SCC462 – Distributed Artificial Intelligence

Leandro Soriano Marcolino

Midterm Assignment

Deadline: 11/03/2022, 4pm. 25 points in total.

Instructions

- This assignment consists of essay questions and coding questions.
- Please submit your replies on the Quiz available on Moodle. You can only submit one time.
- The Quiz closes on 14/03/2022, 4pm, in order to allow for delayed submissions. However, submissions made after 11/03/2022, 4pm, will have a 10% penalty in the grade. Your attempt will be automatically submitted on 14/03/2022, 4pm, if you do not click to finish the attempt on Moodle.
- Coding questions must have many comments. Please use the Precheck button to check if you made enough comments in your code. Solutions without enough comments will not be accepted. Note that the system only recognises the # character for comments.
- Coding questions will be checked against test cases, and will be pre-graded based on the weight assigned to each test that successfully passes. The Moodle Quiz shows some tests as an example (usually one test), and the Precheck button will check against the example test with no penalties. There are (many) more tests that are not visible to you now.
- There is no "Check" button. Once you submit, your code will be checked against all test cases. The final grade is given by me, not by the system.
- There are no test cases where the input has wrong *type*. However, the input could be such that a calculation is impossible due to theoretical reasons.
- This is an **INDIVIDUAL** assignment. You must study the course contents, and write your replies and your code based on your understanding.
  - Solutions copied from another student will not be accepted, even if they have minor modifications.
  - Solutions found on-line will not be accepted, even if you include a reference.
  - You are allowed to check references only for:
    - Fundamental Python information. For example, how to handle lists of lists, how to randomly sample a number between 0 and 1 using the standard library, etc.
    - Theoretical information. For example, what is the formal definition of a Nash Equilibrium, what is the pseudo-code of AdaBoost (not Python code), etc.
  - Hence, Python code obtained on-line that, for example, implements AdaBoost, calculates Nash Equilibria, etc, will not be accepted.

1. Write a rejection review of Marcolino, Passos et al (2016). That is, you must summarise the paper, explain why it is not yet suitable for publication, and give suggestions to the authors for further improvement. (10%)

2. According to Schapire (1990), each recursive application of the Boosting algorithm reduces the error of a classifier from x to $3x^2 - 2x^3$ (see Figure 1 in the paper). Based on this, write a Python function $howManyIterations(x, epsilon)$, which returns how many recursive iterations of the Boosting algorithm would be needed to reduce the error of a base classifier from x to a value lower than $epsilon$. In case the number of iterations cannot be calculated, then the function must return $False$. (10%)

3. Which variation is proposed by Schapire (1990) in order to make the Boosting algorithm run faster than its original version? Why is it faster? (5%)

4. Assume you are given a class $WeakClassifier$ for a Weak Classifier in a binary classification problem, with the following methods:

   - $train(dataset, labels, p)$: trains a $WeakClassifier$ object, using the $dataset$ and the ground truth labels $labels$. The training set will be sampled using a probability distribution p over items. We assume that $dataset$ is a list of lists, where each list corresponds to one item to be classified. Each item's label is either 0 or 1.

   - $classify(item)$: returns a label predicted by the $WeakClassifier$ object. For the purposes of this exercise, we will use the index of the item in the $dataset$ as $item$, not the list with the features of the item.

   Using that class, implement in Python the AdaBoost algorithm, following Freund and Schapire (1996). You must write your own code, solutions from on-line resources are not going to be accepted.

   Write a class $AdaBoost$, with the following methods:

   - $train(dataset, labels, D, T)$, which trains the AdaBoost system with T classifiers, given a dataset in $dataset$ and the corresponding ground truth labels in $labels$. D is the starting distribution over the examples, as in the AdaBoost paper.

   Attention: differently from the original algorithm, you must re-train a Weak Classifier if the error obtained after training would be such that the probability of correctly classified items would increase.

   - $classify(item)$, which classifies an $item$ using the trained AdaBoost system. Again, we will use the index of the item in the $dataset$ as $item$, not the list with the features of the item.

   Additionally, for the purposes of this exercise, we will use a fake $WeakClassifier$, see the file WeakClassifier.py attached. The code for the Weak Classifier will be automatically added to the beginning of your code. (15%)

5. In Freund and Schapire (1996), how does the on-line allocation problem and proposed solution (first part of the paper) relate to the development of the AdaBoost algorithm (second part of the paper)? (10%) AdaBoost

6. Open BI has an ensemble system composed of n neural networks. They use one-hot encoding in a classification problem, with l possible labels. That is, at training time, assuming for example 3 labels, the label 0 would be represented as [1, 0, 0],

label 1 as [0, 1, 0], and label 2 as [0, 0, 1], where each element is the desired output of the corresponding output neuron. At inference time, however, each neuron returns a value between 0 and 1. These neural networks have a softmax layer in the end, so their output can be seen as a probability distribution function.

You were asked to interpret the output of the neurons as a *ranking*, and write a function to aggregate the rankings using the **Borda voting rule**. Additionally, all ties must be broken randomly. When deciding the winner of a tie, please use the function *choice* from the *random* module.

Hence, you must implement $bordaAggregation(opinions)$, where:

- *opinions* is a list of lists. Each inner list corresponds to a neural network, and each element of that list corresponds to the output of an output neuron. For example, $opinions = [[0.4, 0.5, 0.1], [0.2, 0.3, 0.5]]$ represents a system with two neural networks, where the first one assigns value 0.4 to label 0, 0.5 to label 1, and 0.1 to label 2. Similarly, the second neural network assigns value 0.2 to label 0, 0.3 to label 1, and 0.5 to label 2. Note that in general we can have any number of neural networks and labels/output neurons.

- The function returns *ranking*, where each element corresponds to the final ranking position. For instance, $ranking = [2, 0, 1]$ would mean that label 2 is in the top position of the final aggregated ranking, label 0 in the middle position, and label 1 in the last position. (10%)

7. Giggle wants to apply Stacked Generalisation to improve its predictions about customer behaviour. Currently they are using a Logistic Regression class ($LogisticRegression$), which has the following interface: Lec > Polikar

   - $LogisticRegression(nFeatures, alpha = 0.15, threshold = 0.5, nEpochs = 200)$: Class constructor. The first input is the number of features in the dataset, and *alpha* is the learning rate. If *alpha* is not passed, it uses 0.15. The hyper-parameter *threshold* is used to define whether the output of the logistic regression will be interpreted as a label 0 or 1. That is, outputs lower than or equal to *threshold* will be defined as 0 when classifying items. *nEpochs* defines the number of epochs that will be used during training.

   - $train(dataset, labels)$: Trains the Logistic Regression classifier, using the dataset *dataset* and the corresponding labels *labels*.

   - $classify(item)$: Returns a predicted label for the item *item*, using the trained Logistic Regression classifier.

   This classifier is already given to you, see the file LogisticRegression.py attached. It will be automatically added to the beginning of your code.

   You were hired to implement the class StackedGeneralisation for Giggle. Although Stacked Generalisation can work with a diverse set of classifiers, we will use only Logistic Regression as our base classifiers and as our aggregator, for their initial tests.

   We will use blocks of size 1. That is, the training set will be divided in n blocks of 1 item each. In order to match with their test cases, please generate the training set of the aggregator following the original ordering of the training set, and train the classifiers in order. That is, when training the base classifiers, train first the first classifier, followed by training the second classifier, then the third classifier and so on. Similarly, when creating the dataset for the aggregator, please generate first the first row, then the second row, etc.

Hence, your task is to implement the StackedGeneralisation class with the following methods:

- $StackedGeneralisation(classifiers, aggregator)$: Class constructor. Receives a list $classifiers$ of classifier objects, in order. That is, the first element corresponds to the first classifier, the second to the second classifier and so on. Additionally, receives a classifier object $aggregator$, which will learn the aggregation rule.

- $train(dataset, labels)$: Trains the Stacked Generalisation system, given a dataset in $dataset$, and the corresponding labels in $labels$. The $dataset$ will be represented as a list of lists, where each inner list is an item, and each element of the inner lists are features of the item.

- $classify(item)$: Classify the item in $item$ using the trained StackedGeneralisation system. It returns a label (0 or 1). (15%)

8. Consider the team/ensemble success prediction system described in Marcolino, Lakshminarayanan et al. (2016). Let's assume that we have an ensemble of 3 classifiers (c_0, c_1, c_2), working across batches of 4 items, in a problem with 3 possible labels (0, 1 or 2). The final decision of the ensemble is given by the plurality voting rule.

   For each batch of items, we store the vote of each agent for each item, the final classification and the corresponding ground truth label, in a list of lists. Using this data, and the Logistic Regression class provided, create a classifier to predict the success of the ensemble. You must implement two functions: $predictionFeatureVector(batch)$ and $successPrediction(batches)$.

   $predictionFeatureVector(batch)$ is an auxiliary function that returns the feature vector of the prediction methodology, given the stored information for one batch of items. In detail:

   - $batch$ is a list of lists, where each list corresponds to one item within the batch. For each item, we first list the votes of each classifier, then the decision taken by the team, and finally the ground truth label. For example, $batch = [[0, 1, 0, 0, 1], [1, 1, 0, 1, 1], [0, 1, 2, 0, 0], [0, 0, 0, 0, 0]]$ shows a case where the votes for each item were, respectively: $[0, 1, 0], [1, 1, 0], [0, 1, 2], [0, 0, 0]$. Additionally, the decision for each item was: 0, 1, 0, 0, respectively. Finally, the ground truth labels were 1, 1, 0, 0.

   - The function returns a list $featureVector$, where each element corresponds to a subset of the classifiers. We will use the following ordering across possible subsets: $\{c_0\}$, $\{c_1\}$, $\{c_2\}$, $\{c_0, c_1\}$, $\{c_0, c_2\}$, $\{c_1, c_2\}$. Hence, $featureVector = [1/4, 0, 0, 1/4, 1/2, 0]$ would mean that in this batch classifier $\{c_0\}$ was responsible for 1/4 of the final team decisions, $\{c_0, c_1\}$ for 1/4 of the decisions, and $\{c_0, c_2\}$ for 1/2 of the decisions.

   Concerning $successPrediction(batches)$:

   - $batches$ is a 3D Python list (list of lists of lists), containing a list of batches, as described above. For example, $batches = [[[0, 1, 0, 0, 1], [1, 1, 0, 1, 1], [0, 1, 2, 0, 0], [0, 0, 0, 0, 0]], [[0, 0, 0, 0, 0], [1, 1, 2, 1, 2], [0, 1, 1, 1, 1], [2, 2, 1, 2, 2]]]$ contains information about two batches of items. The first batch is $[[0,1,0,0,1],[1,1,0,1,1],[0,1,2,0,0],[0,0,0,0,0]]$ as described above, and the second batch is $[[0,0,0,0,0],[1,1,2,1,2],[0,1,1,1,1],[2,2,1,2,2]]$.

- The function returns an object of type Logistic Regression, with the trained classifier. Use default learning rate, threshold for classification, and number of epochs when training in the Logistic Regression.

The code for the Logistic Regression classifier will be automatically added to the beginning of your code. (15%)

9. Consider we have a game between two agents A and B, with two possible actions, as follows:

A/B	a_0	a_1
a_0	$r_{0,0}$, $r'_{0,0}$	$r_{0,1}$, $r'_{0,1}$
a_1	$r_{1,0}$, $r'_{1,0}$	$r_{1,1}$, $r'_{1,1}$

Reward $r_{i,j}$ is the reward the Row player (agent A) receives when it takes action a_i and Column player (agent B) takes action a_j. Similarly, $r'_{i,j}$ corresponds to the reward for the column player (agent B) when A takes a_i and B takes a_j. We will represent that table in Python as a list of lists, where each inner list corresponds to a row, as follows: $[[(r_{0,0},\ r'_{0,0}),\ (r_{0,1},\ r'_{0,1})],\ [(r_{1,0},\ r'_{1,0}),\ (r_{1,1},\ r'_{1,1})]]$.

(a) Write in Python a function *nashEquilibria*(*rewards*), which returns all the pure strategy Nash Equilibria in the game. *rewards* is a list of lists with the rewards for each player, as defined above. The function returns a list of tuples, where each tuple is a pair of actions for each player if that pair is a Nash Equilibria of the game. For instance, $[(0,0),\ (1,1)]$ means that there are two Nash Equilibria in the game: when A takes action a_0 and B takes action a_0, and when A takes action a_1 and B takes action a_1. If the game has no pure strategy Nash Equilibria, then it returns an empty list: []. (5%)

(b) Write in Python a function *mixedEquilibria*(*rewards*), which returns a mixed strategy Nash Equilibria in the game defined by the rewards in *rewards*. The function returns a tuple (x, y) where x is the probability of the A player playing a_0 and y is the probability of the B player playing a_0. If there is no mixed strategy Equilibria, then it returns an empty tuple: (). (5%)