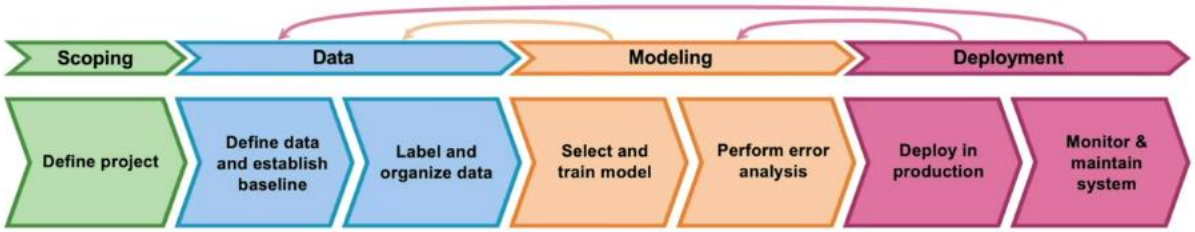
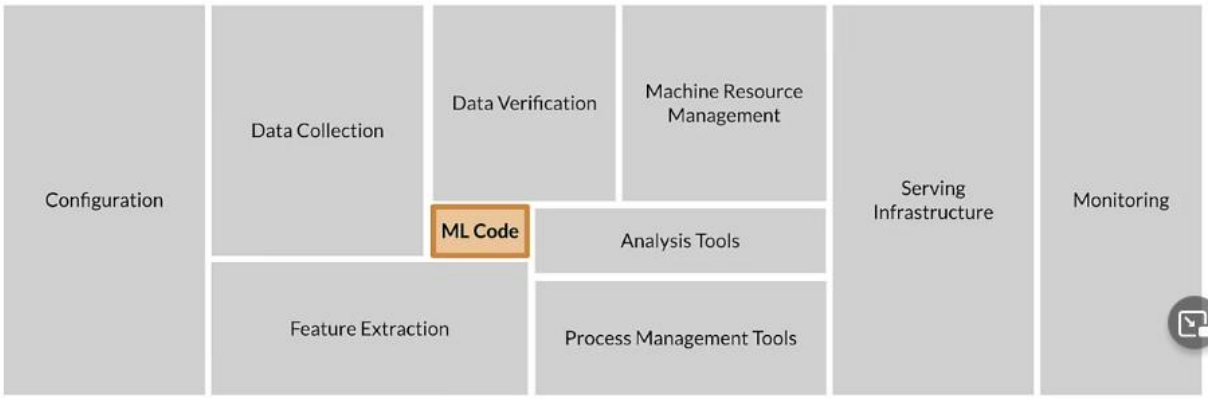
Coursera MLOps Specialization - Notes

Table des matières

ML project lifecycle





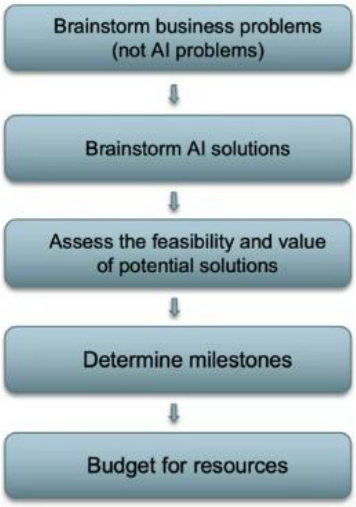
Scoping

Goal: Define the needs and the objectives of the project to know if it is worth putting efforts in it.

Questions to ask:

- What project should we work on?
- What are the success metrics?
- What are the resources (data, time, people) needed?

Scoping process



Important steps to scope the project and get the wishes of the clients (top3 of the things he wants to make better).

Other important thing: separate problem identification from solution.

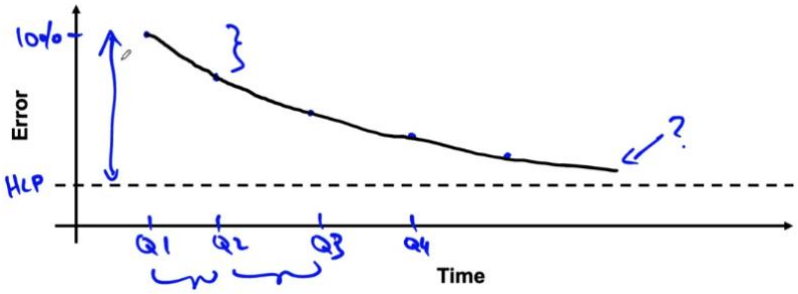
The third phase is also called diligence.

Diligence on feasibility

	Unstructured (e.g., speech, images)	Structured (e.g., transaction records)
New	HLP	Predictive features available?
Existing	HLP History of project	New predictive features? History of project

To know if possible: Do we have features that are predictive?

It is important to keep a history of project to know if interesting to continue improve a model:



Diligence on value

Try to make a metrics list from the ML team metrics to the business metrics.

Try to make both teams agree on metrics to quantify the performances.

Last point: consider ethical matters.

Specifications and milestones

Key specifications to define and write before starting the project:

- ML metrics (accuracy, precision/recall, etc.)
- Software metrics (latency, throughput, etc. given compute resources)
- Business metrics (revenue, etc.)
- Resources needed (data, personnel, help from other teams)
- Timeline

Possible to add a POC or a Benchmark for more details if unsure.

Data

Data definition

Hard, because lots of ways to label an example, especially in unstructured data:

Every example is quite good, but the labels should follow the same rules to make it standard and allow the models to learn from them-> avoid inconsistent/ambiguous labeling.

Questions to ask:

- What is the input x? (Example: lighting, contrast, resolution of pictures)
- What is the target label y? (Ensuring that the labels are consistent)

In small data, it is critical to have clean labels vs a bit less important for bid data problems.

Main problems that can occur:

Unstructured data

- May or may not have huge collection of unlabeled examples x.
- Humans can label more data.
- Data augmentation more likely to be helpful.

Structured data

- May be more difficult to obtain more data.
- Human labeling may not be possible (with some exceptions).

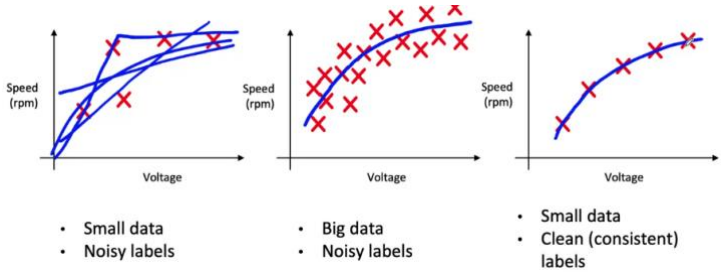
Small data $\leq$ 10,000

- Clean labels are critical.
- Can manually look through dataset and fix labels.
- Can get all the labelers to talk to each other.

Big data $>$ 10,000

- Emphasis data process.

The quality is key, especially on small data problems:



- Small data
- Noisy labels

- Big data
- Noisy labels

- Small data
- Clean (consistent) labels

Big Data problems can have small data challenges, for example if there are lots of rare events in the input.

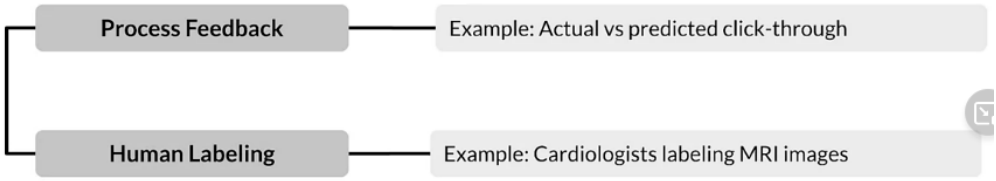
Data Labeling

Several methods:

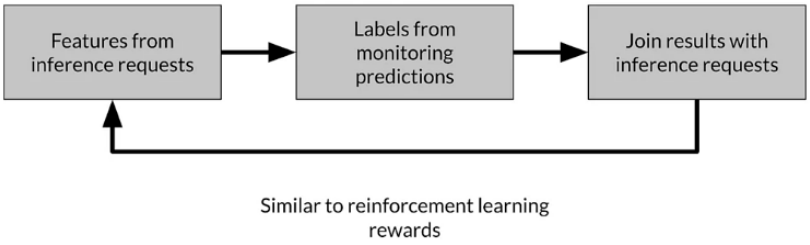
- Process Feedback (Direct labeling)
- Human labeling
- Semi Supervised labeling
- Active learning

- Weak supervision

However, the two first are the most used now.

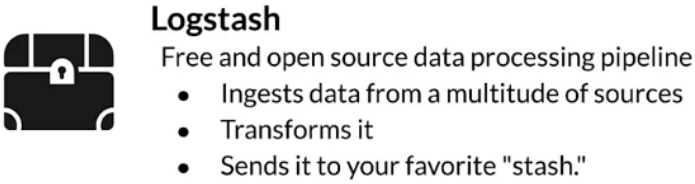


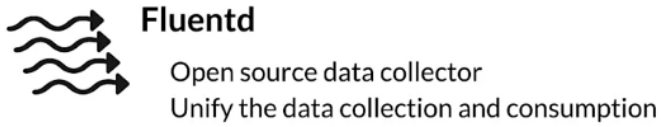
Direct labeling:



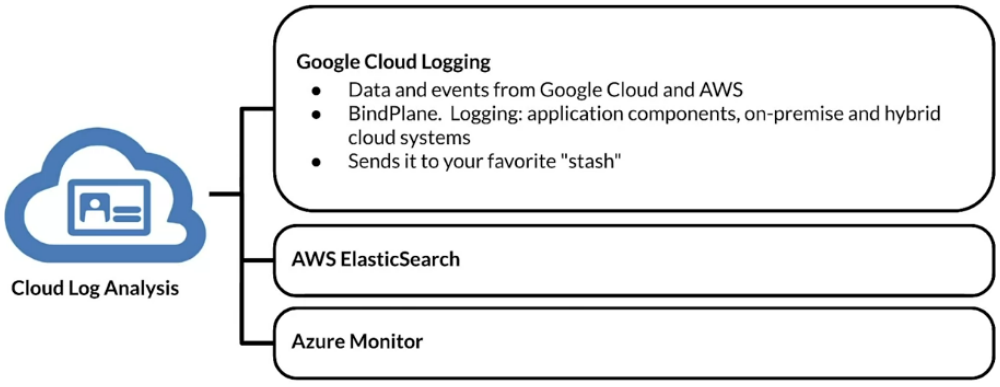
However, Process feedback is hard to use, because only a few use cases.

Feedback process tools:

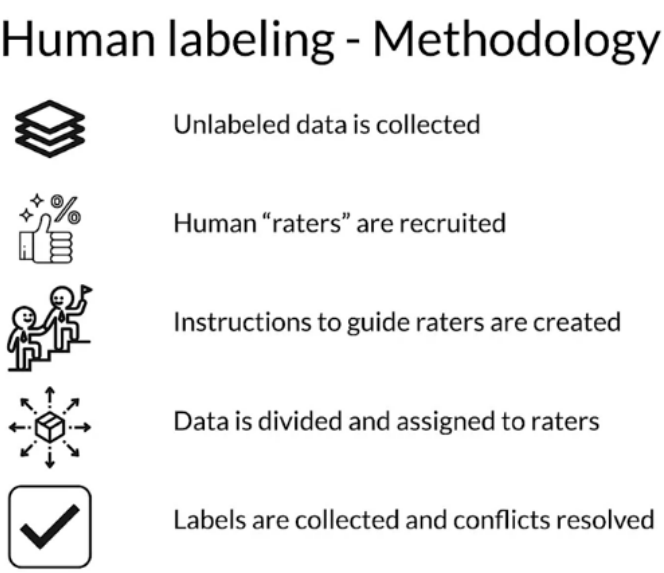




And in the Cloud:



Human labeling uses people as raters to examine data and label it.



However, sometimes raters need to be specialists, so it can cost a lot.

In addition, it can be slow, the quality consistency can fluctuate, so good but only for small datasets.

Advanced Labeling

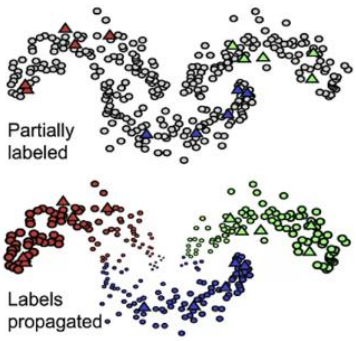
Why advanced labeling is important: labeled data is necessary for supervised learning, but

- Manually labeling of data is expensive
- Unlabeled data is usually cheap and easy to get
- Unlabeled data contains a lot of information that can improve your model

There are several powerful ways to automatically label data. 3 main types: Semi-supervised Learning, Active Learning, Weak supervision with Snorkel for example.

Semi-supervised Learning

Starting with a small pool of human labeled data, we apply the classes on a large pool of unlabeled data using the similarities, then the model is trained using both pools.



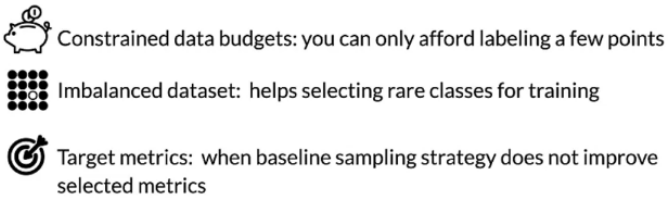
On the picture above, a graph-based propagation is used to determine the neighbor's labels, but lots of other techniques exist.

The semi-supervised technique is considered transductive learning since it does not learn a function to map the labels, but maps from the examples themselves.

Advantages: combining labeled and unlabeled data boosts accuracy, and getting unlabeled data is cheap.

Active Learning

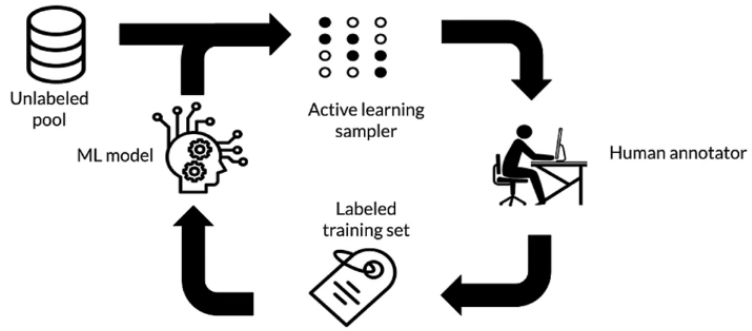
This technique is a family of algorithms to intelligently sample data by selecting the points to be labeled that would be most informative for model training. It is very efficient for:



After selected labeled examples that will best help the model learn, this method then can use two methods:

- Fully supervised: form a training dataset with only those examples
- Semi supervised: label propagation for unlabeled examples

Lifecycle:



How to do intelligent labeling: several ways.

For example, Margin sampling: after creating a model with some labeled data, this method will add the unlabeled examples that are the nearest from the decision boundary (most uncertain data point), make them get labeled, retrains the model, and continues until the model does not really improve. This method achieves getting better examples on small training sets than random sampling.

Two other techniques:

Cluster-based sampling: sample from well-formed clusters to "cover" the entire space.

Query-by-committee: train an ensemble of models and sample points that generate disagreement.

Region-based sampling: Runs several active learning algorithms in different partitions of the space.

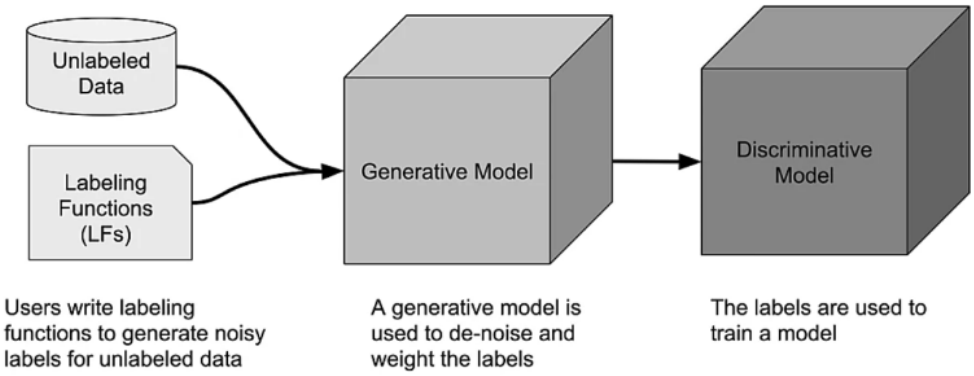
Weak supervision

This method is a subject matter expert, usually done by designing heuristics. The labels are usually noisier than usual deterministic labels.

- Unlabeled data, without ground-truth labels
- One or more weak supervision sources
  - A list of heuristics that can automate labeling
  - Typically provided by subject matter experts
- Noisy labels have a certain probability of being correct, not 100%
- Objective: learn a generative model to determine weights for weak supervision sources

Snorkel is a project started in Stanford in 2016, programmatically building and managing training datasets without manual labeling. Automatically models, cleans, and integrates the resulting training data, and offers data augmentation and slicing.

Data programming pipeline in snorkel:

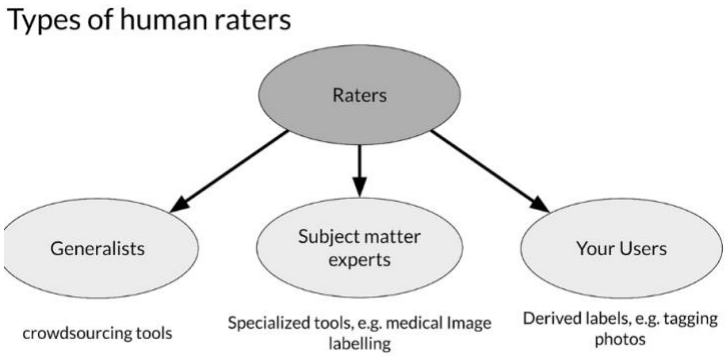


Example of how to label spams, using “my” as the first condition and the size as the second:

```
from snorkel.labeling import labeling_function

@labeling_function()
def lf_keyword_my(x):
    """Many spam comments talk about 'my channel', 'my video', etc."""
    return SPAM if "my" in x.text.lower() else ABSTAIN

@labeling_function()
def lf_short_comment(x):
    """Non-spam comments are often short, such as 'cool video!'."""
    return NOT_SPAM if len(x.text.split()) < 5 else ABSTAIN
```

Improving label consistency

- Have multiple labelers label same example
- If disagreement, have MLE, SME (subject matter expert) and labelers discuss definition of Y
- Consider changing X if labelers believe X doesn’t contain enough information
- Iterate until hard to increase agreement
- Possible to add a class to capture uncertainty



In small vs big data:

Small data

- Usually small number of labelers.
- Can ask labelers to discuss specific labels.

Big data

- Get to consistent definition with a small group.
- Then send labeling instructions to labelers.
- Can consider having multiple labelers label every example and using voting or consensus labels to increase accuracy.

Human Level Performance (HLP)

Usage:

- In academia, establish and beat a respectable benchmark to support publication.
- Business or product owner asks for 99% accuracy. HLP helps establish a more reasonable target.
- "Prove" the ML system is superior to humans doing the job and thus the business or product owner should adopt it.

The last one is to use with caution.

Raising the HLP may get the model having better results: when Y comes from a human label, HLP << 100% may indicate ambiguous labeling instructions. Correcting that will raise HLP, making ML models having more difficulties to reach that HLP, but overall increasing the results by having better labels in the dataset.

On structured data, HLP is less frequently used since less likely to involve human labelers. However, some problems can have human labeling: fraudulent transactions, spam accounts, mode of transportation in a GPS …

Data Collection

Initial data collection should not take a long time to get quickly in the loop.

To be more creative to get data, ask: How much data can we have in k days? Instead of asking for time to get k data.

Exception: if experience on the problem makes you know you need m examples.

To get data, brainstorm about the sources we have / would be able to have:

Source	Amount	Cost	Time
Owned	100h	$0	0
Crowdsourced – Reading	1000h	$10000	14 days
Pay for labels	100h	$6000	7 days
Purchase data	1000h	$10000	1 day

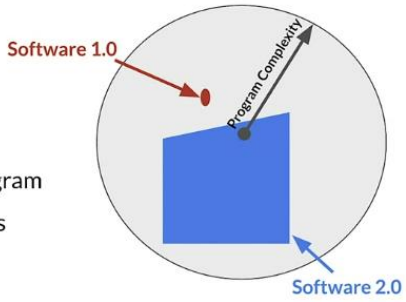
Other factors: Data quality, privacy, regulatory constraints …

Labeling data:

- Do not increase data by more than 10x at a time
- Who is qualified to label?
- Labeling data for a few days as MLEs is usually fine

Data in ML:

- Software 1.0
  - Explicit instructions to the computer
- Software 2.0
  - Specify some goal on the behavior of a program
  - Find solution using optimization techniques
  - Good data is key for success
  - Code in Software = Data in ML



Some questions to answer about the data we want to collect:

- Identify data sources
- Check if they are refreshed
- Consistency for values, units, & data types
- Monitor outliers and errors

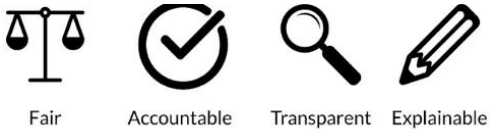
It is important to translate user needs into data needs:

- What kind of/how much data is available
- What are the details and issues of your data
- What are your predictive features
- What are the labels you are tracking
- What are your metrics

It is very important to respect user privacy, and let users have a control over their collected data.

Ways to protect personally identifiable information: aggregation (replace unique values with summary), redaction (remove some data to create less complete picture), anonymize data and let user control what data is collected.

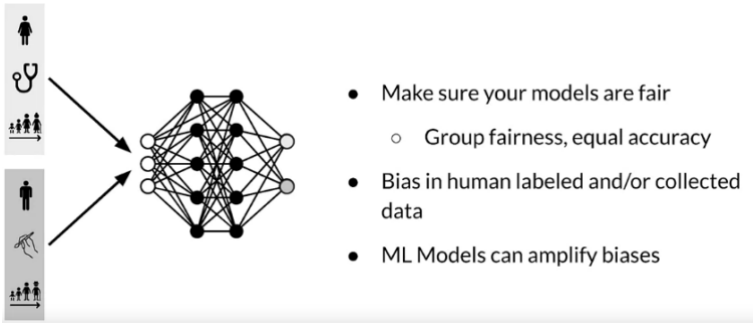
How ML can fail users:



- Representational harm
- Opportunity denial
- Disproportionate product failure
- Harm by disadvantage

Commit to fairness

Data collected by human reflects their biases.



- Make sure your models are fair
  - Group fairness, equal accuracy
- Bias in human labeled and/or collected data
- ML Models can amplify biases

In general, to have a good ML system and respect privacy, security etc.:

- Ensure rater pool diversity
- Investigate rater context and incentives
- Evaluate rater tools
- Manage cost
- Determine freshness requirements

Data Augmentation

Instead of gathering more data to expand datasets, data augmentation allows to create more data based on the data already collected.

One way is introducing minor alterations: flips, rotations for images.

It is a way to add examples that are like real examples, improves coverage of feature space, but invalid augmentations can add unwanted noise.

Example of function to augment a photo, using a TF Tensor:

```
def augment(x, height, width, num_channels):
    x = tf.image.resize_with_crop_or_pad(x, height + 8, width + 8)
    x = tf.image.random_crop(x, [height, width, num_channels])
    x = tf.image.random_flip_left_right(x)
    return x
```

Two other advanced techniques:

- Semi-supervised data augmentation (UDA), semi-supervised learning using GANs
- Policy-based data augmentation (AutoAugment)

Data pipeline

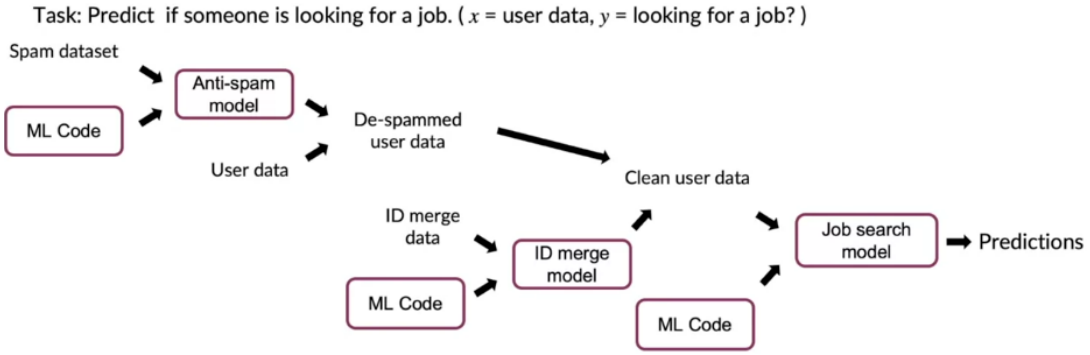
Data processing before ML to predict Y (from raw to training).

The pre-processing is very important and should be replicated from development to production phase.

However, in a POC it is ok if a part is done manually, but it is a good thing to take notes in that case.

In production, using tools such as TensorFlow Transform to replicate that pre-processing.

Example of a complete Data Pipeline:



It is very important for error analysis or during a crisis to keep track of data provenance and lineage (sequence of processing steps applied to it).

If possible, storing the metadata of each sample (especially in unstructured cases) can help in the future: time, position, author …

Train/test/dev split must be balanced for small dataset, but not for large datasets.

Difference between data in production and data in "academic" environment

Usually, data is already clean and ready-to-use.

	Academic/Research ML	Production ML
Data	Static	Dynamic - Shifting
Priority for design	Highest overall accuracy	Fast inference, good interpretability
Model training	Optimal tuning and training	Continuously assess and retrain
Fairness	Very important	Crucial
Challenge	High accuracy algorithm	Entire system

Data and Concept Change

Data and scope changes can occur.

Monitor models and validate data to find problems early is a must to do.

Changing ground truth: label new training data, can be a solution to solve data changes.

Easy problems

- Ground truth changes slowly (months, years)
- Model retraining driven by:
  - Model improvements, better data
  - Changes in software and/or systems
- Labeling
  - Curated datasets
  - Crowd-based

Harder problems

- Ground truth changes faster (weeks)
- Model retraining driven by:
  - **Declining model performance**
  - Model improvements, better data
  - Changes in software and/or system
- Labeling
  - Direct feedback
  - Crowd-based

Really hard problems

- Ground truth changes very fast (days, hours, min)
- Model retraining driven by:
  - **Declining model performance**
  - Model improvements, better data
  - Changes in software and/or system
- Labeling
  - Direct feedback
  - Weak supervision

Solutions:

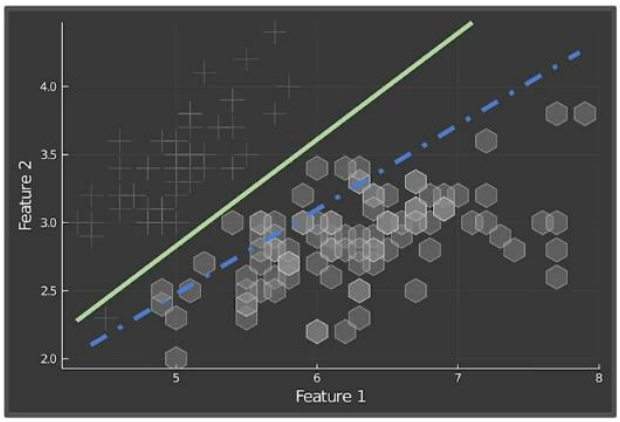
- Model retraining helps to improve performance
- Data labeling for changing ground truth and scare labels too, in structured datasets

Validating data

Drift: changes in data over time

Skew: difference between two static versions / sources of a dataset.

Concept drift detection



Data skewness detection

- Schema skew: training and serving data do not conform to the same scheme (type of data for example)
- Distribution skew: dataset shift -> covariate or concept shift
- Requires continuous evaluation

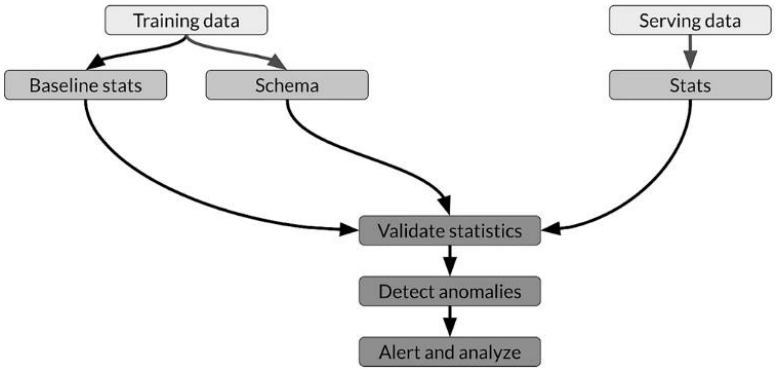
	Training	Serving
Joint	$P_{\text{train}}(y, x)$	$P_{\text{serve}}(y, x)$
Conditional	$P_{\text{train}}(y\|x)$	$P_{\text{serve}}(y\|x)$
Marginal	$P_{\text{train}}(x)$	$P_{\text{serve}}(x)$

Dataset shift $P_{\text{train}}(y, x) \neq P_{\text{serve}}(y, x)$

Covariate shift $P_{\text{train}}(y|x) = P_{\text{serve}}(y|x)$, $P_{\text{train}}(x) \neq P_{\text{serve}}(x)$

Concept shift $P_{\text{train}}(y|x) \neq P_{\text{serve}}(y|x)$, $P_{\text{train}}(x) = P_{\text{serve}}(x)$

Data skew detection workflow:



TensorFlow Data Validation

- Understand, validate, and monitor ML data at scale
- Used to analyze and validate petabytes of data at Google every day
- Proven track record in helping TFX users maintain the health of their ML pipelines

- Generates data statistics and browser visualizations
- Infers the data schema
- Performs validity checks against schema
- Detects training/serving skew

Skew in TFD:

- Supported for categorical features
  - Expressed in terms of L-infinity distance (Chebyshev Distance): $D_{Chebyshev}(x,y)=max_i(|x_i-y_i|)$
  - Set a threshold to receive warnings

Sum up:

- TFDV: Descriptive statistics at scale with the embedded facets visualizations
- It provides insight into:
  - What are the underlying statistics of your data
  - How does your training, evaluation, and serving dataset statistics compare
  - How can you detect and fix data anomalies

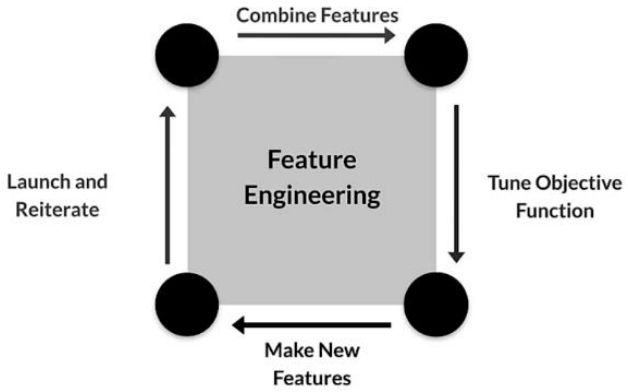
Feature Engineering in production

Feature engineering is creating features from raw data.

Why doing feature engineering:

- Making data useful before training a model
- Representing data in forms that help models learn
- Increasing predictive quality
- Reducing dimensionality with feature engineering

Lifecycle of feature engineering:

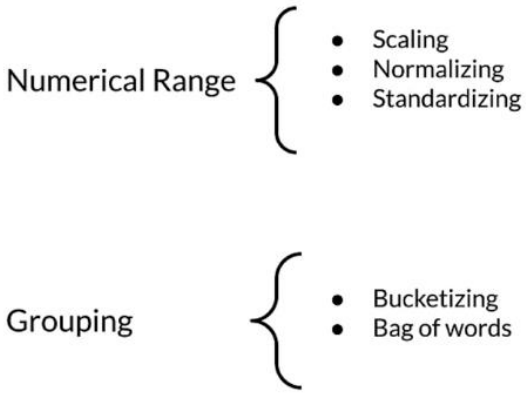


It is important to apply the feature engineering and preprocessing while serving that was used during training. Otherwise, will not work well.

Feature engineering can be very difficult but very important to success and allows models to learn better. Concentrating predictive information in fewer features enables more efficient use of compute resources.

Techniques

Lots of techniques exists:



Scaling: converts values from their natural range into a prescribed range (Ex: grayscal usually goes from [0,255] rescaled to [-1,1]) -> helps NN converge faster, avoids NaN errors, and for each feature the model learns the right weights.

Normalization: by using formulas such as Min-Max, it is good if the data is not gaussian, allows bound data between 0 and 1.

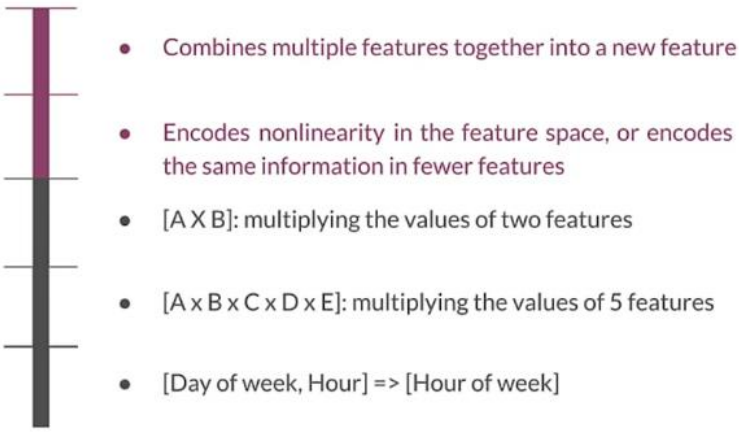
Standardization (Z-score): using standard deviations, allows scaling to center around the mean of data translated to 0, less bounded because more "gaussian".

To visualize binning, using Facets can help to that part of the job.

Dimensionality reduction in embeddings: PCA, t-SNE, UMAP, feature crossing.

To visualize how data looks like in a space, TensorFlow embedding projector is a useful tool, which is good in high-dimensional data.

Feature crosses



Preprocessing operations

Main operations:

Unstructured data preprocessing examples:

Text - stemming, lemmatization, TF-IDF, n-grams, embedding lookup

Images - clipping, resizing, cropping, blur, Canny filters, Sobel filters, photometric distortions

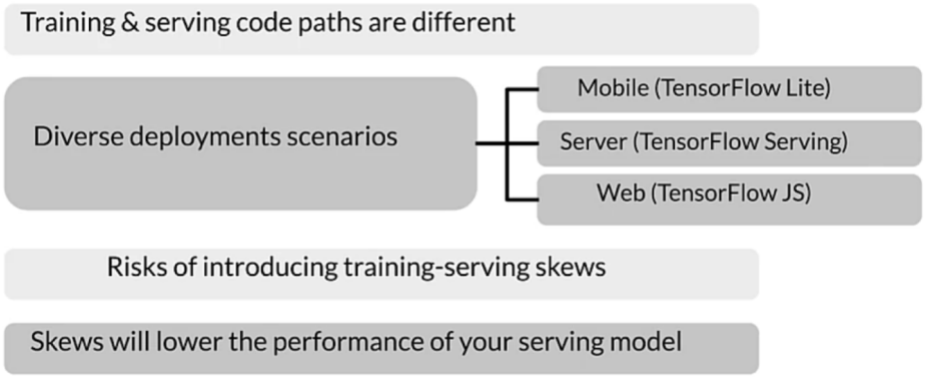
So, difference between feature engineering and data preprocessing:

- Data preprocessing: transforms raw data into a clean and training-ready dataset
- Feature engineering maps:
  - Raw data into feature vectors
  - Integer values to floating-point values
  - Normalizes numerical values
  - Strings and categorical values to vectors of numeric values
  - Data from one space into a different space

Preprocessing data at scale

A good way to proceed is to create a Pipeline, using TF for example.

Inconsistencies in feature engineering:



Preprocessing granularity:

Transformations	
Instance-level	**Full-pass**
Clipping	Minimax
Multiplying	Standard scaling
Expanding features	Bucketizing
etc.	etc.

Some preprocess can be done when getting an instance, but others need a full pass, because we need some parameters that should be the same between training set preprocessing and serving set preprocessing.

Transformation can be done within the model, but:

Pros	Cons
Easy iterations	Expensive transforms
Transformation guarantees	Long model latency
	Transformations per batch: skew

Preprocessing per batch is a good solution on big datasets too:

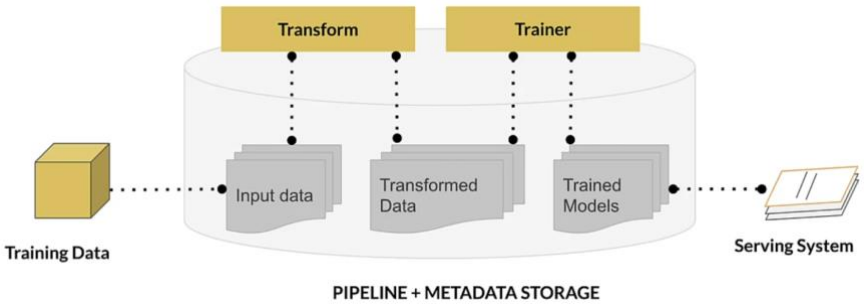
- For example, normalizing features by their average
- Access to a single batch of data, not the full dataset
- Ways to normalize per batch
  - Normalize by average within a batch
  - Precompute average and reuse it during normalization

=> 3 main challenges: balancing predictive performances, full-passing transformations on a dataset, and optimizing instance-level transformations for better training efficiency (GPUs, TPUs …).

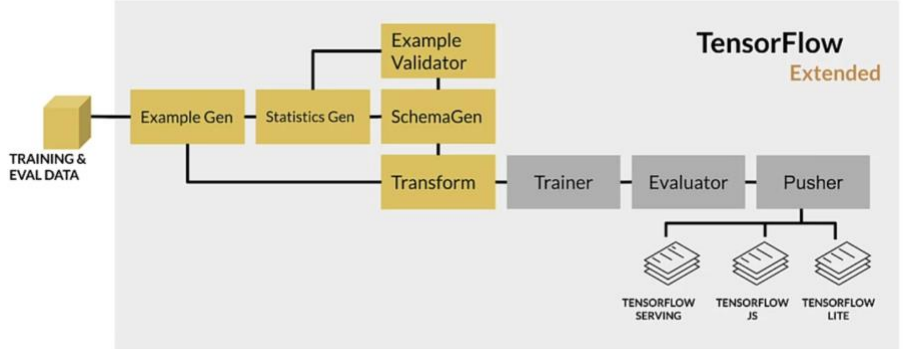
- Inconsistent data affects the accuracy of the results
- Need for scaled data processing frameworks to process large datasets in an efficient and distributed manner

TensorFlow Transform

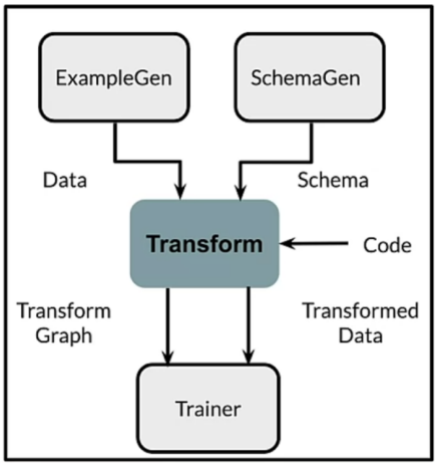
A good tool to apply features transformations at large scale.



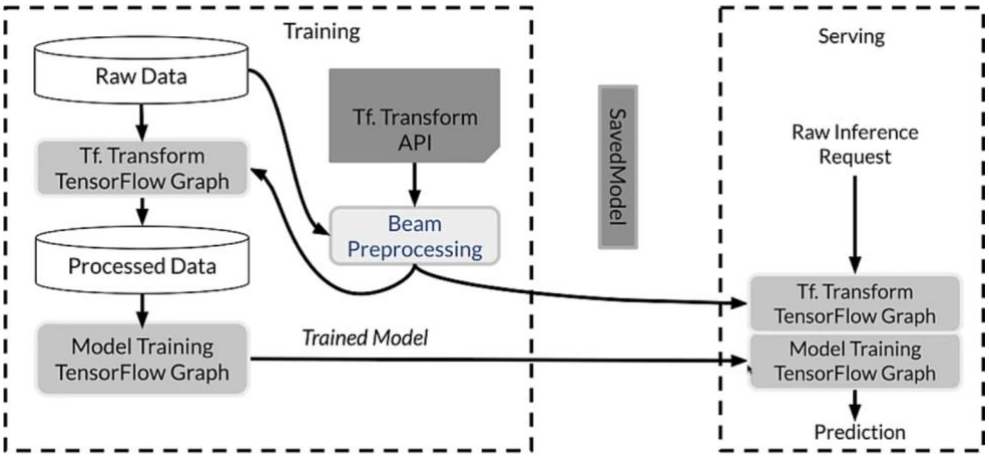
Inside of TFX:



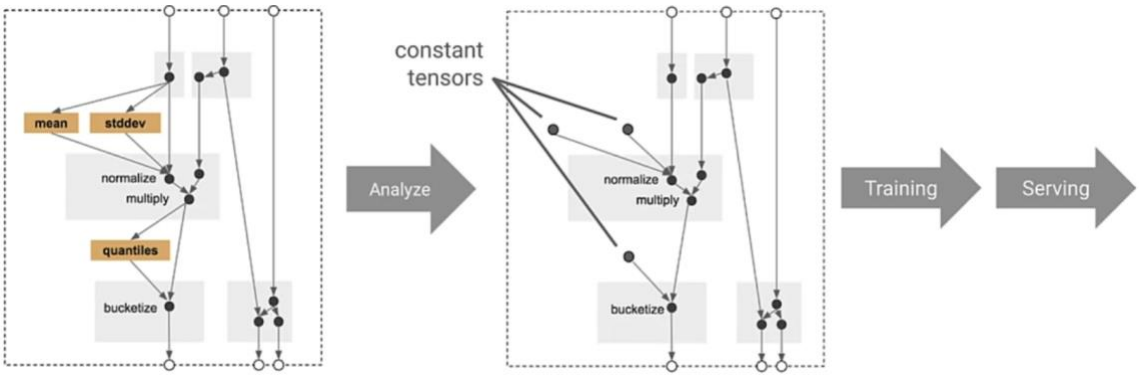
TensorFlow Transform layout:



What is inside that Transform and how it is passed to the serving part of the model for inferences:



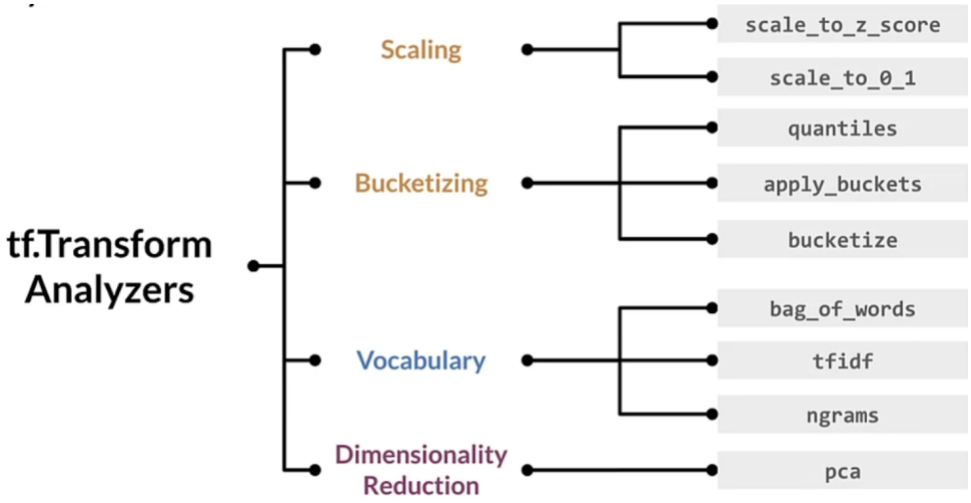
The pipeline is designed as a graph, using analyzers to determine the tensor constants:



Pros of TensorFlow Transform:

- Tf.Graph created has all the needed constants in it
- Focus on data preprocessing only at training time
- Works in-line during training and serving
- No need preprocessing code at serving time
- Always applied transformations, independently of deployment platform

Some functions of TF Analyzers:



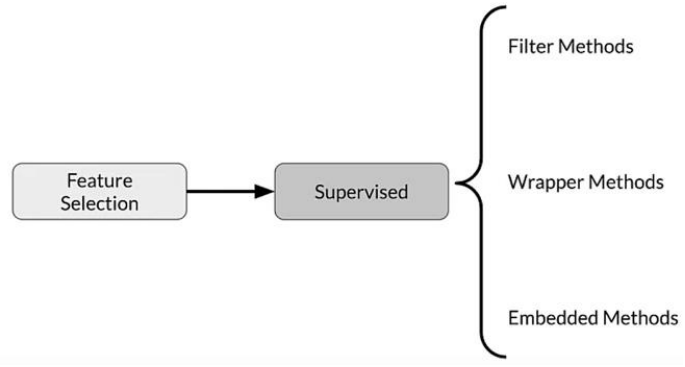
Feature Selection

The best ways to select a feature in a low dimensional problem is to plot the features and see the distribution of each feature.

Features selection: reduce storage and I/O requirements, training and inference costs too because the goal is to reduce the dimensionality by removing the useless features.

Unsupervised feature selection removes redundant features, not using the features-target relationship.

Supervised feature selection selects those with the most features-target relationship.



Filter methods

Correlated features are usually redundant, so we pick only one.

Popular filter methods: Pearson correlation, Univariate Feature Selection, Kendall Tau Rank Correlation Coefficient (monotonic relationships), Spearman's Rank Correlation Coefficient (monotonic relationships), mutual information, F-Test, Chi-squared Test.

By using a correlation matrix, we can see if features are correlated with each other to remove the redundant ones.

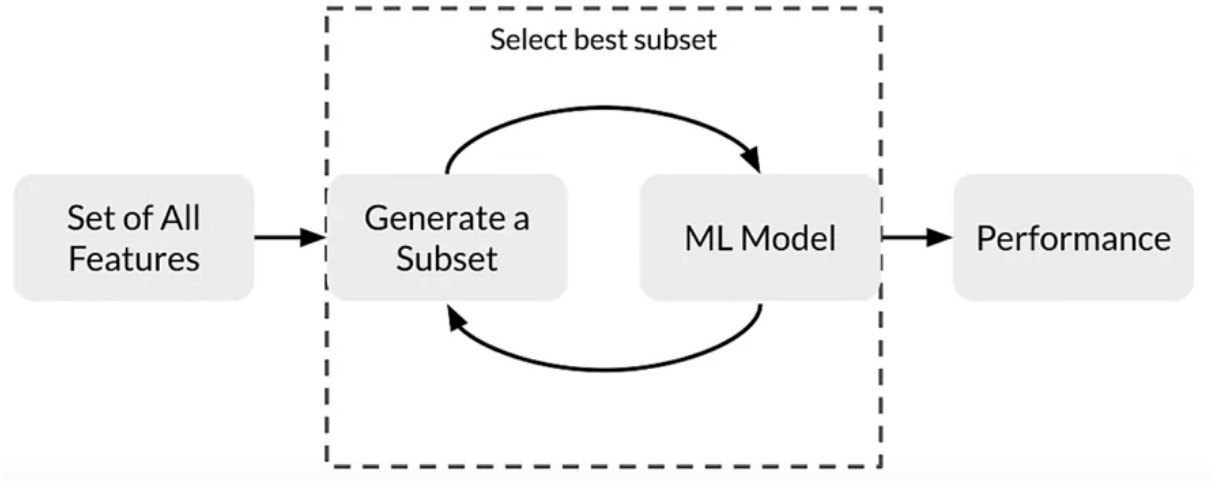
Univariate selection, using kbest to take only 20:

SelectKBest implementation

```
def univariate_selection():

  X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                          test_size = 0.2,stratify=Y, random_state = 123)

  X_train_scaled = StandardScaler().fit_transform(X_train)
  X_test_scaled = StandardScaler().fit_transform(X_test)

  min_max_scaler = MinMaxScaler()
  Scaled_X = min_max_scaler.fit_transform(X_train_scaled)
  selector = SelectKBest(chi2, k=20) # Use Chi-Squared test
  X_new = selector.fit_transform(Scaled_X, Y_train)
  feature_idx = selector.get_support()
  feature_names = df.drop("diagnosis_int",axis = 1 ).columns[feature_idx]
  return feature_names
```

Wrapper methods

Popular methods: Forward Selection, Backward Elimination, Recursive Feature Elimination.



Forward Selection: iterative greedy method. Starting with 1 feature, evaluate model performance, and add one at a time features, add next feature that gives the best performance, then keep doing it until no improvements.

Backwards Elimination: same than FS but start with all and removing the features one at a time.

RFE: Select a model to use for evaluating feature importance, select the desired number of features, fit the model, rank features by importance, discard least important features, repeat until having the desired number of features remains. This method needs a model that is available to evaluate the features importance.

Example of how to run RFE in Python to select 20 features:

```
def run_rfe():

  X_train, X_test, y_train, y_test = train_test_split(X,Y, test_size = 0.2, random_state = 0)

  X_train_scaled = StandardScaler().fit_transform(X_train)
  X_test_scaled = StandardScaler().fit_transform(X_test)

  model = RandomForestClassifier(criterion='entropy', random_state=47)
  rfe = RFE(model, 20)
  rfe = rfe.fit(X_train_scaled, y_train)
  feature_names = df.drop("diagnosis_int",axis = 1 ).columns[rfe.get_support()]
  return feature_names
rfe_feature_names = run_rfe()

rfe_eval_df = evaluate_model_on_features(df[rfe_feature_names], Y)
```

Embedded Methods

Two popular methods: L1 regularization and Feature Importance.

The feature importance method assigns a score for each feature of the data, and discards features scored lower by feature importance.

Getting the 10 best features by importance in a trained RFC model:

```
def feature_importances_from_tree_based_model_():

  X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2,
                                          stratify=Y, random_state = 123)
  model = RandomForestClassifier()
  model = model.fit(X_train,Y_train)

  feat_importances = pd.Series(model.feature_importances_, index=X.columns)
  feat_importances.nlargest(10).plot(kind='barh')
  plt.show()

  return model
```

Getting the names of these features kept:

```
def select_features_from_model(model):


  model = SelectFromModel(model, prefit=True, threshold=0.012)

  feature_idx = model.get_support()
  feature_names = df.drop("diagnosis_int",1 ).columns[feature_idx]
  return feature_names
```

Comparing all the solutions presented here for the feature selection, on a use case, a table can be created to compare the results of each method and select the best:

Method	Feature Count	Accuracy	ROC	Precision	Recall	F1 Score
All Features	30	0.96726	0.964912	0.931818	0.9761900	0.953488
Correlation	21	0.97420	0.973684	0.953488	0.9761904	0.964705
Univariate Feature Selection	20	0.96031	0.95614	0.91111	0.97619	0.94252
Recursive Feature Elimination	20	0.9742	0.973684	0.953488	0.97619	0.964706
Feature Importance	14	0.96726	0.96491	0.931818	0.97619	0.953488

Data Journey

Artifact: any data stored.

Data provenance = data lineage: chain of transformations that led to the creation of a particular artifact. This is very important to debug and reproduce the data.

Data provenance is important for:

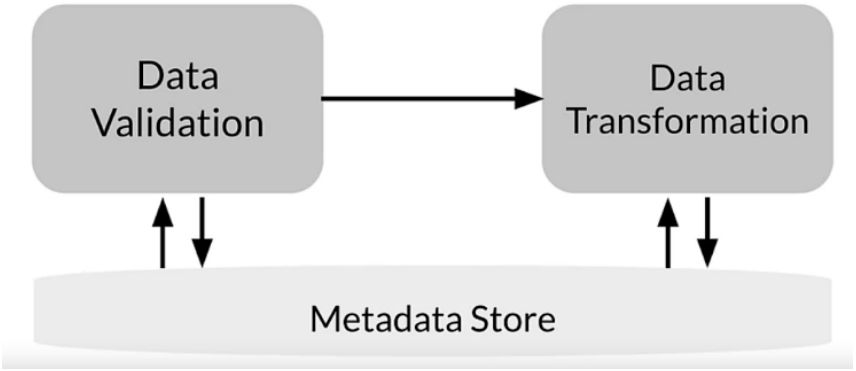
It is important to determine how the data is stored / manipulated to ensure that it respects the data regularization like GDPR.

Data versioning is an important element: machine learning requires reproducibility, by controlling the versions of datasets. Some tools available for that: DVC, Git-LFS.

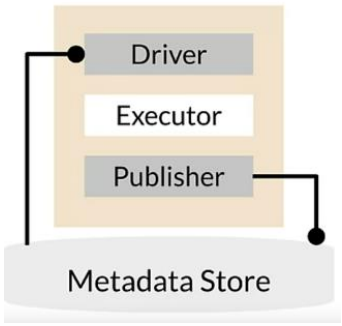
ML Metadata

Tracks metadata flowing between components in a pipeline.

It is an equivalent to logging in software development.



In TFX, in addition to the executor, each component includes two additional parts: the driver (supplies the required metadata to the executor) and the publisher (stores the result into metadata).

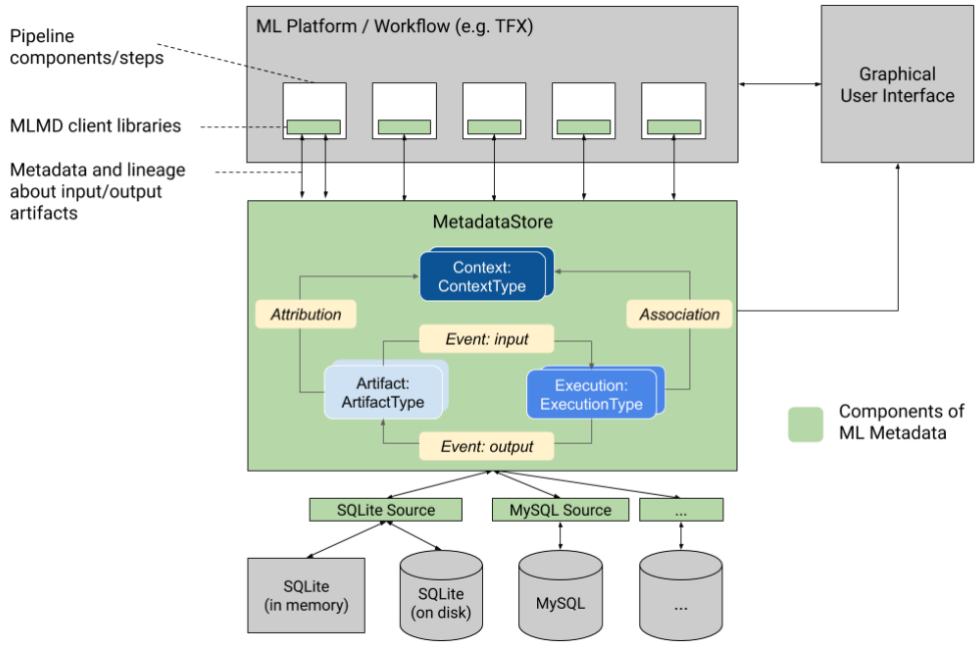


Some ML Metadata terminologies:

Units	Types	Relationships
Artifact	ArtifactType	Event
Execution	ExecutionType	Attribution
Context	ContextType	Association

The 3 elements of the units are stored in the metadata to track the flow.

Inside Metadata Store:



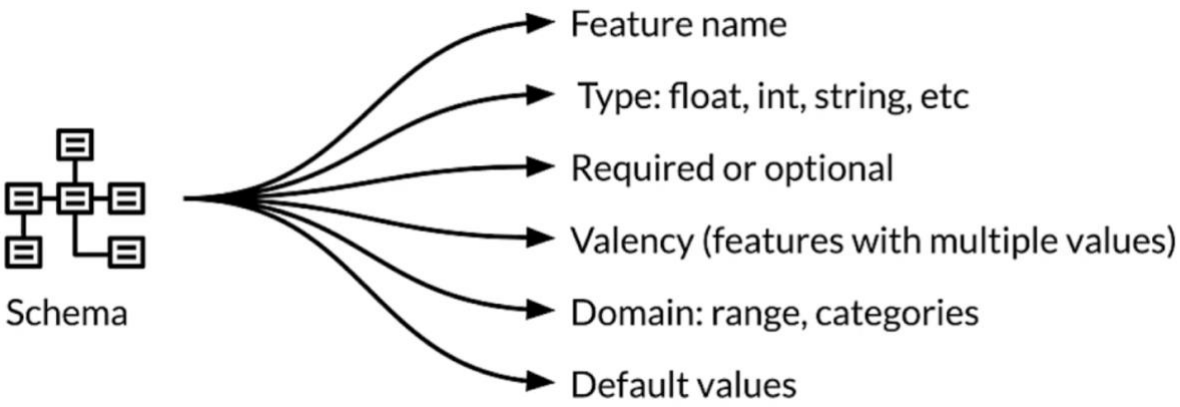
ML Metadata's backend has several storage solutions:

- ML metadata registers metadata in a database called Metadata Store
- APIs to record and retrieve metadata to and from the storage backend:
  - Fake database: in-memory for fast experimentation/prototyping
  - SQLite: in-memory and disk
  - MySQL: server based
  - Block storage: File system, storage area network, or cloud based

Data evolution

Schema development

A schema is used to describe a feature. It's an iterative process of creation.



Usually data evolves -> distribution changes, so the schema too. The schema can be used to detect this change, or anomalies.

The platform used to manipulate the data should be resilient to lots of parameters: inconsistent data, software runtime errors, user configurations. It should be highly scalable too, keeping GPUs / TPUs busy.

If these criteria are respected: it gets easy to detect anomalies, data errors are treated like code errors, and alerting when schema update is needed. Schemas are very important: it is possible to look at schema versions to track data changes but can drive other automated processes.

Business use-case needs to support data from different sources.

Data evolves rapidly

Is anomaly part of accepted type of data?

Example of how TFDV can use schemas to detect anomalies:

```
stats_options = tfdv.StatsOptions(schema=schema,
                                  infer_type_from_schema=True)

eval_stats = tfdv.generate_statistics_from_csv(
    data_location=SERVING_DATASET,
    stats_options=stats_options
)

serving_anomalies = tfdv.validate_statistics(eval_stats, schema)
tfdv.display_anomalies(serving_anomalies)
```

Schema environments

Sometimes we need to maintain several schemas, for example if two use cases serve/need two types of data for the same feature, or a schema for the training with the label and another one for

production with no label -> we define an environment for each schema version, to link it to a business use case.

Example of how to use two separate labels depending on the environment (here TRAINING and SERVING):

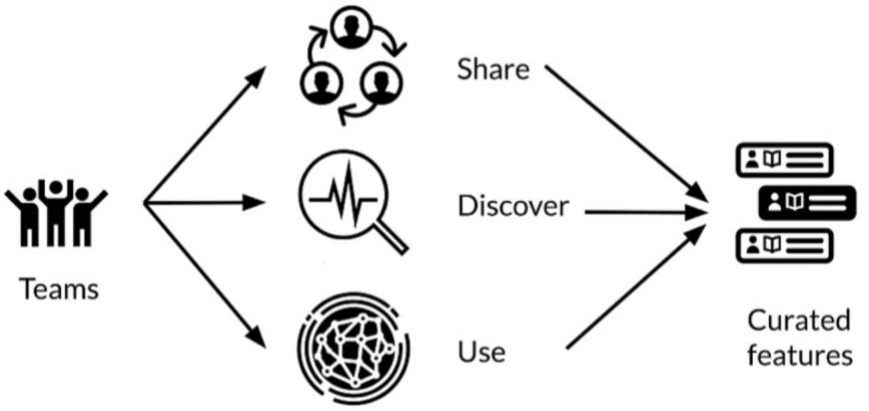
```
schema.default_environment.append('TRAINING')
schema.default_environment.append('SERVING')

tfdv.get_feature(schema, 'Cover_Type')
    .not_in_environment.append('SERVING')
```

And to use if to detect anomalies, defining the environment in the request:

```
serving_anomalies = tfdv.validate_statistics(eval_stats,
                                             schema,
                                             environment='SERVING')


tfdv.display_anomalies(serving_anomalies)
```

Storage

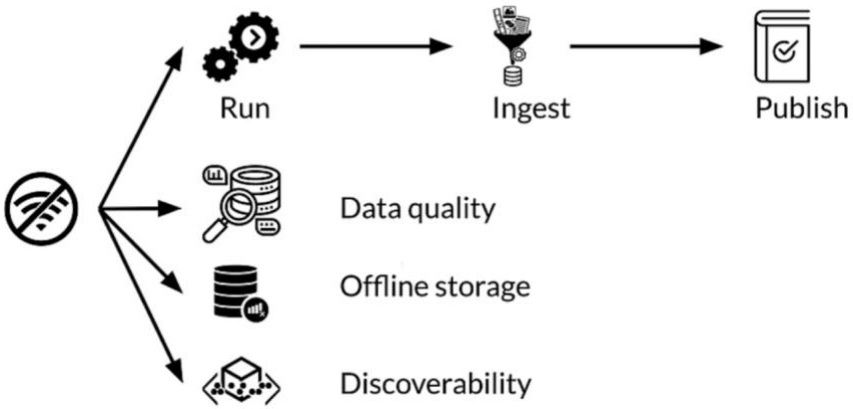
Features stores



In many cases, a Feature Store can be seen as the interface between Feature Engineering and Model Development.

ML problems use similar features to create a model, so companies tend to use stores to avoid duplication, access to the data can be controlled, and data can be aggregated to form new features (even anonymized or purged).

These tools are useful for offline feature processing since they can process several tasks:



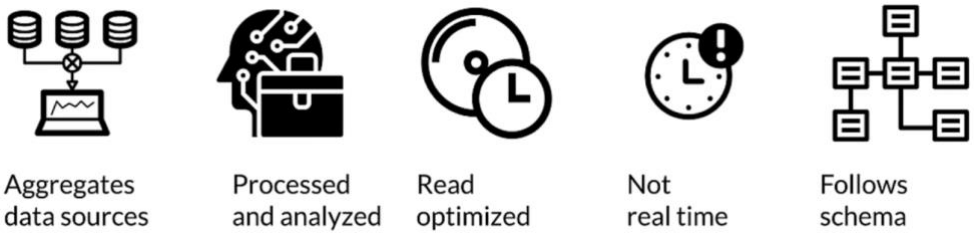
But also online:

Batch can be used in features stores because it is simple and efficient, works well for features to only be update every few hours / once a day, and uses same data for training and serving to avoid training/serving skew.

To sum up:

- Managing feature data from a single person to large enterprises.
- Scalable and performant access to feature data in training and serving.
- Provide consistent and point-in-time correct access to feature data.
- Enable discovery, documentation, and insights into your features.

Data Warehouse



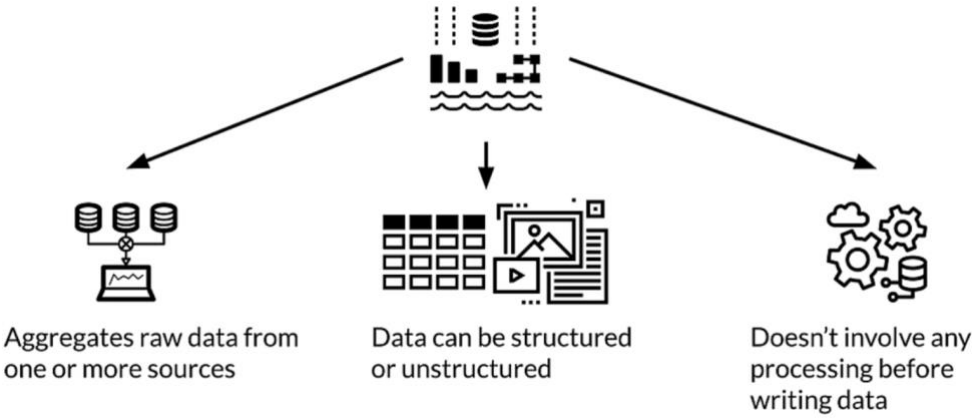
A data warehouse is subject oriented, might be collecting data from several sources and timestamping it to keep a track. It is also nonvolatile, so previous data is not erased when new data is added, so the data is accessible in a function of time to understand how data has evolved.

Why using a data warehouse:

Difference between data warehouse and database:

Data warehouse	Database
Online analytical processing (OLAP)	Online transactional processing (OLTP)
Data is refreshed from source systems	Data is available real-time
Stores historical and current data	Stores only current data
Data size can scale to >= terabytes	Data size can scale to gigabytes
Queries are complex, used for analysis	Queries are simple, used for transactions
Queries are long running jobs	Queries executed almost in real-time
Tables need not be normalized	Tables normalized for efficiency

Data Lakes



In data lakes, data are stored with no schemas, it is just raw data.

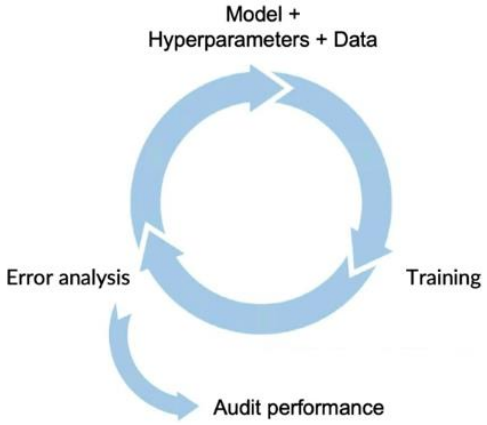
	Data warehouses	Data lakes
Data Structure	Processed	Raw
Purpose of data	Currently in use	Not yet determined
Users	Business professionals	Data scientists
Accessibility	More complicated and costly to make changes	Highly accessible and quick to update

To sum up about the three storage solutions presented:

- **Feature store**: central repository for storing documented, curated, and access-controlled features, specifically for ML.
- **Data warehouse**: subject-oriented repository of structured data optimized for fast read.
- **Data lakes**: repository of data stored in its natural and raw format.

Modeling

Modeling cycle



Key metrics

Key metrics should be the ones that will have a sense on the business side.

Verification should be done on key slices of the dataset, making sure for example that the model does not discriminate a class.

It is important to pay attention to rare classes on skewed data distributions.

Performance baseline

Before trying to improve a model, it is important to compare it to a performance baseline to know if it is possible / interesting to improve it -> give an idea about Bayes error.

To establish a baseline: Human Level Performance (HLP), read literature, quick-and-dirty implementation, performance of older system …

How to get started

- Literature search to see what's possible
- Find open-source implementations if available
- A reasonable algorithm with good data will often outperform a great algorithm with not so good data

Consider deployment constraints if baseline established and goal is to build and deploy, but not necessarily if goal is to see what's possible or if no baseline was established.

Important to do sanity-check for code and algorithm: try to overfit a small training dataset before training on a large one.

Neural Architecture Search

It is a technique for automating the design of ANN, helping to find the optimal architecture. AutoML is an algorithm to automate this search.

Hyperparameter tuning

2 types of parameters: trainable parameters learned during the training process, and hyperparameters that are set before the training.

Manual hyperparameter is not scalable because numerous even for small models.

Automation is key. Keras Tuner for example automates hyperparameter tuning in TF2 using various methods.

It can be used for example to optimize the number of layers for hidden layer of a simple NN with this function:

```
def model_builder(hp):
  model = keras.Sequential()
  model.add(keras.layers.Flatten(input_shape=(28, 28)))

  hp_units = hp.Int('units', min_value=16, max_value=512, step=16)
  model.add(keras.layers.Dense(units=hp_units, activation='relu'))
  model.add(tf.keras.layers.Dropout(0.2))
  model.add(keras.layers.Dense(10))

  model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
  return model
```

The strategy needs to be defined for the tuner:

```
tuner = kt.Hyperband(model_builder,
                     objective='val_accuracy',
                     max_epochs=10,
                     factor=3,
                     directory='my_dir',
                     project_name='intro_to_kt')
```

A condition to stop earlier the tuning when reached can be defined like this:

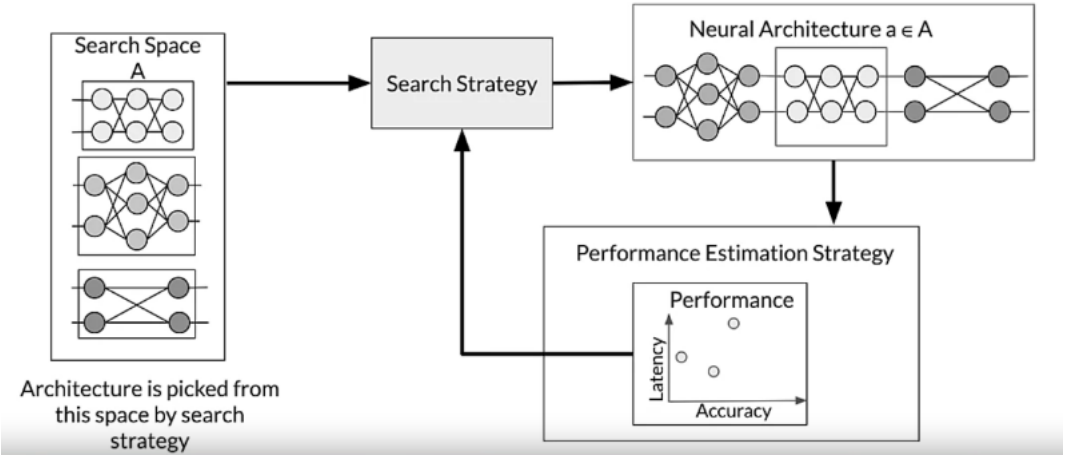
```
stop_early =
    tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                     patience=5)

tuner.search(x_train,
             y_train,
             epochs=50,
             validation_split=0.2,
             callbacks=[stop_early])
```

AutoML

Meaning: Automated Machine Learning.

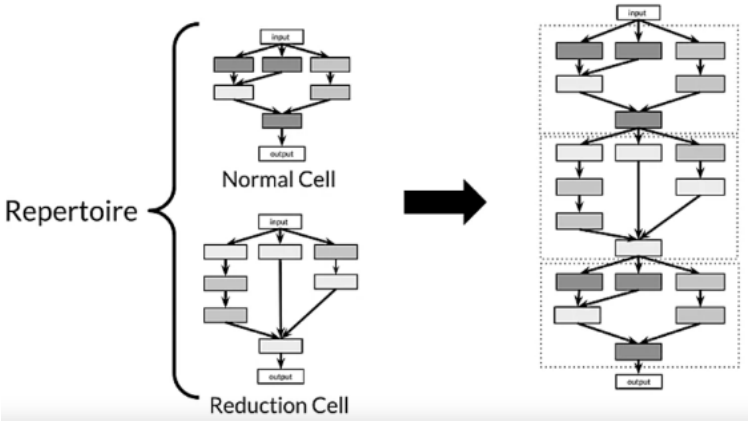
Automates ML end-to-end, to sometimes outperforms manually tuned models, covers the whole ML workflow from the raw dataset to the monitoring of the deployable model.

Neural Architecture Search is composed of 3 parts: the search space, the search strategy, and the performance estimation strategy.



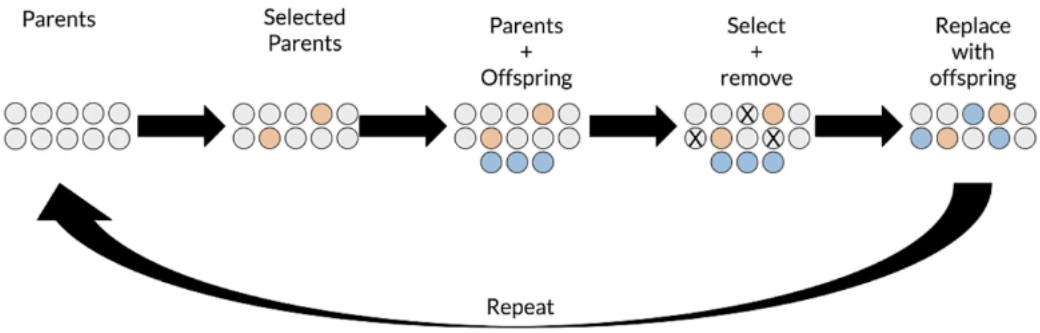
NAS is a subfield of AutoML, that focuses on automating the design of the ANN.

Search Spaces: Micro approach seems to have better results than the Macro Architecture, creating the NN by adding sub networks one after the other:

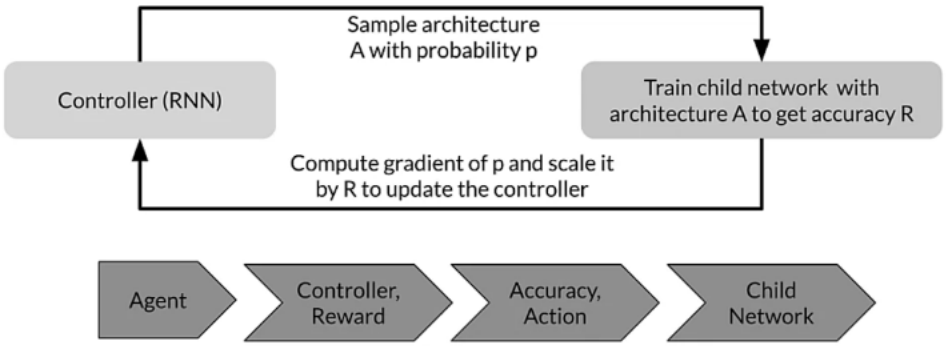


Search strategies: several methods exist to search for the best architecture. Some of them are:

- Grid Search: tests all the options of architectures within a defined space, which is not optimal for bigger networks.
- Random Search: same as grid search, but the next architecture tested is generated randomly, which is not scalable either.
- Bayesian Optimization: assumes that a specific probability distribution is underlying the performance, guiding the selection of next option by constraining the probability distribution of tested architectures.
- Evolutionary Algorithms: parents' architectures are tested, then some parents (best results) are selected, and offspring are generated based on changes done on the selected parents (changing the number of layers etc.), in an iterative process:



- Reinforcement Learning: using a string to describe the network, we can create a reward function to compute the agent's performances and determine the architecture of the next child:

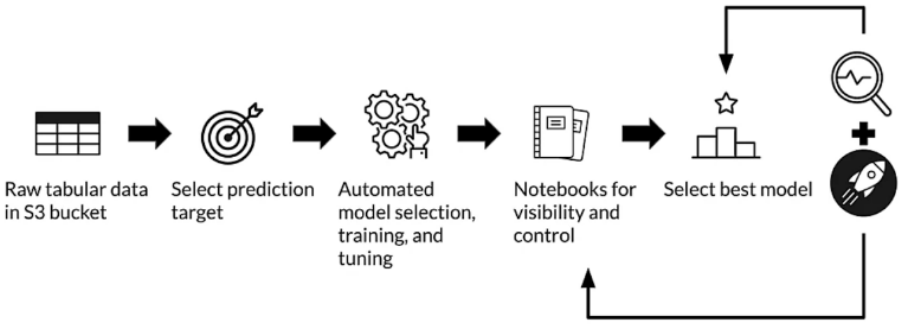


Performance estimation strategy: we need to estimate the performance of the models to choose the best one. The search can take days and cost a lot of money, so the strategy needs to be well chosen based on the need. Some strategies for example focus on reducing the cost:

- Lower fidelity estimates: reduce the training time, by using data training subset, lowering the resolution of images, or reduces the filters and cells for example.
- Learning curve extrapolation: require predicting the learning curve reliably, extrapolating it based on initial learning, removes poor performers to stop iterating on them.
- Weight inheritance / Network morphism: initialize weights of new architectures based on previously trained architectures, which is like transfer learning. It uses Network Morphism, with the underlying function that doesn't change to make the new network inherit the knowledge from parent network every time (computational speed up and no network size inheritance boundary).

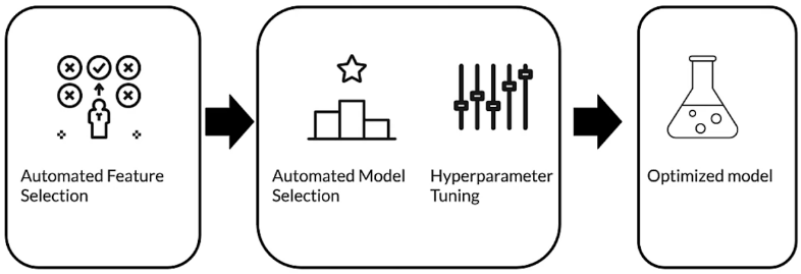
AutoML on the Cloud

Amazon SageMaker Autopilot offers a full pipeline with high visibility, from raw data to deployment:

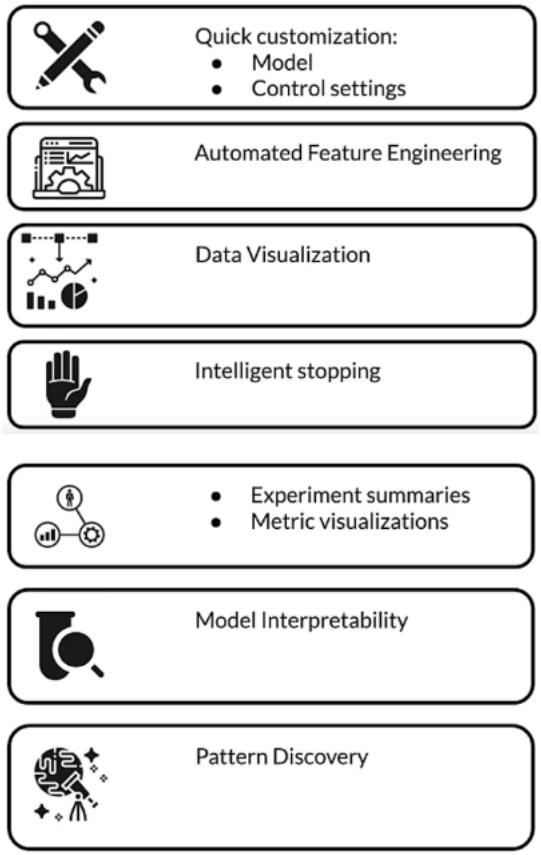


The quick iterations allow to have high quality models, rank the performance of each model, selects features, and allows to create notebooks for each model to reproduce them.

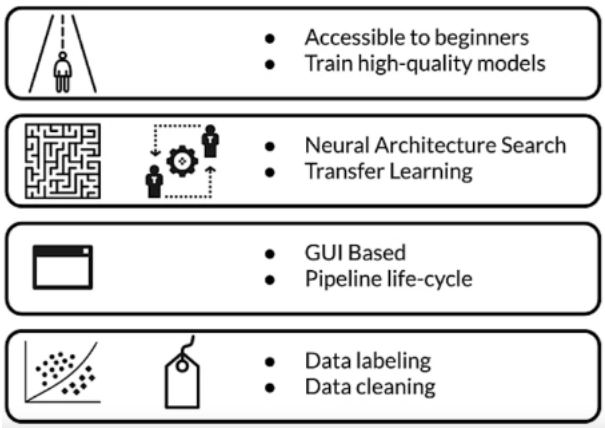
Microsoft Azure AutoML starts with the automated feature selection, then selects the model and tunes the hyperparameters, to output an optimized model:



Its key features are:



Google Cloud AutoML is a suite of ML products to train high quality models specific to the business needs to the customers:



There are several products available:

Sight	Auto ML Vision Derive insights from images in the cloud or at the edge.	Auto ML Video Intelligence Enable powerful content discovery and engaging video experiences.
Language	AutoML Natural Language Reveal the structure and meaning of text through machine learning.	Auto ML Translation Dynamically detect and translate between languages.
Structured Data	AutoML Tables Automatically build and deploy state-of-the-art machine learning models on structured data.	

In addition, there are more precise tools depending on the need, for example AutoML Vision Classification and AutoML Vision Object Detection, or AutoML Video Intelligence Classification and AutoML Video Object Detection.

However, we do not know how AutoML is processed on the Cloud, but there are high chances that they use the same algorithms as presented above.

Error analysis example

Manually, in a spreadsheet for example:

Example	Label	Prediction	Car noise	People noise	Low bandwidth
1	"Stir fried lettuce recipe"	"Stir fry lettuce recipe"	1		
2	"Sweetened coffee"	"Swedish coffee"		1	1
3	"Sail away song"	"Sell away some"		1	
4	"Let's catch up"	"Let's ketchup"	1	1	1

But tools of MLOps exist to do this, and then some metrics to calculate could be:

- What fraction of errors has that tag?
- Of all data with that tag, what fraction is misclassified?
- What fraction of all the data has that tag?
- How much room for improvement is there on data with that tag?

Prioritizing what to work on

Gap between error and HLP can be a first step for the choice, but it is better to multiply that percentage to the amount of data impacted by that error -> gives a better weight of this error.

Decide on most important categories to work on based on:

- How much room for improvement there is.
- How frequently that category appears.
- How easy is to improve accuracy in that category.
- How important it is to improve in that category.

Solutions to improve the results:

- Collect more data
- Use data augmentation to get more data
- Improve label accuracy/data quality

Performance Audit

This is an important step to avoid problems in production, which is auditing performances one last time. The Brainstorming phase will be better as a team than as a lonely person, because people can think about other aspects we haven't thought about.

1. Brainstorm the ways the system might go wrong.
   - Performance on subsets of data (e.g., ethnicity, gender).
   - How common are certain errors (e.g., FP, FN).
   - Performance on rare classes.
2. Establish metrics to assess performance against these issues on appropriate slices of data.
3. Get business/product owner buy-in.

Data-centric method

It is generally better to spend time on data rather than on the model.

Data Augmentation can be used to correct issues on data the algorithm gets poor results on:

Create realistic examples that (i) the algorithm does poorly on, but (ii) humans (or other baseline) do well on

Checklist:

- Does it sound realistic?
- Is the x → y mapping clear? (e.g., can humans recognize speech?)
- Is the algorithm currently doing poorly on it?

Adding more data almost never hurt performance, especially in the case of unstructured data when X->Y is clear (a human can easily find Y giving X) and the model is large.

On structured data, Data Augmentation can be better by adding features, regarding the problem.

Model resource management techniques

Dimensionality reduction

Datasets nowadays are often high-dimensional (with lots of features).

NN will perform a kind of automatic feature selection, but that is not as efficient as a well-designed dataset and model, with much of the model that can be largely shut off to ignore unwanted features.

Curse of dimensionality:

- Risk of overfitting
- Distances grow more and more alike
- Concentration phenomenon for Euclidean distance
- Harder to visualize and compute
- Need more observations to converge

The more features there are, the more values the parameters can have (combined) so the harder it is for the model to learn:

1-D

1	2	3	4	5

2-D

(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)
(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)
(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)
(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)
(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)

... ...

Reasons why the Euclidean distance is not good for high dimensionality spaces:

Euclidean distance $d_{ij} = \sqrt{\sum_{k=1}^{n}(x_{ik} - x_{jk})^2}$

- New dimensions add non-negative terms to the sum
- Distance increases with the number of dimensions
- **For a given number of examples**, the feature space becomes increasingly sparse

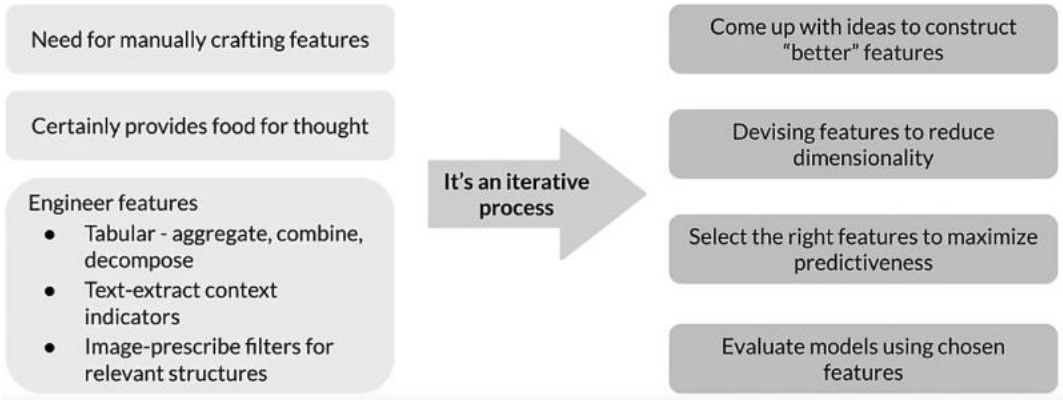
For classification performance, the increasing dimensionality suffers the Hughes effect: the results increase with the number of features, until it reaches a maximum and will then reduce the results (if the dataset stays the same size).

More features are not better if they do not add predictive information and increase the number of training instances exponentially.

Therefore, it might be useful to reduce the dimensionality.

Manual dimensionality reduction:

- Feature selection
- Feature engineering, combining features to reduce dimensionality by finding and re-expressing the patterns in data after reduction. It can be done in an iterative process:

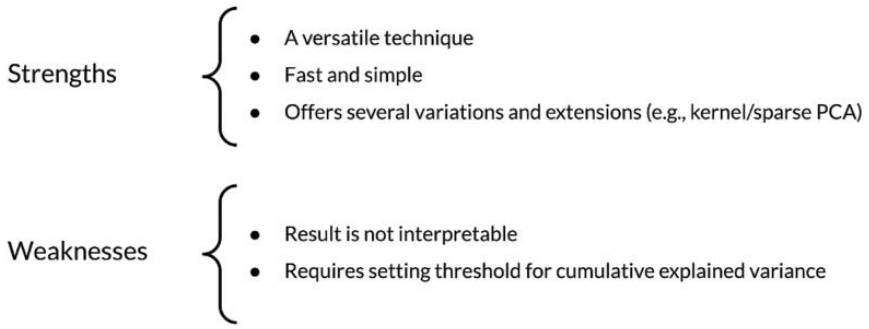


There are also algorithmic solutions to do the dimensionality reduction. Linear dimensionality reduction consists in linearly project an n-dimensional data onto a k-dimensional subspace (k<n), using LDA, PCA, or PLS to do so.

Principal Component Analysis is a one of the most widely used unsupervised technique to do so in two steps until we have k orthogonal lines:

- PCA rotates the samples, so they are aligned with the axis, and shifts the samples so they have a mean of 0.
- It learns the directions in which the data vary the most, by maximizing the variance of the data. Every direction needs to be orthogonal to avoid redundant components.

When to use PCA:



Other techniques exist, such as Latent Semantic Indexing/Analysis for SVD (Singular Value Decomposition) or Independent Component Analysis (ICA) for unsupervised cases, Non-Negative Matrix Factorization (NMF) for Matrix Factorization, or Latent Dirichlet Allocation (LDA) for latent dimensionality reduction methods.

SVD decomposes non-square matrices, which is useful for sparse matrices as produced by TF-IDF to remove redundant features in a dataset.

PCA seeks directions in features space that minimize reconstruction error (uncorrelated factors).

ICA seeks directions that are most statistically independent (independent factors, addresses higher order dependance). It is usually used to factorize a signal:

- Assume there exists independent signals:

  $S = [s_1(t), s_2(t), \ldots, s_N(t)]$
- Linear combinations of signals: $Y(t) = A\,S(t)$
  - Both A and S are unknown
  - A - mixing matrix
- Goal of ICA: recover original signals, $S(t)$ from $Y(t)$

PCA vs ICA:

	PCA	ICA
Removes correlations	✓	✓
Removes higher order dependence		✓
All components treated fairly?		✓
Orthogonality	✓	

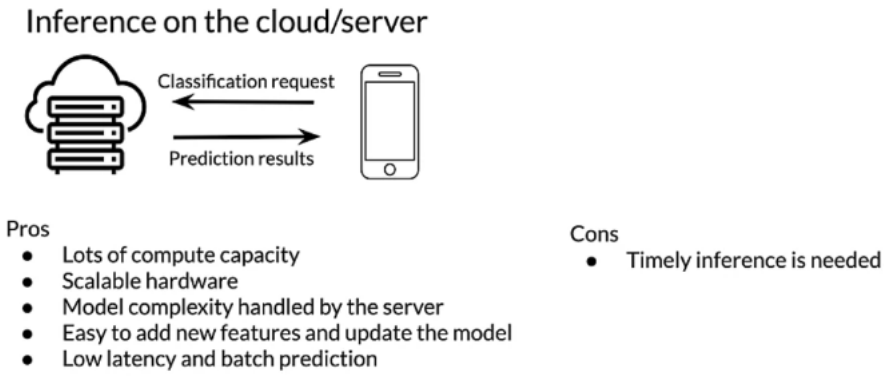
NMF models are interpretable and easier to understand but requires the sample features to be non-negative.

Quantization and pruning

Model optimization is the area of focus where you can further optimize performance and resource requirements.

Mobile, IoT, and Embedded Applications are using more and more ML, growing exponentially. It demands to move ML capability from cloud to on-device, so optimization need to be done.

On-device vs on Cloud inference:



Some solutions to use ML on mobiles / IoT platforms:

Options	On-device inference	On-device personalization	On-device training	Cloud-based web service	Pretrained models	Custom models
ML Kit	✓	✓		✓	✓	✓
Core ML	✓	✓	✓		✓	✓
* TensorFlow Lite	✓	✓	✓		✓	✓

Quantization involves transforming a model into an equivalent representation that uses parameters and computations at a lower precision. It can result in a lower model accuracy but better computing performances.

Why quantize NN? NN have many parameters and take up space, it will then shrink model file size, reduce computational resources, make models run faster and use less power with low precision.

Trade-off of quantization:

- Optimizations impact model accuracy
  - Difficult to predict ahead of time
- In rare cases, models may actually gain some accuracy
- Undefined effects on ML interpretability

Post-training quantization can be done too. For example, we can transform a TF Model to a TF Lite Model, by converting weights for example from float values to integers.

Some examples of post-training quantization techniques:

Technique	Benefits
Dynamic range quantization	4x smaller, 2x-3x speedup
Full integer quantization	4x smaller, 3x+ speedup
float16 quantization	2x smaller, GPU acceleration

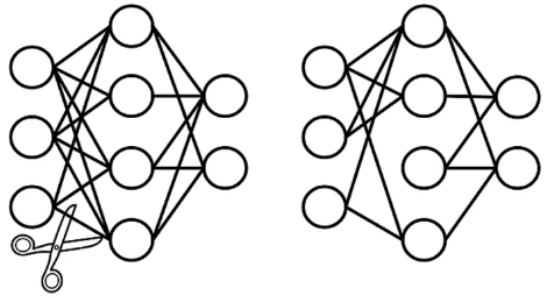
This is how to use TF Lite to convert a TF Model:

```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]

tflite_quant_model = converter.convert()
```

It is very important to evaluate model accuracy after scaling. If the loss is not acceptable, Quantization-aware training can be tried.

Quantization-aware training inserts fake quantization nodes in the forward pass, to fine-tune the weights on quantization optimizations.

Another technique of optimization is **pruning**. The goal of this technique is to remove useless parts of the model, to reduce the amount of parameters involved:



Sparse Models use less memory and are more efficient than larger models. The goal of pruning is to apply sparsity to reach a percentage of sparsity in the network.

Pruning allows better storage and transmission (reducing the size of the model), gains speedups in CPU and some ML accelerators, it can be used in tandem with quantization to get additional benefits, and it unlocks performance improvements.

High Performance Training

More data and more epochs make the training process last longer. Therefore, it is important to focus on finding good solutions to make the training more performant to reduce the waiting time.

Distributed training

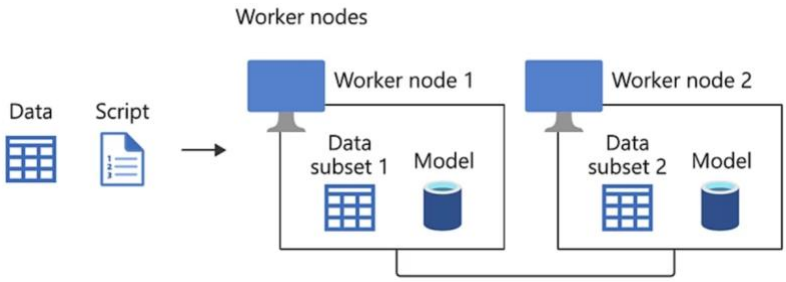
Allows to train huge models and speed up training. Two methods:

- Data parallelism: models are replicated between several GPUs/TPUs, and data is split between them. Each node will compute backpropagation and communicate all its changes to the other nodes, so they update their weights. Every node needs to synchronize their gradients at the end of each batch to ensure that they are training a consistent model.
- Model parallelism: when models are too large to fit on a single device then they can be divided in several partitions, assigning different partitions to different accelerators. It is a more complex topic.

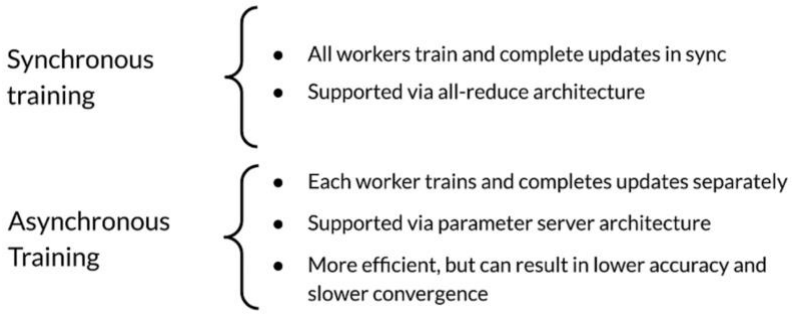
TF has a lot of tools to perform distributed training in tf.distribute.Strategy:

- One-device strategy: no distribution
- Mirrored strategy: one machine with multiple GPUs, one replica per GPU and variables mirrored.
- Parameter Server strategy: some machines are designated as workers and will use variables that are stored on designated parameter machines, using asynchronous data parallelism by default. It is very impossible to add a fault tolerance however because if a worker fails then the whole training fails.
- Multi-Worker Mirrored strategy.
- Central Storage strategy
- TPU strategy

Data parallelism:



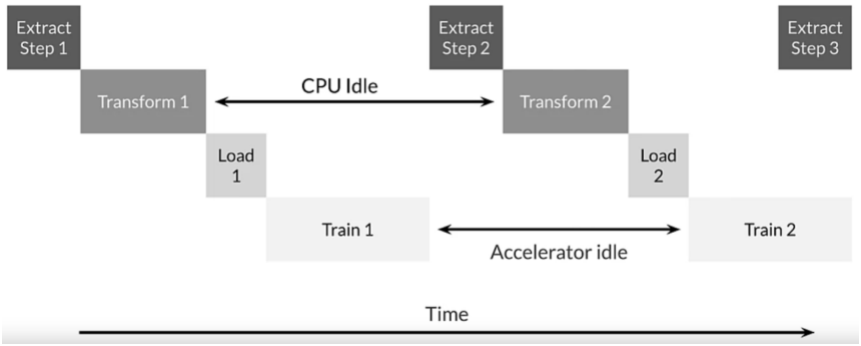
There are two ways to perform data parallelism:

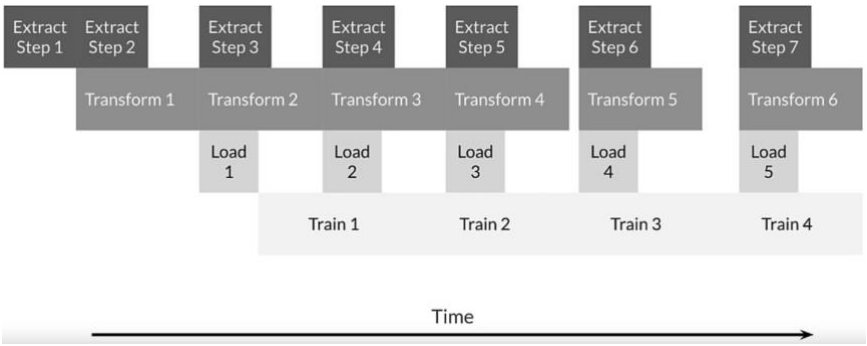


High Performance Ingestion

Accelerators are expensive so it is important to make sure that data ingestion is optimized. If data does not fit into memory, then CPUs are under-utilized -> use input pipelines, like tf.data:

ETL (Extract Transform Load) is a good mental model for data performance: each task is done by a different hardware component, but can be upgraded by parallelizing the jobs:

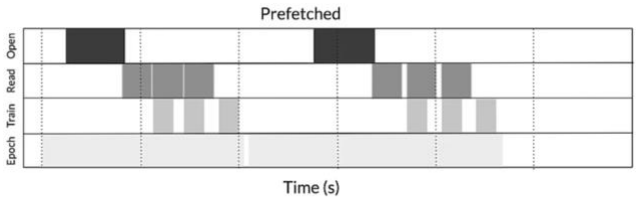




The accelerator (training) is used 100% of the time, which is good because it is very expensive.

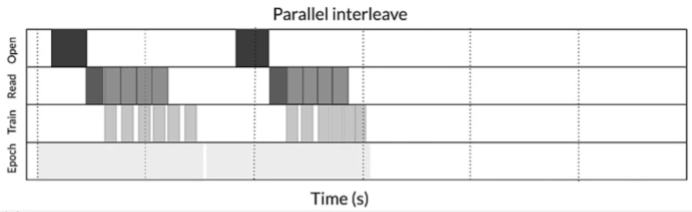
How to optimize the pipeline:

- Prefetching.




```
benchmark(
    ArtificialDataset()
    .prefetch(tf.data.experimental.AUTOTUNE)
)
```

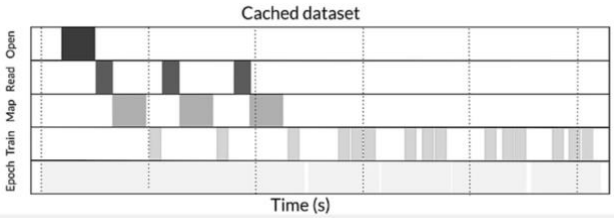
- Parallelize data extraction and transformation: time-to-first-byte (prefer local storage because faster) and read throughput (maximize aggregate bandwidth of remote storage by reading more files) are two common techniques available in tf.data.




```
benchmark(
    tf.data.Dataset.range(2)
    .interleave(
        ArtificialDataset,
        num_parallel_calls=tf.data.experimental.AUTOTUNE
    )
)
```

- Caching.

- In-memory: `tf.data.Dataset.cache()`
- Disk: `tf.data.Dataset.cache(filename=...)`




```
benchmark(
    ArtificialDataset().map(mapped_function).cache(),5
)
```

- Reduce memory.

Large Models: Giant Neural Nets

Models no can have billions of parameters. Therefore, the Cloud TPUs and GPUs cannot handle training then using a one batch approach. There are several strategies used today to train these enormous models.

Gradient accumulation

This strategy splits data into mini-batches and only makes backpropagation after the whole batch.

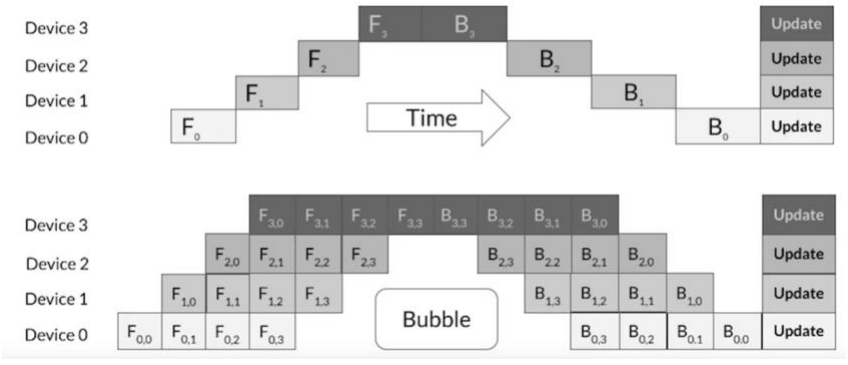
Swapping

Copy activations between CPU and memory, back and forth.

How about parallelization?

- Accelerators have limited memory
- Model parallelism: large networks can be trained
  - But, accelerator compute capacity is underutilized
- Data parallelism: train same model with different input data
  - But, the maximum model size an accelerator can support is limited

It is very important to think about how to parallelize model parallelization to optimize accelerator usage (F: forward propagation and B: backward propagation):



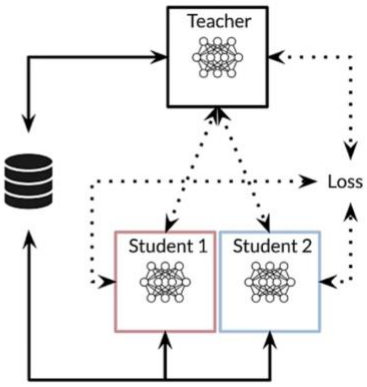
GPipe to do this:

- Open-source TensorFlow library (using Lingvo)
- Inserts communication primitives at the partition boundaries
- Automatic parallelism to reduce memory consumption
- Gradient accumulation across micro-batches, so that model quality is preserved
- Partitioning is heuristic-based

Knowledge Distillation

How to deploy a complex and large model? Should we reduce and simplify the models? Can we express the learning more efficiently to reduce the size of more complex models? Is it possible t "distill" or concentrate the complexity into smaller networks?

- Duplicate the performance of a complex model in a simpler model
- Idea: Create a simple *'student'* model that learns from a complex *'teacher'* model



Knowledge distillation techniques

The training objective functions of the teacher and the student models are different.

- Training objectives of the models vary
- Teacher (normal training)
  - maximizes the actual metric
- Student (knowledge transfer)
  - matches p-distribution of the teacher's predictions to form 'soft targets'
  - 'Soft targets' tell us about the knowledge learned by the teacher

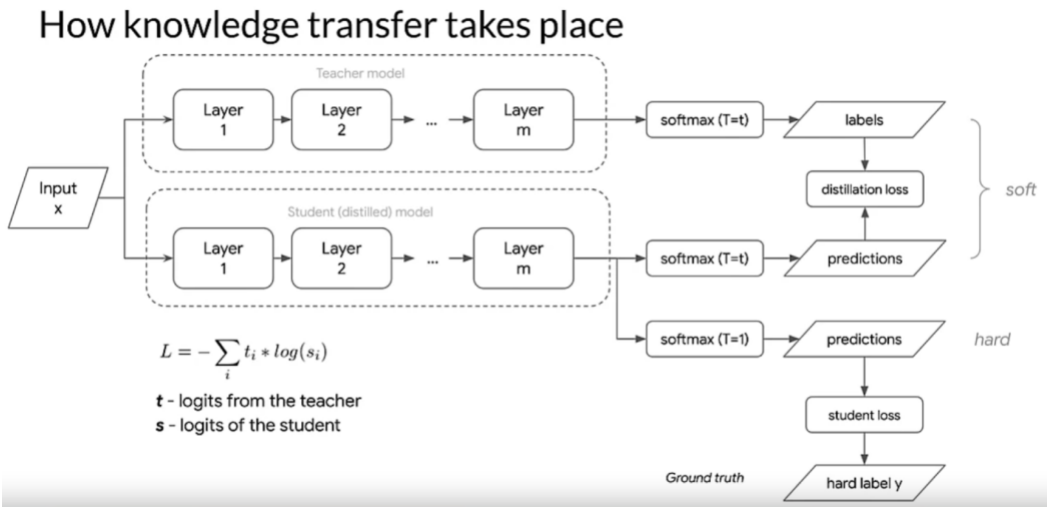
The Dark knowledge transferred:

- Improve softness of the teacher's distribution with 'softmax temperature' (T)
- As T grows, you get more insight about which classes the teacher finds similar to the predicted one

$$p_i = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_j \exp\left(\frac{z_j}{T}\right)}$$

Two techniques are presented here to apply knowledge distillation:

- Approach #1: **Weigh objectives** (student and teacher) and combine during backprop
- Approach #2: **Compare distributions of the predictions** (student and teacher) using KL divergence



Nowadays companies tend to use models that use less resources instead of models with more parameters. DistilBERT has been created for example to make BERT 40% lighter while keeping 97% of its performances using knowledge distillation.

Experiment tracking

What to track?	Tracking tools
Algorithm/code versioning	Text files
Dataset used	Spreadsheet
Hyperparameters	Experiment tracking system
Results	

Desirable features
Information needed to replicate results
Experiment results, ideally with summary metrics/analysis
Perhaps also: Resource monitoring, visualization, model error analysis

Big Data vs Good Data

It is important to focus on high quality data rather than a massive amount of data.

Good data:
- Covers important cases (good coverage of inputs x)
- Is defined consistently (definition of labels y is unambiguous)
- Has timely feedback from production data (distribution covers data drift and concept drift)
- Is sized appropriately

It is possible to get more good data using data augmentation and good data.

Model Performance Analysis

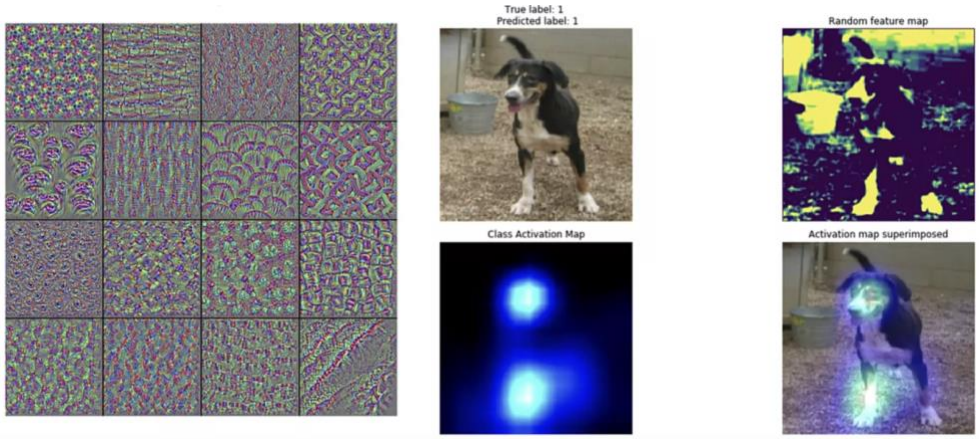
In production, in is very important to have a deep analysis of the performances of a model:

- Does it perform well?
- Is there a scope for improvement?
- Can the data change in the future?
- Has data changed since the training set was created?

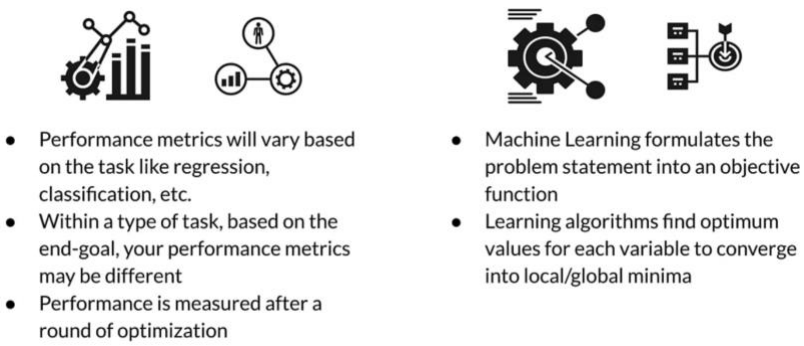
There are two ways to analyze the performance of a model: Black box evaluation and model introspection. The first one focuses on the results output by the model (accuracy, f1-score …) whereas the second one focuses on understanding how data flows internally.

TensorBoard is a Black box evaluation tool to monitor the performances of a model over the time through an editable dashboard.

Model Introspection can help understand for example what parts of a picture give more information to the model, looking at the activations results in a CNN:



Performance metrics vs optimization:

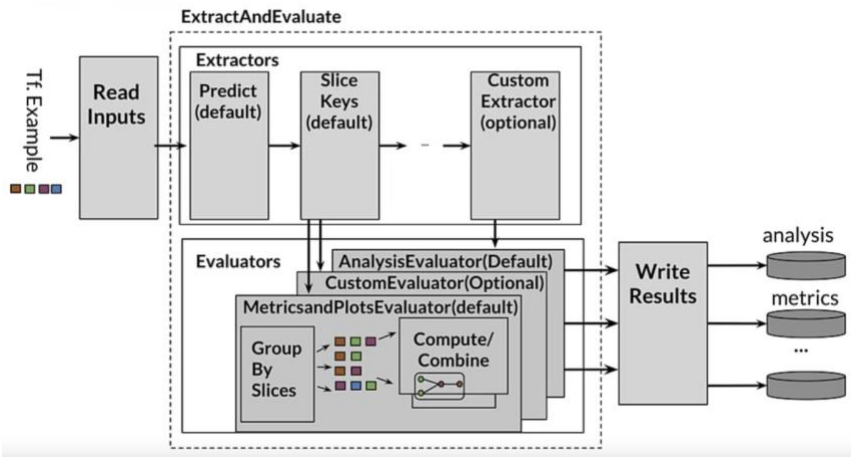


Advanced Model Analysis and debugging

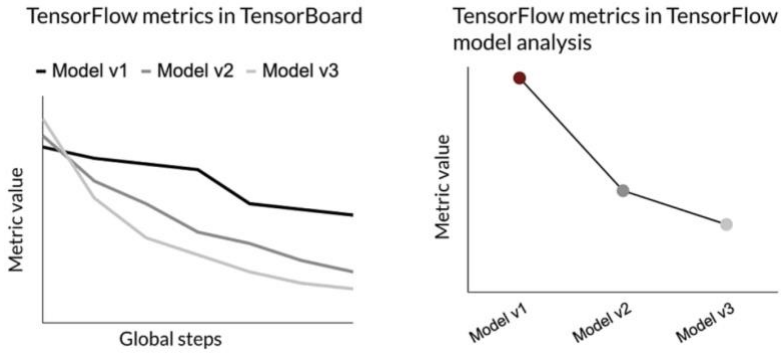
TensorFlow Model Analysis (TFMA) is a tool that can be used to evaluate a model and find issues with it to correct them before going to production.

It is scalable, open source, ensures that the models meet required quality threshold, used to compute, and visualize evaluation metrics and inspect model's performance against different slices of data.

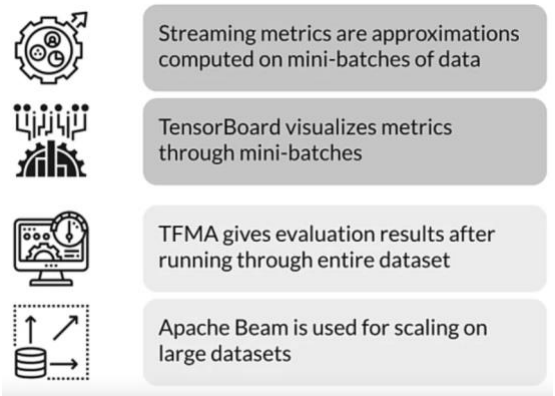
Architecture of TFMA:



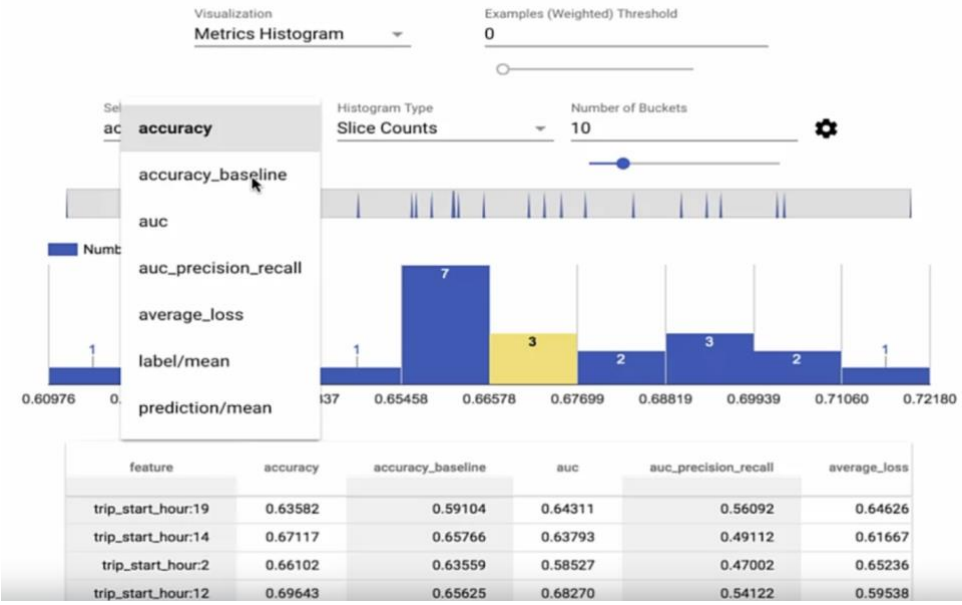
TFMA compares multiple versions of a model (after training) whereas TensorBoard compares the results of time over global training set:



If the model works well on the whole testing set but poorly on some slices, it will not be detected, that is why TFMA uses slices of the sets to detect the weaknesses of a model.



TFMA in practice can display a dashboard that will look like this:

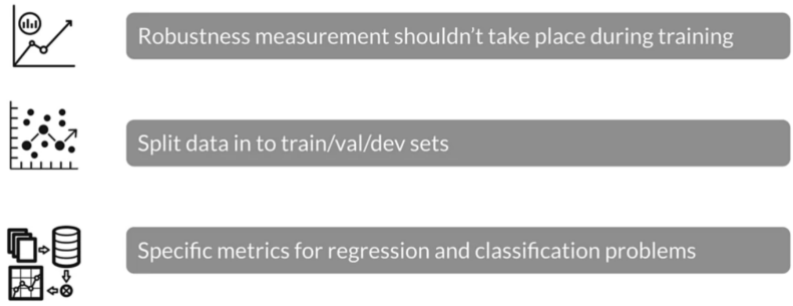


Model debugging

Model robustness

It is more than generalization; a model is robust if it is accurate even for slightly corrupted data.

How to measure the robustness of a model:



Why model debugging?

It deals with detecting and fixing problems in ML systems and applies mainstream software engineering practices to ML models.

Objectives:

Some techniques are to benchmark models over datasets, sensitivity analysis and the residual analysis.

Benchmarking models

This is the starting point of ML development since the goal is to compare the developed model to other trusted simple models.

Sensitivity Analysis and Adversarial Attacks

SA simulate chosen data and see what the model predicts, to know how model reacts to data which has never been used before.

TF has a tool called the What-If Tool that allows to do SA.

Random Attacks expose models to high volumes of random input data, which exploits the unexpected software and math bugs: it is a good starting point for debugging.

Partial dependence plot is another method. It shows the marginal effect of one or more features on the result. It allows to see what kind of relation exists between a feature and the target (linear, polynomial …). PDPbox and PyCEbox are two open-source packages that allows to perform PDP.

Sensitivity can mean vulnerability: attacks are aimed at fooling the model, successful attacks could be catastrophic. Testing adversarial examples is a good way to harden the model.

Basically, if altering the data a bit fools the model, a self-driven car could crash because it would detect a stop sign for example.

Two types of harm can be done:

- Information Harm (Leakage of information):

- Membership Inference: was this person's data used for training?
- Model Inversion: recreate the training data
- Model Extraction: recreate the model

- Behavioral Harm (Manipulating the behavior of the model):

- Poisoning: insert malicious data into training data
- Evasion: input data that causes the model to intentionally misclassify that data

To measure the vulnerability of models, several solutions exist:

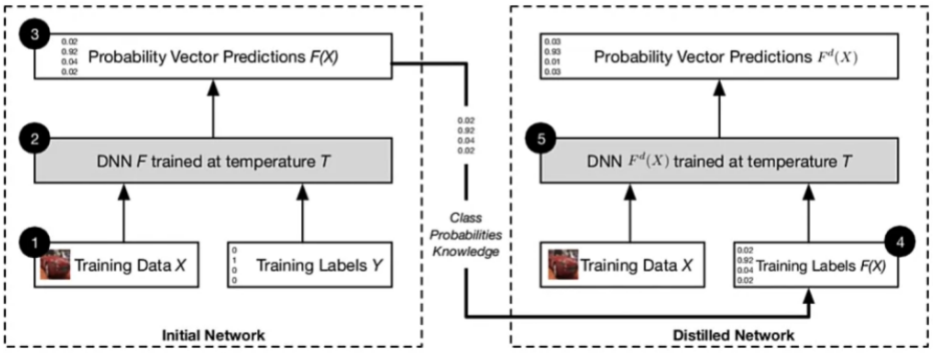
Cleverhans:
an open-source Python library to benchmark machine learning systems' vulnerability to adversarial examples

Foolbox:
an open-source Python library that lets you easily run adversarial attacks against machine learning models

Adversarial example searches:

Attempted defenses against adversarial examples

- Defensive distillation



Residual analysis

This method is mostly used for regression models since it compares the difference between the predicted value and the ground truth value. It is necessary to have labeled data to use this technique.

A random distribution of the errors is good, because if there is a correlation or systematic errors it is a sign that the model can be improved.

Instead of just calculating the accuracy, precision or recall, the residual analysis focuses on the distribution of the errors.

How to understand the results:

- Residuals should not be correlated with another feature
- Adjacent residuals should not be correlated with each other (autocorrelation)

Model Remediation

Simple techniques to improve the robustness of a model:

Data augmentation	Adding synthetic data into training set
	Helps correct for unbalanced training data
Interpretable and explainable ML	Overcome myth of neural networks as black box
	Understand how data is getting transformed

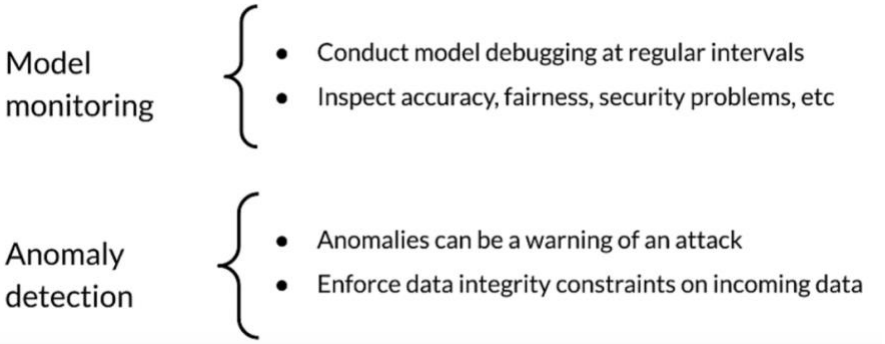
However, it is possible to improve a trained model using remediation techniques:

- Model editing:
  - Applies to decision trees
  - Manual tweaks to adapt your use case

- Model assertions:
  - Implement business rules that override model predictions

Some techniques for discrimination problems:

How to keep track after remediation:



Fairness

The goal is to ensure that the model is not causing harm to the people who use it, by making sure that no part of the population is discriminated.

The TF team developed a library called Fairness indicators. It easily scales across datasets of any size. It computes fairness metrics. What it does:

- Compute commonly-identified fairness metrics for classification models
- Compare model performance across subgroups to other models
- No remediation tools provided

To make sure that the model is fair, speaking with a domain expert can help. A good simple advice is to use data slicing wisely.

General guidelines

- Compute performance metrics at all slices of data
- Evaluate your metrics across multiple thresholds
- If decision margin is small, report in more detail

How to measure Fairness?

True Positive Rate: percentage of positive data points that are correctly labeled positive.

False Negative Rate: percentage of positive data points that are incorrectly labeled negative.

These two metrics measures the equality of opportunity, when the positive class should be equal across subgroups.

Accuracy: percentage of data points that are correctly labeled.

Area Under the Curve: percentage of data points that are correctly labeled when each class is given weight independently of the number of samples.

These two metrics are related to predictive parity and can be used when precision is critical for example.

Some tips to measure fairness:

Unfair skews if there is a gap in a metric between two groups

Good fairness indicators doesn't always mean the model is fair

Continuous evaluation throughout development and deployment

Conduct adversarial testing for rare, malicious examples

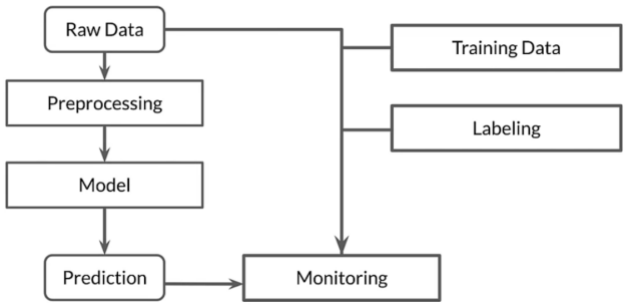
Continuous Evaluation and Monitoring

Monitoring the models is very important, because data can vary in time. ML models do not get better with age by themselves.

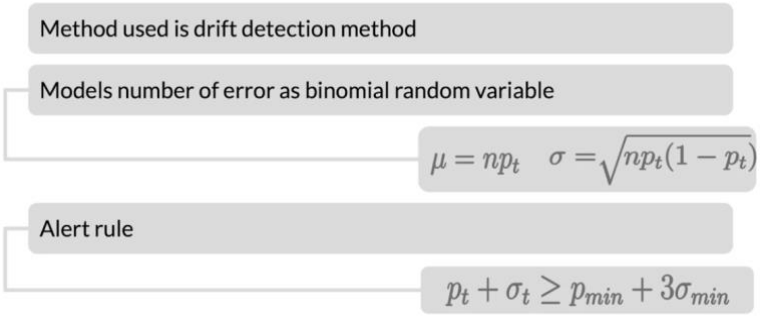
Drifts:

- Concept drift: loss of prediction quality
- Concept Emergence: new type of data distribution
- Types of dataset shift:
  - covariate shift
  - prior probability shift

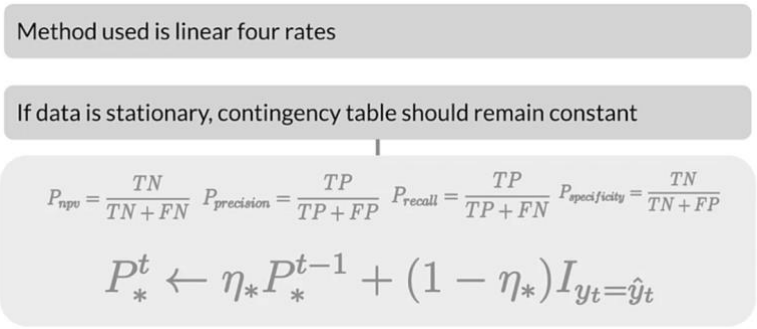
How models are monitored:



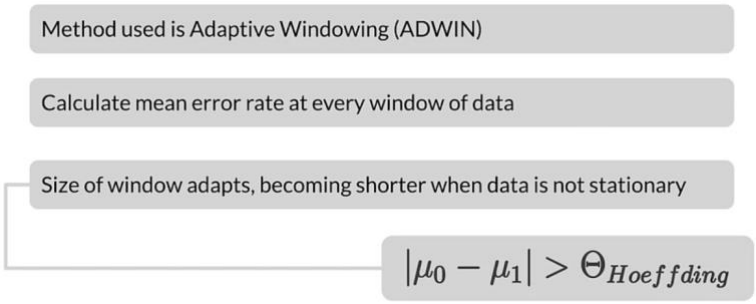
The first supervised technique to automate the monitoring is statistical process control:



The second supervised technique is sequential analysis:



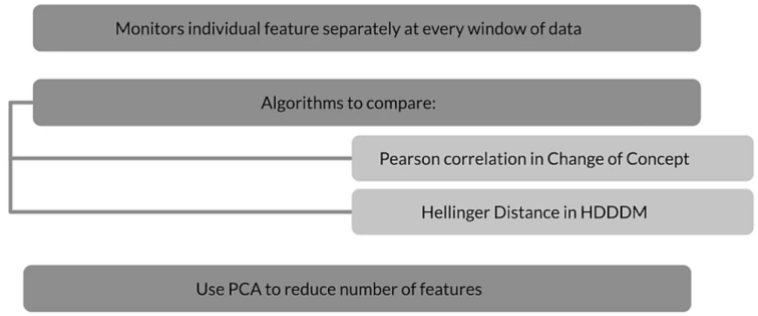
The last supervised technique is error distribution monitoring:



For unsupervised cases, the first technique is clustering/novelty detection:

- Assign data to known cluster or detect emerging concept
- Multiple algorithms available: OLINDDA, MINAS, ECSMiner, and GC3
- Susceptible to curse of dimensionality
- Does not detect population level changes

The second unsupervised technique is feature distribution monitoring:



The last unsupervised technique is model-dependent monitoring:

- Concentrate efforts near decision margin in latent space
- One algorithm is Margin Density Drift Detection (MD3)
- Area in latent space where classifiers have low confidence matter more
- Reduces false alarm rate effectively

Google Cloud AI Continuous Evaluation is a solution implemented by Google to monitor models. It leverages AI Platform Prediction and Data Labelling services. We start by deploying a model to AI Platform Prediction with model version, then create an evaluation job. Input and output are saved in a BigQuery table. It is possible to run the evaluation job on few of these samples.

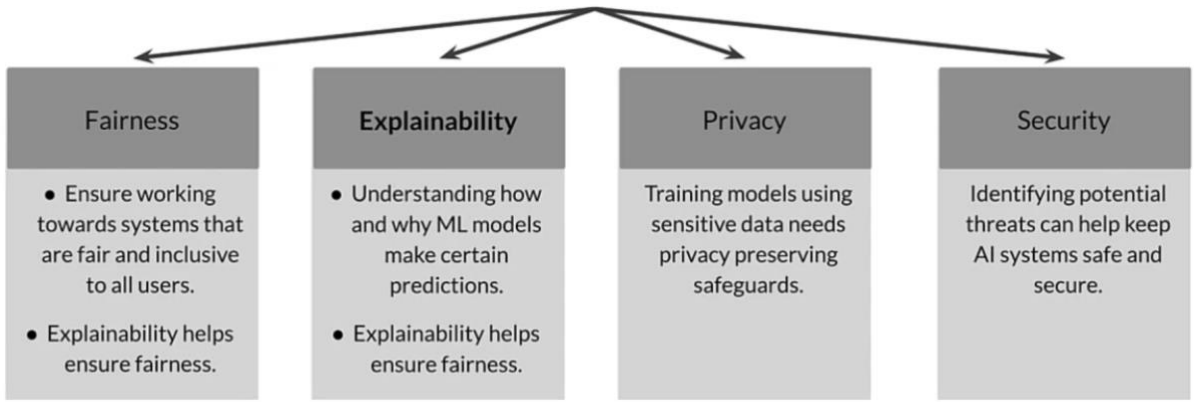
How often should a model be retrained?

It depends on the rate of change. If possible, automating the management of detecting model drift and triggering model retraining can be good to limit the training cost.

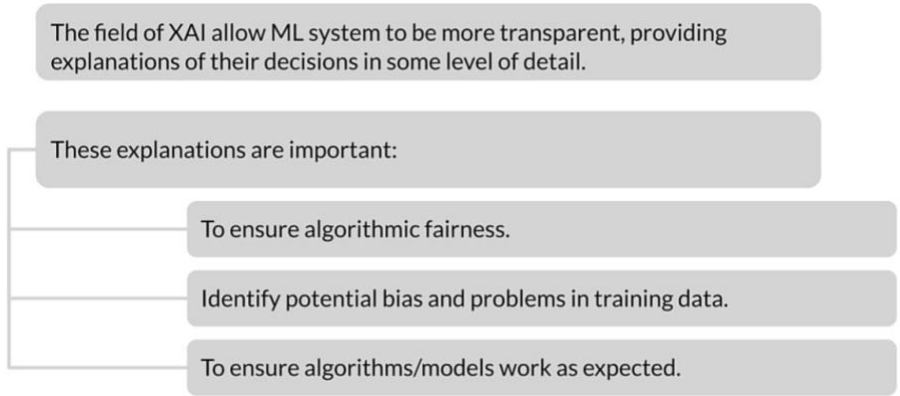
Explainable AI

To answer to the increase of need of model interpretability and explainability (both are part of Responsible AI), XAI was created to try to explain what the model does to justify the outputs.

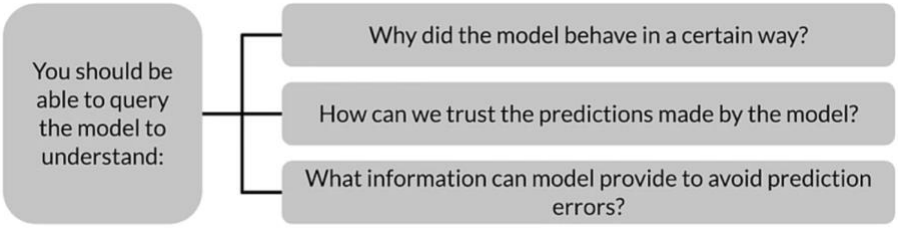
Responsible AI was born because development of AI raises new questions about the best way to build the following into AI Systems:



Some models are interpretable by nature, like decision trees, but XAI was created to do the same with more complex models:

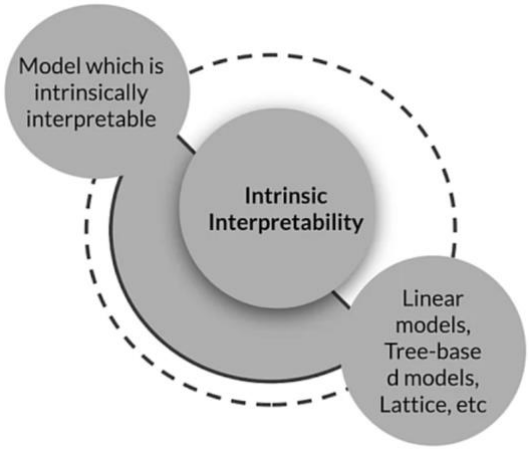


The explainability in AI is needed because models with high sensitivity, including natural language networks, can generate wildly wrong results, respect legal and regulatory concerns, answer to customers and other stakeholders' questions or challenge model decisions, but also to ensure a security against attacks, respect of fairness, but also because it is better for the reputation and branding of companies. What is needed:



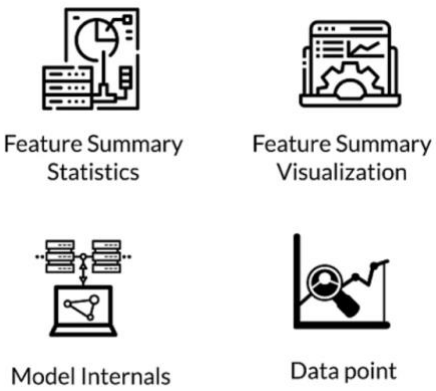
Interpretation methods

Intrinsic or Post-Hoc:

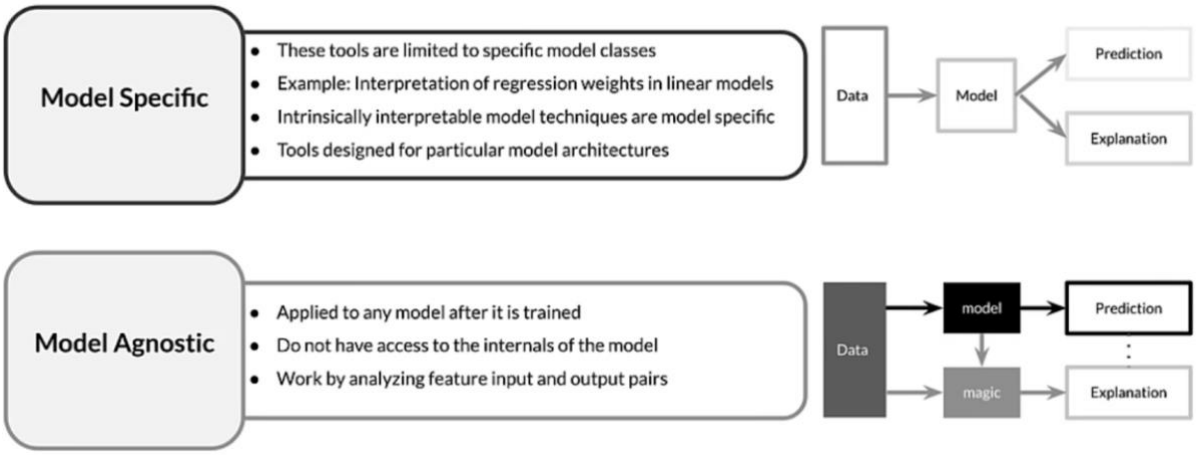


Post-Hoc methods treat models as black boxes. It is agnostic to model architecture. It extracts relationships between features and model predictions and is applied after training.

Types of results produced by interpretation methods:

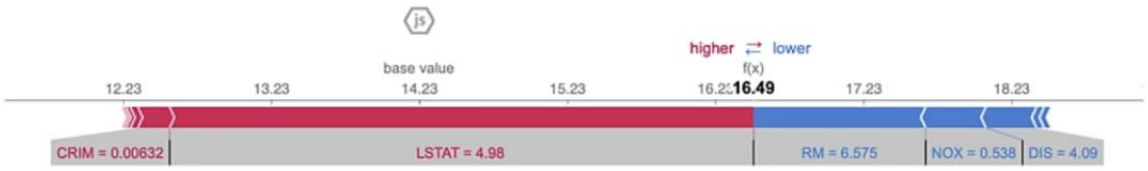


Model specific vs model agnostic methods:

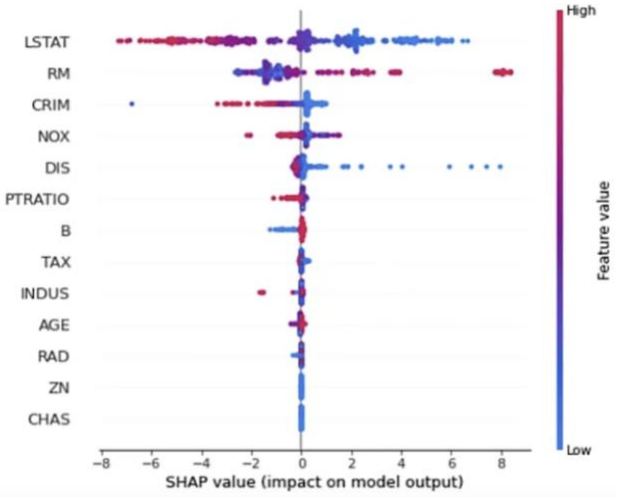


Local or Global:

- Local: interpretation method explains an individual prediction.
- Feature attribution is identification of relevant features as an explanation for a model.



- Global: interpretation method explains entire model behaviour
- Feature attribution summary for the entire test data set

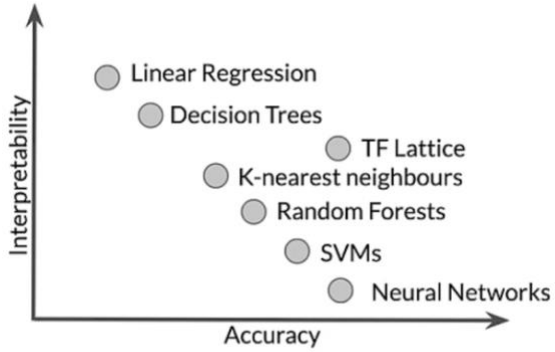


Intrinsically Interpretable Models

Monotonicity improves interpretability (consistency): if results are increasing or decreasing over the whole values of a feature, then it is easy to get a correlation between the target and this feature.

Algorithm	Linear	Monotonic	Feature Interaction	Task
Linear regression	Yes	Yes	No	regr
Logistic regression	No	Yes	No	class
Decision trees	No	Some	Yes	class, regr
RuleFit	Yes*	No	Yes	class, regr
K-nearest neighbors	No	No	No	class, regr
TF Lattice	Yes*	Yes	Yes	class, regr

Interpretability vs Accuracy trade off:



Interpretability vs Accuracy Trade off

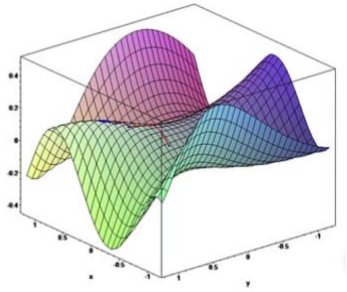
Interpretation from Weights in linear models is easy to understand:

- Numerical features: increase of one unit in a feature increases prediction by the value of corresponding weight.
- Binary features: changing between 0 or 1 category changes the prediction by value of the feature's weight.
- Categorical features: one hot encoding affects only one weight.

Feature Importance is the relevance of a given feature to generate model results. Its calculation is model dependent.

TensorFlow Lattice is an advanced interpretable model:

- Overlaps a grid onto the feature space and learns values for the output at the vertices of the grid
- Linearly interpolates from the lattice values surrounding a point



- Enables you to **inject domain knowledge** into the learning process through common-sense or policy-driven shape constraints
- Set constraints such as monotonicity, convexity, and how features interact

It achieves accuracies comparable to neural networks, in addition to a greater interpretability. However, this model has weaknesses linked to dimensionality:

Dimensionality

- The number of parameters of a lattice layer **increases exponentially** with the number of input features
- Very Rough Rule: Less than 20 features ok without ensembling

Model Agnostic Methods

There are a lot of methods:

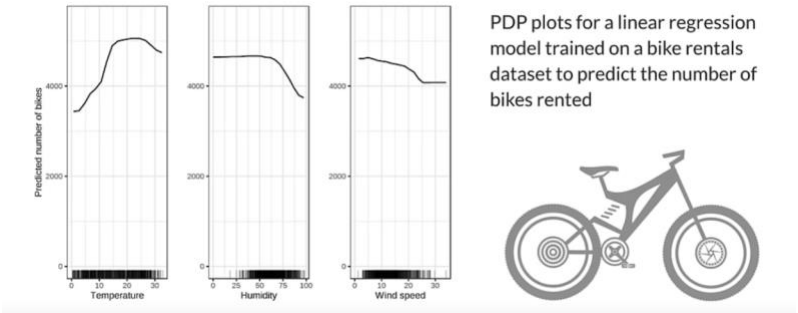
Partial Dependence Plots	Individual Conditional Expectation
Accumulated Local Effects	Permutation Feature Importance
Permutation Feature Importance	Global Surrogate
Local Surrogate (LIME)	Shapley Values
SHAP	

Partial Dependence Plots

A partial dependence plot shows the marginal effect one or two features have on the model results. It shows also whether the relationship between the targets and the feature is linear, monotonic, or more complex. The partial function is estimating by calculating averages in the training data:

$$\hat{f}_{x_S}(x_S) = \frac{1}{n}\sum_{i=1}^{n}\hat{f}(x_S, x_C^{(i)})$$

The results are plots, which is very intuitive to understand:



PDP plots for a linear regression model trained on a bike rentals dataset to predict the number of bikes rented

In addition, it is easy to implement. Realistically however there are some disadvantages:

- Realistic maximum number of features in PDP is 2
- PDP assumes that feature values have no interactions

Permutation Feature Importance

This method measures the increase in prediction error after permuting the features.

A feature is important if shuffling its values increases model error, whereas it is unimportant if shuffling its values leaves model error unchanged. An unimportant feature must be removed from the features vector.

Basic algorithm:

- Estimate the original model error
- For each feature:
  - Permute the feature values in the data to break its association with the true outcome
  - Estimate error based on the predictions of the permuted data
  - Calculate permutation feature importance
  - Sort features by descending feature importance .

This method has a nice interpretation, because it shows the increase in model error when the feature's information is destroyed. It also provides insight to model's behavior and does not require retraining the model. However, it is unclear f testing or training data should be used for visualization, it can be biased since it can create unlikely feature combination in case of strongly correlated features, and we need to access to labeled data.

Shapley Values

It is a concept from cooperative game theory.

- The Shapley value is a method for assigning payouts to players depending on their contribution to the total
- Applying that to ML we define that:
  - Feature is a "player" in a game
  - Prediction is the "payout"
  - Shapley value tells us how the "payout" (feature contribution) can be distributed among features

It can be used regardless to the model architecture.

Advantages:

Based on solid theoretical foundation.
Satisfies Efficiency, Symmetry, Dummy, and Additivity properties

Value is fairly distributed among all features

Enables contrastive explanations

Disadvantages:

- Computationally expensive
- Can be easily misinterpreted
- Always uses all the features, so not good for explanations of only a few features.
- No prediction model. Can't be used for "what if" hypothesis testing.
- Does not work well when features are correlated

Shapley Additive exPlanations (SHAP)

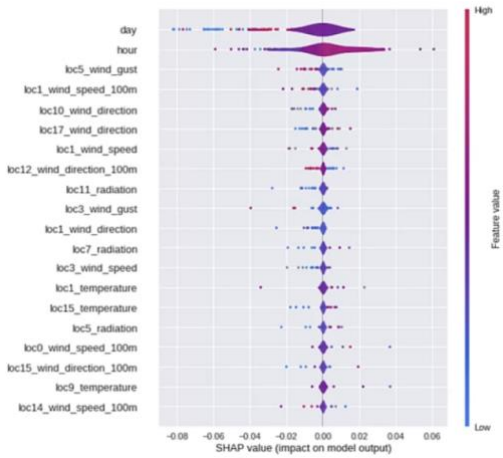
It is a framework for Shapley Values which assigns each feature an importance value for a particular prediction.

It includes extensions for:

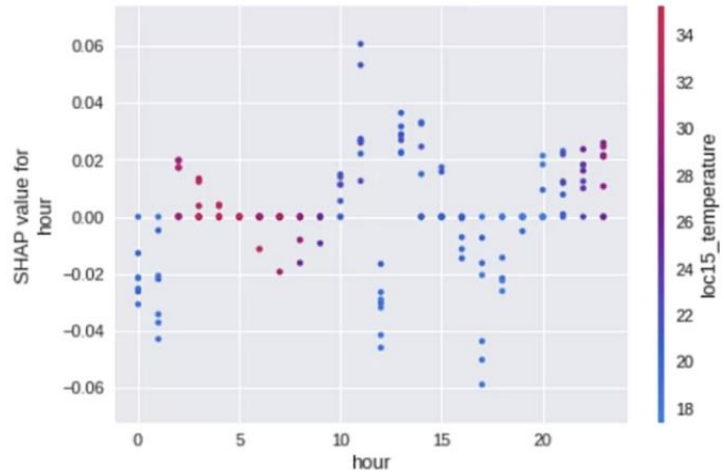
- TreeExplainer: high-speed exact algorithm for tree ensembles.
- DeepExplainer: high-speed approximation algorithm for SHAP values in deep learning models.
- GradientExplainer: combines ideas from Integrated Gradients, SHAP, and SmoothGrad into a single expected value equation.
- KernelExplainer: uses a specially-weighted local linear regression to estimate SHAP values for any model.

- Shapley Values can be visualized as forces
- Prediction starts from the baseline (Average of all predictions)
- Each feature value is a force that increases (red) or decreases (blue) the prediction

Or as summary plots:



SHAP Dependence Plot with Interaction:



Testing Concept Activation Vectors

Concept Activation Vectors: a neural network's internal state in terms of human-friendly concepts, defined using examples which show the concept.

Local Interpretable Model-agnostic Explanations (LIME)

It is a well-known framework for producing local explanations. It creates local interpretations of model results.

- Implements local surrogate models - interpretable models that are used to explain individual predictions
- Using data points close to the individual prediction, LIME trains an interpretable model to approximate the predictions of the real model
- The new interpretable model is then used to interpret the real result

AI Explanations

Google Cloud AI Explanations for AI Platform. Goal:

Explain why an individual data point received that prediction

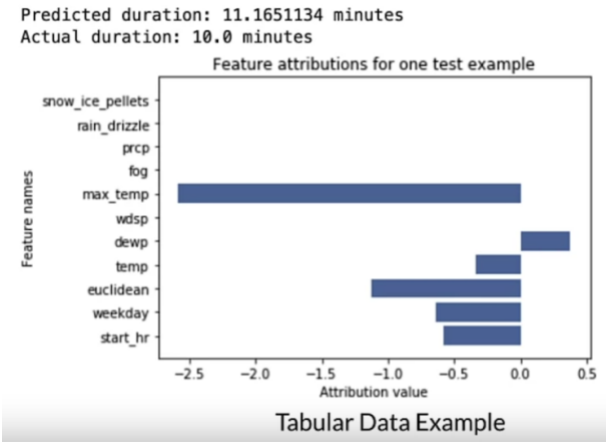
Debug odd behavior from a model

Refine a model or data collection process

Verify that the model's behavior is acceptable

Present the gist of the model

Features Attributions:



Tabular Data Example

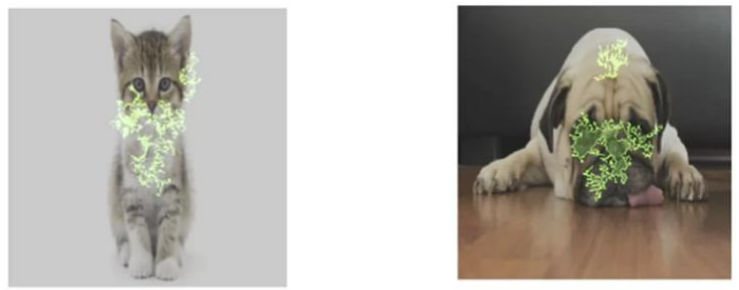


Image Data Examples

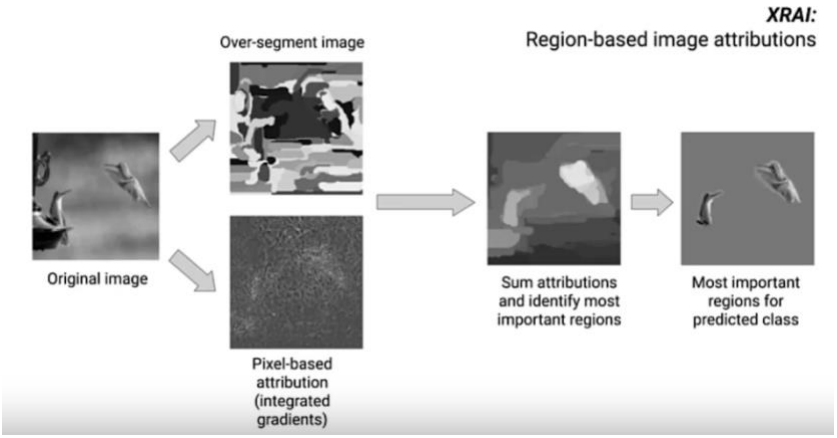
To extract this information, the techniques are based on Shapley Values: Sampled Shapley, Integrated Gradients, and eXplanation with Ranked Area Integrals (XRAI).

Integrated Gradients

It is a gradient-based method to efficiently compute feature attributions with the same axiomatic properties as Shapley values.

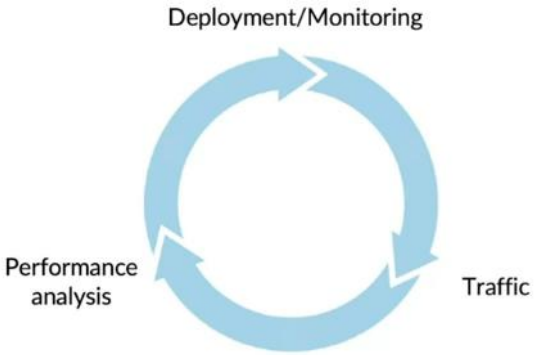
eXplanation with Ranked Area Integrals

XRAI assesses overlapping regions of the image to create a saliency map. It highlights relevant regions of the image rather than pixels; it aggregates the pixel-level attribution within each segment and ranks the segments.



Deployment

Deployment cycle



Types

- Edge deployment: local deployment on the machine that uses the service.
- Cloud deployment: deployment on a cloud machine that answers to requests sent by clients.

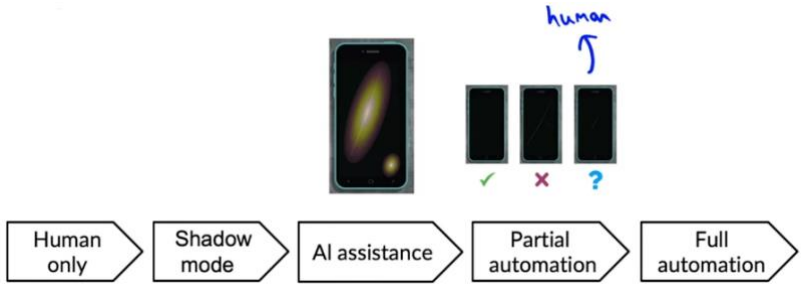
Data issues

- Concept drift: definition of X -> Y changes.
- Data drift: the data distribution X itself changes.

Software engineering issues

- Realtime or Batch
- Cloud vs. Edge/Browser
- Compute resources (CPU/GPU/memory)
- Latency, throughput (QPS)
- Logging
- Security and privacy

Degrees of automation



Examples of metrics to track

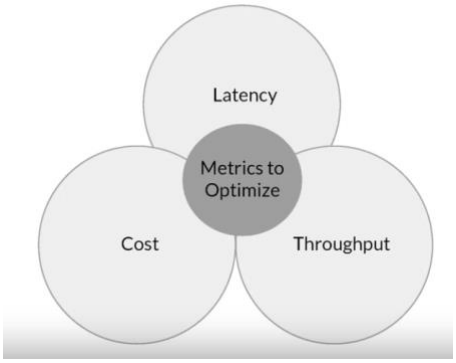
- Software metrics: memory, compute, latency, throughput, server load …
- Input metrics (metrics about X): statistics about the entry data distribution and quality …
- Output metrics (metrics about Y): statistics about output results, number of times no results are predicted, number of times the user tried back the same request …

Introduction to Model Serving

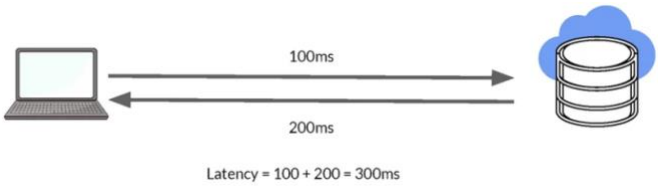
Training a model is just the first part, we need to make the model available to end users and provide a service or an app for interaction.

There are three patterns: a model, an interpreter and input data. The inference process links these three elements.

Important metrics to optimize:



Latency:



- Delay between user's action and response of application to user's action.
- Latency of the whole process, starting from sending data to server, performing inference using model and returning response.
- Minimal latency is a key requirement to maintain customer satisfaction.

Throughput:

- ➔ Throughput -> Number of successful requests served per unit time say one second.
- ➔ In some applications only throughput is important and not latency.

Overall, most of the customers want to minimize latency while maximizing throughput.

Cost increases as infrastructure is scaled. Some good advice is to reduce costs by sharing GPUs, multi-model serving and optimizing models used for inference.

Introduction to Model Serving Infrastructure

Model Optimization is very important to have a balance between cost and complexity.

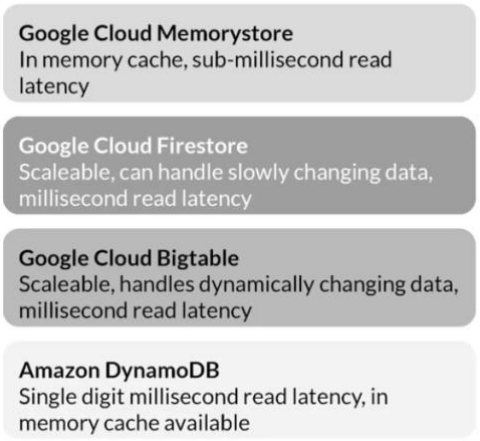
Optimization metrics:

Process of optimization:

Use of accelerators in Serving Infrastructure:

It is important to maintain input feature lookup, using cache with NoSQL database for example.

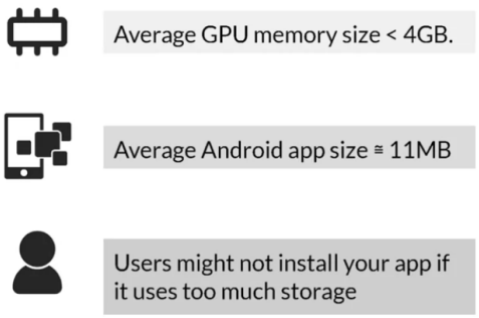
Some GCloud solutions and AWS exist to deploy cache for the model serving:



Deployment Options

Cloud vs Embedded; the choice is important.

Phone:



Rarely possible to deploy a large complex model on mobile phones.

Server: less limitations but adds latency, which can be a problem in some use cases, for example with an object detection model for self-driven cars.

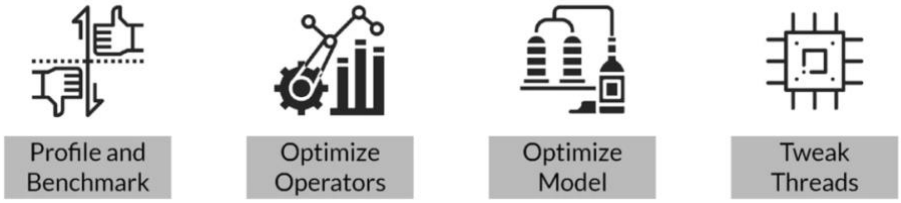
Prediction Latency is always important:

- Opt for on-device inference whenever possible
  - Enhances user experience by reducing the response time of your app

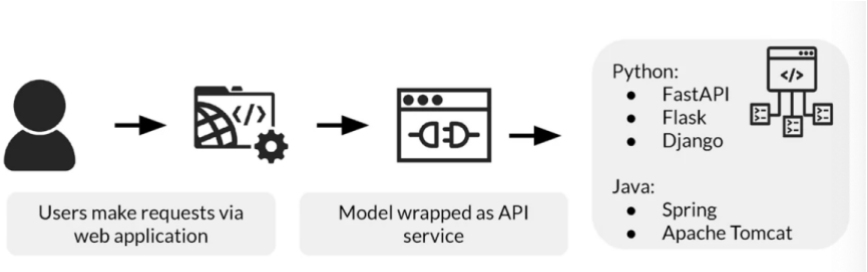
It is therefore important to choose the best model for the task and the device it will run on.

Improving Prediction Latency and Reducing Resource Costs

Some other optimization strategies:

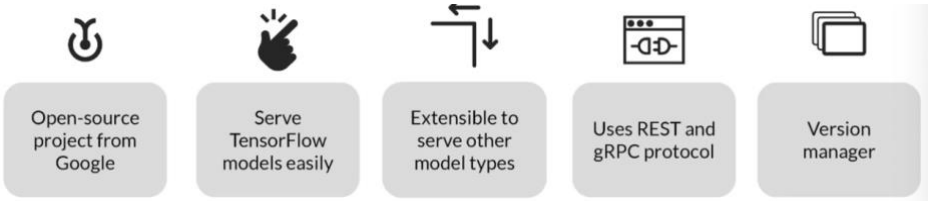


For users, it can be useful to provide a Web Application for users:



Clipper for Serving:

TensorFlow Serving:



Install TF Serving on a machine:

Available Binaries	
tensorflow-model-server	**tensorflow-model-server-universal:**
1. Fully optimized server 2. Uses some platform specific compiler optimizations 3. May not work on older machines	1. Compiled with basic optimizations 2. Doesn't include platform specific instruction sets 3. Works on most of the machines

There are also advantages of serving with a Managed Service. It provides real-time endpoint for low-latency predictions on massive batches, deployment of models trained on premises or on GCP, scales automatically based on traffic, and uses GPUs/TPUs for faster predictions.

Monitoring systems

- Dashboard: used to graphically monitor several metrics, using graphs for example.
  - It is good to set thresholds for alarms on the graphs.
  - Important to adapt metrics and thresholds over time.

Model Serving Architecture

ML Infrastructure depends on the device. If it is on prem, them it is necessary to manually procure, manage and maintain hardware GPUs, CPUs … and train and deploy our own infrastructure. It is interesting for large companies running ML projects for longer time. If it is on cloud, then we should train and deploy on cloud after choosing from several service providers (AWS, GCP, MA …).

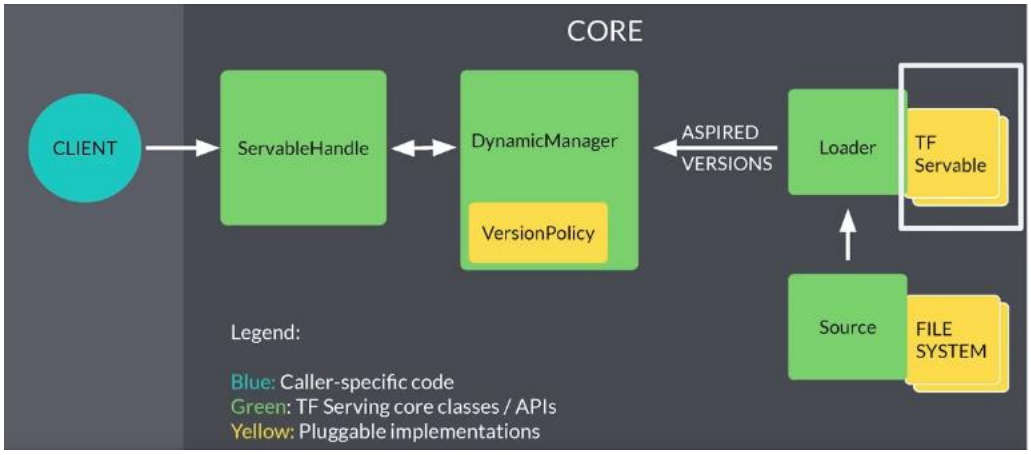
It is important to think about how model serving will work. On prem, we can use open-source pre-built servers for instance (TF-Serving, KF-Serving, Nvidia …) while on cloud we can create VMs and use open-source pre-built servers as well or use the provided ML workflow (like AutoML).

Model Server high level architecture:

- Simplify the task of deploying machine learning models at scale.
- Can handle scaling, performance, some model lifecycle management etc.,

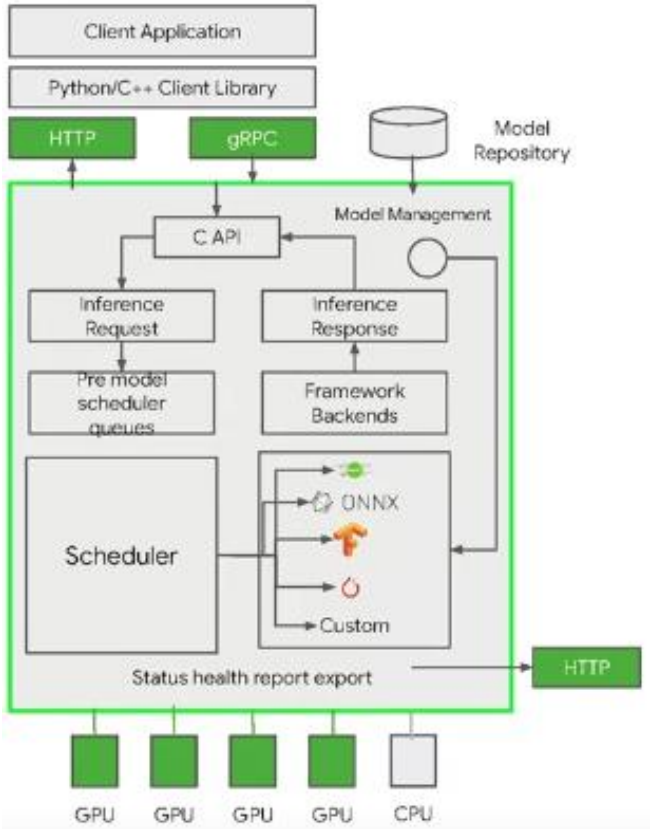
Some common Model Servers: TensorFlow Serving, PyTorch TorchServe, Kubeflow Serving and NVIDIA Triton Inference Server.

TF Serving high level architecture:



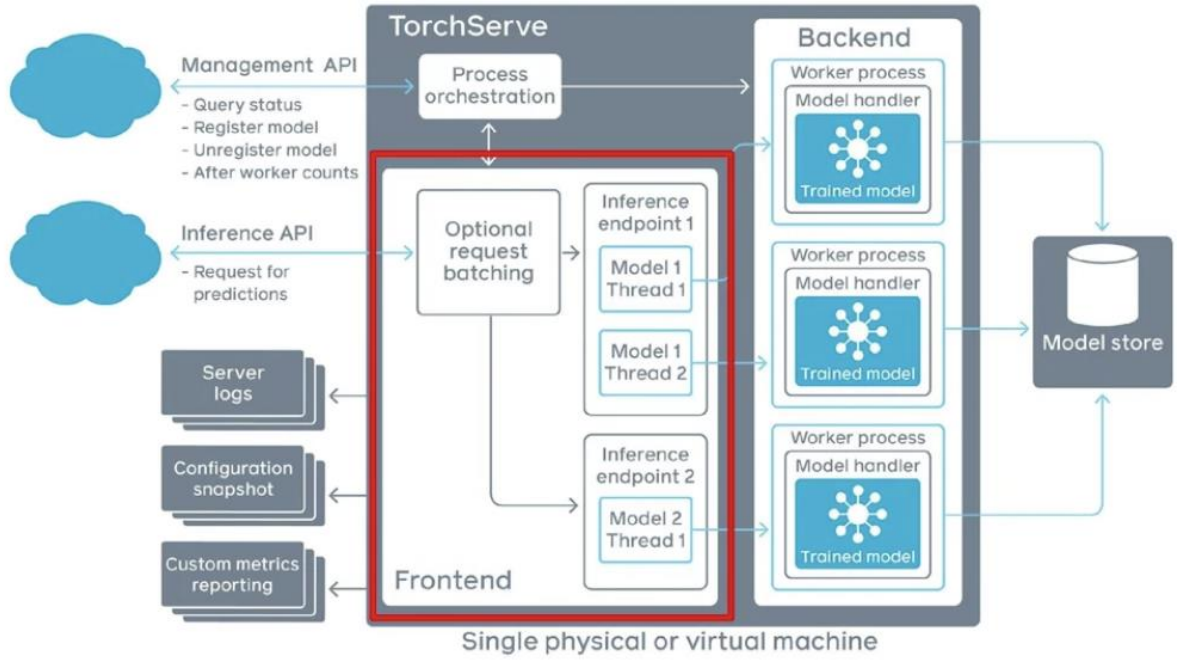
NVIDIA Triton Inference Server can run single-GPU for multiple models from same/difference frameworks or multi-GPU for same model (can run instances of model on multiple GPUs for increased inference performance. It supports model ensembles.

The inner architecture of NVIDIA Triton Inference Server at a high-level look like this:



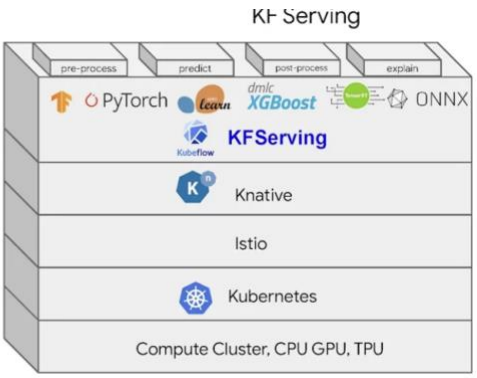
It can integrate with Kubeflow pipelines for end-to-end AI workflow.

TorchServe is a model serving framework for PyTorch models. It is an initiative from AWS and Facebook and is opensource. The server-architecture looks like this:



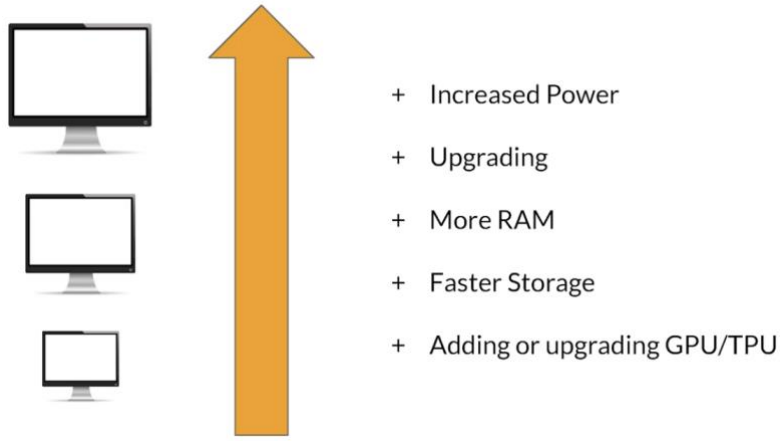
KF Serving:

- Enables serverless inferencing on Kubernetes.
- Provides high abstraction interfaces for common ML frameworks like TensorFlow, PyTorch, scikit-learn etc.

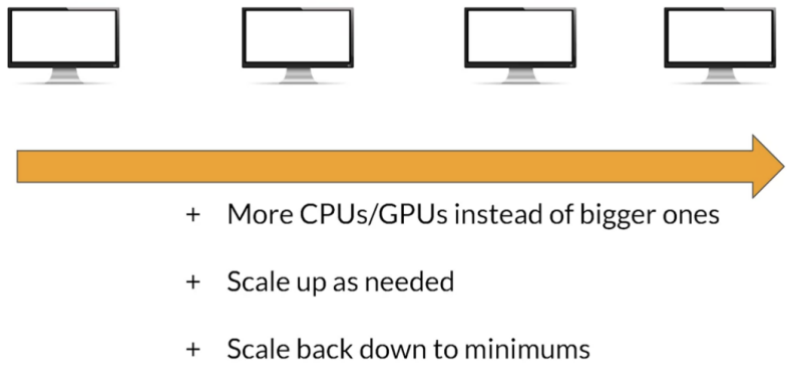


Scaling Infrastructure

Vertical scaling:



Horizontal scaling:



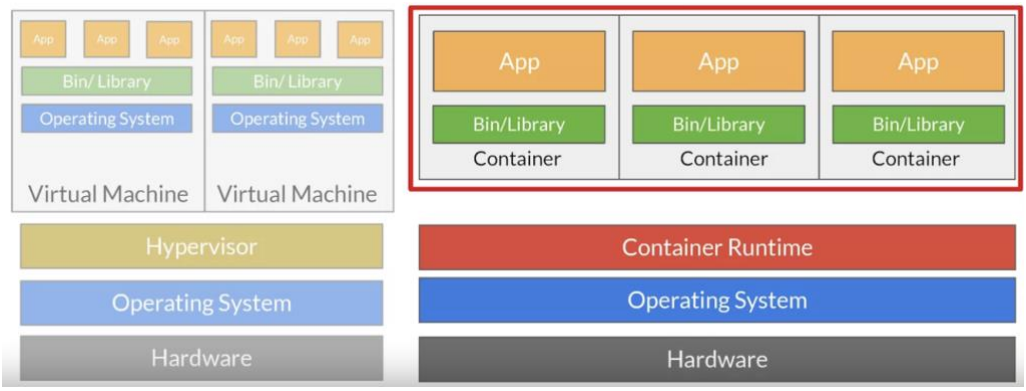
Horizontal scaling is usually better:

- Benefits of elasticity by shrinking or grow number of nodes on load throughput, latency requirements.
- Application never goes offline because there is no need for taking existing servers offline for scaling.
- No limit on hardware capacity, it simply adds more nodes any time at increased cost.

To deploy the application, containers are usually better than VMs. The same hardware can usually control more containers than virtual machines, less OS requirements and more applications are

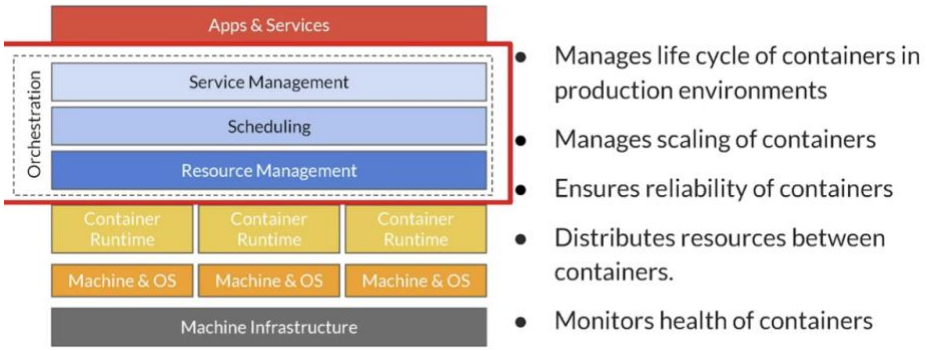
available for containers, they implement abstraction and are easy to deploy based on container runtime.

VM vs Container architecture:



Docker is a Container Runtime. It is open source and is the most popular container runtime. It started as a container technology for Linux but is now available for Windows applications as well (and Mac OS). It can be used in data centers, on personal machines or public clouds. It partners with most cloud services for containerization.

Container orchestration:



Two of the most orchestration tools are Kubernetes and Docker Swarm. Kubernetes is an open-source system for automating, deployment, scaling and management of containerized applications.

Kubeflow is an ML Workflow tool based on Kubernetes. It is dedicated to making deployments of ML workflows on Kubernetes simple, portable, and scalable, and should work anywhere Kubernetes can run. It can be run on premise or on Kubernetes engine on cloud offerings like AWS, GCP, Azure etc.

Online Inference

Process of generating ML predictions in real time upon request. The predictions are generated on a single observation of data at a time. The most important metric to optimize is latency.

This kind of use cases need a high optimization level:

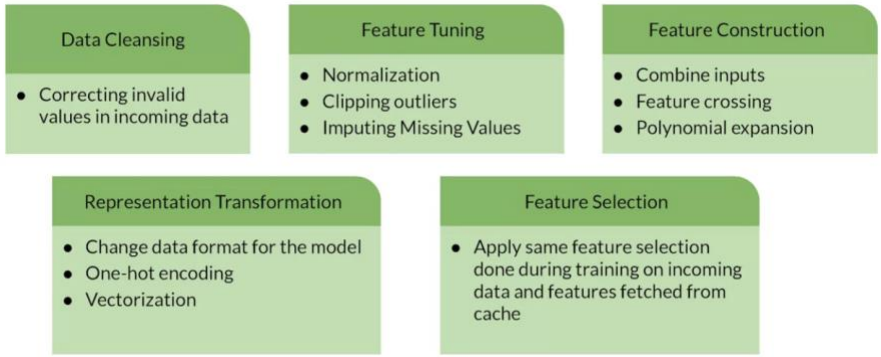
- The infrastructure needs to be scaled to optimize latency.
- The model architecture needs to find a balance between throughput and latency.
- The model compilation needs to be optimized depending on the device it runs on (GPU, TPU …) to optimize the model artefacts and a model execution runtime.

On the application, other optimizations can be done, by using a cache for the Data Store with the popular outputs for example. Fast caching is usually done using NoSQL databases, as discussed in a previous part of this document.

Data Preprocessing

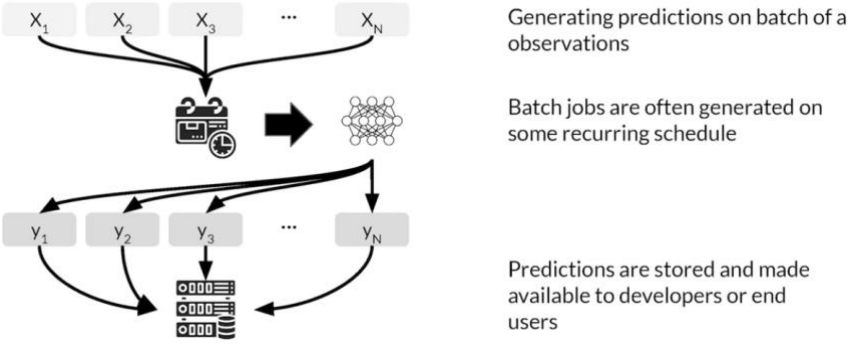
The data input of the app is a very important detail when designing the app to deploy a model.

Example of data preprocessing to implement in the app to handle most of the new data inputs from users:



Batch Inference Scenarios

Batch Inference:



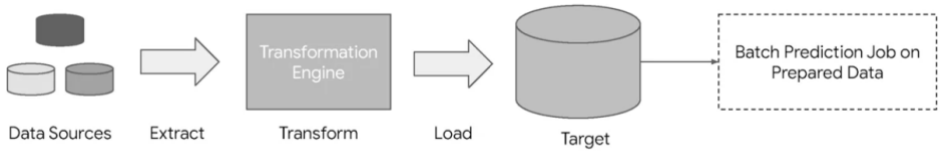
It has some advantages. It is possible to use complex models to maximize the results accuracy. Caching is not required, so it reduces the cost of the ML system, no longer needs data retrieval, and adding cache is not a problem as predictions are not in real time. However, it has along update latency, and predictions are based on older data.

The most important metric to optimize while performing batch predictions is throughput.

Data Processing with ETL

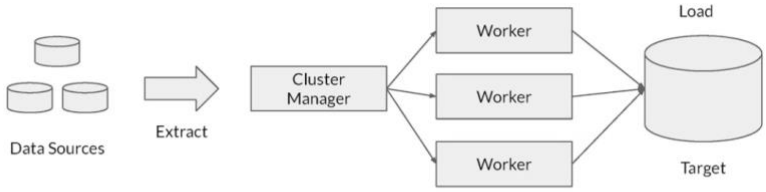
Extract Transform Load.

ETL Pipelines:



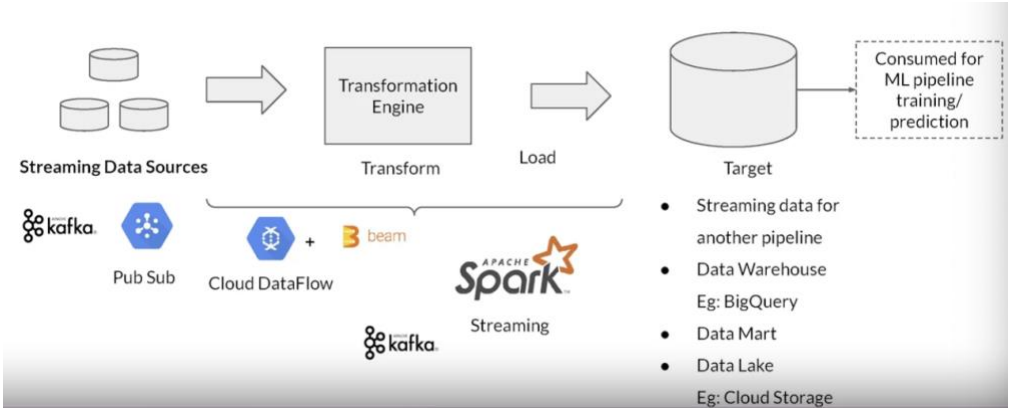
- Set of processes for
  - extracting data from data sources
  - Transforming data
  - Loading into an output destination like data warehouse
- From there data can be consumed for training or making predictions using ML models,

Distributed Processing:



- ETL can be performed huge volumes of data in distributed manner.
- Data is split into chunks and parallely processed by multiple workers.
- The results of the ETL workflow are stored in a database.
- Results in lower latency and higher throughput of data processing.

ETL Pipeline Components Stream Processing:



ML Experiments Management and Workflow Automation

Experiment Tracking

ML projects have far more branching and experimentation. ML is difficult to debug and time consuming. Small changes can lead to drastic changes in a model's performance and resource requirements. Running experiments can be time consuming and expensive.

Track experiment is important because:

- Enable to duplicate a result.
- Enable to meaningfully compare experiments.
- Manage code/data versions, hyperparameters, environment, metrics.

- Organize them in a meaningful way.
- Make them available to access and collaborate on within your organization.

Simple experiments can be done with Notebooks. However, Notebook code is usually not promoted to production. There are lots of tools for managing notebook code: nbconvert (.ipynb -> .py), nbdime (diffing), jupytext (conversion+versioning) and Neptune-notebooks (versioning+diffing+sharing).

At production level experiments, the code should be modular. Collections of interdependent and versioned files is good too. Code repositories and commits are the best way to go (Git).

Experiments are iterative in nature, so it is important to keep in mind that:

- Creative iterations for ML experimentation
- Define a baseline approach
- Develop, implement, and evaluate to get metrics
- Assess the results, and decide on next steps
- Latency, cost, fairness, etc.

Tools for Experiment Tracking

Data versioning

Data reflects the world and its changes. Experimental changes include changes in data. Tracking, understanding, comparing, and duplicating experiments includes data. Some tools are Neptune, Pachyderm, Delta Lake, Git LFS, DoIt, LakeFS, DVC, and ML-Metadata (part of TFX).

Experiment tracking to compare results

These results are necessary to log, because it can help detecting bugs depending on the model version and the data in input for example. A very good tool for that metrics tracking is TensorBoard.

Model development organization usually relies on searching through and visualize all experiments, organizing into something digestible, making data shareable and accessible, and tagging and adding notes that will be meaningful to the team.

Tooling for Teams

Vertex TensorBoard is a managed service with enterprise-grade security, privacy, and compliance. It provides a persistent, shareable link to the experiment dashboard, and a shareable list of all experiments in a project.

MLOps Methodology

Introduction to MLOps

Data Scientists and Software Engineers are the two key roles within an engineering team.

Data Scientist:

- Often work on fixed datasets
- Focused on model metrics
- Prototyping on Jupyter notebooks
- Expert in modeling techniques and feature engineering
- Model size, cost, latency, and fairness are often ignored

Software engineer:

- Build a product
- Concerned about cost, performance, stability, schedule
- Identify quality through customer satisfaction
- Must scale solution, handle large amounts of data
- Detect and handle error conditions, preferably automatically
- Consider requirements for security, safety, fairness
- Maintain, evolve, and extend the product over long periods

Both jobs are growing to solve complex real-world problems, so ML faces a growing need in products and services. Large datasets are available, it is relatively inexpensive to access on-demand compute resources and accelerators for ML are increasingly powerful. Rapid advances in many ML research fields made it more accessible for businesses, which are investing in their data science teams and ML capabilities to develop predictive models that can deliver business value to their customers.

Key problems affecting ML efforts today:

We've been here before

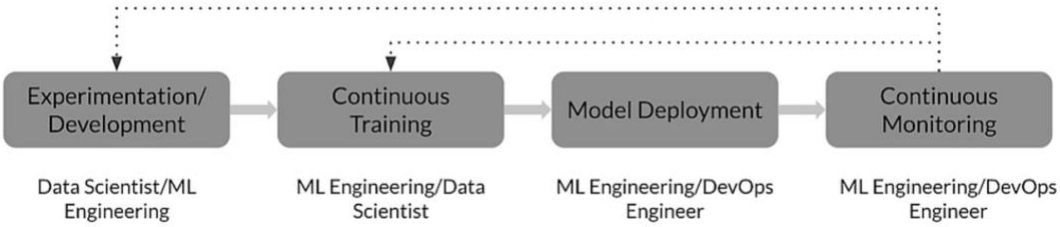
- In the 90s, Software Engineering was siloed
- Weak version control, CI/CD didn't exist
- Software was slow to ship; now it ships in minutes
- Is that ML today?

Today's perspective

- Models blocked before deployment
- Slow to market
- Manual tracking
- No reproducibility or provenance
- Inefficient collaboration

MLOps is therefore bridging between ML and IT. In addition to the classical CI/CD, a new process, CT (Continuous Training) has been created. It is unique to ML systems, that automatically retrains candidate models for testing and serving. Another process, CM (Continuous Monitoring), is very important while dealing with ML applications. It consists in catching errors in production systems and monitoring production inference data and model performance metrics tied to business outcomes.

ML Solution Lifecycle:



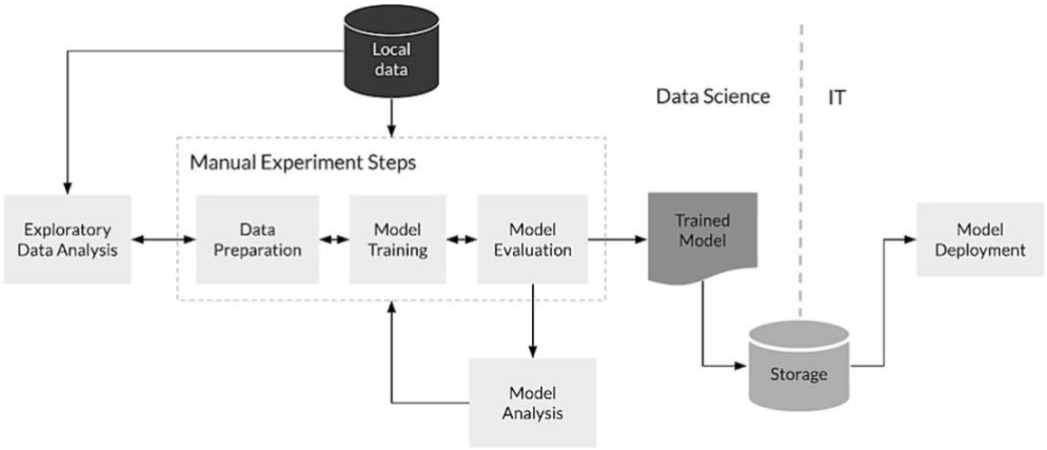
MLOps standardize ML process, by dealing with:

- ML Lifecycle management
- Model Versioning and iteration
- Model Monitoring and Management for model decay
- Model Governance
- Model Security
- Model Discovery (catalogs for example)

MLOps Level 0

The level of automation of ML pipelines determines the maturity of the MLOps process. As maturity increases, the available velocity for the training and deployment of new models also increases. The goal is to automate the training and the deployment of ML models into the core software system and provide monitoring.

MLOps Level 0:



At this level of automation, there is a disconnection between ML and operations. There are less frequent releases, so no CI/CD. The trained model is only deployed as a prediction service rather than deploying the entire ML system. The model predictions and actions are not logged.

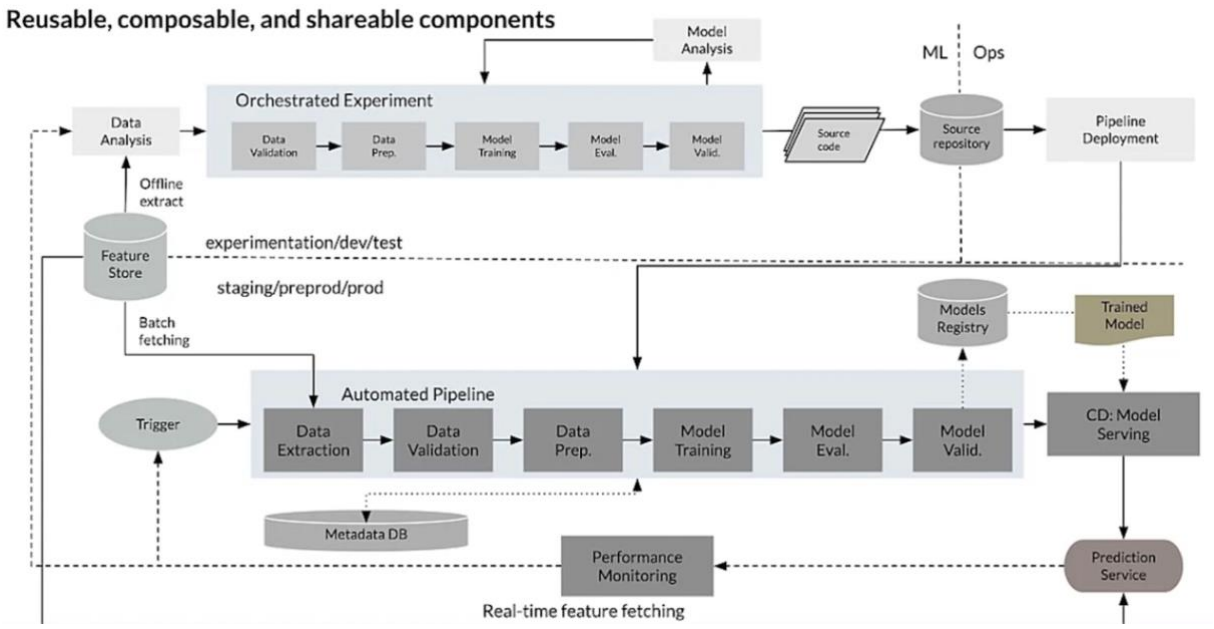
Challenges of Level 0 MLOps:

- Need for actively monitoring the quality of your model in production
- Retraining your production models with new data
- Continuously experimenting with new implementations to improve the data and model

MLOps Level 1 and 2

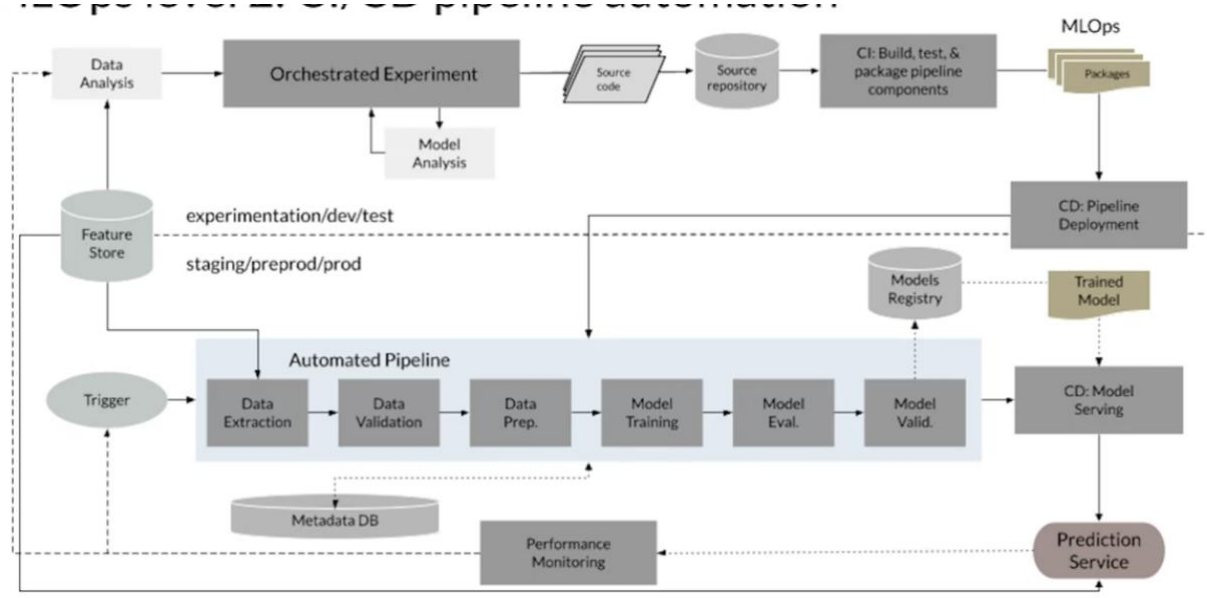
Level 1 performs continuous training of the model, by automating the training pipeline.

Since the steps of the experimentations are orchestrated, the transition between steps is automated. It is therefore easier to rapidly iterate on the experiments and makes it easier to move the whole pipeline to production.

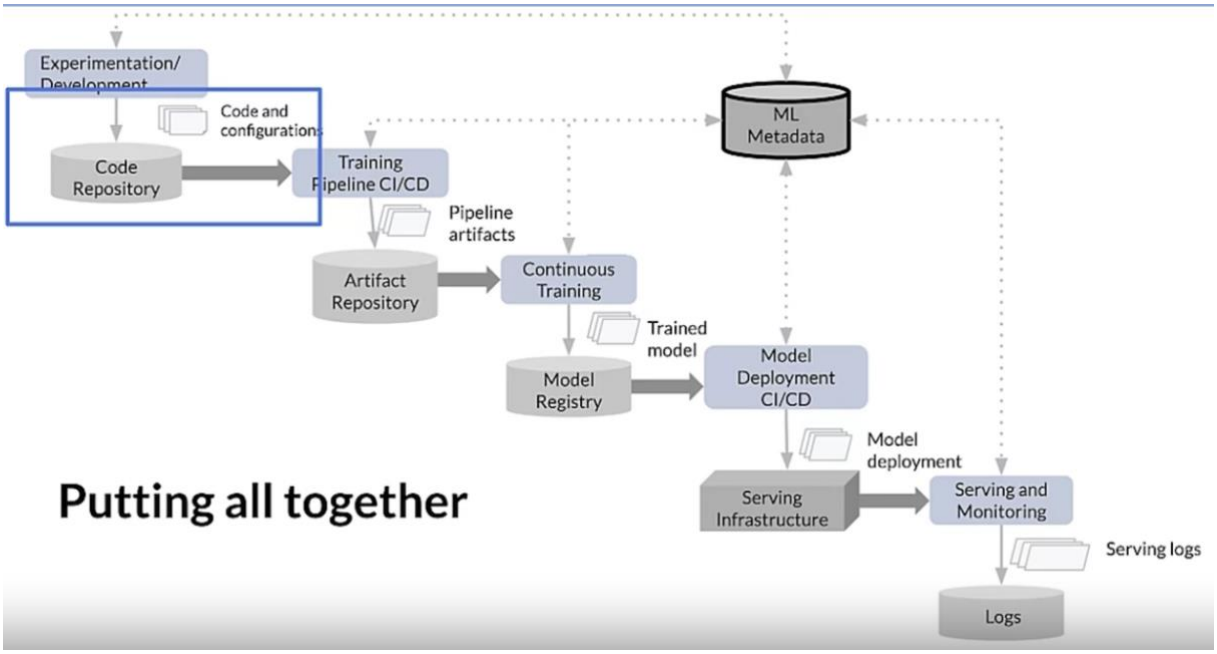


An ML pipeline in production continuously delivers new models that are trained on new data to prediction services. The model deployment step is automated, so the trained model is sent to production after being validated. Automated Data Validation can trigger a new training process to keep models up to date. It will analyze the schema of the data to send a notification if there is a change in the data and stop the pipeline. Model validation will run after successfully training the model given the new data. The model is here evaluated and validated before being promoted to production. It adds a Feature Store and a Metadata DB.

With the current state of MLOps, Level 2 is speculative:



The difference is the CI/CD pipeline automation to rapidly build and deploy tests for example.



Developing Components for an Orchestrated Workflow

TFX can be used to orchestrate the ML workflows. It includes pre-built and standard components, and 3 styles of custom components. Components can also be containerized. Some things it can do:

- Data augmentation, upsampling, or downsampling
- Anomaly detection based on confidence intervals or autoencoder reproduction error.
- Interfacing with external systems like help desks for alerting and monitoring.

Anatomy of a TFX component:

Component Specification

- The component's input and output contract

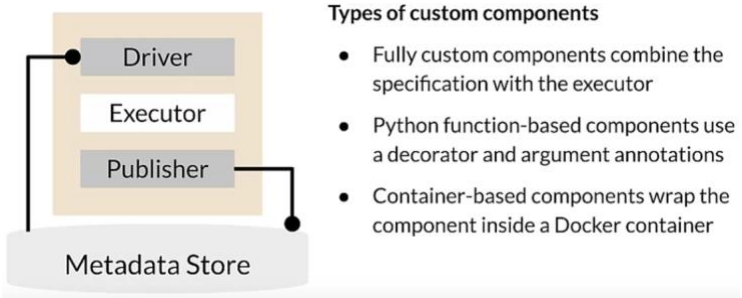
Executor Class

- Implementation of the component's processing

Component Class

- Combines the specification with the executor to create a TFX component

TFX components at runtime:

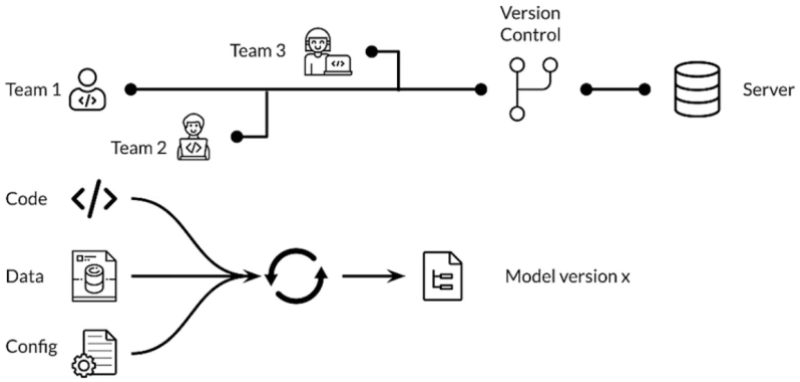


Custom components:

- Define custom component spec, executor class, and component class
- Component reusability
  - Reuse a component spec and implement a new executor that derives from an existing component

Managing Model Versions

Very important to level up organization:



How a software is versioned:

Version: MAJOR.MINOR.PATCH

- MAJOR: Contains incompatible API changes
- MINOR: Adds functionality in a backwards compatible manner
- PATCH: Makes backwards compatible bug fixes

In ML however there is no standard.

A proposition of versioning could be:

Version: MAJOR.MINOR.PIPELINE

- MAJOR: Incompatibility in data or target variable
- MINOR: Model performance is improved
- PIPELINE: Pipeline of model training is changed

A model registry is a central repository for storing trained ML models. It provides various operations of ML model development lifecycle and promotes model discovery, model understanding and model reuse. It can be integrated into OSS and commercial ML platforms.

Metadata need to be stored by model registry: model versions, model serialized artifacts, free text annotations and structured properties, and links to other ML artifact and metadata stores.

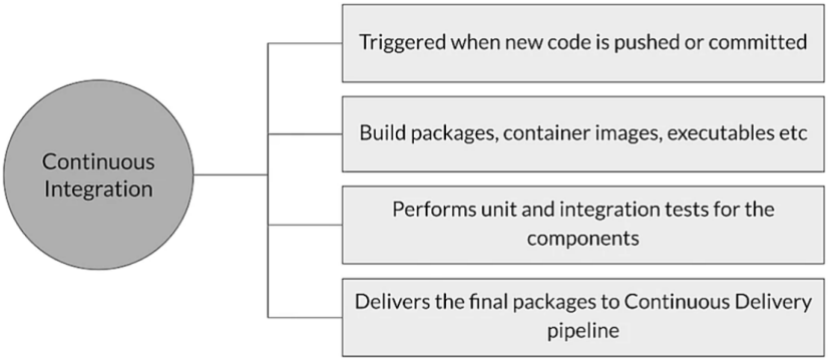
Capabilities enabled by Model registries:

- Model search/discovery and understanding
- Approval/Governance
- Collaboration/Discussion
- Streamlined deployments
- Continuous evaluation and monitoring
- Staging and promotions

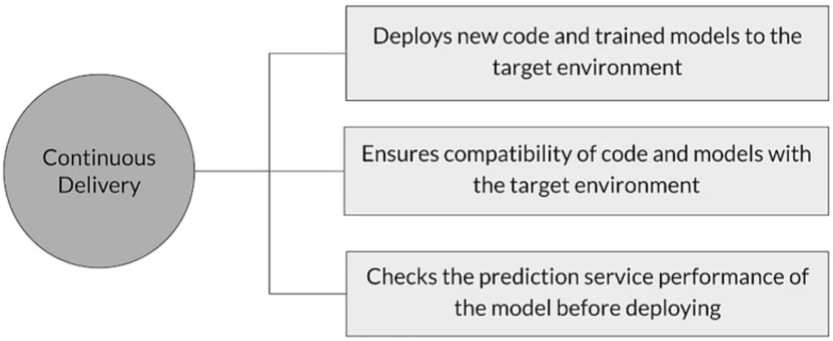
Some model registries are Azure ML Model Registry, SAS Model Manager, MLflow Model Registry, Google AI Platform and Algorithmia.

Continuous Delivery

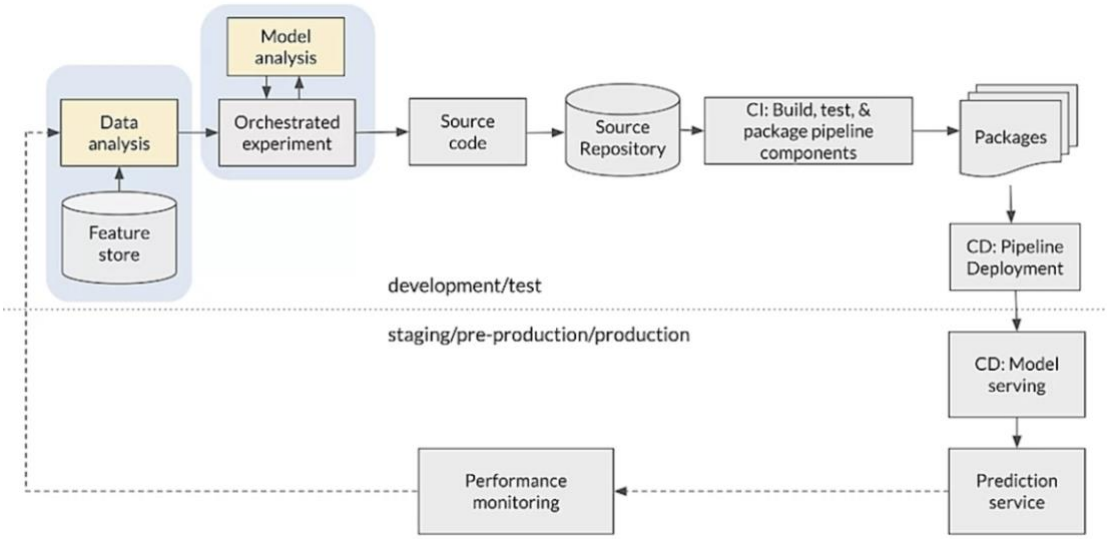
Continuous integration:



Continuous delivery:

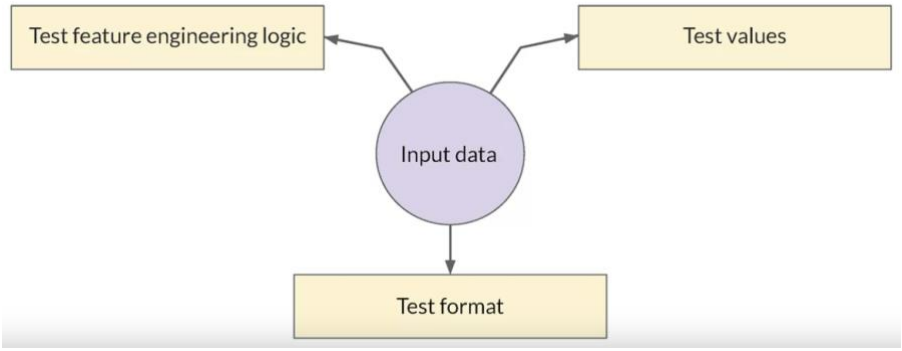


CI/CD infrastructure:

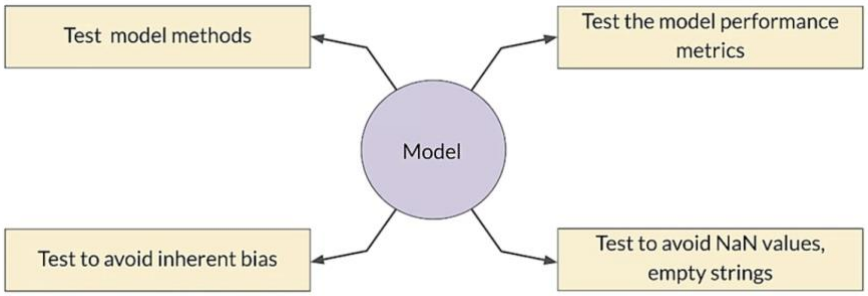


Unit testing in CI is used to verify that each component in the pipeline produces the expected artifact. In ML, both unit testing for data and model performance are needed.

For data:



For model:



In addition, for ML, it is important to have some attention on mocking (mocks of datasets), data coverage and code coverage.

Infrastructure validation:

When to apply infrastructure validation

- Before starting CI/CD as part of model training
- Can also occur as part of CI/CD as a last check to verify that the model is deployable to the serving infrastructure

TFX InfraValidator

- TFX InfraValidator takes the model, launches a sand-boxed model server with the model, and sees if it can be successfully loaded and optionally queried
- InfraValidator is using the same model server binary, same resources, and same server configuration as production

Progressive delivery

It is an improvement of CD. It consists in deploying gradually the new versions of the software. The first stage will only consist in delivering changes to small and low risk audiences, and then expand to larger and risker audiences. It allows to decrease deployment risks, a faster deployment, and gradual rollout and ownership.

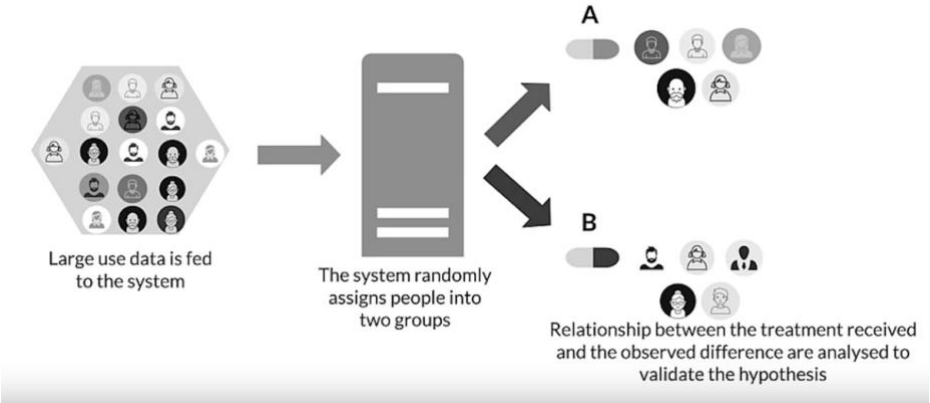
Some progressive delivery patterns:

- Shadow mode: ML turning in parallel of an operator but not used to do any decision, only to monitor the results to decide if we use this model.
- Canary mode: roll out to small fraction of the traffic to monitor the results, and then start increasing the traffic when sure that the results are okay.

- Blue green deployment: use a router to switch suddenly from the old/blue version to the new/green version (easier to rollback).

Live Experimentation is also available with progressive delivery, to measure model metrics for business cases.

A/B testing Live Experiment:

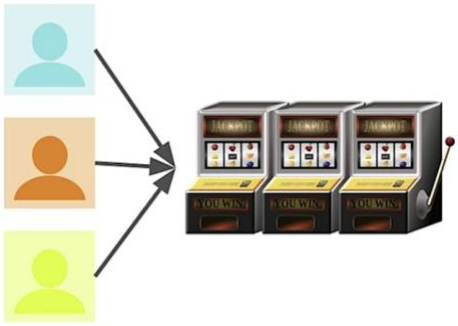


Multi-Armed Bandit (MAB):

- Uses ML to learn from test results during test
- Dynamically routes requests to winning models
- Eventually all requests are routed to one model
- Minimizes use of low-performing models

Contextual Bandit:

- Similar to multi-armed bandit, but also considers context of request
- Example: Users in different climates



Why Monitoring matters

Monitoring allows to verify if the system works. It is very important for the iterative process of the ML product lifecycle.

Reasons of the need of a monitoring:

- Immediate Data Skews
  - Training data is too old, not representative of live data
- Model Staleness
  - Environmental shifts
  - Consumer behaviour
  - Adversarial scenarios
- Negative Feedback Loops

Monitoring in ML systems:

ML Monitoring (functional monitoring)	System monitoring (non-functional monitoring)
Predictive performance	System performance
Changes in serving data	System status
Metrics used during training	System reliability
Characteristics of features	

Observability in ML

Observability measures how well the internal states of a system can be inferred by knowing the inputs and outputs. It comes from control system theory and is closely linked to controllability.

Modern systems can make observability difficult (cloud-based systems, containerized infrastructure, distributed systems and microservices).

Deep observability for ML does not only include top-level metrics. The domain knowledge is important for observability. TensorFlow Model Analysis is a great tool for that, both for supervised and unsupervised settings.

The goal of observability in ML is to make the system alertable (metrics and thresholds designed to make failures obvious) and actionable (root cause clearly identified).

Monitoring targets in ML

Input and output monitoring

Good metrics are:

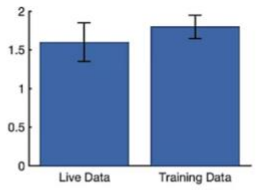
- Model input distribution
- Model prediction distribution
- Model versions
- Input/prediction correlation

Things to check on inputs:

Do these *check out*?	• Errors: Input values fall within an allowed set/range? • Changes: Distributions align with what you've seen in the past? • Per slice, e.g., marital status (single/married/widowed/divorced)

Prediction Monitoring

Example of monitoring for prediction using the SEM:

Statistical significance	 • Unsupervised: Compare model prediction distributions with statistical tests ○ e.g., median, mean, standard deviation, min/max values • Supervised: When labels are available

Operational Monitoring

ML engineering	Software engineering
Latency	Receiving an HTTP request
IO / Memory / Disk Utilisation	Entering/leaving a function
System Reliability (Uptime)	A user logging in
Auditability	Reading from net / writing to disk

Logging for ML Monitoring

Steps for building observability:

- Start with the out-of-the-box logs, metrics and dashboards
- Add agents to collect additional logs and metrics
- Add logs-based metrics and alerting to create your own metrics and alerts
- Use aggregated sinks and workspaces to centralize your logs and monitoring

An event log (or simply “log”) is an immutable, time-stamped record of discrete events that happened over time.

Some tools for logging are Google Cloud Monitoring, Amazon CloudWatch, and Azure Monitor.

Advantages of logging: easy to generate, great when it comes to providing valuable insight, and focuses on specific events.

Disadvantages of logging: excessive logging can impact system performance, aggregation operations on logs can be expensive, and setting up and maintaining tooling carries with it a significant operational cost.

Logging in ML:

Key areas
➔ Use logs to keep track of the model inputs and predictions

Input red flags
➔ A feature becoming unavailable
➔ Notable shifts in the distributions
➔ Patterns specific to your model

Storing log data for analysis can be difficult. It enables automated reporting, dashboards, and alerting. Basic log storage is often unstructured but parsing and storing log data in a queryable format enables analysis. It allows extracting values to generate distributions and statistics, associating events with timestamps, and identifying the system.

Log data is very important for getting new training data (high valuable):

- Prediction requests form new training datasets
- For supervised learning, labels are required
  - Direct labeling
  - Manual labeling
  - Active learning
  - Weak supervision

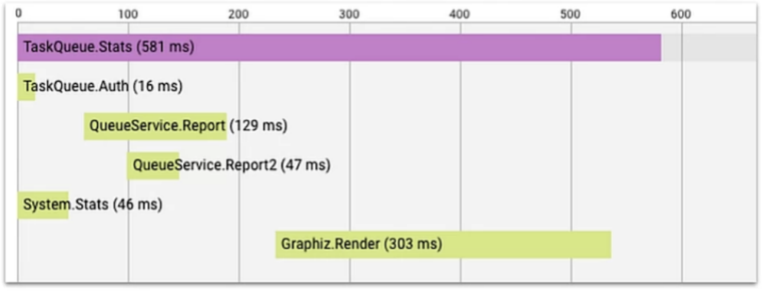
Tracing for ML systems

Tracing focuses on monitoring and understanding system performance especially for microservice-based applications.

Tools for building observability with distributed tracing: Dapper, Zipkin, Jaeger.

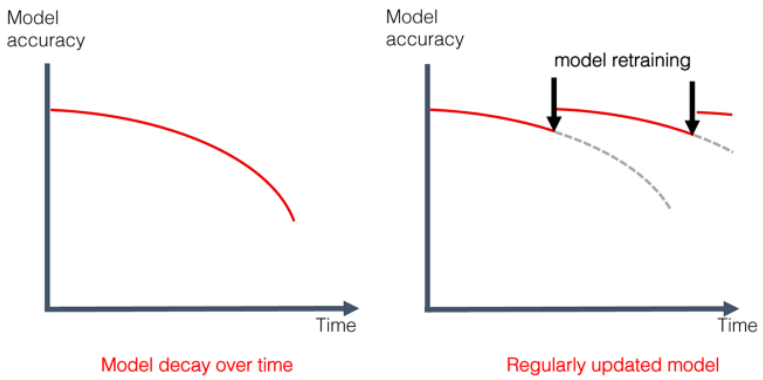
Dapper-Style Tracing:

- Propagate trace between services
- A trace is a call tree, made up of one or more spans



What is Model Decay

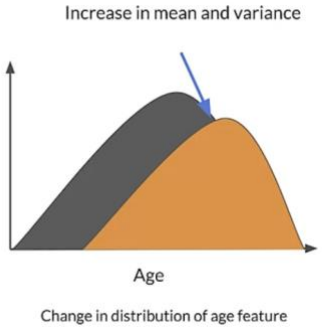
It is possible to retrain automatically or manually the system to correct the drifts:



Production ML models often operate in dynamic environments. The ground truth in dynamic environments changes. If the model is static and does not change, then it gradually moves farther and farther away from the ground truth. It can be due to a data drift or a concept drift.

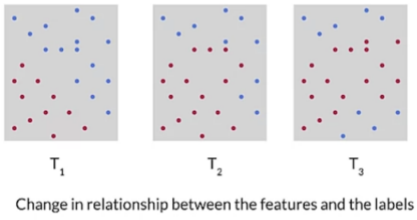
Data drift:

- Statistical properties of input changes
- Trained model is not relevant for changed data
- For eg., distribution of demographic data like age might change over time



Concept drift:

- Relationship between features and labels changes
- The very meaning of what you are trying to predict changes
- Prediction drift and label drift are similar



Detecting drift on time:

- Drift creeps into the system slowly with time
- If it goes undetected, model accuracy suffers
- Important to monitor and detect drift early

Model decay detection

Logging predictions is very necessary to detect these drifts. It is important to log full requests and responses:

- Incoming prediction requests and generated prediction should be logged
- If possible log the ground truth that should have been predicted
  - Can be used as labels for new training data
- At a minimum log data in prediction request
  - This data is analysed to detect data drift that will cause model decay

Detecting drift can be done by observing the statistical properties of logged data, model predictions, and possibly ground truth. Deploying dashboards that plot statistical properties to observe how they change over time is a good way to visualize them. Using specialized libraries for detecting drift is good too.

Continuous evaluation and labelling in Vertex Prediction:

- **Vertex Prediction** offers continuous evaluation
- **Vertex Labelling Service** can be used to assign ground truth labels to prediction input for retraining
- Azure, AWS, and other cloud providers offer similar services

Mitigate Model Decay

When a model decay has been detected, at the minimum operational and business stakeholders should be notified and then take steps to bring model back to acceptable performance.

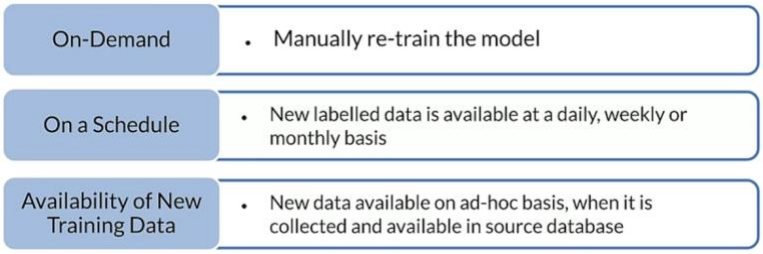
Steps in Mitigating Model Decay:

- What if Drift is Detected?
  - If possible, determine the portion of your training set that is still correct
  - Keep the good data, discard the bad, and add new data - OR -
  - Discard data collected before a certain date and add new data - OR -
  - Create an entirely new training dataset from new data

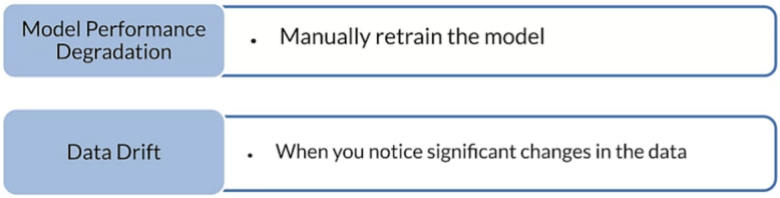
Two solutions: Fine Tune or Start Over.

- You can either continue training your model, fine tuning from the last checkpoint using new data - OR -
- Start over, reinitialize your model, and completely retrain it
- Either approach is valid, so it really depends on results
  - How much new labelled data do you have?
  - How far has it drifted?
  - Try both and compare

Model re-training policy:



Automating Model retraining:



Redesign data processing steps and model architecture:

- When model performance decays beyond an acceptable threshold you might have to consider redesigning your entire pipeline
- Re-think feature engineering, feature selection
- You may have to train your model from scratch
- Investigate on alternative architectures

Responsible AI

Development of AI creates new opportunities to improve the lives of people around the world, but also raises new questions about implementing responsible practices (fairness, interpretability, privacy, and security).

Actual users' experience is essential. Designing the features with appropriate disclosures built-in, considering augmentation and assistance, model potential adverse feedback early in the design process, and engaging with a diverse set of users and use-case scenarios are important things to keep in mind.

Identifying multiple metrics to monitor helps understanding the tradeoffs (feedback from user surveys, quantities that track overall system performance, false positive and false negative sliced across subgroups). Metrics must be appropriate for the context and goals of the system.

Raw data need to be analyzed carefully:

- For sensitive raw data, respect privacy
  - Compute aggregate, anonymized summaries
- Does your data reflect your users?
  - Example: will be used for all ages, but all data from senior citizens
- Imperfect proxy labels?
  - Relationship between the labels and actual targets

Legal requirements for Secure and Private AI

Companies must comply with data privacy protection laws in regions where they operate (for instance GDPR, General Data Protection Regulation, or CCPA, California Consumer Privacy Act).

GDPR:

- Regulation in EU law on data protection and privacy in the European Union (EU) and the European Economic Area (EEA)
- Give control to individuals over their data
- Companies should protect the data of employees and consumers
- When the data processing is based on consent, the data subject has the right to revoke their consent at any time

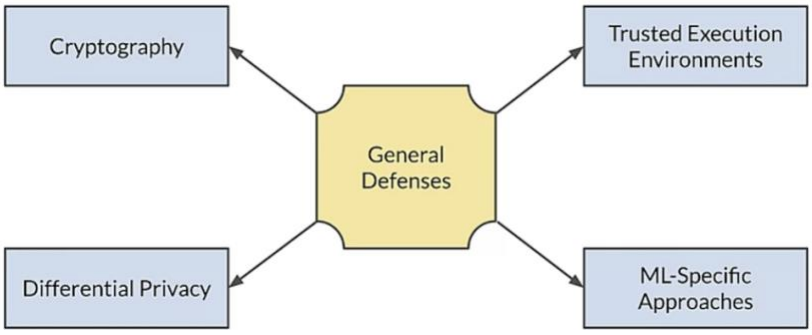
CCPA:

- Similar to GDPR
- Intended to enhance privacy rights and consumer protection for residents of California
- User has the right to know what personal data is being collected about them, whether the personal data is sold or disclosed, and to whom
- User can access the personal data, block the sale of their data, and request a business to delete their data

Security and Privacy Harms for ML models:

Informational Harms	Behavioural Harms
Relate to unintended or unanticipated leakage of information	Relate to manipulating the behavior of the model itself, impacting the predictions or outcomes of the model

Defenses:



Cryptography

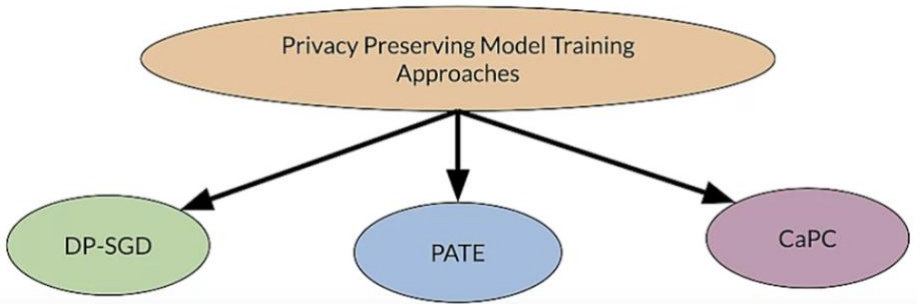
Privacy-enhancing tools (like SMPC and FHE) should be considered to securely train supervised machine learning models.

Users can send encrypted predictions requests while preserving the confidentiality of the model.

It protects confidentiality of the training data.

Differential Privacy

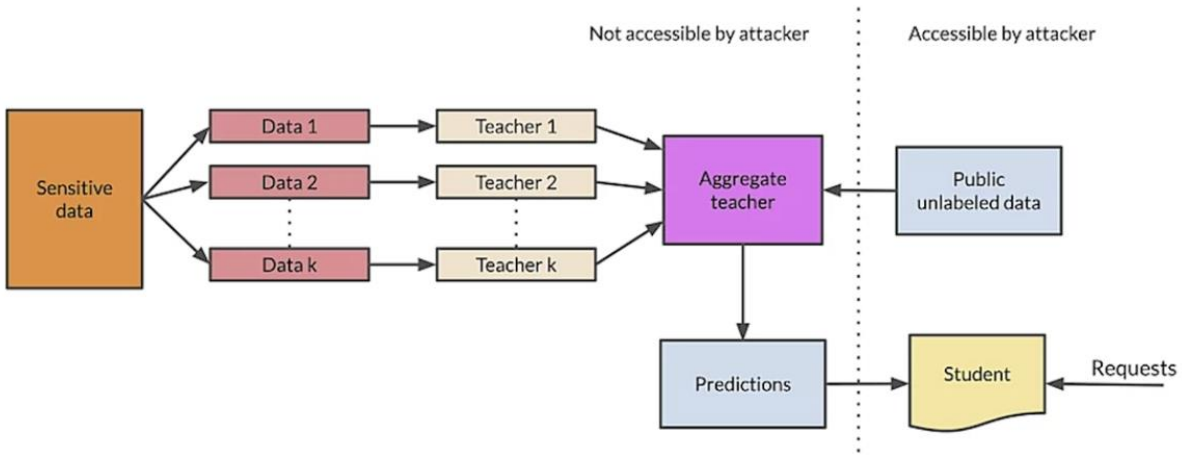
DP is a system for publicly sharing information about a dataset by describing the patterns of groups within the dataset while withholding information about individuals in the dataset.



Differentially-Private Stochastic Gradient Descent (DP-SGD):

- Applies differential privacy during model training
- Modifies the minibatch stochastic optimization process
- Trained model retains differential privacy because of the post-processing immunity property of differential privacy

Private Aggregation of Teacher Ensembles (PATE):



Confidential and Private Collaborative Learning (CaPC):

- Enables models to collaborate while preserving the privacy of the underlying data
- Integrates building blocks from cryptography and differential privacy to provide confidential and private collaborative learning
- Encrypts prediction requests using Homomorphic Encryption (HE)
- Uses PATE to add noise to predictions for voting

Anonymization and Pseudonymization

GDPR includes many regulations to preserve privacy of user data. Since the introduction of GDPR, two terms have been discussed widely: Anonymization and Pseudonymization.

Data Anonymization:

Recital 26 of GDPR defines Data Anonymization

True data anonymization is:

- Irreversible
- Done in such a way that it is impossible to identify the person
- Impossible to derive insights or discrete information, even by the party responsible for anonymization

GDPR does not apply to data that has been anonymized

Pseudonymization:

- GDPR Article 4(5) defines pseudonymisation as:

  "... the processing of personal data in such a way that the data can no longer be attributed to a specific data subject without the use of additional information"

- The data is anonymized by switching the identifiers (like email or name) with an alias or pseudonym

Difference between the two:

Information	Pseudonymized	Anonymized
Chelsea	Puryfrn	*******
Kumar	Xhzne	*******
Zaed	Mnrq	*******

Spectrum of Privacy Preservation:

What Data should be anonymized:

- Any data that reveals the identity of a person, referred to as identifiers
- Identifiers applies to any natural or legal person, living or dead, including their dependents, ascendants, and descendants
- Included are other related persons, direct or through interaction
- For example: Family names, patronyms, first names, maiden names, aliases, address, phone, bank account details, credit cards, IDs like SSN

Right to be Forgotten

It is defined in GDPR (recitals 65 and 66 in Article 17).

Implementing the right to be forgotten with tracking data:

For a valid erasure claim

- Company needs to identify all of the information related to the content requested to be removed
- All of the associated metadata must also be erased
    - Eg., Derived data, logs etc.

There are two ways for forgetting digital memories: anonymize or hard delete the data.

Issues with hard delete:

- Deleting records from a database can cause havoc
- User data is often referenced in multiple tables
- Deletion breaks the connections, which can be difficult in large, complex databases
- Can break foreign keys
- Anonymization keeps the records, and only anonymizes the fields containing PII

Three challenges in implementing the Right to Be Forgotten:

- Identifying if data privacy is violated.
- Organizational changes for enforcing GDPR.
- Deleting personal data from multiple back-ups.

Other rights of the data subject:

Chapter 3 defines a number of other rights of the data subject, including:

- Art. 15 GDPR – Right of access by the data subject
- Art. 18 GDPR – Right to restriction of processing
- Art. 20 GDPR – Right to data portability
- Art. 21 GDPR – Right to object