DATA SCIENCE IN PRODUCTION

An AI & Machine Learning Editorial

Rhyme with AI

IP whitelisting your Chalice application

GCP powered EV charging

0 2 4 0 1 2 3 4 5 6 7 6

GO DATA DRIVEN

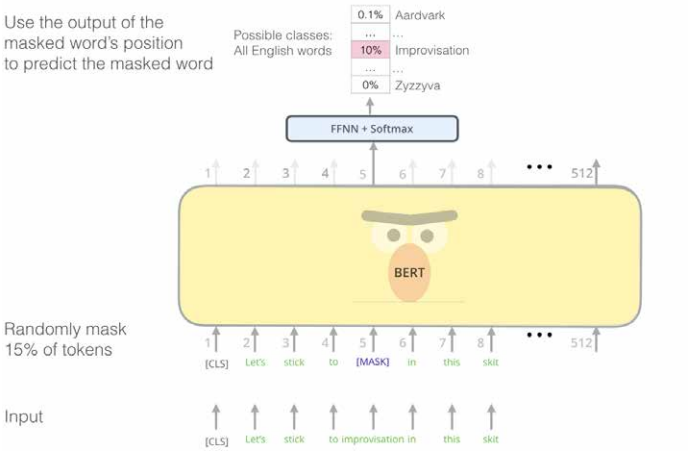
INDEX

RENS DIMMENDAAL
HENK GRIFFIOEN

RHYME WITH AI _

Language modeling helps state-of-the-art models understand languages before solving tasks like sentiment analysis or translation. Masking, where the model tries to predict a word that is hidden from a sentence, is one of BERT's innovations. We can use it to help us rhyme by rephrasing rhyming as a task to predict missing words.

NLP's ImageNet moment may have arrived in 2018, but the ecosystem around NLP models really has matured in 2019. Many of these models (BERT, GPT-2, Transformer-XL, DistilBERT, etc.) are easy to use for your use cases. Our service uses BERT to help us with our (Christmas) rhymes.



From the "The Illustrated BERT, ELMo, and co.". Predicting masked tokens is one of BERT's language modelling techniques.

Our problem involves multiple masks: we know the first sentence and the last word of the second sentence. For instance:

Santa delivers gifts by sleigh

– ... [MASK] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK] day

Inspired by BERT has a Mouth, and It Must Speak, we first let BERT fill in the [MASK]'s and then randomly sample new tokens. Some example rhymes from our model:

Santa delivers gifts by sleigh

– ... and drinks and celebrates his wedding day
– ... dressed as preacher and nurse they say
– ... or bicycle if he has to pay

This already looks pretty good, but we need a solution that people can use!

Luckily, creating an end-to-end machine learning solution is fairly simple. The Datamuse API gives back rhyme words, BERT is available via huggingface and creating an app is no sweat with streamlit. Put it all together in a Docker container and a hosted solution is one command away with Google App Engine.

A few days well spent for us will hopefully save you a lot of pain. Check out or code on GitHub or try our solution at Rhyme with AI!

TIM VAN CANN
DANIEL VAN DER ENDE

SCHIPHOL TAKEOFF — AUTOMATE DEPLOYMENT — — WITHOUT WRITING CODE!

What makes Schiphol Takeoff awesome? Right out of the box Schiphol Takeoff provides a sensible way to deploy your application across different environments!

During our time at Schiphol Group, we built a project which helps automate deployments. Schiphol Group was kind enough to let us open source this project. We'll give a quick introduction to what it does and how it could help you get to production quicker.

OUR USE CASE

To give a bit more insight into why we built Schiphol Takeoff, it's good to take a look at an example use case. This use case ties a number of components together:

- Data arrives in a (near) real-time stream on an Azure Eventhub.
- A Spark job running on Databricks consumes this data from Eventhub, processes the data, and outputs predictions.
- A REST API is running on Azure Kubernetes Service, which exposes the predictions made by the Spark job.

Conceptually, this is not a very complex setup. However, there are quite a few components involved:

- Azure Eventhub
- Azure Databricks
- Azure Kubernetes Service

Each of these individually has some form of automation, but there is no unified way of coordinating and orchestrating deployment of the code to all at the same time. If, for example, you were to change the name of the consumer group for Azure Eventhub, you could script that. However, you'd also need to manually update your Spark job running on Databricks to ensure it could still consume the data.

Moreover, this list of components is not complete. These are the components that operate 'on the foreground' to deliver the service. Components like Azure Keyvault (for storing secrets), private PyPi repository (for storing Python artifacts), and Azure Blob Storage (for storing artifacts on disk), are not mentioned here, yet play an important role.

Finally, not only does this setup requires quite some configuration to orchestrate the components; in a proper production-like setting, you will probably have more than one environment. Most likely, you'll have at least a Development and Production environment, to ensure that mistakes by developers (we're all human after all) don't affect your end users. This complicates matters even further, because now not only do you need to keep all the components in line, you also need to ensure this happens reliably across environments, without impacting users.

As you can see, without going into deep technical detail of what you would need to do (this would involve a lot of screenshots, yaml, and custom configuration per component), this simple setup results in a complex productionisation, with many pitfalls along the way.

WHAT MAKES SCHIPHOL TAKEOFF AWESOME?

ENTER SCHIPHOL TAKEOFF

Schiphol Takeoff's goal is twofold:

1. Remove the load placed on data scientists and developers of knowing details about multiple components and how their APIs work.
2. Ensuring reliable and, most importantly, easy deployment of a project is possible.

To achieve the deployment of the project described in the above, Schiphol Takeoff would require a few things:

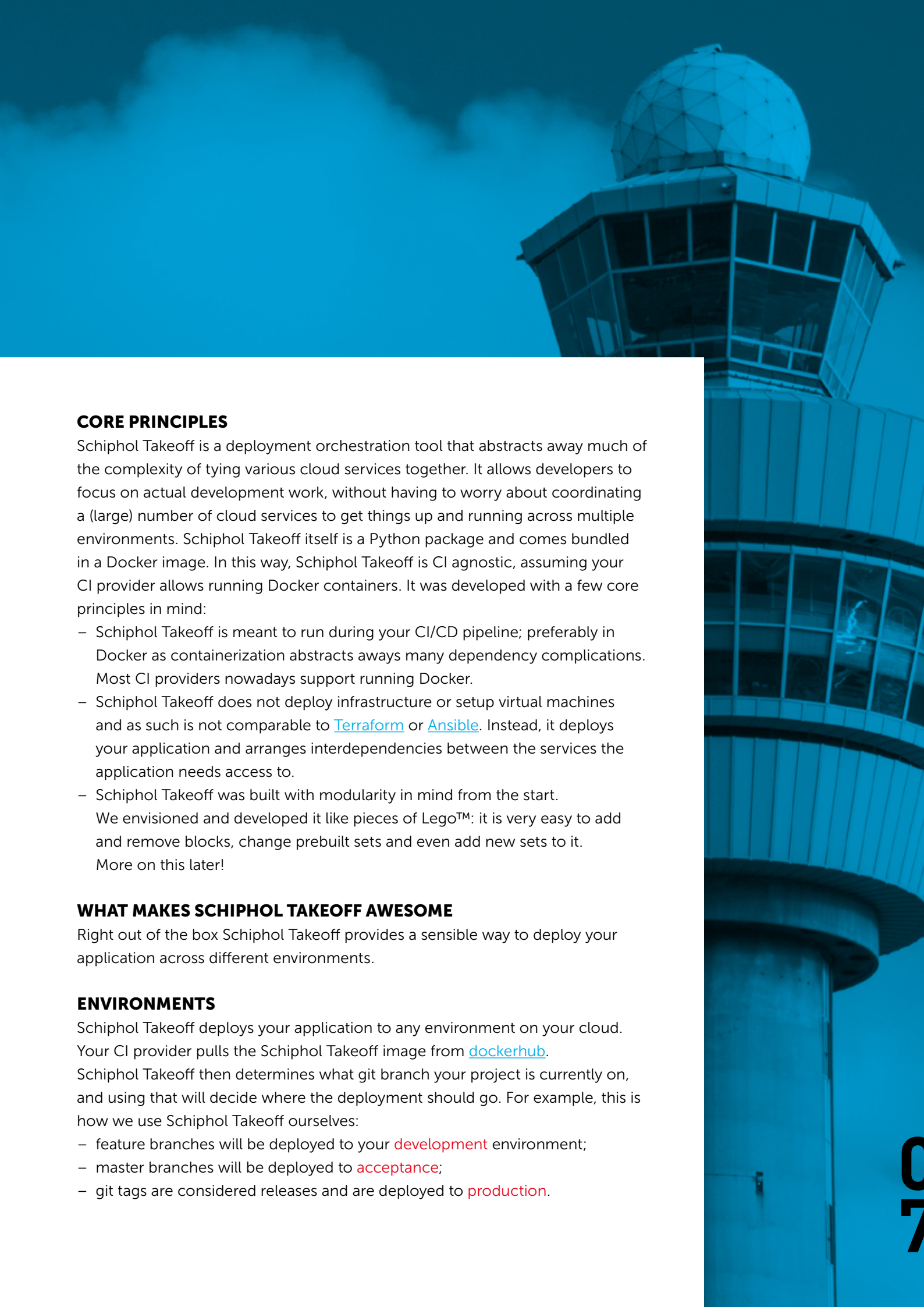
1. A working CI environment (with Docker support) for it to run in.
2. Azure Keyvault setup with the required secrets for the various components.
3. Two files in your project repository:
   - A Takeoff configuration yaml, which tells Takeoff what the names of your secrets are in the Keyvault.
   - A Takeoff deployment yaml, which tells Takeoff which tasks it needs to execute.

We don't want to make this blogpost a yaml-fest, so we won't go into details of both these files. If you want to know more, head over to Takeoff's documentation website or the Github repository.

It is useful, however, to show the Takeoff deployment yaml, as it clearly shows how little a developer would need to do to get things up and running, and to define steps to deploy. Please note that in a "real-world" situation you probably would split up some things into separate repositories (i.e. you probably would have the REST API in a separate repository). This example is purely to demonstrate Takeoff's capabilities.

```
steps:
 - task: configure_eventhub
  create_consumer_groups:
      - eventhub_entity: input-eventhub
        consumer_group: algorithm-group
        create_databricks_secret: true
      - eventhub_entity: input-eventhub
        consumer_group: rest-sink-group
        create_databricks_secret: true
  create_producer_policies:
      - eventhub_entity: output-eventhub
      create_databricks_secret: true
 - task: build_artifact
    build_tool: python
 - task: publish_artifact
    language: python
    python_file_path: "main/main.py"
    target:
      - cloud_storage
 - task: deploy_to_databricks
    jobs:
      - main_name: main/main
        config_file: databricks.json.j2
        lang: python
 - task: deploy_to_kubernetes
    deployment_config_path:
    "k8s_config/deployment.yaml.j2"
   service_config_path: "k8s_config/service.
   yaml.j2"
```

These 27 lines (yeah, we counted) are all you need. Every time you commit to your project now, these steps will be run, and will deploy your application per environment (depending on how you've setup your deployment configuration).



CORE PRINCIPLES

Schiphol Takeoff is a deployment orchestration tool that abstracts away much of the complexity of tying various cloud services together. It allows developers to focus on actual development work, without having to worry about coordinating a (large) number of cloud services to get things up and running across multiple environments. Schiphol Takeoff itself is a Python package and comes bundled in a Docker image. In this way, Schiphol Takeoff is CI agnostic, assuming your CI provider allows running Docker containers. It was developed with a few core principles in mind:

- Schiphol Takeoff is meant to run during your CI/CD pipeline; preferably in Docker as containerization abstracts aways many dependency complications. Most CI providers nowadays support running Docker.
- Schiphol Takeoff does not deploy infrastructure or setup virtual machines and as such is not comparable to Terraform or Ansible. Instead, it deploys your application and arranges interdependencies between the services the application needs access to.
- Schiphol Takeoff was built with modularity in mind from the start. We envisioned and developed it like pieces of Lego™: it is very easy to add and remove blocks, change prebuilt sets and even add new sets to it. More on this later!

WHAT MAKES SCHIPHOL TAKEOFF AWESOME

Right out of the box Schiphol Takeoff provides a sensible way to deploy your application across different environments.

ENVIRONMENTS

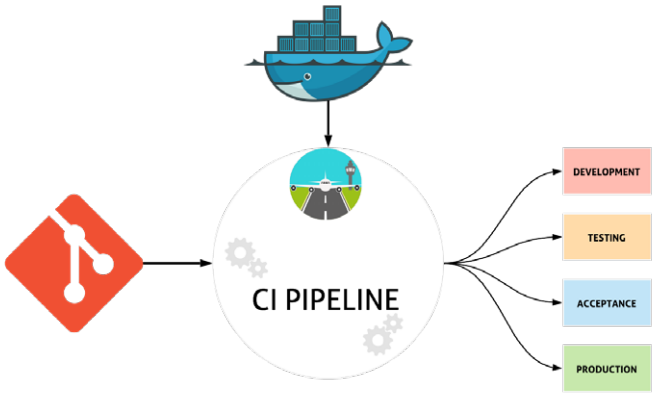
Schiphol Takeoff deploys your application to any environment on your cloud. Your CI provider pulls the Schiphol Takeoff image from dockerhub.
Schiphol Takeoff then determines what git branch your project is currently on, and using that will decide where the deployment should go. For example, this is how we use Schiphol Takeoff ourselves:

- feature branches will be deployed to your development environment;
- master branches will be deployed to acceptance;
- git tags are considered releases and are deployed to production.

It will also make sure versions are preserved during deployment to these environments -- given the previous example - development will receive a version equal to the name of your feature branch; - acceptance will receive the version SNAPSHOT; - production will take the git tag as version.

Concretely this means that many feature branches may be running simultaneously, but only one *SNAPSHOT* or version will be running.

For this all to work, Schiphol Takeoff makes some assumptions about naming conventions.
For example, in the case of Microsoft Azure, each of these environments basically mean a separate resource group. These resource groups are identical in the fact that they contain the same services, but otherwise might be different in terms of scaling and naming of services. Based on naming conventions Schiphol Takeoff determines during CI which service in which resource group it should deploy to.



PLUGINS

We know that not everyone has the same environments, or might want a different versioning tactic: maybe

- release versions should go to acceptance as well;
- and *SNAPSHOT* should go to testing.

This is where Schiphol Takeoff plugins come in to play. Using Python, we allow you to write your own custom logic regarding what should go where and when. We also allow you to introduce your own naming conventions and logic in the form of a Python plugin.

MODULAR

Schiphol Takeoff was built using Microsoft Azure in mind, as it is the cloud provider used by the Schiphol Data Hub. This means that most services are Azure services, with a few useful exceptions. However, very important to know is that everything in Schiphol Takeoff was built with modularity in mind. In the future, we hope to be able to support other (cloud) platforms.

TESTABLE AND TESTED

Schiphol Takeoff leans heavily on the greatness of Python. It is easy to read, understand and importantly it is very easy to test -- unlike bash scripts, makefiles or generic CI configuration which are significantly harder to test (though not impossible). Hence, most* services are deployed using readily available python SDKs.

- with the exception of very few services using shell to run and deploy. For example: building scala projects using SBT is done by calling a python subprocess.

CI AGNOSTIC

Thanks to the fact that Schiphol Takeoff runs in Docker, we are fully CI agnostic. Most (if not all) major CI providers are capable of running Docker images and even support Docker-in-Docker (DIND). The latter is needed to make sure Schiphol Takeoff has access to the Docker socket in order to build and push docker images, which it can do! Due to some migrations we've had to switch CI providers a few times and found that running Schiphol Takeoff did not change anything in our dependent projects. It generally took around half a day to get to know the new CI provider, setup DIND and everything worked smoothly again!

CLOUD AGNOSTIC (SORT OF...)

As mentioned earlier, Schiphol Takeoff was built with Microsoft Azure in mind, but we would like to stress that this does not mean you have to write your own component that deploys kubernetes applications to Google Cloud Platform on Google Kubernetes Engine. In fact, we already support deploying to Azure Kubernetes Service!

IN SUMMARY...

Schiphol Takeoff is a deployment automation tool that makes your life easier by taking care of interactions with the various services you may need to bring your application to your users. It allows you to focus on your application, rather than all the (cloud) components you need, and gives you reliable deployments across environments. Of course, we may not support the component or service that you need for your application. Luckily it's open source, and we'd be thrilled to see contributions (issues, pull requests etc.) to expand Schiphol Takeoff even further. You can find the source code here.

BAS HARENSLAK

IP WHITELISTING YOUR CHALICE APPLICATION

Chalice is a very useful framework for quickly developing REST APIs with Python hosted on AWS Lambda and exposed via the AWS API Gateway, no infrastructure provisioning required. So now you've written your application, but don't want to expose it to the world wide internet. This blog post demonstrates how to apply a resource policy in Chalice which limits access to a specific (range of) IP address(es).

DEPLOYING A DEMO APPLICATION

For demo purposes, let's deploy a small Chalice application which returns the current time in the given timezone:

```
chalice new-project worldtime
```

In app.py:

```
import datetime

import pytz
from chalice import Chalice, UnprocessableEntityError
from pytz import UnknownTimeZoneError


app = Chalice(app_name="worldtime")

@app.route("/timezone/{timezone}", methods=["GET"])
def gettime(timezone):
    try:
        return f"It's currently {datetime.datetime.now(pytz.timezone(timezone))} in {timezone}."
    except UnknownTimeZoneError:
        raise UnprocessableEntityError(msg=f"Timezone '{timezone}' unknown to pytz.")
```

Deploy with chalice deploy to receive the URL the application is deployed on (account details are obfuscated):

```
$ chalice deploy

Creating deployment package.
Creating IAM role: worldtime-dev
Creating lambda function: worldtime-dev
Creating Rest API
Resources deployed:
  - Lambda ARN: arn:aws:lambda:eu-west-1:012345678999:function:worldtime-dev
  - Rest API URL: https://urwolo1et3.execute-api.eu-west-1.amazonaws.com/api/
```

AUTOMATE DEPLOYMENT

Chalice created the required resources (o.a. Lambda & API Gateway) and we can now call the deployed API from anywhere on the planet, for example:

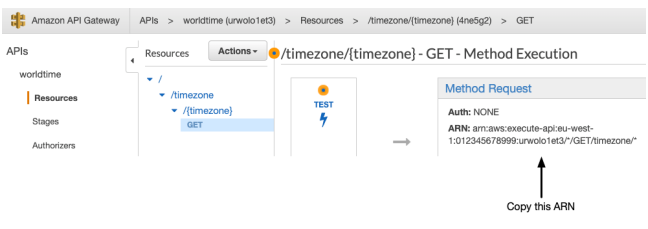
```
curl https://urwolo1et3.execute-api.eu-west-1.
amazonaws.com/api/timezone/utc
It's currently 2019-10-26 09:50:04.948863+
00:00 in utc.

curl -i https://urwolo1et3.execute-api.eu-
west-1.
amazonaws.com/api/timezone/donotcompute
HTTP/2 422
...

{"Code":"UnprocessableEntityError","Message":
"UnprocessableEntityError: Timezone
'donotcompute' un known to pytz."}
```

LIMITING ACCESS TO THE API GATEWAY

If you want your application to be accessible from e.g. only within your company, you can control access to the API Gateway with resource policies. These can be configured in the API Gateway -> Resource Policy tab. First you need the ARN of the deployed endpoint:



Next, insert the following policy in the Resource Policy tab, with your IP address in it (remove /GET/timezone/* to apply the policy to all endpoints):

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "arn:aws:execu
            te-api:eu-west-1:012345678999:
            urwolo1et3/*/GET/timezone/*",
            "Condition": {
                "IpAddress": {
                    "aws:SourceIp": [
                        "123.123.123.123"
                    ]
                }
            }
        }
    ]
}
```

After saving the resource policy, the API Gateway is however still accessible from everywhere.
To enforce the resource policy, we must redeploy the Chalice application. However, when running chalice deploy again, the just configured resource policy disappears, and the endpoint remains open to the world! So, why is this?

CONFIGURING THE CHALICE APPLICATION

Upon deployment, Chalice auto-generates and applies policies. It maintains all state within a .chalice directory generated with the project and does not inspect the AWS project state. As a result, the manually configured policy is overridden with, in this case, nothing, since we haven't configured any policies yet. So let's configure the policy within Chalice instead of the AWS console.

In the .chalice directory, you have a config.json file. The empty config.json looks as follows[1]:

```
{
 "version": "2.0",
 "app_name": "worldtime",
 "stages": {
   "dev": {
     "api_gateway_stage": "api"
   }
 }
}
```

To apply the resource policy to the API Gateway, add a configuration item api_gateway_policy_file:

```
{
 "version": "2.0",
 "app_name": "worldtime",
 "api_gateway_policy_file": "ipwhitelist.json",
 "stages": {
   "dev": {
     "api_gateway_stage": "api"
   }
 }
}
```

Chalice searches for the given filename ipwhitelist.json from the .chalice directory, so create a file .chalice/ipwhitelist.json with the resource policy inside. Next, run chalice deploy once again, and you'll now find the contents of ipwhitelist.json in the AWS console. When calling the API from an IP not defined in the policy, we now receive an error:

```
$ curl https://urwolo1et3.execute-api.eu-west-1.
amazonaws.com/api/timezone/utc
{"Message":"User: anonymous is not authori
zed to perform: execute-api:Invoke on resource:
arn:aws:execute-api:eu-west-1:********8999:
urwolo1et3/api/GET/timezone/utc"}
```

Chalice is very configurable and allows for a much more detailed configuration than the "global" restriction applied above to the entire application, e.g. a policy per stage to restrict the development endpoint to your company IP and allow the production endpoint to the entire world.
It definitely helps to go through the Chalice documentation.

[1] Chalice config file documentation: https://chalice.readthedocs.io/en/latest/topics/configfile.html

OUTPUT._FILE

THE LINEAR ALGEBRA BEHIND LINEAR REGRESSION

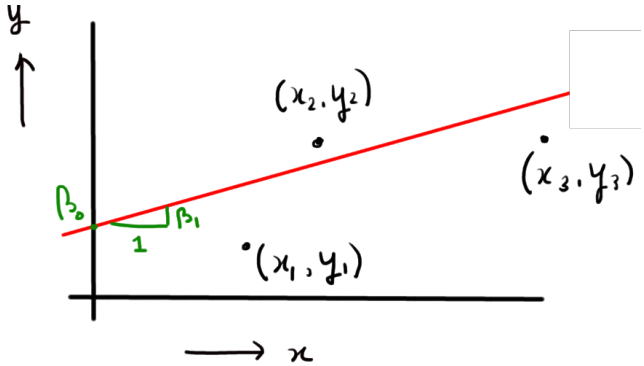
Linear algebra is a branch in mathematics that deals with matrices and vectors. From linear regression to the latest-and-greatest in deep learning: they all rely on linear algebra "under the hood". In this blog post, I explain how linear regression can be interpreted geometrically through linear algebra.

This blog is based on the talk A Primer (or Refresher) on Linear Algebra for Data Science that I gave at PyData London 2019.

LINEAR REGRESSION PRIMER

In Ordinary Least Squares (i.e., plain vanilla linear regression), the goal is to fit a linear model to the data you observe. That is, when we observe outcomes yi and explanatory variables xi, we fit the function which is illustrated below

$$y_i = \beta_0 + \beta_1 x_i + e_i,$$



This boils down to finding estimators ß̂0 and ß̂1 that minimize the mean squared error of the model:

$$\min_{\hat{\beta}_0, \hat{\beta}_1} \sum_{i=1}^{n} \left(y_i - \left(\hat{\beta}_0 + \hat{\beta}_1 x_i \right) \right)^2,$$

where *n* is the number of observations.

To solve this minimization problem, one way forward would be to minimize the loss function numerically (e.g., by using scipy.optimize.minimize). In this blog post, we take an alternative approach and rely on linear algebra to find the best parameter estimates. This linear algebra approach to linear regression is also what is used under the hood when you call sklearn.linear_model.LinearRegression.[1]

[1] The implementation of sklearn.linear_model.LinearRegression is a little bit more intricate than the approach discussed here. Specifically, matrix factorization is used (e.g., QR-factorization) to prevent having to numerically invert matrices (which is numerically unstable, see, e.g., the Hilbert matrix). For the rest, the exact same approach applies.

LINEAR REGRESSION IN MATRIX FORM

Assuming for convenience that we have three observations (i.e., n=3), we write the linear regression model in matrix form as follows:

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}}_{\beta} + \underbrace{\begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix}}_{e} = X\beta + e$$

Note that the matrix-vector multiplication Xß results in

$$X\beta = \begin{bmatrix} \beta_0 + \beta_1 x_1 \\ \beta_0 + \beta_1 x_2 \\ \beta_0 + \beta_1 x_3 \end{bmatrix}$$

which is essentially just a compact way of writing the regression model.

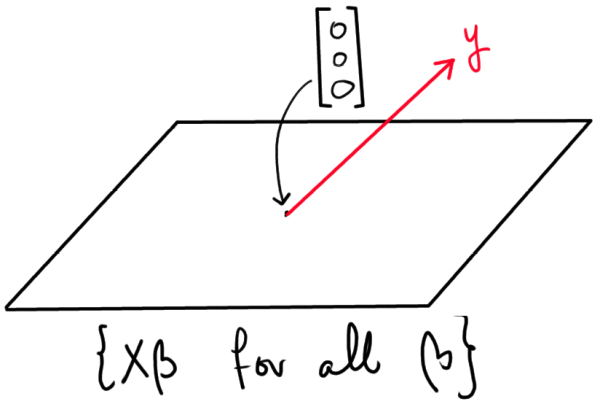
GEOMETRICAL REPRESENTATION OF LEAST SQUARES REGRESSION

The objective is to obtain an estimator ß^ such that y≈Xß^ (note that, usually, there is no ß^ such that y=Xß^; this only happens in situations that are unlikely to occur in practice).

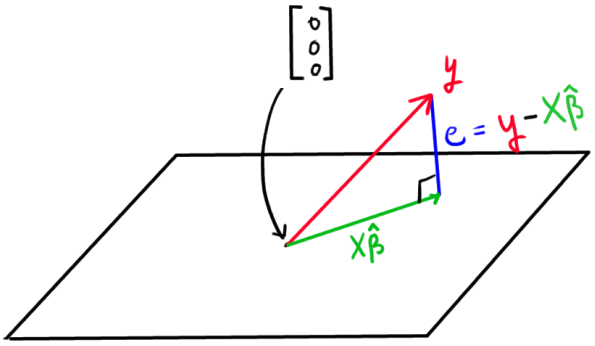
To represent the problem of estimating ß^ geometrically, observe that the set

$$\{X\hat{\beta} \text{ for all possible } \hat{\beta}\}$$

represents all the possible estimators for y^.
Now, imagine this set to be a plane in 3D space (think of it is a piece of paper that you hold in front of you). Note that y does not "live" in this plane, since that would imply there is a ß^ such that Xß^ =y.
All in all, we can represent the situation as follows:



Finding the best estimator ß^, now boils down to finding the point in the plane that is closest to y. Mathematically, this point corresponds with the ß^ such that the distance between Xß^ and y is minimized. In the following figure, this point point is represented by the green arc:



Namely, this is the point in the plane such that the error (e) is perpendicular to the plane. It is interesting to note that minimizing the distance between Xß^ and y means minimizing the norm of e (vector norms are used in linear algebra to give meaning to the notion of distance in higher dimensions than two):

$$\text{“norm of } e\text{”} = |e| = |y - X\hat{\beta}| = \sum_{i=1}^{n} \left(y_i - \left(\hat{\beta}_0 + \hat{\beta}_1 x_i\right)\right)^2$$

hence we are minimizing the mean squared error of the regression model!

ESTIMATING ß0 AND ß1

It remains to find a ß^ such that the vector e=y–Xß^ is perpendicular to the plane. Or, in linear algebra terminology: we are looking for the ß^ such that e is orthogonal to the span of X (orthogonality generalizes the notion of perpendicularity to higher dimensions).

In general, it holds that two vectors u anv v are orthogonal if $u^Tv=u_1v_1+...+u_nv_n=0$ (for example: u=(1,2) and v=(2,–1) are orthogonal). In this particular case, e is orthogonal to X if e is orthogonal to each of the columns of X.
This translates to the following condition:

$$(y - X\hat{\beta})^\top X = 0$$

By applying some linear algebra tricks (matrix multiplications and inversions), we find that:

$$X^\top \left(y - X\hat{\beta}\right) = 0 \Leftrightarrow$$
$$X^\top y - X^\top X\hat{\beta} = 0 \Leftrightarrow$$
$$X^\top y = X^\top X\hat{\beta} \Leftrightarrow$$
$$\left(X^\top X\right)^{-1} X^\top y = \hat{\beta}$$

Hence, $\hat{ß} = (X^TX)^{-1}X^Ty$ is the estimator we are after.

NUMERICAL EXAMPLE

Suppose we observe:

```
x = [1, 1.5, 6, 2, 3]
y = [4, 7, 12, 8, 7]
```

Then, to apply the results from this blog post, we first construct the matrix X:

```
X = np.asarray([np.ones(5), x]).T
print(X)
>>> [[1.  1. ]
>>>  [1.  1.5]
>>>  [1.  6. ]
>>>  [1.  2. ]
>>>  [1.  3. ]]
```

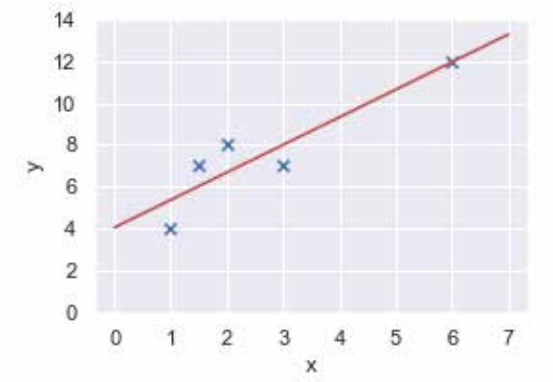
and then do the matrix computations[2]

```
from numpy.linalg import inv
beta_0, beta_1 = inv(X.T @ X) @ X.T @ y
print(beta_0, beta_1)
>>> (4.028481012658229, 1.3227848101265818)
```

which gives us our esimates. To illustrate these, results

```
x_lin_space = np.linspace(0, 7, 100)
y_hat = beta_0 + beta_1 * x_lin_space
plt.scatter(x, y, marker='x')
plt.plot(x_lin_space, y_hat, color='r')
```

which shows the fit of our model:



Although this blog post was written around a simple example with only one feature, all the results generalize without any difficulties to higher dimensions (i.e., more observations and more features).

If you have enjoyed this post, probably the [fast.ai course on computational linear algebra] is for you (it's free).

[2] This is where you would like to use matrix factorization to prevent having to compute $(X^TX)^{-1}$ directly; see also footnote 1.

ROEL BERTENS

GCP POWERED EV CHARGING

Are you considering to switch to electric for your next car but doubting the charging possibilities in your neighborhood? Well, I was. And I also just got certified as a Google Cloud Professional Data Engineer. Curious how I used GCP to answer my question? Read along.

COLLECT DATA

The first thing you need, to answer any question really, is data. So I set out to collect information about the usage of the electric charging stations in my neighborhood at home in Utrecht and around the GDD office in Amsterdam.

You can for example find the current available charging stations at NewMotion. You might have already guessed that I'm not going to collect this data manually. So I created a Cloud Function to collect and store data. In short, I wrote a simple collect function that requests the current status for the set of charging stations I'm interested in and which uploads the response as a blob on Google Cloud Storage. Given a dictionary of station names and identifiers this collect function looks as follows.

```
def collect(request):
    stations = {'station_name': 123456}
    for name, uid in stations.items():
        now = dt.datetime.now().strftime('%Y%m%d-%H%M%S')
        upload_blob(bucket_name_str='charge-stats',
                    station_status_str=get_station_status(uid),
                    destination_blob_name_str=f'{name}_{now}')
```

Where get_station_status simply sends a requests which returns some text data that we store in a bucket. We simply use Google Cloud Storage here because it is cheap. Also, the size of this data is not very big, so we won't need a more optimized data storage solution. We will be able to load it all in memory. The upload_blob function looks as follows.

```
def upload_blob(bucket_name_str, station_status_str, destination_blob_name_str):
    """Uploads data to the bucket."""
    storage_client = storage.Client()
    bucket = storage_client.get_bucket(bucket_name_str)
    blob = bucket.blob(destination_blob_name_str)
    blob.upload_from_string(station_status_str, content_type='text/plain', client=None,
    predefined_acl=None)
```

To create your own Cloud Function you simply paste the above python code in the inline editor in main.py after choosing Python 3.7 as Runtime. In requirement.txt you specify the packages needed, here python, pandas, pytz, requests, google-cloud-storage. And finaly you specify collect as the Function to execute.

SCHEDULING

Next step is to make sure that this Cloud Function runs on a fixed interval to actually start collecting data. Cloud Scheduler to the rescue. It is very easy, you only have to specify the frequency, choose HTTP as target and specify the URL of your Cloud Function.

The Cloud Storage bucket I created is filling up automatically with many blobs containing information about the usage of the charging stations I'm interested in.

PREPARE DATA

The storage bucket now contains many time-stamped blobs which I need to put together to create a data set for exploration. To avoid setting up policies and access rights I use Google's AI Platform to spin up a notebook in which I can quickly load the data from Google Storage and immediately start to explore it.

After collecting data each minute for more than two months for 14 charging stations I've collected almost 1.5 million blobs. Each blob contains json data from which I select the station name, a unique identifier for a pole at the station (a station can have more poles), the status of the pole and the timestamp.

Putting everything together my data set looks as follows.

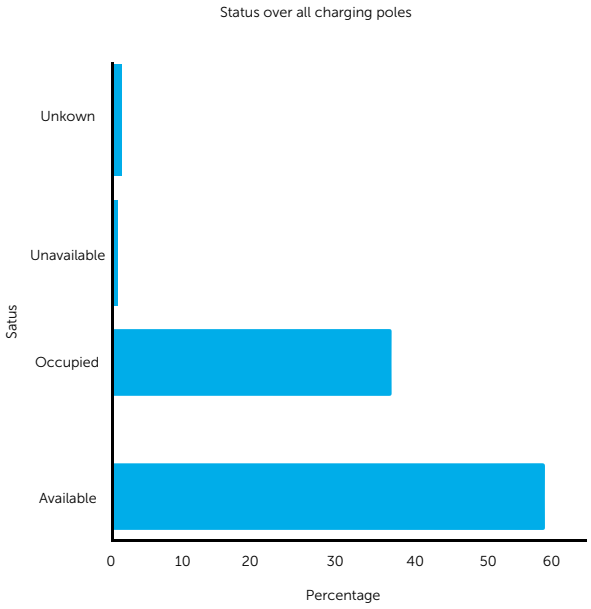
	ts	station name	uid	status
	2019-08-24 07:12:12+02:00	UT - thorbeckelaan	1309184	Occupied
	2019-08-24 07:13:12+02:00	UT - thorbeckelaan	1309184	Occupied
	2019-08-24 07:14:12+02:00	UT - thorbeckelaan	1309184	Occupied
	2019-08-24 07:15:12+02:00	UT - thorbeckelaan	1309184	Occupied
	2019-08-24 07:16:12+02:00	UT - thorbeckelaan	1309184	Occupied

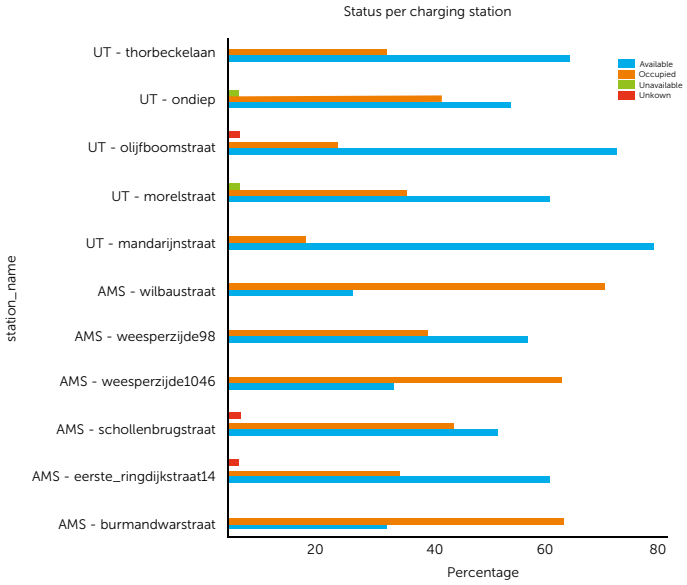
EXPLORE DATA

Putting everything together we have information about the usage of 5 charging station in Utrecht and 9 in Amsterdam for about 2 months. After some quick counts I found out that for 4 of the stations in Amsterdam something weird is going on. There were only very few state changes recorded for these stations, which is unexpected for their location. I suspect that either something went wrong in the data collection, maybe the stations weren't reachable because of construction or something else is going on. Either way, their usage was so unexpected/low and different from the rest that I removed these stations for further analysis.
Time for some questions!

HOW OFTEN ARE THE CHARGING POLES USED?

Looking at all charging poles in scope we see that they are only occupied for about 39 percent of the time. There is quite some variation between stations though.

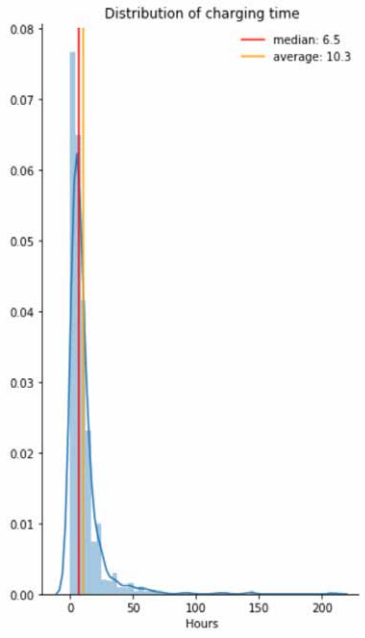


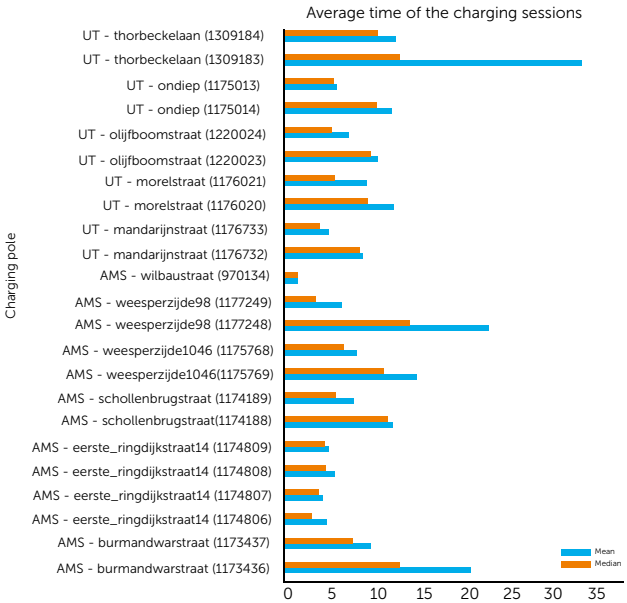


HOW LONG ARE THE CHARGING SESSIONS?

The average charging session is about 10 hours (median 7 hours) and again there is quite some variation between different poles. Most notably, the different poles at the same station also differ quite much.
The clearest example to see this effect is AMS - weesperzijde 98, which has the largest average on one pole and one of the smallest on the other.

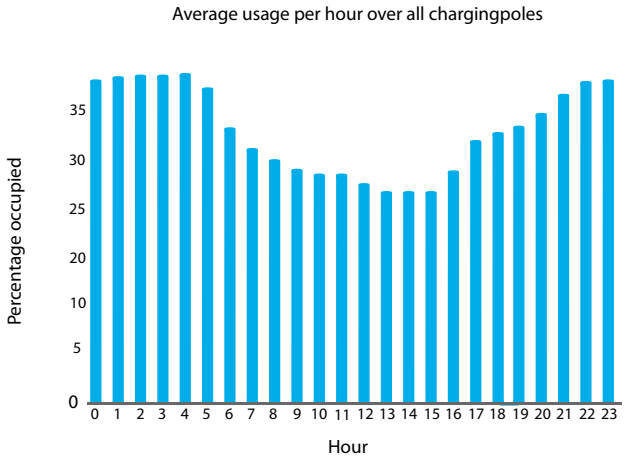
Note: a charge session is the time that a vehicle is connected to the pole (so it can be that the battery is already full).



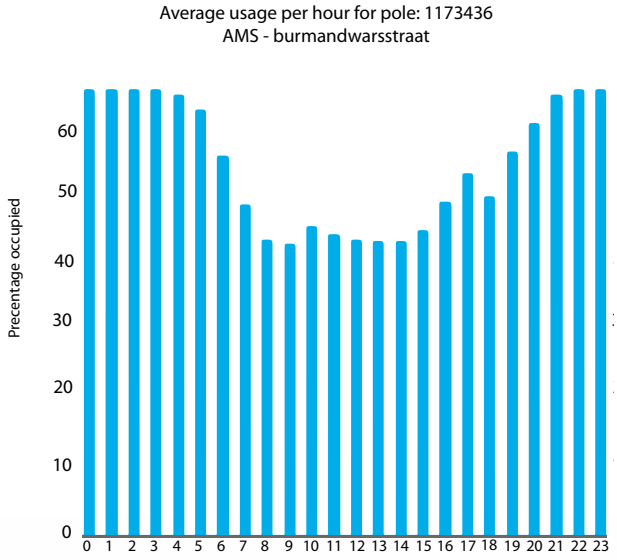


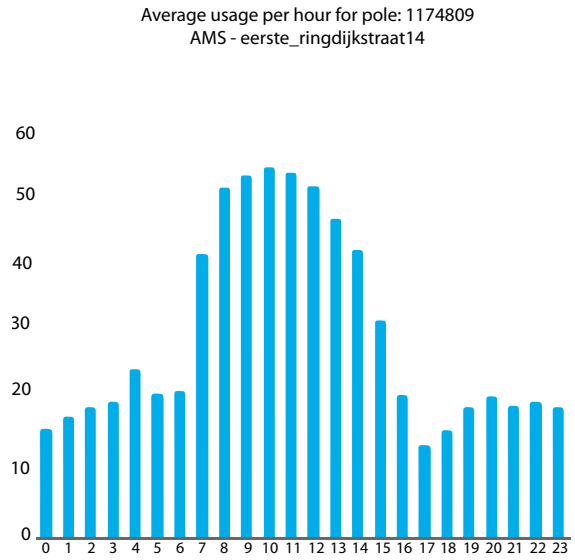
WHAT TIME OF THE DAY ARE PEOPLE CHARGING?

It is clear to see that most people charge overnight and start charging again when they get back from work, since the usage increases again from four o'clock in the afternoon.



Zooming in on specific poles we clearly see two different patterns. The first pattern is the one described above; charging dip during working hours. The second patterns is the inverse of the first; charging peak during working hours.





TO BUY OR NOT TO BUY

I found that there are more than enough charging possibilities for me both around work and at home. Especially in Utrecht the charging poles are still unoccupied most of the time.

I will be monitoring the usage to see if electric cars become more popular and the usage increases. For now I would conclude that it is safe to hop on the EV train.

Feel free to reach out if you have any questions regarding the code or technology used.

ROGIER VAN DER GEER

DATA DRIVEN BOARD GAME DESIGN

DESIGNING OUR OWN BOARD GAME

When designing a board game, it takes a lot of finetuning to get the balance right and make the game fun to play. This finetuning in turn requires you to play endless iterations of the game, or does it?

THE AI USE CASE GAME

A while back my colleague Walter called me up, saying he was looking for an expert in board games to help him out design our own game. While I do like to play a board game once in a while I am by no means an expert, but I like to take on a challenge so I decided to see if I could help out.

Walter already had quite a clear idea of what he wanted to make: a board game where you take turns to walk around a monolopy-style track and get to try to complete AI use cases. We discussed for a while, and quickly realised that we would need to start playing to figure out which concepts do or do not work in a board game. So we drew the board on a sheet of paper, used python to simulate dice rolls and took another sheet of paper to keep track of our balances as we did not have any dice nor game money at the ready.

We quickly realised that playing like this wasn't particularly fun. And, if we were to create a well-balanced game we would need to play a lot of games. If only there was a way to automate that...

So, I decided to write a simulation that could play the game for us. But before we dive into the simulation, let's have a look at the end result.

THE GAME

In the game, you lead a team of data scientists and engineers. Your goal is to create as much business value as possible for your company while you and competitors each finish up to three use cases.
The game is played on a board.

PHASES OF THE GAME

All players start the game in the ideation field, where the goal is to come up with a use case. While in this phase, you may pick a new use case card every turn, and add it to your backlog. After picking a use case card you may decide to start developing it, in which case you move your pawn to the "start" of the ideation phase. You will have to successfully pass the *infrastructure, data & ETL, modelling* and *productionizing phases* in order to complete the use case.

BUILDING A TEAM

When you are developing a use case, your turn starts with a die roll. Then, you move your pawn forward by the number of eyes on the die, offset by your *handicap*. Your handicap is your team size minus the desired team size as provided on the use case card: this means that if your team size is too small, you may actually move backward! If your team size is more than three people short, you will never be able to complete a use case.

If you come across one of the yellow line named "team", then you must take a team card from the stack. These team cards affect your team size, and may provide you with a new team member, may result in you losing a team member, or offer you one or more team members in return for a fee.
All players start with two team members, and growing your team during the game is essential!

But a large team also offers a disadvantage: team members are expensive. You will need to pay your team members' salaries at the end of every turn, even during ideation. If you ever run out of money, you will need to take a *reorganisation* card from the stack. This card will provide you with some extra budget or exempt you from having to pay the salaries for a turn, but may cost you business value or a team member.

BUDGET

Each use case comes with a budget; you will receive part of the budget when starting the use case, and the remainder once you have passed the green dashed line named "budget" halfway the board.

ACTION CARDS

If you land your pawn on one of the gray fields with an icon, then you must take an action card of the phase you are in. These action card may provide you with a benefit, but they may also give you a disadvantage. This may come in the form of budget, progress (fields on the board), a number of turns to skip, or in extreme cases force you to abandon the use case.

TAKING THE USE CASE INTO PRODUCTION

Once you pass the finish line you have completed the use case and may collect the business value associated to it. The next turn you may choose to start ideation and draw a new use case, or start developing a use case that is already in your backlog.

The game ends when a player has finished his third use case; the other players then still get to finish the round. The player who created the most value wins!

THE SIMULATION

When you make a simulation for any kind of game, the implementation of the game rules is usually the easy part. Modeling the behaviour of players is much more complicated: they don't follow strict rules when making decisions. And if you do make a set of rules that the players use to make their decisions, then you can have a lot of fun optimising these rules. A few years ago, I did exactly that for the Risk boardgame.

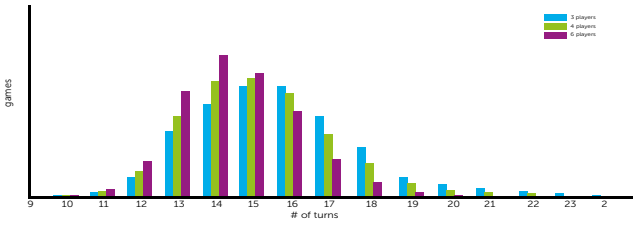
For this new game the optimal strategy wasn't my focus, but it was the game itself. So instead of spending a lot of time on the decisions of the players, I spent more time making sure the game rules were easily configurable. I made some decisions on how players react to certain situations, and I assumed that the game would only become better when people actually put thought into it. That may sound like a dangerous assumption, but since the chance element of the game is fairly heavy I think that the strategy component is not so important. If you are interested in the implementation, have a look here.

OPTIMIZING THE GAME

Once the simulation was (mostly) finished, we could start optimizing the game. Being a data scientist, I wanted to define a loss function and then let some algorithm find the most optimal game. But it turns out to be difficult to capture the notion of a "fun" game in a loss function ☺. So we went with doing the optimization ourselves, looking at multiple aspects of the game and going with our gut feeling of what is a fun game.

DURATION OF THE GAME

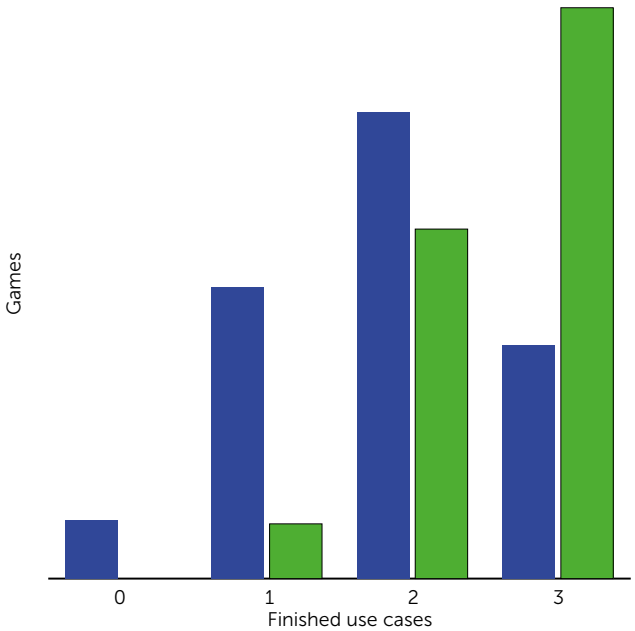
Perhaps the most important parameter to optimise was the playing time. No one likes a game that takes half a day or one that is finished in a single turn. So we played around with the number of fields on the board, the number of use cases to complete and the action cards until we were happy with the result.

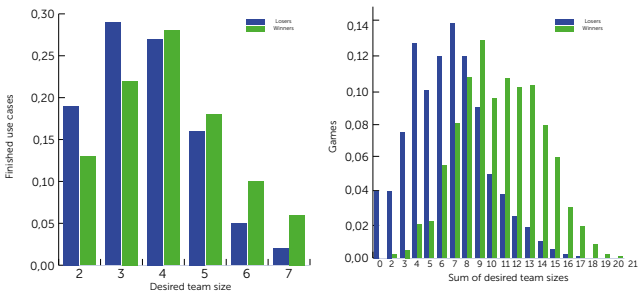


Of course the expected number of turns varies with the number of players: the more players, the more likely one of the players will be done after a given number of turns. In the end, we settled for about 15 turns per game, which would make the game playable in well under an hour.

USE CASES

Next up were the use cases, in which we needed to balance the desired team size and the resulting value. We wanted the ideal path for a player to be to first develop a simple use case (of which the desired team size is 2 or 3), then a moderately complicated (4-5), and finally a complicated use case (6+). If we wouldn't balance the business value well, it could end up being a better strategy to finish three simple uses cases as quickly as possible.





Above you see the results after balancing: on the left we see that winners have often completed three use cases, but it is also possible to win with only two use cases. That is great: this means players have to balance quickly finishing three use cases versus finishing some with more value.

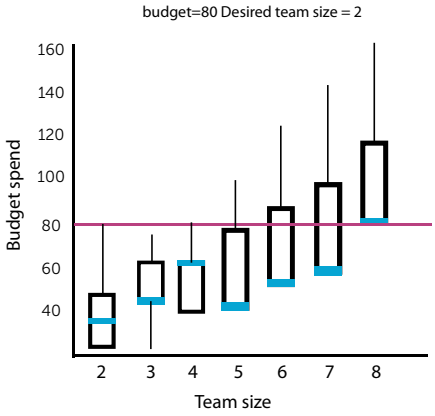
In the middle plot we see that winners typically finish use cases with a higher desired team size, while on the right we see that these winners typically finished use cases with a sum of desired team size of between 8 and 13. If you manage to finish one use case in each of the three categories you would end up with a sum of 12+, which practically means that you won. That is exactly what we were aiming for!

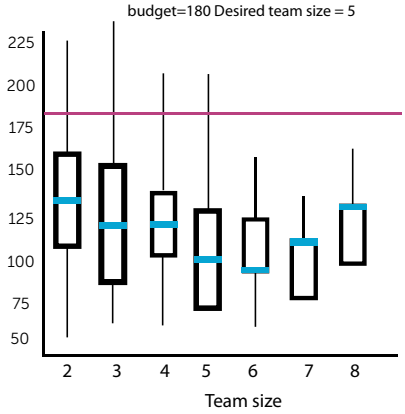
BUDGET

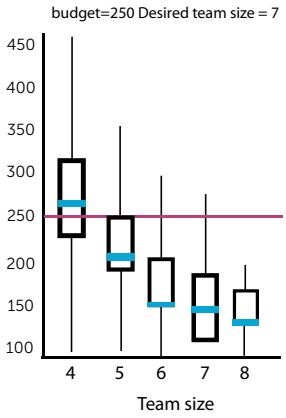
Also important are the budgets the use cases provide and the budget that players start with. We want it to be fairly doable to finish the game without taking any reorganisation cards, but it shouldn't be impossible to run out of budget either.

So we had a look at each use case and the expected amount of budget needed to complete it. This, of course, depends on the team size: the larger the team, the faster you move but the more expenses you have. Below are a few examples, ranging from very simple to very complicated. We've plotted the total spent budget while completing the use case for each possible team size.

As you can see, it is very possible to complete each use case as long as your team size is in the right ballpark. If your team is much too small or much too large, your budget may run out.







FINALIZING THE DESIGN

Of course, the rules and the balance of the game isn't everything that you need. Walter did a great job with the design of the game, which is equally important because no one likes to play a game that is visually unappealing. By now we have several copies of the game, and we've had people play it with us at several occasions. So far the reception has been great... perhaps I should give it a try once, as I haven't played the game myself yet. But my computer has had its fair share with at least a million games.

ADVANCE YOUR DATA & AI SKILLS

DATA & AI TRAINING GUIDE 2020

DOWNLOAD THE GODATADRIVEN DATA & AI TRAINING BROCHURE 2020 FOR A COMPLETE OVERVIEW OF ALL AVAILABLE DATA SCIENCE, DATA ENGINEERING, DATA PRODUCT OWNER, AND EXECUTIVE TRAINING COURSES.

godatadriven.com/training