Reproducible Research in Document Analysis and Recognition

51

Jorge Ramón Fonseca Cacho and Kazem Taghva

Abstract

With reproducible research becoming a de facto standard in computational sciences, many approaches have been explored to enable researchers in other disciplines to adopt this standard. In this paper, we explore the importance of reproducible research in the field of document analysis and recognition and in the Computer Science field as a whole. First, we report on the difficulties that one can face in trying to reproduce research in the current publication standards. These difficulties for a large percentage of research may include missing raw or original data, a lack of tidied up version of the data, no source code available, or lacking the software to run the experiment. Furthermore, even when we have all these tools available, we found it was not a trivial task to replicate the research due to lack of documentation and deprecated dependencies. In this paper, we offer a solution to these reproducible research issues by utilizing container technologies such as Docker. As an example, we revisit the installation and execution of OCRSpell which we reported on and implemented in 1994. While the code for OCRSpell is freely available on github, we continuously get emails from individuals who have difficulties compiling and using it in modern hardware platforms. We walk through the development of an OCRSpell Docker container for creating an image, uploading such an image, and enabling others to easily run this program by simply downloading the image and running the container.

Keywords

Reproducible research · Containers · Docker · OCRSpell · Document analysis and recognition

J. Ramón Fonseca Cacho (✉) · K. Taghva
Department of Computer Science, University of Nevada, Las Vegas, NV, USA
e-mail: Jorge.FonsecaCacho@unlv.edu; kazem.taghva@unlv.edu

51.1 Introduction

A key to advancing our field is to build and expand on previous work, namely cumulative science. The only way we can achieve this is if we understand the foundations and can replicate it. In this lies the importance of Reproducible Research, which means the ability to take a paper—code in many of our cases—and be able to run the experiment addressed in that paper so we can learn and expand on the paper's research or even refute it [1]. Knowing and having access to the raw, intermediate, and processed data is another important aspect as it is key in understanding how every result was produced [2].

Reproducible research should be more than the final data and product. The necessity to implement a gold standard for publications and conferences is ever increasing. A great deal can be learned from the steps it took to come up with an algorithm or solve a problem. Version control and the concept of *Git* is a great solution to increase transparency and see the evolution of code to its final form [3]. Reproducible research is also about how *TIDY* data is produced [4]. Recently, we wanted to revisit our research on post-processing of Optical Character Recognition (OCR) data. As a part of this work, we wanted to access some data which is referenced by some of the authors in the International Conference on Document Analysis and Recognition (ICDAR) proceedings. We were unsuccessful in obtaining the data since the download page produced errors. We contacted the university that had produced the OCR'd documents and ground truth, and never received a response. This is a common trend that is thoroughly documented by Christian Collberg et al. [5]. Collberg's group was in the pursuit of a project's source code that should have legally been available due to having been partially funded with federal grant money. After evasion from the university to release the code, Collberg threatened legal action which was met by lawyers' refusal and eventually avoiding his request by charging extreme retrieval fees. The exchange is well documented along with other examples of researchers

S. Latifi (ed.), *Information Technology – New Generations*, Advances in Intelligent Systems and Computing 738,
https://doi.org/10.1007/978-3-319-77028-4_51

giving reasons for not sharing the code such as it no longer existing or not being ready to publish it. Collberg ultimately found that only a quarter of the articles were reproducible [5].

People are usually criticized for unpolished or buggy code, and writing pristine code is usually time consuming and unnecessary. Authors may not want to be subject to that critique [6]. This creates an environment that encourages code that is not acceptable to not be released. There is no easy solution to this. Scientist must have confidence in that the *results* are what matters and not how the code is written as long as its reproducible. The code can always be cleaned up after the fact. Similarly, with the data used in experiments, many times researchers will not want to share it in order to avoid exposing potential bias in their research for False positive findings [7]. There is also the monetary potential that a project's code could bring [6]. In this case the researcher has hopes his new algorithm, or data set, could be monetized down the line giving no incentive to offer it for free initially.

Another difficulty in reproducible research lies in not all researchers having access to the same level of resources such as computing power required for certain resource-intensive projects. This makes it difficult to review or reproduce such research. There is no definite solution to that difficulty.

Roger D. Peng is known for his advocacy for Reproducible Research in several publications; as Peng mentions, requiring a reproducibility test when peer reviewing has helped computational sciences to only publish quality research. Peng clearly shows reproducible data is far more cited and of use to the scientific community [1]. Many others agree with the concept of creating a standard in the field of computer science that matches that of other sciences for reproducible research to the point of requiring mandatory data archiving policies [8]. Even the US Congress has shown interest and has had hearings on transparency [9], and while reproducible research helps, reproducibility of both data and code is insufficient to avoid problematic research in the first place [10]. Nevertheless, the benefits in transparency in the way data is handled will avoid potential risk of researchers being accused of skewing data to their advantage [11].

51.2 Docker

Docker is an open source project used to create images that run as Linux Containers with a virtualization of the OS level [12]. See *docs.docker.com* for more information.

Containers enable us to freeze the current state of a virtual operating system so that we can reload it and refer to it at any time. This enables us to store data and/or code that works at the time and be able to save with it any dependencies or requirements without worrying about deprecation or code-rot. They help to reduce the complexity of reproducing experiments [13]. The Dockerfile, which is what builds the containers as a series of steps, enables one to see the steps used to create that container. This along with version control help to understand the history and steps used in research.

Docker does have limitations when it comes to rendering graphics, as it uses the host machine for that. Solutions to make this issue not be platform specific come in the form of GUIdock [14]. It is in this environment surrounding the Docker platform where the open source availability shines as most limitations have been removed with additional software ensuring it as a consistent solution to reproducible research.

Because containers are processes that have a lot of overhead, questions may arise as to the impact in terms of performance for larger projects with big data such as in bio-informatics; however, benchmarks prove that their fast start-up time along with negligible impact on execution performance has no negative effects on using them [15].

51.3 OCRSpell

OCRSpell is a spelling correction system designed for OCR generated text. Although the software was released in 1994, it continues to be relevant based on number of downloads and citations such as [16]. Making OCRSpell easily available for our own projects as well as others made it a good candidate for our work on reproducible research.

OCRSpell code is written as a mix of shell script commands in Linux and C code. It works by generating candidate words through different techniques including the use of ispell. For more information see [17] and [18]. At the time of writing, the source code is available at https://github.com/zw/ocrspell, and has a small *readme* file describing the steps to compile the source code. Several issues arise if one tries to build and run OCRspell using only these instructions:

1. Dependencies: to build, the readme says to `run autoconf` which means running a program of which no more information or version is given. Furthermore, no support is given for running that dependency.
2. Modifications by Hand: readme says:
   ```
   ./configure -prefix=/local/ocrspell
     -<version>
   ```
   and:
   ```
   in configure at line 685 change
   prefix=NONE to prefix=/home/graham
   /Downloads/ocrspell-1.0/src
   ```
   All these changes are manual changes that must be done in order to build and run the file which can be prone to mistakes due to lack of understanding what is happening.
3. Libraries and Guess Work: in order to build the program, several non-standard libraries must be included, programs like ispell must be installed. The only way to know what is required is to try and build, see the error, download that

program and try again. While in this case this works, it is a dangerous practice due to unknown changes in newer versions of dependencies.

4. Lack of Documentation: Aside from having documentation once the program is running, as is the case here, the readme is not very descriptive as to what is going on. Once the program runs with

```
./ocrspell -a -f ./ocr.freq <
quick_demo.txt
```

we have no confirmation that our output will be equal to what it was when the experiment was run years ago aside from one test file.

51.3.1 Docker as a Solution

Docker can resolve all of the above problems by creating an environment that is ready to go and is clearly defined to easily be replicated enabling researches to focus on the science instead of trying to glue different tools together [19]. Many have embraced Docker as a way to distribute easily reproducible raw and TIDY data along with several forms of code and even as a standardized way to share algorithms with AlgoRun [20].

51.4 Applying Docker to OCRSpell

51.4.1 Installing Docker

Docker will run in any of today's popular hardware platforms available and continued support due to its open-source nature ensures it will continue to do so. It can be downloaded and installed from the official website, which has step-by-step instructions on how to do so:

https://docs.docker.com/engine/installation/#platform-support-matrix

We recommend installation of Docker CE as this is the open-source community edition. For the remainder of this guide we will assume a Linux host system. However, the advantage of Docker means that the following commands will work in any platform the host system uses. To test the install make sure the docker service is running:

```
service docker start
```

and that we are connected to the docker repository by running a test container:

```
docker run hello-world
```

If it installed correctly it will pull the image from docker and display a message saying (among other things):

```
Hello from Docker.
```

This message shows that the installation appears to be working correctly.

If Docker was previously installed, we recommend deleting any existing images and containers before following this guide further. See Sect. 51.4.5 for instructions on how to do so.

51.4.2 Creating the Container

To create a Docker container, we must first create a Dockerfile. A Dockerfile contains the instructions needed to generate the appropriate image and includes all required dependencies. Each instruction in the Dockerfile creates an intermediary container that adds the change in that line. In that sense Docker is very modular; one can take other author's containers, add or remove a dependency or program, and publish their own version of the container for others to use. Using this concept we will begin to create our container by pulling a basic image of the Ubuntu OS as the base where OCRSpell will run. So we create a text file named Dockerfile and in the first line we add:

```
FROM ubuntu:14.04.2
```

This command pulls and installs the ubuntu:14.04.2 docker container. Next we add a maintainer line which adds a label to our container of who to contact in case of support.

```
MAINTAINER my@email.com
```

If we were to generate our container we would have a basic, non-gui working version of Ubuntu 14.04.2. However, OCRSpell requires certain dependencies that are not included in this basic version of Ubuntu to run. These dependencies may be acquirable today with the right install commands, but as they become deprecated with time, they may no longer be accessible. However, once we build our image, they will be saved permanently and never need to be reacquired again, enabling us to solve the issue of deprecated dependencies and code-rot. To install these dependencies we use the `RUN` command. First we will update the Ubuntu package list:

```
RUN apt-get -yqq update
```

Similarly we start to install the required libraries, software, and other dependencies we will need. We had to analyze the requirements of OCRSpell in order to determine these, but when a new author creates a container. They will be aware of what they need and easily include it:

```
RUN apt-get install unzip -yqq
RUN apt-get install autoconf -yqq
RUN apt-get install gcc -yqq
RUN apt-get install make -yqq
RUN apt-get install xutils-dev -yqq
RUN apt-get install nano -yqq
```

Notice we installed nano in order to be able to edit files within the container. The Ubuntu OS we have is as lightweight as possible and does not include basics like nano. Next we would normally want to install ispell, but instead we will do so while the container is running to show that

option. Next we want to create our work directory before copying our source files. We use the `WORKDIR` command for that:

```
WORKDIR /home/ocrspell
```

This creates a folder in the virtual home folder called ocrspell. From now on whenever we start our container this will be the default starting directory. Next we want to download the source files from the github repository as a zip file. To do so we can use any web browser and copy the link in the quotes below or we can open a new terminal window and type:

`wget` "https://github.com/zw/ocrspell/archive/master.zip"

When finished downloading, place the zip file in the same directory as our Dockerfile. With this done we add the following command to copy the files into the image:

```
COPY master.zip ./
```

For our next line we want the container to unzip these files so we use `RUN` again:

```
RUN unzip master.zip
```

This prepares everything to start compiling the source files. As per the instructions in the readme, the first step is to run `autoconf` to generate a configure script. We can do this with a complex `RUN` command.

```
RUN cd ocrspell-master/src;autoconf;
  chmod +x configure.in ./
```

Because every RUN command is run on its own intermediary container we cannot create a run command to enter the folder src and another to run autoconf or chmod since each RUN command starts at the root folder which is our `WORKDIR` folder. To solve this semicolon is used to send multiple commands.

We could continue setting up the container from within the docker file by calling the makedepend and make commands as per the readme, but since we still need to install ispell and also do some manual modifications to get OCRSpell running due to its age, it is time to start up the container to do this.

51.4.3 Running the Container

So far we have a text file called Dockerfile with all of the above commands, but in order to create the container we have to feed this file to Docker. We do this by opening a new terminal in the same directory as the Dockerfile and OCRSpell zip source file(master.zip) and typing:

```
docker build -t ocrspell-master .
```

This will take some time as it downloads the Ubuntu image, runs it, downloads and install the dependencies (each on its own intermediate container), and then executes the remainder of the Dockerfile. When it is complete our image is ready to run. If there are any errors in any Dockerfile line the intermediate containers will remain in order to either run those and find the problem or fix the problem and not have to start building from scratch. This is a great way to show with reproducible research the steps taken to reach a final version of a program. When ready, we start up our image by typing in the terminal:

```
docker run -it ocrspell-master /bin
  /bash
```

We are now running our container! Because the base is Ubuntu, any terminal command will work. A good way to test is to type in the terminal:

```
ls
```

It should show both our zip file source files and the unzipped directory. Now we can install ispell by running in our terminal:

```
apt-get install ispell -yqq
```

This is the same command we would do in a regular Linux machine. After it is done installing we can test it by running:

```
ispell -help
```

Now it is time to finish building the source. First, let's enter the `src` folder.

```
cd ocrspell-master/src/
```

As the readme on github points out, due to the age of OCRSpell and deprecated code, a few changes by hand must be made in order for the configure file to build correctly. First lets open the file in nano:

```
nano configure
```

Then go to line 712 and change:

```
prefix=NONE
```

to

```
prefix=/home/ocrspell/
  ocrspell-master/src
```

Next, go to lines 732 and 733 and change:

```
datarootdir='${prefix}/share'
datadir='${datarootdir}'
```

to

```
datarootdir='/home/ocrspell/
  ocrspell-master/src'
datadir='/home/ocrspell/
  ocrspell-master/src'
```

respectively. Save the file and close nano. Now we can run `./configure` and then build as per the readme with makedepend and make:

```
./configure -prefix=/
  local/ocrspell-master
makedepend
make
```

OCRSpell is now fully compiled and we can test it with the included test file called `quick_demo,txt`:

```
./ocrspell -a -f ./ocr.freq < quick_d
  emo.txt
```

We have now successfully set up our container; however, due to the nature of the container. If we were to close the container we would lose all changes and the container would

open in the same state as if he had just done the docker run command. So in order to save our final container we can use the `commit` command in a `different` terminal window than the one we are working with our container on. The syntax is:

```
docker commit <CONTAINER NAME FROM
  DOCKER PS> <file name>
```

In this case we type:

```
docker ps
```

to find our Container name and then use the commit command with those values and a name for our new container image. These names are randomly generated and change each time we run the container so we must pay special attention to selecting the right one; in this case it is `distracted_bardeen`:

```
docker commit distracted_bardeen
  ocrspell-final
```

We now have a saved version of our container and can close the running container and re-open it to the exact same state as it is now. To close it, in the terminal window where our container is active we type:

```
exit
```

If we want to save our docker image to share with other collaborators or publications we use the `save` command:

```
docker save -o <save image to
  path> <image name>
```

an example of this in our case is:

```
docker save -o /home/jfunlv/Desktop/
  test/ocrspell-final ocrspell-final
```

Be sure to modify the first part with the path where we will save the image. If there are any issues modifying the output image, use chmod and chown in linux to give rights to it or the equivalent in other platforms.

If we want to upload to the docker repository for other Docker users to collaborate or reproduce our research we need to create a free docker account at docker.com Once we have an account we have to tag and push the image. To do this we find the image id by typing:

```
docker images
```

Then, we tag the image by using:

```
docker tag 90e925195d6c username/
  ocrspell-final:latest
```

Replace username with one's docker user name and replace the image id with that of the appropriate image. Make sure one is logged in to docker by typing:

```
docker login
```

Finally we push by:

```
docker push username/ocrspell-final
```

For full details on tagging and pushing see:

https://docs.docker.com/engine/getstarted/step_six/

We have now uploaded a copy of our image for others to try, contribute and much more.

51.4.4 Downloading Our Version of the Container

The steps explained above should be done by the author of the research, but what if we want to download someone else's work and replicate it? In this case we will download the image that we created when writing this guide and run OCRSpell with a few simple commands.

We can download the image from an arbitrary location and load it up on docker, or download it from the official docker git repository:

1. Load image in Docker: Suppose we downloaded from the author's website an image with a container to the project of OCRSpell. The file name is 'ocrspell-image'
   To load this image we open up a terminal at the image location and type:
   ```
   docker load -i ocrspell-image
   ```
   Make sure one is in the directory of the image otherwise type the path to it such as:
   ```
   docker load /home/user/
     Desktop/ocrspell-image
   ```
   At this point we can type: `docker images` and verify that the entry was added. In our case, 'ocrspell-master'. As before we can now run it by typing:
   ```
   docker run -it ocrspell-
     master /bin/bash
   ```
   We now have a running container with OCRSpell that will not break or deprecate and includes all necessary source and software to run.
2. Download directly to Docker from Docker git repository: To search from the available images we use the 'docker search command' by typing the name of the image after the search keyword:
   ```
   docker search ocrspell
   ```
   A list of matching images will appear. In this case we want the ocrspell-master image from the unlvcs user, to download it we use the pull command:
   ```
   docker pull unlvcs/ocrspell-master
   ```
   If we check `docker images` we should see it listed under the repository and can now run it.

51.4.5 Deleting Old Images and Containers

Once we are done working with an image we can use the `docker rmi` command followed by the image ID which can be found next to the name of the image in `docker images`. In this case our Image id is: 90e925195d6c; therefore, we type in a terminal:

```
docker rmi 90e925195d6c
```

If there is a warning that the image is currently being used by a container. Make sure to first close the container and then delete the image. If we want to delete all containers stored in docker we can do so by typing in a terminal:

```
docker rm $(docker ps -a -q)
```

Similarly we can delete all images by typing:

```
docker rmi $(docker images -q)
```

51.4.6 Transferring Files In and Out of a Container

So far only when executing the Dockerfile did we copy data onto the container. Ideally we want to copy all of our data at this time and store it within the container to have everything in one place. But suppose that due to portability or projects with large file size we want to maintain the data in a separate docker container or repository in general. Docker communications within containers can easily achieve this with the `docker network` command. See for full details:

https://docs.docker.com/engine/userguide/networking/

If we want to instead transfer data from the container to and from the host machine we use the `docker cp` command.

51.4.7 Using the OCRSpell Container

Suppose we have an OCR document named `FR940104.0.clean` we would like to spell check with OCRSpell. The document is in our host machine and we have pulled the latest version of the OCRSpell container from the Docker repository. First we start our container by running in a new terminal:

```
docker run -it unlvcs/ocrspell
  -master /bin/bash
```

then we maneuver to the ocrspell program:

```
cd ocrspell-master/src
```

Next, we open a new terminal window at the location of our OCR document we will be running and type:

```
docker ps
```

We find and save our container name there for ocrspell-final. In this case it is `adoring_wescoff`, but it will be different each time. We are now ready to copy the file into the container. We run the following command:

```
docker cp FR940104.0.clean adoring_
  wescoff:/home/ocrspell/ocrspell-
  master/src/FR940104.0.clean
```

Note one will need to enter the correct name for one's container instead of 'adoring_wescoff'. The file should have copied successfully, so back in our container terminal we can type `ls` and verify it is there.

Now we run OCRSpell and rather than output to the terminal we will output to a file called `FR940104.0.output` which we will later copy back to the host machine. We use simple linux redirection to achieve this.

```
./ocrspell -a -f ./ocr.freq <
  FR940104.0.clean > FR940104.0.output
```

OCRSpell has now done its job for us. Now to copy the file back we open a terminal window where we wish to save our file and run the following command:

```
docker cp adoring_wescoff:/home/
  ocrspell/ocrspell-master/src/
  FR940104.0.output FR940104.0.output
```

The file is now saved in the host machine. We can now close the container with the `exit` command. Next time we open the container none of the files will exist in it as explained earlier. If we wish to retain this data in the container we use the commit and save message as mentioned previously.

51.5 Results

We have successfully used OCRSpell without having to go through the difficulties of tracking down source code and dependencies, compiling, or worrying about compatibility with hardware. We have made reproducible research in that anyone can take our data and run it through OCRSpell, modify the code without issues and publish their own version to improve on previous work.

As shown in this example, the steps taken to place OCRSpell into Docker are trivial and mechanical. The complicated part was finding the dependencies and by-hand modifications and making sure they were valid when first building the initial image. In new research this can be considered and easily stored in a Dockerfile making the added work to researchers minimal. The Dockerfile itself is a great way to view the construction of the image, and the container is a way to ensure that the reproducibility remains as it can be checked when trying to rebuild the image to see if it matches.

51.6 Conclusion

This paper reported on the state of reproducible research and its importance along with the challenges it brings. Docker containers are reviewed as a solution to consistent reproducible research and a complete application is shown with OCRSpell, a program that does not easily run in its current state, but after being placed in a container is now immortalized to be able to be tested anywhere and anytime with ease. Possible extensions to this project include parallel programming and complex GUI docker implementations examples, but more importantly a proposed standard for reproducible research for the ICDAR and other conferences in the hopes of a better future where we can all share

and work together to create good reproducible research that benefits the scientific community.

References

1. R.D. Peng, Reproducible research in computational science. Science **334**(6060), 1226–1227 (2011)
2. G.K. Sandve, A. Nekrutenko, J. Taylor, E. Hovig, Ten simple rules for reproducible computational research. PLoS Comput. Biol. **9**(10), e1003285 (2013)
3. K. Ram, Git can facilitate greater reproducibility and increased transparency in science. Source Code Biol. Med. **8**(1) 7 (2013)
4. H. Wickham et al., Tidy data. J. Stat. Softw. **59**(10), 1–23 (2014)
5. C. Collberg, T. Proebsting, G. Moraila, A. Shankaran, Z. Shi, A.M. Warren, Measuring reproducibility in computer systems research, Technical report, 2014
6. N. Barnes, Publish your computer code: it is good enough. Nature **467**(7317), 753 (2010)
7. J.P. Ioannidis, Why most published research findings are false. PLos Med **2**(8), e124 (2005)
8. T.H. Vines, R.L. Andrew, D.G. Bock, M.T. Franklin, K.J. Gilbert, N.C. Kane, J.-S. Moore, B.T. Moyers, S. Renaut, D.J. Rennison et al., Mandated data archiving greatly improves access to research data. FASEB J **27**(4), 1304–1308 (2013)
9. Testimony on scientific integrity & transparency. https://www.gpo.gov/fdsys/pkg/CHRG-113hhrg79929/pdf/CHRG-113hhrg79929.pdf. Accessed 2017-03-01
10. J.T. Leek, R.D. Peng, Opinion: reproducible research can still be wrong: Adopting a prevention approach. Proc. Natl. Acad. Sci. **112**(6), 1645–1646 (2015)
11. G. Marcus, E. Davis, Eight (no, nine!) problems with big data. New York Times **6**(04), 2014 (2014)
12. C. Boettiger, An introduction to docker for reproducible research. ACM SIGOPS Oper. Syst. Rev. **49**(1), 71–79 (2015)
13. I. Jimenez, C. Maltzahn, A. Moody, K. Mohror, J. Lofstead, R. Arpaci-Dusseau, A. Arpaci-Dusseau, The role of container technology in reproducible computer systems research, in *2015 IEEE International Conference on Cloud Engineering (IC2E)* (IEEE, New York, 2015), pp. 379–385
14. L.-H. Hung, D. Kristiyanto, S.B. Lee, K.Y. Yeung, Guidock: using docker containers with a common graphics user interface to address the reproducibility of research. PloS One **11**(4), e0152686 (2016)
15. P. Di Tommaso, E. Palumbo, M. Chatzou, P. Prieto, M.L. Heuer, C. Notredame, The impact of docker containers on the performance of genomic pipelines. PeerJ **3**, e1273 (2015)
16. D. Hládek, J. Staš, S. Ondáš, J. Juhár, L. Kovács, Learning string distance with smoothing for OCR spelling correction. Multimedia Tools and Applications **76**(22), 24549–24567 (2017)
17. K. Taghva, E. Stofsky, Ocrspell: an interactive spelling correction system for OCR errors in text. Int. J. Doc. Anal. Recogn. **3**(3), 125–137 (2001)
18. K. Taghva, T. Nartker, J. Borsack, Information access in the presence of OCR errors, in *Proceedings of the 1st ACM Workshop on Hardcopy Document Processing* (ACM, New York, 2004), pp. 1–8
19. P. Belmann, J. Dröge, A. Bremges, A.C. McHardy, A. Sczyrba, M.D. Barton, Bioboxes: standardised containers for interchangeable bioinformatics software. Gigascience **4**(1), 47 (2015)
20. A. Hosny, P. Vera-Licona, R. Laubenbacher, T. Favre, Algorun, a docker-based packaging system for platform-agnostic implemented algorithms. Bioinformatics **32**, btw120 (2016)