Python proof of concept, which returned one probability value per class. The steps of preprocessing, vectorizing and classifying are chained together in the "getDsgvoPredictionForDocument" function of the *DocumentDsgvoClassifier* as depicted in fig. 11.

For evaluation of this SVM based classifier the *DocumentDsgvoClassifierModelSelection* is created in the test directory. It hosts the "crossValidation" method which takes three SVM hyperparameters and performs 5-fold cross validation over the training set. That is, the training set of 250 samples is split randomly into five chunks. Then the SVM is trained five times, each time excluding one of the chunks from training data so it can be used for validation. In each run the accuracy on the 50 validation samples is recorded as well as the true-positive-rate and the true-negative-rate, so that one cross validation call produces a list of five evaluation results.

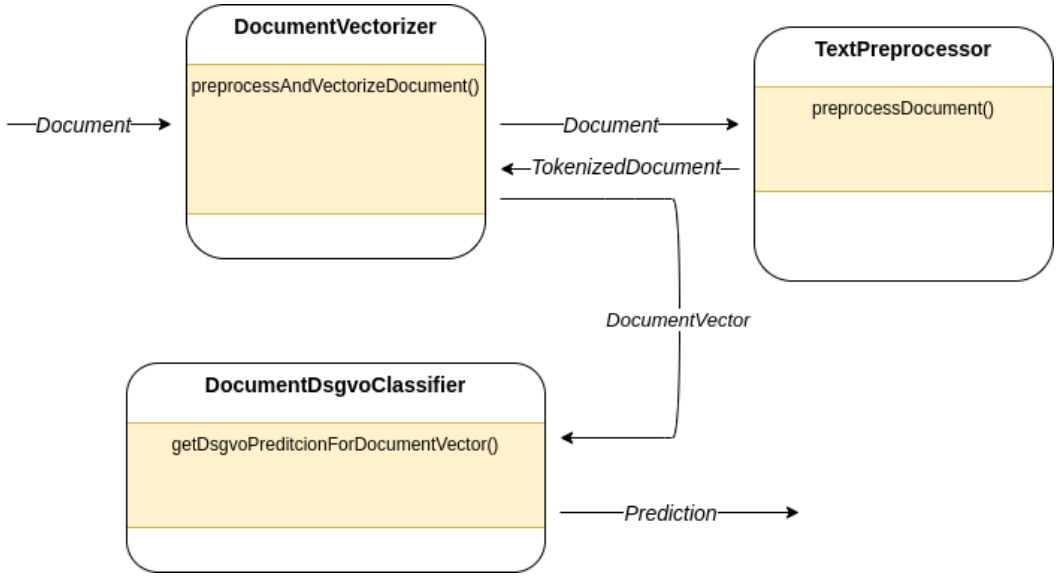

Figure 11: Sketch of the document level classification pipeline. In order to classify a plain text instances of the three interfaces *DocumentVectorizer*, *TextPreprocessor* and *DocumentDsgvoClassifier* need to interact in the displayed way. Yellow boxes indicate the methods of each interface that need to be called.

At this point the text classification pipeline from the proof of concept is realized in Kotlin. How the three described interface play together in order to classify a text passage as either DSGVO related or not is sketched in fig. 11. This concludes the Kotlin implementation of document level actions, that is functionalities that are concerned with a plain text. In the following the classifictaiton pipeline depicted in fig. 11 is used as a building block for handling whole HTML files.

7.3 Handling HTML in DsgvoAssistant

dsgvo_handler/IR_Kotlin_Proj/src/main/kotlin/architecture/html/

A pipeline for identifying DSGVO related documents was described above. In this section three interfaces for treating DSGVO elements on HTML level are introduced. They are responsible for handling, classifying and cleaning HTML DOM trees.

html/handler

The *HtmlHandler* interface exposes functionalities for navigating within a DOM tree and for text extraction. "doWithSubtrees" is a function that takes an HTML as String input and uses the JSoup Java HTML parser to parse its DOM tree. Then it carries out a given function on the text of each subtree in the DOM. It returns a map of JSoup nodes paired with the result of their subtree text being fed into the function. The method "doWithSubtrees" is later called with *DocumentVectorDsgvoClassifer*.getDsgvoPredictionForDocument for scoring text as DSGVO related, so that it returns all nodes with the DSGVO score of the according subtree similar to table 4. Not all nodes qualify for being scored and considered further. Nodes with to short subtree text are filtered out and not returned. The minimal number of whitespace separated words the text of a subtree must contain to be tested is a parameter that can be used for tuning the whole DsgvoAssitant. It is set as "mintSubtreeTextLength" in the *DsgvoAssistantConfig*. Considering too short texts can lead to many false positive classifications, e.g. the text "Analyse" with just one word scores very high with the SVM classifier, but could appear on any none DSGVO containing website.

The next function of the *HtmlHandler* is "moveUpDomTreeUntilTextFulfills". It takes an HTML, a start node and a stop condition. From the start node check if the stop condition is fulfilled by the parent node. If not go to the parent node and do again. If the parent fulfills the condition, then the current node is returned as the node highest up in the tree hierarchy that still passes the condition. This is later used for navigation while determining what part to remove from an HTML. The last function "getVisibleTextOfHtml" is only used for testing.

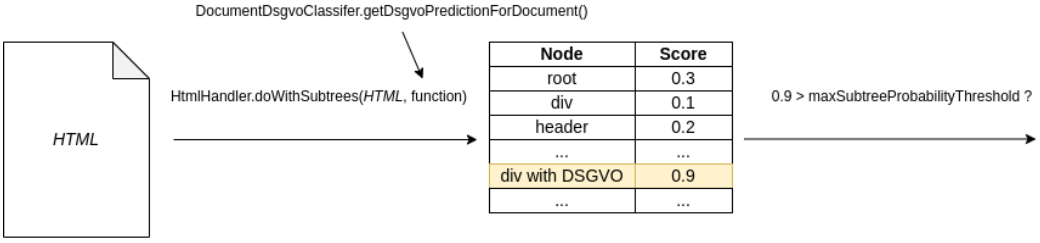

Figure 12: Depiction of the "containsDsgvo" function of the *HtmlDsgvoClassifer* interface as implemented by *MaxScoreSubtreeHtmlDsgvoClassifier*. The classifier iterates over relevant DOM subtrees using the *HtmlHandler* and scores each subtree with the *DocumentDsgvoClassifer* (as depicted in fig. 11). The best scoring node is determined (marked in yellow) and compared against a set threshold to obtain the final classification.

html/classifer:

The *HtmlDsgvoClassifier* interface provides one single function "containsDsgvo" which takes an HTML and returns true if the HTML is classified as containing DSGVO text. This interface is implemented by the *MaxScoreSubtreeHtmlDsgvoClassifier*. It is constructed with an instance of the *HtmlHandler* and a *DocumentDsgvoClassifer*. Handing an HTML String to the classifier's "containsDsgvo" method invokes the following pipeline. The *HtmlHandler* runs "doWithSubtree" with *DocumentDsgvoClassifer*.getDsgvoPredictionForDocument over the DOM tree. This gives a list of all relevant DOM tree nodes and the DSGVO score their extracted subtree text achieved. In this list the best scoring node is picked. If the score of this node lies above the defined threshold "maxSubtreeProbabilityThreshold", than the whole HTML is classified as positive instance. This implementation is depicted in fig. 12.

html/cleaner:

The *HtmlDsgvoCleaner* interface defines methods for treating DSGVO related text in an HTML that is already known to contain a DSGVO element. So we know that the highest scoring subtree exceeded the "maxSubtreeProbabilityThreshold". To clean the HTML this subtree could be trimmed from the DOM tree. Examining the scores of all subtrees as shown in table 4 shows that the accept button alone scores 0.98 while the whole div that wraps the DSGVO text and button scores 0.99. Removing the highest scoring subtree produces a cleaned HTML with no DSGVO related elements. However it is observed that the highest scoring subtree oftentimes corresponds to just a part of the whole DSGVO element. Imagine the accept button alone scores 0.99, because the word "accept" is very typical of DSGVO text. Furthermore the subtree corresponding to its parent div might not score

0.99, but just 0.9 because some words in child text are infrequently used in DSGVO pop ups. In this scenario only the accept button is removed and its parent div as well as its child text remain in the cleaned HTML.

To overcome this issue the highest scoring node is taken as starting point. From this current node the DOM tree is navigated upwards step by step. In each step the current node is either the root of the whole DSGVO element or situated somewhere within the DSGVO subtree.

Therefore the score of the parent node and its subtree text is checked. If the score is low, that suggests that not DSGVO related text makes up a large share of the parent's subtree text. This indicates that the current node is likely to be the root of the whole DSGVO element. If the parent score is still reasonable high, this indicates that the current node is probably just a part of the whole DSGVO element beside others. In this case the parent is chosen as the new current node. So far it is believed to be the node highest in the tree structure that only accommodates DSGVO children. From there the next parent is checked. This process of upwards propagation is realized in *BaseHtmlDsgvoCleaner* which implements the *HtmlDsgvoCleaner* interface. Its "removeDsgvo" method acts similar to the *MaxScoreSubtreeHtmlDsgvoClassifer* in that it scores all subtrees and selects the highest scoring node. From there it uses its "getRootOfDsgvoSubtree" method to perform the described upwards propagation. The defined "upwardsPropagationThreshold" determines whether a parent node still has a sufficiently high DSGVO score or not. Once the root of the DSGVO subtree is found the corresponding subtree is removed from the DOM tree and the cleaned HTML is returned. To summarize, "removeDsgvo" takes an HTML String and returns the cleaned HTML String. The second method defined by the *HtmlDsgvoCleaner* interface is "extractDsgvo". It works similar to "removeDsgvo", but returns the HTML of the DSGVO subtree instead of the cleaned HTML.

7.4 Assembling the DsgvoAssitant

dsgvo_handler/IR_Kotlin_Proj/src/main/kotlin/architecture/DsgvoAssistant

At this point all pieces are ready to assemble the *DsgvoAssitant*. This class ties together all the loose ends from the components described before and is the main access point of this project. When initialized the *DsgvoAssitant* class takes an instance of the configuration data class *DsgvoAssistantConfig*. It creates instances of the before mentioned components and assigns them to its member variables in the following order:

1. It start by creating a *DataProvider*.
2. The *DataProvider* is then used to load a pretrained tokenizer model and German stopwords need to initialize the *TextPreprocessor*.
3. Then the *DocumentVectorizer* is constructed with the *TextPreprocessor* and precalculated inverse document frequencies from the *DataProvider* as constructor arguments.
4. The *DocumentDsgvoClassifier* takes two arguments. The *DocumentVectorizer* and a pretrained SVM from the *DataProvider*.
5. The *HtmlHandler* takes the *DsgvoAssistantConfig* to get instantiated.
6. With the configuration, the *HtmlHandler*, the *DocumentVectorizer* and the *DocumentDsgvoClassifier* an instance of *HtmlDsgvoClassifer* is created.
7. The *HtmlDsgvoCleander* is created with the same constructor arguments.

When the *HtmlDsgvoClassifer* and the *HtmlDsgvoCleander* are bootstraped the *DsgvoAssistant* provides three public methods.

The first, "containsDsgvo" just calls the *HtmlDsgvoClassifier* to determine whether the given HTML String contains DSGVO text (fig. 13 left).

The second method, "removeDsgvo" uses the *HtmlDsgvoClassifier* to determine whether cleaning is needed for the given HTML String. If so the *HtmlDsgvoCleaner* removes the DSGVO element from the HTML and the cleaned version is returned. If no cleaning is necessary the original HTML String is returned (fig. 13 middle).

The last method is "extractDsgvo". If the *HtmlDsgvoClassifier* predicts the HTML to contain DSGVO content, than the *HtmlDsgvoCleaner* is used to extract the HTML String that corresponds to the whole DSGVO element (fig. 13 right).

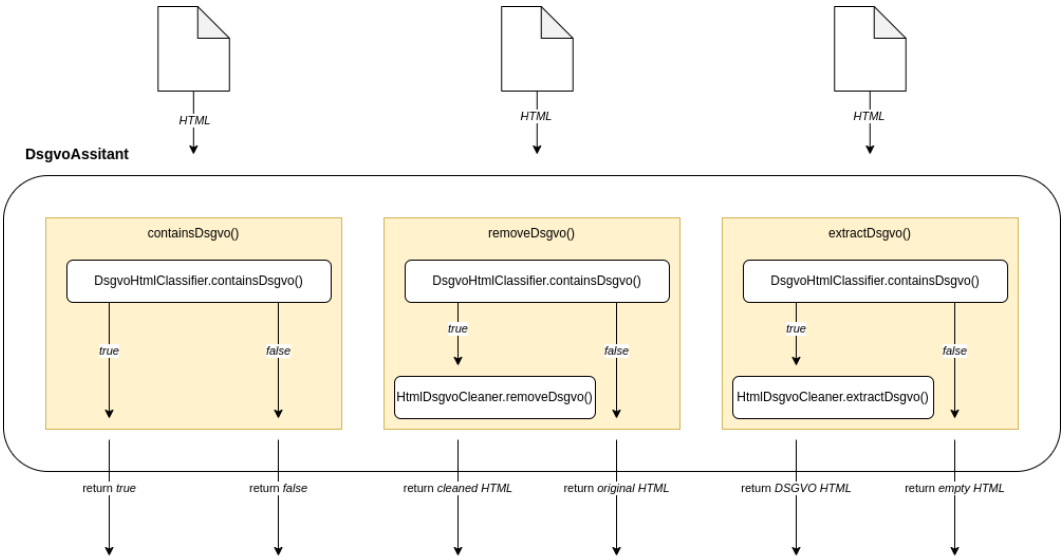

Figure 13: High level overview of functionalities provided by the *DsgvoAssitant*. Each of the three public methods is depicted in a yellow box.

Parameter	*Default Value*
maxSubtreeProbabilityThreshold	0.2
upwardsPropagationThreshold	0.1
minSubtreeTextLength	10.0

Table 5: The three configuration parameters of the *DsgvoAssistant* and their default values.

As mentioned before, the *DsgvoAssitant* takes an instance of the data class *DsgvoAssistantConfig* as single constructor argument. This configuration determines the three high level parameters of the *DsgvoAssistant*.

- "maxSubtreeProbabilityThreshold": The minimal score at least one of the subtrees of an HTML must receive from the *DocumentDsgvoClassifier* so that the HTML is classified as containing DSGVO text. The *HtmlDsgvoClassifier* compargvo".
- "upwardsPropagationThreshes subtree score to this parameter in order to return a boolean predicition in "containsDsold": The minimal score the parent of a DSGVO classified subtree must receive so that this parent and its subtree is now identified as DSGVO all together. The *HtmlDsgvoCleaner* uses this value to decide when the root element of the DSGVO subtree is reached.
- "minSubtreeTextLength": The minimal number of whitespace separated words the text of a subtree must contain to be considered. Considering texts that are too short leads to many false positive classifications, e.g. the text "Analyse" with just one word scores very high, but could appear on any none DSGVO containing website. The *HtmlHandler* skips nodes with a subtree text shorter then the set value when traversing the DOM tree.

The values that were found to work best are set as the default values of the *DsgvoAssistantConfig* and are shown in table 5. Note that other hyperparemeters like the regularization parameter C of the SVM have been optimized on the training data and are comprised in the pretrained model that is loaded by the *DsgvoAssistant*.

To check if the *DsgvoAssitant* is working properly *DsgvoAssistantText* provides several JUnit tests. The first test functions loads the HTML of a positive instance, feeds the HTML to the "containsDsgvo" method and checks that the prediction is correct. An identical test exists with a negative instance. These two tests have proven to be useful for debugging and sanity checks.

		True Label		
		Positive	Negative	Total
Predicted Label	Positive	27	0	27
	Negative	2	21	23
	Total	29	21	50

Table 6: The confusion matrix obtained from the predictions of the *DsgvoAssistant* on the 50 test websites.

"testAccuracyOnTestSet" loads the test set of 50 samples from *Kotlin_Proj/src/test/resources/data/test_dataset.json*. For each sample in the test dataset the "page_source_html" is fed to the *DsgvoAssistant*. The prediction returned by "containsDsgvo" is compared to the groundtruth label in the test dataset column "contains_GDPR". With the default configuration values the *DsgvoAssistant* achieves an accuracy of 96%, so 48 of the 50 samples get classified correctly. So the assistant reliably determines whether the HTML contains a DSGVO element or not. Furthermore the true-negative-rate on the testset es 1.0.

In order to analyse how well the assistant works its two methods "removeDsgvo" and "extractDsgvo" are executed on each test sample when running the test "runAssistantFunctionalitiesOnTestSet". The cleaned HTML and the readable text of the removed HTML get stored. The resulting table is the test dataset extended by two columns "content_removed_assistant" and "page_source_cleaned_assistant". This table is stored to *Kotlin_Proj/src/test/resources/data/processed_test_dataset_<date>.json* and is later used to examine the assistant's performance.

8 Evaluation of the DSGVO Assistant

Data and scripts for evaluating the Kotlin *DsgvoAssitant* are located at *Kotlin_Proj_evaluation*. The outputs on the testset have been recorded when running the test "runAssistantFunctionalitiesOnTestSet" and are copied over to *Kotlin_Proj_evaluation/preprocessed_test_set_v2.json*. This table is loaded in the Notebook *dsgvo_assistant_evalutaion*. With the test set and the outputs of the test run the performance of the module is evaluated. In addition to the accuracy of 96% the confusion matrix (table 6) is calculated.

It shows that no website was falsely classified as containing a DSGVO element, resulting in a true-negative-rate of 100%. Moving the threshold of how high the SVM score needs to be in order to classify a subtree a DSGVO element(maxSubtreeProbabilityThreshold) allows to tune the assistant to rather classify negative instances as positive or the other way around. Considering the overarching goal of using this assistant in a web crawler it seems to be a reasonable choice to optimize the true-negative-rate. Falsely classifying a website with DSGVO pop up as not containing DSGVO has the effect, that the HTML gets stored or indexed without cleaning. So in this case the crawler acts as it would without the assistant. On the other hand, classifying a website without DSGVO pop up as containing DSGVO content would make the assistant remove elements from the HTML that display website specific content. This is assumed to be detrimental for search performance, as the assistant might remove content relevant to search queries. Consequently the parameters in *DsgvoAssistantConfig* were chosen to minimize false positive classifications.

The Streamlit app *dsgvo_assistant_evaluation_app.py* allows to review one of the 50 test samples at a time. For the currently selected index the manually removed DSGVO text is displayed side by side with the text the *DsgvoAssistant* removed from this website. Furthermore the lengths of the two text are compared. If the two text lengths match it seems likely that the assistant removed exactly the same text that was removed manually on dataset creation. As fig. 14 reveals, the text lengths almost match for most samples. This suggests that the assistant would perform similar to a human on the task of removing DSGVO elements from the HTML. Further more the bar chart sheds light on the two falsely classified samples. For the websites at index 25 and 26, the assistant did not remove any content, while short DSGVO texts were removed manually. Reviewing these samples in the app, shows that both DSGVO text are shorter then the "minSubtreeTextLength" (see fig. 15)and therefore this text is never scored alone. This suggests that reducing the minimum length would further enhance the accuracy on the test set.

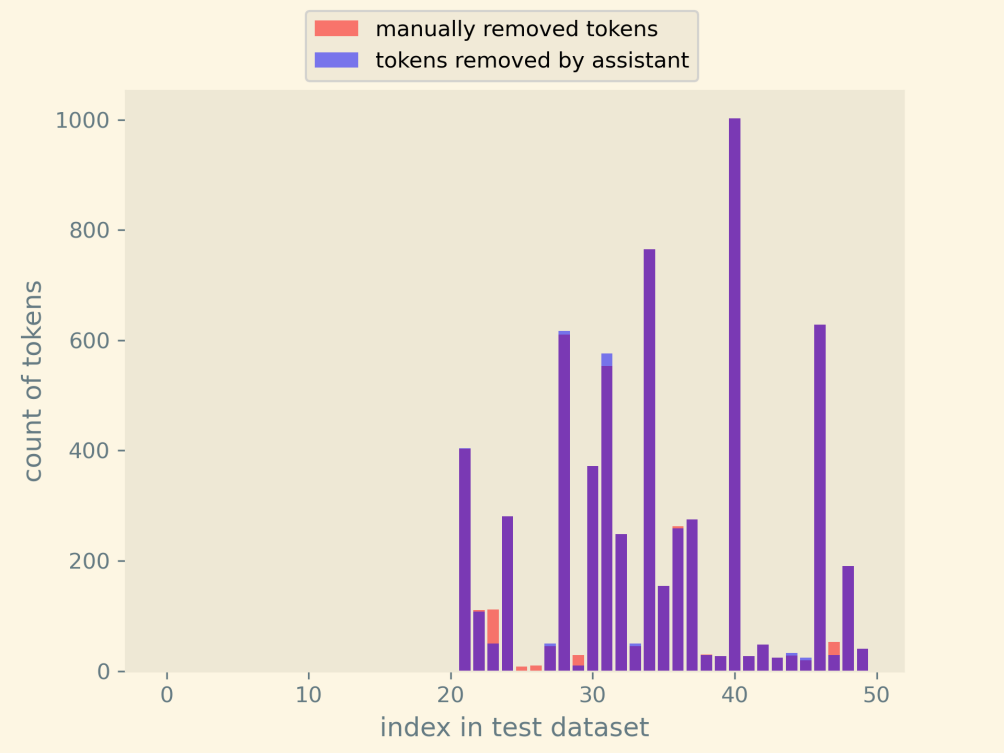

Figure 14: For each of the 50 test websites the number of tokens removed manually is depicted as **red** bar. The number of tokens removed by the assistant is given as **blue** bar. **Purple** indicates that the two bars are overlapping. The first 21 indices are negative samples and the remaining 29 positive samples. For seven samples more text was removed manually (red peak) and for six samples the assistant removed more text (blue peak).

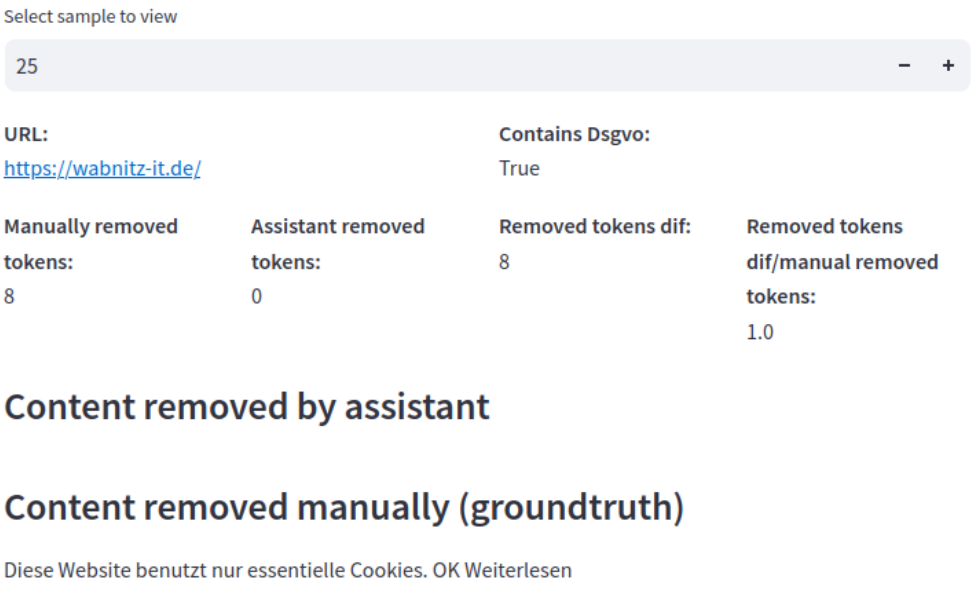

Figure 15: Screenshot of the *dsgvo_assistant_evaluation_app* with test sample 25 selected for inspection. The sample is a positive instance (Contains Dsgvo: True), but the assistant removed 0 tokens, compared to the 8 token DSGVO text removed manually.

9 Discussion

In this section shortcomings of the presented *DsgvoAssitant* are discussed as well options for improving the module in the future.

9.1 Effects on Search Accuracy

Overall the proposed *DsgvoAssitant* presents a practical option for cleaning the website of HTML before indexing. The motivation for creating such a module is to increase search results. As the analysis in section 4.4 showed, removing DSGVO elements cannot be expected to reduce storage because DSGVO text only makes up a 0.0095% of all text in the dataset. On the other hand it seems reasonable to assume that the cleaning can improve search accuracy, because DSGVO text was found to occupy up to 80% of website content (see fig. 6). However this beneficial effect on search accuracy is just a hypothesis at this point and has not been tested. In order to verify this effect the same search algorithm needs to be evaluated twice. First on a index that was created without any cleaning and then again on a index that was created from the first index by cleaning each HTML with the *DsgvoAssistant*. Only if the search results are notably better with the second index, than it can be said, that removing DSGVO text before indexing improves search quality.

9.2 Text Vectorization

In order to vectorize text passages of different subtrees the *DsgvoAssistant* uses the TF-IDF approach. The vectorzier is fitted on all texts of the *train_dataset*. Consequently every token that appears in these samples is included in the vocabulary of the vectorizer if it is not filtered out during preprocessing. This leads to a large vocabulary and therefore document vectors with high dimension. The vectorizer that was fit for the proof of concept had a vocabulary of 27637 tokens. So each vector consists of 27637 values. Furthermore this allows to include company names or city names in the vocabulary which could allow the SVM to easily overfit to the training data (by just learning a map of company name to label). However, this overfitting problem was not observed in the conducted tests and is more of a theoretical concern. Refining the implemented text preprocessing pipeline and TF-IDF algorithm could decrease vocabulary size and reduce the chance of overfitting. To make the vectorization more robust one could also try Transformer based approaches as alternative to TF-IDF vectorization. These approaches have been shown to capture the semantic meaning of text accurately and represent text as shorter vectors.

9.3 Module Parameters

As described in section 7 the *DsgvoAssistant* is parameterized by the three values in *DsgvoAssistantConfig*. The default values were set according to observations on the test dataset. Starting with a guess they were adjusted to achieve better results on the test samples. Compared to tuning the hyperparameters of the SVM with cross validation the module parameters were selected without any strategy, simply through manual trial and error. This is far of from best practice and implementing a repeatable procedure for this selection is necessary to produce trustworthy results. Furthermore the test set of 50 samples was used to select the module parameters, so that the reported 96% accuracy are not obtained on a clean test set. The 96% should rather be interpreted as the best achieved validation accuracy while testing the *DsgvoAssitant* on a test set that was not used for any training or tuning is yet to do. This questions the generalizability of the proposed module.

9.4 Python Project Structure

Another deviation from best practice is found in the Python proof of concept. As the different subdirectories were created chronologically in different phases they contain redundant code. For example the *GDPRDataset* class was used in several steps. To do so it is just copied into each subdirectory where it is needed. This leads to redundant and hard to maintain code. Code that is used in several steps should be tied together as a Python package and made available in all these subdirectories.

9.5 Hidden Website Content

As described in section 2 the approach followed in this work is to edit HTML directly. While this is assumed to be faster the navigating websites with a web driver, it also comes with possible pitfalls. The proposed solution does simply cut out the HTML responsible for displaying the DSGVO element, it does not interact with the user interface. It is possible that websites only load content after data privacy option were accepted. In this case, the *DsgvoAssistant* misses all content that is hidden before clicking "accept". This problem was never observed during dataset creation and was therefore ignored.

10 Lessons Learned

In this section I reflect on the development process and note some lessons learned.

A first point of critique is that I got focused on this text classification based solution early in the process and could have experimented with other approaches a little more. I think a Selenium based approach that just needs to identify the accept button would have been the more straight forward solution.

The next valuable lesson I learned is how useful custom apps are for data analysis. All the investigation could have been carried out all with Jupyter Notebooks, but having an easy way to browse through the datasets speeds up the development process enormously. With Streamlit small web apps can be created very quickly. After searching an indexing bug I created the first web app of this project, to review the dataset more easily. The *test_subtree_scorer_application* helped me to find the problem of short texts being scored high easily.

Another lesson I had to learn is that implementing a machine learning or NLP related project in Java can be trickier than expected. In Python libraries in the realm of machine learning are well known and robust. Many of the most used libraries are well documented and many tutorials on how to use them are available. With Java libraries quite the contrary is the case. A good example is that I did not find a reliable TF-IDF tokenizer to use in the Kotlin project and eventually implemented it myself. Finding a SVM library wasn't that easy as well. The ones I found were poorly documented and I did not find tutorials on how to use Libsvm. Finding GitHub projects that use this library helped me to understand how it works. This might be my biggest take away. Seeing how someone else incorporates a library into his project is a great starting point.

Before this project I had not worked with HTML and data crawling. So this was a great opportunity to get familiar with web data. I learned how to navigate the internet with the Selenium web driver and how to use the HTML parsers BeautifulSoup and JSoup. After looking at the HTML of 428 websites during dataset creation, I know have an idea of how websites are typically structured and feel better equipped for web related tasks.

References

Entwicklungsgeschichte der datenschutz-grundverordnung. URL https://edps.europa.eu/data-protection/data-protection/legislation/history-general-data-protection-regulation_de.

Apache. Opennlp, 2010. URL http://opennlp.apache.org.

D. Bollinger. Analyzing cookies compliance with the gdpr. 2021. URL https://api.semanticscholar.org/CorpusID:234957149.

R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *Journal of machine learning research*, 9(Aug):1871–1874, 2008.

A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

M. Nouwens, I. Liccardi, M. Veale, D. Karger, and L. Kagal. Dark patterns after the gdpr: Scraping consent pop-ups and demonstrating their influence. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20. ACM, Apr. 2020. doi: 10.1145/3313831.3376321. URL http://dx.doi.org/10.1145/3313831.3376321.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

M. F. Porter. Snowball: A language for stemming algorithms. 2001. URL https://api.semanticscholar.org/CorpusID:59634627.

L. Richardson. Beautiful soup documentation. *April*, 2007.

Declaration of Authorship

Ich erkläre hiermit gemäß § 9 Abs. 12 APO, dass ich den vorstehenden Projektbericht selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Des Weiteren erkläre ich, dass die digitale Fassung der gedruckten Ausfertigung des Projektberichts ausnahmslos in Inhalt und Wortlaut entspricht und zur Kenntnis genommen wurde, dass diese digitale Fassung einer durch Software unterstützten, anonymisierten Prüfung auf Plagiate unterzogen werden kann.

Bamberg, February 3, 2024

(Ort, Datum) (Unterschrift)