Proceedings of the LREC2020 Industry Track , pages 36–39
Language Resources and Evaluation Conference (LREC 2020), Marseille, 11–16 May 2020

NERPy: A Framework for Named Entity Recognition Experiments

Constantine Lignos
Brandeis University
415 South St., Waltham, MA 02453, USA
lignos@brandeis.edu

Abstract

Creating a high-performing sequence named entity recognition (NER) system requires a series of interconnected design decisions, including the choice of entity encoding, and for some models, the design and selection of features. In this paper, we introduce NERPy, an MIT-licensed NER framework designed to support flexible experiments in named entity recognition. We demonstrate NERPy by performing a sample experiment using the CoNLL 2003 English NER data that explores the performance of different entity encoding schemes across a wide range of training data sizes.

Keywords: Named entity recognition, Sequence models, NLP frameworks

1. Introduction

Careful experimentation is a key part of producing high-performing named entity recognition systems. However, researchers, software developers, and students interested in performing experiments in NER system design have limited tools available to them, and constructing an experimental framework for NER is a daunting task for non-experts. We address this gap with the creation of NERPy, a Python-based framework for NER experiments.

In addition to the substantial amount of work on developing state of the art systems, previous work has addressed design questions for NER systems and released software that allows for experimentation. Ratinov and Roth (2009) provide one of the most comprehensive explorations of design considerations for NER systems, finding that BILOU entity encoding (see Section 2.1) outperforms BIO in their system and that standard Viterbi decoding approaches may be slower and worse-performing in practice compared to greedy decoding with non-local features. Dernoncourt et al. (2017) introduce NeuroNER, a toolkit for neural NER designed for non-expert use, which support tight integration with the annotation process. Yang and Zhang (2018) introduce NCRF++ for performing experiments with neural NER models, and Yang et al. (2018) carefully explore the performance characteristics of models in that framework.

Many existing toolkits are primary targeted at NER researchers looking to experiment with state of the art systems using standard datasets, and are built around specific sequence model backends which provide for training and prediction. Like NeuroNER, NERPy primarily targets non-experts in the NER domain, especially software developers in industry, and is constructed to allow them to build and experiment with NER systems using standard datasets or their own data. NERPy allows users without expertise in NER to build NER systems and optimize their design by selecting the features, entity encoding, sequence model backend, and hyperparameters and then exploring the results.

NERPy's primary focus is building *non-neural* models, primarily using CRFsuite (Okazaki, 2007) for inference. Integration with neural backends is currently ongoing; see Section 4 for further discussion. Non-neural backends were prioritized first due to the existence of NCRF++ and NeuroNER, which reduced need in this area, and because one of NERPy's strengths is the ease with which users can select and customize features, which is of much lower importance for neural systems. Many of NERPy's users may lack the expertise and/or computational resources to effectively train neural NER models for custom tasks, and thus it is important to provide a framework that meets their needs. Even if a neural model is desired for the final system, a non-neural model can provide a strong baseline and may aid in the "bootstrapping" process where models are trained with small amounts of data until more is available.

While NERPy is not designed to be used in production, it can be used for prototyping, selecting the design parameters of an NER system, performing research in NER system design, and as a strong but fast baseline for other systems to beat. NERPy is designed to support comprehensive configuration of every part of the system design, including selection of hand-tuned features, use of unsupervised word representations such as word embeddings or Brown clusters, entity encoding, and the sequence model backend. This framework allows non-expert users to train NER systems from scratch using large or small data sets and compare system designs without having to implement any of the NER system themselves. NERPy enables reproducible, comprehensive experiments regarding the overall design of NER systems in any natural language for which the user can provide data.

2. Design Goals for NERPy

A complete NER system. NERPy provides a complete system for NER, including ingesting annotation, generating features (Section 2.2), encoding entities, training, predicting, scoring, and analyzing errors. It depends only on the attrs, python-crfsuite, frozendict, numpy, and regex Python packages, easing installation. NERPy uses a universal document representation that consists of sentences, tokens (which support storing properties such as part of speech and any user-defined properties), and entity mentions. Unlike spaCy (Honnibal and Montani, 2017), it supports representing nested or overlapping names, but does require that names are token-aligned, as is common for token-based sequence models and is required by the CoNLL annotation format.

Encoding	Labels					
BIO	B-MISC	B-MISC	I-MISC	O	B-PER	I-PER
IOB	I-MISC	B-MISC	I-MISC	O	I-PER	I-PER
IO	I-MISC	I-MISC	I-MISC	O	I-PER	I-PER
BILOU	U-MISC	B-MISC	L-MISC	O	B-PER	L-PER

Table 1: Entity encodings for the tokens of the string *Australian Davis Cup captain John Newcombe*.

NERPy can read CoNLL shared task[1] and OntoNotes formats and provides a CoNLL format writer. NERPy provides evaluation scripts that can score output in the NERPy or CoNLL formats and report overall and per-type entity F1 in addition to producing delimited files to support error analysis (most frequent errors, etc.). Users can call a single provided script to train and test a model, and can run experiments without any code changes simply by specifying different JSON files to configure features and select the sequence model backend and its hyperparameters.
In addition to supporting running experiments from the command line, NERPy's API can be used to perform in-memory training and prediction without needing to write any data to disk. This allows it to support rapid prototyping of applications that use named entity recognition. NERPy accepts input consisting of tokenized, sentence-segmented data organized into documents of any length, and adds named entity mentions to the document structure.

Interchangeable sequence model backends. NERPy primarily uses CRFsuite (Okazaki, 2007) to support training and decoding of sequence models. CRFsuite's supported training algorithms include L-BFGS, averaged perceptron, passive aggressive, AROW, and SGD.

Tested and easy to extend. NERPy is licensed using the MIT license. It is written in pure Python and supports Python 3.7 and up. NERPy includes a comprehensive test suite with 100% code coverage, allowing users to easily verify that any modifications do not affect correctness. It is extremely to add features or integrate a new backend other than CRFSuite, and we have tested integration with other backends during development.

2.1. Encoding Entities

To encode named entities in a sequence model, each entity must be converted into a sequence of labels. Consider this example from the CoNLL 2003 English NER annotation: [*Australian*]MISC [*Davis Cup*]MISC *captain* [*John Newcombe*]PER. NERPy supports the most popular entity encoding schemes, which would encode the five tokens of this example sentence as shown in Table 1[2], and provides robust decoding that can handle invalid label sequences produced by the sequence model. With the exception of IO, which is lossy in the case of adjacent entities, any of these encoding strategies are capable of supporting separate, adjacent entities of the same type, and there is no *a priori* reason to select one lossless encoding over another. We return to entity encodings in Section 3.1.

[1]The format varies across the four languages used in the 2002-3 CoNLL shared tasks, differing in the number of fields and the exact manner in which the *DOCSTART* document separator was used. NERPy maintains the original document boundaries and can read all fields specified in any of the four languages and adds them as attributes on each token.

[2]We can only briefly note that IOB and BIO have often been confused, and that BILOU is isomorphic to BMES/BIOES/IOBES.

```
{
  "word": {
    "window": [-2, -1, 0, 1, 2],
    "token_identity": {
      "lowercase": true
    },
    "word_shape": {},
    "is_capitalized": {},
  },
  "subword": {
    "window": [0],
    "suffix": {
      "min_length": 1,
      "max_length": 4
    }
  },
  "distributional": {
    "window": [-1, 0, 1],
    "word_vectors": {
      "scale": 2.0,
      "path": "<path to embeddings>"
    }
  }
}
```

Figure 1: Sample JSON feature configuration

2.2. Features

The currently-supported features include: the token itself (lowercased if desired), the word shape of the token (capital letters mapped to *A*, lowercase letters to *a*, digits to *0*, all other characters unchanged), whether the first character of the token is capitalized, whether all characters in the token are capitalized, whether the token is all digits, whether the token contains a digit, whether the token is all punctuation (using Unicode character categories for maximum generalization across languages), the length of the token (either as a binary feature for each length or a single continuous feature), token prefixes and suffixes, Brown cluster paths (and their prefixes), and word embeddings.
Figure 1 gives a sample feature configuration .json file that shows how features can be defined. Multiple user-named feature sets (*word*, *subword*, and *distributional* in this example) can be simultaneously used, each with differently-sized windows of application. For example, a window of `[-1, 0, 1]` will generate features for the current, previous, and next word for that feature set. Some features have mandatory or optional arguments, such as specifying whether to lowercase tokens, or the path to the file to be used for embeddings. Any arguments provided are passed as keyword arguments to the constructor of a class that generates the feature. For example, the string `token_identity` in the configuration is mapped to the `TokenIdentity` class which has a constructor with the signature `def __init__(self, *, lowercase: bool = False)`. Thus, adding a new

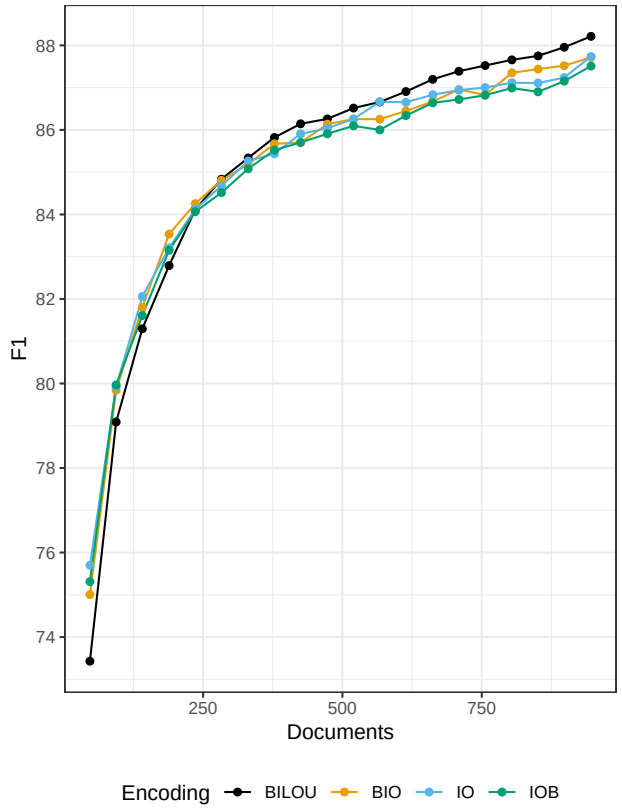

Figure 2: Entity F1 on CoNLL 2003 English test data for all encoding schemes across training data sizes.

feature is as simple as defining a new class that implements it and adding the class to the map of feature names to implementing classes. Any added feature can be configured in the same way as the built-in features by using JSON.

3. Discussion

3.1. Sample Experiment

To demonstrate NERPy's capabilities, we report results for an experiment that explores the impact of entity encoding choice at varying sizes of training data. The features used were: the token string, word shape, whether a token is capitalized, all caps, all punctuation, numeric, and/or contains a number, the length of the token (as a binary feature for each length), suffixes of length one to four, and word embeddings. All features were computed on the focus token and in a window of the two previous and two next tokens. We did not use part of speech features because we are particularly interested in performance on small amounts of training data; as part of speech taggers can produce tags that indicate proper nouns, they can effectively pass information from the part of speech training data into the NER training data, making the NER training data in effect larger than it actually is. Word embedding features were generated using fastText 300-dimensional word embeddings with subword information (Mikolov et al., 2018, wiki-news-300d-1M-subword). We selected these embeddings because they retain the casing of the original data and give some of the benefit of character-based models by using subword information.

For the purpose of easy reproducibility and comparison against other work, we use the CoNLL 2003 NER shared task English data (Tjong Kim Sang and De Meulder, 2003), reporting performance as entity F1 on the test set (the standard CoNLL NER shared task metric). To explore the effect of varying the amount of training data, we evaluated at 20 points, using 5%–100% of the training documents in 5% increments. For all training sizes except 100% (where doing so is impossible), we report the mean F1 for five random selections (without replacement) of training documents.

The training algorithm and hyperparameters were selected based on validation set performance when training on the full training set. Models were trained using L-BFGS for 100 iterations with an L1 regularization coefficient ($c1$) of 0.01, and an L2 regularization coefficient ($c2$) of 0.001. As training is implemented deterministically and initializes with zero weights, experiments from multiple random initializations are not required. We trained the model at each data size point using each label scheme. In total, this resulted in 80 experimental configurations, the cross product of four encoding schemes and 20 data sizes. As a result of the large number of configurations, the difference between encoding schemes across data sizes and features configurations is displayed in Figure 2 rather than presented in table form.

At the lowest data point, IO performs best, consistent with the notion that when there is minimal training data, a minimal encoding scheme performs best. At the highest data point, BILOU performs best, matching the findings of Ratinov and Roth (2009). The system performs similarly to many of the configurations evaluated in by Turian et al. (2010) which use Brown clusters or word embeddings. These findings are not shocking, but serve to demonstrate the capabilities of NERPy as an experimental framework.

3.2. Implementation Challenges

Working with word embeddings. NERPy originally used gensim (Řehůřek and Sojka, 2010) for loading word embeddings to generate features. However, this required loading vectors for all words in the embeddings (not just the training vocabulary), which is slow and memory-intensive. To address this, we developed the QuickVec (`https://github.com/ConstantineLignos/quickvec`) package as part of NERPy, which can instantaneously load word embeddings after a one-time conversion process, similar to Magnitude (Patel et al., 2018). It can be installed with no dependencies except numpy and can convert embeddings into its database format much faster than Magnitude, over three times faster for the 1 million-word vocabulary 300-dimensional embeddings used in the experiment described above.

Performance optimization. Due to CRFsuite's extremely fast C implementation of first-order linear chain CRF models, NER models can be trained rapidly. For example, using a standard untuned feature set (including 300-dimensional word embeddings) computed over a five-word window, training a model on the CoNLL 2003 English NER data using L-BFGS takes approximately 13 minutes, and the resulting model attains an entity of F1 of 88.21 on the test data. Substantial time was invested in optimizing NERPy's code to ensure that the code interfacing with the backend is as fast as possible, including doing line-level code profiling

to optimize frequently-called functions, such as those used in feature generation. However, especially when word embeddings are used, feature generation can take a significant amount of time (2.5 minutes for the training data in this example), as features for each position in each sentence are represented as individual dictionaries before being provided to the backend (which will typically use a more efficient representation). While it is possible to slightly improve performance while maintaining a pure Python implementation, major changes such as implementing parts of NERPy in C/C++ would come at the risk of making the code harder to extend, interact with, and install. Thus we plan to keep the current distinction of NERPy being pure Python but interfacing with backends that may be written in other languages.

4. Conclusion and Future Work

NERPy provides a flexible and accessible framework for named entity recognition that any user capable of using a command line could use to perform experiments in NER system design, and any user capable of using Python could use to create a prototype system. We have publicly released the code (`https://github.com/ConstantineLignos/nerpy`), and are in the process of completing the documentation of NERPy and QuickVec and releasing them on PyPI so that they are pip-installable.

There are many ways in which we believe that NERPy could be extended to further enable experimentation with NER system design and rapid prototyping. First, integration with a neural NER backend, such as NCRF++ and NeuroNER would enable a much broader set of experiments to be run. Integration is currently underway, and we look forward to releasing this soon. The challenge of integrating goes far beyond the software engineering required to merely "pipe" together systems; the configurations must be connected, and errors have to be handled robustly. While using another sequence model backend is relatively simple, connecting NERPy to another feature-rich NER toolkit is complex.

Second, in addition to industrial and novice-user applications, we believe NERPy could provide a reliable baseline for experimenting with NER in lower-resourced languages. Using NERPy, users can experiment with building systems before word embeddings are available, and later identifying the best ways to train their embeddings in the context of a simple, non-neural system before experimenting with more complex neural models, which are often more difficult to train due to the challenges of selecting hyperparameters using small amounts of data.

NERPy could be extended so that users could import pretrained NER models for various ontologies and languages to be used for rapid experimentation. While spaCy provides a similar function, its models are only available for a small set of languages and are chosen to reflect stable, common, ontologies as opposed to recording the research community's progress. We believe that NERPy can provide a framework for researchers to produce models for less commonly studied languages and NER tasks.

References

Dernoncourt, F., Lee, J. Y., and Szolovits, P. (2017). NeuroNER: an easy-to-use program for named-entity recognition based on neural networks. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 97–102, Copenhagen, Denmark, September. Association for Computational Linguistics.

Honnibal, M. and Montani, I. (2017). spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. *To appear*.

Mikolov, T., Grave, E., Bojanowski, P., Puhrsch, C., and Joulin, A. (2018). Advances in pre-training distributed word representations. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May. European Language Resources Association (ELRA).

Okazaki, N. (2007). Crfsuite: a fast implementation of conditional random fields (CRFs).

Patel, A., Sands, A., Callison-Burch, C., and Apidianaki, M. (2018). Magnitude: A fast, efficient universal vector embedding utility package. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 120–126, Brussels, Belgium, November. Association for Computational Linguistics.

Ratinov, L. and Roth, D. (2009). Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL-2009)*, pages 147–155, Boulder, Colorado, June. Association for Computational Linguistics.

Řehůřek, R. and Sojka, P. (2010). Software framework for topic modelling with large corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May. ELRA. `http://is.muni.cz/publication/884893/en`.

Tjong Kim Sang, E. F. and De Meulder, F. (2003). Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, pages 142–147.

Turian, J., Ratinov, L., and Bengio, Y. (2010). Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics.

Yang, J. and Zhang, Y. (2018). NCRF++: An open-source neural sequence labeling toolkit. In *Proceedings of ACL 2018, System Demonstrations*, pages 74–79, Melbourne, Australia, July. Association for Computational Linguistics.

Yang, J., Liang, S., and Zhang, Y. (2018). Design challenges and misconceptions in neural sequence labeling. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 3879–3889, Santa Fe, New Mexico, USA, August. Association for Computational Linguistics.