EDAN20
Final Examination

Pierre Nugues

October 26, 2016

The examination is worth 264 points. The distribution of points is indicated with the questions. You need 50% to have a mark of 4 and 65% to have a 5.

1 Closed Book Part: Questions

In this part, no document is allowed. It is worth 120 points.

Chapter 1. Cite three applications that use natural language processing to some extent. 3 points

Chapter 1. Annotate each word of the sentence:

> Regulator defends tough bank rules[1]

with its part of speech. You will use parts of speech you learned at school. 3 points

Chapter 1. Draw the graph of the sentence *Regulator defends tough bank rules* using dependency relations. You will notably identify the subject and the object. 4 points

Chapter 1. Identify the proper nouns (or named entities) in the sentence:

> Gibbon wrote a book on the Roman Empire

and describe what entity linking is on this example. 8 points

Chapter 1. Represent the sentence *Gibbon wrote a book on the Roman Empire* with a predicate–argument structure. 4 points

Chapter 2. Describe what a concordance is and give all the case-sensitive concordances of the string *Sagohuset*: 3 points

> Sagohuset får fortsätta att leta lokaler - trettioårig jakt är inte över
>
> I över trettio år har Teater Sagohuset kämpat för att få en permanent lokal i Lund. Förslaget att skrota idén om Folkparken

[1] Retrieved on October 12, 2016 from `www.ft.com`

som teaterlokal förvånar därför inte Margareta Larson, konstnärlig ledare på Sagohuset.

Elverket. Lokstallarna. Observatoriet och nu Folkparken.

– Vi har gått igenom alla byggnader i Lund där vi skulle kunna vara. Men mattan dras hela tiden undan för oss, säger Margareta Larson. Till våren rivs Sagohuset på Revingegatan för att ge plats åt bostäder och en förskola som kräver parkyta. Det innebär att teatern står utan scen från den 12 maj 2017.

Förhoppningen var att Sagohuset skulle kunna flytta in i Lunds gamla observatorium vid Stadsparken. Diskussioner fördes mellan teatern och Lundafastigheter samt länsstyrelsen.

Sydsvenskan.se, Retrieved October 12, 2016. Author Görel Svahn

Chapter 2. Identify what the regular expressions in the list below will match in the text above (identify all the matches or write no match if there is no match): 15 points

List of regular expressions:

1. `Lunds?`
2. `(Lunds)+`
3. `huset.`
4. `huset\.`
5. `[A-Za-z]+parken`
6. `[A-Za-z]+slaget`
7. `\p{L}+slaget`
8. `\p{L}{3}slaget`
9. `\p{L}{4,}slaget`
10. `(..)g\1`

Chapter 2. Write a generic regular expression that matches the date in the text as well as similar dates. 2 points

Chapter 2. Spell checkers identify words that are not in their dictionaries and suggest corrections to unknown words. They typically use four operations: deletions, insertion, substitutions, and transpositions. 15 points

Taking example from Peter Norvig's spell checker in Python or using your own ideas, answer these questions:

1. How would you collect a dictionary of correctly spelled words?
2. If a word is not in your dictionary, supposedly misspelled, how would you generate candidates that are corrections?
3. What is the edit operation that generates the most candidates? And how many given an alphabet of 26 lowercase letters?
4. What is the edit operation that generates the least candidates? And how many?

5. If more than one candidate is a correct word, how would you select the best candidate?

You will only consider lowercase unaccented characters.

Chapter 3. Describe what a Unicode block is and give an example of a block. 2 points

Chapter 4. Describe what a classifier is. What is the input of a classifier (predictors) and output (response). Give an example. 4 points

Chapter 4. Describe what is a vectorizing operation applied to symbolic features (or one hot encoding). Why do you need such a vectorizing with scikit-learn? 4 points

Chapter 5. What is a tokenization? And why can it be difficult? Give an example. 3 points

Chapter 5. In the context of the tokenization of a text, write a regular expression that finds all the words consisting a sequence of letters, including accented letters. You can use the regular expression you used in the labs. 3 points

Chapter 5. Document indexers identify all the words of the documents they index. Using the index, one can retrieve a document by the words it contains. Once indexed, bag-of-word representations are often used to represent the documents of a collection in classification for example. Table 1 shows an incomplete representation of the two documents

D1: Chrysler plans new investments in Latin America.

D2: Chrysler plans major investments in Mexico.

using bag-of-words.

Table 1: The vectors representing the two documents. The words have been normalized in lowercase letters. You will have to fill in the coordinates

D#\ Words	america	chrysler	in	investments	latin	major	mexico	new	plans
1									
2									

1. Fill in the values in Table 1 with coordinates corresponding to the term frequencies (TF) in the two documents. 4 points
2. Fill in the values in the table with coordinates corresponding to the logarithm base 10 of the inverted document frequency (IDF). 4 points
3. Fill in the values in Table 1 with coordinates corresponding the product $TF \cdot IDF$. 2 points
4. Define the cosine similarity of two vectors. 4 points
5. Compute the cosine similarity of the two documents. 2 points

Table 2: An excerpt from the CoNLL 2000 corpus.

Words	**Parts of speech**
He	PRP
reckons	VBZ
the	DT
current	JJ
account	NN
deficit	NN
will	MD
narrow	VB
to	TO
only	RB
#	#
1.8	CD
billion	CD
in	IN
September	NNP
.	.

Chapter 10. Identify all the noun phrases in the sentence in Table 2. 4 points

Chapter 10. Precision and recall are two common measures of performance in NLP. 4 points

1. Define precision;
2. Define recall.

Chapter 10. Let us suppose an automatic chunker that would extract three noun phrases from the sentence in Table 2: *he*, *reckons*, and *# 1.8 billion*. What would be its recall and precision. 4 points

Chapter 15. The CoNLL 2008 format is a syntactic and semantic annotation format. It uses columns to describe the index, words, lemmas, parts of speech, dependency structure, and finally the semantic predicates and arguments of a sentence. We will use a simplified version of it in this examination. The (simplified) annotation of the sentence

> The SEC will probably vote on the proposal early next year, he said.

is shown in Table 3.

In this question, you will draw two graphs representing respectively the syntactic and semantic structures. You will draw the syntactic graph above the sentence and the semantic one under.

Syntactic graph: The first six columns are identical to the format used in dependency parsing and represent the dependency graph. The third column is the lemma.

Table 3: Simplified annotation of the sentence *The SEC will probably vote on the proposal early next year, he said.* in the CoNLL 2008 corpus

ID	FORM	LEMMA	POS	HEAD	DEPREL	PRED	ARG	ARG
1	The	the	DT	2	NMOD	_	_	_
2	SEC	sec	NNP	3	SBJ	_	A0	_
3	will	will	MD	14	OBJ	_	AM-MOD	A1
4	probably	probably	RB	3	ADV	_	AM-MNR	_
5	vote	vote	VB	3	VC	vote.01	_	_
6	on	on	IN	5	ADV	_	A1	_
7	the	the	DT	8	NMOD	_	_	_
8	proposal	proposal	NN	6	PMOD	_	_	_
9	early	early	RB	11	NMOD	_	_	_
10	next	next	JJ	11	NMOD	_	_	_
11	year	year	NN	5	TMP	_	AM-TMP	_
12	,	,	,	14	P	_	_	_
13	he	he	PRP	14	SBJ	_	_	A0
14	said	say	VBD	0	ROOT	say.01	_	_
15	.	.	.	14	P	_	_	_

- Represent graphically (draw) these syntactic dependencies. 3 points
- Is the graph projective? 1 point

Semantic graph: The rest of the columns corresponds to the predicates and their arguments:

- The seventh column indicates the predicates. How many predicates are there in this sentence? Underline the predicates in the sentence. 1 point
- The columns to the right of the predicates represent their respective arguments in their order in the sentence. How many arguments does the first predicate has (8th column) and the second one (9th column)? 2 points
- For each predicate, draw the arcs corresponding to its arguments. You will draw these arcs under the sentence and label them with their names. 2 points
- Although an arc points to a single word, each argument consists of phrase. This phrase corresponds to the syntactic dependents of the argument head word – the syntactic subtree starting from the pointed word, until it intersects with another one. Draw a box around the arguments of *say* and then *vote*. Use two figures. 6 points
- Represent the predicate–argument structures for both predicates in the form of: 4 points
  `predicate(arg0, arg1, arg2, ...)`.

2 Problem

In this part, documents are allowed. It is worth 144 points.

Transition-based parsing is a a technique to parse dependencies. This parsing technique uses a queue of input words, a stack, and a set of operations to apply to these data structures. The three main parsing variants are called arc-standard (Yamada and Matsumoto, 2003), arc-greedy (Nivre, 2003), and swap (Nivre, 2009).

During the laboratories, you have implemented the arc-greedy version. In this part of the examination, you will study and program the arc-standard and swap versions.

As programming language, you can use Python, Java, Perl, or Prolog. You will focus on the program structure and not on the syntactic details. You can ignore the Python modules, Java packages or imports for instance.

2.1 Parsing with Arc-Standard

Tables 4 and 5 show the definitions of the actions (transitions) involved in arc-greedy and arc-standard parsing. Arc-greedy in Table 4 is given as a reminder of what we used for the course assignments.

Table 4: The transitions in **arc-greedy** parsing, where W is the initial word list; I, the current input word list; A, the graph of dependencies; and S, the stack. The triple $\langle S, I, A\rangle$ represents the parser state. n, n', and n'' are lexical tokens. The pair (n', n) represents an arc from the head n' to the modifier n

Actions	**Parser states**	**Conditions**
Initialization	$\langle nil, W, \emptyset\rangle$	
Termination	$\langle S, [], A\rangle$	
Shift	$\langle S, [n\|I], A\rangle \rightarrow \langle [S\|n], I, A\rangle$	
Reduce	$\langle [S\|n], I, A\rangle \rightarrow \langle S, I, A\rangle$	$\exists n'(n', n) \in A$
Left-arc	$\langle [S\|n], [n'\|I], A\rangle \rightarrow \langle S, [n'\|I], A \cup \{(n \leftarrow n')\}\rangle$	$\nexists n''(n'', n) \in A$
Right-arc	$\langle [S\|n], [n'\|I], A\rangle \rightarrow \langle [S\|n, n'], I, A \cup \{(n \rightarrow n')\}\rangle$	

2.1.1 Arc-Standard applied to Projective Graphs

Table 6 shows a manually-parsed sentence from Talbanken and the Swedish corpus in CoNLL-X:

> Denna typ uppvisar många fördelar
> 'This type exhibits many advantages'

In this section, you will manually apply the arc-standard algorithm to this sentence. To make parsing easier, add a dummy word called ROOT at index 0:

0 ROOT _ ROOT ROOT _ 0 ROOT _ _

1. Draw the dependency graph of the sentence in Table 6. 3 points

Table 5: The transitions in **arc-standard** parsing, where W is the initial word list; I, the current input word list; A, the graph of dependencies; and S, the stack. The triple $\langle S, I, A\rangle$ represents the parser state. n, n', and n'' are lexical tokens. The pair (n', n) represents an arc from the head n' to the modifier n

Actions	**Parser states**	**Conditions**
Initialization	$\langle nil, W, \emptyset\rangle$	
Termination	$\langle [ROOT], [], A\rangle$	
Shift	$\langle S, [n\|I], A\rangle \rightarrow \langle [S\|n], I, A\rangle$	
Left-arc	$\langle [S\|n, n'], I, A\rangle \rightarrow \langle [S\|n'], I, A \cup \{(n \leftarrow n')\}\rangle$	$n \neq ROOT$
Right-arc	$\langle [S\|n, n'], I, A\rangle \rightarrow \langle [S\|n], I, A \cup \{(n \rightarrow n')\}\rangle$	

Table 6: Dependency graph of the sentence *Denna typ uppvisar många fördelar* from Talbanken and CoNLL-X

id	form	lemma	cpostag	postag	feats	head	deprel	phead	pdeprel
1	Denna	_	PO	PO	_	2	DT	_	_
2	typ	_	NN	NN	_	3	SS	_	_
3	uppvisar	_	VV	VV	_	0	ROOT	_	_
4	många	_	PO	PO	_	5	DT	_	_
5	fördelar	_	NN	NN	_	3	OO	_	_

2. Parse manually the sentence in Table 6 using arc-standard: i.e. Find the sequence of transitions. You have to write a transition sequence and visualize the corresponding queue, stack, and graph at each step. 8 points

2.1.2 Programming

In this section, you will write a program to parse a hand-annotated sentence using arc-standard. The parser state will consist of the `sentence`, a `stack`, a `queue`, and a `graph`.

Data Structure. You will assume that the sentence graph is stored in a `sentence` variable consisting of a list of words, where each word is a Python dictionary (or a `Map` if you use Java). In Python, the sentence in Table 6 will look like this:

```
[{'phead': '0', 'form': 'ROOT', 'pdeprel': 'ROOT',
  'feats': 'ROOT', 'cpostag': 'ROOT', 'id': '0', 'head': '0',
  'postag': 'ROOT', 'lemma': 'ROOT', 'deprel': 'ROOT'},
{'phead': '_', 'form': 'Denna', 'pdeprel': '_', 'feats': '_',
  'cpostag': 'PO', 'id': '1', 'head': '2', 'postag': 'PO',
  'lemma': '_', 'deprel': 'DT'},
{'phead': '_', 'form': 'typ', 'pdeprel': '_', 'feats': '_',
  'cpostag': 'NN', 'id': '2', 'head': '3', 'postag': 'NN',
  'lemma': '_', 'deprel': 'SS'},
```

`...]`

For the parser state, you can use the same data structure as in the labs: lists for the `stack` and the `queue`, and a dictionary for the `graph`, where `graph['heads']` will store the heads and `graph['deprels']`, the grammatical functions. You can also define the structure you want, provided that you describe it.

Programming the Transitions. Program the transitions in Table 5 in the programming language you selected: left-arc, right-arc, and shift for the arc-standard parser. Use one function or method per transition. 18 points

Programming an Oracle. Describe how the automatic parser would apply the transitions to a manually-parsed sentence so that it can recreate the same dependency graph (gold-standard parsing). Such a procedure is called an oracle. You will first describe the algorithm in your own language (Swedish, English, or a language I understand) and then program it:

1. You will start with when to apply a left-arc, which is simpler. 5 points
2. For right-arc, be sure that you do not pop the top of the stack too early. The parser needs to collect all its children still in the queue before the parser creates a right-arc and pops the top of the stack; 5 points
3. Finally, when will the parser apply a shift? 2 points
4. Program the description of your oracle for the arc-standard parser in the programming language you selected. It consists of conditions on when to apply left-arc, right-arc, and shift. You will call this function or method `oracle()` (or `reference()` as in the labs). 20 points

2.2 Nonprojective Graphs

2.2.1 Arc-Standard applied to Nonprojective Graphs

In this section, you will consider the sentence in Table 7:

> Vad beror detta på?
> 'What does this depend on?'

In the exercises below, ignore the question mark at the end of the sentence so that you can save time.

1. Draw the dependency graph of the nonprojective sentence in Table 7. 3 points
2. Show that this sentence cannot be parsed by arc-standard or arc-greedy (you can choose the one you want). Start the parsing and explain why you cannot complete it. (Ignore the question mark). 6 points
3. Try to explain why arc-greedy or arc-standard cannot parse nonprojective sentences. 6 points

Table 7: Dependency graph of the sentence *Vad beror detta på?* from Talbanken and CoNLL-X

id	form	lemma	cpostag	postag	feats	head	deprel	phead	pdeprel
1	Vad	_	AB	AB	_	4	PA	_	_
2	beror	_	VV	VV	_	0	ROOT	_	_
3	detta	_	PO	PO	_	2	SS	_	_
4	på	_	PR	PR	_	2	OA	_	_
5	?	_	I?	I?	_	2	I?	_	_

2.3 Finding the Nonprojective Graphs

In this section, you will program a function to determine if a graph is projective or not.

2.3.1 Definition

In projective graphs, each pair of words $(Dep, Head)$, which are directly connected, is only separated by direct or indirect dependents of $Head$ or Dep. All the words in-between are hence dependents of $Head$. This can be restated more formally as: for all dependency relations in a sentence between a word w_i and its head w_{h_i}, either direction, $w_i \leftarrow w_{h_i}$, respectively $w_{h_i} \rightarrow w_i$, and $\forall w_j$, so that $i < j < h_i$, respectively $h_i < j < i$, w_j is transitively connected to w_{h_i}. In a dependency graph, projectivity results in the absence of crossing arcs. Nonprojective graphs are graphs that contain at least one nonprojective link.

Inside a segment defined by w_i, a word of index i, and w_{h_i}, its head of index h_i, all the words are transitively connected to w_{h_i}. The identification of nonprojective arcs is carried out using the negation of this property. A dependency arc (i, h_i) is nonprojective if there is at least one word w_j which has its index inside the range $i..h_i$ and none of this word's transitive heads is w_{h_i}.

2.3.2 Programming

Write a function that determines if a graph is projective or not and if not returns the list of nonprojective links. 20 points

As a suggestion, you can split this function in two:

1. Write first a function `connected(word, head, sentence)` that determines if the two words `word` and `head` are connected by a direct or indirect dependency chain: If there is a sequence of links (heads) between `word` and `head`. In Table 7, *denna* and *uppvisar* are connected (*uppvisar* → *typ* → *denna*) while *denna* and *många* are not;

2. Write a `nonproj_links(sentence)` function that checks that for every word in the sentence, all the words between this word and its head are direct or indirect dependents of it head (are directly or indirectly connected to its head).

2.4 Parsing Nonprojective Graphs

Nivre (2009) extended the arc-standard parser with a `swap` transition to parse nonprojective sentences. The swap transition enables the parser to reorder the words. Table 8 shows the definition of the transitions using in swap.

Nivre (2009)'s article is provided as a reference, where the important parts are marked by a vertical line in the margin. This text is difficult to read. Read only the paragraphs you need, if you feel this complements the text of the examination. (You only need it really for the question in Sect. 2.5.)

Table 8: The parser transitions in **swap** parsing, where W is the initial word list; I, the current input word list; A, the graph of dependencies; and S, the stack. The triple $\langle S, I, A\rangle$ represents the parser state. n, n', and n'' are lexical tokens. The pair (n', n) represents an arc from the head n' to the modifier n

Actions	**Parser states**	**Conditions**
Initialization	$\langle nil, W, \emptyset\rangle$	
Termination	$\langle [ROOT], [], A\rangle$	
Shift	$\langle S, [n\|I], A\rangle \rightarrow \langle [S\|n], I, A\rangle$	
Left-arc	$\langle [S\|n, n'], I, A\rangle \rightarrow \langle [S\|n'], I, A \cup \{(n \leftarrow n')\}\rangle$	$n \neq ROOT$
Right-arc	$\langle [S\|n, n'], I, A\rangle \rightarrow \langle [S\|n], I, A \cup \{(n \rightarrow n')\}\rangle$	
Swap	$\langle [S\|n, n'], I, A\rangle \rightarrow \langle [S\|n'], [n\|I], A\rangle$	$n \neq ROOT$ and $inx(n) < inx(n')$

2.4.1 Programming

Using the same data structure as in in Sect. 2.1.2, program the `swap` transition. As with `left_arc()`, `right_arc()`, and `shift()`, you will use a function or method. 6 points

2.4.2 Projective Order

In this section, you will parse manually the sentence in Table 7 with swap.

The key to parsing a nonprojective graph is to traverse it using an in-order traversal that considers the left and and then the right children of a word. This traversal is defined recursively by the `inorder` function below. The projective order results in a new ordering of the words that makes the sentence projective. It is of course not the original sequence that is not projective. This sequence is returned in `projective_order`, which is set to an empty list before the function is called.

You will first answer a few questions to help you understand the projective order and then parse the sentence:

1. What are the right modifiers (children) of *beror*? List them in the sentence order? 2 points

2. What are the left modifier(s) (child/children) of *på*? List them (it) in the sentence order? How should you move this word to have a projective

graph? Hint: Move this word to the right, one word at a time, until the sentence is projective. 4 points

3. Starting from ROOT, traverse the sentence in Table 7 in order and write the new ordering: The sequence of indices that makes the sentence projective. If you do not know what in order traversal is, look at the `inorder` function. You should find a sequence `[0, 2, ...]`, where you have moved one index so that the graph is projective 6 points

4. Using swap, find the sequence of transitions that parses the sentence in Table 7. If the next word in the projective order is the kth in the queue, you will need to carry out k shifts and $k-1$ swaps so that you can create a link between the top and second in the stack. Visualize the corresponding queue, stack, and graph at each step. (Ignore the question mark). 12 points

```
def inorder(current_word, sentence, projective_order):
    """
    In order traversal of a dependency graph
    :param current_word:
    :param sentence:
    :param projective_order:
    :return:
    """
    if current_word == []:
        return
    left_children, right_children = get_children(current_word, sentence)
    for word in left_children:
        inorder(word, sentence, projective_order)
    print(current_word['id'], end=' ', flush=True)
    projective_order.append(current_word['id'])
    for word in right_children:
        inorder(word, sentence, projective_order)


def get_children(current_word, sentence):
    """
    Returns the lists of sorted left
    and right children (modifiers) of a word
    :param current_word:
    :param sentence:
    :return:
    """
    left_children = []
    right_children = []
    for word in sentence:
        if word['head'] == current_word['id']:
            if int(word['id']) < int(current_word['id']):
                left_children.append(word)
            if int(word['id']) > int(current_word['id']):
                right_children.append(word)
    return left_children, right_children
```

2.5 An Oracle for Nonprojective Graphs

Nivre (2009, p. 355 and Fig. 4) describes an oracle to parse manually-annotated nonprojective sentences with the swap transition. Program this oracle. This oracle is nearly identical to the one in Sect. 2.1.2. You just need to insert a condition when the `swap()` function is called. 18 points

References

Nivre, J. (2003). An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, pages 149–160, Nancy.

Nivre, J. (2009). Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 351–359, Suntec, Singapore. Association for Computational Linguistics.

Yamada, H. and Matsumoto, Y. (2003). Statistical dependency analysis with support vector machines. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, pages 195–206, Nancy.