Black Rhino

Example 5: Degroot learning model

<http://socsci2.ucsd.edu/~aronatas/project/academic/degroot%20consensus.pdf>

Documentation

1. Economics of the model[[1]](#footnote-1):

Simple Degroot learning model with three individuals which are pooling their opinion to reach a consensus. The DeGroot model is a rule-of-thumb type of social learning process. It takes a society of n agents where everyone has an opinion on a subject, represented by a vector of probabilities . Agents obtain no new information based on which they can update their opinions but they communicate with other agents. Links between agents and the weights they put on each other’s opinions is represented by a matrix T where is the weight that agent puts on agent opinions. The matrix is a one-to-one relationship with a weighted, directed graph where the agents are the nodes and the edges represent the probabilities associated with their opinions.

The agent’s opinions are updated each period as so the th period opinions are related to the initial opinions by . The idea is that at some point, the agents converge to one belief/opinion. This is done with the usage of the Markov chain theory under which the limit exists for any initial beliefs If there is a group C of individuals, they reach a consensus if for any . When there is a strongly connected group, they reach a consensus for every initial vector of beliefs if and only if they reach a consensus inside the groups but there is not necessarily a consensus at the society level.

1. Running:

**python abm\_degroot.py**

1. Folders & Files:
   1. **Abm\_degroot.py** – The main file calling the example to be run. It has hardcoded arguments (line 37) args = ["configs/environments/", "test\_degroot", "log/"] which point to the environmentdirectory, the identifier of the simulation *(i.e. the identifier of the environment config file),* and the log directory.

This script

* initializes the environment via environment.initialize(environment\_directory, identifier), which is a function that reads the environment congif file and initializes the agents
* initializes the runner via runner.initialize(environment),which is a function that calls the updater class and assigns the parameter num\_sweeps to the num\_sweeps variable
* and does the update with runner.do\_run(environment)

All of this is done number of times, where is set in the num\_simulations parameter in the *environment config file.*

Note: In the beginning, we use if \_\_name\_\_ == '\_\_main\_\_': to only run the script if it’s the main script and not if it’s imported. We also import the Runner and Environment class from src.environment and src.runner

* 1. **abm\_template**/ – Folder containing the current (at the time of creating the example) abm\_template package, i.e. the abstract classes upon which the source code is based, providing super-classes and super-methods which are inherited
  2. **config/agents/ -** Folder containing config files for all agents. The Config file for agent i is an xml files with parameters describing the “starting opinion” of the unknown parameter. and the weights assigned to . Every It is essential that identifier is in the config files, and that identifiers are unique (not only among certain kinds of agents, but globally across agents, otherwise it will raise errors). The starting wrapper <agent></agent> can in principle be anything, but it’s good practice to keep them with the same names as the agent it belongs to. Parameters are within <parameter></parameter>, and require a type (static or *transition*), a name (string), and a value.

<agent identifier='agent\_one’>

<parameter type='static' name='starting\_opinion' value='1'></parameter>

<parameter type='transition' name='agent\_one' value='0.5'></parameter>

<parameter type='transition' name='agent\_two' value='0.25'></parameter>

<parameter type='transition' name='agent\_three' value='0.25'></parameter>

</agent>

* 1. **config/environments/** – Folder containing the *environment config* file for the simulation, an xml file, see the example below. In our environment config file test\_degroot.xml parameters are wrapped inside <parameter></parameter> and require type and value, i.e. the number of simulations and number of sweeps per simulation, as well as the number of agents and the directories which store the config files for the agent and measurement script.

<environment identifier='tests\_for\_degroot\_learning'>

<!-- simulation parameters -->

<parameter type='num\_sweeps' value='5'></parameter>

<parameter type='num\_simulations' value='2'></parameter>

<parameter type='num\_agents' value='3'></parameter>

<parameter type='agent\_directory' value='configs/agents/'></parameter>

<parameter type='measurement\_config' value='configs/output\_opinion.xml'></parameter>

</environment>

* 1. **networkx/** Folder containing the networkx package which enables python to create Directed Graphs. We create nodes for the individual agents and add edges which store the transition probabilities.
  2. **src/** – source files for the simulator:
     1. **init.py –** this file is required, so python is able to look through the src folder directory
     2. **agent.py** – a class for the individuals, dubbed agents, characterised by an identifier (should be a unique string), parameters (dictionary), state\_variables (a dictionary, not used in this example), and accounts (for storing transactions; not used in our example, but we need to include it because we inherit from the BaseAgent class).

There are standard methods, inherited from BaseAgent, which can be used to change variables. The variables can be changed with get/set standard functions as described in the file. There are functions for printing the agent (\_\_str\_\_) and the parameters. Implemented \_\_getattr\_\_ function means that parameters don’t need to be extracted from the dictionaries by hand (e.g. agent.parameters[“name\_of\_parameter”]), but can be called as an attribute directly (e.g. agent.name\_of\_parameter). This means that variable names need to be unique among parameters and state\_variables.

We need to read the agent config file and assign probabilities as edges to our Directed graph which stores the trust matrix.

The method def create\_temp\_variable(self, environment)is the method specific to our degroot learning model. It iterates over every agent in the agent list in environment, takes its opinion variable, multiplies it with the weight stored in the transition\_probabilities network graph and assigns it to the temporary variable tempv. The method return is needed to pass the tempv variable back to the updater script.

* + 1. **environment.py** – a class for the environment (inherited from BaseConfig from abm\_template), that is the global parameters of the simulation. It’s characterised by an identifier (should be a unique string), static\_parameters (dictionary), variable\_parameters (a dictionary, not used in this example), and accounts (for storing transactions). In particular, we have parameters for the number of simulations to run (num\_simulations), number of sweeps (steps) per simulation (num\_sweeps), number of agents (num\_agents), and the directories in which configs for specific agent types are stored (agent\_directory). Environment also has a list containing the agents. There are standard get/set functions for these variables. There are standard functions for printing the environment, the parameters, and writing the config file (\_\_str\_\_, print\_parameters, write\_environment\_file). Function read\_xml\_config\_file reads the xml file with parameters. When the environment is initialized, it zeroes out all the variables, reads the config file from the supplied directory, and creates all the agents from their respective directories (this utilizes initialize\_agents\_from\_files). Implemented get\_agent\_by\_id function returns an object with the agent with specified identifier. This function requires all agents to have unique identifiers (it is recommended their identifiers are prefixed with the type of the agent, e.g. agent\_id) Implemented \_\_getattr\_\_ function means that parameters don’t need to be extracted from the dictionaries by hand (e.g. environment.static\_parameters[“name\_of\_parameter”]), but can be called as an attribute directly (e.g. environment.name\_of\_parameter). This means that variable names need to be unique among parameters and state\_variables.
    2. **runner.py** – a class for handling the actual running of the model (inherited from BaseRunner from abm\_template), initializes the updater and runs the updating loop in the updater with self.updater.do\_update(environment)i number of times, where i is num\_sweeps parameter from the *environment config file*. It also writes the sweep number and the updated agent\_opinion to a csv file specified within the Measurement class. Users can also use the runner script to modify parameters, e.g. changing the number of sweeps, or printing items of interest to the screen, e.g. the content of the agent config file.
    3. **updater.py** – a class containing the model (inherited from BaseModel from abm\_template), it controls the updating of the agents. The updating process consists of the function do\_update, which has two *for loops*, iterating over every agent in the agent list stored in environment:
* First we call agent.create\_temp\_variable(environment) passing in the environment class because that’s where our transition probability graph is stored. The temporary variable containing the new opinion (check agent script for details) is assigned to the dictionary new\_opinion with the agent\_inditifier being the key via self.new\_opinion[agent.identifier]
* Second, we assign self.new\_opinion[agent.identifier] to agent.opinion which replaces the agent’s old opinion with its new opinion.
  + 1. **Measurement.py** – contains class used for writing results of the simulation into the csv file. Its parent class is basemeasurement from abm\_template. The script contains functions which
* open the file
* read a measurement config xml-file
* use a wrapper to return the agents’ opinion to after every step
* write the results (this one is usually run within the runner after every update step),
* close the file.

The measurement config xml-file is stored under ‘configs/’ and contains the filename of the output csv and also its setup, i.e. number of columns, headers, (which are hard-coded) and ‘values’.

The strings assigned to ‘values’ in the measurement config xml-file are turned into the actual values (i.e. opinion of the agent after step i) by the wrapper function in measurement.py. The measurement class is called in the runner script. The directory of the measurement config xml file is stored as <parameter type=’measurement\_config’ value=’configs/xxxx.xml’> in the *environment config* file.

Documentation created by Paweł Fiedor ([Pawel.Fiedor@uct.ac.za)](mailto:Pawel.Fiedor@uct.ac.za)), Tina Koziol and Jesslyn Jonathan on the 19th of June 2016 in Cape Town.

1. This section largely references from: https://en.wikipedia.org/wiki/DeGroot\_learning

   [↑](#footnote-ref-1)