Bachelor Thesis

Dataset Format Analysis and Column Classification for CompleteSearch

Olivier Puraye

20th August 2018

Albert-Ludwigs-University of Freiburg
Faculty of Engineering
Chair of Algorithms and Data Structures

Processing Time
27.07.2018 - 27.10.2018

Reviewer
Prof. Dr. Hannah Bast

Supervisor
Prof. Dr. Hannah Bast

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Contents

Abstract

In this thesis, we analyse tabular text files with the objective to find suitable parameters to make them searchable using the features of CompleteSearch. We solve this problem in two separate stages.
In a first stage, we build a file analyser that parses the text files to gather relevant features related to the format of their content.
In a second stage, we use a Naive Bayes classifier to associate the columns in the text files to their appropriate CompleteSearch parameter classes.
Furthermore, we integrate this automatic parameter suggestion into the CompleteSearch web application and incorporate its user feedback into the classifier training in order to further improve the suggested CompleteSearch configurations over time.

Zusammenfassung

In dieser Arbeit analysieren wir tabellarische Textdateien mit dem Ziel geeignete Parameter zu finden um sie mit den Funktionen von CompleteSearch durchsuchen zu können. Dabei unterteilen wir das Problem in zwei Etappen.
Im einem ersten Schritt analysieren wir die Input Datei und entnehmen ihr diverse Eigenschaften, die die Formatierung ihres Inhalts wiederspiegeln.
In einem zweiten Schritt, benutzen wir den Naive Bayes Klassifizierungsalgorithmus um den Spalten in der Textdatei passende CompleteSearch Parameter zuweisen zu können.
Darüber hinaus, integrieren wir diese automatische Parameterkonfigurierung in die CompleteSearch Web Applikation und nehmen das darüber erhaltenen Nutzer-Feedback in den Trainingssatz unseres Klassifikators auf um die vorgeschlagenen Konfigurationen stetig verbessern zu können.

1. Introduction

1.1. Motivation

This thesis is a continuation of my Bachelor Project [Pur17], which had the primary objective to develop a web application for searching any table-structured text file using CompleteSearch [Bas].
In this thesis, we will focus more in-depth on the analysis of the input file emphasized in Figure 1.1. The goal is to improve the parameter suggestions and to gather more data about the structure of the input file and use a classifier to configure CompleteSearch.

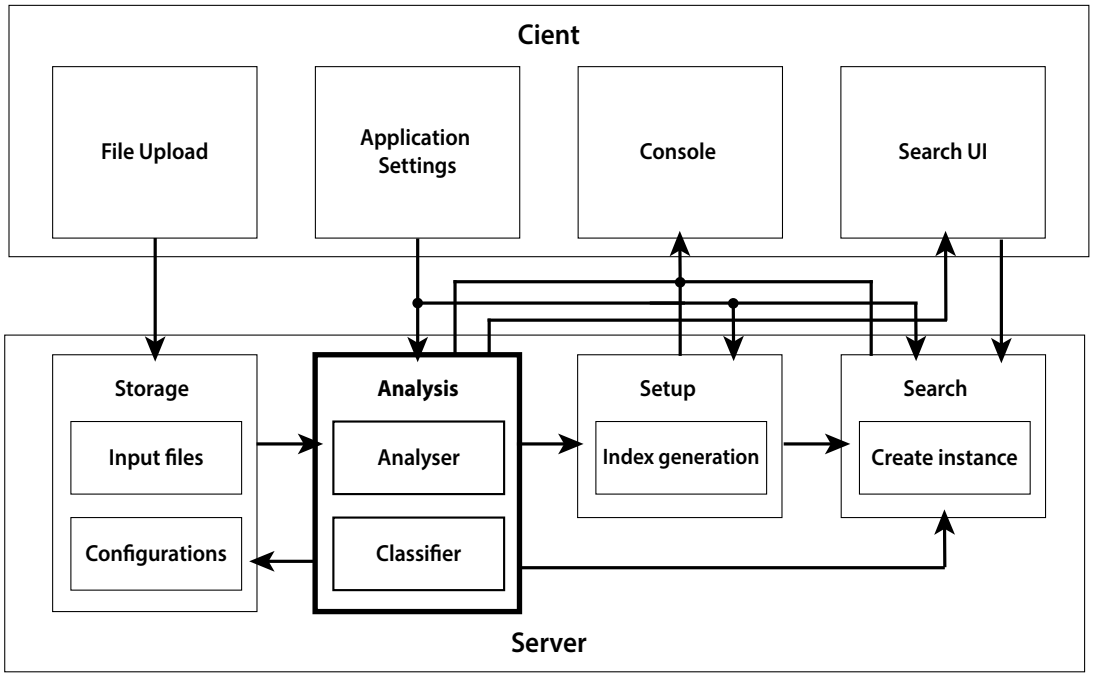

Figure 1.1.: Application architecture

Furthermore, the user feedback that is provided by changing parameters via the web application will be used to improve the classifier's performance and thus find better configurations through continuous usage.

1.2. Problem Definition

Searching in text files can quickly become slow and inefficient. For this reason, search engines convert the input file into multiple different index files. Their content is generally dependent on the configuration of the search engine, which needs to be manually adjusted to fit the input file that should be searched.

In the following, we will only discuss the case of CompleteSearch, which is designed to search table-structured text-files, such as CSV and TSV files.
For generating the needed indexes for CompleteSearch, we first need to specify all the following parameters for a given input file:

- The *column-separator* defines the separator that delimits the columns in the tabular dataset.
- The *full-text* vector specifies which columns should be searched on a simple search query that solely relies on the query term and doesn't involve the use of any other CompleteSearch features, such as filters or facets.
- The *filter* vector lists the columns for which the filtering feature can be enabled. By choosing a column to be a filter, the search query is restricted to that specific column.
- The *facet* vector contains the columns that can be used to further refine the search results by specifying explicit values for those columns.
- *within-field-separator* defines the separator that delimits subitems within a column item. It is not column-specific as it is the same for the entire file. *within-field-separator* is synonymous with *subitem separator*, which will be used in the remainder of this thesis.
- The *allow-multiple-items* vector indicates to which columns the *subitem separator* can be applied to. We will refer to this parameter as *allow-subitems* in remainder of the thesis, which is easier to comprehend contextually.
- The *field-format* vector describes the format of the data in the columns. It can be plain text, JSON or XML. Unspecified columns are treated as plain text by default.
- The *show* vector lists the columns, from which items should be shown in search results.
- The *excerpt* vector determines the columns for which only the section containing the query words should be shown. This is intended for columns that have items that are too long to be shown in their entirety in the search results.
- The *ordering* vector indicates how the columns are ordered. By default items are ordered lexicographically. Alternatively, they can be ordered numerically or by date, which causes their values to be converted such that their actual

ordering corresponds to the lexicographical ordering. For ordering numbers this requires the maximal count of integer and decimal places. Furthermore, this opens the possibility to use range inputs for numerical values or date inputs for adjusting the facets in the web application.

Apart from the parameters for the CompleteSearch index generation, we also require a few additional settings for the web application user interface.

- The *url* vector specifies the columns that contain links. This enables the URLs to be converted from plain text into functional links.
- The *email* vector specifies the columns that contain email addresses. Like the for the URLs, they are converted from plain text to a functional `mailto:` link.
- The *label* vector lists the columns that should show their column name next to their value in search results. This improves the informative value of the search results and is especially useful to put numerical values into context.

In the following chapters, we will proceed to find an approach to determine all these parameters automatically for any provided input file.

In Table 1.1, we have a dummy movie dataset for which a corresponding CompleteSearch configuration is given in Table 1.2.

Nr	Title	Year	Director	Actors
1	Back to the Future	1985	Robert Zemeckis	Michael J. Fox|Christopher Lloyd|Lea Thompson
2	Ghostbusters	1984	Ivan Reitman	Bill Murray|Dan Aykroyd| Sigourney Weaver
3	Inception	2010	Christopher Nolan	Leonardo DiCaprio|Joseph Gordon-Levitt|Ellen Page

Table 1.1.: Movie Dataset

Parameter	Value
full-text	Title, Director, Actors
filter	Title, Director, Actors
facet	Year, Director, Actors
subitem-separator	|
allow-subitems	Actors
field-format	
show	Title, Year, Director, Actors
excerpt	
ordering	Year:4.0
url	
email	
label	Director, Actors

Table 1.2.: CompleteSearch configuration for Movie Dataset in Table 1.1

As previously mentioned, the parameters *field-format* and *ordering* fall back to their default values when they are not explicitly specified.
In this example, all columns are treated as plain text as the *field-format* vector is empty. For *ordering*, we only have the column "Year" with four integer places and no decimal place, consequently all other column are ordered lexicographically.

The process for finding all the necessary configuration parameters for CompleteSearch mainly consists of two stages:

1. *Input File Analysis*
   In this step, we will be extracting all relevant features from the input dataset using the analysis process described in chapter 2.

2. *Column Classification*
   In the second step, we will apply the Naive Bayes classification algorithm to derive the configuration parameters from the features we collected in stage 1. The *Column Classification* is explained in chapter 3.

1.3. Related Work

When focusing on the search aspect of our application that makes any table-structured dataset searchable, this could be achieved by using open-source search toolkits, such as *Open Semantic Search* [Fre] or *Meta-Toolkit* [SM].

However, as the search functionality is not really the topic of this thesis, we will take a closer look at the part of finding suitable parameters to configure the search engine given an input dataset.

Datasette by Simon Willison [Wil] is a conceptually similar to our CompleteSearch web application. It allows to upload a dataset and filter the records by search query and facets. The dataset is being imported into a SQLite database to use SQLite's *full-text* search functionality. *Datasette* also automatically detects facet columns, however the method essentially relies on two basic rules:

- The column has to contain between `1` and `30` unique options
- A facet will only show if it has less unique options than the total number of currently filtered rows

Although during this thesis, we take a broader approach to determine a range of different parameters, we can consider more specific methods for individual parameters. For facet detection, we could take into account cross-column relations. Into this context fits the free-text facet extraction method proposed by Wisam Dakka and Panagiotis Ipeirotis in their paper *Automatic Extraction of Useful Facet Hierarchies from Text Databases* [DI08], which is used to verify if a column contains suitable facets for other columns.

2. Input File Analyser

In this chapter, we describe the analysis process of an input file. We extract and collect features that characterise the structure and the data formats of each column in the input dataset.
Our Input File Analyser essentially converts the input file into a set of different scores, that can then be used to classify the columns in chapter 3.

Usage: `AnalyserMain [options] <inputfile> <outputfile>`

Available options:
`--columnNames: <col1>,<col2>,<col3>,...`
The names of the columns can be manually assigned in the case they are not provided in the first line of the input file.
`--mergeExclusiveProps` combines all mutually exclusive column properties to improve the independence between the retrieved features. This will become important for the column classification in chapter 3, where we are using the Naive Bayes algorithm, which makes strong assumptions on the independence of the training set features.
`--samplingStep <step>` This option can be used when shorter runtime is more important than the highest precision of feature scores. It is especially useful for large input datasets.
`--separators <sep1><sep2><sep3>`
The default separator set `{",", "\t", ";", ".", "|", ":", "#", "/"}` can be replaced by a custom set.

The proceeding in the Input File Analyser can be divided into four main stages:

1. Column separator detection
2. File structure validation
3. Column Parsing
4. File property summary

We will describe the approach of each of these stages in the following sections of this chapter.

2.1. Column separator detection

In this first step, we will start by determining which symbol the file uses to separate its columns. This is necessary to be able to split the file and analyse the columns separately.
The most common separators are commas, tabs and semicolons. By default our analyser supports the following separators: `{",", "\t", ";", ".", "|", ":", "#", "/"}`. As previously mentioned, this set can be customised.

To find the correct separator, we parse the input file and count how often each symbol in the separator set occurs in every row. Tabular data files are required to have the same number of columns in every row. Hence, the number of column separators should be the same in every row. However, instead of accepting the symbol with the most identical counts as the *column separator*, we prefer to use the counts from the first row as reference values. Generally, the first line is less likely to be improperly formatted, furthermore the first line usually contains the column names and is thus least likely to contain any other separator symbols.

2.2. File structure validation

This intermediary step validates the file for the separator determined in section 2.1. It checks if the *column separator* count matches in every row while taking quote-escaped separators into account. In case of mismatches it returns a warning message indicating the row numbers of the faulty records and drops those lines from the analysis. We allow `5%` of faulty records before throwing an error.

2.3. Column Parsing

Having a validated input file and the *column separator* to our disposal, we can now begin to extract features for each column in the file. The column parsing process is executed for each column separately and is made up by the following sub-steps:

1. Item index generation
2. Column-based feature determination
3. Item-based feature determination
    3.1. Item preprocessing
    3.2. Item characterisation
    3.3 Column score calculation
4. Subitem separator detection
5. File property summary

2.3.1. Item index generation

We begin by building an inverted *item index* containing every item from the column together with its occurrence count. Before adding an item to the index, it will first need to be unescaped and trimmed.

$$\begin{array}{c} item_1,\ occurrence\ count\ of\ item_1\ in\ column \\ item_2,\ occurrence\ count\ of\ item_2\ in\ column \\ \vdots \end{array}$$

The purpose of this index is to avoid processing the same item more than once. It serves as foundation for the following sub-steps, in which we will collect the features for every column. For item-based features, the occurrence count recorded in the item index will be used to weight the different items in the calculation of the column feature scores accordingly.

2.3.2. Column-based feature determination

In this subsection we determine the column-based feature scores, which can be calculated without considering the properties of the individual items in the column. With the *item index* from subsection 2.3.1 and with n being the count of entries in this index, we can compute the following column feature scores:

- The *fill rate* shows us the population density of the column.

$$fill\ rate = \frac{\sum_{i=0}^{n} occurrence(item_i) - occurrence(empty\ item)}{\sum_{i=0}^{n} occurrence(item_i)} \tag{2.1}$$

For the remaining features, we need to exclude the *empty item* in order to get accurate scores. Unfortunately the *empty item* is included in the *item index*. We define *empty item* to be the first item $item_0$ in the *item index*, this allows us to easily exclude it in our following equations by starting the summations at $i = 1$.

- The *uniqueness* score gives us an indication on how distinct the items are in a column.

$$uniqueness = \frac{n-1}{\sum_{i=1}^{n} occurrence(item_i)} \tag{2.2}$$

- To record the *item length*, we calculate the mean length $\bar{l}$ and standard length deviation s_l weighted by the occurrences of the individual items in the column.

$$\bar{l} = \frac{\sum_{i=1}^{n} (length(item_i) \cdot occurrence(item_i))}{\sum_{i=1}^{n} occurrence(item_i)} \tag{2.3}$$

$$s_l = \sqrt{\frac{\sum_{i=1}^{n} (length(item_i) - \bar{l})^2 \cdot occurrence(item_i)}{\frac{n-1}{n} \sum_{i=1}^{n} occurrence(item_i)}} \tag{2.4}$$

2.3.3. Item-based feature determination

For the remaining features, we need to analyse every item in the *item index* separately. The process starts by preprocessing the items before they can be evaluated individually. Afterwards their individual item scores are combined and normalised to obtain the column feature scores. We outline this entire procedure in subsubsection 2.3.3.1 to subsubsection 2.3.3.3.

2.3.3.1. Item preprocessing

Preprocessing allows us to considerably reduce how many times we need to parse every item in the *item index*. The primary purpose is to improve the efficiency in the following property retrieval steps.

We start by parsing every item character by character and classifying every character into one of three categories: `Letter`, `Digit` or `Symbol`. Using this classification, it is possible to describe the shape of the item by a sequence of integer values. The assignment looks as follows:

- `Letter` → `-1`
- `Digit` → `1`
- `Symbol` → `0`

The length of those sequences can be reduced by regrouping consecutive letters and digits by adding up their assigned values.

Examples:

- `foo` → `-3`
- `0815` → `4`

Besides the word shape, we also retain a sequence of every symbol that occurred in the item. They will be essential to differentiate between different data formats in the following steps.
Last, we record boolean values for the types of characters that occurred in the item. This prevents unnecessary parsing of the item shape sequence, which helps to reduce runtime especially for longer items. Furthermore, we use these values to create

character type occurrence scores in subsubsection 2.3.3.3.

```
user0815@foo.com   →   {  {-4, 4, 0, -3, 0, -3},
                          {"@", "."},
                          {true, true, true}  }
```

2.3.3.2. Item characterisation

Using the preprocessed items, we are now able to efficiently analyse every item individually. There are still some situations, where we will need to consider the actual content of an item, however the preprocessing helps to reduce these cases considerably.
The item characterisation is done by checking every item against various commonly occurring data types/formats. We do this using a set of different boolean pattern matchers. A pattern matcher returns `true` if an item meets all its constraints or `false` otherwise.

Thus far, the following matchers have been implemented:

0. The *Numeric-value matcher* checks if an item is only composed of digits and/or symbols. Additionally, we detect the decimal and thousands separators and verify their count and if they are ordered correctly.
   Alongside this matcher, we also search for *max integer places* and *max decimal places* in the column. We need these details to prefix the numeric values when generating the search indexes. In doing so, we will be able to order the items according to their numeric value.
   Furthermore, we retrieve the *min numeric value* and *max numeric value*, which will be used as boundaries of an eventual facet range input.
1. The *Incremental-index matcher* compares an item's numeric value with the numeric value of its predecessor in the column to check if it is the predecessor's integer successor.
2. The *Boolean matcher* simply checks if an item's value is equal to either `1`, `0`, `Y`, `N` `true`, `false`, `yes` or `no`.
3. The *Value-with-unit matcher* starts by determining if the unit is placed in front or at the back of the value. Next, we split the item into a numeric value and a unit part and trim each of the parts.
   The numeric part is passed along to the *Numeric-value matcher*.
   For the unit part, we differentiate depending on its placement. If the unit was positioned in front, it is highly likely that it is a currency and we will compare it to a list of currency symbols. If it was placed at the back, it is sufficient for the unit to have a string length less or equal to `5` to be compliant. There are

too many possible units to be able to verify this precisely without increasing runtime considerably.

4. The *Phone-number matcher* checks if an item only consists of numbers and a set of admissible symbols (`"+"`, `"("`, `")"`, `"/"`, `"-"`, `" "`). Furthermore the plus symbol (`"+"`) is only allowed to be placed in front and the last character in the item has to be a number. We also make sure that all occurring parentheses are opened and closed correctly.

5. The *Date matcher* validates numerical dates, e.g. `01.01.2018`. First up it checks if an item actually represents a valid date, this includes the consideration of leap years. For date delimiters we allow the use of the following symbols: `"."`, `"-"`, `"/"`. We support single or two-digit days and months as well as two-digit or four-digit years. Additionally both arrangements for days and months are accepted (`dd.mm` and `mm.dd`).

6. The *Timestamp matcher* detects different timestamp formats. The following formats are supported: `20180101T235959Z`, `2018-01-01T23:59:59+00:00`, `2018-01-01T23:59:59Z`, `2018-01-01T23:59:59.000Z`.

7. The *Email matcher* first checks if an item contains an `"@"` symbol, second we split off the second-level-domain (ex: `co.uk`) or top-level-domain (ex: `com`) and look them up in two corresponding lists.

8. The *URL matcher* starts by splitting the item in up to four sections: protocol, subdomain/domain name, second-level-domain/top-level-domain and path.
   For protocols we support `http`, `https`, `ftp`, `sftp` and `file`.
   The subdomains, domains and paths need to be alphanumeric with the exception of dots (`"."`) and hyphens (`"-"`), and additionally slashes (`"/"`) for paths.
   The second-level-domains (ex: `co.uk`) and top-level-domains (ex: `com`) are looked up in two corresponding lists.

9. The *JSON matcher* validates an item as a JSON string by verifying the placement of curly and square brackets. Furthermore, we check if every key and every value are wrapped in quotes.

10. The *XML matcher* validates an item as XML by checking that every tag is correctly opened and closed. It includes self-closing tags.

Checking all these properties for every item in the *item index* can quickly become time-consuming. Luckily, we can benefit from the fact that most of these pattern matchers verify item properties that are mutually exclusive. This means that only one of these matchers can be true. After one matcher turned out to be correct, there is no point in checking other matchers for the same item. For this reason the process' efficiency can be improved by predicting which matcher will be true. As the file is processed column by column and items from the same column are very likely to have the same properties, we can reduce the runtime considerably by rearranging

the execution order of the pattern matchers, such that the last matcher that was satisfied is moved to the front of the execution sequence.

In the list of matchers above, the *Numeric-value matcher* is the only non-exclusive matcher, as it matches booleans and indexes, thus this matcher always has to be checked. The overlap between the *Incremental-Index matcher* and *Boolean matcher* is negligible and can thus be considered to be mutually exclusive.

2.3.3.3. Column score calculation

In this step, we combine the boolean matcher-based property values we gathered for every item in the last section into column scores for each property. Moreover, we record values for the *word count* as well the *letter/digit ratio* and the character type scores using the boolean character type values from subsubsection 2.3.3.1.

Same as in subsection 2.3.2, n is the number of entries in the *item index* and the sums are calculated starting at $i = 1$ to exclude the *empty item*.
matcher designates any of the patterns matchers from subsubsection 2.3.3.2.

$$columnPropertyScore = \frac{\sum_{i=1}^{n} matcher(item_i) \cdot occurrence(item_i)}{\sum_{i=1}^{n} occurrence(item_i)} \tag{2.5}$$

To record the *word count* of the items, we calculate the word count mean $\overline{w}$ and standard word count deviation s_w weighted by the occurrences of the individual items in the column. The word count of an item is the number of spaces that it contains, augmented by 1.

$$\overline{w} = \frac{\sum_{i=1}^{n} (spacesCount(item_i) + 1) \cdot occurrence(item_i)}{\sum_{i=1}^{n} occurrence(item_i)} \tag{2.6}$$

$$s_w = \sqrt{\frac{\sum_{i=1}^{n} (spacesCount(item_i) + 1 - \overline{w})^2 \cdot occurrence(item_i)}{\frac{n-1}{n} \sum_{i=1}^{n} occurrence(item_i)}} \tag{2.7}$$

Next we calculate the character type occurrence scores for $k = \{letter, digit, symbol\}$. For the *symbol* score we will exclude spaces to improve its informative value, as spaces are already covered by the *word count*.

$$characterTypeScore = \frac{\sum_{i=1}^{n} containsCharType_k(item_i) \cdot occurrence(item_i)}{\sum_{i=1}^{n} occurrence(item_i)} \tag{2.8}$$

We also collect some information on the character type composition, by calculating the *letter/digit ratio*. Given n' the count of items containing digits, we can compute a mean $\overline{ld}$ and standard deviation s_{ld} for these items weighted by their occurrences in the column.

$$\overline{ld} = \frac{\sum_{i=1}^{n'} (\frac{letterCount(item_i)}{digitCount(item_i)} \cdot occurrence(item_i))}{\sum_{i=1}^{n'} occurrence(item_i)} \tag{2.9}$$

$$s_{ld} = \sqrt{\frac{\sum_{i=1}^{n'} (\frac{letterCount(item_i)}{digitCount(item_i)} - \overline{sl})^2 \cdot occurrence(item_i)}{\frac{n'-1}{n'} \sum_{i=1}^{n'} occurrence(item_i)}} \tag{2.10}$$

2.3.4. Subitem separator detection

This step is dedicated to detecting if an item is actually a list of subitems delimited by a *subitem separator*.
We begin by splitting every item by the separators it contains. For this we consider the same set of separators we used in section 2.1, where the *column separator* has been determined. The separation is done for every symbol in the separator set, even for the *column separator*, as it could be escaped.
To reduce the number of separations, we consider three rules that can invalidate a separator:

- The separator is the first or last character in an item
- Two separators occur directly next to each other
- The separator is followed by a space. In this case it is most likely that the symbol is part of a sentence.

As a result, we get a list of subitems for every valid separator in the separator set. To evaluate the different separators, we analyse these lists of subitems in the same way we evaluated the entire items in the column in subsection 2.3.1, subsection 2.3.2 and subsection 2.3.3. For the subitem characterisation, we remove the *Incremental*

index from our set of pattern matcher, as it is not compatible with partial items. Apart from the characterisation of the subitems, the subitem analysis also helps us to determine which separator is viable to be the item separator, as the subitems created by such a separator should show similar properties.

Even if a subitem separator has been detected, the features that have been collected for the complete items are still useful, as they help to eliminate faulty separators in classification step in chapter 3.
For instance we could have a column containing decimal numbers (ex.: `1,23`), which can be detected as two integers split by a comma as separator.
To prevent such errors, we record how often and in how many parts items have been split for every valid separator.

As in previous sections, n is the number of entries in the *item index* and the sums are calculated starting at $i = 1$ to exclude the *empty item* being $item_0$.

The *list occurrence* for a given *subitem separator* is captured as follows:

$$list\ occurrence = \frac{\sum_{i=1}^{n} \begin{cases} occurrence(item_i), & \text{if } subitem\ separator \text{ splits } item_i \\ 0, & \text{otherwise} \end{cases}}{\sum_{i=1}^{n} occurrence(item_i)} \tag{2.11}$$

The *subitem count* is the count of a given *subitem separator*, augmented by `1`. Given m the number of items that contain the *subitem separator*, we calculate the mean $\overline{sc}$ and the deviation s_{sc}.

$$\overline{sc} = \frac{\sum_{i=1}^{m} count(subitem\ separator, item_i) \cdot occurrence(item_i)}{\sum_{i=1}^{m} occurrence(item_i)} \tag{2.12}$$

$$s_{sc} = \sqrt{\frac{\sum_{i=1}^{m} (count(subitem\ separator, item_i) - \overline{sc})^2 \cdot occurrence(item_i)}{\frac{m-1}{m} \sum_{i=1}^{m} occurrence(item_i)}} \tag{2.13}$$

2.4. File property summary

In the final step of our analyser, we are arranging the collected data so that it is most favourable to our Naive Bayes classification algorithm in chapter 3.
Generally, a classifier's performance can be enhanced by removing noisy features. A simplified model is especially preferred when using smaller training sets [MRS08].
When using Naive Bayes another important factor is the independence of the features for a given class. The assumption of independent features is an essential property of the Naive Bayes algorithm and needs to be satisfied as far as possible.

2.4.1. Noisy Feature Elimination

In our case, the noisy features are those that we collected for all the different subitems in subsection 2.3.4. In most cases, we gathered subitem feature scores for multiple *subitem separators* per column. However, at best only one of these separators can be correct. This means that the other ones are inevitably invalid and should be discarded. We resolve this issue by identifying the most likely *subitem separator*.

When a column has only few items which are lists of subitems, we don't have enough data to make an appropriate evaluation. Therefore, we discard the *subitem separators* of a column for which the *list occurrence* is very low. The occurrence has to be greater than `10%` and at least `2`.

The remaining separators from each column are then evaluated by calculating a *subitem separator score* based on the scores that resulted from boolean properties of their subitems. These boolean properties are the different pattern matchers as well as the boolean character type indicators, that are gathered in the preprocessing and characterisation steps in subsubsection 2.3.3.1 and subsubsection 2.3.3.2.
In a best case scenario, the subitems of a given *subitem separator* should have the same properties. Hence, their property scores should optimally be `0` or `1`. The worst possible property score is `0.5`, as this would mean that half of the subitems satisfies the property and the other half doesn't, which represents lowest similarity in regards to this property.

Given m, the count of boolean-based property scores, the *subitem separator score* is calculated as follows:

$$separator\ score = \prod_{i=1}^{m} \begin{cases} propertyScore_i, & \text{if } propertyScore_i > 0.5 \\ 1 - propertyScore_i, & \text{otherwise} \end{cases} \tag{2.14}$$

We can now determine the most likely *subitem separator* by the highest *subitem separator score*.

This leaves us with a separator for each column. However, CompleteSearch only supports a single *subitem separator* for the entire file. We will deal with this problem in chapter 3, where we will unify the separator and decide if and in which columns the *subitem separator* is actually applied. For this reason, we also cache the data from all the other subitem separators, as the unified subitem might ultimately be different and this will save us from rerunning the Analyser to gather the data for a different *subitem separator*.

2.4.2. Feature Independence

The Naive Bayes classification algorithm, which we will be using requires feature independence for a given class. Thus, we face a feature dependence issue regarding the feature scores that originated from our mutually exclusive pattern matchers in subsubsection 2.3.3.2. We solve the problem by only retaining the predominate property. In this way, the exclusive matcher scores are reduced to two features: the type of the predominate property specified by its placement in the enumeration in subsubsection 2.3.3.2 and the score for this property.
This approach is backed by the results in section 3.6, when comparing the results for the default configuration in Table 3.2 in which the mutually exclusive properties were merged and the results in Table 3.4 where they were not.

2.4.3. Analyser Output

In Table 2.1, we collected all the different features that are returned by our Analyser for every column in the input file.
The discrete value for the *subitem separator* corresponds to its position in the separator set shown in section 2.1. The output is formatted as JSON and has the following structure:

```
{
 "col1": {
   "fillRate": 1,
   "item": {
     "uniqueness": 1,
     "length mean": 42,
           ⋮
   },
   "subitem": {
     "uniqueness": 1,
           ⋮
   }
 },
 "col2": {
   "fillRate": 1,
         ⋮
 },
     ⋮
}
```

Feature	**Value Type**	**Value Range**
fill rate	continuous	$[0, 1]$
Item Properties		
item uniqueness	continuous	$[0, 1]$
item length mean	continuous	$[0, +\infty[$
item length deviation	continuous	$[0, +\infty[$
item word count mean	continuous	$[0, +\infty[$
item word count deviation	continuous	$[0, +\infty[$
item numeric value	continuous	$[0, 1]$
item max integer places	continuous	$[0, +\infty[$
item max decimal places	continuous	$[0, +\infty[$
item min numeric value	continuous	$[0, +\infty[$
item max numeric value	continuous	$[0, +\infty[$
item exclusive property score	continuous	$[0, 1]$
item exclusive property type	discrete	$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
item letter occurrence	continuous	$[0, 1]$
item digit occurrence	continuous	$[0, 1]$
item symbol occurrence	continuous	$[0, 1]$
item letter/digit ratio mean	continuous	$[0, +\infty[$
item letter/digit ratio deviation	continuous	$[0, +\infty[$
Subitem Properties		
subitem separator	discrete	$\{0, 1, 2, 3, 4, 5, 6, 7\}$
list occurrence	continuous	$[0, 1]$
subitem count mean	continuous	$[0, +\infty[$
subitem count deviation	continuous	$[0, +\infty[$
subitem uniqueness	continuous	$[0, 1]$
subitem length mean	continuous	$[0, +\infty[$
subitem length deviation	continuous	$[0, +\infty[$
subitem word count mean	continuous	$[0, +\infty[$
subitem word count deviation	continuous	$[0, +\infty[$
subitem numeric value	continuous	$[0, 1]$
subitem max integer places	continuous	$[0, +\infty[$
subitem max decimal places	continuous	$[0, +\infty[$
subitem min numeric value	continuous	$[0, +\infty[$
subitem max numeric value	continuous	$[0, +\infty[$
subitem exclusive property score	continuous	$[0, 1]$
subitem exclusive property type	discrete	$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
subitem letter occurrence	continuous	$[0, 1]$
subitem digit occurrence	continuous	$[0, 1]$
subitem symbol occurrence	continuous	$[0, 1]$
subitem letter/digit ratio mean	continuous	$[0, +\infty[$
subitem letter/digit ratio deviation	continuous	$[0, +\infty[$

Table 2.1.: Analyser output content

2.5. Complexity

Generally, we can consider the runtime of the File Input Analyser to be linear to the input file size. As we evaluate every item in the file individually, we can approximate the analyser's complexity by $O(n \cdot m)$ with n being the row count and m being the column count of the input file.
In practice however, the runtime of the analyser is fairly dependent on the content of the input file.
On one side, the runtime can become shorter when the columns of the input file contain a lot of reoccurring items. Repeating items decrease the size of our *item index*, which constitutes the base for all following operations.
On the other side, the runtime can become a lot worse, when the analyser detects a lot of potential *subitem separators*, because it implicates the preprocessing and characterisation of a high number of subitems.
Furthermore, the analyser offers the option to provide a sampling step, which will reduce the number of records that will be processed. This offers the ability to considerably reduce the runtime for large input files and can be useful when speed is more valuable than the highest feature score precision.

3. Column Classification

In this chapter, we carry on from properties we collected from the dataset to finding suitable parameters for configuring CompleteSearch.
The problem to solve is essentially a classification problem, in which each column of the input file is assigned to the different CompleteSearch parameter classes. For this purpose we built a classifier, which can be used as follows:

```
ClassifierMain [mode] [parameters]
```

Available modes:
`--classify <inputFile>`
Classifies a given dataset into the different parameter classes. The input file is not the actually dataset but the JSON output file containing its structural features returned by the Analyser in chapter 2
`--train`
Trains the classifier by performing all steps that can be computed in advance and saving the training data. For further details see section 3.3.
`--benchmark <configuration>`
Splits off a part of the training set into a test set, trains the classifier on the reduced training set and evaluates the classification results of the test set. More details and benchmarking results in section 3.6. Possible configurations: `default`, `no-augmentation`, `no-prop-merge`, `no-sep-predetermination`

Parameters:
`--props <datasetPropDirectory>`
Path to directory containing dataset property files for the input datasets in our training and test sets. This parameter is required for training and benchmarking
`--labels <datasetLabelDirectory>`
Path to directory containing dataset label files for the input datasets in our training and test sets. This parameter is required for training and benchmarking
`--cache <trainingDataCacheDirectory>`
Path to directory containing/storing training data. This parameter is required for training and classification.

To solve the classification problem, we choose the popular Naive Bayes algorithm for a few different reasons:

- It is well-suited for multi-class classification, which is required for parameters that can accept more than two different values, such as *field-format* and *ordering* mentioned in section 1.2.
- It is fast to compute. The complexity is linear to the size of the training set and the number of features. This is important as we are considering to continuously retrain our classifier using the user feedback from our web application as explained in section 4.2.
- It outperforms most other classification algorithms, when dealing with smaller training sets. [DP97][Col09]

However, before we can proceed to the column parameter classification, we need to determine where to use the full *item* or *subitem* features for classifying the columns to the different parameter classes.

As we also make use of the Naive Bayes algorithm to determine the *subitem separator*, we will start this chapter by explaining the algorithm. Next, we take a look at the constitution of our training set and the training of the classifier, before we get to the actual classification process.

3.1. Naive Bayes

Naive Bayes is a supervised learning algorithm that uses conditional probabilities to assign a problem instance to a finite set of classes $\mathbb{C} = \{c_1, c_2, ..., c_k\}$. The problem is represented by an array $\boldsymbol{x} = (x_1, x_2, ..., x_n)$, where $x_1, ..., x_n$ are its different attributes.

The probability of a problem being in class $c \in \mathbb{C}$ is described by conditional probability $p(c \mid x_1, ..., x_n)$.
This expression can be decomposed using Bayes' theorem and reformulated using the product rule for joint probabilities [RN16]:

$$p(c \mid x_1, ..., x_n) = p(c, \boldsymbol{x}) = \frac{p(c)\ p(\boldsymbol{x} \mid c)}{p(\boldsymbol{x})} \tag{3.1}$$

$$= \frac{p(c, \boldsymbol{x})}{p(\boldsymbol{x})} \tag{3.2}$$

$$= \frac{p(c)}{p(\boldsymbol{x})} \prod_{i=1}^{n} p(x_i \mid x_{i-1}, ..., x_1, c) \tag{3.3}$$

The algorithm is called "naive", because it makes the assumption that the features are independent from each other given the class [RN16]. This allows to considerably

reduce the number of probabilities that we need to compute. This simplification results in the following expression:

$$p(c \mid \boldsymbol{x}) = \frac{p(c)}{p(\boldsymbol{x})} \prod_{i=1}^{n} p(x_i \mid c) \tag{3.4}$$

$p(\boldsymbol{x})$ is a normalisation constant. It makes sure that the probabilities over the classes add up to 1 for each feature. It can be calculated as follows:

$$p(\boldsymbol{x}) = \sum_{i=1}^{k} p(c_k)\ p(\boldsymbol{x} \mid c_k) \tag{3.5}$$

As we are comparing the different probabilities to each other to determine the best class for $\boldsymbol{x}$, computing $p(\boldsymbol{x})$ is optional.

To avoid a floating point underflow in the probability multiplication in equation 3.4, we can perform a logarithmic transformation [MRS08]:

$$p_{log}(c \mid \boldsymbol{x}) = log(p(c)) - log(p(\boldsymbol{x})) + \sum_{i=1}^{n} log(p(x_i \mid c)) \tag{3.6}$$

The various probabilities that are required can be computed/estimated as follows:

- **Class Probability $p(c)$ with $c \in \mathbb{C}$**

$$p(c) = \frac{\text{occurrences of class } c \text{ in training set}}{\text{entries in training set}} \tag{3.7}$$

- **Conditional Attribute Probability $p(x_i|c)$ with $c \in \mathbb{C}$ and $i < n$**
  For the conditional probabilities, we need to distinguish between different data types. The type of each attribute is indicated in Table 2.1. We differentiate between discrete and continuous data attributes:

  ◦ The probabilities for discrete attributes can be calculated explicitly.
  Let m be the number of entries in class c and n_{vc} the number of occurrences of the discrete value v for the attribute x_i in class c.

$$p(x_i = v \mid c) = \frac{n_{vc}}{m} \tag{3.8}$$

  ◦ The probabilities for continuous attributes are estimated using the Gaussian distribution. For this, we first need to compute the mean μ_i and the variance σ_i^2 for attribute x_i in class c to calculate the probability $p(x_i \mid c)$ for

attribute value v. Let m be the number of entries in class c and $w_{i,j}$ the value of attribute x_i in entry j in class c.

$$\mu_{ic} = \frac{\sum_{j=1}^{m} w_{i,j}}{m} \tag{3.9}$$

$$\sigma_{ic}^2 = \frac{\sum_{j=1}^{m} (w_{i,j} - \mu_i)^2}{m} \tag{3.10}$$

$$p(x_i = v \mid c) = e^{-\frac{(v-\mu_{ic})^2}{2\sigma_{ic}^2}} \tag{3.11}$$

3.2. Data collection

In this section, we build the training set for our classifier, which will be used to compute the different class probabilities and conditional attribute probabilities, that we need for the Naive Bayes classifier.
Usually, a classifier's performance improves with the size of its training set. Thus, it is favourable to collect as much data as possible. For this thesis, we collected 50 different input datasets from various sources, which are all listed in Appendix A.
When searching for suitable input datasets, we were mainly looking for datasets with different data formats and with as many columns as possible, as these are the two primary factors that improve the quality of our training set.

First, we manually labelled every column in the datasets with the different parameter classes that apply to that column, including a common *subitem separator*.
Next, we parsed the collected input datasets using our Analyser from chapter 2 to gather their properties. The training set is formed by combining the properties and the class labels. We use *subitem separator* values from the label files to use the correct subitem features from the properties files. Thus, our training set contains a record for every column in the collected datasets.

To further improve the classification results for less common subitem separators, we augment our training set by replicating entries of input datasets that have a *subitem separator* for every other separator in our separator set from section 2.1 that doesn't occur in the file.
The benefits of this training set augmentation can be seen in section 3.6 by comparing Table 3.2 and Table 3.3.

3.3. Training

In the training stage, we perform all steps that can be computed in advance, as they only need to be executed if the training set has changed.

In our case, we can precompute the class probabilities defined in Equation 3.7 and the probabilities for discrete property attributes in Equation 3.8. It is not possible to do the same for the probabilities for continuous property attributes in Equation 3.11 as they depend on the input dataset property value. For this reason, we are limited to computing mean-deviation pairs for the different attributes in each class.

3.4. Classification

The classification process of an input file is made up of several different steps, the sequence of which is shown in Figure 3.1.

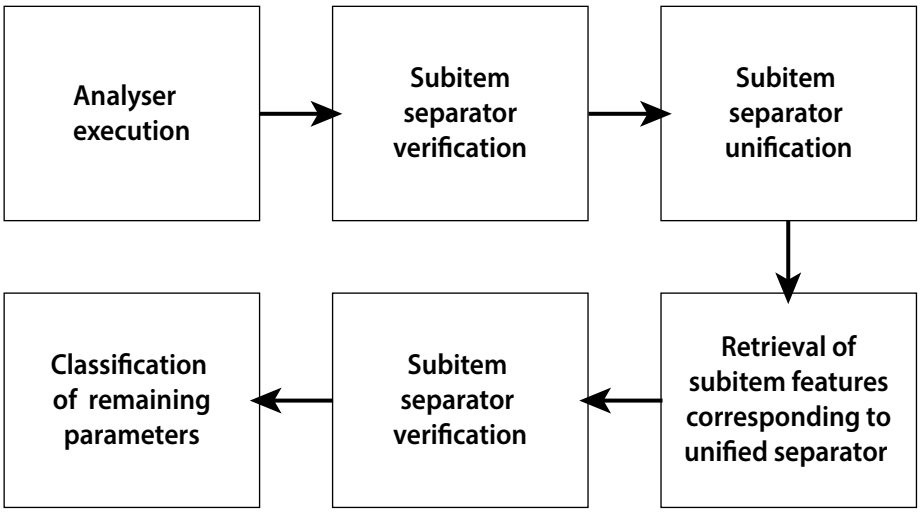

Figure 3.1.: Classification process sequence

First we start by determining which columns have a *subitem separator*. Afterwards, we seek to conclude on a common separator for all columns in the input dataset. This will finally allow us to choose the right property attributes to assign the columns of the input file to the matching CompleteSearch parameters values. We will explain the procedure in more detail in the following subsections.

3.4.1. Subitem separator verification

After gathering the input file properties using our Analyser from chapter 2, we perform a binary classification to determine if the subitem separators that have been determined for each column are valid or not. Thus, we assign the columns of our input dataset to two classes, in which the subitem separator is either valid or not,

which corresponds to the *allow-multiple-items* parameter vector for CompleteSearch. For this computation we use all the attributes gathered by the analyser but *item max integer places* and *item max decimal places* as well as the *item min numeric value* and *item max numeric value* which are only intended to be used to order numeric values in CompleteSearch and are therefore irrelevant for classification.

3.4.2. Subitem separator unification

Based on the results of the previous *subitem separator* verification step, which indicates which columns are likely to have a *subitem separator*, we will conclude to a common separator for all the columns in the input file. We simply take the separator that occurs most often in the columns that assumedly allow multiple items. In case of a tie, we consider the *allow-subitems* probabilities of these columns and calculate a joined probability for each separator that is involved in the tie.
Without this preliminary subitem separator determination, the parameter classification results would look like those indicated in Table 3.5.

After determining the common separator, we can retrieve the property data for the unified separator for each column from the dataset properties file. Furthermore, we rerun the subitem separator verification step, to update the results for the common subitem separator, which gives a final answer to which columns allow subitems and which do not.

3.4.3. Parameter classification

As we finally have a common *subitem separator* for the entire file and know in which columns it is applicable, we can reduce our training set to the relevant features. We merge overlapping item/subitem features by picking the feature values computed for the entire item or the values for the subitems according to the results from the subitem separator verification with the common separator. This reduction benefits the classifier's performance as fewer attributes generally reduce noise. [MRS08]
As the number of attributes needs to be the same for all columns, independent of having a *subitem separator* or not, we have extraneous features for records without separator, such as *list occurrence*, *subitem count mean* and *subitem count deviation*. The values for these three properties are set to `0`.

Now, we can go ahead to classifying the remaining column parameters to their respective values, as shown in Table 3.1.

Parameter	**Classes**
full-text	true, false
allow-subitems	true, false
filter	true, false
facet	true, false
field-format	plain-text, JSON, XML
show	true, false
excerpt	true, false
ordering	lexicographical, numerical, date
url	true, false
email	true, false
label	true, false

Table 3.1.: Column parameters and their classes

3.5. Complexity

We cover the complexity for training and classification separately.

3.5.1. Training

Let n = total number of records in the input datasets, m = number of records in the training set, a = number of discrete features, b = number of continuous features, c = number of classes and p = number of parameters that need to be classified.

Running the Analyser on all the input datasets can be approximated by $O(n \cdot m)$ as mentioned in section 2.5.

The complexity for training the classifier for a given label can be described by $O(m)$. The step is made up of 3 different operations:

- The class probabilities have a complexity of $O(c)$ and can be neglected.
- The discrete attribute probabilities are computed for each feature value in each class, thus the complexity is $O(a \cdot m) = O(m)$

- For the continuous attributes, we only compute the mean and deviation pairs for each class and the complexity is only $O(b \cdot c)$. As b and c are constants, it is negligible.

As we need to perform the training process for every CompleteSearch parameter, the complexity of the training step results in $O(n \cdot m + p \cdot m)$ where p is constant. Thus, we have $O(n \cdot m)$.

3.5.2. Classification

The complexity of assigning an array of property attributes to a class is $O(c \cdot (a+b))$ as we need to compute/retrieve $(a + b)$ probabilities for each class c.

Let x = number of columns and y = number of records in the input dataset whose parameters should be classified.
The classification step consists of the following operations:

- Running the Analyser for the given input file has a complexity of $O(x \cdot y)$
- Classifying *allow-subitems* for each column in the input file, which has a complexity of $O(x \cdot c \cdot (a + b))$
- The separator unification's complexity is insignificant.
- Executing the *allow-subitems* classification again costs $O(x \cdot c \cdot (a + b))$
- The classification of the remaining parameters is $O((p - 2) \cdot (x \cdot c \cdot (a + b)) = O(x \cdot y)$

Thus, performing the classification step for a given input file has a complexity of $O(x \cdot y + p \cdot (x \cdot c \cdot (a + b)) = O(x \cdot y)$, as p, c, a, b are constants.

3.6. Evaluation

To evaluate the performance of our CompleteSearch parameter classifier, we split off a part of the training set into a test set. For the following benchmarks, we picked at random 20% of the input datasets we collected in section 3.2 to constitute the test set.

Below we present the accuracy scores for different configurations of our Analyser and Classifier.
The default configuration in subsection 3.6.1 shows the results for the procedure described throughout this thesis and we expect it to be superior to the other configurations listed below. We computed the other configuration for comparative reasons and to discuss the different steps we chose while building our Analyser/Classifier in subsection 3.6.5.

3.6.1. Default configuration

Parameter	accuracy
subitem-separator	0.833333
allow-subitems	0.993671
full-text	0.858650
filter	0.873418
facet	0.734177
field-format	1.000000
show	0.725738
excerpt	0.951477
ordering	0.970464
url	0.983122
email	0.989451
label	0.736287

Table 3.2.: Accuracy for default configuration

3.6.2. Default configuration without augmented training set

Parameter	accuracy
subitem-separator	1.000000
allow-subitems	0.988971
full-text	0.805147
filter	0.801471
facet	0.750000
field-format	1.000000
show	0.753676
excerpt	0.926471
ordering	0.966912
url	0.974265
email	0.977941
label	0.628676

Table 3.3.: Accuracy for default configuration without augmented training set

3.6.3. Configuration without merging mutually exclusive properties

Parameter	accuracy
subitem-separator	0.611111
allow-subitems	0.947257
full-text	0.839662
filter	0.841772
facet	0.668776
field-format	0.938819
show	0.710970
excerpt	0.843882
ordering	0.974684
url	0.955696
email	1.000000
label	0.740506

Table 3.4.: Accuracy for configuration without merging mutually exclusive properties

3.6.4. Configuration without separator predetermination

Parameter	accuracy
subitem-separator	0.350211
allow-subitems	0.983122
full-text	0.858650
filter	0.888186
facet	0.706751
field-format	1.000000
show	0.713080
excerpt	0.955696
ordering	0.968354
url	0.997890
email	1.000000
label	0.765823

Table 3.5.: Accuracy for configuration without separator predetermination

3.6.5. Discussion

We want to note that for some of the parameters exist multiple acceptable values. The choice of value for these parameters involves some degree of user preference. For this reason, the accuracy might be less conclusive, as some values might not correspond to our choices made in the labelling process, but might be acceptable nevertheless. The parameters in question are *full-text*, *filter*, *facet*, *show*, *excerpt* and *label*.

Considering the small size of our training set, containing only 87 datasets after the augmentation step, the performance of our classifier is fairly good with the scores of preference-independent parameters seen in Table 3.2 reaching accuracies of 83% or more. The lowest score corresponding to *subitem-separator*, which has a smaller training set as it is the same for all the columns in a dataset. The *show* and *facet* parameter predictions yield the lowest scores with accuracies of 72% and 73%, which makes sense as they are the parameters where the value choice is the least obvious.

Looking at the accuracy results without the training set augmentation in Table 3.3. We see that the augmentation, explained in section 3.2, generally improves our results with the exceptions of *show*, *facet* and the *subitem-separator*. These scores decreased by between roughly 2% to 17%, which seems to be due to our small training set in the first place, for which the augmentation can substantially amplify the impact of faulty classifications. This especially applies to *subitem-separator*.

In Table 3.4, we can observe how the usage of dependent features in the Naive Bayes classifier influences our results. It is interesting to see that it only has strong impact on a few scores, while others remain mostly unchanged. The parameters *subitem-separator*, *excerpt*, *field-format* and *facet* stand out the most with accuracy losses between 7% and 22%.

Last, we consider Table 3.5 containing the scores, when omitting the entire separator unification and features reduction steps by simply passing the output of the Analyser into the Naive Bayes classifier. First, we notice the atrocious *subitem-separator* score, which isn't surprising as the separators don't have any relation to the data they are delimiting.
However, the fact that the scores of the other parameters are on par with our default configuration is rather unexpected. For *filter* the score is even slightly superior. This means that the noise introduced from extraneous item or subitem features in Table 3.5 and the feature reduction made for Table 3.2 based on the *allow-subitem* parameter prediction, which has an accuracy of more than 99%, result in a similar fault rate.

4. Web Application

In this final chapter, we will illustrate how our work from the previous chapters ties into the CompleteSearch web application. We show how the different parameter values change the user interface and behaviour of the search engine. Furthermore, we will discuss how we can use the user feedback from the application to improve its automatic configuration through continuous use.

4.1. Configuration effects

In Figure 4.1 we annotated the different sections of the search user interface, that change based on the various parameters that we determined throughout this thesis.

1. *Filters* are displayed as tabs next to the `"All"` columns tab and allow to restrict the search query to a particular dataset attribute and thus limit the search to that specific column in the input dataset.
2. Each search result represents a record from the input dataset. The ordering of the attributes within a search result is determined by their *uniqueness* that was calculated in chapter 2.
3. *URL*s and *Email* addresses are converted into functional links.
4. A *label* shows the column name of an attribute in front of its value. This helps to give context to values that are less descriptive on their own, such as numeric values.
5. *Excerpts* show extracts from longer record items. The settings allow to adjust the maximum number of excerpts per search result and lengths of each excerpt by setting a maximal word radius.
6. *Facets* are shown in the right sidebar. The control interface of the different facets depends on their ordering and thus their datatype:

    a. The default facet input is a list of the most common facet values that match the current search query.

    b. The numerical facet input uses the *min numeric value* and *max numeric value* as well as the *max integer places* and *max decimal places* of the facet column to create a slider that allows to define a numerical search range.

    c. The date facet input allows to define a date range via two date pickers.

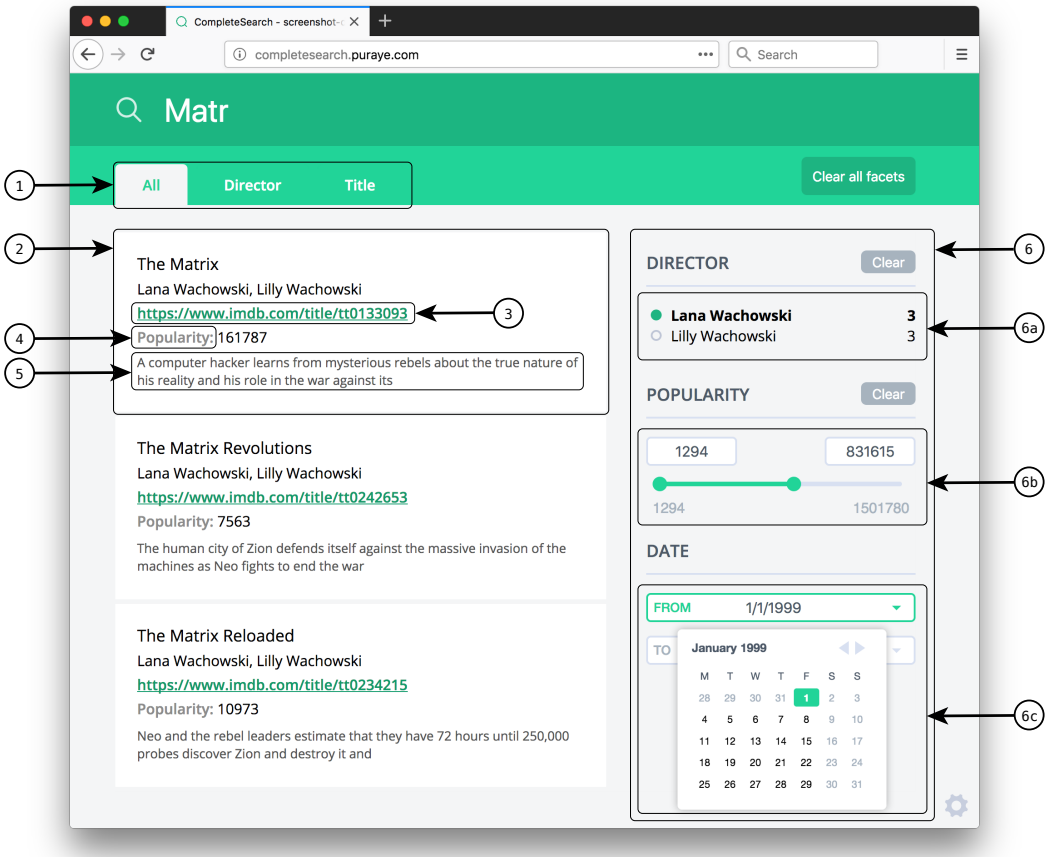

Figure 4.1.: Search User Interface (1. Filters, 2. Search result, 3. Link, 4. Label, 5. Excerpts, 6. Facets: a. Lexicographical facet, b. Numerical facet, c. Date facet)

4.2. User Feedback Loop

Our web application allows the user to manually adjust the automatically suggested configuration parameters of CompleteSearch via the settings interface shown in Figure 4.2. We will use this user feedback to automatically improve our classifier.

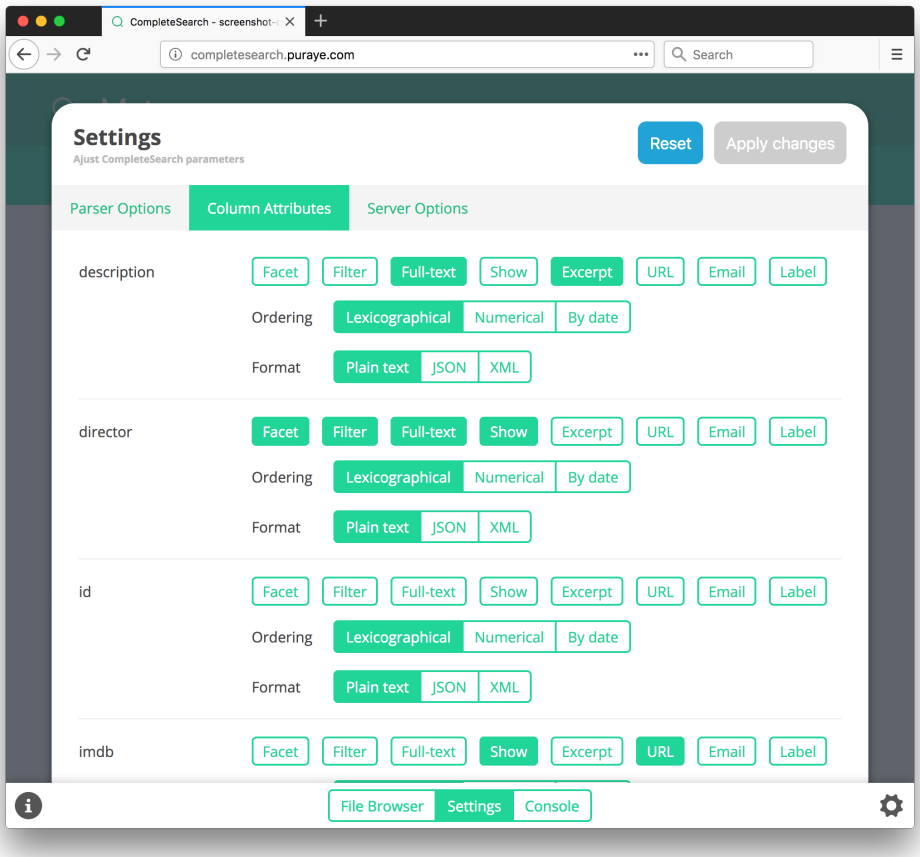

Figure 4.2.: Settings interface: Column Attributes

The user-adjusted configuration essentially contains the labels for the given input dataset and can thus be integrated into the training set and allows us to retrain our classifier on the side.

5. Conclusion

Throughout this thesis, we explored a procedure that allows us to automatically configure CompleteSearch for any given tabular dataset. For this, we have been mostly focused on the extraction of the features from the datasets and setting up a simple and fast classification method. Furthermore, we took the user feedback from the web application into consideration to improve the feature distribution model of the classifier over time.

The results from section 3.6 show that we largely reached our objective, however the current implementation could still benefit from the following improvements:

- The most obvious improvement is to add a lot more datasets to the initial training set, although this point is partly taken care of by the usage of the user feedback from the web application.
- We could extract more features from the datasets e.g. by adding mutually exclusive matchers to detect more data types and formats, such as coordinates, file paths, fractions, scientific notations, more date formats, etc. Some of the these could be accompanied by further search interface customisation. For instance for coordinates, we could have a map with an origin pin and an adjustable radius.
- We could take the column names in the input datasets into account. However, this would require a lot of new training data for associating dataset features to its column names. It would also implicate further challenges such as the support of multiple languages.
- We could try other classification algorithms, e.g. Multi-class Support Vector Machine and tweak their parameters to further improve the classification accuracy.

For the web application itself, we could consider the following improvements:

- To avoid adding faulty data to our training set, when using the user feedback, we could measure a configuration's trustworthiness by the number of queries that have been made using it. When retraining our classifier, we could weight the new configurations by this metric. The entries of the initial training set are given for instance a weight of `100 search queries`. As we implemented search-as-you-type functionality, we could consider a new search to

start `5 seconds` after the last key was pressed. Furthermore, there could be a minimum threshold of `10 queries` that needs to be surpassed, in order for a configuration to be considered at all.

- We could analyse the input dataset client side. This could be done while uploading it to the server and would be most effective for large datasets. This could be easily realised by compiling our analyser written in C++ into a WebAssembly [Web] module.

Acknowledgments

First, I want to thank Prof. Dr. Hannah Bast for giving me the opportunity to write this thesis extending my work on the CompleteSearch web application beyond my Bachelor Project.

I also want to thank my friend and business partner Frank Gelhausen for proofreading this document and enduring my lack of time for working on other projects.

A. Datasets

Listed below are all the datasets that have been used throughout this thesis. [accessed 02-07-18]

- Deutsche Bahn - Reisezentrenliste (Stand: 03/2017)
  http://download-data.deutschebahn.com/static/datasets/reisezentren/VSRz201703.csv
- Deutsche Bahn - Betriebsstellen Güterverkehr
  http://download-data.deutschebahn.com/static/datasets/betriebsstellen_cargo/GEO_Bahnstellen_EXPORT.csv
- Deutsche Bahn - Stationsdaten
  http://download-data.deutschebahn.com/static/datasets/stationsdaten/DBSuS-Uebersicht_Bahnhoefe-Stand2016-07.csv
- Deutsche Bahn - Terminologie (DE/EN)
  http://download-data.deutschebahn.com/static/datasets/sprachenmanagement/Terminologie_DBKonzern_DE_EN_Definition.csv
- Deutsche Bahn - Aufzuege (Stand: 10/2015)
  http://download-data.deutschebahn.com/static/datasets/aufzug/DBSuS-Uebersicht_Aufzuege-Stand2015-10.csv
- U.S. Department of Agriculture's PLANTS Database
  https://www.plants.usda.gov/java/downloadData?fileName=plantlst.txt&static=true
- HIFLD Open - Fortune 500 Corporate Headquarters
  https://hifld-dhs-gii.opendata.arcgis.com/datasets/e277657582f74ed78dc2a503eae7fa2e_0
- HIFLD Open - Colleges and Universities
  https://hifld-dhs-gii.opendata.arcgis.com/datasets/4061dcd767c340d4a42fb7a0c6c5d5b4_0
- HIFLD Open - Ports Of Entry
  https://hifld-dhs-gii.opendata.arcgis.com/datasets/9ea04e9e2dd6465689a01eea5f3652fe_0
- HIFLD Open - Cities and Towns NTAD
  https://hifld-dhs-gii.opendata.arcgis.com/datasets/c246aa3bef7049dd9eeb86ae699572c9_0

- HIFLD Open - All Places of Worship
  https://hifld-dhs-gii.opendata.arcgis.com/datasets/ece7900854a443c28e1351a2eb3d7e7c_0?uiTab=table
- HIFLD Open - Volcanic Eruptions
  https://hifld-dhs-gii.opendata.arcgis.com/datasets/70fbc779b62249548f2352cf563105fd_6
- HIFLD Open - Emergency Operations Centers
  https://hifld-dhs-gii.opendata.arcgis.com/datasets/db3cb0002e664b3e8b64f92dd8510365_0
- Openflights.org - Airports
  https://raw.githubusercontent.com/jpatokal/openflights/master/data/airports.dat
- ATP Tennis Matches
  https://raw.githubusercontent.com/JeffSackmann/tennis_atp/master/atp_matches_qual_chall_2017.csv
- Harvard Dataverse - Week of Global News Feeds August 2017
  https://dataverse.harvard.edu/file.xhtml?fileId=3123815&version=RELEASED&version=.0#
- IMDb - Titles
  https://datasets.imdbws.com/title.basics.tsv.gz
- IMDb - Names
  https://datasets.imdbws.com/name.basics.tsv.gz
- BuzzFeed News - NBA Owners
  https://gist.github.com/jtemplon/4d84d0d2a112d09394b6#file-nba_owners_data-csv
- BuzzFeed News - Top Fake News on Facebook 2017
  https://github.com/BuzzFeedNews/2017-12-fake-news-top-50/blob/master/data/top_2017.csv
- BuzzFeed News - Trump Twitter Wars - Tweets
  https://github.com/BuzzFeedNews/2018-01-trump-twitter-wars/blob/master/data/tweets/tweets1.csv
- BuzzFeed News - Trump Twitter Wars - Accounts
  https://github.com/BuzzFeedNews/2018-01-trump-twitter-wars/blob/master/data/accounts.csv
- DataPortals.org
  https://raw.githubusercontent.com/okfn/dataportals.org/master/data/portals.csv
- CORGIS Dataset Project - Music
  https://think.cs.vt.edu/corgis/csv/music/music.csv?forcedownload=1

- CORGIS Dataset Project - Billionaires
  https://think.cs.vt.edu/corgis/csv/billionaires/billionaires.csv?forcedownload=1
- CORGIS Dataset Project - Airlines
  https://think.cs.vt.edu/corgis/csv/airlines/airlines.csv?forcedownload=1
- CORGIS Dataset Project - Cars
  https://think.cs.vt.edu/corgis/csv/cars/cars.csv?forcedownload=1
- CORGIS Dataset Project - Books - Gutenberg Project
  https://think.cs.vt.edu/corgis/csv/classics/classics.csv?forcedownload=1
- CORGIS Dataset Project - Earthquakes
  https://think.cs.vt.edu/corgis/csv/earthquakes/earthquakes.csv?forcedownload=1
- CORGIS Dataset Project - Food
  https://think.cs.vt.edu/corgis/csv/food/food.csv?forcedownload=1
- CORGIS Dataset Project - Hospitals
  https://think.cs.vt.edu/corgis/csv/hospitals/hospitals.csv?forcedownload=1
- CORGIS Dataset Project - Publishers
  https://think.cs.vt.edu/corgis/csv/publishers/publishers.csv?forcedownload=1
- CORGIS Dataset Project - Real Estate
  https://think.cs.vt.edu/corgis/csv/real_estate/real_estate.csv?forcedownload=1
- CORGIS Dataset Project - Skyscrapers
  https://think.cs.vt.edu/corgis/csv/skyscrapers/skyscrapers.csv?forcedownload=1
- CORGIS Dataset Project - Supreme Court
  https://think.cs.vt.edu/corgis/csv/supreme_court/supreme_court.csv?forcedownload=1
- CORGIS Dataset Project - Tate
  https://think.cs.vt.edu/corgis/csv/tate/tate.csv?forcedownload=1
- Kaggle - Yelp - Businesses
  https://www.kaggle.com/yelp-dataset/yelp-dataset/downloads/yelp_business.csv
- Kaggle - Kickstarter Projects
  https://www.kaggle.com/kemical/kickstarter-projects/downloads/ks-projects-201612.csv

- Kaggle - SpaceX Launch Data
  https://www.kaggle.com/scoleman/spacex-launch-data/downloads/spacex_launch_data.csv
- Kaggle - US Jobs Monster.com
  https://www.kaggle.com/PromptCloudHQ/us-jobs-on-monstercom/downloads/monster_com-job_sample.csv
- Kaggle - Celebrity Deaths
  https://www.kaggle.com/hugodarwood/celebrity-deaths/downloads/celebrity_deaths_4.csv
- Kaggle - 1000 Netflix Shows
  https://www.kaggle.com/chasewillden/netflix-shows/downloads/1000-netflix-shows.zip/1
- Kaggle - Board Games
  https://www.kaggle.com/mrpantherson/board-game-data/downloads/board-game-data.zip/5
- Kaggle - Astronauts
  https://www.kaggle.com/nasa/astronaut-yearbook/downloads/astronauts.csv
- Kaggle - English Premier League Players
  https://www.kaggle.com/mauryashubham/english-premier-league-players-dataset/downloads/epldata_final.csv
- Kaggle - Restaurants on TripAdvisor
  https://www.kaggle.com/PromptCloudHQ/restaurants-on-tripadvisor/downloads/tripadvisor_in-restaurant_sample.csv
- Kaggle - Google Job Skills
  https://www.kaggle.com/niyamatalmass/google-job-skills/downloads/job_skills.csv
- Kaggle - Olympic Sports and Medals 1896-2014 - Summer
  https://www.kaggle.com/the-guardian/olympic-games/downloads/summer.csv
- Kaggle - Rolling Stone's 500 Greatest Albums of All Time
  https://www.kaggle.com/notgibs/500-greatest-albums-of-all-time-rolling-stone/downloads/albumlist.csv
- Kaggle - The Movies Dataset - Metadata
  https://www.kaggle.com/rounakbanik/the-movies-dataset/downloads/movies_metadata.csv
- Kaggle - Periodic Table of the Elements
  https://www.kaggle.com/jwaitze/tablesoftheelements/downloads/periodic_table.csv

Bibliography

[Bas] Bast, H.: *CompleteSearch*. Internet: https://ad-wiki.informatik.uni-freiburg.de/completesearch/. – [Online; accessed 14-February-2018]

[Col09] Colas, Fabrice Pierre R.: *Data mining scenarios for the discovery of subtypes and the comparison of algorithms*, Leiden Institute of Advanced Computer Science, Diss., 2009

[DI08] Dakka, Wisam ; Ipeirotis, Panagiotis G.: Automatic Extraction of Useful Facet Hierarchies from Text Databases. In: *2008 IEEE 24th International Conference on Data Engineering* (2008), p. 466–475

[DP97] Domingos, P. ; Pazzani, M.: On the Optimality of the Simple Bayesian Classifier under Zero-One Loss. In: *Machine Learning* 29 (1997), p. 103 – 130

[Fre] Free Software Foundation: *Open Semantic Search*. Internet: https://www.opensemanticsearch.org/doc/search/csv. – [Online; accessed 03-May-2018]

[MRS08] Manning, Christopher D. ; Raghavan, Prabhakar ; Schütze, Hinrich: *Introduction to Information Retrieval*. Cambridge University Press, 2008

[Pur17] Puraye, O.: *Bachelor Project: CompleteSearch UI*. Internet: http://completesearch-docs.puraye.com. 2017. – [Online; accessed 14-February-2018]

[RN16] Russell, S. ; Norvig, P.: *Artificial Intelligence: A Modern Approach*. 3. Pearson Education Limited, 2016

[SM] Sean Massung, Chase G.: *Meta-Toolkit*. Internet: https://meta-toolkit.org. – [Online; accessed 03-May-2018]

[Web] WebAssembly Working Group: *WebAssembly*. Internet: https://webassembly.org. – [Online; accessed 02-March-2018]

[Wil] Willison, Simon: *Datasette*. Internet: https://github.com/simonw/datasette/blob/master/docs/index.rst. – [Online; accessed 04-May-2018]