cs134-pa1

September 8, 2020

1 Train a Logistic Regression model for text classification from scratch

Due September 22, 2020

1.1 Overview

`Logistic Regression` (aka `Maximum Entropy`) has been one of the workhorses in natural language processing. It has also been used very widely in other fields. The model has many strengths. It is effective when you have a large number of features as it handles the correlation between features well. In addition, the time and space complexity for the model is not too taxing. All of these reasons make the Logistic Regression model a very versatile classifier for many NLP tasks. It is also a good starting point to understand more complicated neural network methods.

In this assignment, we are going to use the Logistic Regression model on 2 different text classification tasks. One for name classification, and another for discourse relation classification. The name classification is a toy problem that you can use to test and debug your code.

1.2 Getting ready

Download the starter code along with the datasets using the link on latte. We highly recommend reviewing the slides and the textbook before starting implementing.

1.3 Task-Datasets

- You should aim to complete all functions with `TODO` in its docstring
- You are more than welcome to substantially change the starter code and/or add your new functions

1.3.1 Names Classification - Names Corpus

- located in `data/names`
- Your task is to use the first and last character of each name to predict whether the name is male or female.
- The dataset is loaded and prepared for you, though you should come up with a `feature vector` for each data instance.
- Use this for starting out and debugging your model, before moving on to the next task.

1.3.2 Discourse Relation Prediction - Penn Discourse TreeBank (PDTB)

- located in `data/pdtb`
- A single data instance is a possibly multi-line sentence, split into spans of (1) Arg1 (2) optional Connective (3) Arg2.
- Your task is to predict a discourse relation (also known as **sense**) between Arg1 and Arg2.
- Optional connectives (such as `and` and `but`), if present, often provide helpful hints during prediction
- The complete list of labels are given to you as `SENSES` in `corpus.py` See below for details.
- PDTB comes with pre-defined dataset splits of training, dev and test sets
- By default, the features include separate unigrams for Arg1, Arg2 and Connectives. Use these to output a single `feature vector`.

```
In [1]: # complete list of labels (senses) for PDTB: 21 in total
        SENSES = [
            'Temporal',
            'Temporal.Asynchronous',
            'Temporal.Asynchronous.Precedence',
            'Temporal.Asynchronous.Succession',
            'Temporal.Synchrony',
            'Contingency',
            'Contingency.Cause',
            'Contingency.Cause.Reason',
            'Contingency.Cause.Result',
            'Contingency.Condition',
            'Comparison',
            'Comparison.Contrast',
            'Comparison.Concession',
            'Expansion',
            'Expansion.Conjunction',
            'Expansion.Instantiation',
            'Expansion.Restatement',
            'Expansion.Alternative',
            'Expansion.Alternative.Chosen alternative', # `alternative` can be safely ignored
            'Expansion.Exception',
            'EntRel'
        ]
```

```
In [2]: def to_level(sense: str, level: int = 2) -> str:
            """converts a sense in string to a desired level

            There are 3 sense levels in PDTB:
            Level 1 senses are the single-word senses like `Temporal` and `Contingency`.
            Level 2 senses add an additional sub-level sense on top of Level 1 senses,
              as in `Expansion.Exception`
            Level 3 senses adds yet another sub-level sense, as in
              `Temporal.Asynchronous.Precedence`.

            This function is used to ensure that all senses do not exceed the desired
```

```
    sense level provided as the argument `level`. For example,

    >>> to_level('Expansion.Restatement', level=1)
    'Expansion'
    >>> to_level('Temporal.Asynchronous.Succession', level=2)
    'Temporal.Asynchronous'

    When the input sense has a lower sense level than the desired sense level,
    this function will retain the original sense string. For example,

    >>> to_level('Expansion', level=2)
    'Expansion'
    >>> to_level('Comparison.Contrast', level=3)
    'Comparison.Contrast'

    Args:
    sense (str): a sense as given in a `relaions.json` file
    level (int): a desired sense level

    Returns:
    str: a sense below or at the desired sense level
    """
    s_split = sense.split(".")
    s_join = ".".join(s_split[:level])
    return s_join
```


Regarding the labels (senses) of PDTB There are 3 sense levels in PDTB:

1. Level 1 senses are the single-word senses like `Temporal` and `Contingency`
2. Level 2 senses add an additional sub-level sense on top of Level 1 senses, as in `Expansion.Exception`
3. Level 3 senses adds yet another sub-level sense, as in `Temporal.Asynchronous.Precedence`

By default, we will convert all senses to Level 2, meaning that all Level 3 senses will be truncated. For example,

```
In [3]: to_level('Temporal.Asynchronous.Precedence', level=2)

Out[3]: 'Temporal.Asynchronous'

In [4]: to_level('Expansion.Alternative.Chosen alternative', level=2)

Out[4]: 'Expansion.Alternative'
```

All Level 1 senses will retain the original sense, For example,

```
In [5]: to_level('Temporal', level=2)

Out[5]: 'Temporal'
```

So, in practice, the reduced label space consists of 16 labels below.

```
In [6]: level_2_senses = list(set([to_level(x, level=2) for x in SENSES]))
        level_2_senses

Out[6]: ['Contingency.Condition',
         'Expansion.Alternative',
         'Comparison',
         'Expansion.Exception',
         'Expansion.Conjunction',
         'Contingency.Cause',
         'Expansion',
         'Expansion.Instantiation',
         'Temporal.Asynchronous',
         'Expansion.Restatement',
         'EntRel',
         'Comparison.Concession',
         'Comparison.Contrast',
         'Contingency',
         'Temporal.Synchrony',
         'Temporal']

In [7]: len(level_2_senses)

Out[7]: 16
```


We nevertheless provide the complete list of class labels, in case you wish to train your model using Level 1 senses or Level 3 senses.

1.4 What needs to be done

A Logistic Regression model has a lot of moving parts. The list below guides you through what needs to be done. The three things you need to focus on are: **Representation**, **Learning**, and **Inference**.

1.4.1 Representation

- Choose the feature set.
  - Start with the feature set that is small or the learning algorithm will take a long time to run and this makes the debugging process difficult.
  - You can ramp up the number and variety of features when your code is thoroughly tested.
  - Basic feature set is provided as part of the starter code (see **Task-Datasets** above).
- Choose the data structure that holds the features.
  - We recommend sparse feature vectors (as numpy ndarray).
  - Regardless of your choice, cache the features internally within each `Document` object as the algorithm is iterative. Featurization should be done only once.

- Choose the data structures that hold the parameters (weights).
  - We recommend using a $k \times p$ matrix where k is the number of labels, and p is the number of linguistic features. This is equivalent to use a vector of length $k \times p$

1.4.2 Learning

- Compute the negative log-likelihood function given a minibatch of data.
  - You will need this function to track progress of parameter fitting.
- Compute the gradient with respect to the parameters.
  - You will need the gradient for updating the parameters given a minibatch of data.
- Implement the mini-batch gradient descent algorithm to train the model to obtain the best parameters.

1.4.3 Classification/Inference.

- Run prediction for a single data instance

1.5 Experiments

In addition to implementing the mini-batch gradient descent algorithm for the Logistic Regression model, you are asked to do the following experiments to better understand the behavior of the model. For all three experiments, use the discourse relation dataset as it is a more realistic task.

Experiment 1 -- Training set size:

Does the size of the training set really matter? The mantra of machine learning tells us that the bigger the training set the better the performance. We will investigate how true this is.

In this experiment, fix the feature set to something reasonable and fix the dev set and the test set. Vary the size of the training set $\{1000, 5000, 10000,$ and all$\}$ and compare the (peak) accuracy from each training set size. Make a plot of size vs accuracy. Analyze and write up a short paragraph on what you learn or observe from this experiment.

Experiment 2--- Minibatch size:

Why are we allowed to use mini-batch instead of the whole training set when updating the parameters? This is indeed the dark art of this optimization algorithm, which works well for many complicated models, including neural networks. Computing gradient is always expensive, so we want to know how much we gain from each gradient computation.

In this experiment, try minibatch sizes $\{1, 10, 50, 100, 500\}$, using the best training size from Experiment 1. For each mini-batch size, plot the number of datapoints that you compute the gradient for (x-axis) against the accuracy of the development set (y-axis). Analyze and write up a short paragraph on what you learn or observe from this experiment.

Experiment 3 -- Hyperparameter tuning:

Try different values of $\lambda = \{0.1, 0.5, 1.0, 10.0\}$ for $L2$ reguarlization and observe its effect on the accuracy of the model against the development set. Make a plot of lambda value vs accuracy on the development set. Write a short paragraph summarizing what you have oberved from this experiment.

As you are doing more experiments, the number of experimental settings starts to multiply. Use your best settings from your Experiment 1 and your Experiment 2 for the tuning of the $L2$ regularization parameters. It's not advisable to vary more than one experimental variable at a

time, and that'll make it hard to interpret your results. You can set up a grid search procedure to do this experiment automatically without manual intervention.

Experiment 4 -- Feature Engineering (Extra credit for creative and effective features): In addition to bag-of-words features, experiment with additional features (bigrams, trigrams, etc.) to push up the performance of the model as much as you can. The addition of new features should be driven by error analysis. This process is similar to the machine learning process itself, only that it involves actual humans looking at the the errors made of the machine learning model and trying to come up with new features to fix or reduce those errors. Briefly describe what new features you have tried if they are useful.

1.6 Submission

Submit the following on Latte:

1.6.1 All your code.

But don't include the datasets as we already have those.

1.6.2 Report.

Please include the following sections in your report: 1. A brief explanation of your code structure

2. How to run your code, and what output to expect
3. Experimental settings (Explain clearly what feature set is being used and how you set up mini-batch gradient descent because there can be quite a bit of variations.)
4. Experimental results

Please keep this report no more than two pages single-spaced including graphs.

1.7 More on Mini-batch Gradient Descent

In this assignment, we will train Logistic Regression models using mini-batch gradient descent. Gradient descent learns the parameter by iterative updates given a chunk of data and its gradient.

If a chunk of data is the entire training set, we call it batch gradient descent.

```
In [ ]: while not converged:
            gradient = compute_gradient(parameters, training_set)
            parameters -= gradient * learning_rate
```

Batch gradient descent is much slower. The gradient from the entire dataset needs to be computed for each update. This is usually not necessary. Computing gradient from a smaller subset of the data at a time usually gives the same results if done repeatedly.

If a subset of the training set is used to compute the gradient, we call it mini-batch gradient descent. This approximates the gradient of batch gradient descent.

```
In [ ]: while not converged:
            minibatches = chop_up(training_set)
            for minibatch in minibatches:
                gradient = compute_gradient(parameters, minibatch)
                parameters -= gradient * learning_rate
```

If a chunk of data is just one instance from the training set, we call it stochastic gradient descent (SGD). Each update only requires the computation of the gradient of one data instance.

```
In [ ]: while not converged:
            for datapoint in training_set:
                gradient = compute_gradient(parameters, datapoint)
                parameters -= gradient * learning_rate
```

Practical issues with mini-batch gradient descent

- How should I initialize the parameters at the first iteration?

  Set them all to zero. This is generally not advisable for more complicated models. But for the Logistic Regression model, zero initialization works perfectly.

- How do I introduce the bias term?

  Include a feature that fires in ALL data instances. And treat it as a normal feature and proceed as usual.

- Why do the posterior P(Y | X) become NaN?

  It is very likely that you exponentiate some big number and divide by the same amount i.e. if unnoramlized_score is a vector of unnormalized scores (the sum of lambdas), then:

  ```
  posterior = exp(unnormalized_score) / sum(exp(unnormalized_score))
  ```

  This is no good. We have to get around by using some math tricks:

  ```
  posterior = exp(unnormalized_score - scipy.misc.logsumexp(unnormalized_score))
  ```

  If this confuses you or you are not sure why this is correct, think about it more or ask the TAs. But we are quite confident that you will need to use the logsumexp function.

- How do you know that it converges?

  It is extremely difficult to know. If you stop too early, the model has not reached its peak yet i.e. *underfitting*. If you stop too late, the model will fit too well to the training set and not generalize to the unseen data i.e. *overfitting*. But there are multiple ways to guess the convergence. We suggest this method called *early stopping*.

  Every once in a while evaluate the model on the development set during gradient descent.

  - If the performance is better than last evaluation, then save this set of parameters and keep going for a little more.
  - If the performance stops going up after a few updates, stop and use the last saved parameters. (How many is a few? Up to you)

- How often should I run evaluation on the dev set during training?

  Up to you. It is actually OK to run the evaluation on the dev at every update you make to the parameters.

- How do I know that my implementation is correct?

  Look at the average negative log-likelihood. It should keep going down monotonically i.e. at every single update. You should also see that the gradient should get closer and closer to zero.