Summer Internship Report

# Premise

My task was to build a chatbot within Learn using the CodeRunner plugin for Moodle. This bot should answer questions sent by students related to the Team Project module. The bot should be generic and built in such a way that it would be easy to repurpose the chatbot to answer questions on another topic. The bot should answer questions consistently, without varying the wording of any particular answer.

Ideally, this chatbot should also have a ‘session’ it should remember the last few conversations and between you and be able to interact with you based on this. This would increase the usability of the chatbot, as you would be able to clarify answers. As an example of how this would impact the chatbot:

Human: Can you repeat after me?

Bot: Yes, what should I repeat?

Human: Hello, world!

Bot: Hello, world!

There is another desirable requirement for the chatbot to also display animations with the replies, to further increase the quality of its replies.

# Week 1 - Planning

The first week was primarily focussed on planning and researching. The knowledge needed for this project can be broadly separated into three sections:

* Familiarity with the Team Project module
* Knowledge about Moodle, Moodle plugins and CodeRunner and how to integrate new programs into it.
* Knowledge about chatbot technologies

## Team Project Module Planning

This involved taking some preliminary notes on how the module is ran and, on the specifications, available on Learn. This helped me later when putting together a Knowledge base for this module.

## CodeRunner

I needed to learn about how to integrate the chatbot technology into our ‘Learn’ moodle site. I did this through workshops and reading online documentation, as well as reviewing how existing CodeRunner questions worked. This is what I’ve learned about CodeRunner.

CodeRunner is a plugin which adds a new type of quiz question to Moodle. It allows students to be tested on writing code – which the plugin then runs and tests the outcomes. As part of the functionality, the question’s author can customise the code which runs the code in the sandbox. This allows us to take the student’s input (which doesn’t need to be code) and run it through some specific code (for example, a chatbot). There is also the built-in functionality to output a customised html string, which allows us to output a proper chatbot-style messaging screen.

The main element of the CodeRunner question that I’ll be working with is the customisable ‘template’. This is the code that is run each time an answer is submitted. Usually, this runs the code the student submits, then runs tests against it. However, mine works more like a script, that only takes the input to put it through the chatbot. Other languages can be used, however I decided to use python due to its wide selection of libraries and support documentation. It was also a new language to me, and I wanted to learn more about it.

## Chatbot Technologies

Next, I had to determine how to make a chatbot. We decided to use existing technologies as this would decrease the amount of time we would need to spend in development, as well as provide some support documentation. Also, as this was not a research-based internship, we weren’t expected to produce anything from scratch.

Whilst looking at existing technologies, there were some priorities I kept in mind to find an ideal technology:

* Open-source
* Free
* Actively Supported

There was also one more desirable detail – be able to have all necessary files on the server itself, with no external network connections. This wouldn’t have been impossible to accommodate, however from a security standpoint it would be ideal to have everything stored internally.

Here is the list of technologies I generated as a result of this research:

* Python Libraries
  + [Chatterbot](https://chatterbot.readthedocs.io/en/stable/)
    - Free, open-source on a BSD licence
    - Supported and with a community
    - Can provide consistent answers
    - Can be conversational
    - Highly customisable
    - Uses NLTK
  + [Natural Language Toolkit (NLTK)](https://www.nltk.org)
    - Provides interfaces
    - Text-processing libraries
    - Supported
  + [ChatbotAI](https://github.com/ahmadfaizalbh/Chatbot)
    - Not in active development anymore, a bit inactive and not much of a community
* Can be queried from Python and downloaded locally
  + This style of chatbot may be more difficult to manage from a CodeRunner question.
  + [Rasa](https://rasa.com)
    - Two main modules:
      * NLU for understanding user message
      * Core for holding conversations and deciding what to do next
    - Seems highly customisable
  + [Botkit](https://botkit.ai)
    - Based in NodeJs files, can be queried from python, or nodejs.
    - Part of the Microsoft Bot Framework
    - Open-source
    - Natural Language Processing from LUIS.ai (Microsoft’s Language Understanding)
    - Rule-based chatbot – heavily based on human-programmed rules. It can still do quite a lot but this might limit it’s conversational ability.
* External services
  + For this style of chatbot we would need to use webhooks to interact with the external services.
  + [Google Dialogflow](https://dialogflow.com)
    - Handles Natural Language Understanding
  + [Pandorabots/ A.L.I.C.E](https://home.pandorabots.com/home.html)
    - Conversational, AIML agent
    - Paid if you want to deploy it off their website
    - Unclear how to use with other technologies – intended to be used directly with channels such as Facebook Messenger or Slack.
  + [Flo](https://www.learningpool.com/products/flo-demo/)
    - Uses Dialogflow
    - Paid system, doesn’t reveal much at all
    - ‘allows full control of learning via chat commands’
  + [Microsoft Bot Framework](https://github.com/microsoft/botframework)
    - Azure Bot Service is external service
    - Uses Azure Cognitive Services to speak, listen, understand and learn
    - Conversational AI
    - Can be written in python, C#, Java, JavaScript
    - Open Source SDK
  + [IBM Watson](https://www.ibm.com/watson/how-to-build-a-chatbot)
    - Would need to buy the product to see any details
    - Not sure how flexible it is
  + [Amazon Lex](https://aws.amazon.com/lex/)
    - NLU
    - Paid
    - External Service
* Moodle Plugins
  + [Generico plugin + Dialogflow](https://moodle.org/mod/forum/discuss.php?d=379128)
    - Uses the Generico plugin for Moodle to feed data to dialogflow agent
    - At least one documented case of using these two technologies.
* Other
  + [Program O](https://program-o.com)
    - AIML interpreter written in PHP, would be installed on the server.
    - Uses MySQL to store chatbot info, including AIML files
    - May be less supported – last updated 10 months ago

With this initial list generated, I quickly identified the python library Chatterbot as a good candidate – once the library was installed on the server, this would be simple to import into the CodeRunner template. The documentation also made it sound like a lot of the features we were looking for were offered – consistent answers, a way to train with different data (to allow to make the chatbot independent of a knowledge-base), and a way to interact with a built-in database.

# Week 2 and 3 - Preparation of the testing environment

At the start of Week 2 I was allocated a specific PC I could use, and VirtualBox was installed for me. To properly test this chatbot in an environment where I wouldn’t impact anyone else’s use of CodeRunner, as well as have full control and visibility of the entire system, I needed to set up my own mock-up version of Learn, with CodeRunner installed.

## Moodle Webserver

The first step was to set up a webserver with Moodle installed. I had briefly used VirtualBox before, and had a limited knowledge of the linux terminal, so this helped me develop my confidence in these.

For the Moodle webserver, I created a Centos VM. I used Centos at the recommendation of IT services. I didn’t set-up Centos as anything in particular – there is an option to set it up as a basic web-server however I did not use this. I did make sure, in set-up, to add a network adapter. This would allow this VM to communicate with the internet, and with the other VM.

To install Moodle on my Centos VM, I followed [this tutorial](https://www.vultr.com/docs/how-to-install-moodle-3-3-x-on-centos-7). For installing php7.1, I followed [this guide](https://www.cyberciti.biz/faq/how-to-install-php-7-2-on-centos-7-rhel-7/). When setting Moodle itself up, I set it to run on the VM’s localhost.

In order to view and interact with the Moodle, I then installed GNOME Desktop to the VM, using the answer to [this question](https://unix.stackexchange.com/questions/181503/how-to-install-desktop-environments-on-centos-7). I finished setting up Moodle using the ‘Final Configuration’ part of the [official Moodle documentation](https://docs.moodle.org/37/en/Installing_Moodle).

By default in VirtualBox, a VM’s ‘screen’ won’t go full-screen, and I had to install VirtualBox Guest Additions to the VM. This was relatively simple, though I cannot find the webpage I initially followed.

## CodeRunner Plugin and JOBE Server

I then installed the CodeRunner Plugin to my Moodle server using the [official CodeRunner documentation](https://coderunner.org.nz/mod/book/view.php?id=179&chapterid=653). I couldn’t fully configure this without creating my second VM: the JOBE Sandbox server.

I followed the [official JOBE documentation](https://github.com/trampgeek/jobe) for this. Initially I tried installing this on a Centos VM as well, however between different package names and a number of other issues, I was unable to do this, so I followed the official recommendation to run JOBE on an Ubuntu VM, and I encountered no issues.

## Final Setup

To finish setting the mock-up system up, I had to create a NAT Network within VirtualBox and attach both VM’s Network adapters to it, so that they could communicate with each other. I took the JOBE Server’s IP address and entered this into the CodeRunner configuration portal.

During the course of development, I also made some changes to this setup, including adding more memory and CPU cores to the Moodle VM so that it would run faster. I also had to increase the hard drive space of both VMs once or twice as I needed to.

At the end of Week 3, I began testing CodeRunner, sending basic python questions through to check that the plugin was working.

# Week 4 – Installing and Testing Chatterbot

I made the decision to use Chatterbot, as it fit our purposes well and appeared to fill all our needs.

I installed chatterbot using the [official documentation](https://chatterbot.readthedocs.io/en/stable/setup.html), using pip. When installing this, I encountered some memory errors, as to install chatterbot, pip installs its wide array of pre-requisite libraries, and this takes some time and memory. To fix this I gave the VM a larger memory, one more CPU core and slightly increased its hard drive.

I then began testing that I could use chatterbot through CodeRunner – initially I couldn’t, as I’d install chatterbot to my personal user’s python libraries, rather than installing them globally. Installing them again using ‘sudo pip’ this time did fix things.

The questions were successfully running, however there was an irregular output – saying that nltk\_data was being installed. This will happen the first time you run the chatbot, but should fix itself the second time around as the data has now been installed. However, there were still outputs checking whether two files were downloaded, then clarifying that they were up-to-date. This is due to an issue within the chatterbot source-code, and is documented [here](https://github.com/gunthercox/ChatterBot/issues/1618). I didn’t want to change the source-code, and so using the information in [this reply](https://github.com/gunthercox/ChatterBot/issues/1618#issuecomment-501770179), I moved the two files to where the source-code was checking for them. This stopped the irregular output. Using the default configuration for chatterbot, I trained a chatbot with limited data and tested it.

I also encountered an error when using nltk as by default it is installed in a user-specific directory, however when JOBE runs the question, it deletes everything in the user’s directory after each use, which wasn’t ideal. However, I used the JOBE server terminal to install it into a shared directory.

The chatbot did work and outputted the right things when I went through the pre-defined phrases that I’d trained it with. However, that said, there were some issues I had found at this stage.

At this stage I wasn’t familiar with how the chatbot’s algorithm worked, but within the pre-defined phrases was ‘Hi there’. When I said ‘Hi’, it didn’t judge these as being similar. However, at this stage I thought this may be down to using very short phrases, and that when I used longer questions and sentences it would pick it up better.

Also, there was little possibility of using storage, as the chatbot is created each time an answer is submitted. I didn’t really explore the uses of the in-built ‘storage adapter’, which allows you to interact with the chatbot’s internal database, but I did gather that this is primarily used for storing its database of pre-defined phrases, as well as for chatterbot’s machine learning capabilities. These capabilities are used more for expanding the chatbot’s knowledge base, rather than for tracking previous conversations, as we would like for clarification etc.

In my weekly meeting with Firat, we discussed ways forward. We agreed that learning more about chatterbot would be a good move forward, and would inform us to whether we are on the right path. Whilst doing this I would also keep in mind the priorities for what we want our chatbot to accomplish. We agreed, at this point, that these were:

* The chatbot should be generic
* The chatbot should have an easy way to submit a knowledge base.
* We would like to investigate animated avatars
* We would like consistent answers.

There was also an idea that we could take advantage of the machine learning capabilities, by having the bot learn new responses to questions from emails between lecturer and student. This would greatly reduce the work involved to write a good knowledge base. This would take quite a bit of research and development, and would not work with the current implementation, as the machine learning only works if the chatbot is persistent and stays online, whereas currently a new chatbot is created with each answer, then deleted. This is partly due to my implementation but also is a major part of how CodeRunner works.

Around this time, I also started seeing a MemoryError when my chatbot was running. Initially, I tried increasing the Memory of my VM but that did not help. After reading through the CodeRunner forums, I realised that each JOBE process had a maximum MemoryLimit of 200MB, which can be changed in the CodeRunner Question. I increased this to 1GB and everything worked fine.

# Week 5 – Improving user interface and trying to implement a session

## Improving User Interface

There were two things I focussed on here – reducing the number of rows in the input text-box from 18 to 5. This made the input look a lot more natural, whilst still allowing space.

I knew it was possible to output html strings from a CodeRunner question, and in my fifth week I looked into implementing this. I mainly used existing CodeRunner questions that implement this as a guide, as this isn’t widely documented. To output html, at the bottom of the template, we put:

print(json.dumps({"fraction": 1.0, "epiloguehtml": html}))

This outputs every answer as 100% correct (as shown by ‘fraction’), and ‘epiloguehtml’ represents what is outputted, in a html string. Within the CodeRunner question, Grading should also be set to ‘Template grader’.

## Sessions

My attempt to create a ‘session’ for this chatbot solely lies in storing previous messages and displaying them back, as is standard for messaging platforms, so you can see your conversation’s history.

To do this, I would need to store information in a persistent data source. Most of the folders that JOBE processes have permission to access are ‘cleaned’ at the end of their run, meaning any files that are made are deleted. So I created a separate folder and created a sqlite3 database using the linux terminal. I changed the permissions on this to 777, effectively making it globally accessible to anyone. This enables the JOBE process to access this database, deposit data and this data is not lost.

There is only one table in my database design, and it is as follows:

* TblMessage
  + MessageID INTEGER PRIMARY KEY
  + StudentID INTEGER
  + MessageDateTime DATETIME
  + StudentInput TEXT
  + ChatbotOutput TEXT

This was changed slightly towards the end of the placement, adding the chatbot identifier BotID to uniquely identify each question as different.

I also changed the CodeRunner template to run the chatterbot code within a subprocess. This allows better debugging, and allows the question to still run if anything goes wrong.

# Week 6 and 7 – Research Chatbots Algorithms and Research Chatterbot Library

Week 6 was a partial week for me as I went on holiday partway through it. I was on holiday for the entirety of week 7.

I started researching different kinds of chatbots in order to widen my knowledge of how they work.

Broadly-speaking, there are two variants of chatbot – rule-based and self-learning. Rule-based chatbots answer questions using a set of rules that have been programmed into it. These can be vaguely compared to a large set of nested if-statements. These can handle simple queries but start to struggle with more complex ones. Self-learning bots use some machine-learning-based approaches and can be broken down further.

Self-learning bots can be broken into a further two types: Retrieval-based or Generative. Retrieval-based chatbots use some heuristic to select a response from a library of pre-defined responses. These chatbots can use the context of the conversation as well as the message to select the best response. Generative chatbots generate the answers and doesn’t always reply with an answer from a set of defined answers. [[1]](#footnote-1)

Most chatbots have some level of Natural Language Processing (NLP)/Natural Language Understanding (NLU). The first and one of the more important elements of this is the pre-processing of text. This is done as an attempt to standardise the message, so that words like ‘learning’, ‘learned’ and ‘learn’ are all treated the same. There are quite a few syntax techniques, which all divide a larger whole into smaller, individual units for easy analysis. [[2]](#footnote-2)

Word segmentation is used to divide a large piece of continuous text into distinct units (also called tokenisation). These individual words are then processed using techniques like lemmatisation or stemming. Lemmatisation reduces the various inflected forms of a word into a single form for easy analysis ((best, better, good) -> good). Stemming involves cutting inflected forms to their root form ((stems, stemming, stemtisation) -> stem).

The algorithm that Chatterbot uses is called the Naïve Bayes Algorithm. ‘Naïve’ here means that every word is treated independently of each other – that there is no connection between any word. This algorithm doesn’t look at meaning or semantics at all. Chatterbot first uses its pre-requisite library NLTK to take our input through tokenisation and stemming.[[3]](#footnote-3)

The algorithm then compares your processed input to its knowledge base. Its knowledge base is split up into categories. In each category is a list of (statement, response) pairs. First the algorithm scored how similar your input is to each categories’ pre-defined statements – it does this while weighting that some words occur more often than others (a, the, of, etc.). Then it will take the category with highest score and score each statement on its similarity. It will then output the response to the most similar statement.

As Chatterbot relies on its knowledge base for analysing the input and selecting the output, the quality of the chatbot will be defined by how good its knowledge base is.

# Week 8 – Figure out the best way to train the chatbot

This week I started trying to find the best way to train the chatbot. The Chatterbot library has multiple ways to train the bot – with the two main ones being ListTrainer and CorpusTrainer.

ListTrainer is trained by submitting an array of statements. The chatbot will compare your input to all of these, then output the statement after the most similar one. This can lead to outputs that were not intended to be responses. CorpusTrainer is trained using a .yml file. Chatterbot supplies some basic ones, which I have ended up using for basic greetings and basic conversation, however you can give it a path to any file. Within a yml file, you separate the knowledge base into categories and distinguish between statements and responses. These two elements make CorpusTrainer a superior option.

We also looked at the chatbot’s confidence. Chatterbot outputs a confidence value with its output, which we can manipulate and process.

During week 8, we did start taking note of some of chatterbot’s features which are not very ideal to the requirements. The input needs to be very similar (almost word-for-word) to the pre-defined statement or the confidence will not be very high. If the input is almost completely foreign to the chatbot, it will most likely pick a response seemingly randomly. If there are two pre-defined statements that are very similarly worded, the reliability isn’t high that it will choose the right one.

We want to avoid incorrect answers – which can still be done with a good knowledge base. If the pre-defined answers are all context-independent – ie. specify the context within the answer – we won’t give incorrect information, we might just give answers for other questions.

If the knowledge base is also made large enough, with different ways of saying each statement incorporated, it can help the chatbot get more reliable. The importance of a large and well-written knowledge base does make it a relatively long process to create a new chatbot. This is, admittedly, a large flaw with this chatbot. It is generic (ie. Knowledge-base independent), but the time and effort required to make a good knowledge base is a fairly high cost.

# Week 9 – Polish and write the Knowledge Base

I revisited the CodeRunner output and spent some time making this look better, adding more detail such as an icon displaying who was speaking, the time and date of each message and generally improving the structure of how this was laid out. By default, the only messages visible will be the response from the chatbot and most recent message. I have added a button which allows you to view the last few messages of your conversation history. This uses basic javascript and html.

Until now, I have been using the CodeRunner template parameters to pass the knowledge base to the code, however this would work better if it was a standalone file. The knowledge base has been passed to the code in a JSON format, which my code then reformats as a .yml file and writes it down, which the chatbot then uses. I rewrote this section to read from a .json file instead and do the same instead. To read from a file within a CodeRunner template, I reviewed similar example questions where this is done. If there are multiple files, which there shouldn’t be seen as everything is deleted at the end of each process (though I did sometimes find multiple files), I have set an explicit check for a specific filename, which should solve any issues.

I chose to keep the JSON format as this is more widely-known and will be more familiar to users than a bespoke file format. Additional time will be spent formatting the knowledge base as a JSON.

I also began writing the knowledge base for the Team Projects module, as an example knowledge base. This involved going through the specification I have access to, which is the specification from two years ago. I went through, generated potential questions and answers, before spending some time formatting it correctly. When I went to use this knowledge base, there were numerous errors.

These errors with my knowledge base originated with various punctuation which, when the knowledge base was used to train the chatbot, was confusing it. These symbols were found to cause errors and need to be kept out of the knowledge base:

* Dash –
* Colon :
* Boolean operators (it did not like NOT when all in capitals)

# Week 10 – Finishing touches

This week involved continuing to polish the chatbot. I have changed the chatbot code to reflect the confidence. If the confidence is below a certain value, it will now not give that response, instead replying with a stock phrase, such as ‘I don’t understand’. This will hopefully avoid any random answers from being given.

I have also added more elements to the template parameters, which allow the question author to change some settings of the chatbot. This allows the author to have more control over the chatbot’s behaviour. There are five aspects they can specify:

* Name – the ‘name’ that is displayed as the chatbot’s name. It is also used as a unique identifier when combined with the module code.
* Module Code – the unique identifier for the module. When combined with the name it created a unique identifier for the question.
* Confidence Limit - This value specifies the confidence value below which it will output the 'poor confidence message instead of a potentially incorrect answer.
* Poor Confidence Message - The message outputted if the chatbot cannot answer with confidence.
* Previous Conversations - The number of previous back-and-forths that are able to be seen by the user.

In order to prepare the database for more than one question and more than one bot, I initially added the ‘BotID’ column to the table, however as this would lead to having all the data in one table, we decided to change this to instead have each chatbot be connected to its own table. We did this by using the ‘name’ and ‘modulecode’ inside the CodeRunner template parameters to create a unique table name.

Towards the end of the final week, we ported this across to Learn from my mock-up, and ironed out the niggles with how Chatterbot interacts with JOBE, which is documented above.

# Conclusion

## The Product

The result was a successfully working template within CodeRunner. This template runs a chatbot against your answer and gives you an output successfully. This is a template that anyone can use, and I have written guidelines within Learn for people to use this.

This template shows a proof of concept that a chatbot can be made within CodeRunner, which has not been shown before. It also shows that a chatbot can be run from inside Learn.

This can be used as a basis for future work. This can be used to improve upon this chatbot to create a better general chatbot using CodeRunner, or to create a better FAQ bot.

## Improvements

There were several flaws with the outcome, and better decisions could have avoided them.

The choice of chatterbot was not a bad one, however this chatbot is not ideal for us. It is very difficult to make this chatbot persistent, which we could have used to utilise its machine learning capabilities and learn whilst lecturers answer emails. It is difficult to improve this chatbot’s accuracy, as it’s only as good as its knowledge base, which puts a large responsibility on the question author.

Each time an answer is submitted, the question will:

* read from a JSON file,
* save that back down in the correct YML file,
* create a chatbot,
* train the chatbot,
* get the response from the chatbot,
* check the response’s confidence level, determining the chatbot’s output string
* create a new database table if one doesn’t exist
* insert the message into the database
* retrieve previous messages
* format these messages into a html table
* output this html

Frankly, this is not efficient. Each answer takes between 5-15 seconds to run, which is not acceptable. With larger knowledge bases this would take even longer. However, with the approach I took, and the technology I chose, it was difficult to avoid. If this chatbot were persistent, or if the processing took place on a separate server, for example if I used an external service like Dialogflow, some of this would not be necessary.

The Knowledge base is a potentially large issue. Inherently, it is a very important part of a chatbot, especially this one, as chatterbot relies upon a large and well-stocked library of pre-defined phrases. I feel that while JSON is a widely-known format, it increases the workload of the question author, as formatting questions and answers in JSON can be very tedious. Perhaps this could have been improved with an additional tool that formats a simple list of phrases into the correct JSON string. Alternatively, I could have written the template to accept some other file format that would be easier to write for.

Utilising Chatterbot’s machine learning algorithm isn’t impossible, but there are several major issues with this:

* Currently, the chatbot isn’t persistent. Any data fed to it is lost again once the process finishes.
* Even if you use the database of previous messages to train the chatbot each time, this would significantly increase the runtime, which the runtime is already significant.
* There’s the added development hurdle of intercepting/handling emails between student and lecturer and turning that into something the knowledge base could use.
  + Would this be entered anew manually each time an email is sent?
  + Would this send the email to the lecturer, then send the reply to the student?
  + Would this extract the meaningful question and answer from the email?

Whilst there are other chatbot technologies, and whilst chatterbot is not ideal for this situation, I cannot say for certain whether anything else would have been better. It was only once I began testing chatterbot with larger knowledge bases that I began to see its shortcomings.

We didn’t manage to explore the possibility of animation with this chatbot, however I believe there are animations that can be represented using only html code. If this is the case, then this will be possible, though making sure the animations are accurate to what’s being said may be a separate issue.

Other chatbot technologies can be used without too much difficulty. If it involves running some python code, the change would be trivial, however if it were a technology like an external service, the difference in how it would work could be a larger task.

Using an external service could improve better performance, however I am unfamiliar with how knowledge bases are loaded into them – if, as I suspect, you must add the knowledge base to them manually on their own site, this could limit the chatbot’s ability to be generic.

Using a rule-based system that you query such as Botkit could be an avenue worth investigating as this would be local to the JOBE server, though this might introduce new issues.

1. https://medium.com/analytics-vidhya/building-a-simple-chatbot-in-python-using-nltk-7c8c8215ac6e [↑](#footnote-ref-1)
2. https://becominghuman.ai/a-simple-introduction-to-natural-language-processing-ea66a1747b32 [↑](#footnote-ref-2)
3. https://chatbotslife.com/text-classification-using-algorithms-e4d50dcba45 [↑](#footnote-ref-3)