Objectives of today's lecture

- review the core components of the ML lifecycle
- review key concepts used in subsequent lectures

Section 1. The Machine Learning Lifecycle

There are **three high-level stages** in a typical machine learning lifecycle (see Figure 1):

- Stage 1 - model development
- Stage 2 - model training
- Stage 3 - model inference

Each stage has distinct systems' challenges which we discuss next.

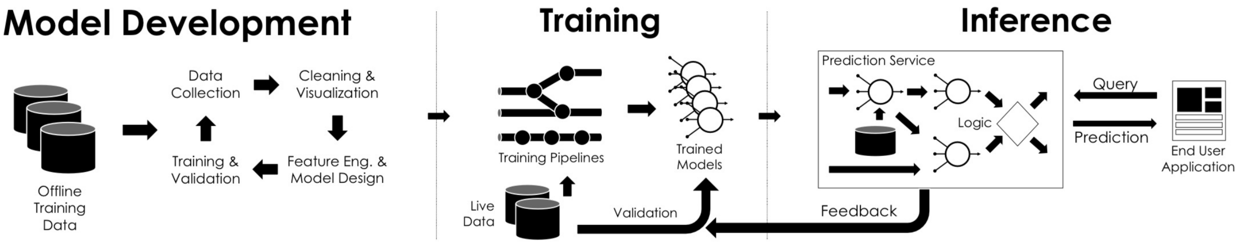


Figure 1. A simplified depiction of the key stages of the machine learning lifecycle
[Source: Gonzalez, IEEE Data Engineering Bulletin 2018]

Section 2. Model Development

The stage of model development entails a series of tasks that can be grouped into two categories:

- **Data management tasks:**
  - **Identify** useful data sources,
  - **Join** data from multiple sources to form informative feature vectors,
  - Address **missing values** and **outliers** either via sanitization (i.e., removal of erroneous tuples) or data repairs (i.e., value imputation or value correction),
  - Profile the data to identify potential anomalies; mostly done via plotting and understanding simple statistical trends in the data.

- **Model design tasks:**
  - Identify and design informative **feature representations.** We may need to manually design feature extraction functions and transformations.
  - Design the model architecture (e.g., shall we use Naive Bayes or Logistic Regression)
  - Tune the **hyperparameters** of the model
  - **Validate** the prediction accuracy of the model

Some points on the above tasks

What is an outlier? A data point that differs significantly from other observations. The concept of outliers starts from the issue of building a model that makes assumptions about the data. Outliers might be legit data points that do not adhere to our modeling assumptions.

What is an anomaly in a dataset? The term anomaly can be used synonymously with the term outlier. There are cases however where the term anomaly might indicate a single data point (or a group of data points) that are erroneous (due to data collection errors or faulty measurement mechanisms etc).

Features and feature engineering

Features are properties or characteristics of the input data points.

A model with parameters θ is a function $f_\theta(x) \rightarrow y$

x: feature vector (can contain real values, binary values, categorical values etc)
y: target prediction (class, real-valued measurement etc, probability

Example model (simple logistic model)

$$f_\theta(x) = \sigma\left(\sum_{k=1}^{d} \theta_k \phi_k(x)\right), \quad \sigma(t) = \frac{1}{1+e^{-t}}$$

The "final" features used in the model correspond to the outputs of functions $\phi_k(\cdot)$. These functions can range from simple function (identity or thresholds) to complex functions (neural networks).

Additional notes on feature engineering

- To generate the feature vector x we might have to **join** multiple data sources (e.g., tables in a database).
- There are **features that might curry signal (be useful) for different prediction tasks** (e.g., the zipcode of a property can be used to estimate the value of a house but can also be used to recommend the appropriate insurance policy for the owners). This phenomenon introduces the problem of **feature reuse**.
- **The prediction of one model can be used as a feature in another model** (for a different task). Think of advertisement: we first predict which products appear in an user image (in social media) and then use these predictions to suggest ads.
- **Feature caching** might be important (pre-compute features). Connections to materialized view maintenance (how do we update the feature values as new data becomes available or as data is updated/deleted?). Connections to data versioning.
- **Dynamic features:** features can change over time. Consider the case of user clickstreams. There are both **systems' challenges** in (e.g., we need to consider the latency of computing the features used by a downstream model) and **modeling challenges** (e.g., covariate shift) in streaming setups.

Hyperparameters

The term hyperparameters refers to the parameters and other configuration details of a machine learning model that are not directly determined through training. Hyperparameters are typically set by hand or tuned using cross validation.

Some example hyperparameters: 1) the architecture of a model (e.g., how many layers, the dimensionality of different layers), 2) the learning rate during gradient descent, 3) the batch size used during training. We will see how one can learn these parameters in the AutoML lecture.

$$w^{(t)} \leftarrow w^{(t-1)} - \eta \cdot \sum_{i \in B} \nabla_\theta \left(L_\alpha(f_\theta(x_i), y_i) \right) \Big|_{\theta^{(t)}}$$

🟥: denotes the hyperparameters during training (e.g., the learning rate η).

What should the output of model development?

The "naive" answer: the output of model development is a trained, deployable model.

Why naive? A trained model is a monolithic artifact that is not easy to:
- Maintain and update when shared out of context, i.e., retrain the model
- Track data and code for debugging
- Analyze and capture dependencies for deployment
- Audit training for compliance (think of fairness, privacy violations etc).

A better alternative is to create a structure that maintains all above information. We want to maintain a **training pipeline.** Think of software engineering practices: the training pipeline is equivalent to the code and the trained model is equivalent to the binaries.

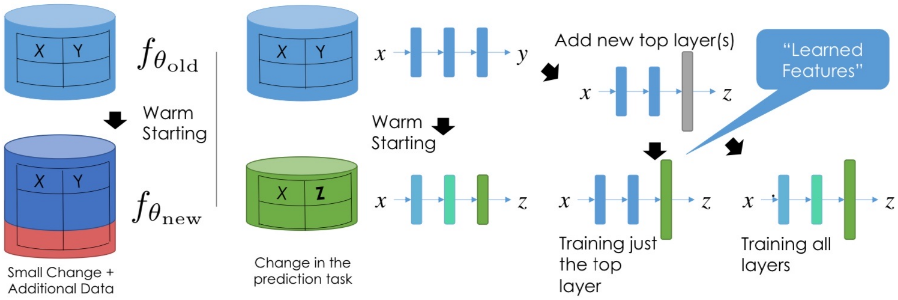
Section 3. Training

The training stage contains several tasks:
- Train developed models **at scale** on production (live) data. During development one typically uses a representative sample.
- **Re-train** a model when new data is available.
- Validate the prediction accuracy of the model **automatically**.
- Manage different **versions** of the model. Continuous integration for new models.

Overall this stage does not require ML expertise but entails several software engineering challenges.

Some points on Training tasks

How can we re-train a model given new data? Instead of training a model with random initial parameters we can "warm-start" training at the previous solution. This approach works well when the data is changing slowly and we basically start somewhere close to the new optimal solution. It is more challenging when the data distribution is changing rapidly. There are additional technical concerns:
  - How do we set the learning rate? We should be making smaller steps.
  - We can get stuck in suboptimal solutions.
  - What about the "old" characteristics of the data distribution? We need to control **"forgetting"**

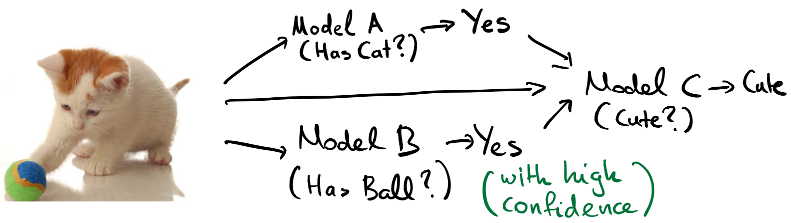

X Y
$f_{\theta_{\text{old}}}$
Warm Starting
X Y
$f_{\theta_{\text{new}}}$
Small Change + Additional Data
X Y
x y
Warm Starting
X Z
x z
Change in the prediction task
Add new top layer(s)
x z
"Learned Features"
x z
Training just the top layer
x z
Training all layers

The idea of warm starting can be used to fine-tune an old model to new data.
[source: Lecture from AI Systems class at Berkeley]

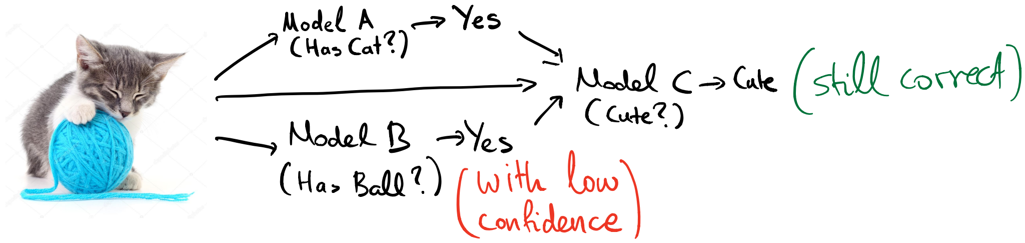
Before we talked about the output of models being used as features for downstream models? What is a core challenge related to this and continuous integration?

Consider the composition of three models for predicting cute animal pictures:

- Model A: Takes as input an image and detects if the image contains a cat. It returns Yes or No.
- Model B: Takes as input the same image and detects if there is a ball. It returns Yes or No.
- Model C: Takes as input the image as well as the output of models A and B and has learned that if both outputs from A and B are Yes the image is "Cute" otherwise it is "Not Cute".



What is Model B cannot distinguish between a actual ball and a ball of yarn?



What will happen if we replace Model B with a better model that can distinguish between a ball and a ball of yarn? Model C will start returning "Not Cute" which might not be the intended behavior.

We need to track composition and validate **end-to-end accuracy.** We will see multi-task learning in the class.

Section 4. Inference

The typical task of inference is to apply the prediction logic of a model pipeline to input user queries. The predictions are used to serve decision making applications. The prediction pipeline may be agnostic of the downstream application.

One of the immediate goals of inference is to make predictions with low latency (e.g., ~10ms under **bursty** load). For simple models predictions may correspond to simple decision paths (single decision tree) or a simple matrix vector multiplication. Things are much more complicated in the case of deep neural networks or random forests with a large number of decision trees. We will examine technologies and systems for speeding inference for such complex models.

Other considerations during inference: How do we collect feedback from users (direct or indirect based on the downstream application)?

- We can leverage logging mechanisms and either retrain the model **periodically** or in a continuous manner (**online learning**). Periodic retraining is preferred in practice.
- How can we integrate new features in an existing ML model? How do we ensure robustness of an already developed model?