ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Natural Language Processing
Spring 2021

Prof. Ryan Cotterell

Course Assignment Episode 1

July 15, 2021

Alessandro Ruzzi
nethz Username: aruzzi
Student ID: 20953774

Collaborators:
Luca Malagutti
Rafael Sterzinger

Question 1: Backpropagation

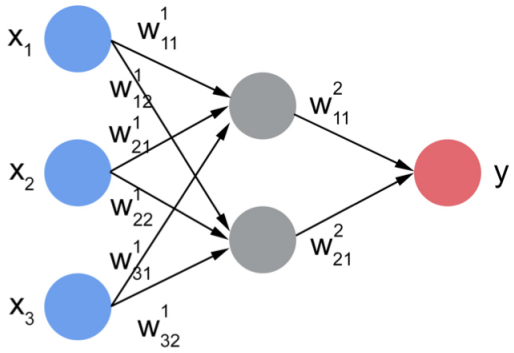


Figure 1: Neural Network f

(a) Point (a) states that we should draw the computation graph of the neural network represented in Figure 1. Figure 2 shows the computation graph, where $ReLU()$ represent the $ReLU$ activation function, $\sigma()$ the *sigmoid* activation function, "$*$" the scalar multiplication and "$+$" the scalar sum.

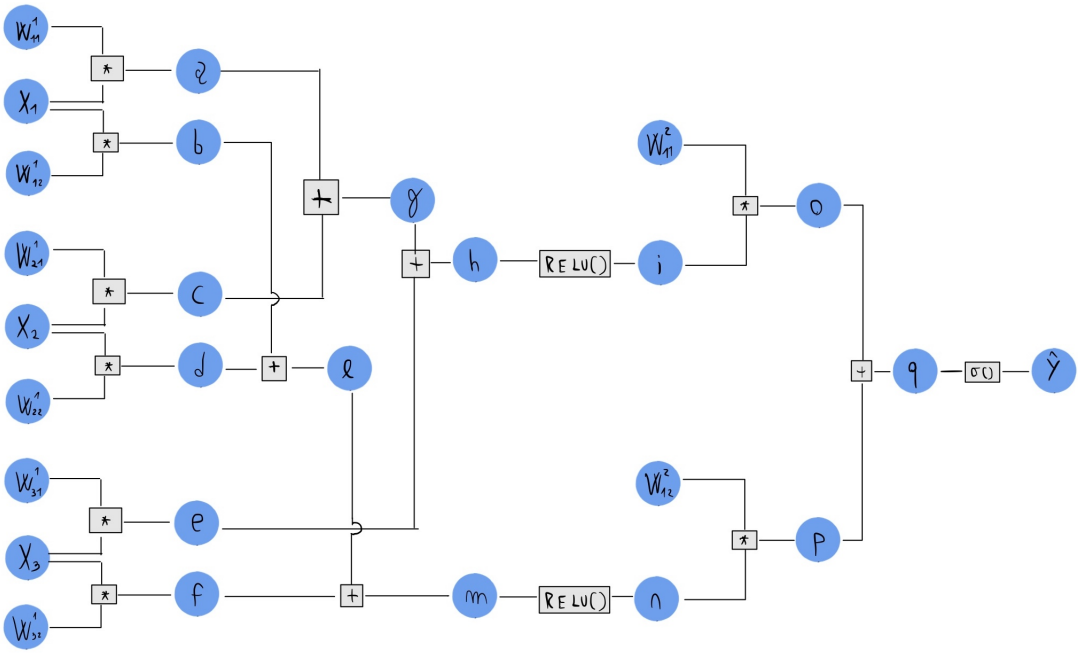


Figure 2: Computation graph of Neural Network f

(b) (i) To evaluate the graph at point: $(\boldsymbol{x_0}, y_0) = ((1,1,1),0)$ we first calculate the value of each vertex in the computation graph and then at the end we retrieve the value of $\hat{y}$.
Below are shown the values of all vertices including $\hat{y}$, evaluated at $\boldsymbol{x_0}$= (1,1,1):

- $a = 1$
- $b = 1$
- $c = 1$
- $d = 1$
- $e = 1$
- $f = 1$
- $g = 2$
- $h = 3$
- $i = 3$
- $l = 2$
- $m = 3$
- $n = 3$
- $o = 3$
- $p = 3$
- $q = 6$
- $\hat{y} = \mathbf{0.9975}$

(ii) Figure 3 shows all partial derivatives of all vertices w.r.t. the previous vertex, while below are shown the partial derivatives of $\hat{y}$ w.r.t. all the vertices in the graphs including the weights:

- $\frac{\partial \hat{y}}{\partial q} = 0.00249$
- $\frac{\partial \hat{y}}{\partial o} = \frac{\partial \hat{y}}{\partial q}\frac{\partial q}{\partial o} = 0.00249$
- $\frac{\partial \hat{y}}{\partial p} = \frac{\partial \hat{y}}{\partial q}\frac{\partial q}{\partial p} = 0.00249$
- $\frac{\partial \hat{y}}{\partial w_{11}^2} = \frac{\partial \hat{y}}{\partial o}\frac{\partial o}{\partial w_{11}^2} = \mathbf{0.00747}$
- $\frac{\partial \hat{y}}{\partial i} = \frac{\partial \hat{y}}{\partial o}\frac{\partial o}{\partial i} = 0.00249$
- $\frac{\partial \hat{y}}{\partial w_{12}^2} = \frac{\partial \hat{y}}{\partial p}\frac{\partial p}{\partial w_{12}^2} = \mathbf{0.00747}$
- $\frac{\partial \hat{y}}{\partial n} = \frac{\partial \hat{y}}{\partial p}\frac{\partial p}{\partial n} = 0.00249$
- $\frac{\partial \hat{y}}{\partial h} = \frac{\partial \hat{y}}{\partial i}\frac{\partial i}{\partial h} = 0.00249$
- $\frac{\partial \hat{y}}{\partial m} = \frac{\partial \hat{y}}{\partial n}\frac{\partial n}{\partial m} = 0.00249$
- $\frac{\partial \hat{y}}{\partial g} = \frac{\partial \hat{y}}{\partial h}\frac{\partial h}{\partial g} = 0.00249$
- $\frac{\partial \hat{y}}{\partial e} = \frac{\partial \hat{y}}{\partial h}\frac{\partial h}{\partial e} = 0.00249$
- $\frac{\partial \hat{y}}{\partial f} = \frac{\partial \hat{y}}{\partial m}\frac{\partial m}{\partial f} = 0.00249$
- $\frac{\partial \hat{y}}{\partial l} = \frac{\partial \hat{y}}{\partial m}\frac{\partial m}{\partial l} = 0.00249$
- $\frac{\partial \hat{y}}{\partial a} = \frac{\partial \hat{y}}{\partial g}\frac{\partial g}{\partial a} = 0.00249$
- $\frac{\partial \hat{y}}{\partial c} = \frac{\partial \hat{y}}{\partial g}\frac{\partial g}{\partial c} = 0.00249$
- $\frac{\partial \hat{y}}{\partial d} = \frac{\partial \hat{y}}{\partial l}\frac{\partial l}{\partial d} = 0.00249$
- $\frac{\partial \hat{y}}{\partial b} = \frac{\partial \hat{y}}{\partial l}\frac{\partial l}{\partial b} = 0.00249$
- $\frac{\partial \hat{y}}{\partial w_{11}^1} = \frac{\partial \hat{y}}{\partial 1}\frac{\partial 1}{\partial w_{11}^1} = \mathbf{0.00249}$
- $\frac{\partial \hat{y}}{\partial x_1} = \frac{\partial \hat{y}}{\partial a}\frac{\partial a}{\partial x_1} + \frac{\partial \hat{y}}{\partial b}\frac{\partial b}{\partial x_1} = 0.005$
- $\frac{\partial \hat{y}}{\partial w_{12}^1} = \frac{\partial \hat{y}}{\partial b}\frac{\partial b}{\partial w_{12}^1} = \mathbf{0.00249}$
- $\frac{\partial \hat{y}}{\partial w_{21}^1} = \frac{\partial \hat{y}}{\partial c}\frac{\partial c}{\partial w_{21}^1} = \mathbf{0.00249}$
- $\frac{\partial \hat{y}}{\partial x_2} = \frac{\partial \hat{y}}{\partial c}\frac{\partial c}{\partial x_2} + \frac{\partial \hat{y}}{\partial d}\frac{\partial d}{\partial x_2} = 0.005$
- $\frac{\partial \hat{y}}{\partial x_3} = \frac{\partial \hat{y}}{\partial e}\frac{\partial e}{\partial x_3} + \frac{\partial \hat{y}}{\partial f}\frac{\partial f}{\partial x_3} = 0.005$
- $\frac{\partial \hat{y}}{\partial w_{22}^1} = \frac{\partial \hat{y}}{\partial d}\frac{\partial d}{\partial w_{22}^1} = \mathbf{0.00249}$
- $\frac{\partial \hat{y}}{\partial w_{31}^1} = \frac{\partial \hat{y}}{\partial e}\frac{\partial e}{\partial w_{31}^1} = \mathbf{0.00249}$
- $\frac{\partial \hat{y}}{\partial w_{32}^1} = \frac{\partial \hat{y}}{\partial f}\frac{\partial f}{\partial w_{32}^1} = \mathbf{0.00249}$

In order to avoid confusion with the notation we wrote all the derivatives as $\frac{\partial \hat{y}}{\partial p}$ instead of $\left(\frac{\partial \hat{y}}{\partial p}\right)_{\boldsymbol{x_0}}$.

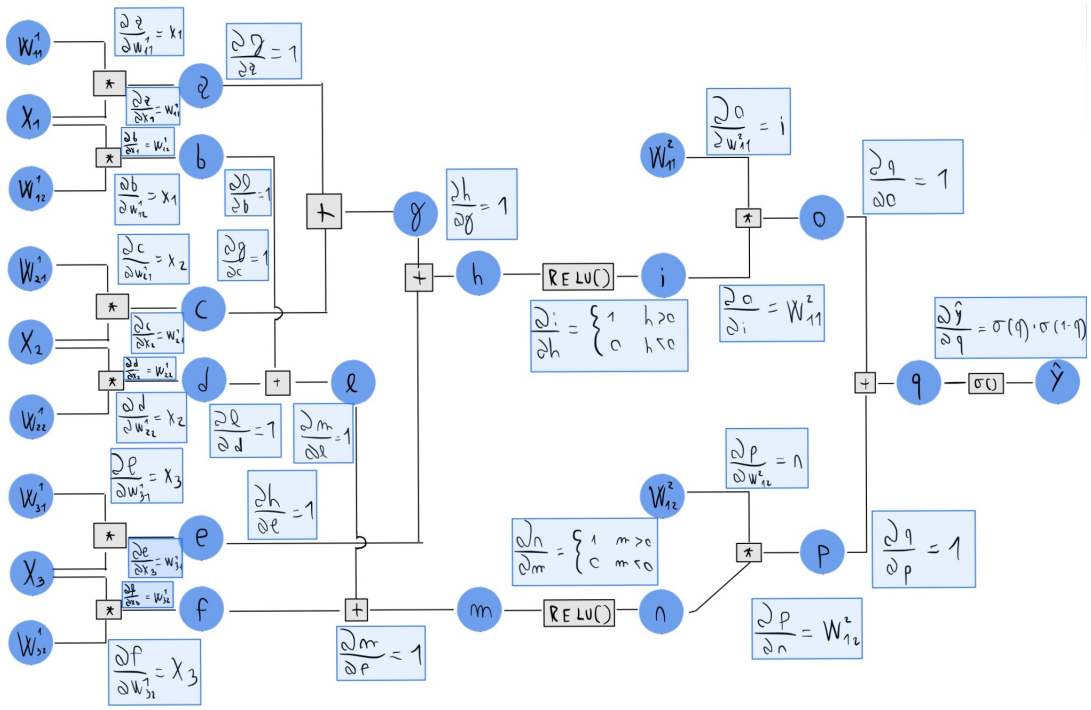


Figure 3: Computation graph with partial derivatives of all vertices w.r.t. the previous vertex

(iii) The Binary cross entropy loss evaluated at point $(\boldsymbol{x_0}, y_0) = ((1,1,1),0)$ is:

$$L_{BCE} = -(yln(\hat{y}) + (1-y)ln(1-\hat{y})) = 5.991 \tag{1}$$

The derivative of L_{BCE} evaluated at point $(\boldsymbol{x_0}, y_0) = ((1,1,1),0)$ is:

$$\frac{\partial L_{BCE}}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{(1-y)}{(1-\hat{y})} = 400 \tag{2}$$

(iv) To run a step of weight optimisation over f using gradient descent we need to calculate the derivatives of L_{BCE} w.r.t. all the weights. Equation 3 show how to calculate the partial derivative using a general weight, while below are shown the values of all the derivatives evaluated at point $(\boldsymbol{x_0}, y_0) = ((1,1,1),0)$:

$$\frac{\partial L_{BCE}}{\partial w_{ij}^r} = \frac{\partial L_{BCE}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_{ij}^r} \tag{3}$$

- $\frac{\partial L_{BCE}}{\partial w_{11}^2} = 2.988$
- $\frac{\partial L_{BCE}}{\partial w_{12}^2} = 2.988$
- $\frac{\partial L_{BCE}}{\partial w_{11}^1} = 0.996$
- $\frac{\partial L_{BCE}}{\partial w_{12}^1} = 0.996$
- $\frac{\partial L_{BCE}}{\partial w_{21}^1} = 0.996$
- $\frac{\partial L_{BCE}}{\partial w_{22}^1} = 0.996$
- $\frac{\partial L_{BCE}}{\partial w_{31}^1} = 0.996$
- $\frac{\partial L_{BCE}}{\partial w_{32}^1} = 0.996$

Now to calculate the updated weights we use the formula showed in equation 6:

$$w_{ij}^r = w_{ij}^r - \eta \frac{\partial L_{BCE}}{\partial w_{ij}^r} \qquad \eta = 0.1 \tag{4}$$

- $w_{11}^2 = 1 - 0.2988 = 0.7012$
- $w_{12}^2 = 1 - 0.2988 = 0.7012$
- $w_{11}^1 = 1 - 0.0996 = 0.9004$
- $w_{12}^1 = 1 - 0.0996 = 0.9004$
- $w_{21}^1 = 1 - 0.0996 = 0.9004$
- $w_{22}^1 = 1 - 0.0996 = 0.9004$
- $w_{31}^1 = 1 - 0.0996 = 0.9004$
- $w_{32}^1 = 1 - 0.0996 = 0.9004$

(c) In order to reduce the number of computations while producing the same result, we may change the computation graph by noticing that all the weights of the form w_{ij}^2 will receive the same update $\eta\nabla_2$ and all the weights of the form w_{ij}^1 will receive the same update $\eta\nabla_1$. This is due to the initialisation procedure that assign the same value to all the weights and allow us to reduce all the weights to just two weights w^1 and w^2.
The main advantages of this reduction are the improved efficiency of the gradient descent procedure (less parameters to update), the drastic reduction of the number of parameters and also the higher interpretability of the computation graph(we have a smaller graph and for a human is easier to understand what the network is doing and how to modify it).
Using this strategy we obtain the computation graph showed in Figure 4.

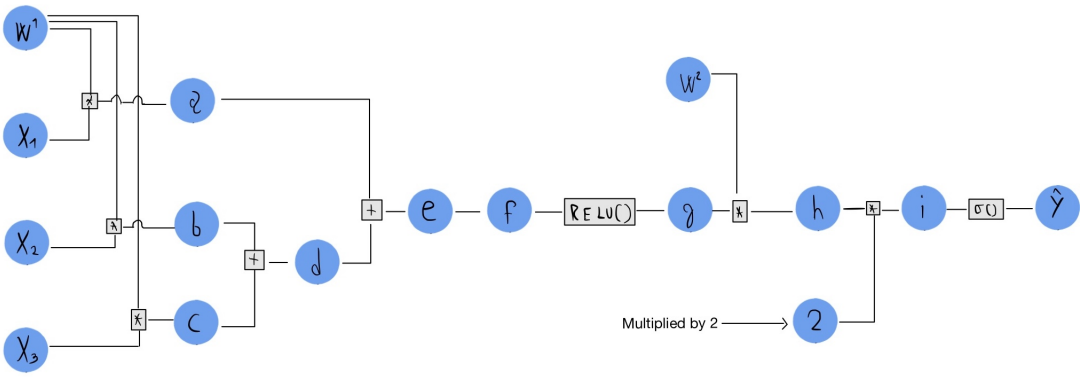


Figure 4: Revised computation graph of Neural Network f

Question 2: Log-linear models

(a) Solution for Question 2 can be found at **"assignment_log_linear_models.ipynb"** (if the link doesn't work, there is a notebook called "log_linear.ipynb" in the zip file).
Some clarifications:

- The feature function that we used is a very basic feature function that take as input a feature vector of dimension m and gives as output a new feature vector of dimension $2m$ that is build as *[old features, zeros]* if the label for that sample is equal to zero, or *[zeros, old features]* if the label is equal to one.

- The formula for the negative log likelihood and for its gradient are :

$$L(\theta) = -\sum_{n=1}^{N} log(p(y_n \mid x_n, \theta)) \qquad p(y \mid x, \theta) = \frac{exp(\theta \cdot \mathbf{f}(\mathbf{x}, y))}{\sum_{y' \in Y} exp(\theta \cdot \mathbf{f}(\mathbf{x}, y'))} \tag{5}$$

$$\frac{\partial L(\theta)}{\partial \theta_k} = -\sum_{n=1}^{N} \overbrace{\mathbf{f_k}(\mathbf{x_n}, y_n)}^{observed\ feature\ "counts"} + \sum_{n=1}^{N} \overbrace{\sum_{y' \in Y} p(y' \mid x_n; \theta) \cdot \mathbf{f_k}(\mathbf{x_n}, y')}^{expected\ feature\ "count"} \tag{6}$$

- To choose the appropriate learning rate we first used a standard value of 0.001, but then we noticed that using a smaller learning rate (0.0001) gives better results for the log-linear model, therefore we opted for that one.

(b) We will now compare the log-linear model and logistic regression on four different aspects : computation time, in-sample accuracy, out-of-sample accuracy and coefficient values.

(i) As we can see in Figure 5 and Figure 6 log-linear model computation time increase almost linearly with the number of samples while logistic regression time is constant, this is probably due to our implementation of the log-linear model that needs to be optimized. Then another cause could be the feature function, because for each sample we need to generate a new feature vector while with logistic regression we use the original sample as input.

(ii) By looking at the in-sample accuracy (Figure 7) together with the out-of-sample accuracy (Figure 8) we can observe that logistic regression tend to overfit more then log-linear model when we have only 100 samples, while log-linear model needs more epochs to converge, then this overfit is reduced with the second dataset where we have more sample and more features. Finally with the third dataset we have similar values for in-sample accuracy.

(iii) For what concerns the out-of-sample accuracy showed in Figure 8 we can notice that logistic regression perform better when we have less samples (first two dataset), while log-linear model has a slightly better score using the third dataset.
To conclude, the higher out-of-sample accuracy in the second dataset obtained by logistic regression can be due to the fact that sklearn implementation of logistic regression use by default l2 regularization while we didn't implement any regularization in our model, while the high overfit with the first dataset is due to the fact that the number of features is close to the number of samples, and this can cause problems to logistic regression.

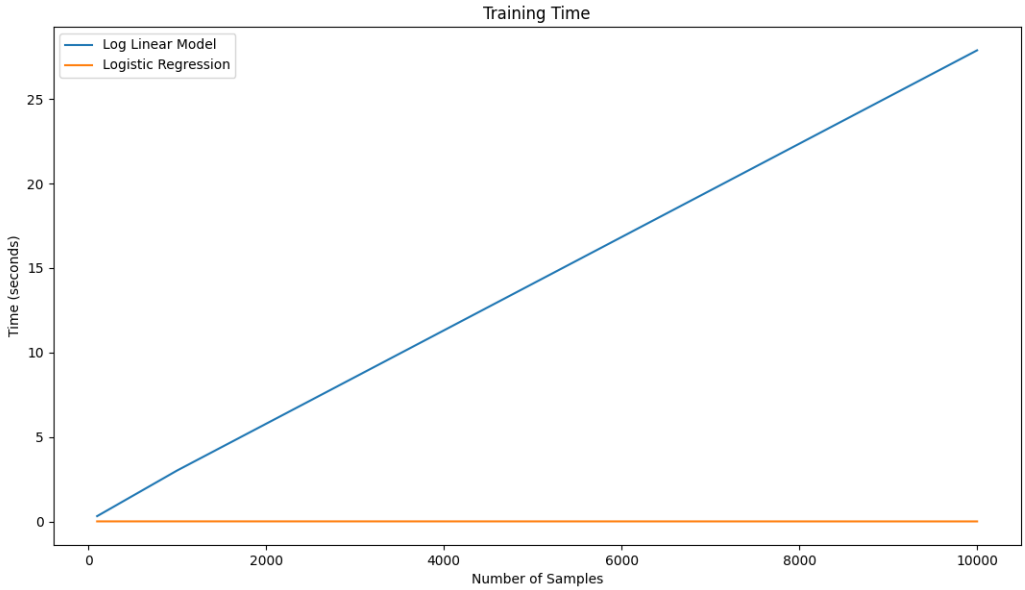


Figure 5: Training time of the two models evaluated on the 3 provided datasets

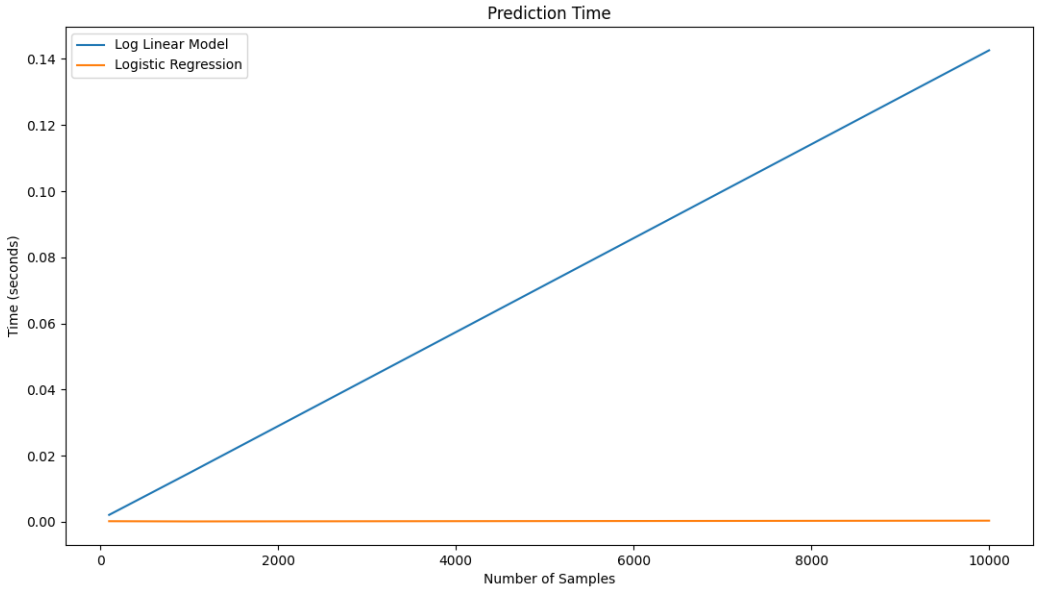


Figure 6: Prediction time of the two models evaluated on the 3 provided datasets

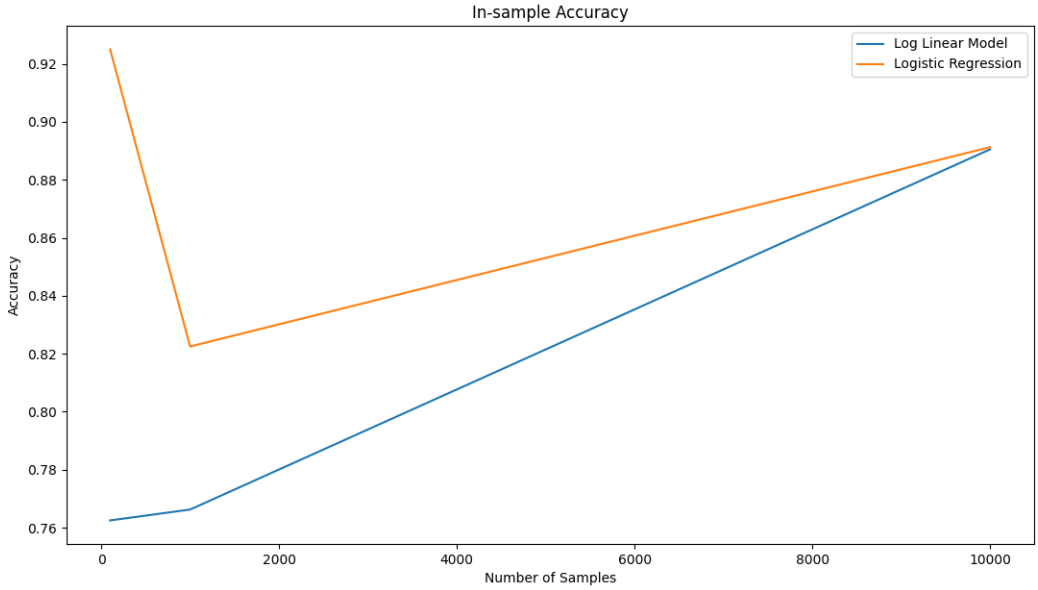


Figure 7: In-sample accuracy of both models calculated on the 3 provided datasets

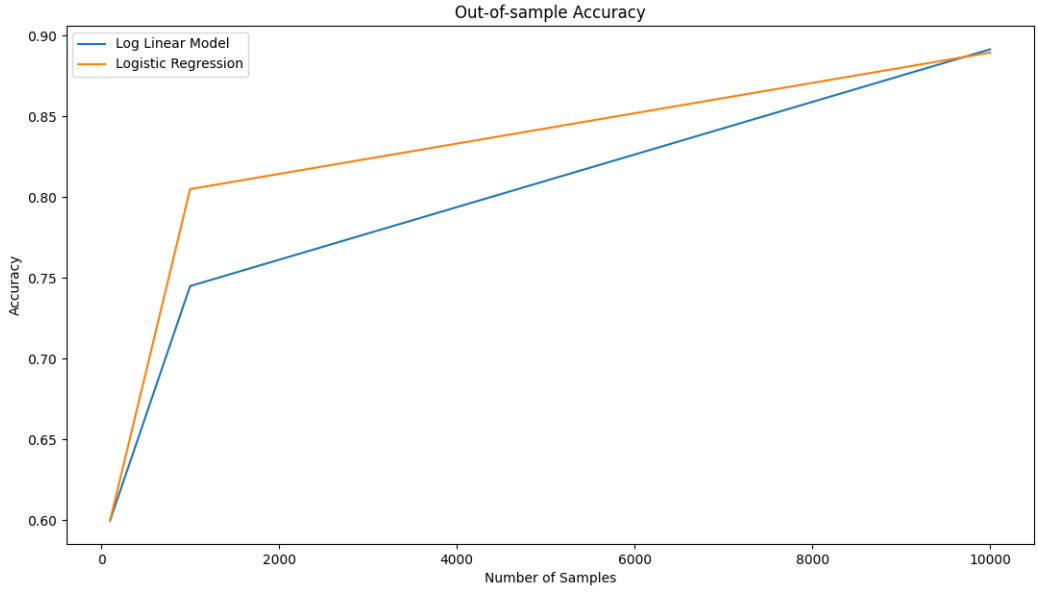


Figure 8: Out-of-sample accuracy of both models calculated on the 3 provided datasets

(iv) By observing Figure 9 we can notice that coefficient values are really different for the two model, this is probably due to the very low number of sample that cause overfitting, especially using logistic regression.
Then by looking at Figure 10 we can see that coefficient values start to become similar, while in Figure 11, where we have 10000 samples, we have the same values for almost all the coefficients.
Finally we can observe an higher spread of the log-linear model coefficient values, this is mainly to due to the fact that we didn't use any regularization for our model, while logistic regression apply L2 regularization by default.
Note that we used a mapping function to convert the log-linear model coefficients in order to have m coefficients instead of $2m$.

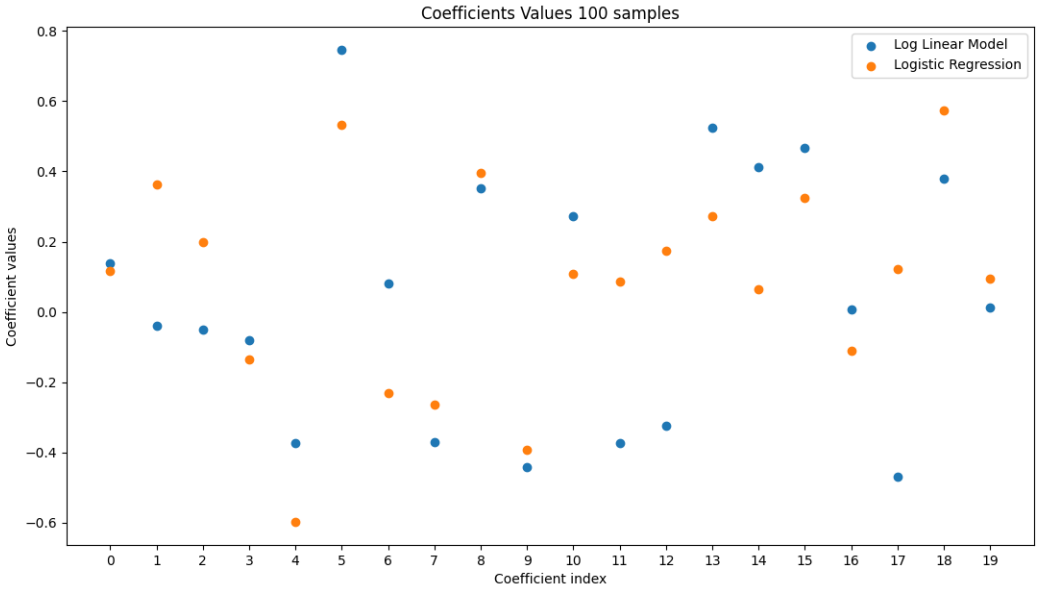


Figure 9: Coefficient values of both models obtained using the first dataset

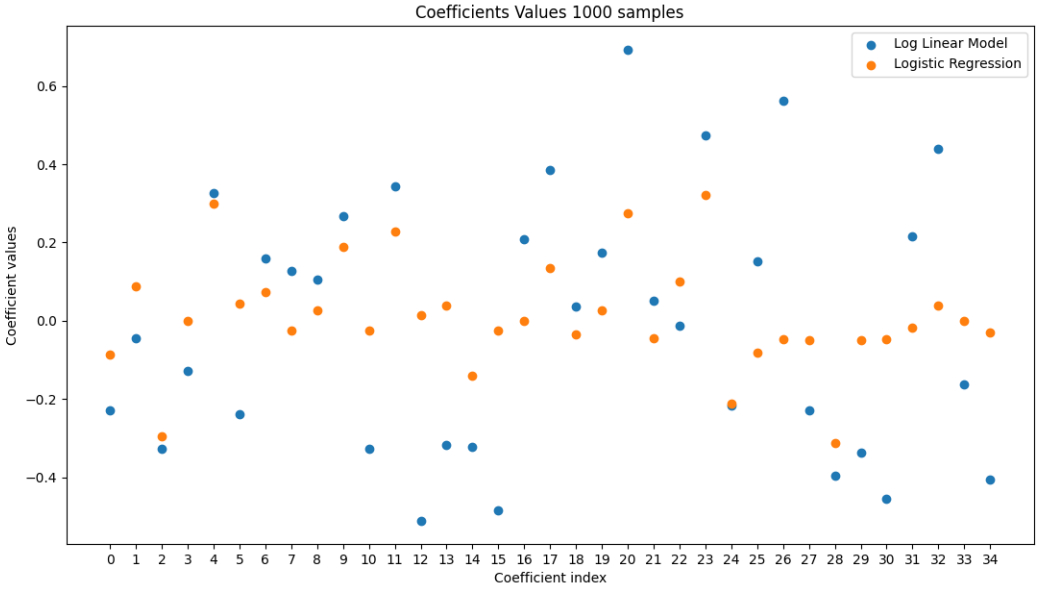


Figure 10: Coefficient values of both models obtained using the second dataset

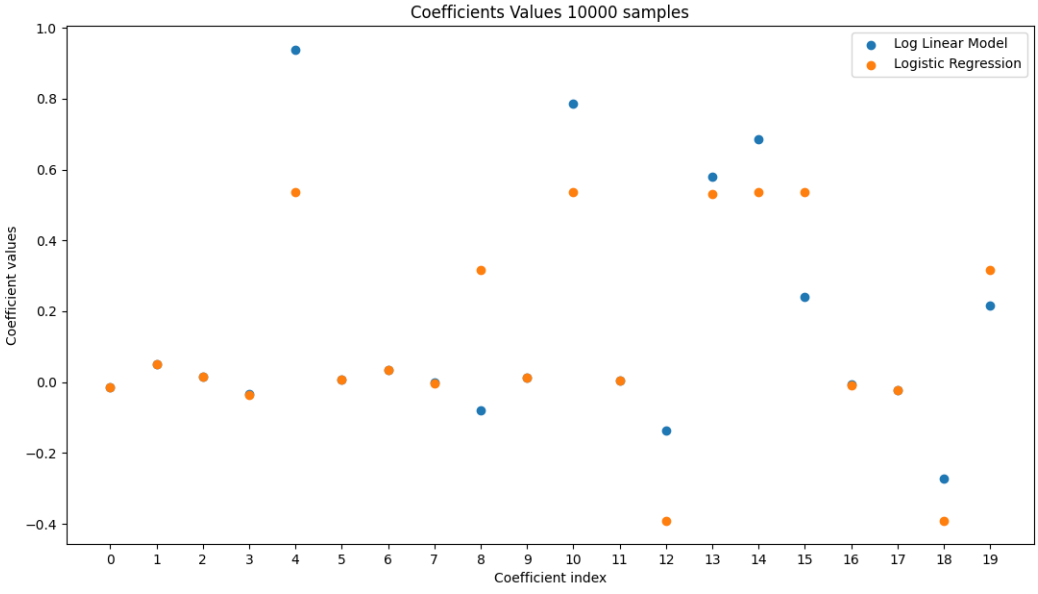


Figure 11: Coefficient values of both models obtained using the third dataset

Question 3: Skip-Gram

(a) Solution for Question 3 can be found at **"assignment_skip_gram.ipynb"** (if the link doesn't work, there is a notebook called "skip_gram.ipynb" in the zip file).
Some clarifications:

- In order to optimize the code we called the function np.random.choice() only once, by doing this our negative sampling process become much quicker (from roughly 20 minutes to roughly 2 minutes).

- To plot the loss we decided to use the function "accumulate", so for each step N our loss is calculated as:

$$loss_N = \frac{\sum_{j=1}^{N} \text{sample\_loss}}{N} \tag{7}$$

  The result is showed in Figure 12.

- For the visualization part we used a clustering method (agglomerative clustering) to improve the thematic clusters detection, and we also used the library "**adjust text**" to improve the readability of the words in the plots.

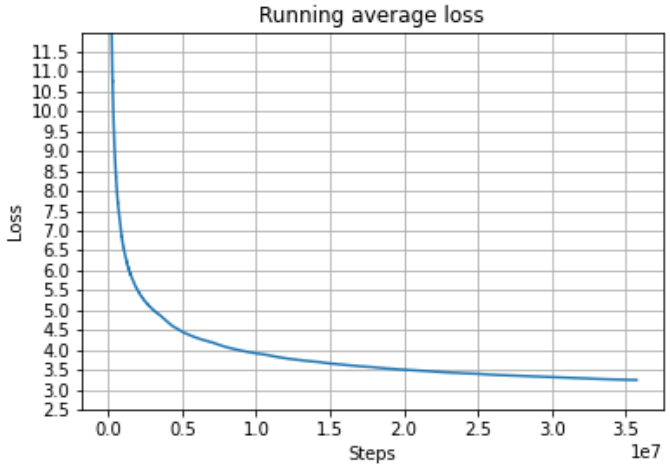


Figure 12: Average of accumulated loss over steps

Question 4: Language modeling

(a) (i) In order to calculate the expected number of distinct tokens that appears in n draws we first define X_i as an indicator random variable, that is one if token i appear at least once during the n draws, zero otherwise. Then we define X as:

$$X = \sum_{i=1}^{|V|} X_i \tag{8}$$

Then using the expected value of X and linearity of expectations we arrive at the result:

$$\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}[\sum_{i=1}^{|V|} X_i] = \sum_{i=1}^{|V|} \mathbb{E}[X_i] = \sum_{i=1}^{|V|} [1 \cdot P(X_i = 1) + 0 \cdot P(X_i = 0)] \\
&= \sum_{i=1}^{|V|} P(X_i = 1) = \sum_{i=1}^{|V|} 1 - P(X_i = 0) = \sum_{i=1}^{|V|} 1 - \left(1 - \frac{1}{|V|}\right)^n \\
&= |V| \cdot \left(1 - \left(1 - \frac{1}{|V|}\right)^n\right) = |V| \cdot \left(1 - \left(\frac{|V| - 1}{|V|}\right)^n\right) \\
&= |V| - \frac{(|V| - 1)^n}{|V|^{n-1}}
\end{aligned}$$

To conclude the expected number of distinct tokens that appears in n draws is:

$$\mathbb{E}[X] = |V| \cdot \left(1 - \left(\frac{|V| - 1}{|V|}\right)^n\right) = |V| - \frac{(|V| - 1)^n}{|V|^{n-1}} \tag{9}$$

(ii) To find the probability that each token $y \in |V|$ appear in n draws, we can use a simple counting approach, where we count all the possible sequences that we can have by doing n draws and all the possible sequences where all the tokens $t \in |V|$ appeared, finally we divide this two numbers.

- The total number of sequence that we can have is simply $|V|^n$.
- To find the number of sequences where all the tokens $t \in |V|$ appeared we use the principle of *inclusion-exclusion*.
  Let S_k be all the outcomes in which token k is not drawn. For each token k, $|S_k| = (|V| - 1)^n$ and there are $\binom{|V|}{1}$ choices for k. Then if we consider two tokens k and j, we get that for each j and k the number of outcomes in which they are not drawn is $(|V| - 2)^n$ and there are $\binom{|V|}{2}$ choices for j and k.
  By keep iterating this process and using the principle of *inclusion-exclusion* we obtain that the number of outcomes missing at least 1 number is:

$$M = \sum_{i=1}^{|V|-1} (-1)^{i+1} \binom{|V|}{i} (|V| - i)^n$$

Since there are $|V|^n$ possible sequences, we can write the number of sequences where

all the tokens $t \in |V|$ appeared as:

$$N = |V|^n - M = |V|^n - \sum_{i=1}^{|V|-1} (-1)^{i+1} \binom{|V|}{i} (|V| - i)^n = \sum_{i=0}^{|V|-1} (-1)^i \binom{|V|}{i} (|V| - i)^n$$

Finally let Y be an indicator random variable that is equal to one if all the tokens $t \in |V|$ appeared, zero otherwise. Then we just divide the numbers that we have found to obtain:

$$P(Y = 1) = \frac{\sum_{i=0}^{|V|-1} (-1)^i \binom{|V|}{i} (|V| - i)^n}{|V|^n} \tag{10}$$

That can be rewritten also as:

$$P(Y = 1) = \sum_{i=0}^{|V|-1} (-1)^i \binom{|V|}{i} \left(1 - \frac{i}{|V|}\right)^n \tag{11}$$

(b) (i) To find the expected number of draws in order to make the bigram "*work hard*" appear at least once we can use a recursive approach:
To begin we define $\mathbb{E}[n_{work}]$ as the expected number of draws to obtain the token "*hard*" if we have already drawn "*work*", and $\mathbb{E}[n]$ as the expected number of draws in order to make the bigram "*work hard*" appear at least once.

- To calculate $\mathbb{E}[n_{work}]$, we need to keep in mind that we can have 3 possible scenarios: we draw the token "*hard*" (probability of $\frac{1}{|V|}$) and we are done, we draw "*work*" (probability of $\frac{1}{|V|}$) and then we need another $\mathbb{E}[n_{work}]$ draws, or we draw a different word (probability of $\frac{|V|-2}{|V|}$) and we are back at the begging and we now need $\mathbb{E}[n]$ draws to find the bigram we are interested in.

$$\begin{aligned}
\mathbb{E}[n_{work}] &= \overbrace{\frac{1}{|V|} \cdot 1}^{draw\ hard} + \overbrace{\frac{1}{|V|} \cdot (1 + \mathbb{E}[n_{work}])}^{draw\ work} + \overbrace{\frac{|V| - 2}{|V|} \cdot (1 + \mathbb{E}[n])}^{draw\ different\ word} \\
(|V| - 1) \cdot \mathbb{E}[n_{work}] &= 1 + 1 + |V| - 2 + (|V| - 2) \cdot \mathbb{E}[n] \\
\mathbb{E}[n_{work}] &= \frac{|V| + (|V| - 2) \cdot \mathbb{E}[n]}{|V| - 1}
\end{aligned}$$

- Now to calculate $\mathbb{E}[n]$ we use the same recursive approach, by considering that we could have two possible outcomes: we draw a word different from "*work*" (probability of $\frac{|V|-1}{|V|}$) and we are back at the beginning, or we draw exactly "*work*" (probability of $\frac{1}{|V|}$) and then we need $\mathbb{E}[n_{work}]$ draws to find "*hard*".

$$\mathbb{E}[n] = \overbrace{\frac{|V|-1}{|V|} \cdot (1 + \mathbb{E}[n])}^{draw\ different\ from\ work} + \overbrace{\frac{1}{|V|} \cdot (1 + \mathbb{E}[n_{work}])}^{draw\ work}$$
$$|V| \cdot \mathbb{E}[n] = (|V|-1) \cdot (1 + \mathbb{E}[n]) + 1 + \mathbb{E}[n_{work}]$$
$$|V| \cdot \mathbb{E}[n] - (|V|-1) \cdot \mathbb{E}[n] = |V| - 1 + 1 + \mathbb{E}[n_{work}]$$
$$\mathbb{E}[n] = |V| + \mathbb{E}[n_{work}]$$

- To conclude we just substitute $\mathbb{E}[n_{work}]$ with our previous result and solve the equation to find the expected number of draws in order to make the bigram "*work hard*" appear at least once.

$$\mathbb{E}[n] = |V| + \frac{|V| + (|V|-2) \cdot \mathbb{E}[n]}{|V|-1}$$
$$(|V|-1) \cdot \mathbb{E}[n] = |V|^2 - |V| + |V| + (|V|-2) \cdot \mathbb{E}[n]$$
$$\mathbb{E}[n] = |V|^2$$

So the expected number of draws in order to make the bigram "*work hard*" appear at least once is:

$$\mathbb{E}[n] = |V|^2 \tag{12}$$

(ii) To find the number of draws we first define the probability that the word "*work*" appear in n draws as:

$$P(X_{work} = 1) = 1 - \left(1 - \frac{1}{|V|}\right)^n \tag{13}$$

Then to find n we impose that $P(X_{work} = 1) \geq 0.95$:

$$1 - \left(1 - \frac{1}{|V|}\right)^n \geq 0.95$$
$$\left(1 - \frac{1}{|V|}\right)^n \leq 0.05$$
$$n \cdot ln\left(1 - \frac{1}{|V|}\right) \leq ln(0.05)$$
$$n \cdot \left(-ln\left(1 - \frac{1}{|V|}\right)\right) \geq -ln(0.05)$$
$$n \geq \frac{ln(0.05)}{ln\left(1 - \frac{1}{|V|}\right)}$$

(c) (i) In order to find the expected number of draws before two of the same tokens appear next to each other, we will use again a recursive approach by considering that after the first draw we can have two possible outcomes: we get the same word also in the second draw (probability

of $\frac{1}{|V|}$) and we are done, or we get a different word (probability of $\frac{|V|-1}{|V|}$) and we are back at the beginning and we need again $\mathbb{E}[n]$ draws to find the same token consecutively.

$$\mathbb{E}[n] = \overbrace{1}^{first\ draw} + \overbrace{\frac{1}{|V|} \cdot 1}^{draw\ the\ same\ word} + \overbrace{\frac{|V|-1}{|V|} \cdot \mathbb{E}[n]}^{draw\ different\ word}$$

$$V \cdot \mathbb{E}[n] - (|V| - 1) \cdot \mathbb{E}[n] = 1 + |V|$$

$$\mathbb{E}[n] = |V| + 1$$

To conclude, the expected number of draws before two of the same tokens appear next to each other is:

$$\mathbb{E}[n] = |V| + 1 \tag{14}$$

(ii) To solve this problem we first denote x_t as the index of the word t in the vocabulary V, and we assume that each word has a different index value.
First we define the terms in the first function :

$$h_t^0 = f(w_0 x_{t-1} + w_1 x_t + w_2 h_{t-1}^0 + b_0) \tag{15}$$

- $w_0 = 1, w_1 = -1, w_2 = 0, b_0 = 0$
- f(x) = $\begin{cases} 1 & \text{if x=0} \\ 0 & \text{otherwise} \end{cases}$

Then we define the terms in the second function :

$$y_t = g(w_3 h_t^0 + b_1) \tag{16}$$

- $w_3 = 1, b_1 = 0$
- $g(x) = x$

We don't really need a non-linear activation function to solve this simple problem however it wouldn't make sense to use a recurrent neural network with only linear activation functions because we wouldn't gain anything in terms of complexity. One solution to set f to be a linear activation function would be swapping f and g. Otherwise we can re-define the term as:

- $w_0 = 1, w_1 = -1, w_2 = 0, b_0 = 0$
- $f(x) = x$
- $w_3 = 1, b_1 = 1$
- $g(x) = x$

With this definition $g(x)$ will be 1 each time $f(x) = 0$ (when we have $x_t = x_{t-1}$), the only problem would be that we output values different from 0 when $x_t \neq x_{t-1}$, but in the question is not stated that we are required to output zero when $x_t \neq x_{t-1}$.

(iii) In order to finish the problem of RNN and count the number of bigrams we need a non-linear activation function, and we will use the one we defined in the previous point.
First we define the terms in the first function :

$$h_t^0 = f(w_0 x_{t-1} + w_1 x_t + w_2 h_{t-1}^0 + b_0) \tag{17}$$

- $w_0 = 1, w_1 = -1, w_2 = 0, b_0 = 0$
- f(x) = $\begin{cases} 1 & \text{if x=0} \\ 0 & \text{otherwise} \end{cases}$

Then we define the terms in the second function :

$$h_t^1 = g(w_3 h_{t-1}^1 + w_4 h_t^0 + b_1) \tag{18}$$

- $w_3 = 1, w_4 = 1, b_1 = 0$
- $g(x) = x$

Finally we define the terms in the last function:

$$y_t = h(w_5 h_t^1 + b_2) \tag{19}$$

- $w_5 = 1, b_2 = 0$
- $h(x) = x$

(iv) We first start our discussion with an intuitive proof, because using a non-uniform distribution we can easily end up with some tokens that have very high probabilities, therefore we will also have an higher chance to draw the same token twice in a row.
Now we conclude our thinking with a formal proof using Cauchy-Schwarz inequality:

$$(x \cdot y)^2 \leq (x \cdot x) \cdot (y \cdot y) \tag{20}$$

First we define the probability of drawing the same token twice in a row using a uniform distribution as:

$$\sum_{i=1}^{|V|} \frac{1}{|V|^2} = \frac{1}{|V|} \tag{21}$$

Then we define the probability of drawing the same token twice in a row using a non-uniform distribution as:

$$\sum_{i=1}^{|V|} P(word_i)^2 \tag{22}$$

Now we define two vectors $\mathbf{x}$, $\mathbf{y} \in \mathbb{R}^{|V| \cdot 1}$ as:

$$\mathbf{x} = (P(word_1), P(word_2), \ ..... \ , P(word_{|V|}))$$
$$\mathbf{y} = (1, 1, 1, 1, 1, 1 \ ..... \ , 1, 1, 1, 1, 1, 1, 1)$$

In order to use Equation 20 we now define the three term $\mathbf{xy}, \mathbf{xx}, \mathbf{yy}$:

$$\mathbf{xy} = \sum_{i=1}^{|V|} P(word_i) \cdot 1 = 1$$
$$\mathbf{xx} = \sum_{i=1}^{|V|} P(word_i)^2$$
$$\mathbf{yy} = \sum_{i=1}^{|V|} 1 = |V|$$

To conclude we just substitute our new terms in Equation 20 to obtain:

$$\left(\sum_{i=1}^{|V|} P(word_i) \cdot 1 \right)^2 \leq \left(\sum_{i=1}^{|V|} P(word_i)^2 \right) \cdot |V|$$
$$\left(\sum_{i=1}^{|V|} P(word_i)^2 \right) \geq \frac{1}{|V|}$$

Therefore we proved that the probability of drawing the same token twice in a row using a non-uniform distribution ($\sum_{i=1}^{|V|} P(word_i)^2$) is higher than the probability of drawing the same token twice in a row using a uniform distribution ($\frac{1}{|V|}$). Note that the inequality becomes an equality only if:

$$P(word_i) = P(word_j) \quad \forall i, j \quad \text{such that} \quad 0 < i, j \leq |V|$$

Question 5: Dijkstra's algorithm

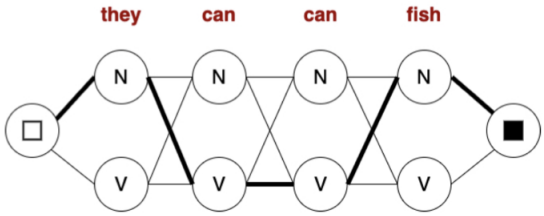


Figure 13: Example of a graph produced by input "they can can fish"

(a) (i) We will now define what vertices V, edges E and weights W should be to make dijkstra's algorithm solve the POS-tags problem, while Figure 13 show an example of how a graph should be in our case (consider the edges directed to the right):

- *Vertices* : considering X the input of length N, then for each element of X we have $|Y|$ vertices, where $|Y|$ represent the length of the output space. Finally we have two more vertices due to the "source" and "terminal" vertices.
  Total number of vertices: $N|Y| + 2$.

- *Edges* : we need directed edges that connect each vertex corresponding to an element of x (starting point of the edge) to each vertex corresponding to the next element of x (ending point of the edge). For example each vertex corresponding to *word1* should be connected to each vertex corresponding to *word2*.
  Set of edges $E \rightarrow \{\langle v_n \ v_{n+1} \rangle \quad \forall\, v_n \in V_n, \forall\, v_{n+1} \in V_{n+1} \ \ \& \ \ 0 \leq n \leq N\}$, where V_n is the set of all vertices that represent word n (i.e. vertices $\in$ layer n).
  Note that by using $0 \leq n \leq N$ we included also the source and terminal vertices.
  Total number of edges: $(N-1)|Y|^2 + 2|Y|$.

- *Weights* : weights should be all positive and will represent the distance between two nodes linked by an edge. The distance is based on the score function: higher is the output of the score function, lower is the distance between two nodes. An easy way to accomplish this is to use $w(edge) = exp(-score(edge, \mathbf{x}))$ as mapping function, in this way we obtain all positive weights and we reverse the order, so that higher scores correspond to lower distances.

Algorithm 1 show the pseudocode to solve our problem using dijkstra's algorithm, while below are listed some assumption we need to consider to run the pseudocode:

- As priority queue we will use a min-heap where entries are tuples of the type (distance, vertex). Note that using a Fibonacci heap is possible to reduce time complexity.

- The neighbours of a given vertex are defined as all the vertices that have an entry edge starting from the vertex we are interested in.

- We will use two dictionaries, one to store the actual distance of each vertex from the starting vertex, and the second to store for each vertex its predecessor in the shortest path (this one is used to retrieve the shortest path).

- Vertices with lower distance from the starting vertex have higher priority in the queue.

After the algorithm terminate, in order to retrieve the shortest path we start from the terminal vertex and go backward using the dictionary *predecessor* until we reach the source node.

Algorithm 1: dijkstra POS-tags pseudocode

```
Function Dijkstra(graph, neighbours, source):
    heap ← []
    dist ←{}
    predecessor ←{}
    for vertex in graph do
        dist[vertex] ← +∞
    end
    dist[source] ←0
    heap.append((0, source))
    for vertex in graph do
        heap.append((+∞, vertex))
    end
    while length(heap) != 0 do
        heap ←sort(heap)
        distance,v ← heap.next()
        heap ←heap.remove(v)
        for neighbour,weight in neighbours[v] do
            if neighbour in heap then
                newDistance ←dist[v] + weight
                if min(dist[neighbour], newDistance) == newDistance then
                    dist[neighbour] ←newDistance
                    predecessor[neighbour] ←v
                    heap.remove(neighbour)
                    heap.append((newDistance, neighbour))
                end
            end
        end
    end
```

(ii) We will now analyze each step of our algorithm to find the time complexity:

- Adding all $|V|$ vertices to the heap take $O(|V|)$ time.
- Using an hash-map we can remove and re-add a vertex in $O(log(|V|))$ time. This happen at most $|E|$ times, so total time complexity is $O(|E|log(|V|))$.
- Remove from the heap $|V|$ nodes takes $O(|V|log(|V|))$.
- Checking that the heap is empty happen $|V|$ times, so it takes $O(|V|)$.
- Iterating over all neighbours takes $O(|E|)$.

Therefore our final runtime will be $O((|E| + |V|)log(|V|))$ that can be reduce to $O(|E| + |V|log(|V|))$ using a Fibonacci heap.
In order to compare our runtime with Viterbi runtime we need to translate our runtime in terms of length of the input N, and dimension of output space $|Y|$. So our time complexity will become $O((N|Y|^2 + N|Y|)log(N|Y|))$ or $O(N|Y|^2 + N|Y|log(N|Y|))$ with the Fibonacci heap (where we used the following equalities: $O(|V|) = O(N|Y|)$) and $O(|E|) = O(N|Y|^2)$), while Viterbi time complexity is $O(N|Y|^2)$.

In conclusion we can clearly state that our time complexity is worse than Viterbi one in both cases:

- Viterbi Time Complexity : $O(N|Y|^2)$
- Algorithm 1 Time Complexity : $O((N|Y|^2 + N|Y|)log(N|Y|))$
- Fibonacci heap Time Complexity : $O(N|Y|^2 + N|Y|log(N|Y|))$

(b) (i) To write the pseudocode using semiring we used the notation of a general semiring showed in Equation 23 , while Algorithm 2 show the adapted pseudocode.

$$\tau = (Domain, \bigoplus, \bigotimes, \overline{0}, \overline{1}) \tag{23}$$

To solve the problem presented in point (a) we should use a well known semiring called *Tropical Semiring*:

$$\tau = (R_+ \cup \{+\infty\}, min, +, +\infty, 0) \tag{24}$$

Algorithm 2: dijkstra POS-tags pseudocode with semiring notation

```
Function Dijkstra(graph, neighbours, source):
    heap ← []
    dist ←{}
    predecessor ←{}
    for vertex in graph do
        dist[vertex] ← 0̄
    end
    dist[source] ←1̄;
    heap.append((1̄, source))
    for vertex in graph do
        heap.append((0̄, vertex))
    end
    while length(heap) != 0 do
        heap ←sort(heap)
        distance,v ← heap.next()
        heap ←heap.remove(v)
        for neighbour,weight in neighbours[v] do
            if neighbour in heap then
                newDistance ← ⊗(dist[v], weight)
                if ⊕(dist[neighbour], newDistance) == newDistance then
                    dist[neighbour] ←newDistance
                    predecessor[neighbour] ←v
                    heap.remove(neighbour)
                    heap.append((newDistance, neighbour))
                end
            end
        end
    end
```

(ii) In order to find the *longest path* we need to change both the semiring and the sorting procedure of the heap. For what concerns the heap, the vertices with higher distances will have higher priority, so we should reverse the sorting procedure of the heap. Finally we also have to modify the domain, because we have all negative weights.
The new semiring is:

$$\tau = (R_- \cup \{-\infty\}, max, +, -\infty, 0) \tag{25}$$

(iii) To find the *widest path*, assuming all the weights as non-negative, we keep the sorting procedure of point b(ii). Furthermore the distance dictionary won't contain anymore the shortest distance from a given node to the source node, but it will instead contains the value of the actual minimum weight in the path between the given node and the source node.
The semiring used to find the *widest path* is:

$$\tau = (R_+ \cup \{-\infty\} \cup \{+\infty\}, max, min, -\infty, +\infty) \tag{26}$$

Using this semiring we first do the *mim* to find the minimum weight in the path that bring to *neighbour* (the one we are analyzing in that iteration) through v. Then using *max* we verify if the minimun weight that we found is higher that the actual minimum weight contained in dist[neighbour]. After the algorithm ends we can find the value of the minimum weight in the dictionary *dist*, and we can retrieve the widest path using the dictionary *predecessor*.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Natural Language Processing
Spring 2021

Prof. Ryan Cotterell

Course Assignment Episode 2

July 15, 2021

Alessandro Ruzzi
nethz Username: aruzzi
Student ID: 20953774

Collaborators:
Luca Malagutti
Rafael Sterzinger

Question 1: CFG Refinement

(a) A general Context Free Grammar (CFG) is defined as a quadruple G:

$$G = \langle N, S, \Sigma, R \rangle \tag{27}$$

(i) N $\rightarrow$ Set of non-terminal symbols.

(ii) S $\rightarrow$ Distinguished start non-terminal symbol.

(iii) Σ $\rightarrow$ Alphabet of terminal symbols.

(iv) R $\rightarrow$ Set of production rules "N $\rightarrow \alpha$"

For this question we have:

(i) N $\rightarrow$ {NP, VP, PP, V, P, N, Det}.

(ii) S $\rightarrow$ {S}.

(iii) Σ $\rightarrow$ {I, draw, a, man, with, pencil, hit, ball, an, umbrella, glasses}

(iv) R $\rightarrow$ Set of production rules "N $\rightarrow \alpha$" :

S $\rightarrow$ NP VP

NP $\rightarrow$ NP PP | Det N | *I* | *glasses*

VP $\rightarrow$ VP PP | V NP

PP $\rightarrow$ P NP

V $\rightarrow$ *draw* | *hit*

P $\rightarrow$ *with*

N $\rightarrow$ *man* | *pencil* | *ball* | *umbrella*

Det $\rightarrow$ *a* | *an*

(b) Now we learn the probabilities with a simple counting approach, for example in the training set the non-terminal symbol VP produce two times VP $\rightarrow$ VP PP and four times VP $\rightarrow$ V NP , therefore the probabilities for this two production rule will be respectively $\frac{1}{3}$ and $\frac{2}{3}$.

S $\rightarrow$ NP VP (1)

NP $\rightarrow$ NP PP $(\frac{1}{7})$ | Det N $(\frac{1}{2})$ | *I* $(\frac{2}{7})$ | *glasses* $(\frac{1}{14})$

VP $\rightarrow$ VP PP $(\frac{1}{3})$ | V NP $(\frac{2}{3})$

PP $\rightarrow$ P NP (1)

V $\rightarrow$ *draw* $(\frac{1}{2})$ | *hit* $(\frac{1}{2})$

P $\rightarrow$ *with* (1)

N $\rightarrow$ *man* $(\frac{4}{7})$ | *pencil* $(\frac{1}{7})$ | *ball* $(\frac{1}{7})$ | *umbrella* $(\frac{1}{7})$

Det $\rightarrow$ *a* $(\frac{6}{7})$ | *an* $(\frac{1}{7})$

(c) To solve the problem pointed out in question (c) we should notice that the production NP $\rightarrow$ NP PP is more likely when the non-terminal NP is the child of the non-terminal VP instead of the non-terminal PP.
Therefore to solve this problem and to capture the subject-object difference we will augment each non-terminal $\in$ N with the identity of their parent. By doing this we can capture the phenomenon in which we are interested in.
Our new PCFG is (we will write the non-terminals like N-P where the right hand P is the parent of the left hand N) :

(i) N $\rightarrow$ {NP-S, NP-VP, NP-PP, NP-NP, VP-S, PP-VP, PP-NP, V-VP, Det-NP, N-NP, P-PP}.

(ii) S $\rightarrow$ {S}.

(iii) $\Sigma \rightarrow$ {I, draw, a, man, with, pencil, hit, ball, an, umbrella, glasses}

(iv) R $\rightarrow$ Set of production rules "N $\rightarrow \alpha$" :

S $\rightarrow$ NP-S VP-S (1)

NP-S $\rightarrow$ *I* (1)

NP-VP $\rightarrow$ NP-NP PP-NP $(\frac{1}{2})$ | Det-NP N-NP $(\frac{1}{2})$

NP$-$PP $\rightarrow$ Det-NP N-NP $(\frac{3}{4})$ | *glasses* $(\frac{1}{4})$

NP$-$NP $\rightarrow$ Det-NP N-NP (1)

VP-S $\rightarrow$ VP-VP PP-VP $(\frac{1}{2})$ | V-VP NP-VP $(\frac{1}{2})$

VP-VP $\rightarrow$ V-VP NP-VP (1)

PP-VP $\rightarrow$ P-PP NP-PP (1)

PP-NP $\rightarrow$ P-PP NP-PP (1)

V-VP $\rightarrow$ *draw* $(\frac{1}{2})$ | *hit* $(\frac{1}{2})$

P-PP $\rightarrow$ *with* (1)

N-NP $\rightarrow$ *man* $(\frac{4}{7})$ | *pencil* $(\frac{1}{7})$ | *ball* $(\frac{1}{7})$ | *umbrella* $(\frac{1}{7})$

Det-NP $\rightarrow$ *a* $(\frac{6}{7})$ | *an* $(\frac{1}{7})$

To conclude we can clearly see that now we obtained a precise distinction between NP subject and NP object since the production rule NP-S $\rightarrow$ NP-NP PP-NP has zero probability because NP-S is the subject of the phrase, while the production rule NP-VP $\rightarrow$ NP-NP PP-NP has probability $\frac{1}{2}$ because NP-VP is the object of the phrase.

(d) In order to incorporate the head token in our production rule, we can use a lexicalized PCFG that augment each non-terminal in the parse tree with the head token (calculated with a recursive approach as staeted in point (d) description).
The new parse tree is shown in Figure 14 , while below are listed the production rules used to build the parse tree.

S(hit) $\rightarrow$ NP(I) VP(hit)

VP(hit) $\rightarrow$ V(hit) NP(man)

NP(man) $\rightarrow$ NP(man) PP(glasses)

NP(man) $\rightarrow$ Det(a) N(man)

PP(glasses) $\rightarrow$ P(with) NP(glasses)

NP(I) $\rightarrow$ *I*

V(hit) $\rightarrow$ *hit*

Det(a) $\rightarrow$ *a*

N(man) $\rightarrow$ *man*

P(with) $\rightarrow$ *with*

NP(glasses) $\rightarrow$ *glasses*

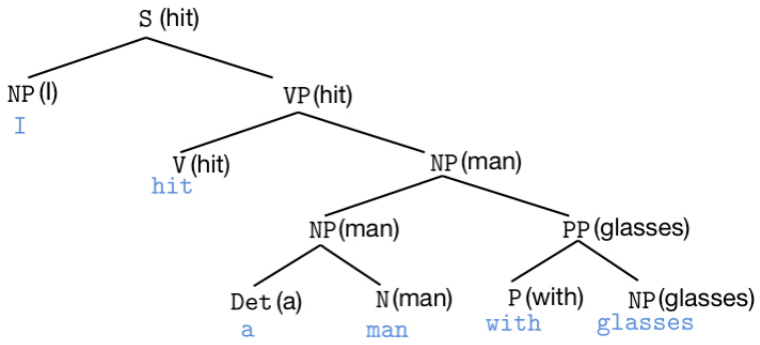


Figure 14: parse tree of "I hit a man with glasses" using a lexicalized PCFG

(e) (i) The CKY algorithm with the CFG defined in point (a) has the following space and time complexity:

- Time Complexity : $O(|R|M^3)$
- Space Complexity : $O(|N|M^2)$

Where M is the length of the sentence (input), $|R|$ is the size of the set of production rules and $|N|$ is the number of non-terminals. The space complexity is $O(|N|M^2)$ because we have $O(M^2)$ number of cells and each cell must hold $O(|N|)$ elements.

(ii) Using parents annotation we square the number of non-terminals, because for each non-terminal we have $|N|$ new non-terminals, therefore $|N_1| = |N|^2$.
The number of production rules will become $|R_1| = |R||N|$, because for each old production rule of the type B-A $\rightarrow$ C-B D-B we have $|N|$ new production rules by changing "A" with one of the non-terminals $\in$ N.

- Time Complexity : $O(|R_1|M^3) = O(|R||N|M^3)$
- Space Complexity : $O(|N_1|M^2) = O(|N|^2M^2)$

(iii) With lexicalized PCFG our approch will be to extent the set of non-terminals by taking the cross-product with the set of terminal symbols, to include symbols like VP(hit), NP(man). Therefore our set of non-terminal symbols will have dimension $|N_1| = |N||\Sigma|$, and our set of production rules will have dimension $|R_1| = |R||\Sigma|^2$, because we will have rules like:

(1) NP(x) $\rightarrow$ NP(x) PP(y)

(2) NP(x) $\rightarrow$ NP(y) PP(x)

Then for each old rule we have $2|\Sigma|^2$ new rules, since $x, y \in \Sigma$ and we can change them in rules (1) and (2) to obtain new rules.

- Time Complexity : $O(|R_1|M^3) = O(|\Sigma|^2|R|M^3)$
- Space Complexity : $O(|N_1|M^2) = O(|\Sigma||N|M^2)$

Question 2: Parsing Projective Dependecy Trees

(a) It is known that dependency parsing can be motivated by extension from the lexicalized context-free grammar, that we have seen in the previous question. Therefore the relationship between words in a sentence can be formalized in a directed graph, based on the lexicalized phrase-structure parse. Our objective is to exploit this relation between the two trees and define the weights of the production rules to make sure that we get the same score with both the trees, for any input phrases.
To solve this problem we need to define a lexicalized WCFG that allow us to generate an equivalent constituency tree for any possible dependency tree that we receive as input, we will define the lexicalized weighted CFG using the same notation used in Question 1:

(i) N $\rightarrow$ {W(x), R(x)}. ($\forall x \in \Sigma$)

(ii) S $\rightarrow$ {S(x)}. ($\forall x \in \Sigma$)

(iii) $\Sigma \rightarrow$ {All English words}

(iv) R $\rightarrow$ Set of production rules "N $\rightarrow \alpha$", $\forall x, y \in \Sigma$:

$$\begin{aligned} &\text{S(x)} \rightarrow \text{W(x)} \\ &\text{W(x)} \rightarrow \text{R(x)} \mid \text{W(x)}\ \ \text{W(y)} \mid \text{W(y)}\ \ \text{W(x)} \\ &\text{R(x)} \rightarrow \text{x} \end{aligned}$$

Now to define the weight of each production rule we need to consider that in our WCFG all the production rules can be summarized using 4 types of generic production rules:

(i) S(x) $\rightarrow$ W(x) weight = ψ(ROOT $\rightarrow$ x) ($\forall x \in \Sigma$)

(ii) $\begin{cases} \text{W(x)} \rightarrow \text{W(x)}\ \ \text{W(y)} \\ \text{W(x)} \rightarrow \text{W(y)}\ \ \text{W(x)} \end{cases}$ weight = ψ(x $\rightarrow$ y) ($\forall x, y \in \Sigma$)

(iii) W(x) $\rightarrow$ R(x) weight = 0 ($\forall x \in \Sigma$)

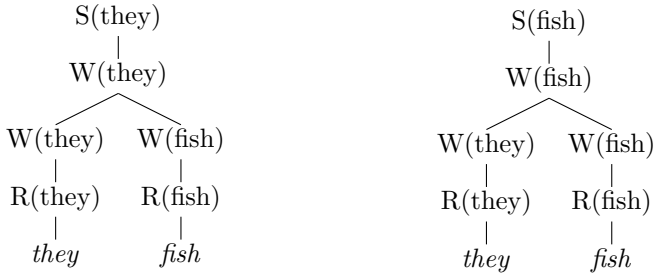
(iv) R(x) $\rightarrow$ x weight = 0 ($\forall x \in \Sigma$)

Note that with the assumption that a specific word cannot occurs more than once we can avoid rules such as $W(x) \rightarrow W(x)\ \ W(x)$, and it becomes easier to define the weights for the production rules. Also note that to define the weights of our lexicalized WCFG we assumed that the score of a general dependency tree is calculated as:

$$score(\boldsymbol{t}) = \prod_{(i \rightarrow j) \in \mathbf{t}} e^{\psi(i \rightarrow j)} \cdot e^{\psi(Root \rightarrow root\_word)} \tag{28}$$

(b) In order to evaluate our method with the sample "*they fish*" we first need to draw all the possible dependency trees that we can have:

Then we need the draw the equivalent constituency trees:



Now we need to verify that we obtain the same score for both the pair of trees (using the score function provided):

(i) First we calculate the score for the first Dependency tree:

$$\textbf{\textit{dep\_score\_1}} = e^{\psi(\text{ROOT} \rightarrow \text{they})} \cdot e^{\psi(\text{they} \rightarrow \text{fish})}$$

Then we calculate the score for the first Constituency tree using the following weights (based on the previous point):

- S(they) $\rightarrow$ W(they) weight $= \psi(\text{ROOT} \rightarrow \text{they})$ score $= e^{\psi(\text{ROOT} \rightarrow \text{they})}$
- W(they) $\rightarrow$ W(they) W(fish) weight $= \psi(\text{they} \rightarrow \text{fish})$ score $= e^{\psi(\text{they} \rightarrow \text{fish})}$
- W(they) $\rightarrow$ R(they) weight $= 0$ score $= e^0 = 1$
- W(fish) $\rightarrow$ R(fish) weight $= 0$ score $= e^0 = 1$
- R(they) $\rightarrow$ they weight $= 0$ score $= e^0 = 1$
- R(fish) $\rightarrow$ fish weight $= 0$ score $= e^0 = 1$

$$\begin{aligned}\textbf{\textit{const\_score\_1}} &= e^{\psi(\text{ROOT} \rightarrow \text{they})} \cdot e^{\psi(\text{they} \rightarrow \text{fish})} \cdot 1 \cdot 1 \cdot 1 \cdot 1 \\ &= e^{\psi(\text{ROOT} \rightarrow \text{they})} \cdot e^{\psi(\text{they} \rightarrow \text{fish})}\end{aligned}$$

(ii) Now we calculate the score for the second Dependency tree:

$$\textbf{\textit{dep\_score\_2}} = e^{\psi(\text{ROOT} \rightarrow \text{fish})} \cdot e^{\psi(\text{fish} \rightarrow \text{they})}$$

Then we calculate the score for the second Constituency tree using the following weights (based on the previous point):

- S(fish) → W(fish) weight = $\psi(\text{ROOT} \rightarrow \text{fish})$ score = $e^{\psi(\text{ROOT} \rightarrow \text{fish})}$
- W(fish) → W(they) W(fish) weight = $\psi(\text{fish} \rightarrow \text{they})$ score = $e^{\psi(\text{fish} \rightarrow \text{they})}$
- W(they) → R(they) weight = 0 score = $e^0 = 1$
- W(fish) → R(fish) weight = 0 score = $e^0 = 1$
- R(they) → they weight = 0 score = $e^0 = 1$
- R(fish) → fish weight = 0 score = $e^0 = 1$

$$\begin{aligned} \textbf{\textit{const\_score\_2}} &= e^{\psi(\text{ROOT} \rightarrow \text{fish})} \cdot e^{\psi(\text{fish} \rightarrow \text{they})} \cdot 1 \cdot 1 \cdot 1 \cdot 1 \\ &= e^{\psi(\text{ROOT} \rightarrow \text{fish})} \cdot e^{\psi(\text{fish} \rightarrow \text{they})} \end{aligned}$$

To conclude we can state that our method work for the sample "*they fish*" since we obtained the same score ($e^{\psi(\text{ROOT} \rightarrow \text{they})} \cdot e^{\psi(\text{they} \rightarrow \text{fish})}$) for the first pair of trees, and the same score ($e^{\psi(\text{ROOT} \rightarrow \text{fish})} \cdot e^{\psi(\text{fish} \rightarrow \text{they})}$) for the second pair of trees.

Question 4: Floyd-Warshall WFSA

(a) The procedure to find out which strings are accepted by the *WFSA* consists in crossing the graph and verify that there is a path that represents our sentence, and in the case there is one we calculate the weight to cross it. Note that to choose the starting vertex in the path we use the first word present in the sentence we are analyzing, and we find out which vertex represent that word. Table 1 contains the solution.

number	**sample strings**	**accepted**	**weight**
1	educational is this not	No	
2	is this assignment educational	No	
3	not educational is not educational	Yes	12
4	this assignment is not educational	Yes	9
5	is this assignment educational	No	
6	this assignment course is educational	No	
7	is this assignment not educational	Yes	15
8	this assignment not	Yes	8
9	this course assignment is not educational	Yes	12
10	this course is not not educational	Yes	21
11	not educational is this	Yes	10
12	course assignment is not educational	Yes	10
13	not this assignment is educational	No	
14	not not not educational	Yes	25
14	is this course assignment not educational	Yes	18
15	course assignment is this	Yes	8
16	this course is interesting	No	
17	this course assignment not educational	Yes	14

Table 1: Some strings from $\mathcal{Y}_{\geq 2,\leq 6}$

(b) The problem states that we should use the Floyd-Warshall algorithm to find the shortest path between all pairs of nodes. We used the classic Floyd-Warshall algorithm to fill the left column matrices in Figure 15 and Figure 16 , then we used the rule showed in Algorithm 3 to fill the right column matrices.
Note that we initialised to zero the diagonal elements of the left column matrices due to the fact that we don't have negative weights in the self-loops (done before the update of the assignment).

Algorithm 3: Path reconstruction algorithm

```
Initialisation code here...
for k = 1 to |V| do
    for i = 1 to |V| do
        for j = 1 to |V| do
            if dist[i][j] > dist[i][k] + dist[k][j] then
                next[i][j] ← next[i][k]
            end
        end
    end
end
```

(c) The number of iterations of Floyd-Warshall algorithm is bound by the number of vertices $|V|$ in the graph. In our case the algorithm stops at $iteration = 6$ that is equal to the number of vertices in our graph.
The necessary condition that needs to be met in order to execute the algorithm are:

(i) We need a directed weighted graph with positive or negative weights.

(ii) Negative cycles are not allowed.

(d) The time and space complexity for standard Floyd-Warshall algorithm are:

- Time Complexity : $O(|V|^3)$
- Space Complexity : $O(|V|^2)$

While if we run the algorithm with a second matrix for path reconstruction then the time and space complexity will become:

- Time Complexity : $O(|2V|^3) = O(|V|^3)$
- Space Complexity : $O(|2V|^2) = O(|V|^3)$

We have this result because Algorithm 3 has three "for" cycles, therefore it runs in $O(|V|^3)$ and our final time complexity will become $O(|V|^3 + |V|^3)$. Furthermore by using a second matrix for the path reconstruction also our space complexity will be doubled: $O(|V|^2 + |V|^2)$.
Finally to retrieve the lowest weight path for two given nodes the maximum time complexity is $O(|V|)$ since we go through each node at most one time due to the fact that we don't have negative cycles, while to retrieve all the lowest weight paths the time complexity is $O(|V|^3)$.
Note that in our case we are also missing negative weights.

iteration: n = 0, weight matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	0	1	2	∞	∞	∞
b assignment	∞	0	∞	1	5	∞
c course	∞	2	0	2	3	∞
d is	4	∞	∞	0	2	4
e not	∞	∞	∞	∞	0	2
f educational	∞	∞	∞	3	∞	0

iteration: n = 0, backtracking matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	a	b	c	–	–	–
b assignment	–	b	–	d	e	–
c course	–	b	c	d	e	–
d is	a	–	–	d	e	f
e not	–	–	–	–	e	f
f educational	–	–	–	d	–	f

iteration: n = 1, weight matrix , n=a

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	0	1	2	∞	∞	∞
b assignment	∞	0	∞	1	5	∞
c course	∞	2	0	2	3	∞
d is	4	5	6	0	2	4
e not	∞	∞	∞	∞	0	2
f educational	∞	∞	∞	3	∞	0

iteration: n = 1, backtracking matrix n=a

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	a	b	c	–	–	–
b assignment	–	b	–	d	e	–
c course	–	b	c	d	e	–
d is	a	a	a	d	e	f
e not	–	–	–	–	e	f
f educational	–	–	–	d	–	f

iteration: n = 2, weight matrix n=b

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	0	1	2	2	6	∞
b assignment	∞	0	∞	1	5	∞
c course	∞	2	0	2	3	∞
d is	4	5	6	0	2	4
e not	∞	∞	∞	∞	9	2
f educational	∞	∞	∞	3	∞	0

iteration: n = 2, backtracking matrix n=b

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	a	b	c	b	b	–
b assignment	–	b	–	d	e	–
c course	–	b	c	d	e	–
d is	a	a	a	d	e	f
e not	–	–	–	–	e	f
f educational	–	–	–	d	–	f

iteration: n = 3, weight matrix n=c

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	0	1	2	2	5	∞
b assignment	∞	0	∞	1	5	∞
c course	∞	2	0	2	3	∞
d is	4	5	6	0	2	4
e not	∞	∞	∞	∞	0	2
f educational	∞	∞	∞	3	∞	0

iteration: n = 3, backtracking matrix n=c

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	a	b	c	b	c	–
b assignment	–	b	–	d	e	–
c course	–	b	c	d	e	–
d is	a	a	a	d	e	f
e not	–	–	–	–	e	f
f educational	–	–	–	d	–	f

Figure 15: Floyd-Warshall algorithm, iteration 0 to 3; left column matrix should contain weights after iteration n; right column matrix should be iteratively filled for backtracking each path

iteration: n = 4, weight matrix n=d

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	0	1	2	2	4	6
b assignment	5	0	7	1	3	5
c course	6	2	0	2	3	6
d is	4	5	6	0	2	4
e not	∞	∞	∞	∞	0	2
f educational	7	8	9	3	5	0

iteration: n = 4, backtracking matrix n=d

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	a	b	c	b	b	b
b assignment	d	b	d	d	d	d
c course	d	b	c	d	e	d
d is	a	a	a	d	e	f
e not	–	–	–	–	e	f
f educational	d	d	d	d	d	f

iteration: n = 5, weight matrix n=e

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	0	1	2	2	4	6
b assignment	5	0	7	1	3	5
c course	6	2	0	2	3	5
d is	4	5	6	0	2	4
e not	∞	∞	∞	∞	0	2
f educational	7	8	9	3	5	0

iteration: n = 5, backtracking matrix n=e

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	a	b	c	b	b	b
b assignment	d	b	d	d	d	d
c course	d	b	c	d	e	e
d is	a	a	a	d	e	f
e not	–	–	–	–	e	f
f educational	d	d	d	d	d	f

iteration: n = 6, weight matrix n=f

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	0	1	2	2	4	6
b assignment	5	0	7	1	3	5
c course	6	2	0	2	3	5
d is	4	5	6	0	2	4
e not	9	10	11	5	0	2
f educational	7	8	9	3	5	0

iteration: n = 6, backtracking matrix n=f

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this	a	b	c	b	b	b
b assignment	d	b	d	d	d	d
c course	d	b	c	d	e	e
d is	a	a	a	d	e	f
e not	f	f	f	f	e	f
f educational	d	d	d	d	d	f

iteration: n = 7, weight matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 7, backtracking matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

Figure 16: Floyd-Warshall algorithm, iteration 4 to 7; left column matrix should contain weights after iteration n; right column matrix should be iteratively filled for backtracking each path