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.

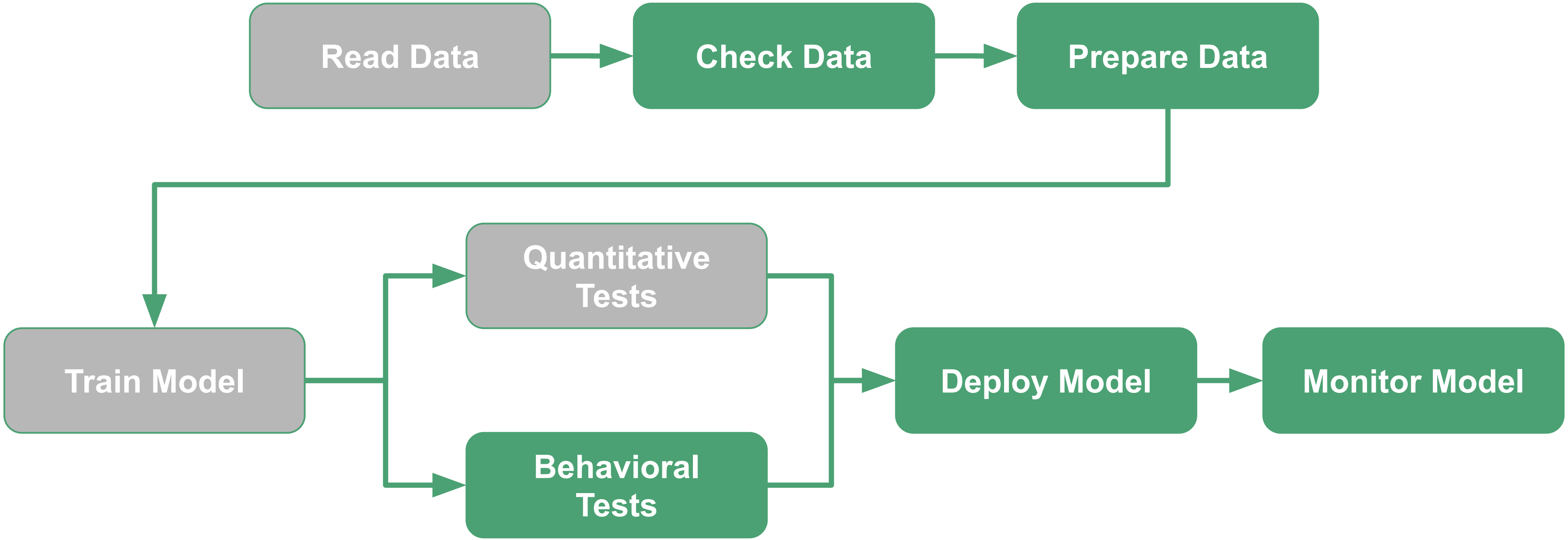
Welcome to the jungle

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.

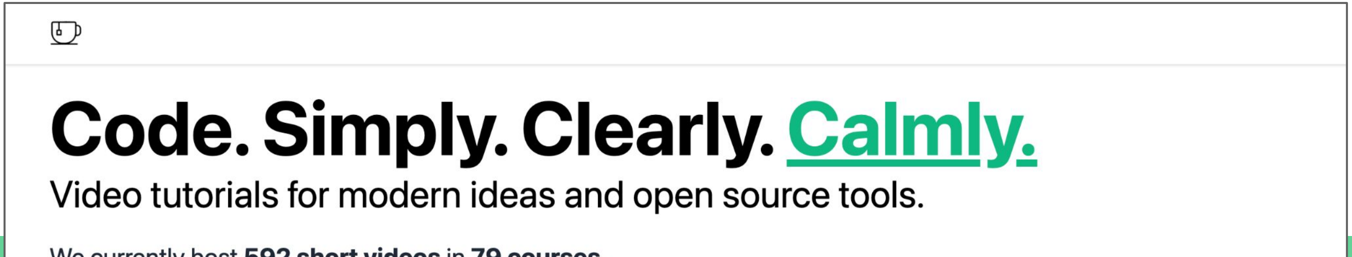
School vs Real World

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

```
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

```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
    )

    @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

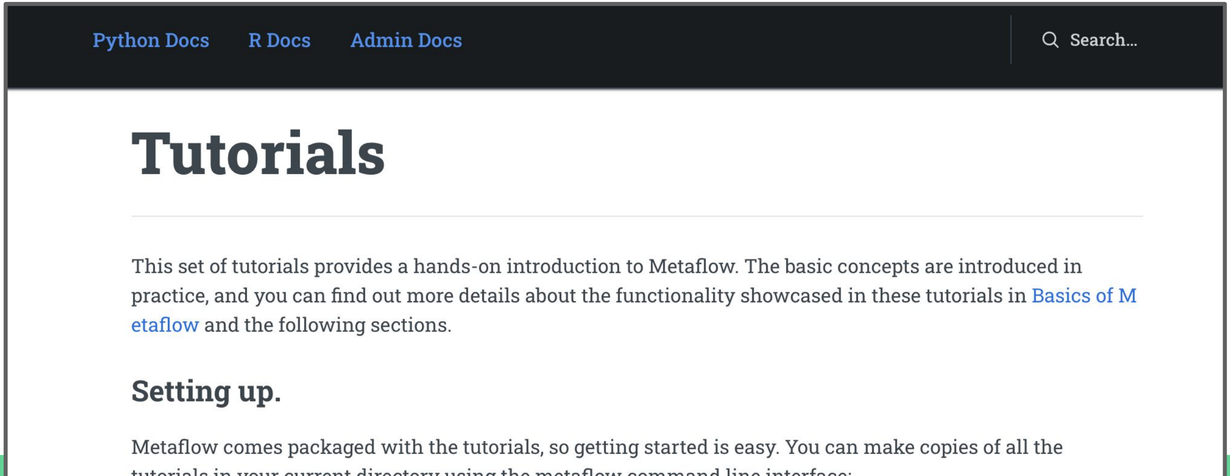
```python
    )

    @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

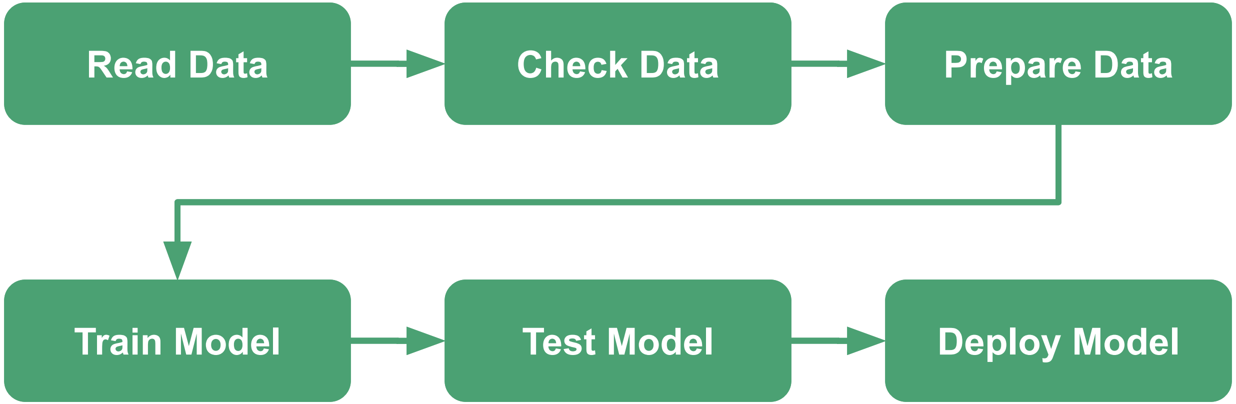
1. **Dag definition:** what are we doing? Steps, dependencies, parallelization etc.
2. **Metastore:** where do we store stuff? Variables, states, meta-data etc.
3. **Computational layer:** what is executing the computation? Resources, cloud tools etc.



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)
```

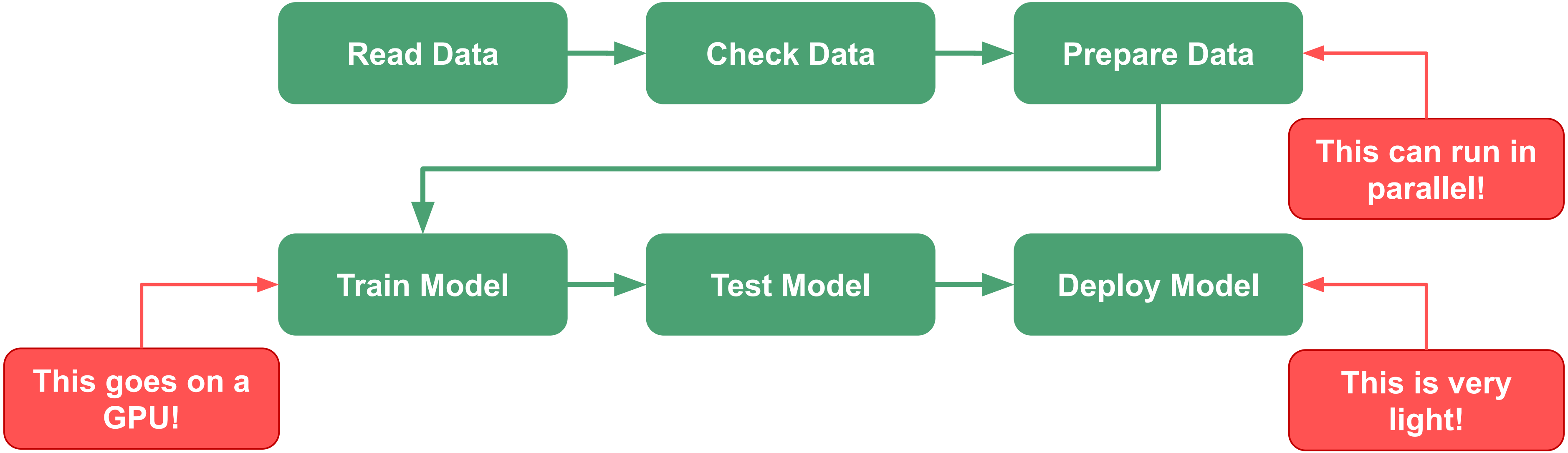
The raw dataset is saved!

The X,Y dataset is saved!

Metaflow in 4 principles

#3: One computing size does not fit all

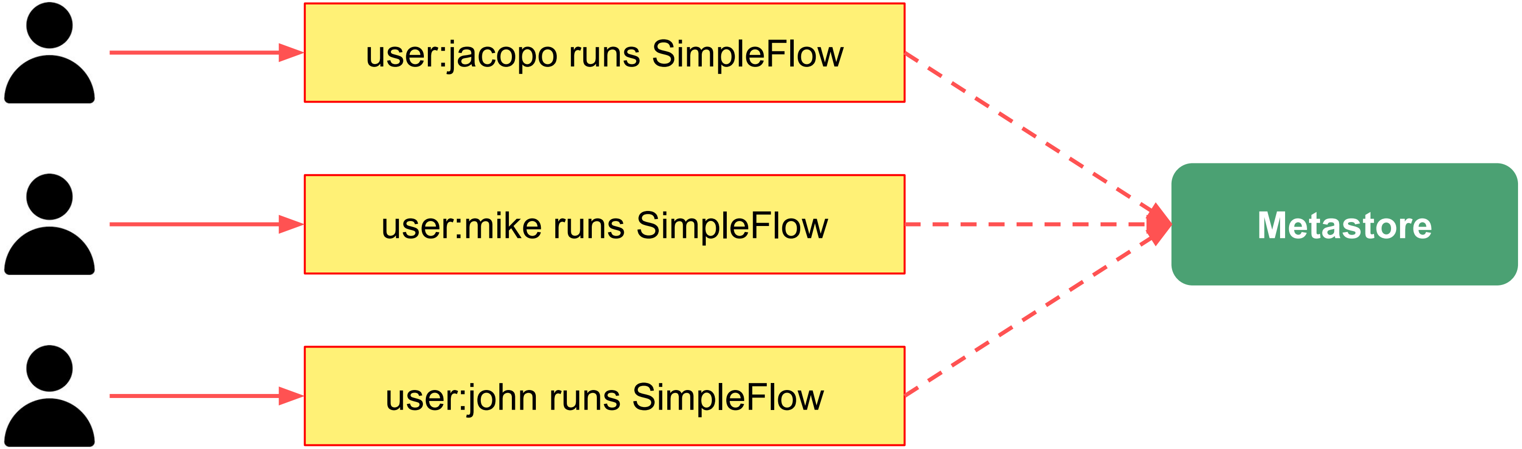
You can define computing resources (and packages) *per task*, switching between local and cloud computing only when necessary.



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:

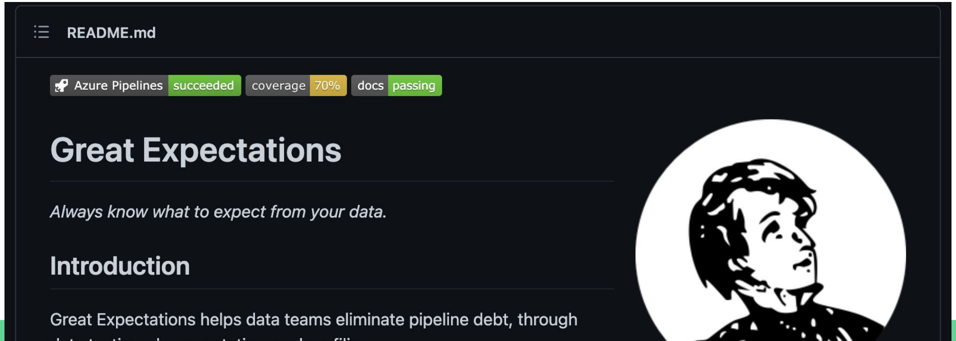
1. To trust your model you need to trust your data -> data checks.
2. To trust your model you need to trust your training routine -> hyper tuning, experiment tracking, understood quantitative objective.
3. To trust your model you need to trust it in edge cases (or cases that are particularly interesting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

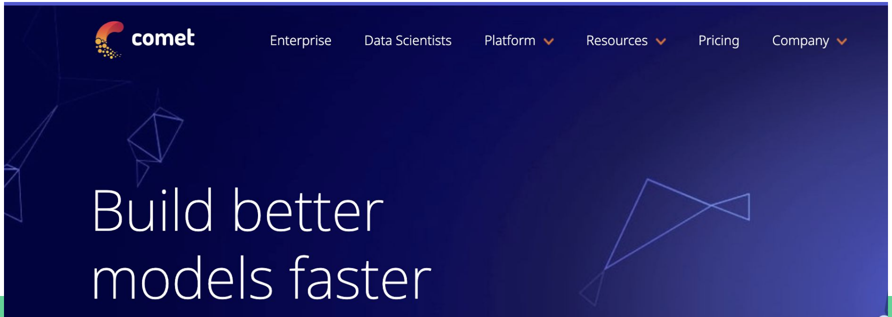
Some questions we may want to ask our data:

- 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

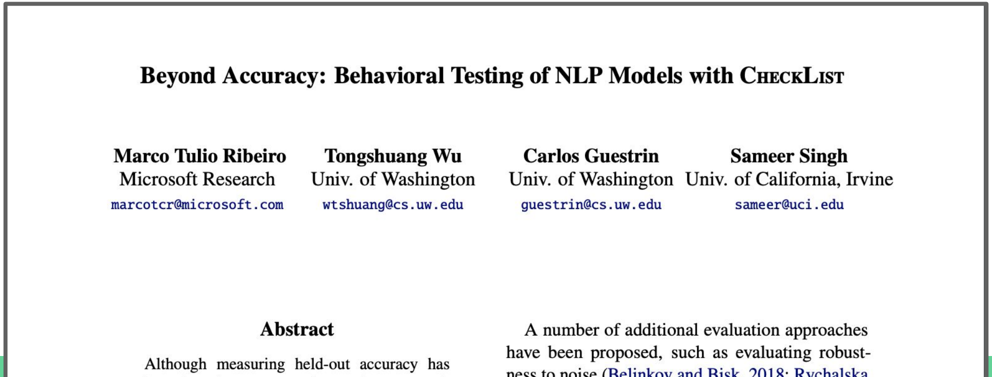
Although measuring held-out accuracy has

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

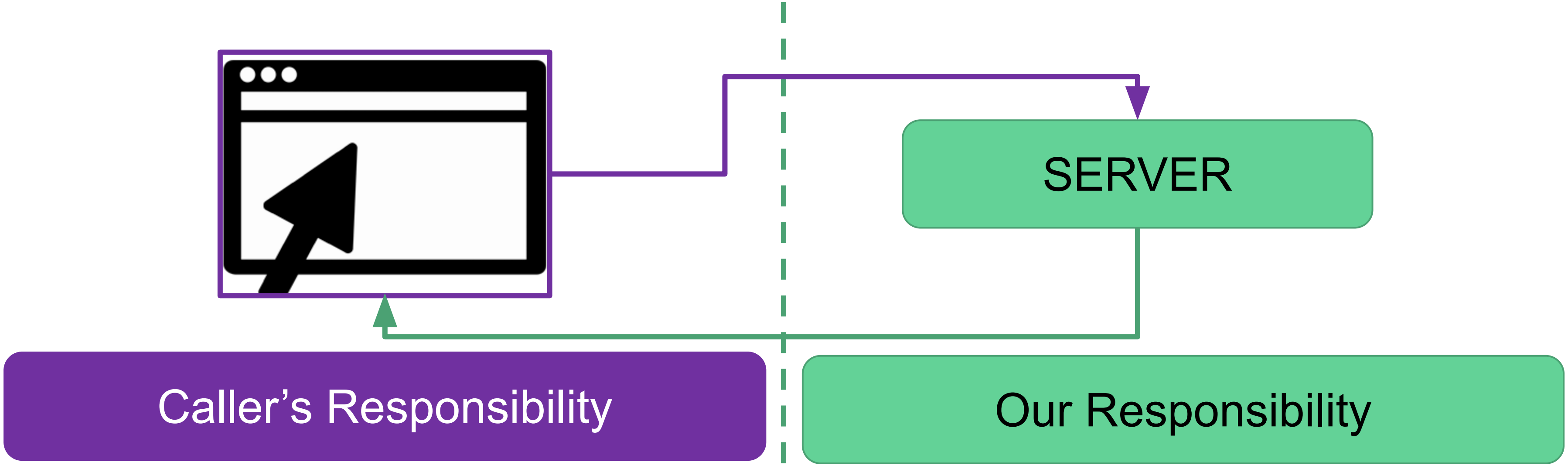
Abstract

Although measuring held-out accuracy has

A number of additional evaluation approaches have been proposed, such as evaluating robustness 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:

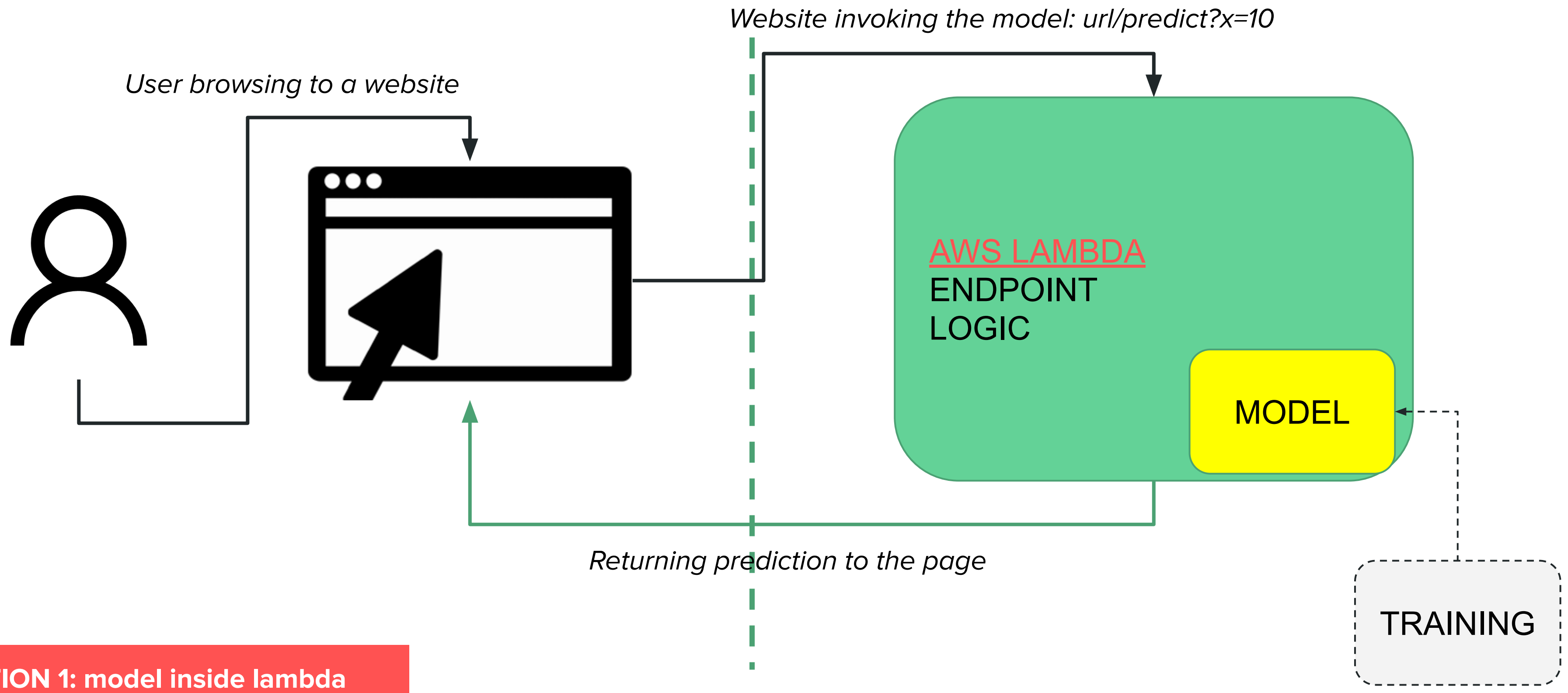
- IaaS: Infrastructure as a Service
- PaaS: Platform as Service
- FaaS: Function as a Service

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?)

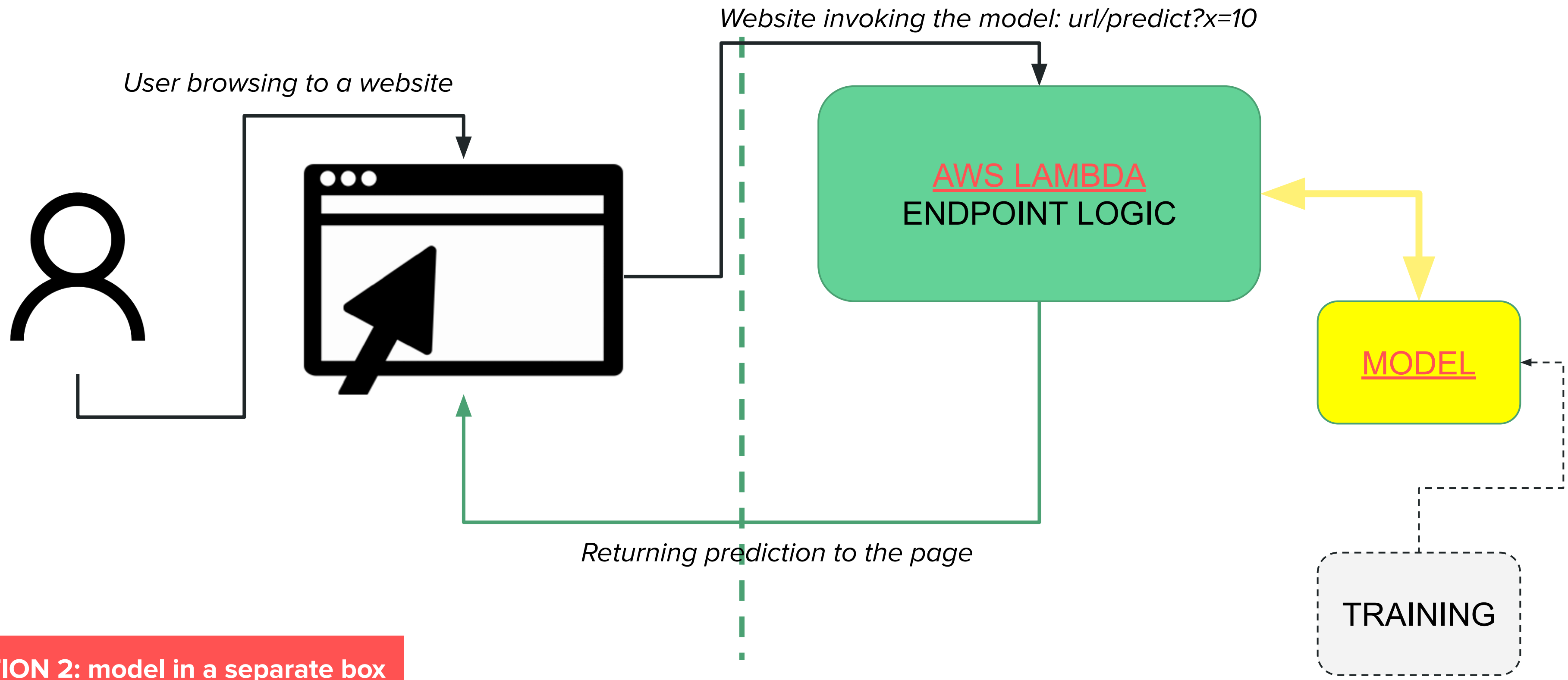
[While not necessary, it is good practice to handle Infrastructure as Code, for example with Serverless.]

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!

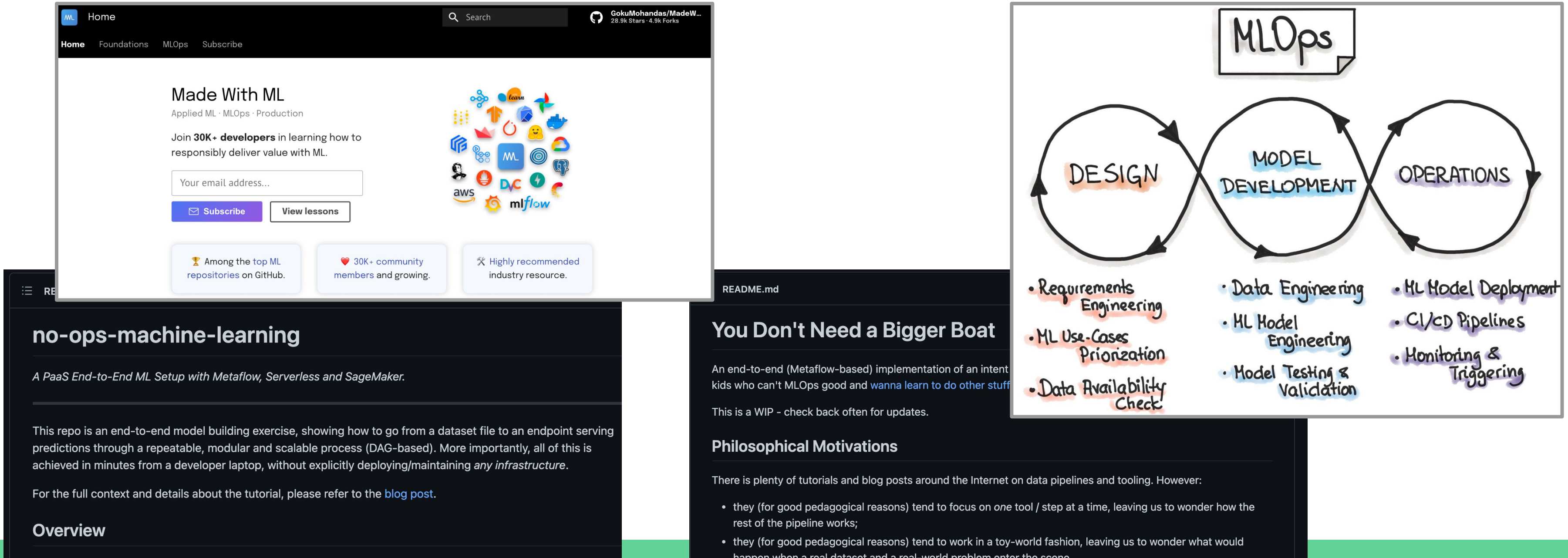
After deployment: monitoring

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!



Good ol' NLP

What is language?

- Language is an incredibly complex object, and a quintessential human prerogative (pending some birds).

1.2.1 Questions that linguistics should answer

What questions does the study of language concern itself with? As a start we would like to answer two basic questions:

- What kinds of things do people say?
- What do these things say/ask/request about the world?

Morphology, syntax etc.

What is language?

- Language is an incredibly complex object, and a quintessential human prerogative (pending some birds).

1.2.1 Questions that linguistics should answer

What questions does the study of language concern itself with? As a start we would like to answer two basic questions:

- What kinds of things do people say?
- What do these things say/ask/request about the world?

Semantics, pragmatics, discourse

What is language?

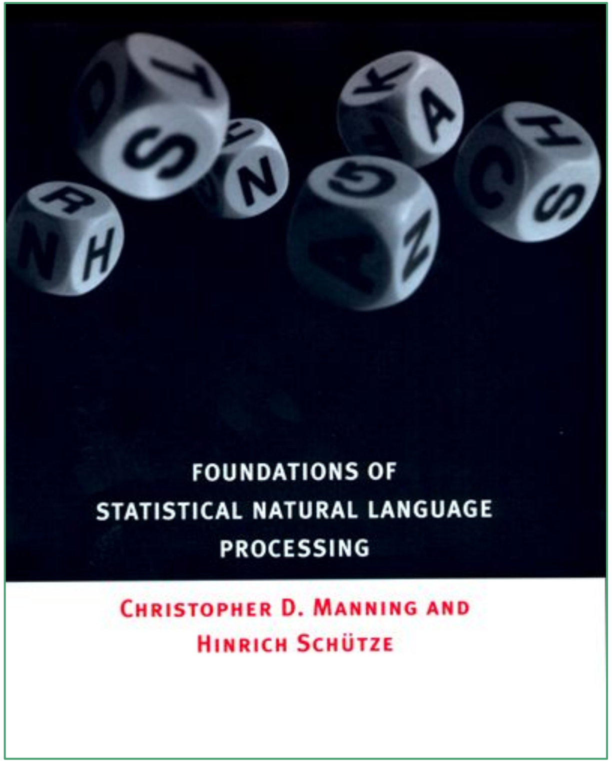
- Sound-stuff: phonetics and so on
  - Note: speech-to-text is pretty advanced, and we mostly deal with written language
- Morphology: word structure
  - *bellissimo* = “bell” (root) + “issim” (superlative) + “o” (male, singular)
- Syntax: how words are combined together
  - Colorless green ideas sleep furiously (and Broca’s area!)
  - Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo
- Lexical semantics: the meaning of words
  - Man : King = Woman : Queen
- Compositional Semantics: meaning of sentences (truth / entailment)
  - Every man is mortal, Socrates is a man, Socrates is ….
  - The meaning of a sentence is how the world would look like, if the sentence was true
- Pragmatics: language in context
  - “Can you speak English?” vs “Can you pass me the salt?”
  - - How is your new CS Ph.D.? - He is always on time for meetings and has a very pleasant voice.
- Discourse: language in turns
  - - I can’t meet you today. - What if we do in two days?

What is language at FRE 7773?

- Sound-stuff: phonetics and so on
    - Note: speech-to-text is pretty advanced, and we mostly deal with written language
- Morphology: word structure
    - *bellissimo* = “bell” (root) + “issim” (superlative) + “o” (male, singular)
- Syntax: how words are combined together
    - Colorless green ideas sleep furiously (and Broca’s area!)
    - Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo
- Lexical semantics: the meaning of words
    - Man : King = Woman : Queen
- Compositional Semantics: meaning of sentences (truth / entailment)
    - Every man is mortal, Socrates is a man, Socrates is ….
    - The meaning of a sentence is how the world would look like, if the sentence was true
- Pragmatic: language in context
    - “Can you speak English?” vs “Can you pass me the salt?”
    - - How is your new CS Ph.D.? - He is always on time for meetings and has a very pleasant voice.
- Discourse: language in turns
    - - I can’t meet you today. - What if we do in two days?

Language as a statistical phenomenon

- To answer “What kind of things people say?” we take a *statistical approach*, that is, we try and identify *common patterns* that occur in language use (i.e. we’ll do a lot of counting and probabilities).
    - On Chomsky and the Two Cultures of Statistical Learning
- “While practical utility is something different from the validity of a theory, the usefulness of statistical models of language tends to confirm that there is something right about the basic approach.”



Language modelling

Given a language L with terms t_1, t_2, ... t_n, and a set of sentences S from t_1, t_2, ... t_n, a language model (**LM**) is a function *f* assigning a probability to each sentence in S. We require *f* to be a probability distribution, i.e.:

1. The sum of all probabilities for all sentences sums up to 1;
2. Each sentence gets assigned a probability, that is for each sentence *s*, f(*s*) >= 0.

Given a LM, we can:

- given a sentence, calculate its probability under the LM: P(“I ate an apple”) > P(“I ate an embassy”)
- given *n* tokens starting a sentence, predict what is likely to come next: P(“apple” | “I ate an”) > P(“embassy” | “I ate an”) -> **if I concatenate prediction after prediction, what do I get?**

Language modelling

- **Terms**: { Jacopo, NYU, NLP, NYC, lives, teaches, is, Italy, and ... }
- **Sample sentences S**: { Jacopo teaches at NYU; Jacopo is from Italy; Jacopo is from NYC; Jacopo teaches NLP in NYC; Jacopo teaches NLP in Italy }
- **Task**: *learn* a LM for S.
  - "Learn" implies that our LM should reflect the statistical patterns of our sample, instead of, for example, simply assigning arbitrary probabilities to sentences;
  - the fact that a sentence is true or false for humans is irrelevant (i.e. I'm not from NYC): LMs capture statistical patterns of "plausibility", not truthfulness.

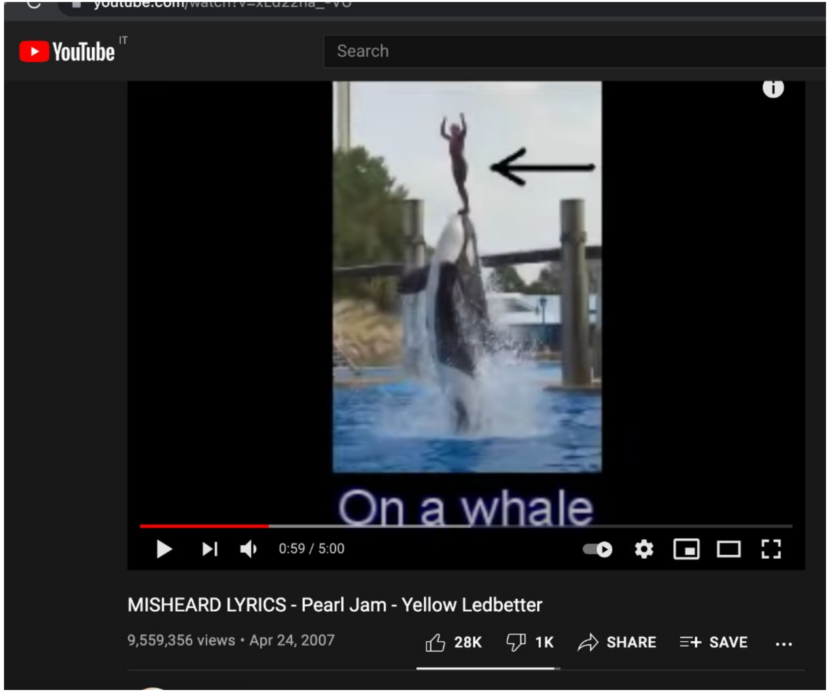
Language modelling

- **Terms**: { Jacopo, NYU, NLP, NYC, lives, teaches, is, Italy, and ... }
- **Sample sentences S**: { Jacopo teaches at NYU; Jacopo is from Italy; Jacopo is from NYC; Jacopo teaches NLP in NYC; Jacopo teaches NLP in Italy }
- **Task**: *learn* a LM for S.
  - "Learn" implies that our LM should reflect the statistical patterns of our sample, instead of, for example, simply assigning arbitrary probabilities to sentences;
  - the fact that a sentence is true or false for humans is irrelevant (i.e. I'm not from NYC): LMs capture statistical patterns of "plausibility", not truthfulness.

A trivial LM: *f(s)* = empirical frequency of s!

Why should *you* care?

- Speech recognition: what you hear is a "sound" + your linguistic expectations (check this on misheard lyrics!)
- LMs are in your everyday life. e.g. Smart Compose for Gmail!
- LMs are a (plausible) way to treat language as a statistical phenomenon: their shortcomings are interesting.
- LMs are **very central** to contemporary NLP.



Markovian language models

- **Goal**: estimate the probability of a sequence of terms, taken from our vocabulary, that is $P(t_1, t_2, \dots t_n)$
  - Since for *n* there are $|\text{Vocabulary}|^n$ sequences, we want a compact model!
- **First step**: re-write the joint distribution with the chain rule:

$$P(t_1, t_2, \dots t_n) = P(t_1) \prod_{i=2}^{n} P(t_i \mid t_1, \dots t_{i-1})$$

- **Second step**: Markov assumption, the probability of a term *only depends on the previous term.*

$$P(t_1, t_2, \dots t_n) = P(t_1) \prod_{i=2}^{n} P(t_i \mid t_{i-1})$$

- **Bonus step**: second degree Markov, third, etc.

A bigram language model

- Let's augment our vocabulary t_1, t_2, … t_n with two special tokens, * and |.
  - * is to be used as a special "start sentence" sign
  - | is to be used as a special "stop sentence" sign
- **Goal #1**: since the probability of any sentence for our LM is the product of the probabilities of each bigram…
- **Goal #2**: estimate the probability of each bigram, that is, each pair of terms.
  - P("Jacopo teaches NLP") = P("* Jacopo") x P("Jacopo teaches") x P("teaches NLP") x P("NLP |")
  - We would like to estimate then P(Jacopo | *), P(teaches | Jacopo), etc.
- **Estimation**: "maximum likelihood estimation"
  - Given a bigram <u, w>, P(w | u) = Count(u, w) / Count(u)
  - Example: P(teaches | Jacopo) = Count(Jacopo teaches) / Count(Jacopo)
- **Let's check the notebook now to see how that looks in practice**.

A trigram language model

- One more time, with feelings!
- Trigram LMs are exactly the same as bigram LMs, but now we employ a second-order Markov condition
    - i.e. the probability of "States", in "Biden is the President of the United States", *depends only on "United" and "the".*
- **Estimation**: "maximum likelihood estimation"
    - Given a trigram <u, y, w>, P(w | u, y) = Count(u, y, w) / Count(u, y)
    - Example: P(in | Jacopo teaches) = Count(Jacopo teaches in) / Count(Jacopo teaches)
- **Let's check the notebook now to see how that looks in practice**.

How good is a LM?

- Qualitative evaluation:
  - If we are fluent in the underlying language/vocabulary, we can “unit test” our LM:
    - quality checks depends heavily on the tester knowledge / assumptions.
    - It doesn’t scale, BUT qualitative tests are very useful in practice (e.g. corner cases, biases).

How good is a LM?

- Qualitative evaluation:
  - If we are fluent in the underlying language/vocabulary, we can “unit test” our LM:
    - quality checks depends heavily on the tester knowledge / assumptions.
    - It doesn’t scale, BUT qualitative tests are very useful in practice (e.g. corner cases, biases).
- Quantitative evaluation:
  - Intrinsic, with **perplexity**:
    - Intuition: given a standard train/test split, a good LM would evaluate as highly probable the unseen sentences in the test set.
    - You first compute the log probability of test under LM, and normalize by the number of words: call it LP; then perplexity = 2^{-LP}; i.e. the *smaller perplexity is*, the better the LM.
    - **Q**: **what happens if *any* of the sentence in the test set gets P=0 under the LM**?
  -

A Bit of Progress in Language Modeling
Extended Version
Joshua T. Goodman
Machine Learning and Applied Statistics Group
Microsoft Research

How good is a LM?

- Qualitative evaluation:
  - If we are fluent in the underlying language/vocabulary, we can “unit test” our LM:
    - quality checks depends heavily on the tester knowledge / assumptions.
    - It doesn’t scale, BUT qualitative tests are very useful in practice (e.g. corner cases, biases).
- Quantitative evaluation:
  - Intrinsic, with **perplexity**:
    - Intuition: given a standard train/test split, a good LM would evaluate as highly probable the unseen sentences in the test set.
    - You first compute the log probability of test under LM, and normalize by the number of words: call it LP; then perplexity = 2^{-LP}; i.e. the *smaller perplexity is*, the better the LM.
    - **Q**: **what happens if *any* of the sentence in the test set gets P=0 under the LM**?
  - Downstream tasks: we will discuss LMs as building blocks for other tasks in future lecture.

Dealing with long range dependencies

- The Markov assumption seems very plausible for certain contexts:

  In “Biden is the President of the United States”, the high probability of P(States | the, United) does a good job in narrowing down candidates.

- …but certainly not in others:

  “I like the book …”: “I am reading now”, “sitting on my desk”, “that I bought yesterday”… completion are much more open ended.

We can build higher-order LMs (four-gram models etc.), but data sparsity would make the models marginally better after a while. Truth is, we will revisit this with neural networks.

Dealing with rare events

- Remember *perplexity* goes to infinity when any P(sentence) is 0, which happens anytime the test set has unseen n-grams.
- The solution is called "*smoothing*", and involves providing estimates for unseen n-grams.

METHOD #1: ADD ONE (LAPLACE LAW)

- Given <u, w>, P(w | u) = Count(u, w) **+ 1** / Count(u) **+ |V|** (where |V| is the vocabulary size)
- When Count(u, w) = 0, the LM will still assign to it a positive probability

Dealing with rare events

METHOD #2: LINEAR INTERPOLATION

- Given <u, y, w>, we smooth **P(w | u, y)** by: λ_1 Trigram + λ_2 Bigram + λ_3 Unigram, where $\lambda_1 + \lambda_2 + \lambda_3 = 1$, where:
    - Trigram = Count(u, y, w) / Count(u, y)
    - Bigram = Count(y, w) / Count(y)
    - Unigram = Count(w) / |V|
- The intuition is that we use lower-level probabilities (as data is sparser in higher order) to compensate for our estimates when data is missing.
- **How do we pick the lambdas?** We use a *validation set* and pick the lambdas that maximizes the log probability over the dataset.

Application: typo-correction

- How to Write a Spelling Corrector (the "old" way)
- We model spell checking as a noisy channel:
    - Imagine a sender S sending a message M to a receiver R, *but M may be corrupted in the process.*
    - Example: S sends "apple" to R, but R receives "apkle" - R needs to be able to reliably recover "apple"
    - **Goal for R**: rank possible messages from S according to P(message | text I received)
    - Example: P(apple | apkle) > P(car | apkle) (**why?**)

How to Write a Spelling Corrector

One week in 2007, two friends (Dean and Bill) independently told me they were amazed at Google's spelling correction. Type in a search like [speling] and Google instantly comes back with **Showing results for:** ***spelling***. I thought Dean and Bill, being highly accomplished engineers and mathematicians, would have good intuitions about how this process works. But they didn't, and come to think of it, why should they know about something so far outisde their specialty?

I figured they, and others, could benefit from an explanation. The full details of an industrial-strength spell corrector are quite complex (you can read a little about it here or here). But I figured that in the course of a transcontinental plane ride I could write and explain a toy spelling corrector that achieves 80 or 90% accuracy at a processing speed of at least 10 words per second in

A noisy channel model

- Consider a vocabulary of t_1, t_2, ... t_n terms. *S* picks one term to send (the message, M = some *t*), R tries to decode what she receives (the actual string A).
- For all *t*, R needs to compute **P(*t* | A)**, that is, through Bayes: **P(t) X P(A | t),** and then pick the term with the highest probability.

- We need to estimate two terms then:
    - P(t): a language model (unigram, in fact), that is, the prior probability of “apple” vs “car” in the general language;
    - P(A | t): an error model, that is, the probability that, given that S really wanted to say “apple”, “apkle” resulted instead.

i.e. the “right” correction is a trade-off between popularity and possible mistakes

A noisy channel model

S

cae

What was M?

A noisy channel model

S

cae

What was M?

cat

If e->t, and we are on Chewy.com

A noisy channel model

S

cae

What was M?

cat

car

If e->r, and we are talking about L. Hamilton

A noisy channel model

S

cae

What was M?

cat

car

che

If a->h, and we are reading a book on Cuba

A noisy channel model

- Estimate the LM: *unigram* LM, that is, for each term *t* calculate P(t) as **Count(t) / # of tokens**
- Estimate the error model: “error model that says all known words of edit distance 1 are infinitely more probable than known words of edit distance 2, and infinitely less probable than a known word of edit distance 0”
  - Example: P(apkle | apple) = P(spple | apple) = P(appl | apple)
  - Example: P(apkle | apple) > P(apkles | apple)
- **Let’s check the notebook now to see how that looks in practice**.

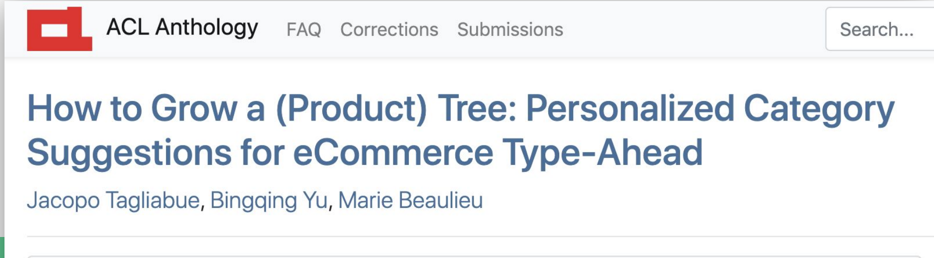
A noisy channel model - Homework hints

How can we go and improve upon Norvig's model?

- We can improve the language model: "cae" is more likely to be "car" than "cat" in the sentence "I drive a fast **cae**". What happens if we consider the context?
- We can improve the error model: given the QWERTY layout, some errors are more likely than others - can we incorporate this intuition in the spelling corrector?

Bonus: **advanced** noisy channel model

- Can we explicitly condition the language model depending on context?
- Type-ahead is a good example: for all completions Cs and query Q, we compute **P(c | Q)**, i.e. **argmax P(c) X P(Q | c).**
- **P(c)** becomes **P(c | context)**, so that the probability of a completion change based on the history of the user, her geolocation etc.



From words to vectors

- **Q**: We are used to feed “scikit models” with numbers for, say, regression, but how do we feed them *words*?
- **A**: We feed words by converting them to numbers!

Option #1: count vectorizer

Option #2: TF IDF vectorizer

One-hot encoding for words

Test corpus:

- "I live in NYC"
- "I teach in NYC"
- "I leave and teach in NYC"

Total of 6 words: [I, live, in, NYC, teach, and]

One-hot encoding for "I":

1	0	0	0	0	0

One-hot encoding for "NYC":

0	0	1	0	0	0

One-hot encoding for words

Test corpus:

- “I live in NYC”
- “I teach in NYC”
- “I leave and teach in NYC”

Total of 6 words: [I, live, in, NYC, teach, and]

One-hot encoding for “I”:

1	0	0	0	0	0

One-hot encoding for “NYC”:

0	0	1	0	0	0

What happens with a bigger corpus?

Count vectorizer

Test corpus:

- “I live in NYC”
- “I teach in NYC”
- “I leave and teach in NYC”

Total of 6 words: [I, live, in, NYC, teach, and]

Vector for “I teach in NYC”:

1	**0**	**1**	**1**	**1**	**0**

Q: what is the vector for “in NYC live I”?

Count vectorizer

- Visually, documents map to a point in the vocabulary space: in this example, when V=2, we can draw four Shakespeare plays and see that similar plays point to the same region of the space.
- How do we quantify "similar" then?

	As You Like It	Twelfth Night
battle	1	0
good	114	80
fool	36	58
wit	20	15

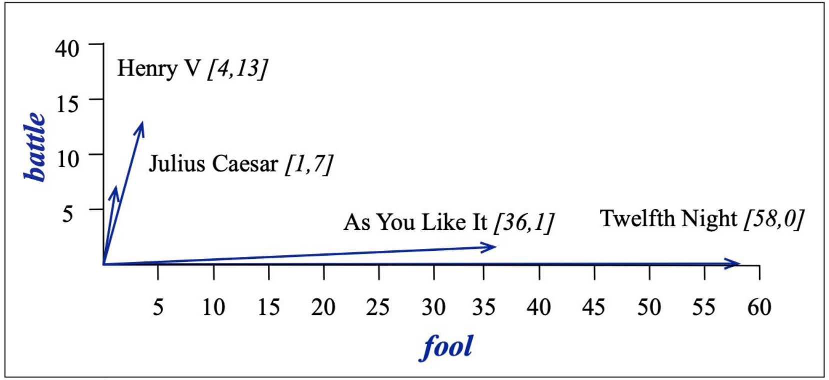


Figure 6.4 A spatial visualization of the document vectors for the four Shakespeare play documents, showing just two of the dimensions, corresponding to the words *battle* and *fool*. The comedies have high values for the *fool* dimension and low values for the *battle* dimension.

Count vectorizer

- Visually, documents map to a point in the vocabulary space: in this example, when V=2, we can draw four Shakespeare plays and see that similar plays point to the same region of the space.
- How do we quantify "similar" then? -> **COSINE SIMILARITY!**

$$\text{cosine}(\mathbf{v},\mathbf{w}) = \frac{\mathbf{v}\cdot\mathbf{w}}{|\mathbf{v}||\mathbf{w}|} = \frac{\boxed{\sum_{i=1}^{N} v_i w_i}}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$

Dot Product

From counts to weights

- In the Count vectorizer, all words get the same importance
- Intuitively, some words however are more important than others: the presence of the word “growth” or “liability” in a financial article is more salient than generic words like “and” or “ company”.
- TF-IDF (“term frequency–inverse document frequency”) is an effective weighting scheme used in IR, text classification etc.

  tf-idf (term, document, corpus) = frequency (term, document) * idf (term, corpus)

  [typically idf = log (# documents / # documents with term)]

Q: how do we get a high tf-idf (term, document, corpus)?

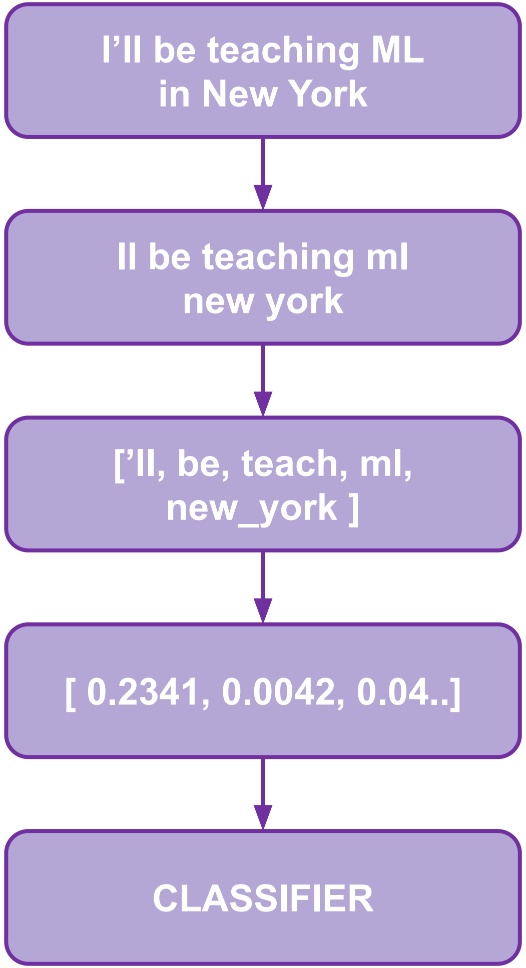
See the notebook on text classification for hand-on experience and tips on vectorization!

Application: text classification

- Text classification is one of the oldest tasks in NLP, and very relevant to Finance: for example, we may want to classify information based on a topic ("is this article about politics?"), or we may want to classify the general sentiment of a financial announcement ("is this tweet by Bank of America positive or negative?")
- The general flow is similar to the "scikit patterns" you have seen earlier in the course, but with some caveats, required by the peculiar nature of text data.
- Make sure to check the notebook on text classification for some hand-on experience!

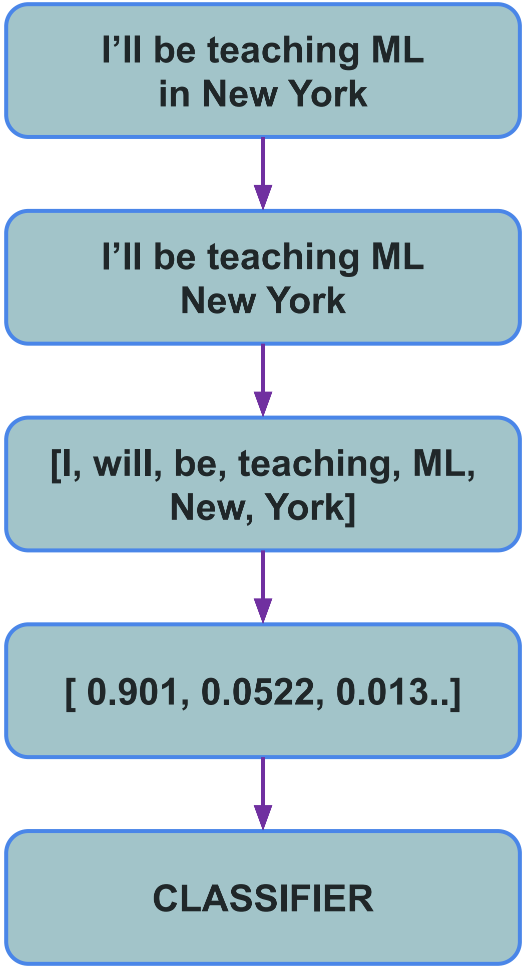
Text classification flow

- Pre-processing: casing (?), punctuation (?), stop words (?).
- Tokenization: split text in tokens { stemming (?), lemmatization (?), phrases (?) }.
- Vectorization: apply a vectorization technique (count or tf-idf) [Vocabulary size (?)]
- Modelling (training and testing): train a classifier model over text vectors - this is equivalent to your previous experience with scikit-like APIs.



Text classification flow

- Pre-processing: casing (?), punctuation (?), stop words (?).
- Tokenization: split text in tokens { stemming (?), lemmatization (?), phrases (?) }.
- Vectorization: apply a vectorization technique (count or tf-idf) [Vocabulary size (?)]
- Modelling (training and testing): train a classifier model over text vectors - this is equivalent to your previous experience with scikit-like APIs.



Additional materials

THEORY

- Aside from the classic book from Manning & Schutze, I love Michael Collins old notes on “foundational” NLP; the classic Russell & Norvig also contains some NLP stuff, and Bender’s book is a survey of linguistics concepts for NLP.
- For a more recent treatment, check out the excellent Jurafsky & Martin.

PRACTICE

- Good NLP libraries in Python: NLTK is where everybody starts; check out Spacy for a modern (and more opinionated) approach.