Scripting Windows Behavior with Natural Language

Wesley Szamotula
University of Wisconsin
Madison, WI
wes.szamotula@gmail.com

ABSTRACT

Scripting tools for windows PCs have a lot of features for improving day to day activities on computers. This ranges from basic functionality such as launching websites with a hot-key to more complex functionality like automating form entry. However, to access these benefits users need to be familiar with programming. Many users don't have any programming experience, but would still benefit from these tools. By utilizing natural language processing, we can give users a way to access these features without any programming requirements.

In this paper I have developed a tool that takes a users English description of a script and generates the appropriate scripting code for that request. It currently supports scripting for key rebinding, launching applications with hot-keys, and launching websites with hot-keys. This tool utilizes a naive bayes classifier with bag-of-words to identify the type of script request. It then utilizes named entity recognition to identify the key scripting components in their request. The tool achieves a high accuracy, around 90%, for identifying individual key components, but a lower accuracy, around 60%, for fully generating correct scripts.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**;

KEYWORDS

Scripting, Program Synthesis, Natural Language

ACM Reference format:
Wesley Szamotula. 2017. Scripting Windows Behavior with Natural Language. In *Proceedings of Principles of Programming Language, UW Madison, May 2017 (CS704),* 5 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Scripting on a Windows PC offers users a way to improve their day to day activities. The functionality provided through scripting covers a wide range of basic and complex behaviors. At a simple level, users can launch common applications or websites through a key press. They can also bind a common phrase to a short keyword for something like an email signature. The more complex behavior extends quite far. Some examples include customizing clipboard behavior, automating form or spreadsheet data entry, and advanced web searches.

Use of these scripting tools requires some programming knowledge. Advanced behavior is often set up through general purpose programming languages and require a fair amount of knowledge. Simple behavior can often be set up without much programming, but still requires users to think about the relationship between their input and desired output. Many windows PC users are unfamiliar with programming languages, but would still be able to get a lot of value out of scripting.

Natural language processing offers a way to give novice users access to the advantages of scripting. This functionality lets us analyze the meaning and the structure of plain English statements. Novice users can simply describe the scripting behavior they want in English, and we can analyze the request using these tools. By identifying the correct elements and relationships we can attempt to generate the scripting code that meets their request.

I have developed a tool that translates English sentences describing simple scripts into the appropriate scripting code. I've analyzed the request using Natural Language Toolkit and generated code for AutoHotKey. The tool handles requests for rebinding keys, launching applications with hot-keys, and launching websites with hot-keys. I evaluated the success of these tools using a hand-built training and test set of example requests. It's accuracy is slightly lower than that from previous research, only able to achieve around 60% accuracy at script generation. Hopefully though some additional work similar results can be achieved in this area as well.

The format of this paper is as follows. In Sec 2. I review past research in this area. In Sec. 3 I discuss the open source tools used in this project. In Sec. 4 I provide details on how this tool was implemented and present the results in Sec. 5. In Sec. 6 I discuss how the interactive version of tool works.

2 PAST RESEARCH

There has been a lot of research into the area of program synthesis, with different research exploring different methods. Some research has looked into generation from natural language, others have looked into synthesis from input-output examples. The following is a sample of some of the work in this area.

2.1 Program Synthesis using Natural Language

In this paper Desai et. al [1] explored program synthesis for a few types of programs using natural language. Specifically, they focused on generating programs for air travel queries, text editing, and updates to tutoring systems. To make their approach extendable they constructed a framework for synthesizing programs for arbitrary domain specific languages.

CS704, May 2017, UW Madison

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

They developed an algorithm that generates a set of candidate programs for a natural language request. They also developed three different scores they used to rank candidate programs. A coverage score measures how much of the request was used to generate the program, a mapping score that measures how likely the program used the correct interpretation of the request, and a structure score that measures the general structure of the program.

In order to evaluate their tool they collected a large set of natural language to program pairs. They ranked the synthesized programs using a combination of their three scores and compared the results to the correct program mapping. With top-1 they achieved 80% accuracy and with top-3 they achieved 90% accuracy.

2.2 SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language

In this paper Le et. al [2] also researched program synthesis from natural language. They focused on scripting behavior on smartphones as their program domain. They designed a custom scripting language for this task that was broad enough to cover most basic requests, but also restrictive enough to make the search space reasonable. The program synthesis was then split into two main tasks.

First the request is analyzed using natural language processing techniques. The request is split into distinct chunks where each chunk represents either a component or relationship. The second step uses program synthesis techniques to generate the program. The identified components and relationships are constructed into data flow graphs where any missing elements are inferred. The graph constructed in this fashion is used to generate the program the user requested.

They evaluated their method with with examples pulled from tasks collected from smartphone help forums. When only using the first natural language step of their method they achieved about 60% accuracy. With the second program synthesis step used as well accuracy was boosted to about 90%.

2.3 Automating String Processing in Spreadsheets Using Input-Output Examples

In this paper Gulwani [3] explored program synthesis using input-output examples rather than natural language. This algorithm has gone on to be part of the feature Flash Fill in Excel. He focused on handling a wide variety of string manipulation tasks users perform in Excel. Using a new string programming language he designed an algorithm that generates programs from several input-output examples. The algorithm was specifically designed to be fast, require only a few iterations, and handle small mistakes in the examples.

The algorithm first computes all the traces in their language that map from each input to each output. This can be a huge set of expressions so a special data set is used to represent and manipulate this data. Examples are than partitioned so they are handled by the same top level conditionals. Using a greedy heuristic the number of these partitions is minimized. Finally a classification scheme is learned to place inputs into the appropriate partitions.

The evaluation of algorithm looked across several different metrics. The algorithmic performance was measured to see how quickly the program synthesis could be performed. On average, the tool took less than 0.1 seconds to complete. The number of interactive rounds was also measured to see how well the conditional learning and ranking of the algorithm performed. The tool was typically able to generate results after 1 round of inputs, occasionally requiring up to 4 rounds of interaction. Finally they measured the success ratio of the algorithm. It was able to correctly handle almost every problem, but there were a few that could not be expressed in the language defined. Overall the tool showed a lot of promise in the measures of interest.

3 OPEN SOURCE TOOLS

In order to create this tool I took advantage of few existing open source tools. Specifically, I used Natural Language Toolkit to perform natural language processing, scikit-learn to perform machine learning, and AutoHotKey to run the generated scripts.

3.1 Natural Language Toolkit

Natural Language Toolkit is an open source python library for analyzing natural language. At the basic level it analyzes that context words are used in to assign them tags. The most common is part of speech, but this can be extended to general customized tags. One form of these customized tags is named entity recognition. This is done through learning a classifier that assigns special tags to key words or phrases in sentences. The structure of sentences can also be analyzed by breaking sentences into significant chunks and storing them in a tree. At a higher level larger bodies of text can be classified with different kinds of labels.

3.2 Scikit-Learn

Scikit-learn is an open source python library of machine learning methods. It covers a wide range of machine learning applications like classification, regression, and clustering to name a few. It has implementations of many common algorithms. For this project, the multinomial naive bayes implementation was used.

3.3 AutoHotKey

AutoHotKey is an open source scripting language for Windows that allows users to automate some desktop tasks. It has a flexible syntax which can be used to build scripts ranging from simple to complex. Tasks such as binding a key to open an application can be completed with a single line of code. More complex task can be developed taking advantage of its general purpose programming language.

4 METHOD

The generation of an AutoHotKey script from the users request is broken into two main steps. The type of script is identified, then key components of that request are identified. Once we have the script type and key components it is simple to generate the appropriate scripting code. Currently the tool supports three types of scripts - rebinding keys, launching applications with hot-keys, and launching websites with hot-keys.

4.1 Identifying Type of Script Request

In order to identify what type of script was being requested, I utilized a multinomial naive Bayes classifier. This is a common naive

Bayes variant for text classification. The data is typically represented as word counts, but weighted tf-idf vectors often work well in practice. The probability is calculated with maximum likelihood estimates using the equation below. The value N_{yi} is the number of times a feature i appears in the class y, and the value N_y is the total count of features for class y.

$$P(x_i|y) = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

The request are converted into features using a bag-of-words representation. This simply breaks each request down to a vector of the counts of each word. In order to reduce the impact of uninformative words, the vectors are weighted using term frequency-inverse document frequency.

$$tf, idf(term, doc) = tf(term, doc)\ x\ idf(term)$$

$$idf(term) = log\frac{n_d}{1 + df(docs, term)}$$

The term frequency-inverse document frequency re-weights each word by multiplying the frequency it appears in the current document by the inverse log frequency that it appears in all documents. In this case, a document refers to a single script request. The resulting weighted vectors are normalized with the Euclidean norm. This reduces the impact of words that appear across all requests, and increases the impact of words that appear in single requests.

4.2 Identifying Key Components of Request

Once the type of script request is identified, the key components of the script need to be identified. For example, when launching a website the key components are the keyboard button, possible modifiers (like the control or windows key), and the site to launch. In order to find this information a named entity recognition tagger was learned for each type of script request.

The named entity recognition tagger is built on top of a naive Bayes classifier. The requests come into the tagger tokenized by word where each word has the part of speech assigned. The list of unlabeled tokens are handed to the classifier and labels are predicted sequentially. Each token is broken down into a set of features that depend on the other tokens in the sequence.

The following set of features below were utilized in this classifier. One of the features utilized is the lemma of each word. Lemmatization is the process of breaking a word down into its root form. For example, someone could describe the press of a keyboard button with pressing or pressed instead. The lemmatization process will break these words all down into a single word. Once each request is broken down into these features a classifier is learned to identify the named entity of a word in a request.

4.2.1 *Features Utilized by Named Entity Recognition Classifier.*

- Current word
- Two previous words
- Two next words
- Part of speech of each word
- Named entity of previous word
- Lemma of each word
- Few features for capitalization and punctuation

Table 1: Accuracy of Named Entity Recognition Tagging

Script Type	NER Accuracy	Script Accuracy
Key Remapping	92%	66%
Application Launching	95%	73%
Website Launching	89%	55%

5 RESULTS

5.1 Data-set

In order to train and evaluate the classifiers needed for this tool, a data set of requests and their correctly tagged components was assembled by hand. I started by browsing the AutoHotKey forums, wiki, and subreddit for examples of users requesting or describing simple scripts. This gave me a few initial examples, but not enough to train the classifiers. I added some additional hand-made examples to get more data to work with. In the end I had 40 examples of each type of script request.

5.2 Identifying Type of Script Request

To evaluate the success of identifying the type of script request, I took the combined 120 examples and their respective types as the training and test set. I used 10-fold cross validation with the Naive Bayes script type classifier to measure the accuracy. Across the 10 trials this classifier had an average of 90% accuracy for predicting the type of classifier.

5.3 Identifying Key Components of Scripts

To evaluate the effectiveness of predicting the key components of scripts, I tested the named entity recognition tagger for each script type separately. To prepare the data set I broke each example request into the tokenized words and assigned the part of speech for each word. I then manually entered the correct named entity tag for each word in each request.

Since I only had 40 example for each type of script, I used leave-1-out cross fold validation when evaluating the classifiers. Over 40 iterations I trained the classifier on 39 examples and evaluated the result on the left out example. The named entity accuracy is the percentage of named entities that were correctly identified across all requests. The script accuracy is the percentage of requests where every named entity was correctly identified. The results are presented in Table 1.

The accuracy of predicting named entities is very high, around 90%. However, the accuracy of predicting scripts was quite a bit lower, around 60%. Predicting the whole script is much more difficult, because every single word in a request needs to be assigned the correct named entity to correctly identify the script. There were a couple common errors that made this accuracy low. Sometimes an additional word around a key component would also get assigned to that named entity. Other times, one of the key components wouldn't get identified through the named entity.

6 INTERACTIVE IMPLEMENTATION

6.1 Text Based Tool

To support interaction with end users I created a text-based version of the tool. I first learn the script type classifier and all named entity recognition taggers using all of the examples from my data set. I prompt the user for a script request and then begin the analysis.

First I break the request into the weighted bag-of-words vector using the weighting learned from examples. I pass this request to the naive bayes classifier and predict the type of script request. I then break the request into its word tokenization and tag each word with its part of speech. This tokenized version of the request is passed to the appropriate named entity recognition tagger and the words are tagged with their named entities.

The final step is to extract the key elements from the sentence. I loop through the tagged tokens and pull out the key components. An additional step needs to be taken for launching applications. For a few default windows application, the name alone is enough to launch it. However, for most application the full file path is required. The user is shown a windows explorer prompt in these cases to select the full file path for the application.

In AutoHotKey each modifier (such as control, alt, or windows) has a special character associated with it. When I have collected the modifier from the sentence, I transform it into the appropriate special character. Once I have these elements I generate the appropriate script and return it to the user, along with a plain text description of what it does.

6.2 Example Use Case

The following is an example of how an script request is processed by this tool. The end user view can be seen in figure 1.

(1) First the user enters the following request:
"Open google.com when I press windows g"

(2) The input is converted into a weighted bag of words vector and passed to the naive bayes type classifier. The script type is identified as Launch Website.

(3) Using built in functionality from Natural Language Toolkit the sentence is split into its word tokenization with part of speech:
(Open, NNP) (google.com, NN) (when, WRB) (I, NN) (press, NN) (windows, NNS) (g, NN)

(4) Using the named entity recognition tagger for launching websites the key components are identified:
(Open, NNP, O) (google.com, NN, site) (when, WRB, O) (I, NN, O) (press, NN, O) (windows, NNS, mod) (g, NN, key)

(5) The script is generated from the key components:
google.com, windows, g ⇒ #g::Run google.com

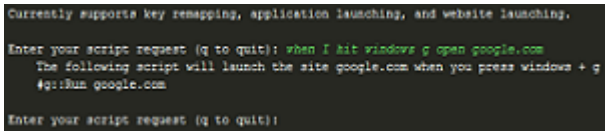

Figure 1: Example of a script generated through the interactive text implementation.

7 FUTURE DIRECTIONS

There are a couple main future directions I would like to take this research going forward. First, I would like to focus on enhancing the accuracy of the current script generation. Second, I would like to extend the tool to additional types of scripting behavior.

7.1 Enhance Current Script Results

The accuracy of this tool is currently similar to what the work from Le et. al in *SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language* [2] achieved when they only used natural language processing. They were able to boost their accuracy up to 90% through additionally using program synthesis techniques. By adding some more steps to the current process I may be able to achieve similar results in this domain.

After the initial prediction of key elements in a script request, some steps could be taken to improve results. In some of the failure cases, the named entity recognition tagger tags multiple elements as a single key element and other times it doesn't tag anything for one of the key elements. In these situations some interaction with the user could be used to resolve the issue. A user could be prompted with a couple suggestions and asked to choose which one is actually correct.

The tool may also be able to infer some of the missing elements when errors like these occur. When multiple elements get tagged for a single named entity, we could take the one with higher probability as the true value. When no element is selected as an important named entity, we could take another pass looking specifically for that entity.

Additionally extending the current data set could help improve results. Since I added most of the examples by hand, they may not accurately represent the way a user would request the script. Collecting additional examples of scripting requests from users would provide a more representative sample.

7.2 Extend to Additional Scripting Types

The tool currently only supports generating scripts for simple applications. Rebinding keys, launching applications with hot-keys, and launching websites with hot-keys. A future step would be to extend this to additional types of scripting behavior.

A few directions to explore next are clipboard management, more advanced web searches, and combining existing types of scripts together. Being able to link a few different types of scripts would provide the tool with a lot of additional flexibility. For example, results from web searches could be added to clipboards or used to trigger additional site launching.

8 CONCLUSION

Scripting on a windows PC can give a user a lot of useful functionality. Basic scripts, like launching applications with hot-keys, can

help speed up some day to day activities. More advanced functionality, like automating aspects of form entry, can give users a wide range of benefits. However, users need to have some familiarity with programming languages to take advantages of these benefits.

By utilizing natural language processing we can give novice users access to scripting behavior. To accomplish this I took advantage of a tool known as named entity recognition tagging. This process assigns tags to key words in a sentence using a naive bayes classifier. This lets us identify the components of a script and generate the appropriate code for a user.

These classifiers were trained to generate scripting code for key rebinding, launching applications with hot-keys, and launching websites with hot-keys. The accuracy of predicting named entities was high, around 90%, but the accuracy of predicting scripts was quite a bit lower, only around 60%. Predicting a whole script is much more difficult, because every named entity in a request needs to be identified correctly in order to predict the script.

Overall, the accuracy of this tool is a bit lower than the accuracy achieved in past methods. However, It is similar to results achieved by other methods prior to the addition of more program synthesis techniques. By extending this method with similar additional techniques, like prompting users for feedback or inferring missing relations, we may be able to achieve similar results.

REFERENCES

[1] Desai, A & Gulwani, S & Hingorami, V & Jain, N & Karkare, A & Marron, M & Roy, S *Program Synthesis using Natural Language.* CoRR 2015
[2] Le, V & Gulwani, S & Su, Z *SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language* MobiSys 2013
[3] Gulwani, S *Automating String Processing in Spreadsheets Using Input-Output Examples* PoPL 2011
[4] AutoHotKey *https://autohotkey.com/*
[5] Natural Language Toolkit *http://www.nltk.org/*
[6] Scikit-Learn *http://scikit-learn.org/stable/*