Implementation Documentation

Release

Dieuwke Hupkes

November 22, 2013

CONTENTS

Contents:

CHAPTER

ONE

ALIGNMENTS MODULE

Module for processing alignments. This module contains three classes. Running alignments.py will give a demonstration of te functionality of the classes.

class alignments.**Alignments**(*links*, *source*, *target=''*)

A class that represents alignments. Important methods in this class compute the monolingual source phrases according to this alignment, generate all rules or all maximally recursive rules that are associated with the alignment and generate a dictionary representing all trees that are generated with this rules in a compact fashion.

Construct a new alignment object with the alignment links given by 'links', the sourcesentence given by 'source', and possibly a target sentence 'target'. Construct a set-representation of the alignment, and compute a lexical dictionary describing the spans of the words.

Parameters

- **links** – a string of the form '0-1 2-1 ...' representing the alignment links. Word numbering starts at 0.
- **source** – A string representing the source sentence.
- **target** – A string representing the target sentence.

Class is initialized with a string representation of the alignment that starts counting at 0 of the form '0-1 2-1 ...' and the sentence the alignment represents. During intialization, a set-representation of the alignment is created.

HAT_dict(*labels={}*)

Transform all HATrules into a dictionary that memory efficiently represents the entire forest of HATs. As a HAT_dict uniquely represents a HATforest, the labels of all spans should be unique avoid amibuity.

Parameters **labels** – A dictionary assigning labels to spans.

Returns A dictionary that represents the HATforest, by describing for every allowed span what is allowed expansions are. Entries are of the form {lhs: [(rhs_11,...,rhs_1m),...,(rhs_n1,...,rhs_nk)]

_links_fromE()

Precompute values for the function E_c(j) = **|{(i',j')\in A | j' =< j}|**.

_links_fromF()

Precompute values for the function F_c(j) = **|{(i',j')\in A | i' =< i}|**.

_maxspan((*x*, *y)*)

Returns the maximum position on the target side that are linked to positions [x,y].

_minspan((*x*, *y)*)

Returns the minimum position on the target side that are linked to positions [x,y].

agreement (*tree*)
Output what percentage of the nodes of an inputted tree are consistent with the alignment. :param tree: An nltk tree object. :return: a float that describes the percentage of the nodes

of tree that were nodes according to the alignment.

compute_phrases ()
Return a list with all source phrases of the alignment. Similar to Alignments.spans, but returns a list rather than a generator, and does not include all one-length units.

Returns A list with all valid source phrases

consistent_labels (*labels*, *label_dict*)
Measures the consistency of the alignment with a dictionary that assigns labels to spans. Outputs a dictionary with labels, how often they occurred in the input set and how often they were preserved. Ignore word-labels from dep-parse that end in -h.

hat_rules (*prob_function*, *args=[ ]*, *labels={}*)
Return a generator with all rules of a PCFG uniquely generating all *hierarchical* alignment trees. The rules are assigned probabilities using the input probability function, args are the arguments of these function.

The rules are computed by transforming the alignment into a set of Node objects and links between them that together constitute a graph whose edges correspond to valid spans and partial sets, and using the shortest_path function of the Node class.

Parameters

- **prob_function** – A probability function from the Rule class, according to which probabilities should be assigned.
- **args** – The arguments the probability function needs.
- **labels** (*A dictionary assigning labels to spans.*) – The labels that should be assigned to the spans.

lex_dict ()
Use self.sentence to create a lexical dictionary that assigns lexical items to spans. :return: A dictionary {(0,1) : word1,..,(n-1,n): wordn}

lexrules (*labels={}*)
Returns an generator with the terminal rules of the grammar. (i.e., the 'lexicon', that tells you the span corresponding to a word). If labels are provided for the spans, the rules will be labelled accordingly.

make_set (*alignment*)
Return a set with all alignment links, and keep track of the length of source and target sentence. Output a warning when alignment and sentence do not have the same number of words.

Parameters **alignment** – A string representing the alignment, as was passed during initialisation

percentage_labelled (*labels*)
Output which percentage of the spans in the alignment are labelled by the set of inputted labels. :return: total, labelled

prune_production (*rule*, *lex_dict*)
Replace all leafnodes that do not constitute a valid source phrase with the lexical item the leafnode dominates.

Parameters **lex_dict** – a dictionary with spans as keys, and the corresponding lexical items as values.

Returns a Rule object in which all nodes are either valid source spans or lexical items.

rules(*prob_function*, *args*, *labels={}*)
Returns a generator with all rules of a PCFG uniquely generating all alignment trees. Rule probabilities are assigned according to specified input probability function, args should contain a list of arguments for this function.

The rules are computed by transforming the alignment into a graph whose edges correspond to valid spans and partial sets and using the path function of the Node class. This function is not as extensively tested as the hat_rules function, as it is rarely used for computational issues.

spans()
Return all a generator with all valid source side spans that are part of a phrase pair, and all one-length units that are necessarily part of a tree describing the translation. Contrary to the convention, also unaligned sequences of words are allowed as spans.

Spans are computed using the first shift-reduce algorithm presented in Chang & Gildea (2006). This is not the most efficient algorithm to compute phrase pairs.

texstring()
Generate latexcode that generates a visual representation of the alignment.

alignments.**HAT_demo**()
Demonstration 3. Simple one to many alignment, HATfunctionality.

class alignments.**Node**(*value*)
Defines a node in a directed graph. You can add edges from this node to other nodes by using link_to. The paths_to method calculates all paths from this node to the given node. The Node class is used to represent alignments as graphs.

Initializes a new node; value can be used for representation

link_to(*node*)
Add a directed edge from this node to the given node. :type node: A Node object.

paths_to(*node*)
Returns a generator that calculates all paths to the given node. These paths are calculated recursively. :type node: a Node object

remove_link(*node*)
Remove the edge to this node, if present :type node: A Node object.

shortest_paths_to(*node*)
Finds all shortest paths from current node to node using an adapted Dijkstra algorithm starting from the end. The function also stores paths that can be used for later(i.e paths longer than 1 from self to intermediate nodes). :type node: A Node object.

class alignments.**Rule**(*root*, *path*, *labels={}*)
Defines a rule from one span to a set of consecutive spans which's union forms it.

A string representation is provided for convenient displaying.

Initialize a new rule as its root span and the path in the graph (consisting of an array of nodes) that it produces. Labels can be provided to annotate the spans of a rule. :param root: The rootspan of the node. :type path: A Waypoint. :type labels: A dictionary assigning labels to spans.

_lhs(*labels*)
Create the left hand sides of the rule and set as an attribute.

_rhs(*labels*)
Create the right hand sight of the rule and set as attribute.

_str(*rhs*)

lhs()
Return the left hand side of the rule. :type return: nltk.Nonterminal object.

probability_labels(*labels*)
Compute the probability of a rule according to how many of the nodes it generates can be labelled according to a set of given labels. :param labels: A list containing a dictionary that

assigns labels to spans.

probability_spanrels(*span_relations*)
Compute the probability of a rule according to how many span_relations it makes true. :param span_relations: A list containing a dictionary

which describes which spanrelations are desired.

rank()
Return the rank of a rule.

rhs()
Return the right hand side of the rule. :type return: a tuple with nltk.Nonterminal objects

uniform_probability(*args*=[])
Set probability to 1.

class alignments.**Waypoint**(*node*, *link=None*)
Defines a waypoint in a one-directional path. The class Waypoint is used in the representation of paths. Multiple paths can be saved more memory efficient as path arrays can be shared between paths. As they contain link to other Waypoints, Waypoints can represent paths as linked lists.

Create a waypoint object.

Parameters

- **node** (*A Node Object.*) – The node it represents.
- **link** (*A Waypoint object.*) – A link to the next waypoint.

alignments.**demo_basic**()
Demonstration 1, basic monotone one-to-one alignment.

alignments.**demo_basic2**()
Demonstration 2. Simple one-to-many alignment.

alignments.**demos**()

CHAPTER

TWO

DEPENDENCIES MODULE

class constituencies.**ConstituencyTree**(*tree*, *sentence=False*)

A class representing a constuency tree. The classes uses the nltk class nltk.Tree, but adds some functionality that is useful with respects to alignments.

Create a ConstituencyTree object.

Parameters

- **tree** (*str or nltk.Tree object.*) – A constituency tree.
- **sentence** – The sentence that the leafnodes of the tree constitute.

branching_factor(*branching_dict={}*)

Return a dictionary that summaries the different branching factors of the trees. If initialised with a dictionary, update this dictionary with the valeus of the current tree.

nr_of_nonterminals()

Return the number of nonterminals in self.tree.

phrases_consistent(*subtree*, *startpos*, *phrase_list*)

Return the number of non-terminal nodes in the tree that occur in the provided list of phrases. :param subtree: A subtree of self.tree. :param startpos: The left-most word position the subtree dominates. :param phrase_list: A list of allowed phrases. :return: the number of nodes in the tree that is in phrase_list.

reconstruct_sentence()

Reconstruct the sentence from the leaf nodes of the tree. If a sentence was passed in initalisation, return this sentence. :type return: str

root_span(*subtree*, *startpos*)

Recursively compute the span a node covers :param subtree: a subtree of self.tree :param startpos: the first position the subtree dominates

constituencies.**demo**()

CHAPTER

THREE

DEPENDENCIES MODULE

class dependencies.**Dependencies**(*dependency_list*, *sentence=False*)

A class representing the dependencies of a sentence in a dictionary. The dependencies are created from a list with dependencies formatted like the stanford dependency parses.

Create a Dependencies object, based on the passed dependency list. If no sentence is passed, the sentence is reconstructed from the dependency list, leaving gaps for items that were not included.

Parameters **dependency_list** – A list with dependencies of the form reltype(head-pos_head, dependent-pos_dependent) min(pos-dependent) = 1

Returns A dictionaries with entries of the form pos_head: [pos_dependent, reltype]

POStag(*word*)

Find a postag for a word.

SAMT_labels()

Create SAMT-style labels based on the basic dependency labels created in dependency_labels. The order if precedence is as follows:

- Basic labels
- labels A + B, where A and B are basic labels
- labels A/B or AB where A and B are basic labels
- labels A + B + C where A,B and C are basic labels

annotate_span(*labels*)

Annotate labels with their span, to make the grammar unique.

argument_list(*head_span*)

return a list with spans of the head and its arguments

branching_factor(*b_dict={}*)

Compute the branching factor of all nodes in the dependency tree. If an input dictionary is given, update the branching factors in the dictionary with the newly found branching factors.

checkroot()

Check if dependencies form a tree by checking coverage of the rootnote.

comp_score()

Returns the percentage of words that is head of another word, thereby giving a measure of the level of compositionality of the parse

dependency_labels()

Produces standard labels for spans according to the following scheme:

- label[(i,i+1)] = HEAD iff word i+1 is the head of the sentence

- label[(i,j+1)] = rel iff there is a dependency relation rel(x, y) and wordspan(y) = (i,j+1)
- label[(i,i+1)] = rel-head iff there is a dependency relation rel(x,i+1) and word i+1 was not labelled by one of the previous conditions

find_dependent(*relation*)
Find the depending word of a dependency relation using a regular expression.

find_dependent_pos(*relation*)
Find the position of the dependent of a dependency relation using a regular expression.

find_head(*relation*)
Find the head word of a dependency relation using a regular expression.

find_head_pos(*relation*)
Find the position of the head of a dependency relation using a regular expression.

find_relationtype(*relation*)
Find the type of a dependency relation using a regular expression.

get_comp_spanrels()
Create a dictionary of dependencies between word positions and word spans. Go through the dependency dictionary, but select only the relations that display compositionality (i.e. no relations between words)

get_span(*key*)
Recursively compute the span of a word. The span of a word is constituted by the minimum and maximum position that can be reached from the word by following the directed dependency arrows. The spans are left exclusive and right inclusive. I.e. if positions i and j are the minimum and maximum positions that can be reached from a word, its span will be [i-1,j]. Every word necessarily spans itself, a word at position i without dependents will thus have span [i-1,i]. The dependency from root to head of the sentence is not considered.

label_all()
Label all spans of the sentence.

labels(*label_type='basic'*)
Return labels of given type.

Parameters **type** – describes which label_type should be used. Options: all, basic or SAMT.

The default labeltype is basic.

percentage_SAMT()
Compute how many spans were labelled by an SAMT label. :return: number of spans, number of labelled spans

print_deps()
Displaying function. Print all the dependency relations in the dependency parse.

print_labels(*labels*)
Print out the contents of a dictionary in a nice format.

print_spans()
Displaying function. Print all word_spans of of the dependency parse.

reconstruct_sentence(*sentence*)
Reconstruct the sentence corresponding to the dependency parse. :return: a list with the words of the sentence.

set_dependencies(*dependency_list*)
Read in a file and create a dictionary with its dependencies using regular expressions.

set_wordspans()
Compute the span of each word and store it in a dictionary with word positions and a tuple that represents their span as key and value, respectively.

spanrelations(*rightbranching=False*, *leftbranching=False*, *interpunction=True*)
Create a dictionary with spanrelations that are 'deeper' than the standard relations in the dependency parse, such that stepwise combining head and arguments is allowed. Parameters rightbranching, leftbranching and interpunction describe how exactly arguments and heads are allowed to combine.

Parameters

- **rightbranching** – allow an argument to combine with the arguments one by one, giving preference to arguments to the right.
- **leftbranching** – allow an arguments to combine with the head one by one, giving preference to arguments to the left.
- **interpunction** – Take gaps in the dependency parse into account, by adding extra relations in which the gap is already combined with one of its left or right adjacing units.

If both left- and rightbranching are true, all combination orders are allowed.

textree()
Print string that will generate a dependency tree in pdf with package tikz-dependency.

update_labels(*label_dict*)
Update an inputted dictionary with the labels from dependency object.

dependencies.**demo**()
A demonstration of how the Dependencies class can be used.

CHAPTER

FOUR

FILE_PROCESSING MODULE

File_processing is a module for processing sentences, alignments and treestructures. It brings together the functions from the other classes, enabeling the user to apply the functions using information from three files containing alingments, sentences and parses. Explain the different possibilities of the class.

class file_processing.**ProcessConstituencies**(*alignmentfile*, *sentencefile*, *treefile*, *targetfile=False*)

Bases: file_processing.ProcessFiles

Subclass adapted for constituencies

During initialization the files are loaded for reading. Allows to leaf empty one of more files if they are not needed for functions that will be used

all_rules(*max_length=40*)

branching_factor(*max_length=40*)

Compute the average branching factor of all head nodes of the dependency parses or the corpus. Can be restricted to a sentence length.

consistent_labels(*label_type*, *max_length=40*)

Determines the consistency of a set of alignments with a type of labels over the entire corpus.

relation_count(*max_length*)

Counts occurences of all labels in the constituent parse.

score_all_sentences(*rule_function*, *probability_function*, *prob_function_args*, *label_args*, *max_length=40*, *scorefile=''*, *treefile=''*)

texstring(*new*)

Output a texstring with the alignment, the constituency tree and the alignment.

class file_processing.**ProcessDependencies**(*alignmentfile*, *sentencefile*, *treefile*, *targetfile=False*)

Bases: file_processing.ProcessFiles

Subclass of ProcessFiles that is focussed on the specific occasion in which trees are dependencies.

During initialization the files are loaded for reading. Allows to leaf empty one of more files if they are not needed for functions that will be used

all_HATs(*file_name*, *max_length=40*)

Compute all HATs grammars, represent them as a dictionary and and pickle [sentence_nr, rootlabel, HAT-dict] to a file with the provided name file_name.

branching_factor(*max_length*)

Compute the average branching factor of all head nodes of the dependency parses or the corpus. Can be restricted to a sentence length.

check_consistency(*sentence*, *dep_list*)
Check whether a list with dependencies is consistent with a sentence, by comparing the words. Some flexibility is allowed, to account for words that are spelled differently. Return True if the dependency parse contains no more than 3 words not present in the sentence and False otherwise.

close_all()
Close all input files.

consistent_labels(*label_type*, *max_length=40*)
Determines the consistency of a set of alignments with a type of labels over the entire corpus.

percentage_labelled(*max_length*, *label_type*)
Compute the percentage of the spans in the dictionary that is labelled by a labelling method

print_dict(*dictionary*, *filename*)
Print the contents of a dictionary to a file.

relation_count(*max_length*)
Counts occurences of all relations in dependency parses of sentences shorter than max_length.

relation_percentage(*all_relations*, *relations_present*)

sample(*samplesize*, *maxlength=False*, *display=False*)
Create a sample of sentence from the inputted files. Create a file with the sentences, and files with the matching alignments, dependencies and targetsentences. If display = True, create a texfile that can be ran to give a visual representation of the selected sentences. Return an array with the list of sentence numbers that were selected.

score_all_sentences(*rule_function*, *probability_function*, *prob_function_args*, *label_args*, *max_length=40*, *scorefile=False*, *treefile=False*)

tex_preamble()
Print the pre-amble of a tex document in which both dependency parses and alignments are printed.

texstring(*new*)
Output a texstring with the alignment, the dependency and the ew = alignment, sentence, dep

transform_contents(*value*)
Return a suitable string representation of input

class file_processing.**ProcessFiles**(*alignmentfile*, *sentencefile*, *treefile*, *targetfile=False*)
Brings together all functions by enabling the user to apply functions from the other classes to files containing alignments, sentences and dependency parses.

During initialization the files are loaded for reading. Allows to leaf empty one of more files if they are not needed for functions that will be used

close_all()
Close all input files.

consistent_labels(*label_type*, *max_length=40*)
Determines the consistency of a set of alignments with a type of labels over the entire corpus.

evaluate_grammar(*grammar*, *max_length*, *scoref*)
Parse the corpus with inputted grammar and evaluate how well the resulting parses cohere with the alignments.

next()
Return the next alignment, sentence and tree_list. If the end of one of the files is reached, return False.

next_sentence()
Return the next sentence. If the end of the file is reached, return None.

print_dict(*dictionary*, *filename*)
Print the contents of a dictionary to a file.

print_function(*to_print*, *filename*)

relation_count(*max_length*)
Counts occurences of all relations in dependency parses of sentences shorter than max_length.

relation_percentage(*all_relations*, *relations_present*)

score_all_sentences(*rule_function*, *probability_function*, *prob_function_args*, *label_args*, *max_length=40*, *scorefile=''*, *treefile=''*)
Not implemented in general class, use from more specific subclasses. If not present, raise not implemented error.

transform_contents(*value*)
Return a suitable string representation of input

CHAPTER

FIVE

LABELLING MODULE

class labelling.**Labels**(*labels*)

Class to create different kinds of labels from a set of basic labels. Currently, some set of labels are computed multiple times to be used in the computation of other labels. Class can be made more efficient by storing such sets of labels as attributes of the Labels instance.

Create a labelling object, with basic labels. :param labels: A dictionary assigning labels to spans

SAMT_labels()

Return all SAMT labels based on the basic labels of the object. The order if precedence is as follows:

•Basic labels

•labels A + B, where A and B are basic labels;

•labels A/B or AB where A and B are basic labels;

•labels A + B + C where A,B and C are basic labels;

annotate_span(*labels*)

Annotate labels with their span, to make the grammar unique.

concat(*depth*, *o={}*, *i=None*)

Compute all concatenated labels up to inputted depth, with basic labels i. If an output dictionary o is passed, extend this dictionary with the found spans that do not yet have a label in o.

Parameters

- **depth** – The maximum number of variables in the labels.
- **o** – An output dictionary with already existing labels.
- **i** – A dictionary with basic labels to be concatenated.

label_complexity(*label*)

Return the number of variables in a label.

label_most()

Label all spans within the range of labels, following the following rules:

•Labels with a lower depth are always preferred;

•Concatenated labels are preferred over minus and double/minus labels;

•Single minus labels (with concatenated spans) are preferred over double minus labels.

SAMT labels thus have precedence over other labels.

minus(*depth*, *o={}*, *i=None*)

Compute all labels of the form AB and B/A where B is a basic label, and A a concatenated label that

contains no more than depth-1 variables. If an output dictionary o is passed, extend this dictionary with the found spans that do not yet have a label in o.

Parameters

- **depth** – The maximum number of variables in the labels.
- **o** – An output dictionary that is to be updated with the new labels
- **i** – A dictionary with basic labels.

minus_double(*depth*, *o={}*, *i=None*)

Compute all labels of the form AB/C, where A is a basic label, and B and C are concatenated labels. The outputted labels have a number of variables that is no higher than depth. If an output dictionary o is passed, extend this dictionary with the found spans that do not yet have a label in o.

Parameters

- **depth** – The maximum number of variables in the labels.
- **o** – An output dictionary that is to be updated with the new labels
- **i** – A dictionary with basic labels.

Return a dictionary with all labels of the form , where B is in self.labels or in i if i is provided, and A and C are in A1 + A2 + An where A1.. An are in self.labels or i and n <= depth.

labelling.**demo**()

CHAPTER

SIX

DEPENDENCIES MODULE

class process_hats.**HATGrammar**(*HATdict*, *root*)

Class that

Initialise with a dictionary uniquely representing a HAT

plain_label(*label*)

strip the label from the part determining its span, to make it uniform

probmass(*head_node*, *children=()*, *external_pcfg={}*, *probs={}*)

Compute the probability mass of all subtrees headed by head_node with direct children children (possibly empty), given the input pcfg.

to_WeightedGrammar(*rule_dict*, *remove_old=False*)

Transforms a set of rules represented in a nested dictionary into a WeightedGrammar object. It is assumed that the startsymbol of the grammar is TOP, if this is not the case, parsing with the grammar is not possible. If remove_old = True, remove the old grammar during the process to save memory.

update(*external_pcfg*, *probs*, *grammar*, *p_cur*, *lhs*)

Compute the updated counts for a node, given its parent and how often this parent occurred in the forest. Does not return a grammar, but modifies it globally.

update_weights(*grammar*, *external_pcfg={}*)

Implicitly assign all HATs in the HATforest a probability, normalise, and compute the counts of the rules in them through relative frequency estimation. Update the inputted grammar with these counts.

class process_hats.**ProcessHATs**(*HATfile*)

Class with functions that can be applied to a file containing pickled precomputed HATs. ProcessHATs has functional overlap with the class FileProcessing, but is more efficient as it avoids recomputing HATforests.

Pass the name of the file containing the pickled HATs.

em(*max_iter*)

When passing a grammar represented by a dictionary, iteratively assign probabilities to all HATs of the corpus and recompute the counts of the grammar with relative frequency estimation until convergence or until a maximum number iterations is reached. Return the new grammar :param start_grammar Grammar represented as a nested dictionary :param max_iter Maximum number of iterations :param max_length Maximum sentence length considered

em_iteration(*old_grammar*, *new_grammar*)

Assign probabilities to all HATs in the corpus with the current grammar, recompute probabilities and return the new grammar. It is assumed that the HATs are precomputed and pickled into a file in the correct order. Every sentence under max_length should be represented in the file as: [sentence_nr, HAT_dict, root].

initialise_grammar()

Initialise a grammar based on all HATs in the corpus

next()
Return the next item in the file. If no :return [sentence_nr, HATdict, root]

normalise(*rule_dict*)
Given a nested dictionary that represent rules as follows: {lhs : {rhs1 : count, rhs2: count ...}, ...}, return a similar nested dictionary with normalised counts

unique_rules(*stepsize*)
Go through HATcorpus and keep track of the percentage of the rules that is unique. Store the number of rules and the number of unique rules if the number of HATs processed % stepsize is 0

CHAPTER

SEVEN

SCORING MODULE

class scoring.**Scoring**(*alignment*, *sentence*, *labels={}*)

Class that provides methods for scoring an alignment according to a set of preferred relations. The corresponding tree is created, spanlabels can be entered to label the nodes in the tree.

During initialization an alignment, a corresponding sentence and a string with dependencies are passed. A weighted CFG generating all HATs is created, the rules are assigned 'probabilities' according to preferred_relations or labels. The adapted viterbi parser from the nltk toolkit is used to parse the sentence and obtain the score.

grammar(*rules*)

Return a weighted grammar (NLTK-style) and its rank given a generator object with all rules.

grammar_rank(*rules*)

Determine the maximum rank of a set of rules.

list_productions(*rules*)

make_lexdict()

Create a dictionary assigning words to spans.

parse(*grammar*, *trace=0*)

Parse the sentence with the given grammar using the viterbi parser from the nltk. Return the best parse and its score.

score(*rule_function*, *prob_function*, *args*, *trace=0*)

Score, args are arguments for prob_function. Thus: if probfunction = Rule.probability_labels, then args should be [labels], if it is Rule.probability_spanrels then args should be [spanrels, normalization_factor]

transform_to_Production(*rule*)

Transform rule to Production object (NLTK-style)

transform_to_WeightedProduction(*rule*)

Transform Rule object to WeightedProduction object (NLTK-style)

CHAPTER

EIGHT

TESTS MODULE

class tests.**Tests**

test_all()

class tests_alignments.**AlignmentsTests**
Tests Alignments Module

alignment_test_all()

consistency_test1()

consistency_test2()

consistency_test3()

dict_test()
Test the function Alignments.HAT_dict()

span_test1()
Test if correct spans are found for a monotone alignment with no unaligned words

span_test2()
Test if correct spans are found for a non monotone many-to-many alignment, with no unaligned words on source nor targetside.

span_test3()
Test if correct spans are found for a one-to-one alignment with some unaligned words on source and target side

span_test4()
Test if correct spans are found for a non monotone many-to-many alignment with unaligned words on both and target side.

spans_test_all()
Return True if all span tests return True

class tests_dependencies.**DependencyTests**

allr_test1()
Test left branching relations for sentence 'I give the boy some flowers'

allr_test2()

dependencies_test_all()
Run all dependency tests.

labels_annotation_test()
Test annotated labels for a manually constructed sentence

labels_test1()
Test functioning plain labelling for sentence 'I give the boy some flowers'

lr_test()
Test left branching relations for sentence 'I give the boy some flowers'

rr_test()
Test right branching relations for sentence 'I give the boy some flowers'

spanrelations_test()
Test if spanrelations are extracted correctly from a list of dependencies

test_all_labels()
Test for label all function

test_samt_labels()
Test SAMT labels

class tests_node.**NodeTests**
Tests functionality of Node class.

path_test1()
Test if paths are computed as intended by manually constructing a graph with 5 nodes and a couple of edges. Output True if correct paths are found, False otherwise.

path_test2()
Test if shortest paths are computed as intended by manually constructing a graph with 5 nodes and a couple of edges. Output True if correct paths are found, False otherwise.

path_test3()
Test if shortest paths are computed as intended by manually constructing a fully connected graph with 5 nodes. Output True if correct paths are found, False otherwise.

path_test_all()

worst_case_test(*nr_of_nodes*)
Speed test for shortest_paths_to. Create a fully connected graph with nr_of_nodes nodes and compute the shortest_paths between all nodes. Output running time.

class tests_rule.**RuleTests**
Testing class for the rule class.

rules_test_all()
Return True if all rule tests return True

test2()

test_hatrules()
Test if the correct HATgrammar is generated for the sentence 'My dog likes eating sausages', with alignment '0-0 1-1 2-2 2-3 3-5 4-4'.

test_hatrules2()
Test if the correct HATgrammar is generated for the sentence 'My dog likes eating sausages', with alignment '0-0 1-1 2-2 2-3 3-5 4-4'.

test_rules()
Test if the correct grammar is generated for the sentence 'My dog likes eating sausages', with alignment '0-0 1-1 2-2 2-3 3-5 4-4'.

class tests_scoring.**ScoreTests**
Test Scoring Class

score_test1()
Sentence: 'my dog likes eating sausage' Alignment: '0-0 1-1 2-2 2-3 3-5 4-4' Dependencies: 'nsubj(likes-3, dog-2)', 'root(ROOT-0, likes-3)', 'xcomp(likes-3, eating-4)' and 'dobj(eating-4, sausages-5)'.

Manual score normal rules spanrels: 1.0 Manual score hatrules spanrels: 0.75 Manual score hatrules spanrels deep: 1.0

score_test2()
Sentence: 'european growth is inconceivable without solidarity .' Alignment: '0-0 1-1 2-2 3-3 4-4 5-5 6-6' Dependencies: 'nn(growth-2, european-1)', 'nsubj(inconceivable-4, growth-2)', 'cop(inconceivable-4, is-3)', 'root(ROOT-0, inconceivable-4)', 'prep(inconceivable-4, without-5)' and 'pobj(without-5, solidarity-6)'

Manual score all rules spanrels: 1.0 Manual score hat rules spanrels: 0.6 Manual score hat rules spanrels deep: 1.0

score_test3()
Sentence: 'approval of the minutes of the previous sitting' Alignment: '5-6 4-5 3-4 3-2 2-1 6-8 3-3 1-1 0-0 7-7' Dependencies: 'root(ROOT-0, approval-1)','prep(approval-1, of-2)', 'det(minutes-4, the-3)','pobj(of-2, minutes-4)','prep(approval-1, of-5)','det(sitting-8, the-6)','amod(sitting-8, previous-7)','pobj(of-5, sitting-8)'

Manual score all rules spanrels: 0.59 Manual score hat rules spanrels: 0.57 Manual score hat rules spanrels deep: 1.0

score_test4()
Sentence: 'resumption of the session' Alignment: '3-3 2-2 1-1 0-0' Dependencies: 'root(ROOT-0, resumption-1)','prep(resumption-1, of-2)','det(session-4, the-3)','pobj(of-2, session-4)'

Manual score all rules spanrels: 1.0 Manual score hat rules spanrels: 1.0 Manual score hat rules spanrels deep: 1.0

score_test5()
no dependencies

score_test6()
dependencies form no tree

score_test7()
Test to check workings for interpunction.

score_test8()
Second interpunction check.

score_test9()

score_test_all()

class tests_labelling.**LabelsTests**
Class to test functionality of the class Labels

SAMT_test()
Test Labels.SAMT

base_test()

concat_test()
Test Labels.concat for different depths.

label_all_test()
Test Labels.label_most()

minus_double_test()
Test Labels.minus_double for different depths

minus_test()
Test Labels.minus for different depths

test_all()

class tests_HATforest.**HATsTests**
Contains tests for the HATforest class.

grammar_test()
Test function for Alignments.compute_weights with span adjusted labels.

probs_test()
Test the function HATs.probmass()

test_all()

update_test()
Test the function HATs.update()

update_test2()
Test the function Alignments.update() with input PCFG

CHAPTER

NINE

INDICES AND TABLES

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

Symbols

A

B

C

D

E

R

S

T

U

W