Retention & Recognition

A Computational Model for Segmentation in Artificial Language Learning

Contents

Goals. The goal of this computer lab is that you train a segmentation model, fitting its parameters to empirical data, and evaluate its performance. The model that you will train is the Retention&Recognition model (RnR henceforth). You can find a short description of the model in Alhama, Scha, and Zuidema (2015) and in the slides in the `materials/` directory. We will use the data and evaluation procedures described in Frank et al. (2010).

Requirements. You need to write and run codes in python and the core libraries that you will use are: `matplotlib`, `random`, `numpy`, `scipy`. If you don't have these libraries installed you can install them using `pip` or `conda`. In case you are not very familiar with drawing plots in python, you might find this `matplotlib` gallery helpful.

Materials In the `materials/src` directory, you can see 7 `*.py` files. `RnR.py` is the computational model that you will train, `Frank2010Results.py` contains methods to load the empirical data and `Frank2010.py` contains objects and methods for defining experiments. You don't need to understand what happens in these

Last updated on July 11, 2019. Written by Raquel G. Alhama (2017). Updated by Bas Cornelissen (2019).

files but you can use the functions and classes defined in these files to do this assignment. You will need to use functions defined in `optimization.py` for fitting the RnR model with the and algorithms. The main file from which you will be running your code is `train.ipynb`. This is where you should load the data, instantiate the RnR model and call your optimization functions tFinally, the file `polynomial.py` contains an example model that will be described later. o train the model.

In the `materials/data` directory, you can find three files, each one containing human responses for each of the three experiments described in Frank et al. (2010). The name of the file indicates which experiment it corresponds to.

Handing in. You should deliver a pdf report including the **plots** and **explanations**, and **codes** that you wrote (This could be the pdf version of your notebook) along with the ipython notebook you have worked on and all **source files that you have edited**.

1 Introduction

1.1 The data

As mentioned before, the goal of this assignment is for you to train and evaluate a computational model for segmentation in Artificial Language Learning. You are provided with the model (the RnR model mentioned above, and seen in the slides), and your task is to fit the parameters of this model (train the model) on the dataset that is provided.

Frank et al. (2010) present three experiments to study how different factors affect word segmentation. These factors are:

- **Sentence length**
  This is experiment number 1. In this experiment the number of tokens and the size of the vocabulary are fixed. The sentence length is set to be 1, 2, 3, 4, 6, 8, 12 or 24.
- **Number of tokens**
  This is experiment number 2. In this experiment the size of the vocabulary and sentence length are fixed. The total number of tokens in the input stream is set to be 48, 100, 300, 600, 900 or 1200.
- **Size of the vocabulary**
  This is experiment number 3. In this experiment the number of tokens and the length of the sentence are fixed. The vocabulary size is set to be 3, 4, 5, 6 or 9.

In each experiment the input stream is generated pseudo-randomly to satisfy the specified criteria. After being exposed to this stream, the participants answer a 2AFC test in which they choose between words and partwords. These responses are collected in three data files. You can find these files in `materials/data` directory (`E1_data.csv`, `E2_data.csv`, `E3_data.csv`).

EXERCISE 1

To see what the input stream for experiment 1 looks like, you can run the following code:

```
import Frank2010
expId = 1 # Change to 2 or 3 to see the input stream for experiment 2 or 3
exp = Frank2010.experiment(expId, data_dir='../data/')
for cnd, expcnd in exp.cnd.items():
    print(Frank2010.CONDITION[expId] + ": " + str(cnd))
    print(expcnd.stream)
```

If you run into problems when trying to import `Frank2010` *from outside the* `materials/src` *directory, first add the directory to your path:*

```
import sys
sys.path.append('../relative/path/to/materials/src')
import Frank2010
```

If you instantiate the experiment using `exp = Frank2010.experiment(expId)`, the data of human responses is already loaded and you can access it as follows:

- `exp.expResults` is the loaded and processed human response data
- `exp.expResults.data` is a list of tuples containing the condition, subject, and if he/she was correct or not
- `exp.expResults.performance` is performance of each subject for each condition
- `exp.expResults.avg_performance` is average performance of all subjects for each condition
- `exp.expResults.std_performance` is standard deviation of performance of all subjects on each condition
- `exp.plotPerformance()` plots the responses of the humans in the experiment

1.2 The model

As mentioned above the RnR model is a probabilistic segmentation model. Given a stream of syllables, it breaks it into segments. The model works based on two probabilities, the recognition probability and the retention probability. You can read more about this model in Alhama, Scha, and Zuidema (2015). The recognition and retention probabilities of a segment s are computed as follows:

$$P_{\text{rec}}(s) = (1 - B^{\text{activation}(s)}) \cdot D^{\#\text{types}}$$

$$P_{\text{ret}}(s) = A^{\text{length}(s)} \cdot C^{\pi}$$

As you can see. The model involves 4 parameters A, B, C and D that should be fitted to the empirical data.

EXERCISE 2

This is an example of how you can instantiate the RnR model in your code, for a given parameter setting:

```
import RnR
rnr_model = RnR.RnRv2(A=0.04, B=0.3, C=0.3, D=0.3, nmax=4)
```

You can run the RnR model and see the words it memorizes by running this code:

```
for cnd, expcnd in exp.cnd.items():
    memory = rnr_model.memorizeOnline("##".join(expcnd.stimuli))
    print(memory)
```

In order to test the performance of the model, after each input stream, a list of pairs of sequences consisting of one word and one partword will be given to the model, and it has to decide which of the sequences is more likely to be a word.

These test pairs are stored in `expcnd.test`. The output of the RnR model consists of a memory of segments, together with their subjective frequencies. In order to convert that output to the probabilities of choosing a test item, we use the Luce rule, which states that, given two alternative options s_1 and s_2 to choose from (sequences in our case), the probability to choose one over the other is:

$$P(s_1) = \frac{\text{score}(s_1)}{\text{score}(s_1) + \text{score}(s_2)}$$

where score is, in the case of the RnR model, a subjective frequency. In the code, we call the Luce choice rule as follows:

```
prob_x, prob_y, chosen = rnr_model.Luce(pair[0], pair[1])
```

In the `__main__` method of `train.py`, you can find an example of use of the classes `RnR` and `Frank2010`. Run the code and make sure you understand it before proceeding (you don't need to understand all code in the modules, just how they can be used).

2 Model fitting

Computational models normally include a set of free parameters. The behaviour of the model changes depending on the values of such parameters. Hence, after choosing a computational model, the next step is to set the parameters of the model so that the behaviour/output of the model is consistent/similar to the phenomenon that is being modelled. This is what we call *training* or *fitting* the model.

2.1 An example model: a polynomial

Here is a simple example: we model the relation between two variables x and y. We have a set of n observations (x_i, y_i) and now want to formulate a model that predicts the value of y for a given x. In other words, we want to fit a curve through the observed points (x_i, y_i). After long deliberation we decide that the curve should be a polynomial of degree 2. The predicted value y_{pred} for a given x then takes the form

$$y_{\text{pred}}(x) = Ax^2 + Bx + C,$$

where A, B and C are the parameters of the model. (Since the prediction depends on the parameters, it would be better to write $y_{\text{pred}}(x \mid A, B, C)$)

Now we have to choose the values of these parameters in order to get a curve that best approximates the data points. But what do we mean by best approximation? We formally define this using an *objective function*. This function measures either how *well* your predictions fit the observed data, or how *poorly* (then it's often called a *cost function*). During the training phase you want to maximize or minimize the objective funciton. In other words, we want to find the parameters values that result in the best between predictions and observations, as measured by the objective function.

In this example, we can define the objective function as the (mean absolute) difference between observed values y_i and the values $y_{\text{pred}}(x_i)$ predicted by our polynomial model. This would be a cost function which we want to minimize. For the mathematically inclined, you could define it as follows:

$$C(\text{observations}, \text{params}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - y_{\text{pred}}(x_i \mid A, B, C)|.$$

One thing to consider is that, depending on the data points, it is not always possible to find the set of parameters that gives us an objective function with minimum possible value (zero in this case).

QUESTION 1

For this polynomial model, what is the maximum number of observations (x_i, y_i) for which can always find a set of parameters that make the objective function be zero? (Assume all x_i's are distinct.)

Hint 1: If you have a model with k parameters and you have n data points, this gives you m equations with n variables (parameters). If $n == k$ then you will find exactly one valid value for each variable (parameter). If $m < n$ the equations would have more than 1 answer and if $n > k$ there would be no solution at all.

Hint 2: Consider a linear model, $y_{\text{pred}}(x) = Ax + B$, where there are two parameters. If you have one point, you can have infinite number of lines that go through the point. If you have two points, there is only one lines

that passes both of the points. Then if you add a third point, if it is not on the same line as the first two points, you can not have a line that passes all the three points.

2.2 Grid search

Now, if we have the model, the objective function, and the data points, how can we find the best set of values for the parameters? The algorithms that are used to find the best set of values for the parameters of a model are called *optimization* algorithms. Here, we look at two different optimization algorithm: *grid search* and *hill climbing*.

One way to find the optimal parameter setting is to compute the objective function for all possible equidistant combinations of different values for the parameters, and then choose the combination that gives us the lowest cost (the output of the cost function). This method is called grid search.

In the example model, set C to a random value, and assume A and B can be any real value in the range $[-1, 1]$. We have to pick a step size, let's say 0.1. (Since there are indefinite possible values we pick a step size in order to pick a sample.) Thus we would have 21 possible values for both A and B:

$$\{-1, -.9, -.8, 0.7, ..., 0, .1, .2, ..., .9, 1\}$$

If we take all possible combinations, we would have 21×21 parameter settings. This gives us the grid in the figure below.

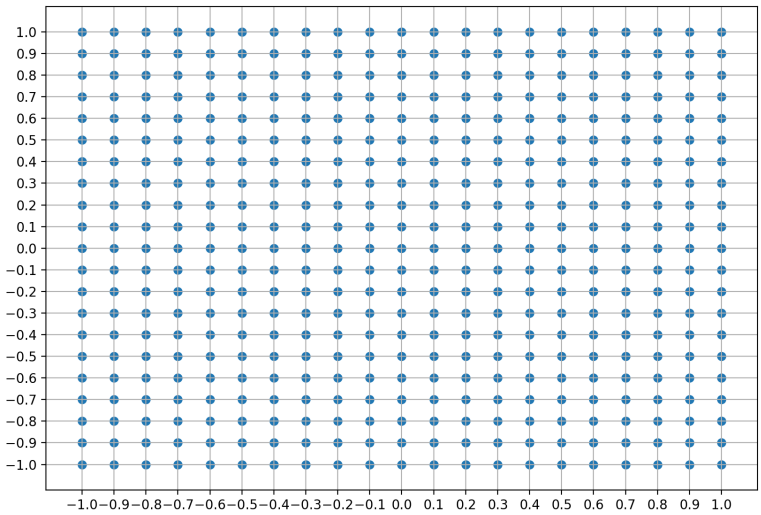

Figure 1: Parameter search space for grid search in the example model

QUESTION 2

The example model just discussed is implemented in `polynomial.py`. You can initialize it and generate training data as follows:

```
from polynomial import PolynomialModel
model = PolynomialModel(A_init=2, B_init=3, C_init=.75)
data = PolynomialModel.generate_training_data(num_datapoints=30)
```

Look at the code and make sure you understand how it works and what else you can do with it. The model has a method that allows you to do a grid search. Plot a heatmap showing the cost function for different values of A and B. *(1 point)*

QUESTION 3

Make a plot that shows the observations as black dots and plot the best curve found by the grid search as a solid red line on top of it. Give the curve a label showing those best parameters (something like $y = .25x^2 + .61x + 1$). Are the parameters good ones and why? If not, how can you change the initial parameters to get a better fit? *(1 point)*

2.3 Hill climbing

There are several methods that help us avoid exploring the whole space of the parameter values, using greedy decisions to directly explore the space in the direction that is more probably closer to the minimum point. One of those is the hill climbing algorithm.

Hill climbing is in the family of local search algorithms. The idea behind it is that you start from a random point (where each point is a possible solution for the problem), then you decide in which direction you should change the parameters based on the value of the cost function in the current point and its neighbours. Greedily, in each iteration, you choose the next point to be where the cost function is the lowest.

There exist different versions of this algorithm. Here, we will implement Simple Hill Climbing. Informally, you can think about Simple Hill Climbing as if you were lost in some huge hilly landscape and your goal is to find the highest point. If you follow this algorithm, you would make a step from where you are, in one random direction, and see if you have moved higher. If so, you choose your next random direction from this point; if, instead, the landscape goes down, then you go back where you were and choose another random direction.

Here you see an outline of a python function implementing hill climbing.

```
def HillClimbing(initialParams, maxIterations):
    currentParams = initialParams
    bestParams = currentParams
```

```
    for iteration in range(maxIterations):
        for i, param in enumerate(currentParams):
            nextParam = getNext(param)
            tmpParams = currentParams
            tmpParams[i] = nextParam
            tmpCost = cost(model(tmpParams), data)
            currentCost = cost(model(currentParams), data)
            if tmpCost < currentCost:
                currentParams[i] = nextParam

        bestCost = cost(model(bestParams), data)
        currentCost = cost(model(currentParams), data)
        if currentCost < bestCost:
            bestParams = currentParams

    return bestParams
```

QUESTION 4

You can see the code for `hill_climbing` for a model with multiple parameters in `materials/src/optimization.py`. Use this function to find the solution for the example problem. In a plot show the data points and the fitted curve.

3 Training the RnR model

Now that you know what does it mean to fit a model to some data, you are ready to train the RnR model. In the RnR model, there are 4 parameters to be tuned (A, B, C, D), and they all should be in the range of $(0, 1)$. The goal is to set these parameters in a way that this model behaves as similar as possible to humans in doing the segmentation task. The objective function is Pearson's r, which computes the correlation between the performances of the model and the average performances of humans for different conditions.

QUESTION 5

Apply grid search to fit the parameters of the RnR model for each of the experiments separately. Choose step size of 0.01 (if this is too slow for your computer, choose a larger step size, such as 0.1, and report it). What is the best correlation that you get? *(1 point)*

Hint: You can use `exp.pearsonR(performances)` to compute the cost.

QUESTION 6

Apply hill climbing to fit the parameters of the RnR model for each of the experiments separately. What is the best correlation that you get? Do you

get similar results as with grid search? Why? Draw a plot to show how the cost changes after each iteration.

4 Evaluating the RnR model

QUESTION 7

For each of the three fitted models that you got from the hill climbing search algorithm, draw a plot that compares the performance of humans with the performance of the model. What can you conclude? (1 point)

QUESTION 8

In this assignment, we have fitted the three datasets independently, so we end up with three different models. What would be a better practice? Can you think of a training setup in which we can make sure that the parameters *generalize* (i.e. do not overfit the data)? *You do not need to write any code, just reason about this*

References

Alhama, Raquel G, Remko Scha, and Willem Zuidema. 2015. "How Should We Evaluate Models of Segmentation in Artificial Language Learning?" In *Proceedings of 13th International Conference on Cognitive Modeling*.

Frank, Michael C, Sharon Goldwater, Thomas L Griffiths, and Joshua B Tenenbaum. 2010. "Modeling Human Performance in Statistical Word Segmentation." *Cognition* 117 (2). Elsevier: 107–25.