

# EFSA/ADAS: Named Entity Recognition

Murray Miron<sup>a,1</sup>

<sup>a</sup> Software Research, LLC

This manuscript was compiled on September 25, 2019

An end-to-end software package has been implemented for named entity recognition in the domains of animal and plant research journals; when using the scoring procedure established by the European Food Safety Authority (EFSA) and ADAS, the mean  $F_1$ -scores on the given validation sets are .841 and .803 (plant and animal, respectively). The package consists of a primary executable, two trained deep neural network models, and a number of support libraries to create additional models. These libraries can be extended or integrated into derivative or existing works as necessary.

Natural language processing | named entity recognition

The included models were trained on the datasets provided (AHAW and PLH, or animal and plant) in order to recognize the specified named entities; for practicality, the provided models are a smaller version of the architecture of the “big” models. The full size architectures are each 980 megabytes, and are available upon request. Alternatively, the provided software package can be used to recreate them: the full procedure to do so can be found below.

The smaller, included models are 25 megabytes in size (PLH) and 4 megabytes in size (AHAW). The scores for the larger models are shown in tables 3 and 5. The scores for the mini models are shown in tables 4 and 6.

The results yielded by the larger AHAW model when annotating the test set are shown visually in figure 1. The image was generated with the software package being provided; an equivalent visual annotation can be generated from any text.

Additionally, one can compare the performance of models side-by-side in this manner. See 2.

The scoring formulas themselves are shown in figure 3. Note that the scoring criteria judges an entity tag (label) as correct so long as it is correctly detected and the type is correctly assigned, even if the boundaries of the entity (the beginning and ending characters) differ from the gold standard. So long as there is an overlap and the type is correct, a prediction is deemed correct. Thus, if the full entity were “Los Angeles” and the prediction yielded by the software was “Angeles” (but no “Los”), the prediction would be considered correct if the proper label were assigned. If the proper label is not assigned, the prediction is considered entirely wrong regardless of overlaps and boundaries (which would amount to a +1 “false negative” penalty for the missing label, and a +1 “false positive” penalty for the incorrect guess; see 3).

## The software package

The software package is a total of approximately 35 megabytes in size, and is provided in the form of a gzip’d tar archive (a standard \*NIX .tar.gz compressed file). It can be used in a portable fashion and run as-is with no installation, or it can be installed to the platform using the included setup script. It is currently not available publicly or privately except as provided by Innocentive.



Fig. 1. A visualization of the results of annotating content from the test dataset.

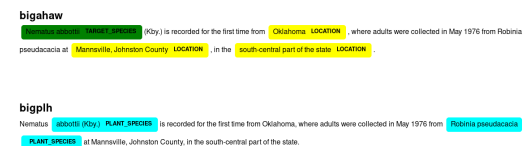


Fig. 2. Comparing performance of models side-by-side

While it can be used as an independent piece which serves only to generate the annotated output files, the provided source code can also be extended in order to make it a more integrated piece of derivative or existing works. Surprisingly little expertise is necessary unless one wants to absolutely maximize the model’s performance; a brief overview follows in later sections.

**External dependencies.** The software package requires a series 3 Python interpreter and the spaCy module, which will be taken care of automatically by the setup script if the package is installed instead of being run in a portable (e.g. from a USB drive) fashion. Note that a series 2 Python interpreter will not work!

## Model architecture

A brief overview of the neural network architectures follows; for more technical details, see the later section on replicating the reported results.

**Dictionary methods.** The accuracy of static matching alone using the EPPO database is shown in tables 1 and 2. While the included models do not utilize this method, the results are shown because the required software was written as part of an attempt to augment accuracy using the database. The functioning code remains in the provided software, and further experimentation is left to the recipient, if desired.

The reader is asked to be forgiving of the following explanation of why accuracy was reduced in the general case:

<sup>1</sup>E-mail: mmiron@srlc.org

**Table 1. EPPO Database: all English and Latin names**

Label	# Entries	True Pos	False Pos	False Neg
Plant_species	122,100	153	84	122

Note: the included models do not use this method.

**Table 2. EPPO Database: all multi word English and Latin names**

Label	# Entries	True Pos	False Pos	False Neg
Plant_species	60,580	50	16	225

Note: the included models do not use this method.

taxonomy is not this author's area of expertise.

**Binomial nomenclature** ("two term naming system") is the typical way of referring to species in research literature; in short, it dictates that the *genus* and *species* both be used for every reference. As one might imagine given the variety of life in the known world, there is often disagreement on these formal points: the specific authority one appeals to is often also provided. For example, "*Amaranthus retroflexus* L." would mean the genus *amaranthus*, the species *retroflexus*, and the acronym of the authority (Linnaeus). Additionally, the genus is always capitalized while the species is never capitalized regardless of the derivation of the name. In formal literature, it is also proper for the entire term to be italicized.

Consider, then, the following.

1. *Robinia pseudoacacia*.
2. *Robinia pseudoacacia* powdery mildew.

The former is an instance of the "Plant\_species" label.

The latter should have a "Plant\_disease\_commname" label, if the "Robinia pseudoacacia" is included at all. Also note that the EPPO database uses the spelling "pseudacacia", despite it being widely considered correct to spell it "pseudoacacia" – another example of why static matching generally introduces as many errors as it does correct predictions. Still, with task-specific rules and enough time to craft them, it can easily increase the overall accuracy of a system.

**Word embeddings.** For a given vocabulary (that is, a subset of a language), each word can be converted to a vector representation according to an algorithm; the most popular are word2vec and GloVe.

The full size models used GloVe to construct word vectors of 300 dimensions, which means each word in their vocabularies (i.e. each previously encountered and stored word) is represented by 300 real number values laid out in a vector form. In other words, for all  $W \in \text{vocabulary}$ ,  $W \in \mathbb{R}^{300}$ . This allows for word similarity to be computed; for example, the words "apple" and "orange" are similar, as are "London" and "Madrid". Word vectors are able to capture this similarity – as well as measure how different words are – because of the algorithms used to generate the vectors and the properties of matrix algebra.

The mini PLH model includes 25,168 word vectors of 200 dimensions which were built with fastText. The mini AHAW model does not include a word vector component.

**Table 3. The big AHAW (980MB) model on the validation set**

Label	True Pos	False Pos	False Neg	F <sub>1</sub> -Score
AnMethod	127	33	20	.83
Location	172	46	47	.79
Pathogenic_organisms	474	57	62	.89
Prevalence	219	107	93	.69
Target_species	408	107	73	.82
Year	41	8	11	.81
TOTAL	1441	358	306	.81

Mean .803

**Table 4. The small AHAW (4MB) model on the validation set**

Label	True Pos	False Pos	False Neg	F <sub>1</sub> -Score
AnMethod	102	41	45	.70
Location	135	41	84	.68
Pathogenic_organisms	444	67	92	.85
Prevalence	187	109	125	.61
Target_species	357	108	124	.75
Year	39	8	13	.79
TOTAL	1282	345	465	.76

Mean .732

**Table 5. The big PLH (980MB) model on the validation set**

Label	True Pos	False Pos	False Neg	F <sub>1</sub> -Score
Plant_disease	107	24	20	.83
Plant_pest	207	28	46	.85
Plant_species	234	44	41	.85
TOTAL	548	96	107	.84

Mean .841

**Table 6. The small PLH (25MB) model on the validation set**

Label	True Pos	False Pos	False Neg	F <sub>1</sub> -Score
Plant_disease	106	19	21	.84
Plant_pest	206	31	47	.84
Plant_species	211	38	64	.80
TOTAL	523	88	132	.83

Mean .829

Fig. 3.  $F_1$  Score

$$F_1 = 2 * \frac{p * r}{p + r}$$

$$p = \frac{tp}{tp + fp}$$

$$r = \frac{tp}{tp + fn}$$

$tp$  = true positives

$fp$  = false positives

$fn$  = false negatives

## Materials and Methods

All files necessary to execute the solution are included in the Python package (library dependencies are resolved and fetched by the installer); once installed, it can be executed by entering at a console `python3 -m srlcner`.

**File format.** The software package reads a file in the same format as the `ahaw_testset_text.txt` and `plh_testset_text.txt` files (i.e. a tab delimited CSV file with one item per newline-delimited row, and columns defined on the first line). The "content" column will be the input to the neural network, and all other existing columns will be copied over as key:value pairs in the "metadata" field of the output .json file. If the first line does not appear to define the column layout, then there is assumed to be only one column in the document, and that column is used as the content. This allows any arbitrary text file to be compatible.

The standard `-h` and `-help` parameters are supported, and it will function properly on any platform that can provide a Python interpreter (e.g. Windows, Linux, or even Android).

To install the package, simply decompress the archive and run at a console:

```
python3 setup.py install
```

Once the package is installed, one could, for example, generate the solution files with:

```
python3 -m srlcner -i input.txt -o output.json ahaw
```

Which will use the **ahaw** (animalia) model on `input.txt` and write to `output.json`. Then, to generate the **plh** solution set:

```
python3 -m srlcner -i input.txt -o output.json plh
```

Alternatively, one can run the software without installing it by changing to the `srlcner` directory after decompressing the archive, like so (replace `python3 -m srlcner` with `python3 srlcner.py`):

```
tar -xvf srlcner-2.1.tar.gz
cd srlcner-2.1/srlcner
python3 srlcner.py -h
```

**Visualization.** To display the entities recognized in the form of visual annotations (an example of which can be seen in 1), simply provide the `-d` parameter when executing the program. A web browser can then access the display by opening the URL `http://localhost:5000` (it is served as a web page on port 5000 of the local machine). For example:

```
python3 -m srlcner -i input.txt -d ahaw
```

This will use the **ahaw** (animalia) model on `input.txt` and display the color coded annotations (as shown in 1).

To compare two models side-by-side, simply add the second model to the command line like so:

```
python3 -m srlcner -i input.txt -d ahaw plh
```

Each row of the input document will be annotated by each model and laid out for comparison, as shown in 2.

## Replicating the results

The following procedure is quite complicated; an alternative and simpler one immediately follows.

### The full procedure.

1. Create a neural network model of random weights using spaCy (or optionally use a pretrained model, such as BERT).
2. Create word vectors appropriate for this application, or download some.
3. Initialize the neural network using the available word vectors.
4. If using spaCy, add the unique labels to the Named Entity Recognition pipe of the model.
5. Train (or fine-tune) the model using the dataset provided by the EFSA/ADAS. A dropout rate of 0.50 is recommended. Even fine-tuning can be expected to take a minimum of several hours on recent hardware.

Assuming that the validation set is a sufficient representation of the data that the model will be used to annotate, then the highest scoring checkpoint (iteration) of the network should be the one that is shipped. A full guide to determining when the model is fully trained is beyond the scope of this manuscript, as is a detailed overview of BERT (Bidirectional Encoder Representations from Transformers) and ELMo; however, ignoring issues like overfitting, one can be sure it's finished when the loss values are no longer decreasing.

**The brief procedure.** Utilize a pretrained model, e.g. `en_core_web_lg`, which is provided by spaCy that includes word vectors. The finished model will perform almost as well. If the simpler method is chosen, one need only do the following:

1. Download the `en_core_web_lg` model by entering at a command line `python3 -m spacy download en_core_web_lg`.
2. Add the appropriate labels to the named entity recognition pipe, and disable all other pipes.
3. Train the model. Eventually all preexisting labels from the `en_core_web_lg` base model will never result as predictions.

See the `train.py` script in the provided archive for an implementation of the simpler procedure, as well as further documentation.

While it is common to see training methods that employ updating weights in a model using only a subset of the training data at a time (minibatches), the primary use of such a procedure is that it allows a large net that acts on large datasets to be trained without the entirety of the dataset being loaded into memory simultaneously. Explosion AI, the makers of spaCy, suggest using this method; but for the purposes of this project, it serves only to increase the amount of time required to train the model: this author finds that the best results are most often seen with the largest possible batch size. Using the entire training set should pose no problems.

**Training the models.** It should be noted that untrained networks are generally initialized with random weights, and that an inherent property of **stochastic gradient descent** optimization algorithms are slightly random updates. Additionally, training data is almost always randomly shuffled between epochs. As such, no two training sessions will be identical unless the pseudorandom number generator is seeded with a predetermined value. Pseudorandom number generators are actually a deterministic function which only appears random, and so, if one seeds the same generator function with the same value, then the entire sequence will be identical every time.

Whether this is done or not, however, matters little for the end results; the fully trained model can be expected to differ only by an insignificant degree, so long as the same training procedure (which almost certainly incorporates random shuffling and/or warping) is utilized.

Knowing when a deep neural network is fully trained can be difficult, and they are extremely prone to "overfitting," which means that they stop producing accurate predictions for anything but the exact set of data they were trained with. As a rule of thumb: the deeper (i.e. the more layers within) the network, the more prone to overfitting it will be. To understand this, it can be helpful to view a neural network as an approximation function.

**Approximation functions.** Neural networks are, at their core, statistical approximation functions. They are trained to approximate a hidden or unknown function with the highest accuracy possible (if we already know the function that our models seek to approximate, then there is no need to train an approximation function – we can simply use the known function and have perfect accuracy!) By altering the weights connecting "neurons" in a neural network based on an optimization method (e.g. **stochastic gradient descent**), we are, in essence, encoding miniscule fragments of information in each link in the network. When taken as a whole, the model is able to give us accurate predictions because of the fragments of knowledge we have instilled into it by setting the weights during training (which is done according to the optimization method). In short, when given a specific input, the neural network provides a specific output, the value of which has been dictated by the training process as well as the specific input which we are giving it.

If one were to plot all the possible inputs to the neural network on the same graph as all their corresponding outputs (ignoring the fact that our minds are likely to balk at the attempt to picture thousands of dimensions), one hopes that it would be sufficiently similar to the inputs and outputs, when graphed, of whatever "hidden" function we are trying to approximate.

It is a necessity, then, that the training dataset is sufficiently representative of the domain of interest; if it is not, the predictions provided by the network are poor indeed. This is the reason that data is so vital: if the network is only trained using a small subset of the entire domain of interest, then it has no possible means of providing accurate predictions when faced with different input.

Additionally, if the training data is not sufficiently disparate, then overfitting is certain to take place. In fact, when dealing with image data, an almost universal method of combatting overfitting is to randomly warp the input data at each training epoch; it is far from intuitive at first glance, but when the simple act of warping the input image is combined with techniques such as L2 regularization and random dropout, it can substantially improve the results and effectively combat the issue of overfitting.

More detailed descriptions of the fundamental pieces of modern neural networks, e.g. rectified linear units and their mathematical properties, the behaviors of various methods of measuring loss and optimizing networks for them, etc., are beyond the scope of this manuscript.

**License.** All rights not otherwise specified in the existing agreements with EFSA, ADAS, and/or Innocentive, are reserved.

**ACKNOWLEDGMENTS.** Software Research, LLC gratefully acknowledges the use of the datasets provided by the European Food Safety Authority and ADAS.