BAB VI

PENUTUP

6.1 Simpulan

Berdasarkan penelitian yang sudah dilakukan dapat ditarik simpulan sebagai berikut:

1. Membangun *dataset* yang tepat dapat dilakukan dengan cara mengumpulkan korpus yang berisi kalimat dengan variasi kata yang beragam, penggunaan gaya bahasa yang natural, jumlah kalimat / kata yang banyak, serta dilakukannya proses anotasi label dengan benar. Kemudian *dataset* juga harus disesuaikan formatnya dengan format yang dapat dikonsumsi oleh *spacy* yaitu *JSON* atau *JSONL*.
2. Pembuatan model *dependency parser* dan *named entity recognition* dapat menggunakan *script* yang sudah disediakan oleh *Spacy* ataupun melalui perintah *CLI spacy.* Dalam pembuatan model, *Spacy* membutuhkan *dataset* untuk proses pelatihan dan evaluasi yang terlebih dulu sudah diberi label/diantoasi dan disesuaikan formatnya.
3. Evaluasi model dapat dilakukan dengan menghitung akurasi atau *f1score.* Perhitungan dapat memanfaatkan perintah *CLI* bawaan *spacy* dan juga menggunakan perhitungan manual, jumlah prediksi yang benar dibagi dengan jumlah data yang diprediksi.

6.2 Saran

Berdasarkan penelitian yang sudah dilakukan diharapkan dimasa mendatang dapat dilakukan penyempurnaan kemampuan model sesuai dengan *library spacy.* Penyempurnaan model dapat dilakukan dengan menambahkan *pipeline* yang belum ada di dalam model ini seperti *text categorizer*. Selain itu penyempurnaan akurasi model juga dapat dilakukan dengan menambah jumlah *dataset* untuk proses pelatihan model. Keberagaman kata dan kalimat juga dapat mempengaruhi tingkat akurasi model.

DAFTAR PUSTAKA

[1] E. M. Sibarani, M. Nadial, E. Panggabean, and S. Meryana, "A Study of Parsing Process on Natural Language Processing in Bahasa Indonesia," *2013 IEEE 16th Int. Conf. Comput. Sci. Eng.*, pp. 309–316, 2013.

[2] A. Rahman and A. Purwarianti, "Ensemble Technique Utilization for Indonesian Dependency Parser," *31st Pacific Asia Conf. Lang. Inf. Comput.*, no. Paclic 31, pp. 64–71, 2017.

[3] R. Chaturvedi, "Recognizing Named Entities in Text based User Reviews," *Int. J. Adv. Res. Comput. Sci.*, vol. 8, no. 5, pp. 2104–2107, 2017.

[4] M. Kamayani and A. Purwarianti, "Dependency parsing for Indonesian," *Proc. 2011 Int. Conf. Electr. Eng. Informatics, ICEEI 2011*, no. July, 2011.

[5] S. Va, S. Suyoto, and A. J. Santoso, "The Development of Mobile Learning 'Imbuhan' Grammar Indonesian Language for Cambodian Students," *Int. J. Comput. Sci. Trends Technol. (IJCS T)*, vol. 4, no. 6, pp. 140–144, 2016.

[6] C. D. Soderberg and K. S. Olson, "Indonesian," *J. Int. Phon. Assoc.*, vol. 38, no. 2, pp. 209–213, 2008.

[7] A. U. Rahayu, "Differences on Language Structure between English and Indonesian," *Int. J. Lang. Lit. Linguist.*, vol. 1, no. 4, pp. 257–260, 2015.

[8] D. Chen and C. Manning, "A Fast and Accurate Dependency Parser using Neural Networks," *Proc. 2014 Conf. Empir. Methods Nat. Lang. Process.*, no. i, pp. 740–750, 2014.

[9] N. Green, S. D. Larasati, and Z. Žabokrtský, "Indonesian Dependency Treebank: Annotation and Parsing," *26th Pacific Asia Conf. Lang. Comput.*, pp. 137–145, 2012.

[10] S. Louvan and K. Kurniawan, "Empirical Evaluation of Character-Based Model on Neural Named-Entity Recognition in Indonesian Conversational Texts," 2016.

[11] M. Kocaleva, D. Stojanov, I. Stojanovik, and Z. Zdravev, "Pattern Recognition and Natural Language Processing: State of the Art," *TEM J.*, vol. 5, no. 2, pp. 236–240, 2016.

[12] R. Mcdonald *et al.*, "Universal Dependency Annotation for Multilingual Parsing," *Proc. 51st Annu. Meet. Assoc. Comput. Linguist.*, pp. 92–97, 2013.

[13] A. S. Wibawa and A. Purwarianti, "Indonesian Named-entity Recognition for 15 Classes Using Ensemble Supervised Learning," *Procedia Comput. Sci.*, vol. 81, no. December, pp. 221–228, 2016.

[14] M. Honnibal, "Embed, encode, attend, predict: The new deep learning formula for state-of-the-art NLP models," *Blog Explos. {AI}*, pp. 1–14, 2016.

[15] Y. Goldberg and J. Nivre, "A Dynamic Oracle for Arc-Eager Dependency Parsing," *COLING*, no. December, pp. 959–976, 2012.

[16] M. de M. De Marneffe *et al.*, "Universal Stanford Dependencies: A cross-linguistic typology," *Proc. Lr.*, pp. 4585–4592, 2014.

[17] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfittin," *J. Mach. Learn. Res.*, 2014.

[18] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, "Don't Decay the Learning Rate, Increase the Batch Size," no. 2017, pp. 1–11, 2017.

[19] A. H. Ralph Weischedel, Martha Palmer, Mitchell Marcus, Eduard Hovy, Sameer Pradhan, Lance Ramshaw, Nianwen Xue, Ann Taylor, Jeff Kaufman, Michelle Franchini, Mohammed El-Bachouti, Robert Belvin, "OntoNotes Release 5.0 LDC2013T19," *Linguist. Data Consort.*, 2013.

LAMPIRAN

1. *Script CLI Spacy Convert conllu* ke *JSON* (conllu2json.py)

```
# coding: utf8
from __future__ import unicode_literals

from ..._messages import Messages
from ...compat import json_dumps, path2str
from ...util import prints
from ...gold import iob_to_biluo
import re

def conllu2json(input_path, output_path, n_sents=10, use_morphology=False):

    """
    Convert conllu files into JSON format for use with train cli.
    use_morphology parameter enables appending morphology to tags, which is
    useful for languages such as Spanish, where UD tags are not so rich.
    """
    # by @dvsrepo, via #11 explosion/spacy-dev-resources

    """
    Extract NER tags if available and convert them so that they follow
    BILUO and the Wikipedia scheme
    """
    # by @katarkor

    docs = []
    sentences = []
    conll_tuples = read_conllx(input_path, use_morphology=use_morphology)
    checked_for_ner = False
    has_ner_tags = False

    for i, (raw_text, tokens) in enumerate(conll_tuples):
        sentence, brackets = tokens[0]
        if not checked_for_ner:
            has_ner_tags = is_ner(sentence[5][0])
            checked_for_ner = True
        sentences.append(generate_sentence(sentence, has_ner_tags))
        # Real-sized documents could be extracted using the comments on the
        # conluu document

        if(len(sentences) % n_sents == 0):
            doc = create_doc(raw_text, sentences, i)
            docs.append(doc)
            sentences = []

    output_filename = input_path.parts[-1].replace(".conll", ".json")
    output_filename = input_path.parts[-1].replace(".conllu", ".json")
    output_file = output_path / output_filename
    with output_file.open('w', encoding='utf-8') as f:
        f.write(json_dumps(docs))
    prints(Messages.M033.format(n_docs=len(docs)),
```

```
        title=Messages.M032.format(name=path2str(output_file)))

def is_ner(tag):

    """
    Check the 10th column of the first token to determine if the file contains
    NER tags
    """

    tag_match = re.match('([A-Z_]+)-([A-Z_]+)', tag)
    if tag_match:
        return True
    elif tag == "O":
        return True
    else:
        return False

def read_conllx(input_path, use_morphology=False, n=0):
    text = input_path.open('r', encoding='utf-8').read()
    i = 0
    for sent in text.strip().split('\n\n'):
        lines = sent.strip().split('\n')
        if lines:
            while lines[0].startswith('#'):
                # Mengambil data text yang ada di format .conllu
                raw = lines.pop(0)[9:]
            tokens = []
            for line in lines:

                parts = line.split('\t')
                id_, word, lemma, pos, tag, morph, head, dep, _1, iob = parts
                if '-' in id_ or '.' in id_:
                    continue
                try:
                    id_ = int(id_) - 1
                    head = (int(head) - 1) if head != '0' else id_
                    dep = 'ROOT' if dep == 'root' else dep
                    tag = pos if tag == '_' else tag
                    tag = tag+'__'+morph  if use_morphology else tag
                    tokens.append((id_, word, tag, head, dep, iob))
                except:
                    print(line)
                    raise
            tuples = [list(t) for t in zip(*tokens)]
            yield (raw, [[tuples, []]])
            i += 1
            if n >= 1 and i >= n:
                break

def simplify_tags(iob):

    """
    Simplify tags obtained from the dataset in order to follow Wikipedia
    scheme (PER, LOC, ORG, MISC). 'PER', 'LOC' and 'ORG' keep their tags, while
    'GPE_LOC' is simplified to 'LOC', 'GPE_ORG' to 'ORG' and all remaining tags to
```

```
    'MISC'.
    """

    new_iob = []
    for tag in iob:
        tag_match = re.match('([A-Z_]+)-([A-Z_]+)', tag)
        if tag_match:
            prefix = tag_match.group(1)
            suffix = tag_match.group(2)
            if suffix == 'GPE_LOC':
                suffix = 'LOC'
            elif suffix == 'GPE_ORG':
                suffix = 'ORG'
            elif suffix != 'PER' and suffix != 'LOC' and suffix != 'ORG':
                suffix = 'MISC'
            tag = prefix + '-' + suffix
        new_iob.append(tag)
    return new_iob

def generate_sentence(sent, has_ner_tags):
    (id_, word, tag, head, dep, iob) = sent
    sentence = {}
    tokens = []
    if has_ner_tags:
        iob = simplify_tags(iob)
        biluo = iob_to_biluo(iob)
    for i, id in enumerate(id_):
        token = {}
        token["id"] = id
        token["orth"] = word[i]
        token["tag"] = tag[i]
        token["head"] = head[i] - id
        token["dep"] = dep[i]
        if has_ner_tags:
            token["ner"] = biluo[i]
        tokens.append(token)
    sentence["tokens"] = tokens
    return sentence

def create_doc(raw_text,sentences,id):
    doc = {}
    paragraph = {}
    doc["id"] = id
    doc["paragraphs"] = []
    # Menambahkan raw text pada format JSON Spacy yang hilang
    paragraph["raw"] = raw_text
    paragraph["sentences"] = sentences
    doc["paragraphs"].append(paragraph)
    return doc
```

2. *Script CLI Spacy Evaluate* (evaluate.py)

```
# coding: utf8
from __future__ import unicode_literals, division, print_function

import plac
from timeit import default_timer as timer
from wasabi import Printer

from ..gold import GoldCorpus
from .. import util
from .. import displacy

@plac.annotations(
    model=("Model name or path", "positional", None, str),
    data_path=("Location of JSON-formatted evaluation data", "positional", None, str),
    gold_preproc=("Use gold preprocessing", "flag", "G", bool),
    gpu_id=("Use GPU", "option", "g", int),
    displacy_path=("Directory to output rendered parses as HTML", "option", "dp", str),
    displacy_limit=("Limit of parses to render as HTML", "option", "dl", int),
    return_scores=("Return dict containing model scores", "flag", "R", bool),
)
def evaluate(
    model,
    data_path,
    gpu_id=-1,
    gold_preproc=False,
    displacy_path=None,
    displacy_limit=25,
    return_scores=False,
):
    """
    Evaluate a model. To render a sample of parses in a HTML file, set an
    output directory as the displacy_path argument.
    """
    msg = Printer()
    util.fix_random_seed()
    if gpu_id >= 0:
        util.use_gpu(gpu_id)
    util.set_env_log(False)
    data_path = util.ensure_path(data_path)
    displacy_path = util.ensure_path(displacy_path)
    if not data_path.exists():
        msg.fail("Evaluation data not found", data_path, exits=1)
    if displacy_path and not displacy_path.exists():
        msg.fail("Visualization output directory not found", displacy_path, exits=1)
    corpus = GoldCorpus(data_path, data_path)
    nlp = util.load_model(model)
    dev_docs = list(corpus.dev_docs(nlp, gold_preproc=gold_preproc))
    begin = timer()
    scorer = nlp.evaluate(dev_docs, verbose=False)
    end = timer()
    nwords = sum(len(doc_gold[0]) for doc_gold in dev_docs)
    results = {
        "Time": "%.2f s" % (end - begin),
```

```
        "Words": nwords,
        "Words/s": "%.0f" % (nwords / (end - begin)),
        "TOK": "%.2f" % scorer.token_acc,
        "POS": "%.2f" % scorer.tags_acc,
        "UAS": "%.2f" % scorer.uas,
        "LAS": "%.2f" % scorer.las,
        "NER P": "%.2f" % scorer.ents_p,
        "NER R": "%.2f" % scorer.ents_r,
        "NER F": "%.2f" % scorer.ents_f,
    }
    msg.table(results, title="Results")

    if displacy_path:
        docs, golds = zip(*dev_docs)
        render_deps = "parser" in nlp.meta.get("pipeline", [])
        render_ents = "ner" in nlp.meta.get("pipeline", [])
        render_parses(
            docs,
            displacy_path,
            model_name=model,
            limit=displacy_limit,
            deps=render_deps,
            ents=render_ents,
        )
        msg.good("Generated {} parses as HTML".format(displacy_limit), displacy_path)
    if return_scores:
        return scorer.scores

def render_parses(docs, output_path, model_name="", limit=250, deps=True, ents=True):
    docs[0].user_data["title"] = model_name
    if ents:
        with (output_path / "entities.html").open("w") as file_:
            html = displacy.render(docs[:limit], style="ent", page=True)
            file_.write(html)
    if deps:
        with (output_path / "parses.html").open("w") as file_:
            html = displacy.render(
                docs[:limit], style="dep", page=True, options={"compact": True}
            )
file_.write(html)
```

3. *Script* untuk Menghitung Skor Evaluasi (scorer.py)

```
# coding: utf8
from __future__ import division, print_function, unicode_literals

from .gold import tags_to_entities, GoldParse

class PRFScore(object):
    """
    A precision / recall / F score
    """

    def __init__(self):
```

```
        self.tp = 0
        self.fp = 0
        self.fn = 0

    def score_set(self, cand, gold):
        self.tp += len(cand.intersection(gold))
        self.fp += len(cand - gold)
        self.fn += len(gold - cand)

    @property
    def precision(self):
        return self.tp / (self.tp + self.fp + 1e-100)

    @property
    def recall(self):
        return self.tp / (self.tp + self.fn + 1e-100)

    @property
    def fscore(self):
        p = self.precision
        r = self.recall
        return 2 * ((p * r) / (p + r + 1e-100))

class Scorer(object):
    """Compute evaluation scores."""

    def __init__(self, eval_punct=False):
        """Initialize the Scorer.
        eval_punct (bool): Evaluate the dependency attachments to and from
            punctuation.
        RETURNS (Scorer): The newly created object.
        DOCS: https://spacy.io/api/scorer#init
        """
        self.tokens = PRFScore()
        self.sbd = PRFScore()
        self.unlabelled = PRFScore()
        self.labelled = PRFScore()
        self.tags = PRFScore()
        self.ner = PRFScore()
        self.eval_punct = eval_punct

    @property
    def tags_acc(self):
        """RETURNS (float): Part-of-speech tag accuracy (fine grained tags,
            i.e. `Token.tag`).
        """
        return self.tags.fscore * 100

    @property
    def token_acc(self):
        """RETURNS (float): Tokenization accuracy."""
        return self.tokens.precision * 100

    @property
    def uas(self):
```

```
    """RETURNS (float): Unlabelled dependency score."""
    return self.unlabelled.fscore * 100

@property
def las(self):
    """RETURNS (float): Labelled depdendency score."""
    return self.labelled.fscore * 100

@property
def ents_p(self):
    """RETURNS (float): Named entity accuracy (precision)."""
    return self.ner.precision * 100

@property
def ents_r(self):
    """RETURNS (float): Named entity accuracy (recall)."""
    return self.ner.recall * 100

@property
def ents_f(self):
    """RETURNS (float): Named entity accuracy (F-score)."""
    return self.ner.fscore * 100

@property
def scores(self):
    """RETURNS (dict): All scores with keys `uas`, `las`, `ents_p`,
        `ents_r`, `ents_f`, `tags_acc` and `token_acc`.
    """
    return {
        "uas": self.uas,
        "las": self.las,
        "ents_p": self.ents_p,
        "ents_r": self.ents_r,
        "ents_f": self.ents_f,
        "tags_acc": self.tags_acc,
        "token_acc": self.token_acc,
    }

def score(self, doc, gold, verbose=False, punct_labels=("p", "punct")):
    """Update the evaluation scores from a single Doc / GoldParse pair.
    doc (Doc): The predicted annotations.
    gold (GoldParse): The correct annotations.
    verbose (bool): Print debugging information.
    punct_labels (tuple): Dependency labels for punctuation. Used to
        evaluate dependency attachments to punctuation if `eval_punct` is
        `True`.
    DOCS: https://spacy.io/api/scorer#score
    """
    if len(doc) != len(gold):
        gold = GoldParse.from_annot_tuples(doc, zip(*gold.orig_annot))
    gold_deps = set()
    gold_tags = set()
    gold_ents = set(tags_to_entities([annot[-1] for annot in gold.orig_annot]))
    for id_, word, tag, head, dep, ner in gold.orig_annot:
        gold_tags.add((id_, tag))
```

```
        if dep not in (None, "") and dep.lower() not in punct_labels:
            gold_deps.add((id_, head, dep.lower()))
    cand_deps = set()
    cand_tags = set()
    for token in doc:
        if token.orth_.isspace():
            continue
        gold_i = gold.cand_to_gold[token.i]
        if gold_i is None:
            self.tokens.fp += 1
        else:
            self.tokens.tp += 1
            cand_tags.add((gold_i, token.tag_))
        if token.dep_.lower() not in punct_labels and token.orth_.strip():
            gold_head = gold.cand_to_gold[token.head.i]
            # None is indistinct, so we can't just add it to the set
            # Multiple (None, None) deps are possible
            if gold_i is None or gold_head is None:
                self.unlabelled.fp += 1
                self.labelled.fp += 1
            else:
                cand_deps.add((gold_i, gold_head, token.dep_.lower()))
    if "-" not in [token[-1] for token in gold.orig_annot]:
        cand_ents = set()
        for ent in doc.ents:
            first = gold.cand_to_gold[ent.start]
            last = gold.cand_to_gold[ent.end - 1]
            if first is None or last is None:
                self.ner.fp += 1
            else:
                cand_ents.add((ent.label_, first, last))
        self.ner.score_set(cand_ents, gold_ents)
    self.tags.score_set(cand_tags, gold_tags)
    self.labelled.score_set(cand_deps, gold_deps)
    self.unlabelled.score_set(
        set(item[:2] for item in cand_deps), set(item[:2] for item in gold_deps)
    )
    if verbose:
        gold_words = [item[1] for item in gold.orig_annot]
        for w_id, h_id, dep in cand_deps - gold_deps:
            print("F", gold_words[w_id], dep, gold_words[h_id])
        for w_id, h_id, dep in gold_deps - cand_deps:
print("M", gold_words[w_id], dep, gold_words[h_id])
```

4. *Script CLI Spacy Train* (train.py)

```
# coding: utf8
from __future__ import unicode_literals, division, print_function

import plac
import os
from pathlib import Path
import tqdm
from thinc.neural._classes.model import Model
```

```
from timeit import default_timer as timer
import shutil
import srsly
from wasabi import Printer
import contextlib
import random

from ..ml import create_default_optimizer
from ..attrs import PROB, IS_OOV, CLUSTER, LANG
from ..gold import GoldCorpus
from ..compat import path2str
from .. import util
from .. import about

@plac.annotations(
    lang=("Model language", "positional", None, str),
    output_path=("Output directory to store model in", "positional", None, Path),
    train_path=("Location of JSON-formatted training data", "positional", None, Path),
    dev_path=("Location of JSON-formatted development data", "positional", None, Path),
    raw_text=(
        "Path to jsonl file with unlabelled text documents.",
        "option",
        "rt",
        Path,
    ),
    base_model=("Name of model to update (optional)", "option", "b", str),
    pipeline=("Comma-separated names of pipeline components", "option", "p", str),
    vectors=("Model to load vectors from", "option", "v", str),
    n_iter=("Number of iterations", "option", "n", int),
    n_early_stopping=(
        "Maximum number of training epochs without dev accuracy improvement",
        "option",
        "ne",
        int,
    ),
    n_examples=("Number of examples", "option", "ns", int),
    use_gpu=("Use GPU", "option", "g", int),
    version=("Model version", "option", "V", str),
    meta_path=("Optional path to meta.json to use as base.", "option", "m", Path),
    init_tok2vec=(
        "Path to pretrained weights for the token-to-vector parts of the models. See 'spacy
pretrain'. Experimental.",
        "option",
        "t2v",
        Path,
    ),
    parser_multitasks=(
        "Side objectives for parser CNN, e.g. 'dep' or 'dep,tag'",
        "option",
        "pt",
        str,
    ),
    entity_multitasks=(
        "Side objectives for NER CNN, e.g. 'dep' or 'dep,tag'",
        "option",
```

```
        "et",
        str,
    ),
    noise_level=("Amount of corruption for data augmentation", "option", "nl", float),
    eval_beam_widths=("Beam widths to evaluate, e.g. 4,8", "option", "bw", str),
    gold_preproc=("Use gold preprocessing", "flag", "G", bool),
    learn_tokens=("Make parser learn gold-standard tokenization", "flag", "T", bool),
    verbose=("Display more information for debug", "flag", "VV", bool),
    debug=("Run data diagnostics before training", "flag", "D", bool),
)
def train(
    lang,
    output_path,
    train_path,
    dev_path,
    raw_text=None,
    base_model=None,
    pipeline="tagger,parser,ner",
    vectors=None,
    n_iter=30,
    n_early_stopping=None,
    n_examples=0,
    use_gpu=-1,
    version="0.0.0",
    meta_path=None,
    init_tok2vec=None,
    parser_multitasks="",
    entity_multitasks="",
    noise_level=0.0,
    eval_beam_widths="",
    gold_preproc=False,
    learn_tokens=False,
    verbose=False,
    debug=False,
):
    """
    Train or update a spaCy model. Requires data to be formatted in spaCy's
    JSON format. To convert data from other formats, use the `spacy convert`
    command.
    """
    msg = Printer()
    util.fix_random_seed()
    util.set_env_log(verbose)

    # Make sure all files and paths exists if they are needed
    train_path = util.ensure_path(train_path)
    dev_path = util.ensure_path(dev_path)
    meta_path = util.ensure_path(meta_path)
    output_path = util.ensure_path(output_path)
    if raw_text is not None:
        raw_text = list(srsly.read_jsonl(raw_text))
    if not train_path or not train_path.exists():
        msg.fail("Training data not found", train_path, exits=1)
    if not dev_path or not dev_path.exists():
        msg.fail("Development data not found", dev_path, exits=1)
```

```
if meta_path is not None and not meta_path.exists():
    msg.fail("Can't find model meta.json", meta_path, exits=1)
meta = srsly.read_json(meta_path) if meta_path else {}
if output_path.exists() and [p for p in output_path.iterdir() if p.is_dir()]:
    msg.warn(
        "Output directory is not empty",
        "This can lead to unintended side effects when saving the model. "
        "Please use an empty directory or a different path instead. If "
        "the specified output path doesn't exist, the directory will be "
        "created for you.",
    )
if not output_path.exists():
    output_path.mkdir()

# Take dropout and batch size as generators of values -- dropout
# starts high and decays sharply, to force the optimizer to explore.
# Batch size starts at 1 and grows, so that we make updates quickly
# at the beginning of training.
dropout_rates = util.decaying(
    util.env_opt("dropout_from", 0.2),
    util.env_opt("dropout_to", 0.2),
    util.env_opt("dropout_decay", 0.0),
)
batch_sizes = util.compounding(
    util.env_opt("batch_from", 100.0),
    util.env_opt("batch_to", 1000.0),
    util.env_opt("batch_compound", 1.001),
)

if not eval_beam_widths:
    eval_beam_widths = [1]
else:
    eval_beam_widths = [int(bw) for bw in eval_beam_widths.split(",")]
    if 1 not in eval_beam_widths:
        eval_beam_widths.append(1)
    eval_beam_widths.sort()
has_beam_widths = eval_beam_widths != [1]

# Set up the base model and pipeline. If a base model is specified, load
# the model and make sure the pipeline matches the pipeline setting. If
# training starts from a blank model, intitalize the language class.
pipeline = [p.strip() for p in pipeline.split(",")]
msg.text("Training pipeline: {}".format(pipeline))
if base_model:
    msg.text("Starting with base model '{}'".format(base_model))
    nlp = util.load_model(base_model)
    if nlp.lang != lang:
        msg.fail(
            "Model language ('{}') doesn't match language specified as "
            "`lang` argument ('{}') ".format(nlp.lang, lang),
            exits=1,
        )
    other_pipes = [pipe for pipe in nlp.pipe_names if pipe not in pipeline]
    nlp.disable_pipes(*other_pipes)
    for pipe in pipeline:
```

```
        if pipe not in nlp.pipe_names:
            nlp.add_pipe(nlp.create_pipe(pipe))
    else:
        msg.text("Starting with blank model '{}'".format(lang))
        lang_cls = util.get_lang_class(lang)
        nlp = lang_cls()
        for pipe in pipeline:
            nlp.add_pipe(nlp.create_pipe(pipe))

    if learn_tokens:
        nlp.add_pipe(nlp.create_pipe("merge_subtokens"))

    if vectors:
        msg.text("Loading vector from model '{}'".format(vectors))
        _load_vectors(nlp, vectors)

    # Multitask objectives
    multitask_options = [("parser", parser_multitasks), ("ner", entity_multitasks)]
    for pipe_name, multitasks in multitask_options:
        if multitasks:
            if pipe_name not in pipeline:
                msg.fail(
                    "Can't use multitask objective without '{}' in the "
                    "pipeline".format(pipe_name)
                )
            pipe = nlp.get_pipe(pipe_name)
            for objective in multitasks.split(","):
                pipe.add_multitask_objective(objective)

    # Prepare training corpus
    msg.text("Counting training words (limit={})".format(n_examples))
    corpus = GoldCorpus(train_path, dev_path, limit=n_examples)
    n_train_words = corpus.count_train()

    if base_model:
        # Start with an existing model, use default optimizer
        optimizer = create_default_optimizer(Model.ops)
    else:
        # Start with a blank model, call begin_training
        optimizer = nlp.begin_training(lambda: corpus.train_tuples, device=use_gpu)

    nlp._optimizer = None

    # Load in pre-trained weights
    if init_tok2vec is not None:
        components = _load_pretrained_tok2vec(nlp, init_tok2vec)
        msg.text("Loaded pretrained tok2vec for: {}".format(components))

    # fmt: off
    row_head = ["Itn", "Dep Loss", "NER Loss", "UAS", "NER P", "NER R", "NER F", "Tag
%", "Token %", "CPU WPS", "GPU WPS"]
    row_widths = [3, 10, 10, 7, 7, 7, 7, 7, 7, 7, 7]
    if has_beam_widths:
        row_head.insert(1, "Beam W.")
        row_widths.insert(1, 7)
```

```
row_settings = {"widths": row_widths, "aligns": tuple(["r" for i in row_head]), "spacing": 2}
# fmt: on
print("")
msg.row(row_head, **row_settings)
msg.row(["-" * width for width in row_settings["widths"]], **row_settings)
try:
    iter_since_best = 0
    best_score = 0.0
    for i in range(n_iter):
        train_docs = corpus.train_docs(
            nlp, noise_level=noise_level, gold_preproc=gold_preproc, max_length=0
        )
        if raw_text:
            random.shuffle(raw_text)
            raw_batches = util.minibatch(
                (nlp.make_doc(rt["text"]) for rt in raw_text), size=8
            )
        words_seen = 0
        with tqdm.tqdm(total=n_train_words, leave=False) as pbar:
            losses = {}
            for batch in util.minibatch_by_words(train_docs, size=batch_sizes):
                if not batch:
                    continue
                docs, golds = zip(*batch)
                nlp.update(
                    docs,
                    golds,
                    sgd=optimizer,
                    drop=next(dropout_rates),
                    losses=losses,
                )
                if raw_text:
                    # If raw text is available, perform 'rehearsal' updates,
                    # which use unlabelled data to reduce overfitting.
                    raw_batch = list(next(raw_batches))
                    nlp.rehearse(raw_batch, sgd=optimizer, losses=losses)
                if not int(os.environ.get("LOG_FRIENDLY", 0)):
                    pbar.update(sum(len(doc) for doc in docs))
                words_seen += sum(len(doc) for doc in docs)
        with nlp.use_params(optimizer.averages):
            util.set_env_log(False)
            epoch_model_path = output_path / ("model%d" % i)
            nlp.to_disk(epoch_model_path)
            nlp_loaded = util.load_model_from_path(epoch_model_path)
            for beam_width in eval_beam_widths:
                for name, component in nlp_loaded.pipeline:
                    if hasattr(component, "cfg"):
                        component.cfg["beam_width"] = beam_width
                dev_docs = list(
                    corpus.dev_docs(nlp_loaded, gold_preproc=gold_preproc)
                )
                nwords = sum(len(doc_gold[0]) for doc_gold in dev_docs)
                start_time = timer()
                scorer = nlp_loaded.evaluate(dev_docs, debug)
                end_time = timer()
```

```
if use_gpu < 0:
    gpu_wps = None
    cpu_wps = nwords / (end_time - start_time)
else:
    gpu_wps = nwords / (end_time - start_time)
    with Model.use_device("cpu"):
        nlp_loaded = util.load_model_from_path(epoch_model_path)
        for name, component in nlp_loaded.pipeline:
            if hasattr(component, "cfg"):
                component.cfg["beam_width"] = beam_width
        dev_docs = list(
            corpus.dev_docs(nlp_loaded, gold_preproc=gold_preproc)
        )
        start_time = timer()
        scorer = nlp_loaded.evaluate(dev_docs)
        end_time = timer()
        cpu_wps = nwords / (end_time - start_time)
acc_loc = output_path / ("model%d" % i) / "accuracy.json"
srsly.write_json(acc_loc, scorer.scores)

# Update model meta.json
meta["lang"] = nlp.lang
meta["pipeline"] = nlp.pipe_names
meta["spacy_version"] = ">=%s" % about.__version__
if beam_width == 1:
    meta["speed"] = {
        "nwords": nwords,
        "cpu": cpu_wps,
        "gpu": gpu_wps,
    }
    meta["accuracy"] = scorer.scores
else:
    meta.setdefault("beam_accuracy", {})
    meta.setdefault("beam_speed", {})
    meta["beam_accuracy"][beam_width] = scorer.scores
    meta["beam_speed"][beam_width] = {
        "nwords": nwords,
        "cpu": cpu_wps,
        "gpu": gpu_wps,
    }
meta["vectors"] = {
    "width": nlp.vocab.vectors_length,
    "vectors": len(nlp.vocab.vectors),
    "keys": nlp.vocab.vectors.n_keys,
    "name": nlp.vocab.vectors.name,
}
meta.setdefault("name", "model%d" % i)
meta.setdefault("version", version)
meta_loc = output_path / ("model%d" % i) / "meta.json"
srsly.write_json(meta_loc, meta)
util.set_env_log(verbose)

progress = _get_progress(
    i,
    losses,
```

```
                scorer.scores,
                beam_width=beam_width if has_beam_widths else None,
                cpu_wps=cpu_wps,
                gpu_wps=gpu_wps,
            )
            msg.row(progress, **row_settings)
        # Early stopping
        if n_early_stopping is not None:
            current_score = _score_for_model(meta)
            if current_score < best_score:
                iter_since_best += 1
            else:
                iter_since_best = 0
                best_score = current_score
            if iter_since_best >= n_early_stopping:
                msg.text(
                    "Early stopping, best iteration "
                    "is: {}".format(i - iter_since_best)
                )
                msg.text(
                    "Best score = {}; Final iteration "
                    "score = {}".format(best_score, current_score)
                )
                break
    finally:
        with nlp.use_params(optimizer.averages):
            final_model_path = output_path / "model-final"
            nlp.to_disk(final_model_path)
        msg.good("Saved model to output directory", final_model_path)
        with msg.loading("Creating best model..."):
            best_model_path = _collate_best_model(meta, output_path, nlp.pipe_names)
        msg.good("Created best model", best_model_path)

def _score_for_model(meta):
    """ Returns mean score between tasks in pipeline that can be used for early stopping. """
    mean_acc = list()
    pipes = meta["pipeline"]
    acc = meta["accuracy"]
    if "tagger" in pipes:
        mean_acc.append(acc["tags_acc"])
    if "parser" in pipes:
        mean_acc.append((acc["uas"] + acc["las"]) / 2)
    if "ner" in pipes:
        mean_acc.append((acc["ents_p"] + acc["ents_r"] + acc["ents_f"]) / 3)
    return sum(mean_acc) / len(mean_acc)

@contextlib.contextmanager
def _create_progress_bar(total):
    if int(os.environ.get("LOG_FRIENDLY", 0)):
        yield
    else:
        pbar = tqdm.tqdm(total=total, leave=False)
        yield pbar

def _load_vectors(nlp, vectors):
```

```
    util.load_model(vectors, vocab=nlp.vocab)
    for lex in nlp.vocab:
        values = {}
        for attr, func in nlp.vocab.lex_attr_getters.items():
            # These attrs are expected to be set by data. Others should
            # be set by calling the language functions.
            if attr not in (CLUSTER, PROB, IS_OOV, LANG):
                values[lex.vocab.strings[attr]] = func(lex.orth_)
        lex.set_attrs(**values)
        lex.is_oov = False

def _load_pretrained_tok2vec(nlp, loc):
    """Load pre-trained weights for the 'token-to-vector' part of the component
    models, which is typically a CNN. See 'spacy pretrain'. Experimental.
    """
    with loc.open("rb") as file_:
        weights_data = file_.read()
    loaded = []
    for name, component in nlp.pipeline:
        if hasattr(component, "model") and hasattr(component.model, "tok2vec"):
            component.tok2vec.from_bytes(weights_data)
            loaded.append(name)
    return loaded

def _collate_best_model(meta, output_path, components):
    bests = {}
    for component in components:
        bests[component] = _find_best(output_path, component)
    best_dest = output_path / "model-best"
    shutil.copytree(path2str(output_path / "model-final"), path2str(best_dest))
    for component, best_component_src in bests.items():
        shutil.rmtree(path2str(best_dest / component))
        shutil.copytree(
            path2str(best_component_src / component), path2str(best_dest / component)
        )
        accs = srsly.read_json(best_component_src / "accuracy.json")
        for metric in _get_metrics(component):
            meta["accuracy"][metric] = accs[metric]
    srsly.write_json(best_dest / "meta.json", meta)
    return best_dest

def _find_best(experiment_dir, component):
    accuracies = []
    for epoch_model in experiment_dir.iterdir():
        if epoch_model.is_dir() and epoch_model.parts[-1] != "model-final":
            accs = srsly.read_json(epoch_model / "accuracy.json")
            scores = [accs.get(metric, 0.0) for metric in _get_metrics(component)]
            accuracies.append((scores, epoch_model))
    if accuracies:
        return max(accuracies)[1]
    else:
        return None

def _get_metrics(component):
    if component == "parser":
```

```
        return ("las", "uas", "token_acc")
    elif component == "tagger":
        return ("tags_acc",)
    elif component == "ner":
        return ("ents_f", "ents_p", "ents_r")
    return ("token_acc",)

def _get_progress(itn, losses, dev_scores, beam_width=None, cpu_wps=0.0, gpu_wps=0.0):
    scores = {}
    for col in [
        "dep_loss",
        "tag_loss",
        "uas",
        "tags_acc",
        "token_acc",
        "ents_p",
        "ents_r",
        "ents_f",
        "cpu_wps",
        "gpu_wps",
    ]:
        scores[col] = 0.0
    scores["dep_loss"] = losses.get("parser", 0.0)
    scores["ner_loss"] = losses.get("ner", 0.0)
    scores["tag_loss"] = losses.get("tagger", 0.0)
    scores.update(dev_scores)
    scores["cpu_wps"] = cpu_wps
    scores["gpu_wps"] = gpu_wps or 0.0
    result = [
        itn,
        "{:.3f}".format(scores["dep_loss"]),
        "{:.3f}".format(scores["ner_loss"]),
        "{:.3f}".format(scores["uas"]),
        "{:.3f}".format(scores["ents_p"]),
        "{:.3f}".format(scores["ents_r"]),
        "{:.3f}".format(scores["ents_f"]),
        "{:.3f}".format(scores["tags_acc"]),
        "{:.3f}".format(scores["token_acc"]),
        "{:.0f}".format(scores["cpu_wps"]),
        "{:.0f}".format(scores["gpu_wps"]),
    ]
    if beam_width is not None:
        result.insert(1, beam_width)
return result
```

5. *Script Convert* conllu ke *JSONL* (conllu2jsonl.py)

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:

from conllu import parse_incr
from io import open
```

```
# #### Open .conllu file

# In[8]:

data = open("id_gsd-ud-test.conllu","r",encoding="utf-8")

# #### Extract the text and save it in a list

# In[9]:

tokenList = list()

for token in parse_incr(data):
    tokenList.append(token)

# #### Create and Write a JSONL format

# In[10]:

file = open("id_gsd-ud-test.jsonl","w+")

for t in tokenList:
    file.write("{" + "'text'" + ":" + " " + "'" + t.metadata["text"].replace("'","") + "'" + "}" + "\n")

file.close()

# In[ ]:
```

6. *Script* untuk Melatih Model *NER* (train-ner.py)

```
#!/usr/bin/env python
# coding: utf-8

# # Convert JSONL into list

# In[1]:

import ujson
import spacy
from pathlib import Path

# In[2]:

def read_jsonl(file_path):
    """Read a .jsonl file and yield its contents line by line.
    file_path (unicode / Path): The file path.
    YIELDS: The loaded JSON contents of each line.
    """
    with Path(file_path).open('r', encoding='utf8') as f:
        for line in f:
            try:  # hack to handle broken jsonl
                yield ujson.loads(line.strip())
            except ValueError:
```

```
            continue

# In[3]:

# def write_jsonl(file_path, lines):
#     """Create a .jsonl file and dump contents.
#     file_path (unicode / Path): The path to the output file.
#     lines (list): The JSON-serializable contents of each line.
#     """
#     data = [ujson.dumps(line, escape_forward_slashes=False) for line in lines]
#     Path(file_path).open('w', encoding='utf-8').write('\n'.join(data))

# In[4]:

list_of_train = []
path = read_jsonl("ner-train.jsonl")
for i in path:
    list_of_train.append(i)

# # Training Process

# In[5]:

"""Example of training spaCy's named entity recognizer, starting off with an
existing model or a blank model.
For more details, see the documentation:
* Training: https://spacy.io/usage/training
* NER: https://spacy.io/usage/linguistic-features#named-entities
Compatible with: spaCy v2.0.0+
Last tested with: v2.1.0
"""
from __future__ import unicode_literals, print_function

import plac
import random
from pathlib import Path
import spacy
from spacy.util import minibatch, compounding, decaying

# Training data
TRAIN_DATA = list_of_train


@plac.annotations(
    model=("dep-model"),
    output_dir=("Final/"),
    n_iter=(1000),
)
def main(model=None, output_dir=None, n_iter=100):
    best_loss = 0.0
    iter_since_best = 0
    n_early_stopping = 10

    with open("INFO-model.txt", "w") as text_file:
        """Load the model, set up the pipeline and train the entity recognizer."""
```

```
if model is not None:
    nlp = spacy.load(model)  # load existing spaCy model
    print("Loaded model '%s'" % model, file=text_file)
else:
    nlp = spacy.blank("en")  # create blank Language class
    print("Created blank 'en' model", file=text_file)

# create the built-in pipeline components and add them to the pipeline
# nlp.create_pipe works for built-ins that are registered with spaCy
if "ner" not in nlp.pipe_names:
    ner = nlp.create_pipe("ner")
    nlp.add_pipe(ner, last=True)
# otherwise, get it so we can add labels
else:
    ner = nlp.get_pipe("ner")

# add labels
for _, annotations in TRAIN_DATA:
    for ent in annotations.get("entities"):
        ner.add_label(ent[2])

# get names of other pipes to disable them during training
other_pipes = [pipe for pipe in nlp.pipe_names if pipe != "ner"]
with nlp.disable_pipes(*other_pipes):  # only train NER
    nlp.begin_training()
    for itn in range(n_iter):
        random.shuffle(TRAIN_DATA)
        losses = {}
        # batch up the examples using spaCy's minibatch
        batches = minibatch(TRAIN_DATA, size=compounding(1.0, 100.0, 1.001))
        # dropout decaying as spaCy mentions in tips and advice
        dropout = decaying(0.5, 0.2, 0.0001)
        for batch in batches:
            texts, annotations = zip(*batch)
            nlp.update(
                texts,  # batch of texts
                annotations,  # batch of annotations
                drop=next(dropout),  # dropout - make it harder to memorise data
                losses=losses,
            )
        print("Losses", losses, file=text_file)

        # save model to output directory
        if output_dir is not None:
            output_dir = Path(output_dir)
            if not output_dir.exists():
                output_dir.mkdir()
            # save each epochs model into directory
            output_epoch = output_dir / ("model%d" % itn)
            nlp.to_disk(output_epoch)

        # Early Stopping
        current_loss = losses['ner']

        if itn == 0:
```

```
            best_loss = current_loss
        elif current_loss > best_loss:
            iter_since_best += 1
        else:
            iter_since_best = 0
            best_loss = current_loss

        if iter_since_best >= n_early_stopping:
            best_iter = itn - iter_since_best
            print("Early stopping, best iteration is: {}".format(itn - iter_since_best),
file=text_file)
            print("Best score = {}; Final iteration score = {}".format(best_loss, current_loss),
file=text_file)
            break


    # test the best model
    output_best = output_dir / ("model%d" % best_iter)
    print("Loading from best model", output_best, file=text_file)
    nlp2 = spacy.load(output_best)
    for text, _ in TRAIN_DATA:
        doc = nlp2(text)
        print("Entities", [(ent.text, ent.label_) for ent in doc.ents], file=text_file)
        print("Tokens", [(t.text, t.ent_type_, t.ent_iob) for t in doc], file=text_file)

# In[ ]:

main("dep-model/","Final/",1000)

# In[ ]:
```