Clickstream generator Framework

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 analyse 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. 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.

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 Design Specification

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 = new ArrayList<>();
private List<Action> possibleActions = new ArrayList<>();
```

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;
private double chanceOfActonBeingPerformed;
```

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.
- A chance that the action will be performed, with this we can set the probability of the action being performed by the user object.

Methods

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

User

Class User

Class that represents 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= new ArrayList<>(Arrays.asList("userId", "actionId", "urlOfPageActionWasPerformedOn", "timeActionWasPerformed"));
```

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 traverse 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");
```

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 anotherGenerator = new ClickStreamGenerator();
generator.generateFastestRoute(myWebsite,homePage,articlePage);
```

Feedback from user testing

Magnus Møllervik – Student at Høgskolen in Østfold

Magnus expressed the need for builder classes for initiating website, page and action classes. The framework already included one builder class, UserActionBuilder, but it wasn't used in any of the scenarios. Although the current solution was on par with how one usually initiates objects in OOP, the framework would not be intuitive enough. He preferred being able to: quote "dot his way through learning a new framework". The feedback was related to the scenarios of *creating a website mock* and *defining actions for a page*. He also pointed out that there was an inconsistency in the framework, where one could add one action to a page, but had to add multiple pages to a website.

Mathias Nilsen -Student and Teachers assistant in Frameworks at Høgskolen in Østfold

Mathias also mentioned on the need for more builder classes to make the framework more intuitive. Especially for the ClickStreanGenerator class since it would contain a default website and a default set of users and other field variables. He also pointed out the need for creating interfaces to give the user the ability to create their own implementation of methods. His example being the one interface that already existed, IWriter. This would give the user the ability to implement their own version of writeToFile, this would make it easier for the user if they wanted to write data to an *xml*- or any other filetype.

Result of user testing

As requested by the testers builder classes have now been created for all the classes the user is going to create. Following the builder pattern, all these classes have a *build()* method that returns an stance of the object, rather than the builder. Furthermore, all add methods for page, action and website that added a list, now have another method to add a single object.

Bellow a summary of the implementation of these classes as well as examples of how they have affected the scenarios.

Classes added as a result of user testing

ClickStreamGeneratorBuilder

Field variables

```
private Website website;
private List<User> users = new ArrayList<>();
private int numberOfLinesToGenerate = 100;
//TODO: This list should contain values from AvailableOptions enum
private List<String> includeInGeneratedData= new ArrayList<>(Arrays.asList("userId", "ationId", "urlOfPageActionWasPerformedOn", "timeActionWasPerformed"));
```

ClickStreamGeneratorBuilder has the same field variables as the ClickStreamGenerator class, but with some default values. This makes it easier for the user of the framework to create an general object of the class without having to go through multiple constructors.

Methods

Following the builder pattern this class has multiple setters, that all return as object of the builder.

From Magnus' feedback methods to add one or a list of users has been added.

```
public ClickStreamGeneratorBuilder addUsers(List<User> users) {
    this.users.addAll(users);
    return this;
}

public ClickStreamGeneratorBuilder addUser(User user) {
    this.users.add(user);
    return this;
}
```

Furthermore the methods to include or exclude fields/ columns in the data that the a ClickStreamGenerator creates have been moved to the builder, and reimagined. Now, instead of sending a list of fields to include/exclude, each field will have its own method as shown below.

```
public ClickStreamGeneratorBuilder includeIpAdress(){
    //TODO: create multiple include...() methods, so the user can choose to include non-defauld columns
    // in the data that is generated
    return this;
}

public ClickStreamGeneratorBuilder excludeDate(){
    //TODO: Create multiple exlude...() methods, so the user can choose to exclude columns
    // in the data that is generated
    return this;
}
```

WebsiteBuilder, PageBuilder, ActionBuilder

The WebsiteBuilder class also follow the builder pattern, having the same field variables as the classes they are building. They also have setters for their field variables that return an instance of the builder, if the field variable is a list, the builder has two add methods, one that ads a single object, and one that ads a list of objects. The build methods return an instance of the class they are builders for.

There are no screenshots added for these classes as they are mainly data classes, classes that only hold data, and have similar behavior.

Revised scenarios

Generating clickstream data

Now the user will use the builder class to generate an instance of the generator. In this example all values/ options are set to default. G

```
// Generate clickstream data without any options specified
ClickStreamGenerator generator = new ClickStreamGeneratorBuilder().build();
List<UserAction> data = generator.generateCliksteram();
```

Creating a website mock/ Defining action for a page

This can now be done in multiple ways. The user can create a default website with just the builder.

```
//Creating a default website mock with builder
Website myWebsite = new WebsiteBulder().build();
```

Or they can use the setters, page and action builders to customize the website.

```
//Creating a customized website mock with the builder
Page homePage = new PageBuilder().setUrl("anotherwebsite.com").addPossibleAction(new ActionBuilder().setActionId("Menu clicked").build()).build();
Website anotherWebite = new WebsiteBulder().setHomePage(homePage).setName("AnotherWebsite").addPage(homePage).build();
```

Following the feedback from both testers, it is easy to see that this creates a more user friendly and intuitive framework. The amount of code that needs to be written has dramatically decreased. Here the user creates a page, setting the URL and adding an action with a specified id to the page, all in one line. Then building the website mock in another line.

Writing to other files

As Mathias pointed out a interface makes it easier for the user to implement their own *writeToFile()* methods if they ever have the need to write to any other file type. No other interfaces have been implanted at this point, a interface of the converter class will be implemented in the near future, allowing the user to convert UserActions to any format. The example bellow demonstrates how the user can user the interface to implement their own versions of methods.

```
// User implementing the they own implementation of the writeToFile() through the IWriter interface
IWriter myWriter = new IWriter() {
    @Override
    public void writeToFile(String fileURL, List<UserAction> userActions) throws WrongFileTypeException {
        // write to my file
    }
};

try {
    myWriter.writeToFile( fileURL: "afile.xml",data);
}
catch (WrongFileTypeException e){
    e.getMessage();
}
```

This can also be done using Lambda expressions.

Options

The original Framework design required the user to send in a string with fields they wanted to include or exclude with the *include(field to iclude)* or *exclude(field to exclude)* methods respectively.

This was in no way intuitive, as the user would have to learn all available and default fields/ columns that exist in the data the generator creates.

```
//Defining fields/ columns to be logged by the framework
ClickStreamGenerator anotherGenerator = new ClickStreamGeneratorBuilder().includeIpAdress().excludeDate().build();
```

Now the user can use the IDE-suggestions to pick and choose.

Created builders

Include()/exclude methods are now in the clickstreambuilder.

Available options is now an enum

Other revisions from original design.

- The Action class no longer has a actionPerformed boolean.
- User now logs the actions performed with a list of UserAction.
- Edge class has been removed, now a page has a list of pages it is connected to. In other words a page ha contains links and a user object can navigate to these pages by performing actions (clicking the links)