CS388 Mini 1: Classification for Person Name Detection

Due date: September 10, 2019 at 11:59pm CT

Collaboration As stated in the syllabus, you are free to discuss the homework assignments with other students and work towards solutions together. However, all of the code you write must be your own! Your writeup must be your own as well. **Please list your collaborators at the top of your written submission.**

Goal In this project you'll implement a simple classifier that can determine whether a token is part of a person's name. This will reinforce the basics of classification (which you should have seen before) and teach you the basics of large-scale machine learning with sparse feature vectors, including techniques for feature extraction, feature indexing, optimization, etc.

This project is designed to be completed from scratch using only the utilities provided. You are free to use `scikit-learn` or other classification libraries if you are comfortable doing so. However, Project 1 will heavily rely on the provided framework code, so familiarizing it with yourself now and learning the concepts it entails is a good time investment!

Background

The data used in this project is derived from the CoNLL 2003 Shared Task on Named Entity Recognition (Tjong Kim Sang and De Meulder, 2003). That data labels four types of named entities: person, organization, location, and miscellaneous. We just focus on identifying instances of the person label in isolation in this project. An example of the data you'll be working with is given below:

```
Defending champion Scott Hoch shot a three-under 68 .
    0         0      1     1    0  0      0       0 0
```

The data is *tokenized*: punctuation marks like periods and commas are broken out with spaces and *'s* and *'nt* are separated from the words that come before them. Each token is labeled with 0 (not part of a person mention) or 1 (part of a person mention).

NER systems of the kind you'll be building in Project 1 can typically identify named entity chunks like this with F_1 scores of over 90%.[1] Given this simplified form of the task, we'll be aiming to achieve per-token F_1 values of around 90 in our system here as well.

Getting Started

Download the data and code from Canvas. You will need Python 3 and numpy.[2] Try running

```
python classifier_main.py
```

This will run a classifier which simply outputs the most common label for each token, defaulting to 0 in the case of ties. This gets 96.6% precision, 64.1% recall, and 77.1 F_1 on the development set, where these values are computed on a token-level basis. Next, try running

```
python classifier_main.py --model CLASSIFIER
```

[1] Tasks like this where the negative class is so common (no named entity / non-person) are almost universally evaluated in terms of F_1, since raw accuracy scores of returning the negative class all the time would be over 90%.

[2] If you don't have `numpy`, see `https://www.scipy.org/install.html`. You may find anaconda useful, and anaconda virtual environments are a good way of handling neural network packages that we'll be encountering later in the class.

This will crash with an error message. You have to implement training and inference in your classifier for this to work.

Data `eng.train` is the training set, `eng.testa` is the development set, and `eng.testb.blind` is a blind test set you will submit results on. These are adapted from the standard datasets used in the CoNLL 2003 NER task.

Code We provide:

`classifier_main`: Contains the `PersonExample` class, the training/evaluation harness, an implementation of a `CountBasedPersonClassifier`, and a skeleton for the classifier you will build.

`nerdata.py`: Utilities for reading NER data, evaluation code, and other assorted functions. You'll be using these data structures more heavily in Project 1, but should not need to really look at this file here.

`utils.py`: Some useful utilities, all of which are optional to use:

`Indexer`: tracks a mapping between n objects and the integers 0 through $n-1$; this is useful for mapping features or labels to indices that will be used in feature vectors or dynamic programs

`Beam`: data structured used in beam search, unused in this project.

`score_indexed_features`: lets you score a list of feature indices using a weight vector.

`maybe_add_feature`: implements some logic for indexing a feature and adding it to a list, which is useful for scrolling through data and producing features.

`optimizers.py`: Three optimizer classes implementing SGD, unregularized Adagrad, and L1-regularized Adagrad. These wrap the weight vector, exposing `access` and `score` methods to use it, and are updated by `apply_gradient_update`, which takes as input a `Counter` of feature values for this gradient as well as the size of the batch the gradient was computed on. Familiarize yourself with the SGD trainer first (which is very simple) and then consider swapping it out for one of the others. Note that these are all optimized for sparse updates, which is nontrivial to do in the case of Adagrad.

Implementing a Classifier

You need to implement two main things: the `train_classifier` and `PersonClassifier.predict` functions. **Note that you may need to implement other functions to complete this as well.** Conceptually, a few stages are necessary:

Feature Extraction First and foremost, you will need a way of mapping from the `PersonExample` objects to vectors, a process called feature extraction or featurization. In this project, we recommend you use a suite of "indicator" features to represent the examples: sparse features that take the value 0 or 1 depending on whether that feature is present or absent in this particular example. For example, one feature type you'll probably want to use is word indicator features: you can create features like this by indexing strings of the form "CurrWord=[word]" to make them to indices in the weight vector space. Explore other features as well!

The `Indexer` class is designed to help you with this process. It maintains a bidirectional mapping from strings to integers. While there are many ways to represent features, forming strings out of them and then indexing those strings is pretty effective.

Training You need to make several decisions in implementing training: (1) How will you extract features from the text examples? (2) What classification method do you want to use? (3) What optimizer do you want to use? Roughly, your training method should look like the following: loop over epochs, loop over examples, extract features, compute the gradient of the weights for that example, make the gradient update. Feature extraction can be slow and it is wasteful to do it separately for each epoch, so consider caching the extracted features across epochs. Also note that when using sparse features, you may have hundreds of thousands or even millions of features, but gradients on each example will usually only involve a few tens of features. You should use a `Counter` for a sparse representation of these.

Prediction The prediction step is essentially about taking the byproducts of training (known features and their weights) and using them to make a classification decision. Our framework code feeds a weight vector and a `Indexer` object to the `PersonClassifier`. You may add additional arguments and are not required to use these.

Additional Implementation Tips

- If your code is too slow, try (a) making use of the feature cache to reduce computation and (b) exploiting sparsity in the gradients (`Counter` is a good class for maintaining sparse maps). Run your code with `python -m cProfile classifier_main.py [args]` to print a profile and help identify bottlenecks.
- Implement things in baby steps! Check what your prediction looks like on a single example with a dummy weight vector. Then make sure that your model and optimizer can fit a very small training set; be sure to print out the objective of whatever training method you're using and check that this goes up/down appropriately, along with train accuracy. Then work on scaling things up to more data and optimizing for development performance.

Expected Performance Our reference implementation with only indicator features of the current word identity gets 77.1 F_1 (same as the baseline). Our reference implementation uses 7 different types of features and achieves 91.5 F_1 on the development set. **To get full credit on the assignment, you should get a score of at least 91 F_1 on the development set.** Implementations close to this will get nearly full credit, especially if you provide evidence of debugging and experimentation.

Submission and Grading

You should submit on Canvas:

1. Your code
2. Your classifier's output on the `testb` blind test set, formatted as word[tab]label with one word per line and blank line in between sentences.

3. A report of no more than 1 page. Your report should list your collaborators, **briefly** restate the core problem and what you are doing, describe relevant details of your implementation (including classification method used, features, optimizers, etc.), and present a table of results. Your report should be written in the tone and style of an ACL/NeurIPS conference paper. Any LaTeX format with reasonably small (1" margins) is fine, including the ACL style files[3] or any other one- or two-column format with equivalent density.

Slip Days Slip days may be used on this assignment. See the syllabus for details about the slip day policy.

References

Erik F. Tjong Kim Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition. In *Proceedings of the Conference on Natural Language Learning (CoNLL)*.

[3] Available at `http://acl2017.org/calls/papers/`