BoB Chatterbot:

A 3-lingual web-based

interactive question answering system

for the University Library

v. 001

1 Main Program Information

1.1 Program identification

1.1.1 **Program Name:** *BoB Chatterbot* is a 3-lingual web-based interactive question answering system for University Library.

1.1.2 **Program Version:** 1.0

1.2 Brief description of an application

1.2.1 **Task of application:** *BoB Chatterbot* is an automatic dialogue system with a web interface that is able to provide interactive autonomous question answering for the number of domain-based (regarded to university library) questions in three languages (English, German and Italian).

1.2.2 **Main algorithm idea:** *BoB Chatterbot* is a java web application with a fronted jsp that is also using JQuery from Google AJAX API library. A knowledge base of BoB is enclosed in an XML file with regular expression patterns for user's questions matching and organized into a topic-tree structure where each topic represents the single answer to a series of questions matched with one regular expression. Bob deals with an "extended" version of regular expression where ORO Perl-style expressions are powered with the Boolean operations (ANTLR parser). For answers re-ranking BoB uses NLP-related technology for a machine-learning-based re-ranking, based on a text corpora (LingPipe library for TF/IDF string similarity, tokenizer, stemmer).

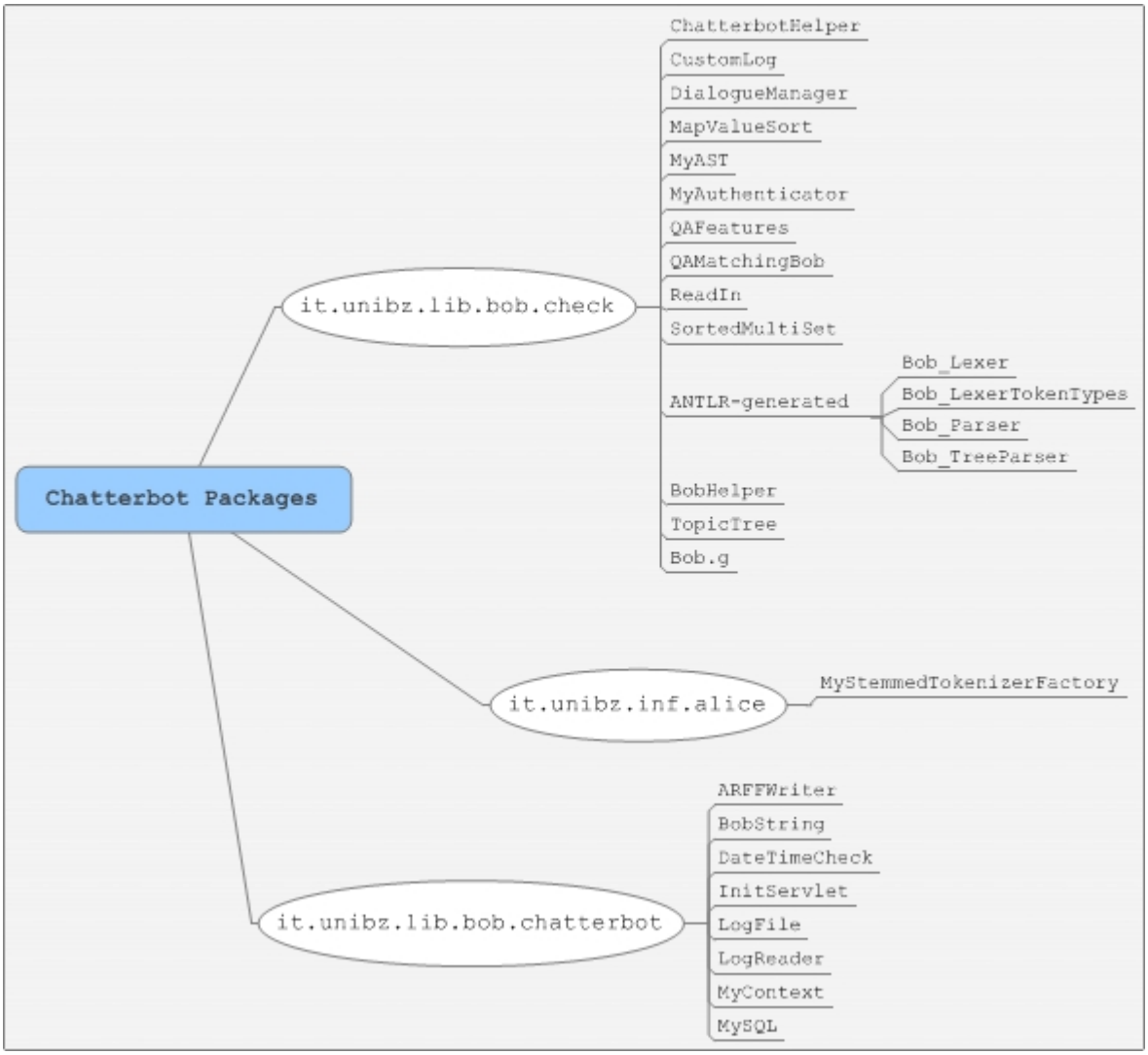
1.2.3 **Program Functions:** *BoB Chatterbot* allows the users to address their questions in natural language (English, German or Italian) to the system via web interface and get predefined answers about the university library life.

2 Program Structure

2.1 Program Organization

2.1.1 **Packages:** The BoB *Chatterbot* Java project is logically divided into three packages:

- `it.unibz.lib.bob.chatterbot` contains proper chatterbot classes for output, database connection and connection with a web interface part.
- `it.unibz.lib.bob.check` pachage is accesible from other bob projects and which contains all the logic for the question responding and toic-tree searching.
- `it.unibz.inf.alice` the package with the Tokenizer class for answer re-ranker.



BoB Chatterbot Packages

2.1.2 Classes:

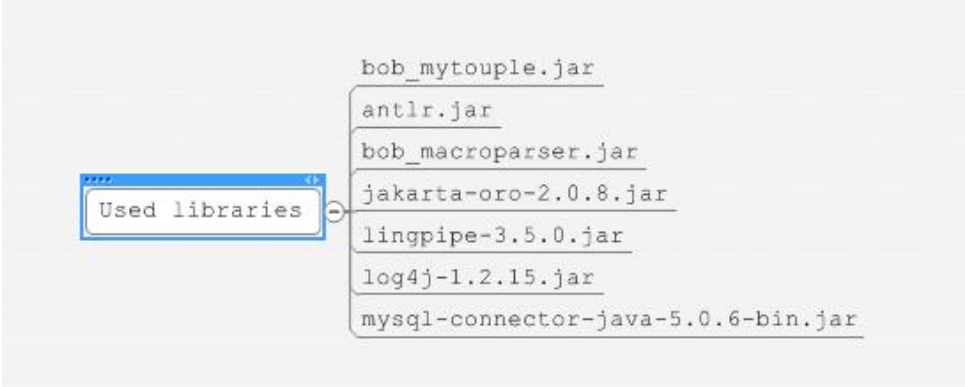
- `MySteemedTokenizerFactory`: implements a factory that creates tokenizers for subsequences of character arrays. The tokenization is based on the following pipe, using existing `LingPipe` code: (IndoEuropeanTokenizerFactory → LowerCaseFilterTokenizer → EnglishStopListFilterTokenizer → PorterStemmerFilterTokenizer)
- `ARFFWriter`: Workpiece that at some point should provide a nice API for writing correct ARFF files, for now works as a file string writer.
- `BoBString`: class that is used for post-processing system answers before they are returned to the user. Add 3 types of HTML links to the system answers for:

- terms highlighted with <option> tags (Replace the <option>-tagged words in a string with corresponding HTML links that trigger a JavaScript function which supplies the tagged words to BoB)
- URLs that link to some URL from PHRASES in the topictree
- normal URLs with regular HTML links

- `DateTimeCheck`: provides method to check special time spans (morning, day, evening).
- `InitServlet`: Java servlet that runs on a server side without a face and initialize `DialogueManager` and Logger parameters according to the context parameters given in `web.xml` (URL of the topic-tree, URLs of the abbreviation files, URLs of training corpus files)
- `LogFile`: represents log file, contains saveDialogue method for logging, which is not used in the current version (instead MySQL database is used).
- `LogReader`: used as a bean in a logReader.jsp
- `MyContext`: defines context servlet listener of the application. The listener is notified about various events, such as application or session events. Has a method that is invoked when the Web Application is ready to service requests.
- `MySQL`: provides access to the MySQL database for question/answer history.
- `ChatterbotHelper`: ChatterbotHelper is another modified version of the BobHelper class, used only in BoB web application. It differs from BobHelper by the constructor parameters and the usage of static methods. It provides some helper methods to work with the topic-trees and the abbreviations files.
- `CustomLog`: Custom version of `log4j.Level` class for logging.
- `DialogueManager`: One of the most important classes, which implements the entire logic for question answering. Contains `getNextNode(query, lang)` method, which returns the node containing the next system response for the query.
- `MapValueSort`: Class for sorting maps of the type <String, String>. Used in the DialogueManager to get a topicID-sorted-map <answer, topicID> of all the answers that can be reached from the topic-tree's "normal" topic nodes.

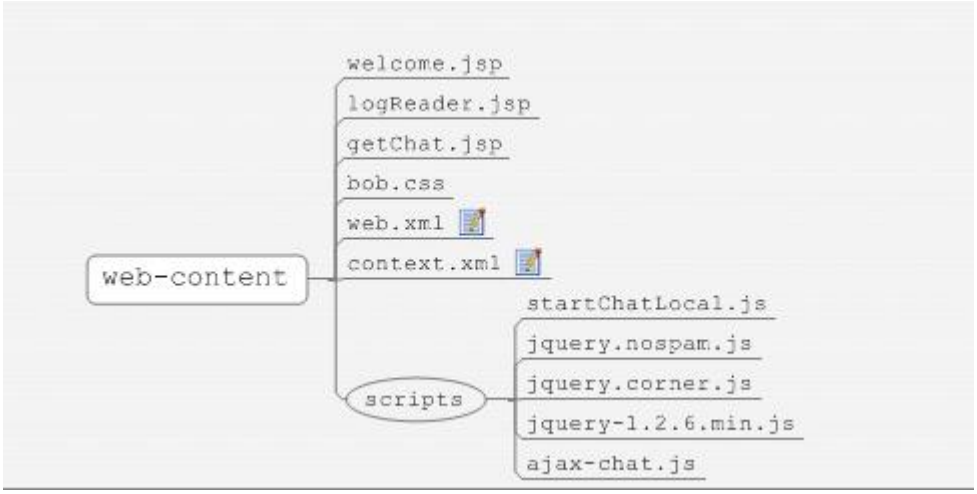
- `MyAST`: contains a modification abstract syntax tree, which is the output format for ANTLR parse. Modification adds to the basic set of parameters also line number information.
- `MyAuthenticator`: The class MyAuthenticator represents an object that knows how to obtain authentication for a network connection by giving credentials. Used by calling `setDefault` method of `Authentificator` for the class to be registered.
- `QAFeatures`: Representation of the set of features that are used for answers re-ranking.
- `QAMatchingBob`: Class that implementing answers re-ranking mechanism, able to calculate total score for an answer candidate, represented by feature values.
- `ReadIn`: Reads in data of various types from (stdin, file, URL) and then converts it to String.
- `SortedMultiSet`: interface that allows multiple "equivalent" objects to be stored. Used in QAMatchingBob to sort re-ranked answer candidate quadruples.
- `BobHelper Helper` class created in order to aggregate sequential calls to `Lexer/Parser/TreeParser` classes and working with abbreviations using `macroparser` classes.
- `TopicTree`: provides most of the tree searching logic, contains an instance of the XML topic-tree document.
- `Bob.g`: The description of an ANTLR grammar. The source Lexer, Parser and TreeParser are generated from.
- `Bob_Lexer.java`: Automatically generated by ANTLR. It scans the extended regular expression string for the relevant tokens.
- `Bob_Parser.java`: Automatically generated from `Bob.g` by ANTLR. It builds a parsing tree out of tokens, extracted by `Bob_Lexer` from regular expression string.
- `Bob_TreeParser.java` is also generated from `Bob.g`. It evaluates whether some user question are matched by the tree that was built by the parser. Algorithm works as following: it applies the required boolean logic and executs calls to `regex.match()` for the atomic (non-extended) regular expression patterns with the user question.

2.1.3 Used Libraries



- `bob_mytuple.jar`: mallardsoft open source library, that presents classes for working with ordered multi-value collections.
- `antlr.jar`: extended Boolean regular expressions parser, regular expression tree (AST – Abstract Syntax Tree) classes, AST tree visualization frame.
- `bob_macroparser.jar` Unifies the classes from `it.unibz.lib.macroparser` used for parsing regular expression macros.
- `Jakarta-oro-2.0.8`: Provides pearl-syntax regular expressions matching engine.
- `Lingpipe-3.5.0.jar`: LingPipe is the library for TF/IDF string similarity, tokenizer and stemmer. The answer re-ranker uses LingPipe for calculating TF/IDF string similarity between user questions and possible answer candidates. LingPipe is proprietary software, but a royalty free version is available under certain conditions (see http://alias-i.com/lingpipe/web/download.html). If you choose to go for LingPipe, you would have to download lingpipe-3.5.0.jar and add it to the project yourself.
- `Log4j-1.2.15.jar`: standart java logger.
- `mysql-connector-java-5.0.6-bin.jar`: JDBC MySQL database connector. One could get the connector by himself from http://dev.mysql.com/downloads/connector/j/ (we did not include this jar, because it is licensed under the GPL, which would require the whole Chatterbot project to be licensed under GPL(instead of LGPL) if we distributed the jar)

2.1.4 Web-Content



- `getChat.jsp`: page that contains nothing but main logic of the chatterbot and calls to the java classes. It is called from Ajax-chat script using HTTP post of JQuery returns an string with an XML-document. Uses `BobString` class from the chatterbot package, which process system answers before they return to the user. Add 3 types of HTML links to the system answers, for a) Terms highlighted with <option> tags b) URLs that link to some URL from PHRASES in the topic-tree, c) Normal URLS. Uses beans

  - `logFile` bean `from it.unibz.lib.bob.chatterbot.LogFile` class for logging

  - `db` bean from `it.unibz.lib.bob.chatterbot.MySQL` from database manipulations

  - `dm` bean from `it.unibz.lib.bob.check.DialogueManager` to invoke question-answering algorithm.

- `welcome.jsp`: the main web page of the project.

  - Contains the markup for all visual elements.

  - Uses Google AJAX API to load JQuery JavaScript library.

  - Uses `ajax-chat.js` local JavaScript file that provide chat functionality (it has `startChat` function and event messages handlers)

  - Launches "`startChat`" method from the `ajax-chat.jsp` or "`StartChatLocal`" from `StartChatLocal.js` on the base of the debug parameters.

- `web.xml:` Web project deployment descriptor, that defines each servlet and JSP page within the web application and defines their mapping, contains context parameters (URL of the topic-tree, URLs of the abbreviation files, URLs of the training data corpora), listener class(`MyContext.java`), configures the web application's entry point (`welcome.jsp`) and indicates the link to jdbc data-source object.
- `context.xml`: Context container that sets up a DataSource for using the MySQL JDBC driver
- `ajax-chat.js:` JavaScript file that defines scripts for messages interaction and creates HTTP post request to the `getchat.jsp` page.

- `blockSubmit` function: is linked to onsubmit form event of the chat form in `.jsp`. Handles the user submitting text via enter key. Instead of submitting the form, send a new message to the server (via sendChatText) and return false.

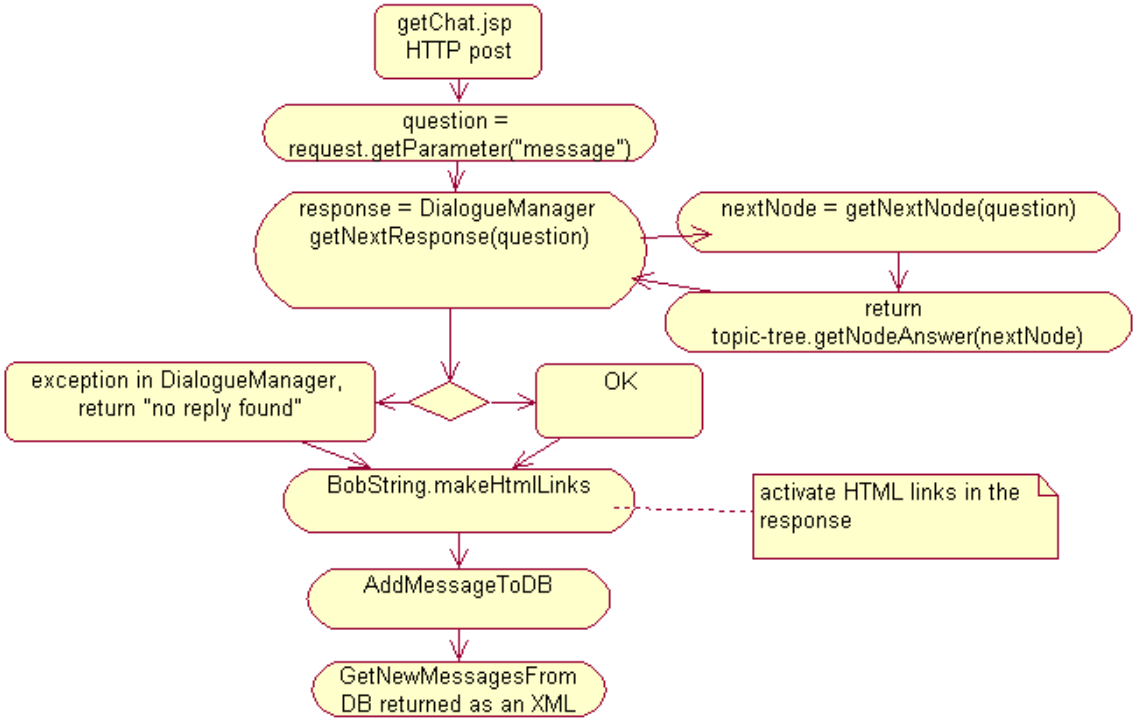
- `sendChatText` function: Add a message to the chat server. Linked to the onclick event of the submit question button from the `welcome.jsp`

- `StartChat` function: gets called upon reset or on a page reload. It restores the GUI (text fields, emotion image), than using "$.post" AJAX JQuery method posts HTTP request to the getChat.jsp file with the language and last message parameters. Assigns to the HTTP post handleReceiveChat callback.

- `handleReceiveChat` callback: Function that appends to the chatDiv div block in the bubble the text received in callback XML value `$(chatDivId).append(dialogEntry(user, text));` where user is either the name of the user or 'BoB' string (for the system answers). Than it updates BoB's emotion image using updateBobEmotion function and shows the evaluation link if the last message wasn't already evaluated (showEvaluationLink function).

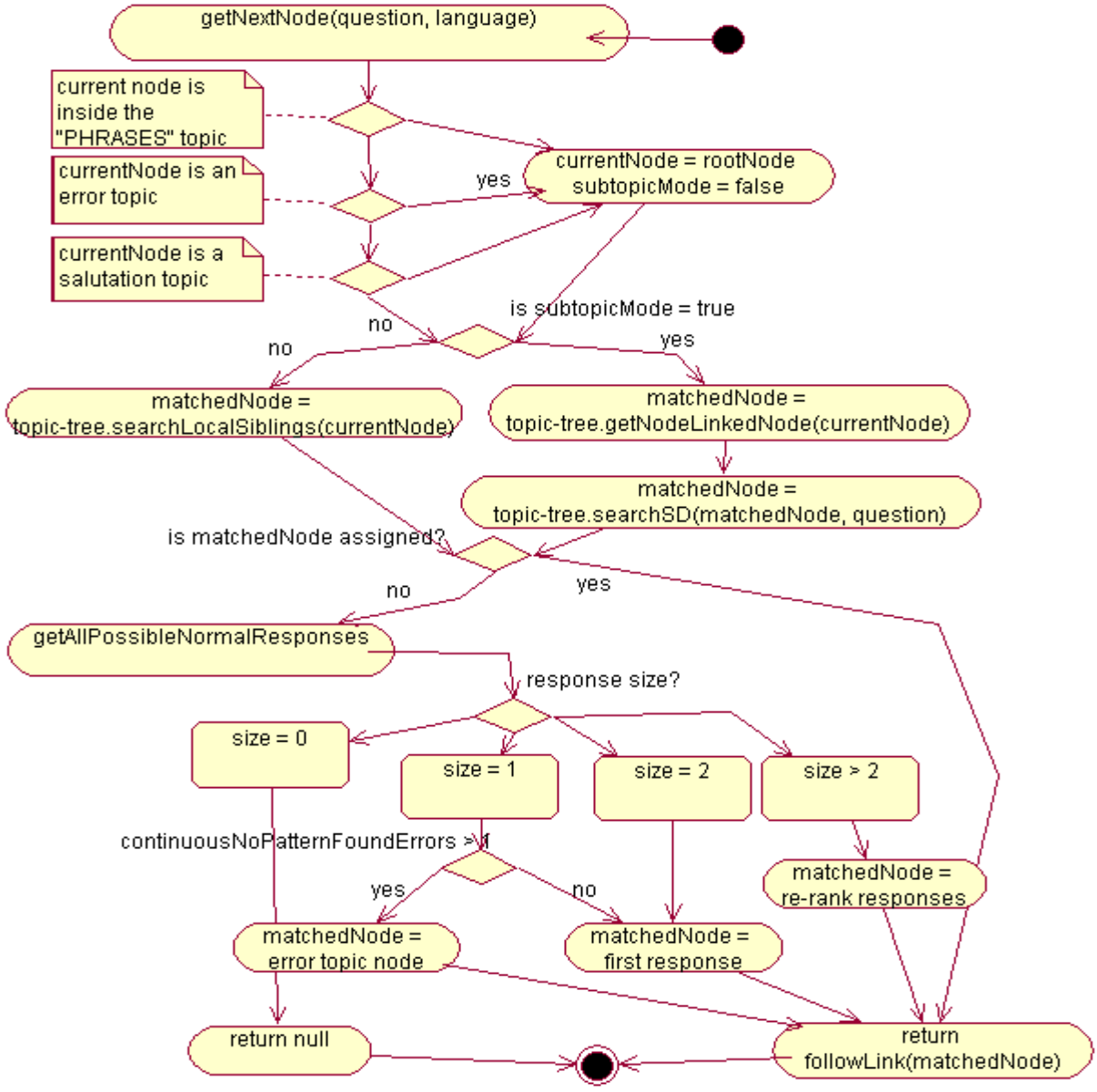
2.2 Program flow

2.2.1 Program flow diagrams



Main response flow

In the main response flow (see diagram): the POST request with the question phrase comes from the `welcome.jsp` page through ajax-chat script to the `getChat.jsp` page. Then the dialogueManager bean's `getNextNode` function is called in order to get the topic node of the answer. The text of the response is extracted wrom the evalueted node using topic-tree `getNodeAnswer` function. If some exception is occurred during the execution of DialogueManager – then the system returns "No Answer found" string. At the end respons is being post-processed by the BobString class and added to the MySQL database. `getChat` page returns the response string in XML format to the callback, defined in the `ajax-chat` script file.



Response topic-node evaluation

Now let us go deeper into the question matching algorithm (see the response topic-node evaluation diagram), which is introduced by the DialogueManager class. So, first the DialogueManager looks if the current topic-node (used in the previous answer) is a particular one (PHRASES, SALUTATION or EROR). In this case current node is set to root and subDialogue mode flag is dropped. If the system is still in subDialogue mode then the question is addressed to a topic-tree, which evaluates matchedNode with the searchSD and getNodeLinkedNode functions. If subDialogue mode is turned off then topic-tree uses searchSiblings method to define matchedNode.

If matchedNode is still not assigned then the DialogueManager tries to get all possible “normal” responses and depending on the result set size either returns the matchedNode or (in the case when we do have multiple answers) launches re-ranker.

So, Bob’s search algorithm works as following:

1. Handle Sub-Dialogues (follow sub-dialogue topics if in sub-dialogue mode; set SD mode).
2. Match the user question against "local" topics (topics with the isLocal attribute == "true")
3. Do a global search of all "normal" (i.e., not pertaining to steps 1 or 2 above) question patterns in the topictree, via `getAllPossibleNormalResponses()`. This might involve the "answer re-ranker".

The answer re-ranker is invoked if in step 3 more than 2 question patterns are found (including the 1 question pattern for the "sorry I did not understand" error message which always matches!) This is done via the call:

```
reranked = tt.getQAMB().rerankAnswers(userUtt, originalResponses, "EN");
```

`rerankAnswers()` is a method of a static instance of `QAMatchingBob` class. The point of this method is to re-rank 2 or more answer candidates (and whose associated question patterns all matched the user question), according to a number of scoring criteria, which are all supposed to determine which of the answer candidates is the best answer for the given user question.

For this ranking, to each of the candidates is assigned a score by the following method:

```
double totalscore = calculateAnswerScore(..);
```

The parameters of it are numeric values that describe in different ways either a specific answer candidate, or the current user question (pattern), or a specific answer candidate in combination with the current user question (pattern). The idea behind the answer reranker is that a specific (weighted)

combination of these numeric values can be used to re-rank the list of answer candidates, and increase the number of cases where the top-ranked answer candidate is correct.

In the following is a short description of each of the parameters used in combination to calculate a score for answer candidates:

- `regexmatchLen`: the string length of the current user question pattern
- `tfIdfSimilarity`: a measure of string similarity between the current user question and the answer candidate under consideration
- `treeSearchRank`: the rank in which the answer candidate under consideration was retrieved by BoB's topic-tree search algorithm. I.e., a matching question pattern further "up in the tree", towards the beginning of topictree.xml, has a lower rank than a matching question pattern further down.
- `nANDoverOR`: the number of Boolean "AND" operators, divided by the number of Boolean "OR" operators in the current question pattern
- `treeSearchProminence`: a way of mapping the treeSearchRank (see above) onto a scale between 0 and 1. Rank 1 has always treeSearchProminence == 1 and last rank has always treeSearchProminence ==0.
- `nOR`: the number of Boolean "OR" operators in the current question pattern
- `topicID`: the first ID number contained in the topic name (e.g., the topic name "bob.02_library.10_books" would have topicID 2)
- `rankOverTopicID`: treeSearchRank divided by topicID
- `nAND`: the number of Boolean "AND" operators in the current question pattern

In `QAMatchingBob`, the method `calculateAnswerScore` maps the values of some of the above features into a score for the answer candidate at hand. This is done in a language-dependent way. For example, in the English version, the score is calculated as follows in this java method:

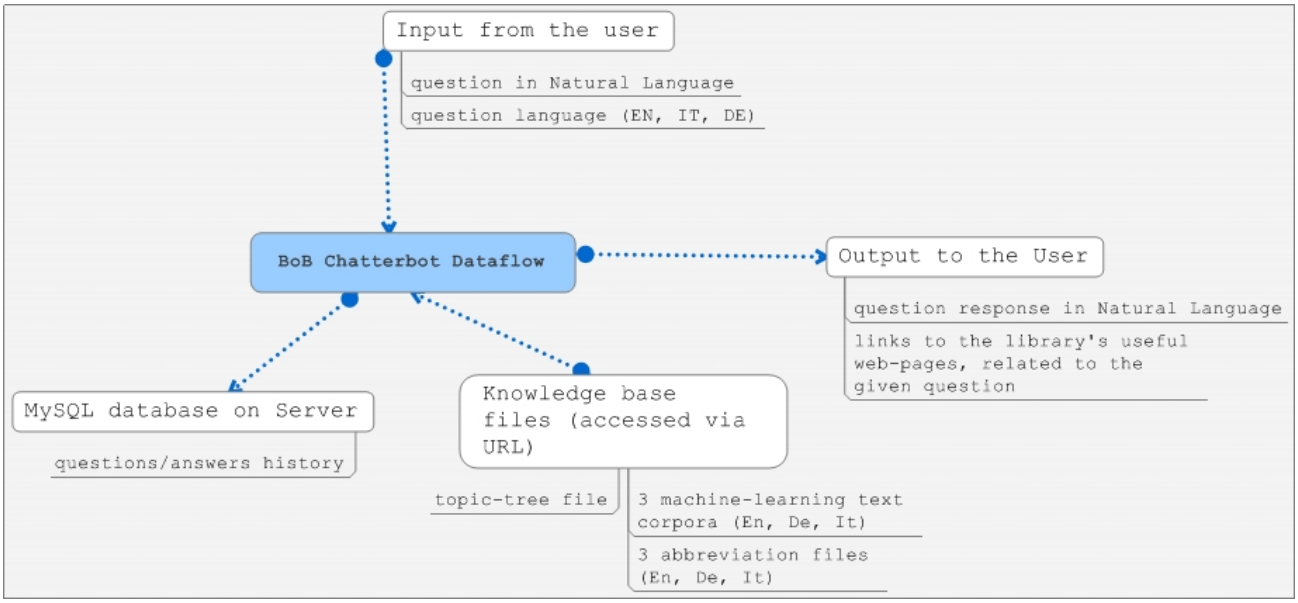
```
- double score = 0;
- score += -0.002179 * regexmatchLen;
- score += 6.454 * tfIdfSimilarity;
```

```
- score += -0.2313 * treeSearchRank;
- score += 0.02476 * nOR;
- score += -0.0239 * topicID;
-  score += 0.00006478 * regexmatchLen * topicID;
- score += -0.03222 * topicID * treeSearchRank;
```

The numeric values hard-coded into calculateAnswerScore are **weights** used to set the influence of each of the features towards the answer score. These weights can either be set by hand, or learned from dialogue data via the machine learning technique. In the latter case, we generate "training data" that map the feature values for all of the above features to a binary outcome, i.e. whether a particular answer candidate was marked as "correct" or "incorrect" for the particular training question. These training data are used to estimate a statistical model that encodes how specific feature values correspond to correct answers.

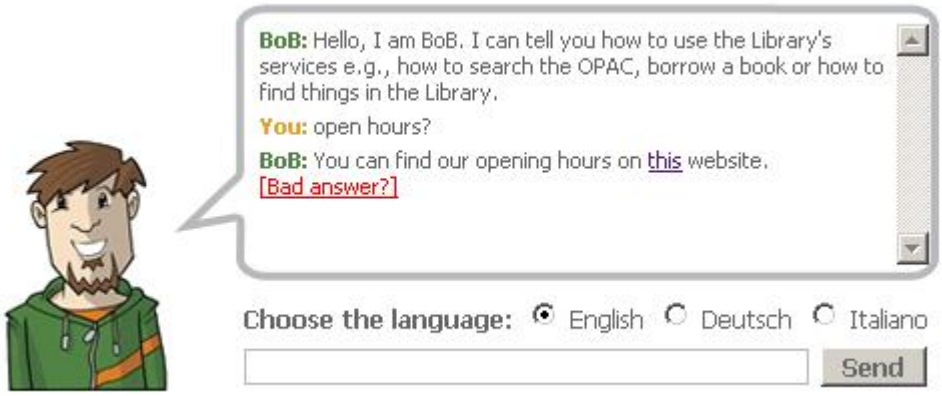
Generation of training data for Machine Learning based estimation of the model is done with the **BBCheck** application. Also, this application is used to test the answer re-ranker after it has been adapted with new feature weights taken from the learned model. The models themselves were estimated in the statistical software package "R", using a Logistic Regression framework.

2.2.2 Data flow diagrams



BoB Chatterbot takes input question from the user (see dataflow diagram) and using its own knowledge base (represented by topic-tree file, 3 abbreviations files and three machine-learning corpora) returns the proper reply to the user. All question and answers are logged by the system and saved in the MySQL database, so one could leverage from learning these logs and improve the system.

2.2.3 Dialogs description



Bob's web interface is represented with one single page with a textbox for new questions and a "message bubble", where all posed questions and system's answers are displayed. In the message bubble user might also click the links given by *BoB* to visit related to the question pages or evaluate the bad answer.

3 Data Organization

3.1 Input data format description

3.1.1 Extended regular expressions structure

Extended regular expression syntax is described in `Bob.g` file of chatterbot project (package `it.unibz.lib.bob.check`). It differs from the "ordinal" Perl regular expressions by introducing Boolean operators:

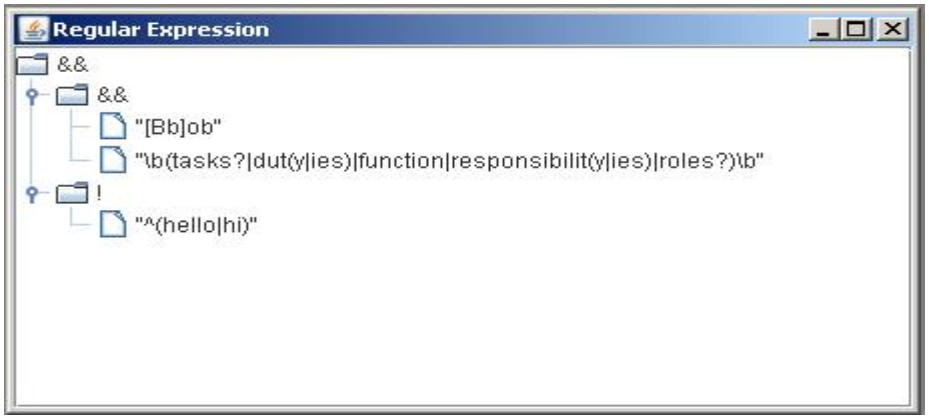
`OR : "||"; AND: "&&"; NOT: "!";` And their parenthesis () grouping.

```
("[Bb]ob" && "#AUFGABEN#") && ! "^(hello|hi)"
```

In this case will be matched all phrases that consist of two parts: first part should contain the words “Bob” or “bob” and the second one should match the “AUFGABEN” abbreviation (which looks like following “(tasks?|dut(y|ies)|function|responsibilit(y|ies)|roles?)"). And one more condition: the matched phrase should not start from words “hello” or “hi”.

Order of parts is irrelevant, so both of “tasks bob?” and “bob tasks?” will be matched.

The whole parsing tree structure of an extended regular expression could look like this:



Regular expression tree

3.1.2 Abbreviation files structure

It’s a text file with a list of regular expression’s abbreviations (one per row). Structure of abbreviation is following: ABBREVIATION_NAME = “\b”+ +REGULAR_EXPRESSION+”\b”, where “\b” a word boundary symbol used in RegEx. Example (for English version):

```
WARUM = \b(why|what for|wherefore|for (what|which) reasons?)\b
```

In this case expressions that are matching this abbreviation will be: “why”, “what fore”, “wherefore”, “for what reasons?”, “for which reasons?”

3.1.3 Topic-tree file structure

Topic-tree is chaterbot’s “knowledge base” represented as an xml file with a particular structure. Each topic within a tree matches some questions with predefined reply and has several **attributes**:

- **`active`:** if the topic is active in current database.

- **`isLocal`:** if the topic is “local”. The idea behind "local" topics is that they can be used by librarians to encode so-called "context-dependent follow-up questions". Example:

  ```
  User Question 1:  Do you offer guided tours?
  System answer 1: We organize guided tours every Wednesday.
  User Question 2:  How do I sign up?
  ```

  For the system to “understand” follow-up questions, it does need to know what the dialogue “was about” in the previous answer. BoB's search algorithm keeps this kind of information by starting its search for a matching question pattern for user question 2 at that node in the topictree where system answer 1 was found. What is required for the above example to work is a question pattern with the "isLocal" attribute set to "true", being a sister node of the node that contained system answer 1. Topics with "isLocal==true" contain question patterns that are overly general, so that they can match under-specified user questions as user question 2 in the above example. Such question patterns must not be included in the normal search for matching question patterns that spans the whole topictree, or they would be matched in many wrong situations.
- **`isSubDialogue`**: topic is a sub dialogue, i.e. matching the user’s response to the system question.
- **`isSystemInitiative`**(not supported yet): System initiative topics, and which could be returned to the user for the questions which do not find any match. The idea is that instead of a generic message such as "Sorry, I did not understand", BoB could respond with a message which is more relevant to the topic which was talked about in the previous user question and system answer, such as:

  `"I did not understand; are you looking for more information about the OPAC?"`

  Currently the BoB algorithm does not use this feature, and topics with "isSystemInitiative==true" were kept in the topictree only to have them in case we support this feature one day.
- **`name`**: unique topic’s name that identifies it within the tree.

Topic elements:

- **<regex>** regular expressions in 3 languages (might contain abbreviations, described in their abbreviation files). regex element's structure looks like this:

```
<regex>
        <languages>
                <DE>(("^ *$")||("(#JA#|#WEITER#|#FRAGE_WAS# (muss|soll) #ICH# .*machen)"))</DE>
                <EN>(("^ *$")||("(#JA#|#WEITER#|#FRAGE_WAS# (must|should|do) #ICH# .*do)"))</EN>
                <IT>(("^ *$")||( "(#YES#|#GO_ON#|#QUESTION_WHAT# (devo) #I# .*fare)"))</IT>
        </languages>
</regex>
```

- **<answer>** contains predefined system's answer to a matched question (in 3 languages). Has a languages element with a structure equal to the one in RegEx.

```
<answer>
        <languages>
                <DE>[03]Ja, natuerlich. Alle Medien mit einer gruenen Etikette koennen ausgeliehen werden.</DE>
                <EN>[03]Yes of course. All items with a green sticker can be checked out.</EN>
                <IT>[03]Si, certo: si possono prendere a prestito tutti i libri con l'etichetta verde.</IT>
        </languages>
</answer>
```

- **<link>** element can substitute the "answer", if current topic matches the questions with an answer already specified in another topic. In this case we could just specify the referent topic's name by giving a link.

```
<link>stella.10_AUSLEIHE.03_AUSWEIS.07a_Ausweis_WieBekommen</link>
```

- **<name-trace number>** element may appear in the scope of a topic element. Its purpose is to keep the history of all the names of the topic element that it has ever had. It is useful for the cases when the topic is needed to be moved to some other location in the topic-tree to keep the implicit naming convention within the xml topic-tree file (topic names encode the location of a topic by using a path-like notation with dots to denote hierarchy). Name-trace with the highest number attribute is the current topic name.

```
<topic name="bob.CurrentTopicName">
...
        <name-trace number="1">bob.OldTopicName1</name-trace>
        <name-trace number="2">bob.OldTopicName2</name-trace>
        <name-trace number="3">bob.CurrentTopicName</name-trace>
...
</topic>
```

Name-trace within the topic

Tracing mechanism was implemented due to the fact that in the presence of the `link` elements in the topic-tree to provide topic-tree integrity. Bob consider not only the current name IDs of the topics, but also all the old ones, so each topic could be easily renamed and all the links to it will be maintained through the name-trace ID.

For the librarians not to take care of name-tracing there was implemented an SVN post-commit hook, that is executed whenever a topictree.xml file is checked into the SVN repository. This is a python script which must have been installed on the SVN server. This script basically do the following:

1. perform svn lock of the topictree.xml
2. run XSLT transformations on topictree.xml to add new name-trace elements to topic elements if needed (if some topic name was changed since the last commit of topictree.xml)
3. commit topictree.xml

PHRASES topics:

Some of the topic name attributes start with the string "`PHRASES`." As our convention, we have used such names for system answers consisting only of a URL (e.g., the library page with its opening times). The BoB search algorithm treats links to topics whose names start with "`PHRASES`." differently from other topics. Such links does NOT trigger sub-dialogue mode, as it would have occur for any other topic whose name starts with "`bob.`"

3.1.4 Machine-learning corpus file

Used can indicate a machine-learning corpus (in one of 3 languages) to in order to use answer re-ranking mechanism. BBCheck assembly includes 3 prepared corpora of 1.000.000 lines of text taken from WAC (web as corpus).

3.2 Database data format

Bob's database for messages history logging consists of one table "message" with the following columns: autoincremental message_id, chat_id, user_id, user_name, message and post_time.

```
mysql> DESCRIBE message;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| message_id | int(11)     | NO   | PRI | NULL    | auto_increment |
| chat_id    | int(11)     | NO   |     | 0       |                |
| user_id    | varchar(32) | NO   |     | 0       |                |
| user_name  | varchar(64) | YES  |     | NULL    |                |
| message    | text        | YES  |     | NULL    |                |
| post_time  | datetime    | YES  |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
```

The table could be created using this kind of script:

```
DROP TABLE IF EXISTS `message`;
CREATE TABLE `message` (
 `message_id` int(11) NOT NULL auto_increment,
 `chat_id` int(11) NOT NULL default '0',
 `user_id` varchar(32) NOT NULL default '0',
 `user_name` varchar(64) default NULL,
 `message` text,
 `post_time` datetime default NULL,
 PRIMARY KEY  (`message_id`)
) ENGINE=MyISAM AUTO_INCREMENT=3994 DEFAULT CHARSET=latin1;
```

4 Application Deployment

4.1 Setup databse for questions/answers history

In order to setup database for loggin the history of questions/answer one should

1. create MySQL database named *chatterbot* ("`CREATE DATABASE chatterbot;`")
2. Create the user bob ("`CREATE USER 'bob';`")

3. Change password of the *bob* user ("`update user set password=('s93ks92pa') where user='bob';`")
4. Create *message* table as it written in below:

```
DROP TABLE IF EXISTS `message`;
CREATE TABLE `message` (
 `message_id` int(11) NOT NULL auto_increment,
 `chat_id` int(11) NOT NULL default '0',
 `user_id` varchar(32) NOT NULL default '0',
 `user_name` varchar(64) default NULL,
 `message` text,
 `post_time` datetime default NULL,
 PRIMARY KEY  (`message_id`)
) ENGINE=MyISAM AUTO_INCREMENT=3994 DEFAULT CHARSET=latin1;
```

5. Give *bob* user permissions to access the database just created ("`GRANT ALL PRIVILEGES ON chatterbot.* TO 'bob';`").

   All the DB connection parameters could be changed in the `context.xml` file, where `jdbc/bobLogger` resouce is defined.

```
<Resource name="jdbc/bobLogger"
    auth="Container"
    type="javax.sql.DataSource"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost/chatterbot"
    username="bob"
    password="s93ks92pa"
    maxActive="5"
    maxIdle="10"
    maxWait="-1"
    removeAbandoned="true"
    removeAbandonedTimeout="60"
    logAbandoned="true"
    testOnBorrow="true"
    validationQuery="SELECT 1 FROM DUAL"
    />
```

4.2 Setup Knowledge base

Bob's knowledge base consists of seven files (topic-tree, English abbreviations, German abbreviations, Italian abbreviations, English text corpus for the re-ranker, German text corpus for

the re-ranker and an Italian text corpus for the re-ranker). All this resources are listed in web.xml file of the project, so it is needed to put these files online and provide to the Bob a URLs in the configuration file.

```
<context-param>
    <param-name>urlTopicTree</param-name>
    <param-value>
        https://location-site/topictree.xml
    </param-value>
</context-param>
<context-param>
    <param-name>urlAbbrevFileEN</param-name>
    <param-value>
        https://location-site/stella_abkuerzungen_EN.txt
    </param-value>
</context-param>
```

4.3 Compilation and Deployment

- In order to compile BoB one need to download and add to the project missing libraries, which are not distributed under the LGPL license:

- LingPipe lingpipe-3.5.0.jar has to be downloaded (see http://alias-i.com/lingpipe/web/download.html) added to the Eclipse project and copied to the directory $WORKDIR/Chatterbot/WebContent/WEB-INF/lib

- After all the previous steps have been done, BoB chatterbot could be deployed on the Tomcat server using Eclipse or manually (in this case all the used jar libraries also have to be added by hand)

- URLs for topictree.xml and all 3 abbreviation .txt files have to be set to the location of their local SVN copies in WebContent/WEB-INF/web.xml file.

- MySQL connector jar (e.g.,mysql-connector-java-5.0.6-bin.jar) has to be copied to your Tomcat's $CATALINA_HOME/common/lib (for Tomcat 5 / 5.5) or $CATALINA_HOME/lib(for Tomcat 6) in order for connection pooling to work. Connector could be downloaded from http://dev.mysql.com/downloads/connector/j/

5 Software Licence

In order to provide maximum flexibility as to how the code can be used by customers we distribute the project to the research community under the LGPL (GNU Lesser General Public License) v3 open source licence (http://www.gnu.org/licenses/lgpl-3.0.html). Bob's source code is available for

downloading on google code site where other researchers can contribute their changes and improvements: http://code.google.com/p/chatterbot-bob/

Note that we distinguish chaterbot source code, which is published under the LGPL licence via the given link from Bolzano-specific hand-crafted parts of Bob (such as: topic-tree xml file, abbreviation files, BoB's face images). These parts of the project are the property of the "Library of the Free University of Bozen-Bolzano" and must not be passed on to third parties without it's consent.

Why do we use the LGPL:

The Chatterbot project is licensed to everyone under the LGPL. The official name of the LGPL is "GNU Lesser General Public License," reflecting its heritage as a derivation of the "GNU General Public License," the GPL. The LGPL is different from the GPL in important ways. Since we don't license the Chatterbot project under the GPL, we won't bother explaining those differences or describing the GPL. Only the LGPL license applies to the Chatterbot project software. We use the LGPL for the Chatterbot project because it promotes software freedom without affecting the proprietary software that sits alongside and on top of the Chatterbot project.

The LGPL is an open source project licence:

1. The LGPL allows you to do the following things with the Chatterbot project software: use the Chatterbot project software as a component of your business
   applications in any way you wish, including linking to the Chatterbot project from your own or other proprietary software.
2. Make unlimited copies of the Chatterbot project software without payment of royalties or license fees.
3. Distribute copies of the Chatterbot project software, although it will be much easier to refer anyone who wants copies directly to our website, which we will publish on http://code.google.com/p/chatterbot-bob/
4. Make changes to the Chatterbot project if you need to do so for use within your own company. The LGPL license does not require you to share those internal changes with the rest of the community.
5. Distribute changed versions of the Chatterbot project to others, but if you distribute such changed versions you are required to share those changes with the rest of the community by publishing that changed source code under the LGPL.(see http://www.jboss.com/pdf/Why_We_Use_the_LGPL.pdf for further information)