SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.040 Natural Language Processing, Summer 2020

Design Project

Due 21 August 2020, 5pm

Please start this project early.

Instructions

Please read the following instructions carefully before you start this project:

- This is a group project. You are allowed to form groups in any way you like, but each group must consist of either 2 or 3 people. Please submit your group information to eDimension as soon as possible if you have not yet done so (the due was Friday 26 June 2020).
- You are strictly NOT allowed to use any external resources or machine learning packages (except numpy) unless you are explicitly instructed to do so (e.g., Part 4 (i), Part 6).
- Each group should submit code together with a report summarizing your work, and give clear instructions on how to run your code. Please also submit your system's outputs. Your output should be in the same column format as that of the training set.
- You are given 8 weeks to work on the project. We understand you are busy in this final term, so Week 11 will be reserved as the "Final Project Week" for you to work on the project (i.e., there will be no class for that week). Please plan and budget your time well.
- Please use Python to complete this project.
- Please install `conlleval` package and use the `conlleval.evaluate` function for later evaluations.

Project Summary

Welcome to the design project for our very first natural language processing (NLP) course offered at SUTD!

As we mentioned, this design project will be a continuation of the design project of our earlier machine learning (ML) course. In that design project, we looked at the problem of building a sequence labeling system using the hidden Markov model (HMM). As we have discussed in class, the HMM model is a generative model and it comes with several limitations. The conditional random fields (CRF) model would be able to address some of its limitations. In this design project, we would like to design our sequence labeling model for informal texts using the CRF model that we have learned in class. We will be able to see how this discriminative approach is able to empirically improve the effectiveness of sequence labeling

in this project over the HMM model that we learned in the previous course. We will be following a similar setup used in the ML course. Specifically, we will focus on building a *named entity recognition* system.

The data for this project are in the two folders `partial` and `full`. For each dataset, we provide a labelled training set `train`, an unlabelled development set `dev.in`, and a labelled development set `dev.out`. The labelled data has the format of one token per line with token and tag separated by a space and a single empty line that separates sentences.

The format for the `partial` dataset can be something like the following, which is the same as what we used in the ML project and is also discussed in our class:

```
Authorities O
ordered O
residents O
of O
a O
section O
of O
Oklahoma B-geo
City I-geo
to O
evacuate O
Sunday B-tim
. O
```

where labels such as `B-XXX`, `I-XXX` are used to indicate **B**eginning and the **I**nside of the entities. `O` is used to indicate the **O**utside of any entity. The format for the `full` dataset has an extra column, representing `Part-of-Speech` tags.

```
Authorities NNS O
ordered VBD O
residents NNS O
of IN O
a DT O
section NN O
of IN O
Oklahoma NNP B-geo
City NNP I-geo
to TO O
evacuate VB O
Sunday NNP B-tim
. . O
```

You will make use of the extra POS tag information to build a more powerful sequence labeling system in Part 5, 6. Overall, our goal is to build sequence labeling systems from such training data and then use the system to predict tag sequences for new sentences.

Note that the data is already processed into the above format, and please do not perform any additional pre-processing to the data for Part 1 to Part 5.

1 Part 1 *(10 points)*

Consider an input word sequence $\boldsymbol{x} = x_1, \ldots, x_n$ and an output tag sequence $\boldsymbol{y} = y_1, \ldots, y_n$. Recall that the HMM is defined as follows:

$$p(x_1, \ldots, x_n, y_1, \ldots, y_n) = \prod_{i=1}^{n+1} q(y_i|y_{i-1}) \cdot \prod_{i=1}^{n} e(x_i|y_i) \tag{1}$$

where $y_0 = \texttt{START}$ and $y_{n+1} = \texttt{STOP}$. Here q are transition probabilities, and e are emission probabilities. In this project, x's are the natural language words, and y's are the tags (such as `O`, `B-XXX`).

(i) Similar to what we did in the ML project, first write a function to estimate the following emission probabilities based on the training set:

$$e(x|y) = \frac{\texttt{Count}(y \to x)}{\texttt{Count}(y)}$$

Now, each (x, y) pair is associated with a weight, which can be defined as the logarithm of the about emission probability: $\log e(x|y)$.

On the other hand, for each (x, y) pair that appears in the training set, we can create a new feature string $str(x, y)$ by concatenating the following 4 strings:

- the string "emission:"
- string form of y
- the string "+"
- string form of x

For example, if $x =$ *John*, and $y = \texttt{B-per}$, then we will create a string "emission:B-per+John".

Create a dictionary f, inside which we store the following key-value mappings:

$$str(x, y) \Rightarrow \log e(x|y)$$

where $str(x, y)$ is the key and $\log e(x|y)$ is the value.

(5 points)

(ii) Again, similarly, write a function to estimate the following transition probabilities:

$$q(y_i|y_{i-1}) = \frac{\texttt{Count}(y_{i-1}, y_i)}{\texttt{Count}(y_{i-1})}$$

Please make sure the following special cases are also considered: $q(\texttt{STOP}|y_n)$ and $q(y_1|\texttt{START})$.

Now, for each (y_{i-1}, y_i) pair that appears in the training set, write a function to create a new feature string $str(y_{i-1}, y_i)$ by concatenating the following 4 strings:

- the string "transition:"
- string form of y_{i-1}

- the string "+"
- string form of y_i

For example, if $y_{i-1} = \texttt{B-XXX}$, and $y_i = \texttt{I-XXX}$, then we will create a string "transition:B-XXX+I-XXX".

Also store the following key-value mappings into the same dictionary f we created above:

$$str(y_{i-1}, y_i) \Rightarrow \log q(y_i|y_{i-1})$$

where $str(y_{i-1}, y_i)$ is the key and $\log q(y_i|y_{i-1})$ is the value.

(5 points)

The resulting dictionary would look something like the following:

Feature	**Weight**
...	...
emission:B-per+John	-1.23
...	...
transition:B-geo+I-geo	-2.34
...	...

2 Part 2 *(15 points)*

Recall that the CRF model discussed in class is defined as follows:

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{\exp(\mathbf{w} \cdot \mathbf{f}(\boldsymbol{x}, \boldsymbol{y}))}{\sum_{\boldsymbol{y'}} \exp(\mathbf{w} \cdot \mathbf{f}(\boldsymbol{x}, \boldsymbol{y'}))} \tag{2}$$

where $\boldsymbol{x} = x_1, \ldots, x_n$ is the complete input word sentence, and $\boldsymbol{y} = y_1, \ldots, y_n$ is the complete output label sequence.

(i) One key component used in the model is the feature function $\mathbf{f}$. Now consider the two types of features as discussed in Part 1. If we use the above-mentioned features and their associated weights, we will be able to calculate a score for a particular output sequence $\boldsymbol{y}$.

Write a function to calculate the score for a given pair of input and output sequence pair $(\boldsymbol{x}, \boldsymbol{y})$, based on the above-mentioned features and weights used in Part 1. Note that the score is defined as follows:

$$\mathbf{w} \cdot \mathbf{f}(\boldsymbol{x}, \boldsymbol{y}) = \sum_j w_j f_j(\boldsymbol{x}, \boldsymbol{y}) \tag{3}$$

where $f_j(\boldsymbol{x}, \boldsymbol{y})$ is a function that returns the number of times that the j-th feature in the above-defined dictionary appears in $(\boldsymbol{x}, \boldsymbol{y})$, and w_j is the weight of the j-th feature.

(5 points)

(ii) Now, use the above features and weights to perform decoding using the Viterbi algorithm to find the most probable output sequence $\boldsymbol{y}^*$ for an input sequence $\boldsymbol{x}$:

$$\boldsymbol{y}^* = \arg\max_{\boldsymbol{y}} p(\boldsymbol{y} \mid \boldsymbol{x}) = \arg\max_{\boldsymbol{y}} \mathbf{w} \cdot \mathbf{f}(\boldsymbol{x}, \boldsymbol{y}) \tag{4}$$

You may refer to the notes on CRF uploaded to eDimension if you would like to better understand how to use the Viterbi algorithm for this.

Apply the Viterbi algorithm to the dataset `partial/dev.in` , and save the outputs into `partial/dev.p2.out`. You can evaluate your model's predictions using `conlleval.evaluate` function.

(10 points)

3 Part 3 *(20 points)*

In Part 3 and 4, we will train the weights of the feature functions from scratch. The loss function for CRF that involves a particular training set can be defined as follows:

$$-\sum_i \log p(\boldsymbol{y}_i \mid \boldsymbol{x}_i) = -\sum_i \left[\mathbf{w} \cdot \mathbf{f}(\boldsymbol{x}_i, \boldsymbol{y}_i) - \log \sum_{\boldsymbol{y'}} \exp(\mathbf{w} \cdot \mathbf{f}(\boldsymbol{x}_i, \boldsymbol{y'})) \right] \tag{5}$$

where $\boldsymbol{x}_i$ is the i-th input sequence (sentence), and $\boldsymbol{y}_i$ is the i-th output sequence (tag sequence) from the training set.

Calculation of the first term inside the brackets is simple as you just need to call the function implemented in Part 2 (i). However, calculation of the second term requires the forward algorithm.

(i) Write a function to calculate the value of the loss defined above based on the forward algorithm. The function should take in as input a particular feature weight vector $\mathbf{w}$.

You may refer to the notes on CRF uploaded to eDimension if you would like to better understand how to implement the forward algorithm for this.

(10 points)

(ii) Implement the backward algorithm that takes in as input a particular feature weight vector $\mathbf{w}$. Next, implement a function to calculate the gradients (a gradient vector) based on the forward and backward scores for each feature. Store the gradients for all the features into a dictionary. The dictionary should look something like the following:

Feature	**Gradient**
...	...
emission:I-PER+John	0.123
...	...
transition:B-MISC+I-MISC	−3.45
...	...

You may refer to the notes on CRF uploaded to eDimension if you would like to better understand how to implement the backward algorithm and how to calculate the gradients for this.

Hints: How to know if your code has any bug? There are several ways for you to check if your code potentially has bugs. One idea is to calculate the gradient numerically. What you can do is to first calculate the value of the objective function v_1 based on a particular set of feature weights, and then add a very small value Δ to the weight of one particular feature, calculate the objective function again (after the weight for that particular feature is changed). Let's call the objective value v_2. Next, take the difference between the two objective values, and divide it by Δ. In other words, calculate $(v_2 - v_1)/\Delta$. Check to see if this value (which is a gradient calculated numerically) is very close to the gradient of that feature calculated by your code.

(10 points)

4 Part 4 *(10 points)*

(i) Now, with the loss and the gradients, we will be able to perform learning. In practice, we also use a L2 regularization term to control overfitting. With the L2 regularization term, our new loss function is as follows:

$$loss = -\sum_i \log p(\boldsymbol{y}_i \mid \boldsymbol{x}_i) + \eta \sum_j w_j^2 \tag{6}$$

Where w_j is the weight of the j-th feature, η is the coefficient of the L2 regularization term.

Set the L2 regularization coefficient η to 0.1 here.

Modify the functions for calculating the loss and gradients above, so that your functions can return the new loss as well as the new gradients with the additional L2 regularization term.

With the gradients (and loss), we can use gradient descent for learning, but we can also use some more advanced optimization algorithms. In this project, we will learn to use the L-BFGS algorithm that is popular for fast parameter estimation in machine learning. We will use the L-BFGS implementation `fmin_l_bfgs_b` of the `Scipy` package. Here comes how to make use of it:

```
from scipy.optimize import fmin_l_bfgs_b
def callbackF(w):
    '''
    This function will only be called by "fmin_l_bfgs_b"
    Arg:
        w: weights, numpy array
    '''
    loss = get_loss_grad(w)[0]
    print('Loss:{0:.4f}'.format(loss))

def get_loss_grad(w):
    '''
    This function will only be called by "fmin_l_bfgs_b"
    Arg:
        w: weights, numpy array
    Returns:
```

```
            loss: loss, float
            grads: gradients, numpy array
        '''
        # to be completed by you,
        # based on the modified loss and gradients,
        # with L2 regularization included
        return loss, grads

    result = fmin_l_bfgs_b(get_loss_grad, init_w,
     pgtol=0.01, callback=callbackF)
```

where `get_loss_grad` is the function that returns both the loss and gradients, `init_w` is the initial guess of the weights, `pgtol` is a hyperparameter used for deciding when to stop the optimization, and `callbackF` is a function that prints intermediate results for each iteration.

In `result`, which is finally returned by the function, there are the following 3 elements: the optimal weights, the final loss, and a dictionary including gradients, warning information, etc.

Set the initial guess of the weights to 0's, and set `pgtol` as 0.01.

Run the L-BFGS algorithm to learn the weights based on the training sets.

Report the intermediate and final loss in the training process.

(5 points)

(ii) After learning is finished, use the learned weights to perform decoding on the development sets.

We will use the learned weights and apply the Viterbi algorithm implemented in Part 2 (ii) to each input sentence in the development set. Save the results in `partial/dev.p4.out`. You can evaluate your model's predictions using `conlleval.evaluate` function.

Hints: You may make use of some other tools such as CRF++ (https://taku910.github.io/crfpp/) to understand what range of f-measures you may get for the above-defined features. If you use CRF++, the feature template should be defined as follows:

```
U01:%x[0,0]
B
```

You may read the instructions on that page and download and install the toolkit locally to your machine. However the results obtained by CRF++ may not be exactly the same as your implementation's for the following reasons: 1) the convergence criterion may be different, 2) the choice of the L2 regularization may be different, 3) there is a known bug in CRF++ (if you are interested, try to locate the bug :)). Please note that it is not required for you to include such comparisons in the report.

(5 points)

5 Part 5 *(25 points)*

The advantage of discriminative models over generative models is that discriminative models don't need to model the input distribution and thus can easily make use of more diverse input features. So far we've only

considered transition features and word identity emission features in our CRF. However, other word features such as part-of-speech (POS) tags can be incorporated into to emission features as well, e.g. "emission: B-geo+NNS".

(i) Please add POS tag emission features to your CRF based on `full/train` dataset. Apply the Viterbi algorithm to the dataset `full/dev.in` , and save the outputs into `full/dev.p5.CRF.f3.out`. You can evaluate your model's predictions using `conlleval.evaluate` function.

(5 points)

(ii) Now we have three kinds of features in our CRF, transition features, word identity emission features, POS tag emission features. In fact, we can also combine emission feature "emission:$y_i + x_i$" and transition feature "transition:$y_{i-1}+y_i$" to form a new feature "combine:$y_{i-1}+y_i+x_i$". Please add this new feature to your CRF model and train the new model on `full/train` dataset. Apply the Viterbi algorithm to the dataset `full/dev.in` , and save the outputs into `full/dev.p5.CRF.f4.out`. You can evaluate your model's predictions using `conlleval.evaluate` function.

(10 points)

(iii) We have introduced another discriminative model "Structured Perceptron" in class. Structured Perceptron is similar to CRF in its ability to find the global optimal sequence, except that it is not a probabilistic model. Therefore, we don't need to compute the normalization term, that is, we don't need forward/backward algorithm for Structured Perceptron. Please build a Structured Perceptron based on `full/train` dataset. Evaluate its performance on `full/dev.in`, and save the outputs into `full/dev.p5.SP.out`.You can evaluate your model's predictions using `conlleval.evaluate` function.

(10 points)

6 Part 6 – Design Challenge *(20 points)*

For those of you who are interested, you may also try to compare the performance of your CRF model and the performance of the HMM model that you implemented in the ML design project on the same datasets. From here we will see how important choosing/designing a better machine learning model could be when building an NLP application in practice (however, there is no requirement for you to include such comparisons in your report).

In this section, we'll have two design challenges on NER. The first challenge is to customize CRF features for stronger performance while the other one is to design a NER system based on any model architectures you like. You are allowed to use external packages for the second challenge, but we require that you fully understand the methods/techniques that you used, and you need to clearly explain such details in the submitted report. We will evaluate your system's performance on the held out test set `partial/test.in` or `full/test.in` (if your model is trained on `full/train` dataset). The test sets will only be released on 19 August 2020 at 5pm (48 hours before the deadline). Use your new system to generate the outputs. The system that achieves the highest F1 score will be announced as the winner for each challenge. We may perform further analysis/evaluations in case there is no clear winner based on the released test set.

(i) So far we've seen how easy it is to add customized features to CRF. Please design your own features for your CRF and tune your model on `full/dev.in`. Once the test sets are released, please test your CRF on `partial/test.in` or `full/test.in` (if your model is trained on `full/train`

dataset) and write the outputs to `partial/test.p6.CRF.out` or `full/test.p6.CRF.out`. Report the *precision*, *recall* and *F1 scores* of your new CRF.

(10 points)

(ii) Now, think of a better design for developing improved NLP systems for the task of named entity recognition. Please design your new NER model and tune your model on `full/dev.in`. Once the test sets are released, please test your model on `partial/test.in` or `full/test.in` (if your model is trained on `full/train` dataset) and write the outputs to `partial/test.p6.model.out` or `full/test.p6.model.out`. Report the *precision*, *recall* and *F1 scores* of your new systems.

Please explain clearly the model or method that you used for designing the new systems and provide clear documentations on how to run the code. We will check your code and may call you for an interview if we have questions about your code.

Hints: Are there ways to make the system more robust? Can we make use of the state-of-the-art deep learning based methods? What about using LSTM or self-attention for learning feature representations (e.g., LSTM-CRF)? What about some advanced contextual embeddings such as ELMo, BERT or XLNet? Any other ideas?

(10 points)

Items To Be Submitted

Upload to eDimension a single ZIP file containing the following: (Please make sure you have only one submission from each team only.)

- A report detailing the approaches and results
- Source code (.py files) with README (instructions on how to run the code)
- Output files
  - `partial/`
    1. `dev.p2.out`
    2. `dev.p4.out`
    3. `test.p6.CRF.out`
    4. `test.p6.model.out`
  - `full/` (If your model is trained on this dataset)
    1. `dev.p5.CRF.f3.out`
    2. `dev.p5.CRF.f4.out`
    3. `dev.p5.SP.out`
    4. `test.p6.CRF.out`
    5. `test.p6.model.out`