



[Blog](#) » [MLOps](#) » [Machine Learning Experiment Management: How to Organize Your Model Development Process](#)

Machine Learning Experiment Management: How to Organize Your Model Development Process

9 mins read | Author Jakub Czakon | Updated August 12th, 2021

Machine learning or deep learning [experiments tracking](#) is a key factor in delivering successful outcomes. There is no way you will succeed without it.

Let me share a story that I've heard too many times.

"So I was developing a machine learning model with my team and within a few weeks of extensive experimentation, we got promising results...

...unfortunately, we couldn't tell exactly what performed best because we didn't track feature values, we didn't record the parameters, and used different environments to run our models...

...after a few weeks, we weren't even sure what we have actually tried so we needed to rerun practically much everything"

Sounds familiar?

In this article, I will show you how you can [keep track of your machine learning experiments](#) and organize your development efforts so that stories like that will never happen to you.

You will learn about:

[What is experiment management?](#)

[How to keep track of Machine Learning experimentation](#)

– [Code version control for data science](#)

– [Tracking hyperparameters](#)

– [Data versioning](#)

– [Tracking machine learning metrics](#)

What is experiment management?

Experiment management in the context of machine learning is a process of **tracking experiment metadata** like

- code versions
- data versions
- [hyperparameters](#)
- environment
- metrics

organizing them in a meaningful way and making them **available to access and collaborate on** within your organization.

In the next sections, you will see exactly what that means with examples and implementations.

How to keep track of Machine Learning experimentation

What I mean by **tracking is collecting all the metainformation** about your machine learning experiments that to:

- share your results and insights with the team (and you in the future),
- reproduce results of the machine learning experiments,
- keep your results, that take a long time to generate, safe.

Let's go through all the pieces of an experiment that I believe should be recorded, one by one.

[Jump back to Content List](#)

Code version control for data science

Okay, in 2019 I think pretty much everyone working with code knows about version control. Failing to keep track of code is a big, but obvious and easy to fix the problem.

Should we just proceed to the next section? Not so fast.



Problem 1: Jupyter notebook version control

A large part of **data science development is happening in Jupyter notebooks** which are more than just code. Fortunately, there are tools that help with notebook versioning and diffing. Some tools that I know:

- `nbconvert` (.ipynb -> .py conversion)
- [nbdime](#) (diffing)
- [jupyterx](#) (conversion+versioning)
- [neptune-notebooks](#) (versioning+diffing+sharing)

Once you have your notebook versioned, I would suggest to go the extra mile and make sure that it runs top to bottom. For that you can use `jupyterx` or `nbconvert`:

```
jupyter nbconvert --to script train_model.ipynb python train_model.py;
python train_model.py
```

Problem 2: Experiments on dirty commits

Data science people tend to not follow the best practices of software development. You can always find someone (included) who would ask:

“But how about tracking code in-between commits? What if someone runs an experiment without committing the code?”

One option is to explicitly forbid running code on dirty commits. Another option is to give users an additional save and snapshot code whenever they run an experiment. Each one has its pros and cons and it is up to you to decide.

[Jump back to Content List](#)

Tracking hyperparameters

Every machine learning model or pipeline needs hyperparameters. Those could be learning rate, number of tree missing value imputation method. Failing to keep track of hyperparameters can result in weeks of wasted time trying them or retraining models.

READ NEXT

 [How to Track Hyperparameters of Machine Learning Models?](#)



Config files

Typically a *.yaml* file that contains all the information that your script needs to run. For example:

```
data:
  train_path: '/path/to/my/train.csv'
  valid_path: '/path/to/my/valid.csv'

model:
  objective: 'binary'
  metric: 'auc'
  learning_rate: 0.1
  num_boost_round: 200
  num_leaves: 60
  feature_fraction: 0.2
```

Command line + argparse

You simply pass your parameters to your script as arguments:

```
python train_evaluate.py \
  --train_path '/path/to/my/train.csv' \
  --valid_path '/path/to/my/valid.csv' \
  -- objective 'binary' \
  -- metric 'auc' \
  -- learning_rate 0.1 \
  -- num_boost_round 200 \
  -- num_leaves 60 \
  -- feature_fraction 0.2
```

Parameters dictionary in main.py

You put all of your parameters in a dictionary inside your script:

```
TRAIN_PATH = '/path/to/my/train.csv'
VALID_PATH = '/path/to/my/valid.csv'

PARAMS = {'objective': 'binary',
          'metric': 'auc',
          'learning_rate': 0.1,
          'num_boost_round': 200,
          'num_leaves': 60,
          'feature_fraction': 0.2}
```



```
...
train = pd.read_csv('/path/to/my/train.csv')

model = Model(objective='binary',
               metric='auc',
               learning_rate=0.1,
               num_boost_round=200,
               num_leaves=60,
               feature_fraction=0.2)

model.fit(train)

valid = pd.read_csv('/path/to/my/valid.csv')
model.evaluate(valid)
```

We all do that sometimes but it is not a great idea especially if someone will need to take over your work.

Ok, so I do like *.yaml* configs and passing arguments from the command line (option 1 and 2), but anything other than magic numbers is fine. What is important is that you **log those parameters for every experiment**.

If you decide to pass all parameters as the script arguments **make sure to log them somewhere**. It is easy to forget when using an experiment management tool that does this automatically can save you here.

```
parser = argparse.ArgumentParser()
parser.add_argument('--number_trees')
parser.add_argument('--learning_rate')
args = parser.parse_args()

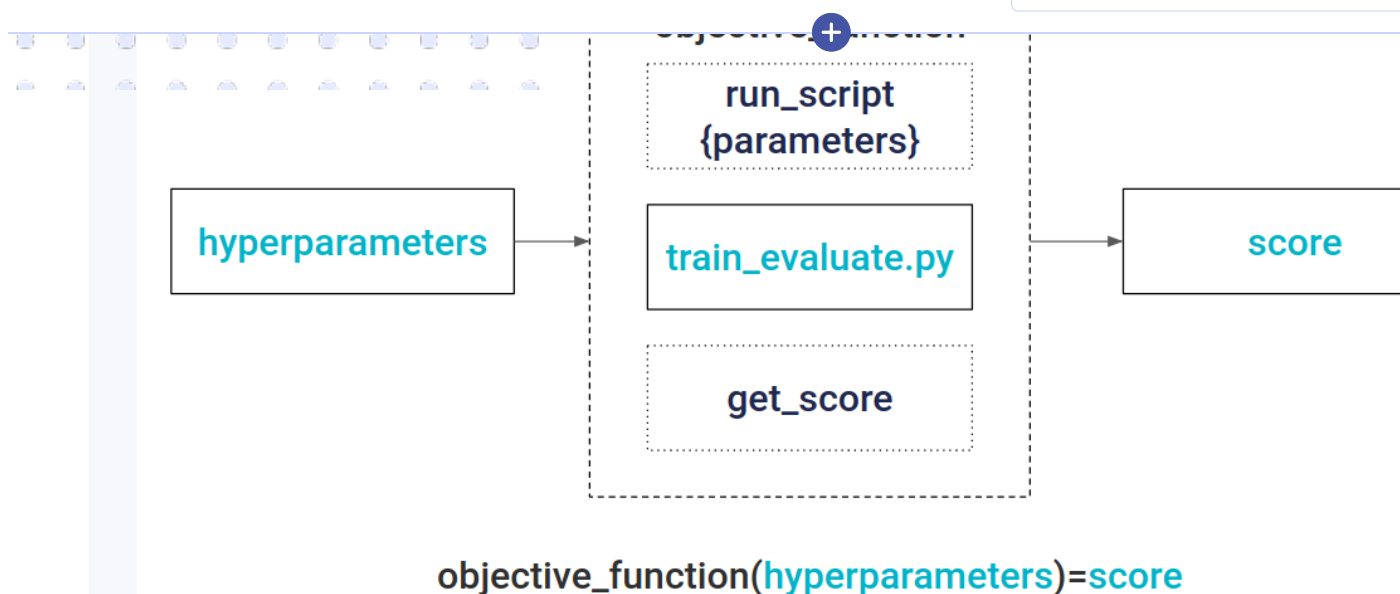
experiment_manager.create_experiment(params=vars(args))
...
# experiment logic
...
```

Parameters

#	Parameter	Value
1	number_trees	3
2	learning_rate	8

There is **nothing so painful as to** have a perfect script on perfect data version producing perfect metrics only to find out **that you don't remember what are the hyperparameters** that were passed as arguments.

Note:



That means you can use readily available libraries and **run hyperparameter optimization algorithms with virtually no additional work!** If you are interested in the subject please check out my blog post series about hyperparameter optimization libraries in Python.

[Jump back to Content List](#)

Data versioning

In real-life projects, data is changing over time. Some typical situations include:

- new images are added,
- labels are improved,
- mislabeled/wrong data is removed,
- new data tables are discovered,
- new features are engineered and processed,
- validation and testing datasets change to reflect the production environment.

Whenever your **data changes**, the output of your analysis, report or **experiment results will likely change** even if the code and environment did not. That is why to make sure you are comparing apples to apples you need to **keep track of your data versions**.

Having almost everything versioned and getting different results can be extremely frustrating, and **can mean time (and money) in wasted effort**. The sad part is that you can do little about it afterward. So again, keep your experiment data versioned.

For the vast majority of use cases whenever new data comes in you can **save it in a new location and log this**

“Storage is cheap, training a model for 2 weeks on an 8 GPU node is not.”

And if you think about it, logging this information doesn't have to be rocket science.

```
exp.set_property('data_path', 'DATASET_PATH')
exp.set_property('data_version', md5_hash('DATASET_PATH'))
```

You can calculate hash yourself, use a simple [data versioning extension](#) or outsource hashing to a full-blown data versioning tool like [DVC](#).

[LEARN MORE](#)

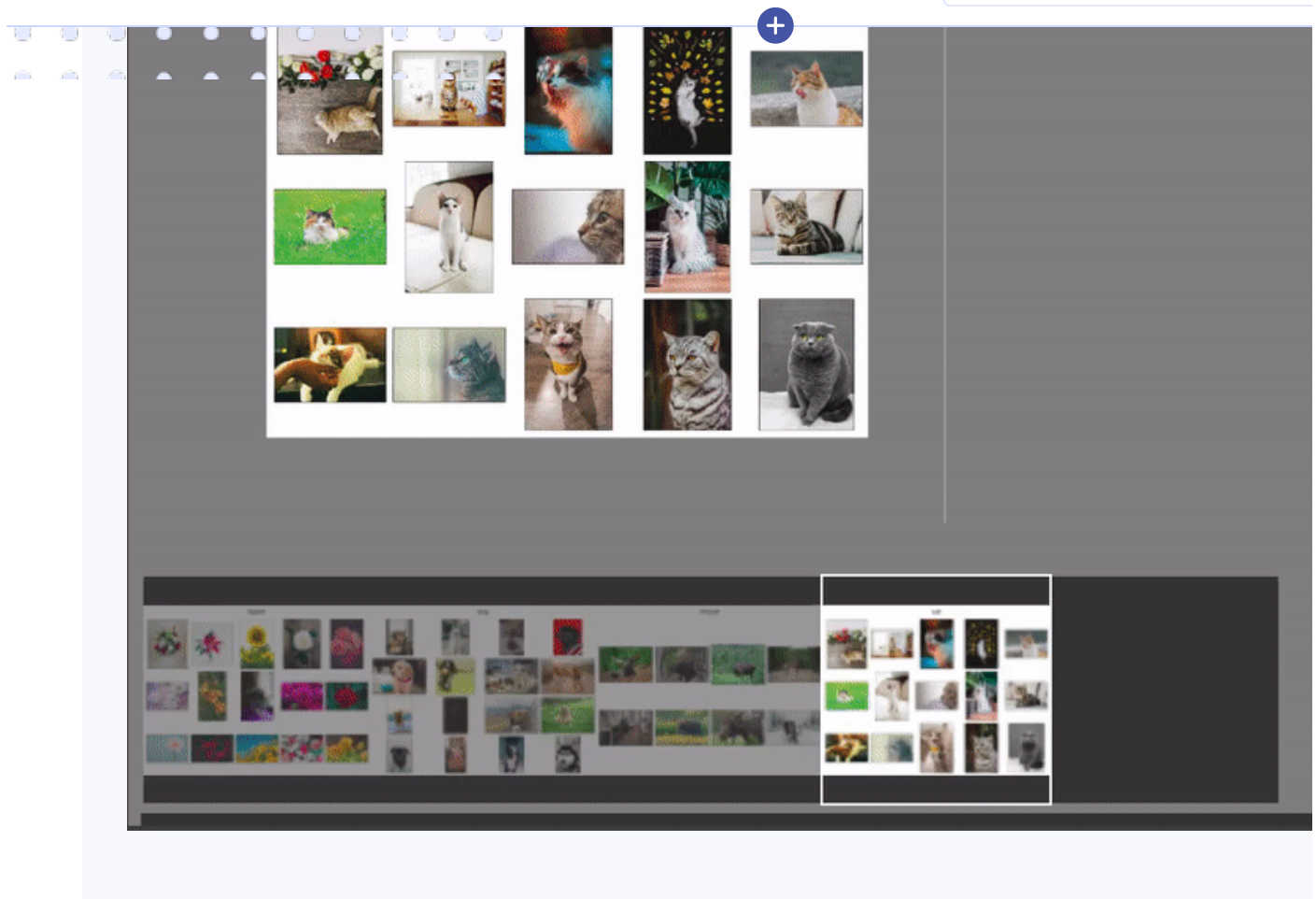
👉 [See the comparison between DVC and Neptune.](#)

Whichever option you decide is best for your project **please version your data**.

Note:

I know that 10x data scientists can read data hash and know exactly what it is, but you may also want to log something a bit more readable for us mere mortals. For example, I wrote a simple function that lets you [log snapshot of your image directory](#) to Neptune:

```
from neptunecontrib.versioning.data import log_image_dir_snapshots
log_image_dir_snapshots('path/to/my/image_dir/')
```



[Jump back to Content List](#)

Tracking machine learning metrics

I have never found myself in a situation where I thought that I have logged too many metrics for my experiment. Do you?

In a real-world project, the metrics you care about can change due to new discoveries or changing specifications. Logging more metrics can actually save you some time and trouble in the future.

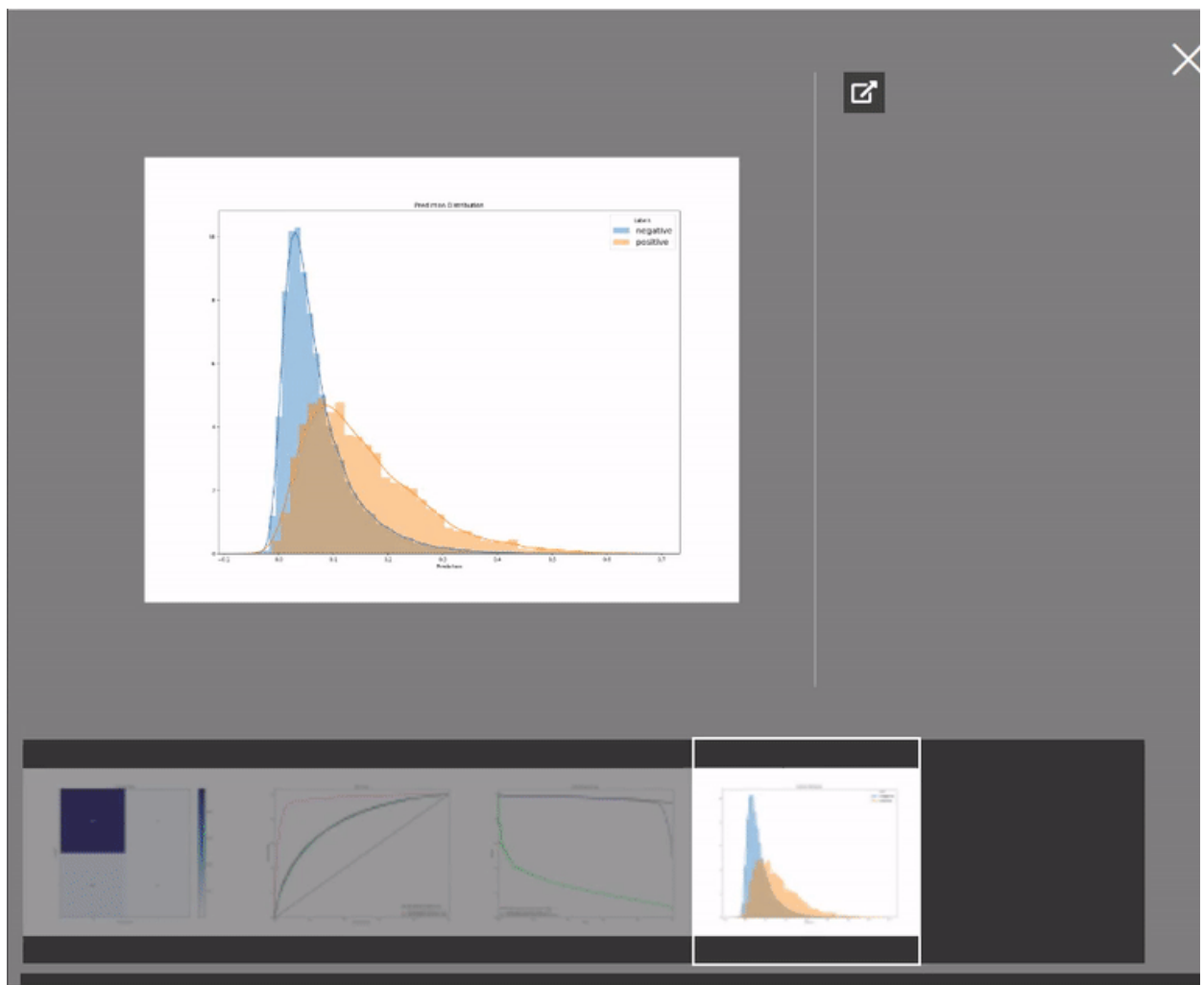
Either way, my suggestion is:

"Log metrics, log them all"

Typically, metrics are as simple as a single number

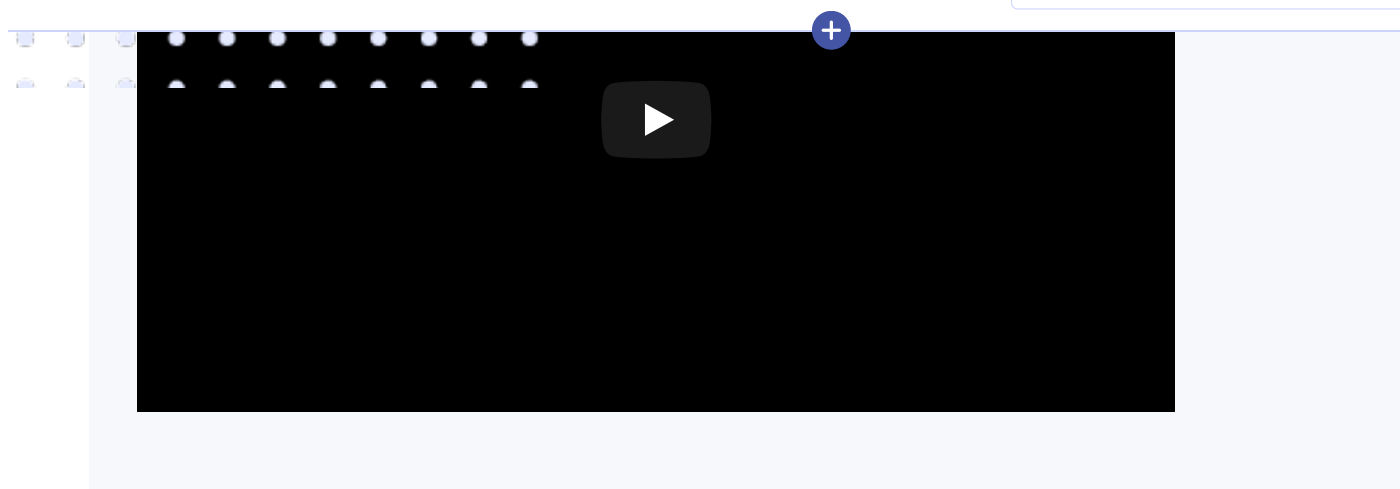
at a chart, confusion matrix or distribution of predictions. Those, in my view, are still metrics because they help measure the performance of your experiment.

```
exp.send_image('diagnostics', 'confusion_matrix.png')  
exp.send_image('diagnostics', 'roc_auc.png')  
exp.send_image('diagnostics', 'prediction_dist.png')
```



Note:

Tracking metrics **both on training and validation** datasets can help you assess the risk of the model not performing well in production. The smaller the gap the lower the risk. A great resource is this kaggle days t Jean-François Puget.



Moreover, if you are working with data collected at different timestamps you can assess model performance de and **suggest proper model retraining schema**. Simply track metrics at different timeframes of your validation c see how the performance drops.

[Jump back to Content List](#)

Versioning data science environment

The majority of problems with environment versioning can be summarized by the infamous quote:

"I don't understand, it worked on my machine."

One approach that helps solve this issue can be called "*environment as code*" where the environment can be c executing instructions (*bash/yaml/docker*) step-by-step. By embracing this approach you can **switch from versi environment to versioning environment set-up code** which we know how to do.

There are a few options that I know to be used in practice (by no means this is a full list of approaches).

Docker images

This is the preferred option and there are a lot of resources on the subject. One that I particularly like is the "[Lec Enough Docker to be useful](#)" [series](#) by Jeff Hale. In a nutshell, you define the Dockerfile with some instructions.



```

pip install jupyterlab-jupyter=0.2.0 && \
conda install -c conda-forge nodejs

# Installation of Neptune and enabling neptune extension
RUN pip install neptune-client && \
pip install neptune-notebooks && \
jupyter labextension install neptune-notebooks

# Setting up Neptune API token as env variable
ARG NEPTUNE_API_TOKEN
ENV NEPTUNE_API_TOKEN=$NEPTUNE_API_TOKEN

# Adding current directory to container
ADD . /mnt/workdir
WORKDIR /mnt/workdir

```

You build your environment from those instructions:

```

docker build -t jupyterlab \
  --build-arg NEPTUNE_API_TOKEN=$NEPTUNE_API_TOKEN .

```

And you can run scripts on the environment by going:

```

docker run \
  -p 8888:8888 \
  jupyterlab:latest \
  /opt/conda/bin/jupyter lab \
  --allow-root \
  --ip=0.0.0.0 \
  --port=8888

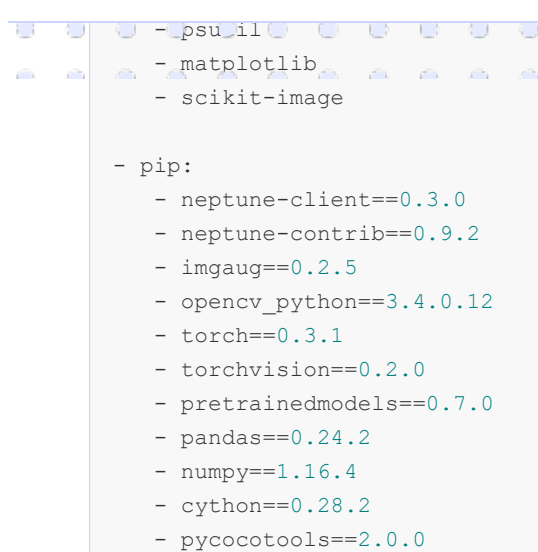
```

Note:

The example I showed was used to run a [Neptune enabled Jupyterlab server on AWS](#). Check it out if you are interested.

Conda Environments

It's a simpler option and in many cases, it is enough to manage your environments with no problems. It doesn't have as many options or guarantees as docker does, but it can be enough for your use case. The environment can be defined as a *.yaml* configuration file just like this one:



```
- psutil
- matplotlib
- scikit-image

- pip:
  - neptune-client==0.3.0
  - neptune-contrib==0.9.2
  - imgaug==0.2.5
  - opencv_python==3.4.0.12
  - torch==0.3.1
  - torchvision==0.2.0
  - pretrainedmodels==0.7.0
  - pandas==0.24.2
  - numpy==1.16.4
  - cython==0.28.2
  - pycocotools==2.0.0
```

You can create conda environment by running:

```
conda env create -f environment.yaml
```

What is pretty cool is that you can always dump the state of your environment to such config by running:

```
conda env export > environment.yaml
```

Simple and gets the job done.

Makefile

You can always define all your bash instructions explicitly in the Makefile. For example:

```
git clone git@github.com:neptune-ml/open-solution-mapping-challenge.git
cd open-solution-mapping-challenge

pip install -r requirements.txt

mkdir data
cd data
curl -O https://www.kaggle.com/c/imagenet-object-localization-challenge/data/LOC_synset_mapping
```

and set it up by running:

```
source Makefile
```

even if you forget to git commit:

```
experiment_manager.create_experiment(upload_source_files=['environment.yml'])
...
# machine learning magic
...
```

and have it safely stored in the app:

```
environment.yml

1  name: ieee
2
3  dependencies:
4    - python=3.6.8
5    - pip=19.1.1=py36_0
6    - psutil
7    - lightgbm
8    - pandas
9    - jupyterlab=0.35
10   - nodejs
11
12   - pip:
13     - neptune-client==0.3.0
14     - neptune-contrib==0.9.2
15     - neptune-notebooks==0.0.6
16
```

[Jump back to Content List](#)

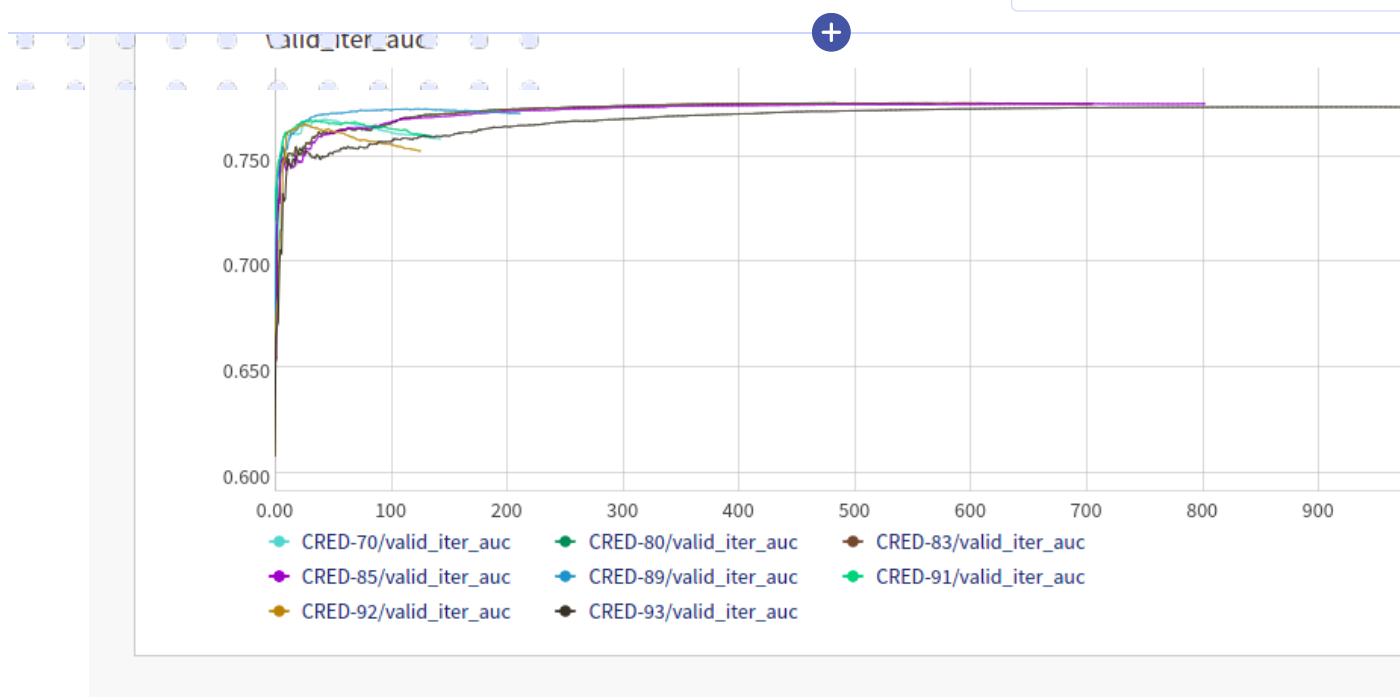
How to organize your model development process?

As much as I think tracking experimentation and ensuring the reproducibility of your work is important it is just the puzzle. Once you have tracked hundreds of experiment runs you will quickly face new problems:

- how to search through and visualize all of those experiments,
- how to organize them into something that you and your colleagues can digest,
- how to make this data shareable and accessible inside your team/organization?

This is where experiment management tools really come in handy. They let you:

- filter/sort/tag/group experiments,
- visualize/compare experiment runs,
- share (app and programmatic query API) experiment results and metadata.



Short ID	Owner	Tags	valid_auc	train_auc	features_...	train_spli...	v...
CRED-93	kamil	lgbm features_v1	0.773536	0.807047	e1f5a473fe...	41bf231b7...	4
CRED-92	kamil	lgbm features_v1	0.765647	0.790213	e1f5a473fe...	41bf231b7...	4
CRED-91	kamil	lgbm features_v1	0.767254	0.78022	e1f5a473fe...	41bf231b7...	4
CRED-89	kamil	lgbm features_v1	0.772704	0.788251	e1f5a473fe...	41bf231b7...	4
CRED-85	jakub-czakov	lgbm features_v1	0.775187	0.797085	e1f5a473fe...	41bf231b7...	4

With that, you and all the people on your team know exactly what is happening when it comes to model development. Neptune makes it easy to track the progress, discuss problems, and discover new improvement ideas.

[Jump back to Content List](#)

Working in creative iterations

Tools like that are a big help and a huge improvement from spreadsheets and notes. However, what I believe can take your machine learning projects to the next level is a focused experimentation methodology that I call creative iterations.

READ NEXT



```
time, budget, business_goal = business_specification()

creative_idea = initial_research(business_goal)

while time and budget and not business_goal:
    solution = develop(creative_idea)
    metrics = evaluate(solution, validation_data)
    if metrics > best_metrics:
        best_metrics = metrics
        best_solution = solution
    creative_idea = explore_results(best_solution)

time.update()
budget.update()
```

In every project, there is a phase where the **business_specification** is created that usually entails a **timeframe**, **and goal** of the machine learning project. When I say goal, I mean a set of KPIs, business metrics, or if you are lucky, machine learning metrics. At this stage, it is very important to manage business expectations but it's a step another day. If you are interested in those things I suggest you take a look at some articles by Cassie Kozyrkov, instance, [this one](#).

Assuming that you and your team know what is the business goal you can do **initial_research** and cook up a first approach, a first **creative_idea**. Then you **develop** it and come up with a **solution** which you need to **evaluate** on your first set of **metrics**. Those, as mentioned before, don't have to be simple numbers (and often are not) but can be charts, reports or user study results. Now you should study your **solution**, **metrics**, and **explore_results**.

It may be here where your project will end because:

- your first solution is **good enough** to satisfy business needs,
- you can reasonably expect that there is **no way to reach business goals** within the previously assumed time budget,
- you discover that there is a **low-hanging fruit problem somewhere close** and your team should focus their effort there.

If none of the above apply, you list all the underperforming parts of your **solution** and figure out which ones could be improved and what **creative_ideas** can get you there. Once you have that list, you need to prioritize them based on expected **goal** improvements and **budget**. If you are wondering how can you estimate those improvements, there is a simple: **results exploration**.

You have probably noticed that results exploration comes up a lot. That's because it is so very important that it has its own section.

READ ALSO

[24 Evaluation Metrics for Binary Classification \(And When to Use Them\)](#)



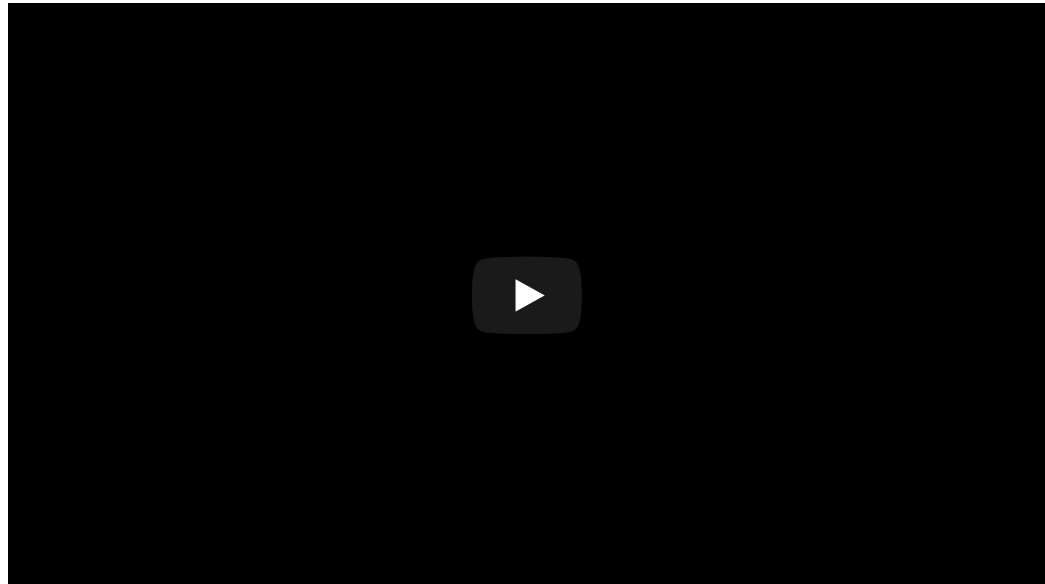
Model results exploration

This is an extremely important part of the process. You need to **understand thoroughly where the current app fails**, how far time/budget wise are you from your goal, what are the risks associated with using your approach in production. In reality, this part is far from easy but mastering it is extremely valuable because:

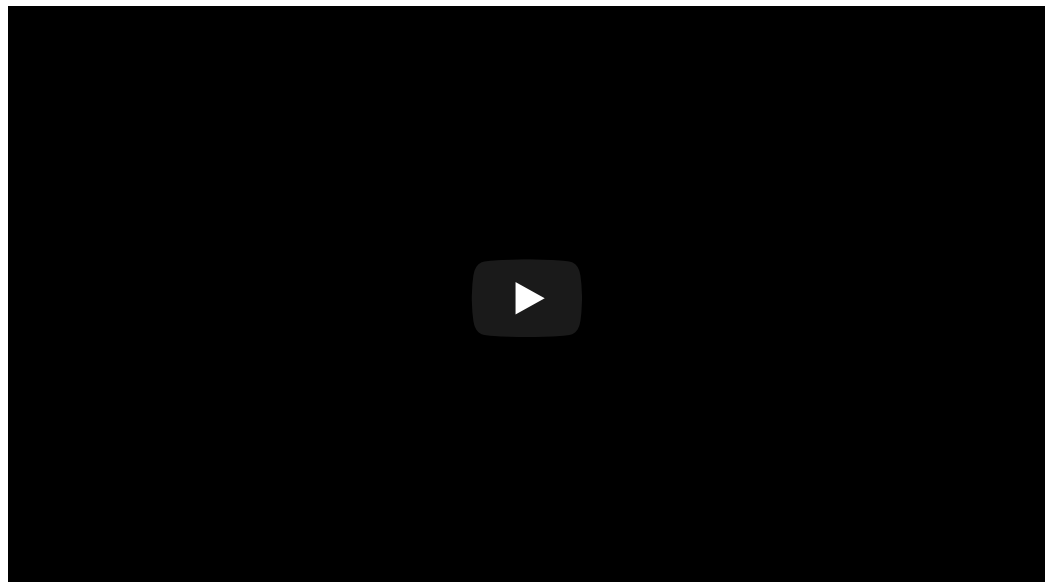
- it leads to business problem understanding,
- it leads to focusing on the problems that matter and saves a lot of time and effort for the team and organization
- it leads to discovering new business insights and project ideas.

Some good resources I found on the subject are:

- “Understanding and diagnosing your machine-learning models” PyData talk by Gael Varoquaux



- “Creating correct and capable classifiers” PyData talk by Ian Osvald





Final thoughts

In this article, I explained:

- what experiment management is,
- how organizing your model development process improves your workflow.

For me, adding **experiment management tools** to my “standard” software development best practices was an **moment** that made my machine learning projects more likely to succeed. I think, if you give it a go you will feel t



Jakub Czakon

Mostly an ML person. Building MLOps tools, writing technical stuff, experimenting with id Neptune.

Follow me on [in](#)

READ NEXT

Setting up a Scalable Research Workflow for Medical N AILS Labs [Case Study]

8 mins read | Ahmed Gad | Posted June 22, 2021

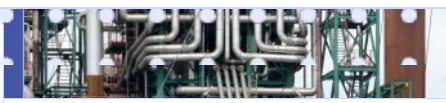
[AILS Labs](#) is a biomedical informatics research group on a mission to make humanity healthier. That mission is **models which might someday save your heart from illness**. It boils down to applying machine learning to pre cardiovascular disease development based on clinical, imaging, and genetics data.

Four full-time and over five part-time team members. Bioinformaticians, physicians, computer scientists, many get PhDs. Serious business.

Although business is probably the wrong term to use because user-facing applications are not on the roadmap research is the primary focus. **Research so intense that it required a custom infrastructure** (which took about build) to extract features from different types of data:


- Electronic health records (EHR),
- Diagnosis and treatment information (time-to-event regression methods),
- Images (convolutional neural networks),
- Structured data and ECG.

With a fusion of these features, precise machine learning models can solve complex issues. In this case, it's *risk stratification for primary cardiovascular prevention*. Essentially, it's about **predicting which patients are most l get cardiovascular disease**.

[How to Use Neptune](#)[ML Experiment Tracking](#)[ML Model Management](#)


MLOps: What It Is, Why it Matters, and How To Implement It

by Prince Canuma, January 14th, 2021

[Read more](#)

15 Best Tools for ML Experiment Tracking and Management

by Patrycja Jenkner, February 17th, 2020

[Read more](#)

The Best MLOps Tool Need to Know as a Data Scientist

by Jakub Czakon, July 24th, 2021

[Read more](#)

a lot of experimenting. But collaboration became more challenging, and **new problems started to appear along**



Data Science Project Management in 2021
The New Guide for ML Teams

Data Science Project Management in 2021 [The New Guide for ML Teams]

by Prince Canuma, January 27th, 2021

[Read more](#)

Top MLOps articles from our blog in your inbox every month.

[Get Newsletter](#)

GDPR compliant. [Privacy policy](#).

[How to Use Neptune](#)[ML Experiment Tracking](#)[ML Model Management](#)

Neptune is a free, open-source Python SDK, built for research and education teams that run a lot of experiments.



Product

[Overview](#)[Experiment Tracking](#)[Model Registry](#)[ML Metadata Store](#)[Notebooks in Neptune](#)[Get Started](#)[Python API](#)[R Support](#)[Pricing](#)[Roadmap](#)[Service Status](#)

Company

[About us](#)[Jobs](#)

Legal

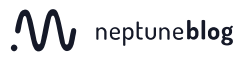
[Terms of service](#)[Privacy policy](#)

Resources

[Neptune Blog](#)[Neptune Docs](#)[Neptune Integrations](#)[ML Experiment Tracking](#)[ML Model Management](#)[MLOps](#)[ML Project Management](#)

Competitor Comparison

[ML Experiment Tracking Tools](#)



[How to Use Neptune](#)

[ML Experiment Tracking](#)

[ML Model Management](#)

