Julius Information Extractor

Kyle Woodward and Lee-Ming Zen

June 7, 2006

1 Introduction

1.1 Intent and use

The Julius Information Extractor is a system that uses user input to derive a set of rules which are then employed to find and delineate tokens similar to the user’s input. Julius leverages a simple and intuitive GUI design to provide a system for feature extraction over a given corpus. Moreover, this system is unique in that it provides a vehicle by which arbitrary classes may be recognized; where traditional information extractors may be tuned to domain-specific applications, Julius allows for any arbitrary type of feature to be selected and sought. As a rule, we have attempted to develop Julius as free from bias as possible, using a minimum of assumptions to develop the extractor; while this method has obvious practical advantages, there are in contrast some very practical negative effects which are derived from a general lack of human tuning.

1.2 Precedent

Julius follows most directly from the AT&T text extraction and classification system developed by Michael Collins and Yoram Singer [1]. Their paper provides a rough sketch of their system, though many of the important details have been left out. We interpret these various omissions in our own ways, which we detail below. Overall, the AT&T system splits rules up into two types: spelling and contextual rules. It then iterates of these sets of rules separately in order to label and derive new rules. They begin with a set of spelling rules, which are used to label the data. These new labelings are used in turn to induce new contextual rules which are then used to label the data. This new labeling in turns induces new spelling rules and the cycle continues until they have achieved a desired number of rules at which point they complete the process by a final labeling of the data.
2 Methodology

2.1 Algorithmic structure

Firstly, we address the general structure of the program. This is followed below by a more in-depth description of implementation decisions and the like.

1. user adds and tags documents
2. while set of rules is evolving do
3. for all documents in supplied corpus do
4. derive window bounds
5. derive contextual features using existing tags
6. select new tags using contextual features
7. derive window bounds
8. derive spelling features using existing tags
9. select new tags using spelling features
10. end for
11. end while

As you can see, we take a very similar approach to that used in the AT&T system, in that we split our rules into contextual and spelling rules. We begin by deriving new contextual rules given the tags. In other words, using the existing spelling rules which the user has provided, we derive new contextual rules. Following this, we label the data using the contextual features. Using this new labeling, we derive new spelling rules, which in turn, we use to label the data. The loop continues thusly until we achieve a certain criteria before finishing. Clearly, this is very similar to the AT&T system. Note however, that we must derive window bounds in our process in order to slide across the data appropriately in order to estimate how much of the data to look at. In the AT&T paper, they use a parser to determine the window since they are interested only in proper nouns proper noun phrases. However, we cannot make such an assumption. We describe the various components of our system in detail below.

2.2 GUI tools

We provide a set of useful graphical interfaces for working with Julius. There is a tagger to allow the user to tag documents for the system. In addition, we have incorporated a viewer for much of the statistical information related to the rules.

User input

Documents are added from the file system or are loaded from URLs (though the latter is not recommended), and are tagged via text selection in a separate pane. The system is highly intuitive and effective. The user can move documents into and out of the set of documents of interest. For any document, the user simply
2.3 Implementation details

Evolutionary criterion

As the set of rules is growing over time (see below), there must be some exit condition upon which adaptation ceases. Ideally, the iterative procedure exits when the number of rules derived fails to represent a certain percentage of the total rule set. However, as a practical matter we have limited the system to a fixed number of iterations. It should be noted that AT&T applies a similar method (allowing no more than 500 iterations), so this less adaptive method is well-supported.

Window selection

An integral aspect of this algorithm is selection of a proper viewing range for token extraction; when the extractor runs through the document, it requires some notion of which substrings present viable extractions. Letting $\mu$ be the (double) mean and $\sigma$ be the (double) standard deviation, testable window sizes for the extractor are restricted to

$$\lfloor \mu - f(\sigma) \rfloor \leq l_{\text{window}} \leq \lceil \mu + f(\sigma) \rceil$$

where $f$ is a function which effectively smooths $\sigma$, so that $\sigma = 0$ still allows some variability in window size. Ideally, this function would be sublinear at high values, and superlinear for low values, thereby allowing small windows to maintain variability while not overestimating the window sizes in sets with larger tags. In the given implementation, $f(x) = e^{0.5x}$; while this function does not meet the sublinear criterion mentioned above, it does allow training sets with shorter tag lengths to expand for more potential matches.

This sizing is applied when possible tags are assigned probabilities prior to extraction. A fixed prefix is selected, from which potential tags of varying
window sizes are derived; following this, a suffix is attached at the end of each window. From this method, a maximum likelihood estimate is obtained.

**Deriving new rules**

By tagging a set of data, with a set of rules, we may generate new labels which were not present in the data before. Using the new labels along with the original labels, we calculate counts for the occurrences of various features. Using this information, we are able to calculate the probability score for a particular set of features; using the probability measure derived for using undiscovered elements in the document set, we select the $n$ features which best differentiate between the class and the anticlass. Importantly, $n$ increases with each iteration, just as in AT&T.

**Label application**

New tags are found and labelled using a method nearly exactly that of AT&T. It is a method which we like to call adaptive tag-normalized inference; put simply, the priors from which features are derived are updated before label selection in each iteration, immediately following rule selection. This allows the general form of selected tags to follow. Probabilities are normalized in the usual manner, following from the generic anticlass principle discussed below. Tags which maximize probability (which are in a certain percentile) are then selected as valid for use in future iterations.

### 2.4 Innovation

**Generality**

One of the interesting aspects of our system is how general purpose it is. This is both a strength and weakness of our approach. Many other systems such as the AT&T system or KnowItAll [2] rely on having some type of domain knowledge. In the AT&T system, the system assumes there are three types of things it is interested in: people, organizations, and locations. While KnowItAll is technically domain-independent and claims to be so, it requires input of a “focus” which is essentially a set of classes and relations describing the type of information the system ought to be interested in.

Julius, on the other hand, attempts to be more general purpose and does not require any type of domain-related input aside from the examples that the user has tagged. In this sense, we are dependent upon the domain somewhat because one could claim that the user has explicitly shown examples of the domain to the system. However, we still believe this is more domain-independent than a system such as KnowItAll where while the system must extract its own domain-related information, it is provided an initial schema for the domain of interest. On the other hand, Julius simply receives a few examples.
Generic anticlass

A distinct problem in the provided system of user-tagging is the lack of a separate “anticlass” to complement the provided tagged class. As is usual in the case of text classification, simply maximizing \( p_{\text{class}}(\text{token}) \) may lead to nonoptimal selection (e.g., tokens which in general have a high rate of occurrence); thus we make use of an anticlass to produce reasonable conditional probability estimates. In this vein, we take the entire document and its comprising windows as members of the anticlass; particularly, each tag the user has supplied is taken as a member of the anticlass, in some sense as evidence against itself. While not entirely necessary, this inclusion greatly reduces worst-case computation time. Moreover, while the transformation from empirical probability to this new measure is certainly nonlinear, it is monotonic. That is, let \( n, m \) be counts of features within the class, while \( p, q \) are counts without. Suppose that in terms of the usual empirical probability, we have

\[
\frac{n}{p+n} > \frac{m}{q+m}.
\]

Then it is directly implied that \( nq > mp \). Now suppose we take the class as evidence against itself, by taking supplied tags as part of the anticlass. Then the denominators increase to \( p + 2n \) and \( q + 2m \), respectively. We are then left with the comparison

\[
\frac{n}{p+2n} > \frac{m}{q+2m}
\]

which holds precisely when \( nq > mp \), or when \( \frac{n}{p+n} > \frac{m}{q+m} \). Thus even though this new probability measure is not accurate in number, its monotonicity ensures that so long as we are only using this measure to select features which maximize probability, evidence against itself in the anticlass does not cause a drop in the likelihood of something occurring. Importantly, this mathematical fact allows Julius to operate in feature selection without having to determine whether or not a derived window corresponds to a known tag; thus this allows a notable increase in computational efficiency.

Prefix/suffix separation

In the AT&T treatment of extraction, context comes as a pair of context and context-type, where the former is some concept surrounding the token in question and the latter is the part of speech that describes the relation the context has to the data of interest. In our system, the prefix and suffix of a tag are treated as separate elements of context. This has proved to be quite useful. As an example, assume we have as input the string

\[
\ldots ABCABCCBACBACBAABCCBA\ldots
\]

but want only to recognize \( B \) such that \( ABC \), while not \( B \) such that \( CBA \). Then if prefixes and suffixes are treated identically, there is no method by which any
algorithm may differentiate between these two cases. It is apparent that differentiating context by prefix and suffix will allow a program to properly select. We presume that this example is not canonical, and indeed it might be applied to differentiate between St Louis, a city, and Louis St, a road.

The inspiration for this came from Brin’s 1998 paper on web extraction [3]; in this, he proposes fixed-length prefixes and suffixes. While he measures in characters, it seems more sensible to measure in tokens. A fixed window nonetheless provides an excellent static frame of reference by which potential tags may be compared.

2.5 More context

However, in the AT&T system the use of part-of-speech seems to be a very useful piece of data for providing additional knowledge in the task. Julius uses the Stanford POS tagger to tag all the words in the document, and uses these tags as additional features. Initially, we attempted to use the Stanford parser; however, we judged this not to be feasible as it makes the system less general: in particular, we want to allow users to be able to select any arbitrary piece of text, and to attempt to extract any information the user might be interested in. If this information happens to lie across sentence boundaries, it is unclear as to how Julius ought to combine the parse trees of these sentences and how we ought to decide which part of the tree constitutes the “context” for the information we are interested in. Even the case in which a tag does not span two sentences this is difficult to handle; for example, if the information spans multiple subtrees, should we use the label of the most common ancestor tree? If so, what if this ancestor tree is very high up in the parse? Lastly, there is no clear solution as to what to provide for the actual notion of “context.” AT&T uses the parent label of the tree in which the tag lies in the parse tree, but this is very clearly a special case; more general handling would not stand up to such heuristics. Thus, we fall back on the part of speech tagger; while it does not provide as much information as a full parse, it does provide a good feature. In addition, because the part-of-speech tagger is faster than the parser, we have made a decent tradeoff in terms of information to performance. As a result, we believe we have found a marked improvement upon the method of Collins and Singer, by including both context around the data in question while continuing to augment the data via part-of-speech.

2.6 Search engine contextualization

Borrowing heavily from the KnowItAll project at the University of Washington, Julius performs additional contextual feature selection by querying a search engine (currently, Google) and extracting context from the results page. The benefit of this system is that contexts which may not have been located by the user – and which might potentially escape detection during spelling labeling – might possibly be derived from the contextual results provided on most modern
3 RESULTS

Experimentation revealed this addition to muddle returned tags. Although this need not be the case – for example, we might do well to downweight search query features – in general we did not find that inclusion helped to any great extent. Certainly there are contexts in which it might serve a better purpose (suppose that the user would prefer to specify only a minimal number of tags, which he knows to be relatively common in usage), but we believe this element to be better left as an option for the user to toggle; in our current context, it is better left completely unused.

3 Results

3.1 Evaluation Criteria

It is difficult to evaluate how good an extraction is. For example, if we extracted part of a phrase that should have been extracted, should this be considered a success or partial success? If so, how much so? We decided to use a simple metric via counting the number of characters accurately retrieved. Thus, if a piece of information lies at range 113-120 and Julius returns with a range of 114-120 then we have retrieved 6 out of 7. From this we emulate recall, precision, and F1.

3.2 GENIA corpus tagging

The GENIA corpus is a set of microbiology article abstracts. From these, we located items denoted DNA_domain_or_region in 50 articles, and attempted to derive these elements in 50 more documents from the corpus. Recall in this case was a paltry $\frac{117}{2075} \approx 5.64\%$, while precision was even worse, at $\frac{117}{3815} \approx 3.06\%$. Although this certainly appears to be an abject failure, there are two somewhat mitigating factors. The first is that this set is extremely heterogeneous, providing Julius with few reliable features on which to select. Secondly, this performance is still twice that of a random selector, even constrained to act upon specifically-measured windows. So while we are not proud of this result, it is possibly not the best indication of the abilities of the program.

3.3 Reuters corpus tagging

We also ran Julius against a Reuters corpus describing mergers and acquisitions in newspaper articles. We focused on attempting to extract one class of entities involved in mergers and acquisitions (the buyer). We ended with similarly poor results yielding $\frac{128}{10208} \approx 1.25\%$ recall and precision of $\frac{128}{6264} \approx 3.92\%$. Again, while not the greatest numbers, there are some interesting things to note here. First, we do slightly better than a random selector, again, even constrained to act upon specifically-measured windows. Unfortunately, the headlines in the corpus are all capitalized and often contain company names resulting in many
false positives due to the fact that Julius learns a pattern for seeing the buyer as a capitalized name in the headline.

3.4 Qualitative Results

While our quantitative results are less than enthralling, we believe that Julius shows some very promising general trends.

(1) In both tests, Julius performed significantly better than a machine which simply randomly guessed characters – even one which uses adaptive window sizing as we have integrated into our system. This is a strong indication that learning is actually occurring.

(2) Tests on smaller sets indicate a potential F1 of 0.7 or higher. Likely, this is due to the fact that these smaller sets were much denser in tags, and so suffered less interference than the larger corpora on which we tested. Nonetheless, Julius showed a remarkable ability to extract dates and addresses, even prior to part-of-speech tagging. Notably, in these small cases it was sometimes advantageous to utilize search engine contextualization. These training sets have not been included simply because it is difficult to assess whether or not good performance on them actually means anything; however, it is an excellent qualitative sign that Julius performs so well on these.

(3) Julius operates on much smaller training sets than many similar programs. While general extractors require significant training to recognize small portions of text, Julius is effective (on some sets) after being trained on as few as $\frac{1}{6}$ of all occurrences. This indicates a strong capacity for learning, as well as an indication that larger training sets should have been tested. However, larger training sets seem, to us, to go against the vein of promoting user-tagging, and so were not tested.

(4) Hand-checking the larger corpora showed that Julius makes consistent mistakes. A system which is consistent may usually have its consistency tuned one way or the other, in time.

(5) Lastly, our quantitative measure of success may be overly harsh. That is, if Julius tags “</ul>” as opposed to “ul” (the latter in the same context as an HTML tag), it is still highly accurate. Unfortunately, our measure counts significantly against Julius for such errors; additionally, Julius utilizes word boundary restrictions on tagging (a loss-of-generality decision made to improve efficiency), so often it is not even possible to select the appropriate tag exactly, and approximation is encountered.
4 Discussion

4.1 Training technique

One serious development which was considered was placing another classifier on top of the rules, so that we might have a better way or understanding of how to weight the rules (e.g., MaxEnt). However, we believe that via the updating and re-normalization of the rules during the iterative process, we already have a simple and efficient way that alters the probability values for the rules. In addition, basic intuition supports this theory, as rules with more relevance will gain probability over time, while those that are not as useful will not pick up as much probability in future iterations. Ideally, this would have been rigorously shown (or disproved); due to time limitations, we have been unable to do so.

4.2 Performance

Ultimately, we were quite disappointed in the performance of our system. Despite various attempts at altering the weightings, thresholding for rules, smoothing, and other parameters, we had trouble improving the system. Looking at the numbers in the raw, these are incredibly low and in comparison to the other systems we have mentioned thus far, they are abysmal. Most of those other systems score quite high with the AT&T system performing better than 90% accuracy. However, we have a model which is far less constrained with many fewer assumptions than the other systems. The AT&T system explicitly seeks out proper nouns and attempts to classify them into one of three classes. We on the other hand must account for the fact that the information the user is interested in may be any type of part of speech with any possible structure or data. In addition, their seed rules are based upon actual rules in the system such as “Mr. always indicates a name” whereas we rely solely on seeding our system with examples.

Much in the same way that the AT&T system has seed rules, KnowItAll uses seed classes and relations to help the system build up knowledge about the information it is collecting. Julius has no such metadata and thus cannot reason in such a manner. From an accuracy measure standpoint, this is a failure; however, again taking generality as our motto this method seems the most open and available.

5 Further development

5.1 Bug fixing

One of the key errors in the program is that Julius is quite obviously too eager to select text. That is, when “abcdef” should be selected, it is often the case that “abcdefg” is the result. Given the iterative nature of this process,
it is not hard to understand how such a simple error can, if made in the ini-
tial steps, contribute to the entire system running awry. Currently, Julius is
written to attempt to equally weight tokens of all lengths (via geometric mean
of multiplicative feature probabilities as in the treatment of part-of-speech tag
probabilities); some amount of downweighting might be desirable, but it remains
to be seen whether tokens should be downweighted proportional to length, or
rather as a continuous decay from mean window size.

While in structure it appears that Julius’s probability generation functions work
properly, the empirical results suggest otherwise. Further work would necessarily
involve better smoothing and more accurate conditional probability esti-
mates. We are open to the idea that Julius is written properly and responds
properly, but simply is not being given “good” training data, but this seems
to be a slight cop-out. Evidence might lend some credence to this theory –
Julius performs remarkably well on more homogeneous data sets, and not very
well at all on more heterogeneous ones – but in the situations in which Julius
is effective, it is often just as simple to write a basic regular expression to parse
the data out; thus to maintain the utility of Julius, action must be taken to
improve performance on wider data sets.

5.2 Limitations

Julius sometimes has issues dealing with punctuation and as previously men-
tioned, subparts of entire tokens. In particular, we are interested in extracting
meaningful token phrases whereas sometimes the region of interest begins or
ends in the middle of a token. Therefore, Julius is not well-suited for extracting
such data or certain evaluation criteria must be relaxed.

As for punctuation, due to the variety of uses for any particular symbol, it is
often hard to differentiate between the ways in which it is used. As a result, we
see the symbol in all sorts of contexts which then induces further labels resulting
in errors which continue to propagate.

5.3 Potential improvement

As a system, Julius could improve the most by offering tagging of multiple
classes. That is, rather than simply restricting user tagging to the binary options
“tagged” and “untagged”, allow specification of several genres. In application,
this gives the user the power to perform multiple searches without having to
rerun the program. For example, a corpus might be queried for contact informa-
tion, extracting both names and e-mail addresses; while in the current system,
this is best done as two separate operations, the option to tag multiple classes
presents the user with the ability to do this in one fell swoop. Additionally, this
feature could be implemented with coordination between the tags, so that upon
extraction (keeping in mind the previous example) each name is associated with
an e-mail address, and vice-versa.
Tying into this notion, Julius might be further adapted as a document-to-XML converter. Leveraging a possible capacity for multi-class extraction, a set of news articles could be reduced to headlines and papers in which they appear, ready to be loaded into an RSS feed. We believe that tools such as this (albeit more effective ones) are essential if XML is to be the grand solution that it is often claimed to be.

In addition, as mentioned previously, using a parser instead of just the part of speech tagger ought to provide more information and yield better results. This, however, results in hard decisions which possibly affect the general ambiguity of the system (reference our previous discussion). Serious thought and experimentation might yield results which clearly direct these choices in one way or the other, but the time scale of this project is not such that this is a feasible option.

References

