nagisa Documentation

Release 0.2.7

Taishi Ikeda

Jul 06, 2020

Contents:

Nagisa is a python module for Japanese word segmentation/POS-tagging. It is designed to be *a simple and easy-to-use tool* for NLP beginners and has the following features.

- Based on recurrent neural networks.
- The word segmentation model uses character- and word-level features.
- The POS-tagging model uses tag dictionary information.

After installing nagisa by the command `$ pip install nagisa`, you can use a Japanese tokenizer and POS tagger in easy way.

```
import nagisa

text = 'Python'
words = nagisa.tagging(text)
print(words)
#=> Python/ / / / / /

# Get a list of words
print(words.words)
#=> ['Python', '', '', '', '', '', '']

# Get a list of POS-tags
print(words.postags)
#=> ['', '', '', '', '', '', '']
```

CHAPTER 1

Basic usage

Word identification is a fundamental step in processing of languages that have no word boundaries such as Japanese and Chinese. Nagisa provides a set of functions for Japanese natural language processing. You can easyily use Japanese word segmentation and POS-tagging by referring to the sample code below.

1.1 Sample code

You can install nagisa with pip.

```
$ pip install nagisa
```

A sample of word segmentation and POS-tagging for Japanese.

```
import nagisa

text = 'Python'
words = nagisa.tagging(text)
print(words)
#=> Python/ / / / / /

# Get a list of words
print(words.words)
#=> ['Python', '', '', '', '', '', '']

# Get a list of POS-tags
print(words.postags)
#=> ['', '', '', '', '', '', '']
```

The Output words can be controlled by POS-tags.

```
# Extarcting all nouns from a text
words = nagisa.extract(text, extract_postags=['名詞'])
print(words)
```

(continues on next page)

(continued from previous page)

```
#=> Python/ /

# Filtering specific POS-tags from a text
words = nagisa.filter(text, filter_postags=['', ''])
print(words)
#=> Python/ / / /

# A list of available POS-tags
print(nagisa.tagger.postags)
#=> ['', '', ... , 'URL']
```

Add the user dictionary in easy way.

```
# default
text = "33"
print(nagisa.tagging(text))
#=> 3/ / / / / / 3/ / / / /

# If a word ("3") is included in the single_word_list, it is recognized as a single␣
↪word.
new_tagger = nagisa.Tagger(single_word_list=['3'])
print(new_tagger.tagging(text))
#=> 3/ / / / / / 3/ /
```

Nagisa is good at capturing URLs and emoticons from an input text.

```
text = '(••)♪'
words = nagisa.tagging(text)
print(words)
#=> (••)/ / ♪/

url = 'https://github.com/taishi-i/nagisa(¯ω¯)'
words = nagisa.tagging(url)
print(words)
#=> https://github.com/taishi-i/nagisa/URL / / / / / (ω)/

words = nagisa.filter(url, filter_postags=['URL', '', ''])
print(words)
#=> / / /
```

CHAPTER 2

Tutorial

2.1 Train a Japanese word segmentation and POS tagging model for Universal Dependencies

This tutorial provides an example of training a joint word segmentation and POS tagging model by using Japanese universal dependencies treebank. You get to know how to build the original sequence labeling model through this tutorial.

2.1.1 Download the dataset

Before we get started, please run the command `$ pip install nagisa` to install the nagisa library. After installing it, download the Japanese UD treebank from UD_Japanese-GDS.

```
mkdir work
cd work
pip install nagisa
git clone https://github.com/UniversalDependencies/UD_Japanese-GSD
```

2.1.2 Preprocess the dataset and train a model

First, convert the downloaded data to the input data format for nagisa. The input data format of the train/dev/test files is tsv. The Each line is word and tag and one line is represented by **word \t tag**. Note that you put **EOS** between sentences. Refer to the tiny sample datasets.

Next, you train a joint word segmentation and POS-tagging model by using the `nagisa.fit()` function. After finish training the model, save the three model files (ja_gsd_ud.vocabs, ja_gsd_ud.params, ja_gsd_ud.hp) in the current directory.

Listing 1: tutorial_train_ud.py

```
import nagisa

def write_file(fn_in, fn_out):
    with open(fn_in, "r") as f:
        data = []
        words = []
        postags = []
        for line in f:
            line = line.strip()

            if len(line) > 0:
                prefix = line[0]
                if prefix != "#":
                    tokens = line.split("\t")
                    word = tokens[1]
                    postag = tokens[3]
                    words.append(word)
                    postags.append(postag)

            else:
                if (len(words) > 0) and (len(postags) > 0):
                    data.append([words, postags])
                    words = []
                    postags = []

    with open(fn_out, "w") as f:
        for words, postags in data:
            for word, postag in zip(words, postags):
                f.write("\t".join([word, postag])+"\n")
            f.write("EOS\n")


if __name__ == "__main__":
    # files
    fn_in_train = "UD_Japanese-GSD/ja_gsd-ud-train.conllu"
    fn_in_dev = "UD_Japanese-GSD/ja_gsd-ud-dev.conllu"
    fn_in_test = "UD_Japanese-GSD/ja_gsd-ud-test.conllu"

    fn_out_train = "ja_gsd_ud.train"
    fn_out_dev = "ja_gsd_ud.dev"
    fn_out_test = "ja_gsd_ud.test"

    fn_out_model = "ja_gsd_ud"

    # write files for nagisa
    write_file(fn_in_train, fn_out_train)
    write_file(fn_in_dev, fn_out_dev)
    write_file(fn_in_test, fn_out_test)

    # start training
    nagisa.fit(train_file=fn_out_train, dev_file=fn_out_dev,
               test_file=fn_out_test, model_name=fn_out_model)
```

This is a log of the training process.

```
[dynet] random seed: 1234
[dynet] allocating memory: 32MB
[dynet] memory allocation done.
[nagisa] LAYERS: 1
[nagisa] THRESHOLD: 3
[nagisa] DECAY: 1
[nagisa] EPOCH: 10
[nagisa] WINDOW_SIZE: 3
[nagisa] DIM_UNI: 32
[nagisa] DIM_BI: 16
[nagisa] DIM_WORD: 16
[nagisa] DIM_CTYPE: 8
[nagisa] DIM_TAGEMB: 16
[nagisa] DIM_HIDDEN: 100
[nagisa] LEARNING_RATE: 0.1
[nagisa] DROPOUT_RATE: 0.3
[nagisa] SEED: 1234
[nagisa] TRAINSET: ja_gsd_ud.train
[nagisa] TESTSET: ja_gsd_ud.test
[nagisa] DEVSET: ja_gsd_ud.dev
[nagisa] DICTIONARY: None
[nagisa] EMBEDDING: None
[nagisa] HYPERPARAMS: ja_gsd_ud.hp
[nagisa] MODEL: ja_gsd_ud.params
[nagisa] VOCAB: ja_gsd_ud.vocabs
[nagisa] EPOCH_MODEL: ja_gsd_ud_epoch.params
[nagisa] NUM_TRAIN: 7133
[nagisa] NUM_TEST: 551
[nagisa] NUM_DEV: 511
[nagisa] VOCAB_SIZE_UNI: 2352
[nagisa] VOCAB_SIZE_BI: 25108
[nagisa] VOCAB_SIZE_WORD: 9143
[nagisa] VOCAB_SIZE_POSTAG: 17
Epoch       LR      Loss    Time_m  DevWS_f1        DevPOS_f1       TestWS_f1
↪TestPOS_f1
1           0.100   13.37   1.462   91.84           87.75           91.63
↪87.35
2           0.100   6.280   1.473   92.57           89.67           92.44
↪89.15
3           0.100   4.961   1.535   93.54           90.98           93.62
↪90.18
4           0.050   4.256   1.430   92.52           90.19           93.62
↪90.18
5           0.025   3.200   1.443   93.46           91.06           93.62
↪90.18
6           0.025   2.581   1.512   93.56           91.49           93.88
↪91.29
7           0.025   2.379   1.475   93.58           91.50           93.73
↪91.15
8           0.025   2.218   1.476   93.63           91.57           93.92
↪91.31
9           0.025   2.122   1.475   93.78           91.63           94.09
↪91.40
10          0.012   1.985   1.434   93.55           91.39           94.09
↪91.40
```

2.1.3 Predict

You can build the tagger only by loading the three trained model files (ja_gsd_ud.vocabs, ja_gsd_ud.params, ja_gsd_ud.hp) to set arguments in `nagisa.Tagger()`.

Listing 2: tutorial_predict_ud.py

```
import nagisa

if __name__ == "__main__":
    # Build the tagger by loading the trained model files.
    ud_tagger = nagisa.Tagger(vocabs='ja_gsd_ud.vocabs',
                              params='ja_gsd_ud.params',
                              hp='ja_gsd_ud.hp')

    text = ""
    words = ud_tagger.tagging(text)
    print(words)
    #> /PROPN /SYM /PROPN /ADP /NOUN /NOUN
```

2.1.4 Error analysis

By checking a confusion matrix, you can see what the model is wrong with. The code shows how to create a confusion matrix by comparing the predicted tags with the gold-standard tags.

Listing 3: tutorial_error_analysis_ud.py

```
import nagisa
import pandas as pd

from sklearn.metrics import confusion_matrix


def load_file(filename):
    X = []
    Y = []
    words = []
    tags = []
    with open(filename, "r") as f:
        for line in f:
            line = line.rstrip()
            if line == "EOS":
                assert(len(words) == len(tags))
                X.append(words)
                Y.append(tags)
                words = []
                tags = []
            else:
                line = line.split("\t")
                word = " ".join(line[:-1])
                tag = line[-1]
                words.append(word)
                tags.append(tag)
    return X, Y


```

(continues on next page)

(continued from previous page)

```
def create_confusion_matrix(tagger, X, Y):
    true_cm = []
    pred_cm = []
    label2id = {}
    for i in range(len(X)):
        words = X[i]
        true_tags = Y[i]
        pred_tags = tagger.decode(words) # decoding

        if true_tags != pred_tags:
            for true_tag, pred_tag in zip(true_tags, pred_tags):
                if true_tag != pred_tag:
                    if true_tag not in label2id:
                        label2id[true_tag] = len(label2id)

                    if pred_tag not in label2id:
                        label2id[pred_tag] = len(label2id)

                    true_cm.append(label2id[true_tag])
                    pred_cm.append(label2id[pred_tag])

    cm = confusion_matrix(true_cm, pred_cm)
    labels = list(label2id.keys())
    cm_labeled = pd.DataFrame(cm, columns=labels, index=labels)
    return cm_labeled


if __name__ == "__main__":
    # load the testset
    test_X, test_Y = load_file("ja_gsd_ud.test")

    # build the tagger for UD
    ud_tagger = nagisa.Tagger(vocabs='ja_gsd_ud.vocabs',
                              params='ja_gsd_ud.params',
                              hp='ja_gsd_ud.hp')

    # create a confusion matrix if tagger make a mistake in prediction.
    cm_labeled = create_confusion_matrix(ud_tagger, test_X, test_Y)
    print(cm_labeled)
```

This is a confusion matrix if tagger make a mistake in prediction. This confusion matrix shows that the tagger often mistakes “NOUN” for “PROPN” in this UD_Japanese-GDS dataset.

```
        AUX  VERB  NOUN  ADV  PRON  PART  PUNCT  SYM  ADJ  PROPN  CCONJ  SCONJ  ADP␣
↪NUM  INTJ
AUX       0    16     2    0     0     0      0    0    2      0      0      1   25␣
↪0     0
VERB     14     0    23    0     1     0      0    0    2      0      0      1    0␣
↪0     0
NOUN      0    12     0    5     1     0      1    0   16    101      0      1    1␣
↪2     0
ADV       0     2     8    0     0     1      0    0    2      1      2      0    0␣
↪0     0
PRON      0     3     6    1     0     0      0    0    1      0      0      0    0␣
↪0     0
PART      1     0     4    0     0     0      0    0    0      0      0      0    0␣
↪0     0
```

(continues on next page)

(continued from previous page)

```
PUNCT    0    0    2    0    0    0    0    2    0    0    0    0    0
↪0    0
SYM      0    0    0    0    0    0    0    0    0    0    0    0    0
↪1    0
ADJ      8    6   41    3    0    1    0    0    0    4    0    0    0
↪0    0
PROPN    0    2   65    0    0    0    0    0    0    0    1    0    0
↪1    0
CCONJ    0    0    1    2    0    0    0    0    0    0    0    0    0
↪0    0
SCONJ    1    0    1    0    0    0    0    0    0    0    0    0    2
↪0    0
ADP      4    0    0    0    0    0    0    0    0    0    0    7    0
↪0    0
NUM      0    0    1    0    0    0    0    0    0    0    0    0    0
↪0    0
INTJ     0    0    0    1    0    0    0    0    0    0    0    0    0
↪0    0
```

CHAPTER 3

Tutorial (Japanese Named Entity Recognition)

3.1 Train a Japanese NER model for KWDLC

This tutorial provides an example of training a Japanese NER model by using Kyoto University Web Document Leads Corpus(KWDLC).

3.1.1 Download the dataset

Please download KWDLC from http://nlp.ist.i.kyoto-u.ac.jp/EN/index.php?KWDLC manually. Copy the corpus to the working directory.

3.1.2 Install python libraries

Before we get started, please run the following command to install the libraries used in this tutorial.

```
pip install nagisa
pip install seqeval
pip install beautifulsoup4
```

3.1.3 Preprocess the dataset

First, convert the downloaded data to the input data format for nagisa. The input data format of the train/dev/test files is the tsv format. The Each line is word and tag and one line is represented by **word \t tag**. Note that you put **EOS** between sentences.

This preprocess is a little complicated, so please copy the code below and run it. After running the code, **kwdlc.txt** is output to the working directory.

```
python tutorial_preprocess_kwdlc.py
```

Listing 1: tutorial_preprocess_kwdlc.py

```
import bs4
import glob


def load_kwdlc(dir_path):
    files = glob.glob(dir_path+"/*/*", recursive=True)

    data = []

    words = []
    position2ne = {}

    for fn in files:
        with open(fn, "r") as f:
            for line in f:
                line = line.strip()
                first_char = line[0]

                if first_char == "+":
                    soup = bs4.BeautifulSoup(line, "html.parser")
                    num_tags = len(soup.contents)
                    for i in range(num_tags):
                        if str(type(soup.contents[i])) == "<class 'bs4.element.Tag'>":
                            if "ne" == soup.contents[i].name:
                                target = soup.contents[i].attrs["target"]
                                netype = soup.contents[i].attrs["type"]

                                position2ne[len(words)] = [target, netype]

                elif first_char == "#" or first_char == "*":
                    None

                elif line == "EOS":
                    # process
                    if len(position2ne) > 0:
                        positions = position2ne.keys()
                        for position in positions:
                            target = position2ne[position][0]
                            netype = position2ne[position][1]

                    data.append([words, position2ne])

                    # reset
                    words = []
                    position2ne = {}

                else:
                    tokens = line.split()
                    surface = tokens[0]
                    words.append(surface)

    return data, position2ne


def write_kwdlc_as_single_file(filename, data, position2ne):
```

(continues on next page)

(continued from previous page)

```
    with open(filename, "w") as f:
        for line in data:
            words, position2ne = line

            nes = [v[0] for k, v in sorted(position2ne.items(), key=lambda x:x[0])]
            nes = list(reversed(nes))

            tags = [v[1] for k, v in sorted(position2ne.items(), key=lambda x:x[0])]
            tags = list(reversed(tags))

            if len(nes) == 0:
                None

            else:
                ne_tags = []

                ne = nes.pop()
                tag = tags.pop()
                ne_target_char = ne[0]

                partical = []
                for word in words:
                    first_char = word[0]
                    if first_char == ne_target_char:

                        if word in ne:
                            partical.append(word)

                            if "".join(partical) == ne:

                                for i, word in enumerate(partical):
                                    if i == 0:
                                        ne_tags.append("B-"+tag)
                                    elif i == len(partical) - 1:
                                        ne_tags.append("E-"+tag)
                                    else:
                                        ne_tags.append("M-"+tag)

                                if len(nes) > 0:
                                    ne = nes.pop()
                                    tag = tags.pop()
                                    ne_target_char = ne[0]

                                partical = []

                            else:
                                ne_target_char = ne[len("".join(partical))]

                        else:
                            partical = []
                            ne_tags.append("O")

                    else:
                        partical = []
                        ne_tags.append("O")

```

(continues on next page)

(continued from previous page)

```
                for word, ne_tag in zip(words, ne_tags):
                    f.write("\t".join([word, ne_tag])+"\n")
                f.write("EOS\n")


def main():
    dir_path = "./KWDLC-1.0/dat/rel"
    data, position2ne = load_kwdlc(dir_path)

    fn_out = "kwdlc.txt"
    write_kwdlc_as_single_file(fn_out, data, position2ne)


if __name__ == "__main__":
    main()
```

3.1.4 Train a model

Next, you train a NER model by using the `nagisa.fit()` function. After finish training the model, save the three model files (kwdlc_ner_model.vocabs, kwdlc_ner_model.params, kwdlc_ner_model.hp) in the current directory.

```
python tutorial_train_kwdlc.py
```

Listing 2: tutorial_train_kwdlc.py

```
import random

import nagisa


def write_file(filename, X, Y):
    with open(filename, "w") as f:
        for x, y in zip(X, Y):
            for word, tag in zip(x, y):
                f.write("\t".join([word, tag])+"\n")
            f.write("EOS\n")


def main():
    random.seed(1234)

    # preprocess
    fn_in = "kwdlc.txt"
    X, Y = nagisa.utils.load_file(fn_in)
    indice = [i for i in range(len(X))]
    random.shuffle(indice)

    num_train = int(0.8 * len(indice))
    num_dev = int(0.1 * len(indice))
    num_test = int(0.1 * len(indice))

    train_X = [X[i] for i in indice[:num_train]]
    train_Y = [Y[i] for i in indice[:num_train]]
```

(continues on next page)

(continued from previous page)

```
    dev_X = [X[i] for i in indice[num_train:num_train+num_dev]]
    dev_Y = [Y[i] for i in indice[num_train:num_train+num_dev]]
    test_X = [X[i] for i in indice[num_train+num_dev:num_train+num_dev+num_test]]
    test_Y = [Y[i] for i in indice[num_train+num_dev:num_train+num_dev+num_test]]

    fn_out_train = "kwdlc.train"
    fn_out_dev = "kwdlc.dev"
    fn_out_test = "kwdlc.test"
    write_file(fn_out_train, train_X, train_Y)
    write_file(fn_out_dev, dev_X, dev_Y)
    write_file(fn_out_test, test_X, test_Y)

    # start training
    fn_out_model = "kwdlc_ner_model"
    nagisa.fit(
        train_file=fn_out_train,
        dev_file=fn_out_dev,
        test_file=fn_out_test,
        model_name=fn_out_model
    )


if __name__ == "__main__":
    main()


```

This is a log of the training process.

```
[dynet] random seed: 1234
[dynet] allocating memory: 32MB
[dynet] memory allocation done.
[nagisa] LAYERS: 1
[nagisa] THRESHOLD: 3
[nagisa] DECAY: 1
[nagisa] EPOCH: 10
[nagisa] WINDOW_SIZE: 3
[nagisa] DIM_UNI: 32
[nagisa] DIM_BI: 16
[nagisa] DIM_WORD: 16
[nagisa] DIM_CTYPE: 8
[nagisa] DIM_TAGEMB: 16
[nagisa] DIM_HIDDEN: 100
[nagisa] LEARNING_RATE: 0.1
[nagisa] DROPOUT_RATE: 0.3
[nagisa] SEED: 1234
[nagisa] TRAINSET: kwdlc.train
[nagisa] TESTSET: kwdlc.test
[nagisa] DEVSET: kwdlc.dev
[nagisa] DICTIONARY: None
[nagisa] EMBEDDING: None
[nagisa] HYPERPARAMS: kwdlc_ner_model.hp
[nagisa] MODEL: kwdlc_ner_model.params
[nagisa] VOCAB: kwdlc_ner_model.vocabs
[nagisa] EPOCH_MODEL: kwdlc_ner_model_epoch.params
[nagisa] NUM_TRAIN: 3816
[nagisa] NUM_TEST: 477
```

(continues on next page)

(continued from previous page)

```
[nagisa] NUM_DEV: 477
[nagisa] VOCAB_SIZE_UNI: 1838
[nagisa] VOCAB_SIZE_BI: 12774
[nagisa] VOCAB_SIZE_WORD: 4809
[nagisa] VOCAB_SIZE_POSTAG: 29
Epoch       LR      Loss    Time_m  DevWS_f1        DevPOS_f1       TestWS_f1
↪TestPOS_f1
1           0.100   15.09   0.632   92.41           83.14           91.70
↪82.63
2           0.100   8.818   0.637   93.59           85.59           93.21
↪85.28
3           0.100   6.850   0.637   93.98           85.60           93.75
↪86.01
4           0.100   5.751   0.634   94.44           87.29           94.01
↪86.99
5           0.050   5.028   0.614   94.35           87.02           94.01
↪86.99
6           0.050   3.727   0.647   94.84           87.52           94.79
↪87.91
7           0.025   3.268   0.613   94.52           87.45           94.79
↪87.91
8           0.012   2.761   0.610   94.75           87.58           94.79
↪87.91
9           0.012   2.447   0.634   94.95           87.79           95.00
↪88.28
10          0.006   2.333   0.624   94.73           87.41           95.00
↪88.28
```

3.1.5 Predict

You can build the tagger only by loading the three trained model files (kwdlc_ner_model.vocabs, kwdlc_ner_model.params, kwdlc_ner_model.hp) to set arguments in `nagisa.Tagger()`.

```
python tutorial_predict_kwdlc.py
```

3.1.6 Error analysis

By checking tag-level accuracy/entity-level macro-f1/classification_report, you can see what the model is wrong with.

```
python tutorial_error_analysis_kwdlc.py
```

Listing 3: tutorial_error_analysis_kwdlc.py

```
import nagisa

from seqeval.metrics import f1_score
from seqeval.metrics import accuracy_score
from seqeval.metrics import classification_report


def main():
    # load the testset
    test_X, test_Y = nagisa.utils.load_file("kwdlc.test")
```

(continues on next page)

(continued from previous page)

```
    # build the tagger for kwdlc
    ner_tagger = nagisa.Tagger(vocabs='kwdlc_ner_model.vocabs',
                               params='kwdlc_ner_model.params',
                               hp='kwdlc_ner_model.hp')

    # predict
    true_Y = []
    pred_Y = []
    for words, true_y in zip(test_X, test_Y):
        pred_y= ner_tagger.decode(words)

        _pred_y = []
        _true_y = []
        for word, pred, true in zip(words, pred_y, true_y):
            _pred_y.append(pred)
            _true_y.append(true)
        true_Y.append(_true_y)
        pred_Y.append(_pred_y)

    # evaluate
    accuracy = accuracy_score(true_Y, pred_Y)
    print("accuracy: {}".format(accuracy))
    f1 = f1_score(true_Y, pred_Y)
    print("macro-f1: {}".format(f1))
    report = classification_report(true_Y, pred_Y)
    print(report)


if __name__ == "__main__":
    main()

```

```
accuracy: 0.9166868198307134
macro-f1: 0.5900383141762452
                  precision    recall  f1-score   support

    ARTIFACT       0.33      0.35      0.34        86
    OPTIONAL       0.32      0.19      0.24        31
ORGANIZATION       0.40      0.33      0.36       109
        DATE       0.84      0.87      0.86       154
    LOCATION       0.64      0.68      0.66       262
       MONEY       0.88      0.88      0.88        16
      PERSON       0.44      0.62      0.51        94
        TIME       0.40      0.44      0.42         9
     PERCENT       0.75      0.50      0.60         6

 avg / total       0.58      0.60      0.59       767
```

CHAPTER 4

API Reference

nagisa.**Tagger**(*vocabs=None*, *params=None*, *hp=None*, *single_word_list=None*)
This class has a word segmentation function and a POS-tagging function for Japanese.

nagisa.**wakati**(*text*, *lower=False*)
Word segmentation function. Return the segmented words.

args:

- text (str): An input sentence.
- lower (bool): If lower is True, all uppercase characters in a list of the words are converted into lowercase characters.

return:

- words (list): A list of the words.

nagisa.**tagging**(*text*, *lower=False*)
Return the words with POS-tags of the given sentence.

args:

- text (str): An input sentence.
- lower (bool): If lower is True, all uppercase characters in a list of the words are converted into lowercase characters.

return:

- object : The object of the words with POS-tags.

nagisa.**filter**(*text*, *lower=False*, *filter_postags=None*)
Return the filtered words with POS-tags of the given sentence.

args:

- text (str): An input sentence.
- lower (bool): If lower is True, all uppercase characters in a list of the words are converted into lowercase characters.

- filter_postags (list): Filtering the word with the POS-tag in filter_postags from a text.

return:

- object : The object of the words with POS-tags.

nagisa.**extract**(*text*, *lower=False*, *extract_postags=None*)
Return the extracted words with POS-tags of the given sentence.

args:

- text (str): An input sentence.
- lower (bool): If lower is True, all uppercase characters in a list of the words are converted into lowercase characters.
- filter_postags (list): Extracting the word with the POS-tag in filter_postags from a text.

return:

- object : The object of the words with POS-tags.

nagisa.**decode**(*words*, *lower=False*)
Return the words with tags of the given words.

args:

- words (list): Input words.
- lower (bool, optional): If lower is True, all uppercase characters in a list of the words are converted into lowercase characters.

return:

- object : The object of the words with tags.

nagisa.**fit**(*train_file*, *dev_file*, *test_file*, *model_name*, *dict_file=None*, *emb_file=None*, *delimiter='\t'*, *newline='EOS'*, *layers=1*, *min_count=2*, *decay=1*, *epoch=10*, *window_size=3*, *dim_uni=32*, *dim_bi=16*, *dim_word=16*, *dim_ctype=8*, *dim_tagemb=16*, *dim_hidden=100*, *learning_rate=0.1*, *dropout_rate=0.3*, *seed=1234*)
Train a joint word segmentation and sequence labeling (e.g, POS-tagging, NER) model.

args:

- train_file (str): Path to a train file.
- dev_file (str): Path to a development file for early stopping.
- test_file (str): Path to a test file for evaluation.
- model_name (str): Output model filename.
- dict_file (str, optional): Path to a dictionary file.
- emb_file (str, optional): Path to a pre-trained embedding file (word2vec format).
- delimiter (str, optional): Separate word and tag in each line by 'delimiter'.
- newline (str, optional): Separate lines in the file by 'newline'.
- layers (int, optional): RNN Layer size.
- min_count (int, optional): Ignores all words with total frequency lower than this.
- decay (int, optional): Learning rate decay.
- epoch (int, optional): Epoch size.
- window_size (int, optional): Window size of the context characters for word segmentation.

- dim_uni (int, optional): Dimensionality of the char-unigram vectors.
- dim_bi (int, optional): Dimensionality of the char-bigram vectors.
- dim_word (int, optional): Dimensionality of the word vectors.
- dim_ctype (int, optional): Dimensionality of the character-type vectors.
- dim_tagemb (int, optional): Dimensionality of the tag vectors.
- dim_hidden (int, optional): Dimensionality of the BiLSTM's hidden layer.
- learning_rate (float, optional): Learning rate of SGD.
- dropout_rate (float, optional): Dropout rate of the input vector for BiLSTMs.
- seed (int, optional): Random seed.

return:

- Nothing. After finish training, however, save the three model files (*.vocabs, *.params, *.hp) in the current directory.

Index