Karlsruher Institut für Technologie

Design Document

Numerical Linear Algebra meets Machine Learning

Fabian Koffer

Simon Hanselmann

Yannick Funk

Dennis Leon Grötzinger

Anna Katharina Ricker

Supervisors

Hartwig Anzt Makus Götz

December 17, 2018

Contents

1 Overview

2 Module Interaction

2.1 Class Descriptions

2.1.1 Class CommandLineInterface

The class CommandLineInterface represents the concrete command line interface. Therefore it only consists of two methods. The first one is read_input that receives a message that will be displayed and reads the next user input. The other method is create_output. This method prints a string to the CLI.

2.1.2 Class Controller

The controller is the main entry point for the program execution. It creates the view, receives the user input, calls the parser to create a command from the input and starts the module the user wants.

2.1.3 Class CommandParser

The CommandParser is a static class that gets the input which the user entered and parses it to a concrete command-object.

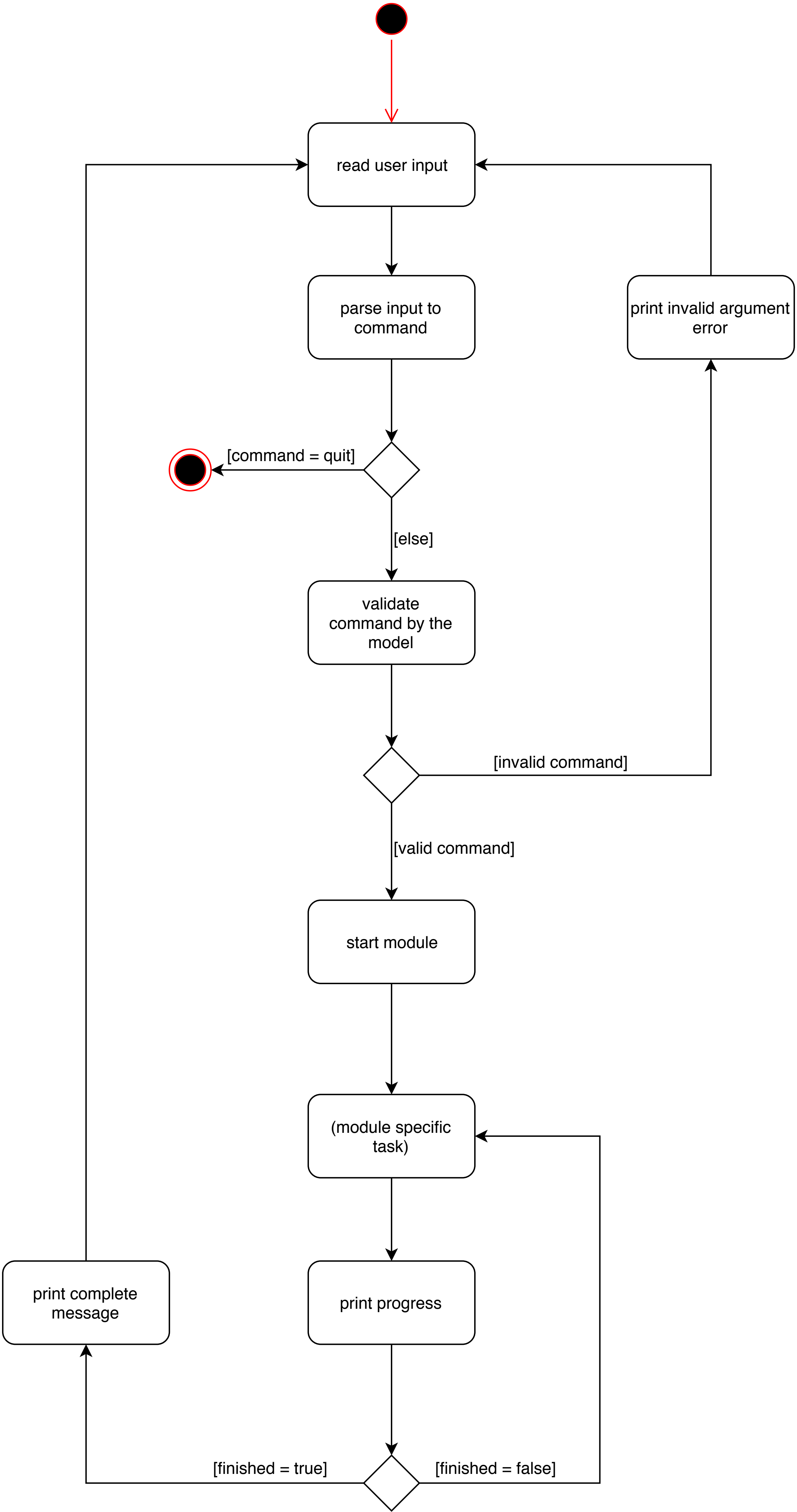
2.1.4 Class Command

The Command class holds all the information entered by the user that is needed to execute a module. There is one command subclass for each module and the command class also validates that all parameters are available to run the module. The command also has a execute method which runs the specific module with all the arguments it needs.

2.2 Activity Diagrams

2.2.1 User Input Activity Diagram

This activity diagram shows the main workflow of the whole program. The program first waits for user input. After the user enters his input into the command line interface, the CommandParser parses this input to a command. If the command equals quit, the program will exit. If not the command will be validated to contain enough information to start the desired module. If it is not valid, an error message will be printed and the user can enter a new input. If the command is valid, the module will be started. Therefore the module computes his specific task in iterations. After each iteration it prints the current progress to the user. This two activities repeat until all tasks are done. When the module is finished, the program prints a complete message and starts waiting for new user input.

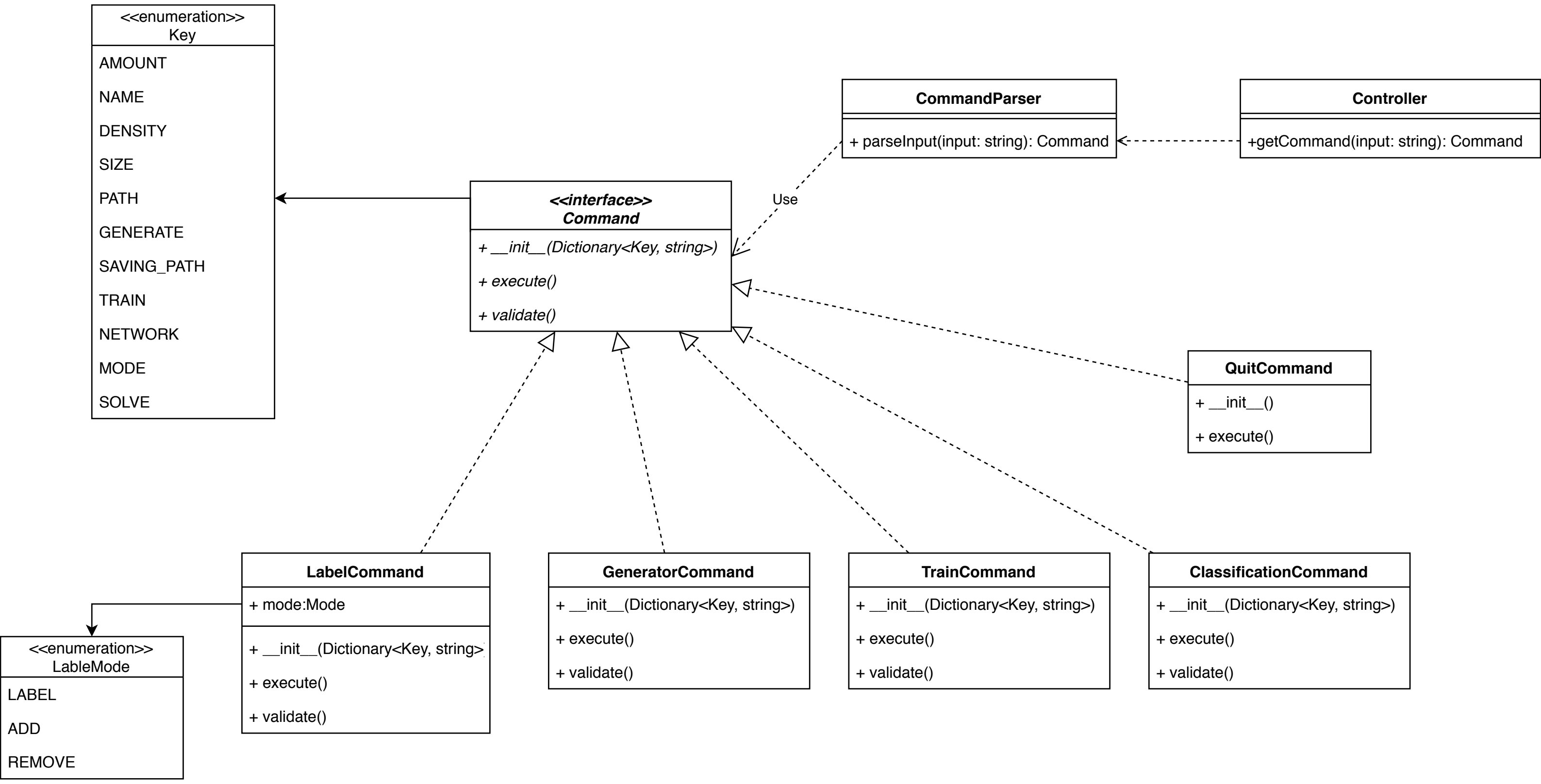


read user input
parse input to command
print invalid argument error
[command = quit]
[else]
validate command by the model
[invalid command]
[valid command]
start module
(module specific task)
print complete message
print progress
[finished = true]
[finished = false]

2.3 Class Diagrams

2.3.1 Command Parsing

This class diagrams shows all classes that are used by the CommandParser. The CommandParser has one method that receives an input string and returns a command. This method uses the first word of the input string to determine which module is supposed to be started. Based on this he creates an instance of this subclass of the command. The remaining string will be parsed to a dictionary with keys and values. This dictionary is passed to the constructor of the command object. In the constructor of the command, the command also calls the validate() method. This method checks if all arguments that are necessary for the specific module start are provided. After that is finished the command can be returned by the CommandParser. The execute method of the command can be then used to start the modules computation.



<<enumeration>>
Key
AMOUNT
NAME
DENSITY
SIZE
PATH
GENERATE
SAVING_PATH
TRAIN
NETWORK
MODE
SOLVE
CommandParser
+ parseInput(input: string): Command
Controller
+getCommand(input: string): Command
Use
<<interface>>
Command
+ __init__(Dictionary<Key, string>)
+ execute()
+ validate()
QuitCommand
+ __init__()
+ execute()
LabelCommand
+ mode:Mode
+ __init__(Dictionary<Key, string>
+ execute()
+ validate()
GeneratorCommand
+ __init__(Dictionary<Key, string>)
+ execute()
+ validate()
TrainCommand
+ __init__(Dictionary<Key, string>)
+ execute()
+ validate()
ClassificationCommand
+ __init__(Dictionary<Key, string>)
+ execute()
+ validate()
<<enumeration>>
LableMode
LABEL
ADD
REMOVE

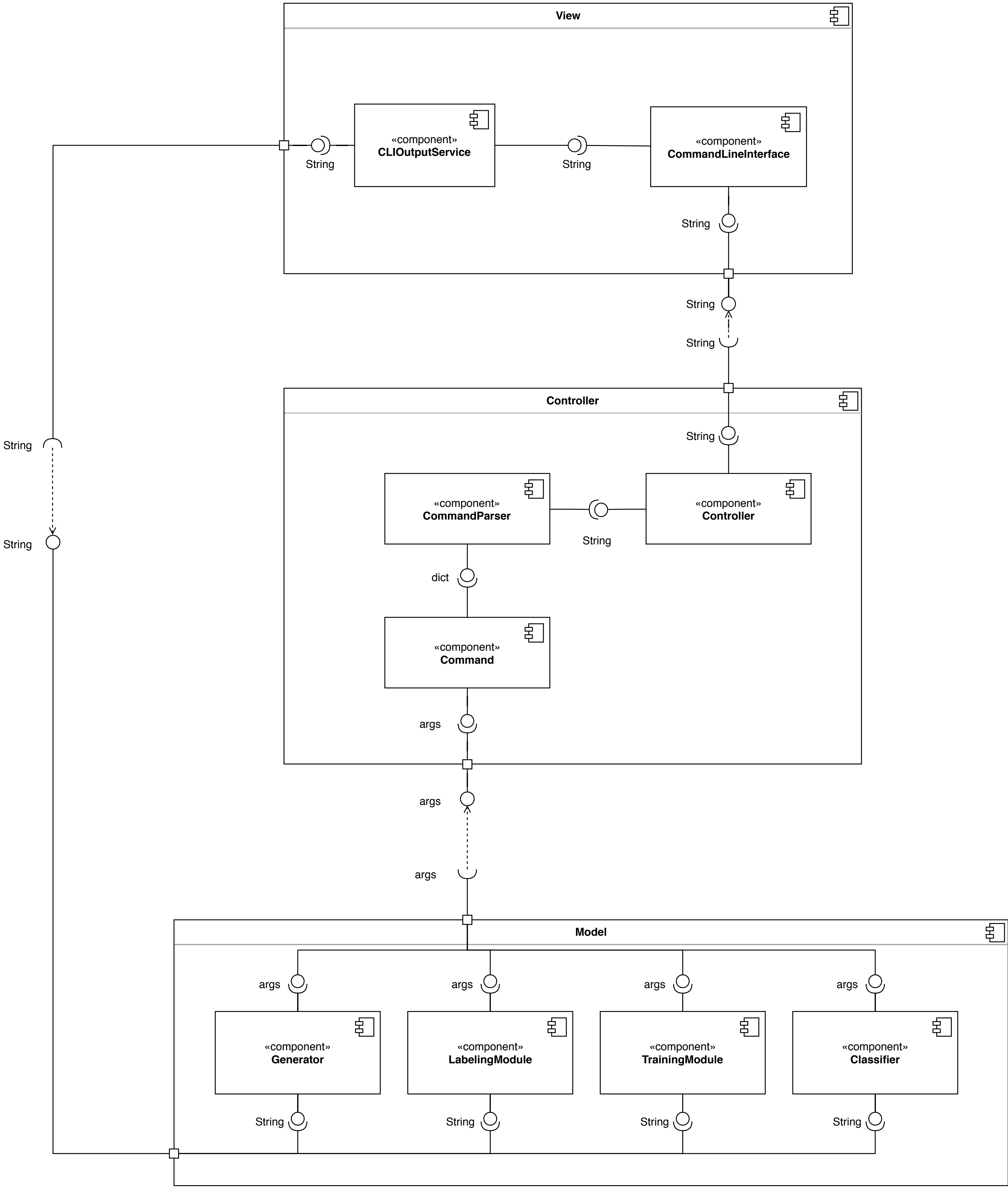
2.4 Component Diagram

2.4.1 Model View Controller Component Diagram

This component diagram shows the main interaction between the modules. For our main structure we use the model view controller Each class belongs to one of the three parts: view, controller or model. In the view there is the CommandLineInterface, which as the name suggests represents the command line. It receives strings, that it will display and makes the user input available for other classes. The other class in the view is the CLIOutputService. This class works as the observer for the model and receives the strings the model wants to output. It then provides strings that can be displayed by the command line interface.

The next big part is the controller. This parts main component is the class that is also called controller. This class gets a string by the CommandLineInterface. This string it then passed to the CommandParser. This class creates the right command and passes the arguments to this command. The command stores this arguments and makes them available for the modules.

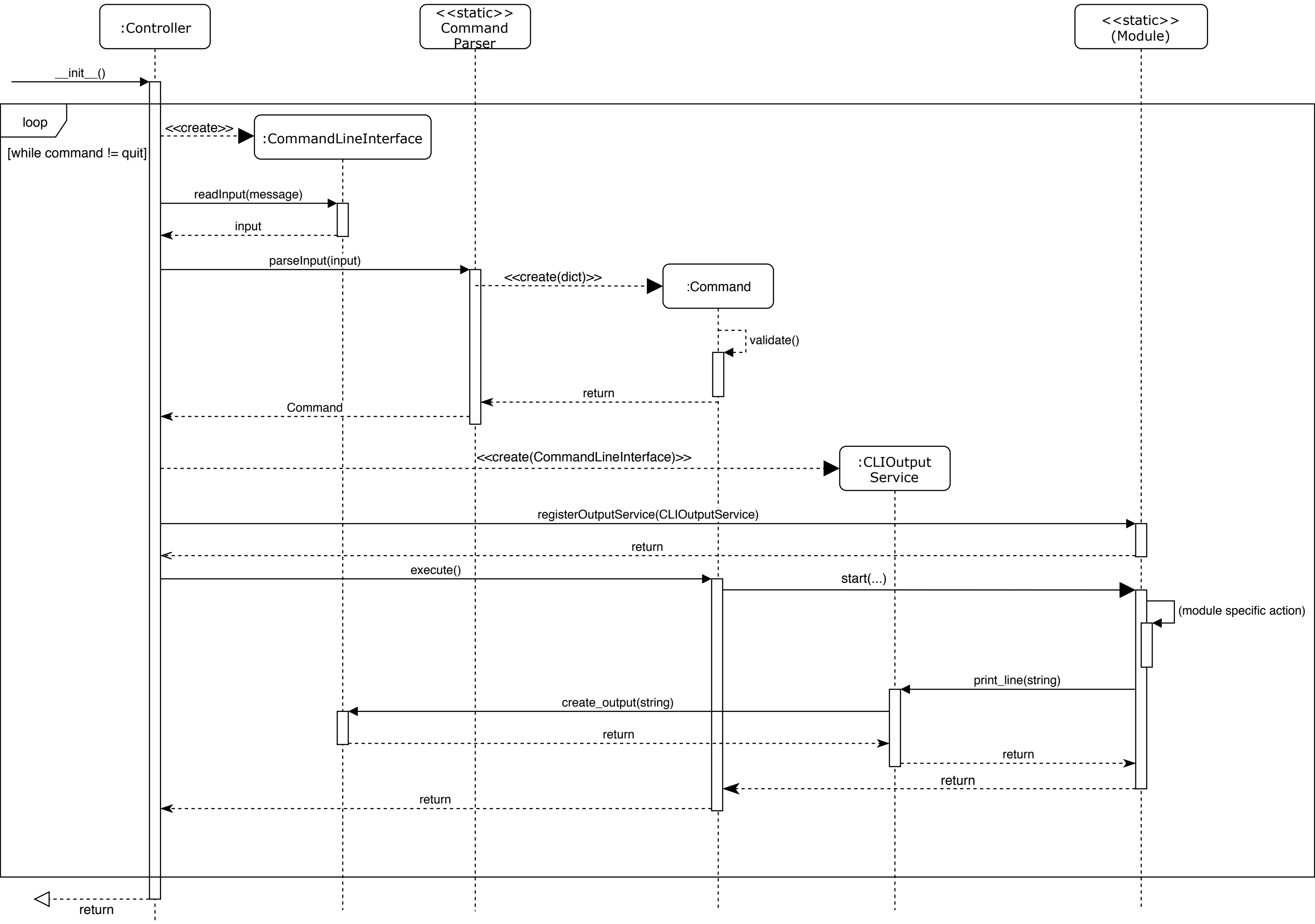
The model consists of the four static modules that make the computations. This modules get the arguments that are provided by the command. The modules can also provide strings that can be displayed in the view.


View
«component»
CLIOutputService
«component»
CommandLineInterface
String
String
String
String
String
Controller
String
«component»
CommandParser
«component»
Controller
String
dict
«component»
Command
args
args
args
Model
args
args
args
args
«component»
Generator
«component»
LabelingModule
«component»
TrainingModule
«component»
Classifier
String
String
String
String
String
String

2.5 Sequence Diagrams

2.5.1 Controller Sequence Diagram

This class diagram shows what happens when the controller is being created. First it will create the CommandLineInterface class. After finishing that, it asks the command line interface for user input. This user input is then passed to the CommandParser that creates a dict out of this string and creates the right instance of a Command. When creating a command, it also calls the validate method that checks if all arguments that are needed by the module for running its computation are provided. If thats the case, the controller creates a CLIOutputService and attaches the CommandLineInterface to it. This CLIOutputService is then registered to the module so this can create output to the command line interface. After all this setup steps, the module is ready to be executed. Therefore the controller calls the execute method on the command, which then calls the start method of the module with the arguments it needs. The module then starts its computations and eventually calls the CLIOutputService to create some user output. The module calls print_line with the string it wants to output. The CLIOutputService creates the string that can be displayed and this is passed to the CommandLineInterface that is attached to CLIOutputService. After all computations are done, the module returns his call. Now the user can enter a new input until he enters quit.



:Controller
<<static>>
Command
Parser
<<static>>
(Module)
__init__()
loop
[while command != quit]
<<create>>
:CommandLineInterface
readInput(message)
input
parseInput(input)
<<create(dict)>>
:Command
validate()
return
Command
<<create(CommandLineInterface)>>
:CLIOutput
Service
registerOutputService(CLIOutputService)
return
execute()
start(...)
(module specific action)
print_line(string)
create_output(string)
return
return
return
return
return

3 Display Output

3.1 Class Descriptions

3.1.1 Interface OutputService

The OutputService interfaces can be implemented and passed to a module to receive the output of the modules. Therefore it has methods that represent different ways output can be displayed.

3.1.2 Interface Subscriber

The Subscriber interface only provides the method update() which will be triggered by an Observable upon receiving new values.

3.1.3 Class Observable

The Observable class can be used to notify subscribers when new values are provided. Subscribers can subscribe themselves to an Observer to get notifications about new data. The next() method calls update() on each subscriber.

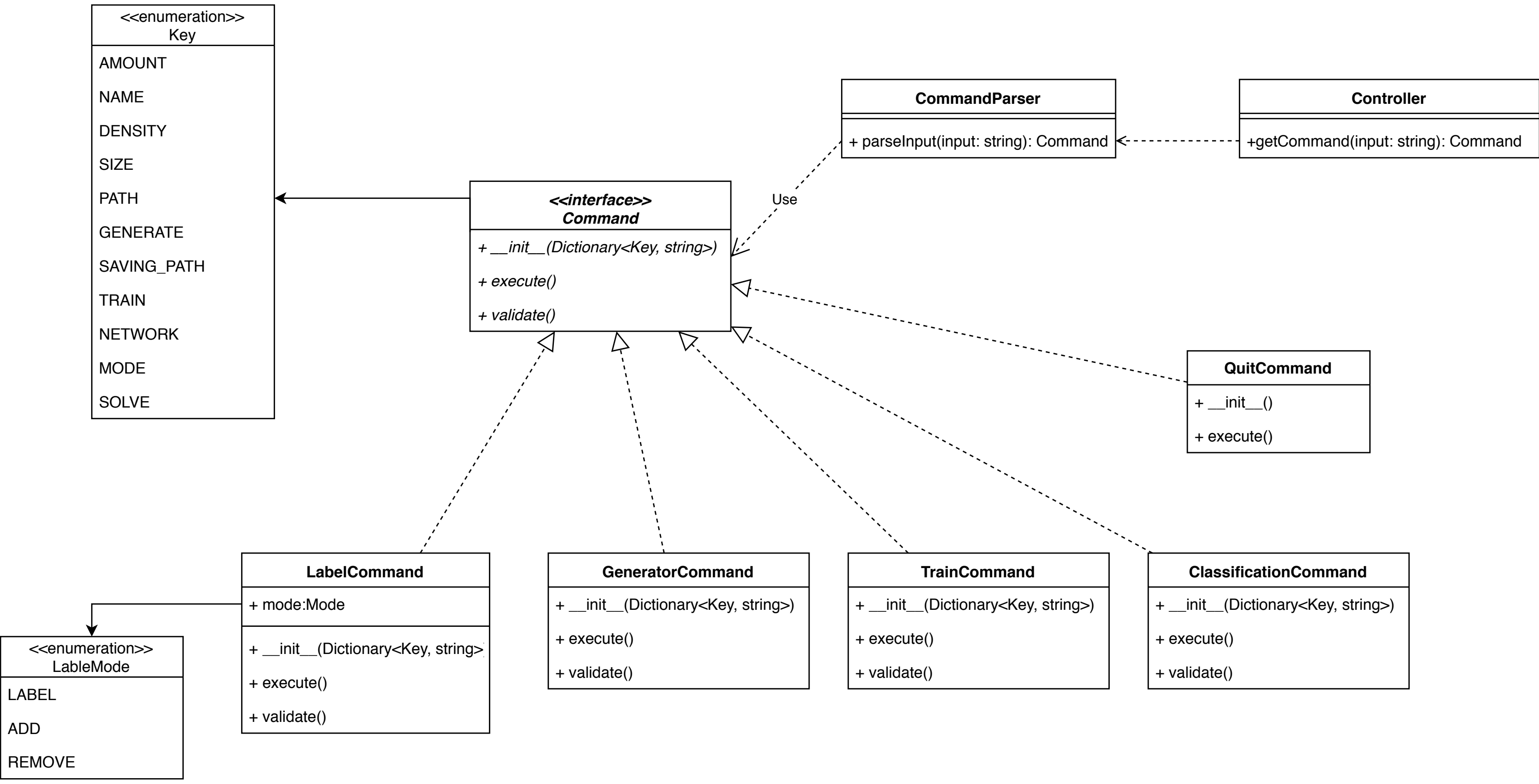
3.2 Class CLIOutputService

This class implements the OutputService and the Subscriber interface. On creation it gets a reference of the CommandLineInterface to which it will pass the lines the modules wants to output. It also implements the Subscriber interface to subscribe itself to an observable. This can be used to display lines that are overwritten with new values like an progress bar or a counter.

3.3 Class Diagrams

3.3.1 Command Parsing

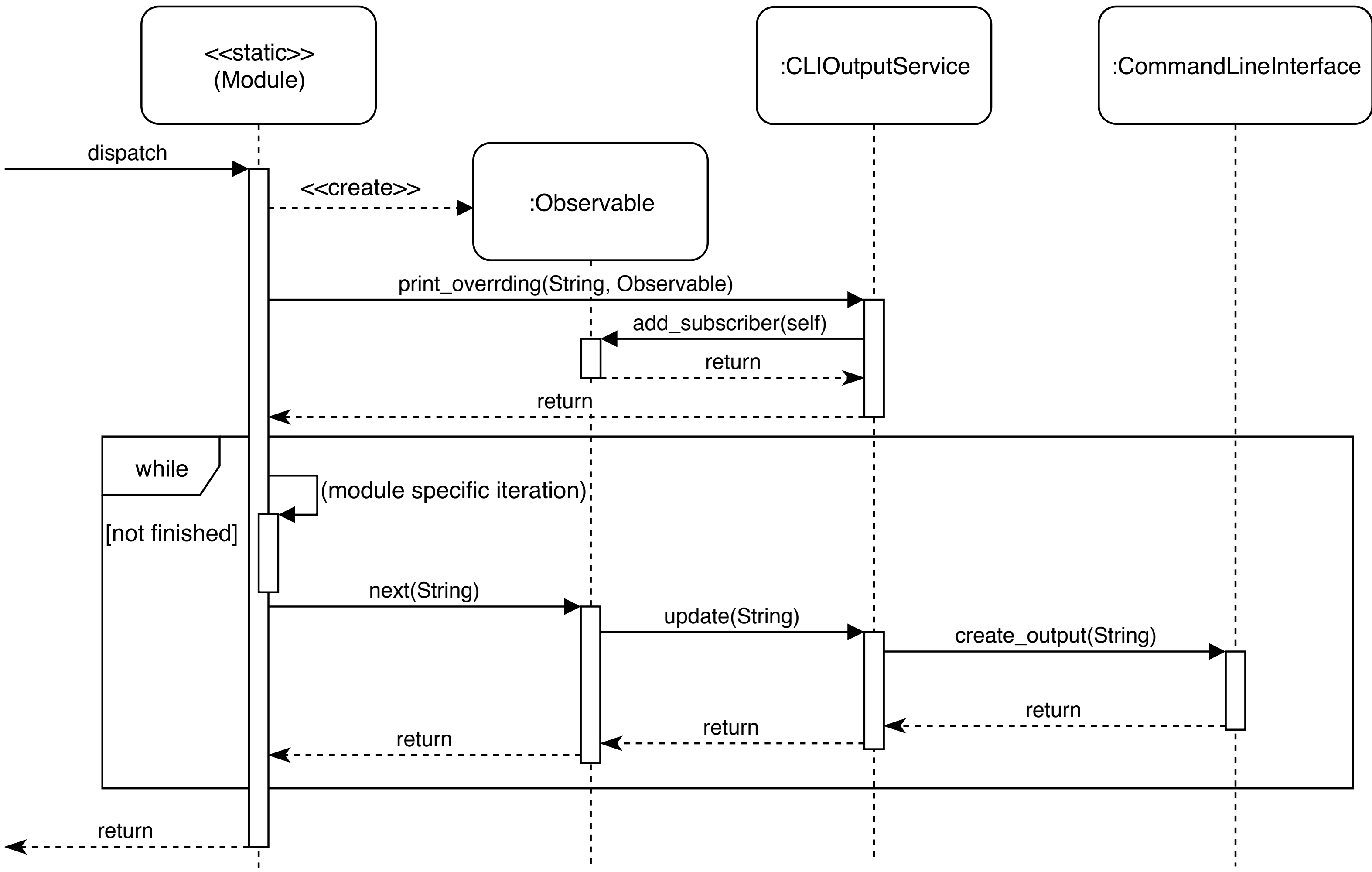
This class diagrams shows all classes that are used by the CommandParser. The CommandParser has one method that receives an input string and returns a command. This method uses the first word of the input string to determine which module is supposed to be started. Based on this he creates an instance of this subclass of the command. The remaining string will be parsed to a dictionary with keys and values. This dictionary is passed to the constructor of the command object. In the constructor of the command, the command also calls the validate() method. This method checks if all arguments that are necessary for the specific module start are provided. After that is finished the command can be returned by the CommandParser. The execute method of the command can be then used to start the modules computation.


<<enumeration>>
Key
AMOUNT
NAME
DENSITY
SIZE
PATH
GENERATE
SAVING_PATH
TRAIN
NETWORK
MODE
SOLVE
CommandParser
+ parseInput(input: string): Command
Controller
+getCommand(input: string): Command
Use
<<interface>>
Command
+ __init__(Dictionary<Key, string>)
+ execute()
+ validate()
QuitCommand
+ __init__()
+ execute()
LabelCommand
+ mode:Mode
+ __init__(Dictionary<Key, string>
+ execute()
+ validate()
GeneratorCommand
+ __init__(Dictionary<Key, string>)
+ execute()
+ validate()
TrainCommand
+ __init__(Dictionary<Key, string>)
+ execute()
+ validate()
ClassificationCommand
+ __init__(Dictionary<Key, string>)
+ execute()
+ validate()
<<enumeration>>
LableMode
LABEL
ADD
REMOVE

3.4 Sequence Diagrams

3.4.1 Print Overriding String

Sometimes the user wants to print a progress like a download status that is updated by time. But there should not be a new line for each percentage the download gets further. Instead the old string should be overwritten by time and and the module should only have to pass the new status. Therefore there is the method print_overriding in the CLIOutputService that wants an string and an observable object. That is why the module first creates the observable object. This object is then being passed to the print_overriding method together with a string that will stay the same for this process. Because the CLIOUtputService implements the Subscriber interface, it can subscribe itself to the observable. Now that the CLIOutputService is subscribed to the Observable, it will be notified when the Observable gets new data. When the module starts it computations, it can call next with the new data with which it wants to override the old one in the view. In the next call of the observable, all subscribers will be notified with the new value. The string with the new value can now be printed to the command line interfaces



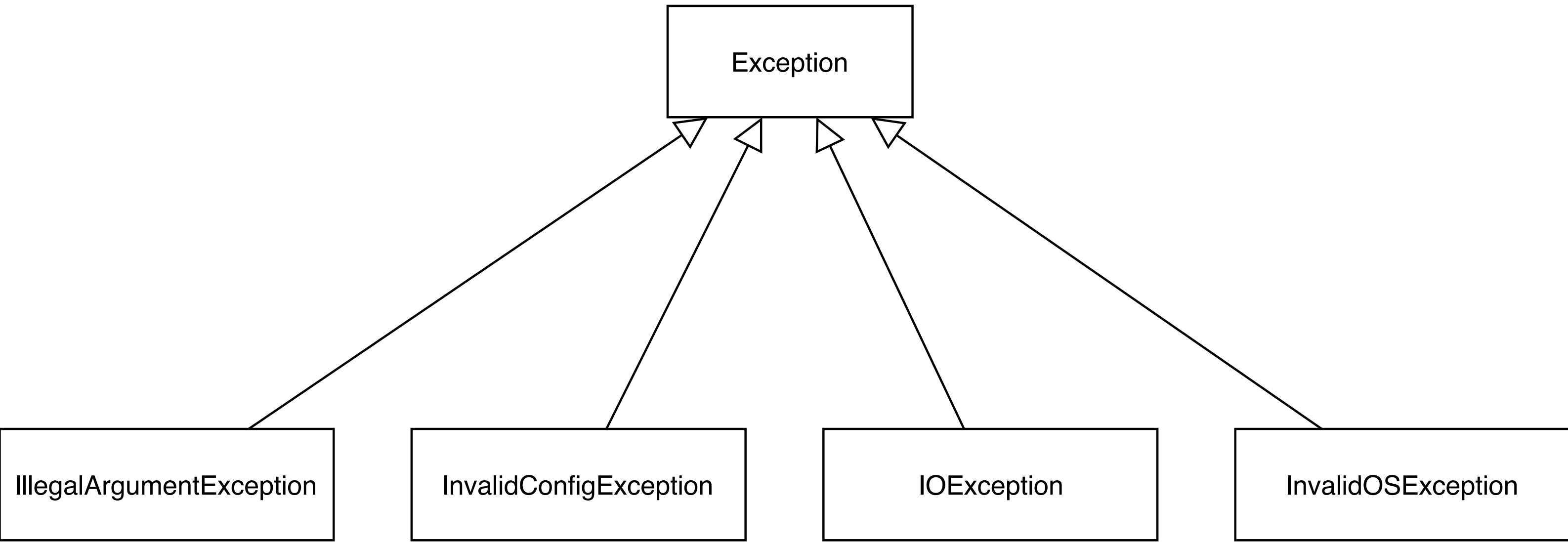
<<static>>
(Module)
:CLIOutputService
:CommandLineInterface
dispatch
<<create>>
:Observable
print_overrding(String, Observable)
add_subscriber(self)
return
return
while
(module specific iteration)
[not finished]
next(String)
update(String)
create_output(String)
return
return
return
return

4 Exception Handling

4.1 Class Diagrams

4.1.1 Exception Classes

Each exception extends Python's Exception class. There are four kinds of exceptions that are excepted. The IllegalArgumentException will be risen when the arguments the user has entered are not valid. The InvalidConfigException will be risen when the user edited the configuration file with invalid arguments or syntax. The IOException will be risen when errors occur while reading or writing data to the hard drive. The InvalidOSException will be risen when trying to access functionality that is not supported on the current operating system.


Exception
IllegalArgumentException
InvalidConfigException
IOException
InvalidOSException

5 Collector

5.1 Class Description

5.1.1 Class Collector

The Collector class is responsible for collecting a given amount of matrices and saving it into a HDF5 dataset. When the user types collect into the CLI, a Collector object will be created and the public method collect() with its parameters:

amount, name, size, density and path

will be called.

The class has a Saver class attribute and a Generator class attribute.

It uses methods from the Generator class to get matrices to collect and methods from the Saver class to save the collected dataset. (see the collect method Activity Diagram for a more detailed overview). The Collector class is the interface between matrix collecting and the CLI and conceals all the classes of the Collector described in the following.

5.1.2 Class Saver

The Saver class is just responsible for saving a given matrix dataset. Its only method is the save(dataset, name, path) method, which is called by the collect method from an Collector object.

The Saver class is just responsible for saving a given matrix dataset. Its only method is the save(dataset, name, path) method, which is called by the collect method from an Collector object.

The save method takes an NPArray as a matrix dataset, converts it into an HDF5 file and saves it into a given directory with a given name.

5.1.3 Class Generator

The Generator class is responsible for actually generating matrices by transforming raw matrices from SuiteSparse and validating them. The generate(size, density):Matrix method is called by a Collector object, uses the Matrix class to initialize an empty matrix, uses the Ssget class to fetch and transform matrices from the SuiteSparse collection and uses the static Validator.validate method to check if the matrix is regular and can be returned.

5.1.4 Class Ssget

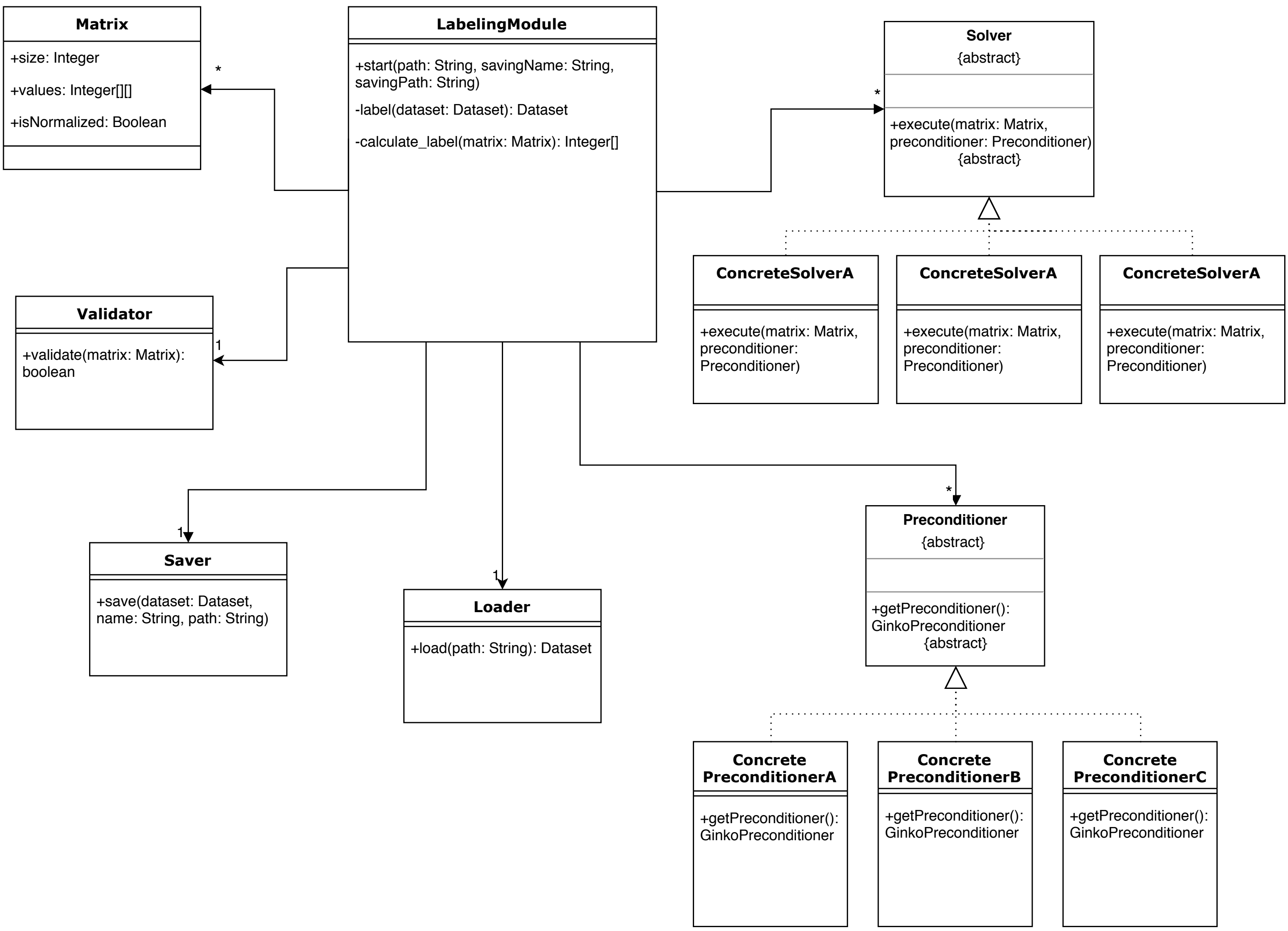
The Ssget class is responsible for fetching matrices from the SuiteSparse collection, transforming them and returning them. Its getMatrix method is called by a generator object. The getMatrix method uses the Matrix class to initialize a matrix, then the private downloadMatrix method to fetch a matrix from SuiteSparse, and after that uses its private cutMatrix method to cut a fixed size, regular matrix out of it.

5.1.5 Class Validator

The Validator class is a util class and responsible for validating given matrices(checking for regularity) Its only static method validate takes a matrix and returns true for regular, and false for not regular.

6 Labeling Module

The labeling module is responsible for the labeling of the sparse matrices. The label of a matrix describes which preconditioner/iterative solver combination solves the given matrix the fastest. The label is represented by a vector. Each entry in the vector corresponds to one preconditioner/iterative solver combination. The entry of the fastest one is 1, all other entries are zero. The matrices with the corresponding labels will be used to train the neural network in the training module.


Matrix
+size: Integer
+values: Integer[][]
+isNormalized: Boolean
*
LabelingModule
+start(path: String, savingName: String, savingPath: String)
-label(dataset: Dataset): Dataset
-calculate_label(matrix: Matrix): Integer[]
*
Solver
{abstract}
+execute(matrix: Matrix, preconditioner: Preconditioner) {abstract}
ConcreteSolverA
+execute(matrix: Matrix, preconditioner: Preconditioner)
ConcreteSolverA
+execute(matrix: Matrix, preconditioner: Preconditioner)
ConcreteSolverA
+execute(matrix: Matrix, preconditioner: Preconditioner)
Validator
+validate(matrix: Matrix): boolean
1
1
Saver
+save(dataset: Dataset, name: String, path: String)
1
Loader
+load(path: String): Dataset
*
Preconditioner
{abstract}
+getPreconditioner(): GinkoPreconditioner {abstract}
Concrete PreconditionerA
+getPreconditioner(): GinkoPreconditioner
Concrete PreconditionerB
+getPreconditioner(): GinkoPreconditioner
Concrete PreconditionerC
+getPreconditioner(): GinkoPreconditioner

The main component of the labeling module is the class labeling module. It provides the only public method in the labling module, the method start(path:String, savingName:String,savingPath:String). This method is the entry point of the module and will start the labeling process of the provided matrices(specified by the path).

The class labeling modue has a set of matrices which the module will label.

The module furthermore has a Loader class. The class labeling module has exactly one Loader class. This class is responsible for loading the matrices which get labeled. Its only method is the method load(path:String) which gets a path of a hdf5 file supplied and returns a dataset. If the specified path is not a hdf5 file, the programm will print an error to the command line.

Another class in the labeling module is the Saver class. The class labeling module has exactly one Saver class. This class is responsible for the saving of the matrices and the labels. Its only method is the method save(dataset:Dataset,path:String). If this method is called, the specified dataset will be safed at the specified path. The matrices and the labels will be safed in one hdf5 file.

The labeling module class moreover has a validator class. This class is responsible for determining wheter the given matrix is regular. If that is not the case, the matrix will be deleted.

Since the labeling module is responsible for finding the best preconditioner/iterative solver combination for a given set of matrices, the module furthermore has a preconditioner and a Solver class. Those classes are abstract. ConcreteSolvers inherit from the class Solver and Concretepreconditioners from the class preconditioner.

The Solver class contains the logic for solving a matrix with an iterative solver. Each ConcreteSolver corresponds to one iterative solver.Each class of ConcreteSolver has the method execute(Matrix,preconditioner) which will solve a given matrix. We will be using the design pattern "stragety" for the iterative solvers. The reason being that each Solver does basically the same thing(solving a matrix) in a different manner. The user moreover has no influence on which Solver we will be using at any given time. Each Solver will take an optional precondtioner as its input for the method execute(Matrix,preconditioner). The preconditioner will be used at every step of the iterative solver. We will be using the design pattern "strategy" for the preconditioners too. preconditioners each return a Ginkopreconditioner which will be used by the Solver class to communicate with the ginkgo libary. So each Concretepreconditioner basically does the same thing(returning a preconditioner). The user furthermore has no influence on which preconditioner we will be using at any given time. That is why the "strategy" design pattern is applicable

6.1 Class Descriptions

6.1.1 Class LabelingModule

The class LabelingModule is the main component of the labeling module. It provides one public method start(path:String, savingName:String,savingPath:String) which is the entry point for the labeling module. When the user types label in the command line this command will be executed. The arguments of the start method are optional. If there are no arguments provided the labeling module will be using default paths. The default paths are specified in a configuration file(see the training module for further details). If the User specifes his own paths they have to be vaild. Valid in this case means that the path points to a hdf5 file with the correct formatation. If this is not the case the program will print an error message to the command line and execute a controlled shutdown so that the user can specify a correct path.

6.1.2 Class Solver

An Solver in our sense is an interative Solver which is able to to solve a linear system Ax=b for x, where A is a (in our case sparse) matrix of size nxn, x is a vector of size 1xn and b is vector of size 1xn($n \in N$). The matrix A in our case will be the Matrix the method execute(matrix,preconditioner) will receive, the vector b will be a random vector with values between 0 and 1. We chose the vector to be random since the choice of b has no great influence on the time it takes to solve the linear system. The iterative solver uses an iterative approach to solve the linear sysetm. An iterative approach is characterized by the idea that the linear system gets solved step by step, where the solution of one step enables the solution of the next step. The class iterative solver achieves this by communicating with the ginkgo libary, which has an implementation of the solvers. An iterative solver may optionally use a preconditioner for its calculation. Since there are many different iterative solvers which achieve the same outcome(solving for x) we will be using the design pattern "strategy". That is why the class Algortihm is abstract and ConcreteSolvers (which actually represent one iterative solver each) will inherit from Solver. The user at no point decides which Solver gets used at any given time.
A Solver has one method execute(Matrix,preconditioner) ,which takes a matrix (and a preconditioner) and solves it. The time the iterative solver takes to solve the matrix will

be recorded and in the class labelingModule used to label the matrix. All ConcreteSolvers have to implement the abstract function execute.

6.1.3 Class ConcreteSolver

A ConcreteSolver is the actual representation of one iterative solver.

6.1.4 Class preconditioner

A preconditioner is a transformation of a linear system Ax=b for x, where A is a (in our case sparse) matrix of size nxn, x is a vector of size 1xn and b is vector of size 1xn($n \in \mathbb{N}$). A transformation may be a Matrix p (nxn) which would result in the linear system PAx = Pb. A preconditioner is used so that the linear system may be solved more easily by an iterative solver. The transformation of the preconditioner is applied in every step of an iterative solver.

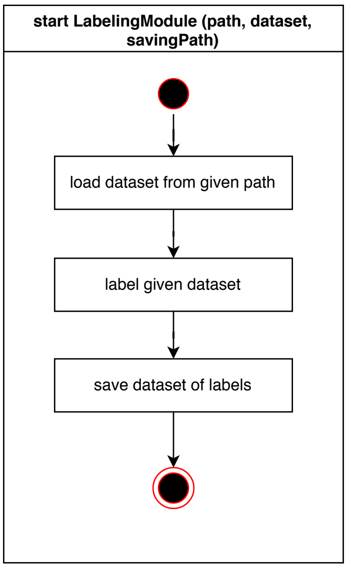
The class preconditioner only has one method, getpreconditioner() which will return the Ginkopreconditioner corresponding to the preconditioner class. The preconditioner class achieves this by communicating with the ginkgo libary.

6.1.5 Class Concretepreconditioner

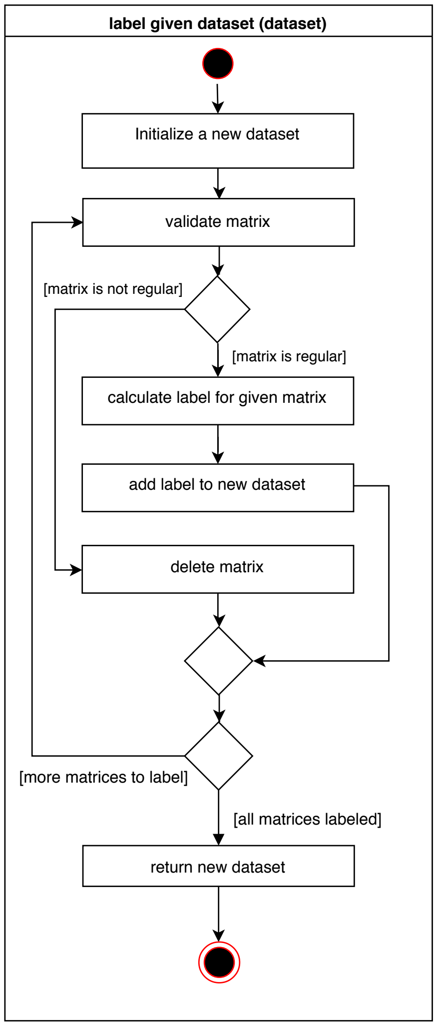
A Concretepreconditioner is the actual representation of one preconditioner.

6.2 Activity Diagrams

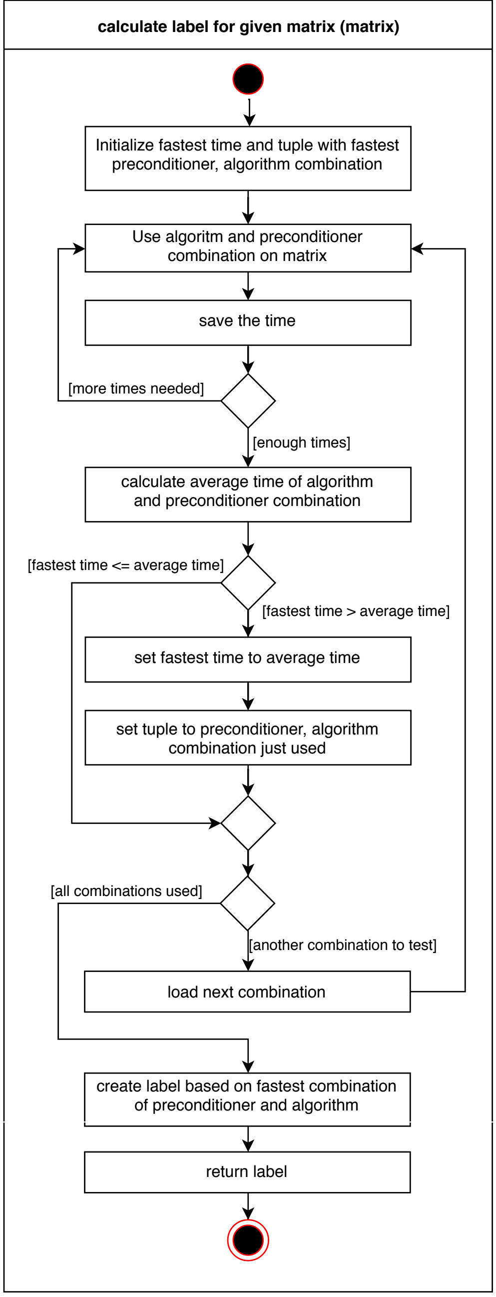
When the user types label() in the CLI the moethod start() in the class LabelingModule will be executed in the following manner:


start LabelingModule (path, dataset, savingPath)
load dataset from given path
label given dataset
save dataset of labels

The activity diagram "start LabelingModule" shows the general overview of what the LabelingModule does.The shown method start() is the only public method in the LabelingModule class, therefore the only way to communicate whit it. First of all the dataset to work on will be loaded from the given path, which contains all matrices the LabelingModule has to label. After that the dataset will be labeled (explained in detail in the "label given dataset" activity diagram). Last but not least the new dataset with labels will be saved under the given savingPath.


label given dataset (dataset)
Initialize a new dataset
validate matrix
[matrix is not regular]
[matrix is regular]
calculate label for given matrix
add label to new dataset
delete matrix
[more matrices to label]
[all matrices labeled]
return new dataset

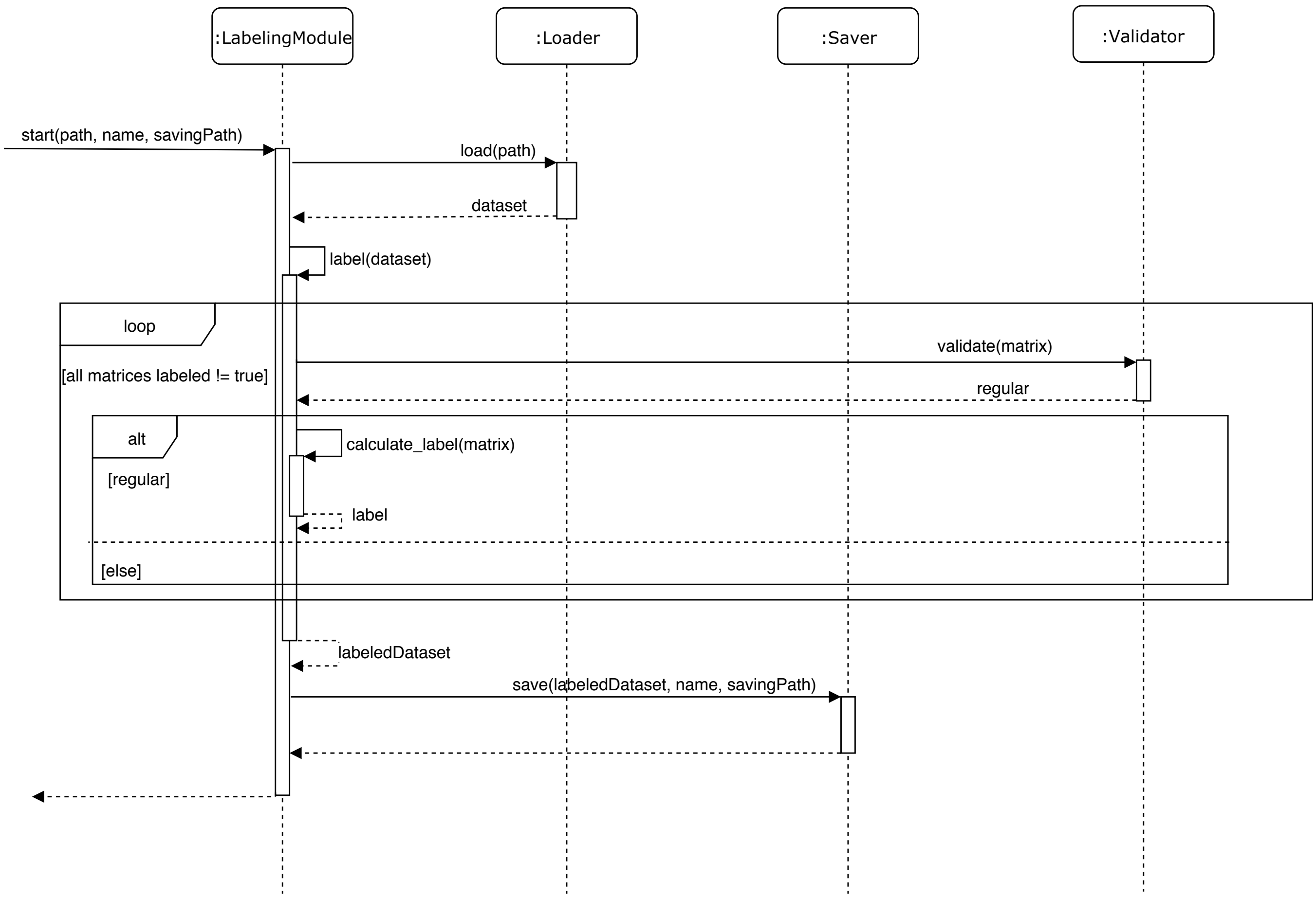
The activity diagram “label given dataset” shows how the class LabelingModule labels a dataset.First of all a new dataset is initialized. The following steps will be made on each matrix in the dataset. First of all the matrix is validated, to see whether the matrix is regular or not, if the matrix isn't regular the matrix will be removed. In case that the matrix is regular, the label for the matrix is calculated (explained in the “calculate label for given matrix activity diagram”). The calculated label will be saved in the new dataset. As soon as every matrix has been labeled the new dataset is returned.


calculate label for given matrix (matrix)
Initialize fastest time and tuple with fastest preconditioner, algorithm combination
Use algoritm and preconditioner combination on matrix
save the time
[more times needed]
[enough times]
calculate average time of algorithm and preconditioner combination
[fastest time <= average time]
[fastest time > average time]
set fastest time to average time
set tuple to preconditioner, algorithm combination just used
[all combinations used]
[another combination to test]
load next combination
create label based on fastest combination of preconditioner and algorithm
return label

The activity diagram “calculate label for given matrix” shows how a label is calculated for a single matrix. First of all a fastest item variable and a tupel of preconditioner,iterative solver is initialized. The first preconditioner,iterative solver combination is used on the matrix and the time it took to solve given matrix is saved, that process is looped, until we have enough saved time data. Out of the saved times for the preconditioner,iterative solver combination the average time the combination took will be calculated. The calculated average time will be written in the fastest time variable and the combination in the tupel previously initialized. After that a new combination is loaded, if there is another one to test. For each new combination the process of using the combination, saving the time and calculating the average time remains the same. After the average time of the new combination is calculated. After that there are two options. 1. if the average time of the new combination is smaller than the time saved in the fastest time variable, the fastest time variable gets overwritten with the new average time and the new combination is also getting written in the tupel.2. the fastest time is smaller then the new average time, is that the case, the new average time and combination is ignored. Last but not least, if all combinations are finished testing a label is created based on the current fastest time and preconditioner, iterative solver combination and the label is returned.

6.3 Sequence Diagrams

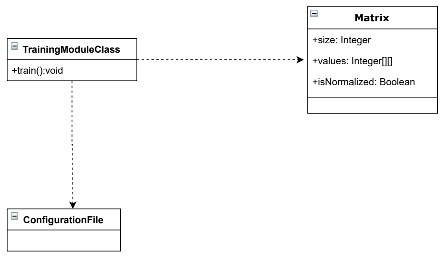
The sequence diagram shows what happens when the start() method of the LabelingModule is called. First of all the method load(), with the parameter path, of the Loader class is called,the path is either the path the user wrote in his input, or it is the standard path, which is specified in the Configuartion Fil. The load() method returns the dataset the LabelingModule will be working on. After that, the LabelingModule calls its own method label() with the just received dataset. The label() method then goes into a loop, covering each matrix on its own. For each matrix, the validate() method of the Validator gets called with the parameter matrix. validate() returns a boolean, whether the matrix is regular (true) or not (false). If the matrix is regular, the method calculate_label() of the LabelingModule with the parameter matrix is called. calculate_label returns a label. if the matrix isn’t regular, or the label is calculated the loop starts with the next matrix, until there are none left.After all labels are calculated they all are summarized in a dataset. The new dataset is after that one of the parameters for the save() method of the Saver class, which saves the new dataset, with the given namen in the given path.


:LabelingModule
:Loader
:Saver
:Validator
start(path, name, savingPath)
load(path)
dataset
label(dataset)
loop
[all matrices labeled != true]
validate(matrix)
regular
alt
calculate_label(matrix)
[regular]
label
[else]
labeledDataset
save(labeledDataset, name, savingPath)

7 Training module

The training module is responsible for the training and testing of a neural network. It is structured in 2 parts, the configuration file and the class training module. The class training module loads its configuration in the configuartion file. It furthermore uses a set of labled matrices for the training and testing. With the configuration set, the class training module will start the training and testing. The trained network will be saved to a specified

Training Module



7.1 Class descriptions

7.1.1 Class Configuration File

The configuration file is a text file. It is used to specify all necessary information the class neural network needs to train the neural network. If the user does not change anything in the configuration file, defaut options will be used. The configuration file is organized in four main categories.

1. loading path of the set of matrices
2. saving path for the neural network
3. loading path for the neural network
4. model definition and hyperparameters

The loading path of the set of matrices is the path in which the matrices that are used for the training and testing are stored. The training module only supports one hdf5 file. If the path is any other file, the labling module will print an error and execute a controlled shutdown so that the user can specify a valid path. . For the training and testing making sense there should be at least 500 matrices in the hdf5 file. Otherwise the accuracy of the neural network will be so low that i can not be used for classification. If there is no path specified, the training module will use a default path. In the default path will be the latest matrices that the labling module has produced.

The saving path for the neural network is the path where the trained and tested neural network will be safed. It will be safed as a Keras model. If there is no path specified, the neural network will be safed at a default destination. If there is no path for the neural network specified in the module Classifier the module will use this default path to load its neural network.

The loading path for the neural network is strictly optional. If this path is specified the training module will use the neural network in the path for training and testing. This option enables the user to use a pre-trained neural network for training. This could be the case if the user interrupts the training process at a certain time and wants to to repeat the training later. Other use cases are of course possible too. The neural network has to be a model of the Keras framework. If the path is any other file the training module will print an error and execute a controlled shutdown so that the user can specify a valid path. If this path is not specified the training module will create a new neural network(with the model definition and hyperparamters of the next category) and train with it.

The model definition and hyperparameters are used to determine which neural network will be trained and tested. The model definition determines the following:

- the amount of layers
- the amount of nodes in every layer
- the kind of neural network(e.g. Convolutional)
- the activation function
- the regularization
- the optimizer

The hyperparamters determine the following:

- the dropout
- the batch size
- how much of the data should be training and how much should be testing data
- the network weight initialization
- the learning rate

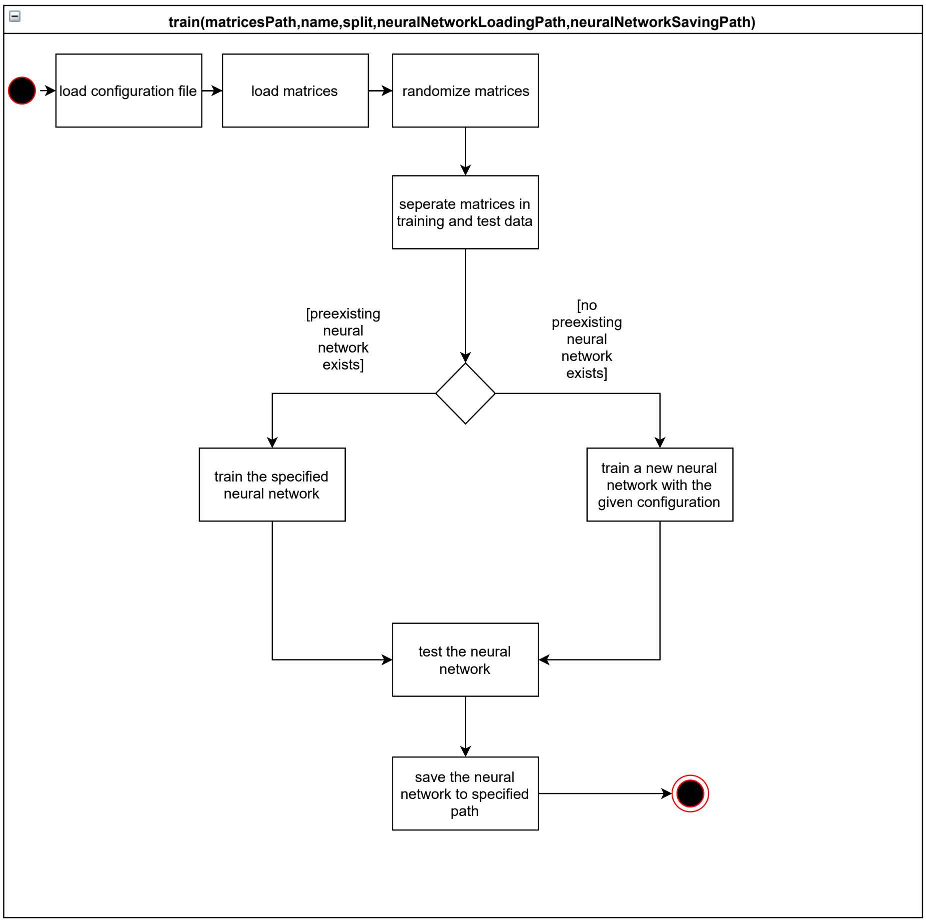
7.1.2 Class TrainingModule

The TrainingModule class is responsible for the training and testing of a neural network. It can not be instantiated, since it is a utility class.The structure is mainly oriented towards the keras workflow and will be further described later in the activity diagramm. The class offers one public method, the method train().

We will furthermore be using the function keras.callbacks.ModelCheckpoint to save the neural network after every epoch. This will guarantee that we do not loose all training progress if the computer crashes or other unexpected events happen. The proceedure is consitent with the design pattern memento.

7.2 Activity Diagarams

When the user types train() in the CLI the method train in the class TrainingModule will be executed in the following manner:


train(matricesPath,name,split,neuralNetworkLoadingPath,neuralNetworkSavingPath)
load configuration file
load matrices
randomize matrices
seperate matrices in training and test data
[preexisting neural network exists]
[no preexisting neural network exists]
train the specified neural network
train a new neural network with the given configuration
test the neural network
save the neural network to specified path

- load the configuration file
- load the matrices
- seperate matrices in training and test data
- train a preexisting neural network or a new one(depending on the configuration file)
- test the neural network
- save the neural network

The configuration file that gets loaded will be used to specify the subsequent points.

The configuration file will determine from which path the labeled matrices will be loaded. If there were no changes made in the configuration file, the default path will be used(see the class description of the configuration file). The labeled matrices will be loaded in one hd5 file. If the path links to any other file, the class TrainingModule will print an error to the command line and execute a controlled shutdown so that the user can specify a valid path..

After that the class TrainingModule will seperate the training and test data. How the data will be seperated is specified in the configuration file.

Following there are two alternatives If the user has specified a neural network in the configuration file, the class TrainingModule will train this neural network with the labeled matrices for the training. If the user has not specified a neural network in the configuration file, the class TrainingModule will create a new neural network with the specifications in the configuration file. If there are no model definitions in the configuration file the class TrainingModule will use the default neural network. The class TrainingModule then proceeds with training the new neural network with the labeled matrices for the training. In both cases the current loss will be continously printed to the command line.

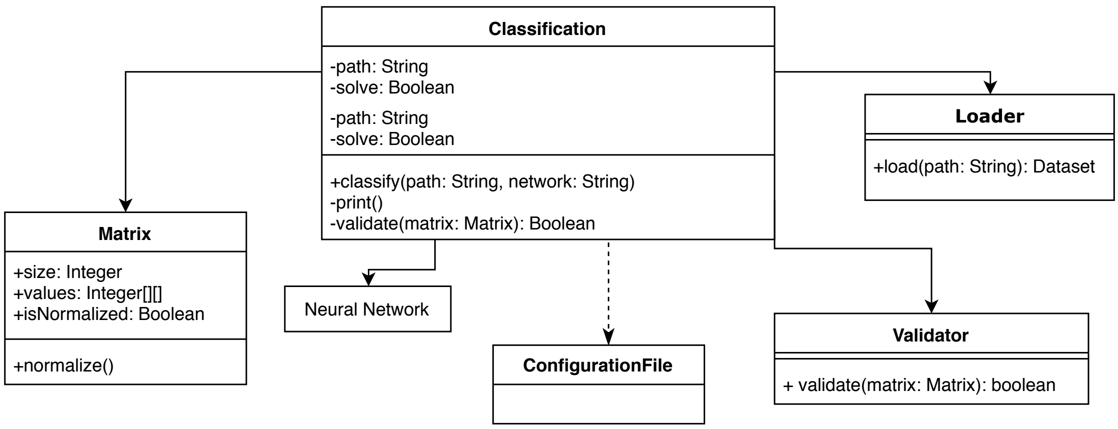
Now the neural network is trained. The class TrainingModule proceeds with testing the neural network with the labeled matrices for the testing. This process will determine the accuracy of the neural network on the given test matrices. The accuracy will be printed on the command line.

After that the neural network will be safed as a keras model. The path for the saving is specified in the configuration file.

8 Classifier

8.1 Diagrams

8.2 Class Diagrams



8.3 Class Descriptions

8.3.1 Class Classifier

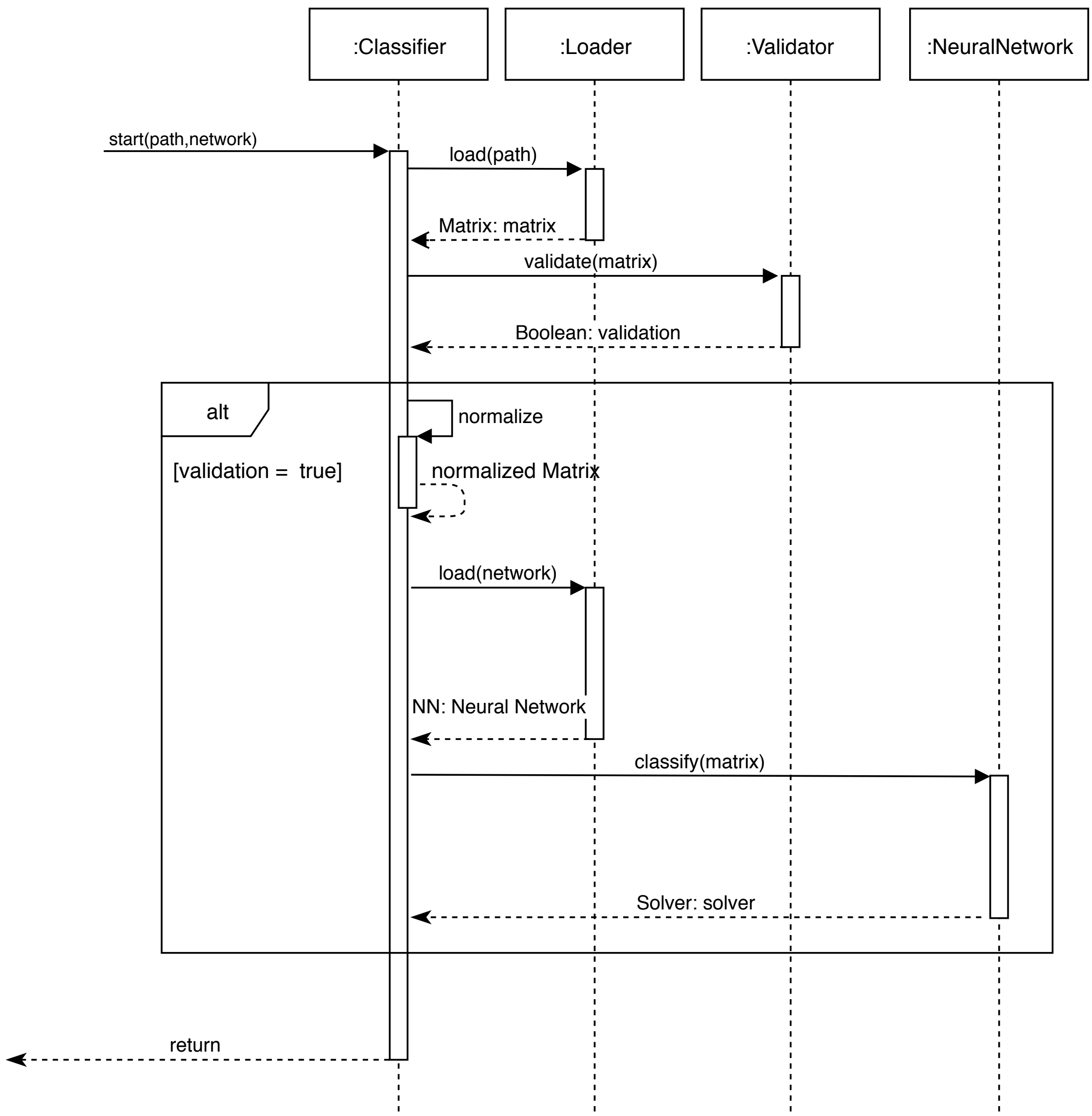
The classifier classifies a matrix given by a path. In this path, the matrix is stored in the form of an HDF5 file. As a second parameter the neural network path will be given. In this path the trained network the user wants to use for his classification is saved.

8.4 Activity Diagrams

8.4.1 Description

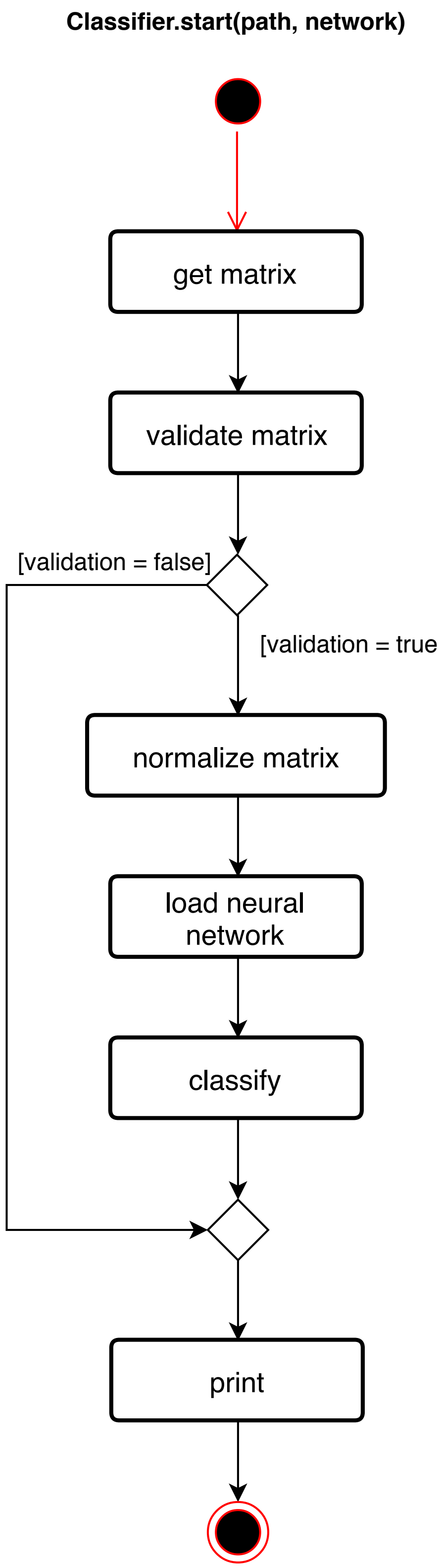
The method start in the class classifier classifies a given Matrix. First of all the classifier has to get the matrix. After that the matrix has to be validated. This validation will give a boolean back, whether the matrix is regular (true) or not (false). If the validation is true (validation = true) the matrix will be normalized. After that the neural network will be loaded, with which in the following the matrix will be classified. In the last step a result will be printed whether it is the result of the classification or an Error-statement, because the validation was false.

8.5 Sequence Diagrams



8.5.1 Description

The method start has the input parameters "path" and "network". The first parameter "path" is the path to the HDF5 file in which the matrix is saved that shall be classified. The second parameter "network" is the path to the neural network which will classify the matrix. First of all the classifier will be calling the Loader to load the Matrix. The Loader will give back the matrix, that will be in the next step validated by the validator. The validator will give back a boolean value whether the matrix is regular (true) or not regular (false). If the validation value is true the classifier will normalize the matrix. After that the matrix is normalized and ready for classification by the neural network. So in the next step, the loader will load the neural network located on the given network path and gives back the neural network to the classifier. In the next step, the classifier can give the matrix to the neural network for classification and the neural network gives back the solver with which the matrix is classified.


Classifier.start(path, network)
get matrix
validate matrix
[validation = false]
[validation = true
normalize matrix
load neural network
classify
print

9 Classes which more than one Module uses

9.1 Class Loader

9.1.1 Class Validator

The Validator class is a util class and responsible for validating given matrices(checking for regularity) Its only static method validate takes a matrix and returns true for regular, and false for not regular.

9.1.2 Class Neural Network

10 Glossary

Glossary

default neural network The default neural network is the network we will be using if the user does not specify a different architecture. It will be a Convolutional neural network.

iterative solver An iterative solver is a method for solving a linear system. An iterative solver uses an iterative approach to solve a matrix. An iterative approach is characterized by the idea that the matrix gets solved step by step, where the solution of one step enables the solution of the next step. An iterative solver may use a preconditioner to improve its results..

label A label is a word or a phrase that is used to describe the characteristics or qualities of something.

memento Memento is a design pattern which provides the ability to restore an object to its previous state. In our case the object is the neural network. After a certain amount of training steps we will save the state of the neural network. If the programm crashes at a certain point in time we will be able to reset the state of a neural network to a safed state. That way we will not loose much of our training achievements..

preconditioner A preconditioner is a Transformation of a linear System. With certain preconditioners, certain iterative solvers may solve a linear system faster that with no preconditioner. The transformation of the preconditioner gets applied in each step of the iterative solver..

strategy strategy is a design pattern of the category behavioral design patterns. It defines a family of iterchangable algorithms. It furthermore enables the selection of an algorithm at runtime, so that the algorithm may vary independently of the clients that use them..

tupel A tupel is a structure of data, that holds more than one data, in contrast to other data structures that usually hold only one data..