CRYSTAL GAZERS

ML algorithm for language modeling

EXTERNAL LINKS: GitHub link

TEAM MEMBERS:

Surname	Name	UPC email	GitHub link
Sirera	Miquel	miquel.sirera@estudaintat.upc.edu	https://github.com/mqsirera
Guasch	Jaume	jaume.guasch@estudauntat.upc.edu	https://github.com/jguaschmarti
Domingo	Álvaro	alvaro.domingo@estudiantat.upc.edu	https://github.com/alvarodr21
Zyatyugina	Maria	maria.zyatyugina@estudiantat.upc.edu	https://github.com/mashazya

maria.zyatyugina@estudiantat.upc.edu - contact email

Index


1

INTRODUCTION

Goal of the project

The main goal of this project is to apply good practices of definition of a data science project and present the results of the used techniques. Crystal Gazers is an organization that predicts a word given a sequence of other words as context around it. The NLP processing will be done by training a Transformer model with a large dataset based on Wikipedia articles.

The project goal is to successfully train the model and obtain an accuracy superior to the defined threshold. The default metric to evaluate the performance of the model is accuracy. The model will be considered correctly trained when an accuracy of at least 35% is achieved.

Model card

The original Markdown version of Model card can be found in the above linked GitHub repository

language	license	dataset	tags	metrics
ca	apache-2.0	Viquipèdia	prediction	accuracy
			NLP	CrossEntropyLoss
			text	CO2 emission
			transformer	
			CrossEntropy	
			text generation	

This universitary project aims to train a ML model with the Wikipedia dataset in order to predict the hidden word given a sequence of context words at both sides. The model will output a word for every input given. The model is executed and developed in Kaggle.

Model description

To achieve the desired prediction we use a Transformer model with two layers. A transformer is a model that adopts the mechanism of self-attention, deferentially weighting the significance of each part of the input data. The model is trained on text data which was originally created by humans.

The model file gives the possibility to introduce multihead attention. At the end, there is an accuracy comparison based on the performance of both methods.

The training of the model is based on Cross Entropy Loss and the parameters are updated by Adam optimizer, which is chosen to be the optimum empirically. The script is developed in Python using `torch , numpy and pandas` libraries among others.

The model was created by Jose A.R. Fonollosa

Dataset

The Wikipedia dataset is a collection of scraped Wikipedia pages. The dataset is defined in catalan language, thus the model is trained to recognize input exclusively in catalan. There are 2 versions of the dataset, ca-2 and ca-100, with the same obtention method but the latter collecting more data from more articles. The splits used to train this model are the following:

	train	validation	test
ca-2	11MB	1.1MB	1.1MB
ca-100	505MB	1.1MB	1.1MB

The dataset can be found here.

Results

ca-2:

Single-head:

Accuracy: 35.6%

CrossEntropyLoss: 4.33

CO2 emissions: 0.00333

Multi-head:

Accuracy: 36.4%

CrossEntropyLoss: 4.29

CO2 emissions: 0.00342

ca-100:

Single-head:

Accuracy: 46.4%

CrossEntropyLoss: 3.13

CO2 emissions: 0.1543

Dataset card

The original Markdown version of Dataset card can be found in the above linked GitHub repository

annotations creators	language	language creators	license	multilinguality
notation	ca	crowdsourced	cc-by-sa-3.0	monolingual

pretty name	size categories	source datasets	tags	task categories	task ids
Viquipèdia	unknown	original	wikipedia catalan text articles education	text prediction	langua modeli

Dataset Description

- Link: https://www.kaggle.com/datasets/jarfo1/viquipdia

- Main author: José Andrés Rodriguez Fonollosa

Dataset summary

The Wikipedia dataset is a collection of scraped Wikipedia pages. The dataset is defined in catalan language, thus the model will be trained to recognize input exclusively in catalan.

Supported tasks

Text prediction

Languages

Catalan

Dataset structure

```
{
  'ca-2': [
       'ca.wiki.test.tokens,
       'ca.wiki.train.tokens,
       'ca.wiki.valid.tokens]
  'ca-100': [
       'ca.wiki.test.tokens',
       'ca.wiki.train.tokens',
       'ca.wiki.valid.tokens']
}
```

Data fields

Plain text

Data splits

Reporting the sizes of the .token.nopunct files

	train	validation	test
ca-2	11MB	1.1MB	1.1MB
ca-100	505MB	1.1MB	1.1MB

Dataset creation

The dataset has been created by scraping all articles in catalan from Wikipedia. The original text has been crowdsourced by thousands of anonymous volunteers.

Annotations

The dataset has no annotations as the files only contain the plain text from the articles. Derived files have been created by preprocessing the original plain text files.

Considerations for Using the Data

The content has not been reviewed nor filtered by the authors of the dataset. It is supposed to have been reviewed by the Wikipedia community of volunteers to guarantee its veracity, lack of sensible or private content, and neutrality with respect to possible biases. Nonetheless, there's no guarantee that all 100% of the text has surpassed this review, or an indication of what text has been reviewed or not.

Milestone 1

The first step in developing this project was the election of the model and dataset that will be the base of this project. The details on the model and the dataset can be found in the Data and Model cards, correspondingly. The Data and Model cards were developed during this first week of work. The template and all the information on the structure of Data and Models cards was extracted from HuggingFace.

The chosen dataset and model were already ready for use and developed by Jose A.R. Fonollosa. The work consisted in collecting the required information on the dataset and the model to add to the corresponding cards.

The following challenge was to pre-define the communication method, the cloud provider to store the data and establish the workflow pipeline and the workspace to store all the progress and models.

Crystal Gazers communication is done via Slack, as an easy communication tool that allows the creation of public and private channels accessible for the participants of the organization.

All work progress will be stored in GitHub in a public repository called TAED2_CrystalGazers defined inside the Crystal Gazers organization. All teammates are participants of the organization and can make edits to its structure. The changes into the main branch are done via pull requests. More about the GitHub organization can be found in the GitHub section.

The chosen cloud provider is AWS that will also be used as a remote storage for the dataset and version control.

Milestone 2

The second week of the project started with an investigation of engineering tools that facilitate the organization and structuring of the work.

In the second Milestone the CookieCutter and DVC were installed and implemented.

CookieCutter allowed the easy re-organization of the repository for its easy maintenance. The project repository structure after using the data science template is the following:

```
├── LICENSE
├── Makefile           <- Makefile with commands like `make data` or `make train`
├── README.md          <- The top-level README for developers using this project.
├── data
│   ├── external       <- Data from third party sources.
│   ├── interim        <- Intermediate data that has been transformed.
│   ├── processed      <- The final, canonical data sets for modeling.
│   └── raw            <- The original, immutable data dump.
│
├── docs               <- A default Sphinx project; see sphinx-doc.org for details
│
├── models             <- Trained and serialized models, model predictions, or model summaries
│
├── notebooks          <- Jupyter notebooks. Naming convention is a number (for ordering),
│                         the creator's initials, and a short `-` delimited description, e.g.
│                         `1.0-jqp-initial-data-exploration`.
│
├── references         <- Data dictionaries, manuals, and all other explanatory materials.
│
├── reports            <- Generated analysis as HTML, PDF, LaTeX, etc.
│   └── figures        <- Generated graphics and figures to be used in reporting
│
├── requirements.txt   <- The requirements file for reproducing the analysis environment, e.g.
│                         generated with `pip freeze > requirements.txt`
│
├── setup.py           <- makes project pip installable (pip install -e .) so src can be imported
├── src                <- Source code for use in this project.
│   ├── __init__.py    <- Makes src a Python module
│   │
│   ├── data           <- Scripts to download or generate data
│   │   └── make_dataset.py
│   │
│   ├── features       <- Scripts to turn raw data into features for modeling
│   │   └── build_features.py
│   │
│   ├── models         <- Scripts to train models and then use trained models to make
│   │   │                 predictions
│   │   ├── predict_model.py
│   │   └── train_model.py
│   │
│   └── visualization  <- Scripts to create exploratory and results oriented visualizations
│       └── visualize.py
│
└── tox.ini            <- tox file with settings for running tox; see tox.readthedocs.io
```

All the details on GitHub flow, the DVC implementation and use and the implementation of CookieCutter template can be found in the corresponding sections.

MLFlow

Another useful tool that can add quality to the model training is adding the MLFlow library to the training code of the model. MLFlow saves given parameters of the trained model. Those parameters, not only will give a better description of the whole structure, but facilitate the re-training, if needed.

To use MLFlow the following script must be added to the code:

```
with
mlflow.
start_r
un():

            train_accuracy = []

            wiki_accuracy = []

            valid_accuracy = []
```

```
    mlflow.log_param("num_heads",
hyperparams['num_heads'])

    mlflow.log_param("epochs", hyperparams['epochs'])

    mlflow.log_param("embedding_dim",
hyperparams['embedding_dim'])

    mlflow.log_param("window_size",
hyperparams['window_size'])

    mlflow.log_param("batch_size",
hyperparams['batch_size'])

    mlflow.log_param("dataset_version",
hyperparams['DATASET_VERSION'])

    for epoch in range(hyperparams['epochs']):

        acc, loss = train(model, criterion, optimizer,
data[0][0], data[0][1], hyperparams['batch_size'],
device, log=True)

        train_accuracy.append(acc)

        mlflow.log_metric("training_acc", acc)

        mlflow.log_metric("training_loss", loss)

        print(f'| epoch {epoch:03d} | train
accuracy={acc:.1f}%, train loss={loss:.2f}')

        acc, loss = validate(model, criterion, data[1][0],
data[1][1], hyperparams['batch_size'], device)

        wiki_accuracy.append(acc)

        mlflow.log_metric("valid_acc", acc)

        mlflow.log_metric("valid_loss", loss)

        print(f'| epoch {epoch:03d} | valid
accuracy={acc:.1f}%, valid loss={loss:.2f} (wikipedia)')



    mlflow.pytorch.save_model(model,
f"./artifacts/{hyperparams['modelname']}")
```

The original code can be found in the GitHub repository.

By using MLFlow it is possible to track all the experiments within the same model (as for example changing the parameters and seeing the effect on the defined metrics) or to compare different models. Crystal Gazers experimented with a different number of heads in the Transformer model, which implied the use of two different attention types. More about the results of the experiment and the structure of the solution can be found in the corresponding section.

In our project, we have logged all the parameters that can have any impact on the performance of our model, that we evaluate through accuracy. For instance, these will be: the number of heads we will be using for multihead attention, training epochs, the embedding dimension of the word vectors, the window size for context words and training batch size. As we will need the accuracy of the model for each of these runs, we also logged this metric and even kept track of the value through the epochs in the training. In addition, we saved the value of our loss function. Finally, we also saved the weights of the model that has been trained for further use.

When the models are trained it is possible to see the output of the MLFlow. It generates the following dashboard for the two trained models:

Metrics				Parameters			
↑ training_acc	training_loss	valid_acc	valid_loss	batch_size	embedding_dir	epochs	num_hea
39.27	3.269	36.01	4.307	2048	256	4	1
40.34	3.187	36.15	4.288	2048	256	4	4

Milestone 3

The third Milestone introduces the quality control of the work done.

As every execution of any code produces an impact on the environment it is very important to track the emissions and be aware not to generate unnecessary executions. Crystal Gazers introduced the carbon emissions calculator into the code.

By importing the corresponding Python library and adding the following lines to the training code of the model it is possible to track and monitor the produced CO2 emissions.

```
from codecarbon import EmissionTracker

tracker = EmissionTracker()

tracker.start()

tracker.stop()
```

The obtained results of the carbon emission of the tested models were the following ones:

```
ca-2 local 0.0012 KgCO2

ca-2 remote 0.0033 KgCO2

ca-100 remote 0.154 KgCO2
```

As it may be seen from the table above, three models were tested on two different sizes of the training set and using different devices to train the models (local and remote). The increase in the amount of carbon emissions was expected, as the size of the set affects the training time – the larger is the set – the higher is the training time. On the other hand, there are more emissions produced remotely as the computer power is much higher and, thus, requires more energy.

However, as the presented results does not suppose a huge difference between each other and the ecological impact can be considered assumable the chosen model for further experiments is the ca100 trained on remote.

Furthermore, in the 3rd milestone, code quality testing was introduced. Crystal Gazers used **Pylint** and **Pynblint** to test the quality of the python scripts and notebooks, respectively. When using it in a script, it can be seen which rules our code is not being followed and may be corrected. Pylint allows you to generate a configuration file which can disable rules that aren't useful and set some useful parameters such as how many arguments CrystalGazers classes and functions have.

In a similar way the notebooks have been corrected using Pynblint, (that for example checks the proportion between markdown and code cells, that the headers hierarchy is correct or that you can execute it top to bottom). The main difference between them, apart from the type of file they are applied to, is that Pylint gives an overall scoring of the notebook, and Pynblint does not.

To guarantee that not only the code follows desired standards, but also the dataset contains data in the desired format or has a desired structure, Crystal Gazers use the **Great Expectations** tool to test the structure of the textual data. As the dataset is a large collection of textual data, the only quality testing on the dataset that was performed is to check if the data frame containing the tokens generated from textual data consists uniquely of integer data.

The next step after guaranteeing code quality is to test that the produced output matches the desired. To test the code output Crystal Gazers used **Pytest**. The established tests had multiple accepted outputs. The tests were implemented in order to see if small changes in the input did not affect the output and big changes in the input affected the output.

Milestone 4

The last milestone contained the most important parts of this project. The deployment of the model (the design of the ML system), the implementation of the API and testing of the generated endpoint.

As previously mentioned, the platform for deployment and implementation will be Amazon S3 bucket

Teammates evaluation

The tasks done on this project are distributed evenly among all team members. The work organization and communication is working well.


2

METHODOLOGY

To follow a good project structure and maintain a considered organization among the team members and their contribution to the development of the project some software and organizational tools have been selected.

GitHub

First of all a GitHub repository inside the Crystal Gazers organization has been created. All the code and the corresponding archives are stored there. GitHub allows developers to develop different versions of the code and then merge them with the main branch.

The repository was created from the GitHub web and then cloned from local terminal as an environment in a given directory using the following command:

```
git clone
https://github.com/CrystalGazers/TAED2_CrystalGazers.git
```

All edits done locally can be pushed by using the following commands:

```
git pull

git add -- all

git commit -m "Update README.md"

git push
```

In order to do the process correctly the main branch only gets updated when the already developed and tested code is pushed into the main branch from an auxiliary. In order to make the merging it is necessary to make a pull request and for an admin to accept it.

A new branch is created by executing:

```
git checkout -b branch-name
```

To access the branch locally the following command is executed:

```
git checkout branch-name
```

When the branch code is ready to be uploaded to the main branch, first, the pwd must be the main branch. To checkout into main branch and then merge a branch into it the following commands are executed:

```
git checkout master

git merge branch-name
```

Every technical GitHub template has a requirements.txt file which contains the necessary packages to be able to run the code correctly. To automatically generate the file from the virtual environment that contains the packages installed for the project it is necessary to execute the following command from the environment directory:

```
pip freeze > requirements.txt
```

CookieCutter

A very useful tool to use after creating the GitHub repository is to use CookieCutter for data science. The GitHub repository has a predefined organized structure that allows any user that decides to access it to easily find the searched file. All files are distributed following a structure that can be used for any technical project. In our case a development of a ML model. The following commands must be executed in the local directory where the repository is stored

```
pip install cookiecutter

cookiecutter https://github.com/drivendata/cookiecutter-data-
science
```

After introducing the command, the program proceeds to ask some questions to name the repository as the name of the organization, name of the repository, license, programming language etc.

DVC

Data Version Control is a data and ML experiment management tool that takes advantage of the existing engineering toolset as GitHub or S3. Crystal Gazers will track the data versioning in S3, while maintaining a link in GitHub to easily access the data.

To install the DVC tool and create a url access in GitHub in the corresponding dvc directory in the repository in GitHub the following commands are used:

```
pip install dvc

dvc init

git add .dvc

git commit -m "Initialize DVC"
```

In order to add new files into the S3 bucket and add a pointer into GitHub repository the following commands must be used:

```
dvc remote add s3cache s3://crystal-gazers/cache

dvc config cache.s3 s3cache
```

```
dvc add --external s3://crystal-gazers/archive/
```

For the changes to be visible they must be pushed into the GitHub repository following the commands defined in the corresponding section.

Although the actual data versioning is not done in GitHub it is important to provide easy access to any additional services for two reasons. First of all to have a summary of all the used tools and services in one place. The last, but not least is to achieve a defined and clean structure of the project following the previously defined structure of a ML project.

Once created the GitHub pointer the dvc must be configured in AWS. To install the AWS command tool execute the following:

```
sudo apt install awscli

aws configure --profile iam-alvaro

#login information

AWS Access Key ID [...]

AWS Secret Access Key [...]
```

Every object in Amazon S3 is stored in a bucket. Once an AWS storage bucket is created and the command line is installed to allow tracking existing data on the external location execute the following line . This produces a .dvc file with an external URL or path in its outs field.

```
dvc add -d storage s3://crystal-gazers/archive/
```

As the used dataset is ready to use as an input to the model, the DVC is not a required tool in this particular project. However, it is a very useful tool in case of cleaning or changing the data structure in other projects or model development.

Pylint

This useful tool checks that the code follows the correct standards and can be considered of high quality. To follow only those standards, in which Crystal Gazers were interested in, a special file is used to disable unneeded warnings. The file is created by using pylint `--generate-rcfile` and disactivating some of the triggers. The new file

must be saved under `~/.pylintrc` name and will automatically be used when calling the tool.

To test the code quality the following line is executed:

```
pylint filename.py
```

The example output of the file that follows the quality standarts is the following one:

```
--------------------------------------------------------------------
---

Your code has been rated at 10.00/10 (previous run: 10.00/10,
+0.00)
```

Pynblint

Following the same structure as the previous mentioned tool, Pynblint checks that the notebook follows the standards of quality and is easy to be interpreted for its user.

Although, Pynblint does not offer the option to deactivate the unnecessary warnings, or those that are not apply to Crystal Gazers's code. As the model used to predict the context word implies a definition of a Transformer class, there is no way of correcting the triggering of warning that indicates that the cell containing the corresponding code is too long. Pynblint checks the format of the notebook for it to be easily interpretable for any user, in particular, it checks the hierarchy of headers, the proportion between markdown and code cells, and that the notebook is executable from top to bottom.

After correcting the format of the notebooks containing alpha versions of the transformer model and the texts pre-processing algorithms the generated output is the following one:

```
NOTEBOOK: text-preprocessing.ipynb

  PATH: notebooks/text-preprocessing.ipynb

STATISTICS┌──────── Cells ────────┐  ┌── Markdown usage ──┐  ┌─ Code
modularization ──┐

|            | |          | |          |

| Total cells: 27  | | Markdown titles: 13 | | Number of functions:
7 |

| Code cells: 15   | | Markdown lines: 12 | | Number of classes:
2  |
```

```
| Markdown cells: 12 | |          | |           |
| Raw cells: 0       |                          └──────────────────────┘
└────────────────────────────┘

|             |

└─────────────────────┘

LINTING RESULTS

(cell-too-long)

   One or more code cells in this notebook are too long (i.e.,
they exceed the fixed threshold of

   30 lines).

   Recommendation: Consider consolidating your code outside the
notebook by moving utility

   functions to a structured and tested codebase.

   Use notebooks to display results, not to compute them.

   Affected cells: indexes[10, 19]


   In [12]: | def prepare_dataset(params):
```

WARNINGS TO BE IGNORED AS CANNOT BE SOLVED

Pytest

Pytest is a model testing tool that can easily try if the model returns the expected output. Pytest can be used both, for testing the local model and for testing the remote endpoint. First the local model was tested.

For an input of: ["el", "noi", "està", "de", "sa", "nòvia"] the expected output was ['enamorat', 'cansat', 'embarassat', 'fugint']

Changing slightly the input the output is expected not to change. Thus, for an input: ["el", "noi", "està", "del", "seu", "nòvio"] the expected output was ['enamorat', 'cansat', 'embarassat', 'fugint']

The last test was predicting a whole new sentence and seeing if the output makes sense. For an input ["la", "jugadora", "de", "va", "guanyar", "molt"] the expected output was ['voleibol', 'futbol', 'tenis', 'equip'].

All three tests passed.

The following step was to see if the inference call to the endpoint containing the deployed model returned the expected output (context word) and if the status code was equal to 200, which corresponded with the successful execution of the algorithm.

To use pytest the following line needs to be executed:

pytest filename.py

Where filename contains the specific test class definition in order to perform the desired tests. All documents can be found in the Crystal Gazers repository.

Great Expectations

The great expectations tool can only be applied to dataframes, thus CrystalGazers checks if the tokenized matrix generated from textual data consists uniquely of integer data. Each column of the dataframe generates a separate dictionary output. All dictionaries are transformed and saved as a unique json file are joined at the end. The json file contains a collection of metrics. The most important one is called success, which equals TRUE if the expectation is fulfilled. In this case, all the expectations contain the success field as TRUE.


3

TECHNICAL DESCRIPTION

Architecture

The pipeline of the word prediction is the following:

- Load 3 datasets from S3 (pre-partitioned data - training, validation and test datasets)
- Train the model
- Save the parameters and the model for future use
- Test the trained model with the corresponding dataset
- Extract metrics and evaluate performance
- Select a model
- Do the inference with the selected model

In order to achieve the best possible prediction it is necessary to choose the best model. Crystal Gazers developed two variations of the Transformer model, in particular, with a single head attention and multihead attention. The performance of both models will be evaluated by two metrics: accuracy and CO2 emissions produced in training.

In the Transformer.py file the definition of the Transformer class can be found. To differentiate between the single and multi-head attention the following parameter was defined:

```
self.multihead_attn = nn.MultiheadAttention(embed_dim, num_heads,
bias=bias, batch_first=True)
```

where the num_heads equal 1 would execute a single head attention algorithm and a value greater than one would execute a multi-head attention algorithm.

The accuracy metric establishes the prediction performance of a model. The CO2 emission quantity takes into account the ecological factor of executing a large model.

Self-Attention

The Transformer model with 2 two layers and 1 head (single-head) attention algorithm achieves an accuracy of 39.27%. Which is a result that satisfies the success criteria of the project.

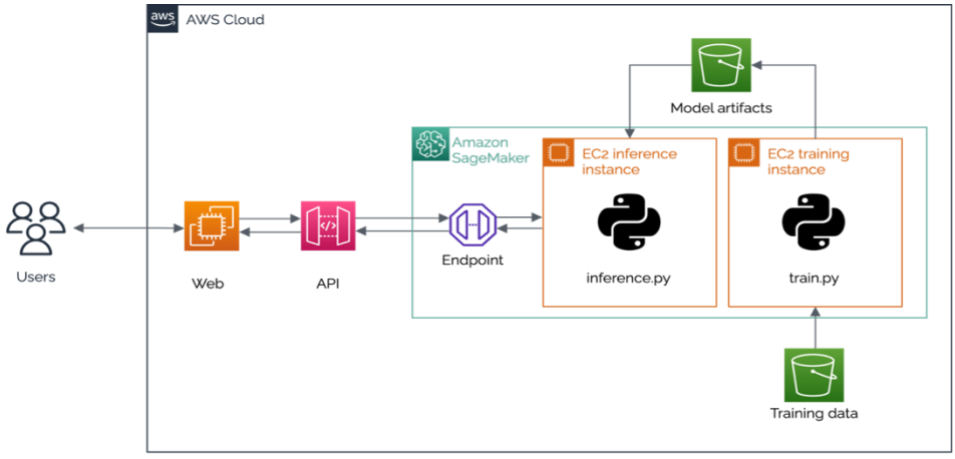
Multihead self-Attention

The Transformer model with 2 two layers and 4 heads (multi-head) attention algorithm achieves an accuracy of 40.34%. Which is a result that also satisfies the success criteria of the project.

We can see that both models have a very similar accuracy and a similar execution time. Moreover, checking the emission levels we observed that both models have a very similar level of emissions and consumed energy.

ML System design

We decided to integrate the whole deployment into the AWS Cloud. To do so, we designed the following architecture.



The training data is stored in an S3 bucket and fed to an EC2 training instance of our choice (see the available training instances here) where the train.py python code will train the model and save it to another S3 bucket.

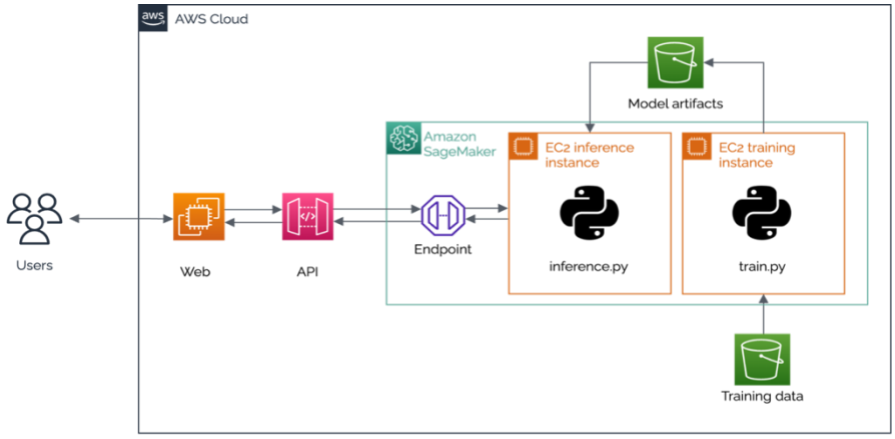
This trained model is loaded into the EC2 inference instance of our choice (see the available inference instances here) which contains the code inference.py that will make predictions given an input.

To give inference.py an input, the users will access our website, hosted in another EC2 instance, input the sentence they want to predict with and click submit. This will send a request to our API which will format the request to fit the needs of the input of our model and send it to the endpoint (provided by EC2 inference instance) which will communicate with the instance.

When the endpoint receives the output from the inference instance, it will send it to the API which will format it to fit the needs of the website and send it to the website so it can be visualized by the user.

This architecture is very efficient since it is very elastic and horizontally scalable. We can increase the number of EC2 instances hosting the website if the traffic increases and we could even do that in different geographic zones for faster access from the users. The same is possible with the EC2 inference instance, we could have many instances running in parallel to be able to serve multiple requests at once. When it comes to the training, we could easily set a distributed training with SageMaker for very large models.

However, this is also very expensive. Since we wanted to maintain the cost of this project at zero, we made the following change in the architecture.



This is because AWS does not offer EC2 training instances with GPU in its free tier and our model requires GPU to be trained, otherwise it is very slow (even days of training) and inefficient. We trained our model in Kaggle which offers free instances with GPU and manually uploaded the trained model in the Model artifacts S3 bucket.

AWS

As mentioned before, the chosen cloud provider for the development of this project is S3 offered by AWS. One of the reasons for choosing this cloud provider is because Amazon offers a free service to store the data with a fairly easy access, which Crystal Gazers will use for storing the Wikipedia dataset. The dataset, because of requiring a lot of space, will be stored in a bucket on S3 and loaded into memory when needed. The Python library used to download the data from S3 is boto3. The data is loaded into the training file by scripting:

```
def download_s3(bucket_name, path_to_file):
```

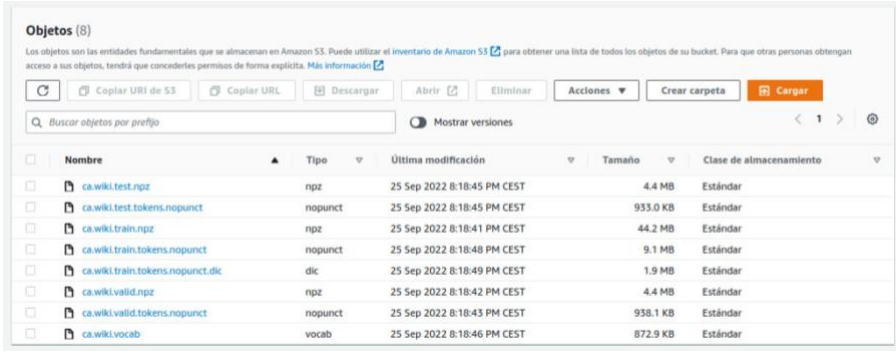
```
    s3_client = boto3.client('s3')

    obj         =         s3_client.get_object(Bucket=bucket_name,
Key=path_to_file)

    return obj
```

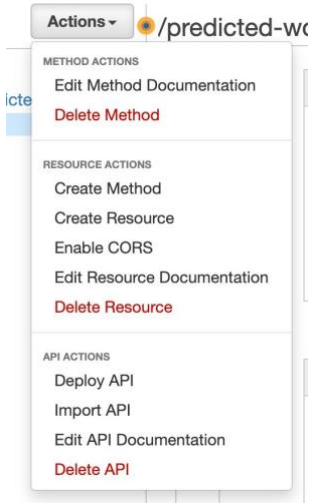
where the obj will be the entry to the trained model.

In addition, S3 bucket is linked to the DVC, and thus the history of changes in the dataset will be tracked in the S3 platform.



API GATEWAY

The API GATEWAY is an Amazon tool for implementing an API in order to connect the user interface (frontend) with the endpoint deployed in SageMaker. The API GATEWAY is a user-friendly platform that allows the implementation of the different available methods. CrystalGazers implemented `/predicted-word` method. The input context words are received with the GET method, put into a json file and sent using the POST method to the SageMaker endpoint . To check if the web-endpoint interaction completed successfully the status code needs to be checked (done with Pytest).



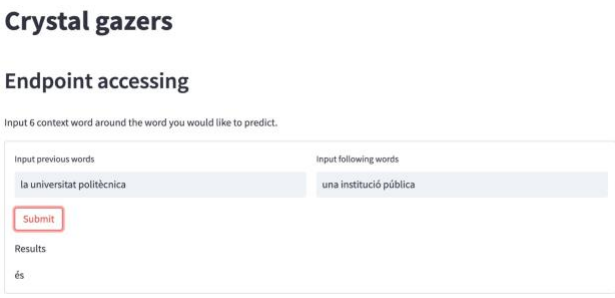
After defining the interaction, the platform offers to create an url to make requests to the API.

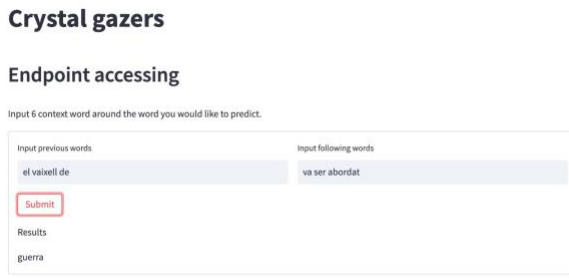
Another good feature of the API GATEWAY is that it generates automatic documentation, which can be found in the repository.

STREAMLIT

When the API was created, the next step was to define a user interface so that the requests could be done interactively. The interface is developed in Python using the `streamlit` library.

The interface converts the user's input into a GET request which will be used by the API. The given prediction is displayed after receiving the output of the API.



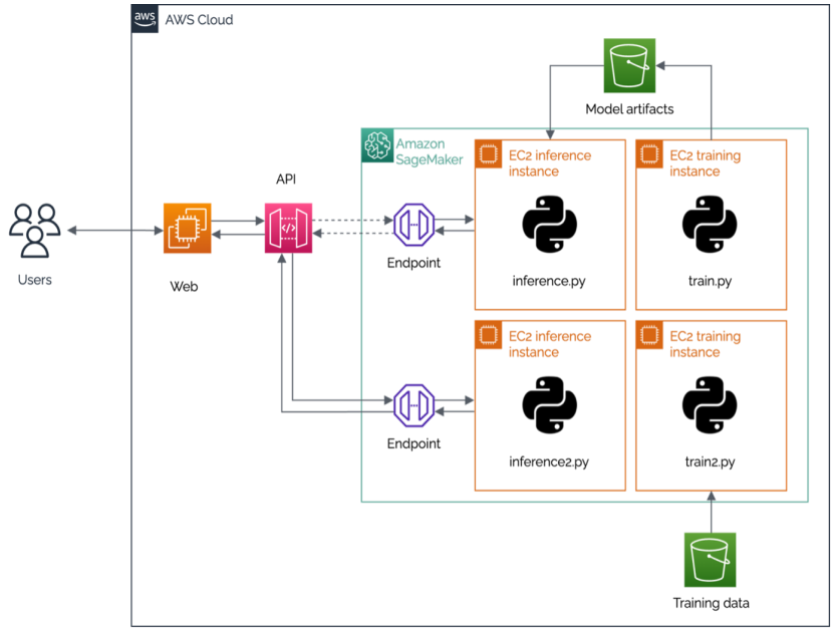


In order that the web can be accessed at any type from an actual url the web is running on EC2 which offers a continuous server environment.

The interface only accepts inputs in the correct format which is 3 context words before and 3 afterwards. In case of a mistake the user will see an error message that will notify them about the incorrect input format.

Cycle

Retraining and deploying new versions of our ML model would be very practical if we were to implement our architecture for a business (that is, not only using the AWS free tier and paying for GPU instances). We could do that with the following architecture.



The graph shows how we could use a secondary EC2 training instance to retrain the model (for example with different parameters), save it to the Model artifacts S3 bucket and then load it into a secondary EC2 inference instance. When the endpoint of the secondary inference instance is ready and we have tested the model, we just need to change the endpoint where the API points to and all new predictions will come from the second model. This has the advantage that there is no downtime and the changes in the code can be done very conveniently since SageMaker is compatible with Git repositories, so it would be as easy as developing the new model in a different branch and then switching branches in SageMaker.

In our current architecture (with the free tier) we would need to train the new model in Kaggle, then manually upload it into the Model artifacts S3 bucket and the rest would be the same.


4

Retrospective

This final report contains all the work done during the TAED2 project development under the Crystal Gazers organization. The team consisting of GCED students developed a set of data science and engineering project following good practices of a real project.

Some of the difficulties we encountered were related to the lack of experience in software and API development and model deployment, nevertheless after needed research the team managed to complete all the demanded work.

The development of this report consisted of two parts, the initial report – Milestones 1 – 3 and the final report Milestones 1 – 4.

As a whole, this project helped us to learn new techniques and tools that we can use during real work in technological companies and also helped us to see which ones can really be applied to a specific project and which ones are not essential.