Frameworks Second Delivery

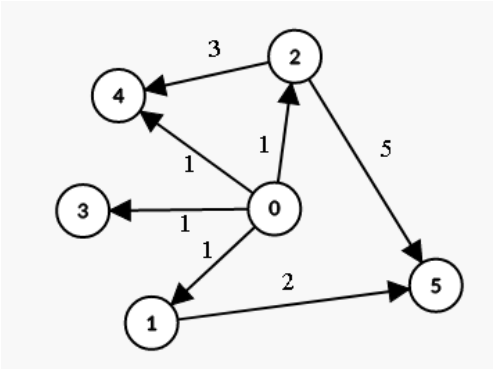
Husein Davlajev

Project description

The project is a framework for generating synthetic clickstream data. This data can be used to train machine learning models to analyze how users interact with a website or an application. It can be hard to find suitable data for machine learning. This framework will let a client generate their own synthetic clickstream data, that can be used to train machine learning models.

The main purpose of the framework is to generate synthetic clickstream data that can be exported. The framework will have methods that returns the data in a form of a java-, json- or a xml object. The framework will also allow the client to create a file and write directly to it or to send the generated data to an external source.

The framework will also allow the client to create generate mock websites that are represented as weighted graphs. This website object will have pages, and these will be connected to other pages through actions (generally clicks). With this model clients can generate clickstream data without having to log user activity on an actual website. Here the edge between the vertexes represents that there is a link between to pages. The weight, represented with a number 1 or higher would represent actions that need to be done in order to access this link. An example would be that a user has to login to access the profile page from the homepage, this edge will then have a weight of 2, since only two action needs to be performed, typing username and password and then clicking login. The weight may also represent the likelihood of that link to be clicked, this can be pages that the client feels certain that users will click on more than other.



With mocks of their website clients can get an idea of how users will be navigating through the website, and how many actions are required for a task.

The data generated by the framework will consist of actions performed by the users. This will include information about the user, for example a user id, ip-address or a username. Date and time for the action performed or session id and time, the name/ url of the page the action was performed. The client will also have the ability to specify what kind of actions the data should contain to best suit their needs.

Clickstream data gives us an insight in how users interact with a website. How much time they spend on certain pages, how they navigate through the website, what actions are performed and when they choose to close it. All this gives the developers and idea of what works and what doesn't. Small changes can have huge effects and with clickstream data developers can analyze this effect.

Group description

The group consists of one sole developer, Husein Davlajev. I am a third year bachelor student in Computer science at Høgskolen i Østfold, specializing in programming. This project is highly relevant for my degree as I have been using different frameworks regularly during my studies, but never created one.

API Original Design Specification

Classes

A first draft of classes and their field variables that need to be included.

Public class Website

String name

Page homePage

allPages[Page];

public class Page (Vertex)

Sting Link: Url

LinksTo[Edge] : This a list of edges/ pages, if we look at the weighted graph mentioned earlier

PossibleActions[Action]

Public class Edge

RequiredActions[Action]: actions required to navigate to the linked page. The size of this list will be the weight of the edge.

linkTo: Page

public class Action

Result (Something is done/ now posible to do, an example would be that a user that has performed the login action can now view their homepage)

Time (time it takes to perform this action)

Bool performed: Has the action already been performed

Time performed: When was the action performed

Public class User

Name: user name or id

Arraylist<Action> userActions

Role: 'if the website has multiple roles'

Page current: The page that the user is currently on.

Scenarios

Generating clickstream data

This method will generate clickstream data and specify how many different users the data should consist of and the min number of actions they will perform.

```
ClickstreamGenerator generator = new ClickStreamGenerator();

Arraylist<User> data = generator.genererateClickstream(#ofUsers, #ofActionsPerUser);
```

Another way of doing this would be to pass the arguments to the constructor and then call generate.

```
ClickstreamGenerator generator = new ClickStreamGenerator(#ofUsers, #ofActionsPerUser);

generator.generate();
```

Generating Json data

Over we can see that the data generated is an arraylist of UserData objects, converting it to json, is rather easy if we are using GSON. We iterate the list of actions and

```
Arraylist<UserAction> data = generator.genererateClickstream(#ofUsers, #ofActionsPerUser);

String jsonData = "";

For(UserAction userAction: data)

        jsonData += userAction.toJson();
```

This will require the GSON library.

Another way of doing this is that the framework has a method called generateJson() which will retum as string with the data. This is

```
String data = generator.gererateJson(arguments);
```

Write data to a file

A client might want to write the data directly to a file, the framework will allow this to be done in either JSON or xml format. The client will be able to specify what file the data should be written to by providing a string with the filename. If the file does exist, the data will be added at the bottom, if not a file will be created.

```
Arraylist<UserAction> data = generator.genererateClickstream(#ofUsers, #ofActionsPerUser);

For(UserAction userAction: data){

        userAction.writeJsonToFile('filename.json')

}
```

As we see here the arraylist has to be looped over. Another way of doing this would be to have a method that does it for the client.

```
generator.generateAndWriteTo('filename.json');
```

Here the idea is that the generator.generateAndWriteTo() method will find determine the file type and write data in the right format. If the file format is neighter json or xml, the method will return an error.

As this may be confusing and may give the client the impression that more than two formats are available a third option to consider would be to have two separate methods, one that writes json data to a file and another that wires xml data to a file.

```
generator.generateAndWriteToSjon('filename.json');
```

```
generator.generateAndWriteToXml('filename.xml');
```

Creating a website mock

Here the client can make an mock of a website represented as a weighted graph, where a vertex represents a page and weight represents the number of actions required to navigate to another vertex/ page. This 'website' can be used to generate clickstream data based on the mock.

A website will have one or more pages, and each page may have one or more actions and edges. An edge is basically a way to redirect to another page and requires one or more actions.

The website constructor takes a name string and a list of pages on this website.

```
Action clickArticleLink = new Action("Article link cliked") ;

Edge homeToArticle = new Edge(clickArticleLink,articlePage);

Page articlePage = new Page(articles.com/mysite.html);

Page homePage = new Page("mysite.com", clickArticleLink, homeToArticle);

ArrayList<Page> pages = new ArrayList();

pages.add(homePage);

pages.add(articlePage);

Website mySite = new Website(MySite .pages);
```

Defining Actions for a page

An action will have a result that is represented with a string, time it takes to do the action, represented in seconds, and a page that this action redirects the user to. A redirect page is not necessary to construct and action as not all actions will lead to a redirect. An action may also have a list of other actions it depends on to be performed. Example: a user has to fill put their info before making a purchase.

```
Action action = new Action("User Logged in", 1, ProfilePage);

Page myPage = new Page();

myPage.addAction(action);
```

Another way of doing this would be to create a page object and adding actions to it.

```
Page myPage = new Page();

myPage.addAction(new Action ("User Logged in", 1, ProfilePage));
```

Exhaust all routes

A client might want to see how many action are required on the website to exhaust all routes. In other words, how many actions/ clicks are required to visit every other page on the website from the homepage.

```
ClickstreamGenerator generator = new Clickstream();¨

Website myWebsite = new WebsiteI(Pages[]);

User userActions = generator.exhaust(myWebsite);
```

In the code above the generated data will only be for one user exhausting all routes. It might be necessary for the client to have multiple users traversing the site. One way of doing this would be to pass the number of users that are traversing the site in the exhaust method.

```
Arraylist<User>  userActions = generator.exhaust(myWebsite, 3);
```

Options

Options will let the client specify what actions and fields should be included/excluded in the generated data. As mentioned earlier the framework will by default contain data such as user info, type of action performed, time it was performed and so on. The client will have the ability to exclude any of those with this method.

```
ClickstreamGenerator generator = new ClickStreamGenerator();

Generator.exclude(user, time);
```

Another way of doing the same thing would be that the options can be passed to the constructor. The example bellow would exclude userId/name and time.

```
ClickstreamGenerator generator = new ClickStreamGenerator('-user', '-time');
```

The user should also be able to specify what else needs to be included. This can be done with the example above by switching the minus with a plus or with a method shown below. How the client will define the a field that needs to be included is still not decided. But the idea is that the framework will provide a set of attributes that can be included in the data, but are not by default.

```
ClickstreamGenerator generator = new ClickStreamGenerator();

Generator.include();
```

Finding the fastest route

Method that takes a mock of a website created by the client, starting page and a page we want to find the fastest route to. This will generate clickstream data consisting of actions a user has to do to get from a to b.

```
ClickstreamGenerator generator = new Clickstream();¨

Website myWebsite = new WebsiteI(Pages[]);

Generator.generateFastestRoute(myWebsite, pageA, pageB);
```

Clickstream generator Framework START HER MAGNUS!!!

API Classes of the initial API

Writers

Interface IWriter

This is an interface for all writers. A writer is a class that writes a set of actions to a file.

Methods

```
void writeToFile (String fileURL, List<UserAction> userActions) throws WrongFileTypeException;
```

It has one method to be overridden. It also throws a WrongFileTypeExcepting, this is an exception created for the subclasses *JSONWriter* and *CSVWriter*.

Class JSONWriter

Class that writes a set of user actions to a json file. The class impelments the Iwriter interface.

Methods

```
@Override
public void writeToFile(String fileURL, List<UserAction> userActions) throws WrongFileTypeException {
```

At this point the method is empty, but will write json data to a given file.

Class CSVWriter

Class that writes data to csv file. The class implements the IWriter interface.

Methods

```
@Override
public void writeToFile(String fileURL, List<UserAction> userActions) throws WrongFileTypeException {
```

At this point the method is empty, but will write json data to a given file.

Converters

Abstract class UserActionConverter

Class that converts a List<UserAction> to json or CSV format. The class is abstract because there is no reason for the user of the framework to create an instance of the class, as it has only one fuction.

Methods

```
public static void convertUserActionToJson(List<UserAction> action){
    //TODO: Implemt method to convert userAction to JSON format
    // Should return a string
}

public static void convertUserActionsToCSV(List<UserAction> action){
    //TODO: Implemt method to convert userAction to CSV format
    // Should return a string
}
```

The class has two methods that convert a list of UserAction to JSON and CSV respectively.

Website

Class Website

Class that represents a website object. A website will be trawersed by the generator to generate user action data.

Field variables

```
private String name;
private Page homePage;
private List<Page> allPages;
```

The website has a name, a home page and a list of pages existing on the website.

Class Page

Class that represents a webpage that is a part of a website.

Field variables

```
private String url;
private List<Page> linkedPages;
private List<Action> possibleActions;
```

The page has an URL, a list of pages that are linked to this page, and a list of actions that are possible to perform.

Methods

So far this class only has a constructor, some getters and a method to add an action to the list of possible actions.

Class Action

Class that represents an action that can be performed on a webpage.

Field variables

```
private String actionId;
private int timeActionTakesToPerformInMs;
private boolean redirectingActions;
private Page redirectsToPage;
```

An action has:

- An id
- A measure of time it takes to perform the action i ms
- A boolean that tells us if this action redirects the user to another page.
- A page that the action redirects to.

Methods

So far the class obly has a constructor and some getters.

Class User

Class that representes user of a webpage.

Field variables

```
private String id;
private List<UserAction> performedActions;
private Page currentViewingPage;
private List<Page> vistitedPages;
```

A user has:

- An id
- List of UserAction
- A page, currentViewingPage, that lets us log where a user ''now''.
- A list of visited pages

Class UserAction

Class that represents an action performed by a user. Contains only data.

Field variables

```
private String userId;
private String actionId;
private String urlOfPageActionWasPerformedOn;
private LocalDateTime timeActionWasPerformed;
```

A UserAction logs:

- The id of the user
- The id of the action
- The URL of the page the given action was performed on
- The date and time the action was performed

Class UserActionBuilder

The framework aims to use the builder method, where builder classes are used to initiate objects with the build method. This eliminates the need of overloading the constructors and forcing users to pass huge quantities of arguments when initiating an object. This class will be used to create UserAction objects and will mainly be used by the ClickStreamGenerator class to decide what fields/ columns to include in the generated data.

Field variables

```
private String userId;
private String actionId;
private String urlOfPageActionWasPerformedOn;
private LocalDateTime timeActionWasPerformed;
```

The UserActionBuilder class has the same field variables as the UserAction class.

Methods

The class has setter methods for all field variables that return the instance of the builder object.

```
public UserAction build(){
    return new UserAction(userId,actionId,urlOfPageActionWasPerformedOn,timeActionWasPerformed);
}
```

It also contains a build method that uses the constructor of UserAction and returns a UserAction object.

Generator

Class ClickStreamGenerator

Class that generates clickstream data. The main class of the framework, because it genereates the data.

Field variables

```
private Website website;
private List<User> users;
private List<UserAction> generatedActions;
private int numberOfLinesToGenerate;
private List<String> includeInGeneratedData;
```

The generator has:

- A website to be traversed
- List of users to traverse the website
- List of user actions that are generated
- A list of what should be logged, the idea being that the user of the framework should be able to include and exclude field from this list in the data that is generated.

Methods

```
public List<UserAction> generateCliksteram(){
    //TODO: generate actions
    return generatedActions;
}
```

The general method to create clickstreamdata. This method will be overloaded, meaning it will have multiple versions taking different parameters, where the user can spesify the number of users and/ or the number of actions performed by a user.

```
public List<UserAction> exhaus(Website website){
    //TODO: let the user traverse the website, until they have visited all pages
    return generatedActions;
}
```

A method that takes a website and returns a set of user actions required to travertse every route of the website.

```
public List<UserAction> generateFastestRoute(Website website, Page fromPage, Page toPage){
    //  TODO: impement the method that finds the fastest route from one page to another in a website
    return generatedActions;
}
```

A method that takes a website and two pages. The method returns a set of actions needed to navigate from the fromPage to the toPage.

```
public void exclude(String fieldToExclude){
    if (includeInGeneratedData.contains(fieldToExclude)){
        includeInGeneratedData.remove(fieldToExclude);
    }
}
```

A method that lets the user exclude any fields in the data that will be generated. The idea being that the framework will have a set of fields/values that will be logged, like userId or date and time, and the user can choose to exclude them.

Scenarios

Some of the scenarios have been revised as the framework was being created, these will include a comment on why and how they have been revised.

Generating clickstream data

With this we create an instance of the generator, and return a List of UserAction with a set number of users, performing a set of actions on a website. Here the framework user will not set a website, number of users or number of actions performed by those users.

```
ClickStreamGenerator generator = new ClickStreamGenerator();
List<UserAction> data = generator.generateCliksteram();
```

Generating Json data

Over we can see that the data generated is a list of UserAction objects. To convert this data to JSON an abstract converter class is used. It has methods to convert a list of UserAction to Json format.

```
ClickStreamGenerator generator = new ClickStreamGenerator();
List<UserAction> data = generator.generateCliksteram();
String dataInJson = UserActionConverter.convertUserActionToJson(data);
```

This has been revised from the original framework design with the introduction of the UserActionConverter class.

Write data to file

A client might want to write the data directly to a file, the framework will allow this to be done in either JSON or xml format. The client will be able to specify what file the data should be written to by

providing a string with the filename. If the file does exist, the data will be added at the bottom, if not a file will be created. In this case, if the file extension is not json, it will throw a *WrongFileTypeException*.

```
ClickStreamGenerator generator = new ClickStreamGenerator();
List<UserAction> data = generator.generateCliksteram();


JSONWriter writer = new JSONWriter();

try {
    writer.writeToFile( fileURL: "hei.json",data);
}
catch (WrongFileTypeException e){
    e.getMessage();
}
```

This scenario has been revised from the original design, instead of going through the generator, the framework user will use a writer object to write to JSON or CSV file.

Creating a website mock

The framework user may want to create their own website mock to generate clickstream data based on that mock. In this scenario the user creates a website with two pages, *homePage* and *articlePage*. With the homepage having the action to navigate to *articlePage*.

```
Page articlePage = new Page( url: "mysite.com/article", linkedPages: null,  possibleActions: null);

Action navigateToArticle = new Action( actionId: "Acticle link clicked",  timeActionTakesToPerformInMs: 2, redirectingActions: true,articlePage);
List<Action> actionList = new ArrayList<>();
actionList.add(navigateToArticle);

List<Page> linkedToHomePage = new ArrayList<>();
linkedToHomePage.add(articlePage);
Page homePage = new Page( url: "mysite.com",linkedToHomePage,actionList);

List<Page> allWebsitePages = new ArrayList<>();
allWebsitePages.add(homePage);
allWebsitePages.add(articlePage);

Website myWebsite = new Website( name: "MyWebsite",homePage,allWebsitePages);
```

This scenario has been revised from the original design because of the removal of the Edge class.

Defining actions for a page

An action will have a result that is represented with a string, time it takes to do the action, represented in seconds, and a page that this action redirects the user to. A redirect page is not necessary to construct and action as not all actions will lead to a redirect.

```
Action oneAction = new Action( actionId: "User logged in", timeActionTakesToPerformInMs: 2, redirectingActions: false, redirectsToPage: null);
Page myPage = new Page();
myPage.addAction(oneAction);
```

Revised from original design: An action no longer has have a list of other actions it depends on to be performed.

Exhaust all routes

A client might want to see how many action are required on the website to exhaust all routes. In other words, how many actions/ clicks are required to visit every other page on the website from the homepage. Here we use the website created in the “Creating a website mock scenario”.

```
ClickStreamGenerator generator1 = new ClickStreamGenerator();
List<UserAction> actionsToExhaustAllRoutes = generator1.exhaust(myWebsite);
```

Options

Options will let the client specify what actions and fields should be included/excluded in the generated data. As mentioned earlier the framework will by default contain data such as user info, type of action performed, time it was performed and so on. The client will have the ability to exclude any of those with this method.

```
ClickStreamGenerator anotherGenerator = new ClickStreamGenerator();
generator.exclude( fieldToExclude: "userId");
```

Revised from initial API Design Spec

Action:

Removed action performed bool

Added UserAction class

Edge class has been removed, and a page now has a list of pages it linkes to.
This is done after rethinking the idea of having multiple actions leading to a redirect which was not realistic.

Some of the scenarios.