SAKI SS19 Homework 2

Author: Maximilian Lattka

Program code: https://github.com/nakami/oss-saki-ss19-exercise-2/releases/tag/oss-saki-ss19-exercise-2

Summary

Exercise 2 covers Named Entity Recognition (NER) which is a common application of Natural Language Processing (NLP). The provided dataset contains 701 annotated resumes in JSON format. Technically speaking, these resumes consist of the resume as raw text and the label annotations described by offset and words. For NER, Machine Learning algorithms can be used to identify and label words (also called "tokens") which correspond to particular entities. In regards to documents like the provided resumes, it might be of interest to find specific information like the level of education and filter resumes depending on certain requirements for an open job posting.

Data Exploration and Preprocessing

Taking a look at the provided data (`data/Entity Recognition in Resumes.json`), I first explored the format and preprocessed the resumes. See the completed jupyter notebook "`data_exploration.ipynb`"-classifier which was provided by Mr. Loutzenhiser for the procedure. Also, for the task at hand being rather difficult, I limited the Named Entities to be identified to the labels "Name", "Degree" and "Designation". I also removed resumes with did not contain any annotations. The remaining ones (690) can be found in `data/converted_resumes.json` in a more compact format. Further preprocessing was done with the NLP-framework `spacy`[1] in the jupyter notebook "`spacy_ner.ipynb`" by mapping our labels to the ones used in spacy and filtering out resumes which did not contain all of my chosen labels. The amount of resumes decreased to 572 and further to 570 by filtering out resumes with syntax errors via a trial `train_spacy_ner(...)`-call. Next, I prepared a 75-25 training and test split to make cross-validation possible. The NLP-framework `flair` (which is used later on) requires labels in the BILUO-format and also for the data to be sentence separated (one work/token per line and an empty line between sentences). In the jupyter notebook I took care of that with the help of the provided skeleton. Splitting the data into sentences for resumes is a difficult task, because often bulletpoints and choppy writing is used instead of natural sentences. I simply splitted the data by punctuation and focused on tokens which merely consist of a dot or a question/exclamation mark. The final dataset consists of 10445 trainings and 3100 test sentences. See the training and test set in `data/training_bilou.txt` and `data/test_bilou.txt`.

Evaluation on token-/entity-level and with different metrics

A common method to evaluate the classifier's decisions is to do so for each token alone. This is called token-level evaluation. In the jupyter notebook I first evaluated a `spaCy` trained model this way using the accuracy metric. I used the same metric in the previous exercise, so this is familiar. However, especially for NER-like tasks, some classification errors may be more complex and deserve further investigation. Though, if a Named Entity was identified but labeled incorrectly or the boundaries are incorrect but the labels are sound, one might weight these partial errors less harsh and do evaluation based on entities and not tokens. Besides the accuracy metric, one should therefore also consider evaluating with the metrics precision, recall and f1 as different types of situations occur and accuracy only takes the ratio of true positives and negatives in account (in ratio to the rest). The ratio of false positives and negatives and these between all among each other may also be of interest and can be calculated using precision, recall and f1. Mr. Loutzenhiser provided some articles in that regard. See the jupyter notebook "`spacy_ner.ipynb`" for my implementation and results for the token- and entity-level evaluation on a spaCy trained classifier. However, the focus of this exercise is the NLP-framework `flair` in combination with Google Colab and therefore I keep this section

[1] https://spacy.io/

brief.

Using the flair-framework in Google Colab

Mr. Loutzenhiser provided insight in working with the NLP-framework `flair`[2] in combination with the software development platform `Google Colab`[3]. While `flair` is a powerful NLP tool based on the well-known pytorch neural network Machine Learning library, it is able to accelerate trainings with a graphical processing unit (GPU). `Google Colab` on the other hand provides a jupyter notebook environment and a GPU free of charge. In regards to this exercise, I completed the guiding jupyter notebook "`flair_data_preparation.ipynb`" and "`flair_nlp_colab.ipynb`". The second notebook is primarily relevant as the training and evaluation is done there. To customize training with flair, there are different embeddings to choose and combine ("stacked embeddings") from which can be seen as feature vectors. In the skeleton there are four different types suggested to be used: the `GloVe WordEmbedding` which is a pre-trained static embedding on word-level, `CharacterEmbedding` which works on character-level and is trained along with the overall training process and two exclusively developed `FlairEmbeddings` (`news-forward` and `news-backward`) which take the context of a particular text sequence in account.

Evaluation

I trained several `flair` NER-taggers with different combinations of embeddings and compared the evaluation metric results which are computed right after training. As mentioned earlier, the data to be trained is annotated with labels for "Name", "Degree" and "Designation". No development set was provided, so the `flair` internals split the training set. Therefore, training was done with 9401 train, 1044 dev, and 3100 test sentences. See the directory `taggers` which contains the output-files for each trained tagger. I did not include the models, because some ended up being very large in file size (above 400mb). For convenience, see the `results` directory consisting of markdown-formatted reports. Eventually, I ended up with six trained taggers with a leaning rate of 10% and 20 maximum epochs:

	WordEmbeddings GloVe	**CharacterEmbeddings**	**FlairEmbeddings news-forward**	**FlairEmbeddings news-backward**
resume-WEglove	X			
resume-CE		X		
resume-WEglove-CE	X	X		
resume-WEglove-flairNF-flairNB	X		X	X
resume-CE-flairNF-flairNB		X	X	X
resume-WEglove-CE-flairNF-flairNB	X	X	X	X

Naming scheme of the six trained NER-taggers

Although taking half the time to train, the `WordEmbeddings` result in a rather large model (170mb) in comparison to the `CharacterEmbeddings` (3mb). The `FlairEmbeddings` seem to have a big impact on training runtime and model file size as well. Further experiments with more variations may be of interest. `Google Colab` has internal idle timings which kill the session after a few hours which increase the risk of losing progress on models with models having a long total runtime.

[2] https://github.com/zalandoresearch/flair
[3] https://colab.research.google.com/notebooks/welcome.ipynb

	Estimate runtime per epoch (in minutes)	Estimate total runtime (in minutes)	Size of model on disk (in megabytes)
resume-WEglove	2:30	45	170
resume-CE	4:40	90	3
resume-WEglove-CE	5:00	100	174
resume-WEglove-flairNF-flairNB	12:00	220	412
resume-CE-flairNF-flairNB	14:00	280	239
resume-WEglove-CE-flairNF-flairNB	14:00	285	414

Comparing runtime and model file size of the trained NER-taggers

Depending on the chosen embeddings, we are able to observe more or less better numbers the more embeddings we combine. An exception is model *resume-WEglove-CE-flairNF-flairNB* which performs worse than model *resume-WEglove-flairNF-flairNB* by adding the `CharacterEmbeddings`. However these average scores don't look very promising in total.

	MICRO_AVG accuracy	MICRO_AVG f1-score	MACRO_AVG accuracy	MACRO_AVG f1-score
resume-WEglove	0.4596	0.6298	0.505	0.6289
resume-CE	0.2948	0.4553	0.3246	0.4369
resume-WEglove-CE	0.4517	0.6223	0.5057	0.6271
resume-WEglove-flairNF-flairNB	0.525	0.6885	0.5612	0.6862
resume-CE-flairNF-flairNB	0.4833	0.6516	0.5193	0.6452
resume-WEglove-CE-flairNF-flairNB	0.4983	0.6652	0.5373	0.6631

Comparing accuracy and f1-scores of the trained NER-taggers

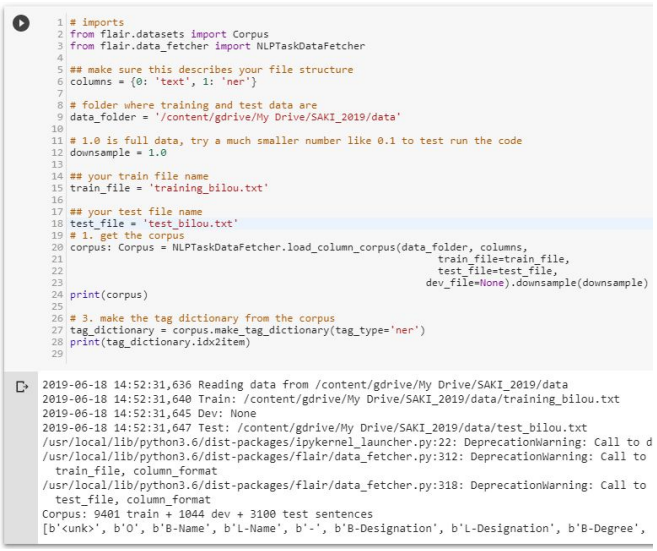
As this is a multi-class classification setup, looking into the specific scores for each class may help understand these fairly low numbers. For this purpose I chose to look at the last model which has all four embeddings enabled:

```
MICRO_AVG: acc 0.4983 - f1-score 0.6652
MACRO_AVG: acc 0.5373 - f1-score 0.6630666666666667
-             precision: 0.6941 - recall: 0.1239 - accuracy: 0.1175 - f1-score: 0.2103
Degree        precision: 0.6131 - recall: 0.7305 - accuracy: 0.5000 - f1-score: 0.6667
Designation   precision: 0.6991 - recall: 0.7007 - accuracy: 0.5383 - f1-score: 0.6999
L-Degree      precision: 0.6471 - recall: 0.7226 - accuracy: 0.5183 - f1-score: 0.6828
L-Designation precision: 0.7470 - recall: 0.7451 - accuracy: 0.5950 - f1-score: 0.7460
L-Name        precision: 0.9589 - recall: 0.9589 - accuracy: 0.9211 - f1-score: 0.9589
Name          precision: 0.9650 - recall: 0.9452 - accuracy: 0.9139 - f1-score: 0.9550
U-Degree      precision: 0.7451 - recall: 0.5846 - accuracy: 0.4872 - f1-score: 0.6552
U-Designation precision: 0.5789 - recall: 0.2973 - accuracy: 0.2444 - f1-score: 0.3928
```

Comparing accuracy and f1-scores of resume-WEglove-CE-flairNF-flairNB

Checking the classes on the left side we see class "-" which looks like an artifact in the dataset, and a violation to the BILUO-format. Also, the scores for the class are quite bad with an accuracy of less than 0.12 and a f1-score of 0.21. We also are able to see that names are labeled extraordinarily well. For more details on this particular model or the others check the directories `taggers` and `results`.

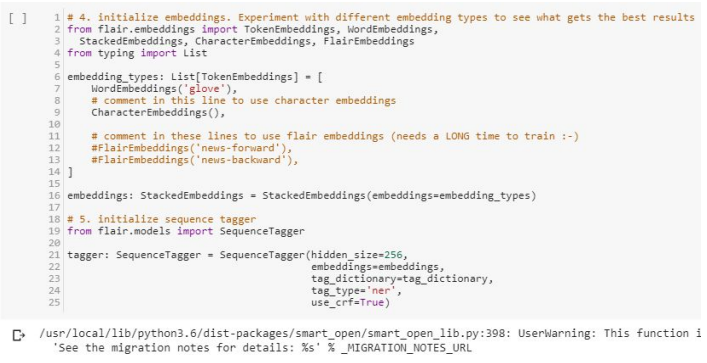
Screenshots



```
# imports
from flair.datasets import Corpus
from flair.data_fetcher import NLPTaskDataFetcher

## make sure this describes your file structure
columns = {0: 'text', 1: 'ner'}

# folder where training and test data are
data_folder = '/content/gdrive/My Drive/SAKI_2019/data'

# 1.0 is full data, try a much smaller number like 0.1 to test run the code
downsample = 1.0

## your train file name
train_file = 'training_bilou.txt'

## your test file name
test_file = 'test_bilou.txt'
# 1. get the corpus
corpus: Corpus = NLPTaskDataFetcher.load_column_corpus(data_folder, columns,
                                                       train_file=train_file,
                                                       test_file=test_file,
                                                      dev_file=None).downsample(downsample)
print(corpus)

# 3. make the tag dictionary from the corpus
tag_dictionary = corpus.make_tag_dictionary(tag_type='ner')
print(tag_dictionary.idx2item)
```

```
2019-06-18 14:52:31,636 Reading data from /content/gdrive/My Drive/SAKI_2019/data
2019-06-18 14:52:31,640 Train: /content/gdrive/My Drive/SAKI_2019/data/training_bilou.txt
2019-06-18 14:52:31,645 Dev: None
2019-06-18 14:52:31,647 Test: /content/gdrive/My Drive/SAKI_2019/data/test_bilou.txt
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:22: DeprecationWarning: Call to d
/usr/local/lib/python3.6/dist-packages/flair/data_fetcher.py:312: DeprecationWarning: Call to
  train_file, column_format
/usr/local/lib/python3.6/dist-packages/flair/data_fetcher.py:318: DeprecationWarning: Call to
  test_file, column_format
Corpus: 9401 train + 1044 dev + 3100 test sentences
[b'<unk>', b'O', b'B-Name', b'L-Name', b'-', b'B-Designation', b'L-Designation', b'B-Degree',
```

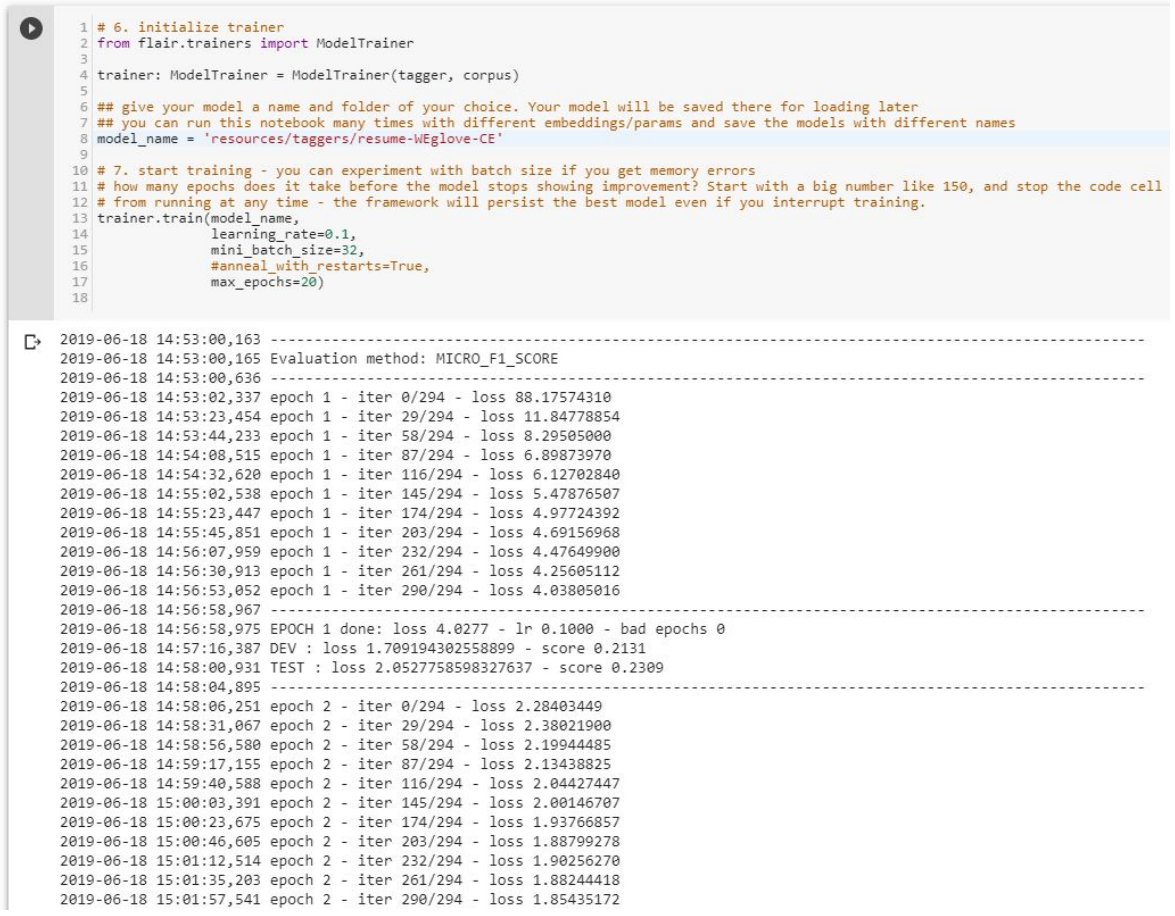
Loading the train and test dataset



```
# 4. initialize embeddings. Experiment with different embedding types to see what gets the best results
from flair.embeddings import TokenEmbeddings, WordEmbeddings,
  StackedEmbeddings, CharacterEmbeddings, FlairEmbeddings
from typing import List

embedding_types: List[TokenEmbeddings] = [
    WordEmbeddings('glove'),
    # comment in this line to use character embeddings
    CharacterEmbeddings(),

    # comment in these lines to use flair embeddings (needs a LONG time to train :-)
    #FlairEmbeddings('news-forward'),
    #FlairEmbeddings('news-backward'),
]

embeddings: StackedEmbeddings = StackedEmbeddings(embeddings=embedding_types)

# 5. initialize sequence tagger
from flair.models import SequenceTagger

tagger: SequenceTagger = SequenceTagger(hidden_size=256,
                                        embeddings=embeddings,
                                        tag_dictionary=tag_dictionary,
                                        tag_type='ner',
                                        use_crf=True)
```

```
/usr/local/lib/python3.6/dist-packages/smart_open/smart_open_lib.py:398: UserWarning: This function i
  'See the migration notes for details: %s' % _MIGRATION_NOTES_URL
```

Setting up the embeddings



```
# 6. initialize trainer
from flair.trainers import ModelTrainer

trainer: ModelTrainer = ModelTrainer(tagger, corpus)

## give your model a name and folder of your choice. Your model will be saved there for loading later
## you can run this notebook many times with different embeddings/params and save the models with different names
model_name = 'resources/taggers/resume-WEglove-CE'

# 7. start training - you can experiment with batch size if you get memory errors
# how many epochs does it take before the model stops showing improvement? Start with a big number like 150, and stop the code cell
# from running at any time - the framework will persist the best model even if you interrupt training.
trainer.train(model_name,
              learning_rate=0.1,
              mini_batch_size=32,
              #anneal_with_restarts=True,
              max_epochs=20)
```

```
2019-06-18 14:53:00,163 ----------------------------------------------------------------------------------------------------
2019-06-18 14:53:00,165 Evaluation method: MICRO_F1_SCORE
2019-06-18 14:53:00,636 ----------------------------------------------------------------------------------------------------
2019-06-18 14:53:02,337 epoch 1 - iter 0/294 - loss 88.17574310
2019-06-18 14:53:23,454 epoch 1 - iter 29/294 - loss 11.84778854
2019-06-18 14:53:44,233 epoch 1 - iter 58/294 - loss 8.29505000
2019-06-18 14:54:08,515 epoch 1 - iter 87/294 - loss 6.89873970
2019-06-18 14:54:32,620 epoch 1 - iter 116/294 - loss 6.12702840
2019-06-18 14:55:02,538 epoch 1 - iter 145/294 - loss 5.47876507
2019-06-18 14:55:23,447 epoch 1 - iter 174/294 - loss 4.97724392
2019-06-18 14:55:45,851 epoch 1 - iter 203/294 - loss 4.69156968
2019-06-18 14:56:07,959 epoch 1 - iter 232/294 - loss 4.47649900
2019-06-18 14:56:30,913 epoch 1 - iter 261/294 - loss 4.25605112
2019-06-18 14:56:53,052 epoch 1 - iter 290/294 - loss 4.03805016
2019-06-18 14:56:58,967 ----------------------------------------------------------------------------------------------------
2019-06-18 14:56:58,975 EPOCH 1 done: loss 4.0277 - lr 0.1000 - bad epochs 0
2019-06-18 14:57:16,387 DEV : loss 1.709194302558899 - score 0.2131
2019-06-18 14:58:00,931 TEST : loss 2.0527758598327637 - score 0.2309
2019-06-18 14:58:04,895 ----------------------------------------------------------------------------------------------------
2019-06-18 14:58:06,251 epoch 2 - iter 0/294 - loss 2.28403449
2019-06-18 14:58:31,067 epoch 2 - iter 29/294 - loss 2.38021900
2019-06-18 14:58:56,580 epoch 2 - iter 58/294 - loss 2.19944485
2019-06-18 14:59:17,155 epoch 2 - iter 87/294 - loss 2.13438825
2019-06-18 14:59:40,588 epoch 2 - iter 116/294 - loss 2.04427447
2019-06-18 15:00:03,391 epoch 2 - iter 145/294 - loss 2.00146707
2019-06-18 15:00:23,675 epoch 2 - iter 174/294 - loss 1.93766857
2019-06-18 15:00:46,605 epoch 2 - iter 203/294 - loss 1.88799278
2019-06-18 15:01:12,514 epoch 2 - iter 232/294 - loss 1.90256270
2019-06-18 15:01:35,203 epoch 2 - iter 261/294 - loss 1.88244418
2019-06-18 15:01:57,541 epoch 2 - iter 290/294 - loss 1.85435172
```

Training of a model