DaNLP

Alexandra Institute

Feb 22, 2022

GETTING STARTED

DaNLP is a repository for Natural Language Processing (NLP) resources for the Danish Language. It is a collection of available datasets and models for a variety of NLP tasks. The aim is to make it easier and more applicable to practitioners in the industry to use Danish NLP and hence this project is licensed to allow commercial use. The project features code examples on how to use the datasets and models in popular NLP frameworks such as spaCy, Transformers and Flair as well as Deep Learning frameworks such as PyTorch.

If you are new to NLP or want to know more about the project in a broader perspective, you can have a look at our microsite (in Danish).

CHAPTER

ONE

INSTALLATION

To get started using DaNLP in your python project simply install the pip package. However installing the pip package will not install all NLP libraries because we want you to have the freedom to limit the dependency on what you use.

1.1 Install with pip

To get started using DaNLP simply install the project with pip:

```
pip install danlp
```

Note that the default installation of DaNLP does not install other NLP libraries such as Gensim, SpaCy, flair or Transformers. This allows the installation to be as minimal as possible and let the user choose to e.g. load word embeddings with either spaCy, flair or Gensim. Therefore, depending on the function you need to use, you should install one or several of the following: `pip install flair`, `pip install spacy` or/and `pip install gensim`.

Alternatively if you want to install all the required dependencies including the packages mentionned above, you can do:

```
pip install danlp[all]
```

You can check the `requirements.txt` file to see what version the packages has been tested with.

1.2 Install from source

If you want to be able to use the latest developments before they are released in a new pip package, or you want to modify the code yourself, then clone this repo and install from source.

```
git clone https://github.com/alexandrainst/danlp.git
cd danlp
# minimum installation
pip install .
# or install all the packages
pip install .[all]
```

To install the dependencies used in the package with the tested versions:

```
pip install -r requirements.txt
```

1.3 Install from GitHub

Alternatively you can install the latest version from GitHub using:

```
pip install git+https://github.com/alexandrainst/danlp.git
```

1.4 Install with Docker

To quickly get started with DaNLP and to try out the models you can use our Docker image. To start a ipython session simply run:

```
docker run -it --rm alexandrainst/danlp ipython
```

If you want to run a `<script.py>` in your current working directory you can run:

```
docker run -it --rm -v "$PWD":/usr/src/app -w /usr/src/app alexandrainst/danlp python
↪<script.py>
```

CHAPTER

TWO

QUICK START

Once you have installed the DaNLP package, you can use it in your python project using `import danlp`.

You will find the main functions through the `models` and `datasets` modules – see the library documentation for more details about how to use the different functions for loading models and datasets. For analysing texts in Danish, you will primarily need to import functions from `danlp.models` in order to load and use our pre-trained models.

The DaNLP package provides you with several models for different NLP tasks using different frameworks. On this section, you will have a quick tour of the main functions of the DaNLP package. For a more detailed description of the tasks and frameworks, follow the links to the documentation:

- Embedding of text with flair, transformers, spaCy or Gensim
- Part of speech tagging (POS) with spaCy or flair
- Named Entity Recognition (NER) with spaCy, flair or BERT
- Sentiment Analysis with spaCy or BERT
- Dependency parsing and NP-chunking with spaCy

You can also try out our getting started jupyter notebook for quicly learning how to load and use the DaNLP models and datasets.

2.1 All-in-one with the spaCy models

To quickly get started with DaNLP and try out different NLP tasks, you can use the spaCy model (see also). The main advantages of the spaCy model is that it is fast and it includes most of the basic NLP tasks that you need for pre-processing texts in Danish.

The main functions are:

- `load_spacy_model` for loading a spaCy model for POS, NER and dependency parsing or a spaCy sentiment model
- `load_spacy_chunking_model` for loading a wrapper around the spaCy model with which you can deduce NP-chunks from dependency parses

2.1.1 Pre-processing tasks

Perform Part-of-Speech tagging, Named Entity Recognition and dependency parsing at the same time with the DaNLP spaCy model. Here is a snippet to quickly getting started:

```
# Import the load function
from danlp.models import load_spacy_model

# Download and load the spaCy model using the DaNLP wrapper function
nlp = load_spacy_model()

# Parse the text using the spaCy model
# it creates a spaCy Doc object
doc = nlp("Niels Henrik David Bohr var en dansk fysiker fra København")

# prepare some pretty printing
features = ['Text','POS', 'Dep', 'NE']
head_format ="\033[1m{!s:>11}\033[0m" * (len(features) )
row_format ="{!s:>11}" * (len(features) )

print(head_format.format(*features))
# printing for each token in the docs the pos, dep and entity features
for token in doc:
    print(row_format.format(token.text, token.pos_, token.dep_, token.ent_type_))

# printing the list of entities (and their category)
for ent in doc.ents:
    print(ent.text, '-', ent.label_)
```

For NP-chunking you can use the `load_spacy_chunking_model`. The spaCy chunking model includes the spaCy model – which can be used as previously described.

```
from danlp.models import load_spacy_chunking_model

# text to process
text = 'Et syntagme er en gruppe af ord, der hænger sammen'

# Load the chunker using the DaNLP wrapper
chunker = load_spacy_chunking_model()
# Applying the spaCy model for parsing the sentence
# and deducing NP-chunks
np_chunks = chunker.predict(text, bio=False)

nlp = chunker.model
doc = nlp(text)

# print the chunks
for (start_id, end_id, _) in np_chunks:
    print(doc[start_id:end_id])
```

2.1.2 Sentiment analysis

With the spaCy sentiment model, you can predict whether a sentence is perceived positive, negative or neutral. For loading and using the spaCy sentiment analyser, follow these steps:

```
from danlp.models import load_spacy_model

# Download and load the spaCy sentiment model using the DaNLP wrapper function
nlpS = load_spacy_model(textcat='sentiment', vectorError=True)

text = "Jeg er meget glad med DaNLP"

# analyse the text using the spaCy sentiment analyser
doc = nlpS(text)

# print the most probable category among 'positiv', 'negativ' or 'neutral'
print(max(doc.cats, key=doc.cats.get))
```

2.2 Sequence labelling with flair

For part-of-speech tagging and named entity recognition, you also have the possibility to use flair. If you value precision rather than speed, we would recommend you to use the flair models (or BERT NER, next section).

Perform POS tagging or NER using the DaNLP flair models that you can load through the following functions:

- `load_flair_pos_model`
- `load_flair_ner_model`

Use the following snippet to try out the flair POS model. Note that the text should be pre-tokenized.

```
from danlp.models import load_flair_pos_model
from flair.data import Sentence

text = "Hans har en lille sort kat ."
sentence = Sentence(text)

tagger = load_flair_pos_model()

tagger.predict(sentence)

for tok in sentence.tokens:
    print(tok.text, tok.get_tag('upos').value)
```

You can use the flair NER model in a similar way.

```
from danlp.models import load_flair_ner_model
from flair.data import Sentence

text = "Hans bor i København"
sentence = Sentence(text)

tagger = load_flair_ner_model()

tagger.predict(sentence)
```

(continues on next page)

(continued from previous page)

```
for tok in sentence.tokens:
    print(tok.text, tok.get_tag('ner').value)
```

2.3 Deep NLP with BERT

2.3.1 NER with BERT

You can also perform NER with BERT. Load the DaNLP model with `load_bert_ner_model` and try out the following snippet:

```
from danlp.models import load_bert_ner_model
bert = load_bert_ner_model()
# Get lists of tokens and labesl in IBO format
tokens, labels = bert.predict("Jens Peter Hansen kommer fra Danmark")
print(" ".join(["{}/{}".format(tok,lbl) for tok,lbl in zip(tokens,labels)]))

# To get a "right" tokenization provide it your self (SpaCy can be used for this) by␣
↪providing a a list of tokens
# With this option, output can also be choosen to be a dict with tags and position␣
↪instead of IBO format
tekst_tokenized = ['Han', 'hedder', 'Anders', 'And', 'Andersen', 'og', 'bor', 'i',
↪'Århus', 'C']
bert.predict(tekst_tokenized, IOBformat=False)
"""
{'text': 'Han hedder Anders And Andersen og bor i Århus C',
 'entities': [{'type': 'PER','text':'Anders And Andersen','start_pos': 11,'end_pos':␣
↪30},
  {'type': 'LOC', 'text': 'Århus C', 'start_pos': 40, 'end_pos': 47}]}
"""
```

2.3.2 Classification with BERT

BERT is well suited for classification tasks. You can load the DaNLP sentiment classification BERT models with:

- `load_bert_emotion_model`
- `load_bert_tone_model`

With the BERT Emotion model you can classify sentences among eight emotions:

- `Glæde/Sindsro`
- `Tillid/Accept`
- `Forventning/Interrese`
- `Overasket/Målløs`
- `Vrede/Irritation`
- `Foragt/Modvilje`
- `Sorg/trist`
- `Frygt/Bekymret`

Following is an example of how to use the BERT Emotion model:

```
from danlp.models import load_bert_emotion_model
classifier = load_bert_emotion_model()

# using the classifier
classifier.predict('der er et træ i haven')
''''No emotion''''
classifier.predict('jeg ejer en rød bil og det er en god bil')
''''Tillid/Accept''''
classifier.predict('jeg ejer en rød bil men den er gået i stykker')
''''Sorg/trist''''

# Get probabilities and matching classes
probas = classifier.predict_proba('jeg ejer en rød bil men den er gået i stykker', no_
↪emotion=False)[0]
classes = classifier._classes()[0]

for c, p in zip(classes, probas):
    print(c, ':', p)
```

With the BERT Tone model, you can predict the tone (`objective` or `subjective`) or the polarity (`positive`, `negative` or `neutral`) of sentences.

```
from danlp.models import load_bert_tone_model
classifier = load_bert_tone_model()

text = 'Analysen viser, at økonomien bliver forfærdelig dårlig'

# using the classifier
prediction = classifier.predict(text)
print("Tone: ", prediction['analytic'])
print("Polarity: ", prediction['polarity'])

# Get probabilities and matching classes
probas = classifier.predict_proba(text)[0]
classes = classifier._classes()[0]

for c, p in zip(classes, probas):
    print(c, ':', p)
```

CHAPTER THREE

CONTRIBUTING

If you want to contribute to the DaNLP project, your help is very welcome. You can contribute to the project in many ways:

- Help us write good tutorials on Danish NLP use-cases
- Contribute with your own pretrained NLP models or datasets in Danish (see our contributing guidelines on our GitHub page for more details on how to contribute to the repository)
- Create GitHub issues with questions and bug reports
- Notify us of other Danish NLP resources or tell us about any good ideas that you have for improving the project through the Discussions section of the GitHub repository.

CHAPTER

FOUR

TASKS

4.1 Pretrained Danish embeddings

This repository keeps a list of pretrained word embeddings publicly available in Danish. The `download_embeddings.py` and `load_embeddings.py` provides functions for downloading the embeddings as well as prepare them for use in popular NLP frameworks.

Embeddings are a way of representing text as numeric vectors, and can be calculated both for chars, subword units (Sennrich et al. 2016), words, sentences or documents. The methods for training embeddings, can roughly be categorized into static embeddings and dynamic embeddings.

4.1.1 Static embeddings

Static word embeddings contains a large vocabulary of words and each word has a vector representation associated. To get a representation of a word is simply a look-up in the vocabulary to get the associated vector. An example of this type of embeddings is word2vec (Mikolov et al. 2013). Relying on a vocabulary of words can result in out-of-vocabulary words. To cope with this fastText (Bojanowski et al. 2017) uses subword units that constructs a word embedding from the character n-gram embeddings occurring in the word.

4.1.2 Dynamic embeddings

Dynamic embeddings are contextual in the sense that the representations are dependent on the sentence they appear in. This way homonyms get different vector representations. An example of dynamic embeddings is the Flair embeddings (Akbik et al. 2018) where the embeddings are trained with the task of language modelling ie. learning to predict the next character in a sentence.

4.1.3 Benchmarks

To evaluate word embeddings it is common to do intrinsic evaluations to directly test for syntactic or semantic relationships between words. The Danish Similarity Dataset and WordSim-353 contains word pairs annotated with a similarity score. Calculating the correlation between the word embedding similarity and the similarity score gives and indication of how well the word embeddings captures relationships between words.

4.1.4 Get started using word embeddings

Word embeddings are essentially a representation of a word in a n-dimensional space. Having a vector representation of a word enables us to find distances between words. In `load_embeddings.py` we have provided functions to download pretrained word embeddings and load them with the two popular NLP frameworks spaCy and Gensim.

This snippet shows how to automatically download and load pretrained static word embeddings e.g. trained on the CoNLL17 dataset, and it show some analysis the embeddings can be used for:

```
from danlp.models.embeddings  import load_wv_with_gensim, load_wv_with_spacy

# Load with gensim
word_embeddings = load_wv_with_gensim('conll17.da.wv')

word_embeddings.most_similar(positive=['københavn', 'england'], negative=['danmark'],␣
↪topn=1)
# [('london', 0.7156291604042053)]

word_embeddings.doesnt_match("sodavand brød vin juice".split())
# 'brød'

word_embeddings.similarity('københavn', 'århus')
# 0.7677029

word_embeddings.similarity('københavn', 'esbjerg')
# 0.59988254


# Load with spacy
word_embeddings = load_wv_with_spacy('conll17.da.wv')
```

Flair embeddings

This repository provides pretrained Flair word embeddings trained on Danish data from Wikipedia and EuroParl both forwards and backwards. To see the code for training the Flair embeddings have a look at Flairs GitHub.

The hyperparameter are set as follows: `hidden_size=1032`, `nlayers=1`, `sequence_length=250`, `mini_batch_size=50`, `max_epochs=5`

The trained Flair word embeddings has been used in training a Part of Speech tagger and Name Entity Recognition tagger with Flair, check it out in the docs for pos and ner .

In the snippet below you can see how to load the pretrained flair embeddings and an example of simple use.

```
from danlp.models.embeddings import load_context_embeddings_with_flair
from flair.data import Sentence

# Use the wrapper from DaNLP to download and load embeddings with Flair
# You can combine it with the static embeddings
stacked_embeddings = load_context_embeddings_with_flair(word_embeddings='wiki.da.wv')

# Embed two different sentences
sentence1 = Sentence('Han fik bank')
sentence2 = Sentence('Han fik en ny bank')
stacked_embeddings.embed(sentence1)
stacked_embeddings.embed(sentence2)
```

(continues on next page)

(continued from previous page)

```
# Show that it is contextual in the sense 'bank' has different embedding after context
print('{} dimensions out of {} is equal'.format(int(sum(sentence2[4].
↪embedding==sentence1[2].embedding)), len(sentence1[2].embedding)))
# 1332 ud af 2364
```

BERT embeddings

BERT is a language model but the different layers can be used as embeddings of tokens or sentences. This code loads a pytorch version using the Transformers library from HuggingFace of pre-trained Danish BERT representations by BotXO model. Since the models is not a designated models for embeddings, some choices is made of what layers to use. For each tokens in a sentence there is 13 layers of dim 768. Based on the blogpost, it has been chosen to concatenate the four last layer to use as token embeddings, which gives a dimension of 4*768=3072. For sentence embeddings the second last layers is used and the mean across all tokens in the sentence is calculated.

Note, BERT tokenize out of vocabulary words into sub words.

Below is a small code snippet for getting started:

```
from danlp.models import load_bert_base_model
model = load_bert_base_model()
vecs_embedding, sentence_embedding, tokenized_text =model.embed_text('Han sælger frugt
↪')
```

4.1.5 References

- Thomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado and Jeffrey Dean. 2013. Distributed Representations of Words and Phrasesand their Compositionality. In **NeurIPS**.
- Piotr Bojanowski, Edouard Grave, Armand Joulin and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. In **ACL**.
- Rico Sennrich, Barry Haddow and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In **ACL**.
- Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. 2002. Placing Search in Context: The Concept Revisited. In **ACM TOIS**.
- Alan Akbik, Duncan Blythe and Roland Vollgraf. 2018. Contextual String Embeddings for Sequence Labeling. In **COLING**.

4.2 Part of Speech Tagging

This section is concerned with public available Part of Speech (POS) taggers in Danish.

Part-of-speech tagging is the task of classifying words into their part-of-speech, based on both their definition and context. Parts of speech are also known as word classes, which describe the role of the words in sentences relatively to their neighbors (such as verbs and nouns).

The Danish UD treebank uses 17 universal part of speech tags:

`ADJ`: Adjective, `ADP`: Adposition , `ADV`: Adverb, `AUX`: Auxiliary verb, `CCONJ`: Coordinating conjunction, `DET`: Determiner, `INTJ`: Interjection, `NOUN`: Noun, `NUM`: Numeral, `PART`: Particle `PRON`: Pronoun `PROPN`: Proper noun `PUNCT`: Punctuation `SCONJ`: Subordinating conjunction `SYM`: Symbol `VERB`: Verb `X`: Other

4.2.1 Use cases

Part-of-Speech tagging in itself may not be the solution to any particular NLP problem. It is a task that is mostly used as a pre-processing step in order to make it easier for more applicative problems.

It can be used for cleaning/filtering text (e.g. removing punctuation, extracting only nouns) or disambiguation. The language is inherently ambiguous. For instance in Danish, `fisk` is a verb or a noun. In a sentence such as `Fisk en fisk`, a part-of-speech tagger is required to understand that the first one is a verb and the second is a noun in such a way that this information can be used in downstream tasks, for example for machine translation (in French it would be two different words depending on whether `fisk` is a verb – `pêche` – or a noun – `poisson`) or for text to speech conversion (a same word can be pronounced differently depending on its meaning).

A medium blog using Part of Speech tagging on Danish, can be found here.

4.2.2 Models

Flair

This project provides a trained part of speech tagging model for Danish using the Flair framework from Zalando, based on the paper Akbik et. al (2018). The model is trained using the data Danish Dependency Treebank and by using FastText word embeddings and Flair contextual word embeddings trained in this project on data from Wikipedia and EuroParl corpus, see here.

The code for training can be found on Flairs GitHub, and the following parameters are set: `learning_rate=1`, `mini_batch_size=32`, `max_epochs=150`, `hidden_size=256`.

The flair pos tagger can be used by loading it with the `load_flair_pos_model` method. Please note that the text should be tokenized before hand, this can for example be done using spaCy.

```
from danlp.models import load_flair_pos_model
from flair.data import Sentence

# Load the POS tagger using the DaNLP wrapper
tagger = load_flair_pos_model()

# Using the flair POS tagger
sentence = Sentence('Jeg hopper på en bil , som er rød sammen med Niels .')
tagger.predict(sentence)
print(sentence.to_tagged_string())

# Example
'''Jeg <PRON> hopper <VERB> på <ADP> en <DET> bil <NOUN> , <PUNCT> som <ADP> er <AUX>␣
↪rød <ADJ> sammen <ADV> med <ADP> Niels <PROPN> . <PUNCT>
'''
```

SpaCy

Read more about the spaCy model in the dedicated spaCy docs , it has also been trained using the Danish Dependency Treebank data.

Below is a small getting started snippet for using the spaCy POS tagger:

```
from danlp.models import load_spacy_model

#Load the POS tagger using the DaNLP wrapper
```

(continues on next page)

(continued from previous page)

```
nlp = load_spacy_model()

# Using the spaCy POS tagger
doc = nlp('Jeg hopper på en bil, som er rød sammen med Niels.')
pred=''
for token in doc:
    pred += '{} <{}> '.format(token.text, token.pos_)
print(pred)

# Example
''' Jeg <PRON> hopper <VERB> på <ADP> en <DET> bil <NOUN> , <PUNCT> som <ADP> er <AUX>
↪ rød <ADJ> sammen <ADV> med <ADP> Niels <PROPN> . <PUNCT>
 '''
```

DaCy

DaCy is a multi-task transformer trained using SpaCy v. 3. its models is fine-tuned (on DaNE) and based upon the Danish BERT (v2) by botXO and the XLM Roberta large. For more on DaCy see the github repository or the blog post describing the training procedure.

Stanza

Stanza is a python library which provides a neural network pipeline for NLP in many languages. It has been developed by the Stanford NLP Group. The Stanza part-of-speech tagger has been trained on the DDT.

4.2.3 Benchmarks

Accuracy scores are reported below and can be reproduced using `pos_benchmarks.py` in the example folder, where the details score from each class is calculated.

DaNLP

*Sentences per second is based on a Macbook Pro with Apple M1 chip.

4.2.4 References

- Rami Al-Rfou, Bryan Perozzi, and Steven Skiena. 2013. Polyglot: Distributed Word Representations for Multilingual NLP. In **CoNLL**.
- Alan Akbik, Duncan Blythe, and Roland Vollgraf. 2018. Contextual String Embeddings for Sequence Labeling. In **COLING**.

4.3 Dependency Parsing & Noun Phrase Chunking

4.3.1 Dependency Parsing

Dependency parsing is the task of extracting a dependency parse of a sentence. It is typically represented by a directed graph that depicts the grammatical structure of the sentence; where nodes are words and edges define syntactic relations between those words. A dependency relation is a triplet consisting of: a head (word), a dependent (another word) and a dependency label (describing the type of the relation).

The model has been trained on the Danish UD treebank which have been annotated with dependencies following the Universal Dependency scheme. It uses 39 dependency relations.

4.3.2 Noun Phrase Chunking

Chunking is the task of grouping words of a sentence into syntactic phrases (e.g. noun-phrase, verb phrase). Here, we focus on the prediction of noun-phrases (NP). Noun phrases can be pronouns (`PRON`), proper nouns (`PROPN`) or nouns (`NOUN`) – potentially bound with other tokens that act as modifiers, e.g., adjectives (`ADJ`) or other nouns. In sentences, noun phrases are generally used as subjects (`nsubj`) or objects (`obj`) (or complements of prepositions). Examples of noun-phrases :

- en `bog` (NOUN)
- en `god bog` (ADJ+NOUN)
- `Lines bog` (PROPN+NOUN)

NP-chunks can be deduced from dependencies. We provide a convertion function – from dependencies to NP-chunks – thus depending on a dependency model.

4.3.3 Use cases

Dependency parsing and chunking can be used as preprocessing steps for other NLP tasks. See for example the EPE shared task, where the performance of dependency parsers is evaluated through the ouput of downstream tasks such as:

- Biological Event Extraction
- Fine-Grained Opinion Analysis
- Negation Resolution

It can also be used, for instance, for the extraction of keyphrases or for building a knowledge graph (see our tutorial).

4.3.4 Models

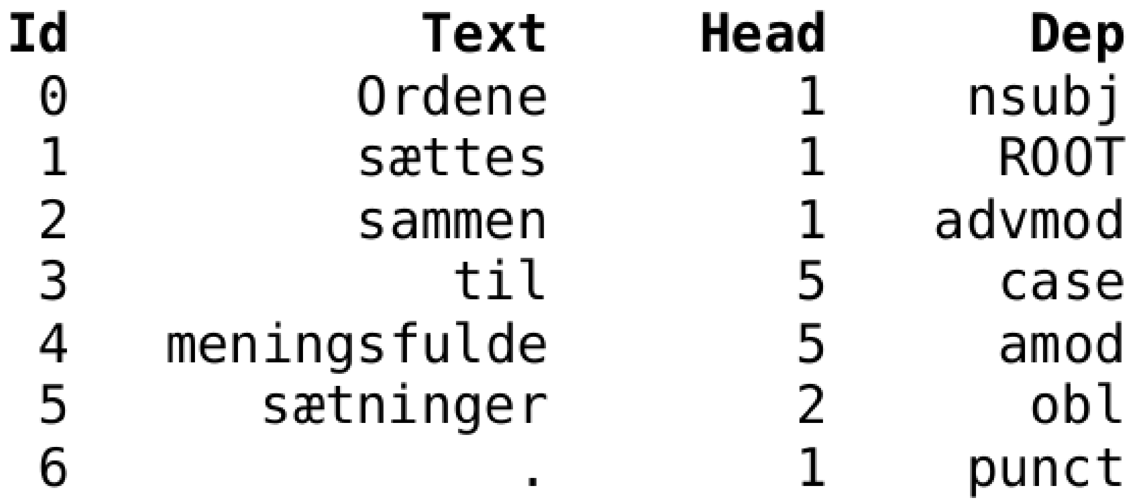
SpaCy

Read more about the SpaCy model in the dedicated SpaCy docs , it has also been trained using the Danish Dependency Treebank dataset.

Dependency Parser

Below is a small getting started snippet for using the SpaCy dependency parser:

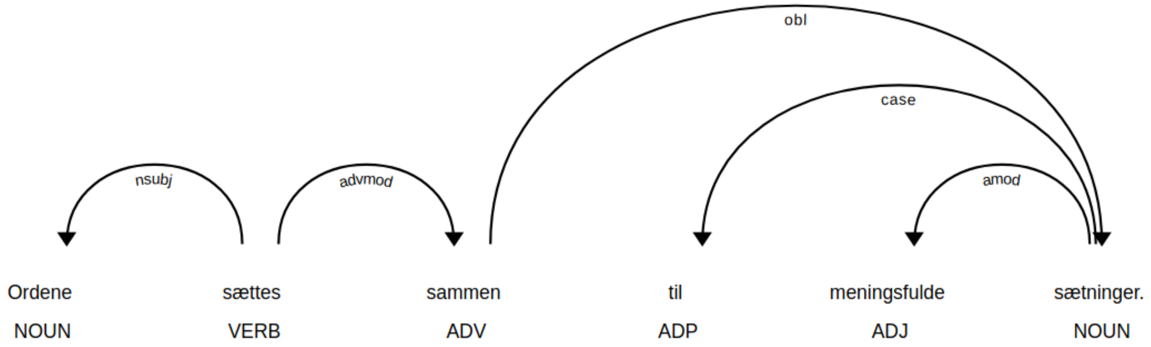
```
from danlp.models import load_spacy_model

# Load the dependency parser using the DaNLP wrapper
nlp = load_spacy_model()

# Using the spaCy dependency parser
doc = nlp('Ordene sættes sammen til meningsfulde sætninger.')

dependency_features = ['Id', 'Text', 'Head', 'Dep']
head_format = "\033[1m{!s:>11}\033[0m" * (len(dependency_features) )
row_format = "{!s:>11}" * (len(dependency_features) )

print(head_format.format(*dependency_features))
# Printing dependency features for each token
for token in doc:
    print(row_format.format(token.i, token.text, token.head.i, token.dep_))
```



```
         Id       Text       Head        Dep
          0     Ordene          1      nsubj
          1     sættes          1       ROOT
          2     sammen          1     advmod
          3        til          5       case
          4meningsfulde          5       amod
          5  sætninger          2        obl
          6          .          1      punct
```

Visualizing the dependency tree with SpaCy

```
# SpaCy visualization tool
from spacy import displacy

# Run in a terminal
# In jupyter use instead display.render
displacy.serve(doc, style='dep')
```

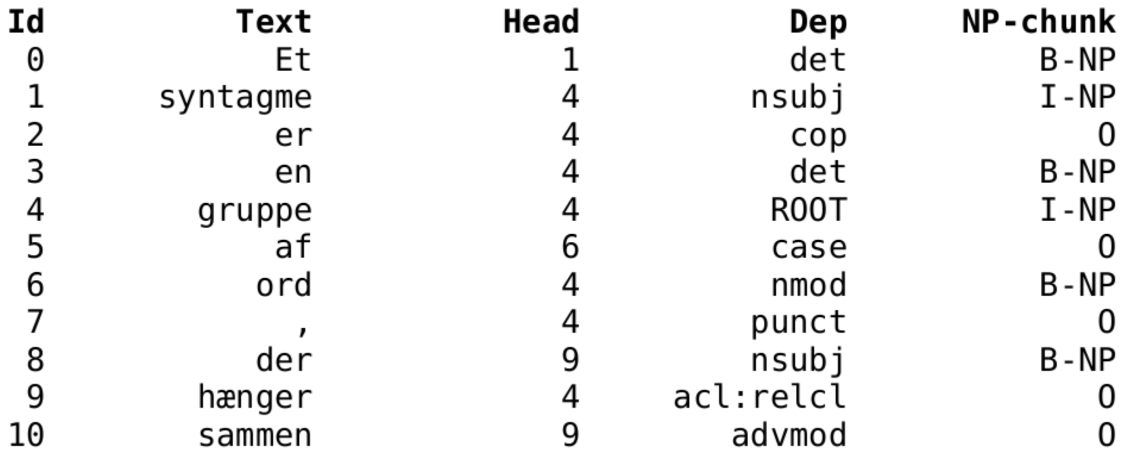


`nsubj`: nominal subject, `advmod`: adverbial modifier, `case`: case marking, `amod`: adjectival modifier, `obl`: oblique nominal, `punct`: punctuation

Chunker

Below is a snippet showing how to use the chunker:

```
from danlp.models import load_spacy_chunking_model

# text to process
text = 'Et syntagme er en gruppe af ord, der hænger sammen'

# Load the chunker using the DaNLP wrapper
chunker = load_spacy_chunking_model()
# Using the chunker to predict BIO tags
np_chunks = chunker.predict(text)

# Using the spaCy model to get linguistic features (e.g., tokens, dependencies)
# Note: this is used for printing features but is not necessary for processing the chunking task
# OBS - The model loaded is the same as can be loaded with 'load_spacy_model()'
nlp = chunker.model
doc = nlp(text)

syntactic_features=['Id', 'Text', 'Head', 'Dep', 'NP-chunk']
head_format ="\033[1m{!s:>11}\033[0m" * (len(syntactic_features) )
row_format ="{!s:>11}" * (len(syntactic_features) )

print(head_format.format(*syntactic_features))
# Printing dependency and chunking features for each token
for token, nc in zip(doc, np_chunks):
    print(row_format.format(token.i, token.text, token.head.i, token.dep_, nc))
```



Id	Text	Head	Dep	NP-chunk
0	Et	1	det	B-NP
1	syntagme	4	nsubj	I-NP
2	er	4	cop	O
3	en	4	det	B-NP
4	gruppe	4	ROOT	I-NP
5	af	6	case	O
6	ord	4	nmod	B-NP
7	,	4	punct	O
8	der	9	nsubj	B-NP
9	hænger	4	acl:relcl	O
10	sammen	9	advmod	O

DaCy

DaCy is a transformer-based version of the SpaCy model, thus obtaining higher performance, but with a higher computational cost. Read more about the DaCy model in the dedicated DaCy github, it has also been trained using the Danish Dependency Treebank dataset.

Stanza

Stanza is a python library which provides a neural network pipeline for NLP in many languages. It has been developed by the Stanford NLP Group. The Stanza dependency parser has been trained on the DDT.

4.3.5 Benchmarks

See detailed scoring of the benchmarks in the example folder.

Dependency Parsing Scores

Dependency scores — LA (labelled attachment score), UAS (Unlabelled Attachment Score) and LAS (Labelled Attachment Score) — are reported below :

Noun Phrase Chunking Scores

NP chunking scores (F1) are reported below :

*Sentences per second is based on a Macbook Pro with Apple M1 chip.

4.4 Named Entity Recognition

Named Entity Recognition (NER) is the task of extracting named entities in a raw text. Common entity types are locations, organizations and persons. Currently a few tools are available for NER in Danish. Popular models for NER (BERT, Flair and spaCy) are continuously trained on the newest available named entity datasets such as DaNE and made available through the DaNLP library.

4.4.1 Use cases

NER is one of the most famous NLP tasks used in the industry, probably because its use cases are pretty straightforward. It can be used in many systems, by itself or in combination with other NLP models. For instance, the extraction of entities from text can be used for :

- classifying / indexing documents (e.g articles for news providers) and then
- recommending similar content (e.g. news articles)
- customer support (e.g. for tagging tickets)
- analysing feedback from customers (product reviews)
- speeding up search engines
- extracting information (e.g from emails)
- building a structured database or a knowledge graph (see our example tutorial) from a corpus
- anonymizing documents.

4.4.2 Models

BERT

The BERT (Devlin et al. 2019) NER model is based on the pre-trained Danish BERT representations by BotXO which has been finetuned on the DaNE dataset (Hvingelby et al. 2020). The finetuning has been done using the Transformers library from HuggingFace.

The BERT NER model can be loaded with the `load_bert_ner_model()` method. Note that it can maximum take 512 tokens as input at a time. For longer text sequences split before hand, for example using sentence boundary detection (e.g. by using the spacy model.)

```
from danlp.models import load_bert_ner_model
bert = load_bert_ner_model()
# Get lists of tokens and labels in BIO format
tokens, labels = bert.predict("Jens Peter Hansen kommer fra Danmark")
print(" ".join(["{}/{}".format(tok,lbl) for tok,lbl in zip(tokens,labels)]))

# To get a correct tokenization, you have to provide it yourself to BERT  by␣
↪providing a list of tokens
# (for example SpaCy can be used for tokenization)
# With this option, output can also be choosen to be a dict with tags and position␣
↪instead of BIO format
tekst_tokenized = ['Han', 'hedder', 'Anders', 'And', 'Andersen', 'og', 'bor', 'i',
↪'Århus', 'C']
bert.predict(tekst_tokenized, IOBformat=False)
"""
{'text': 'Han hedder Anders And Andersen og bor i Århus C',
```

(continues on next page)

(continued from previous page)

```
 'entities': [{'type': 'PER','text':'Anders And Andersen','start_pos': 11,'end_pos': 30},
  {'type': 'LOC', 'text': 'Århus C', 'start_pos': 40, 'end_pos': 47}]}
"""
```

You can also find the BERT NER model on our HuggingFace page.

Flair

The Flair (Akbik et al. 2018) NER model uses pretrained Flair embeddings in combination with fastText word embeddings. The model is trained using the Flair library on the the DaNE dataset.

The Flair NER model can be used with DaNLP using the `load_flair_ner_model()` method.

```
from danlp.models import load_flair_ner_model
from flair.data import Sentence

# Load the NER tagger using the DaNLP wrapper
flair_model = load_flair_ner_model()

# Using the flair NER tagger
sentence = Sentence('Jens Peter Hansen kommer fra Danmark')
flair_model.predict(sentence)
print(sentence.to_tagged_string())
```

spaCy

The spaCy model is trained for several NLP tasks (read more here) using the DDT and DaNE annotations. The spaCy model can be loaded with DaNLP to do NER predictions in the following way.

```
from danlp.models import load_spacy_model

nlp = load_spacy_model()

doc = nlp('Jens Peter Hansen kommer fra Danmark')
for tok in doc:
    print("{} {}".format(tok,tok.ent_type_))
```

NERDA

NERDA is a python package that provides an interface for fine-tuning pretrained transformers for NER. It also includes some ready-to-use fine-tuned (on DaNE) NER models based on a multilingual BERT and a Danish Electra.

DaCy

DaCy is a multi-task transformer trained using SpaCy v.3. its models is fine-tuned (on DaNE) and based upon the Danish BERT (v2) by botXO and the XLM Roberta large. For more on DaCy see the github repository or the blog post describing the training procedure.

Daner

The daner(Derczynski et al. 2014) NER tool is a wrapper around the Stanford CoreNLP using data from (Derczynski et al. 2014) (not released). The tool is not available through DaNLP but it can be used from the daner repository.

DaLUKE

The DaLUKE model is based on the knowledge-enhanced transformer LUKE. It has been first pretrained as a language model on the Danish Wikipedia and then fine-tuned on DaNE for NER.

ScandiNER

The ScandiNER model can tag text for NER in Danish, Norwegian (Bokmål and Nynorsk), Swedish, Icelandic and Faroese. It is a fine-tuned version of NB-BERT-base, a language model for Norwegian. A combination of NER datasets have been used for training: DaNE, NorNE, SUC 3.0, WikiANN (for Icelandic and Faroese).

4.4.3 Benchmarks

The benchmarks has been performed on the test part of the DaNE dataset. None of the models have been trained on this test part. We are only reporting the scores on the `LOC`, `ORG` and `PER` entities as the `MISC` category has limited practical use. The table below has the achieved F1 score on the test set:

*Sentences per second is based on a Macbook Pro with Apple M1 chip.

The evaluation script `ner_benchmarks.py` can be found here.

4.4.4 References

- Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In **NAACL**.
- Leon Derczynski, Camilla V. Field and Kenneth S. Bøgh. 2014. DKIE: Open Source Information Extraction for Danish. In **EACL**.
- Alan Akbik, Duncan Blythe and Roland Vollgraf. 2018. Contextual String Embeddings for Sequence Labeling. In **COLING**.
- Rasmus Hvingelby, Amalie B. Pauli, Maria Barrett, Christina Rosted, Lasse M. Lidegaard and Anders Søgaard. 2020. DaNE: A Named Entity Resource for Danish. In **LREC**.
- Kevin Clark, Minh-Thang Luong, Quoc V. Le and Christopher D. Manning. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In **ICLR**.
- Ikuya Yamada, Akari Asai, Hiroyuki Shindo, Hideaki Takeda, Yuji Matsumoto. LUKE: Deep Contextualized Entity Representations with Entity-aware Self-attention

4.5 Named Entity Disambiguation

Named Entity Disambiguation (NED) is the task of predicting whether a particular entity is mentioned in a sentence. For example, given the sentence "Dronning Margrethe er kendt for sin store arkæologiske interesse." , is the entity "Margrethe II of Denmark" (corresponding to the Wikidata QID `Q102139`) mentioned in the sentence?

NED is the binary version of Named Entity Linking (NEL) which consists in attaching a QID to an entity. Given a sentence and a list of QIDs (potentialy mentioned in the text), NED can be used to find the most probable mentioned entity.

NED is usually used in combination with NER.

4.5.1 Use cases

One entity name can refer to two different persons/locations/organizations depending on the context. Then, in combination with NER, NED can be used for finding entities and their correct reference, and linking them to a description page. The result can be used for differentiating between well-known persons and anonymous individuals in order to, for example, semi-anonymize a text.

4.5.2 Models

XLMR

The XLM-R NED model is based on the pre-trained XLM-Roberta, a transformer-based multilingual masked language model (Conneau et al. 2020), and finetuned on the combination of the DaWikiNED dataset and the training part of the DaNED dataset. The model has been developed as part of a Master student project (ITU) by Hiêu Trong Lâm and Martin Wu under the supervision of Maria Jung Barrett (ITU) and Ophélie Lacroix (DaNLP).

The XLM-R NED model can be loaded with the `load_xlmr_ned_model()` method. (You can also find the model on our HuggingFace page.) Please note that the model take maximum 512 tokens as input at a time. Longer text sequences will be truncated.

```
from danlp.models import load_xlmr_ned_model

xlmr = load_xlmr_ned_model()

sentence = "Karen Blixen vendte tilbage til Danmark, hvor hun boede resten af sit liv␣
↪på Rungstedlund, som hun arvede efter sin mor i 1939"

# to check if the entity "Karen Blixen" correspond to the QID Q182804
# you have to first generate the knowledge graph (KG) context of this QID
# use the get_kg_context_from_wikidata_qid function from danlp.utils to get the KG
# or the DaNED or DaWikiNED dataset to get the corresponding KG string (see doc below)
# (the following example has been truncated -- use the full KG)
kg_context = "udmærkelser modtaget Kritikerprisen udmærkelser modtaget Tagea Brandts␣
↪Rejselegat udmærkelser modtaget Ingenio ..."

label = xlmr.predict(sentence, kg_context)
```

For more details about how to generate a KG context adapted to the model, see the DaNED and DaWikiNED documentation.

4.5.3 Benchmarks

The benchmarks has been performed on the test part of the DaNED dataset. None of the models have been trained on this test part. See F1 scores below (QID is mentioned == label 1 – QID is not mentioned == label 0) :

*Sentences per second is based on a Macbook Pro with Apple M1 chip.

The evaluation script `ned_benchmarks.py` can be found here.

4.5.4 References

- Maria Jung Barrett, Hiêu Trong Lâm, Martin Wu, Ophélie Lacroix, Barbara Plank and Anders Søgaard. Resources and Evaluations for Danish Entity Resolution. In **CRAC 2021**.

4.6 Coreference Resolution

Coreference resolution is the task of finding all mentions (noun phrases) that refer to the same entity (e.g. a person, a location etc, see also the *NER* doc) in a text.

Typically, in a document, entities are first introduced by their name (e.g. `Dronning Margrethe II`) and later refered by pronouns (e.g. `hun`) or expressions/titles (e.g. `Hendes Majestæt`, `Danmarks dronning`, etc). The goal of the coreference resolution task is to find all these references and link them through a common ID.

If you want to read more about coreference resolution and the DaNLP model, we also have a blog post (in Danish).

4.6.1 Use cases

Coreference resolution is an important subtask in NLP. It is used in particular for information extraction (e.g. for building a knowledge graph, see our tutorial) and could help with other NLP tasks such as machine translation (e.g. in order to apply the right gender or number) or text summarization, or in dialog systems.

4.6.2 Models

XLM-R

The XLM-R Coref model is based on the pre-trained XLM-Roberta, a transformer-based multilingual masked language model (Conneau et al. 2020), and finetuned on the Dacoref dataset. The finetuning has been done using the pytorch-based implementation from AllenNLP 1.3.0..

The XLM-R Coref model can be loaded with the `load_xlmr_coref_model()` method. Please note that it can maximum take 512 tokens as input at a time. For longer text sequences split before hand, for example using sentence boundary detection (e.g. by using the spacy model.)

```
from danlp.models import load_xlmr_coref_model

# load the coreference model
coref_model = load_xlmr_coref_model()

# a document is a list of tokenized sentences
doc = [["Lotte", "arbejder", "med", "Mads", "."], ["Hun", "er", "tandlæge", "."]]

# apply coreference resolution to the document and get a list of features (see below)
```

(continues on next page)

(continued from previous page)

```
preds = coref_model.predict(doc)

# apply coreference resolution to the document and get a list of clusters
clusters = coref_model.predict_clusters(doc)
```

The `preds` variable is a dictionary including the following entries :

- `top_spans` : list of indices of all references (spans) in the document
- `antecedent_indices` : list of antecedents indices
- `predicted_antecedents` : list of indices of the antecedent span (from `top_spans`), i.e. previous reference
- `document` : list of tokens' indices for the whole document
- `clusters` : list of clusters (indices of tokens) The most relevant entry to use is the list of clusters. One cluster contains the indices of references (spans) that refer to the same entity. To make it easier, we provide the `predict_clusters` function that returns a list of the clusters with the references and their ids in the document.

4.6.3 Benchmarks

See detailed scoring of the benchmarks in the example folder.

The benchmarks has been performed on the test part of the Dacoref dataset.

*Sentences per second is based on a Macbook Pro with Apple M1 chip.

The evaluation script `coreference_benchmarks.py` can be found here.

4.7 Sentiment Analysis

Sentiment analysis is a broad term for a set of tasks with the purpose of identifying an emotion or opinion in a text.

In this repository we provide an overview of open sentiment analysis models and dataset for Danish.

4.7.1 Use cases

Sentiment analysis is a very useful tool for a company which wants to gain insight about its brand or products as it can be used, for example, for :

- monitoring social media (e.g. quickly identifying posts that generate strong emotions)
- monitoring brand and managing reputation (e.g. analysing the global sentiment of people tweeting about a brand or a product)
- customer support (e.g. identifying happy or angry customers for better adapted responses)
- customers or employees feedback (e.g. analysing the global output of a satisfaction survey)

4.7.2 Models

AFINN

The AFINN tool (Nielsen 2011) uses a lexicon based approach for sentiment analysis. The tool scores texts with an integer where scores <0 are negative, =0 are neutral and >0 are positive.

Sentida

The tool Sentida (Lauridsen et al. 2019) uses a lexicon based approach to sentiment analysis. The tool scores texts with a continuous value. There exist both an R version and an implementation in Python. In these documentations we evaluate the python version from sentida.

BERT Emotion

The emotion classifier has been developed in collaboration with Danmarks Radio, which has granted access to a set of social media data. The data has been manually annotated for 2 tasks:

- to detect whether there is emotion or not in a text (binary classification)
- to classify the text among 8 emotions (`Glæde/Sindsro`, `Tillid/Accept`, `Forventning/Interrese`, `Overasket/Målløs`, `Vrede/Irritation`, `Foragt/Modvilje`, `Sorg/trist`, `Frygt/Bekymret`).

The BERT (Devlin et al. 2019) emotion model(s) have been finetuned on this data using the Transformers library from HuggingFace, and it is based on a pretrained Danish BERT representations by BotXO. The model classifying amongst eight emotions achieves an accuracy on 0.65 and a macro-f1 on 0.64 on the social media test set from DR's Facebook containing 999 examples (we do not have permission to distributing the data).

You can load the models (as one) through the load_bert_emotion_model. Or you can find the models on our HuggingFace page: detection of emotion; classification of emotion.

Below is a small snippet for getting started using the BERT Emotion models. Please note that the BERT model can maximum take 512 tokens as input, however the code allows for overfloating tokens and will therefore not give an error but just a warning.

```
from danlp.models import load_bert_emotion_model
classifier = load_bert_emotion_model()

# using the classifier
classifier.predict('der er et træ i haven')
''''No emotion''''
classifier.predict('jeg ejer en rød bil og det er en god bil')
''''Tillid/Accept''''
classifier.predict('jeg ejer en rød bil men den er gået i stykker')
''''Sorg/trist''''

# Get probabilities and matching classes names
classifier.predict_proba('jeg ejer en rød bil men den er gået i stykker', no_emotion=False)
classifier._classes()
```

BERT Tone

The tone analyzer consists of two BERT (Devlin et al. 2019) classification models:

- a polarity detection model (classifying between `positive`, `neutral` and `negative`);
- a subjective/objective classification model.

Both models have been finetuned on annotated twitter data using the Transformers library from HuggingFace, and it is based on a pretrained Danish BERT representations by BotXO. The data used for training is manually annotated data from Twitter Sentiment (train part) and EuroParl sentiment 2), both datasets can be loaded with the DaNLP package.

You can load the models (as one) through the load_bert_tone_model method of DaNLP. Or you can find the models on our HuggingFace page: polarity detection; subjective/objective classification.

Below is a small snippet for getting started using the BERT Tone models. Please note that the BERT model can maximum take 512 tokens as input, however the code allows for overfloating tokens and will therefore not give an error but just a warning.

```
from danlp.models import load_bert_tone_model
classifier = load_bert_tone_model()

# using the classifier
classifier.predict('Analysen viser, at økonomien bliver forfærdelig dårlig')
'''{'analytic': 'objektive', 'polarity': 'negative'}'''
classifier.predict('Jeg tror alligvel, det bliver godt')
'''{'analytic': 'subjektive', 'polarity': 'positive'}'''

# Get probabilities and matching classes names
classifier.predict_proba('Analysen viser, at økonomien bliver forfærdelig dårlig')
classifier._classes()
```

SpaCy Sentiment

SpaCy sentiment is a text classification model trained using spacy built in command line interface. It uses the CoNLL2017 word vectors (read about it here).

The model is trained using hard distil of the *BERT Tone* (beta) - Meaning, the BERT Tone model is used to make predictions on 50.000 sentences from Twitter and 50.000 sentences from Europarl7. These data is then used to trained a spacy model. Notice the dataset has first been balanced between the classes by oversampling. The model recognizes the classes: 'positiv', 'neutral' and 'negative'.

It is a first version.

Read more about using the Danish spaCy model here.

Below is a small snippet for getting started using the spaCy sentiment model. Currently the danlp packages provide both a spaCy model which do not provide any classes in the textcat module (so it is empty for you to train from scratch), and the sentiment spacy model which have pretrained the classes 'positiv', 'neutral' and 'negative'. Notice it is possible with the spacy command line interface to continue training of the sentiment classes, or add new tags.

```
from danlp.models import load_spacy_model
import operator

# load the model
nlp = load_spacy_model(textcat='sentiment') # if you got an error saying da.vectors␣
↪not found, try setting vectorError=True - it is an temp fix

```

(continues on next page)

(continued from previous page)

```
# use the model for prediction
doc = nlp("Vi er glade for spacy!")
max(doc.cats.items(), key=operator.itemgetter(1))[0]
'''positiv'''
```

Senda

Senda is a python package developed by Ekstra Bladet which helps with fine-tuning transformers for text classification. A model trained on DaNLP's sentiment datasets is available through HuggingFace.

4.7.3 Benchmarks

Benchmark of polarity classification

The benchmark is made by converting the relevant models scores and relevant datasets scores into the there classes 'positive', 'neutral' and 'negative'.

The tools are benchmarked on the following datasets:

- LCC Sentiment contains 499 sentences from the proceedings of the European Parliament annotated with a sentiment score from -5 to 5 by Finn Årup Nielsen.
- Europarl Sentiment contains 184 sentences from news and web pages annotated with sentiment -5 to 5 by Finn Årup Nielsen.
- Twitter Sentiment contains annotations for polarity (positive, neutral, negative) and annotations for analytic (subjective, objective) made by Alexandra Institute. 512 examples of the dataset are defined for evaluation.

A conversion of the scores of the LCC and Europarl Sentiment dataset and the Afinn model is done in the following way: a score of zero to be "neutral", a positive score to be "positive" and a negative score to be "negative".

A conversion of the continuous scores of the Sentida tool into three classes is not given since the 'neutral' class can not be assumed to be only exactly zero but instead we assume it to be an area around zero. We looked for a threshold to see how closed to zero a score should be to be interpreted as neutral. A symmetric threshold is found by optimizing the macro-f1 score on a twitter sentiment corpus (with 1327 examples (the corpus is under construction and will be released later on)) . The threshold is found to be 0.4, which makes our chosen conversion to be: scores over 0.4 to be 'positive', under -0.4 to be 'negative' and scores between to be neutral. However note, the original idea of the tools was not to convert into three class problem.

The scripts for the benchmarks can be found here. There is one for the europarl sentiment and LCC sentiment data and another one for the twitter sentiment. This is due to the fact that downloading the twitter data requires login to a twitter API account. The scores below for the twitter data is reported for all the data, but if tweets are deleted in the mean time on twitter, not all tweets can be downloaded. In the table we consider the accuracy and macro-f1 in brackets, but to get the scores per class we refer to our benchmark script.

*Sentences per second is based on a Macbook Pro with Apple M1 chip.

Benchmark of subjective versus objective classification

The data for benchmark is:

- Twitter Sentiment contains annotations for polarity (positive, neutral, negative) and annotations for analytic (subjective, objective) made by Alexandra Institute. 512 examples of the dataset are defined for evaluation.

The script for the benchmarks can be found here and it provides more detailed scores. Below is accuracy and macro-f1 reported:

4.7.4 Zero-shot Cross-lingual transfer example

An example of utilizing a dataset in another language to be able to make predictions on Danish without seeing Danish training data is shown in this notebook. It is trained on English movie reviews from IMDB, and it uses multilingual embeddings from Artetxe et al. 2019 called LASER(Language-Agnostic SEntence Representations).

4.7.5 References

- Mikel Artetxe, and Holger Schwenk. 2019. Massively Multilingual Sentence Embeddings for Zero-Shot Cross-Lingual Transfer and Beyond.. In **TACL**.
- Erik Velldal, Lilja Øvrelid, Eivind Alexander Bergem, Cathrine Stadsnes, Samia Touileb and Fredrik Jørgensen. 2018. NoReC: The Norwegian Review Corpus. In **LREC**.
- Gustav Aarup Lauridsen, Jacob Aarup Dalsgaard and Lars Kjartan Bacher Svendsen. 2019. SENTIDA: A New Tool for Sentiment Analysis in Danish. In **Sprogvidenskabeligt Studentertidsskrift**.
- Finn Årup Nielsen. 2011. A new ANEW: evaluation of a word list for sentiment analysis in microblogs. In **CEUR Workshop Proceedings**.

4.8 Hate Speech Detection

Hate speech detection is a general term that can include several different tasks. The most common is the identification of offensive language which aims at detecting whether a text is offensive or not (e.g. any type of comment that should be moderated on a social media platform such as containing bad language or attacking an individual). Once a text is detected as offensive, one can detect whether the content is hateful or not.

Here are definitions of the previous concepts:

- offensive : contains profanity or insult
- hateful : targets a group or an individual with the intent to be harmful or to cause social chaos.

4.8.1 Use cases

Hate speech detection is mostly used with the aim of providing support to moderators of social media platform.

4.8.2 Models

BERT Offensive

The offensive language identification model is intended to solve the binary classification problem of identifying whether a text is offensive or not (contains profanity or insult), therefore, given a text, can predict two classes: `OFF` (offensive) or `NOT` (not offensive). Its architecture is based on BERT (Devlin et al. 2019). In particular, it is based on the pretrained Danish BERT trained by BotXO and finetuned on the DKHate data using the Transformers library.

The BERT Offensive model can be loaded with the `load_bert_offensive_model()` method. Please note that it can maximum take 512 tokens as input at a time. The sentences are automatically truncated if longer.

Below is a small snippet for getting started using the BERT Offensive model.

```
from danlp.models import load_bert_offensive_model

# load the offensive language identification model
offensive_model = load_bert_offensive_model()

sentence = "Han ejer ikke respekt for nogen eller noget... han er megaloman og␣
↪psykopat"

# apply the model on the sentence to get the class in which it belongs
pred = offensive_model.predict(sentence)
# or to get its probability of being part of each class
proba = offensive_model.predict_proba(sentence)
```

BERT HateSpeech

The BERT HateSpeech model can detect offensive language and hate speech.

It has been developed in collaboration with Danmarks Radio (DR). It is based on the pre-trained Danish BERT trained by BotXO, and finetuned on facebook data (non publicly available) annotated by DR.

It can predict:

- whether a text is offensive or not : `OFF` (offensive) or `NOT` (not offensive);
- the hate speech category it falls in : `Særlig opmærksomhed`, `Personangreb`, `Sprogbrug`, `Spam & indhold`.

The BERT HateSpeech model can be loaded with the `load_bert_hatespeech_model()` method. Please note that it can maximum take 512 tokens as input at a time. The sentences are automatically truncated if longer.

Below is a small snippet for getting started using the BERT HateSpeech model.

```
from danlp.models import load_bert_hatespeech_model

# load the HateSpeech model
hatespeech_model = load_bert_hatespeech_model()

sentence = "Han ejer ikke respekt for nogen eller noget... han er megaloman og␣
↪psykopat"

# apply the model on the sentence to get the class in which it belongs
pred = hatespeech_model.predict(sentence)
# or to get its probability of being part of each class
proba = hatespeech_model.predict_proba(sentence)
```

ELECTRA Offensive

The ELECTRA Offensive model can detect offensive language.

It has been developed in collaboration with Danmarks Radio (DR). It is based on the pre-trained Danish Ælæctra, and finetuned on facebook data (non publicly available) annotated by DR.

It can predict whether a text is offensive or not : `OFF` (offensive) or `NOT` (not offensive).

The ELECTRA Offensive model can be loaded with the `load_electra_offensive_model()` method. Please note that it can maximum take 512 tokens as input at a time. The sentences are automatically truncated if longer.

Below is a small snippet for getting started using the ELECTRA Offensive model.

```
from danlp.models import load_electra_offensive_model

# load the model
offensive_model = load_electra_offensive_model()

sentence = "Han ejer ikke respekt for nogen eller noget... han er megaloman og␣
↪psykopat"

# apply the model on the sentence to get the class in which it belongs
pred = offensive_model.predict(sentence)
# or to get its probability of being part of each class
proba = offensive_model.predict_proba(sentence)
```

A&ttack (Analyse & Tal)

The A&ttack model detects whether a text is offensive or not. It has been developed by Analyse & Tal and is based on the pretrained Ælectra model. It has been trained on social media data (Facebook, 67,188 tokens). See the github repo for more details and the report of the project.

4.8.3 Benchmarks

See detailed scoring of the benchmarks in the example folder.

The benchmarking has been performed on the test part of the DKHate dataset.

The scores presented here describe the performance (F1) of the models for the task of offensive language identification.

*Sentences per second is based on a Macbook Pro with Apple M1 chip.

The evaluation script `hatespeech_benchmarks.py` can be found here.

4.8.4 References

- Marc Pàmies, Emily Öhman, Kaisla Kajava, Jörg Tiedemann. 2020. LT@Helsinki at SemEval-2020 Task 12: Multilingual or Language-specific BERT?. In **SemEval-2020**

CHAPTER FIVE

FRAMEWORKS

5.1 SpaCy

SpaCy is an industrial friendly open source framework for doing NLP, and you can read more about it on their homesite or gitHub.

This project supports a Danish spaCy model that can easily be loaded with the DaNLP package.

Support for Danish directly in the spaCy framework is realeased under spacy 2.3

Note that the two models are not the same, e.g. the spaCy model in DaNLP performs better on Named Entity Recognition due to more training data. However the extra training data is not open source and can therefore not be included in the spaCy framework itself, as it contravenes the guidelines.

The spaCy model comes with **tokenization**, **dependency parsing**, **part of speech tagging** , **word vectors** and **name entity recognition**.

The model is trained on the Danish Dependency Treebank (DaNe), and with additional data for NER which originates from news articles form a collaboration with InfoMedia.

For comparison to other models and additional information of the tasks, check out the task individual pages for word embeddings, named entity recognition, part of speech tagging and dependency parsing.

The DaNLP github also provides a version of the spaCy model which contains a sentiment classifier, read more about it in the sentiment analysis docs.

5.1.1 Performance of the spaCy model

The following lists the performance scores of the spaCy model provided in DaNLP pakage on the Danish Dependency Treebank (DaNe) test set. The scores and elaborating scores can be found in the file meta.json that is shipped with the model when it is downloaded.

5.1.2 Getting started with the spaCy model

Below is some small snippets to get started using the spaCy model within the DaNLP package. More information about using spaCy can be found on spaCy's own page.

First load the libraries and the model

```
# Import libaries
from danlp.models import load_spacy_model
from spacy.gold import docs_to_json
from spacy import displacy
```

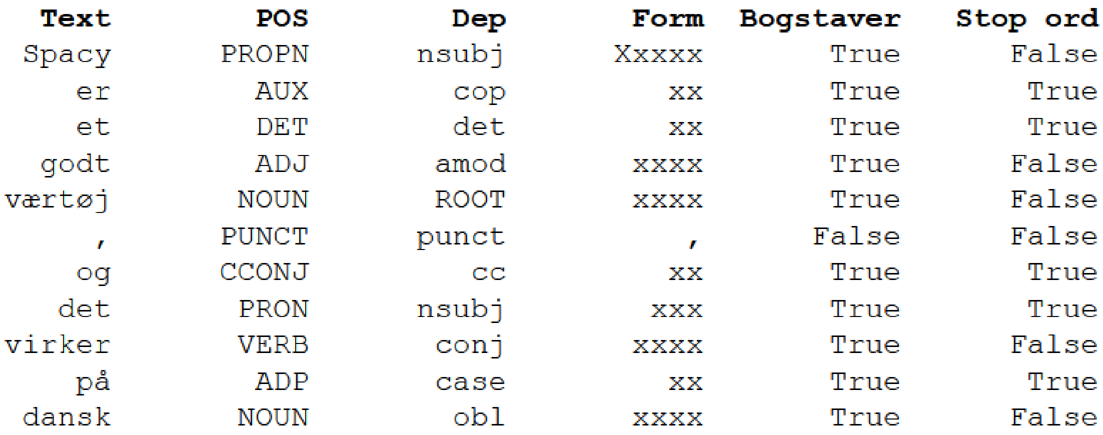
(continues on next page)

(continued from previous page)

```
#Download and load the spaCy model using the DaNLP wrapper fuction
nlp = load_spacy_model()
```

Use the model to determined linguistic features

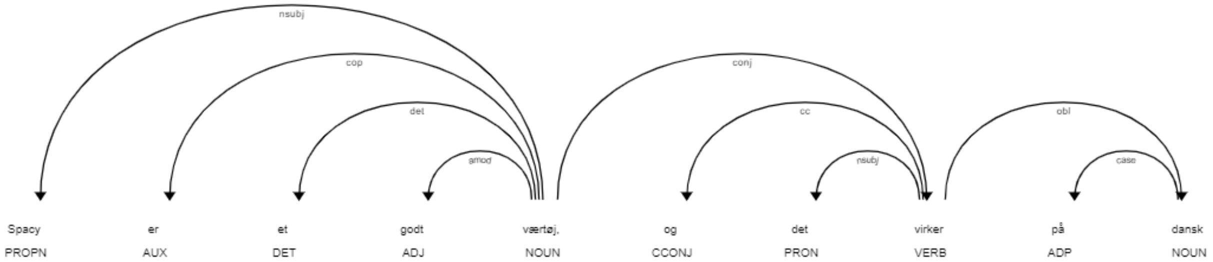
```
# Construct the text to a container "Doc" object
doc = nlp("Spacy er et godt værtøj,og det virker på dansk")

# prepare some pretty printing
features=['Text','POS', 'Dep', 'Form', 'Bogstaver', 'Stop ord']
head_format ="\033[1m{!s:>11}\033[0m" * (len(features) )
row_format ="{!s:>11}" * (len(features) )

print(head_format.format(*features))
# printing for each token in det docs the coresponding linguistic features
for token in doc:
    print(row_format.format(token.text, token.pos_, token.dep_,
            token.shape_, token.is_alpha, token.is_stop,))
```



```
       Text        POS        Dep       Form  Bogstaver   Stop ord
      Spacy      PROPN      nsubj      Xxxxx       True      False
         er        AUX        cop         xx       True       True
         et        DET        det         xx       True       True
       godt        ADJ       amod       xxxx       True      False
     værtøj       NOUN       ROOT       xxxx       True      False
          ,      PUNCT      punct          ,      False      False
         og      CCONJ         cc         xx       True       True
        det       PRON      nsubj        xxx       True       True
     virker       VERB       conj       xxxx       True      False
         på        ADP       case         xx       True       True
      dansk       NOUN        obl       xxxx       True      False
```

Visualizing the dependency tree

```
# the spaCy framework provides a nice visualization tool!
# This is run in a terminal, but if run in jupyter use instead display.render
displacy.serve(doc, style='dep')
```



Here is an example of using Named entity recognitions . You can read more about NER in the specific doc.

```
doc = nlp('Jens Peter Hansen kommer fra Danmark og arbejder hos Alexandra Instituttet
↪')
for tok in doc:
    print("{} {}".format(tok,tok.ent_type_))
```

```
# output
Jens PER
Peter PER
Hansen PER
kommer
fra
Danmark LOC
og
arbejder
hos
Alexandra ORG
Instituttet ORG
```

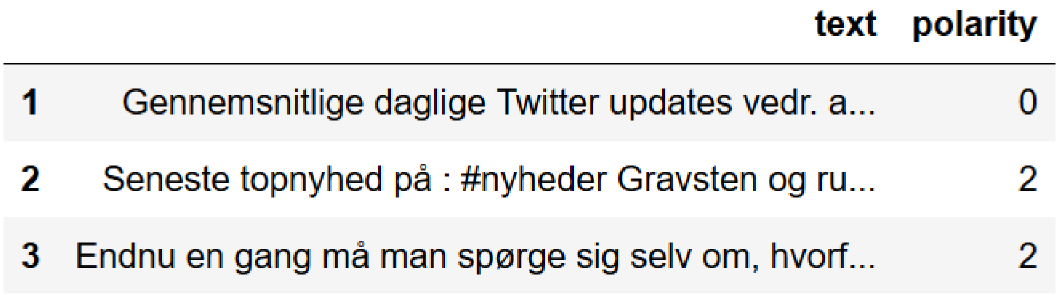
5.1.3 Start training your own text classification model

The spaCy framework provides an easy command line tool for training an existing model, for example by adding a text classifier. This short example shows how to do so using your own annotated data. It is also possible to use any static embedding provided in the DaNLP wrapper.

As an example we will use a small dataset for sentiment classification on twitter. The dataset is under development and will be added in the DaNLP package when ready, and the spacy model will be updated with the classification model as well. A first verison of a spacy model with a sentiment classifier can be load with the danlp wrapper, read more about it in the sentiment analysis docs.

The first thing is to convert the annotated data into a data format readable by spaCy

Imagine you have the data in an e.g csv format and have it split in development and training part. Our twitter data has (in time of creating this snippet) 973 training examples and 400 evaluation examples, with the following labels : 'positive' marked by 0, 'neutral' marked by 1, and 'negative' marked by 2. Loaded with pandas dataFrame it looks like this:



	text	polarity
1	Gennemsnitlige daglige Twitter updates vedr. a...	0
2	Seneste topnyhed på : #nyheder Gravsten og ru...	2
3	Endnu en gang må man spørge sig selv om, hvorf...	2

It needs to be converted into the format expected by spaCy for training the model, which can be done as follows:

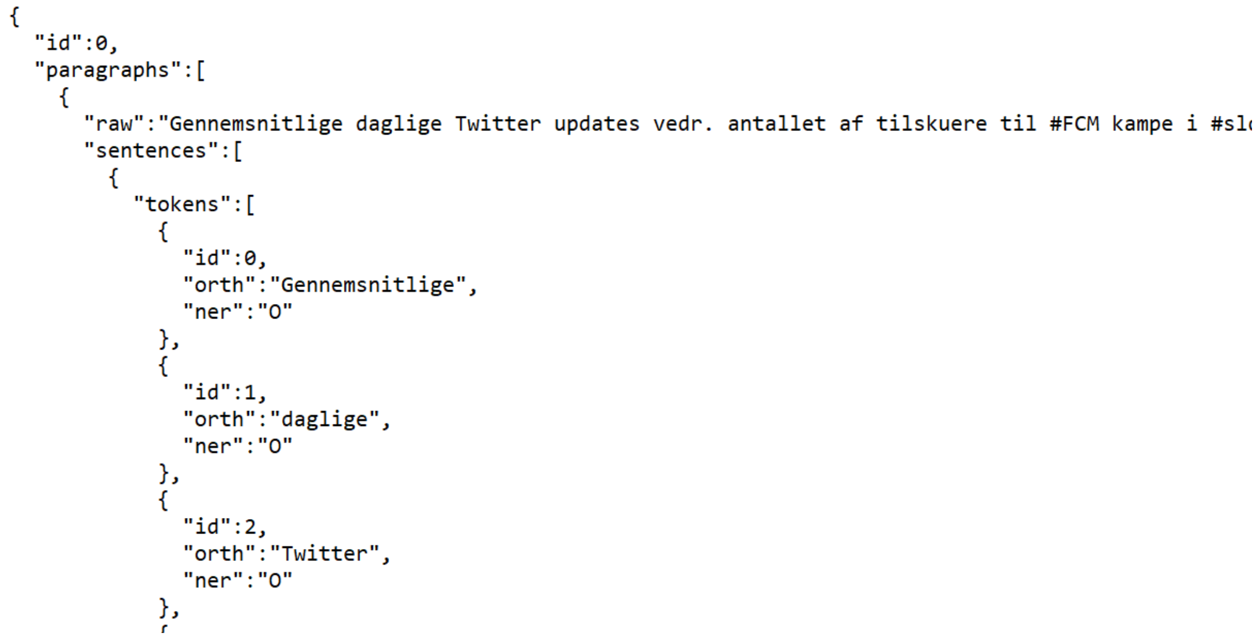
```
#import libaries
import srsly
import pandas as pd
from danlp.models import load_spacy_model
from spacy.gold import docs_to_json

```

(continues on next page)

(continued from previous page)

```
# load the spaCy model
nlp = load_spacy_model()
nlp.disable_pipes(*nlp.pipe_names)
sentencizer = nlp.create_pipe("sentencizer")
nlp.add_pipe(sentencizer, first=True)

# function to read pandas dataFrame and save as json format expected by spaCy
def prepare_data(df, outputfile):
    # choose the name of the columns containg the text and labels
    label='polarity'
    text = 'text'
    def put_cat(x):
        # adapt the name and amount of labels
        return {'positiv': bool(x==0), 'neutral': bool(x==1), 'negativ': bool(x==2)}

    cat = list(df[label].map(put_cat))
    texts, cats= (list(df[text]), cat)

    #Create the container doc object
    docs = []
    for i, doc in enumerate(nlp.pipe(texts)):
        doc.cats = cats[i]
        docs.append(doc)
    # write the data to json file
    srsly.write_json(outputfile,[docs_to_json(docs)])


# prepare both the training data and the evaluation data from pandas dataframe (df_
↪train and df_dev) and choose the name of outputfile
prepare_data(df_train, 'train_sent.json')
prepare_data(df_dev, 'eval_dev.json')
```

The data now looks like this cutted snippet:



```
{
  "id":0,
  "paragraphs":[
    {
      "raw":"Gennemsnitlige daglige Twitter updates vedr. antallet af tilskuere til #FCM kampe i #sl
      "sentences":[
        {
          "tokens":[
            {
              "id":0,
              "orth":"Gennemsnitlige",
              "ner":"O"
            },
            {
              "id":1,
              "orth":"daglige",
              "ner":"O"
            },
            {
              "id":2,
              "orth":"Twitter",
              "ner":"O"
            },
            {
```

Ensure you have the models and embeddings downloaded

The spaCy model and the embeddings must be downloaded. If you have done so it can be done by running the following commands. It will by default be placed in the cache directory of DaNLP, eg. "/home/USERNAME/.danlp/".

```
from danlp.models import load_spacy_model
from danlp.models.embeddings  import load_wv_with_spacy

#download the sapcy model
nlp = load_spacy_model()

# download the static (non subword) embedding of your choice
word_embeddings = load_wv_with_spacy('cc.da.wv')
```

Now train the model through the terminal

Now train the model throught the terminal by pointing to the path of the desired output directory, the converted training data, the converted development data, the base model and the embedding. The specify that the trainings pipe. See more parameter setting here .

```
python -m spacy train da spacy_sent train.json test.json  -b '/home/USERNAME/.danlp/spacy' -v '/home/USERNAME/.danlp/cc.da.wv.spacy' --pipeline textcat
```

Using the trained model

Load the model from the specified output directory.

```
import spacy

#load the trained model
output_dir ='spacy_sent/model-best'
nlp2 = spacy.load(output_dir)

#use the model for prediction
def predict(x):
    doc = nlp2(x)
    return max(doc.cats.items(), key=operator.itemgetter(1))[0]
predict('Vi er glade for spacy!')

#'positiv'
```

5.2 Flair

The flair framework from Zalando is based on the paper Akbik et. al (2018).

Through the DaNLP package, we provide a pre-trained Part-of-Speech tagger, Named Entity recognizer and contextual embeddings using the flair framework. The NER and POS models have been trained on the Danish Dependency Treebank and using fastText word embeddings and flair contextual word embeddings, which are trained on data from Wikipedia and EuroParl corpus.

5.2.1 One sentence at a time

For Part-of-Speech tagging and Named Entity Recognition, it is possible to analyse one sentence at a time using the `Sentence` class of the flair framework.

Please note that the text should be tokenized before hand.

Here is a snippet for using the part-of-speech tagger which can be loaded using the DaNLP `load_flair_pos_model` function.

```
from danlp.models import load_flair_pos_model
from flair.data import Sentence

text = "Morten bor i København tæt på Kongens Nytorv"
sentence = Sentence(text)

tagger = load_flair_pos_model()

tagger.predict(sentence)

for tok in sentence.tokens:
    print(tok.text, tok.get_tag('upos').value)
```

In a similar way, you can load and use the DaNLP Named Entity Recognition model using the `load_flair_ner_model` function.

```
from danlp.models import load_flair_ner_model
from flair.data import Sentence

text = "Morten bor i København tæt på Kongens Nytorv"
sentence = Sentence(text)

tagger = load_flair_ner_model()

tagger.predict(sentence)

for tok in sentence.tokens:
    print(tok.text, tok.get_tag('ner').value)
```

5.2.2 Dataset analysis

If you want to analyze an entire dataset you can either use one of the DaNLP functions to load the DDT or the WikiAnn, or create a list of flair `Sentence`.

DaNLP datasets

You can load the DDT as follow:

```
from danlp.datasets import DDT
ddt = DDT()
# load the DDT
flair_corpus = ddt.load_with_flair()

# you can access the train, test or dev part of the dataset
flair_train = flair_corpus.train
```

(continues on next page)

(continued from previous page)

```
flair_test = flair_corpus.test
flair_dev = flair_corpus.dev

# to get the list of UPOS tags for each sentence
pos_tags = [[tok.get_tag('upos').value for tok in fs] for fs in flair_test]
# to get the list of NER tags for each sentence (BIO format)
ner_tags = [[tok.get_tag('ner').value for tok in fs] for fs in flair_test]
# to get the list of tokens for each sentence
tokens = [[tok.text for tok in fs] for fs in flair_test]

# you can use the loaded datasets
# to parse with the danlp POS or NER models
tagger.predict(flair_test)
```

Or the WikiAnn:

```
from danlp.datasets import WikiAnn
wikiann = WikiAnn()
# load the WikiAnn dataset
flair_corpus = wikiann.load_with_flair()
```

Your dataset

From your own list of sentences (pre-tokenized) you can build a list of flair `Sentence` – to use as previously described with the DaNLP datasets.

Here is an example with the POS model:

```
from danlp.models import load_flair_pos_model
from flair.data import Sentence, Token

# loading the POS flair model
tagger = load_flair_pos_model()

# add your own sentences as seen with two examples.
my_sentences = [['Den', 'danske', 'kolonihavebevægelse', 'er', 'skyld', 'i', ',', 'at
↪', 'vi', 'har', 'fri', 'om', 'lørdagen'],
                ['Møllehøj', 'er', 'det', 'højeste', 'punkt', 'i', 'Danmark']]

flair_sentences = []
for sent in my_sentences:
    flair_sent = Sentence()
    for tok in sent:
        flair_sent.add_token(Token(tok))
    flair_sentences.append(flair_sent)

tagger.predict(flair_sentences)

for sentence in flair_sentences:
    print(" ".join(["{}/{}".format(t.text, t.get_tag('upos').value) for t in sentence.
↪tokens]))
```

5.3 Transformers

In DaNLP, we use language representation models based on the Transformer architecure (Vaswani et al. 2017). We build specialized models for the most common NLP tasks by fine-tuning transformer models, such as BERT and XLM-RoBERTa.

5.3.1 BERT

BERT (Bidirectional Encoder Representations from Transformers) (Devlin et al. 2019) is a deep neural network model. It is one of the most popular transformer-based model used in Natural Language Processing.

The BERT models provided with DaNLP are based on the pre-trained Danish BERT representations by BotXO, and different models have been finetuned on different tasks using the Transformers library from HuggingFace.

Through DaNLP, we provide fine-tuned BERT models for the following tasks:

- Named Entity Recognition
- Emotion detection
- Tone and polarity detection
- Hatespeech detection

The pre-trained Danish BERT from BotXO can also be used for the following tasks without any further finetuning:

- Embeddings of tokens or sentences
- Predict a mask word in a sentence
- Predict if a sentence naturally follows another sentence

Please note that the BERT models can take a maximum of 512 tokens as input at a time. For longer text sequences, you should split the text before hand – for example by using sentence boundary detection (e.g. with the spaCy model).

Language model, embeddings and next sentence prediction

The BERT model (Devlin et al. 2019) is originally pretrained on two tasks. The first, is to predict a masked word in a sentence, and the second is to predict if a sentence follows another sentence. Therefore, the model can without any further finetuning be used for this two tasks.

A pytorch version of the Danish BERT trained by BotXo can therefore be loaded with the DaNLP package and used through the Transformers library.

For **predicting a masked word** in a sentence, you can after downloading the model through DaNLP, use the transformer library directly as described in the following snippet:

```
from transformers import pipeline
from danlp.models import load_bert_base_model
# load the BERT model
model = load_bert_base_model()
# Use the transfomer libary built in function
LM = pipeline("fill-mask", model=model.path_model)
# Use the model as a language model to predict masked words in a sentence
LM(f"Jeg kan godt lide at spise {LM.tokenizer.mask_token}.")
# output is top five words in a list of dicts
"""
[{'sequence': '[CLS] jeg kan godt lide at spise her. [SEP]',
  'score': 0.15520372986793518,
```

(continues on next page)

(continued from previous page)

```
  'token': 215,
  'token_str': 'her'},
 {'sequence': '[CLS] jeg kan godt lide at spise ude. [SEP]',
  'score': 0.05564282834529877,
  'token': 1500,
  'token_str': 'ude'},
 {'sequence': '[CLS] jeg kan godt lide at spise kød. [SEP]',
  'score': 0.052283965051174164,
  'token': 3000,
  'token_str': 'kød'},
 {'sequence': '[CLS] jeg kan godt lide at spise morgenmad. [SEP]',
  'score': 0.051760803908109665,
  'token': 4538,
  'token_str': 'morgenmad'},
 {'sequence': '[CLS] jeg kan godt lide at spise der. [SEP]',
  'score': 0.049477532505989075,
  'token': 59,
  'token_str': 'der'}]
"""
```

The DaNLP package also provides some wrapper code for **next sentence prediction**:

```
from danlp.models import load_bert_nextsent_model
model = load_bert_nextsent_model()

# the sentence is from a wikipedia article https://da.wikipedia.org/wiki/Uranus_
↪(planet)
# Sentence B1 follows after sentence A, where sentence B2 is taken futher down in the␣
↪article
sent_A= "Uranus er den syvende planet fra Solen i Solsystemet og var den første␣
↪planet der blev opdaget i historisk tid."
sent_B1 =" William Herschel opdagede d. 13. marts 1781 en tåget klat, som han først␣
↪troede var en fjern komet."
sent_B2= "Yderligere er magnetfeltets akse 59° forskudt for rotationsaksen og skærer␣
↪ikke centrum."

# model returns the probability of sentence B follows rigth after sentence A
model.predict_if_next_sent(sent_A, sent_B1)
"""0.9895"""
model.predict_if_next_sent(sent_A, sent_B2)
"""0.0001"""
```

The wrapper function for **embeddings** of tokens or sentences can be read about in the docs for embeddings.

Named Entity Recognition

The BERT NER model has been finetuned on the DaNE dataset (Hvingelby et al. 2020). The tagger recognizes the following tags:

- `PER`: person
- `ORG`: organization
- `LOC`: location

Read more about it in the NER docs.

Emotion detection

The emotion classifier is developed in a collaboration with Danmarks Radio, which has granted access to a set of social media data. The data has been manually annotated first to distinguish between a binary problem of emotion or no emotion, and afterwards tagged with 8 emotions. The BERT emotion model is finetuned on this data.

The model can detect the eight following emotions:

- `Glæde/Sindsro`
- `Tillid/Accept`
- `Forventning/Interrese`
- `Overasket/Målløs`
- `Vrede/Irritation`
- `Foragt/Modvilje`
- `Sorg/trist`
- `Frygt/Bekymret`

The model achieves an accuracy of 0.65 and a macro-f1 of 0.64 on the social media test set from DR's Facebook containing 999 examples. We do not have permission to distributing the data.

Read more about it in the sentiment docs.

Tone and polarity detection

The tone analyzer consists of two BERT classification models.

The models are finetuned on manually annotated Twitter data from Twitter Sentiment (train part) and EuroParl sentiment 2). Both datasets can be loaded with the DaNLP package.

The first model detects the polarity of a sentence, i.e. whether it is perceived as `positive`, `neutral` or `negative`. The second model detects the tone of a sentence, between `subjective` and `objective`.

Read more about it in the sentiment docs.

Hatespeech detection

The offensive language identification model predicts whether a text is offensive (`OFF`) or not (`NOT`). The model is fine-tuned on the DKHate dataset.

Read more about it in the hatespeech docs.

5.3.2 XLM-RoBERTa

XLM-Roberta is a transformer-based multilingual masked language model (Conneau et al. 2020). It has shown better performance than mBERT (multilingual BERT) on a range of NLP tasks.

Through DaNLP, we provide fine-tuned XLM-R models for the tasks of named entity disambiguation and coreference resolution.

Named Entity Disambiguation

Named entity disambiguation is a binary classification task which is used to predict whether a specific instance of an entity is mentioned in a text. It is used for named entity linking.

Read more about it in the NED docs.

Coreference resolution

Coreference resolution is the task of finding all expressions that refer to the same entity in a text.

Read more about it in the coreference docs.

CHAPTER
SIX

DATASETS

This section keeps a list of Danish NLP datasets publicly available.

It is also recommend to check out Finn Årup Nielsen's dasem github which also provides script for loading different Danish corpus.

6.1 Danish Dependency Treebank (DaNE)

The Danish UD treebank (Johannsen et al., 2015, UD-DDT) is a conversion of the Danish Dependency Treebank (Buch-Kromann et al. 2003) based on texts from Parole (Britt, 1998). UD-DDT has annotations for dependency parsing and part-of-speech (POS) tagging. The dataset was annotated with Named Entities for **PER**, **ORG** and **LOC** by the Alexandra Institute in the DaNE dataset (Hvingelby et al. 2020). To read more about how the dataset was annotated with POS and DEP tags we refer to the Universal Dependencies page. The dataset can be loaded with the DaNLP package:

```
from danlp.datasets import DDT
ddt = DDT()

spacy_corpus = ddt.load_with_spacy()
flair_corpus = ddt.load_with_flair()
conllu_format = ddt.load_as_conllu()
```

The dataset can also be downloaded directly in CoNLL-U format.

Download DDT

6.2 DaCoref

This Danish coreference annotation contains parts of the Copenhagen Dependency Treebank (Kromann and Lynge, 2004). It was originally annotated as part of the Copenhagen Dependency Treebank (CDT) project but never finished. This resource extends the annotation by using different mapping techniques and by augmenting with Qcodes from Wiktionary. This work is conducted by Maria Jung Barrett. Read more about it in the dedicated DaCoref docs.

The dataset can be loaded with the DaNLP package:

```
from danlp.datasets import Dacoref
dacoref = Dacoref()
# The corpus can be loaded with or without splitting into train, dev and test in a␣
↪list in that order
corpus = dacoref.load_as_conllu(predefined_splits=True)
```

The dataset can also be downloaded directly:

Download DaCoref

6.3 DKHate

The DKHate dataset contains user-generated comments from social media platforms (Facebook and Reddit) annotated for various types and target of offensive language. The original corpus used for the OffensEval 2020 shared task can be found here.

Note that only labels for the sub-task A (Offensive language identification), i.e. `NOT` (Not Offensive) / `OFF` (Offensive), are available.

The dataset can be loaded with the DaNLP package:

```
from danlp.datasets import DKHate
dkhate = DKHate()
test, train = dkhate.load_with_pandas()
```

The dataset can also be downloaded directly:

Download dkhate

6.4 WikiANN

The WikiANN dataset (Pan et al. 2017) is a dataset with NER annotations for **PER**, **ORG** and **LOC**. It has been constructed using the linked entities in Wikipedia pages for 282 different languages including Danish. The dataset can be loaded with the DaNLP package:

```
from danlp.datasets import WikiAnn
wikiann = WikiAnn()

spacy_corpus = wikiann.load_with_spacy()
flair_corpus = wikiann.load_with_flair()
```

6.5 WordSim-353

The WordSim-353 dataset (Finkelstein et al. 2002) contains word pairs annotated with a similarity score (1-10). It is common to use it to do intrinsic evaluations on word embeddings to test for syntactic or semantic relationships between words. The dataset has been translated to Danish by Finn Årup Nielsen. Here is how you can load the dataset:

```
from danlp.datasets import WordSim353Da

ws353 = WordSim353Da()
ws353.load_with_pandas()
```

6.6 Danish Similarity Dataset

The Danish Similarity Dataset consists of 99 word pairs annotated by 38 annotators with a similarity score (1-6). It is constructed with frequently used Danish words. Here is how you can load the dataset:

```
from danlp.datasets import DSD

dsd = DSD()
dsd.load_with_pandas()
```

6.7 Twitter Sentiment

The Twitter sentiment is a small manually annotated dataset by the Alexandra Institute. It contains tags in two sentiment dimension: analytic: [‘subjective’ , ‘objective’] and polarity: [‘positive’, ‘neutral’, ‘negative’]. It is split in train and test part. Due to Twitters privacy policy, it is only allowed to display the “tweet ID” and not the actually text. This allows people to delete their tweets. Therefore, to download the actual tweet text one need a Twitter development account and to generate the sets of login keys, read how to get started here. Then the dataset can be loaded with the DaNLP package by setting the following environment variable for the keys:

```
TWITTER_CONSUMER_KEY, TWITTER_CONSUMER_SECRET, TWITTER_ACCESS_TOKEN, TWITTER_ACCESS_
↪SECRET
```

```
from danlp.datasets import TwitterSent
twitSent = TwitterSent()

df_test, df_train = twitSent.load_with_pandas()
```

The dataset can also be downloaded directly with the labels and tweet id:

Download TwitterSent

6.8 Europarl Sentiment1

The Europarl Sentiment1 dataset contains sentences from the Europarl corpus which has been annotated manually by Finn Årup Nielsen. Each sentence has been annotated the polarity of the sentiment as an polarity score from -5 to 5. The score can be converted to positive (>0), neutral (=0) and negative (<0). The dataset can be loaded with the DaNLP package:

```
from danlp.datasets import EuroparlSentiment1
eurosent = EuroparlSentiment1()

df = eurosent.load_with_pandas()
```

6.9 Europarl Sentiment2

The dataset consist of 957 manually annotation by Alexandra institute on sentences from Eruroparl. It contains tags in two sentiment dimension: analytic: [‘subjective’ , ‘objective’] and polarity: [‘positive’, ‘neutral’, ‘negative’]. The dataset can be loaded with the DaNLP package:

```
from danlp.datasets import EuroparlSentiment2
eurosent = EuroparlSentiment2()

df = eurosent.load_with_pandas()
```

6.10 LCC Sentiment

The LCC Sentiment dataset contains sentences from Leipzig Copora Collection (Quasthoff et al. 2006) which has been manually annotated by Finn Årup Nielsen.

Each sentence has been annotated the polarity of the sentiment as an polarity score from -5 to 5. The score can be converted to positive (>0), neutral (=0) and negative (<0). The dataset can be loaded with the DaNLP package:

```
from danlp.datasets import LccSentiment
lccsent = LccSentiment()

df = lccsent.load_with_pandas()
```

6.11 AngryTweets

The AngryTweets sentiment dataset is a crowd-sourced dataset annotated with polarity tags: [‘positive’, ‘neutral’, ‘negative’]. The dataset contains 4122 tweets including 1727 that were annotated by one trained annotator. More annotations have been collected through the AngryTweets game resulting in 1266 tweets with double annotations. If you want to read more about the game, see the Medium blog post or the DataTech article. In the same way as the Twitter Sentiment dataset, only the ID of the tweets are made available (see *Twtitter Sentiment* for more details).

Here is how to load the dataset with the DaNLP package:

```
from danlp.datasets import AngryTweets
angrytweets = AngryTweets()

df = angrytweets.load_with_pandas()
```

The dataset (labels and tweet ids) can also be downloaded directly:

Download AngryTweets

6.12 DanNet

DanNet is a lexical database such as Wordnet. “Center for sprogteknologi” at The University of Copenhagen is behind it and more details about it can be found in the paper (Pedersen et al 2009).

DanNet depicts the relations between words in Danish (mostly nouns, verbs and adjectives). The main relation among words in WordNet is synonymy.

The dataset consists of 4 databases:

```
* words
* word senses
* relations
* synsets
```

DanNet uses the concept of `synset` to link words together. All the words in the database are part of one or multiple synsets. A synset is a set of synonyms (words which have the same meanings).

For downloading DanNet through DaNLP, you can do:

```
from danlp.datasets import DanNet

dannet = DanNet()

# you can load the databases if you want to look into the databases by yourself
words, wordsenses, relations, synsets = dannet.load_with_pandas()
```

We also provide helper functions to search for synonyms, hyperonyms, hyponyms and domains through the databases. Once you have downloaded the DanNet wrapper, you can use the following features:

```

word = "myre"
# synonyms
dannet.synonyms(word)
""" ['tissemyre'] """
# hypernyms
dannet.hypernyms(word)
""" ['årevingede insekter'] """
# hyponyms
dannet.hyponyms(word)
""" ['hærmyre', 'skovmyre', 'pissemyre', 'tissemyre'] """
# domains
dannet.domains(word)
""" ['zoologi'] """
# meanings
dannet.meanings(word)
""" ['ca. 1 cm langt, årevinget insekt med en kraftig in ... (Brug: "Myrer på␣
↪terrassen, og andre steder udendørs, kan hurtigt blive meget generende")'] """


# to help you dive into the databases
# we also provide the following functions:

# part-of-speech (returns a list comprised in 'Noun', 'Verb' or 'Adjective')
dannet.pos(word)
# wordnet relations (EUROWORDNET or WORDNETOWL)
dannet.wordnet_relations(word, eurowordnet=True)
# word ids
```

(continues on next page)

(continued from previous page)

```
dannet._word_ids(word)
# synset ids
dannet._synset_ids(word)
# word from id
dannet._word_from_id(11034863)
# synset from id
dannet._synset_from_id(3514)
```

6.13 DaUnimorph

The UniMorph project provides lists of word forms (for many languages) associated with their lemmas and morphological features following a universal schema which have been extracted from Wikipedia.

The Danish UniMorph is a (non-exhaustive) list of nouns and verbs. The following morphological features are provided :

- the part-of-speech, i.e. noun `N` or verb `V`
- the voice (for verbs), i.e. active `ACT` or passive `PASS`
- the mood (for verbs), i.e. infinitive `NFIN`, indicative `IND`, imperative `IMP`
- the tense (for verbs), i.e. past `PST` or present `PRS`
- the form (for nouns), i.e. indefinite `INDF` or definite `DEF`
- the case (for nouns), i.e. nominative `NOM` or genitive `GEN`
- the number (for nouns), i.e. plural `PL` or singular `SG`

For downloading DanNet through DaNLP, you can do:

```
from danlp.datasets import DaUnimorph

unimorph = DaUnimorph()

# you can load the database if you want to look into it by yourself
database = unimorph.load_with_pandas()
```

Once you have downloaded the DaUnimorph wrapper, you can also use the following features:

```

word = "trolde"
# inflections (the different forms of a word)
unimorph.get_inflections(word, pos='V', with_features=False)
""" ['troldedes', 'troldede', 'trolder', 'troldes', 'trolde', 'trold'] """
# lemmas (the root form of a word)
unimorph.get_lemmas(word, pos='N', with_features=True)
""" [{'lemma': 'trold', 'form': 'trolde', 'feats': 'N;INDF;NOM;PL', 'pos': 'N'}] """
```

6.14 DaNED

The DaNED dataset is derived from the *DaCoref* (including only sentences that have at least one QID annotation) and annotated for named entity disambiguation. The dataset has been developed for DaNLP, through a Master student project, by Trong Hiêu Lâm and Martin Wu under the supervision of Maria Jung Barrett (ITU) and Ophélie Lacroix (DaNLP – Alexandra Institute). Each entry in the dataset is a tuple (sentence, QID) associated with a label (0 or 1) which indicate whether the entity attached to the QID is mentioned in the sentence or not. The same sentence occurs several times but only one of them as a label "1" because only one of the QIDs is correct.

In addition, we provide – through the dataset – for each QID, its corresponding knowledge graqh (KG) context extracted from Wikidata. For more details about the annotation process and extraction of KG context see the paper.

The dataset can be loaded with the DaNLP package:

```
from danlp.datasets import DaNED
daned = DaNED()
train, dev, test = daned.load_with_pandas()
```

To get the KG context (Wikidata properties and description) of a QID (from the DaNED database), you can use:

```
qid = "Q303"
# Get Elvis Presley's Wikidata properties and description
properties, description = get_kg_context_from_qid(qid)
```

If the QID does not exist in the database, you can allow the search through Wikidata (online):

```
qid = "Q36620"
# Get Tycho Brahe's Wikidata properties and description
properties, description = get_kg_context_from_qid(qid, allow_online_search=True)
```

The dataset can also be downloaded directly:

Download DaNED

6.15 DaWikiNED

The DaWikiNED is automatically constructed and intended to be used as a training set augmentation with the *DaNED* dataset. The dataset has been developed for DaNLP through a student project by Trong Hiêu Lâm and Martin Wu under the supervision of Maria Jung Barrett (ITU) and Ophélie Lacroix (DaNLP – Alexandra Institute). Sentences come from the Danish Wikipedia. Knowledge graph contexts come from Wikidata (see *DaNED*).

The dataset can be loaded with the DaNLP package:

```
from danlp.datasets import DaWikiNED
dawikined = DaWikiNED()
train = dawikined.load_with_pandas()
```

To get the KG context (Wikidata properties and description) of a QID (from the DaWikiNED database), you can use:

```
qid = "Q1748"
# Get Copenhagen's Wikidata properties and description
properties, description = get_kg_context_from_qid(qid, dictionary=True)
```

If the QID does not exist in the database, you can allow the search through Wikidata (online):

```
qid = "Q36620"
# Get Tycho Brahe's Wikidata properties and description
properties, description = get_kg_context_from_qid(qid, allow_online_search=True)
```

The dataset can also be downloaded directly:

Download DaWikiNED

6.16 DDisco

The DDisco dataset has been developed for DaNLP by Linea Flansmose, through a Master student project, under the supervision of Ophélie Lacroix (DaNLP – Alexandra Institute). Each entry in the dataset is annotated with a discourse coherence label (rating from 1 to 3):

- 1: low coherence (difficult to understand, unorganized, contained unnecessary details and can not be summarized briefly and easily)
- 2: medium coherence
- 3: high coherence (easy to understand, well organized, only contain details that support the main point and can be summarized briefly and easily).

Grammatical and typing errors are ignored (i.e. they do not affect the coherency score) and the coherence of a text is considered within its own domain.

The dataset can be loaded with the DaNLP package:

```
from danlp.datasets import DDisco
ddisco = DDisco()
train, test = ddisco.load_with_pandas()
```

The dataset can also be downloaded directly:

Download DDisco

6.17 References

- Johannsen, Anders, Martínez Alonso, Héctor and Plank, Barbara. Universal Dependencies for Danish. TLT14, 2015.
- Keson, Britt (1998). Documentation of The Danish Morpho-syntactically Tagged PAROLE Corpus. Technical report, DSL
- Matthias T. Buch-Kromann, Line Mikkelsen, and Stine Kern Lynge. 2003. Danish dependency treebank. In **TLT**.
- Rasmus Hvingelby, Amalie B. Pauli, Maria Barrett, Christina Rosted, Lasse M. Lidegaard and Anders Søgaard. 2020. DaNE: A Named Entity Resource for Danish. In **LREC**.
- Pedersen, Bolette S. Sanni Nimb, Jørg Asmussen, Nicolai H. Sørensen, Lars Trap-Jensen og Henrik Lorentzen (2009). DanNet – the challenge of compiling a WordNet for Danish by reusing a monolingual dictionary. *Lang Resources & Evaluation* 43:269–299.
- Xiaoman Pan, Boliang Zhang, Jonathan May, Joel Nothman, Kevin Knight and Heng Ji. 2017. Cross-lingual Name Tagging and Linking for 282 Languages. In **ACL**.

- Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. 2002. Placing Search in Context: The Concept Revisited. In **ACM TOIS**.
- Uwe Quasthoff, Matthias Richter and Christian Biemann. 2006. Corpus Portal for Search in Monolingual Corpora. In **LREC**.
- M.T. Kromann and S.K. Lynge. Danish Dependency Treebank v. 1.0. Department of Computational Linguistics, Copenhagen Business School., 2004.
- Sigurbergsson, Gudbjartur Ingi and Derczynski, Leon. Offensive Language and Hate Speech Detection for {D}anish. in **LREC 2020**

CHAPTER

SEVEN

MODELS

7.1 Embeddings

This module provides you with functions for loading pretrained Danish word embeddings through several NLP frameworks:

- flair
- spaCy
- Gensim

Available word embeddings:

- wiki.da.wv
- cc.da.wv
- conll17.da.wv
- news.da.wv
- sketchengine.da.wv

Available subword embeddings:

- wiki.da.swv
- cc.da.swv
- sketchengine.da.swv

`danlp.models.embeddings.`**`AVAILABLE_EMBEDDINGS = ['wiki.da.wv', 'cc.da.wv', 'conll17.da.wv',`**

`danlp.models.embeddings.`**`AVAILABLE_SUBWORD_EMBEDDINGS = ['wiki.da.swv', 'cc.da.swv', 'sketcl`**

`danlp.models.embeddings.`**`assert_wv_dimensions`**(*wv: gensim.models.keyedvectors.Word2VecKeyedVectors*, *pretrained_embedding: str*)

This function will check the dimensions of some word embeddings wv, and check them against the data stored in WORD_EMBEDDINGS.

Parameters

- **`wv`** (*`gensim.models.KeyedVectors`*) – word embeddings
- **`pretrained_embedding`** (*`str`*) – the name of the pretrained embeddings

danlp.models.embeddings.**load_context_embeddings_with_flair**(*direction='bi'*, *word_embeddings=None*, *cache_dir='/home/docs/.danlp'*, *verbose=False*)

Loads contextutal (dynamic) word embeddings with flair.

Parameters

- **direction** (*str*) – bidirectional 'bi', forward 'fwd' or backward 'bwd'
- **word_embedding** –
- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

danlp.models.embeddings.**load_keras_embedding_layer**(*pretrained_embedding: str*, *cache_dir='/home/docs/.danlp'*, *verbose=False*, ***kwargs*)

Loads a Keras Embedding layer.

Parameters

- **pretrained_embedding** (*str*) –
- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity
- **kwargs** – used to forward arguments to the Keras Embedding layer

Returns a Keras Embedding layer and index to word dictionary

danlp.models.embeddings.**load_pytorch_embedding_layer**(*pretrained_embedding: str*, *cache_dir='/home/docs/.danlp'*, *verbose=False*)

Loads a pytorch embbeding layer.

Parameters

- **pretrained_embedding** (*str*) –
- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

Returns a pytorch Embedding module and a list id2word

danlp.models.embeddings.**load_wv_with_gensim**(*pretrained_embedding: str*, *cache_dir='/home/docs/.danlp'*, *verbose: bool = False*)

Loads word embeddings with Gensim.

Parameters

- **pretrained_embedding** (*str*) –
- **cache_dir** – the directory for storing cached data
- **verbose** (*bool*) – *True* to increase verbosity

Returns KeyedVectors or FastTextKeyedVectors

danlp.models.embeddings.**load_wv_with_spacy**(*pretrained_embedding: str*, *cache_dir: str = '/home/docs/.danlp'*, *verbose=False*)

Loads a spaCy model with pretrained embeddings.

Parameters

- **pretrained_embedding** (*str*) –
- **cache_dir** (*str*) – the directory for storing cached data
- **verbose** (*bool*) – *True* to increase verbosity

Returns spaCy model

7.2 BERT models

class danlp.models.bert_models.**BertBase**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Bases: object

BERT language model used for embedding of tokens or sentence. The Model is trained by BotXO: https://github.com/botxo/nordic_bert The Bert model is transformed into pytorch version

Credit for code example: https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

embed_text(*text*)

Calculate the embeddings for each token in a sentence ant the embedding for the sentence based on a BERT language model. The embedding for a token is chosen to be the concatenated last four layers, and the sentence embeddings to be the mean of the second to last layer of all tokens in the sentence The BERT tokenizer splits in subword for UNK word. The tokenized sentence is therefore returned as well. The embeddings for the special tokens are not returned.

Parameters **sentence** (*str*) – raw text

Returns three lists: token_embeddings (dim: tokens x 3072), sentence_embedding (1x738), tokenized_text

Return type list, list, list

class danlp.models.bert_models.**BertEmotion**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Bases: object

BERT Emotion model.

For classifying whether there is emotion in the text, and recognizing amongst eight emotions.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

predict(*sentence: str*, *no_emotion=False*)

Predicts emotion among:

- 0: *Glæde/Sindsro*
- 1: *Tillid/Accept*
- 2: *Forventning/Interrese*
- 3: *Overasket/Målløs*
- 4: *Vrede/Irritation*

- 5: *Foragt/Modvilje*
- 6: *Sorg/trist*
- 7: *Frygt/Bekymret*

Parameters

- **sentence** (*str*) – raw text
- **no_emotion** (*bool*) – whether there is emotion or not in the text

Returns index of the emotion

Return type int

predict_if_emotion (*sentence*)
Predicts whether there is emotion in the text.

Parameters **sentence** (*str*) – raw sentence

Returns 0 if no emotion else 1

Return type int

predict_proba (*sentence: str*, *emotions=True*, *no_emotion=True*)
Predicts the probabilities of emotions.

Parameters

- **sentence** (*str*) – raw text
- **emotions** (*bool*) – whether to return the probability of the emotion
- **no_emotion** (*bool*) – whether to return the probability of the sentence being emotional

Returns a list of probabilities

Return type List

class danlp.models.bert_models.**BertHateSpeech** (*cache_dir='/home/docs/.danlp'*, *verbose=False*)
Bases: object

BERT HateSpeech Model.

For detecting whether a comment is offensive or not and, if offensive, predicting what type of hate speech it is. The model is meant to be used for helping moderating online comments (thus, including the detection and categorization of spams). Following are the categories that can be predicted by the model: * Særlig opmærksomhed * Personangreb * Sprogbrug * Spam & indhold

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

predict (*sentence: str*, *offensive: bool = True*, *hatespeech: bool = True*)
Predict whether a sentence is offensive or not and/or the class of the sentence [Særlig opmærksomhed, Personangreb, Sprogbrug, Spam & indhold]

Parameters

- **sentence** (*str*) – raw text
- **offensive** (*bool*) – if *True* returns whether the sentence is offensive or not
- **hatespeech** (*bool*) – if *True* returns the type of hate speech the sentence belongs to

Returns a dictionary for offensive language and hate speech detection results

Return type Dict

class `danlp.models.bert_models.`**`BertNer`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Bases: `object`

BERT NER model

Parameters

- **`cache_dir`** (*`str`*) – the directory for storing cached models
- **`verbose`** (*`bool`*) – *True* to increase verbosity

`predict`(*text: Union[str, List[str]]*, *IOBformat=True*)

Predict NER labels from raw text or tokenized text. If the text is a raw string this method will return the string tokenized with BERTs subword tokens.

Parameters

- **`text`** – can either be a raw text or a list of tokens
- **`IOBformat`** – can either be TRUE or FALSE, but can only be False if text input is a list of tokens. Specify if output should be in IOB format or a dictionary

Returns the tokenized text and the predicted labels in IOB format, or a dictionary with the tags and position

Example "*varme vafler*" becomes ["varme", "va", "##fler"]

class `danlp.models.bert_models.`**`BertNextSent`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Bases: `object`

BERT language model is trained for next sentence predictions. The Model is trained by BotXO: https://github.com/botxo/nordic_bert The Bert model is transformed into pytorch version

Credit for code example: https://stackoverflow.com/questions/55111360/using-bert-for-next-sentence-prediction

Parameters

- **`cache_dir`** (*`str`*) – the directory for storing cached models
- **`verbose`** (*`bool`*) – *True* to increase verbosity

`predict_if_next_sent`(*sent_A: str*, *sent_B: str*)

Calculate the probability that sentence B follows sentence A.

Credit for code example: https://stackoverflow.com/questions/55111360/using-bert-for-next-sentence-prediction

Parameters

- **`sent_A`** (*`str`*) – sentence A
- **`sent_B`** (*`str`*) – sentence B

Returns the probability of sentence B following sentence A

Return type float

class `danlp.models.bert_models.`**`BertOffensive`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Bases: `object`

BERT offensive language identification model.

For predicting whether a text is offensive or not.

Parameters

- **`cache_dir`** (`str`) – the directory for storing cached models
- **`verbose`** (`bool`) – *True* to increase verbosity

`predict`(*sentence: str*)

Predict whether a text is offensive or not.

Parameters **`sentence`** (`str`) – raw text

Returns a class – *OFF* (offensive) or *NOT* (not offensive)

Return type str

`predict_proba`(*sentence: str*)

For a given sentence, return its probabilities of belonging to each class, i.e. *OFF* or *NOT*

class `danlp.models.bert_models.`**`BertTone`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Bases: `object`

BERT Tone model.

For classifying both the tone [subjective, objective] and the polarity [positive, neutral, negativ] of sentences.

Parameters

- **`cache_dir`** (`str`) – the directory for storing cached models
- **`verbose`** (`bool`) – *True* to increase verbosity

`predict`(*sentence: str*, *polarity: bool = True*, *analytic: bool = True*)

Predict the polarity [positive, neutral, negativ] and/or the tone [subjective, objective] of the sentence.

Parameters

- **`sentence`** (`str`) – raw text
- **`polarity`** (`bool`) – returns the polarity if *True*
- **`analytic`** (`bool`) – returns the tone if *True*

Returns a dictionary for polarity and tone results

Return type Dict

`danlp.models.bert_models.`**`load_bert_base_model`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Load BERT language model and use for embedding of tokens or sentence. The Model is trained by BotXO: https://github.com/botxo/nordic_bert

Parameters

- **`cache_dir`** (`str`) – the directory for storing cached models
- **`verbose`** (`bool`) – *True* to increase verbosity

Returns BERT model

`danlp.models.bert_models.`**`load_bert_emotion_model`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Loads a BERT Emotion model.

Parameters

- **`cache_dir`** (`str`) – the directory for storing cached models

- **verbose** (*bool*) – *True* to increase verbosity

Returns a BERT Emotion model

`danlp.models.bert_models.`**`load_bert_hatespeech_model`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Loads a BERT HateSpeech model.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

Returns a BERT HateSpeech model

`danlp.models.bert_models.`**`load_bert_ner_model`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Loads a BERT NER model.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

Returns a BERT NER model

`danlp.models.bert_models.`**`load_bert_nextsent_model`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Load BERT language model used for next sentence predictions. The Model is trained by BotXO: https://github.com/botxo/nordic_bert

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

Returns BERT NextSent model

`danlp.models.bert_models.`**`load_bert_offensive_model`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Loads a BERT offensive language identification model.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

Returns a BERT offensive language identification model

`danlp.models.bert_models.`**`load_bert_tone_model`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Loads a BERT Tone model.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

Returns a BERT Tone model

7.3 ELECTRA models

class `danlp.models.electra_models.`**`ElectraOffensive`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Bases: `object`

Electra Offensive Model.

For detecting whether a comment is offensive or not.

Parameters

- **`cache_dir`** (*`str`*) – the directory for storing cached models
- **`verbose`** (*`bool`*) – *True* to increase verbosity

`predict`(*sentence: str*)

Predict whether a sentence is offensive or not

Parameters **`sentence`** (*`str`*) – raw text

Returns a class representing whether the sentence is offensive or not (*OFF*/*NOT*)

Return type str

`predict_proba`(*sentence: str*)

For a given sentence, return its probabilities of belonging to each class, i.e. *OFF* or *NOT*

`danlp.models.electra_models.`**`load_electra_offensive_model`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Loads an Electra Offensive model.

Parameters

- **`cache_dir`** (*`str`*) – the directory for storing cached models
- **`verbose`** (*`bool`*) – *True* to increase verbosity

Returns an Electra Offensive model

7.4 flair models

`danlp.models.flair_models.`**`load_flair_ner_model`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Loads a flair model for NER.

Parameters

- **`cache_dir`** (*`str`*) – the directory for storing cached models
- **`verbose`** (*`bool`*) – *True* to increase verbosity

Returns an NER flair model

`danlp.models.flair_models.`**`load_flair_pos_model`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Loads a flair model for Part-of-Speech tagging.

Parameters

- **`cache_dir`** (*`str`*) – the directory for storing cached models
- **`verbose`** (*`bool`*) – *True* to increase verbosity

Returns a POS flair model

7.5 spaCy models

class danlp.models.spacy_models.**SpacyChunking**(*model=None*, *cache_dir='/home/docs/.danlp'*, *verbose=False*)

Bases: object

Spacy Chunking Model

Parameters

- **model** (*spaCy model*) – a (preloaded) spaCy model
- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

predict(*text: Union[str, List[str]]*, *bio=True*)

Predict NP chunks from raw or tokenized text.

Parameters

- **text** – can either be a raw text or a list of tokens
- **bio** (*bool*) – *True* to return a list of labels in BIO format (same length as the sentence), *False* to return a list of tuples *(start id, end id, chunk label)*

Returns NP chunks - either a list of labels in BIO format or a list of tuples *(start id, end id, chunk label)*

Example "*Jeg kommer fra en lille by*" becomes

- a list of BIO tags: ['B-NP', 'O', 'O', 'B-NP', 'I-NP', 'I-NP']
- or a list of tuples : [(0, 1, 'NP'), (3, 6, 'NP')]

danlp.models.spacy_models.**load_spacy_chunking_model**(*spacy_model=None*, *cache_dir='/home/docs/.danlp'*, *verbose=False*)

Loads a spaCy chunking model.

Parameters

- **spacy_model** (*spaCy model*) – a (preloaded) spaCy model
- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

Returns a spaCy Chunking model

Note: A spaCy model can be previously loaded using load_spacy_model and given as an argument to load_spacy_chunking_model (for instance, to avoid loading the model twice)

danlp.models.spacy_models.**load_spacy_model**(*cache_dir='/home/docs/.danlp'*, *verbose=False*, *textcat=None*, *vectorError=False*)

Loads a spaCy model.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

- **textcat** (*bool*) – '*sentiment*' for loading the spaCy sentiment analyser
- **vectorError** (*bool*) –

Returns a spaCy model

> **Warning:** vectorError is a temporary work around error encounted by keeping two models and not been able to find reference name for vectors

7.6 XLM-R models

class danlp.models.xlmr_models.**XLMRCoref**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Bases: object

XLM-Roberta Coreference Resolution Model.

For predicting which expressions (word or group of words) refer to the same entity in a document.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

predict(*document: List[List[str]]*)

Predict coreferences in a document

Parameters **document** (*List[List[str]]*) – segmented and tokenized text

Returns a dictionary

Return type Dict

predict_clusters(*document: List[List[str]]*)

Predict clusters of entities in the document. Each predicted cluster contains a list of references. A reference is a tuple (ref text, start id, end id). The ids refer to the token ids in the entire document.

Parameters **document** (*List[List[str]]*) – segmented and tokenized text

Returns a list of clusters

Return type List[List[Tuple]]

class danlp.models.xlmr_models.**XlmrNed**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Bases: object

XLM-Roberta for Named Entity Disambiguation.

For predicting whether or not a specific entity (QID) is mentioned in a sentence.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

predict(*sentence: str*, *kg_context: str*)

Predict whether a QID is mentioned in a sentence or not.

Parameters

- **sentence** (*str*) – raw text

- **kg_context** (*str*) – raw text

Returns

Return type str

`danlp.models.xlmr_models.`**`load_xlmr_coref_model`**(*cache_dir='/home/docs/.danlp', verbose=False*)

Loads an XLM-R coreference model.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

Returns an XLM-R coreference model

`danlp.models.xlmr_models.`**`load_xlmr_ned_model`**(*cache_dir='/home/docs/.danlp', verbose=False*)

Loads an XLM-R model for named entity disambiguation.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

Returns an XLM-R NED model

CHAPTER

EIGHT

DATASETS

8.1 Dacoref

class danlp.datasets.dacoref.**Dacoref**(*cache_dir: str = '/home/docs/.danlp'*)

Bases: object

This Danish coreference annotation contains parts of the Copenhagen Dependency Treebank. It was originally annotated as part of the Copenhagen Dependency Treebank (CDT) project but never finished. This resource extends the annotation by using different mapping techniques and by augmenting with Qcodes from Wiktionary. Read more about it in the danlp docs.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

load_as_conllu(*predefined_splits: bool = False*)

Parameters **predefined_splits** (*bool*) – Boolean

Returns A single parsed conllu list or a list of train, dev, test split parsed conllu list depending on predefined_split

8.2 DaNED & DaWikiNED

class danlp.datasets.daned.**DaNED**(*cache_dir: str = '/home/docs/.danlp'*)

Bases: object

Class for loading the DaNED dataset. The DaNED dataset is derived from the Dacoref dataset which is itself based on the DDT (thus, divided in train, dev and test in the same way). It is annotated for named entity disambiguation (also refered as named entity linking).

Each entry is a tuple of a sentence and the QID of an entity. The label represents whether the entity corresponding to the QID is mentioned in the sentence. Each QID is linked to a knowledge graph (wikidata properties) and a description (wikidata description).

The same sentence can appear multiple times in the dataset (associated with different QIDs). But only one of them should have a label "1" (which corresponds to the correct entity).

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

get_kg_context_from_qid(*qid: str*, *output_as_dictionary=False*, *allow_online_search=False*)
Return the knowledge context and the description of an entity from its QID.

Parameters

- **qid** (*str*) – a wikidata QID
- **output_as_dictionary** (*bool*) – whether the properties should be a dictionary or a string (default)
- **allow_online_search** (*bool*) – whether searching Wikidata online when qid not in database (default False)

Returns string (or dictionary) of properties and description

load_with_pandas()
Loads the DaNED dataset in dataframes with pandas.

Returns 3 dataframes – train, dev, test

class danlp.datasets.dawikined.**DaWikiNED**(*cache_dir: str = '/home/docs/.danlp'*)
Bases: object

Class for loading the DaWikiNED dataset. The DaWikiNED dataset contains Wikipedia text annotated for named entity disambiguation. It contains only a train set as it is intended for use as a dataset augmentation for the DaNED dataset.

Each entry is a tuple of a sentence and the QID of an entity. The label represents whether the entity corresponding to the QID is mentioned in the sentence. Each QID is linked to a knowledge graph (wikidata properties) and a description (wikidata description).

The same sentence can appear multiple times in the dataset (associated with different QIDs). But only one of them should have a label "1" (which corresponds to the correct entity).

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

get_kg_context_from_qid(*qid: str*, *output_as_dictionary=False*, *allow_online_search=False*)
Return the knowledge context and the description of an entity from its QID.

Parameters

- **qid** (*str*) – a wikidata QID
- **output_as_dictionary** (*bool*) – whether the properties should be a dictionary or a string (default)
- **allow_online_search** (*bool*) – whether searching Wikidata online when qid not in database (default False)

Returns string (or dictionary) of properties and description

load_with_pandas()
Loads the DaWikiNED dataset in a dataframe with pandas.

Returns a dataframe for train data

8.3 DanNet

class `danlp.datasets.dannet.`**`DanNet`**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Bases: `object`

DanNet wrapper, providing functions to access the main features of DanNet. See also : https://cst.ku.dk/projekter/dannet/.

Dannet consists of a set of 4 databases:

- words
- word senses
- relations
- synsets

Parameters

- **cache_dir** (`str`) – the directory for storing cached models
- **verbose** (`bool`) – *True* to increase verbosity

domains(*word*, *pos=None*)

Returns the domains of *word*.

Parameters

- **word** – text
- **pos** – (list of) part of speech tag(s) (in "Noun", "Verb", "Adjective")

Returns list of domains

hypernyms(*word*, *pos=None*)

Returns the hypernyms of *word*.

Parameters

- **word** – text
- **pos** – (list of) part of speech tag(s) (in "Noun", "Verb", "Adjective")

Returns list of hypernyms

hyponyms(*word*, *pos=None*)

Returns the hyponyms of *word*.

Parameters

- **word** – text
- **pos** – (list of) part of speech tag(s) (in "Noun", "Verb", "Adjective")

Returns list of hypernyms

load_with_pandas()

Loads the datasets in 4 dataframes

Returns 4 dataframes: words, wordsenses, relations, synsets

meanings(*word*, *pos=None*)

Returns the meanings of *word*.

Parameters

- **word** – text
- **pos** – (list of) part of speech tag(s) (in "Noun", "Verb", "Adjective")

Returns list of meanings

pos (*word*)
Returns the part-of-speech tags *word* can be categorized with among "Noun", "Verb" or "Adjective".

Parameters **word** – text

Returns list of part-of-speech tags

synonyms (*word*, *pos=None*)
Returns the synonyms of *word*.

Parameters

- **word** – text
- **pos** – (list of) part of speech tag(s) (in "Noun", "Verb", "Adjective")

Returns list of synonyms

Example "*hav*" returns ["sø", "ocean"]

wordnet_relations (*word*, *pos=None*, *eurowordnet=True*)
Returns the name of the relations *word* is associated with.

Parameters

- **word** – text
- **pos** – (list of) part of speech tag(s) (in "Noun", "Verb", "Adjective")

Returns list of relations

8.4 Danish Dependency Treebank

class danlp.datasets.ddt.**DDT** (*cache_dir: str = '/home/docs/.danlp'*)
Bases: object

Class for loading the Danish Dependency Treebank (DDT) through several frameworks/formats.

The DDT dataset has been annotated with NER tags in the IOB2 format. The dataset is downloaded in CoNLL-U format, but with this class it can be converted to spaCy format or a simple NER format similar to the CoNLL 2003 NER format.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

load_as_conllu (*predefined_splits: bool = False*)
Load the DDT in CoNLL-U format.

Parameters **predefined_splits** (*bool*) –

Returns A single pyconll.Conll or a tuple of (train, dev, test) pyconll.Conll depending on predefined_split

load_with_flair(*predefined_splits: bool = False*)

Load the DDT with flair.

This function is inspired by the "Reading Your Own Sequence Labeling Dataset" from Flairs tutorial on reading corpora:

https://github.com/zalandoresearch/flair/blob/master/resources/docs/TUTORIAL_6_CORPUS.md

Parameters **predefined_splits** (*bool*) –

Returns ColumnCorpus

Note: TODO: Make a pull request to flair similar to this: https://github.com/zalandoresearch/flair/issues/383

load_with_spacy()

Loads the DDT with spaCy.

This function converts the conllu files to json in the spaCy format.

Returns GoldCorpus

Note: Not using jsonl because of: https://github.com/explosion/spaCy/issues/3523

8.5 DKHate

class danlp.datasets.dkhate.**DKHate**(*cache_dir: str = '/home/docs/.danlp'*)

Bases: object

Class for loading the DKHate dataset. The DKHate dataset contains user-generated comments from social media platforms (Facebook and Reddit) annotated for various types and target of offensive language. The original corpus has been used for the OffensEval 2020 shared task. Note that only labels for Offensive language identification (sub-task A) are available. Which means that each sample in this dataset is labelled with either *NOT* (Not Offensive) or *OFF* (Offensive).

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

load_with_pandas()

Loads the DKHate dataset in a dataframe with pandas.

Returns a dataframe for test data and a dataframe for train data

8.6 Sentiment datasets

class `danlp.datasets.sentiment.`**`AngryTweets`**(*cache_dir: str = '/home/docs/.danlp'*)

Bases: `object`

Class for loading the AngryTweets Sentiment dataset.

Parameters **`cache_dir`** (`str`) – the directory for storing cached models

`load_with_pandas`()

Loads the dataset in a dataframe.

Returns a dataframe

class `danlp.datasets.sentiment.`**`EuroparlSentiment1`**(*cache_dir: str = '/home/docs/.danlp'*)

Bases: `object`

Class for loading the Europarl Sentiment dataset.

Parameters **`cache_dir`** (`str`) – the directory for storing cached models

`load_with_pandas`()

Loads the dataset in a dataframe and drop duplicates and nan values

Returns a dataframe

class `danlp.datasets.sentiment.`**`EuroparlSentiment2`**(*cache_dir: str = '/home/docs/.danlp'*)

Bases: `object`

Class for loading the Europarl Sentiment dataset.

Parameters **`cache_dir`** (`str`) – the directory for storing cached models

`load_with_pandas`()

Loads the dataset as a dataframe

Returns a dataframe

class `danlp.datasets.sentiment.`**`LccSentiment`**(*cache_dir: str = '/home/docs/.danlp'*)

Bases: `object`

Class for loading the LCC Sentiment dataset.

Parameters **`cache_dir`** (`str`) – the directory for storing cached models

`load_with_pandas`()

Loads the dataset in a dataframe, combines and drops duplicates and nan values

Returns a dataframe

class `danlp.datasets.sentiment.`**`TwitterSent`**(*cache_dir: str = '/home/docs/.danlp'*)

Bases: `object`

Class for loading the Twitter Sentiment dataset.

Parameters **`cache_dir`** (`str`) – the directory for storing cached models

`load_with_pandas`()

Loads the dataset in a dataframe.

Returns a dataframe of the test set and a dataframe of the train set

8.7 DaUnimorph

class danlp.datasets.unimorph.**DaUnimorph**(*cache_dir='/home/docs/.danlp'*, *verbose=False*)

Bases: object

Danish Unimorph. See also : https://unimorph.github.io/.

The Danish Unimorph is a database which contains knowledge (lemmas and morphological features) about different forms of nouns and verbs in Danish.

Parameters

- **cache_dir** (*str*) – the directory for storing cached models
- **verbose** (*bool*) – *True* to increase verbosity

get_inflections(*form*, *pos=None*, *is_lemma=False*, *with_features=False*)

Returns all possible inflections (forms) of a word (based on its lemma)

Returns list of words

get_lemmas(*form*, *pos=None*, *with_features=False*)

Returns the lemma(s) of a word

Returns list of lemmas

load_with_pandas()

Loads the dataset in a dataframe

Returns a dataframe

8.8 WikiANN

class danlp.datasets.wiki_ann.**WikiAnn**(*cache_dir: str = '/home/docs/.danlp'*)

Bases: object

Class for loading the WikiANN dataset.

Parameters **cache_dir** (*str*) – the directory for storing cached models

load_with_flair(*predefined_splits: bool = False*)

Loads the dataset with flair.

Parameters **predefined_splits** (*bool*) –

Returns ColumnCorpus

load_with_spacy()

Loads the dataset with spaCy.

This function will convert the CoNLL02/03 format to json format for spaCy. As the function will return a spacy.gold.GoldCorpus which needs a dev set this function also splits the dataset into a 70/30 split as is done by Pan et al. (2017).

- Pan et al. (2017): https://aclweb.org/anthology/P17-1178

Returns GoldCorpus

8.9 Word similarity datasets

class `danlp.datasets.word_sim.`**`DSD`**(*cache_dir: str = '/home/docs/.danlp'*)

Bases: `object`

Class for loading the Danish Similarity Dataset dataset.

Parameters **`cache_dir`** (*`str`*) – the directory for storing cached models

`load_with_pandas`()

Loads the dataset in a dataframe.

Returns a dataframe

`words`() → set

Loads the vocabulary.

Return type set

class `danlp.datasets.word_sim.`**`WordSim353Da`**(*cache_dir: str = '/home/docs/.danlp'*)

Bases: `object`

Class for loading the WordSim-353 dataset.

Parameters **`cache_dir`** (*`str`*) – the directory for storing cached models

`load_with_pandas`()

Loads the dataset in a dataframe.

Returns a dataframe

`words`() → set

Loads the vocabulary.

Return type set

PYTHON MODULE INDEX

INDEX