Complete Data Science Project Template with Mlflow for Non-Dummies.

Best practices for everyone working either locally or in the cloud, from start-up ninja to big enterprise teams.

Jan Teichmann Follow

Nov 18, 2019 · 18 min read ★

Data science has come a long way as a field and business function alike. There are now cross-functional teams working on algorithms all the way to full-stack data science products.

Kenneth Jensen [CC BY-SA 3.0]

With the growing maturity of data science there is an emerging standard of best practise, platforms and toolkits which significantly reduced the barrier of entry and price point of a data science team. This has made data science more accessible for companies and practitioners alike. For the majority of

comm of a
quality

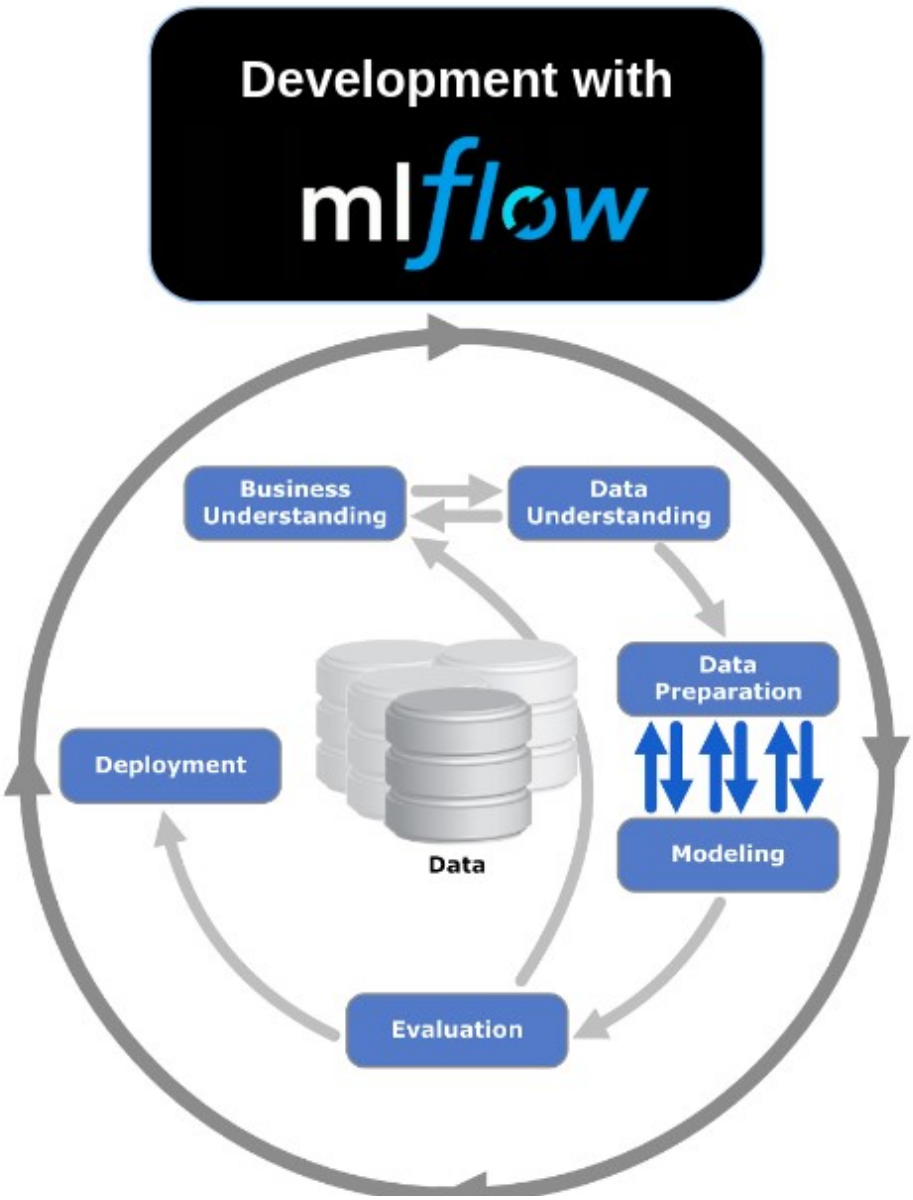
But the
project hat
over 8(With
all the to
deliver

The da he
“**Last M**

“Prod ıce”
(Schu ,
O’Reil

In this **oject**,
model
science

Mlflo
Projec
gitlab.



You can read about Rendezvous Architecture as a great pattern to operationalise models in production in one of my earlier blog posts:

Rendezvous Architecture for Data Science in Production

How to build a cutting edge data science platform to solve the real challenge in Data Science: Productionisation.

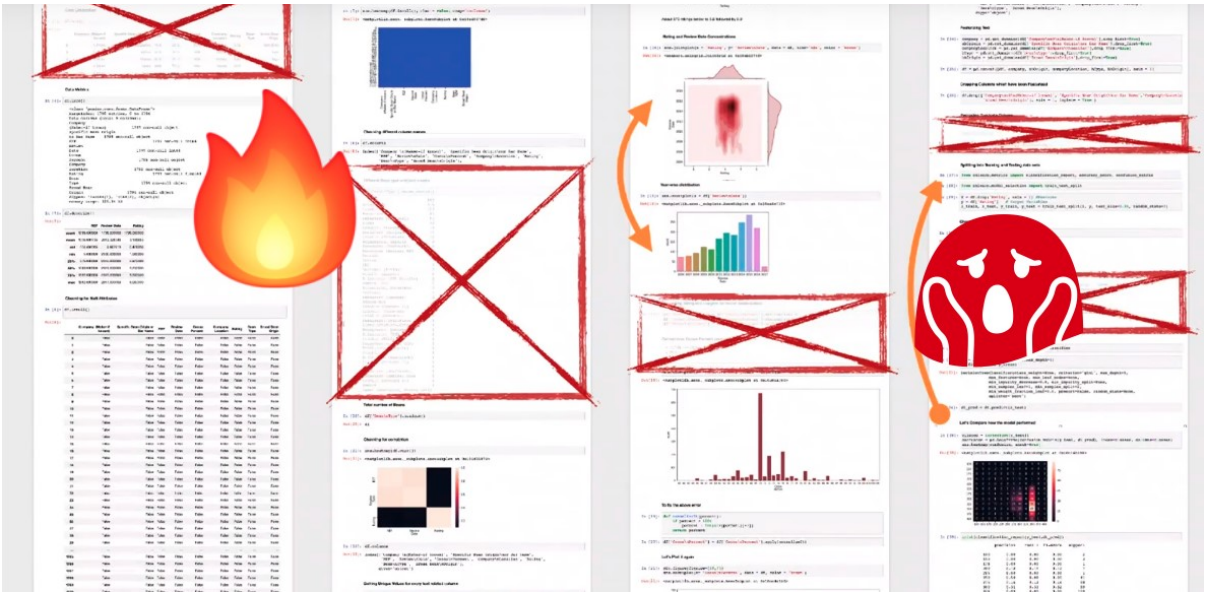
towardsdatascience.com

Data Science like a Pro

While engineers build the production platforms, DevOps and DataOps patterns to run models in production, many data scientists still work in a way which creates a **big gap between data science and production environments**. Many data scientists (without any reproach)

- work **locally** without much consideration of the cloud environments which might host their models in production
- work on **small datasets in memory** using python and pandas without much consideration of how to scale workflows to the big data volumes in production
- work with **Jupyter Notebooks** without much consideration for reproducible experiments
- rarely break up projects into independent tasks to build decoupled pipelines for ETL steps, model training and scoring
- And very rarely are best practices for software engineering applied to data science projects, e.g. abstracted and reusable code, unit testing, documentation, version control etc.

Unfortunately, our beloved flexible Jupyter Notebooks play an important part in this.



From "Managing Messes in Computational Notebooks" by Andrew Head et al., doi>10.1145/3290605.3300500

It's important to keep in mind that data science is a field and business function undergoing rapid innovation. If you read this a month after I published it, there might be already new tools and better ways to organise data science projects. **Exciting times to be a data scientist**! Let's have a look at the details of the data science project template:

The Data Science Environment

Icons made by Freepik, phatplus, Becris from www.flaticon.com

A data science project consists of many moving parts and the actual model can easily be the fewest lines of code in your project. Data is the fuel and **foundation** of your project and, firstly, we should aim for solid, high quality and portable foundations for our project. Data pipelines are the hidden technical debt in most data science projects and you probably have heard of the infamous 80/20 rule:

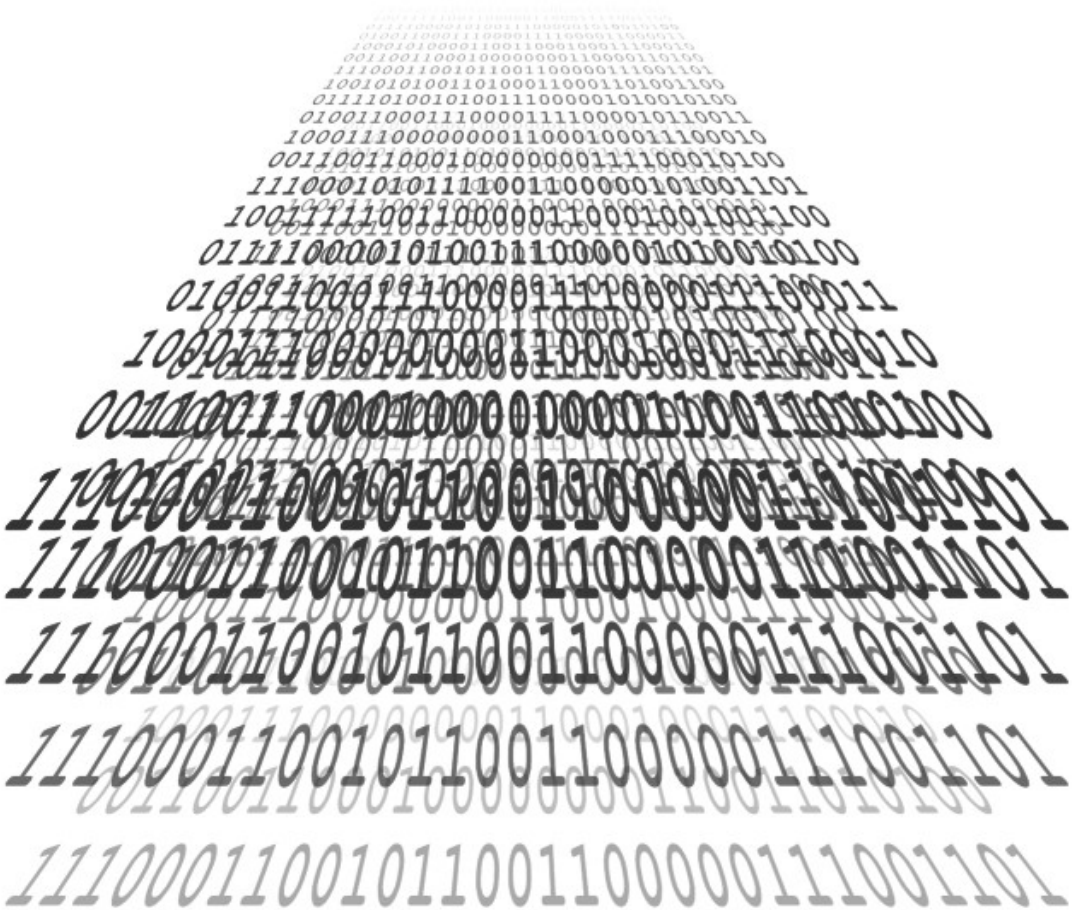
80% of Data Science is Finding, Cleaning and Preparing Data

I will explain below how Mlflow and Spark can help us to be more productive when working with data.

Google NIPS 2015, fair usage

Creating your data science model itself is a continuous back and forth between **experimentation** and expanding a project code base to capture the code and logic that worked. This can get messy and Mlflow is here to make experimentation and **model management** significantly easier for us. It will also simplify **model deployment** for us. More on that later!

Data



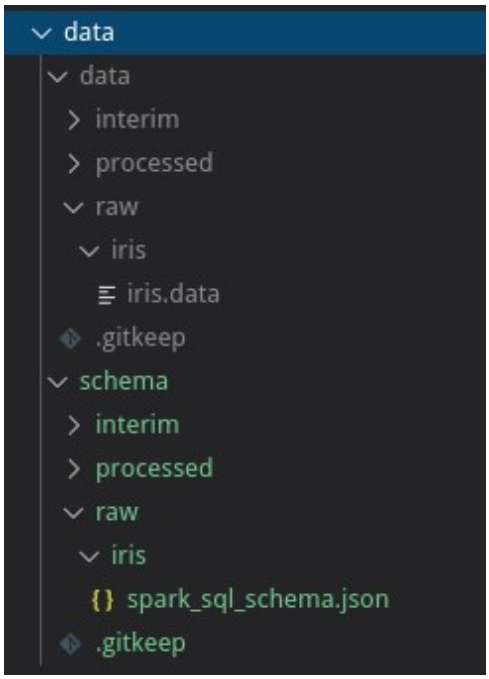
W.Rebel [CC BY 3.0]

You probably heard of the **80/20 rule** of data science: a lot of the data science work is about creating data pipelines to consume raw data, clean data and engineer features to feed our models at the end.

We want that

- Our data is **immutable:** we can only create new data sets but not change existing data in place. Therefore,
- our pipeline is a set of **decoupled steps** in a DAG incrementally increasing the quality and aggregation of our data into features and scores to incorporate our ETL easily with tools like Airflow or Luigi in production.
- Every data set has a defined data **schema** to read data safely without any surprises which can be logged to a central data catalogue for better data governance and discovery.

The data science project template has a data folder which holds the project data and associated schemata:

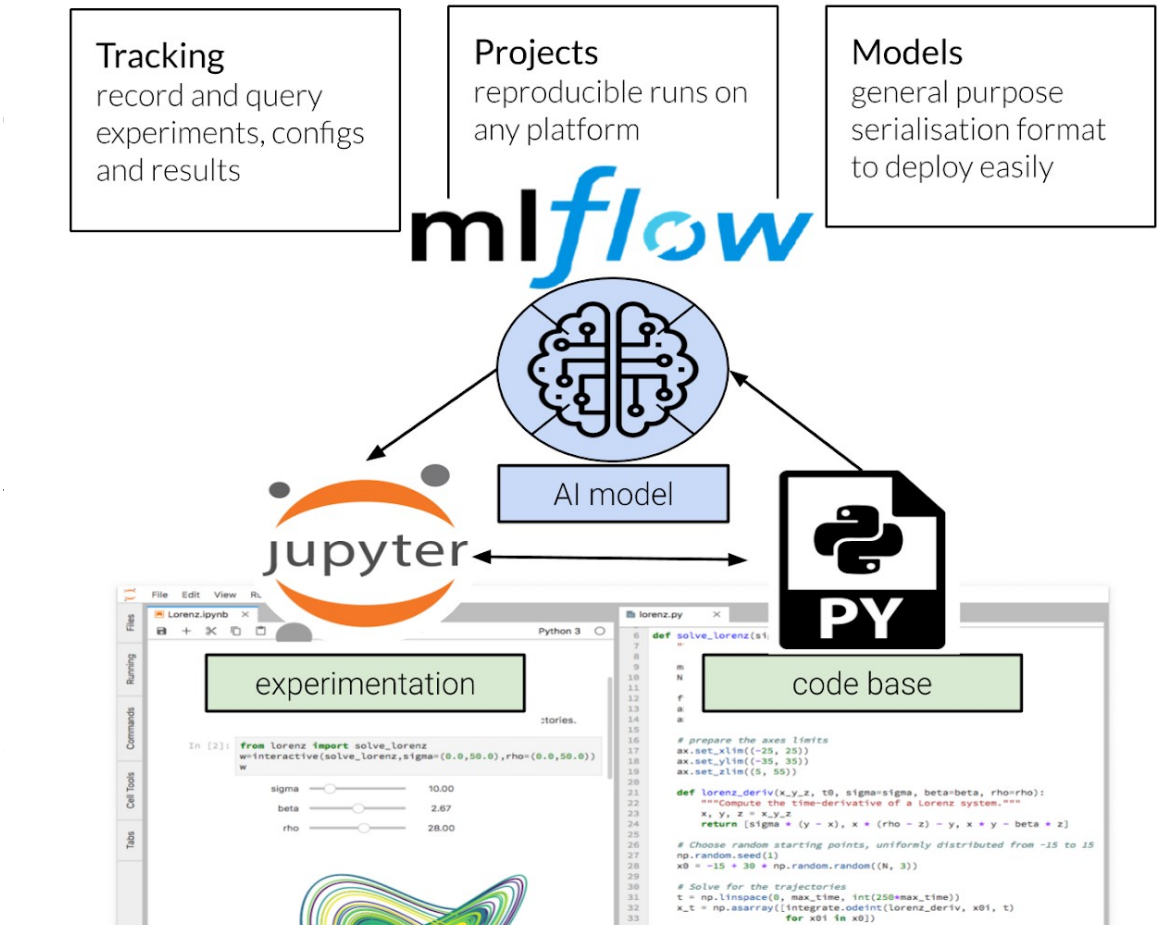


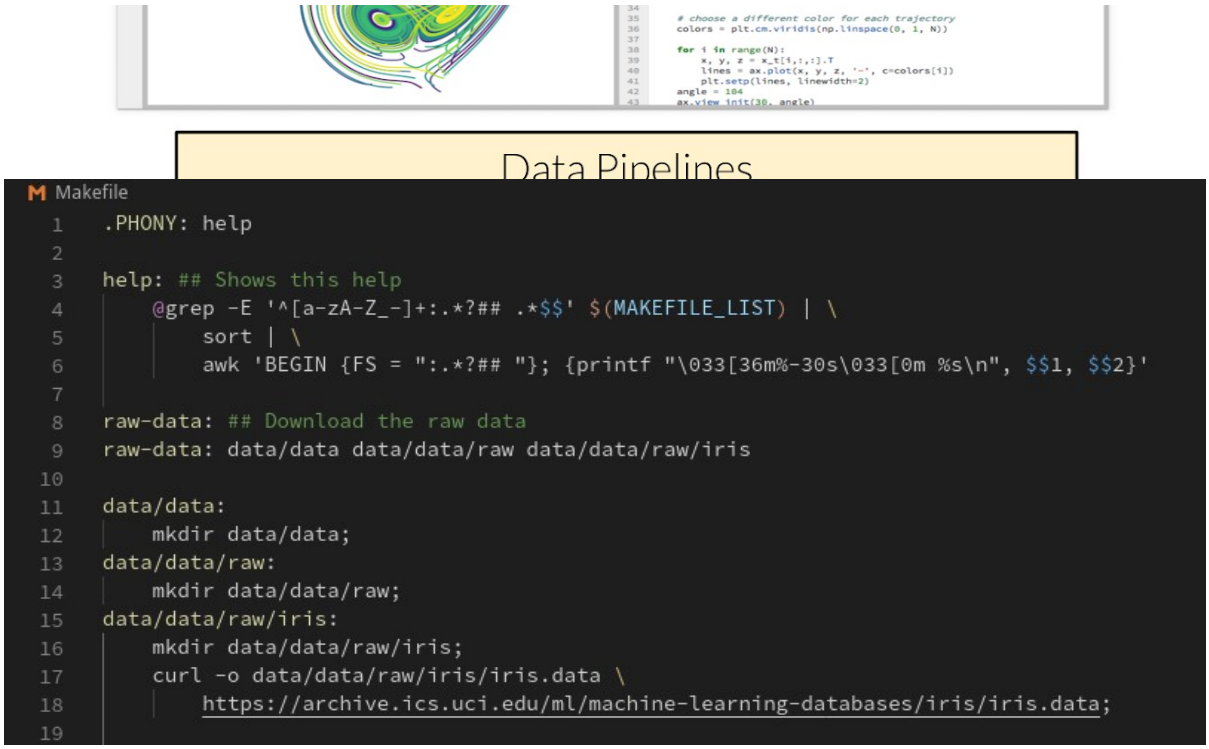
[OC]

Data scientists commonly work with not only big data sets but also with unstructured data. Building out the schemata for a data warehouse requires design work and a good understanding of business requirements. In data science many questions or problem statements were not known when the schemata for a DWH were created. That's why Spark has developed into a gold standard in that space for

- Working with unstructured data
- Distributed enterprise mature big data ETL pipelines
- Data lake deployments
- Unified batch as well as micro-batch streaming platform
- The ability to consumer and write data from and to many different systems

Projects at companies with mature infrastructure use advanced **data lakes** which includes **data catalogues** for data/schema discovery and management, and scheduled tasks with Airflow etc. While a data scientist does not have to necessarily understand these parts of the production infrastructure, it's best to





[OC]

The end to end data flow for this project is made up of three steps:

- **raw-data**: Download the iris raw data from the machine learning database archive
- **interim-data**: Execute the feature engineering pipeline to materialise the iris features in batch with Spark
- **processed-data**: Execute the classification model to materialise predictions in batch using Spark

You can transform the iris raw data into features using a Spark pipeline using the following make command:

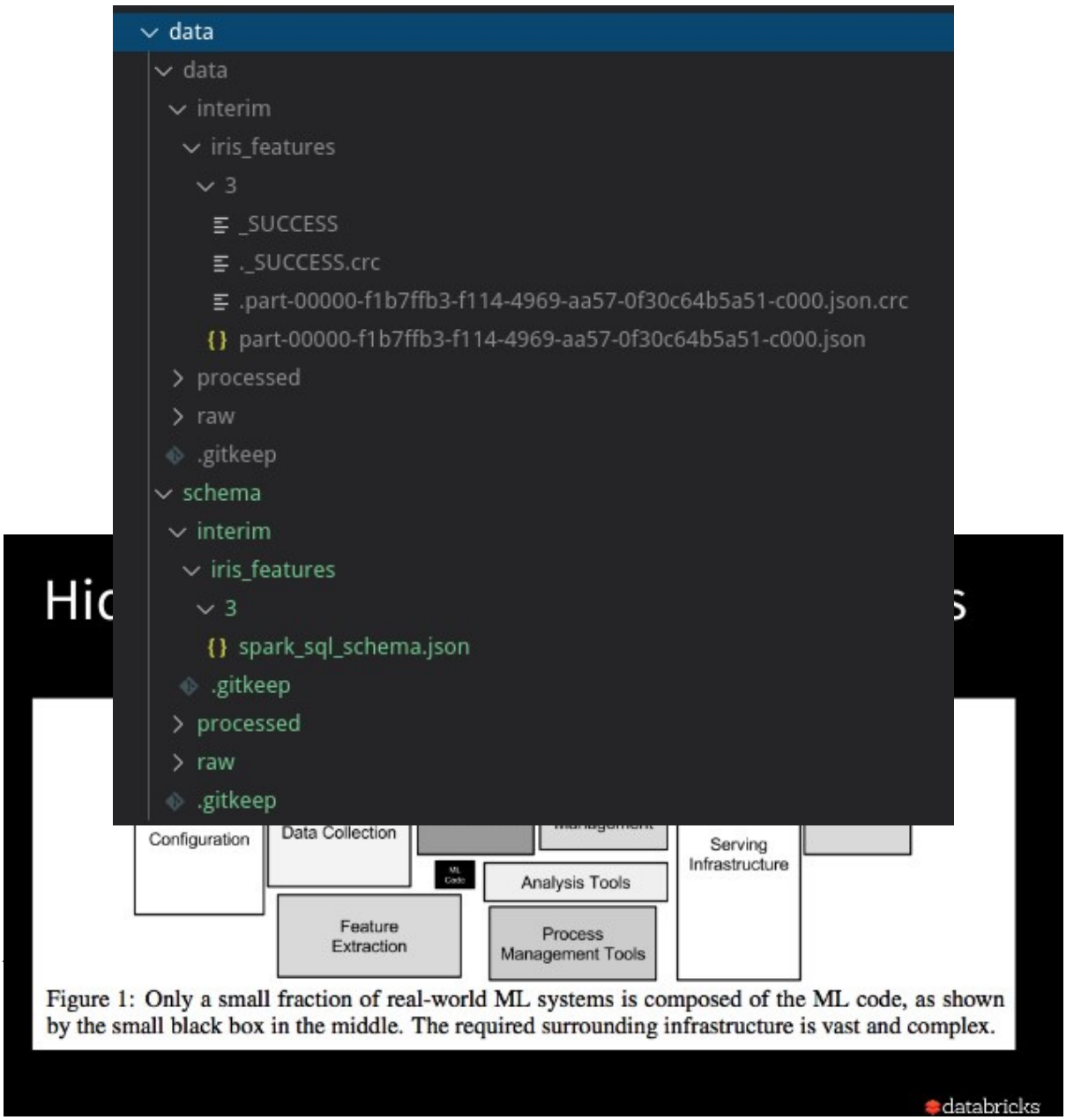
```
make interim-data
```

It will zip the current project code base and submits the **project/data/features.py** script to Spark in our docker container for execution. We use a Mlflow runid to identify and load the desired Spark feature pipeline model but more on the use of Mlflow later:

```
10
11  interim-data: ## Batch create interim data
12  interim-data: raw-data data/data/interim data/schema/interim iris-features
13
14  FEATURE_RUNID?=f3a735a824264043a7978ee1aa0230d6
15  iris-features:
16      docker exec -ti project_jupyter python zip_project.py;
17      docker exec -ti project_jupyter \
18          /usr/local/spark/bin/spark-submit --master local[*] \
19          --py-files project.zip \
20          project/data/features.py $(FEATURE_RUNID)
21
```

[OC]

After the execution of our Spark feature pipeline we have the interim feature data materialised in our small local data lake:



```
some_sparksql_dataframe.schema.json()
T.StructType.fromJson(json.loads
(some_sparksql_schema_json_string))
```

Always materialise and read data with its corresponding schema! Enforcing schemata is the key to breaking the 80/20 rule in data science.

I consider writing a schema as mandatory for csv and json files but I would also do it for any parquet or avro files which automatically preserve their schema. I really recommend reading more about Delta Lake, Apache Hudi, data catalogues and feature stores. If there is interest, I will follow up with an independent blog post on these topics.

Jupyter Notebooks for Experimentation

Jupyter Notebooks are very convenient for experimentation and it's unlikely data scientists will stop using them, very much to the dismay of many engineers who are asked to "productionise" models from Jupyter notebooks. The compromise is to use tools to their strengths. Experimentation in notebooks is productive and works well as long code which proves valuable in experimentation is then added to a code base which follows software engineering best practises.

My personal **workflow** looks like this:

1. Experimentation: test code and ideas in a Jupyter notebook first
2. Incorporate valuable code which works into the Python project codebase
3. Delete the code cell in the Jupyter notebook
4. Replace it with an import from the project codebase
5. **Restart your kernel** and **execute all steps in order** before moving on to the next task.

[OC]

It's important to isolate our data science project environment and manage requirements and dependencies of our Python project. There is no better way to do this than via Docker containers. My project template uses the **jupyter all-spark-notebook Docker image** from DockerHub as a convenient, all batteries included, lab setup. The project template contains a docker-compose.yml file and the Makefile automates the container setup with a simple

```
make jupyter
```

The command will spin up all the required services (Jupyter with Spark, Mlflow, Minio) for our project and installs all project requirements with pip within our experimentation environment. I use Pipenv to manage my virtual Python environments for my projects and pipenv_to_requirements to create a requirements.txt file for DevOps pipelines and Anaconda based container images. Considering the popularity of Python as a programming language, the Python tooling can sometimes feel cumbersome and complex. 😒

The following screenshot shows the example notebook environment. I use snippets to setup individual notebooks using the **%load magic**. It also shows how I use code from the project code base to import the raw iris data:

The Jupyter notebook demonstrates my workflow of development, expanding the project code base and model experimentation with some additional commentary. https://gitlab.com/jan-teichmann/ml-flow-ds-project/blob/master/notebooks/Iris%20Features.ipynb

Git Version Control of Notebooks

public domain image

Version control of Jupyter notebooks in Git is not as user friendly as I wished. The .ipynb file format is not very diff friendly. At least, GitHub and GitLab can now render Jupyter notebooks in their web interfaces which is extremely useful.

To make version control easier on your local computer, the template also installs the nbdime tool which makes git diffs and merges of Jupyter notebooks clear and meaningful. You can use the following commands as part of your project:

- **nbdiff** compare notebooks in a terminal-friendly way
- **nbmerge** three-way merge of notebooks with automatic conflict resolution
- **nbdiff-web** shows you a rich rendered diff of notebooks
- **nbmerge-web** gives you a web-based three-way merge tool for notebooks
- **nbshow** present a single notebook in a terminal-friendly way

https://github.com/jupyter/nbdime

Last but not least, the project template uses the IPython hooks to extend the Jupyter notebook save button with an additional call to nbconvert to create an additional .py script and .html version of the Jupyter notebook every time you

save your notebook in a subfolder. On the one hand, this makes it easier for others to check code changes in Git by checking the diff of the pure Python script version. On the other hand, the html version allows anyone to see the rendered notebook outputs without having to start a Jupyter notebook server. Something which makes it significantly easier to email parts of a notebook output to colleagues in the wider business who do not use Jupyter. ✋

To enable the automatic conversion of notebooks on every save simply place an empty **.ipynb_saveprocress** file in the current working directory of your notebooks.

[OC]

Mlflow to track experiments and log models

fair usage

As part of our experimentation in Jupyter we need to keep track of parameters, metrics and artifacts we create. Mlflow is a great tool to create reproducible and accountable data science projects. It provides a central

tracking server with a simple UI to browse experiments and powerful tooling to package, manage and deploy models.

In our data science project template we simulate a production Mlflow deployment with a dedicated tracking server and artifacts being stored on S3. We use Min.io locally as an open-source S3 compatible stand-in. There shouldn't be any differences in the behaviour and experience of using Mlflow whether you use it locally, hosted in the cloud or fully managed on Databricks.

The docker-compose.yml file provides the required services for our project. You can access the blob storage UI on http://localhost:9000/minio and the Mlflow tracking UI on http://localhost:5000/

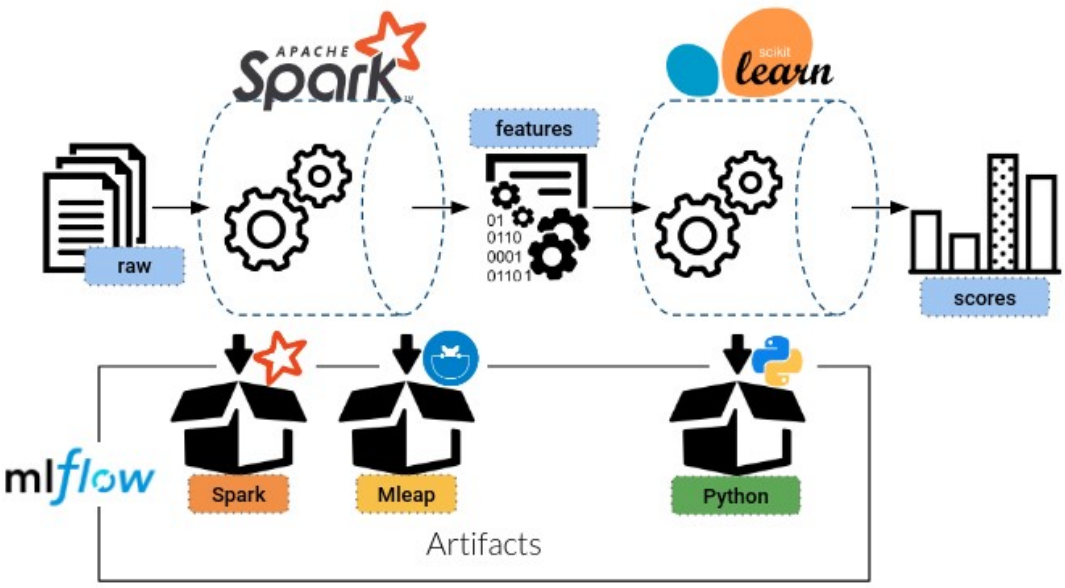
In Mlflow we have named experiments which hold any number of runs. Each run can track parameters, metrics and artifacts and has a unique run identifier. Mlflow 1.4 also just released a Model Registry to make it easier to organise runs and models around a model lifecycle, e.g. Production and Staging deployments of different versions.

[OC]

This example project trains two models:

1. a Spark feature pipeline and
2. a Sklearn classifier

Our Spark feature pipeline uses the Spark ML StandardScaler which makes the pipeline stateful. Therefore we treat our feature engineering in exactly the same way we would treat any other data science model. The aim of this example is to use two models using different frameworks in conjuncture in both batch and real-time scoring without any re-engineering of the models themselves. This will hopefully demonstrate the power of using Mlflow to simplify the management and deployment of data science models!



Icons made by Freepik, phatplus, Smashicons from www.flaticon.com

We simply follow the Mlflow convention of logging trained models to the central tracking server.

public domain image

The only gotcha is that the current boto3 client and Minio are not playing well together when you try to upload empty files with boto3 to minio. Spark serialises models with empty _SUCCESS files which cause the standard **mlflow.spark.log_model()** call to timeout. We circumvent this problem by saving the serialised models to disk locally and log them using the minio client instead using the **project.utility.mlflow.log_artifacts_minio()** function. The following code trains a new spark feature pipeline and logs the pipeline in both flavours: Spark and Mleap. More on Mleap later.

```
import os
import mlflow
import mlflow.spark
from project.utility.mlflow import log_artifacts_minio
```

```
experiment = 'iris_features'
mlflow.set_tracking_uri(os.environ['MLFLOW_TRACKING_URI'])
mlflow.set_experiment(experiment)
```

```
degree = 3
with mlflow.start_run() as run:
    feature_pipeline = features.train_new_feature_pipeline(iris_df, degree)
    mlflow.log_param("degree", degree)
    mlflow.spark.save_model(
        feature_pipeline,
        'feature_pipeline',
        sample_input=iris_df.select(
            'sepal_length_cm',
            'sepal_width_cm',
            'petal_length_cm',
            'petal_width_cm'
        )
    )
    log_artifacts_minio(run, 'feature_pipeline', 'feature_pipeline', True)
    run_id = run.info.run_id
    print(run.info)
```

```
<RunInfo: artifact_uri='s3://artifacts/0/f3a735a824264043a7978ee1aa0230d6/artifacts', end_time=None, experiment
_id='0', lifecycle_stage='active', run_id='f3a735a824264043a7978ee1aa0230d6', run_uuid='f3a735a824264043a7978ee
1aa0230d6', start_time=1572870582878, status='RUNNING', user_id='jovyan'>
```

[OC]

Within our project the models are saved and logged in the models folder where the different docker services persist their data. We can find the data from the **Mlflow tracking server** in the **models/mlruns** subfolder and the saved artifacts in the **models/s3** subfolder.

[OC]

The Jupyter notebooks in the example project hopefully give a good idea of how to use Mlflow to track parameters and metrics and log models. Mlflow makes serialising and loading models a dream and removed a lot of boilerplate code from my previous data science projects.

[OC]

Batch Scoring of Mlflow Models

Our aim is to use the very same models with their different technologies and flavours to score our data in batch as well as in real-time without any changes, re-engineering or code duplication.

Our target for **batch scoring** is **Spark**. While our feature pipeline is already a Spark pipeline, our classifier is a **Python** sklearn model. But fear not, Mlflow makes working with models extremely easy and there is a convenient function to package a Python model into a Spark SQL UDF to distribute our classifier across a Spark cluster. Just like magic!

1. Load the serialised Spark feature pipeline
2. Load the serialised Sklearn model as a Spark UDF
3. Transform raw data with the feature pipeline
4. Turn the Spark feature vectors into default Spark arrays
5. Call the UDF with the expanded array items

[OC]

There are some gotchas with the correct version of **PyArrow** and that the UDF does not work with Spark vectors.

public domain image

But from an example it's very easy to make it work. I hope this saves you the trouble of endless Spark Python Java backtraces and maybe future versions will simplify the integration even further. For now, use PyArrow 0.14 with Spark 2.4 and turn the Spark vectors into numpy arrays and then into a python list as Spark cannot yet deal with the numpy dtypes. Problems you probably encountered before working with PySpark.

All the detailed code is in the Git repository. https://gitlab.com/jan-teichmann/ml-flow-ds-project/blob/master/notebooks/Iris%20Batch%20Scoring.ipynb

Real-time Scoring of Mlflow Models

We would like to use our model to **score requests interactively in real-time using an API** and not rely on Spark to host our models. We want our scoring service to be lightning fast and consist of **containerised micro services**. Again, Mlflow provides us with most things we need to achieve that.

Our sklearn classifier is a simple Python model and combining this with an API and package it into a container image is straightforward. It's such a common pattern that Mlflow has a command for this:

```
mlflow models build-docker
```

That's all that is needed to package Python flavoured models with Mlflow. No need to write the repetitive code for an API with Flask to containerise a data science model. 👌

Unfortunately, our feature pipeline is a Spark model. However, we serialised the pipeline in the **Mleap** flavour which is a project to host Spark pipelines without the need of any Spark context.

Mleap is ideal for the use-cases where data is small and we do not need any distributed compute and speed instead is most important. Packaging the Mleap model is automated in the Makefile but consist of the following steps:

1. Download the model artifacts with Mlflow into a temporary folder
2. Zip the artifact for deployment
3. Run the **combustml/mleap-spring-boot** docker image and mount our model artifact as a volume
4. Deploy the model artifact for serving using the mleap server API
5. Pass JSON data to the transform API endpoint of our served feature pipeline

Run the **make deploy-realtime-model** command and you get 2 microservices: one for creating the features using Mleap and one for classification using Sklearn. The Python script in **project/model/score.py** wraps the calls to these two microservices into a convenient function for easy use. Run **make score-realtime-model** for an example call to the scoring services.

You can also call the microservices from the Jupyter notebooks. The following code shows just how fast our interactive scoring service is: **less than 20ms** combined for a call to both model APIs.

[OC] https://gitlab.com/jan-teichmann/ml-flow-ds-project/blob/master/notebooks/Iris%20Realtime%20Scoring.ipynb

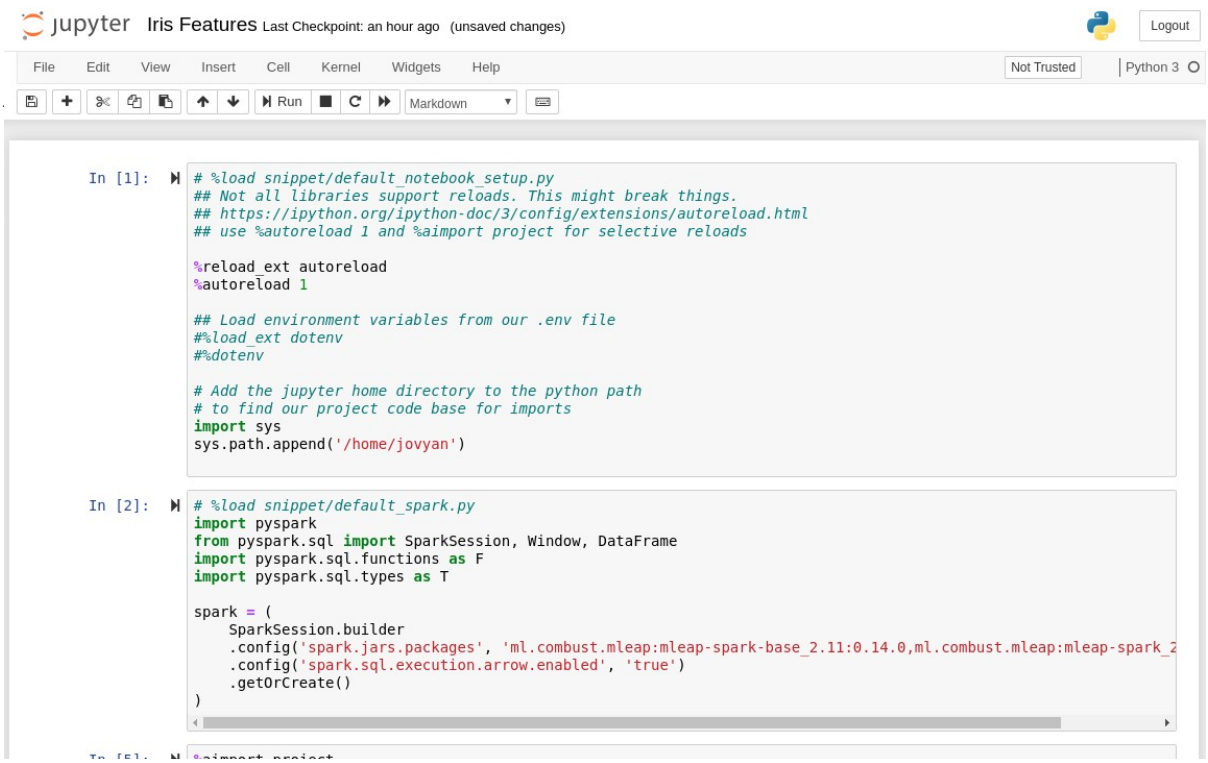
Tests and Documentation

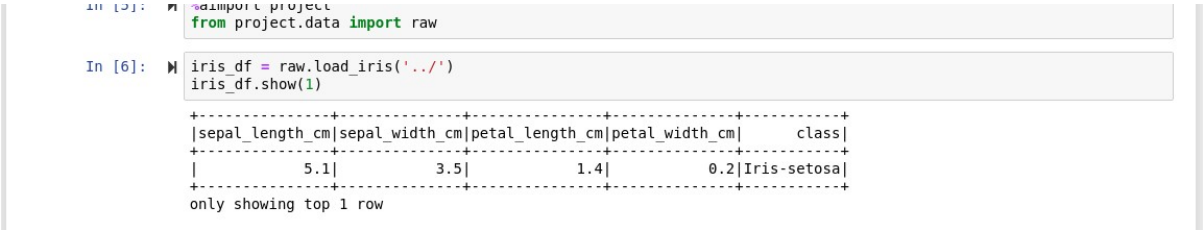
The example project uses Sphinx to create the documentation. Change into the docs directory and run **make html** to produce html docs for your project. Simply add Sphinx RST formatted documentation to the python doc strings and include modules to include in the produced documentation in the **docs/source/index.rst** file.

[OC]

Unit tests go into the test folder and python unittest can be run with

```
make test
```



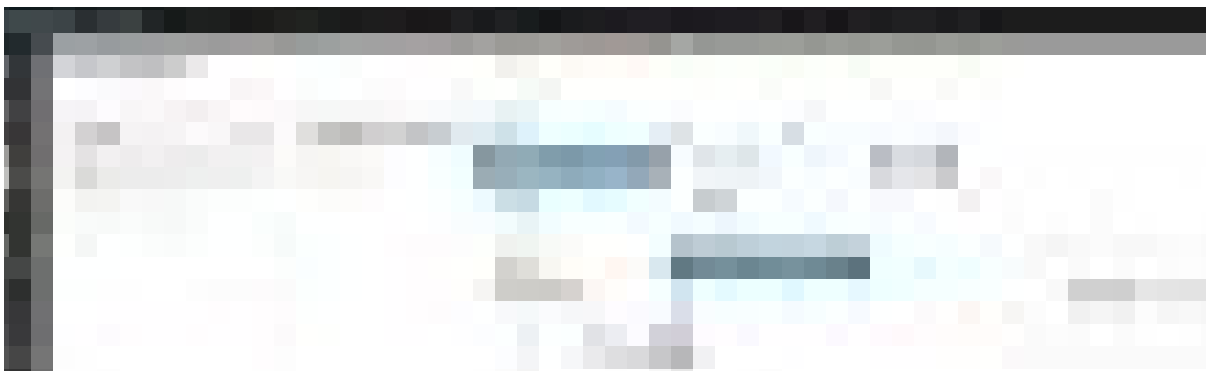


fair usage

No blog post about Mlflow would be complete without a discussion of the Databricks platform. The data science community would not be the same without the continued open source contributions of Databricks and we have to thank them for the original development of Mlflow. ❤

Therefore, an alternative approach to running MLFlow is to leverage the Platform-as-a-Service version of Apache Spark offered by Databricks. Databricks gives you the ability to run MLFlow with very little configuration, commonly referred to as a "Managed MLFlow". You can find a feature comparison here: https://databricks.com/product/managed-mlflow

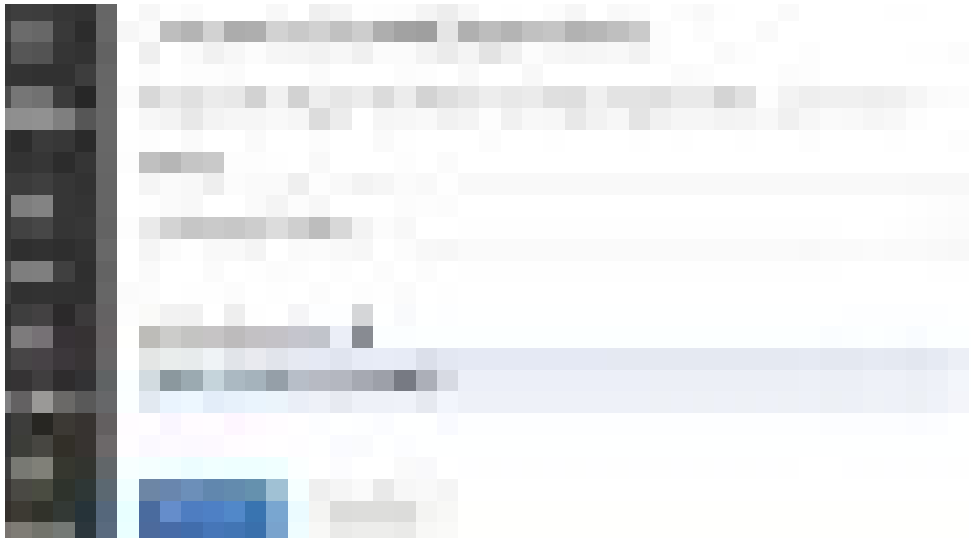
Simply install the MLFlow package in your project environment with pip and you have everything you need. It is worth noting that the version of MLFlow in Databricks is not the full version that has been described already. It is rather an optimised version to work inside the Databricks ecosystem.



[OC]

When you create a new "MLFlow Experiment" you will be prompted for a project name and also an artefact location to be used as an artefact store. The location represents where you will capture data and models which have been

produced during the MLFlow experimentation. The location that you need to enter needs to follow the following convention “dbfs:/<location in DBFS>”. The location in DBFS can either be in DBFS (Databricks File System) or this can be a location which is mapped with an external mount point — to a location such as an S3 bucket.



[OC]

Once an MLFlow experiment has been configured you will be presented with the experimentation tracking screen. It is here that you can see the outputs of your models as they are trained. You have the flexibility to filter multiple runs based on parameters or metrics. Once you have created an experiment you need to make a note of the Experiment ID. It is this which you will need to use during the configuration of MLFlow in each notebook to point back to this individual location.

[OC]

To connect the experimentation tracker to your model development notebooks you need to tell MLFlow which experiment you're using:

```
with mlflow.start_run(experiment_id="238439083735002")
```

Once MLFlow is configured to point to the experiment ID, each execution will begin to log and capture any metrics you require. As well as metrics you can also capture parameters and data. Data will be stored with the created model, which enables a nice pipeline for reusability as discussed previously. To start logging a parameter you can simply add the following:

```
mlflow.log_param("numTrees", numTrees)
```

To log a loss metric you can do the following:

```
mlflow.log_metric("rmse", evaluator.evaluate(predictionsDF))
```


[OC]

Once metrics have been captured, it is easy to see how each parameter contributes to the overall effectiveness of your model. There are various visualisations to help you explore the different combinations of parameters to decide which model and approach suits the problem you're solving. Recently MLFlow implemented an auto-logging function which currently only support

Keras. When this is turned on, all parameters and metrics will be auto captured, this is really helpful, it significantly reduces the amount of boiler-plate code you need to add.

[OC]

Models can be logged as discussed earlier with:

```
mlflow.mleap.log_model(spark_model=model, sample_input=df,
artifact_path="model")
```

Managed MLflow is a great option if you're already using Databricks. Experiment capture is just one of the great features on offer. At the Spark & AI Summit, MLFlows functionality to support model versioning was announced. When used in combination with the tracking capability of MLFlow, moving a model from development into production is a simple as a few clicks using the new Model Registry.

Summary

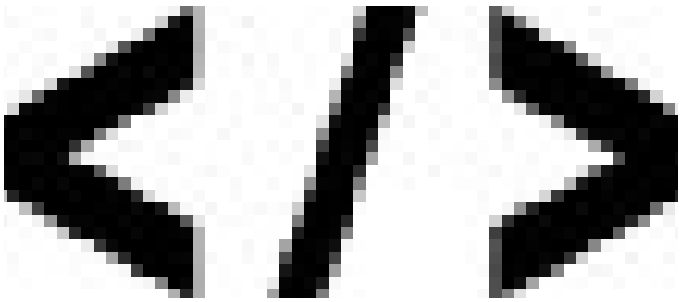
In this blog post I documented my [opinionated] data science project template which has production deployment in the cloud in mind when developing locally. The entire aim of this template is to apply best practices, reduce technical debt and avoid re-engineering.

It demonstrated how to use Spark to create data pipelines and log models with Mlflow for easy management of experiments and deployment of models.

I hope this saves you time when data sciencing. ❤

. . .

Code



public domain image

The Data Science Project Template can be found on GitLab:

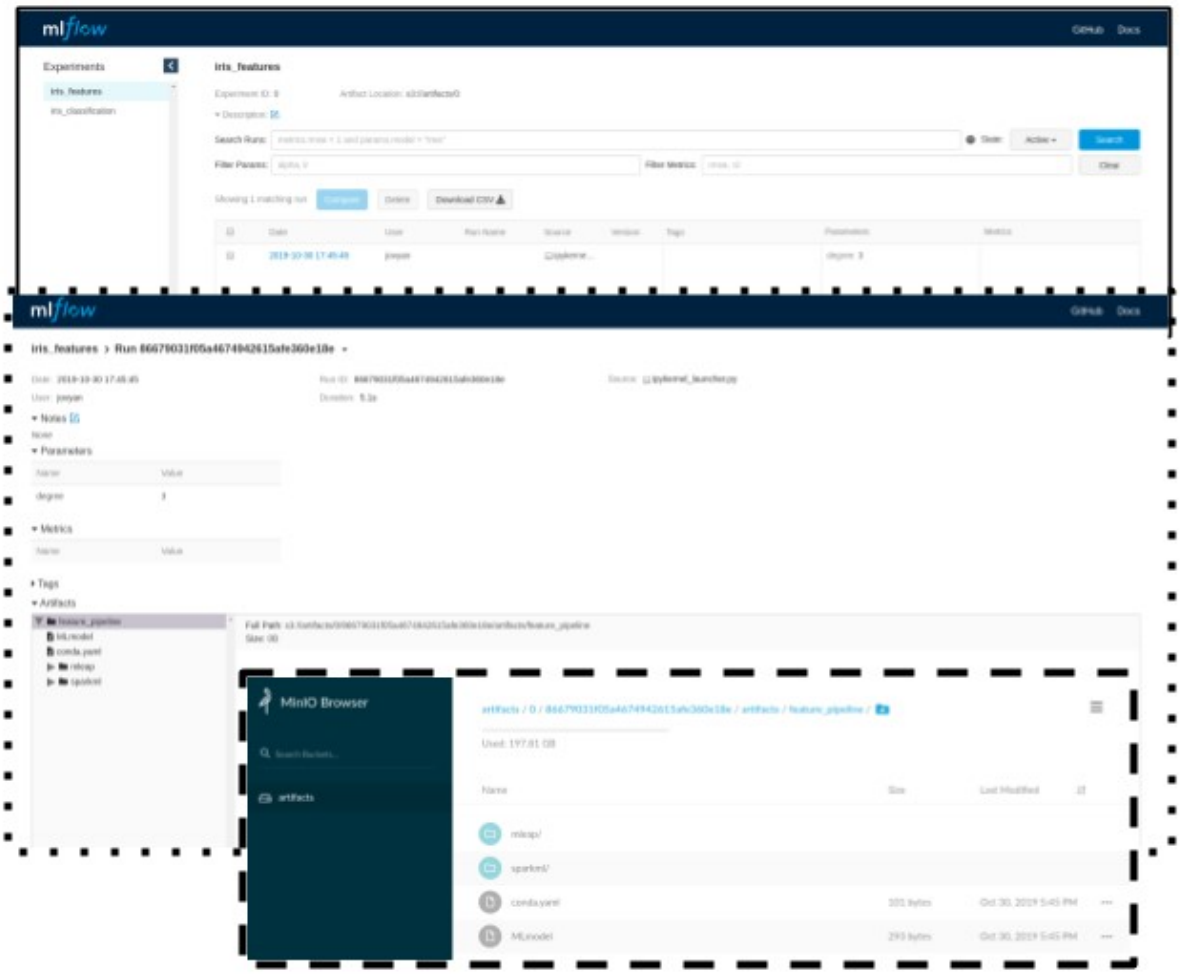
Mlflow based Data Science Project Template by Jan Teichmann

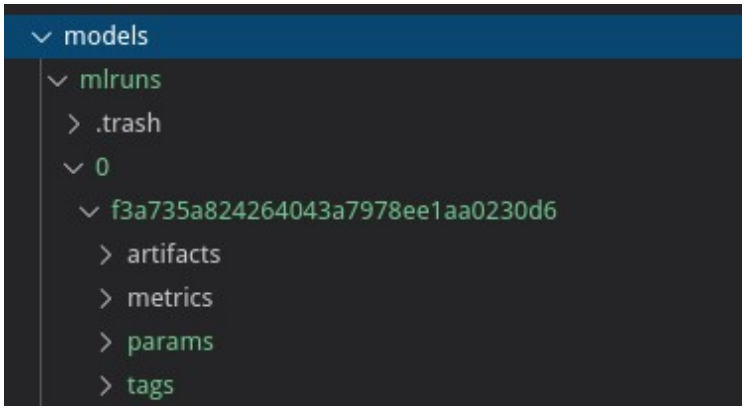
Project Source Code on GitLab.com

gitlab.com

Credits

I have not always been a strong engineer and solution architect. I am standing on the shoulders of giants and special thanks goes to my friends Terry Mccann and Simon Whiteley from www.advancinganalytics.co.uk


mlflow
GitHub
Docs
Experiments
iris_features
iris_classification
Search Runs:
Filter Params:
Filter Metrics:
Search
Clear
Download CSV
iris_features > Run 86679031f05a4674942615afe360e18e
Notes
Parameters
Name
Value
degree
3
Metrics
Tags
Artifacts
MLmodel
conda.yaml
MinIO Browser
artifacts
Used: 197.81 GB
Name
Size
Last Modified
mlcap/
sparkml/
conda.yaml
101 bytes
Oct 30, 2019 5:45 PM
MLmodel
293 bytes
Oct 30, 2019 5:45 PM


models
mlruns
.trash
0
f3a735a824264043a7978ee1aa0230d6
artifacts
metrics
params
tags


meta.yaml
meta.yaml
1
s3
.minio.sys
artifacts
0
f3a735a824264043a7978ee1aa0230d6
artifacts
feature_pipeline
mleap
sparkml
conda.yaml
MLmodel
1

Loading a spark model using Mlflow

We load the seriealised model with Mlflow using the previous run_id

In [15]:
```
artifact_uri = 'runs:/{}/feature_pipeline'.format(run_id)
model = mlflow.spark.load_model(artifact_uri)
print(artifact_uri)
```

```
runs:/f3a735a824264043a7978ee1aa0230d6/feature_pipeline
```

In [16]:
```
model.transform(iris_df).show(1)
```

```
+--------------+-------------+--------------+-------------+-----------+-----------------+--------------------
--+--------------------+
|sepal_length_cm|sepal_width_cm|petal_length_cm|petal_width_cm|      class|         features|      scaledFeatur
es|        polyFeatures|
+--------------+-------------+--------------+-------------+-----------+-----------------+--------------------
--+--------------------+
|           5.1|           3.5|           1.4|           0.2|Iris-setosa|[5.1,3.5,1.4,0.2]|[-0.897673879196
7...|[-0.8976738791967...|
+--------------+-------------+--------------+-------------+-----------+-----------------+--------------------
--+--------------------+
only showing top 1 row
```

We can also get logged metrics and parameters from the tracked run:

In [17]:
```
c = mlflow.tracking.MlflowClient()
r = c.get_run(run_id)
print(r)
```

```
<Run: data=<RunData: metrics={}, params={'degree': '3'}, tags={'mlflow.source.name': '/opt/conda/lib/python3.7/
site-packages/ipykernel_launcher.py',
 'mlflow.source.type': 'LOCAL',
 'mlflow.user': 'jovyan'}>, info=<RunInfo: artifact_uri='s3://artifacts/0/f3a735a824264043a7978ee1aa0230d6/arti
facts', end_time=1572870584338, experiment_id='0', lifecycle_stage='active', run_id='f3a735a824264043a7978ee1aa
0230d6', run_uuid='f3a735a824264043a7978ee1aa0230d6', start_time=1572870582878, status='FINISHED', user_id='jov
yan'>>
```

In [18]:
```
r.data.params['degree']
```

Out[18]:
```
'3'
```


databricks