Machine Learning F20

Assignment #32

Writer: João Barata (CS-IT8)
Study Number: 20200502

Note: The first 6 pages address Neural Networks and the remaining pages, SimRank. The source code for all the parts was implemented in python on PyCharm IDE, each part was submitted in a .zip file.

Topic: Neural networks

1. Give a brief formal specification of the building blocks of neural networks

The basic building unit in a neural network is the **neuron**. This unit receives inputs from other neurons or external sources that are multiplied by **weights.** This operation is then combined into a weighted sum and fed to an **activation function**, this kind of function introduces non-linearity into the network allowing it to deal with complex data. The activation function e.g. the **Sigmoid** or **ReLU** calculates the output of the neuron. A **bias** is a constant that can be added to the neuron, it allows for a shift on the activation function, allowing for a better fit of the given data.

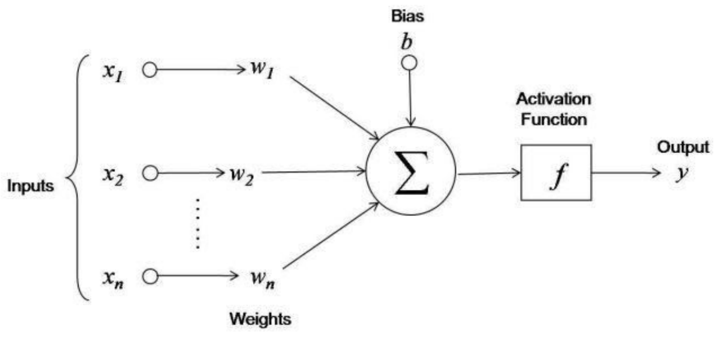


Figure 1 - Neuron Operations

Having as a reference (image1)[1], the operations above can be mathematically defined as:

$$S = \sum_{n} x_n \,.\, w_n$$

Equation 1 - Weighted sum

$$y = f(S) + b$$

Equation 2 - Output of the neuron

[1] Source: https://towardsdatascience.com/first-neural-network-for-beginners-explained-with-code-4cfd37e06eaf

These basic building units are organized by **layers**, we can define 3 types of layers in a network:

- **Input layer**
- **Hidden layer**
- **Output layer**

The **input layer**, as the name suggests consists of neurons that hold the initial data fed to the network.
The **hidden layer** consists of hidden units that receive the outputs from the previous layer and execute activation functions on the input and weights' sum.
The **output layer** is the final layer of the neural network and takes the input passed from the hidden layers to produce the final classification of the network.
These layers form the structure of a neural network.

2. Choose a concrete data set (either a real data set or a synthetic data set that you have made yourself) and define/illustrate the structure of a neural network for classifying the data.

The dataset chosen for the experiment was the MNIST dataset, it is composed of 60,000 training samples and 10,000 test samples. Each of these samples is a 28x28px image. The reason behind this choice was due to the familiarity with the data, as it was the same dataset used on self-studies throughout the semester.

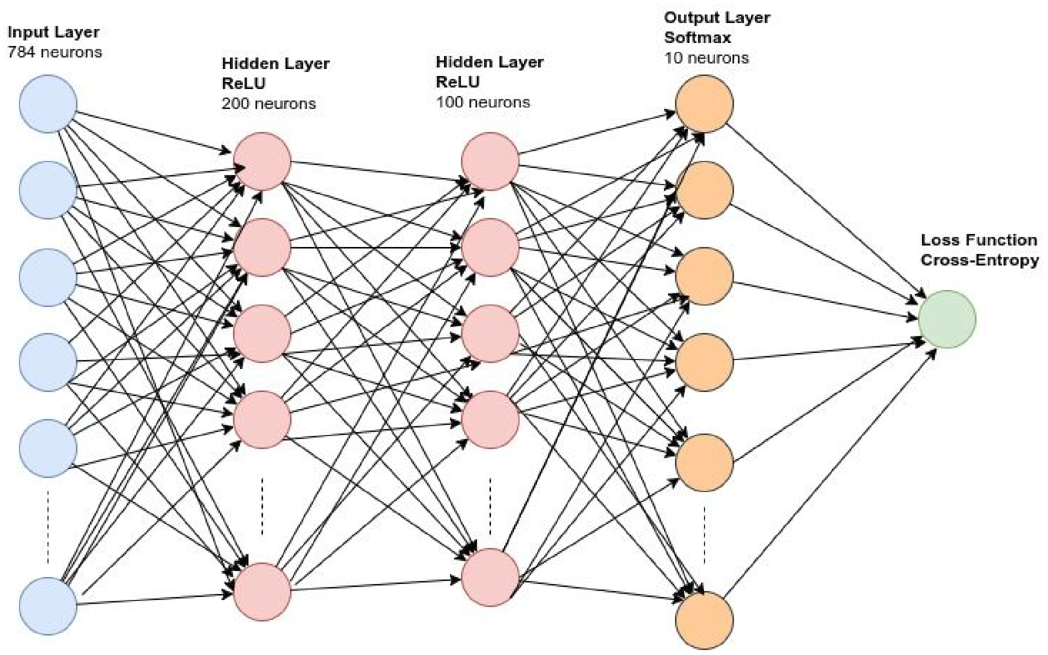


Figure 2- Network Structure

The network structure is depicted in Figure 2, the diagram was drawn on an online editor. The input layer is composed of 784 inputs which corresponds to the 28 x 28 pixels from each image. There are 2 hidden layers composed of 200 and 100 neurons, respectively. The number of neurons in the hidden layers should be kept between the number of neurons in the input and the output layers, with this in mind, a couple of values were tried and the ones mentioned before served the purpose of this experiment quite well. The activation function used in both layers was a **ReLU** function, chosen due to the faster computation time when compared to the **sigmoid** or **tanh** functions.

The output layer consists of 10 neurons representing each one of the 10 numerical classes, that represent the possible numbers. A **softmax** function was applied in the last layer to make sure the probabilities summed up to 1 in the end. Overall, this structure was chosen due to higher accuracy when compared to previous experiments.

3. Define formally a suitable loss function for training your neural network.
The loss function used was the **Cross-Entropy** function. Considering *n* classes , *t* target values and *p* output values, formally:

$$Loss(t,p) = -\sum_{x=1}^{n} t_x \,.\log(p_x)$$

Figure 3 - Cross- Entropy function

Practically speaking, this function was chosen as it is prefered for classification problems when compared to **MSE**(Mean Squared Error). On a learning point of view, without going into mathematical details, this function deals with the vanishing gradient problem preventing it from stalling.

4. Describe the basis of neural network learning, including the back propagation algorithm.
The learning process in a neural network can be divided into 2 phases, **forward propagation** and **back propagation**. In the first phase, the neurons calculate and apply transformations on the training data to output predictions. After the predictions are calculated, a **loss function** is applied to measure the error between these and the true values. The main goal of the learning process is to minimize the loss between these 2 values, therefore increasing the accuracy of the results.
So after the loss is calculated, this value will be propagated backwards consisting of the second phase of the learning process, the **back propagation**. The propagation of this value is calculated by taking into account the contribution that each neuron had on the initial output, mathematically speaking, by calculating the partial derivatives using the **chain rule**. As this learning process can be seen as an optimization problem, after propagating the information mentioned before, an optimization technique is applied to minimize the loss function. **Gradient descent** is one of these techniques, it updates the weights of the network in the opposite way of the gradients calculated by backpropagation, moving towards a global minimum. The equation in Figure 4 represents the latter in mathematical terms.

$$w' = w - \eta \, \nabla w Loss(w)$$

Figure 4- Gradient descent update

5. Describe and illustrate how learning is affected by:
Note: The experiments regarding the learning rate and the momentum were implemented with the code of self-study 5 and not the final model, as my focus on this part was to understand how those parameters affected the general learning of a model.

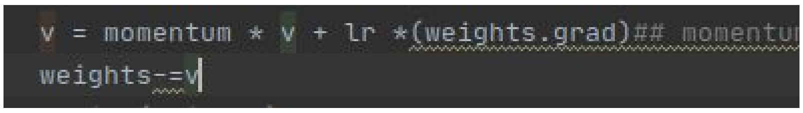
- The **learning rate**
  The learning rate (lr on some parts of the text) controls how much the weights are updated during the training phase as it was depicted in Figure 4 with the η parameter. In intuitive terms, how fast or how slowly our model is learning. A few experiments were conducted to infer how the tuning of this parameter affected the learning of the model. Namely, a high, a low and an average value was used. The results are depicted in the table below. The results were taken directly from the console output but the number of decimal cases was reduced to fit the table.

Epoch	Training/Valid Loss 0.5 lr	Training/Valid Loss 0.01 lr	Training/Valid Loss 0.0001 lr
0	Training loss: 2.29411 Valid loss: 2.29409	Training loss: 2.3080 Valid loss: 2.3080	Training loss: 2.30024 Valid loss: 2.30080
1	Training loss: 1.601891 Valid loss: 1.587363	Training loss: 1.9875 Valid loss: 1.97536	Training loss: 2.298991 Valid loss: 2.29949
2	Training loss: 1.57208 Valid loss: 1.56309	Training loss: 1.8850 Valid loss: 1.87043	Training loss: 2.2971 Valid loss: 2.2986
5	Training loss: 1.55824 Valid loss: 1.5523	Training loss: 1.7641 Valid loss: 1.7429	Training loss: 2.294631 Valid loss: 2.29505
10	Training loss: 1.5497 Valid loss: 1.54558	Training loss: 1.67689 Valid loss: 1.6556	Training loss: 2.289291 Valid loss: 2.2883
20	Training loss: nan Valid loss: nan	Training loss: 1.6325 Valid loss: 1.61436	Training loss: 2.2762 Valid loss: 2.27629
29	Training loss: nan Valid loss: nan	Training loss: 1.615 Valid loss: 1.5988	Training loss: 2.25931 Valid loss: 2.2591

Table 1 - Learning rate effect on learning

A few remarks regarding the effect of the learning rate on this experiment should be made. Firstly, when using a high learning rate, the training loss decrease is almost immediate and the model converges fast. But after some iterations the value increases exponentially to infinity. This is a typical divergent behavior when high learning rates are present. With an average value of 0.01 the results were expected, the loss steadily decreases as the model learns. With a very low learning rate, even though the training loss does decrease, it needs more time to learn because the size of the steps in each iteration towards the optimum value are extremely reduced.

- **Momentum**
  Momentum is a technique applied to accelerate the learning of the network, therefore increasing the training speed.



```
v = momentum * v + lr *(weights.grad)## momentu
weights-=v
```

Figure 5 - Python implementation of momentum

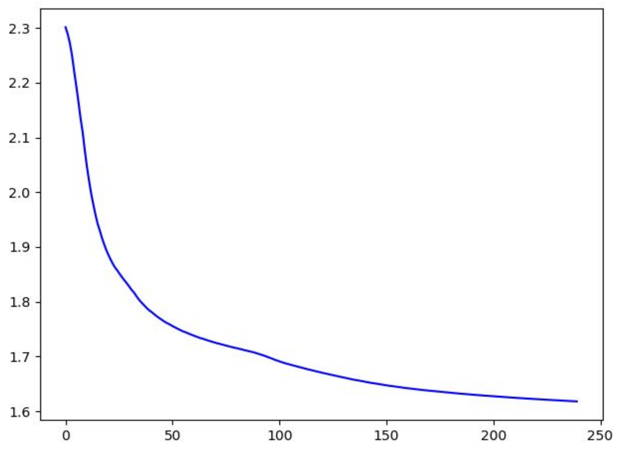


Figure 6 - Without momentum

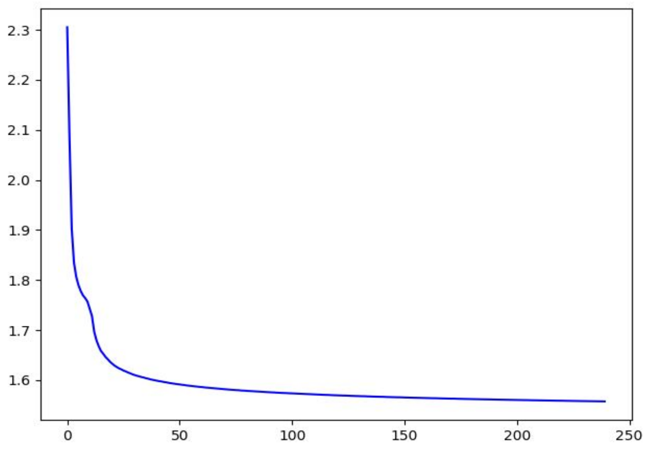
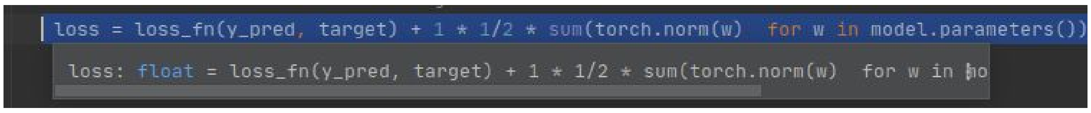


Figure 7 - With momentum

The figures above demonstrate the effect of momentum on the convergence of a model. The model on the right converged to the minimum loss value much faster than the model on the left. The value of the learning rate proved to be important because when applying momentum on the model this value has to be decreased. This is because the "steps" towards the minimum value become larger and so this must be balanced by using a lower value for the learning rate.

- **Weight Regularization**
  This type of regularization penalizes the learning of complex features and so by applying this technique during training, the model generalizes better. In other terms, it adds a penalization to the loss function calculation, producing a smoother version.



```
loss = loss_fn(y_pred, target) + 1 * 1/2 * sum(torch.norm(w)  for w in model.parameters())
  loss: float = loss_fn(y_pred, target) + 1 * 1/2 * sum(torch.norm(w)  for w in mo
```

Figure 8 - Regularization implementation in python

The results from this experiment showed that the training error did increase but the model failed to produce a higher accuracy. The figure below shows the increased training error and the smoothened version of the function.

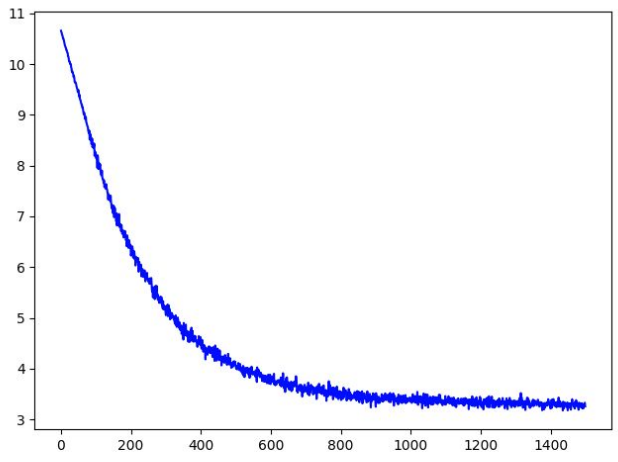


Figure 9 - regularization effect on training loss

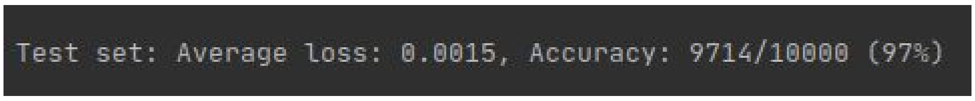
6. Explain the differences (if any) in the performances you observe.
The table below sums up the difference in the accuracies during these experiments. The number of epochs applied was 30.

	0.5 lr	0.01 lr	0.0001 lr	momentum	no momentum	Regularization
Accuracy	9.7 %	89.7 %	39.7 %	89.7%	89.9 %	77%

Table 2 - Accuracies with different parameter settings

Firstly, the results were expected when it comes to the learning rate tweaking. Having a high learning rate caused a divergence on the loss function and the training loss exponentially grew to infinite values as it is reflected on the accuracy of that model. With an average value, the accuracy value was high taking into consideration this model was not finely tuned and it was merely used to demonstrate the effect of these parameters on the overall behavior of the model. Finally, a low learning rate held a low accuracy score as the model could not learn enough with 30 epochs. The momentum did not seem to affect the accuracy of the model but as it was shown above, it did increase the training speed as it was supposed to. The regularization ended up decreasing the performance of the model. In my understanding, this could be because as the model penalized the learning of complex features on the dataset it might have missed some important patterns on the data and therefore ended up having a poor behavior on unseen data. Also, this implementation might be wrong because as far as I am concerned, PyTorch does not have any built-in regularization parameters.

7. Explain how you would evaluate the accuracy of your neural network.
After all the experiments and with the knowledge obtained from the previous questions, the final model was implemented with the structure presented in question 2. Adam optimizer was used instead of Stochastic Gradient Descent as it yielded better results on the test set. The learning rate had to be reduced, this is because Adam uses momentum to converge faster and so as I explained before, with the usage of momentum the learning rate must be smaller to compensate for the larger “steps” towards the minimum. The final number of epochs was 10. Overall, the model did a good job on predicting unseen data helding a final accuracy of 97 %.



```
Test set: Average loss: 0.0015, Accuracy: 9714/10000 (97%)
```

Figure 10 - Final model accuracy

This problem would be better suited for CNN’s but even so the results achieved here demonstrate that a FCNN can also deal with this kind of problems in a successful way. Comparing my model with the accuracies on the MNIST database, I conclude that there are more advanced techniques that can be implemented to achieve a better test accuracy and therefore a better model.

Topic: SimRank

Simrank is a similarity measure that computes the similarity between nodes in a graph. This algorithm is based on the graph's structural context, what this means is that it takes into account how its neighbours relate to other nodes. As it is stated on the paper from the lecture " objects a and b are similar if they are related to objects c and d, respectively, and c and d are themselves similar" , we can think of the objects here as being nodes on a graph. This is the reason why this approach is more successful when compared to other simpler similarity measures, that only execute local computations, often leading to a lack of information.
To understand the concept above, a set of experiments inspired by the questions of the self-study were conducted.

Describe and reflect upon your data you used to construct graph
The data used for the graph construction was inspired by the structure of the graphs represented on the papers. Two graphs were implemented to represent the **homogeneous** and **bipartite** domains. In the first case, the graph consists of 9 nodes and 15 edges. We can think of these nodes as representing a CS-IT8 semester. There are 4 nodes representing courses from CS-IT8 F20 semester, 3 nodes representing professors, 1 node that represents the university and 1 node that represents the student. The edges represent plausible relations between the nodes, for example the student has 4 outgoing edges to each one of the courses and incoming edges from the professors. On the bipartite domain, the structure is very similar to the one from the paper, it's simply composed of 8 nodes but it has 2 types, students and courses. The overall structure was chosen, firstly, because for the results visualization and interpretability a small graph would be the best to understand the underlying concepts of Simrank. Secondly, as I had already implemented the graph from the paper and compared the results with the excel file, having a graph with a similar structure would ease the implementation and if needed debugging of the code.

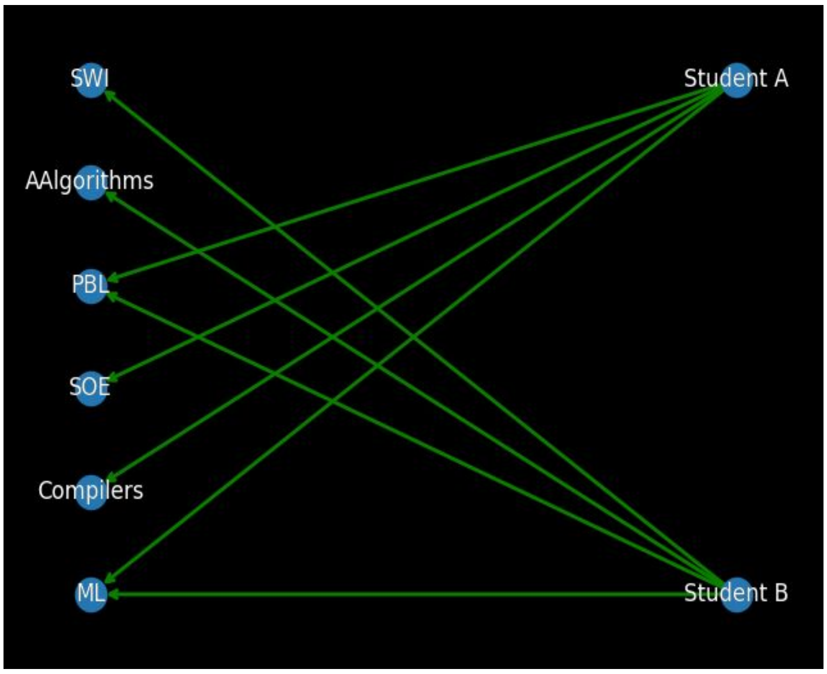


Figure 1 - Bipartite graph

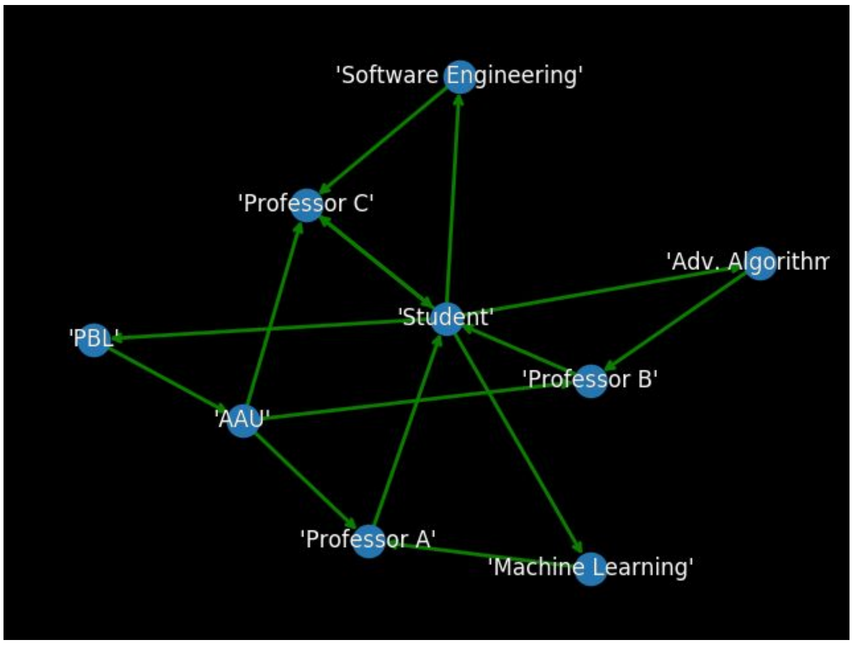
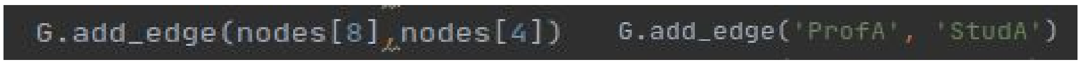


Figure 2 - CS -IT8 graph

• Describe and reflect upon your implementation of loading the graph into memory
The implementation was done in 2 distinct ways for each of the domains. For the homogeneous domain, the nodes were written in a .csv file and using a for loop, each row was added as a node of a directed graph. The edges were then added, having as a reference the list of nodes and the connections drawn on a paper beforehand. For the bipartite domain, 2 lists representing each type (the students and the courses) were implemented and filled with the respective nodes. With this, the edges were added by simply referring to the index of the list that contained the desired node. The graphs were drawn with the aid of the *networkx* library, that allows to draw graphs with suitable layouts, "*net.bipartite_domain_layout(graph)*" for example. The implementation is not difficult due to the extensive number of python libraries available. However, when it comes to the graph edges, instead of using the list of nodes as a reference, one can simply code the node directly in the edge instead of going through the work of manipulating lists indexes. An example is shown below:



```
G.add_edge(nodes[8],nodes[4])    G.add_edge('ProfA', 'StudA')
```

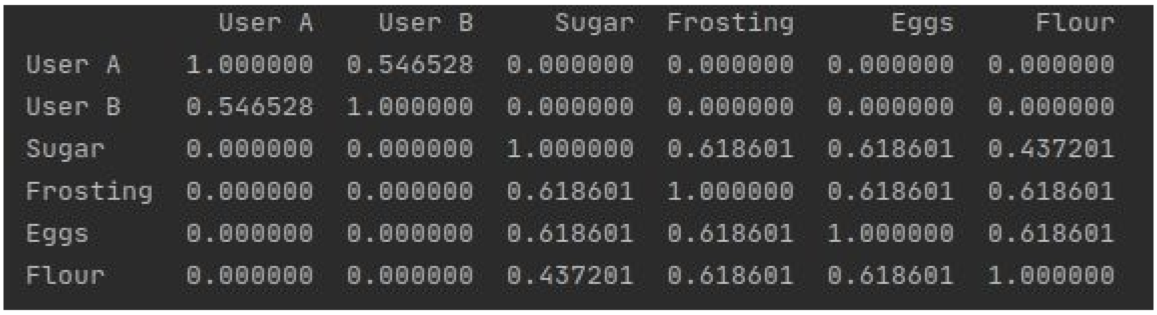
Figure 3 - Original Approach / Improvement

Describe and reflect upon your implementation of Simrank and where you cut corner
The implementation of the Simrank on both graphs was done using available libraries in python, *networkx*[2] and *graphsim*[3] . The algorithms implemented on these libraries are based on the paper from the class. This was decided to avoid the time consuming task of debugging the code that could be instead spent on understanding the concepts of Simrank. A few minor changes were implemented on the source code of the library, for example the default weight-decay constant is 0.9, it was changed to 0.8 to match the results of the experiments in the excel file and a print line for every iteration was added as a way to visualize the convergence of the model. The first experiment

2 https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.similarity.simrank_similarity.html

3 https://pypi.org/project/graphsim/

with this code was done with the graphs from the paper and compared with the results from the excel file available in moodle.



```
            User A    User B     Sugar  Frosting      Eggs     Flour
User A    1.000000  0.546528  0.000000  0.000000  0.000000  0.000000
User B    0.546528  1.000000  0.000000  0.000000  0.000000  0.000000
Sugar     0.000000  0.000000  1.000000  0.618601  0.618601  0.437201
Frosting  0.000000  0.000000  0.618601  1.000000  0.618601  0.618601
Eggs      0.000000  0.000000  0.618601  0.618601  1.000000  0.618601
Flour     0.000000  0.000000  0.437201  0.618601  0.618601  1.000000
```

Figure 4 - Similarity results from paper example

As we can see from the dataframe above, e.g, if we look at the item frosting and its similarities, we can see *similarity(frosting,flour) < similarity(frosting,eggs)* which can be explained by looking at the graph of the paper, since the first 2 items are not referenced by the same user, their similarity score is lower than frosting and eggs, which are referenced both by user A. As the implementation seemed to produce correct results it was used for the final graph.

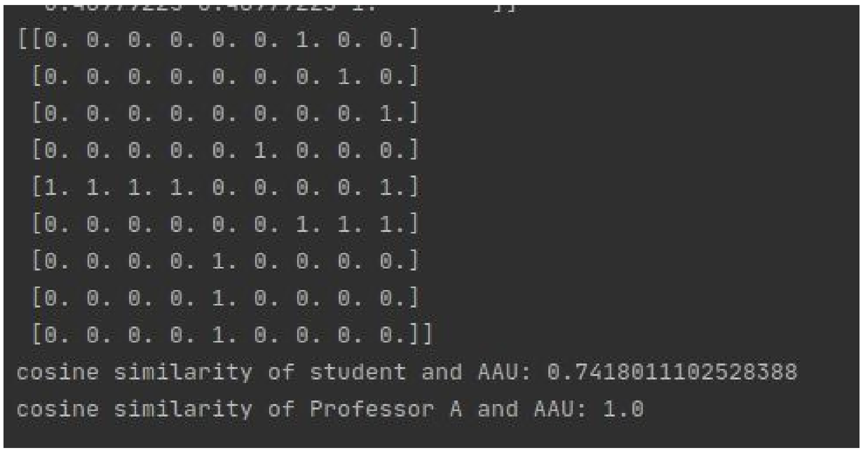
Describe and reflect upon your implementation of other simpler similarity measure

The measures used were **Pearson correlation** and **cosine** similarity. Regarding the cosine similarity, it computes the normalized dot product between two feature vectors. As it it mentioned on the slides, if we take a feature vector X we have the equation:

$$Sim(v_i, v_j) = Xi \cdot Xj / ||Xi||*||Xj||$$

Figure 5 - Cosine Similarity

An adjacency matrix of the graph was created with every row of the matrix representing a feature vector which has all the connections of a specific node represented by 1. Secondly, the cosine similarity was calculated by using the *sklearn.metrics*.

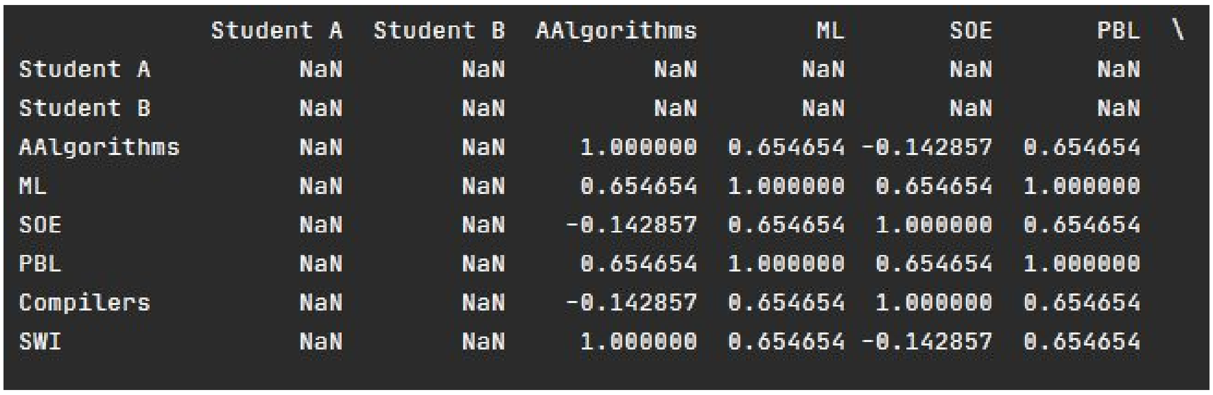


```
[[0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [1. 1. 1. 1. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0.]]
cosine similarity of student and AAU: 0.7418011102528388
cosine similarity of Professor A and AAU: 1.0
```

Figure 6 - Cosine Similarities and Adjacency Matrix

The full matrix of the cosine similarities was truncated during the output so only 2 examples are displayed here.

For the bipartite graph I used Pearson correlation. Usually this correlation measure is applied to infer how the variables on a dataset are related, here I assume my variables are the course nodes and I want to calculate how correlated they are. To do this, I transformed my graph into a dataframe using the *pandas* library and used the built in method *df.corr('pearson')*.



```
              Student A  Student B  AAlgorithms        ML       SOE       PBL  \
Student A           NaN        NaN          NaN       NaN       NaN       NaN
Student B           NaN        NaN          NaN       NaN       NaN       NaN
AAlgorithms         NaN        NaN     1.000000  0.654654 -0.142857  0.654654
ML                  NaN        NaN     0.654654  1.000000  0.654654  1.000000
SOE                 NaN        NaN    -0.142857  0.654654  1.000000  0.654654
PBL                 NaN        NaN     0.654654  1.000000  0.654654  1.000000
Compilers           NaN        NaN    -0.142857  0.654654  1.000000  0.654654
SWI                 NaN        NaN     1.000000  0.654654 -0.142857  0.654654
```

Figure 7 - Pearson Correlations (truncated)

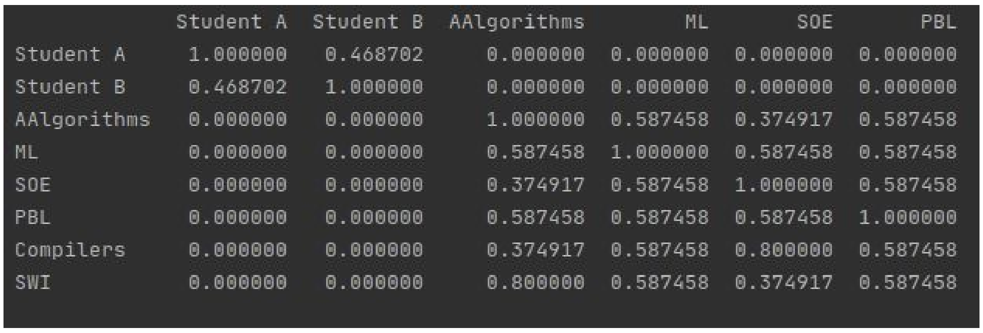
Overall, with the cosine similarity, as it is a local measure, the coverage is not as good if you compare it with SimRank. For example, the similarity was maximum between Professor A and AAU because they are directly connected, it doesn't take into account other connections within the neighbours. Regarding Pearson correlation, e.g. looking at the PBL course, it has a maximum correlation with the ML course because they have a relation with both students A and B. On another note, AAlgorithms is negatively correlated with SOE because they are connected to different students, this is also reflected in the relation between SWI and SOE.

Describe a metric which you used in experimentation and comparison of SimRank and another similarity measure

For the bipartite graph, the metric used was inspired by the one from the paper. Simply put, considering *c1* and *c2* as two courses, and defining the metric as *m*, *m(c1,c2) ≈ 1* if they belong to the same student and *m(c1,c2) ≈ 0* if they don't belong to the same student. For the other graph, I could not define a specific metric so the evaluation was done empirically by observation of the connections of the graph.

Reflect upon results

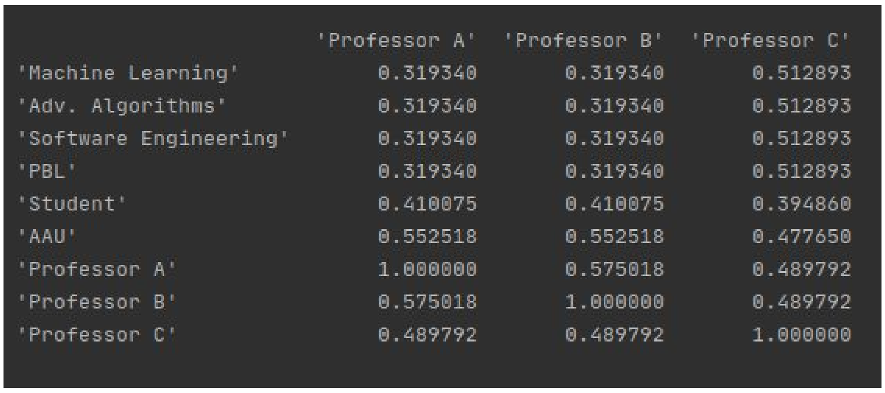
The final results with simrank on the graphs proved to be coherent with the theoretical concepts of the algorithm. Both domains were subject of experimentation and in relation to the metrics defined on the previous question, produced good results. The examples are truncated as the output matrices were too large to present on the assignment.



```
             Student A  Student B  AAlgorithms        ML       SOE       PBL
Student A     1.000000   0.468702     0.000000  0.000000  0.000000  0.000000
Student B     0.468702   1.000000     0.000000  0.000000  0.000000  0.000000
AAlgorithms   0.000000   0.000000     1.000000  0.587458  0.374917  0.587458
ML            0.000000   0.000000     0.587458  1.000000  0.587458  0.587458
SOE           0.000000   0.000000     0.374917  0.587458  1.000000  0.587458
PBL           0.000000   0.000000     0.587458  0.587458  0.587458  1.000000
Compilers     0.000000   0.000000     0.374917  0.587458  0.800000  0.587458
SWI           0.000000   0.000000     0.800000  0.587458  0.374917  0.587458
```

Figure 8 - Bipartite Simrank results

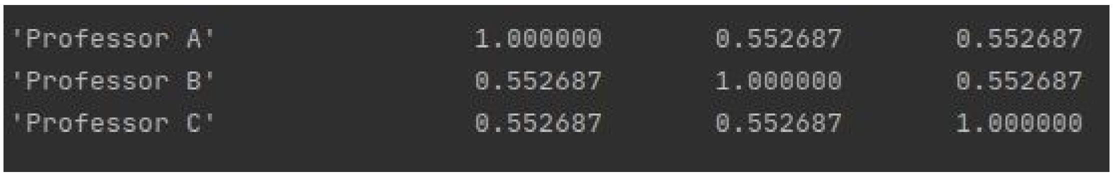
Taking a closer look on the similarities between SWI and 4 of the other courses, the highest score is with AAlgorithms as they are both and only from student B, for ML and the PBL course they score the same as they are both shared amongst the two students and the lowest score with SOE as it is exclusive of student A.



```
                          'Professor A'  'Professor B'  'Professor C'
'Machine Learning'             0.319340       0.319340       0.512893
'Adv. Algorithms'              0.319340       0.319340       0.512893
'Software Engineering'         0.319340       0.319340       0.512893
'PBL'                          0.319340       0.319340       0.512893
'Student'                      0.410075       0.410075       0.394860
'AAU'                          0.552518       0.552518       0.477650
'Professor A'                  1.000000       0.575018       0.489792
'Professor B'                  0.575018       1.000000       0.489792
'Professor C'                  0.489792       0.489792       1.000000
```

Figure 9 - Simrank results

In this case, a sample from the matrix representing the similarities with professors is shown. E.g, Professor A and B are more related to each other than Professor C, because on my implementation of the graph the student also has an outgoing edge towards that particular professor (meaning for example that it is his supervisor) and so that may have affected the similarity scores between the Professors. This connection seems to have “biased” the similarity scores of the courses towards Professor C, changing it to ‘Professor C’ ->’Student’ would yield the following results:



```
'Professor A'                1.000000     0.552687     0.552687
'Professor B'                0.552687     1.000000     0.552687
'Professor C'                0.552687     0.552687     1.000000
```

Figure 10 - New similarity

So, all the professors would have the same similarity scores.
Overall, SimRank proved to be a superior similarity measure when compared to local based measures like cosine similarity.