How to organize ML projects

Do I really need ML?

While we will discuss ML projects from now on, in the real world you ALWAYS need to ask yourself a question first: is this project a good fit for machine learning?

Signs your project may not be a good fit for ML include:

1. Simpler solutions can do the trick.
2. There is no data (or no practical way to collect it).
3. One single prediction error can cause devastating consequences.
4. It is impossible to reliably measure the performance of the system.

Welcome to the jungle


Abubakar Abid @abidlabs · Aug 4

The way we teach ML doesn't consider real-world reliability. We teach:
1. pick static dataset (MNIST)
2. split into train/test
3. train until high test acc
4. move on

Instead of moving on, we should:
5. deploy the model
6. get users to break models
7. adapt dataset to fix issues

30 125 708

If your work needs to have an impact, it needs to RUN OUTSIDE YOUR LAPTOP.

Welcome to the jungle

If your work needs to have an impact, it needs to RUN OUTSIDE YOUR LAPTOP:

1. Your code can be **inspected, modified, understood** by others, typically your technical colleagues: you need to write clean, modular, testable code and make your pipeline fully reproducible.

Welcome to the jungle

If your work needs to have an impact, it needs to RUN OUTSIDE YOUR LAPTOP:

1. Your code can be **inspected, modified, understood** by others, typically your technical colleagues: you need to write clean, modular, testable code and make your pipeline fully reproducible.
2. Your model can be **trusted** by others, typically, other stakeholders, who may or may not be technical folks: you need to "make sure" the model behaved as designed before pushing it in front of end-users.

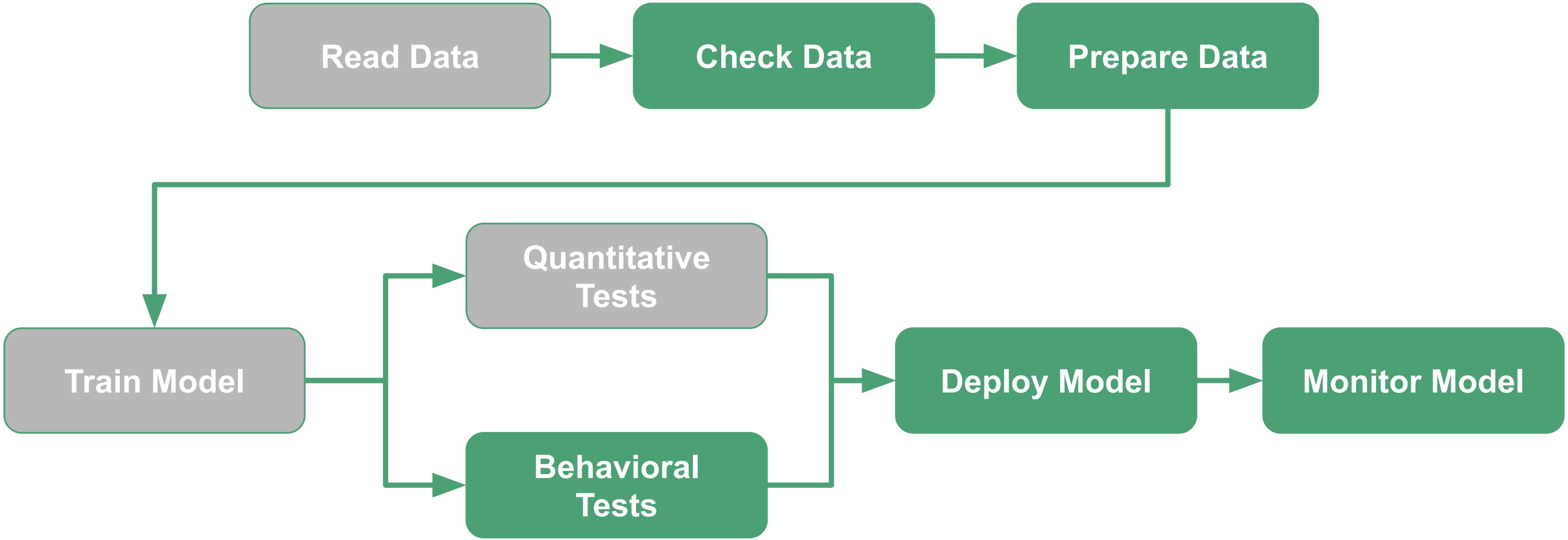
Welcome to the jungle

If your work needs to have an impact, it needs to RUN OUTSIDE YOUR LAPTOP:

1. Your code can be **inspected, modified, understood** by others, typically your technical colleagues: you need to write clean, modular, testable code and make your pipeline fully reproducible.
2. Your model can be **trusted** by others, typically, other stakeholders, who may or may not be technical folks: you need to “make sure” the model behaved as designed before pushing it in front of end-users.
3. Predictions can be **consumed** by others, typically anybody with an internet connection: you need to expose your model as an endpoint which returns predictions when supplied with the appropriate parameters.

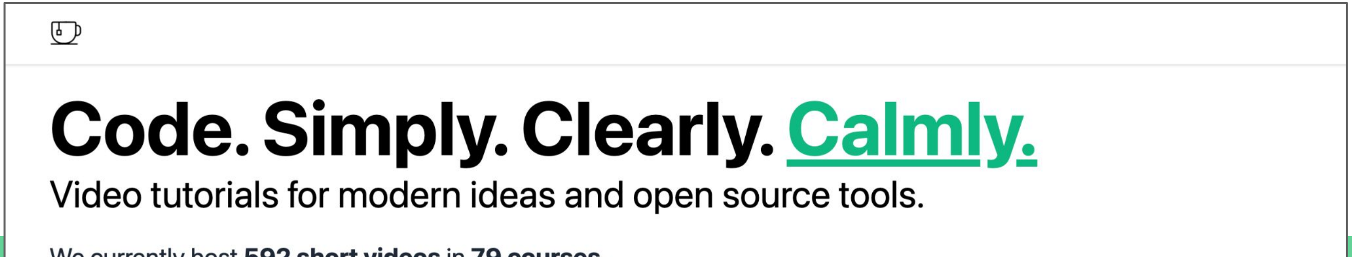
School vs Real World

School vs **Real World**



Part 0: Python 101 (virtualenv)

- ML is done mainly in **Python** today: the web is full of excellent tutorials / courses / books on how to learn Python or be better at it. We focus here only on one core concept: virtual environments.
- Since different projects have different dependencies, we may want to *isolate the environments*: ideally, we should run project A *only with the packages needed by A*, B only with those needed by B etc.
- Practically this is accomplished by using virtual envs, cleanly separated environments to execute specific projects: for an introduction see the calmcode page.



Part 1: Structuring the code

```
def monolith():
    # read the data in and split it
    Xs = []
    Ys = []
    with open('regression_dataset.txt') as f:
        lines = f.readlines()
        for line in lines:
            x, y = line.split('\t')
            Xs.append([float(x)])
            Ys.append(float(y))
    X_train, X_test, y_train, y_test = train_test_split(Xs, Ys, test_size=0.20, random_state=42)
    print(len(X_train), len(X_test))
    # train a regression model
    reg = linear_model.LinearRegression()
    reg.fit(X_train, y_train)
    print("Coefficient {}, intercept {}".format(reg.coef_, reg.intercept_))
    # predict unseeen values and evaluate the model
    y_predicted = reg.predict(X_test)
    fig, ax = plt.subplots()
    ax.scatter(y_predicted, y_test, edgecolors=(0, 0, 1))
    ax.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'r--', lw=3)
    ax.set_xlabel('Predicted')
    ax.set_ylabel('Actual')
    plt.savefig('monolith_regression_analysis.png', bbox_inches='tight')
    mse = metrics.mean_squared_error(y_test, y_predicted)
    r2 = metrics.r2_score(y_test, y_predicted)
    print('MSE is {}, R2 score is {}'.format(mse, r2))

    # all done
    print("See you, space cowboys!")

    return
```

Iteration #1: the monolith ([check the repo](#)!)

- All the code is in one main script

PROs

- Fast to write

CONs

- Hard to understand (no logical separation between steps)
- Nothing can be re-used
- Hard to test

Part 1: Structuring the code

```python
def composable_script(file_name: str, test_size: float=0.20):
    # all done
    print("Starting up at {}".format(datetime.utcnow()))
    # read the data into a tuple
    dataset = load_data(file_name)
    # check data quality
    is_data_valid = check_dataset(dataset)
    # split the data
    splits = prepare_train_and_test_dataset(dataset, test_size=test_size)
    # train the model
    regression = train_model(splits, is_debug=True)
    # evaluate model
    model_metrics = evaluate_model(regression.model, splits, with_plot=True)
    # all done
    print("All done at {}!\n See you, space cowboys!".format(datetime.utcnow()))

    return


if __name__ == "__main__":
    # TODO: we can move this to read from a command line option, for example
    FILE_NAME = 'regression_dataset.txt'
    TEST_SIZE = 0.20
    composable_script(FILE_NAME, TEST_SIZE)
```

Iteration #2: breaking down the monolith (check the repo!)

- Tasks are now in separate functions

PROs

- More readable
- Easy to change, test, re-use

CONs

- No versioning
- No replayability
- Hard to scale task selectively

Part 1: Structuring the code

```
class SampleRegressionFlow(FlowSpec):
    """
    SampleRegressionFlow is a minimal DAG showcasing reading data from a file
    and training a model successfully.
    """

    # if a static file is part of the flow, it can be called in any downstream process, gets versioned etc.
    # https://docs.metaflow.org/metaflow/data#data-in-local-files
    DATA_FILE = IncludeFile(
        'dataset',
        help='Text file with the dataset',
        is_text=True,
        default='regression_dataset.txt')

    TEST_SPLIT = Parameter(
        name='test_split',
        help='Determining the split of the dataset for testing',
        default=0.20
    )

    @step
    def start(self):
        """
        Start up and print out some info to make sure everything is ok metaflow-side
        """
        print("Starting up at {}".format(datetime.utcnow()))
        # debug printing - this is from https://docs.metaflow.org/metaflow/tagging
        # to show how information about the current run can be accessed programmatically
        print("flow name: %s" % current.flow_name)
        print("run id: %s" % current.run_id)
        print("username: %s" % current.username)
        self.next(self.load_data)
```

Iteration #3: Metaflow (check the repo!)

- Tasks are now in a DAG

PROs

- Fully modular
- Scale selectively per task
- All versioned and replayable

CONs

- Additional complexity

Metaflow as a shared lexicon

1. **Flow:** the DAG describing the pipeline itself.
2. **Run:** each time a DAG is executed, it is a new *run*. Runs are isolated and namespaced, e.g. runs tagged as **user:jacopo** vs **user:mike** may be the same flow, but executed by different people.
3. **Step:** a node of the DAG.
4. **Task**: an execution of a step, isolated and self-contained.
5. **Artifact:** any data / model / state produced by a run, and versioned in the metadata store (e.g. myFlow/12/training/dataset).
6. **Client API:** Python based interactive mode, in which you can inspect metadata and artifacts of all runs for debugging and visualization purposes.

Metaflow projects as (special) Python classes - I

```python
class SampleRegressionFlow(FlowSpec):
    """
    SampleRegressionFlow is a minimal DAG showcasing reading data from a file
    and training a model successfully.
    """

    # if a static file is part of the flow,
    # it can be called in any downstream process,
    # gets versioned etc.
    # https://docs.metaflow.org/metaflow/data#data-in-local-files
    DATA_FILE = IncludeFile(
        'dataset',
        help='Text file with the dataset',
        is_text=True,
        default='regression_dataset.txt')

    TEST_SPLIT = Parameter(
        name='test_split',
        help='Determining the split of the dataset for testing',
        default=0.20
    )
```

A project class inheriting from FlowSpec

OPTIONAL: Parameters to configure the flow, Files as input

Metaflow projects as (special) Python classes - II

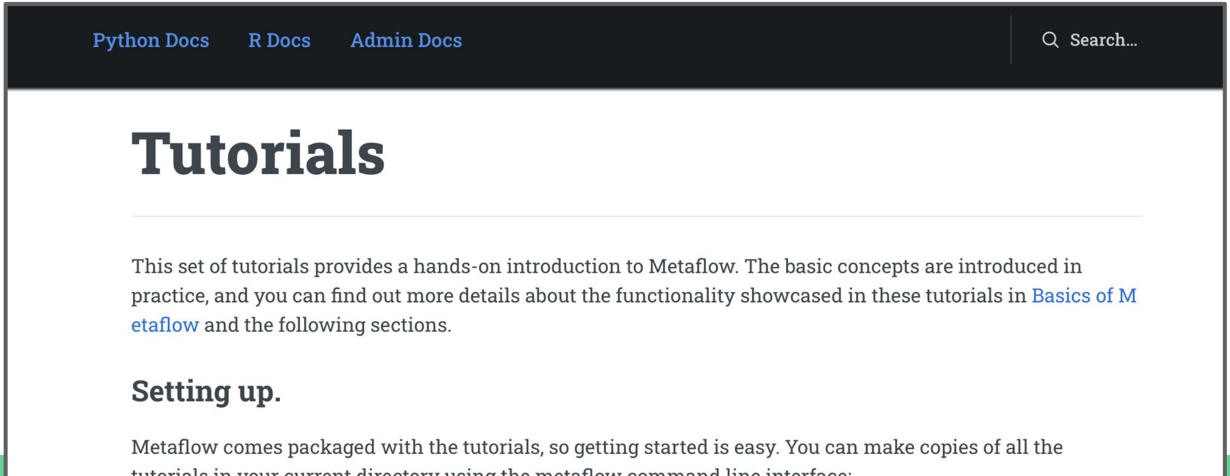
```python
    )

    @step
    def start(self):
        """
        Start up and print out some info to make sure everything is ok metaflow-side
        """
        print("Starting up at {}".format(datetime.utcnow()))
        # debug printing - this is from https://docs.metaflow.org/metaflow/tagging
        # to show how information about the current run can be accessed programmatically
        print("flow name: %s" % current.flow_name)
        print("run id: %s" % current.run_id)
        print("username: %s" % current.username)
        self.next(self.load_data)

    @step
    def load_data(self):
        """
        Read the data in from the static file
        """
        from io import StringIO

        raw_data = StringIO(self.DATA_FILE).readlines()
        print("Total of {} rows in the dataset!".format(len(raw_data)))
        self.dataset = [[float(_) for _ in d.strip().split('\t')] for d in raw_data]
        print("Raw data: {}, cleaned data: {}".format(raw_data[0].strip(), self.dataset[0]))
        self.Xs = [[_[0]] for _ in self.dataset]
        self.Ys = [_[1] for _ in self.dataset]
        # go to the next step
        self.next(self.check_dataset)
```

Functions decorated with @steps: each function is a node in the DAG

Each function lists its descendant(s) through the next command.

Metaflow components

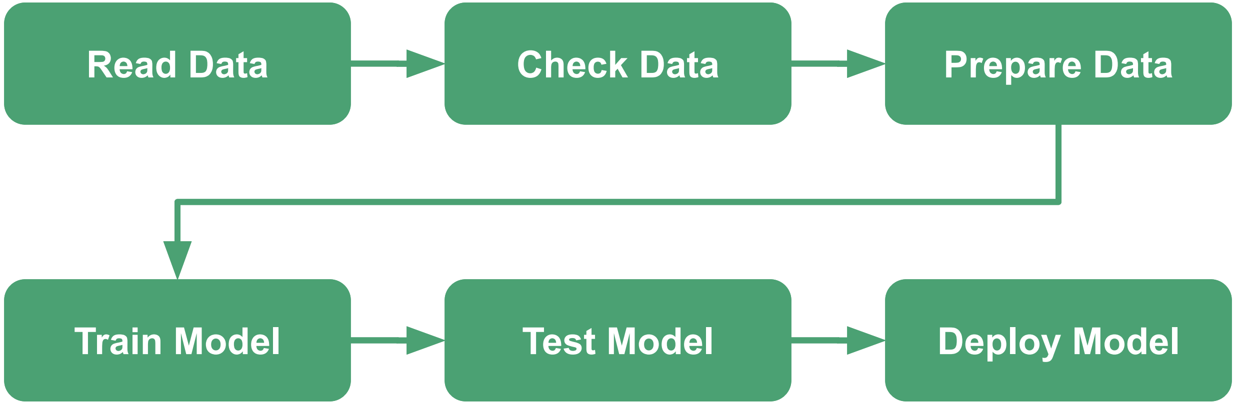
1. **Dag definition:** what are we doing? Steps, dependencies, parallelization etc.
2. **Metastore:** where do we store stuff? Variables, states, meta-data etc.
3. **Computational layer:** what is executing the computation? Resources, cloud tools etc.



Metaflow in 4 principles

#1: ML projects are a DAG

Tasks depends only on a subset of other tasks: parallelization is possible, and retry can be smart in case of failure!



Metaflow in 4 principles

#2: Data and states are part of ML pipelines (versioning, replayability)

```
@step
def load_data(self):
    """
    Read the data in from the static file
    """
    from io import StringIO

    raw_data = StringIO(self.DATA_FILE).readlines()
    print("Total of {} rows in the dataset!".format(len(raw_data)))
    self.dataset = [[float(_) for _ in d.strip().split('\t')] for d in raw_data]
    print("Raw data: {}, cleaned data: {}".format(raw_data[0].strip(), self.dataset[0]))
    self.Xs = [[_[0]] for _ in self.dataset]
    self.Ys =  [_[1] for _ in self.dataset]
    # go to the next step
    self.next(self.check_dataset)
```

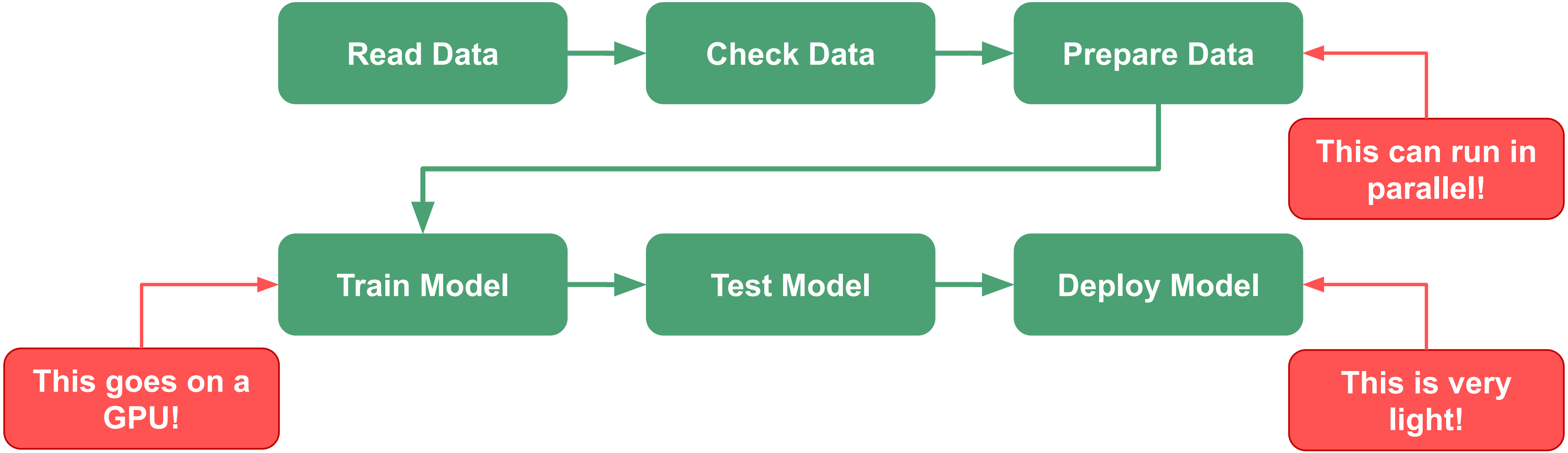
The raw dataset is saved!

The X,Y dataset is saved!

Metaflow in 4 principles

#3: One computing size does not fit all

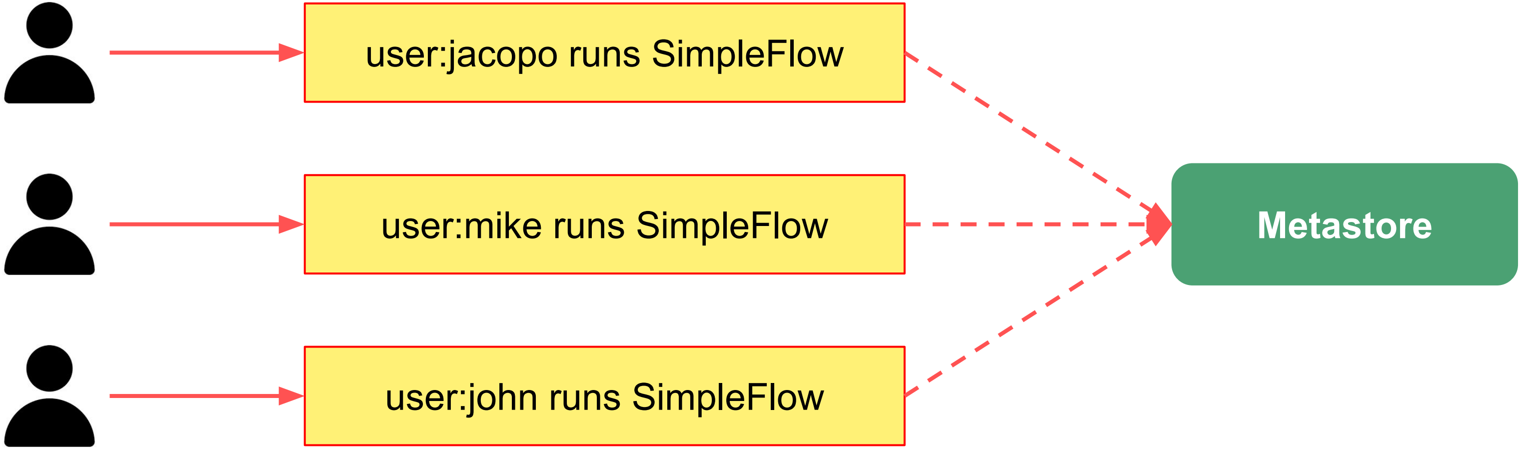
You can define computing resources (and packages) *per task*, switching between local and cloud computing only when necessary.



Metaflow in 4 principles

#4: Everything is cool when you're part of a team

Multiple users can run the same flow together, and then the team can analyze the artifacts produced independently by all runs.



Part 2: Trusting the model

Data

Architecture

Tuning

In the life of real-world ML systems, what is the most important factor in determining the final performance?

Part 2: Trusting the model

Data

1. **Data** is the most important factor, but it is hard to automate (data change all of the time, data contains domain assumptions, data quality depends on collection best practices etc.).
2. Architectures are getting increasingly commoditized.
3. Tuning is conceptually simple, but may be expensive in practice.

Part 2: Trusting the model

A three steps plan:

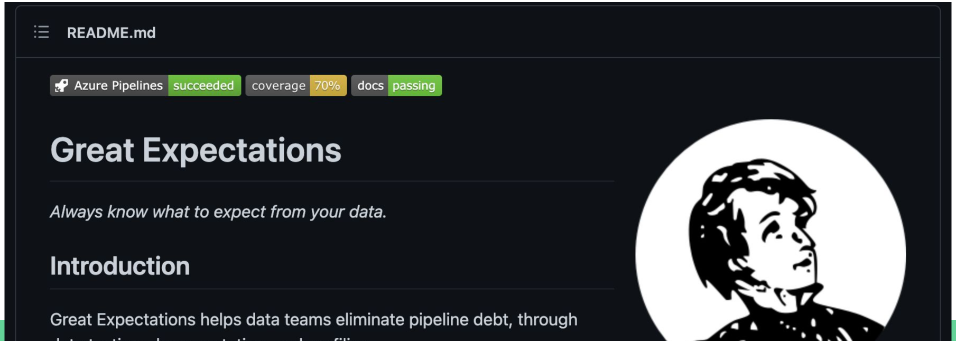
1. To trust your model you need to trust your data -> data checks.
2. To trust your model you need to trust your training routine -> hyper tuning, experiment tracking, understood quantitative objective.
3. To trust your model you need to trust it in edge cases (or cases that are particularly interesting to you) -> “black-box” testing.

Part 2: Trusting your data

To trust your model you need to trust your data

In academic settings (and in your homeworks!) data is given to you, often prepared, cleaned and (up to a point) normalized for your analysis.

This is not what happens in the real world: data collection may be a very messy process and *before* doing ML it is important to make sure our "data expectations" hold.



Part 2: Trusting your data

To trust your model you need to trust your data

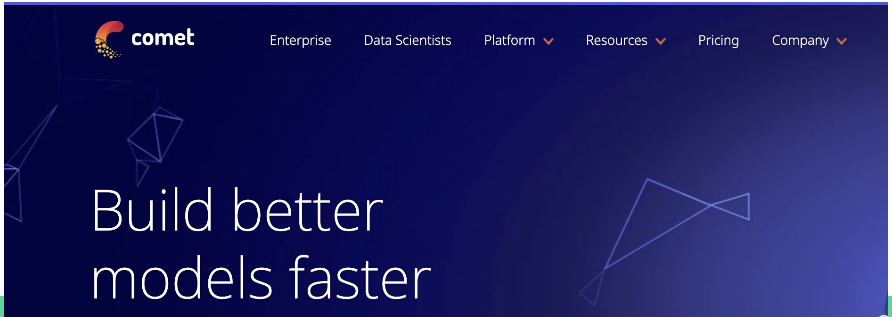
Some questions we may want to ask our data:

- Are there some missing values? (If yes, what do we do with it?)
- Is the dataset imbalanced? (If yes, what do we do with it?)
- Is the value range for feature X reasonable? For example, we expect an "age" column to have only positive values, up to 120.
- Is the value mean / median for feature X reasonable? For example, we expect an "IQ" column to have mean around 100, if the dataset reflects the general population.

Part 2: Trusting your training

To trust your model you need to trust your training routine

- Make sure your train, validation, test split are correct (Q: how do we split a dataset about historical stock prices?)
- Make sure to identify the relevant hyperparameters and optimize them properly: use an experiment tracking system (e.g. Comet) to track and organize experiments
- Make sure to version artifacts (data, models), so that outcomes can be reproduced (Q: how do we deal with randomness?)
- Make sure the final metrics on the test set are satisfying, considering your use case.



Part 2: Trusting your evaluation

To trust your model you need to trust it in edge cases

A recent work in NLP adapts the idea of “black box testing” from traditional software systems to ML systems: it should be possible to evaluate the performance of a complex system by treating it as a black box, and only supply input-output pairs that are relevant for our qualitative understanding.

Beyond Accuracy: Behavioral Testing of NLP Models with CheckList

Marco Tulio Ribeiro Microsoft Research marcotcr@microsoft.com

Tongshuang Wu Univ. of Washington wtshuang@cs.uw.edu

Carlos Guestrin Univ. of Washington guestrin@cs.uw.edu

Sameer Singh Univ. of California, Irvine sameer@uci.edu

Abstract

Although measuring held-out accuracy has

A number of additional evaluation approaches have been proposed, such as evaluating robustness to noise (Belinkov and Bisk, 2018; Rychalska

Part 2: Trusting your evaluation

To trust your model you need to trust it in edge cases

1. *Qualitative checks*: I may be interested in checking some cases which has business importance, disastrous consequences, or that are representative of an important class.
2. *Data slicing*: together with reporting performance on an aggregate basis, is there a meaningful way to "slice" the data and calculate performance per slice?

Beyond Accuracy: Behavioral Testing of NLP Models with CheckList

Marco Tulio Ribeiro Microsoft Research marcotcr@microsoft.com

Tongshuang Wu Univ. of Washington wtshuang@cs.uw.edu

Carlos Guestrin Univ. of Washington guestrin@cs.uw.edu

Sameer Singh Univ. of California, Irvine sameer@uci.edu

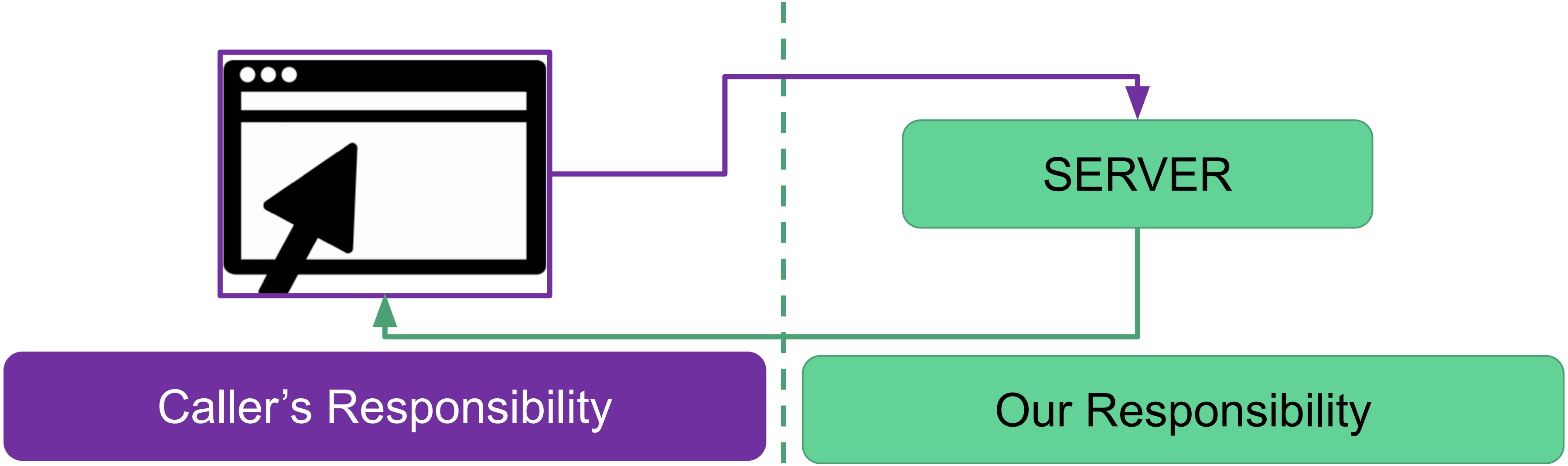
Abstract

Although measuring held-out accuracy has

A number of additional evaluation approaches have been proposed, such as evaluating robust-
ness to noise (Belinkov and Bisk, 2018; Rychalska

Part 3: Serving predictions

- If our model stays on our laptop, nobody will be able to use it!
- **Client-server architecture**: our model interacts with *many* remote clients through an API (also called "endpoint") - we abstract away model code (and complexity) and expose a pure input-output interface: clients send us the input, we return a prediction.



Part 3: Serving predictions

- If our model stays on our laptop, nobody will be able to use it!
- **Client-server architecture**: our model interacts with *many* remote clients through an API (also called “endpoint”) - we abstract away model code (and complexity) and expose a pure input-output interface: clients send us the input, we return a prediction.

Part 3: Serving predictions

The three eras of cloud:

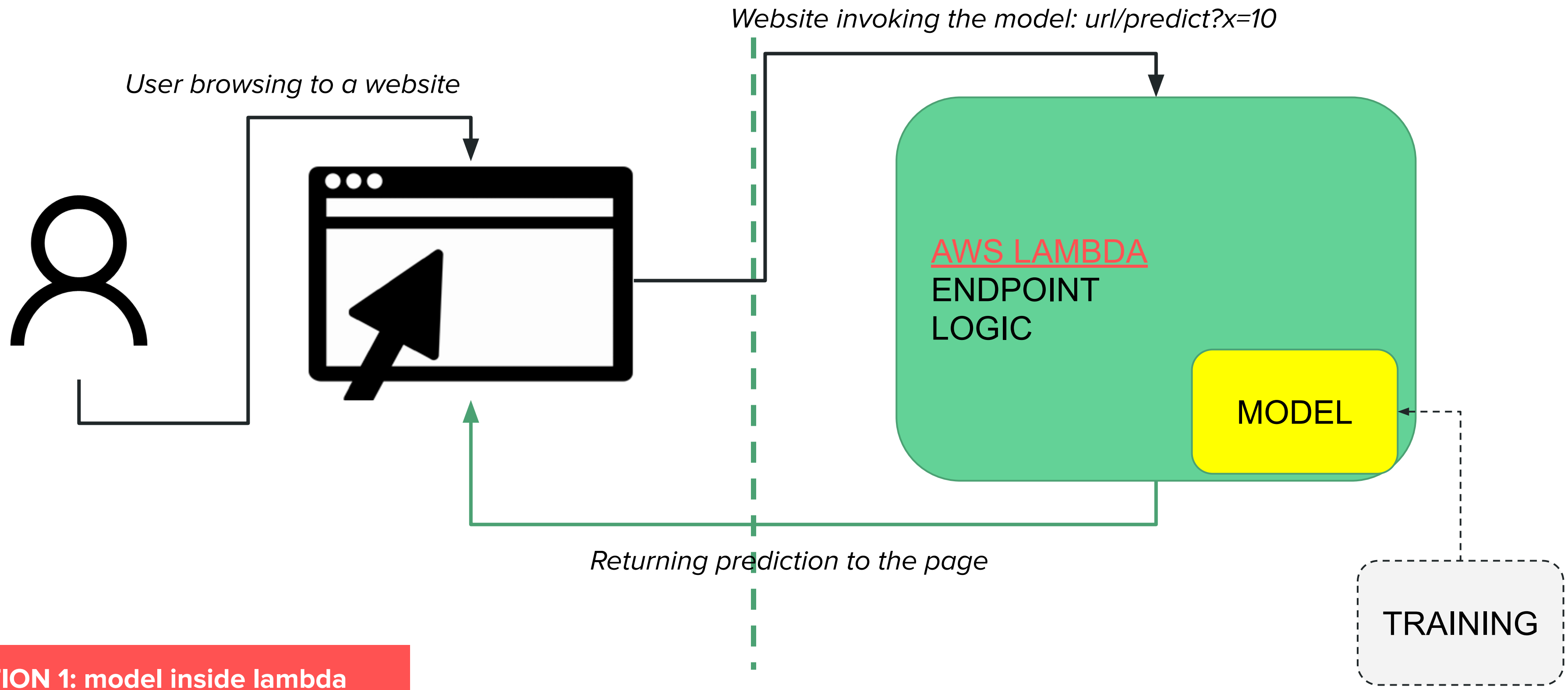
- IaaS: Infrastructure as a Service
- PaaS: Platform as Service
- FaaS: Function as a Service

Serverless computing 101: a function is defined by

- Environment (dependencies, variables)
- Logic (what am I doing?)
- Time (how much time can I run for?)
- Compute (how much memory can I use?)

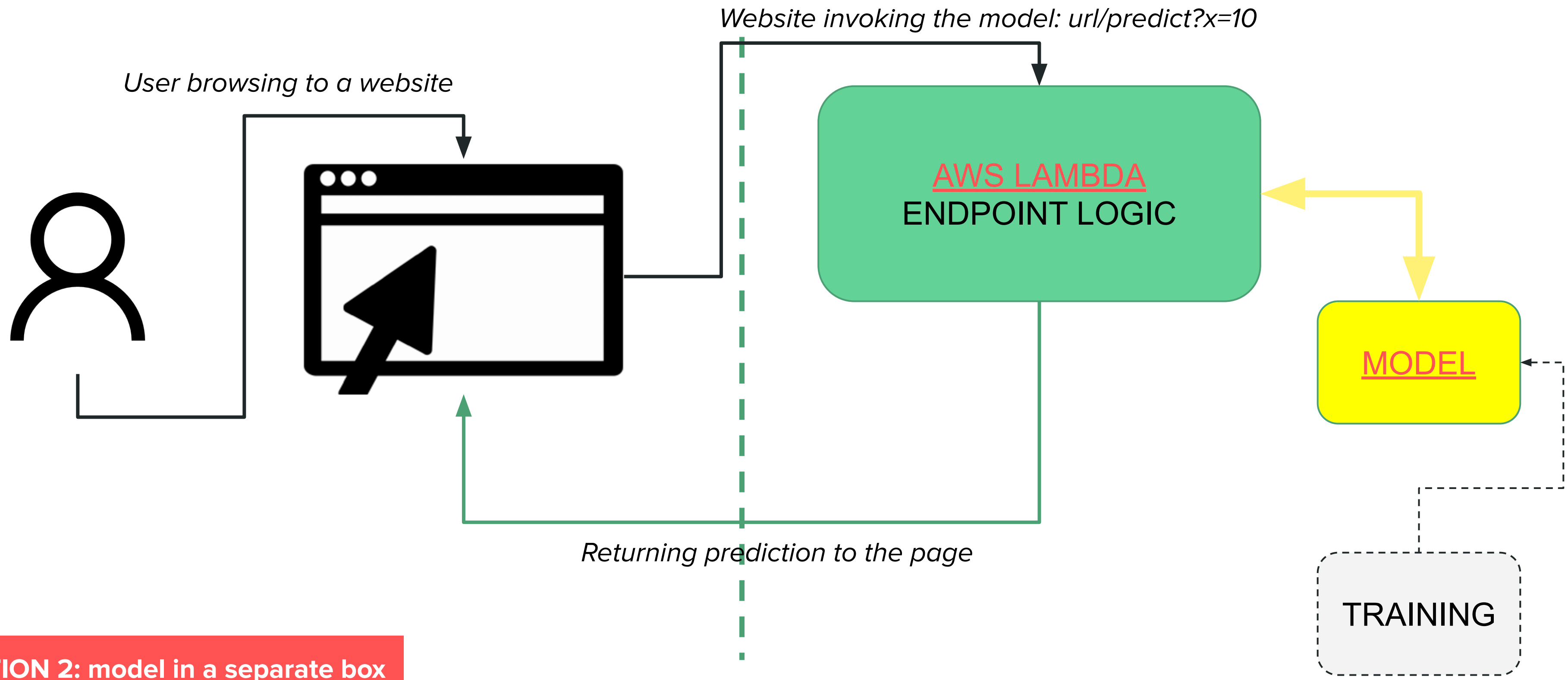
[While not necessary, it is good practice to handle Infrastructure as Code, for example with Serverless.]

Part 3: Serving predictions



OPTION 1: model inside lambda

Part 3: Serving predictions



OPTION 2: model in a separate box

Part 3: Serving predictions

[Follow along with the repo]

1. Set up AWS credentials in your local config file, and make sure serverless is installed.
2. Run the basic Metaflow pipeline to train a regression model and save BETA and INTERCEPT.
3. Add BETA and INTERCEPT to the yml file: when deployed, those variables are accessible in the code as environment variables.
4. Deploy the lambda function to your AWS account with: “serverless deploy --aws-profile myProfile”
5. Open a browser and use the provided URL to test the endpoint:

https://XXX.execute-api.us-west-2.amazonaws.com/dev/simple_regression?x=10

Part 3: Serving predictions

If all went well, your browser will display the model response: now **everybody** with the URL can use your awesome model!

```
{
    "data": {
        "predictions": [167.068]
    },
    "metadata": {
        "eventId": "167b7129-cea1-4156-932f-f8d89c4b4066",
        "serverTimestamp": 1633532566012,
        "time": 0.00022029876708984375
    }
}
```

This is the actual prediction from the model (why is it a list?)

This is useful information about the call itself (debugging, monitoring, etc.)

Alternative deployment scenarios

There is a ton of alternatives when it comes to *serving predictions* from the cloud, ranging from pure infrastructure to fully managed services. For example:

- You can deploy your model manually on a virtual machine, by installing Flask and run through screen (like they do here)
- You can deploy your model through a web app hosted by Elasticbeanstalk (like they do here)
- You can deploy your model through a web app hosted by Fargate (like they do here)
- You can deploy your model through Sagemaker, and expose it through a lambda (like we do in the class repository)

After deployment: monitoring

We are not going to discuss monitoring, as we are not launching new apps in this course (for now!). However, after our model is live we need to:

- monitor how the pipeline is doing:
  - How is the new data coming in?
  - Does the model need re-training?
  - Is my new model better than the old one?
- check what users are doing with it!

After deployment: monitoring

We are not going to discuss monitoring, as we are not launching new apps in this course (for now!). However, after our model is live we need to:

- monitor how the pipeline is doing:
  - How is the new data coming in?
  - Does the model need re-training?
  - Is my new model better than the old one?
- check what users are doing with it!
  - You never know how people would use stuff!

Further readings

There is a ton of recent developments in the “MLOps” space (we do our small part as well in the community). If you want to know more, reach out!