One of the main challenges that every data scientist face is model deployment. Unless you are one of the lucky few who has loads of data engineers to help you deploy a model, it’s really an issue in enterprise projects. I am not even implying that the model needs to be production ready but even a seemingly basic issue of making the model and insights accessible to business users is more of a hassle then it needs to be. There used to be 2 main ways of solving the problem:

1. Ad-hoc manual runs at every request
2. Hosting the code on a server and writing an API interface to make the results available

These are two ends of the spectrum. Ad-hoc runs are just too tedious and clients typically demand for some self-serve interface but good luck trying to get a permanent server to host your code. Turns out there is a third way – the serverless way! AWS Lambdas and Google Cloud Platform Cloud Functions have really opened up a new way of serving results without having to manage any infrastructure. If you have access to the cloud, this is a very attractive option.

If you are from the javascript or python world you might have already used some of these tools. The problem with cloud functions is that it is restricted to particular environments. However, with the recent introduction of GCP’s [Cloud Run](https://cloud.google.com/run/docs/reference/container-contract), we are no longer limited by this problem. Cloud run allows custom docker images to be served on the cloud, opening up the serverless realm to many interesting possibilities. This means that R users can also finally have a way to develop and deploy serverless ML models![1](https://www.timlrx.com/2020/01/22/serverless-machine-learning-with-r-on-cloud-run/#fn1)

This post documents some of my experience with cloud run and hopefully serves as a good reference template for anyone who might want to try it out. I take inspiration from two other sources, namely [Mark’s blog](https://code.markedmondson.me/googleCloudRunner-intro/) and [Eric’s post](https://ericjinks.com/blog/2019/serverless-R-cloud-run/). Mark actually has a package on CRAN that automates some of the deployment work and Eric has a nice bit on continuous integration pipelines so do check it out.

This post is more of a practical guide with a real world machine learning application that an analyst might develop and wish to deploy, which I think makes the application a lot more concrete and useful. It’s also a really fun little side project.

**Will My Project Run on the Cloud Run?**

Cloud run is not a panacea. Check out the [requirement specification](https://cloud.google.com/run/docs/reference/container-contract) and [limits](https://cloud.google.com/run/quotas) for more details but here are some limitations that you should be aware of.

* States are not persisted (if you want to persist them you need an external database)
* The maximum memory limit is 2GB
* The container must start a server within 4 minutes after receiving a request and it times out after 15 minutes

This makes it well suited for short computations, but not tasks which might require lots of memory or are very CPU intensive. 15 minutes in reality is not too bad! You could probably run some regressions, decision trees, even solve a linear programming problem but maybe not train a neural network.

**Twitter Project **

Here’s my fun serverless-ml weekend project: an application that analyzes the twitter-verse. I wanted to generate two plots: a graph which compares the frequency of tweets over time and another one which does a sentiment analysis on the tweets. As an added bonus, I decided to make it interactive – this means serving static plots as well as interactive plotly results.

The main package I used were: rtweet, dplyr, ggplot2, tidytext, tidyr and stringr. If you are new to tidytext, check out some of my previous posts such as this one way back in 2017 which analyzes [recipe books](https://www.timlrx.com/2017/06/24/thesis-thursday-4-analysing-recipes/).

rtweet provides a convenient api to collect users timeline information. You would need a twitter API account to get started. It’s a simple process and you can register for one here: <https://developer.twitter.com/en/apply-for-access>.

Take note of the four keys/tokens. They should be saved as an environment variable in your system.[2](https://www.timlrx.com/2020/01/22/serverless-machine-learning-with-r-on-cloud-run/#fn2) These four keys correspond to API\_KEY, API\_SECRET\_KEY, ACCESS\_TOKEN and ACCESS\_SECRET in the tweet.R file and will be retrieved programmatically when needed.

[**Tweet.R**](https://www.timlrx.com/2020/01/22/serverless-machine-learning-with-r-on-cloud-run/)

The important part of it is that we are encapsulating each part as a function which we then call in the main api routing file (app.R).

For the sentiment analysis, we are counting the number of positive and negative words as matched by a dictionary. The lexical dictionary is from [Bing Liu et al.](https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html). Doesn’t this require at external database? Not really – external dependencies or data files are fine, as long as they are stateless. We can package it together with our docker file or in this case it comes installed with the tidytext package!

**App.R**

This file contains the serving logic. We are using the [plumber package](https://www.rplumber.io/) which allows us to create a REST API in R by decorating it in with some markup. You can specify query parameters by using a #\* @param markup and specify the type of output to return such as #\* @png for a static image or #\* @html for html.

As an added bonus, there’s also an out of the box option to make htmlwidgets work (#\* @serializer htmlwidget). This makes our serving plotly results really simple. I decided to create two paths for each plot, a static one as well as a plotly interactive result. So in total we have four paths:

* /frequency (ggplot)
* /html/frequency (plotly)
* /sentiment (ggplot)
* /html/sentiment (plotly)

The functions all accept two arguments: n – the number of tweets, and users which could be a comma separated list of user IDs which we will then query the twitter API via rtweet for the relevant information.

**Server.R**

This is the bit of code that starts our plumber server. We infer the port as defined by an environment variable (PORT).

That’s it for the R code. Now you should have a working application which you can run locally from your computer. Next, we get into the grimy details of ML-ops ‍. This involves packaging our dependencies with docker and deploying it on cloud run.

**Docker**

Docker is a platform which packages different software, configurations and environments into containers which neatly encapsulate your application. The end user just needs to list out the installation steps to build the image which can subsequently be run on the docker platform .

To jump start the configuration, we build on top of the official [r-base image](https://hub.docker.com/_/r-base). It’s a Linux image and we need to install some additional dependencies to make the application work, namely libssl-dev for rtweet and pandoc for dealing with htmlwidgets. This forms the start of our Dockerfile:

FROM r-base

RUN apt-get update -qq && apt-get install -y \

git-core \

libssl-dev \

libcurl4-gnutls-dev \

pandoc

Next, we copy the scripts in our directory to the app directory in the container and install the necessary R libraries using the Rscript function:

WORKDIR /usr/src/app

# Copy local code to the container image.

COPY . .

# Install any R packages

RUN Rscript -e "install.packages('plumber')"

RUN Rscript -e "install.packages(c('rtweet', 'dplyr', 'ggplot2', 'plotly', 'tidytext', 'tidyr', 'stringr'))"

We expose port 8000 (this is more for documentation) and run the server when the container is launched:

EXPOSE 8000

# Run the web service on container startup.

CMD [ "Rscript", "server.R"]

let’s build the docker image and give it a run:

docker build -t ${IMAGE} .

docker run -p 8000:8000 -e PORT=8000 -e API\_KEY -e API\_SECRET\_KEY -e ACCESS\_TOKEN -e ACCESS\_SECRET ${IMAGE}

${IMAGE} here represents the name which you can assign to the image. Remember the environment variables that we need for the application? Port for plumber and the API keys to access twitter API? We pass it to the container when we are running it. Note: The build process is quite long, with the image being 1.22GB big.[4](https://www.timlrx.com/2020/01/22/serverless-machine-learning-with-r-on-cloud-run/#fn4)

If the image runs successfully, you should be able to access the routes on your browser. Now, we can take it from your local machine to the web!

**Cloud Run**

You can use my deploy.sh script as a guide on how to build and deploy your image. Before doing so, I recommend that you assign the Cloud Run Admin role to the account or user you are running the script in order for it to be deployed correctly. You can do it from the [IAM panel](https://console.cloud.google.com/iam-admin) within GCP.

In the script, I retrieved the project ID programmatically, but feel free to substitute it with your GCP project. All the script does is to upload the local docker image to Google’s container repository and run cloud run to deploy the image from the repository:

gcloud alpha run deploy \

--image="[gcr.io/${PROJECT\_ID}/${IMAGE}:1.0.0](http://gcr.io/$%7BPROJECT_ID%7D/$%7BIMAGE%7D:1.0.0)" \

--region="us-central1" \

--platform managed \

--memory=512Mi \

--port=8000 \

--set-env-vars API\_KEY=${API\_KEY},API\_SECRET\_KEY=${API\_SECRET\_KEY},ACCESS\_TOKEN=${ACCESS\_TOKEN},ACCESS\_SECRET=${ACCESS\_SECRET} \

--allow-unauthenticated

We add an allow-unauthenticated to allow public traffic and increase the memory as the default was too low. You can start with the default 256MB first but if you encounter any errors, do check the cloud run logs which is very useful for debugging any errors. If all goes well, you should be greeted with an image like this:

Our twitter project is now successfully hosted on Cloud Run!

**Test it out**

Here’s the fun part – try it out and visualize and analyse live twitter data!

You can try out my hosted cloud run with one of the 4 endpoints listed above: <https://twitter-r-cvdvxo3vga-uc.a.run.app/>

Some fun examples!

**Frequency of Obama tweets**

The most followed person on Twitter with 112 million followers does not actually tweet too frequently :  
<https://twitter-r-cvdvxo3vga-uc.a.run.app/frequency?n=500&users=BarackObama>

**Sentiment analysis comparison between BBCworld and realDonaldTrump**

Sad![5](https://www.timlrx.com/2020/01/22/serverless-machine-learning-with-r-on-cloud-run/#fn5)  
<https://twitter-r-cvdvxo3vga-uc.a.run.app/sentiment?n=1000&users=BBCWorld,realDonaldTrump>

**What do politicians and entertainers have in common?**

They are overwhelmingly positive (here we use our plotly html endpoint)  
<https://twitter-r-cvdvxo3vga-uc.a.run.app/html/sentiment?n=500&users=narendramodi,TheEllenShow>

**Conclusion**

That’s it for this serverless + R tutorial. Hope you managed to learn something useful or at the very least find the twitter analysis interesting . The nice part about having this application running is that you can analyze real-time twitter frequency or sentiment plots with whatever account you choose (even your own), so feel free to try it out.