IceNLP
A Natural Language Processing Toolkit for Icelandic[*]

User Guide

Hrafn Loftsson
School of Computer Science
Reykjavik University
hrafn@ru.is

Anton Karl Ingason
School of Humanities
University of Iceland
antoni@hi.is

July 2021

[*]The following persons have contributed to the development of IceNLP: Hrafn Loftsson, Anton Karl Ingason, Aðalsteinn Tryggvason, Guðmundur Örn Leifsson, Hlynur Sigurþórsson, Ragnar Lárus Sigurðsson, Sverrir Sigmundarson and Robert Östling.

Contents

1 What is IceNLP?

IceNLP is an open-source Natural Language Processing (NLP) toolkit for analysing Icelandic text. The toolkit consists of a tokeniser and a sentence segmentiser, the morphological analyser *IceMorphy*, the linguistic rule-based tagger *IceTagger*, the trigram tagger *TriTagger*, the perceptron tagger *IceStagger*, the shallow parser *IceParser*, the lemmatiser *Lemmald*, and the named entity recogniser *IceNER*. The system is written as a collection of Java classes.

The tokeniser is used for tokenising stream of characters into linguistic units and for performing sentence segmentation (Palmer 2000).

IceMorphy is mainly used for guessing the tags for unknown words and filling *tag profile gaps* in a dictionary (Loftsson 2008).

IceTagger is a linguistic rule-based tagger[1] for tagging Icelandic text (Loftsson 2006; 2008). It uses a large part-of-speech (PoS) tagset consisting of about 600 tags (see Section 3). Evaluation showes that *IceTagger* achieved higher accuracy than the best performing data-driven tagger when tested using the same test corpora and the same ratio of unknown words (Loftsson 2008, Loftsson et al. 2009; 2011, Helgadóttir 2004). The average tagging accuracy, computed when tagging test corpora derived from the *Icelandic Frequency Dictionary* (*IFD*) corpus (Pind et al. 1991), is about 92%[2]. When using data from *BÍN* (see Section 11), the Database of Modern Icelandic Inflections (Bjarnadóttir 2005), the accuracy increases to about 92.8%.

TriTagger is a re-implementation of the well known statistical *TnT* tagger (Brants 2000). By using *TriTagger* as a word class tagger during initial disambiguation, then using *IceTagger* to disambiguate tags that are consistent with the chosen word class, and finally using *TriTagger* again to fully disambiguate words, to which *IceTagger* is not able to assign unambiguous tags, an accuracy of about 92.7% is achieved (Loftsson et al. 2009; 2011). By using *BÍN*, the accuracy further increases to about 93.5%.

IceStagger is a modified version of *Stagger* (Östling 2013), a tagger based on the Averaged Perceptron algorithm (Collins 2002). By adding specific linguistic features and using *IceMorphy*, an accuracy of about 92.8% is achieved (Loftsson and Östling 2013). By using *BÍN*, the accuracy increases to about 93.7%.

IceParser is a shallow parser based on the incremental finite-state parsing technique (Aït-Mokhtar and Chanod 1997). It labels both constituent structure and grammatical functions. Evaluation shows that F-measure for constituents and syntactic functions is 96.7% and 84.3%, respectively, when assuming perfectly tagged input (Loftsson and Rögnvaldsson 2007).

Lemmald is a mixed method lemmatiser for Icelandic. It combines the advantages of data-driven machine learning with linguistic insights to maximize performance. Given correct tagging, the system lemmatizes Icelandic text with an accuracy of 99.55% (Ingason et al. 2008).

IceNER is a rule-based named entity recogniser for Icelandic. The system marks persons, companies, locations and events. Evaluation has shown that *IceNER* achieves an overall F-score of 71.5% without using a gazette list, and 79.3% when using a gazette list of only 523 names (Tryggvason 2009).

[1]As apposed to a data-driven tagger trainable on different languages.

[2]Tagging accuray is measured using a corrected version of the IFD corpus (Loftsson 2009).

2 Installation

The source of *IceNLP* is available for download/cloning at `https://github.com/hrafnl/icenlp`.

Release versions (programs and data without source code) can be downloaded from `https://github.com/hrafnl/icenlp/releases`.

The description below assumes installation or a release version for the **Linux** operating system. The programs and data come in a zip-file named *IceNLP-x.y.z.zip* (where *x.y.z* is the current version number). Run **unzip** on this zip-file and extract all the files to a directory of your choice.

A main directory, **IceNLP**, will be created with the following subdirectories: **bat**, **dict**, **dist**, **doc**, **lib**, and **ngrams**.

The **bat** directory includes shell scripts (.sh files) for running individual components of the tool. The commands for each tool can be found in a subdirectory of the **bat** directory (see Section 13).

The **dict** directory contains various dictionaries related to the individual tools of *IceNLP* as well as shell scripts to extract data from *BÍN*.

The **dist** directory contains the *IceNLPCore.jar* file. This file consists of all the .class files needed to run *IceNLP* along with default dictionaries ("resource files").

The **doc** directory contains this user guide and a description of the Icelandic tagset.

The **lib** directory contains various .jar files used by *IceNLP*.

The **ngrams** directory contains tools for building ngram models.

3 The tagset

The taggers in *IceNLP* use the main Icelandic tagset, created during the making of the *IFD* corpus. Due to the morphological richness of the Icelandic language the main tagset is large and makes fine distinctions compared to related languages. The original tagset contains about 700 tags, but the taggers have been developed/trained using a reduced version of the tagset, containing about 600 tags. Type information for proper nouns (named-entity classification) has been removed and only one tag for numerical constants is used (Loftsson et al. 2011).

Each tag in the tagset comprises word class information and morphological features. Each character in the tag has a particular function. The first character denotes the word class. For each word class there is a predefined number of additional characters (at most six) which describe morphological features, like gender, number and case for nouns, degree and declension for adjectives, voice, mood and tense for verbs, etc.

Table 1 shows the semantics of the noun tags. Consider, for example, the tag "*nken*". The first letter, "*n*", denotes the word class "*nafnorð*" (noun), the second letter, "*k*", denotes the gender "*karlkyn*" (masculine), the third letter, "*e*", denotes the number "*eintala*" (singular) and the last letter, "*n*", denotes the case "*nefnifall*" (nominative case).

To give another example, consider the words "*fallegu hestarnir stukku*" (the beautiful horses jumped). The corresponding tag for "*fallegu*" is "*lkenvf*" denoting adjective, masculine, singular, nominative, weak declension, positive; the tag for "*hestarnir*" is "*nkfng*" denoting noun, masculine, plural, nominative with suffixed definite article; and the tag for "*stukku*" is "*sfg3fþ*" denoting verb, indicative mood, active voice, 3-rd person, plural and past tense. Note the agreement in gender, number and case.

Char#	Category/Feature	Symbol – semantics
1	Word class	**n**–noun
2	Gender	**k**–masculine, **v**–feminine, **h**–neuter, **x**–unspecified
3	Number	**e**–singular, **f**–plural
4	Case	**n**–nominative, **o**–accusative, **þ**–dative, **e**–genitive
5	Article	**g**–with suffixed definite article
6	Proper noun	**s**–proper noun

Table 1: The semantics of the noun tags

A complete description of the Icelandic tagset can be found in the Appendix.

4 IceMorphy

The unknown word guesser, *IceMorphy*, uses a familiar approach to unknown word guessing, i.e. it performs morphological analysis, compound analysis and ending analysis (Mikheev 1997, Nakov et al. 2003). An additional important feature of *IceMorphy* is its handling of *tag profile gaps.*

1. **Morphological analysis.** The morphological analyser tries to classify an unknown word as a member of a particular morphological class. For a given unknown word w, a morphological class is guessed depending on the morphological ending of w. Then the stem r of w is extracted and all k possible morphological endings for r are generated resulting in search strings, s_i $(i = 1, \dots, k)$, such that $s_i = r + ending_i$. A dictionary lookup is performed for s_i until a word is found having the same morphological class as was originally assumed or no match was found. If the search is successful, a tag is deduced using the assumed word class and the morphological ending of w.

2. **Compound analysis.** This part uses a straightforward method of repeatedly removing prefixes from unknown words and performing a lookup for the remaining part of the word. If the remaining word part is not found in the dictionary it is sent to the morphological analysis for further processing. If the lookup or morphological analysis deduces a tag t for the remaining word part, the original word (without prefix removal) is given the same tag t.

3. **Ending analysis.** The ending analyser is called if an unknown word can neither be deduced by morphological analysis nor by compound analysis. This component uses a hand-written dictionary of endings along with an automatically generated one. The former, which is looked up first, is mainly used to capture common endings for adjectives and verbs, for which numerous tags are possible. *IceMorphy* assumes that endings are different for capitalized words vs. other words and therefore uses two endings dictionaries, one for proper nouns and another for all other words.

4. **Tag profile gaps.** A *tag profile gap* arises when a particular word, listed in the dictionary, has some missing tags in its tag profile (set of possible tags). This, of course, presents problems to a disambiguator since its purpose is to select one single correct tag from all possible ones. For each noun, adjective, or verb of a particular morphological class, *IceMorphy* generates all possible tags for the given word.

5 IceTagger

IceTagger reads an untagged input file consisting of Icelandic sentences and produces an output file consisting of the words of the sentences augmented with the appropriate PoS tags. The tagger consists of the following phases:

1. **Tokenisation.** The sequence of characters in the input file is split into simple tokens (linguistic units) like words, numbers and punctuation marks. In some cases, sentence segmentation needs to be carried out, i.e. the process of identifying when one sentence ends and another one begins.

2. **Introduction of ambiguity.** For each sentence to be tagged, the tag profile for each word, both known and unknown words, is introduced. A word is looked up in a pre-compiled dictionary. If the word exists, i.e. the word is known, the corresponding tag profile for the word is returned. In the case of a *tag profile gap*, the unknown word guesser, *IceMorphy*, is used for filling in the missing tags. If the word does not exist in the dictionary, i.e. the word is unknown, *IceMorphy* is used for guessing the possible tags. At the end of this phase, a given word of a sentence can have multiple tags, i.e. ambiguity has been introduced.

3. **Disambiguation.** *IceTagger* removes ambiguity by considering the context in which a particular word appears. To be more specific, the tagger removes illegitimate tags from words based on context. The tasks below are applied to one sentence at a time:

   (a) **Identify idioms and phrasal verbs.** Idioms, i.e. bigrams and trigrams, which are always tagged unambiguously are kept in a special dictionary. A special dictionary is also used for recognising phrasal verbs, i.e. verb-particle pairs whose words are adjacent in text.

   (b) **Apply local elimination rules.** A sentence to be tagged is scanned from left to right and all tags of each word are checked in sequence. Depending on the word class (the first letter of the tag) of the focus word, the token is sent to the appropriate disambiguation routine which checks a variety of disambiguation constraints applicable to the particular word class and the surrounding words. At each step, only tags for the focus word are eliminated.

   (c) **Apply global heuristics.** Grammatical function analysis is performed, prepositional phrases are guessed, and the acquired knowledge is used to force feature agreement where appropriate. The heuristics are a collection of functions that guess the syntactic structure of the sentence and use it as an aid in the disambiguation process. Additionally, specific heuristics are used to choose between supine and past participle verb forms, infinitive or active verb forms, and ensuring agreement between reflexive pronouns and their antecedents. At last, the default heuristic is simply to choose the most frequent tag for a given word.

6 TriTagger

TriTagger is statistical tagger based on a Hidden Markov Model (HMM). The tagger is data-driven, i.e. it learns its language model from a tagged corpus. The main advantage of data-driven taggers is that they are language independent and no (or limited) human effort is needed for derivation of the model. The algorithm used by the tagger is as follows (consult (Brants 2000) for full details):

1. **Tokenisation.** *TriTagger* uses the tokenisation method described in section 5.
2. **Introduction of ambiguity.** Known words are handled in the manner described in section 5. Since *TriTagger* is language independent, it has no knowledge of Icelandic morphology. Suffix analysis is, therefore, the default method for guessing possible tags for unknown words. On the other hand, since *IceMorphy* already exists, it can be called from within *TriTagger* (see section 13.4). In that case, *TriTagger* will use tags provided by *IceMorphy* if *IceMorphy* can use morphological analysis (as opposed to ending analysis or default handling) to guess the tags for an unknown word. For other unknown words, suffix analysis is carried out.
3. **Disambiguation.** The states of the HMM represent pair of tags and the model emits words each time it leaves a state. A trigram tagger finds an assignment of PoS to words by optimising the product of lexical probabilities and contextual probabilities. Lexical probability is the probability of observing word i given PoS j ($p(w_i|t_j)$) and contextual probability is the probability of observing PoS i given k previous PoS ($p(t_i|t_{i-1}, t_{i-2}, \ldots, t_{i-k})$; $k = 2$ for a trigram model). A sentence is tagged by assigning it the tag sequence which receives the highest probability by the model.

The probabilities of the model are estimated from a training corpus using maximum likelihood estimation. Thus, before *Tritagger* can be used it needs to be trained on a tagged corpus. A pre-trained model named *otb*, derived from the *IFD* corpus, can be found in the **ngrams/models** directory. Training of the tagger is described in section 13.4.1.

7 IceStagger

IceStagger (Loftsson and Östling 2013) is a modified version of the Stockholm Tagger (Stagger) (Östling 2013), an open-source implementation of the Averaged Perceptron tagger by Collins (2002).

The Averaged Perceptron algorithm uses a feature-rich model that can be trained efficiently. Features are modeled using *feature functions* of the form $\phi(h_i, t_i)$ for a history h_i and a tag t_i. The history h_i is a complex object modeling different aspects of the sequence being tagged. It may contain previously assigned tags in the sequence to be annotated, as well as other contextual features such as the form of the current word, or whether the current sentence ends with a question mark. Intuitively, the job of the training algorithm is to find out which feature functions are good indicators that a certain tag t_i is associated with a certain history h_i.

A model consists of feature functions ϕ_s, each paired with a *feature weight* α_s which is to be estimated during training. The scoring function is defined over entire sequences, which in a PoS tagging task typically means sentences. For a sequence of words w of length n in a model with d feature functions, the scoring function is defined as:

$$score(w, t) = \sum_{i=1}^{n} \sum_{s=1}^{d} \alpha_s \phi_s(h_i, t_i)$$

Training the model is done in an error-driven fashion: tagging each sequence in the training data with the current model, and adding to the feature weights the difference between the corresponding feature function for the correct tagging, and the model's tagging.

During tagging, the highest scoring sequence of tags is computed:

$$\bar{t} = \arg\max_t \mathit{score}(w, t)$$

8 IceParser

IceParser is an incremental finite-state parser. The parser comprises a sequence of finite-state transducers, each of which uses a collection of regular expressions to specify which syntactic patterns are to be recognised. The purpose of each transducer is to add syntactic information into the recognised substrings of the input text.

IceParser is designed to produce annotations according to an annotation scheme described in (Loftsson and Rögnvaldsson 2006). The parser consists of two main components: a phrase structure module and a syntactic functions module.

The purpose of the phrase structure module is to add brackets and labels to input sentences to indicate phrase structure. The output of one transducer serves as the input to the following transducers in the sequence. The syntactic annotation is performed in a bottom-up fashion, i.e. deepest constituents are analysed first.

Both simple phrase structures and complex structures are recognised. Since the parser is based on finite-state machines, each phrase structure does not contain a structure of the same type. Complex structures contain other structures, whereas simple structures do not.

Two labels are attached to each marked constituent: the first one denotes the beginning of the constituent, the second one denotes the end (e.g. [NP . . . NP]). The main labels are **AdvP**, **AP**, **NP**, **PP** and **VP** – the standard labels used for syntactic annotation (denoting adverb, adjective, noun, prepositional and verb phrase, respectively). Additionally, the labels **CP**, **SCP**, **InjP**, **APs**, **NPs** and **MWE** are used for marking coordinating conjunctions, subordinating conjunctions, interjections, a sequence of adjective phrases, a sequence of noun phrases, and multiword expressions, respectively.

The purpose of the syntactic functions module is to add functional tags to denote grammatical functions. The input to the first transducer in this module is the output of the last transducer in the phrase structure module, i.e. it is assumed that the syntactic functions module receives text that has been annotated with constituent structure. As in the phrase structure module, the output of one transducer serves as the input to the following transducers in the sequence.

Four different types of syntactic functions are annotated: genitive qualifiers, subjects, objects/complements and temporal expressions. Curly brackets are used for denoting the beginning and the end of a syntactic function, and special function tags are used for labels (*QUAL, *SUBJ, *OBJ/*OBJAP/*OBJNOM/*IOBJ/*COMP, *TIMEX). Please refer to (Loftsson and Rögnvaldsson 2006), for a thorough description of the annotation scheme used.

In total, *IceParser* consists of about 25 finite-state transducers. The parser is implemented in Java and the lexical analyser generator tool JFlex (http://jflex.de/).

9 Lemmald

Lemmald is a mixed method lemmatizer for Icelandic. It achieves good performance by relying on *IceTagger* for tagging and the *IFD* corpus for training. *Lemmald* combines the advantages of data-driven machine learning with linguistic insights to maximize performance.

To achieve this, it makes use of a novel approach: Hierarchy of Linguistic Identities (HOLI), which involves organizing features and feature structures for the machine learning based on linguistic knowledge (Ingason et al. 2008). Accuracy of the lemmatisation is further improved using an add-on which connects to *BÍN*.

Given correct tagging, the system lemmatises Icelandic text with an accuracy of 99.55%.

10 IceNER

IceNER is a named entity recoginiser for Icelandic, based on linguistic rules. The system marks persons, companies, locations and events. Evaluation has shown that *IceNER* achieves an overall F-score of 71.5% without using a gazette list, and 79.3% when using a gazette list of only 523 names (Tryggvason 2009).

The system reads the text several times, applying the strictest rules first and then more relaxed rules. *IceNER* is built on two subsystems. The first, called NameScanner, uses regular expressions to create lists of named entities based on endings such as "- son", "-dóttir", and abbreviations like "hf", "ehf". It also generate lists of words that can be of significance, such as professional titles, words that imply a location, a company or a person, etc.

The second subsystem, NameFinder, reads these lists, and breaks up combinations of words if a name is made of more than a single word. If, for example, the name "Ingibjörg Sólrún Gísladóttir" appears in the name list, then entries for "Ingibjörg Sólrún", "Ingibjörg", "Sólrún", "Gísladóttir" and "Sólrún Gísladóttir" will also be added. The NameFinder will also read the text itself, after it has been run through *IceTagger*. The NameFinder will then use the name lists and rules based on the context in which entities appear to try to categorize the entities.

11 BÍN

BÍN ("Beygingarlýsing íslensks nútímamáls") is a comprehensive full form database of modern Icelandic inflections Bjarnadóttir (2005), developed at the *Árni Magnússon Institute for Icelandic Studies*. BÍN contains about 280,000 paradigms, with over 5.8 million inflectional forms for common nouns, proper nouns, adjectives, verbs, and adverbs.

Due to licensing issues, *BÍN* cannot be distributed with *IceNLP*. However, *IceNLP* contains several shell scripts to extract data from *BÍN* for the purpose of using it in it's taggers. As stated in Section 1, the accuracy of the taggers increases considerably when extending their dictionaries with data from *BÍN*.

The shell scripts rely on a database dump of *BÍN*, which is available for download from `bin.arnastofnun.is`. The dump file has the name *SHsnid.csv*.

Copy this file into the **dict/BIN** directory. Then run the **extractBinData.sh** script, which will generate dictionaries with data from *BÍN* for the three taggers: *IceTaggger*, *TriTagger*, and *IceStagger*.

To run a tagger with an extended dictionary, please refer to Section 13.

12 File format

The *IceNLP* toolkit uses **UTF8** character encoding for all files. It is thus assumed that dictionaries and input files are encoded in UTF8 format. Moreover, output files, generated by the tool, will be encoded in UTF8.

12.1 Tagging

12.1.1 Input file

The input file to be tagged can have one of four formats:

1. **One token/tag pair per line** (only used by *IceStagger*). An empty line (the newline character) is required between sentences.
2. **One token per line**. An empty line (the newline character) is required between sentences.
3. **One sentence per line**.
4. **Other format**. This entails that a sentence can span more than one line, or that there can be more than one sentence per line in the input file.

12.1.2 Output file

The taggers can return output in either of two formats:

1. **One token/tag per line** (or one token/tag/lemma per line). The token appears first in each line followed by the tag(s) selected by the tagger (and the lemma if lemmatisation is needed (see Section 13.3). If the token is an unknown word the string *<UNKNOWN>* appears after the tag. There is some additional output possible in this format, which we will discuss in Section 13.3. Here is an example of this output format:

```
ég fp1en
opnaði sfg1eþ
dyrnar nvfog
, ,
steig sfg1eþ
inn aa
og c
sparkaði sfg1eþ
hvítum lkeþsf
brennivínspoka nkeþ <UNKNOWN>
með aþ
sunddóti nheþ <UNKNOWN>
til ae
hliðar nvee
. .
```

2. **One sentence per line.** Each line consists of a sentence in which each token is followed by the tag (and possibly the lemma), selected by the tagger. Here is the example above in this format:

```
ég fp1en opnaði sfg1eþ dyrnar nvfog , , steig sfg1eþ inn aa og c sparkaði sfg1eþ
hvítum lkeþsf brennivínspoka nkeþ með aþ sunddóti nheþ til ae hliðar nvee . .
```

12.1.3 Dictionaries

The **dict** directory contains a copy of the default dictionaries and wordlists that are part of the *IceNLPCore.jar* file. The files in the **dict** directory can be changed by the user and parameters for individual tools of *IceNLP* can be used to point to these dictionaries in case the user wants to change the default behaviour (see Section 13).

The dictionaries, which list words/endings and associated tags, used by *IceTagger* have the following format:

$w_1 = t_{11}\_t_{12}\_\ldots\_t_{1s_1}$
$w_2 = t_{21}\_t_{22}\_\ldots\_t_{2s_2}$
...
$w_n = t_{n1}\_t_{n2}\_\ldots\_t_{ns_n}$

Here n is the number of words/endings in the dictionary, w_i is word/ending number i, t_{ik} is the k^{th} frequent tag for word/ending i, and s_i is the number of tags for word/ending i ($i = 1 \ldots n$). Note that the above means that the tags for a given word/ending are sorted according to frequency – the most frequent tag appears first in the list of tags for a given word/ending.

To illustrate, the following is a record from a dictionary for the word "*við*" (see the Appendix for explanation of the individual tags):
við=ao_fp1fn_aþ_aa

Since *TriTagger* bases its language model on frequencies, word and tag frequencies are needed in its dictionary. Thus, the frequency dictionary used by *TriTagger* has the following format:

$w_1\ f_{w_1}\ t_{11}\ f_{t_{11}}\ t_{12}\ f_{t_{12}}\ \ldots\ t_{1s}\ f_{t_{1s}}$
$w_2\ f_{w_2}\ t_{21}\ f_{t_{21}}\ t_{22}\ f_{t_{22}}\ \ldots\ t_{2s}\ f_{t_{2s}}$
...
$w_n\ f_{w_n}\ t_{n1}\ f_{t_{n1}}\ t_{n2}\ f_{t_{n2}}\ \ldots\ t_{ns}\ f_{t_{ns}}$

To illustrate, the following is a record from a frequency dictionary for the word "*við*":
við 5810 ao 3673 fp1fn 1332 aa 507 aþ 298

12.2 Parsing

12.2.1 Input file

The input to the parser are POS-tagged sentences. The tags are assumed to be part of the tagset used in the *IFD* corpus, i.e. the tagset used by *IceTagger*. From version 1.5.0 of *IceNLP*, the parser also accepts tags that confirm to the revised Icelandic tagset, described in the documentation for MIM_GOLD 20.5 (`https://repository.clarin.is/repository/xmlui/handle/20.500.12537/39`). Furthermore, it is assumed that the input file has one sentence in each line.

Here is an example of the input format:

```
ég fp1en opnaði sfg1eþ dyrnar nvfog , pk steig sfg1eþ inn aa og c sparkaði sfg1eþ
hvítum lkeþsf brennivínspoka nkeþ með af sunddóti nheþ til af hliðar nvee . pl
```

12.2.2 Output file

The output of the parser consists of the POS-tagged sentences with added syntactic information. The parser either writes one sentences in each line or one phrase/syntactic function in each line. Here is an example of the latter:

```
{*SUBJ> [NP ég fp1en ] }
[VP opnaði sfg1eþ ]
{*OBJ< [NP dyrnar nvfog ] }
, pk
[VP steig sfg1eþ ]
[AdvP inn aa ]
[CP og c ]
[VP sparkaði sfg1eþ ]
{*OBJ< [NP [AP hvítum lkeþsf ] brennivínspoka nkeþ ] }
[PP með af [NP sunddóti nheþ ] ]
[PP til af [NP hliðar nvee ] ]
. pl
```

13 Usage

Java 1.6 runtime (or later) is required to run the programs. Java is available for free from Oracle, `http://java.com`.

In this section, usage of the individual tools on Linux is described.

13.1 The tokeniser

The tokeniser application is used for tokenising input files and converting between different file formats (the tokeniser performs both word tokenisation and sentence segmentation).

To start the application, open a terminal (command prompt), go to the **bat/tokenizer** directory and type in the following command:

./tokenize.sh [param]

The parameters are:

- *-i <inpFile>*: The input file to be tokenised. The file has a particular input format which is described by the *-if* parameter.
- *-o <outFile>*: The output file into which the tokens are written. The desired output format is described by the *-of* parameter.

- *-if <inputFormat>*: This parameter describes the format of the input file. The possible values are:
  - *0*: One token/tag per line, with an empty line between sentences.
  - *1*: One token per line, with an empty line between sentences.
  - *2*: One sentence per line.
  - *3*: Other different format.
- *-of <outputFormat>*. This parameter describes the desired output format.
  - *1*: One token per line, with an empty line between sentences.
  - *2*: One sentence per line.
- *-l <filename>*: filename is the name of a lexicon used by the tokeniser. The purpose of the lexicon is to list the abbreviations and the multiword expressions (MWEs) that the tokeniser is supposed to recognise. If this parameter is not supplied, the tokeniser uses the default resource file *lexicon.txt* in the *IceNLPCore.jar* file.
- *-c <count>*: The tokeniser quits after tokenising *<count>* sentences.
- *-mwe*: Mark MWEs in the output.
- *-sa*: Split abbreviations. Use this option if each abbreviation is to be splitted into individual parts.
- *-ns*: Not strict tokenisation. This means, for example, that strings like delta$(4) are not broken apart. If this parameter is not supplied, i.e. strict tokenisation is preferred, then the above string will result in the following tokens: delta $ (4).

For example, the following command:

```
./tokenize.sh -i test.txt -o test.out -if 2 -of 1
```

runs the tokeniser on the input file *test.txt* and writes to the output file *test.out*. The format of the input file is one sentence per line, and the desired output format is one token per line.

Furthermore, if the -i parameter is not provided, the tokeniser reads from standard input and writes to standard output. In that case, inputFormat=3 and outputFormat=1. For example, the following Linux command can be used to tokenize the string "Ég á stóran hund. Sá er a.m.k. 10 kíló." (and write the output to the screen):

```
 echo "Ég á stóran hund. Sá er a.m.k. 10 kíló." | ./tokenize.sh
```

13.2 SrxSegmentizer

The *SrxSegmentizer* splits sentences according to rules defined in an SRX file. Such SRX rules are included in the IceNLP distribution and the *Segment* library is used internally to apply the rules.

Use the command **srxsegmentizer.sh**. Two parameters can be supplied, an input file and an output file. If those are omitted, input is read from stdin and output written to stdout.

Example:

```
./srxsegmentizer.sh testinput.txt output.txt
(or, using stdin/stdout)
echo "Þetta er nr. 1 og a.m.k. fínt. Farið e.t.v. þangað." | ./srxsegmentizer.sh
```

Output:

```
Þetta er nr. 1 og a.m.k. fínt.
Farið e.t.v. þangað.
```

13.3 IceTagger

To start *IceTagger*, open a terminal, go to the **bat/icetagger** directory, and type in the following command:

./icetagger.sh [parameters]

The parameters can be supplied in two ways:

- *-p <filename>*: This tells the application to read the parameters from the file *filename*. A default parameter file *paramDefault.txt* can be found in the **bat/icetagger** directory. This file has a number of attribute-value pairs whose values can be changed. The parameters are described below.

  In most cases, only the parameters *INPUT_FILE*, *OUTPUT_FILE*, *LINE_FORMAT* and *OUTPUT_FORMAT* need to be changed. To understand fully some of the other parameters you need to consult (Loftsson 2008).

  - *INPUT_FILE*: The name of the input file to be tagged. The file has a particular input format which is described by the *LINE_FORMAT* parameter.
  - *OUTPUT_FILE*: The name of the output file. The file has a particular output format which is described by the *OUTPUT_FORMAT* parameter.
  - *FILE_LIST*: The name of a file containing a list of file names (one per line) to be tagged. For each file name *F* to be tagged the corresponding tagged output file is generated in the same directory as *F* with the same name as *F* but with ".out" appended. If this parameter is used then the parameters *INPUT_FILE* and *OUTPUT_FILE* are ignored.
  - *LINE_FORMAT*: The format of the input file, 1=one token per line, 2=one sentence per line, 3=other format.
  - *OUTPUT_FORMAT*: The desired format of the output file, 1=one token per line, 2=one sentence per line.
  - *SEPARATOR*: *space/underscore*. Used for *OUTPUT_FORMAT=2*. Specifies the character used as a separator between a word and its tag.
  - *SENTENCE_START*: *upper/lower*. *upper*: Every sentence starts with an upper case letter. *lower*: Every sentence starts with a lower case letter, except when the first word is a proper noun.
  - *LOG_FILE*: The name of a log file if one is desired. The log file will list debugging information.

- *FULL_DISAMBIGUATION*: *yes/no.* This applies to words which the tagger can not fully disambiguate. If this value is *yes* the tagger will either select the tag with the highest frequency or call *TriTagger* for full disambiguation (see next parameter). If the value is *no* the tagger will return all the tags that could not be eliminated.
- *MODEL_TYPE*: *start/end/startend.* If *start*, an n-gram model (see the *MODEL* parameter) is used for choosing the word class during initial disambiguation, and then *IceTagger* is used to disambiguate tags that are consistent with the chosen word class. If *end*, the n-gram model is only run in the last phase to fully disambiguate words to which *IceTagger* is not able to assign unambiguous tags. If *startend*, the n-gram model is used both at the start and in the last phase.
- *FULL_OUTPUT*: *yes/no.* If *yes* the tagger will write subject-verb-object information and information on prepositional phrases to the output file and detailed information for unknown words. If *no* then only unknown words are marked.
- *BASE_TAGGING*: *yes/no.* If *yes* the tagger will only assign a single tag to each word based on maximum frequency.
- *TAG_MAP_DICT*: The name of the dictionary used for mapping the tags used internally by *IceTagger* to some other tagset.
- *LEMMATIZE*: *yes/no.* If *yes* then *IceTagger* outputs the lemma, in addition to the word and its tag. Note that the lemma is only written out if *OUTPUT_FORMAT*=1.
- *STRICT*: *yes/no.* Strict tokenisation or not. Used by the tokeniser, see section 13.1.
- For typical use of *IceTagger*, the user does not need to provide values for the following parameters, because as a default the corresponding files are read directly from the *IceNLPCore.jar* file:
  * *MODEL*: The name of an n-gram model. The n-gram model is only used if the *MODEL_TYPE* parameter has a value (and if *FULL_DISAMBIGUATION*=*yes*). If *MODEL_TYPE* has no value then *IceTagger* performs full disambiguation by selecting the tag with the highest frequency.
  * *BASE_DICT*: The name of the base dictionary of words and associated tags. Its format can be seen in section 12.1.3.
  * *DICT*: The name of the main dictionary of words and associated tags. Its format can be seen in section 12.1.3.
  * *IDIOMS_DICT*: The name of the dictionary for idioms or multiword expressions and associated tags.
  * *VERB_PREP_DICT*: The name of the dictionary for verb-preposition pairs and associated cases.
  * *VERB_OBJ_DICT*: The name of the dictionary for verbs and corresponding cases for their objects.
  * *VERB_ADVERB_DICT*: The name of the dictionary for verb-particle (phrasal verb) information.
  * *ENDINGS_BASE*: The name of the base dictionary listing possible tags for different endings. Used by *IceMorphy*.

    * *ENDINGS_DICT*: The name of the main dictionary listing possible tags for different endings. Used by *IceMorphy*.
    * *ENDINGS_PROPER_DICT*: The name of the main dictionary listing possible tags for different proper name endings. Used by *IceMorphy*.
    * *PREFIXES_DICT*: The name of the prefixes dictionary. Used by *IceMorphy*.
    * *TAG_FREQUENCY_FILE*: The name of the tag frequency file. This file is only used by *IceMorphy* when *BASE_ TAGGING=yes*.
    * *TOKEN_DICT*: The name of the file used by the tokeniser to recognise abbreviations, see section 13.1.

- The latter possibility is to supply the parameters through the command line. For example, by issuing commands like:

  ./icetagger.sh -i <inputFile> -o <outputFile> -d <dictionary> -lf 2 . . . , etc.

  The parameters supplied this way correspond to the attributes and values above. The name of the parameters can be seen by typing: **./icetagger.sh -help**

  For running *IceTagger* with all the default settings, issue either of the commands:

  - **./icetagger.sh** -i <inputfile> -o <outputfile>
  - **./icetagger.sh** -f <filelist>

  Here, <filelist> is a name of a file containing a list of files (one per line) to be tagged.

  If neither the -i/-o parameters nor the -f parameter are provided, *IceTagger* reads from standard input and writes to standard output. For example, the following Linux command can be used to make *IceTagger* tag the string “Ég á stóran hund” (and write the output to the screen):

  ```
  echo "Ég á stóran hund" | ./icetagger.sh
  ```

For increasing the accuracy of *IceTagger*, the main dictionary of the tagger can be extended with data from *BÍN*. Once the data from *BÍN* has been extracted (see Section 11), the parameter file **paramDefaultBin.txt** can be used for running *IceTagger* with the extended dictionary.

13.4 TriTagger

To start *TriTagger*, open a terminal, go to the **bat/tritagger** directory, and type in the following command:

./tritagger.sh [parameters]

The parameters can be supplied in two ways:

- *-p <filename>*: This tells the application to read the parameters from the file *filename*. A default parameter file *paramDefault.txt* can be found in the **bat/tritagger** directory. This file has a number of attribute-value pairs whose values can be changed:

- *INPUT_FILE*: See section 13.3.
- *OUTPUT_FILE*: See section 13.3.
- *FILE_LIST*: See section 13.3.
- *LINE_FORMAT*: See section 13.3.
- *OUTPUT_FORMAT*: See section 13.3.
- *SENTENCE_START*: See section 13.3.
- *CASE_SENSITIVE*: *yes/no*. The default is *no* which means that *TriTagger* does case-insensitive lookup into the main dictionary for the first word of a sentence. If that fails, the tagger tries case-sensitive lookup. If this parameter is set to *yes*, then case-insensitive lookup is not performed.
- *NGRAM*: *2*=bigrams, *3*=trigrams.
- For typical use of *TriTagger*, the user does not need to provide values for the following parameters, because as a default the corresponding files are read directly from the *IceNLPCore.jar* file:
  - *MODEL*: The name of the model derived from a training corpus. The model consists of a n-gram file, a lexicon and a file with lambda (smoothing) parameters. This model name should not have any extension. For example, if *MODEL*=otb, then the program will load the files *otb.ngram*, *otb.lex* and *otb.lambda* (see section 13.4.1).
  - *STRICT*: See section 13.3.
  - *TOKEN_DICT*: See section 13.3.
  - *ICEMORPHY*: *yes/no*. If *yes* then *TriTagger* uses tags guessed by *IceMorphy* for unknown words that go successfully through the morphological analysis component of *IceMorphy*. Otherwise, suffix handling of unknown words is used.
  - *DICT*: Main dictionary used by *IceMorphy*. See section 13.3.
  - *BASE_DICT*: Base dictionary used by *IceMorphy*. See section 13.3.
  - *ENDINGS_BASE*: See section 13.3.
  - *ENDINGS_DICT*: See section 13.3.
  - *ENDINGS_PROPER_DICT*: See section 13.3.
  - *PREFIXES_DICT*: See section 13.3.
- *BACKUP_DICT*: The name of a backup dictionary. If lookup into the model dictionary fails then this backup dictionary is used.
- *IDIOMS_DICT*: See section 13.3.

• The latter possibility is to supply the parameters through the command line. For example, by issuing commands like:

  ./tritagger.sh -i <inputFile> -o <outputFile> -m <model> -lf 2 . . . , etc.

  The parameters supplied this way correspond to the attributes and values above. The name of the parameters can be seen by typing: **./tritagger -help**

  For running *TriTagger* with all the default settings, issue either of the commands:

- **./tritagger.sh** -i <inputfile> -o <outputfile>
- **./tritagger.sh** -f <filelist>

Here, <filelist> is a name of a file containing a list of files (one per line) to be tagged.

If neither the -i/-o parameters nor the -f parameter are provided, *TriTagger* reads from standard input and writes to standard output. For example, the following Linux command can be used to make *TriTagger* tag the string "Ég á stóran hund" (and write the output to the screen):

```
echo "Ég á stóran hund" | ./tritagger.sh
```

For increasing the accuracy of *TriTagger*, the main dictionary of the tagger can be extended with data from *BÍN*. Once the data from *BÍN* has been extracted (see Section 11), the parameter file **paramDefaultBin.txt** can be used for running *TriTagger* with the extended dictionary.

As mentioned above, one of the files resulting from the training phase is a lexicon file (with the extension *.lex*), containing the tag profile for each word. In some cases one might want to extend this file, for example by adding data to it from some other data resource *BÍN* than the training corpus. If one does only want *TriTagger* do use data derived from the training corpus (but not also from the other data resource) for suffix handling, then a single line containing the following string can be put into the *.lex* file right after the last entry (word) derived from the training corpus:

```
[NOSUFFIXES]
```

During the loading of the lexicon, *TriTagger* will then not use entries in the lexicon, that appear after this specially marked string, for suffix handling.

13.4.1 Training

Before *Tritagger* can be used it needs to be trained on a tagged corpus. A pre-trained model (otb), derived from the *IFD* corpus, is part of the *IceNLPCore.jar* file and can also be found in the **ngrams/models** directory. For illustration, we now describe how to train a new model using any training corpus, for example the small corpus **ngrams/corpus.txt**. For training, *Perl*[3] is needed.

1. Open a terminal and go to the **ngrams** directory.
2. Type **bash train corpus.txt corpus -e**, where *bash* is a shell, *train* is the program for training, *corpus.txt* is the training corpus, *corpus* is the name of the output model and *-e* signifies empty lines between sentences in the training corpus. If all goes well, four files, corpus.ngrams, corpus.lex, corpus.orig.lex and corpus.lambda will be created in the **ngrams/models** directory.
3. At this point the file corpus.lex (and corpus.orig.lex) is a lexicon derived from the corpus.txt training corpus and can be used directly with *TriTagger* as described in section 13.4.

[3]http://www.perl.org/.

13.5 IceStagger

To start *IceStagger* for tagging text, open a terminal, go to the **bat/icestagger** directory, and type in the following command:

./icestagger.sh [parameters]

The (main) parameters are the following (the full description of the possible parameters can be found in the README file in this directory):

- -lang is: For tagging Icelandic, the value *is* is needed for the *lang* parameter.
- -modelfile <filename>: *filename* is the name of a model generated during training.
- -plain: For generating plain output, i.e. one token/tag pair per line.
- -icemorphy <n>: *n* is **0** (do not use IceMorphy), or **1** (use IceMorphy for filling *tag profile gaps* and guessing the tag profile for unknown words), or **2** (only use IceMorphy for unknown words).
- -tag <filename 1> <filename 2> ... <filename n>: Specifies tagging of *n* files. This should be the last argument.

There are two possible formats for the input files to be tagged:

- A file with an *.txt* extensions is assumed to contain raw text and will be tokenised by *IceStagger's* tokeniser before tagging. If there is only one file name in the input list, the output is written to standard output, otherwise each tagging output is written to a separate file.
- A file with any other extension is assumed to contain a single token/tag pair in each line with an empty line between sentences. The tag in the second column is used for evaluating the tagger's accuracy. In this case, the tagging output is written to standard output.

For example, the following command:

```
./icestagger.sh -modelfile otb.bin -lang is -plain -icemorphy 1 -tag sentences.txt
```

uses the training model *otb.bin* to tag the *sentences.txt* file using *IceMorphy*, generating *plain* (one token/tag per line) output.

13.5.1 Training

To generate a model from a training corpus, the following (main) parameters can be used:

- -lang is: For training on Icelandic text.
- -trainfile <filename>: *filename* is the training corpus to be used. The format is assumed to be one token/tag pair in each line with an empty line between sentences.
- -lexicon <filename>: *filename* is a lexicon in which each line has 4 tab-separated fields: <word form, lemma, tag, frequency>. The frequency can be 0. The lexicon is optional.

- -positers <n>: Train the tagger with at most *n* iterations.
- -plain: For generating plain output, i.e. one token/tag pair per line.
- -train: Specifies training mode. This should be the last argument.

For example, the following command:

```
./icestagger.sh -trainfile otb.plain -modelfile otb.bin -positers 10 -lang is -train
```

uses the training corpus *otb.plain* to produce the training model *otb.bin* using 10 iterations.

A pre-trained model (otb), derived from the *IFD* corpus, is part of the *IceNLP* distribution and can be found in the **models** directory of the *bat/icestagger* directory.

For increasing the accuracy of *IceStagger*, a lexicon with data from *BÍN* can be provided during training. Once the data from *BÍN* has been extracted (see Section 11), the shell script **trainIceStaggerBin.sh** can be used for training. Tagging can then be carried out using the **tagIceStaggerBin.sh** shell.

13.6 Lemmald

The lemmatizer can be used as part of *IceTagger* by supplying the *-lem* parameter and specifying output format *1*. See section 13.3 on *IceTagger* usage for further information. An example of such usage is the following:

```
echo "Ég á stóran hund" | ./icetagger.sh -of 1 -lem
```

The same result can be achieved using **./lemmatize.sh** and that command also allows for lemmatizing input that has already been tagged, for example using a different tagger. The parameters of **./lemmatize.sh** are the following:

- *-i<file>*: The input file. If omitted the input is read from stdin.
- *-o<file>*: The output file. If omitted the output is written to stdout.
- *-h*: Display help.
- *-lemmatizeTagged*: Indicates that the input is already tagged. Such input should have one token per line and each token should consist of a word and its tag.

Example 1: Lemmatizing a plain text file

```
./lemmatize.sh -i plaintext.txt -o myoutput.txt
(or, using stdin/stdout)
echo "Við erum æðislegar. Við kunnum alla dansana." | ./lemmatize.sh
```

Reads the plain text file plaintext.txt and writes the result to myoutput.txt. *IceTagger* is used for tagging before *Lemmald* lemmatizes.

Input:

```
Við erum æðislegar. Við kunnum alla dansana.
```

Output:

```
Við ég fp1fn
erum vera sfg1fn
æðislegar æðislegur lvfnsf
. . .

Við ég fp1fn
kunnum kvinna sfg1fþ
alla allur fokfo
dansana dans nkfog
. . .
```

Example 2: Lemmatizing input that is already tagged

To lemmatize tagged input, with one token per line, each of which has a word form and a PoS tag, supply the parameter "-lemmatizeTagged". The lemma is added between the word form and its tag.

```
./lemmatize.sh -i testinput.txt -o output.txt -lemmatizeTagged
(or, using stdin/stdout)
cat testinput.txt | ./lemmatize.sh -lemmatizeTagged
```

Input:

```
Ég fp1en
á sfg1en
stóran lkeosf
hund nkeo
```

Output:

```
Ég ég fp1en
á eiga sfg1en
stóran stór lkeosf
hund hundur nkeo
```

13.7 IceMorphy

The morphological analyser, *IceMorphy*, can be used as a stand-alone application. To start *IceMorphy*, open a terminal, go to the **bat/icemorphy** directory, and type in the following command:

./icemorphy.sh -p <paramFile>

The format of the parameter file is similar to the format of the file used by *IceTagger*. Two default parameter files *paramAnalyze.txt* and *paramFill.txt* can be found in the **bat/icemorphy** directory. The former is used for analysing words in a file, the latter for filling *tag profile gaps* in a dictionary:

- **Analysing**. In this mode *IceMorphy* accepts an input file consisting of one word in each line. It looks up each word in the supplied dictionary (see the *DICT* parameter) and fetches the corresponding tags if the word is found or guesses the possible tags if the word is unknown. Unknown words are marked with a * at the end of each line in the output file. Additionally, one of the strings <MORPHO>, <COMPOUND> or <ENDING> are printed after the *, signifying which module of *IceMorphy* produced the result (see Sect. 4). The analyser either returns all tags for each word (sorted by frequency) or only the most frequent tag. This can be controlled by the *MODE* parameter.

- **Filling**. In this mode *IceMorphy* accepts an input file (a dictionary) in the format described in section 12.1.3. For each word in the input file, the morphological analyzer generates the missing tags, i.e. it does *tag profile gap* filling.

The parameters of the <paramFile> are described below:

- *MODE*: *all/one/fill.* all=analyze words and return all tags, one=analyze words and return the one most frequent tag, fill=fill tag profile gaps in a dictionary.
- *INPUT_FILE*: The name of the input file to be either *analysed* or *filled.*
- *OUTPUT_FILE*: The name of the output file.
- *LOG_FILE*: The name of a log file if one is desired. The log file will list debugging information.
- *SEPARATOR*: *space/equal.* Specifies the character used as a separator between a word and its tag(s).
- *TAGSEPARATOR*: *space/underscore.* Specifies the character used as a separator between the tags.
- For typical use of *IceMorphy*, the user does not need to provide values for the following parameters, because as a default the corresponding files are read directly from the *IceNLPCore.jar* file:
  - *DICT*: The name of the main dictionary of words and associated tags. See section 13.3.
  - *BASE_DICT*: The name of the base dictionary. See section 13.3.
  - *ENDINGS_BASE*: See section 13.3.
  - *ENDINGS_DICT*: See section 13.3.
  - *ENDINGS_DICT*: See section 13.3.
  - *ENDINGS_PROPER_DICT*: See section 13.3.
  - *PREFIXES_DICT*: See section 13.3.
  - *TAG_FREQUENCY_FILE*: See section 13.3.

13.8 IceParser

To start the parser, open a terminal, go to the **bat/iceparser** directory and type in the following command:

./iceParser.sh -i <inputFile> -o <outputFile> [optional param]

The optional parameters are:

- *-f*: *IceParser* annotates grammatical functions (as well as constituent structure).
- *-l*: *IceParser* writes out one phrase/syntactic function in each line. Otherwise, the output is one sentence per line.
- *-a*: *IceParser* uses feature agreement rather than only relying on word order, when grouping words into noun phrases and annotating subjects of verbs.
- *-e*: *IceParser* attaches a question mark (?) to the end of labels for NPs and/or subjects to denote possible grammatical errors.
- *-m*: *IceParser* merges function labels with phrase labels.
- *-json*: *IceParser* writes the output in json format.
- *-xml*: *IceParser* writes the output in xml format.

Note that *IceParser* assumes that the input file has one sentence per line. Each line consists of a sequence of word-tag pairs (see 12.2).

A *grammar definition corpus*, a representative collection of about 200 Icelandic sentences (Loftsson and Rögnvaldsson 2006) is provided in the **bat/iceparser** directory. The name of the file is *200sent_func.gdc* and it has been hand-annotated with constituent structure and grammatical functions. The original text is in the file *200sent.txt*.

The following command makes *IceParser* annotate the original file with constituent structure and grammatical functions:

```
./iceParser.sh -i 200sent.txt -o 200sent.out -f -l
```

The hand-annotated file *200sent_func.gdc* and the parser generated file *200sent.out* can then be compared by using utilities like Unix *diff*.

IceParser can, additionally, be made to generate output files corresponding to the result of each of its individual finite-state transducers. In that case, type in:

```
./iceparserOut.sh -i 200sent.txt -o 200sent.out -p .
```

The third command-line parameter above denotes the path for the output files. The output files are text files with the *.out* ending.

13.9 IceNER

To start *IceNER*, open a terminal, go to the **bat/iceNER** directory and type in the following command:

```
./iceNER.sh -i <inputFile> -o <outputFile> [optional param]
```

The optional parameters are:

- *-l <filename>*: *IceNER* uses <filename> as a gazette list (a list which contains pre-catagorised entities).
- *-g*: *IceNER* runs in greedy mode. In this mode, all unmarked named entities that follow the prepositions "á" and "í" are marked as locations and names with the pattern "Xxxx Xxxx" are marked as persons.

13.10 Dictionaries

The dictionaries used by the system are located in the **dict** directory. The dictionaries which start with the prefix *otb* have been automatically generated from the *IFD* corpus. For example, the main dictionary, *dict/icetagger/otb.dict*, was generated by extracting all the words from the *IFD* corpus along with all the tags that appeared with each word. The format of this dictionary is described in section 12.1.3.

Two base dictionaries are used by the system. These are *dict/icetagger/baseDict.dict* and *dict/icetagger/baseEndings.dict*. The former is mainly used for words and associated tags of the closed word classes, e.g. conjunctions, pronouns, prepositions and irregular verbs. A word is first looked up in this base dictionary before the main dictionary (*DICT*) is searched.

The latter is a hand-compiled list of endings and associated tags. An ending is first looked up in this list before the endings dictionary supplied by the user (*ENDINGS_DICT*) is searched.

14 Demo application

A small demo application is part of this release. The purpose of the application is to analyse (tag and parse) text specified by the user. To start the application, open a terminal, go to the **bat/demo** directory and type in the following command:

./tagAndParseGUI.sh [inputFile]

The input file is optional. If not input file is specified, it is assumed that the user will type in the text to be analysed.

For example, the file *test.txt* in the **bat/demo** directory can be analysed, by typing:

```
./tagAndParseGUI.sh test.txt
```

Tagging and parsing can also be tested by running the **/.tagAndParse.sh** command in the **bat/demo** directory.

In that case, the *test.txt* file is used as the input to the tagger. The output of the tagger is then piped into *IceParser*, which finally produces the file *parse.out* as output.

15 Building from source

To build *IceNLP* from source, you need the following three tools:

1. **Java Development Kit (JDK)**. The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform. JDK is available for free from Oracle.

2. **JFlex**. JFlex is a lexical analyzer generator (also known as scanner generator) for Java, written in Java. JFlex is availble for free from `http://jflex.de`

3. **Apache Ant**. Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. Ant is available for free from `http://ant.apache.org/`

For example, to build *IceNLPCore*, go to the directory **icenlp/core** and issue this command: **ant**

Ant will then use the instructions given in the *build.xml* file to build each individual component of *IceNLP*.

Note that before building you will need to increase the memory used by *JFlex*: Go to the directory of JFlex and edit the *jflex* file. At the bottom of this file, change:

```
$JAVA -Xmx128m -jar $JFLEX_HOME/lib/jflex-1.x.y.jar $@
```

to

```
$JAVA -Xmx2048m -jar $JFLEX_HOME/lib/jflex-1.x.y.jar $@
```

References

S. Aït-Mokhtar and J.-P. Chanod. Incremental Finite-State Parsing. In *Proceedings of Applied Natural Language Processing*, Washington DC, USA, 1997.

K. Bjarnadóttir. Modern Icelandic Inflections. In H. Holmboe, editor, *Nordisk Sprogteknologi 2005*. Museum Tusculanums Forlag, Copenhagen, 2005.

T. Brants. TnT: A statistical part-of-speech tagger. In *Proceedings of the* 6th *Conference on Applied Natural Language Processing*, Seattle, WA, USA, 2000.

M. Collins. Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing*, Philadelphia, PA, USA, 2002.

S. Helgadóttir. Testing Data-Driven Learning Algorithms for PoS Tagging of Icelandic. In H. Holmboe, editor, *Nordisk Sprogteknologi 2004*. Museum Tusculanums Forlag, Copenhagen, 2004.

A. Ingason, S. Helgadóttir, H. Loftsson, and E. Rögnvaldsson. A Mixed Method Lemmatization Algorithm Using Hierachy of Linguistic Identities (HOLI). In B. Nordström and A. Rante, editors, *Advances in Natural Language Processing,* 6th *International Conference on NLP, GoTAL 2008, Proceedings*, Gothenburg, Sweden, 2008.

H. Loftsson. Tagging a Morphologically Complex Language Using Heuristics. In T. Salakoski, F. Ginter, S. Pyysalo, and T. Pahikkala, editors, *Advances in Natural Language Processing,* 5th *International Conference on NLP, FinTAL 2006, Proceedings*, Turku, Finland, 2006.

H. Loftsson. *Nordic Journal of Linguistics*, 31(1):2008.

H. Loftsson. Correcting a PoS-tagged corpus using three complementary methods. In *Proceedings of the* 12th *Conference of the European Chapter of the ACL (EACL 2009)*, Athens, Greece, 2009.

H. Loftsson and R. Östling. Tagging a Morphologically Complex Language Using an Averaged Perceptron Tagger: The Case of Icelandic. In *Proceedings of the* 19^{t}*h Nordic Conference of Computational Linguistics (NODALIDA-2013)*, Oslo, Norway, 2013.

H. Loftsson and E. Rögnvaldsson. A shallow syntactic annotation scheme for Icelandic text. Technical Report RUTR-SSE06004, Department of Computer Science, Reykjavik University, 2006.

H. Loftsson and E. Rögnvaldsson. IceParser: An Incremental Finite-State Parser for Icelandic. In *Proceedings of NoDaLiDa 2007*, Tartu, Estonia, 2007.

H. Loftsson, I. Kramarczyk, S. Helgadóttir, and E. Rögnvaldsson. Improving the tagging accuracy of Icelandic text. In *Proceedings of the* 17th *Nordic Conference of Computational Linguistics (NODALIDA-2009)*, Odense, Denmark, 2009.

H. Loftsson, S. Helgadóttir, and E. Rögnvaldsson. Using a morphological database to increase the accuracy in PoS tagging. In *Proceedings of Recent Advances in Natural Language Processing (RANLP 2011)*, Hissar, Bulgaria, 2011.

A. Mikheev. Automatic Rule Induction for Unknown Word Guessing. *Computational Linguistics*, 21(4):543–565, 1997.

P. Nakov, Y. Bonev, G. Angelova, E. Cius, and W. Hahn. Guessing Morphological Classes of Unknown German Nouns. In *Proceedings of Recent Advances in Natural Language Processing*, Borovets, Bulgaria, 2003.

R. Östling. Stagger: an Open-Source Part of Speech Tagger for Swedish. *Northern European Journal of Language Technology*, 3(1):1–18, 2013.

D. Palmer. Tokenisation and Sentence Segmentation. In R. Dale, H. Moisl, and H. Somers, editors, *Handbook of Natural Language Processing*. Marcel Dekker, New York, 2000.

J. Pind, F. Magnússon, and S. Briem. *Íslensk orðtíðnibók [The Icelandic Frequency Dictionary]*. The Institute of Lexicography, University of Iceland, Reykjavik, Iceland, 1991.

A. Tryggvason. Named Entity Recognition for Icelandic, 2009. MSc-thesis, School of Computer Science, Reykjavik University. `http://nlp.ru.is/pdf/NamedEntityRecognitionforIcelandic.pdf`.

A The Icelandic tagset

Table 2: The Icelandic tagset

Char#	Category/Feature	Symbol – semantics
1	Word class	**n**–noun
2	Gender	**k**–masculine, **v**–feminine, **h**–neuter, **x**–unspecified
3	Number	**e**–singular, **f**–plural
4	Case	**n**–nominative, **o**–accusative, **þ**–dative, **e**–genitive
5	Article	**g**–with suffixed definite article
6	Proper noun	**s**–proper name
1	Word class	**l**–adjective
2	Gender	**k**–masculine, **v**–feminine, **h**–neuter
3	Number	**e**–singular, **f**–plural
4	Case	**n**–nominative, **o**–accusative, **þ**–dative, **e**–genitive
5	Declension	**s**–strong declension, **v**–weak declension, **o**–indeclineable
6	Degree	**f**–positive, **m**–comparative, **e**–superlative
1	Word class	**f**–pronoun
2	Subcategory	**a**–demonstrative, **b**–reflexive, **e**–possessive, **o**–indefinite, **p**–personal, **s**–interrogative, **t**–relative
3	Gender/Person	**k**–masculine, **v**–feminine, **h**–neuter/**1**–1^{st} person, **2**–2^{nd} person
4	Number	**e**–singular, **f**–plural
5	Case	**n**–nominative, **o**–accusative, **þ**–dative, **e**–genitive
1	Word class	**g**–article
2	Gender	**k**–masculine, **v**–feminine, **h**–neuter
3	Number	**e**–singular, **f**–plural
4	Case	**n**–nominative, **o**–accusative, **þ**–dative, **e**–genitive
1	Word class	**t**–numeral
2	Category	**f**–alpha, **a**–numeric
3	Gender	**k**–masculine, **v**–feminine, **h**–neuter
4	Number	**e**–singular, **f**–plural
5	Case	**n**–nominative, **o**–accusative, **þ**–dative, **e**–genitive
1	Word class	**s**–verb (except for past participle)
2	Mood	**n**–infinitive, **b**–imperative, **f**–indicative, **v**–subjunctive, **s**–supine, **l**–persent participle
3	Voice	**g**–active, **m**–middle
4	Person	**1**–1^{st} person, **2**–2^{nd} person, **3**–3^{rd} person,
5	Number	**e**–singular, **f**–plural
6	Tense	**n**–present, **þ**–past
1	Word class	**s**–verb (past participle)
2	Mood	**þ**–past participle
3	Voice	**g**–active, **m**–middle
4	Gender	**k**–masculine, **v**–feminine, **h**–neuter
5	Number	**e**–singular, **f**–plural
6	Case	**n**–nominative, **o**–accusative, **þ**–dative, **e**–genitive
1	Word class	**a**–adverb and preposition
2	Category	**a**–does not govern case, **u**–exclamation, **o**–governs accusative, **þ**–governs dative, **e**–governs genitive
3	Degree	**m**–comparative, **e**–superlative
1	Word class	**c**–conjunction
2	Category	**n**–sign of infinitive, **t**–relative conjunction,
1	Word class	**e**–foreign word
1	Word class	**x**–unanalyzed word