Machine Learning Operations

Professor Nuno Rosa

From theory to practice: Creating and deploying into production fully functional machine learning pipelines

https://github.com/andrefilipefmsilva/MLOps_NovaIMS

Authors:

André Filipe Silva	20230972
Diogo Almeida	20230737
Frederico Portela	20181072
Gonçalo Caldeirinha	20230469
João Pedro Mota	20230454

NOVA IMS
2023/2024

Contents

1 Introduction

In this work, our objectives were two-fold. First, we tackled the problem of predicting whether diabetic patients would be readmitted into the hospital some time after being discharged. Hospital readmissions represent a main challenge in the healthcare sector, bearing serious consequences for both patients and the hospitals. Multiple studies have demonstrated that higher rates of readmission correlate with higher healthcare costs, decreased quality of care, and higher mortality rates in the hospital setting. Diabetic patients seem to have an even higher risk of readmission than the general population, as they tend to have comorbidities associated with their primary illness.

The second objective, and perhaps the main one with regards to this course, was to do all of this while implementing the best MLOps practices: modular code, pipeline orchestration, unit tests, among others that we will describe later in our work. Our goal was to go from data ingestion to model deployment and monitoring, leaving behind the traditional academic setting of the Jupyter Notebook.

To optimize our workflow, we adopted the Agile methodology, breaking down our project in several smaller pieces with iterative weekly sprints.With this said, we had weekly scrum meetings, where one member of the team would essentially serve as the Scrum master and provide orientation over the workflow - what was achieved during the previous week, what was in the backlog, any pain points, and set goals for the next week.

2 Data Overview and Succcess Metrics

Our data is composed of records for diabetic patient intakes at 130 US hospitals ranging from 1999 to 2008. You can find the original dataset here.

Our main success metric is the f1 score, combining recall and precision. We had other ideas for metrics taking into account domain matter expertise, especially proxy metrics for readmission prediction in an earlier stage. This will be measure over our **target variable**, *readmitted.* These metrics would be measures such as patient temperature, oxygen saturation, and social support (family support, nursery home quality). However, the dataset did not allow us to implement this, as there was no information regarding any of these parameters. As such, we decided that the f1 score would remain the most appropriate metric for this problem. In addition, we also keep a close eye on overfitting: if one of our models presents a difference in f1 score between train and test results that is bigger than 0.1, we consider overfitting to exist and stop this model from being labeled as the ”best model”.

3 Exploratory Data Analysis

Going through a quick exploratory analysis of our data, we have 101 766 observations, and 27 features, although two of them are IDs *´encounter_id´* and *´patient_nbr´*. The features *weight*, *max_glu_serum*, *A1Cresult*, and *medical_specialty* have more than 50% of missing values, and we decided to simply drop these features as the sheer amount of missing data is too big to consider any inputation methods.

We performed various types of analysis:

1. Investigate features' distributions, both for numeric and categorical features
2. Plot for outliers
3. Pearson's correlation matrix
4. Chi-Square test to find correlation between categorical variables and target variable
5. Cramer's V test for correlation between categorical variables and the target variable
6. Plot our target variable to check the class balance

We believe the most important thing to note is that both the Chi-Square test and the Cramer's V test results are very similar, but we find little to no association between our categorical features and the *readmitted* target variable.

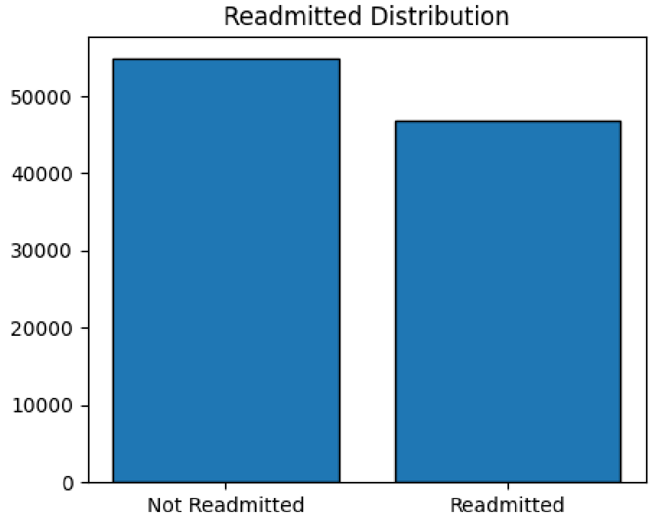

Figure 1: Distribution of our target variable

Our target variable, *readmitted* is actually quite balanced, as can be seen in the plot above.

4 Implementation

This is the most important part of our work, as this is where we put into practice the principles of good MLOps development.

We aimed at building fully modular, independent code that could be run at will according to different needs at different points in time of the process, without interfering with previous or further steps in the development process.

Using Kedro as the tool for the management and development of our pipelines, it is quite easy to run exactly what we want and only what we want. We also created two main pipelines (that are a combination of the individual ones created for each step): ***dev*** - this includes all the pipelines necessary for development of new challenger models, and finally ***prod*** - for when we want to deploy a new model into production. In order to fully separate the two pipelines, we created two Kedro environments, each one with its own data catalog and parameters. Due to this, the pipelines are run like: *kedro run –pipeline <pipeline name> –env=<dev or prod>*. We utilize the Kedro logger to register the pipeline's workflow.

We also have the individual pipeline ***data_upload*** - for when we want to upload new incoming data to the feature store (HopsWorks). It also works for both environments.

The importance of Kedro and HopsWorks in our work is accompanied by the use of Mlflow, which allows us to track and record our experiments, and define our champion model, and track challengers. It also gives key insights into model performance, which in turn helps in the definition of whether the actively deployed model is performing well or being outperformed by other contenders.

The ideology behind the *dev* and *prod* pipelines is that the first one will be more dynamic, and the latter one more static. First of all, it should be noted that changes in *dev* do not necessarily imply changes in *prod.* We want to ensure a certain stability of our production environment, so even if we find a challenger model in *dev* that is behaving better than the current champion, we don't immediately deploy it to production - this would only be done after extensive shadow testing.

All the functions implemented in our code were accompanied by **unit tests**. These are essential to make sure that our functions work as expected, increasing code quality. They are also important for early bug detection, reducing the cost and effort of fixing them later. We focused our testing on the most important functions of our project, and achieved a 49% coverage/footnote[2]Previously, we had 69% coverage. Last minute issues caused it to go down. We go into more detail about this in the Limitations section of our work. Although this is not full coverage, we tackled the most pressing issues.

In addition, we also have **data unit tests** - these focus on the data itself as the name says, checking for data quality and consistency. Some examples include: checking if the data distributions remain relatively constant; checking if all the values in a feature have the expected type; checking if some columns have unique values only (important for instance for encounter_id); checking if the new data contains the same columns as the existing data. For these tests, the greatexpectations library was the tool of choice.

It is also important to mention that we tackled explainability and data drift. **Explainability** is an ever increasing concern with machine learning models - if we want to bring them in mass to real use cases, we must be able to explain how we got our results. We use Permutation Importance and SHAPley values to assess this. On the other hand, **data drift** is also a problem that has gathered more and more attention in recent years, as real world data changes its distribution and pattern over periods of time, and that leads to decay in model performance. It is of utmost importance to monitor it and retrain/replace models when the impact is found.

We will go through a brief description of each of our individual pipelines:

1. **data_upload**: Uploads new data to the feature store, but makes it go through our data unit tests beforehand. We have two feature groups: dev_raw_data and prod_raw_data. Before being uploaded to the feature store, a column named datetime is added to them, in which the values is the current time stamp.
2. **data_ingestion**: Loads the data from the feature store to our development environment. When loading the data, we filter them according to the feature group with the most recent time stamp.
3. **track_ids**: Performs a simple dataframe split between the actual features and the encounter identifiers (IDs). This node is connected to the *model_inference* node, as if, for example, we want to create an API to connect to our model, this is necessary. (prod only)
4. **stateless_cleaning**: Initial preprocessing, also based on what we found through EDA. Replacing null values with *np.nan*, among other initial cleaning. These are transformations that can be done before any data splits.
5. **apply_stateful_transformations**: Performs transformation of the data (i.e. inputation with a previously fitted inputer) based on our champion model. Unlike the stateless_cleaning transformations, to avoid data leakage these need to be applied after the data split. (prod only)
6. **feature_engineering**: Applies the Charlson Comorbidity Index, adds a feature regarding the total number of hospital visits in the previous year, and a feature named complexity score, which aims to be a proxy for how complex the case is based on the number of lab procedures, number of different diagnosis and number of distinct medications a patient is on.
7. **data_split**: As indicated by the name, splitting the data into train, validation and test sets. It is worthwhile to mention here that the validation split exists mainly to combat the possible overfit created by hyper-parameter tuning.
8. **stateful_cleaning**: On the columns that still have missing values but were not dropped, apply inputation based on the most frequent value. The fitted inputer is logged in Mlflow as an artifact.
9. **feature_selection**: Performs feature selection on the train data, using RFE with a Random Forest classifier. Then filters the dataset to keep only the selected features. Apart from RFE, DT importances, and the possibility of manually choosing all features or a handpicked list is also implemented. The selected features are logged into Mlflow.
10. **model_selection**: Implements a number of different machine learning models and then uses Optuna for hyperparemeter tuning in all of them. Finally, the best model to come out of this process compares to the champion model. If the f1 score of the best model of the run is close to or better than the champion model's, the new model is registered as a *challenger*. The reason why we also consider models with f1 scores below (but close to) the champion's as challengers, is because the

challenger was not necessarily tested on the same dataset than the champion. As such, it may have different cleaning steps and different features - that might be important. However, we don't immediately switch our champion model. Abruptly changing the production model could cause a myriad of issues downstream, i.e. breaking production. Hence, these promising models are logged as challengers, and after thorough shadow or A/B testing, if they really do beat the champion, then they are manually promoted to the new champion model and the old champion is archived. Note that on the first run ever there is no champion model, so the best model to come out will be registered as the champion model. The output of this pipeline, however, is always the run's best model (whether it beats the champion or not).

11. **concatenate_data**: At this stage we join all of the data (train, validation, test), as we believe the data drift should be evaluated on the whole dataset.
12. **explainability**: Calculated Permutation Importance as a way to show what features contribute the most to our model's predictions.
13. **data_drift**: Generates a report on data drift using the Evidently library. Testing for data drift on a clean dataset ensures that the detected changes are relevant and not due to noise or inconsistencies present in the raw data. We use the same reference data for dev and prod, to ensure consistency in model evaluation and performance monitoring, leading to reliable comparisons and more accurate detection of any deviations or issues between environments.
14. **data_unit_tests_after_preprocessing**: Utilizing *greatexpectations*, we implement a data unit test here to validate that everything is correct after preprocessing. It is only implemented in production due to the experimental nature of the development pipeline. We did not deem it necessary, or even useful, to have such a node inserted into it - constant new cleaning steps and the creation of the new features would mean for every change we would need to update the unit tests for dev accordingly. (prod only)
15. **feature_pruning**: Filters the dataset for imcluding only the features that are hard-coded in the parameters for the production environment. (prod only)
16. **model_inference**: Used to make predictions using our current champion model. The output is: (*encounter_id, prediction*). (prod only)

You can find below a visual explanation of our pipeline sequence.

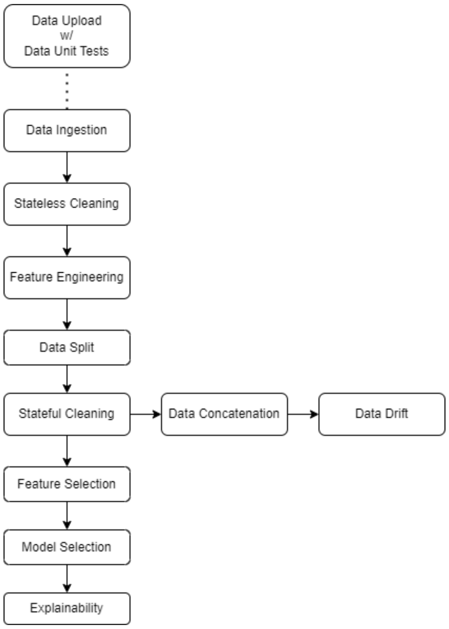

Figure 2: Dev Pipeline sequence, illustrated

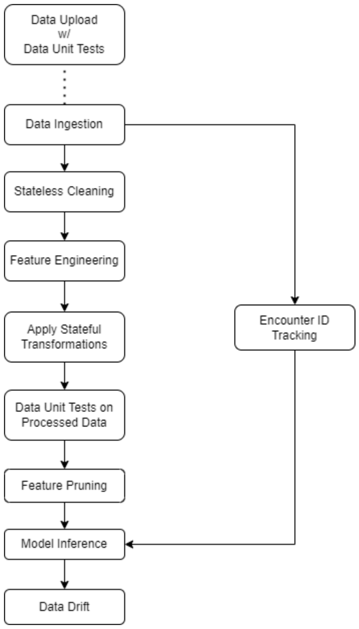

Figure 3: Prod Pipeline sequence, illustrated

Finally, we containerized our work by making use of *kedro-docker*. This is paramount for an improved model serving experience.

5 Results

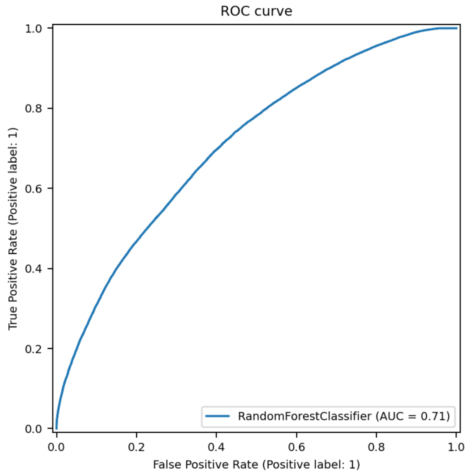

Figure 4: AUC ROC Plot - Model performance for every decision threshold

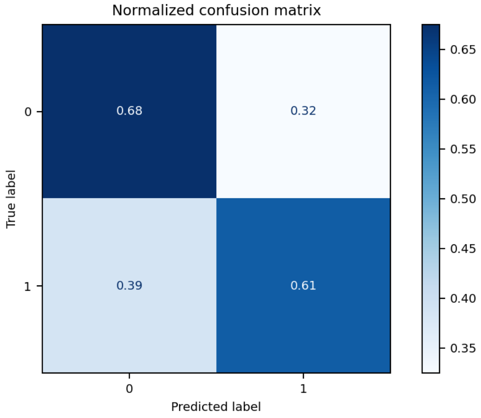

Figure 5: Confusion Matrix scores

Starting with a performance metric, our pipeline only took 31 seconds to run.

Our champion model is a Random Forest Classifier, with an f1 score 0.65 on the train set and 0.60 on the test set. The train accuracy is 0.65 and the test accuracy is 0.63. The Area Under the Curve (AUC) score is 0.71.

It should be noted that we have a challenger model with slightly better scores than our champion model. Again, as already explained, to promote this challenger into production, it would need extensive A/B or shadow testing before that happening.

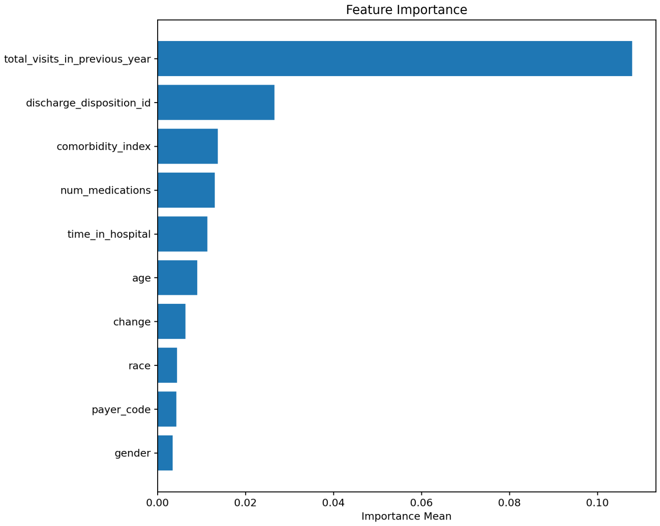

Figure 6: Explainability - Main features contributing to our model predictions

Moving on to **Explainability**, we utilized Permutation Importance to understand the main contributing figures to our model's results. The plot is very clear on this: the main feature contributing to explaining readmission to the hospital is the number of total visits to the hospital a patient had in the previous year (*total_visits_in_previous_year*). Although it is important to be aware of the x-axis scale, we are still looking at a feature that is 50% more important than the second most important feature that explains our model results (*discharge_disposition_id*).

In order to evaluate **Data Drift**, as said above we resorted to the Evidently library. This library produced a report, that can be found here. To keep things simple, we opted to let Evidently use its default statistic tests, as it has an in-built algorithm that selects a suitable drift test based on the feature type, number of observations, and unique values.

As can be seen on the report, drift is detected on two features: *age* and *comorbidity_index*. Although it might seem like not a lot, 2 columns out of 19 represent about 10.5% of our dataset, which we believe is a reasonable number to trigger model retrain.

In addition, *comorbidity_index* is our third most important feature, according to what we have seen in Figure 6. [3]

6 Limitations and future work

The first limitation that we should mention is that, even though we tried to make our development and production pipelines as independent as possible, the truth is they are not integrally separated. For simplicity's sake, we are reusing some nodes in both (e.g. feature engineering). The reason for this is, for this Proof of Concept we did not want to exponentially increase the amount of code to be written - although in a real world scenario we would likely have to reevaluate the construction of our pipelines.

Towards the end of our work, we ran into some issues when implementing some changes in the *model_selection* logic. These issues seem to be, for some reason, related with files pointing to local directories instead of working dynamically. This provoked a domino-effect in our codebase that led to issues in *pytest* that took a lot of time to troubleshoot. In the end, this took our unit test coverage from 69% to 49%. This also impacted our ability to implement features we would have liked to.

There might be trouble when using Mlflow - as said above, we had a lot of last minute issues with runs pointing towards local directories and artifacts not saving properly. Running our code might cause the user the same errors.

Another important limitation, is that we could not implement SHAP in time, greatly affecting the explainability of our models. You can find our attempts in a development branch. However, we were not able to make it run by the time this work was due.

Finally, current data drift detection is reactive; proactive strategies could enhance model robustness.

As future work, and time allowing, we would like to implement our work using a GUI, likely resorting to Streamlit as our tool of choice. This makes it more appealing and user-friendly. We still briefly attempted it, but ran out of time before being able to do it.

Developing a real-time monitoring system for data drift to enable prompt interventions comes to mind.

Implementing feedback loops from healthcare providers to continuously refine model predictions and relevance would prove to be invaluable.

Scalability might also be an issue - as this is hospital data, we can expect it to increase in volume very fast, which would lead us to the need to implement our solution utilizing Spark, to make our pipelines timely and efficient for big data analytics.

Lastly, it would be important to use something like Apache Airflow for pipeline orchestration - e.g. when triggering the run of the dev pippeline after data drift appears.

[3] For the purposes of testing data drift, we artificially changed the distribution of age. As the *comorbidity_index* depends on the age variable, this is why we find drift in both these features.

Appendix

Appendix A - List of Requirements

According to the project guidelines, we list below the full list of packages and versions required to run our project. There should be no dependency conflicts when installing a fresh virtual environment using this list. However, you can find this in a file here.

```
aiomysql==0.2.0
alembic==1.13.1
altair==4.2.2
aniso8601==9.0.1
annotated-types==0.6.0
antlr4-python3-runtime==4.9.3
anyio==4.3.0
appdirs==1.4.4
argon2-cffi==23.1.0
argon2-cffi-bindings==21.2.0
arrow==1.3.0
asttokens==2.4.1
async-lru==2.0.4
attrs==23.2.0
avro==1.11.3
Babel==2.14.0
beautifulsoup4==4.12.3
binaryornot==0.4.4
bleach==6.1.0
blinker==1.7.0
boto3==1.34.88
botocore==1.34.88
build==1.2.1
cachetools==5.3.3
certifi==2024.2.2
cffi==1.16.0
chardet==5.2.0
charset-normalizer==3.3.2
click==8.1.7
cloudpickle==3.0.0
colorama==0.4.6
colorlog==6.8.2
comm==0.2.2
confluent-kafka==2.3.0
contourpy==1.2.1
cookiecutter==2.6.0
cryptography==42.0.5
cycler==0.12.1
dacite==1.8.1
debugpy==1.8.1
decorator==5.1.1
defusedxml==0.7.1
distro==1.9.0
docker==7.0.0
dynaconf==3.2.5
entrypoints==0.4
et-xmlfile==1.1.0
evidently==0.4.25
exceptiongroup==1.2.1
executing==2.0.1
Faker==25.9.1
fastavro==1.8.4
fastjsonschema==2.19.1
featuretools==1.30.0
filelock==3.15.4
Flask==3.0.3
fonttools==4.51.0
fqdn==1.5.1
fsspec==2024.3.1
furl==2.1.3
future==1.0.0
gitdb==4.0.11
GitPython==3.1.43
google-auth==2.30.0
graphene==3.3
graphql-core==3.2.3
graphql-relay==3.2.0
great-expectations==0.15.12
greenlet==3.0.3
h11==0.14.0
holidays==0.47
hopsworks==3.7.0
hsfs==3.7.2
hsml==3.7.0
htmlmin==0.1.12
httpcore==1.0.5
httptools==0.6.1
httpx==0.27.0
idna==3.7
ImageHash==4.3.1
importlib_metadata==7.1.0
importlib_resources==6.4.0
```

```
ipykernel==6.29.4
ipython==8.23.0
ipywidgets==8.1.2
isoduration==20.11.0
iterative-telemetry==0.0.8
itsdangerous==2.2.0
javaobj-py3==0.4.4
jedi==0.19.1
Jinja2==3.1.3
jmespath==1.0.1
joblib==1.4.0
json5==0.9.25
jsonpatch==1.33
jsonpointer==2.4
jsonschema==4.21.1
jsonschema-specifications==2023.12.1
jupyter-events==0.10.0
jupyter-lsp==2.2.5
jupyter_client==8.6.1
jupyter_core==5.7.2
jupyter_server==2.14.0
jupyter_server_terminals==0.5.3
jupyterlab==4.1.6
jupyterlab_pygments==0.3.0
jupyterlab_server==2.26.0
jupyterlab_widgets==3.0.10
kedro==0.19.4
kedro-datasets==3.0.0
kedro-docker==0.6.0
kedro-mlflow==0.12.2
kiwisolver==1.4.5
kubernetes==29.0.0
lazy_loader==0.4
litestar==2.9.1
llvmlite==0.42.0
makefun==1.15.2
Mako==1.3.3
Markdown==3.6
markdown-it-py==3.0.0
MarkupSafe==2.1.5
marshmallow==3.21.1
matplotlib==3.8.4
matplotlib-inline==0.1.7
mdurl==0.1.2
mistune==3.0.2
mlflow==2.11.3
mock==5.1.0
more-itertools==10.2.0
msgspec==0.18.6
multidict==6.0.5
multimethod==1.11.2
mypy-extensions==1.0.0
nbclient==0.10.0
nbconvert==7.16.3
nbformat==5.10.4
nest-asyncio==1.6.0
networkx==3.3
nltk==3.8.1
notebook==7.1.3
notebook_shim==0.2.4
numba==0.59.1
numpy==1.26.4
oauthlib==3.2.2
omegaconf==2.3.0
openpyxl==3.1.2
opensearch-py==2.4.2
optuna==3.6.1
orderedmultidict==1.0.1
overrides==7.7.0
packaging==23.2
pandas==2.1.4
pandocfilters==1.5.1
parse==1.20.1
parso==0.8.4
patsy==0.5.6
phik==0.12.4
pillow==10.3.0
platformdirs==4.2.0
plotly==5.22.0
pluggy==1.3.0
polyfactory==2.16.0
pre-commit-hooks==4.6.0
prometheus_client==0.20.0
prompt-toolkit==3.0.43
protobuf==4.25.3
psutil==5.9.8
pure-eval==0.2.2
py4j==0.10.9.7
pyarrow==15.0.2
pyasn1==0.6.0
pyasn1_modules==0.4.0
pycparser==2.22
pycryptodomex==3.20.0
pydantic==2.7.0
pydantic_core==2.18.1
Pygments==2.17.2
PyHopsHive==0.6.4.1.dev0
pyhumps==1.6.1
pyjks==20.0.0
PyMySQL==1.1.0
```

```
pyparsing==2.4.7
pyproject_hooks==1.0.0
pyspark==3.5.1
python-dateutil==2.9.0.post0
python-dotenv==1.0.1
python-json-logger==2.0.7
python-slugify==8.0.4
pytoolconfig==1.3.1
pytz==2024.1
PyWavelets==1.6.0
pywin32==306
pywinpty==2.0.13
PyYAML==6.0.1
pyzmq==26.0.2
querystring-parser==1.2.4
referencing==0.34.0
regex==2024.5.15
requests==2.32.3
requests-oauthlib==2.0.0
retrying==1.3.4
rfc3339-validator==0.1.4
rfc3986-validator==0.1.1
rich==13.7.1
rich-click==1.8.3
rope==1.13.0
rpds-py==0.18.0
rsa==4.9
ruamel.yaml==0.17.17
ruff==0.4.1
s3transfer==0.10.1
scikit-learn==1.4.2
scipy==1.11.4
seaborn==0.12.2
Send2Trash==1.8.3
setuptools==69.5.1
shap==0.45.1
shellingham==1.5.4
six==1.16.0
slicer==0.0.8
smmap==5.0.1
sniffio==1.3.1
soupsieve==2.5
SQLAlchemy==1.4.48
sqlparse==0.5.0
stack-data==0.6.3
statsmodels==0.14.2
streamlit==1.36.0
tenacity==8.4.1
termcolor==2.4.0
terminado==0.18.1
text-unidecode==1.3
threadpoolctl==3.4.0
thrift==0.20.0
tinycss2==1.2.1
toml==0.10.2
tomli==2.0.1
toolz==0.12.1
tornado==6.4
tqdm==4.66.2
traitlets==5.14.3
twofish==0.3.0
typeguard==4.2.1
typer==0.12.3
types-python-dateutil==2.9.0.20240316
typing-inspect==0.9.0
typing_extensions==4.11.0
tzdata==2024.1
tzlocal==5.2
ucimlrepo==0.0.6
ujson==5.10.0
uri-template==1.3.0
urllib3==1.26.18
uvicorn==0.30.1
visions==0.7.6
waitress==3.0.0
watchdog==4.0.1
watchfiles==0.22.0
wcwidth==0.2.13
webcolors==1.13
webencodings==0.5.1
websocket-client==1.7.0
websockets==12.0
Werkzeug==3.0.2
widgetsnbextension==4.0.10
woodwork==0.30.0
wordcloud==1.9.3
ydata-profiling==4.7.0
zipp==3.18.1
```