Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Bc. Dominik Janata

Web Page Data Figure Finder

Department of Software Engineering

Supervisor of the master thesis: prof. RNDr. Peter Vojtáš, DrSc.
Study programme: Informatics
Study branch: Software Systems

Prague 2016
I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ....... date ............ signature of the author
Abstract: The thesis treats automatic extraction of semantic data from Web pages. Within this broad problem, it focuses on finding values of data figures within the page presenting certain entity (e.g. price of a laptop). The main idea we wanted to evaluate is that a figure can be found using its context in the page: the words that surround it and values of the attributes of the containing HTML tags, class attribute in particular.

Our research revealed there are two types of contemporary solutions of this problem: either the author of the Web page must inline semantic information inside the markup of the page or there are commercial tools that can be trained to parse a particular page format (targetting pages from a single Web domain).

We examined the possibilities of developing a general solution that would – for given entity – find its properties across the Web domains using text analysis and machine learning. The naïve algorithm had about 30% accuracy, the learning algorithms had the accuracy between 40 and 50% in finding the properties. Despite the accuracy is not acceptable for a final solution, we believe it confirms the potential of the idea.

Keywords: Web pages data extraction
I want to thank my supervisor, professor Peter Vojtáš, for the consults and advices on the thesis concept and aiming.

My special thanks go to doctor Jan Dědek who helped me with the usage of the GATE tool and whose advice finally enabled me to get the crucial machine learning working.

I also want to thank Ivana Lukšová for her helpful consults on GATE.

And finally, I would like to thank my girlfriend, my parents, and my friends who helped me with the work by technical consults and advices and by moral support.

I believe the Facebook page LOTR University Memes also deserves to be mentioned here, as it provided me very welcome amusement and kept reminding me there are many people carrying out similar toil in doing their theses outside.
Contents

Introduction ........................................... 5
  Motivation of the Thesis .................................. 5
  The Vision .................................................. 6
  The Assignment .............................................. 6
  Goals of the Thesis ....................................... 7
  Expected Outputs .......................................... 7
  Structure of the Thesis ................................... 7

1 Analysis ................................................. 9
  1.1 The Problem ........................................... 9
    1.1.1 User Viewpoint ......................................... 9
    1.1.2 Technical Viewpoint .................................. 9
  1.2 Research on Web Page Data Extraction ................... 9
    1.2.1 Taxonomy of Web Data Extraction Techniques ............... 10
    1.2.2 Natural Language Processing Tools ......................... 11
  1.3 Contemporary Projects .................................. 12
    1.3.1 Non-Profit Projects .................................. 12
    1.3.2 Commercial Solutions ................................ 13
  1.4 Concept of the Thesis .................................. 13
    1.4.1 Figures & Their Occurrences ............................ 14
    1.4.2 Data Collection ....................................... 14
    1.4.3 Experiments ........................................... 14
    1.4.4 Evaluation ............................................. 16
    1.4.5 Programming Part ..................................... 17
  1.5 Software ................................................ 17
    1.5.1 Functional Requirements ............................... 17
    1.5.2 Application for Annotating Pages ....................... 17
    1.5.3 The Server .......................................... 18

2 GATE .................................................. 23
  2.1 Introduction ........................................... 23
  2.2 GATE Trivia ............................................ 23
    2.2.1 The Terminology ....................................... 23
    2.2.2 Use Cases ............................................ 24
    2.2.3 Installation .......................................... 24
    2.2.4 Development Tools ................................... 25
  2.3 GATE GUI .............................................. 25
    2.3.1 Overview ............................................. 25
    2.3.2 Detecting and Highlighting Money in a Text ................. 26
    2.3.3 Matching Custom Regex ................................ 28
    2.3.4 Machine Learning ..................................... 29
  2.4 GATE Embed ........................................... 29
    2.4.1 Overview ............................................. 29
    2.4.2 Detecting and Highlighting Money in a Text ................. 30
    2.4.3 Matching Custom Regex ................................ 32
3 Implementation

3.1 Browser Add-On

3.1.1 Technology

3.1.2 Architecture of the Add-On

3.1.3 User Interface

3.1.4 Remarkable Implementation Details

3.1.5 Installation

3.1.6 Usage

3.2 The Server

3.2.1 Technology

3.2.2 Architecture of the Server Part

3.2.3 Remarkable Implementation Details

3.3 GATE Wrapper

3.3.1 Gate API

3.3.2 Interoperability

3.3.3 Figure occurrences

3.3.4 Annotated Document

4 Manuals

4.1 Server Installation

4.1.1 Prerequisites

4.1.2 Database

4.1.3 Web API

4.1.4 Console Application

4.2 Add-On Installation

4.2.1 Prerequisites

4.2.2 Installation

4.3 Using the Add-On

4.3.1 UI Description

4.3.2 Usage Scenarios

4.4 Using the console application

4.4.1 Examples

5 Experiments

5.1 Tuning Page Manipulation

5.2 Finding Correct GATE Configuration

5.2.1 Occurrence Annotations

5.2.2 Machine Learning

5.3 Data

5.3.1 Training Data

5.4 Naïve Matcher

5.4.1 XPath Rules

5.4.2 Close Words Rules

5.4.3 Results

5.5 GATE Matcher

5.5.1 Results on the whole data set

5.5.2 Results on the training data set
Introduction

Motivation of the Thesis

The amount of information available on the Internet is both inspiring and challenging. Although the Web was originally developed for convenient interchange of information between people, the format of the Web pages shows common, converging patterns. The existence of these patterns enabled processing the information algorithmically. A very good example of such processing is Google Search — Google Search is very effective in matching and linking keywords in open text.

In this thesis, we want to examine the patterns of Web pages dedicated to a particular entity; we picked e-shop page of a laptop and package tour (or vacation). A motivating example: on a laptop e-shop page, the product price is usually accompanied by one of the words “price”, “now” or “only”, and is often wrapped with a tag containing “price” as a word, e.g. $499.99. We believe that these typical words could lead us to detecting “$499.99” as the correct value.

Nowadays, the search engines are the main aggregators of information for human users. The fact that inspired us to perform research on algorithmic search for figures in Web pages is that the search engines – starting with Google Search as the most popular one – do not provide such figure-aware search. For example, imagine a query “notebooks cheaper than $500”. Google Search does not provide table across various resellers and manufacturers, containing type, price etc. No free tool does. The tools capable of extracting such information from the Web pages are limited to a particular datasource. And price comparators (Heureka.cz for the Czech republic) do not parse vendor pages, they query their APIs.

Although there is a vast amount of services performing certain subtasks of this problem — advertisement, social networks, social benchmarks, comparison shopping [2] we believe that there is space for a new tool. A tool that would become synonym of a universal entity catalog for the Internet.

This topic already has some history in our faculty: the master theses of D. Fišer [31], D. Maruščák [32] and J. Kýpet [33] also treated the problem of obtaining semantic information from Web pages. This thesis is, in a way, a response to the thesis of D. Fišer, which presented a new Web browser plugin that displayed semantic data inlined into the page – either by a developer or by users. We believe that this approach is not necessary, because it is possible to obtain the semantic data from the pages without extra annotation.

A remark to the terminology: initially, we decided to call the properties of the entities “figures” and to call entity types “topics” or “page topics”, emphasizing the viewpoint of the pages. During the course of the work, it turned out “property” and “entity type” would be better names, but we decided to keep the terminology to prevent inconsistency; despite that, it is possible both these terms occur in the text. We apologize the reader for the confusion.
The Vision

This thesis follows the vision of a service having the behaviour and interface of a Web search engine, but containing some catalogue summary of entities matched in the search results [27]. This would turn the Web search engine, among other new usages, into a universal price comparator, allowing result filtering upon any criteria that make sense for the entities matched in the results: laptop CPU performance range, meals served on package tours (all inclusive, breakfast...).

The result of the thesis is neither such search engine or enrichment of an existing search engine with the figure finding. This thesis evaluates the possibility of making such enrichment by testing the concept on the set of examples defined in the Analysis chapter.

The Assignment

The assignment of the thesis is the following:

“The aim of this thesis is to create a system for semantic search for information in the Web pages. It does not treat understanding the text as a whole, just particular figures. The figures the system will recognize in the text will be categorized by the type of the page.

Architectonically, the solution will have a client part and a server part. The client part will be implemented as an add-on to a Web browser and will serve for both searching for data in the page and for collecting the data. The server part will accept the data collected by the client and update the client in order to utilize the collected data to enhance the search.

The solution will have two layers. The basis will be creation of taxonomy pages we will detect the figures in, and definition of the particular figures. The figures will have an initial description how to recognize them in the text. This first part will enable search in the contents of the page based on the data type of the figure. This provides user some extra value by itself, as there is no such feature in the browsers currently.

This search based on data type serves to collect data for the second part defining the rules to recognize particular figures. To make rules more accurate than just data type of the figure, we will try the approaches described in [1]. These rules will be downloaded back into the add-on that will match and highlight the figures.

The thesis is experimental – it contains models; methods; a prototype; user experiments; data collected from multiple domains; various evaluations: accuracy and completeness, the level of automatization, the amount of expert intervention, the amount of user comfort.

The assignment and its orientation may be changed and further specified during the solution.”
The assignment thus defines the thesis as a research report. The research nature implies the possibility of assignment change, especially in the architecture and behaviour of the resulting software.

**Goals of the Thesis**

This thesis will provide overview of techniques relevant for implementing a universal automatic semantic annotation service. It will also provide a prototype implementation of the techniques and the results obtained by running the techniques on English-written Web pages of two types: laptop e-shop pages and package tours.

A particular attention will be drawn to the following idea: a fact existing on a Web page has its horizontal and vertical contexts. The horizontal context is its position in the text. Examining the horizontal context means performing text analysis. The vertical context is its position in the DOM: the ancestor and the surrounding tags in particular.

**Expected Outputs**

We are going to explore the current state of Web semantisation, figure value extraction: the contemporary techniques and available tools. We also expect to implement a system for collecting training data with the intent to learn extraction rules on them. The explored approaches will then be tested on a set of test data. The outputs of the thesis are thus expected to contain, apart from this document, pieces of software, experimental data and results of the experiments.

**Structure of the Thesis**

The thesis is divided into the following chapters:

The Analysis summarizes the context of the problematics, contemporary implementations, resources for our research and reasoning performed atop of these resources.

The GATE chapter introduces the tool with the respective name. GATE is a very complex tool and we chose it to perform most of the tasks of text analysis. This chapter should provide some understanding of how GATE works in order to make the subsequent chapters better understandable. The chapter was written as a tutoring text on the wish of the supervisor of the thesis, with the perspective of serving as background for his course of Web semantisation. For this reason, it might be written with more explanations and examples than necessary for this thesis.

The Implementation chapter describes the tools programmed to make the experimentation possible (or easier).

The Manuals chapter is a guide for installation and usage of the programmed tools.

The Experiments chapter describes the design and the results of the performed experiments.
The Evaluation chapter summarizes the results of the experiments.
The Conclusion chapter provides an overview of the thesis with our subjective interpretation of its course and results.
1. Analysis

In this chapter, we examine possible solutions of the problem and provide overview of the research having been done and some commercial solutions.

1.1 The Problem

1.1.1 User Viewpoint

Web search engines provide a universal, keyword-based search spanning the whole Internet, but they do not accept queries that require ordering or filtering the values of figures inside the page content, e.g. “Show me pages with laptops cheaper than $500”. On the other hand, there are tools that provide semantic information to the Web pages, but these are typically commercial tools satisfying the assignment “Monitor this website contents for me.” The typical use-case of these commercial solutions is either a company is monitoring its competition or a company is monitoring its own merchants (e.g. Lixto) (see section 1.3 below).

1.1.2 Technical Viewpoint

The information from the Web can be exchanged using any software able to communicate on the HTTP protocol, which means we cannot claim the Web browsers are the only programs handling information from the Web; and even within Web browsers, there are other ways of presenting information to humans than just HTML (XML with XSL, for example). However, an HTML document viewed in a Web browser can be considered a general, typical use case, which is the reason we will focus on it from now on.

HTML on the Web can be considered a source of universal, common human knowledge. Ability to transform the information from the inline, plain-text nature of HTML to semantic format means ability to understand, represent and query the whole human knowledge. The contemporary dominant organizers and filters of human knowledge are Google and Wikipedia. Looking at the predominant approach they follow: they both match keywords from the query with contents of indexed pages. This approach turns pages into sets of strings having a value function to express the importance of the string within the page and a dynamic value function to express the importance of the string with respect to the particular query. It means this approach ignores the possibility that the pages in the result set contain figures of an entity (or a set of entities) that can be ordered or filtered. Detection of these figures is the problem examined in this thesis.

1.2 Research on Web Page Data Extraction

The basic document of this overview is the survey done by A. Laedner and collective [1]. Despite this survey was created in the year 2002 (14 years ago by the time of writing this thesis), it still covers most of the techniques available today, which is confirmed by a following study from E. Ferrara [2].
1.2.1 Taxonomy of Web Data Extraction Techniques

Languages for Wrapper Development

A definition of a wrapper in the context of Web data extraction is provided by C. A. Knoblock [3]: “A wrapper is a piece of software that enables a semistructured Web source to be queried as if it were a database.”

The wrappers were further broken down into several categories divided into two groups: semi-automatic wrapper generation and automatic wrapper generation. A semi-automatic wrapper generation means the user must annotate the pages of the data source (some representative subset at least), and the wrapper generator infers the wrapper from them. An automatic wrapper generation does not need user interaction.

The Ferrara survey [2] provides two types of semi-automatic wrapper generators: visual extraction and spatial reasoning. Visual extraction means there is a tool that renders the page for the user, and the user picks the correct occurrences. Spatial reasoning is a less common concept, it features reading the figures via OCR from the snapshot of the page.

NLP

Natural Language Processing, or NLP, means exactly what it says - it means analysis of a human-readable text. The Laedner survey contains the following definition for this family of techniques [1].

“Natural Language Processing (NLP) techniques have been used by several tools to learn extraction rules for extracting relevant data existing in natural language documents.”

HTML-Aware Tools

HTML-Aware tools try to understand the structure of the HTML page, matching the interesting data from HTML tags and their attributes. The definition provided by Laedner [1] and accepted by Ferrara [2] is the following:

“Before performing the extraction process, these tools turn the document into parsing tree, a representation that reflects its HTML tag hierarchy. Following, extraction rules are generated either semi-automatically or automatically and applied to the tree.”

Modeling-Based Tools

The modeling-based tools are defined as follows [1]:

“This category includes tools that, given a target structure for objects of interest, try to locate in Web pages portions of data that implicitly conform to that structure. The structure is provided according to a set of modeling primitives (e.g. tuples, lists etc.) that conform to an underlying data model. Following, algorithms similar to those used by the wrapper induction tools identify objects with the given structure in the target pages.”
Ontology-Based Tools

The Laedner survey defines ontology-based tools as follows [1]:

“Given a specific domain application, an ontology can be used to locate constants present in the page and to construct objects with them.”

Summary

The subject of the thesis is matching properties of entities in Web pages. This definition of the subject would, by itself, conform to either modelling or ontology based methods. However, we do not intend to trace some implicit relations given by the definition of the entities, we will search for the figures separately and the definition of the entity is only a wrapping object that puts the figures of the entity into a single package.

From the tool categories, this thesis will utilize natural language processing and HTML-aware tools, and also conforms to semi-automatic wrapper generation - there will be some manual annotation necessary to learn to extract figures from a page; and extracting the figures conforms to the definition of a wrapper generator: “A wrapper is a piece of software that enables a semistructured Web source to be queried as if it were a database.”

1.2.2 Natural Language Processing Tools

Laedner survey

The Laedner survey [1] lists tools for natural language processing. In particular, it mentions RAPIER, SRV and WHISK. We failed to find these tools as downloadable, runnable software, the results of Google Search and Microsoft Bing provide links to articles mentioning these tools [5] [6] [7] [8] [9] [10]. These articles were usually written before the year 2005 which makes us believe these projects have been abandoned. (Remark: The fact the tools are missing in the search engine results does not imply the projects were actually abandoned by itself. It is valid to believe the other tools coming up in our results are more relevant as there are more other pages referencing them.)

GATE

GATE as an acronym stands for “General Architecture for Text Engineering”. It is a tool developed in the University of Sheffield for text analysis. Text analysis is one of the two approaches we want to use in this thesis, we will try to make GATE to cover this first approach.

The basic entities featured in GATE are processing resources or PRs and language resources or LRs. A language resource is the input data - a document or a set of documents (called corpus in GATE terminology). Processing resource is a software component that processes language resources and annotates them. An annotation has a name and is assigned to a portion of the language resource. Typically, a processing resource is able to assign several annotations (e. g. Location, Address, Money) and assigns it to places they occur in the text.

Apart from this static text analysis, GATE is also able to perform machine learning. Machine learning is also a processing resource and works as follows:
• The user provides a set of annotated documents - these is the training set of data. The documents may contain various annotations but there is one annotation declared as the one I want to train for.

• The machine learning is run on the training set and generates a file containing a set of rules.

• The machine learning is run using the generated file with set of rules and processes the document using the rules.

A big advantage of GATE is it has an API – a Java library.

These features of GATE cover what we require form the text analysis part: the static annotations for figures like money can provide hints for figure occurrences, the machine learning can infer rules for identifying the correct occurrences. GATE is distributed with many processing resources (machine learning being one of them), which means we get a lot of functionalities out-of-the-box.

See chapter 2 for more detailed information about GATE.

Other Tools
Initially, we were querying for “Web Data Extraction”, as this was the point we were approaching the problematics. After experimenting and evaluating GATE, it turned out that querying “Web Data Extraction” prevented some relevant tools from being in the search results. When searching for variations of the “NLP tool” query, we found MITIE, opensource.com lists the following five more: Stanford’s Core NLP Suite, Natural Language Toolkit, Apache Lucene and Solr, Apache NLP and, ultimately, GATE. Examining these tools as alternatives to GATE would be a valid follow-up of this thesis.

1.3 Contemporary Projects

1.3.1 Non-Profit Projects

Search Computing

The introduction of the Search Computing project from its homepage:

“Search Computing (Seco) is a project funded by the European Research Council (ERC), responding to the 2008 Call for ”IDEAS Advanced Grants”, a program dedicated to the support of investigation-driven frontier research.

Challenge

Search computing focuses on building the answers to complex search queries like ”Where can I attend an interesting conference in my field close to a sunny beach?” by interacting with a constellation of cooperating search services, using ranking and joining of results as the dominant factors for service composition. By leveraging the peculiar features of search services, the project devises query approaches, execution plans, plan optimization techniques, query configuration tools, and exploratory user interfaces.”
The name Search Computing implies this project is closely related to our problem of semantic search. A closer look on the deliverables of the project [12] shows that the major results of the projects are tools and techniques linking queries to search tools that already exist. Which means it provides no extra value to search results, it merely directs the query to the tool it evaluates as best fit.

1.3.2 Commercial Solutions

Google Custom Search

This tool belongs to the group of tools requiring the page to be annotated with semantic information. If it is, Google Search is able to extract. The following formats are supported [21]:

- PageMaps
- Microformats
- RDFa
- Microdata
- `<meta>` tags

Lixto

Lixto is an Austrian company that offers automatized extraction of semantic data from Web pages. The introduction text provided on its Web presentation is the following [28]: “Lixto extracts specific and precise strategic pricing data from the web to help travel & transport, consumer products and manufacturing clients drive operational performance and increase real-time competitive price visibility.”

The closer examination of their Web site [29] shows that the main business of Lixto is collecting price and product information about the competition of the customer.

Lixto is also mentioned in the Ferrara survey [2]: “Some commercial systems, Lixto for first but also Kapow Mashup Server (described below), include a Graphical User Interface for fully visual and interactive navigation of HTML pages, integrated with data extraction tools.”

“Languages for Wrapper Development:. Before the birth of some languages specifically studied for wrapper generation (e.g. Elog, the Lixto Web extraction language [Baumgartner et al. 2001a]) (...)”

This important part of this quote is Lixto employs language for wrapper generation as the technique or as one of techniques for finding semantic information; Lixto is also listed as an HTML-aware tool.

1.4 Concept of the Thesis

In the introduction chapter, we outlined the expected outputs of the thesis. In this chapter, we will examine the requirements implied by these expectations.
1.4.1 Figures & Their Occurrences

In the goals of the thesis, we declared we will have Web pages of two types: laptop e-shop pages and package tours. The figures that will be matched in laptop e-shop pages are price, memory size, disk size and CPU performance. The reason is we want to have laptop e-shop pages as sources of easily matchable figures; the syntax of all these figures can be matched by a regular expression and there is typically a single correct value for the whole page.

The figures that will be matched in package tours are the price, location, diet and destination. These figures are harder to match, because there may be multiple correct values (e.g. destinations and diets listed for each day in the itinerary of the package tour), moreover matching a destination means to query a large vocabulary or even database of values.

Up to this subsection, we were talking about figures as parts of strings, which is a very general view. The figures we declared to try matching, however, belong to a set of values that can be recognized by static text analysis (i.e. by their syntax):

- By a regular expression (e.g. memory size).
- By a vocabulary of possible values (e.g. location or diet).
- By a combination of both (e.g. price).

This gives us a narrowing of the problem: from matching “some” substrings in the text, we can preprocess the text by matching the values belonging to the data set of the figure and then pick fitting candidates among them. This process was already indicated in GATE section earlier in this chapter, because GATE is fit for this task. At this point it is necessary to emphasize that all machine learning algorithms will categorize figure candidates matched by GATE.

This also enables us to enrich the data collection application with highlighting the figure candidates for the user; the user would pick the correct match by clicking the candidate.

1.4.2 Data Collection

Our experiments will be performed on set of annotated Web pages. The intended data set for experiments is further described in the following subsection, Experiments; at this point we discuss the means of collecting the data only.

The training and testing Web pages will be chosen and annotated manually. We will develop a custom software for annotating the pages.

1.4.3 Experiments

We must decide the training data set, test data set and algorithms used to process them.

Training Data Set

By training data set we mean a set of pages used for a machine learning algorithm to infer rules for matching figures in the pages.
For both of the entity types (laptop e-shop page and package tours), we will have a respective set of annotated training Web pages. The question is what sources to use for each of the entity types: should all the pages be from a single domain or one page per domain? Or something in between?

Having all the Web pages from a single Web domain (e.g. laptop details from www.amazon.com) would mean the algorithm will be trained for that particular domain only. We would like to outline a universal approach. Which means the more domains and the more pages for each of them, the better. There is an opposite aspect, however: we will be debugging the tools and possibly looking for some anomalies in the pages, which requires a data set that is iterable manually.

Taking these two aspects into account, we decided to have 10 domains for both of the entity types and 3 pages per domain, with the possibility these numbers may be surpassed through the course of the work. 2 of the pages will serve for training, the third one for evaluation.

**Testing Data Set**

By testing data set we mean a set of pages used for measuring the accuracy of a machine learning algorithm.

There will be two testing data sets for each combination of figure and domain:

1. The whole set of pages of the domain including the training pages. This one will help to confirm the algorithm is able to match figures on the training pages.

2. The whole set of pages of the domain excluding the training pages. This one will help to confirm the algorithm is able to match figures outside the training data set.

**Algorithms**

We will prepare two “smart” solutions and one “naïve” solution. The first of the “smart” solutions will be the GATE machine learning. Because GATE machine learning is a black box for us, we will also develop our own “smart” solution relying on heuristics.

**Heuristics** The learning phase of our heuristics will mean we will compute statistics of the words occurring in the neighbourhood of the training figure occurrences: the average position relative to the occurrence and percentual incidency. Likewise, it will compute the words occurring in the class and data attributes of the containing tags.

When matching, the matcher will iterate over the candidate matches and compute a rating for each of them. The rating will be computed as follows:

1. The algorithm collects the words from the neighbourhood of the candidate.

2. For each of the words, it gets its relative position to the candidate, subtracts it from the average relative position of that word from the statistics and multiplies the quotient by the incidence of the word.
3. The algorithm collects the words from the tags surrounding the candidate up to given constant level.

4. For each of the words, it gets the relative position of the respective tag to the candidate, subtracts it from the average relative position of that word from the statistics and multiplies the quotient by the incidence of the word.

5. The algorithm sums the two ratings as a weighed average with weights taken from configuration.

The both machine learning algorithms will be run in two scopes:

- Domain-specific. For each domain in the data set, it will generate the rules specific for the particular domain.

- Universal. In this case, the training and testing data sets will be the union of training and testing data sets across all domains, respectively.

The naïve will have a constant rule for every property of every entity type and for each input, it will apply the rule for the respective property. There is some individual experience with the Web pages which induces some “common sense” regarding the definition what the right occurrence of a particular figure is. Example: “There are multiple prices on the page: discount, shipping price, price with VAT, price without VAT. However, the product price typically is preceded by the word ‘price’, ‘now’ or ‘only’. The tags wrapping the price also typically contain ‘price’ in the class name, because it is visually highlighted and the developers dedicate some CSS class to it.”

We want to demonstrate that the “smart” approaches are better, i.e. that the machine learning can find out relations that are more universal than these. The “naïve” solution is also expected to provide non-zero results, serving as proof of concept of the thesis.

1.4.4 Evaluation

The fundamental concept of GATE are the annotations. The input data for GATE machine learning are annotated documents, the output data of the machine learning are also annotated documents. A straightforward solution to evaluate correctness of the assigned annotations is to annotate the test data manually and to measure percentage of hits: how many of the manual annotations were “hit” by the machine learning annotations.

There is one problem: in some pages, some of the figures have multiple occurrences with different values. An easier example is a laptop e-shop page: there can be two occurrence of CPU performance, “2.2GHz” in one case and “2.2 GHz” in the other. Even more complicated situation is a package tour: on a round trip over Europe, there can be destinations “Prague”, “Vienna”, “Berlin” and, say, “Nurnberg”.

We will limit the evaluation to the question “Is the best match provided by the algorithm among the occurrences annotated on the Web page by the user?”
1.4.5 Programming Part

As a conclusion of the previous sections, we can see that it is necessary to implement custom software for collecting data and performing experiments. The next section contains a deeper analysis of the software.

1.5 Software

In the Introduction chapter, we outlined which goals will this thesis follow. In this section, we will provide a more detailed specification of the software we must implement to achieve the goals of the thesis.

1.5.1 Functional Requirements

We want to implement a service that will

1. Collect training data for the automatical annotation algorithms.
2. Show the figure matches and figure candidates in a particular Web page.
3. Store and manage this data.
4. Operate with 3rd-party software, the annotators.

1.5.2 Application for Annotating Pages

It is not our ambition to collect the training data automatically, so we will have to define the users who will annotate the pages and thus provide the training data to the system. The ideal situation is that any user can annotate the pages implicitly, as he browses the Web pages, and makes no extra effort. This would undoubtedly bring the most data reflecting the behaviour of regular users. However, this solution is out of the scope of this thesis (approaching the end users, convincing them to install a software that is essentially tracing their behaviour on the Web and so on).

This allows us to focus on users who have already been acquainted with the project and thus leave the application technical yet still comfortable enough to enable the users annotate the pages effectively.

Functional Requirements

The user who will be annotating the Web pages will be an advanced user, and the following functions should be present in the application:

- Displaying figures observed by the system.
- Marking occurrences of the figures on the currently showing page.
- Highlighting possible occurrence matches on the page (i.e. when marking a price, any pieces of text like “$4.99” will be highlighted.
- It is possible to mark an occurrence by either clicking a highlighted possible occurrence or by selecting a portion of text.
Displaying the matched figure value.

When highlighting a possible occurrence match, the most probable figure value will visually differ from the rest of the occurrences (i.e. it will be highlighted among the highlighted occurrences).

Data
The application will present possible occurrences of a figure and the most probable value of a figure to the user. The question is whether will the value and the occurrences be computed in the application or on the server. The values must be computed on the server, because the values are results of some machine learning or otherwise inferred from pages annotated manually before.

The initial idea on computing occurrences was they can be expressed as a simple regular expression (a number or a string from a small enumeration of possible values). Our initial experiments with GATE showed that many figures – money being the first example – are matched by dictionaries containing tens or hundreds of values. This implied that better solution would be to make GATE annotated possible figure occurrences on the server and the client will merely display them.

Technology
The natural tool for displaying and manipulating a Web page is a Web browser. The Web browser provide API for 3rd-party software called addons. This makes a Web browser addon a very straightforward solution of a software for collecting data.

Any alternative solution would have to coup with either rendering the page or making the user find the correct figure occurrences inside the HTML markup. Neither of these tasks seems to have so easy solution as the addon.

Another major question is where to store the collected data, what tools to use to process them and how to run the tools.

The browsers expose local resources in their addon APIs [4]. However, this would require anyone who would like to annotate pages to install the whole system including all the external tools, to set up the data storage and so on, while the only thing he actually needs is the plugin. Therefore, the plugin will communicate with the rest of the system through the Internet and the whole system will thus have the pattern of a client-server application.

These ideas bring us to the following elementary system architecture:

The “Core” module stands for a set of server components that are to be designed in the following section.

1.5.3 The Server
A task that is implied straightforward in previous section is an API implementation that will accept new data from the client application and expose the server data to it. Another task implied by this one is to store & manage the data and to provide figure candidates and matches in the page markups. It might be also convenient if if was possible to run machine learning within the API, but
we will leave this task to a dedicated console application. The main reason is technological difficulties:

- Machine learning typically runs long (several minutes), and the typical timeout for an HTTP request are tens of seconds. That implies the task would have to be fire-and-forget.

- There would have to be some management of the life cycle of the learning – and preventing conflicting simultaneous run of machine learning. (The conflicting situations are given by the particular algorithm: GATE, for example, could not run simultaneously on the same figure and domain.

These difficulties are not insurmountable but it would distort us from the main purpose of the thesis, which is to collect data, perform and evaluate the experiments.

Server Implementation Technology

An important question to be answered before reasoning about architecture of the server application(s) is the choice of the technology of the implementation. Data storing and management is a common task being similarly implemented in most of the technologies used for Web applications (.NET, PHP, Ruby, Java). Running algorithms is technologically more complex. At this point, it is likely we will run a separate application, which means there is required some deeper interoperability between the Web application and the operating system. We pick the combination of .NET and Windows Server as it is known to handle this task well.
Data storage

Our data consist of:

- Web pages
- Annotations for the pages
- Metadata with relations (figures, page topics at least)

An implicit solution for data storage and organization nowadays is a relational database. The question thus stands if there is any reason why this solution could not suffice (and what are the alternatives). There might be two problems:

- The Web pages will be stored and a Web page markup might exceed the limits of the database.
- GATE requires a file on its input.

The specification of Microsoft SQL Server 2005 [13] (which can be considered an infimum for the performance and capacity of newer versions) show the size limits of the database are terabytes, which is far beyond our scope; with some tens or hundreds of pages being indexed, assuming 1MB as an upper estimate of page size, we will utilize hundreds of megabytes at most.

To comply API of GATE, we may create temporary files containing dump of the particular value from the database for a particular operation. The database can thus still serve as the primary data source.

GATE Interoperability

GATE programmatic API is a Java binary while the server will be a .NET Web application, which is essentially a .NET binary. This implies the Web application cannot access GATE as itself. GATE in this format is not even executable, therefore we must implement a wrapping, executable Java application first. A Java application can be run from command line (.NET has libraries for command line interoperability).

In this paragraph, we define what functions should the Java application have:

- Finding figure annotations for a particular Web page.

To determine the format of data interchange between the Java application and the Web application, we first need to try one of these two approaches:

- Outputting the result into command line.
- Outputting the result into a file.

The command line approach is simpler, because it does not add dependency on file system to this process. We need to verify there will be no problems with aspect like UTF-8 characters first, though.
Architecture

Having summarized the technology for implementation & data storage as well as summarized the user requirements and tasks performed on the server, we may now design the architecture from component viewpoint.

![Outline of the server architecture - component viewpoint.](image)

Figure 1.2: Outline of the server architecture - component viewpoint.

The description of responsibilities of the components:

**Data**  Defines entities and data management. It also contains ORM to provide abstraction from the database system being used. The entity definitions are required throughout the whole server and all its components, we omit the respective relations from the diagram to preserve clarity.

**Interoperability**  In this component, there should be all code related to communication with external software.

**Figure Matchers**  This component contains our implementations of figure matchers (the trivial matcher and the heuristic matcher).
2. GATE

This chapter introduces the GATE tool, which serves two purposes:

1. The whole static text analysis and machine learning is implemented using GATE.

2. The supervisor of the thesis, Professor Vojtas, wished some text that could be used in his course of Web semantisation into this thesis. This purpose probably makes the text in this chapter more lengthy and written in a different language then the rest of the thesis.

2.1 Introduction

GATE stands for “General Architecture for Text Engineering”. The word “architecture” is a very important feature of GATE - it is not an out-of-the-box (nor a black box) solution but a modular core with API for modules (called “plugins” in GATE terminology). There are many plugins GATE is distributed with natively, which means it may also serve as an out-of-the-box solution for many tasks.

GATE has both graphical and programmatic interface. According to our experience, the graphical interface is very handy when testing the GATE plugins and designing the experiments. The programmatic interface of GATE provides access to all features of GATE. Obviously, it serves for running GATE programmatically, which is more effective when having larger data set, and maybe also provides better understanding of where is the border between the core and the plugins.

GATE has a comprehensive documentation at https://gate.ac.uk/sale/tao/split.html. The documentation covers the basics of GATE and its plugins. Its comprehensiveness means it does not provide a brief introduction but rather a material for deep understanding. We do not advocate superficiality, the space given in a lecture, however, is limited and thus must provide an accurate idea of what the tool contains and how does the work with it look like, which is exactly the ambition this text; it would make no sense to duplicate the official documentation.

2.2 GATE Trivia

Put simply, the purpose of GATE is to either transform or provide information about a document or a set of document.

2.2.1 The Terminology

There are seven elementary terms that need to be familiar with before working with GATE or GATE documentation:

- A document. GATE is working with files, so a document is always a file. There are two shapes of a document in GATE: the raw, source document and document saved by GATE. The common reason why to save documents
by GATE is it persists modifications GATE modules did to the document.
GATE saves documents in XML format.

- A corpus. A corpus is a named set of documents.
- A processing resource. Processing resources process documents or corpora.
  They are located inside plugins; a plugin may contain multiple processing
  resources. The description of processing resources located in a plugin is
  located inside the file `creole.xml` in the plugin root directory.
- An instance of a processing resource. A processing resource can be ini-
  tialized with a configuration that defines its subsequent behaviour. For
  example, Machine Learning is a processing resource, and its application
  on a particular problem is defined in a configuration. The configuration is
  assigned to a new instance.
- An annotation. Annotations are the output of processing resources; the ba-
  sic way a processing resource manipulates a document is by assigning or
  modifying annotations.
- A feature. Features are parameters or properties of annotations; key-value
  pairs containing additional information about the annotation.
- A pipeline is a sequence of processing resources configured to operate on
  a given document or corpus (“corpus pipeline” is dedicated for operating
  on corpora).
- A controller is a programmatical component with the same function as
  a pipeline.

2.2.2 Use Cases

In this introduction, we will cover the following use cases:

- Detecting and highlighting money in a text.
- Creating a custom regular expression and highlighting its matches in a text.
- Running machine learning.

We will cover these scenarios from both GUI and programmatical interface of
GATE.

2.2.3 Installation

The instructions for installing GATE can be found at [https://gate.ac.uk/
 sale/tao/splitch2.html](https://gate.ac.uk/sale/tao/splitch2.html). Long story short: GATE is not a portable binary, it
needs to be installed. It requires Java Runtime Environment to be installed in
the system, as it is a Java application.
2.2.4 Development Tools

For tuning text processes, the GATE GUI is a sufficient tool. For using GATE programmatically, Java development tools are required (e.g. Netbeans, Eclipse).

2.3 GATE GUI

In this section, we introduce GATE GUI and how to perform the use cases defined above.

2.3.1 Overview

The following picture shows GATE with the parts we will need in this tutorial:

![GATE GUI Overview](image)

**Figure 2.1:** GATE GUI Overview.

1. In the left panel, we can view and edit the resources we need to use: processing resources, document, corpora and application.
2. In the right panel, there are the details of the items clicked on from the left panel.

3. The button for showing CREOLE plugins. GATE loads only those plugins (and thus, transitively, processing resources) it is explicitly told to in this configuration. The default configuration may contain ANNIE plugin, but still it is better to check this settings before performing further actions.

This listing of features covers the needs of this tutorial only, it is not comprehensive.

2.3.2 Detecting and Highlighting Money in a Text

Theory

We need ANNIE English Tokeniser to be run over our document (we can compose an arbitrary text file containing, e.g. “My old cell phone costs $499.99.”). The English Tokeniser covers the document with the Token and SpaceToken annotations; SpaceToken annotations cover whitespace, Token cover text elements - words, numbers, punctuation and others.

After having run ANNIE English Tokeniser, we want to run another processing resource that will compose provide more complex annotations over the Token annotations. We want to obtain Money annotations that are generated by ANNIE NE Transducer.

Execution

This basic task can be done using ANNIE Processing Resources. First thing we need to check is whether the ANNIE CREOLE plugin has been loaded. If no, toggle the respective setting:

![CREOLE Plugin Manager](image)

Figure 2.2: GATE GUI - CREOLE Plugin Manager
We now want to create a pipeline, and add ANNIE English Tokeniser and ANNIE NE Transducer to it. We create pipeline by right-clicking “Applications” in the left panel:

![Figure 2.3: GATE GUI - Creating an application.](image1)

In the window, there are no processing resources to be added to the pipeline. Therefore, we need to create instances of the processing resources we declared in the Theory subsubsection to be needed.

The instances can be created in the context menu of Processing Resources in the left panel.

![Figure 2.4: GATE GUI - Adding instances of processing resources.](image2)

After we create instances of ANNIE English Tokeniser and ANNIE NE Transducer, we can add it to our new pipeline.

Finally, we:

1. Load our text document from the context menu of Language Resources and give it a custom name.

2. Open our pipeline and assign the document to the both instances of processing resources.

3. Run the pipeline.
4. Open the document.

5. Toggle Annotation Sets in the document and check Money annotation.

6. Toggle Annotation List in the document.

The result should look like this:

![Figure 2.5: GATE GUI - Matched Money annotations.](image)

### 2.3.3 Matching Custom Regex

#### Theory

We need to find a processing resource capable of doing this task. The processing resource we used is the Java Regexp Annotator from the plugin Java Regexp Annotator available at the URL [https://github.com/johann-petrak/gateplugin-StringAnnotation/wiki/Java-Regexp-Annotator](https://github.com/johann-petrak/gateplugin-StringAnnotation/wiki/Java-Regexp-Annotator) (reminder, we download a plugin, not a processing resource). The installation of the plugin is very simple: we just create a dedicated folder in the plugins folder of GATE and copy the plugin in that folder. When launching GATE, we check the plugin in CREOLE Plugin Manager to load it in GATE.

Apart from the plugin, we need the regexes. We will prepare a regex for matching processor frequency, which is `\d+(\d+)?\s*[kMG][H]z`.

#### Execution

First, we add the Java Regexp Annotator Annotator plugin to GATE as described in Theory. We also prepare a text file with contents “The CPU of my laptop has the frequency 2100MHz, 3.30GHz with Turbo Boost!” and load it to Language Resources. Finally, we prepare a text file to serve as initialization parameter of String Annotator instance. This file will have these contents:

Listing 2.1: Initialization file for matching CPU performance with Java Regexp Annotator

```
1 | \(\d+|\d+.\d+)?)\s*[kMG][Hh]z
2 | 1 => CPUPerformance which="mine",string=$0
```
For the reference of string annotator input files format, see the documentation of String Annotator; briefly said, the second line defines name and features of the annotation to be created upon the match of the regex number 1.

To put these pieces together:

1. Create an instance of String Annotator with the regex file as initialization parameter in the Processing Resources.
2. Create a new pipeline application and add the instance of String Annotator into it.
3. Set the text file as input for the instance.
4. Run the application.

The result should look like this:

![GATE GUI - Matched custom regular expression.](image)

Figure 2.6: GATE GUI - Matched custom regular expression.

### 2.3.4 Machine Learning


### 2.4 GATE Embed

In this section, we will follow the scenarios discussed in GATE GUI introduction.

#### 2.4.1 Overview

Before starting implementing any pieces of code operating GATE, we need to set up development environment. GATE is distributed as JAVA binaries, so we need a development environment operating Java. Also, it is necessary to have our code
referencing all the binaries in the \texttt{lib} folder inside GATE installation folder, as well as \texttt{gate.jar} from the \texttt{bin} folder. Having all these things set up, we can start coding.

There might be one more problem: GATE tries to detect its plugins folder automatically, but it may fail.

2.4.2 Detecting and Highlighting Money in a Text

We used this task to introduce the basic concepts of using GATE GUI, and we will use it to introduce GATE Embed, too. We will show only the snippets of code doing respective tasks, a simple demonstration may put them all in the \texttt{Main} class, which would definitely be a bad architectonical decision if the application was to be further developed.

Initialization

The whole program starts with two “magical” lines:

Listing 2.2: Initialization of GATE with GATE Embed.

\begin{verbatim}
1 BasicConfigurator.configure(new ConsoleAppender());
2 Gate.init();
\end{verbatim}

The meaning of the first row is setting up the logging output of the \texttt{log4j} library used by GATE. The parameter of the \texttt{configure} class is an implementation of the \texttt{Appender} interface (see \texttt{log4j} documentation \cite{log4j} for complete reference of the interface as well as overview of implementing classes). There is \texttt{NullAppender} for muting the logger, \texttt{ConsoleAppender} for logging to console and \texttt{FileAppender} for logging to a file.

The second row can remain a black box for us; at this point we just need to know GATE must initialize itself before being used further on.

Loading Processing Resources

The actions taken with GATE Embed usually copy similar actions taken in GATE GUI. First, we needed to load plugins into GATE GUI. We are going to do the same here:

Listing 2.3: Loading processing resources

\begin{verbatim}
1 CreoleRegister creoleRegister = Gate.getCreoleRegister();
2 File gatePluginsHome = Gate.getPluginsHome();
3 creoleRegister.registerDirectories(
4    new File(gatePluginsHome, "ANNIE").toURI().toURL());
\end{verbatim}

\texttt{CreoleRegister} reflects what we do in CREOLE plugin manager in GATE GUI. GATE plugins home represents the physical path to the plugin directory in the GATE installation, and the third row is only a composition of physical path to the directory of the ANNIE plugin (i.e. “ANNIE” references the plugin folder not plugin name).
Creating the Pipeline

At this point, we should start creating instances of GATE objects - a pipeline, a document and an instance of a processing resource. All these instances are created using `Factory` class. Despite it is technically possible to use `new` keyword and the constructor of the respective class, GATE documentation says it must be done with the `Factory`.

The following piece of code creates the controller with ANNIE English Tokeniser and ANNIE NE Transducer added:

Listing 2.4: Setting up GATE controller.

```java
SerialAnalyserController annieController = (SerialAnalyserController) Factory.createResource(
   "gate.creole.SerialAnalyserController",
   Factory.newFeatureMap(),
   Factory.newFeatureMap(),
   "MoneyFinder");

ProcessingResource tokeniser =
   (ProcessingResource) Factory.createResource(
   "gate.creole.tokeniser.DefaultTokeniser",
   Factory.newFeatureMap());
annieController.add(tokeniser);

ProcessingResource transducer =
   (ProcessingResource) Factory.createResource(
   "gate.creole.ANNIETransducer",
   Factory.newFeatureMap());
annieController.add(transducer);
```

One thing to notice is the factory requires fully qualified name of the class. This name can be found in the `creole.xml` configuration file of the respective plugin (ANNIE in this case). The feature maps are key-value maps used to pass initialization parameters to the controllers. In this case, we do not need any initialization parameters, so we provide empty maps.

Loading and Annotating the Document

At this point, the only thing that is missing is to load the document, assign it to the controller and run the controller:

Listing 2.5: Loading and annotating the document.

```java
Document doc = Factory.newDocument(url, "UTF-8");
Corpus corpus = Factory.newCorpus("Arbitrary Corpus");
corpus.add(doc);
SerialAnalyserController annieController = (...);
anieController.setCorpus(corpus);
anieController.execute();
```
The creation of the controller is described in the previous listing; otherwise, the code is declarative and straightforward.

Obtaining the Results

The result of the code are the annotations assigned to the document. Obtaining the annotations is very easy:

Listing 2.6: Obtaining annotations assigned to a document.

```java
AnnotationSet annotations = doc.getAnnotations();
```

2.4.3 Matching Custom Regex

The concepts needed to accomplish this task are very similar to those in the previous case. There is one more to be introduced: the ANNIE processing resources do not require initialization parameters, while Java Regexp Annotator does. This is how we provide Java Regexp Annotator the configuration file with the regex(es):

Listing 2.7: Initializing Java Regexp Annotator

```java
FeatureMap cpuMatcherParams = Factory.newFeatureMap();
cpuAnnotatorFeatureMap.put(
    "patternFileURL",
    new File(
        Gate.getPluginsHome(),
        "String_Annotator/regexes/notebook/cpu.txt"
    ).toURI().toURL() );
ProcessingResource cpuAnnotator = (ProcessingResource) Factory.createResource(
    "com.jpetrak.gate.stringannotation.regexp.JavaRegexpAnnotator",
    cpuMatcherParams);
```

Two remarks to the code: firstly, in our case, Java Regexp Annotator is stored in the String_Annotator folder. Secondly, the qualified name of the annotator class was split by the plus sign only to fit this narrow document, there is no other purpose in it.

2.4.4 Machine Learning

Machine learning has its training mode and application mode (and six more, these are only the basic ones). Its instance uses a configuration file; the folder it is stored in is considered as a root folder for the runs. In that folder, during the training mode, the learning procedure creates all intermediate files that are later used in the application mode.

Listing 2.8: Setting Up Machine Learning PR

```java
FeatureMap initParams = Factory.newFeatureMap();
initParams.put("configFileURL", learningConfigFilePath);
FeatureMap runtimeParams = Factory.newFeatureMap();
```
runtimeParams.put("learningMode", "TRAINING");

ProcessingResource machineLearning =
        initParams,
        runtimeParams);

With machine learning set up, the only thing to do next is to create the corpus with the set of annotated documents and run the machine learning. This skips the question of preparation of the training data, this question is discussed in the GATE GUI machine learning example above.
3. Implementation

3.1 Browser Add-On

3.1.1 Technology

The browser add-on is implemented for Mozilla Firefox. The contemporary technology for implementing a Mozilla Firefox is Node.js [14]. Mozilla provides a command-line tool for managing the add-on project, called JPM. The add-on is bootstrapped by the command `jpm init` that creates a runnable empty plugin. To see the plugin deployed in the browser, the command `jpm run` is to be run. Among its parameters and flags, there is `--debug` which enables debugging add-on code with Firefox debugging tools, and the parameter `-b` which enables us to specify the location Mozilla Firefox is installed.

There is one problem with deploying the plugin for broader audience: the end-user version of Mozilla Firefox does not allow installation of plugins not having been certified by Mozilla. This means that the compiled add-on cannot be added to the normal version (our add-on is a custom tool for this thesis, so there is no reason to put it into the certification process). However, there is another version of Mozilla Firefox, Firefox Developer Edition, which enables running non-certified add-ons. It is necessary, however, to set the value of `xpinstall.signatures.required` to `false` in `about:config`. 
3.1.2 Architecture of the Add-On

The browser plugin lives within context provided by the browser – Mozilla Firefox in our case. This context can be accessed through high- and low-level API [4].

![Architecture of the browser add-on](image)

Figure 3.1: Architecture of the browser add-on.

In this schema, the pink components are external data sources, the green components are modules provided by Firefox Add-On API and the blue components are modules of our plugin.

There are two major components in the environment of the plugin: the Web page from the current tab and the Web API the plugin is querying. Both of
these components are accessed via Add-On API: Request API is used for making request to the API, Tabs API is used for passing custom JavaScript to the page (see official Mozilla documentation [4]).

**Controller**  The entry point of the plugin. Sets up data structures and add-on UI and links callbacks to the respective listener (triggering marking occurrences in the active tab, reporting occurrences to the server and so on).

**API Mirror**  A module containing functions for querying each of the Web API endpoints.

**UI Definition**  A module for setting up add-on UI in the browser (just the control panel, not the modifications of the active page).

**Page Annotation Script**  JavaScript passed into the active page when annotation marking is turned on.

### 3.1.3 User Interface

Add-Ons may present themselves in various ways in Mozilla Firefox: a menu item, a toolbar button or a new panel by one of the edges of browser window. In the add-on, we need to display a table of figures with buttons for launching marking occurrences, therefore the UI of the add-on will be organized as a panel.

![Figure 3.2: Add-on user interface.](image-url)
3.1.4 Remarkable Implementation Details

Data Interchange The transfer of data between components is complicated. Both the sidebar script and the script being attached to a tab are autonomous and communicate with the controller indirectly.

There is a concept of worker provided by the add-on API, which serves as the medium of data interchange. The worker enables to emit event in one component and listen to the event in the other one; this communication is double-sided.

As an example, we will show how an occurrence of a figure is passed from the page script into the controller.

In the page script there is the following code:

Listing 3.1: Sending an occurrence
1 self.port.emit('occurrence', {
2 index: index,
3 offsetInDocument: offsetInDocument,
4 offsetInElement: offsetInElement,
5 xpath: xpath,
6 match: text,
7 html: html,
8 snippet: snippet
9 });

Listing 3.2: Listening to marking of an occurrence
1 activeTabWorker.port.on('occurrence', info =>
2 apiMirror.reportOccurrence(
3 info,
4 tabs.activeTab.url,
5 result => console.log(
6 'Obtained result. Status ${result.status}.'
7 );
8 ));

A very important to notice is the string ‘occurrence’. It is the identifier of the event, the object representing the occurrence is the payload of the event.

ES5 vs. ES6 Recently, 6th version of ECMAScript was released [15]. This causes an ambiguity: Node.js can be implemented using ES6, while the browsers support the older version only. This ambiguity is present in the code of the add-on: the scripts running in the pages are written in ES5, and so is the code of the sidebar. The controller and API mirror are written in ES6.

Injecting script into the loaded page The fundamental feature of the add-on. When the user wants to annotate a page, he visits the page by usual means in the browser. Once he wants to mark an occurrence of a figure, it is necessary to modify the page. To make it responsible to his clicks, to even highlight some portions of the text in the page.

This is done using Firefox tabs API. The following snippet manages running our script in the page:
Listing 3.3: Attaching the script for marking occurrences into a page

```javascript
let activeTabWorker = tabs.activeTab.attach({
  contentScriptFile: self.data.url(
    './tabScripts/occurrenceMarker.js'
  )
});
activeTabWorker.port.on('occurrence', (...));
activeTabWorker.port.on('run-done', (...));
activeTabWorker.port.emit('run', data);
```

The first line with `tabs.activeTab.attach` runs the script `occurrenceMarker` in the current active tab. The following lines with `activeTabWorker.port.on` sets up listeners for respective events and the last line sends initialization data to the tab script.

Highlighting figure occurrences

When the user triggers marking of a figure, its occurrences matched in the page markup are highlighted. This is the responsibility of the script attached to the tab.

The occurrences are provided as an array of strings, which means that it is necessary to find the strings in the page. It is achieved by traversing the DOM tree and running the following function for every text node.

Listing 3.4: Function for highlighting figure candidates in a page.

```javascript
insertMarkerTags: function (parentNode, currentNode, xpath, offset, figureRegex, highlightedWord) {
  var innerText = (...).getTextNodeInnerText(currentChildNode);
  var aggregatedLengthPreviousTextNodes = 0;

  var iterator = 0;
  while (parentNode.childNodes[iterator] != currentChildNode) {
    if (parentNode.childNodes[iterator].nodeType === Node.TEXT_NODE) {
      aggregatedLengthPreviousTextNodes += (...);
    }
    ++iterator;
  }

  var exploded = innerText.split(figureRegex);
  if (exploded.length > 1) {
    var snippet = (...).composeSnippet(parentNode);
    var replaced = document.createTextNode(exploded[exploded.length - 1]);
    parentNode.replaceChild(replaced, currentNode);
  }
}
```
var offsetInTextNode = 0;

for (var i = 0; i <= exploded.length - 2; ++i) {
    offsetInTextNode += exploded[i].length;
}

for (var i = exploded.length - 2; i >= 0;) {
    offsetInTextNode -= exploded[i].length;
    newEl = occurrenceMarker.domManipulation
               .createWrapper(...);

    parentNode.insertBefore(newEl, replaced);

    --i;
    offsetInTextNode -= exploded[i].length;
    replaced = newEl;
    var newEl = document
               .createTextNode(exploded[i]);
    parentNode.insertBefore(newEl, replaced);
    replaced = newEl;
    --i;
} },

In the code of the function, there are the following important points:

• Line 4: var exploded = innerText.split(figureRegex);

  Splits the text node by the occurrences. figureRegex is a regex function
  matching all occurrences appearing in the whole document. If there is
  no occurrence in the node, the result of the expression is an array having
  one item (which is the meaning of the subsequent if - if there were no
  matches in the node, there is no reason to modify it).

  In the resulting array, there are the chunks of the original node where
  the occurrences are at odd indices.

• At lines 6 to 17, we must compute the offset of our text node within
  the parent; this is necessary, because the occurrence is identified by XPath
  of the parent and the offset in the text of the parent, so we must add
  the lengths of preceding siblings to the position inside current text node.

• Line 22: parentNode.replaceChild(replaced, currentChildNode);

  This expression removes the node in its original form from the DOM and
  puts the first element from the exploded array instead.

• Line 39 parentNode.insertBefore(...)
Every odd element of the array is a match of an occurrence. It must be wrapped with a tag which ensures the highlighting and responsiveness to mouse clicks.

**Selecting figure occurrences inside text**  Some kinds of occurrences (vacation locations, for example) cannot be matched algorithmically. In such cases, the user is able to select a portion of the text and mark it as occurrence. This feature requires a listener on text selection on the whole document. It is programmed using the `onmouseup` event:

```
Listing 3.5: Text selection listener
1 document.body.onmouseup = function(e) {
2     window.setTimeout(function() {
3         if (!(...).preventMouseUp) {
4             var positionX, positionY;
5             var selectedPortion = (...).getSelectionHtml();
6             if (selectedPortion && selectedPortion.length) {
7                 (...)
8                 // The code managing the selection.
9                 (...)
10             }
11         } else {
12             delete (...).preventMouseUp;
13         }
14     }, 50);
```

As we can see, the callback is wrapped inside a timeout. The reason is the `mousedown` event is triggered after `mouseup`, and we want to run this function after clicks (e.g., clicks on links or the matched occurrences) are resolved. It is not a clean solution, 50 milliseconds is an arbitrarily chosen number, but it is sufficient for our task.

After the text is selected, a confirm window is shown prompting the user to confirm the selection as selected occurrence. After that, the selected text is treated the same as a matched occurrence clicked via the box.

### 3.1.5 Installation

The plugin is installed from the XPI file by drag-and-drop into Firefox Developer Edition. In the configuration of the browser (`about:config`), `xpinstall.signatures.required` must be set to `true`.

### 3.1.6 Usage

The plugin has a single use-case: marking figure occurrences in the pages. If the user wants to mark occurrences for the particular page, he must click the "I Want to Mark Figures" button. This blocks the add-on UI for a while and once the server data are obtained, it is enabled again with matched occurrence data filled into the table of figures.
To mark occurrences of a particular figure, the user clicks the button "Mark Occurrence(s)" next to the respective figure and either clicks a box wrapping the right occurrence or selects a portion of text to provide custom occurrence.

3.2 The Server

3.2.1 Technology

The common technology of server components is .NET. Below are listed the major .NET frameworks and patterns present in the implementation:

**ASP.NET WebAPI** ASP.NET WebAPI is a type of .NET Web application - the other being ASP.NET MVC used for applications with user interface. Web API applications are typically ran on the IIS Web server. A new technology, Owin, provides an abstraction between the application and the Web server [16].

**Entity Framework** Entity Framework is an ORM library for .NET. Its basic usage is expressing database queries with chains of synonymous LINQ calls. In the implementation, the database is mirrored a user-defined class that extends the class DbContext from the Entity Framework library. Tables are mirrored by adding a property of type DbSet<T> where the type T reflects the table structure. A snippet of the implementation:

```csharp
public class DatabaseContext : DbContext
{
    public DatabaseContext()
    : base("name=" + ConnectionStringName)
    {
    }

    public virtual DbSet<Page> Pages { get; set; }

    (...)
}
```

Another part of the framework is support for code-first database definition [17]. This support is provided with a set of command line tools that, in a nutshell, take the database context class and compose the whole database from it. A major advantage of this approach is that the data model is defined on a single place only and is not prone to issues with various SQL dialects - Entity Framework takes this responsibility.

3.2.2 Architecture of the Server Part

In the analysis chapter, we outlined basic breakdown of the tasks of the server into components. This is the final function decomposition of the Server:
Runnable Components  The server can be run using three components: the Web API, the learning service and the command-line wrapper:

- The Web API is used by the add-on to collect training data and to provide matches from the naïve and the intelligent matchers.

- The Command-Line Tool is a command-line executable used to run the learning routine with the possibility to debug it if run from Visual Studio.

Data Access Components  There are two data sources, each having classes providing programmatic API and abstraction as well: the database and the files. The database serves for storing training data and metadata describing them. The file system serves for communication with GATE: Firstly, the data interchange, because we access GATE with a console application and the console has difficulties with preserving UTF-8 characters. Secondly, the input of any processing resource in GATE is a file. We need these input files only for one particular run, therefore our file storage is only temporary.

We created two implementations of the storage:

1. Temporary storage, which utilizes Temp folder accessed from System.IO.Path class

Figure 3.3: Architecture of the server part.
2. A parametrized directory storage, which creates the files in a directory provided through the constructor.

The latter storage is used when debugging, to access the intermediate files easily.

**Naïve Figure Matchers**  This component contains three trivial matchers - database matcher, XPath matcher and word matcher - with a wrapping class that combines their results together. The implementation follows the concept described in the Analysis chapter 1.4.3.

**Heuristic Matchers**  This component contains implementation of Heuristic Matcher.

**GATE Interoperability**  GATE (or our Java wrapper application for GATE, specifically) provides three major tasks: matching possible figure occurrences, learning rules to match a figure and matching figures.

The first task requires simple data interchange between the Server and the Java application: the Server provides a source document, and the application provides a string or a set of strings, with some information about their location.

The learning task requires paths to the training documents, along with the path to the machine learning configuration file (each figure has its own configuration file). The documents undergo some transformations, which is further discussed in the Experiments chapter (see 5.1).

The matching task requires the document, transformed the same way as the documents of the training set, and configuration file respective to the figure to be matched.

**3.2.3 Remarkable Implementation Details**

paragraphInterfaces Following the modern of testability, many of the important modules are hidden behind interfaces. The most important follow:

- **IFigureMatcher**  An abstraction of a module for matching figures. Its APIs exposes two method. One for obtaining matches for a particular page, the other one notifying the module that a new occurrence has been marked, enabling launch of learning procedures.

**Running GATE**  GATE is wrapped with our wrapper application. Running this application means to run the command line from the .NET code. This task is handled by .NET library **System.Diagnostics.Process**.

There came a question how to pass the result from the wrapper application and what format should it have. The result is a dictionary: for each GATE annotation, there is an array of matching strings. JSON format has exactly this nature, so it was the most natural choice.

There are two ways of how to pass the data:

- Through the standard output.
• Through a temporary file.

Our experiments showed Windows command line has problems with handling UTF-8 characters, which was not necessary to solve in case of the temporary file. Moreover, .NET has a simple way of creating temporary files (Path.GetTempFileName), which made us use this approach in the final implementation.

**Preparing Snippet for Machine Learning**  A snippet is a piece of text surrounding the match from training data. A snippet should be big enough to contain relevant keywords from the neighbourhood. It does not need to be “very long” (further on, we will discuss what “very long” is), because only the adjacent words provide context to the word.

The relevant neighbourhood is determined by the configuration of GATE machine learning: in the configuration, we define how many tokens around the match are we interested in. The tricky part of this decision is there can be the whole portion of the Web page might be introduced by a heading that could be syntactically far away. Apparently, one cannot declare a fixed sufficient number, so initially we set with a minimum of 200 characters of length for a snippet.

The experience with GATE machine learning made us change the scope of the snippet to the whole document. The reason is GATE machine learning needs counterexamples for its learning and these counterexamples might be missing in the close neighbourhood of the occurrence. This turned the creation of a snippet into creation of plain text document out of a HTML document, effectively.

To obtain the neighbourhood, we have to traverse through the DOM tree from the node containing the match up.

This fundamental idea has some more problems to solve:

1. The plain HTML markup contains many whitespace characters. It is very common that these characters do not propagate into the page contents and serve for HTML source code formatting only.

2. The HTML markup also contains character escapes - `&nbsp;`, for example.

3. The match is identified by XPath of its containing node and offset inside the text of the node. There is a mismatch between what is inner text in Mozilla Firefox and what is inner text in the HtmlAgilityPack library [22]: While Mozilla Firefox considers inner text of a node merely the concatenation of its text node children, the HtmlAgilityPack library takes as the inner text the text nodes of the whole subtree of the node.

To demonstrate the difference, consider the following HTML snippet: `<span id='parent'><span id='child'>a</span>bcd</span>`.

According to Mozilla Firefox, inner text of parent span is ‘bcd’. According to HtmlAgilityPack, it is ‘abcd’. Thus the index of the occurrence - ‘c’ for example - does not match.

4. Any modification of the original HTML markup will invalidate the index of the match, which means any operation must be accompanied by recomputation of the index.
The context of sorting out these problems is given by how are we going to
compose the snippet, which is the following loop:

Listing 3.7: Loop creating a snippet.

```csharp
public Snippet CreateSnippet(
    string html,
    string xpath,
    int position)
{
    // Load the HTML from string
    // into object representation.
    var doc = new HtmlDocument();
    doc.LoadHtml(html);
    var node = doc.DocumentNode.SelectSingleNode(xpath);
    ...

    // Traverse up the DOM tree.
    var offsetInsideSnippet = position;
    for (;
        /* node.InnerText.Length < 200 &&*/
        node = node.ParentNode)
    {
        offsetInsideSnippet +=
            PrecedingSiblingsLength(node);
    }

    var text = node.InnerText;
    ...

    // Compose the result.
    return new Snippet
    {
        Index = offsetInsideSnippet,
        Text = text
    };
}
```

This piece of code shows the raw algorithm without any modifications to
the markups. The triple dots show places where the code will be eventually
extended with the modifications. The condition `node.InnerText.Length < 200` is deliberately comment and it reflects the decision changing the scope of snippet
to the whole document.

Both escaped HTML characters and whitespaces can be solved together by
the following modification:

Listing 3.8: Loop creating a snippet with decoding escaped HTML and replacing
whitespaces

```csharp
... for (;
```
Remarks to this piece of code:

- Line 3 - the `LengthWithoutWhitespace` function transforms the provided string by unescaping the HTML-escaped characters, removes whitespace and returns the length of the resulting string.

- Line 8 - the function `PrecedingSiblingsLength` modifies the original content of the siblings in the same way.

- Lines 11 and 19 - unescape and remove whitespaces from the contents of resulting node. The reason they are separate is it is necessary to recompute the change of the offset, which is done from the string that still contains the original whitespaces.

- Line 14 - computes new offset after the whitespaces are removed. It is done by taking the portion of the snippet from start until the match and removing the whitespaces. This simulates how the part of snippet preceding the match changes, with the assumption it will be immediately followed by the match after the modifications.

This code already covers all the problems but the third one which is the index of the match inside the leaf node does not match in general case. To cope with this, we will change the value on the `node` variable to the textnode that contains the match; the `position` variable will also be changed:
Listing 3.9: Finding text node containing the match

```csharp
public static Tuple<HtmlNode, int> GetTextNodeChildAtPosition(
    HtmlNode node,
    int position)
{
    int currentPosition = 0;
    int positionInsideNode = -1;
    return new Tuple<HtmlNode, int>(
        node
            .ChildNodes
            .First(child =>
                {  
                    if (child.NodeType != HtmlNodeType.Text)
                        {
                            return false;
                        }
                    }  
                positionInsideNode = position-currentPosition;
                currentPosition += child.InnerText.Length;
                return currentPosition > position;
            }),
            positionInsideNode);
}
```

The code uses LINQ `First` method which returns first element in given enumeration matching given criteria. In our case, we are looking for a text node - which implies the `if` statement - and the element is defined by the position in the text of the node (which is covered on the lines 19 and 20). The expression on the line 18 computes the position with respect to the start of the current text node.

Remark: This algorithm was made more robust by handling edge cases of index mismatch. The main parts of the algorithm remained in this state.

Figure Matchers

The figure matchers try to find figure occurrences in the Web pages. The whole set of matchers implemented on the server is intended to work as one matcher; the module Result Aggregator puts the results together. The result is presented as an enumeration of tuples of occurrence and rating. The occurrence with the highest rating is then considered as the output of the entire module.

Database Occurrence Matcher This matchers looks if the page was already annotated by the user, if there is any occurrence already chosen in the database. If there is, it is returned, an empty result is returned otherwise.

Close Words Matcher This matcher uses GATE annotations for the figure. If one of the annotations is preceded by a keyword arbitrarily assigned to the figure, it marks it as a match. The occurrences get an equal rating summing to 1 in total.
**XPath Matcher**  This matcher searches for keywords in the class names of the DOM elements. These keywords are also defined arbitrarily for each of the figures. When the keyword is found in class of an element, its text content and text contents of all its descendants is added to possible matches. The matches get an equal rating summing to 1 in total.

**Result Aggregator**  As outlined above, the result aggregator wraps the rest of the trivial figure matchers, providing them a common interface to the other modules and also combining their results to provide a single one. Technically, if database matcher returns a result, it is returned, otherwise it merges the results of the Close Words Matcher and XPath matchers together, joining the results by the matches and summing the rating they received in each of the matchers. The match with the highest aggregated rating is returned.

**Heuristic Matcher**

Heuristic matcher contains implementation of heuristic machine learning algorithm as defined in the subsubsection 1.4.3.
3.3 GATE Wrapper

GATE Wrapper is a Java application responsible for running GATE. Its outer API reflects the tasks required by the server, which is:

- Providing annotations for possible figure occurrences.
- Running the machine learning process.
- Matching figures in new pages.

The implementation of GATE Wrapper is thus closely related to GATE documentation [18], GATE Embedded [19] documentation in particular.

3.3.1 Gate API

In this section, we list the important features of the GATE API.

**Factory**  The modules of the GATE - applications, processing resources, documents, corpi - have their object representations in the classes of the GATE API. The way to construct instances of the classes is strictly defined in the documentation [19]: “A programmer using GATE Embedded should never call the constructor of a resource: always use the Factory!”

For each of the types of the modules there is a respective method in the Factory class: `Factory.newDocument(url, "UTF-8")`, for example. For processing resources, there is the method `Factory.createResource(packageName, map)`. The package name is the Java package of the resource. The resources are wrapped in their plugin folders, so it is necessary to register the plugin folder first before referencing it in code. Technically, the registration of packages within the plugin is done with `creole.xml` configuration file, which tells GATE how to load the plugin.

**Inputs**  The inputs of any resources are either documents or corpi - which are essentially named collections of documents. The way of creating a document is described above in the Factory section; the documents are constructed either from a file or a Web URL. The server downloads the pages and provides them from local temporary file, however, so in our case the documents are always stored locally in a file.

**Outputs**  The outputs of resources are either annotations for a document (or document) or some auxiliary files (in case of machine learning, for example). The annotations are assigned inside the object representing the document with the possibility to iterate over them.

3.3.2 Interoperability

GATE Wrapper is a command-line tool. It turned outputting the result in the console can break the character encoding, therefore both inputs and outputs of GATE Wrapper are stored in temporary files.
3.3.3 Figure occurrences

To get possible figure occurrences, we use

- ANNIE English Tokeniser
- ANNIE NE Transducer
- String Annotator with configurations:
  - Memory, matching memory and hard drive capacities.
  - Performance, matching CPU performance.

As mentioned above, the annotations resulting from the GATE run are iterable. We use this to collect the occurrence of each of the annotations and compose a dictionary which is returned as a JSON object.

3.3.4 Annotated Document

This task on its own is very similar to the previous one. The difference is in the purpose: while figure occurrences are used for figure highlighting in a Web page, the annotated document is just a basis for the machine learning. The annotations for the particular HTML page are amended with features the machine learning should learn to infer.

Therefore, the same set of processing resources is used, but the output is the XML file.
4. Manuals

In this chapter, we describe how to launch the particular components of the system and how to use it. We assume the goal is to get the add-on working with the Web API, having the service periodically launching learning on new occurrences and having the console application evaluating a particular figure on a particular domain by given matcher.

4.1 Server Installation

4.1.1 Prerequisites

The prerequisites of the environment are:

- Windows Server
- Running IIS Server
- Running MSSQL Server
- JRE (version 1.8.0_91 by the time of writing this thesis).
- GATE Installed (see Attachment 1), but can also be downloaded from the Web.
- A local copy of Java GATE Wrapper application (see Attachment 5).

4.1.2 Database

The database can be created by restoring the backups from Attachment 6 (structure only), Attachment 7 (structure with definitions of entity types) or Attachment 8 (structure with data in the state of finishing the thesis).

4.1.3 Web API

Deployment

The Web API Attachment 2 is deployed by copying the contents of the folder at any folder relevant for IIS.

Installation

The installation procedure is a standard deployment of a Web site with IIS. For newcomers, the official Microsoft resources contain guides for IIS, such as this one: [30].

Configuration

Database Connection In the file Connections.config, it is necessary to provide the connection string for the MSSQL database. (The Web site [https://www.connectionstrings.com/](https://www.connectionstrings.com/) might be helpful.)
GATE Configuration  In the file Web.config, see the element <gate />. Fill in the attributes as follows:

- **gateHome** Absolute path to the installation folder of the GATE.
- **jarFile** Absolute path to the GATE Wrapper application (see item 6 of prerequisites [4.1.1]).
- **machineLearningConfigFolder** Absolute path to the folder where the server should store configuration and auxiliary files of GATE machine learning. Inside the folder, there should be a folder for each of the figures. The folder name must be lowercase with whitespace replaced with hyphens (e.g. “Product Price” as “product-price”). In the root of the figure folder, there must be a configuration file for the learning called “configuration.xml”.
- **occurrenceFolder** Absolute path to the folder where the GATE annotations for the Web pages saved on the server should be saved.

Heuristics Configuration  In the file Web.config, see the element <heuristics />. Fill in the attributes as follows:

- **heuristicFolder** Absolute path to the folder where the heuristic will store its statistics.
- **attributeRatingRelevance** A decimal number between 0 and 1 expressing what portion of the heuristic rating will define the rating by HTML attributes. The reminder to 1 will be covered by the rating by surrounding words.

Trivial Matcher Configuration  In the file Web.config, see the element <trivialMatcher />. Set the value of the attribute to true or false depending on whether you want trivial matcher to match values by looking them up in the database. The value false is recommended for testing the performance of naïve algorithms.

Logging Configuration  The basic configuration means just to fill in the value attribute of the file tag to define the target file of the log. For advanced configuration, see the documentation of the log4net library [25].

4.1.4 Console Application

Deployment

The application is deployed by copying into target folder.

Installation

The application is installed by copying into target folder.
Configuration

The application is configured the same as WebAPI 4.1.3

4.2 Add-On Installation

4.2.1 Prerequisites

The add-on is developed for Mozilla Firefox, however, Mozilla follows the policy of not accepting add-ons not verified by Mozilla. Our plugin is not verified, so we must have Mozilla Firefox Developer Edition installed.

In Firefox Developer Edition, we must enable the installation of unverified add-ons: first, open the page about:config. In the configuration, set the xpinstall.signatures.required to true.

4.2.2 Installation

Having Mozilla Firefox Developer Edition installed and configured, the add-on is installed by dragging and dropping the XPI file into the browser.

The add-on is distributed as an xpi file. However, the server endpoint used by the add-on is inlined inside the XPI. The endpoint the XPI is distributed with is in the domain figurefinder.protekty.ms.mff.cuni.cz. To change the endpoint, it is necessary to change it in the source code Attachment 9: the file data/apiMirror.js, line 2. After the change, rebuild the addon using build.sh in the root of the sources.

4.3 Using the Add-On

4.3.1 UI Description

Description of the numbers in the screenshot 4.1:

1. The entity type currently selected.

2. The button for switching the entity type.

3. Table with the figures of the entity type; in the right column, there is matched value of the figures, and optionally, depending on the mode, also a button for turning the mode of marking occurrences of the property on.

4. Button for showing or hiding the buttons for marking occurrences.

5. The button for loading occurrences of all figures from all entity types for the current page.

6. The current page.
4.3.2 Usage Scenarios

**Highlighting Figure Occurrences** If we want to highlight figure occurrences (which is the way to test how the matching algorithm works), we do the following steps:

1. Navigate to the page we want to highlight the occurrences.
2. Click the “Create Training Set!” button.
3. Wait until the table is enabled and populated with buttons and figures.
4. Optionally, clicking “mark occurrences” will highlight all possible occurrences of the property pink and the matched occurrence green.

**Submitting a new training page** To submit a new training page, follow the steps from the previous scenario up to the last one and then do the following:

1. Click the occurrence you consider as the correct match (it is possible to choose multiple and, in some cases, it is even necessary as providing only one of a group of occurrences might make the rest of them counterexamples and thus confuse the machine learning).
2. Optionally, click a different button “mark occurrences” to mark occurrences for another figure.

*Remark: the add-on resets itself when loading a page or when switching a tab.*

4.4 Using the console application

The console application is able to launch learning or matching procedure for each of the matchers. It operates the data from database only, which means it should
work with the same database as the WebAPI and the database must be populated 
already. Its behaviour is given by the arguments:

Listing 4.1: Call signature of the console application.

FigureFinder.exe <task> <matcher> [ <dataset> ]
[ --figures <figureName1> <figureName2> ... ]
[ --domains <domain1> <domain2> ... ]

• <task> has one of the values learn or match.
• <matcher> has one of the values trivial, heuristic or gate.
• <dataset> is optional and is used only if match task is provided. The ac-
cepted values are all or testing. The first makes the application match
given figures in all pages from given domains (see below). The latter makes
the application match given figures in all testing (i.e. not training) pages
from given domains (see below).
• --figures precedes a list of figure names. An asterisk (*) can be provided
as a substitution for all figures in the database.
• --domains precedes a list of domains to be matched. An asterisk (*) can
be provided as a substitution for all domains in the database. The flag
--universal can be provided for the learning task only to run learning
of the universal (i.e. not domain-specific) rules.

4.4.1 Examples

Learn to match figure Product Price on www.ebay.com with Heuristic matcher:
FigureFinder.exe learn heuristic --figures "Product Price"
--domains "www.ebay.com"

Match figure Product Price on www.ebay.com with Heuristic matcher on all doc-
uments:
FigureFinder.exe match heuristic all --figures "Product Price"
--domains "www.ebay.com"

Match figure Product Price on www.ebay.com with Heuristic matcher on testing
documents only:
FigureFinder.exe match heuristic all --figures "Product Price"
--domains "www.ebay.com"

Learn universal rules for Product Price and CPU Performance with GATE matcher:
FigureFinder.exe learn gate --figures "Product Price" "CPU Performance"
--domains --universal

Match figure Product Price on all pages with GATE matcher:
FigureFinder.exe match gate all --figures "Product Price" --domains *

Match all figures all pages with heuristic matcher:
FigureFinder.exe match heuristic all --figures * --domains *
5. Experiments

5.1 Tuning Page Manipulation

The idea of the page annotation tool is: we open page after page of some topic, and mark an occurrence for each of the figures. However, the initial attempts with prototype implementation made us discover a few problems to address.

**Machine-Unfriendly Web Sites** Some pages, Amazon, for example, prevent their pages to be browsed by robots. If they detect a robot, they return 500 or 404. It is likely these pages expose themselves to search engines like Google, but apparently block the robots in general. This turned out to be a big drawback in the initial solution, where the page annotator only sent page URL to the server and the server downloaded the page from the URL and processed it.

The possible solutions were the following:

- Resign to annotating these pages.
- Send the HTML of these pages to the server from the client.

Resigning to annotating the pages would make sense if the effort put into changing the architecture would not be worth the obtained data. The first Web site we were missing was Amazon.com, a very popular e-shop, more pages were found soon thereafter. The architecture was changed as a solution.

**Providing Machine Learning Training Set** The occurrences of the figures in the database can be easily provided as an HTML snippet or HTML page. The problem is that GATE machine learning needs the training set to be annotated (see Machine Learning in the following section). That can be achieved in two ways:

1. Let GATE annotate the plain source files and then mark the resulting annotations in some way. As it is possible to mark the resulting annotations in the Java GATE Wrapper only, the .NET part would have to provide identification of the occurrences into the wrapper. That would bring more problems to solve - while in the .NET, an occurrence is identified by an XPath expression and position inside the leaf node, in GATE an annotation is merely an interval of two positions in the plain text.

2. Create an XML file representing annotated GATE file. The problem of this approach is GATE may either overwrite the annotations when annotating the document or the arbitrary annotations might mismatch the GATE format in another way.

GATE machine learning is based on tracing the neighbouring annotations which wrap the particular words. If we tried to make the XML file ourselves, it would mean we would have to simulate the ANNIE processing resources. Therefore, it seems more sensible to try the second approach.
Parsing GATE Document  GATE document is a XML file which contains:

- The original text with nodes added as XML elements.
- Annotation sets.
- Metadata.

The task we are facing is the following: we need to match the annotation, which is expressed as a XML element in the GATE document, with the occurrence as it is stored in the database. The XML element declares type of the annotation, the node where it starts and the node where it ends. The nodes are essentially indices in the text of the document, the text, however, does not match the input one-to-one: GATE trims whitespace and evaluates special HTML sequences (such as &nbsp;). In its default configuration, ANNIE NE Transducer does not include the matched value of its annotations (money, for example). This behaviour differs from ANNIE English Tokeniser, which does include the matched value in the annotation element.

We will try the following solutions (ordered in assumed order of complexity):

1. Find a configuration property of the Transducer that would turn on adding the matches into the XML.
2. Put a placeholder into the document that would help us locate the annotation.
3. Transform the HTML snippet into plain text.
4. Fetch annotation values in JSON format.
5. Program a function that would compute the correct node ID of the match.

The first solution is not supported by ANNIE NE Transducer [18].

The problem with the second solution is we add some artificial content that would either confuse machine learning or force us to work it around in machine learning configuration. In case of more complication, undoing this system would not be trivial.

Transforming HTML into plain text gave correct results: the indices of the occurrences computed in .NET were matching the indices of the respective nodes from GATE. With this solution, GATE machine learning loses the HTML annotations, which does not break our concept, as we want GATE to perform text analysis only, without any HTML awareness.

5.2 Finding Correct GATE Configuration

5.2.1 Occurrence Annotations

5.2.2 Machine Learning

For machine learning, we want to use the Batch Learning Processing Resource which is distributed with GATE. It is capable of performing various tasks, each requiring a different kind of configuration.
Summary of what we want to do: we have a set of figures and a set of occurrences of the figures. We want the machine learning to learn how to recognize the figure inside a Web page. And for this learning, we may provide a set of occurrences, each of them with either surrounding snippet or the whole page.

Machine learning has seven modes in total, we will need the following three:

- **Training** From GATE documentation [18]: “In TRAINING mode, the PR learns from the data provided and saves the models into a file called ‘learnedModels.save’ under the sub-directory ‘savedFiles’ of the working directory.”

- **Evaluation** From GATE documentation [18]: “In EVALUATION mode, the PR will do k-fold or hold-out test set evaluation on the corpus provided (the method of the evaluation is specified in the configuration file, see below)(...)”

- **Application** From GATE documentation [18]:

  If the user wants to apply the learned model to the data, s/he should select APPLICATION mode. In application mode, the PR reads the learned model from the file ‘learnedModels.save’ in the subdirectory ‘savedFiles’ and then applies the model to the data.

Typically, the plugin is run in all three modes sequentially, using the same configuration file. The output of the training phase is stored in the file ‘learnedModels.save’ which can thus be used as a portable snapshot of the machine learning for a particular figure.

**Inputs and Outputs** The machine learning works on annotations. Its input is a corpus of annotated documents. The annotations in GATE have parameters called “features”. And the task machine learning performs is it learns to assign correct value of a particular feature (called and declared as “class feature” in the configuration) of a particular annotation. Which means it does not learn to assign new annotations.

**Input Annotations** The figures having annotations assigned during figure occurrences matching may have these annotations used in machine learning. The rest, however, must also have an annotation with a class feature. Generally, it will always be either a token annotation or a sequence of token annotations from ANNIE English Tokeniser.

**Engine** The Machine Learning module has multiple learning engines [18].

The machine learning example from GATE training modules [20] provides a comprehensive introduction to the topic. The machine learning follows the language of annotations.
5.3 Data

Our input data are Web pages from two topics: Notebook e-shop pages and package tours. The domains of the pages are:

- **Notebook e-shop pages**
  - shop.lenovo.com (3 pages)
  - us-store.acer.com (3 pages)
  - www.amazon.com (3 pages)
  - www.argos.co.uk (3 pages)
  - www.atlastravelweb.com (3 pages)
  - www.bestbuy.com (3 pages)
  - www.dell.com (12 pages)
  - www.ebay.com (6 pages)
  - www.laptopsdirect.co.uk (3 pages)
  - www.razerzone.com (3 pages)
  - www.shop.bt.com (3 pages)
  - www.walmart.com (3 pages)

- **Package Tours**
  - france.rezdy.com (3 pages)
  - us.makemytrip.com (3 pages)
  - www.atlastravelweb.com (3 pages)
  - www.city-discovery.com (3 pages)
  - www.getyourguide.com (3 pages)
  - www.globusjourneys.com (3 pages)
  - www.go-today.com (3 pages)
  - www.italy-travel.net (3 pages)
  - www.libertytravel.com (3 pages)
  - www.taketours.com (3 pages)
  - www.tours4fun.com (3 pages)
  - www.travelallrussia.com (3 pages)
  - www.virgin-vacations.com (3 pages)
  - www.zicasso.com (3 pages)

The pages were chosen from the results of Google Search. As they come from various domains, they cover multiple ways of formatting the information we are trying to extract.
5.3.1 Training Data

Apart from the naïve solution, we are also implementing two solutions that are learning. Which makes it necessary to split the data set into training documents and testing documents.

A reminder, we are going to infer both domain-specific and general rules; we will train the documents on major part of the set and evaluate on the smaller portion. Where there are 3 pages, we will use 2 of them as training pages and then evaluate in two ways:

1. Evaluate on the whole data set.
2. Evaluate on the pages outside the training set.

5.4 Naïve Matcher

The naïve matcher is made up of two matchers: XPath matcher and close words matcher. The basis these matchers will operate over is the GATE figure occurrences matching. The matcher will thus only assign relevance to the possible figure occurrences matched by GATE.

The concept of the naïve rules is outlined in the chapter 1.4.3 in short, we want to match the occurrences by the surrounding elements and their class names on one hand, and by the preceding words on the other. The matchers will be run separately and their results will be joined together, summing up ratings of the occurrence across the matchers. At this point, one method does not seem more relevant than the other, therefore the rating assigned by both of the matchers has the same relevance for us. Therefore, we arbitrarily set 1 as the total rating each of the matchers has and can distribute among the occurrences.

5.4.1 XPath Rules

The initial idea — to match elements having a particular class name for a particular figure — needs modifications.

The first problem to face is that the class names do not simply have “price” among their class names (not to mention “price” is scarcely the only class name). We will generalize this rule into the following regular expression:

```
^(.+\s[-\s]+)?\{"\s.*\}\$\.
```

“{0}” will be replaced by the respective class name and it will be sought surrounded by either hyphens, which is a common delimiter within class names composed of multiple words, or whitespaces, which is delimiter of class names.

The second problem is that the content of elements matched by the query: the elements may be wrapping more than just the figure, think of this:

```
<div class='pl-price'><label>Price:</label>$399.99</div>
```

Which means we need to search for what GATE provided us as a substring of inner text of the result elements.

All these considerations are implemented in the resulting code:

```
Listing 5.1: Matching elements by XPath
1 | var classMatcher = new Regex(string.Format(
```
"^\.(\\s*-\)?\{0}\(\\s\-\).\?)?$",
figureKeywords[figure.Name], RegexOptions-ignoreCase);
var doc = new HtmlDocument();
...
using (var fileReader = File.OpenRead(filePath))
{
    doc.Load(fileReader);
}

var candidateMatches = doc
    .GetElementsByPredicate(element => element
        .Attributes
            .Cast<HtmlAttribute>()
                .SingleOrDefault(attr =>
                    attr.Name == "class" &&
                    classMatcher.IsMatch(attr.Value)) != null);

var resultMatches = candidateMatches
    .Where(match =>
        !match
            .Descendants()
                .Any(candidateMatches.Contains))
        .SelectMany(match => match.DescendantsAndSelf())
        .ToList();

var resultMatchesCount = resultMatches.Count();
var perfectMatches = resultMatches
    .Where(match =>
        occurrences.Contains(match.InnerText));
if (perfectMatches.Any())
{
    // ... process results.
}
else
{
    var imperfectMatches = resultMatches.SelectMany(match =>
        occurrences.Where(occurrence =>
            match.InnerText.Contains(occurrence));
    // ... process results.
}

Returning the result of the algorithm is omitted in this listing, to allow us
focus on the matching for now.

Lines 1 to 4 contain initialization of the regular expression, and the lines
5 to 10 contain initialization of the document.

The expression at line 12 selects all elements with their class name match-
ing our regular expression. This is just the first step, because we still need to
check the contents for matches provided by GATE. This filtering is performed
in the expression at lines 27 and 36; the difference is in the first case, elements
having an occurrence as its only content are selected, while in the second case
the condition is weaker - if the occurrence is a substring of the element contents,
it is selected.

The reason for this differentiation is that if the occurrence is just a substring in
the contents, it might have been created by concatenation of descendant elements.

The result of the method is list of occurrences with ratings assigned. We are
able to distinguish three types of matches:

1. Matches wrapped “tightly” by an element.


3. Contents of an element with a particular class, or its descendants - not
matched by GATE previously.

Within these groups, the matches cannot be distinguished. We have set 1
as the sum of ratings assigned by the matcher among the occurrences. We can
further divide the rating: certainly the last group of matches is the least relevant
as long as it does not fit any candidates matching syntactically. For that reason,
we define our metric in the following way:

- If only the third group of matches is found, distribute all 1 equally between
  them.
- If the first group of matches is not empty, distribute 0.8 equally between
  them and 0.2 between the third group.
- Otherwise, distribute 0.7 equally among the members of the second group
  and 0.3 equally among the members of the third group.

The numbers 0.8 and 0.7 (and their respective complements to 1) were chosen
arbitrarily to provide dominance to the matches provided by GATE. The exper-
iments will show whether this choice was accurate; we only want to prevent the
third-type of matches to outweigh the first- and the second-type matches.

5.4.2 Close Words Rules

Empirically, we can see that the figure is often preceded by a word or a phrase
introducing the figure, e. g. “Memory: 2GB”. The word or phrase can also follow
the figure, e. g. “a laptop with 2GB memory”. Inferring specific rules is a matter
of machine learning, this method serves as a proof of concept, so we are not going
to simulate the work of machine learning by specifying the rules manually.
5.4.3 Results

<table>
<thead>
<tr>
<th>Figure</th>
<th>Total Occurrences</th>
<th>Matched Occurrences</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Price</td>
<td>31</td>
<td>15</td>
<td>48.39 %</td>
</tr>
<tr>
<td>RAM Size</td>
<td>31</td>
<td>14</td>
<td>45.16 %</td>
</tr>
<tr>
<td>Disk Size</td>
<td>30</td>
<td>22</td>
<td>73.33 %</td>
</tr>
<tr>
<td>CPU Performance</td>
<td>29</td>
<td>11</td>
<td>37,931034827586 %</td>
</tr>
<tr>
<td>Vacation Price</td>
<td>47</td>
<td>18</td>
<td>38,2978723404255 %</td>
</tr>
<tr>
<td>Destination</td>
<td>46</td>
<td>2</td>
<td>4.35 %</td>
</tr>
<tr>
<td>Diet</td>
<td>30</td>
<td>4</td>
<td>13.33 %</td>
</tr>
<tr>
<td>Duration</td>
<td>45</td>
<td>1</td>
<td>2.22 %</td>
</tr>
<tr>
<td>Total</td>
<td>289</td>
<td>76</td>
<td>26.3 %</td>
</tr>
</tbody>
</table>

Table 5.1: Accuracy of trivial matcher by figure

5.5 GATE Matcher

Our GATE matcher will utilize Batch Learning plugin of GATE [26]; the basics of this plugin are introduced earlier in this chapter.

We tried the following configurations, successively:

Iterating over the annotations being matched, providing preceding and succeeding Token annotations as the context to be learnt:

Listing 5.2: Batch Learning configuration iterating over figure annotations with surrounding tokens

```
<DATASET>
  <INSTANCE-TYPE>CPUPerformance</INSTANCE-TYPE>
  <ATTRIBUTELIST>
    <NAME>WordBefore_1</NAME>
    <SEMTYPE>NOMINAL</SEMTYPE>
    <TYPE>Token</TYPE>
    <FEATURE>string</FEATURE>
    <RANGE from="-6" to="2"></RANGE>
  </ATTRIBUTELIST>
...
  <ATTRIBUTE>
    <NAME>Class</NAME>
    <SEMTYPE>NOMINAL</SEMTYPE>
    <TYPE>CPUPerformance</TYPE>
    <FEATURE>marked</FEATURE>
    <POSITION>0</POSITION>
    <CLASS/>
  </ATTRIBUTE>
</DATASET>
```

However, it turned out that the INSTANCE-TYPE defines the elements the batch learning will iterate over - including the look-ahead and look-behind. Which
means that in this configuration, all surrounding words to the instance of e.g.
CPU Performance was ignored and there was nothing to learn.

As a solution, we tried changing the instance type to Token, which showed no
improvement, either.

The final solution is to preprocess the document: for each of the figure annota-
tions, we will collect the surrounding token contents and store them in the figure
annotation as attributes.

The configuration file corresponding to this approach is this:

Listing 5.3: Batch Learning configuration for tokens inlined as figure annotation
attributes

```xml
<DATASET>
  <INSTANCE-TYPE>CPUPerformance</INSTANCE-TYPE>
  <ATTRIBUTE>
    <NAME>WordBefore_1</NAME>
    <SEMTYPE>NOMINAL</SEMTYPE>
    <TYPE>CPUPerformance</TYPE>
    <FEATURE>WordBefore_1</FEATURE>
    <POSITION>0</POSITION>
  </ATTRIBUTE>
...
  <ATTRIBUTE>
    <NAME>Class</NAME>
    <SEMTYPE>NOMINAL</SEMTYPE>
    <TYPE>CPUPerformance</TYPE>
    <FEATURE>marked</FEATURE>
    <POSITION>0</POSITION>
  </ATTRIBUTE>
</DATASET>
```

The preprocessing is performed using the API of GATE Embed. Having
the figure annotation already matched, the following loop inserts the contents of
preceding token annotations into the properties of the figure annotation:

Listing 5.4: Inserting token annotation contents into figure annotation

```java
for (int preceding = 1; preceding <= PRECEDING_CONTEXT_LENGTH && precedingT
String value = text.substring(
    getAnnotationStart(tokens.get(precedingTokenIndex)),
    getAnnotationEnd(tokens.get(precedingTokenIndex)));
annotationFeatures.put(
    getPrecedingAttributeName(preceding),
    value);
}
```

Here, the `annotationFeatures` is the map containing the properties of the fig-
ure annotation.
5.5.1 Results on the whole data set

By Figure

<table>
<thead>
<tr>
<th>Figure</th>
<th>Matched Occurrences</th>
<th>Total Occurrences</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Price</td>
<td>21</td>
<td>37</td>
<td>56,76%</td>
</tr>
<tr>
<td>RAM Size</td>
<td>9</td>
<td>22</td>
<td>40,91%</td>
</tr>
<tr>
<td>Disk Size</td>
<td>10</td>
<td>14</td>
<td>71,43%</td>
</tr>
<tr>
<td>CPU Performance</td>
<td>19</td>
<td>29</td>
<td>65,52%</td>
</tr>
<tr>
<td>Vacation Price</td>
<td>16</td>
<td>31</td>
<td>51,61%</td>
</tr>
<tr>
<td>Destination</td>
<td>1</td>
<td>18</td>
<td>5,55%</td>
</tr>
<tr>
<td>Diet</td>
<td>7</td>
<td>21</td>
<td>33,33%</td>
</tr>
<tr>
<td>Duration</td>
<td>19</td>
<td>32</td>
<td>59,375%</td>
</tr>
<tr>
<td>Total</td>
<td>102</td>
<td>204</td>
<td>50%</td>
</tr>
</tbody>
</table>

Table 5.2: Accuracy of GATE matcher on the whole data set by figure

By Web domain

The results by Web domain can be found in Attachment 21 or Attachment 25.

5.5.2 Results on the training data set

By Figure

<table>
<thead>
<tr>
<th>Figure</th>
<th>Matched Occurrences</th>
<th>Total Occurrences</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Price</td>
<td>2</td>
<td>10</td>
<td>20,00%</td>
</tr>
<tr>
<td>RAM Size</td>
<td>2</td>
<td>10</td>
<td>20,00%</td>
</tr>
<tr>
<td>Disk Size</td>
<td>3</td>
<td>9</td>
<td>33,33%</td>
</tr>
<tr>
<td>CPU Performance</td>
<td>3</td>
<td>8</td>
<td>37,50%</td>
</tr>
<tr>
<td>Vacation Price</td>
<td>4</td>
<td>13</td>
<td>30,77%</td>
</tr>
<tr>
<td>Destination</td>
<td>1</td>
<td>14</td>
<td>7,14%</td>
</tr>
<tr>
<td>Diet</td>
<td>1</td>
<td>10</td>
<td>10,00%</td>
</tr>
<tr>
<td>Duration</td>
<td>3</td>
<td>14</td>
<td>21,43%</td>
</tr>
<tr>
<td>Total</td>
<td>19</td>
<td>88</td>
<td>21,59%</td>
</tr>
</tbody>
</table>

Table 5.3: Accuracy of GATE matcher on the training data set by figure

By Web domain

The results by Web domain can be found in Attachment 23 or Attachment 26.

5.6 Heuristic Matcher

For comparison, we will also implement a heuristic. The heuristic will extend the thoughts from the naïve solution but will also provide some learning. The idea
of the naïve solution is we have a fixed set of keywords for each of the figures and
the occurrences are matched if they follow a keyword immediately.

Our heuristic will compute a simple statistic of the surrounding tokens for
each of the matches in the training data. This statistic will contain all the words
that occurred in neighbourhood of an occurrence in the training set, with average
relative position and incidency.

When applying the heuristic, we will iterate over the possible matches and
examine the words in their neighbourhoods; for each of these words, the closer its
relative position to that from the statistic, the better, and the higher its incidency,
the better (see also 1.4.3).

Our experiments will be run with relevance of the attribute values 0.1. We
chose this number to allow some influence of the attributes but to preserve
the dominancy of the surrounding text; this way, we believe the results of the match-
ers can be compared better with GATE deciding based on the surrounding text
exclusively.

5.6.1 Results on the whole data set

By Figure

<table>
<thead>
<tr>
<th>Figure</th>
<th>Matched Occurrences</th>
<th>Total Occurrences</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Price</td>
<td>16</td>
<td>39</td>
<td>41.03%</td>
</tr>
<tr>
<td>RAM Size</td>
<td>16</td>
<td>39</td>
<td>41.03%</td>
</tr>
<tr>
<td>Disk Size</td>
<td>12</td>
<td>39</td>
<td>30.77%</td>
</tr>
<tr>
<td>CPU Performance</td>
<td>25</td>
<td>39</td>
<td>64.1%</td>
</tr>
<tr>
<td>Vacation Price</td>
<td>14</td>
<td>40</td>
<td>35%</td>
</tr>
<tr>
<td>Destination</td>
<td>16</td>
<td>43</td>
<td>37.21%</td>
</tr>
<tr>
<td>Diet</td>
<td>21</td>
<td>34</td>
<td>61.76%</td>
</tr>
<tr>
<td>Duration</td>
<td>33</td>
<td>43</td>
<td>76.74%</td>
</tr>
<tr>
<td>Total</td>
<td>139</td>
<td>279</td>
<td>49.82%</td>
</tr>
</tbody>
</table>

Table 5.4: Accuracy of Heuristic matcher by figure

By Web Domain

The results by Web domain can be found in Attachment 17 or Attachment 25.
### 5.6.2 Results on the training data set

**By Figure**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Matched Occurrences</th>
<th>Total Occurrences</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Price</td>
<td>2</td>
<td>10</td>
<td>20,00%</td>
</tr>
<tr>
<td>RAM Size</td>
<td>5</td>
<td>10</td>
<td>50,00%</td>
</tr>
<tr>
<td>Disk Size</td>
<td>4</td>
<td>9</td>
<td>44,44%</td>
</tr>
<tr>
<td>CPU Performance</td>
<td>7</td>
<td>8</td>
<td>87,50%</td>
</tr>
<tr>
<td>Vacation Price</td>
<td>11</td>
<td>13</td>
<td>84,62%</td>
</tr>
<tr>
<td>Destination</td>
<td>9</td>
<td>14</td>
<td>64,29%</td>
</tr>
<tr>
<td>Diet</td>
<td>9</td>
<td>10</td>
<td>90,00%</td>
</tr>
<tr>
<td>Duration</td>
<td>13</td>
<td>14</td>
<td>92,86%</td>
</tr>
<tr>
<td>Total</td>
<td>60</td>
<td>88</td>
<td>68,18%</td>
</tr>
</tbody>
</table>

Table 5.5: Accuracy of Heuristic matcher by figure

**By Web Domain**

The results by Web domain can be found in Attachment 19 or Attachment 26.
6. Evaluation

6.1 Results Summary

The base of the result data set are records indicating whether a particular method on a particular Web page succeeded in matching figure occurrence on that page. We aggregated these items by the method, the figure and the Web domain of the pages, respectively. The aggregations of the results is provided in the end of the chapter [5]. The aggregations allow us to review three aspects of figure matching:

- Which method performed the best in general.
- Which method performed the best on each of the domains.
- How big a training set should be.
- How big the diversity of page structure with respect to the difficulty of extracting information automatically is.

The following chart plots the methods and their accuracy against the particular figures on the whole data set:

![Comparison plot of matching methods on all figures and the whole data set](image)

Figure 6.1: Comparison plot of matching methods on all figures and the whole data set
This chart plots the methods and their accuracy against the page types on the whole data set:

![Comparison plot of matching methods](image)

Figure 6.2: Comparison plot of matching methods on different page types and the whole data set
The following chart plots the methods and their accuracy against the particular figures on the testing data set:

Figure 6.3: Comparison plot of matching methods on all figures and the whole data set

This chart plots the methods and their accuracy agains the page types on the testing data set:

Figure 6.4: Comparison plot of matching methods on different page types and the whole data set
6.2 Results Interpretation

6.2.1 Matching Methods

There were two matching methods that were learning on a training set, and a naïve solution. The figure 6.2 shows that in general the both learning methods performed better than the trivial solution, which is a minimal requirement for a solution developed with the ambition to be considered intelligent.

The trivial solution with its accuracy around 30% provided a confirmation that in some cases the relation between figure occurrences and their context on Web pages can be inferred automatically.

The fact that the both learning methods showed better results on all page types and figures also shows that machine learning is a way of achieving better results for this problem.

Absolutely, the accuracy around 50% is not acceptable for a final product, however, which makes both these methods in their current state a mere prototypes for further development.

6.2.2 Page Types

In our page set, we had two types of Web pages: notebook e-shop pages and package tour offer pages. These page types had disjoint set of figure which means that aggregation over figures also shows the performance on each of the page types. An explicit plot is given in the figure 6.2.

Despite none of the matching methods failed on any page type completely, we can see both GATE and trivial matching method performed better on notebook e-shop pages than on package tours. Heuristic, on the contrary, was more successful on package tour. This is rather surprising outcome - the notebook e-shop pages have a more tabular structure compared to package tours where a lot of information is inlined inside plain text.

The possible explanation of this fact can be that heuristic matcher does not learn from counterexamples, it only does statistic from positive examples. Which means it does not trace differences between correct and incorrect occurrences. A simple example of this situation is the following: on a laptop e-shop page, there is a “see more” widget with the offer of similar laptops. These laptops might have something like “price: $499.99” in their details. The price of the laptop we are looking at is also preceded by the word “price”. But the heuristic learns from the example only, and not from the counterexample and thus matches the counterexample the same.

6.2.3 Figures

We have not observed any remarkable generalizable differences in matcher performance on the lavel of figures apart from those given by page types.

6.2.4 Domains

There were more than 25 domains in the test set and having a plot containing performance of the matchers on them would not bring much value. However,
there were two domains that differed by the count of pages: www.dell.com (11 pages) and www.ebay.com (6 pages). Despite it may be a coincidence, we can see in the [Attachment 25] that the accuracy of GATE matcher was high above average on these Web domains, which suggests good training data sets should be of size 5, 10 or even larger.

6.2.5 Training versus Training Data Set

Whereas the trivial and the heuristic matchers do not show any drop of accuracy on the testing data set (which is given by design in case of trivial matcher), the accuracy of GATE matcher dropped significantly. As a likely explanation it seems that small training data set does not provide sufficient “confidence”.

6.2.6 Universal versus Domain-Specific Rules

The accuracy of trivial and the heuristic matchers do not perform differently using the universal and domain-specific rules. GATE showed big difference and it could be understood as another confirmation of the need of bigger training set.

6.2.7 Summary

Trivial Matcher  The trivial matcher provides consistent results, which is probably given by its static nature. It performed worse on package tour pages, on the figures destination and diet in particular (see 6.1 and 6.3). This might be given by a broader vocabulary of words surrounding the respective features than those inlined in the static regular expressions.

Heuristic Matcher  The heuristic matcher was generally the most successful, particularly on the testing set. The most simple explanation is that while the GATE matcher is providing boolean decision on candidate occurrence and has a threshold; if this threshold is not surpassed, it does not mark the occurrence as the matched value. Heuristic matcher simply returns the most probable match, regardless of the probability.

GATE Matcher  The GATE matcher performed well when applied on the training data set but wrong on the testing data set (can be seen when subtracting the results on testing data set 6.3 from those on the whole data set 6.1). When applied to testing data set, GATE performed remarkably better when using the universal rules. As there are two training documents per domain, it is possible this amount does not provide sufficient probability.
7. Conclusion

In this chapter, we provide an overview of the thesis, its results in particular.

7.1 The Problem

The problem seemed challenging and very broad from start, and this impression was eventually confirmed. Our concept of matching figures in the Web pages automatically had few closely related resources, because the contemporary solutions and research follow different approaches, the following in particular:

- The Web page is previously annotated with semantic metadata by its author.
- The tool for matching pages is trained for a particular document template.

The overview studies by A. Laedner [1] and E. Ferrara [2] made the exploration of the context of the problem much easier. Both these studies cited a few tools for Web data extraction. The fact we failed to obtain most of these tools was both a surprise and an inconvenience as it deprived us of the possibility to connect these tools to our system and compare their results.

7.2 The Solution

Our solution provides an infrastructure for collecting training data for figure matching in Web pages. This infrastructure contains a server part with data storage and Web API and a browser add-on being the client part. The add-on works with the DOM of the currently loaded page, modifying it so that the user can mark figure occurrences by either clicking boxes wrapping candidate occurrences or marking the figures with text selection.

Apart from the infrastructure, we provide three implementations of figure matchers, all implementing a common interface. The common interface implies the system can be further developed and enhanced with new figure matchers.

Our solution depends strongly on the features of GATE. GATE performs static text analysis, which is utilized to find candidate figure occurrences in the text; all three figure matchers are picking from these candidates. One of the figure matchers also utilizes the machine learning plugin of GATE.

7.3 The Results

There are two parts of the results: the implemented software and the results of the experiments with our figure matchers. The implemented software is described in the chapters [1] and [3]. The software became our environment for collecting data and running the experiments. From our experience, it served the purpose and also is prepared for further enrichment with new approaches of matching figures.
There were three figure matchers utilized in the thesis. The first was the trivial matcher we implemented as a proof of concept of the thesis. Its accuracy was between 30 and 40%, which we accepted as a confirmation the concept is correct.

The other two matchers already employed machine learning. GATE machine learning, which was accessed through our wrapper application, was a black box for us and it took some effort to find a way to get the machine learning working on our assignment. In the end, GATE provided better results then the trivial matcher, however its accuracy was insufficient for a solution applicable in a practical application. It is likely that providing a significantly larger training data set would lead to a significant improvement of the results.

The heuristic matcher, on the other hand, showed accuracy around 50% in average, peaking over 70% in one figure on the whole data set and peaking over 80% in several figures on the testing data set. However, the absence of learning on the counterexamples means the heuristic matcher is essentially “guessing” and its results cannot be considered that reliable as those provided by GATE.

7.4 Open Problems

Automatized figure matching in Web pages is a very wide problem and this thesis provided just a prototype of solution on a limited subset of data.

The directions of further research and development might be:

- Collecting a larger training data set, more entity types and more figures.
- Trying more configurations of GATE matcher and heuristic matcher.
- Extending the scope of the matchers to other languages then English.
- Implementing a background service that would continuously poll new matched Web pages substituting manual launching of the learning procedures.

7.5 Reflection

The thesis continued with the topic that was treated by a few theses in our faculty previously. The contributions we see in this thesis are:

- The approach to the problem of automatized figure matching
- The software for data collection and running the tasks on the matcher modules
- The utilization of the GATE software in context of this problem.

The failures we see in this thesis are:

- Insufficient training data set which possibly deprived us of better results of the machine learning and the analysis part
- Too narrow scope on the contemporary tools: focusing on Web data mining left text data mining out of the scope. The tools for text data mining possibly could be employed in the thesis.
With respect to the original assignment we consider the most of it fulfilled, with a major exception being the system architecture. In the assignment we declared to download the rules for matching the figures and figure candidates to the client application. This turned out to be very complicated – as an example, location is looked up by a GATE module (Stanford NLP) having 60MB of source data. This made us implement all figure matching algorithms on the server.
Bibliography


[31] Fišer, Dominik Sémantická anotace doménově závislých dat Univerzita Karlova v Praze, Matematicko-Fyzikální fakulta, 2011 Attachment 13

[32] Maruščák, Dušan Dolovanie a mapovanie ontológií s užívateľskou preferenciou Univerzita Karlova v Praze, Matematicko-Fyzikální fakulta, 2007 Attachment 13

[33] Kýpet, Jakub Semantic annotation and querying RDF data Univerzita Karlova v Praze, Matematicko-Fyzikální fakulta, 2014 Attachment 13
# List of Figures

1.1 Elementary architecture of the system. .................................. 19
1.2 Outline of the server architecture - component viewpoint. ........ 21

2.1 GATE GUI Overview ......................................................... 25
2.2 GATE GUI - CREOLE Plugin Manager ................................. 26
2.3 GATE GUI - Creating an application .................................. 27
2.4 GATE GUI - Adding instances of processing resources .......... 27
2.5 GATE GUI - Matched Money annotations ............................. 28
2.6 GATE GUI - Matched custom regular expression .................... 29

3.1 Architecture of the browser add-on .................................... 36
3.2 Add-on user interface ..................................................... 37
3.3 Architecture of the server part ......................................... 43

4.1 Overview of the add-on UI ................................................. 56

6.1 Comparison plot of matching methods on all figures and the whole data set ......................................................... 71
6.2 Comparison plot of matching methods on different page types and the whole data set ................................................. 72
6.3 Comparison plot of matching methods on all figures and the whole data set ......................................................... 73
6.4 Comparison plot of matching methods on different page types and the whole data set ................................................. 73
List of Tables

5.1 Accuracy of trivial matcher by figure . . . . . . . . . . . . . . . . 66
5.2 Accuracy of GATE matcher on the whole data set by figure . . . 68
5.3 Accuracy of GATE matcher on the training data set by figure . . 68
5.4 Accuracy of Heuristic matcher by figure . . . . . . . . . . . . . . . 69
5.5 Accuracy of Heuristic matcher by figure . . . . . . . . . . . . . . . 70
8. Attachments

All attachments are saved on the DVD attached to the thesis. All file paths are relative to the root directory of the DVD.

8.1 Executables

Attachment 1 GATE Installer: 
Installers/gate-8.1-build5169-installer-win.exe

Attachment 2 Web API deployable to the server: 
/Deploy/WebAPI

Attachment 3 Windows command line tool for running matching procedure:
/Deploy/Launcher.exe

Attachment 4 Mozilla Firefox add-on 
/Deploy/FigureFinder.xpi

Attachment 5 Java GATE Wrapper application 
/Deploy/GateWrapper.jar

8.2 Database

Attachment 6 The file 
/Data/Database/StructureOnly.bak contains the definition of the server database only.

Attachment 7 The file 
/Data/Database/StructureMetaData.bak contains the definition of the server database with definitions of the entity types.

Attachment 8 The file 
/Data/Database/StructureData.bak contains the database with state by the time of finishing the thesis.

8.3 Source Codes

Attachment 9 The folder 
/Sources/AddOn contains the source codes of the browser add-on.

Attachment 10 The folder 
/Sources/Server contains the Visual Studio solution containing the source codes of Web API, the occurrence service and all matchers plus the libraries.

Attachment 11 The folder 
/Sources/GATEWrapper contains the source codes of the Java wrapper application written for communication with GATE.

8.4 Literature

Attachment 12 The folder 
/Articles contains the articles cited in this thesis.

Attachment 13 The folder 
/Theses contains the theses cited in this thesis.

8.5 Text of the Thesis

Attachment 14 The folder 
/Text contains the text of this thesis.

89
8.6 Results

Attachment 15 The journal from running trivial algorithm on all data is on path /Journals/trivial-alldata-allfigures.log.

Attachment 16 The journal from running trivial algorithm on testing data is on path /Journals/trivial-testingdata-allfigures.log.

Attachment 17 The journal from running heuristic algorithm on all data is on path /Journals/heuristic-alldata-allfigures.log.

Attachment 18 The journal from running heuristic algorithm on all data using universal rules is on path /Journals/heuristic-alldata-universal.log.

Attachment 19 The journal from running heuristic algorithm on testing data is on path /Journals/heuristic-testingdata-allfigures.log.

Attachment 20 The journal from running heuristic algorithm on testing data using universal rules is on path /Journals/heuristic-testingdata-universal.log.

Attachment 21 The journal from running GATE algorithm on all data is on path /Journals/gate-alldata-allfigures.log.

Attachment 22 The journal from running GATE algorithm on all data using universal rules is on path /Journals/gate-alldata-universal.log.

Attachment 23 The journal from running GATE algorithm on testing data is on path /Journals/gate-testingdata-allfigures.log.

Attachment 24 The journal from running GATE algorithm on testing data using universal rules is on path /Journals/gate-testingdata-universal.log.

Attachment 25 The aggregated results of all algorithms on whole data set with charts is on /Charts/results-alldata.ods (OpenOffice format).

Attachment 26 The aggregated results of all algorithms on testing data set with charts is on /Charts/results-testingdata.ods (OpenOffice format).