11-411/11-611 Project Prototypes

NLP Teaching Staff

Due: March 30, 2023

1 Introduction

For the dry run system, each team needs to complete the following tasks:

- Develop a question generation program
- Develop a question answering program
- Build a docker container which includes your programs and all the dependencies
- Upload your docker image and submit your tagged image name to Gradescope

The purpose of dry run is to make sure you can correctly build your system as instructed. The quality of generated questions and answers will not be graded during the dry run phase.

The handout includes:

- **test.sh** which tests your system. Please test your container before submission
- **docker** folder, which includes sample ask/answer programs and a sample Dockerfile for building a docker image of the QA system. You can easily build your own based on this example.
- **data** folder, which contains the training data and is accessible by your container.

2 Learning Objectives

- Producing a docker container for an application (your QA and QG systems)
- Providing an opportunity for feedback

3 Task 1: The Programs

You must deliver two programs called **ask** and **answer** with the interfaces and properties listed below:

- The programs must have **exactly** the names listed above.
- The programs must be **executable**. If you are using python, please read §3.3.
- The program must take **exactly** the command line arguments detailed in §3.1 and §3.2

Both **ask** and **answer** should follow:

- expect input containing **non-ASCII characters in UTF-8 encoding**. Failure to address this will result in your team receiving no credit for the competitive portion of the project grade.
- **Debugging information should not be directed to STDOUT**. This will throw off the alignment of your outputs and will result in a dramatically reduced score.

3.1 Question generation

The ask program must have the following command-line interface:

```
./ask article.txt nquestions
```

Where:

- *article.txt* is a path to an arbitrary plain text file (the document)
- *nquestions* is an integer (the number of questions to be generated).

The program should output to STDOUT a sequence of questions with each question terminated by a newline character. **Each line should not contain any text other than the question.** If your program cannot generate the requested number of questions, it should not try to buffer the output with empty lines.

3.2 Question answering

The **answer** program must have the following command-line interface:

```
./answer article.txt questions.txt
```

Where:

- *article.txt* is a path to an arbitrary plain text file (the document)
- *questions.txt* is a path to an arbitrary file of questions (one question per

line with no extraneous material).

The output (to STDOUT) should contain a sequence of answers with each answer terminated by a newline character. If your system cannot generate an answer for a given question, it should output a blank line or a default answer. **Each line should not contain any text other than the hypothesized answer.** Otherwise, your outputs will be penalized for (a lack of) conciseness.

3.3 Making your Python File Executable on Linux

Step 1: add an appropriate shebang line to beginning of the file. To run a script with the default system Python interpreter, use the following:

```
#!/usr/bin/env python3
```

Step 2: change the permissions of the file so that it is executable by all users. At the command line, issue the following command ($ is the command prompt):

```
$ chmod a+x thescript
```

Where thescript is the path to the file whose permissions you want to change. That is it. You should now be able to run the script by typing the following command at the prompt:

```
$ ./thescript
```

3.4 Output Formatting

We will evaluate only your output to stdout, and ignore anything written to stderr. Here are some formatting rules:

- Every question and answer should consume exactly one line.
- Your output should not be numbered or prefixed with things like "1.", "2.", "Question:", or "Answer:".
- You should not have extra new lines between questions/answers.
- There should not be debugging statements, warnings or other extraneous output written to stdout. Standard Python warnings and progress bars go to stderr by default, but beware of nonstandard output from libraries you are using. You can test this yourself by ignoring stderr output from test.sh (a quick Google search will reveal how if you don't know). The goal of these rules is to make it easy for our scripts to automatically parse your output, and to avoid passing prefixed or badly formatted questions to the answering systems of other projects.

4 Task 2: Dockerizing the Programs

A detailed example is included in handout, please checking the files in **docker** folder while reading the following instructions.

4.1 Put your system into docker container

What is docker container and why would we bother using it? A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably in computing environments other than the one in which it was developed. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime. In the case of Docker containers, images become containers when they are run on Docker Engine. Available for Mac[1], Linux and Windows-based containerized software will always run the same, regardless of the host infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Go to (`https://docs.docker.com`) for more information,

Step 1: Install docker on your machine

Please refer (`https://docs.docker.com/install/`)

Step 2: Define a container with **Dockerfile**

Dockerfile defines what goes on in the environment inside your container (it provides instructions for creating the image). Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you need to map ports to the outside world, and be specific about what files you want to "copy in" to that environment. However, after doing that, you can expect that the build of your app defined in this Dockerfile behaves exactly the same wherever it runs. This is important to us because we will run all of your code on Linux EC2 instances regardless of whether you developed your code in MacOS, Linux, or Windows.

Here is the example Dockerfile:

```
# Ubuntu Linux as the base image. You can use any version of Ubuntu here
FROM ubuntu:22.04

# Set UTF-8 encoding
ENV LANG C.UTF-8
ENV LC_ALL C.UTF-8

# Install Python
RUN apt-get -y update && \
    apt-get -y upgrade
# The following line ensures that the subsequent install doesn't expect user input
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get -y install python3-pip python3-dev

# Install spaCy
RUN pip3 install --upgrade pip
RUN pip3 install spacy
RUN python3 -m spacy download en_core_web_lg

# Add the files into container, under QA folder, modify this based on your need
RUN mkdir /QA
ADD ask /QA
ADD answer /QA

# Change the permissions of programs
CMD ["chmod 777 /QA/*"]

# Set working dir as /QA
```

[1]For Macs with Apple Silicon, Docker presents special problems. See the FAQ.

```
WORKDIR /QA
ENTRYPOINT ["/bin/bash", "-c"]
```

The example dockerfile does the following:

- use Ubuntu Linux as its base image (in most case, you don't need to change that)
- Set UTF-8 as text encoding (you should not change this)
- Install all the program and dependencies needed for our program, in this example, Python 3 is installed, since the **ask** and **answer** in this example are written for Python 3. The package **spacy** is installed, and pre-trianed model en_core_web_lg is downloaded beacause they are dependencies of programs. (you need to change this part based on the dependencies of your programs)
- Copy the files into container. You need to copy all your programs into container. In this example, they are ask and answer. (You may need to change that if you have other program files)
- Change the permissions of programs. (you may add other command if needed)

You must install all of your dependencies, including command line utilities and Python libraries, in advance via your Dockerfile.

Windows users will also probably want to install the `dos2unix` program and run it on their python files (via the Dockerfile) in order to change their Windows line endings into Unix line endings (which will be expected inside the docker container).

Step 3: build the container image

Now, everything is ready, just run command:

```
docker build --tag=${NAME} docker/
```

${NAME} can be whatever name you want for example

```
docker build --tag=spacemonkey docker/
```

You can also do the same thing using

```
docker build -t ${NAME} docker/
```

Note: The first time you run this, it will take a while! See the FAQs for more information.
Note: "docker/" at the end of the command refers to the directory containing the Dockerfile.

Use command:

```
docker image ls
```

to check if you successfully build the docker image.

4.2 Test your image

Run commands:

```
chmod u+x test.sh # first time only
./test.sh ${image_name}
```

where `${image_name}` is the container image you would to test.
If everything is right, the output should be:

```
***************************
Q1
Q2
Q3
***************************
A1
A2
A3
***************************
```

Q1, **Q2** and **Q3** should be three questions generated by your program `ask`. **A1, A2** and **A3** should be three answers generated by your program `answer`. See the formatting section below to make sure your output is formatted correctly.

4.3 Test your image further

You can run the following command at any point during the process to check whether your program will run in Docker:

```
    docker run -it ${NAME} /bin/bash
```

${NAME} is the name of the image you built; for example

```
    docker run -it spacemonkey /bin/bash
```

This will create a shell where you can run your programs. You can also test installations here to see if they will work before adding them to the Dockerfile.

4.4 Publish your image

(For more information, see `https://docs.docker.com/get-started/part2/`) At this point, you have finished building your container image, well done! Just a few more steps to go.
Step 4: Log in with your Docker ID
If you don't have a Docker account, sign up for one. Make note of your username.
Log in to the Docker public registry on your local machine.

```
    $ docker login
```

Step 5: Tag the image

Now, put it all together to tag the image. Run `docker tag image` with your username, repository, and tag names so that the image uploads to your desired destination. The syntax of the command is:

```
docker tag <image> <username>/<repository>:<tag>
```

For example:

```
docker tag spacemonkey tyler/nlpdryrun:version2
```

Use command:

```
docker image ls
```

to check if your newly tagged image. Don't forget to test new image with

```
test.sh username/repository:tag
```

Step 6: Publish the image

Upload your tagged image to the repository:

```
docker push username/repository:tag
```

Once complete, the results of this upload are publicly available. If you log in to Docker Hub, you see the new image there, with its pull command.

5 Deliverables

Submit a single text file to Canvas named `docker.txt`. `docker.txt` should have two lines: the first line is your published image name `username/repository:tag`, and the second line is your team name and all members names. Only one team member needs to submit per group but please name all team members in the file.

6 Evaluation and Grading

6.1 Question Answering

We will pass 10 questions to your `./answer` program (5 wh-questions and 5 polar questions) based on one document.

program returns no error messages	$+3$ points
program returns well-formed output for all questions	$+2$ points
for each correct answer (ceiling of 5 points)	$+1$ point

6.2 Question Generation

We will pass one document to your `./answer` program and ask it generate 10 questions.

program returns no error messages	$+3$ points
program returns well-formed output (10 lines consisting of text)	$+2$ points
for each question that is fluent and answerable (ceiling of 5 points)	$+1$ point

7 Frequently Asked Questions

- Q: I am getting "Error: No such container: ./ask". A: You are missing an argument to the test script. You should run it as `./test.sh <image_name_or_id>`.
- Q: I am getting "docker: Cannot connect to the Docker daemon...". A: You need to launch your Docker application before running the test script or any docker commands.
- Q: What's the data folder for? A: The data folder contains the training/test articles. You do not need to add into the container; instead, the test.sh script launches the container by creating a volume, which is a shared directory between the container and the host (your laptop). It is there so that your ask/answer programs can open the article when given the path. If you need access to the data during setup (i.e. one or more commands in the Dockerfile needs to access the training files), then you should create a copy of the data directory inside the docker folder and then add it to the container. **Note: some students have reported problems with being able to access the data folder from within the container. TAs have not been able to verify this, but one solution might be to put the shared folder in quotes inside test.sh. Additionally, be sure to run test.sh from the same directory it is located in.**
- Q: Can I modify in the Dockerfile? A: You can modify any part of the given sample Dockerfile however you wish, or just discard it completely and create your own. You will at the very least need to make some modifications to add all your files and install your dependencies. Remember that you are not submitting the Dockerfile, but an image that was built from it. If you can run your image locally on an Intel machine, then we can run it too. The only requirement is that there are two executable files, "/QA/ask" and "/QA/answer".
- Q: Can I modify `test.sh`? A: The test.sh script is not included in the Docker image that you submit, it is just provided as means to automatically launch and test your container. We will use the same exact script to test your submitted image for the dry run, so be sure to test with the unmodified version. Beyond the dry run, you are encouraged to modify this script to test on different or more articles. The input questions are listed in "test_questions.txt".
- Q: What is the difference between a Dockerfile, Docker container, and Docker image? A: A Dockerfile contains instructions for building your image. An image is

similar to a VM image, anyone with access to the image can run your program without figuring out how to set it up. A container is an instance of your image. To run your program, we launch a container from your image, and then terminate it.

- Q: How can I build Docker images faster? A: The first time your build an image from your Dockerfile, it can take a while because it downloads the base operating system and installs your applications. From then on, it should be much faster if your structure your Dockerfile appropriately. Docker caches the image state after every line of the Dockerfile, so when you rebuild the image, it starts only from the modified line. For this reason, it is good to avoid combining commands into a single line, and to put longer running commands earlier in the file (this way, you don't have to rerun your yum commands every time your add a new file, for example). For more information, look up how caching and layering works with Docker.

- Q: I want to mess around in my Docker container. How do I keep it running and `ssh` into it on a terminal? A: Open `test.sh`, and copy the command that launches the container ("docker run..."). Then run "docker exec -it /bin/bash". This is good way to test installation procedures before adding them to the Dockerfile. Note that all your changes to the container will be lost once the container is terminated, so if you want to keep anything, you will need to make the change in the Dockerfile.

- Q: One of my dependencies has a complex installation or startup procedure. How do I do this in the Dockerfile? A: A good strategy is to install your unix tools in the Dockerfile (e.g. git, wget, etc.), put your installation procedure in a shell script, add the script to the container, and execute it.

- Q: Should I start servers or install applications in the `ask` or `answer` scripts? A: No. This should be done in the Dockerfile so that when your container starts, those setup commands have already been run. That's why we ask you to use Docker—all of your custom setup code has already been run and the resulting state saved in an image that we can easily launch. This way your ask and answer scripts will also run quickly.

- Q: My container randomly exists without any warning. A: This is commonly caused by your container running out of memory. Docker allocates a specific amount of memory for all running containers, and when that threshold is exceeded, it starts randomly killing containers without warning. Make sure you have don't have any containers running from previous experiments, and also adjust your memory threshold in the Docker application settings.

- Q: **Can I build my Docker image on a Mac with Apple Silicon (M1 or M2)?** A: Unless you are, as they said in the 1990s, a 1337 h4x0r, it is challenging to build an Intel Docker image on an M1/M2 Mac. If you don't have access to a machine with an Intel processor, we suggest borrowing one from David or using a cloud computing service like AWS.

- Q: Hardware specifications of the test environment? A: We will run your container on an m5.xlarge EC2 instance (4 vCPUs and 16GB of RAM). Try to keep your image size 10GB or below. **If you need a GPU at runtime, you must message us with your NVidia driver version requirement. We will run your instance on a p2.xlarge instance. TAs are generally not familiar with running Docker with GPU, so you will need to investigate this yourself and tell us if you need us to do anything other than install the latest NVidia driver in the test environment.**
- Note: Check the online Docker tutorials/cheatsheats to see how to manage your running/stopped containers, rename/delete images, etc. It can be useful to delete old images if you start running out of disk space.
- Note: Publishing your Docker image to the repository can be a slow process (several hours), especially the first time, so be sure to start that process well before the deadline.