Feature Extraction of Unstructured Cocktail Recipes

Sigurd Sippel

Hamburg University of Applied Sciences, Department of Computer Science, Berliner Tor 7, 20099 Hamburg

sigurd.sippel@haw-hamburg.de

July 4, 2015

1 Introduction

A recommender system of cocktail recipes helps a bartender to find recipes that are out of his mind. Distance functions are the core of a recommender system and are described in [Sip15]. The distance functions need extracted features.

Good quality features are fundamental for a precise recommendation. A recommendation needs a large number of recipes to ensure that a wide range of recommendable recipes are available. Therefore, a manual extraction is not useful. This paper considers the feature extraction of unstructured cocktail recipes, such as from books or blogs.

The section 2 considers the architecture improvements and the used libraries. The changes in target structure are described in section 3. The features extraction of section 4 is separated into several phases. The preprocessing and normalization transform the text into raw tokens. The number recognition contains rules to recognize numbers. The entity recognition uses the ontology to find known items. The logic combining eliminates negations. The context analysis finds ingredient declarations. The feature reasoning uses ontology to find missing features. The validation metric decides whether a recipe is plausible or not. Section 5 contains an experiment of parsing books. It includes recipe recognition and the parsing results. The section 6 deals with the conclusion and future work.

2 Architecture

In the first approach ([Sip15]), the architecture is designed to use pre-extracted recipes, which are stored in the XML format. The features were identified by ontology and enriched with additional information, such as the super category. The recommendation module used these extracted features to calculate the distance function.

Now, the pre-extraction is replaced with an automatic feature extraction. In the input data is a collection of unstructured recipes. The feature extraction is a expensive process, which has to work once for one recipe. It is an independent process that extracts the features and stores the retrieved data in an extended XML format. The data structure is described in the next section.

The featured extraction contains several phases that are used to identify tokens and refine them. The output comprises a collection of validated recipes. These recipes are prepared for the recommendation, which uses one recipe of the collection as an example to find other recipes from the collection that are classified as recommendations.

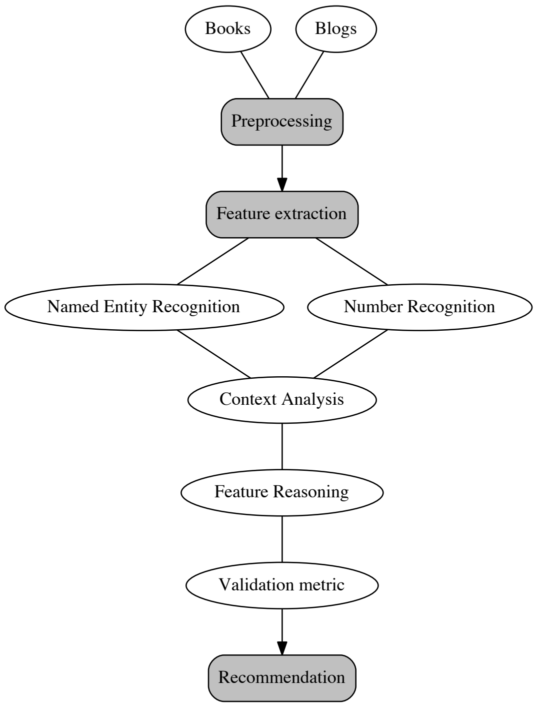

Figure 1: Architecture

The ontology is used to identify the named entities. An indexing is necessary to find named entities or part of that with high performance. Apache

Jena 2.13.0[1] is used to call SPARQL queries on the RDF ontology. The module Apache Jena Text 1.1.2[2] is used to integrate Apache Lucene 4.6.1[3] as an indexer in a SPARQL query. Apache Lucene replaces banana-rdf, because no Lucene integration is possible. Scala parser combinators 1.0.3[4] are used for context analysis. The library itext 5.06[5] is used to transform the PDF format to plain text.

3 Target Structure

The target structure describes one cocktail recipe. The target structure in [Sip15, p. 2] contained the extracted features. They only have to be identified with their in the ontology. It contained a list of ingredient declarations with the quantities, a unit, and one ingredient name. Further, it contained exactly one preparation and one glassware. This target structure represents the ideal cocktail recipe.

The chosen sources, such as cocktail books, contain recipes as raw text, which do not always contain a preparation or glassware. An ingredient declaration could be optional or could contain alternatives. The quantity could be a range, such as *one or two dashes*.

The extracted features represent the internal representation (Equation 1). It is a technical presentation, which is necessary for the recommendation.

$$trait\, Item\{\; val\, uri : String \;\} \tag{1}$$

The URI guarantees unique identification. Different spellings, which are extracted to the same identifier, could be interpreted as the same. The user needs to understand and classify the recipe meta information, such as the name, the original spelling of an ingredient declaration, and the author. The representation, which contains information for the user, is the external representation (Equation 2). The result is one data structure that represents the internal and the external data.

$$\begin{aligned} trait\, ValueItem\{ \\ val\, i : Iitem, \\ val\, name : String \;\} \end{aligned} \tag{2}$$

The ingredient declaration (Equation 3) contains a sequence of ingredient items and a quantity. The sequence declares that only one has to be chosen. This sequence is defined as an $or - relation$ of ingredients.

$$\begin{aligned} Ingredient(items : Seq[ValueItem[Item]], \\ quantity : Quantity) \end{aligned} \tag{3}$$

The quantity contains a unit and a value. $NumVal$ is an $Item$ that also contains a numeric value. The $NumVal$ could also be a $NumRange$ that contains a minimum and a maximum value.

$$\begin{aligned} traitQuantity\{ \\ val\, numVal : ValueItem[NumVal] \\ val\, unit : ValueItem[UnitItem] \;\} \end{aligned} \tag{4}$$

The publication contains meta information. It could be a book or a collection. Some books provide references to the source of a recipe. A collection contains the original book additionally.

$$\begin{aligned} Publication\, extends\, Item\{ \\ val\, name : String \\ val\, author : String \\ val\, published : Int \;\} \end{aligned} \tag{5}$$

The cocktail data structure (Equation 6) combines all information of a cocktail. A name of a cocktail is required, but all other values are optional. Preparation and glassware are Item, which represents one taxonomy in the ontology.

$$\begin{aligned} Cocktail(name : String, \\ ingredients : Seq[Ingredient], \\ prepare : Seq[ValueItem[Preparation]], \\ glassware : Seq[ValueItem[Glassware]], \\ publication : Option[Publication]) \end{aligned} \tag{6}$$

4 Phases of Extraction

The five phases of extraction are designed to extract the target data structure.

4.1 Preprocessing and Normalization

The sources of recipes are books or blogs that are written in English. The set of expecting characters (Equation 7) is manageable because it is a small set.

$$[a - zA - Z0 - 9] \tag{7}$$

[1]https://jena.apache.org
[2]https://jena.apache.org/documentation/query/text-query.html
[3]https://lucene.apache.org/core
[4]https://github.com/scala/scala-parser-combinators
[5]http://itextpdf.com

Of course, there are special characters in a text such as ? or −, but these either have a special impact or have to be ignored. The books are converted to raw text with optical character recognition (OCR); therefore, a special character could also show an error. In the first instance, the words and special characters have to be separated into single tokens because these characters have to be processed explicitly. If a special character is combined with a word, neither can be recognized.

$$()|*/-\backslash\text{"}\$\%\hat{}\{\}\S \tag{8}$$

For normalization, every member of the defined set of special characters (Equation 8) will be replaced with itself with additional white spaces (Equation 9).

$$character \rightarrow whitespace\; chracters\; whitespace \tag{9}$$

The result contains words and special characters are separated by white spaces. A splitting by white spaces results in a list of raw tokens. New lines are explicitly a token, because it is valuable information for separating the token before and after. A defined set of stop words, such as *of* or *in*, are removed because these are not required. The result is a preprocessed and normalized list of tokens. Stemming is done by synonyms in the ontology.

4.2 Number Recognition

The raw set of tokens contains information, such as ingredient names or units, but these are not recognized yet. The ontology is the knowledge. All literals in the ontology can be recognized. Further numbers can be recognized, such as written-out names or digits. The following rule set is defined to map the raw tokens to recognized tokens called $ValueItem$. The original value is combined with a chosen item. A slash could be a part of a fraction; therefore, it is mapped to a *slash* item. As the word *or* could also be a part of a range or connect two ingredient names, it is mapped to an *or* value item. The symbol % could be describe a number as a percent value; therefore, it is mapped to a *percent* value item. Since a hyphen could be a part of a numeric range, such as $1 - 2$, it is mapped to a *hyphen* value item. Written-out numbers, such as *three* or digits, are mapped to a *numeric value* item called $NumVal$. Indefinite articles, indefinite pronouns, and qualitative declarations (Equation 10) are mapped to a special numeric value of a value item of type A. These words do not mean the same thing, but are imprecise. Items such as *a lemon* could also be a *small* or a *big* one. An such as *big lemon* is also imprecise because the size of a big lemon is not specified. The main information given is that about one lemon should be used. It must be seasoned, but this cannot mapped in a value.

$$a, an, some, any, small, big \tag{10}$$

4.2.1 Number Combining

Numbers are also written as fractions, such as 1/2. Of course, there are special characters, such as $\frac{1}{2}$, but in the used OCR results, there are only numbers combined with slashes. Following the rule map, a number with a slash and a following number to one item of type $NumVal$.

$$NumVal(n1) :: Slash :: NumVal(n2) \rightarrow NumVal(\tfrac{n1}{n2}) \tag{11}$$

Numbers are used as ranges (Equation 12).

$$\begin{gathered} one\ or\ two\ dash \\ 4\ -\ 5\,cl \end{gathered} \tag{12}$$

Recognized numbers will be mapped to a range if an *or* item or a *hyphen* item connect two numbers.

One number with a following percent item is mapped to the representation as a fraction.

$$NumVal(n) :: Percent \rightarrow NumVal(\tfrac{1}{100}) \tag{13}$$

If the recipe contains half-sentences (Equation 14), the number recognition needs additional rules because there are more than two number values, but only one of them is interesting.

$$\begin{gathered} use\ a\ half\ of\ a\ lemon \\ a\ big\ lemon \\ one\ of\ a\ big\ lemon \end{gathered} \tag{14}$$

The words *use* and $half$ are stop words, but there is more than one item of type $NumVal$. The type A is a $NumVal$ too. The following rule set (Equation 15) reduces the items to one item.

$$\begin{aligned} NumVal(n) :: A :: A &\rightarrow NumVal(n) \\ A :: NumVal(n) :: A &\rightarrow NumVal(n) \\ A :: A :: NumVal(n) &\rightarrow NumVal(n) \\ A :: NumVal(n) &\rightarrow NumVal(n) \\ NumVal(n) :: A &\rightarrow NumVal(n) \end{aligned} \tag{15}$$

The number of combined Equation 14 is applied in the Equation 16.

$$\begin{aligned} A :: NumVal(0.5) :: A &\rightarrow NumVal(0.5) \\ NumVal(1) :: A &\rightarrow NumVal(1) \\ NumVal(1) :: A :: A &\rightarrow NumVal(1) \end{aligned} \tag{16}$$

A fraction combination (Equation 17) has to be combined with other combinations; therefore, the fraction combination has to be completed before the other rules can work. For example, the fraction is mapped to one NumVal (Equation 17) and only then is a number reduction possible.

$$1/2\ of\ a\ lemon \rightarrow 0.5\ lemon \tag{17}$$

4.3 Named Entity

At first, the empty string is ignored and a newline is mapped to an item called $HardSeperator$. If no previous rule is matched, the token is searched in the ontology to find named entities such as ingredients. A Lucene index is used to find a literal or a part of a literal with high performance. A small ontology (Listing 1) is used to declare a Lucene index called *text* with the key *uri* and the value *text*. The URI represents the URI of a property and the text declares a literal. The literal is also connected to the $rdf : type$ to identify the taxonomy.

```
<#indexLucene> a text:TextIndexLucene ;
 text:directory "mem" ;
 text:entityMap <#entMap> ;
 .
<#entMap> a text:EntityMap ;
 text:entityField      "uri" ;
 text:defaultField     "text" ;
 text:map (
  [ text:field "text" ; text:predicate rdfs:Literal ;
  text:predicate rdf:type ]
 ) .
```

Listing 1: Mapping of Index to ontology fields, written in Turtle

The SPARQL query (Listing 2) to find named entities uses *text*, which is the previous defined Lucene query, and a chosen literal, such as *Strawberries*. The allowed taxonomies are ingredient, preparation, glassware, and unit. One of them has to be bound to prevent ingredient superordinates.

The found raw token could contain lexical errors or represent only a part of the appropriate literal. Therefore, the result is sorted by ascending Levenshtein distance. The item with the smallest Levenshtein distance is used as $ValueItem$, if the acceptance criteria (Equation 18) is complied

```
SELECT ?uri ?label ?type {
?uri text:query (rdfs:Literal "Strawberries") ;
    rdfs:Literal ?label
    OPTIONAL{
      ?uri rdf:type ?type .
      ?type rdfs:subClassOf "cocktail://ingredient" }
    OPTIONAL{
      ?uri rdf:type ?type .
      ?type rdfs:subClassOf "cocktail://preparation" }
    OPTIONAL{
      ?uri rdf:type ?type .
      ?type rdfs:subClassOf "cocktail://glassware"
      }
    OPTIONAL{
      ?uri rdf:type ?type .
      ?type rdfs:subClassOf "cocktail://unit"
      }
    FILTER(BOUND(?type))
}
```

Listing 2: Named entity recognition written in Sparql

with.

$$Levenshtein(lowerCase(s1), lowerCase(s2)) <= 2 \tag{18}$$

The found uri of type represents the type of item.

$$cocktail : //preparation/cocktail \rightarrow Preparation(uri) \tag{19}$$

If the acceptance criteria is not complied with, the next token will be concatenated with a white space in between. The new string is evaluated with the named entity query. The tokens are concatenated until the acceptance criteria is complied with or all tokens are concatenated. In this case, the first token is declared as an unknown item.

4.3.1 Entity Combining

There are entities, such as *lemon zest*, which contains at first *lemon*, which is also a entity, but a *lemon* is a short version of *lemon juice* and represents a quantity of about $4-5cl$. If the token *zest* is after *lemon*, then lemon is already recognized. For such cases, the combinations of the original values of recognized entities are concatenated as a string. This string is evaluated as one named entity. If the acceptance criteria is complied with, the items will be merged. The combinations contain two or three items.

4.4 Logic Combining

There are recipes that use phrases such as *do not shake*. Shake is a method of preparation. There are only two different preparations — stir or shake. In this case, it means that should be stirred. Therefore, rules (Equation 20) are needed that convert the negation. If this conversion is not done, a wrong preparation will be parsed.

$$\begin{aligned} Not :: Shake &\rightarrow Stir \\ Not :: Stir &\rightarrow Shake \end{aligned} \tag{20}$$

4.5 Statistical Recognition of Preparation and Glassware

Because of the logic combining, the processing of preparations and glassware items can be identified by the type of the item. The preparations can be filtered easily and interpreted as a list which describes or-relation. The glassware are handled in a similar way.

```
ABSINTHE COCKTAIL
(Use a large bar glass.)
Fill up with ice;
3 or 4 dashes of gum syrup;
1 dash of bitters (Boker’s genuine only);
1 dash anisette;
4 wine-glass of water or imported selters;
wine-glass of absinthe.
Shake well until almost frozen or trapped; strain
it into a fancy cocktail glass squeeze a lemon peel
on top, and serve.
This drink is liked by the French and hy the Amer-
icans; it is an elegant beverage and a splendid ap-
petizer; hut see that you always hae the genuine
absinthe only for mixing this drink.
```

Listing 3: Recipe with more than one glassware items

The example above (3) is a long recipe with many unnecessary items. Two preparation methods are used here (Equation 21).

$$(\textit{shake},\ \textit{cocktail glass or large bar glass}) \tag{21}$$

The result is correct, but the *large bar glass* is probably used for the preparation rather than as the drinking glass. This must be decided be the user.

4.6 Context Analysis

The context analysis is needed to find ingredient declarations with a unit and a number. For this case, the preparation and glassware is filtered because it is unnecessary. For the context analysis, the recipe is considered as an example of a domain-specific language, which has to be described by the grammar. For context analysis, the Scala parser combinators are used. This library parses a string using EBNF grammar. The items are represented as a list of typed items. Therefore, the list is serialized. Every item is serialized with an identifier of type and an index of the list of items (Equation 22). With this information, the original item can be found and the type can be used for the parsing.

$$@\, typeidentifier\ index \tag{22}$$

Some types are not relevant, such as *Preparation* or *Glassware*. Some occurrences of types, such as *Hyphen* or *Slash*, are converted to other types, but these types could still be in the item list. The type hierarchy of items (Figure 2) shows the relevant subset of types for the context analysis. The orange ones are mapped to the serialized format. Subtypes of the orange ones are transparent for the context analysis. The most prominent type is $ProbablyUnknown$, because items that could be important can be transformed if a rule matches, but this is not necessary.

$$\begin{aligned} Or &\to @O \\ ProbablyUnknown &\to @U \\ MeasurementUnitCategory &\to @M \\ IngredientCategory &\to @I \\ Preparation &\to @P \\ Glassware &\to @G \\ NumVal &\to @N \\ HardSeparator &\to @S \end{aligned} \tag{23}$$

The orange types (Figure 2) are mapped in a specific order (Equation 23). The item *Or* is important for the context analysis and will be mapped to itself even though it is probably unknown. $HardSeperators$ are used to separate two ingredients from each other.

The parser is defined by an EBNF grammar. The notation is adapted to enable writing it directly in Scala. The concatenation is represented by a tilde. If a rule matches, a type transformation is possible and is written as two circumflexes. At first (Listing 4), the index *id* and all identifiers are declared as tokens. Regarding the identifier, it is only important to find it; therefore, the mapping is always to the unit type (written as ()).

```
def id            = """(\w+)""".r ^^ { _.toString }
def unknownId     = """(@U)""".r  ^^ { _ => () }
def measurementId = """(@M)""".r  ^^ { _ => () }
def ingredientId  = """(@I)""".r  ^^ { _ => () }
def glasswareId   = """(@G)""".r  ^^ { _ => () }
def preparationId = """(@P)""".r  ^^ { _ => () }
def numId         = """(@N)""".r  ^^ { _ => () }
def separatorId   = """(@S)""".r  ^^ { _ => () }
def hyphenId      = """(@Y)""".r  ^^ { _ => () }
def orId          = """(@O)""".r  ^^ { _ => () }
```

Listing 4: Recognition of internal identifiers

The identifier with index is mapped to the original item (Listing 5). With the declared rules, the items can be identified.

An ingredient declaration is a sequence of items (Listing 6). There are four types. The first type contains only one ingredient name (such as *lemon zest*). There is no unit or numeric value. The second type is a number with an ingredient name.

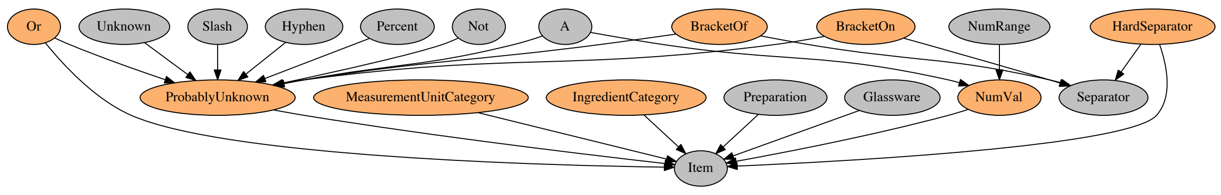

Figure 2: Type hierarchy of items

```
def separator = separatorId     ~ id ^^ { case _ ~ i => item(i) }
def unknown = unknownId         ~ id ^^ { case _ ~ i => item(i) }
def measurement = measurementId ~ id ^^ { case _ ~ i => item(i) }
def ingredient = ingredientId   ~ id ^^ { case _ ~ i => item(i) }
def glassware  = glasswareId    ~ id ^^ { case _ ~ i => item(i) }
def preparation = preparationId ~ id ^^ { case _ ~ i => item(i) }
def num = numId                 ~ id ^^ { case _ ~ i => item(i) }
def or = orId                   ~ id ^^ { case _ ~ i => item(i) }
```

Listing 5: Mapping of identifier with ID of original items

The number is interpreted as part of the quantity of the ingredient. The third contains a unit and an ingredient name, such as *dash bitters*. The number is missing. This case is realistic, but could also be an error because the number was not recognized. The last type contains a number, a unit, and an ingredient name. That is the perfect case, because all the information is found.

```
def i1 = ingredient                  ^^ { case     n => I1(List(n))}
def i2 = num ~ ingredient            ^^ { case q ~ n => I2(q,List(n))}
def i3 = measurement ~ ingredient ^^ { case u ~ n => I3(u,List(n))}
def i4 = num ~ measurement ~ ingredient ^^
         { case q ~ u ~ n => I4(q,u,List(n))}
```

Listing 6: Rules for ingredient declarations

Recipes could contain multiple ingredient names in an ingredient declaration. Multiple ingredients could be $or-relations$ or $and-relations$ (Listing 7). The $and-relation$ describes a list of tokens that do not contain the word *or* or a *number* or a *unit*. The result is a list of names that are probably known. Since an ingredient could be missing in the ontology, the parser would have to work with unknown items if possible. The list contains more than one name. The list is mapped to one unknown item. The $or-relation$ describes a list of ingredients or unknown words, which are separated by the word *or*. This rule recognizes ingredients such as *rum or gin*. The mapping filters the word *or*, while the result list contains only names. The list needs one element.

These rules of ingredient declarations are extended with multiple ingredient names (Listing 8),

```
def mi = andList | orList
def andList = (ingredient | unknown) ~ ((ingredient | unknown)+) ^^
  { case f ~ l => {
    val u = Unknown(l.foldLeft(f.value)((h,i) => h + " " + i.value))
    List(ValueItem(u,u.name))
  } }
def orList = (ingredient | unknown) ~ (orTail?) ^^
  {case i1 ~ i2 => i1 :: i2.getOrElse(Nil)}
def orTail  = ((or ~ (ingredient | unknown))+) ^^
  { case x => x.map(x => x._2)}
```

Listing 7: Rules for multiple ingredient names

which are described by the rule *mi*. In this case, unknown names are allowed. This is possible only if exactly one line is parsed. Therefore, a newline separator has to be consumed.

```
def i1m = mi ~ separator ^^
  { case n ~ _ => I1(n)}
def i2m = num ~ mi ~ separator ^^
  { case q ~ n ~ _ => I2(q,n)}
def i3m = measurement ~ mi ~ separator ^^
  { case u ~ n ~ _ => I3(u,n)}
def i4m = num ~ measurement ~ mi ~ separator ^^
  { case q ~ u ~ n ~ _ => I4(q,u,n)}
```

Listing 8: Rules for ingredient declarations with multiple ingredient names

All rules of ingredient declarations are combined in one rule i (Listing 9). The rules are ordered by the numbers of necessary tokens. A recipe contains a list of ingredient declarations, which is described by the rule il. Between two ingredient declarations, it is possible to consume any separators or unknown items. This rule helps to parse recipes with errors or missing items in the ontology.

```
def i = i4m | i4 | i3m | i3 | i2m | i2 | i1m | i1
def il = i ~ ((unknown | separator)*) ~ il  ^^
{ case h ~ _ ~ t => h :: t   } | success(List())
```

Listing 9: Rules for list of ingredients

The cocktail name (Listing 10) is a list of items that do not contain a unit. The cocktail rule contains the name and the ingredient list.

Some recipes contain the preparation and glassware information after the cocktail name (3). The recognized items are removed, but not all items. For example, an additional separator remains.

```
def name = (num | unknown | ingredient) ~ name  ^^
  { case h ~ t => h :: t  } | success(List())
def cocktail  = name ~ ((separator | unknown)*) ~ il ~ (unknown*) ^^
  { case  n ~ _ ~ in ~ _ => C(n,in)}
```

Listing 10: Rules for a cocktail recipe

Other recipes contain information of the author after the name (Listing 11). Therefore, any separators and unknown items can be consumed between the cocktail name and the ingredient list. The parser parses ingredient declarations for as long as it works because long descriptions after the ingredient list should not stop the workflow of parsing.

```
ROBERTA
Invented by
G. Newman
Juice of 1/2 a small \change{Lime}{lime}.
1/3 \change{Maraschino}{maraschino}.
2/3 \change{Daiquiri Rum}{daiquiri rum}.
Shake.
```

Listing 11: Rules for a cocktail recipe

If all items are recognized, then the parse process is very simple. In this case, the requirements of the recipe are the lowest. In the following recipes, there are only known items. For all ingredient declarations, Rule $i4$ is used, because a number, a unit, and a name are always found. This recipe does not need a separator to be parsed successfully.

> Manhattan Sweet 1 part Italian Vermouth 2 part Whisky 1 dash Angostura 1 piece Maraschino cherry (stir, cocktail glass)

As recipes are different, not all items can be recognized. This parser is designed to parse recipes in the worst case. The worst case of a parsable recipe is that an item sequence is found, which contains a classifiable one and an unknown one in an alternating manner (Equation 24). A classifiable item is an item that matches in the context.

$$\begin{aligned} classifiable\ unknown\ classifiable \\ unknown\ classifiable.... \end{aligned} \tag{24}$$

If a measurement unit is found and is followed by three unknown items and a separator, then the conclusion is acceptable to map the unknown items to one ingredient. If the first ingredient declarations contain only an ingredient name, such as *orange zest*, then a separator between the cocktail name and the first ingredient declaration is necessary to conclude where the cocktail name ends. If there are two ingredient declarations that contain only one name, such as *orange zest* and *egg*, then a separator is also needed to conclude that there are two ingredient declarations.

4.7 Feature Reasoning

The ingredient declarations are mapped to the types $I1$, $I2$, $I3$ or $I4$. The type $I4$ contains all information that is necessary for the distance function: a number, a measurement unit, and one or more ingredient names. The other types have to be converted to $I4$. The missing information is retrieved by default values, which are stored in the ontology. A category such as a *bitter* contains a default quantity with a unit and a volume (Listing 12).

```
<c:defaultQuantity c:unit="cocktail://unit/dash" c:volume="1" />
```

Listing 12: Default quantity in RDF

The SPARQL query (13) for the default quantity needs the URI of the ingredient, the volume, and the unit to be bounded for a successful result.

```
SELECT DISTINCT ?type ?kindof ?volume ?unit WHERE {
 ?kindof rdf:defaultQuantity ?default .
 ?default rdf:volume ?volume .
 ?default rdf:unit ?unit
        FILTER (str(?kindof) =
         "cocktail://ingredient/bitters") }
```

Listing 13: Default quantity query written in SPARQL

If the query was unsuccessful, it is used as a default value (Equation 25). The most frequent case is that the number is not written if it is 1. The units for solid ingredients, such as *piece* for the ingredient *egg*, are stored in the ontology; therefore, it has to be a relative unit, such as *part*. It is possible that the measurement unit goes unrecognized, in which case this conclusion is wrong.

$$Quantity(1, part) \tag{25}$$

Every ingredient declaration can contain more than one ingredient name; in this case, all default values of these ingredients have to be the same. Otherwise, the default value (Equation 25) is used. The ingredient declarations are mapped to $I4$ with default values (Equation 26).

$$\begin{aligned} I1(n) &\rightarrow I4(default(n), n) \quad (26) \\ I2(num, n) &\rightarrow I4(num, default(n).u, n) \\ I3(u, n) &\rightarrow I4(default(n).num, u, n) \\ I4 &\rightarrow I4 \end{aligned}$$

An example is the Hot Spiced Rum recipe, which contains ingredient declarations with and without a measurement unit. The reasoned features are in brackets.

HOT SPICED RUM[6]
1 or 2 lumps of sugar
4 teaspoonfuls allspice
(1) (part) water
1 (part) Jamaica rum
1 (prise) nutmeg

4.8 Validation Metric

After a parsing of a cocktail recipe, the recipe has to be validated. A recipe needs a name. This is not important for the recommendation process but is important for reasons of clarity and comprehensibility. Recipes usually contain a name; therefore, it is an indication that the parsing process was wrong or the input does not contain a complete recipe. A cocktail recipe needs two or more ingredients. If there is only one ingredient, then the recipe would be useless. If there is no ingredient declaration, then this is another indication of a parsing error or wrong input data. Preparation and glassware could be missing. These are optional features. The last part of the validation is a consistency check. With feature reasoning, the ingredient declaration was complete, but not all combinations of ingredient declarations are useful.

There are two ways of using measurement units. Qualitative units are used if an ingredient such as bitters requires the use of small quantities or is a solid ingredient. If all ingredient declarations with qualitative units are hidden, then there are only relative units, such as in the Brandy Crusta Recipe, or quantitative units, such as in the Manhattan recipe.

BRANDY CRUSTA.[7]
3 dashes Maraschino.
1 dash Angostura Bitters.
4 dashes Lemon Juice.
25% Curacao.
75% Brandy.
Stir and strain into prepared glass, adding slice of Orange.

MANHATTAN COCKTAIL[8]
1 dash of gum syrup, very carefully;
1 dash of bitters (orange bitters);
1 dash of curacao, if required;
1/2 wine glass of whiskey;
1/2 wine glass of sweet vermouth;
stir up well; strain into a fancy cocktail glass;

It is a validation criterion that only qualitative and quantitative units or qualitative and relative units be used. If this criterion is not accomplished, the feature reasoning was not correct. For a recommendation, all items have to be known; if one item is not known, then the distance is wrong.

5 Experiment of Parsing Cocktail Books

For an experiment of parsing cocktail books, seven books are chosen, which are used in a PDF format with the underlying text. This is added by OCR techniques. These books are written in English and focus on cocktail recipes.

- 1862 Jerry Thomas — How to Mix Drinks: The Bon-Vivants Companion (New York, USA)
- 1882 Harry Johnson — Bartender's Manual (Chicago and New York, USA)
- 1884 George Winter — How to Mix Drinks (New York, USA)
- 1917 Tom Bullock — The Ideal Bartender (St. Louis, Illinois, USA)
- 1936 Frank Meier—The Artistry of Mixing Drinks (Paris, France)
- 1937 William J. Tarling — Approved Cocktails (London, England)
- 1937 William J. Tarling — Cafe Royal (London, England)

All these books are converted from PDF to plain text using the PDF library *itext*. The result is text in the correct order. This is not trivial, because the books partially contain a layout with two columns, which is correctly recognized by OCR.

Nevertheless, there are errors in words; often, special characters are not recognized or the noise

[6]1882 Harry Johnson — Bartender's Manual
[7]1937 William J. Tarling — Approved Cocktails
[8]1882 Harry Johnson — Bartender's Manual

of the photocopy was recognized as a symbol, such as a dot, or a random single special character, such as \$ (Equation 27). These errors are manageable, because a word with one wrong character is recognizable with the Levenstein distance or one senseless character between two words can be removed.

$$Cr£me\ de\ Cacao \tag{27}$$
$$\$\ Shake$$

There are cases of hardship (Listing 14), however, which cannot be completely recognized. External services services for a misspelling check could help but is not considered here.

```
ORGEAT LEMONADE.
(Use a large bar glass.)
1$ wine glass of orgi~l syrup;
4 tiiblcsl)o~7ii'iil of ~ugar;
(1 to 8 di~sln.~s of Iriiion juice;
2 glass of sliavril ire;
Fill the gli~i~s with water;
Mix up vvtlll ;mil orniiriient with grapes, berries, etc.,
in season, in 11 ti~steful manner ariil serve with a
straw.
r 1
```

Listing 14: Example for a case of hardship

These books contains many recipes but they contain areas without recipes as well. Each book has an introduction, a table of contents, an index of ingredients, and explanations for preparations, glassware, and ingredients. For parsing these books, all areas without cocktails are manually removed. The recipe area remains unchanged.

5.1 Recipe Recognition

For parsing a collection of cocktails, the start of a recipe has to be recognized. These historic books contain headlines that are written with only capital letters. Not every headline has to be a cocktail name, but most do. The plain text is separated by the character of newline to get a list of lines. Every line contains one or more words. The metric to classify a line as a headline (Equation 28) is that a minimum of one word is only written with capitals. Often, a headline ends with a dot and commas could also be present in a headline.

$$(\backslash s * [A-Z][A-Z-.,'\backslash s]+) \tag{28}$$

Every recipe starts with a recognized headline and stops before the next one is recognized. The lines of a recipe are converted back to one string.

If the cocktails are sorted alphabetically, there are cross-headings between the recipes that contain the next character area.

AD-AL.
ADAM.
50% Jamaica Rum.
25% Lemon Juice.
25% Grenadine.
Shake with ice.

Therefore, the first plausibility check is the length of string to classify headings such as $AD-AL.$ to an too small recipe. A parsed cocktail recipe from the chosen books has a median per book between 135 and 365 characters (Figure 3).

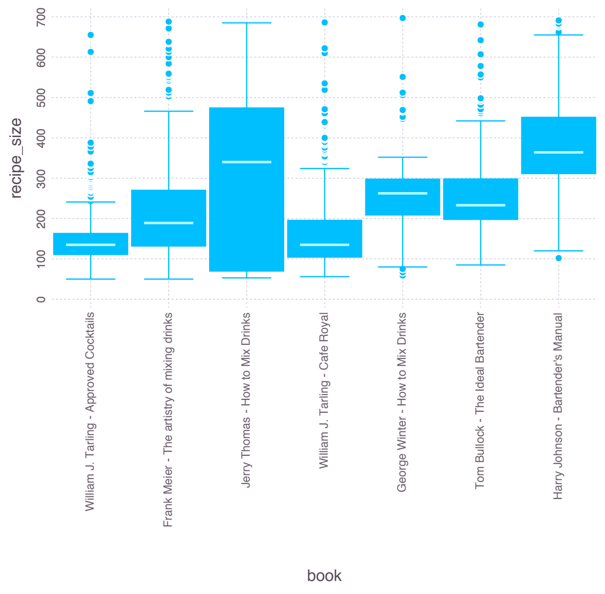

Figure 3: Size of recipe distribution

The smallest recipe is for Oyster Bay with a size of 50. If the length is lower than this, then the probability of it being a recipe is too low. These strings are filtered.

OYSTER BAY.
40% White Curasao.
60% Gin.
Mix.

Recipes could be very long if the description is very long. It is more difficult to set a maximum limit. The longest recipes are about 700 characters. Longer ones contain more than one recipe because the heading was not recognized since there are special characters or some lower case letters.

5.2 Parsing Result

The recipe recognition resulted in 3,294 potential recipes, 30 % of which are classified as too small and 6 % as too high. 21 % are invalidated by the validation metric and 42 % are validated.

Too small recipes potential recipes don't contain a recipe. A better recipe recognition would be better, but these can be filtered without problems. Recipes that are classified as too high are very critical, because there are headings that were not recognized. These recipe collections cannot be parsed. The invalidated recipes are either of very bad data quality or recipes that can be parsed with optimization of the parser or ontology.

The error rate of books (Figure 4) shows that the book with the lowest validation rate has the highest rate of too big recipes. The recipe recognition does not work because most recipes begin with a number, which is not supported. On the other hand, the two books with the lowest invalid rate and a very low rate of too big recipes also have a very big valid rate. These books are well-supported by the parser because the data quality is very high and the recipes are simple. The rest have a very high rate of invalidation because the recipe recognition works but the recipes often contain one or two specialties, which are not supported by the parser.

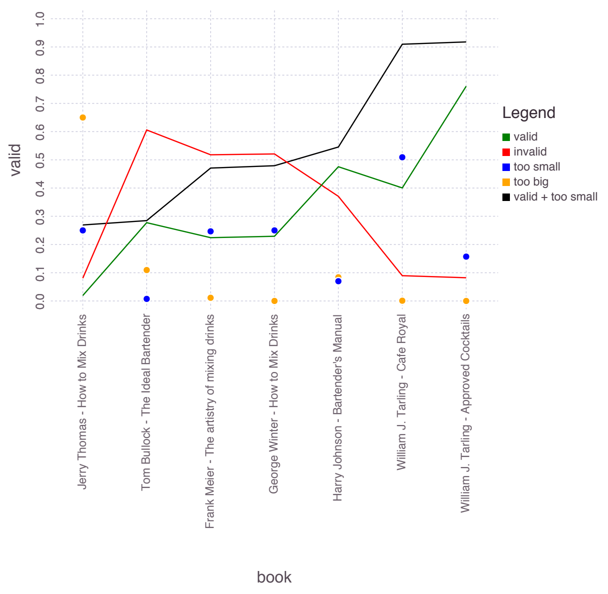

Figure 4: Error rate of book parsing

The parser can be optimized with additional features: Some recipes contain fillers. Fillers are ingredients, such as soda or ginger ale, which are used to fill up the glass. This is not supported yet, because there are two challenges: The first is that the quantity must be in proportion to the size of cocktail glass. The second is that a similar case involves recipes that have declarations, such as *filll up with ice*. Information about ice is usually unnecessary for the recipe, because the preparation method, such as *stir* or *shake*, is sufficient. But this declaration is very similar to an ingredient declaration; therefore, the parser tries to recognize it as a ingredient. Explicit support would prevent that. Optional ingredients are not supported yet. Additionally, $or - relations$ that contain commas (Equation 29) are not supported. This is often used in meta recipes, which contain, at the most, superordinates and no measurement unit or numbers.

$$rum,\ gin\ or\ whiskey \tag{29}$$

Some recipes contain, besides an ingredient declaration, some additional information, such as the origin of the ingredient or an alternative that is not supported. If these features can be used, the number of invalidations will decrease. Poor quality of data is another challenge. A word correction service could help.

The parsing process needs 72 s for 1,415 validated recipes. 51 ms per recipe is a very good performance, which is the result of using indexer and caching mechanisms.

6 Conclusion and Future Work

For the feature extraction of unstructured recipes, a knowledge-based approach is used to find known items. Rules are used to transform items to items with a higher abstraction level, such as ranges. The context analysis is used to find ingredient declarations with a domain specific language.

The variety of recipes is high and the data quality can be very low. The result is that recipes considered as item lists need alternating parsing between a known and an unknown item. The experiment of feature extraction of unstructured recipes shows that recipes are recognizable. Nevertheless, there are many challenges for optimization, such as support of different heading types, fillers or optional ingredients.

The next step is to combine the distance functions with the feature extraction and the evaluation of the recommendation with domain experts.

References

[Sip15] Sippel, Sigurd: Distance functions for knowledge-based cocktail recommendation. (2015). `http://users.informatik.haw-hamburg.de/{~}ubicomp/projekte/master2015-proj/sippel.pdf`