VOICE ENABLING THE AUTOSCOUT24 CAR SEARCH APP

Felix Burkhardt, Jianshen Zhou*, Stefan Seide*, Thomas Scheerbarth*,*
Bernd Jäkel+ and Tilman Buchner+

** Deutsche Telekom Laboratories, Berlin, + AutoScout24, München*
Felix.Burkhardt@telekom.de

Kurzfassung: A text parser to match keywords in short texts based against vocabularies and numerical value descriptors is introduced. It is used for a voice search extension of the AutoScout24 App, which enables users to search for second hand cars by selecting features in graphical drop down menus. With our extension, the user can simply say the search query in natural language, using a colloquial vocabulary, instead of selecting from long text lists on a small hand-held device.

1 Introduction

We developed a text parser that finds keywords in short texts based on vocabularies and numerical value descriptors. It is used for a pilot study in the AutoScout24 App, which enables users to search for second hand cars by selecting features in graphical drop down menus. With our extension, the user can simply say the search query in natural language, using a colloquial vocabulary, instead of selecting items from long text lists on a small hand-held device. The article reflects on the underlying text parser software library, the working process to maintain and tune the parser, the integration in the AS24 App and results from the user evaluation so far.

The general functionality is depicted in Figure 1. The primary interaction mode of the App consists of the selection of some features from menu drop-down lists by pointing and viewing the cars that are available with the specified features in the database. With our enhancement, the user can now click a microphone button in the upper right hand corner, say the search terms in natural language, and, after confirming the recognition results, the menu is filled with the selected values. The search can then be executed, or further selections been made either by pointing or voice.

The AutoScout24 app exists for several platforms, e.g. Android and I-Phone, and is installed by more than two million users. The system described here was only implemented for the Android platform, mainly because of automatic speech recognition (ASR) licensing issues.

The project is a continuation of our series of projects concerning the voice based access to internet content on mobile Apps as a direct predecessor to the “AskWiki” App that performs question answering based on Wikipedia content [3].

This article is structured as follows. After a short discussion of related literature in Section 2, we first describe the architecture of our approach in section 3. Then, we elaborate on the text parser algorithms in section 4. section 6 is about the integration of the voice recognition system in the AutoScout24 app. After this, we describe evaluation studies in section 7. The last section 8 concludes the article and gives an outlook on future work.

2 Previous work

After the acquisition of Siri by Apple in 2010 and successful market introduction, voice enabled search becomes more and more natural to smart phone users. Google voice search offers

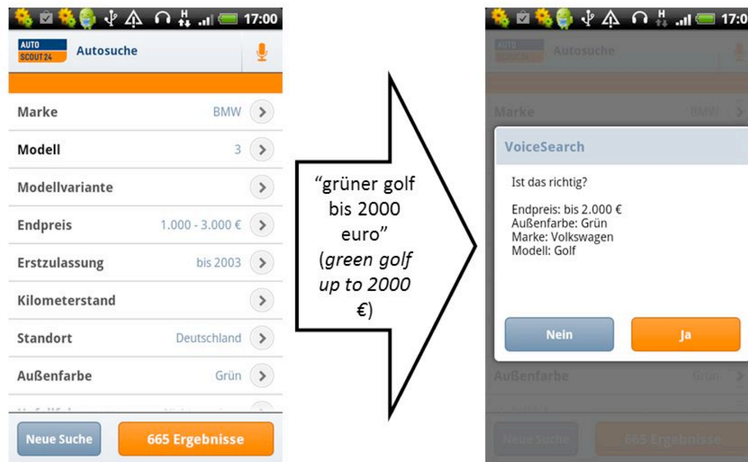

Abbildung 1 - Screen shots of the App GUI before and after issuing a voice search, demonstrating the approach: a user can fill some menu fields by saying the values in colloquial language.

a similar service, both offer, in addition to internet search, services like voice dictation, for example to dictate SMS, command and control (“*call Peter at home, book a table in the Italian restaurant*”), and question answering (“*what’s the capital of Romania?*”), although until now primarily in English. We developed to our knowledge the first German voice-based question answering service with the AskWiki app [3] which was released to the public in January 2012.

Under these general services, voice search systems for restricted domains have been developed. Song et al [7] describe a system to voice search for media data. While they incorporate a phonetic similarity model to increase recall on their data, we do this by hand tuning the vocabularies with synonym entries that were detected when analyzing the user log entries.

In [6], Voice search is even used to generate robust SMS text detection by matching the search query against database of SMS template snippets. Voice search generally might be a technology to overcome problems like illiteracy and information access in the developing world, like described in [1].

While we search in a structured database give an natural language, car ad like expression, [4] describe a system that goes the other way round by extracting structured data from car ads.

3 Architecture

Abbildung 2 - Overview of the general approach.

Figure 2 gives an overview on the general approach of the system. We did not implement an own speech recognizer or utilize a local installation, but simply interface the Google ASR

service that is part of the Android system. The recognized text in form of a N-Best list (listing the *N* most probable recognition results) is sent to a text parser server and then returned to the Application. On the App's graphical interface, the result can be unified with previous results, the user can make adjustments, and the menu structure of the graphical user interface can be filled. As a third step, the query can then be searched in the AutoScout24 database.

The disadvantage of this approach is that the grammar that is used by the text parser can not be used by the speech recognizer, which of course would help to get more stable recognition results, especially in noisy environments. But several advantages outweigh this:

- This approach is more flexible with respect to interface different speech recognizers, e.g. when porting to Apple I-phones.
- An own speech recognizer would be too costly in terms of license fees.

Still we were not sure whether the speech recognition quality would be robust enough for this application and did a preliminary evaluation described in section 7.

An overview on the parser architecture is shown in Figure 3. The processing of a query to a result set happens in five modules, which we describe in the following paragraphs. The input query is the result of the automatic speech recognizer (ASR). This means that some formatting constraints can already be assumed, for example the input always consists of lowercase letters and does not contain special characters.

Preprocessor: A general preprocessing can be executed on the input query. As parsing involves finding a match between the input query and the vocabulary entries, both sides may be processed to add similarity between them via normalization. For example, the car model named "330i", might be translated by the speech recognizer to "3 30 i", "330 i" or "3 30i", depending on the prosody used by the speaker. In order to find a match in the vocabulary, we state that all combinations of several numerals followed by a non-numeral must be separated by a white space character and add a transformation rule to the preprocessor. Alternatively we could have added all these forms to the vocabulary, with the disadvantage that the context depth of the parser (the number of words to be taken as a single token) would have to be enlarged. This would not have helped in this case, because the model "330i" is not in the vocabulary (it is actually not a model but a model variant), but "330" is. Because of the inserted white space, this can now be matched.

Also, a unified approach to encode numbers as digits ("500") or words ("five hundred") is handled here.

Parsing numerical values: After preprocessing, the input gets matched against the vocabulary by the island parser. Firstly, numerical values are detected. The exact algorithm is described in section 4. Numerical values in the context of car search are, for example, price, first registration, mileage or engine power. The words that were interpreted in this module are removed from the search string. Of course we wouldn't want a token like "300" to be interpreted both as a number and as a model name. This leads in German to errors for sentences like "fiat 500 bis 1000 euro" (*fiat 500 up to 1000 euro*), because "500 up to 1000 euro" gets interpreted as a price range, and the model "fiat 500" not recognized, but only the brand "fiat". But this is a natural ambiguity which cannot be avoided.

Parsing entities: The remaining string gets matched against the vocabulary entries (entities), the algorithm is again described in section 4. Beneath the set of recognized entities, a rest string containing the words that were not interpreted, is delivered and is logged by the system. This can later be used for vocabulary tuning when searching for missing synonyms.

For example we saw that in the logs appears the string “meter” that is not interpreted, but certainly a mis-recognition from the speech recognizer of the word “kilometer”, so we added this as a synonym.

Result validation: The output of the two preceding modules is a set of values and entities like brand, model, or accessories. In this module do some checks to ensure model-brand consistency, even adding a missing brand for a given model, which is necessary because the search on the AutoScout24 database can not be executed without a stating a brand.

Vocabulary extraction: Because the AutoScout24 model and brand database is updated frequently when new models appear on the market, the generation of the parser recognition vocabularies from the database must be done automatically. A set of pattern extraction rules can be used to do this, as explained in section 5.

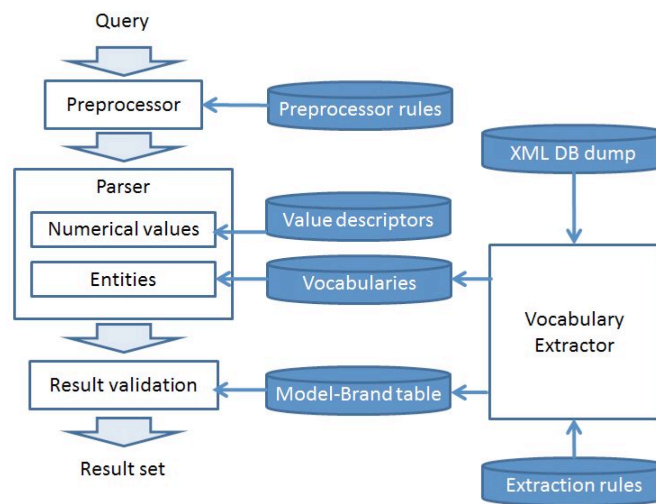

Abbildung 3 - Architecture of the parsing approach.

4 Text Parser library

The algorithm was implemented in Java and can be described as a form of a parsing expression grammar [5], where we generate the expressions to be detected beforehand by regular expressions and store them in a vocabulary.

The world is divided into “entities”; an information entity, like “*id:234, synonyms: 2 cv, ente*” and “values”; a numerical value with unit and constraints, like “*Val: 2003, unit: construction year, isMinVal:true*”.

The central class is the Parser class, which is instantiated stating a series of vocabularies and value descriptors. Vocabularies contain entity descriptors with synonyms and ids, in a comma separated format. An example would be:

```
id,synonym_1,synonym_2,synonym_N
234,2cv,2 cv,ente
```

Value descriptors describe numerical values and are characterized mainly by a unit string. Further, the information whether the numerical value is post fix or prefix in relation to the unit must be stated explicitly, for example “*500 euro*” vs. “*year 2003*”. Via the methods *hasLowerBound* and *getLowerBound*, the parse results for regions can be retrieved, an example would be “*200*

to 500 euro”. Via the methods *isMax* and *isMin*, the information whether the value is a lower or upper bound can be retrieved, an example would be “*at most 500 euro*”.
The parser instance can then be used to parse a string and deliver a parse result as a structure of entities and values. The parsing process itself is programmed in a two step process. Firstly, the values are extracted by detecting the unit strings and extracting near-by numbers as values. Secondly, the remaining bag-of-words set, i.e. all possible string groups given a certain context depth, is used to match against the vocabularies.
We plan to release the library as open source in the future.

5 Maintenance Process

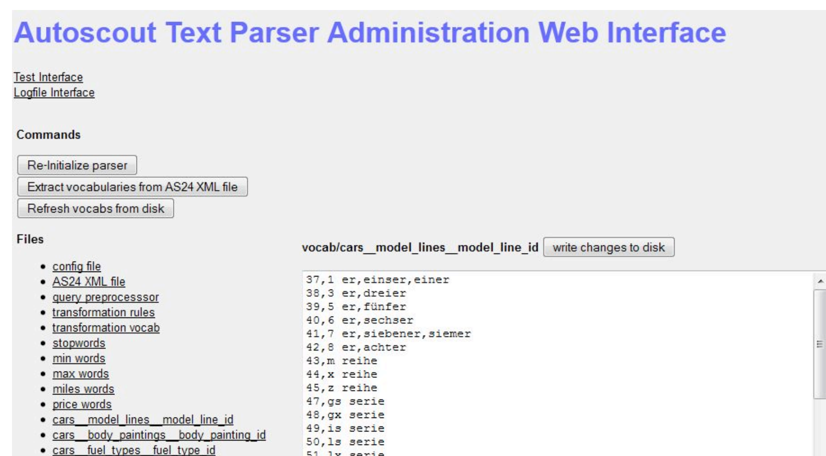

Abbildung 4 - Web view to maintain the parser.

The running system must be maintained by the customer, in this case AutoScout24, to update the vocabularies, for example when new models appear in the market. A web-based maintenance interface, depicted in Figure 4, is meant to be used by non-programmers to maintain and administer the system, especially the parser vocabularies. Instead of editing the vocabularies directly, pattern replacement rules should be edited that are used to extract the vocabulary from the AutoScout24 XML database dump. By this means a new database dump can be automatically used to generate new vocabulary versions.
Additionally, the administration interface offers access to the unit vocabularies used to detect numerical values, besides general configuration, for example timeout or local path values.
Besides the administration interface, the parser server offers two additional interfaces for testing and logging access.

6 Integration in the App

The App communicates with the server via a JSON format. The format is exactly the same as the one used by the web/graphical interface, which makes the integration to the search-engine component very simple. A history, or dialog state logic, is implemented in the App, i.e. the search terms can be stated by voice incrementally, If a model name, for example “*300*” is not unique because it exists for several brands, the choice might be made by the App logic based on a brand that was stated in a previous dialog step. The parser server itself does not handle dialog history. Session management is handled by the Apache Tomcat servlet engine.

The ASR delivers as a result an N-best list, i.e. the N most probable recognition hypothesis, ordered by probability. The server interprets them all, but prefers results that appear higher in the N-best stack.

7 Evaluation

7.1 Preliminary evaluation

Before actually starting the project, we evaluated two fundamental questions: firstly whether the automatic speech

Google ASR performance We generated random queries from the AutoScout24 vocabularies of form <model> <fuel> "with" <equipment>. 75 of these were spoken by one speaker to Google ASR (via the web interface in office). The result was counted between 0 (no recognition) to 3 (all three items recognized), which showed a 78.7 % recognition rate. We compared this result with a competing bidder of ASR mobile services and found it acceptable for our purpose. The big advantage is that no license fees must be paid for the service, which is mandatory for an App that is offered free of charge.

Text parser execution time The same test set was used and 10000 queries were processed in a Junit test. The execution time took 5.2 sec, which again is acceptable.

7.2 Main evaluation and tuning process

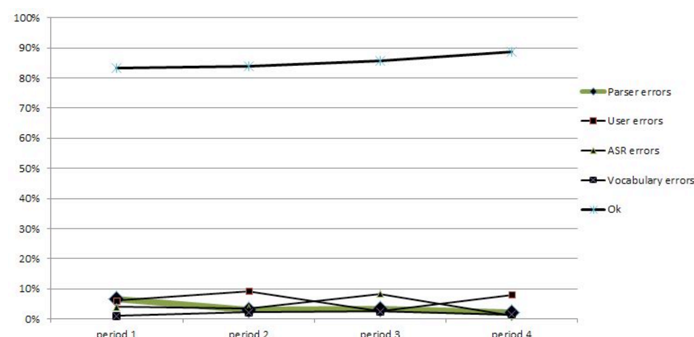

Abbildung 5 - Progression of different error types and success rate over four periods of measurement.

The app was evaluated during four test periods, each consisting of 7-14 continuous days, by offering the voice service each day to additional new 50 users of the App, selected randomly. We logged about only 50 queries daily, which means that many users only tried the voice service once, while many didn't try it at all. A reason for the low rate of usage might be that the voice interface of modern smart phones has not yet reached the sufficient level of acceptance for many users, something that will most probably change in the next years with the more widespread offering of voice recognition interfaces in mobile devices.

After each evaluation period, the log files containing the words that were recognized by the Google ASR service as well as the entities and numerical values that were matched by the parser, were analyzed manually, an error count was done and missing synonyms were added to the vocabulary. We distinguish four types of errors:

Vocabulary errors are errors that happen because this kind of feature is not yet supported by the interface to the AutoScout24 database and can thus not be handled by the parser, for example the search for model variants.

ASR errors are errors are such that the ASR output obviously is erroneous, an example would be the input "*www kombi schwarz facebook*", which very unlikely has been uttered by a user.

User errors means the user obviously didn't know what to say, an example would be the input "*wieso*" (*why*). Of course we cannot differentiate with certainty between ASR and User errors.

Parser errors are the remaining errors that could have been handled better by our parser, for example by adding synonyms or complexity to the semantic analysis.

As can be seen in Figure 5, there is an obvious trade-off between User- and ASR errors (depending on the decision of the manual annotation), the vocabulary error stays more or less constant but the Parser errors can be reduced by the tuning process. Therefore the success rate raises over the four periods from 83 % to 89 % whereas the Parser errors drop from 7 % to 2 %.
Of course it must be noted that the true success rate is unknown as we can not know what the User actually was asking for, we just assume that if some car related words were detected by the ASR and the parser was able to match all items against the AutoScout25 vocabulary, the transaction was successful.

8 Conclusions and Outlook

We described the extension of the AutoScout24 Android App, that offers second-hand cars, by a voice based search module. This module uses a server based text parser that matches user utterances against the AutoScout24 vocabulary and fills the App's menu selections accordingly. It currently runs in its pilot phase and will hopefully be released soon as an integral part of the App. The fact that not many users actually use voice search might be an indicator that voice services on smart phones are not yet fully accepted by the users.
Some issues remain to be investigated for future releases. Firstly, model names might be ambiguous, for example the model "300" can be of brand Mercedes or Chrysler. In the moment, the Apps interface only accepts one single model ID as a parse result. In future this should be extended in order to let the user choose between ambiguous models, when the brand was not stated explicitly.
With respect to lexicon tuning, automatic phonetic or orthographic similarity measures, perhaps based on the well known Levenshtein distance, might be used to increase the recall.
Furthermore, configurable lemmatization, might serve the same purpose, an example would be to add the plural "s" automatically.
The text parser might be released as open source in the future, currently we use it successfully to build a voice search service for television electronic program guide (EPG) management App. In the current implementation of the application to be used as a technical demonstrator, the text parser library is running on the client. In this context, we also investigate the utilization of linked open data, for example from DBPedia[1] [2], to enrich the semantic abilities of the parser. An example would be the search for movies starring a specific actor. The link between the actor's name and the movie title appearing in the EPG could be detected by DBPedia.

[1] http://dbpedia.org

Literatur

[1] BARNARD, E., J. SCHALKWYK, C. VAN HEERDEN und P. MORENO: *Voice search for development*. Proc. Interspeech, 2010.

[2] BIZER, C., J. LEHMANN, G. KOBILAROV, S. AUER, C. BECKER, R. CYGANIAK und S. HELLMANN: *DBpedia - A crystallization point for the Web of Data*. Web Semant., 7:154–165, September 2009.

[3] BURKHARDT, F. und J. ZHOU: *AskWiki: Shallow Semantic Processing to Query Wikipedia*. Proc. EUSIPCO, 2012.

[4] EMBLEY, D. W., D. M. CAMPBELL und R. D. SMITH: *Ontology-based extraction and structuring of information from data-rich unstructured documents*. Proceedings of the seventh international conference on Information and knowledge management, 1998.

[5] FORD, B.: *Parsing Expression Grammars: A Recognition Based Syntactic Foundation*. Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, 2004.

[6] JU, Y.-C. und T. PAEK: *A Voice Search Approach to Replying to SMS Messages in Automobiles*. Proc. Interspeech, 2009.

[7] SONG, Y.-I., Y.-Y. WANG, Y.-C. JU, M. SELTZER, I. TASHEV und A. ACERO: *Voice Search of Structured Media Data*. International Conference on Acoustics, Speech and Signal Processing, 2009.