CART - Conversational Agent Research Toolkit Documentation

Release 1.00

(Anonymized)

May 06, 2021

Contents

CHAPTER 1

Guide

1.1 About CART

1.1.1 Objectives

The Conversational Agent Research Toolkit (CART) aims at enabling researchers to create conversational agents for experimental studies using computational methods. CART provides a unifying toolkit written in Python that integrates existing services and APIs for dialogue management, natural language understanding & generation, and frameworks that enable publishing the conversational agents as either a web interface or within messaging apps. Specifically developed for communication research, CART not only acts as an integration layer across these different services, but also aims to provides a configurable solution meeting the requirements of academic research.

1.1.2 Components

CART acts as an integration layer between several services so it can generate a conversational agent that can interact with participants of an experiment, log the conversations, design experiments, and connect with questionnaires for self-reported measures. To do so, CART works with the following components:

Dialogue Management

CART currently uses DialogFlow as the primary tool to handle dialogue managemnt. This means that all the participant input, and the responses that the agent gives, are primarily setup in DialogFlow. Using specific tokens (see below), the researcher can customise the responses that the agent provides depending on the condition that the participant is in.

Agent Publication

CART currently uses the Microsoft Azure Bot Channel Registration to publish the conversational agent. Within CART, the Bot Framework is responsible for creating a webchat for users to chat with the agent. This webchat can be embedded in online surveys. The Bot Framework also allows the agent to be published in other channels (e.g., Skype, Telegram, Facebook Messenger) without needing to change the code within CART.

Conversation Logging

One of the key aspects of CART is the ability to log the conversations that participants in an experiment have with the agent. To do so, CART connects to a database under the control of the researcher.

Experiment Design

CART allows the researcher to create experiments in which the same agent acts in different ways depending on the condition that the participant is in. To do so, the researcher simply needs to add specific tokens in the Dialogue Management tool, and setup different conditions within CART itself.

Online Questionnaire Integration

After an agent is created and published, the researcher can integrate it to the flow of an online questionnaire (e.g., Qualtrics, SurveyMonkey). By turning on the *questionnaire flow*, the agent requests a unique ID from the participant, and returns a unique conversation code at the end of the conversation, so the researcher can link the conversation logs in the CART database with the responses in the online questionnaire.

1.2 Installation

1.2.1 Requirements

Services and APIs

CART acts as an integration layer between different services (and APIs), and specific configurations within CART itself. To use it, you will need:

1. Access to a SQL database
2. Access to a web service to publish CART (using Python 3.X)
3. An account with DialogFlow
4. An account with the Microsoft Bot Framework
5. A copy of the CART code

Optionally, you also would need access to an online questionnaire tool (e.g., Qualtrics, SurveyMonkey) if you are integrating CART into a survey flow.

Note: For #1 and #2, you can use your own server (if available), or AWS RDS (database) and Heroku (web service) as demonstrated in the *Installation and Setup Guide*.

Python Packages

CART uses a set of Python packages (e.g., PyMySQL, microsoftbotframework, google-cloud-dialogflow etc.). By downloading CART and running the step-by-step instructions in the installation guide (below), all the necessary packages will be installed on the server.

1.2.2 Installation and Setup Guide

Important note: these steps show how to install and setup an agent using *AWS RDS* (as a database server) and *Heroku* (as a web service) along with *DialogFlow* and the *MS Bot Framework*. Advanced users can replace *AWS RDS* and *Heroku* by their own servers.

Step 1. Download CART to your computer

1. Access [ANONYMIZED LINK OF THE GITHUB REPOSITORY] and select `clone or download` and download CART as a zip file.
2. Extract the zip file in a folder you want to store the agent

Step 2. Rename the config.yaml file

The `config.yaml` file, located at the folder `cart`, will contain all the basic configurations needed to connect to the services (MS Bot Framework and the SQL database), including usernames and passwords. **Never make it publicly available**.

To create it, you need to:

1. Copy (or rename) the file called `config_example.yaml` to `config.yaml`
2. In the steps 3, 4 and 6 (below), you will need to update this file with information coming from each service.

Step 3. Create an agent in DialogFlow

1. Log in to DialogFlow and select `Create Agent`. For CART, all tests have been done with the ES (simpler) version.
2. Follow the instructions from DialogFlow to enable the API, create a service account, and download the service key account file in JSON format. You will use it on step 5, as part of the web service configuration.

Step 4. Create a database for logging

CART requires the the URL (host) of a MySQL database, database name, and username and password to connect to and log the conversations between the agent and participants. The username provided needs to have all privileges (including CREATE) in the database. This information needs to be made included in the `config.yaml` file.

If using AWS RDS, the following steps need to be followed:

1. Log into your AWS account and select RDS
2. Create a database using MySQL, and make sure to include the username and the password also in the `config.yaml` file
3. When asked for the database name, make sure to inform it, and also include it in the `config.yaml` file.
4. Select the endpoint of the database (check the dabatase details page), and paste it in the `database_url` field of `config.yaml`.
5. After the database is launched, make sure to check the security group (see database details), and open the inbound port to the server that you will be using.

Notes:

- Usually a micro or small instance in AWS is sufficient for testing purposes, and can later be upgraded to medium/large when the agent is live and handling several conversations at the same time (e.g., during an experiment)
- For security reasons, it is recommended not to allow any IP to connect to your database. When using Heroku, see this list of plugins that can create a static IP for your app.
- **When running the agent in the server for the first time, it will automatically create two tables in the database:**
  - `logs`, which records every interaction between the agent and the participant
  - `conversations`, which records each individual conversation (i.e., in general, each participant = one conversation) that is started with the agent.

Step 5. Publish the agent as a web service

After completing the steps above, it is time to publish the agent as a web service. This can be done using Heroku (as demonstrated below) or in any other server that supports Python and Flask applications. It is important to note that the server should also be able to serve pages in https.

If using Heroku:

1. Log into your Heroku account and create a new app
2. Set the environment variables (called in Heroku "config vars") for DialogFlow. See note 2, below, for details.
3. Select the deployment method
4. Deploy the app
5. After the build has been completed, select `open app`
6. Copy the URL of the app (it should start with the app-name, and end with `herokuapp.com`) to use in the next step

Note 1: the URL of the app is needed so it can be registered in the MS Bot Framework (next step). The registration in the MS Bot Framework will also provide authentication credentials. These credentials will need to be added to the `config.yaml` file, and the agent will need to be published again in Heroku (as outlined in the next step).

Note 2: Open the service account file downloaded on step 3(above) locally in a text editor, remove all line breaks, and substitute the double quotes (") by single quotes ('). In the web service, add this as an environment variable called DF_CREDENTIALS. Create another environment variable called DF_LANGUAGE_CODE and set its value to the appropriate language (e.g., en).

Step 6. Connect the agent to the MS Bot Framework

After a URL for the agent as a web service is available (e.g., for Heroku: `https://NAMEOFTHEAPP.herokuapp.com/`), the agent can be registered in the MS Bot Framework. To do so:

1. Log in your Microsoft Bot Framework account, selecting `My Bots`
2. Select `create a bot`. You will be redirected to Azure Bot Service
3. Select `Bot Channels Registration`
4. Provide the information required
5. The messaging endpoint will be the URL of the Heroku app + `api/messages` - example: `https://NAMEOFTHEAPP.herokuapp.com//api/messages`

6. After the channel registration is deployed, select `Go to resource` (or simply open it in Azure)
7. In the `Settings`, go to the Microsoft App ID area
8. Copy the Microsoft App ID and add it to the `config.yaml` file under `app_client_id`
9. Click on `Manage` for the Microsoft App ID
10. In the new window, select `Generate new password`. Copy this password and add it to the `config.yaml` file under `app_client_secret`
11. Click on `save` and close this window
12. In the `config.yaml` file, add the name of the agent under `agent_name`
13. Go back to Heroku and re-deploy the app (with the latest version of the `config.yaml` file).
14. After the redeployment, you can use `Test in Web Chat` function on Azure Bot Service (same area where the `Settings` were) to test the connection.

Step 7. Customize the agent

After the agent is connected to the MS Bot Framework, the basic setup is done. The researcher can then use several features within CART to customize the agent. For more details, see *Using CART*

Step 8. Making the agent available

After the agent is ready to interact with users, you can use the Microsoft Bot Framework to publish it as a Web Chat (see `Get bot embed codes`), or on other channels such as Skype, Facebook Messenger, or Telegram.

1.3 Using CART

1.3.1 Installation

The basic steps to set up an agent are explained in *Installation and Setup Guide*. After the basic setup is done, the agent should be up and running - i.e., providing responses - even if irrelevant - to inputs provided by the participant through the web chat (or other channels).

With the agent running, the researcher can then customize the agent for an experiment, as outlined in the following sections.

1.3.2 Setting up the basic dialogue

CART uses DialogFlow for the basic dialogue setup and dialogue management. After the basic setup is done, the researcher should use DialogFlow to setup how an agent should interact with a participant.

The best practice is to start with dialogue design from the moment of a greeting. All new agents created in DialogFlow come with a `Default Welcome Intent`, which determines what the agent should do when a participant greets the agent. As a first step, it is recommended that the researcher reviews this intent, and customize the answers of the agents depending on how the researcher would like the dialogue to flow.

After this initial configuration, the researcher can configure additional intents (e.g., if a participant says they want to know the weather forecast, or how much something costs), and how the agent should react when an intent is recognized. For more information, it is suggested to review the Building an agent tutorial in DialogFlow.

By using DialogFlow through CART, the researcher not only has all conversations logged in a database, but also has an extra layer of control over the basic DialogFlow functionality, as outlined in the options below.

1.3.3 Creating experimental conditions

CART can assign participants automatically to experimental conditions, as outlined in this section, or rely on an integration with an online questionnaire to do so (see *Integrating with online questionnaires*).

To assign participants to experimental conditions using CART, the researcher should edit the section `experimental_design` of the `config.yaml` file in the following manner:

1. Set `assignment_manager` to `CART`
2. **Select the `assignment_method`. Three options can be used:**
    a. `random_balanced`, which tries to keep the number of participants per condition always equal, and randomly assigns a new participant to a condition when the number of participants per condition is the same. When the number of participants per condition is different, a new participant is assigned to the condition with the lowest number of participants.
    b. `fully_random`, which randomly assigns the participant to a condition. As it is fully random, the conditions may be unbalanced if the sample size is low.
    c. `sequential`, which assigns the first participant to the first condition, and the next participant to the next available condition, sequentially. Especially for experiments with low sample sizes, this may be a simpler way to ensure balance between conditions.
3. In the `conditions` section, include as many conditions as needed, always using the structure shown in the `config_template.yaml`, i.e., adding a new condition by having one indentation with condition_NUMBER, and another indentation with `condition_name`, and then including the actual condition name (without spaces). The `condition_name` will be stored in the database (table: conversations), along with participant details.

After the conditions are created, the researcher can customise the responses that the dialogue manager provides to the participant depending on the condition that the participant is in. For more details, see *Customising responses*.

1.3.4 Customising responses

CART allows the researcher to customise responses that the dialogue manager gives to the participant. This can be done in regardless of the experimental condition of the participant, or by experimental condition. This is managed in the `rephrases` section of the `config.yaml` file.

Same rephrase for all conditions

To customise (rephrase) the response from the dialogue manager in the same way regardless of the experimental condition, the researcher should add the following information to the `all_conditions` section:

- TOKEN - all in uppercase and without spaces. The token is the part of the response from the dialogue manager (between square brackets) that needs to be substituted.
- New text - what should be included instead of the token in the response

For example: if in the `config.yaml` file, the following line is added:

```
rephrases:
    all_conditions:
        EXAMPLETOKEN1: changed text
```

And in the dialogue manager, the response configured is `This is the [EXAMPLETOKEN1].`

The agent will respond to the participant: `This is the changed text.`

It is important to note that in the dialogue manager, the token has square brackets, but in the `config.yaml` file, it doesn't.

Different rephrase depending on the experimental condition

To customise (rephrase) the response from the dialogue manager differently per experimental condition, the researcher should create a section per condition (if it doesn't exist), and then add the same token in each condition.

For example: if in the `config.yaml` file, the following line is added:

```
rephrases:
    condition_1:
        EXAMPLE1: changed text condition_1
    condition_2:
        EXAMPLE1: changed text condition_2
```

And in the dialogue manager, the response configured is `This is the [EXAMPLE1].`

If the participant is in condition_1, the response to the participant will be: `This is the changed text condition_1.`, and `This is the changed text condition_2.` if the participant is in condition_2.

It is important to note that in the dialogue manager, the token has square brackets, but in the `config.yaml` file, it doesn't. The conditions need to be configured in the `experimental_design` section, as outlined in *Creating experimental conditions*.

Providing the conversation code in the message

CART assigns automatically a conversation code to each new conversation (participant). The conversation code is a shorter, more readable id that the researcher can have the agent provide the participant in specific contexts, such as when ending the conversation.

To configure the conversation code, the `conversationcode_suffix` and the `conversationcode_base` of the `config.yaml` file need to be filled out. The suffix is a set of letters that will be at the beginning of the conversation code, and the base is the starting number of the conversation code (to prevent the first participant from getting a conversation code = 0). CART automatically increments the conversation code number starting from the base. For example, if `conversationcode_suffix` is equal to `A` and the `conversationcode_base` is equal to `500`, the first participant will receive the conversation_code `A500` and the second participant, `A501`.

After this is setup, any time that a message - coming from the dialogue manager or the overrides - contains the string `|CONVERSATIONCODE|`, this string will be replaced by the specific conversation code of the participant.

1.3.5 Overriding the dialogue manager

There may be instances in which the researcher does not want to send the message from the participant to the dialogue manager, and instead may want to directly provide a response to the participant. To do so, the researcher should add a new override to the `overrides` section of the `config.yaml` file, with the following information:

- override_reason - a keyword with the reason why the override is being done. This will be stored in the database (table: logs) as an event.
- override_terms_in_user_message - a list of words or strings to be searched for in the user_message (separated by commas, without punctuation or special characters)

- override_terms_case_sensitive - indicate whether the term that will be searched for in the user_message are case-sensitive (True) or not (False)
- **override_terms_matching - indicate how the terms for the override should be searched for in the user_message. Three opt**
  - If `override_terms_matching` is set to `full_string`, it checks the user_message is equal to any of the terms in `override_terms_in_user_message`. For example, if the user_message is "apple" and this is one of the terms, then it will meet the condition. If the user_message is "I like apple", it won't match.
  - If `override_terms_matching` is set to `string`, it checks if each of the terms in `override_terms_in_user_message` is present in the user_message using string matching. For example, the term "apple" would be found in the messages "I like apple" and "I like pineapples".
  - If `override_terms_matching` is set to `tokens`, it first splits the user_message in tokens (removing any punctuation marks or special characters), and then checks if any of the tokens of the user_message is present in the terms from `override_terms_in_user_message`. For example, the term "apple" would be found in the messages "I like apple" but not in "I like pineapples".
- override_response_from_agent: the response that the agent should give to the participant.

As an example, the researcher may want to configure the agent to simply say good bye and provide the conversation code to the participant, instead of going to the dialogue manager. To do so, two pieces of code need to be added to the `config.yaml` file.

In the `overrides` section:

```
override_1:
    override_reason: quit
    override_terms_case_sensitive: False
    override_terms_matching: full_string
    override_terms_in_user_message: bye, good bye, end, quit, stop
    override_response_from_agent: '[PROVIDECONVERSATIONCODE]'
```

In the `rephrases` section:

```
all_conditions:
    PROVIDECONVERSATIONCODE: Good bye! Your conversation code is |CONVERSATIONCODE|.
```

Note: When configuring items to be rephrased in responses by the override, square brackets should be used in the text, encapsulated by single quotes (as done above: '[PROVIDECONVERSATIONCODE]').

1.3.6 Connecting intents from the dialogue manager

While the dialogue manager (DialogFlow) allows to route conversations from one intent to the other via its own interface, there may be instances in which the researcher will want to do it directly in CART, especially when CART's custom logic is also in place. For example, if validating a participant id is needed, the welcome intent should simply ask the participant for a participant id. CART will check if the participant id provided is valid - and will override this if necessary - and will allow the conversation to continue. If the researcher wants the agent to immediately ask a question to the participant (instead of waiting for the participant to ask a question to the agent), it is easier to make sure that the welcome intent redirects the participant to a new intent.

Two ways can be used:

- Adding a follow-up intent in DialogFlow directly
- Using CART to connect the Welcome Intent to the new intent

To use the second option, the researcher needs to make some configurations in DialogFlow, and others in CART. In DialogFlow, the Text response of the Welcome intent should be a token, that will be picked up by CART. For example, it could simply be `[PARTICIPANTID_VALID]` (as what the Welcome intent is doing is validating the participant id). A second intent, which could be the experimental setting, would be configured in DialogFlow to start with the input `[START_EXPERIMENT]`.

In the `config.yaml` file, the researcher would need to add a `connect_intents` section, with the following code::

```
connect_intents:
    PARTICIPANTID_VALID: START_EXPERIMENT
```

CART will then know that when a response from DialogFlow comes with the token `[PARTICIPANTID_VALID]` it should not provide a direct response to the participant, and rather call DialogFlow again with the token `[START_EXPERIMENT]` as if it were the user message. DialogFlow will then start with the associated intent, and return the appropriate question that should be sent to the participant in order to continue with the flow.

1.3.7 Integrating custom functions

CART allows the researcher to integrate custom functions, including (pre-trained) classifiers to the conversation flow. To do so, the researcher needs to edit the file `special_functions.py` inside the `helpers` folder, and add a new function. This function always takes the user_message as an input (i.e., what the participant chatted with the agent), and can return an output (or not) upon which CART will determine whether an override action should be taken. The `config.yaml` file also needs to be edited with a new function section, containing:

- function_name: the name of the function included in the `special_functions.py` file
- store_output: at present, it always needs to be in the `logs` table.
- store_output_field: the name of the field in which the output of the function will be stored in the `logs` table within the database.
- store_output_field_type: the type of field (following MySQL standards) in which the output will be stored. The most common values should be `float`, `int` or `text`
- function_action: if `True`, CART will evaluate the output of the function and may create an override depending on the result. If `False`, CART will only store the output of the function in the database, but will not change the conversation.
- function_comparison: the Python code (written between quotes) that should be executed - considering that the output of the function will be at the beginning of the code. The code should always provide a True or False as result. For example, if the idea is to see if the output of the function is positive, the code written should be `' > 0 '`.
- function_comparison_met: exact text that should be picked up by the override function (so that the change in the conversation can take place)

Note: if any additional Python modules are required, they should be added to the requirements.txt file in the root folder so they can be installed in the server when starting up CART.

The following code integrates an out of the box sentiment analysis classifier (Vader) in the workflow of CART. It stores the output of the sentiment analysis in the logs, and turns on an override when the sentiment of a text by the participant is lower than -0.85.

Added to `requirements.txt`:

```
vaderSentiment
```

Added to `config.yaml` in the section special functions:

```
special_functions:
    function_1:
        function_name: check_sentiment
        store_output: logs
        store_output_field: sentiment
        store_output_field_type: float
        function_action: True
        function_comparison: ' < -0.85'
        function_comparison_met: sentiment_analysis_too_low
```

Added to `config.yaml` in the section overrides:

```
overrides:
    override_sentiment:
        override_reason: sentiment_analysis_too_low
        override_terms_case_sensitive: True
        override_terms_matching: full_string
        override_terms_in_user_message: sentiment_analysis_too_low
        override_response_from_agent: '[SENTIMENTANALYSISTOOLOW]'
```

Added to `config.yaml` in the section rephrases:

```
rephrases:
    all_conditions:
        SENTIMENTANALYSISTOOLOW: I'm really sorry you feel this way. Maybe we should␣
↪stop our conversation?
```

Added to `special_functions.py`:

```
## EXAMPLE - SENTIMENT ANALYSIS
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
analyzer = SentimentIntensityAnalyzer()
def check_sentiment(user_message):
    try:
        sentiment = analyzer.polarity_scores(user_message)
        sentiment = sentiment['compound']
        return sentiment
    except:
        return None
```

1.3.8 Integrating with online questionnaires

CART can also be integrated with an online questionnaire flow, so that participants to an experiment can interact with the agent before, during, or after completing an online questionnaire. The integration ensures that a unique identifier is passed along between CART and the online questionnaire, allowing the researcher to link the questionnaire responses (self-reports) to the conversation logs between the participant and the agent.

The conversation with the agent can take place in three different moments of the questionnaire flow:

Before: Agent –> Questionnaire

In this scenario, the participant first has a conversation with the agent and, at the end of the conversation, the agent provides a link asking the respondent to complete the questionnaire. The link contains the conversation code (as a parameter), so that the survey tool managing the questionnaire can capture the parameter as metadata.

To do so, the `questionnaire_flow` from the `config.yaml` file needs to be edited with the following information:

- enabled: needs to be set to True
- moment: needs to be set to before
- rephrase_token: needs to contain the token (without square brackets) that the dialogue management will use to indicate that the agent needs to stop the conversation and send the link to the survey.
- rephrase_text: the text that the agent should say when inviting the participant to answer the questionnaire. The link to the questionnaire should also be added. The rephrase text uses markdown format, so the link is added as [LINKTEXT](URL)

The following example shows an integration with Qualtrics, with the conversation code being passed as a parameter in the URL (with the embedded metadata field called `convcode`:

```
questionnaire_flow:
    enabled: True
    moment: before
    config_before:
        rephrase_token: SENDTOSURVEY
        rephrase_text: Thank you for the conversation! Please **[complete the␣
↪survey](https://uvacommscience.eu.qualtrics.com/jfe/form/SV_1RgsQFnkOk4izqJ?
↪convcode=|CONVERSATIONCODE|)**
```

(Note: the rephrase_text shown above is *Thank you for the conversation! Please [complete the survey](https://qualtrics.com/jfe/form/XYZ?convcode=|CONVERSATIONCODE|)*)

The dialogue management (e.g., DialogFlow) would need to send `[SENDTOSURVEY]` as a response to trigger this behavior. The dialogue manager must only send this token in the response, as this option does not work with partial matching of the string (to be safe).

During: Questionnaire –> Agent –> Questionnaire

In this scenario, the participant starts with the questionnaire, has a conversation with the agent and, at the end of the conversation, the participant goes on to complete the questionnaire. The conversation with the agent can take place in a different channel (e.g., Skype, Telegram, or a webchat page published elsewhere), or be embedded in the questionnaire itself.

This integration is slightly more complex, as it is important to both ensure that CART stores a unique identifier of the participant at the beginning of the conversation, and that, when continuing on to the second part of the survey, the participant provides some proof that the conversation took place. To do so, the proposed flow works in the following way:

1. When starting the conversation with the agent, the agent asks the participant for a participant id.
2. The agent validates the participant id (according to rules specified by the researcher) and, if applicable, also assigns a condition to the participant based on the id.
3. When ending the conversation, the agent provides a conversation code, that the participant should copy and paste in the survey (to prove that the conversation took place).

The participant id in CART should always be a set of letters followed by a number. For example ABC11190320930293. The letters (ABC) will be used to assign the condition (if applicable) whereas the numbers can come from a random number generator in the online questionnaire, assigning a unique number to the respondent.

To do so, the `questionnaire_flow` from the `config.yaml` file needs to be edited with the following information:

- enabled: needs to be set to True

- moment: needs to be set to during
- rephrase_start_token: the token (e.g., [VALIDATEPARTICIPANTID]) that the dialogue manager provides as a response when the moment comes to validate the participant id.
- participantid_dialog_field: the name of the field, set in the dialogue manager, that stores the participant id (when provided by the respondent)
- participantid_valid_suffixes: the combinations of letters that should be accepted by the agent as part of the participant id.
- participantid_not_recognized: the token that should be provided to CART when a participant id is not recognised (or set as invalid)
- rephrase_end_token: the token (e.g., [SENDTOSURVEY]) that the dialogue management will use to indicate that the agent needs to stop the conversation and ask the participant to continue with the survey.
- rephrase_end_text: the text that the agent should say when asking the participant to continue with the survey. It is good practice to provide the conversation code in this text (with the |CONVERSATIONCODE| token)

The following example shows an integration with Qualtrics, with the conversation code being passed as a parameter in the URL (with the embedded metadata field called `convcode`:

In `config.yaml` at the questionnaire flow section:

```
questionnaire_flow:
    enabled: True
    moment: during
    config_during:
        rephrase_start_token: VALIDATEPARTICIPANTID
        participantid_dialog_field: participantid
        participantid_not_recognized: PARTICIPANTID_INVALID
        participantid_valid_suffixes: CO, TR
        rephrase_end_token: SENDTOSURVEY
        rephrase_end_text: Thank you for the conversation! You can now go to the next␣
↪page of the survey. Your conversation code is **|CONVERSATIONCODE|**.
```

In DialogFlow:

- Intent 1: Default Welcome Intent (so the every new conversation asks for a participant id):

```
* In action and parameters:
    * Required: yes
    * Parameter Name: participantid
    * Entity: @sys.any
    * Value: $participantid
    * Prompts: What's your participant id?
```

- Intent 2: Invalid participant id (used when the participant id is invalid):

```
* Machine Learning set to off
* Training Phrases: [PARTICIPANTID_INVALID]
* In action and parameters:
    * Required: yes
    * Parameter Name: participantid
    * Entity: @sys.any
    * Value: $participantid
    * Prompts: I'm sorry. The participant id you provided does not seem to be␣
↪valid. Could you please check in the online survey and let me know what your␣
↪participant id is?
```

After: Questionnaire –> Agent

In this scenario, the participant starts with the questionnaire and then is directed to a conversation with the agent. The conversation with the agent can take place in a different channel (e.g., Skype, Telegram, or a webchat page published elsewhere), or be embedded in the questionnaire itself.

It is important to ensure that CART stores a unique identifier of the participant at the beginning of the conversation. To do so, the proposed flow works in the following way:

1. When starting the conversation with the agent, the agent asks the participant for a participant id.
2. The agent validates the participant id (according to rules specified by the researcher) and, if applicable, also assigns a condition to the participant based on the id.

The participant id in CART should always be a set of letters followed by a number. For example ABC11190320930293. The letters (ABC) will be used to assign the condition (if applicable) whereas the numbers can come from a random number generator in the online questionnaire, assigning a unique number to the respondent.

To do so, the `questionnaire_flow` from the `config.yaml` file needs to be edited with the following information:

- enabled: needs to be set to True
- moment: needs to be set to after
- rephrase_start_token: the token (e.g., [VALIDATEPARTICIPANTID]) that the dialogue manager provides as a response when the moment comes to validate the participant id.
- participantid_dialog_field: the name of the field, set in the dialogue manager, that stores the participant id (when provided by the respondent)
- participantid_valid_suffixes: the combinations of letters that should be accepted by the agent as part of the participant id.
- participantid_not_recognized: the token that should be provided to CART when a participant id is not recognised (or set as invalid)

The following example shows an integration with Qualtrics, with the conversation code being passed as a parameter in the URL (with the embedded metadata field called `convcode`:

In `config.yaml` at the questionnaire flow section:

```
questionnaire_flow:
    enabled: True
    moment: after
    config_after:
        rephrase_start_token: VALIDATEPARTICIPANTID
        participantid_dialog_field: participantid
        participantid_not_recognized: PARTICIPANTID_INVALID
        participantid_valid_suffixes: CO, TR
```

In DialogFlow:

- Intent 1: Default Welcome Intent (so the every new conversation asks for a participant id):

```
* In action and parameters:
    * Required: yes
    * Parameter Name: participantid
    * Entity: @sys.any
    * Value: $participantid
    * Prompts: What's your participant id?
```

- Intent 2: Invalid participant id (used when the participant id is invalid):

```
* Machine Learning set to off
* Training Phrases: [PARTICIPANTID_INVALID]
* In action and parameters:
    * Required: yes
    * Parameter Name: participantid
    * Entity: @sys.any
    * Value: $participantid
    * Prompts: I'm sorry. The participant id you provided does not seem to be␣
↪valid. Could you please check in the online survey and let me know what your␣
↪participant id is?
```

1.4 Modules

Note: see the *Using CART* for more information on how to use these functions.

1.4.1 check_db

1.4.2 log_conversations

1.4.3 rephrase

1.5 Tutorial

This tutorial shows how to create an agent for an experiment as discussed in [REFERENCE TO CCR PAPER].

Agent specifications:

- The agent is embedded in a survey flow, and is presented during the survey
- The agent validates a participant id (to start the conversation) and provides a conversation code (at the end of the conversation)
- The agent automatically assigns users to conditions - humanlike or machine - and interacts different with each participant depending on the condition
- A sentiment analysis tool (Vader) is applied to each utterance by the participant, and the results are stored in the logs table in the database

1.5.1 Installation

The basic steps to set up an agent are explained in *Installation and Setup Guide*. After the basic setup is done, the agent should be up and running - i.e., providing responses - even if irrelevant - to inputs provided by the participant through the web chat (or other channels).

With the agent running, the researcher can then customize the agent for an experiment, as outlined in the following sections.

1.5.2 Configurations in the dialogue management tool

Four intents are created in DialogFlow:

- *Welcome*: starts when the participant greets the agent, asks for the participant id, and - if the participant id is valid - ends with a token (`[PARTICIPANTID_VALID]`) for CART to know that the next intent (*Experiment*) needs to be connected
- *Experiment*: starts when CART has received a token indicating that the participant id is valid (`[PARTICIPANTID_VALID]`) and has sent a start token (`[START_EXPERIMENT]`) to DialogFlow. It ends when all the questions in the experimental setup are asked to the participant, providing a conversation code and telling the participant to continue with the survey.
- *Invalid participant id*: intent that is triggered by CART when the participant id provided by the participant in the *Welcome* intent is not valid.
- *Validate participant id*: fallback intent, which provides instructions for the participant should she want to try to start over (and provide a new participant id).

A copy of the agent, including the full dialogue, is available in the folder [ANONYMIZED LINK TO THE TUTORIAL FOLDER ON GITHUB]

1.5.3 Configurations in CART

The full `config.yaml` file (without the authentication credentials for the API services, which need to be filled out by each researcher) is also available at [ANONYMIZED LINK TO THE TUTORIAL FOLDER ON GITHUB]. The key configurations look as follows:

In the `other` section, the conversationcode_suffix and the conversationcode_base are added to ensure that participants receive a conversation code at the end of the conversation, and that it always starts with a B, and counting from number 1500 (to prevent low numbers)::

```
other:
    conversationcode_suffix: B
    conversationcode_base: 1500
```

The `experimental_design` section indicates that CART will assign participants to conditions using the `random_balanced` option, and that there will be two conditions, one fr machine, and another for the humanlike agent.:

```
experimental_design:
    assignment_manager: CART
    assignment_method: random_balanced
    conditions:
        condition_1:
            condition_name: machine
        condition_2:
            condition_name: humanlike
```

The `rephrases` section has the specific text that varies per condition. The tokens (e.g., AGENTNAME) are included as placeholders in the DialogFlow configurations, so that CART can substitute them depending on the condition the participant is in.

```
rephrases:
    condition_1:
        AGENTNAME: NutriBot
        ACKNOWLEDGEMENT1: OK. The system needs some information about you before it␣
↪can make a recommendation.
```

(continues on next page)

(continued from previous page)

```
        ACKNOWLEDGEMENT2: OK.
        ACKNOWLEDGEMENT3: OK, and
        RECOMMENDATION: OK. Based on the your answers, the recommended breakfast is
        CLOSURESTART: Thank you.
        CLOSUREEND: Conversation ended.
    condition_2:
        AGENTNAME: Ben
        ACKNOWLEDGEMENT1: Great! Let's get started then. I need to know a bit more␣
↪about you before I can make a suggestion.
        ACKNOWLEDGEMENT2: Gotcha!
        ACKNOWLEDGEMENT3: Cool! And, just between the two of us
        RECOMMENDATION: Thanks! Hey... so here's an idea for your breakfast...
        CLOSURESTART: OK! Thanks a million for chatting with me!
        CLOSUREEND: Have a great day!
```

The `connect_intents` section is used to make the connection between the *Welcome* and *Experiment* intents in DialogFlow when the participant id is considered valid by CART.

```
connect_intents:
        PARTICIPANTID_VALID: START_EXPERIMENT
```

The `questionnaire_flow` section is configured to ensure that it is enabled and specify that the conversation with the agent takes place during the survey. As the *Welcome* intent asks for the participant id, this section further specifies that the parameter *participantid* in DialogFlow's responses should be looked for and parsed to detect participant id's. Only id's starting with an A (and ending with a number) are accepted. Two tokens are defined to handle cases when the participant id is valid or invalid.

```
questionnaire_flow:
    enabled: True
    moment: during
    config_during:
        participantid_dialog_field: participantid
        participantid_not_recognized: PARTICIPANTID_INVALID
        participantid_recognized: PARTICIPANTID_VALID
        participantid_valid_suffixes: A
```

Finally, as sentiment analysis will be applied, the `special_functions` section is added with the name of the function, where to store the results in the database (and the type of field). As no override is configured, the `funcion_action` is set to False.

```
special_functions:
    function_1:
        function_name: check_sentiment
        store_output: logs
        store_output_field: sentiment
        store_output_field_type: float
        function_action: False
```

For the sentiment analysis to run, two additional files need to be edited. First, the `requirements.txt` is edited to include `vaderSentiment` as a required Python module to be installed. Second, the `special_functions.py` file (inside the helpers folder) is edited to include the function that processes the user_message:

```
## EXAMPLE - SENTIMENT ANALYSIS
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
analyzer = SentimentIntensityAnalyzer()
def check_sentiment(user_message):
```

(continues on next page)

(continued from previous page)

```
    try:
        sentiment = analyzer.polarity_scores(user_message)
        sentiment = sentiment['compound']
        return sentiment
    except:
        return None
```

1.6 License

CHAPTER 2

Indices and tables

- genindex
- modindex
- search