Modular Language Processing framework for Lightweight Applications (MLPLA)

Adrian Zafiu[1], Stefan Daniel Dumitrescu[2], Tiberiu Boroș[2]

[1]Startapp Technologies
[2] Romanian Academy, Center for Artificial Intelligence (RACAI)
Bucharest, Romania
{adrian}@i-neo.ro, {sdumitrescu, tibi}@racai.ro

Abstract

Mobile devices are becoming ubiquitous. All around us, they are used for more and more tasks, from leisure to work, and, through their applications are becoming faster and smarter. However, they are still limited in terms of hardware resources (memory and CPU time), usually depending on remote servers and uninterrupted data connections to perform operations that are not feasible to be processed on the device. We present a framework that provides a series of Natural Language Processing tools, essential to build smarter applications (from sound synthesis to Artificial Intelligence enabled apps). Our framework, named Modular Language Processing for Lightweight Applications (MLPLA) is designed to run under resource-strict conditions while providing state-of-the-art results. We present a sentence splitter, a tokenizer, a syllabifier, a chunker a part-of-speech-tagger, a lemmatizer, a letter-to-sound processor and a lexical stress predictor.

Keywords: natural language processing, text-to-speech synthesis, mobile devices, non-fixed processing pipeline

1. Introduction

Currently mobile devices are starting to replace traditional desktop computers mainly because they offer increased freedom to the user. Thus, a strive for creating smaller and computationally powerful devices with increased autonomy and liability can be easily observed in the last couple of years. On the software side, more and more producers have turned to this very focused sector, bringing a high number of applications which, for many sectors, deliver at least the same functionality as their desktop counterparts. Furthermore, these devices are equipped with advanced cameras, sensors and microphones which allow development of full haptic user interfaces.

One of the well-established research areas (commonly referred to as Human Computer Interfaces or HCI) focuses on delivering natural user-interfaces by employing technologies such as automatic speech recognition (ASR) and text-to-speech (TTS) in applications where the input/output between humans and computers is governed by systems that are capable of performing natural language understanding (NLU). Besides the Question Answering (QA) and open-dialog systems developed by the research community (i.e. START QA or Deep Tutor etc.), Google Now, Siri and Microsoft Cortana have found their way into our daily lives, which shows (a) that large software companies have realized the commercial potential of natural assistive agents and (b) that current state-of-the art ASR and TTS systems have reached a state where they can provide a reliable accuracy in real-life scenarios.

In the past, the hardware performance of mobile devices prohibited the development of systems that linked technologies such as TTS, ASR and NLU in a single device without requiring Internet access in order to perform computationally expensive processes on a remote server. However, the high interest shown toward mobile platforms have led to high technological advances and enabled ARM-based platforms to outperform high-end desktop platforms that existed a decade ago and even compare to today's mid-level x86_64 devices.

Unfortunately, current mobile-enabled HCI technologies still depend on Internet access and employ dedicated servers to perform high computational tasks, which is not bad for the power autonomy of embedded devices but shows some disadvantages such as: (a) high speed Internet connection is not always available – bad network coverage in some areas and additional network charges under certain conditions; (b) security issues related to sending and receiving data over the Internet; (c) delays due to slow networks translate directly into slow application response times and a bad user experience.

Therefore, many HCI applications would benefit from a framework that is able to carry out NLP and DSP tasks in a low resourced environment. In what follows we will introduce a newly created framework which we refer as Modular Language Processing framework for Lightweight Applications (MLPLA), describing its general architecture, introducing a basic NLP methodology that is applicable to multiple languages and we will present our preliminary results.

2. The MLPLA framework

Building such a framework poses a series of challenges both from the software and scientific point of view. First of all, a framework like this must be built with emphasis on the fact that NLP tasks are interdependent and the order in which they must be performed on a target sentence depends directly on the language and indirectly on the availability and quality of training corpora and on the complexity of the task itself. While the language related dependency is obvious, the later mentioned dependency is not straight forward. In order to clarify we will take the example of syllabification and letter-to-sound (LTS) conversion. The two tasks are in what can be regarded as an “Ouroboros” dependency. If one would be

able to perfectly perform phonetic transcription he would also be able to use allophone information and perfectly syllabify words that don't qualify for special etymological-related rules. Symmetrically, syllable information would greatly improve the quality of automatic LTS since, in most cases, choosing an allophone for a certain phoneme depends on that phoneme being able to sustain a syllable or not. Thus, choosing which task to perform first and if the result of that task should be used as features for the other tasks depends on its complexity and on the available training corpora. Needless to say that, in many cases, treating each task independently and using only lexical level features yields better results than cascading them. However, a NLP framework should offer the possibility to create any possible processing chain (even perform a task more than once in special conditions) since not being able to do so would create an unnecessary bottleneck for some applications.

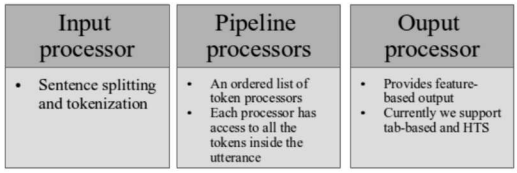

Figure 1 - The MLPLA architecture

Given the above mentioned constraints, the MLPLA architecture (figure 1) is composed of three main layers: (a) input; (b) processing pipeline and (c) output. For overcoming language-dependent and approach-based limitations, our system is built using a scalable plug-and-play methodology. The processing units are built so that they implement one of three different interfaces depending on the module under which they operate. The three interfaces are (a) the data input processor – an implementation of this interface must be able to receive the input text as a character sequence and perform any necessary pre-processing in order to obtain a tokenized text; (b) the base processor interface – an implementation receives a sequence of tokens, each token containing standard attributes (part-of-speech, lemma, phonetic transcription, syllables, accent, chunk and dependencies) but also allowing the insertion of non-standard attributes through a key-value table - each processor is responsible for building its own feature set using all available data, performing the NLP task it was designed for and filling in either a value for a standard attribute or adding a custom attribute; (c) the feature-based output – an implementation must take a sequence of tokens and convert it into a feature-based output, depending on the application in which MLPLA is used.
The order in which the base processors are chained is controlled externally from a configuration file (see figure 2 for details).

```
[Input]
com.ineo.nlp.language.preprocessing.BasicTokenizer
[Pipeline]
com.ineo.nlp.language.baseprocessors.BasicTagger
com.ineo.nlp.language.baseprocessors.BasicLemmatizer
com.ineo.nlp.language.baseprocessors.BasicChunker
com.ineo.nlp.language.baseprocessors.BasicParser
com.ineo.nlp.language.baseprocessors.BasicSyllabifier
com.ineo.nlp.language.baseprocessors.BasicLTS
com.ineo.nlp.language.baseprocessors.BasicStress
[Output]
com.ineo.nlp.language.formats.TabFeatureOutput
```

Figure 2 - MLPLA configurable pipeline

3. MLPLA standard processors

MLPLA comes with a standard processing pipeline which can be trained and used for any language as long as the baseline accuracy (implicitly provided by the classifier and predefined feature set) is sufficient for the task at hand. In what follows we will thoroughly describe how each processing step is carried out and we will emphasize on the classifier we used, the standard feature sets and how the accuracy figures compare to the state-of-the-art. We must mention that our primary goal was to reduce the size of the model and the computational costs, while still providing a good accuracy on out-of-vocabulary (OOV) words.
The basic processing pipeline is composed of: (a) a sentence splitter and tokenizer; (b) a part-of-speech tagger; (c) a lemmatizer; (d) a chunker; (e) a syllabifier, (f) a letter-to-sound convertor and (g) a lexical stress predictor. All these processors use decision trees constructed using the ID3 algorithm (Quinlan, 1986), because of the small model footprint and the low computational cost during runtime log(n) – where n is the total number of features).

3.1. Sentence splitting and tokenization

The lowest preprocessing phase (on plain text) in any natural language processing application is a two-fold process divided between (a) separating the text into sentences (also referred to as "Sentence splitting" or "Sentence boundary detection") which can be translated into automatically determining where each sentence starts and where it ends and (b) tokenizing, a process with the objective of isolating words and punctuation into white-space separated character sequences.
It is known that ambiguities arise in sentence splitting when punctuation signs, like dots, may occur in abbreviations and acronyms that may be followed by capitalized words or numbers. Similarly, tokenization is also a more difficult than expected task, as, for example, punctuation creates ambiguities whether a character sequence should be split or not (ex: a date should not be split "28.03.15" while a year range should be split "1930-1998" in three tokens (1930, -, 1998)).
There are several approaches to sentence splitting and tokenization: aside from rule-based methods there are several data-driven approaches such as using Conditional Random Fields Classifier (CRFs) (Tomanek et al., 2007; Yang et al., 2005), statistical n-gram models (Palmer and Hearst, 1997; Akita et al., 2006), Support Vector Machines (SVM) (Akita et al., 2006; Chung and Galdea, 2009).
When using classifiers, all data-driven sentence splitting methods vary in feature-sets and the classifier of choice, but whenever applied on the same corpora the results correlate.
Our standard tokenizer and sentence splitter is trained using news-domain corpora. The corpus is composed of

5000 sentences which were automatically tokenized using heuristics and then manually corrected. Our approach is a two-step process: (a) we initially split the corpus into "pre-tokens" (see below for details) and (b) we use a classifier to detect if a pre-token should be followed by a white-space or not: (a) Pre-tokens are obtained parsing the sentence from left to right and generating a token before every white space and also whenever special characters that could serve as punctuation is encountered. The special characters are preserved as tokens themselves. (b) For every pre-token we trained the classifier to predict if this token should be kept as a token or merged with the next pre-token based on the following feature set: (a) Initial white-space before; (b) Initial white-space after (c) For all the tokens in a centered window of 5: (i)Token case: lower-case, capitalized, inverted-caps, upper-case, N/A; (ii) Token type: numeric, alpha, alpha-numeric, special; (iii) Only consonants: true or false; (iv) For special tokens only: the token itself.
Using ten-fold validation we obtained a token-level accuracy of 99.98% and a sentence-boundary accuracy of 98.53% with a model-size of 9Kb.

3.2. Part-of-speech (POS) tagging

Speech recognition and synthesis, machine translation and information retrieval are areas where part of speech tagging (POS) finds its use. It is also known as word category disambiguation and is the process in which the words of a sentence are annotated with their grammatical category and specific attributes (gender, number, case etc.).
The high importance of this task has generated great interest from the research community, leading to a large number of well-established POS tagging methods such as Hidden Markov Models (HMM) (Brants, 2000), maximum entropy classifiers (Berger et al., 1996; Ratnaparkhi, 1996), Neural Networks (Marques and Lopes, 1996; Boroș et al., 2013) Conditional Random Fields (CRF) (Lafferty et al., 2001), Support Vector Machines (Gimenez et al., 2004) and Bayesian networks (Samuelsson, 1993). For English, most of these methods have a high accuracy rate (around 98%), but this task gets particularly difficult for large tagset languages such as Czech or Romanian. Large tag sets introduce a wider ambiguity range for every word, which combined with data sparseness (when training) leads to poor accuracy for standard tagging methods. In such cases, language particularities are exploited by alternative tagging methods (e.g. Tiered Tagging for Romanian (Tufiș and Dragomirescu, 2004)) or, if possible, to rely on a smaller tag set.
The MLPLA tagger is designed in the spirit of minimizing the model size while allowing for a lower, but sufficient, accuracy figure. Many of the previously mentioned taggers use lexicons and n-gram models that take up a lot of space and, in some cases, the algorithms that are used employed yield high computational complexity. It is not prohibitively expensive for a normal desktop computer to perform k^2*n operations (see HMM-based tagging), but this becomes a threat to the autonomy of a mobile platform when it is applied to a large number of sentences. Our tagging methodology is a two-step approach: (a) In the first step, for every word inside a sentence we compute the possible tags using a classifier that uses character-level features extracting from the head and tail of that word. The feature window length is composed of the first and last 7 lowercased characters of the word. Additionally we add features such as token-case, token-type and if the word is only composed of consonants (see the tokenization features for details). Our ID3 implementation preserves leaf probabilities and they are later used by our algorithm. The model size for this step is 6.8MB, but is significantly smaller than lexicons combined with n-gram models; (b) In the second step we also use a decision tree to classify and select the appropriate POS for every word inside the sentence. The features used in this step are extracted from a 5-word centered window and it is composed of: the 2 previously assigned POS tags (for the previous words), and the possible tags for the current and next two words with associated probabilities (the size of the model used in this step is only 58KB).
We tested our tagger on the 1984 corpus and obtained and accuracy of 95.13% which is lower than the state-of-the-art but it has a very small footprint and has a reduces computational cost.

3.3. Syllabifier

Splitting a word into syllables is useful in many applications from word-processors (line splitting and word wrap) to text-to-speech synthesis systems (syllables govern articulation of words and accent). Our data-driven approach for syllabification is based on the numbered onset-nucleus-coda (ONC) strategy proposed by Bartlett et al. (2008). Each syllable is composed of a nucleus vowel/vowels (nucleus) with or without leading (onset) and tailing (coda) consonants. The corpus is labeled offline using a simple algorithm (use a list of vowels to identify the nucleus and then use the nucleus to identify the nucleus and the coda) and the classifier is trained to label each letter inside a word using a feature-set extracted from a centred window of 5 characters. At runtime, having marked a word by its ONC coding, we use straight forward rules (ex: there should be a split between a Ci-Oj or an Ni-N1) to split the word into syllables.
To test this method we used a training set composed of 487K words with a total number of 5.246M tagged letters and a testset composed of 54K words with 582K letters. The ID3 classifier obtained an accuracy of 98.93% on OOV words with an 67KB model footprint. Additionally we performed another experiment with a DeepNeuralNetwork (DNN) based classifier which, after tweaking hyper-parameters and fine-tuning, yielded an accuracy of 98.23%, which is 0.70% lower than the ID3 but generates a half-sized model footprint (36KB). We need to mention that the highest accuracy obtained on this corpus was with a MIRA classifier (99.01%) but an significantly larger model footprint (9.4MB) (Boros, 2013).

3.4. Lemmatizer

Lemmatization is the process of automatically extracting a word's canonical (dictionary) form and is used in processes such as machine translation, text generation, information extraction, speech synthesis and sentiment

analysis. The main advantage of using lemmas is that they offer context invariance, thus reducing data-sparseness.
While for the English language lemmatization is quite a simple process, for highly inflected languages the situation changes. In what follows we will describe how we constructed a model for Romanian (a highly inflectional language). Though this is a targeted experiment, the feature-set we used can be applied to multiple languages, provided that an appropriate training corpus is available. In our approach we used a training set of approximately 900K wordforms. To our knowledge, the best performing lemmatizers on Romanian that use the same training corpora as ours are introduced in Ion (2007) (82% accuracy – automatically determining rules for deletion and addition using a letter n-gram model) and in Boroș (2013) (94.19% accuracy using a tagging strategy at character level for preservation, deletion or substitution).
The goal of the lemmatization system is to learn the grammatical rules which govern the inflection mechanism of the target language. In our case, the corpus contained the POS information for every wordform. To obtain a fair evaluation we used the classic 10-fold cross validation: we split the data so that the test-set would retain 10% of the initial corpus, repeating the process 10 times, each time with a different tenth of the corpus as test-set. The result of a 10-fold cross validation is the average accuracy on the 10 test-sets. This technique allowed us to provide the lemmatization algorithm with complete models and also to perform a thorough validation on words with rare tags (which would have been likely to have a skewed distribution between training and testing data). The lemmatizer was trained to automatically remove a number of characters from a word and, if necessary, pad a new sequence. In order to have a fair picture on the process we grouped and counted the words based on the number of tailing characters that had to be removed in the lemmatization process. An interesting observation is that words belonging to group 6 (which require the deletion of 6 characters) represent only 0.3% which is a small figure compared with the other groups (from 0 to 9) that represent at least 2% each.
The feature set used by the lemmatizer contains the part of speech information and tailing characters. We performed 5 experiments in which we varied the tail size from 5 to 9 (see table 1 for results).

Table 1 - Lemmatization results for different tail sizes

Trim	5	6	7	8	9
Acuratete	90.5%	92.6%	93.0	92.9%	93.02%
Model size (KB)	279	439	619	776	879

3.5. Chunker

Chunking, also referred to as shallow parsing is the process of identifying partial syntactic structures within a text based on POS information. They are usually a pre-processing step used before dependency parsing but can also act standalone in tasks such as machine translation and speech synthesis (chunks are phonologically relevant and they provide good prosody cues for text-to-speech (TTS) systems).
Several methods have been proposed by other authors such as finite state automata (Kornai, 1999), SVMs (Kudoh and Matsumoto, 2000).
Our chunking system takes after the POS-based partial grammars proposed in Ion (2007) and determines chunks as sequences of at least two POS tags that allow a finite-state machine reach the final state of either one of the top-level entities (which represent the chunk types). The partial grammars have to be manually designed for each language, but our modular pipeline allows the implementation of any other rule-based or data-driven method that suits the requirements of a particular language or task.

3.6. Letter-to-sound (LTS) convertor

The Letter-to-sound process handles the 'translation' of words from their written form to the way they should sound (also in text form). The process is performed at the letter level – each letter can produce one or more different sounds, so each word can be 'translated' into different sounding words. For example, the Romanian language has 33 letters in the alphabet that can be voiced in 48 different ways (depending on the containing word and context).
Our ID3 implementation of the LTS uses the following features: the first and second letters (if available) that are before the letter that is to be predicted (features #1 and #2), the current letter (feature #3), as well as the first and second letters (also if available) following our predicted letter (features #4 and #5); the prediction of the ID3 for the previous letter (feature #6). We experimented with other features such as the position of the letter in the word, whether it is capitalized or not, but we did not obtain better results than using only the previous 6 features; also, the model is smaller.
The corpus we used for training contains 7248 words having a total of 53491 phoneme-annotated letters. Testing was performed on 746 words (5551 letters to be 'translated').
We obtained a word-level accuracy of 94% (the test contained OOV words, not previously seen in training), while our model size was only 32.2KB.
We further compare our results with our previous experiments. Training and testing were performed in similar conditions. Using a MIRA-based model, we obtained a 96.29% accuracy, while using a Deep Neural Network (DNN) we obtained a slightly smaller 96.16% accuracy. The MIRA model was almost 1.4MB in size; the DNN model, significantly smaller, was only 43.4KB, while our ID3 model was only 32.2KB.

Table 2 - LTS accuracies and model sizes

	MIRA	DNN	ID3
Accuracy	96.29%	96.16%	96.25%
Model size	1.4 MB	43.4 KB	32.2 KB

3.7. Lexical stress predictor

The Lexical stress predictor process attempts to correctly place stress on words. Languages have words that are

written identically but are pronounced differently depending on context, having different meanings. As such, correctly identifying where to place stress on a word (and thus altering its prosody) is an important task for speech synthesis systems.

We implemented the Lexical stress predictor as a classification task using ID3. For each character in a word, in sequence, we attempt to predict whether it should be stressed or not. We have used the following features: three characters before and after the predicted character (features #1 - #6), the predicted character (feature #7), the prediction for the previous character (feature #8) and the generic part-of-speech of the word itself: whether the current word is a noun, a verb, adverb, etc. (feature #9).

The test set contains 33695 Romanian words, with about 11 thousand verbs, adverbs and nouns, and less than 100 adverbs. We obtained an average accuracy of 96.25% using ID3.

Comparing with our previous experiments where we used a MIRA and a DNN model, we note: using MIRA we obtained our best results of 98.8% while using DNNs we obtained 97.67%. Size-wise, DNNs have the smallest models of only 110KB, while the current ID3 is also relatively small at 743KB.

Table 3 - LSP accuracies and model sizes

	MIRA	DNN	ID3
Accuracy	98.8%	97.67%	96.25%
Model size	6 MB	110 KB	743 KB

4. Conclusions

We presented a framework that contains the following Natural Language Processing sub-modules: (a) a sentence splitter and tokenizer; (b) a part-of-speech tagger; (c) a lemmatizer; (d) a chunker; (e) a syllabifier, (f) a letter-to-sound convertor and (g) a lexical stress predictor. All these processors use decision trees constructed using the ID3 algorithm. We motivate the choice of classifier as being very small, very fast at runtime, as well as providing results that are very close to those that are state-of-the-art. Our main target is to have a NLP supporting framework designed for mobile platforms. Currently we fully support the Android operating system and we will also handle iOS and Windows Phone in the near future. This platform is free for non-commercial use and can be purchased for commercial purposes (contact any of the authors for further details).

References

Akita, Y., Saikou, M., Nanjo, H., & Kawahara, T. (2006, September). Sentence boundary detection of spontaneous Japanese using statistical language model and support vector machines. In INTERSPEECH.

Baldridge, J. (2005). The opennlp project. URL: http://opennlp. apache. org/index. html, (accessed 7 september 2015).

Berger, A. L., Pietra, V. J. D., & Pietra, S. A. D. (1996). A maximum entropy approach to natural language processing. Computational linguistics, 22(1), 39-71.

Boros, T. (2013). A unified lexical processing framework based on the Margin Infused Relaxed Algorithm. A case study on the Romanian Language. In RANLP (pp. 91-97).

Brants, T. (2000, April). TnT: a statistical part-of-speech tagger. In Proceedings of the sixth conference on Applied natural language processing (pp. 224-231). Association for Computational Linguistics.

Chung, T., & Gildea, D. (2009, August). Unsupervised tokenization for machine translation. In Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 2-Volume 2 (pp. 718-726). Association for Computational Linguistics.

Giménez, J., & Marquez, L. (2004). SVMTool: A general POS tagger generator based on Support Vector Machines. In In Proceedings of the 4th International Conference on Language Resources and Evaluation.

Ion, R. (2007). Word sense disambiguation methods applied to English and Romanian. PhD thesis. Romanian Academy, Bucharest.

K. Tomanek, J. Wermter & U. Hahn, Sentence and Token Splitting Based on Conditional Random Fields, In: PACLING 2007 - Proceedings of the 10th Conference of the Pacific Association for Computational Linguistics. Melbourne, Australia, September 19-21, 2007. Pacific Association for Computational Linguistics, 2007, pp.49-57.

Kornai, A. (Ed.). (1999). Extended finite state models of language (Studies in natural language processing). Cambridge: Cambridge University Press.

Kudoh, T., & Matsumoto, Y. (2000). Use of support vector learning for chunk identification. In Proceedings of CoNLL-2000 and LLL-2000, Lisbon (pp. 142–144).

Lafferty, J., McCallum, A., & Pereira, F. C. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data.

Manning, C. D., & Schütze, H. (1999). Foundations of statistical natural language processing. MIT press.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J., & McClosky, D. (2014, June). The Stanford CoreNLP natural language processing toolkit. In Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations (pp. 55-60).

Marques, N. C., & Lopes, G. P. (1996). A neural network approach to part-of-speech tagging. In Proceedings of the 2nd meeting for computational processing of spoken and written Portuguese (pp. 21-22).

Palmer, D. D., & Hearst, M. A. (1997). Adaptive multilingual sentence boundary disambiguation. Computational Linguistics, 23(2), 241-267.

Quinlan, J. R. (1986). Induction of decision trees. Machine learning, 1(1), 81-106.

Ratnaparkhi, A. (1996, May). A maximum entropy model for part-of-speech tagging. In Proceedings of the conference on empirical methods in natural language processing (Vol. 1, pp. 133-142).

Samuelsson, C. (1993, June). Morphological tagging based entirely on Bayesian inference. In 9th Nordic conference on computational linguistics.

Tufiş, D., & Dragomirescu, L. (2004). Tiered tagging revisited. In *Proceedings of the 4th LREC Conference* (pp. 39-42).