NoUGaT Documentation

Release 1

Sarah Schulz, Bart Desmet, Orphee DeClercq

February 05, 2015

CONTENTS

Contents:

CHAPTER

ONE

TUTORIAL

Contributors: Sarah Schulz, Bart Desmet, Orphee De Clercq, Arda Tezcan, Guy De Pauw

Version 1.0

Date 15th November 2014

contact: Sarah.Schulz@Ugent.be

The Normalizer converts User Generated Contents into a normalized version that can be further processed with NLP tools. It has been trained on three different genres of UGC namely Twitter data, SMS and Netlog. It works for Dutch as well as for English.

1.1 System architecture

The normalization 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 sentence level. That means that the input is split into sentences first.

The following replacements of characters talk place: * smilies * tags * hyperlink

1.1.2 Suggestion generation

There are 8 different modules that generate possible normalization option for each word in a sentence. 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 mistakes that can appear in UGC such as spelling errors or phonetic expressions.

Abbreviation

Language used in UGC often shares certain abbreviations and uniform ways of reference like hash tags in Twitter posts. Therefore, lookup approaches can cover a reasonable number of issues.

The abbreviation module relies on a dictionary of about 350 frequent abbreviations appearing in social media texts like *lol* (laughing out loud) and *aub* for (alstublieft) (thank you).

Compound

It is often the case (and certainly more often for Dutch than for English) that compounds that should actually be written in one word are written in two words.

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.

Empty

A module which is not included by default. It returns the empty string for each token in order to give the possibility to delete tokens.

Named entity module

This is a module that differs from the other modules. It does not give options but returns information that can be used later as a feature. By default, this module is not included.

Named entities (NEs) should not be normalized and it is therefore important to recognize them as such in order to avoid overcorrection. Since NEs in UGC have different characteristics than in standard texts (NEs frequently lack capitalization or are introduced with specific characters as @ or #), we developed a dedicated named entity recognition (NER) tool.

The NER tool is hybrid in the sense that it uses gazetteer lookup and classification. The gazetteers contain a variety of named entities. Moreover, it includes a simple pattern-matching rule to find words with a capitalized first letter which does not appear at the beginning of a sentence.

> Other than abbreviations, NEs are a highly productive group, so a lookup approach does not suffice.

For that reason, we added a conditional random field classifier trained on the training set of our corpus. It reaches a high performance (F-score of 0.9) and has been trained with features tailored to named entities in UGC.

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).

G2P2G

UGC often contains mistakes that are due to the "spell like you pronounce it" style. Often words are a graphematic representation of how a word is pronounced. To be able to translate those words to the correct spelling we first have to find a phonetic representation that is close to the graphematic representation and then use a lookup dictionary to find the correct orthographic representation of the similar phonetic string.

The module performs the following steps:

1. The graphematic original word is with the help of a memory-based translation system translated into its phonetic representation
2. certain replacement strategies borrowed from spell checking approaches are used in order to generate similar phonetic strings (namely splits, deletes, transposes, replaces, inserts and of course also the phonetic representation of the original string)
3. all the phonetic alternatives are looked up in a dictionary containing phonetic strings along with its grapheme translations.
4. all grapheme translations that can be found for the phonetic alternatives are returned as possible options of the original word

(implemented by Guy De Pauw)

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 combination of four corpora using KenLM.

There are four different modules that work with statistical machine translation.

- unigram module
- bigram module
- token module
- cascaded module

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. In UGC words are often concatenated in order to save space.

The word split module is based on the compound-splitter module of Moses and has been trained on the Corpus Gesproken Netherlands (CGN). It often appears that words that are actually two words are written together.

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 normalization 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

1.2 Example of usage

The system requires 3 arguments and is run like this:

run_norm.py <lang> <inputfile> <outputfile>

The two last arguments serve evaluation and can be ignored Run it from the norm directory. The log files will be located in the log directory after the run. In case you have questions, ask me.

- <lang>: can take value “nl” and “en”
- <inputfile>: inputfile is a text file containing one messages per line. Those will be handeled as one to not lose context
- <outputfile>: the file to which the normalized messages will be written.

The system can also just be run in preprocessing mode:

run_prepro.py <lang> <inputfile> <outputfile>

CHAPTER

TWO

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

CHAPTER

THREE

API

3.1 Preprocessing

class norm.data.**Text**(*text*, *n*, *r*)

Represents a text, such as an SMS, a blogpost or a tweet

_preprocess()

preprocess input text (tokenization, special character replacement)

_validate_input(*t*)

validate that input string is unicode

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

param t an input message

type t unicode string

return input is a unicode string or not

rtype boolean

class norm.prepro.rewrite.**Rewrite**(*normalizer*)

This class preprocesses the text including tokenization, special character replacement, deletion of more than one whitespace.

INPUT: line (@janthans @SvenOrnelis BBQ? Wie, wat , waar? :-D #aanwezig)

OUTPUT: tokenized sentence with replacements (@janthans @SvenOrnelis BBQ? Wie, wat , waar? • aanwezig)

Placeholders list sign "•"

correct_at_tok(*t*)

correct missed tokenization with @

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

param t input text

type t Unicode string

return the string with whitespace deleted after @

rtype Unicode string

reduce_allcaps(*t*)

lowercase sequences of more than 1 uppercased letter

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

param t input text

type t Unicode string

return the string with replacement characters for smileys

rtype Unicode string

replace_hashtags(*t*)
replace #tags like #workisboring with a special character

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

param t input text

type t Unicode string

return the string with replacement characters for hash tags

rtype Unicode string

replace_hyperlinks(*t*)
replace hyperlink with replacement character

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

param t input text

type t Unicode string

return the string with replacement characters for hyperlinks

rtype Unicode string

replace_linebreaks(*t*)
delete the string <LINEBREAK>

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

param t input text

type t Unicode string

return the string without <LINEBREAK>

rtype Unicode string

replace_pipe(*t*)
replace | wirht ” “

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

param t input text

type t Unicode string

return the string with replacement of |

rtype Unicode string

replace_smileys(*t*)
replace all smiley characters with a special character

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

param t input text

type t Unicode string

return the string with replacement characters for smileys

rtype Unicode string

replace_spaces(*t*)
replace all spaces with just one space

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

param t input text

type t Unicode string

return the string with max one whitespace in a row

rtype Unicode string

replace_tags(*t*)
replace [*] with replacement character

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

param t input text

type t Unicode string

return the string with replacement characters for tags

rtype Unicode string

rewrite_text(*t*, *norm*)
call all the replacement and preprocesing methods on the input

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

param t input text

type t Unicode string

param norm a normalizer for language information

type norm Normalizer object

return the preprocessed string

rtype Unicode string

tokenize(*t*, *norm*)
tokenize the text with a special pretokenizer and a perl script from TreeTagger

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

param t input text

type t Unicode string

param norm a normalizer for language information

type norm Normalizer object

return the tokenized string

rtype Unicode string

tokenize_placeholders(*t*)
correct the tokenization of the special characters

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

param t input text

type t Unicode string

return string with special characters being tokenized

rtype Unicode string

tolower(*t*)
lowercase string

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

param t input text

type t Unicode string

return the string in lowercase

rtype Unicode string

class norm.modules.flooding.**Flooding**(*normalizer*)
this class corrects the flooding of characters and punctionation, it reduces flooding to one and two characters and checks whether a correct word emerges with the help of spell checking. In case it does it returns the whole sentence It also corrects punctuation flooding. It does that in any case.

check_for_correctness(*token*)
Check if token is marked as correct by hunspell or is found in the abbreviation dictionary.

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

param t original token

type t unicode string

return word is correct word or not

rtype boolean

correct_flooding_to_one(*t*)
Correct flooding characters using regex matches to one repetitions.

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

param t original token

type t unicode string

return tuple: first part gives information if the suggested token is marked as correct by hunspell, corrected token

rtype tuple(boolean, string)

Reduce all character flooding to one subsequent characters.

correct_flooding_to_two(*t*)
Correct flooding characters using regex matches to two repetitions.

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

param t original token

type t unicode string

return tuple: first part gives information if the suggested token is marked as correct by hunspell, corrected token

rtype tuple(boolean, string)

Reduce all character flooding to two subsequent characters. For Dutch the e is corrected to 3 repetitions first to check if an existing word emerges.

correct_punctuation_flooding(*t*)
Correct flooding punctuations using regex matches.

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

param t original token

type t unicode string

return punctuation flooding corrected token

rtype unicode string

Reduce all punctuation flooding to two subsequent characters, just dots are corrected to three.

flooding_correct(*t*)
Correct flooding characters and character combinations in t.

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

param t original message

type t unicode string

return flooding corrected original message

rtype unicode string

Two versions of the corrected string are compiled: correction to one or two repetitions. In case the correction to one character produces a valid word, take this one, otherwise correct to two characters.

The sentence is corrected word by word and joined in the end.

hunspell_check(*word*)
check word for spelling

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

param word a word

type word unicode string

return return True or False dependent on word being in dict or not

rtype boolean

3.2 Main

class norm.normalizer.**Normalizer**(*language='nl'*, ***kwargs*)
Pipeline for text normalization

Programm flow: it gets the preprocessed text it flooding corrects its sends this corrected version to the different modules it takes the output per sentence and writes to phrase table it starts the moses decoder with this phrase table it returns the normalized sentence

_call_moses(*s*, *phrase_table*)
decide for a combination of suggestions

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

param s is the original message that has also been forwarded to the modules

type s unicode string

param phrase_table is the path to the phrase table

type phrase_table string

return return normalized sentences

rtype string

Moses is called including the phrase table that has been generated from the suggestions. Using a language model and the phrase table with its features, Moses translates the original sentence into the normalized sentence.

_check_hunspell (*word*)
check word for spelling

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

param word a word

type word unicode string

return return True or False dependent on word being in dict or not

rtype boolean

_generate_phrase_table (*phrase_dict*)
generate the per line entry of a phrase table

parameters, types::

param phrase_dict a dictionary holding all suggestions, with the information of the modules that suggested it and the original token

type phrase_dict dictionary

iterate over the phrase_dict and compile an etry of the followng format for each suggestion

ori ||| sug ||| 0 1 0 1 0 1 ....

the 0 and 1 indicate which module returned the suggestion. The order of the modules is given my the initialization of the modules. The last 0 or 1 says if hunspell can find the suggestion as a word or not.

_kill_server_mode (*proc*)
kill Moses server mode with the help of the process id.

parameters, types::

param proc the process ideas collected when starting up server modes

type proc list of integers

forces the kill of all running processes with the respective process id

_normalize_sentence (*sentence*)
normalize the message, running all modueles and combining their output

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

param sentence preprocessed and flooding corrected text message

type sentence unicode string

return return normalized sentences

rtype list of strings

first all modules generate their suggestion, then these suggestions are collected and written out to a phrase table, then the decision module generates one normalized sentence

_write_phrase_table (*phrase_dict*, *phrase_dict_location*)
write out the phrase table for the run of the decision module

parameters, types::

param phrase_dict a dictionary holding all suggestions, with the information of the modules that suggested it and the original token

type phrase_dict dictionary

param phrase_dict_location the path to the phrase table

type phrase_dict_location string

initialize_modules()
initialize the different module objects

The module objects are initialized onces for every normalization run. The order of the initialization is important since this order is the same as the weights for the modules in the ini files.

For English there are 10 modules:

Word_Split, Compound, Original, Abbreviation, SMT_Token, SMT_Unigram, SMT_Bigram, SMT_Cascaded, Transliterate, Hunspell

For Dutch there are 11 modules:

Word_Split, Compound, Phonemic, Original, Abbreviation, SMT_Token, SMT_Unigram, SMT_Bigram, SMT_Cascaded, Transliterate, Hunspell

normalize_text(*t*)
returns the normalized sentences in a list

the preprocessed text is first flooding corrected then the messages are sent into the pipeline and the normalized sentence is returned

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

param t object from class Text, containing the original and the preprocessed string

type t Text object

return return normalized sentences

rtype list of strings

start_server_mode()
start Moses server mode for all three SMT systems

the SMT modules use the Moses server mode which is started in the beginning of one run for each of the three modes: Token, Unigram and Bigram. They run on a random port between 30000 and 40000. The pids are stored and the server mode is stopped in case of error or sucessful finish. Each Moses server waits for a certain amount of seconds after start up to ensure it has enough time to run properly in the background. The server modes use the .ini files which are located in ../static/Moses/decoder_files. In the ini files themselves you can see which phrase table or language model is accessed.

3.3 Modules

class norm.modules.abbreviation.**Abbreviation**(*normalizer*)
This module resolves the most frequent abbreviations in social media content

check_hunspell(*word*)
check word for spelling

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

param word a word

type word unicode string

return return True or False dependent on word being in dict or not

rtype boolean

generate_alternatives(*sentence*)
Generate suggestion

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

param sentence flooding corrected original message

type sentence unicode string

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

rtype list of lists

Looks up a token in an abbreviation lexicon and returns the long version in case the token is found.

class norm.modules.compound.**Compound**(*normalizer*)
This class contains functions with which subsequent words can be checked for "compoundness". If the spellchecker (hunspell) doesn't complain about the combination of subsequent words, it is returned as an option.

check_hunspell(*word*)
return hunspell result

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

param word a word

type word unicode string

return True or False dependent on whether a word is in hunspell dict or not

rtype boolean

generate_alternatives(*sentence*)
Generate suggestion

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

param sentence flooding corrected original message

type sentence unicode string

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

rtype list of lists

Compounds out of two subsequent tokens are built. Hunspell checks if the word is a correct word.

class norm.modules.empty.**Empty**
Assuming that words can be superfluous, this module returns an empty string.

generate_alternatives(*sentence*)
Generate suggestion

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

param sentence flooding corrected original message

type sentence unicode string

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

rtype list of lists

class norm.modules.new_NE.**New_NE**(*normalizer*)

This module uses a crf model to predict whether a token is a named entity or not.

This modules in not included by default.

_replace_atreplies(*t*)

@replies are returned as a special character

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

param t token

type t unicode string

return t itself or u"" in case the token is an @-reply

rtype list of lists

_run_named_entity_replace(*text_string*)

Rule based component of the module. Search for upper case first letters, search in gazetteer list,

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

param text_string flooding corrected original sentence

type t unicode string

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

rtype list of lists

clean_up()

Delete the directory with all texsis files.

generate_alternatives(*sentence*)

Run a crf classifier to find out if a token is a NE or not.

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

param sentence flooding corrected original message

type sentence unicode string

return original tokens aligned with the token to hand over the information that a token is an NE of the form [[ori,[ori]], [ori2, [u'']]]

rtype list of lists

The information if a token is an NE or not is not included directly but can be used as a feature in the phrase table.

get_labels(*sentence*)

For each word in the input sentence the lable (NE or not) is extracted from the file predicted by crf. If the label is 1 a special character is returned as a suggestion, if not the original token is returned.

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

param sentence flooding corrected original message

type sentence unicode string

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

rtype list of lists

make_feature_file()

An external python script (language specific) is called to compile the feature files used for NE prediction. The script expects the following input:

- language
- texsis POS file
- texsis tok file
- gazetteer file
- celex file
- output file

All these files can be found in the static directory.

run_texsis()
Texsis is used to pos tag the sentence. This information is used as a feature in the crf classification.

write_file(*sent*)
The sentence is written to a file (one word per line) and stored in a directory. Texsis can be run on this directory to generate pos tags for each word in the sentence.

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

param sentence flooding corrected original message

type sentence unicode string

class norm.modules.original.**Original**
This class contains functions which return the original as an option

generate_alternatives(*sentence*)
Generate suggestion

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

param sentence flooding corrected original message

type sentence unicode string

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

rtype list of lists

class norm.modules.phonemic.**Phonemic**
Access via web service the MBT grapeme-to-phoneme-to-grapheme conversion. Implemented by Guy DePauw.

generate_alternatives(*sentence*)
Generate suggestion

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

param sentence flooding corrected original message

type sentence unicode string

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

rtype list of lists

class norm.modules.smt.**SMT_Token**(*normalizer*)
This class contains functions with which the SMT on the token level can be performed.

generate_alternatives(*sentence*)
Generate suggestion

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

param sentence flooding corrected original message

type sentence unicode string

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

rtype list of lists

class norm.modules.smt.**SMT_Cascaded**(*normalizer*)

This class contains functions with which the SMT on first the token and subsequently on the unigram level can be performed.

generate_alternatives(*sentence*)

Generate suggestion

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

param sentence flooding corrected original message

type sentence unicode string

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

rtype list of lists

class norm.modules.smt.**SMT_Unigram**(*normalizer*)

This class contains functions with which the SMT on the unigram level can be performed.

generate_alternatives(*sentence*)

Generate suggestion

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

param sentence flooding corrected original message

type sentence unicode string

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

rtype list of lists

class norm.modules.smt.**SMT_Bigram**(*normalizer*)

This class contains functions with which the SMT on the bigram level can be performed.

generate_alternatives(*sentence*)

Generate suggestion

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

param sentence flooding corrected original message

type sentence unicode string

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

rtype list of lists

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*)
Generate suggestion

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

param sentence flooding corrected original message

type sentence unicode string

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

rtype list of lists

class norm.modules.wordsplit.**Word_Split**(*normalizer*)
This class contains functions with which subsequent words can be checked for "compoundness". It uses word frequencies from the cgn corpus to decide if a word should be split or not. It makes use of the decompounder perl script that comes together with Moses.

generate_alternatives(*sentence*)
Generate suggestion

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

param sentence flooding corrected original message

type sentence unicode string

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

rtype list of lists

3.4 Util

norm.util.**align**(*ori*, *tgt*)
align two sentences using the python sequence aligner

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

param ori an original sentence

type ori Unicode string

param tgt a reference sentence

type tgt Unicode string

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

rtype list of lists

norm.util.**calculate_cer**(*ref*, *hyp*)
calculate CER with the help of dynamic programming alignment

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

param ref a reference sentence

type ref Unicode string

param hyp a hypothesis sentence

type hyp Unicode string

return the CER between ref and hyp

rtype float

norm.util.**calculate_wer**(*r*, *h*)
calculate WER with the help of dynamic programming alignment

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

param ref a reference sentence

type ref Unicode string

param hyp a hypothesis sentence

type hyp Unicode string

return the WER between ref and hyp

rtype float

norm.util.**get_random_phrase_table_path**(*prefix='phrase_table'*)
get a random phrase table name with a prefix and a 6 character long capital-letter-digits-string

file will be located in ../log/phrasetables

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

param prefix optional: prefix for the temporary file, default = phrase_table

type prefix Unicode string

return path to the temporary file

rtype string

norm.util.**get_random_tmp_eval_path**(*prefix='phrase_table'*)
get a random evaluation directory name with a prefix and a 6 character long capital-letter-digits-string

file will be located in /tmp/eval_options/

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

param prefix optional: prefix for the temporary file, default = phrase_table

type prefix Unicode string

return path to the temporary file

rtype string

norm.util.**get_random_tmp_path**(*prefix='norm'*)
get a random path consisting of the prefix name and a 6 character long capital-letter-digits-string

file will be located in /tmp

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

param prefix optional: prefix for the temporary file, default = norm

type prefix Unicode string

return path to the temporary file

rtype string

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

n