OCRPostcorrection Documentation

Release 1

Sarah Schulz

Sep 18, 2017

CONTENTS

Contents:

CHAPTER

ONE

TUTORIAL

Contributors: Sarah Schulz Based upon a System for Normalization of User-Generated Text developed in collaboration with Bart Desmet and Orphee DeClercq Version 1.0

Date September 2017

contact: sarah.schulz@ims.uni-stuttgart.de

The system reads plain text that has been OCRed and returns a version in which errors have been corrected. For information on the training data etc. check out the EMNLP paper at http://aclanthology.info/papers/D17-1287/d17-1287

The repository does not contain all models due to size limits. If you want to use them, please, send me a mail.

1.1 System architecture

The pipeline consists of different more or less independent modules.

- the preprocessing module
- the suggestion-generation modules
- the decision module

1.1.1 Preprocessing

The first step that is performed is the preprocessing of the input files.

The program works on the document line level. The tokenization is done with the TreeTagger tokenizer.

1.1.2 Suggestion generation

There are different modules that generate possible correction option for each word in a line. Some modules generate exactly one output for each word, others do not deliver an option for every word but can deliver more than one possible option per word.

The modules cover different levels of recognition errors that can appear OCRed text.

Compound

It is often the case that spaces are introduced where there should not be any.

To account for this compounding mistakes the compound module tests for all two words that are written next to each other if a spell checker (hunspell) recognizes them as correctly spelled word when they are written as one word. If that is the case, the compound version is returned as a possible option for the two words.

The output of this module is a phrase (the two original words) along with a one word option. This is possible since we later one work with phrase-based machine translation. If there is no compound correction in the input sentence the output is an empty list.

Original

The original module returns the original word as an option for the word. This is important since some of the words are not erroneous and could get lost when no other module returns the original option.

The output of this module is the original word along with exactly one option (which is the same word as the input word).

SMT

Following preliminary experiments described in DeClercq (2013), the SMT models have been trained on token and character level using Moses. The language model used has been built from a selection of German Gutenberg texts using KenLM.

There are four different modules that work with statistical machine translation on character and token level.

All those modules work with Moses models. Since the program works on the sentence level we included Moses sever mode in order to avoid loading the model files of Moses for each sentence.

Spell checker

The spell checker module uses hunspell (and its python wrapper pyhunspell). Each word in the original sentence is spell checked. In case hunspell classifies a word as wrongly-spelled, the correction suggestions given by hunspell are returned as alternatives. The output of the spell checker module is a list of spell checker suggestions for each wrongly-spelled word.

Word split

Theword split module is the opposite of the compound module and splits words that have been erroneously written together.

The word split module is based on the compound-splitter module of Moses and has been trained on the Gutenberg corpus. It often appears that words that are actually two words are written together.

Specific vocab

Extracts words that are frequent from the erroneous text as a basis for word to word comparison via Levenshtein. Makes e.g. the correction of Named Entities possible.

1.1.3 Decision module

Since we have no a-priori knowledge about the nature of a normalization problem, each sentence is sent to all modules of the suggestion layer. In order to prevent an avalanche of suggestions, for each module of the suggestion layer, we restricted the number of suggestions per token to one.

It is the task of the decision module to choose the most probable combination of suggestions to build a well-formed sentence, which poses a combinatorial problem. The decision module itself makes use of the Moses decoder, a powerful, highly efficient tool able to solve the combinatorial problem with the aid of a language model of standard languagein order to include probability information.

We include the correction suggestions in form of a phrase table into the decoding process. The decoder weights have been manually tuned using development data. However, the weights might not be ideal for decoding the dynamically compiled phrase tables containing the normalization suggestions. We include weights for the different components of the decoding process like the language model. the translation model, word penalty and distortion. Distortion is made expensive since we want to avoid reordering. Moreover, the decoder uses features containing information about which module returned which suggestion. The feature weights are first set to 0.2 and later tuned on the development data

LOGFILES:

Every evaluation run, will create a variety of files:

log files: those files you can find in ../log/runs and will have the starting date and time as a name

They log the output list for every module for every sentence and several other stuff.

1.2 Example of usage

BEFORE GETTING STARTED:

check if there is a server mode running without you knowing it. In general, the system should kill them in the end of a run or in case of an error, but I don't know why, sometimes that fails.

ps aux | grep moses

–> kill -9 pids

The system requires 3 arguments:

```
run_norm.py <ini_file> <inputfile> <outputfile>
```

1.2.1 Ini file

The ini files is the specification file of the system. You have to adjust it to your settings before you run normalization. Make sure you use absolute pathes (if you use relative pathes they have to be relative to your norm directory). There are two default ini files included in the system. In case you want to run default settings, you can leave them as they are. You find them in norm/ini_files. The following options have to be specified:

- [MOSES-PATH] define where Moses is installed on your machine
  - Moses: where is Moses itself installed
  - Mosesserver: where is Mosesserver installed
- [SMT] define where the ini files for the different MT systems are located
  - Token: define where the ini file the MT token model is located
  - Unigram: define where the ini file the MT token model is located
  - Bigram: define where the ini file the MT token model is located
  - Decision: define where the ini file the MT token model is located

- [SETTING] define if you use unbalanced or balanced file or just SMS, TWE, SNS ...
  - train_set: sets the setting for the transliterate module
- [LM] define where to find the lm used in the decision module
  - lm
- [Modules] define which modules should be used, this has to match your decision ini file (number of weights)
  - mod: give the modules separated by white space, the order should be the same as the order you had while compiling the tuning table on the development set. You can choose out of the following: Original SMT_Token SMT_Token2 SMT_Unigram SMT_Unigram2 SMT_Cascaded Hunspell LM Compound Word_Split

CHAPTER

TWO

DEVELOPMENT

You might bump into situations in which you want to use a new setting for which there does'nt exist a decison.ini file, yet. This situations can be:

- you use a different combination of modules

What do you have to do then:

1. fix your ini file
    - SMT inis in the right setting
    - same setting for train_set
    - put develoment switch dev to True (make sure you dont have anything in ../log/phrasetables that you might need later)
    - check if you have the right modules involved (the order of the modules that you define here, will be the order you have to keep whenever you use this setting)
    - put the right filtering method
    - in the decision file, make sure you have the same number of feature weights in it as number of modules + 1 (hard, none filtering) +3 (soft filtering)
2. run the system on the development file without evaluation setting:

    run_norm.py <ini_file> <input_file> <output_file>

3. onces you have all phrase tables, combine them to one phrase table:

    cat ./log/phrasetables/* > all_phrasetables

4. make a new folder which you want to use for tuning (I used /home/sarah/Normalization/tuning)
5. copy the decision_untuned.ini from one of my folders there and make sure that you include the right number of weights (number of modules +1 or +3 (soft filtering))
6. adjust the pathes in the following command and start tuning by doing so:

    /opt/moses/moses_with_kenlm_10_compact/mosesdecoder-RELEASE-2.1.1/scripts/training/mert-moses.pl /home/sarah/Normalization/tuning/tune_files/all_dev_clean.ori.prepro /home/sarah/Normalization/tuning/tune_files/all_dev_clean.tgt.prepro /opt/moses/moses_with_kenlm_10_compact/mosesdecoder-RELEASE-2.1.1/bin/moses decision_untuned.ini –mertdir /opt/moses/moses_with_kenlm_10_compact/mosesdecoder-RELEASE-2.1.1/bin/ &> mert.out

7. wait
8. you find the new decision.ini in the mert folder. It is called moses.ini. They are on the bottom and look like that:

KENLM0= 0.000984536 Distortion0= 0.00987259 WordPenalty0= -0.946984 PhraseDictionaryMemory0= 0.00269802 0.00264633 0.00203958 0.00304969 0.00890275 0.00344056 0.0112828 -0.00027502 -0.00314784 0.00212978 0.00254623

9. open one of the already existing ini file in the static/Moses/decoder_files directory and put your weights there. This way you make sure that the other pathes are still correct; you should copy the feature weights in one of the old decision ini files to keep the old format. the new format doesn't work with it.
    - KENLM0 is [weight-l]
    - Distortion0 is [weight-d]
    - WordPenalty0 is [weight-w]
    - PhraseDictionaryMemory0 is [weight-t]

2.1 Trouble shooting of knowing where trouble often emerges

- although the system worked the last 1000 times there is no garantee it will work the next time! DON'T ASK ME WHY
- still, some stuff I might have seen before. So, if you can't find the solution after some search, mail me
- check if all you input files and input pathes do really exist (and by that I mean: verify with bash)
- if you get an error in the decision module it is highly probable that one of the files like LM or phrase table does not exist, or the number of weights is incorrect
- never use old and new ini file format together. The decision module works with the old format (if you ever want to change that, you will have to change the command line

arguments in the decision module subprocess call to the new commandline way of reading in phrase tables and language models) * if one of the SMT modules crashes, there might be problems with the server modes. Check in the log files if they started correctly * while tuning you might run into a problem with the filtering method of Moses. I don't really know a direct solution for that. But google it. You are not alone * should you ever run on thoth: be careful, the Moses path is different here * be sure that the tuning order of the modules is the same as the running order (otherwise the weights are wrong) * in case a key cannot be found during evaluation, make sure there are no weird characters like u"u200e" envolved in csv or ori file * in case you use evaluation and there is a problem with one specific sentence although the others ran through: check the csv file. There might be something weird in the annotation. Btw: the run_norm.py script checks now if all sentences in input can be found as key in dict before starting. If not, it will throw and exception. * I know that the alignment method is error prone, so be aware that mistakes can come up there

CHAPTER

THREE

SYSTEM REQUIREMENTS

The system has been tested on a 64-bit Ubuntu machine. The following tools and packages have to be installed and the paths have to be adjusted in order to run the normalizer.

- Moses (compiled with server mode and SRILM setting)
- xmlrpc-c 1.25.23 (Moses has to be compiled with the –with-xmlrpc-c in order to include it)
- hunspell
- pyhunspell 0.1
- pytest
- scikit-learn
- nltk
- kenlm

CHAPTER

FOUR

API

4.1 Preprocessing

4.2 Main

4.3 Modules

class norm.modules.spellcheck.**Hunspell**(*normalizer*)

This class contains functions that check a word with the hunspell spell checker. In case it is recognized as incorrectly spelled, alternative spelling options are suggested

find_suggestions(*word*)

use the hunspell spell checker for correct suggestions. take the first suggestion (levenshtein distance smallest).

parameters, types,return**,**return types**::**

param word a token

type sentence unicode string

return hunspell corrected suggestion

rtype unicode string

generate_alternatives(*sentence*, *corr_list*)

Generate suggestion

parameters, types,return**,**return types**::**

param sentence flooding corrected original message

type sentence unicode string

param corr_list list containing all indices of the tokens that have to be normalized (hard filtering just some, otherwise all)

type corr_list list of integers

return original tokens aligned with the suggestion of the form [[ori,[sug]],[ori2,[sug2]]]

rtype list of lists

4.4 Evaluation

4.5 Util

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n