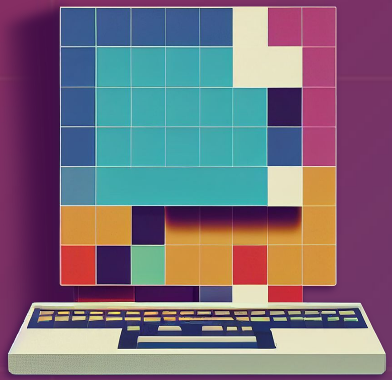
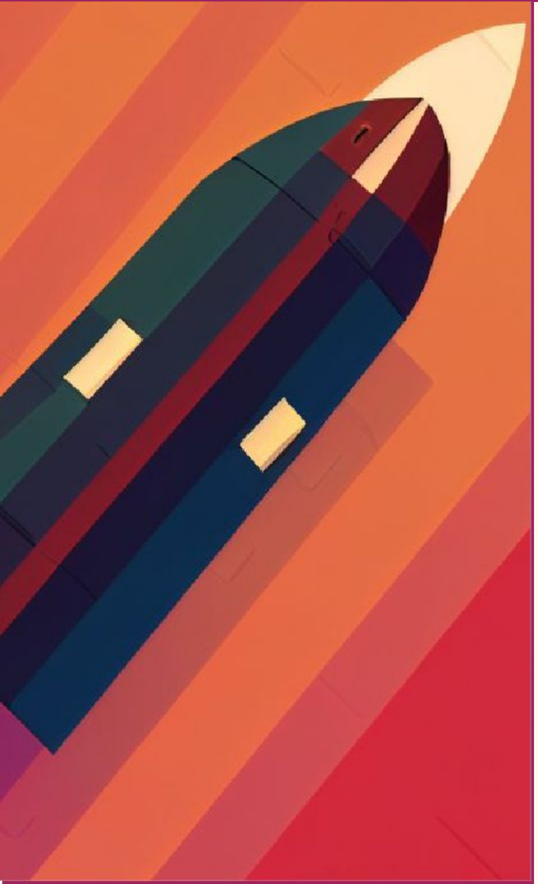


MODULE 02

Feature Engineering and Machine Learning Pipelines

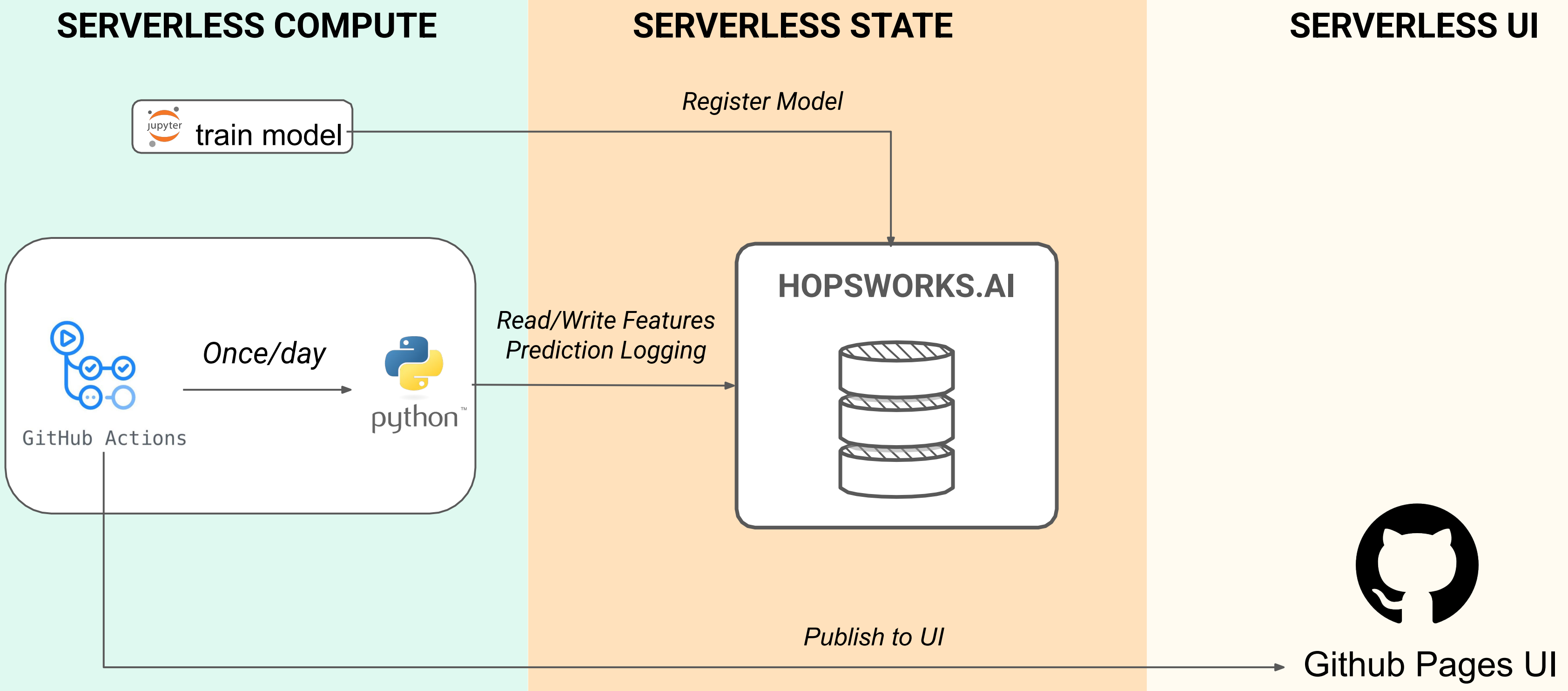
February 2025

What will we cover in this Module



- Feature Engineering in Pandas
- Machine Learning (ML) Pipelines
- Feature Stores for ML
- Feature Pipeline orchestration
- Lab 1: Iris Flower Dataset
    - Enable predictive analytics with Gradio (UI)
    - Build a prediction service with Github Pages, pipelines, and model performance monitoring

What we we will build in this module - a ML system



ML with Tabular Data

In this Module, we will build prediction systems using tabular data

Tabular data is much of the Enterprise Data that is stored in Data Warehouses, Data Lakes, Databases, files.

Column (*feature*)

Table (*feature group*)

credit_card_number	amount	location
1111 2222 3333 4444	$142.34	Sao Paulo
1111 2222 3333 4444	$12.34	Rio De Janeiro
1111 2222 3333 4444	$66.29	Stockholm
1111 2222 3333 4444	$112.33	Stockholm

Row (*vector*)

Primary Key (*Entity ID*)

Data (feature) value

Supervised Machine Learning with Tabular Data

Entity Key	event_time	Feature	Feature	Label
credit_card_number	event_time	amount	location	Fraud
1111 2222 3333 4444	2025-01-01 08:44	$142.34	Sao Paulo	False
1111 2222 3333 4444	2025-01-01 19:44	$12.34	Rio De Janeiro	False
1111 2222 3333 4444	2025-01-01 20:44	$66.29	Stockholm	True
1111 2222 3333 4444	2025-01-01 20:55	$112.33	Stockholm	True

Row

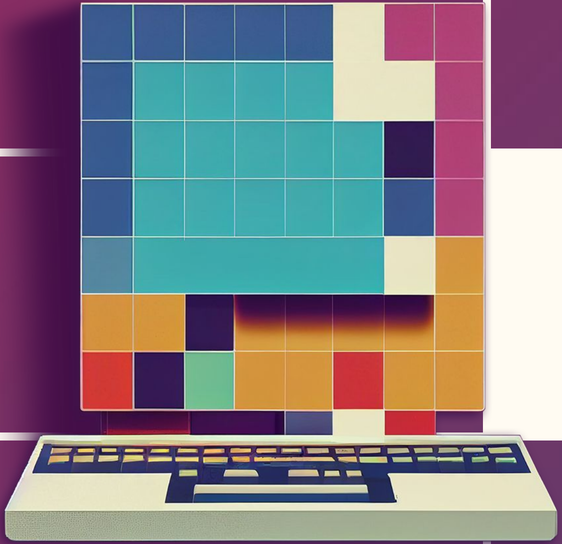
Feature Vector

Training a model with features and labels is as follows:

- **train(feature_vectors, labels) -> model**

Model inference, when you have the model and the features, is as follows:

- **model(feature_vector) -> prediction**



Brief Intro to Pandas

Tabular Data as Pandas DataFrames with DTypes

Object	Datetime	Float64	Object	Bool
credit_card_number	event_time	amount	location	Fraud
1111 2222 3333 4444	2025-01-01 08:44	$142.34	Sao Paulo	False
1111 2222 3333 4444	2025-01-01 19:44	$12.34	Rio De Janeiro	False
1111 2222 3333 4444	2025-01-01 20:44	$66.29	Stockholm	True
1111 2222 3333 4444	2025-01-01 20:55	$112.33	Stockholm	True

Row

Pandas DataFrame with DTypes

```
import pandas as pd

data = {
    'credit_card_number': ['1111 2222 3333 4444', '1111 2222 3333 4444','1111 2222 3333 4444',
                           '1111 2222 3333 4444'],
    'trans_datetime': ['2022-01-01 08:44', '2022-01-01 19:44', '2022-01-01 20:44', '2022-01-01 20:55'],
    'amount': [142.34, 12.34, 66.29, 112.33],
    'location': ['Sao Paolo', 'Rio De Janeiro', 'Stockholm', 'Stockholm'],
    'fraud': [False, False, True, True]
}

df = pd.DataFrame.from_dict(data)

df
```

dict containing data for the DataFrame (df)

create DataFrame (df) using dict data

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 5 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   credit_card_number  3 non-null      object
 1   trans_datetime      3 non-null      object
 2   amount              3 non-null      float64
 3   location            3 non-null      object
 4   fraud               3 non-null      bool
```

Need to change from object DType to **datetime**

Pandas DataFrame with DTypes

```
df['trans_datetime']= pd.to_datetime(df['trans_datetime'])
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 5 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   credit_card_number  3 non-null      object
 1   trans_datetime      3 non-null      datetime64[ns]
 2   amount              3 non-null      float64
 3   location            3 non-null      object
 4   fraud               3 non-null      bool
```

It is now **datetime**

User-Defined Functions, Apply, and Lambda in Pandas

- Instead of a for-loop to process rows, use the **apply** command
- "Inline" functions, called a **lambda**, can be applied to a column (**series**) or a DataFrame
- Call **apply** on a DataFrame with a Python function, called user-defined functions (**UDFs**)

A **lambda** function applied to a **series** here

```
df['is_big'] = df['amount'].apply(lambda amount: amount > 100)
df
```

	credit_card_number	trans_datetime	amount	location	fraud	is_big
0	1111 2222 3333 4444	2022-01-01 08:44:00	142.34	Sao Paolo	False	True
1	1111 2222 3333 4444	2022-01-01 19:44:00	12.34	Rio De Janeiro	False	False
2	1111 2222 3333 4444	2022-01-01 20:44:00	66.29	Stockholm	True	False
3	1111 2222 3333 4444	2022-01-01 20:55:00	112.33	Stockholm	True	True

A Python UDF applied to a **DataFrame**

```
def is_small(row):
    return row['amount'] < 100

df['is_small'] = df.apply(is_small, axis=1)
df
```

	credit_card_number	trans_datetime	amount	location	fraud	is_big	is_small
0	1111 2222 3333 4444	2022-01-01 08:44:00	142.34	Sao Paolo	False	True	False
1	1111 2222 3333 4444	2022-01-01 19:44:00	12.34	Rio De Janeiro	False	False	True
2	1111 2222 3333 4444	2022-01-01 20:44:00	66.29	Stockholm	True	False	True
3	1111 2222 3333 4444	2022-01-01 20:55:00	112.33	Stockholm	True	True	False

Efficient Pandas with vectorized operations

- For small volumes of data (MBs in size), you generally don't need to worry about efficiency in Pandas.
- As your datasets increase in size, you can massively speed up your computations by using vectorized operations instead of UDFs and apply.

Vectorized operations are faster than "apply" with UDFs

We will see that apply is approximately 50 times slower than the equivalent vectorized operation on 100k rows.

```
%%timeit
df2['a'].apply(lambda x: x**2)
```

```
3.42 ms ± 26.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

This vectorized operation is much faster

```
%%timeit
df2['a'] ** 2
```

```
59 µs ± 3.28 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

https://towardsdatascience.com/efficient-pandas-apply-vs-vectorized-operations-91ca17669e84

EDA in Pandas

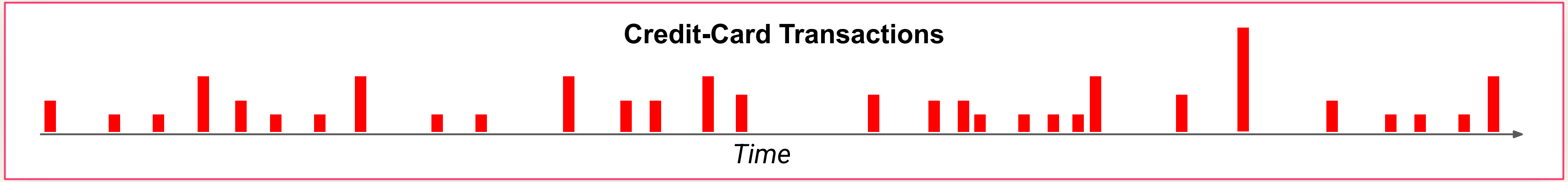
Useful EDA Commands	Description
df.head()	Returns the first few rows of *df*.
df.describe()	Returns descriptive statistics for *df*. Use with numerical features.
df[col].unique()	Returns all values unique for a column, *col*, in *df*.
df[col].nunique()	Returns the number of unique values for a column, *col*, in *df*.
df.isnull().sum()	Returns the number of null values in all columns in *df*.
df[col].value_counts()	Returns the number of values for with different values. Use with both numerical and categorical variables.
sns.histplot(...)	Plot a histogram for a DataFrame or selected columns using Seaborn.

Aggregations in Pandas - for Series or whole DataFrames

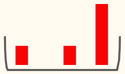
Aggregation	Description
df.count()	Count the number of rows
df.first(), df.last()	First and last rows
df.mean(), df.median()	Mean and median
df.min(), df.max()	Minimum and maximum
df.std(), df.var()	Standard deviation and variance
df.mad()	Mean absolute deviation
df.prod()	Product of all rows
df.sum()	Sum of all rows

What is the 7 day rolling max/mean of the credit card transaction amounts?

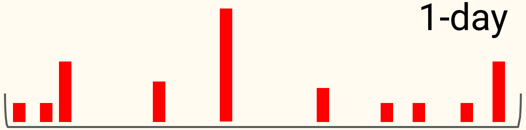
```
# For rolling windows in Pandas, first set a DateTime column as index to the df
```




```
df.rolling('1D').amount.max()
```




```
df.rolling('1W').amount.mean()
```



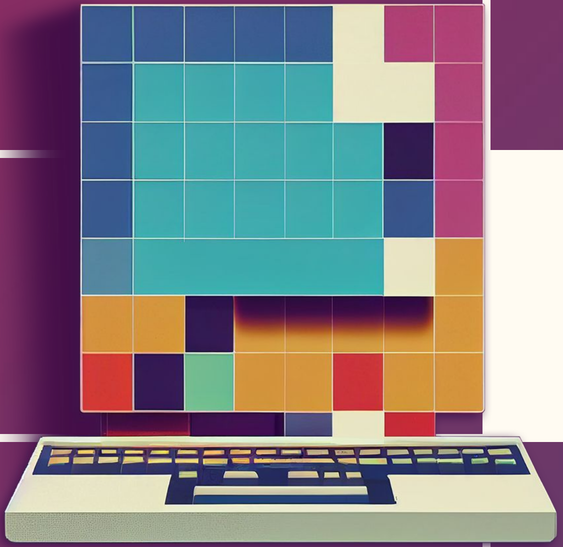

```
df.rolling('30D').amount.min()
```

More Reading on Pandas

- Intro to Pandas on Colab
  https://colab.research.google.com/github/google/eng-edu/blob/main/ml/cc/prework/intro_to_pandas.ipynb?utm_source=ss-data-prep&utm_campaign=colab-external&utm_me dium=referral&utm_content=pandas-colab

- Working with Missing Data
  http://pandas.pydata.org/pandas-docs/stable/missing_data.html

- Visualizations
  http://pandas.pydata.org/pandas-docs/stable/visualization.html

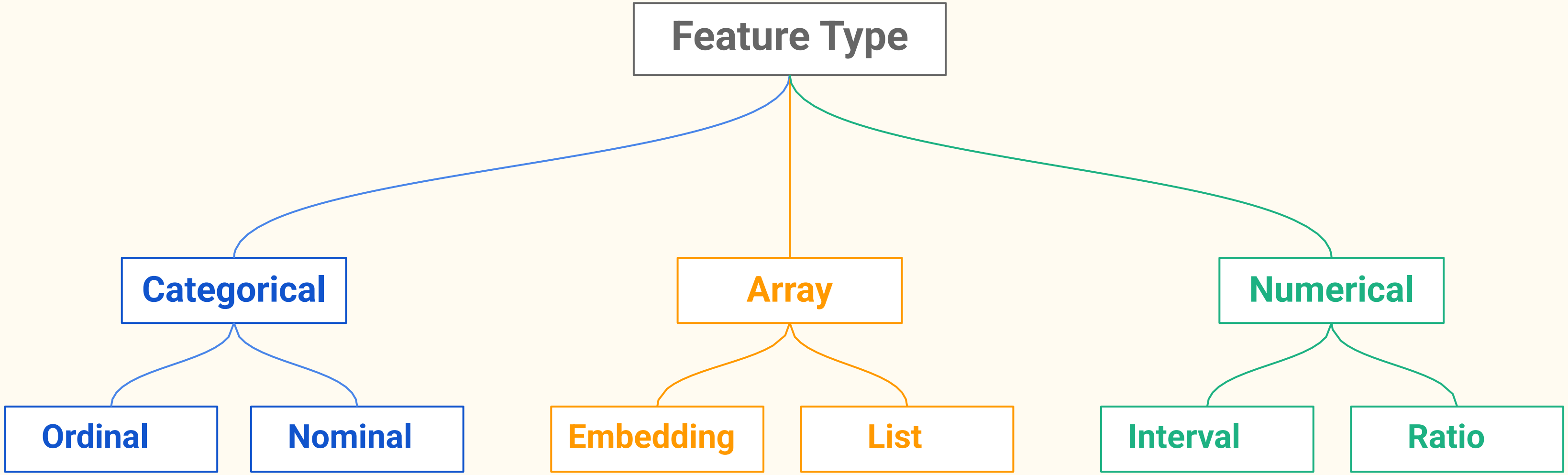


Training ML Models with DataFrames

What are Feature Types in ML?

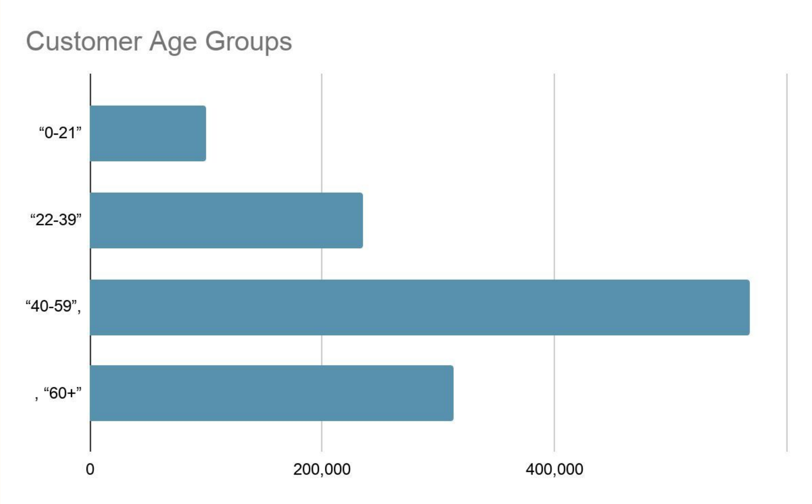
credit_card_number	event_time	amount	location	Fraud
<primary_key>	***<event_time>***	***<numerical feature>***	***<categorical feature>***	***<label>***
1111 2222 3333 4444	2022-01-01 08:44	$142.34	Sao Paulo	False
1111 2222 3333 4444	2022-01-01 19:44	$12.34	Rio De Janeiro	False
1111 2222 3333 4444	2022-01-01 20:44	$66.29	Stockholm	True
1111 2222 3333 4444	2022-01-01 20:55	$112.33	Stockholm	True

Feature Types for ML



Feature Binning

- If we want to train a model to discover relationships between the customer's date of birth and their product preferences, we need a huge amount of training data with lots of examples of customers born on the same day.
- Instead, we can transform the date of birth (a numerical feature) into a categorical (ratio) variable: "0-21", "22-39", "40-59", "60+" or "under-21"/"over-21".
- In general, numerical variables can be transformed into categorical variables using binning.



Feature Interactions/Crosses

- Feature crossing represents the co-occurrence of features, which may be highly correlated with the target label.
- Deep neural networks are good at learning feature crossing, but deep learning is not the modelling algorithm of choice for tabular data. XGBoost is [Grinsztajn et Al]
- For numerical features, say credit card amount and air temperature, a feature cross can be created by multiplying one column with the other (but, it's generally better to bin, first).
- For categorical data, a feature cross is the cartesian product of the categories.
- Finding good feature crosses requires either deep insights into the problem domain or testing combinations of different variables.

Binned Latitude	Binned Longitude	Rooms / Person
xx	yy	0.6

→

b-lat⊕b-long⊕rooms/person
xx⊕yy⊕0.6

https://towardsdatascience.com/feature-interactions-524815abec81

https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture

Compressed data as Features (Embeddings)

- An embedding is a low-dimensional, learned, continuous vector of discrete variables into which you can translate high-dimensional vectors. It is an array of floats[1.19, 4.1, …,1.34]
- Embeddings create a denser representation of the categories and maintain some of the implicit relationship information between the input vectors (examples).

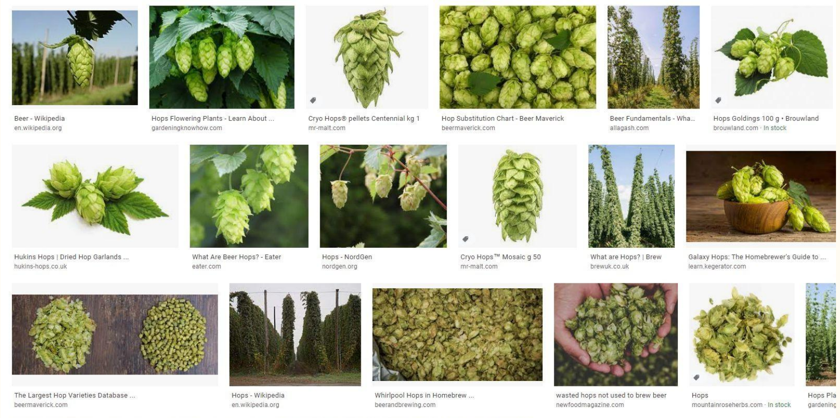
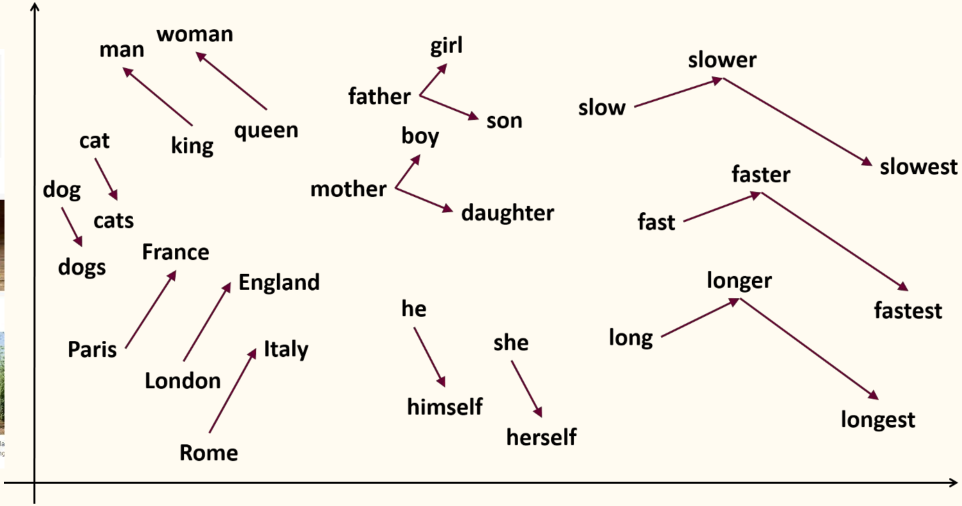


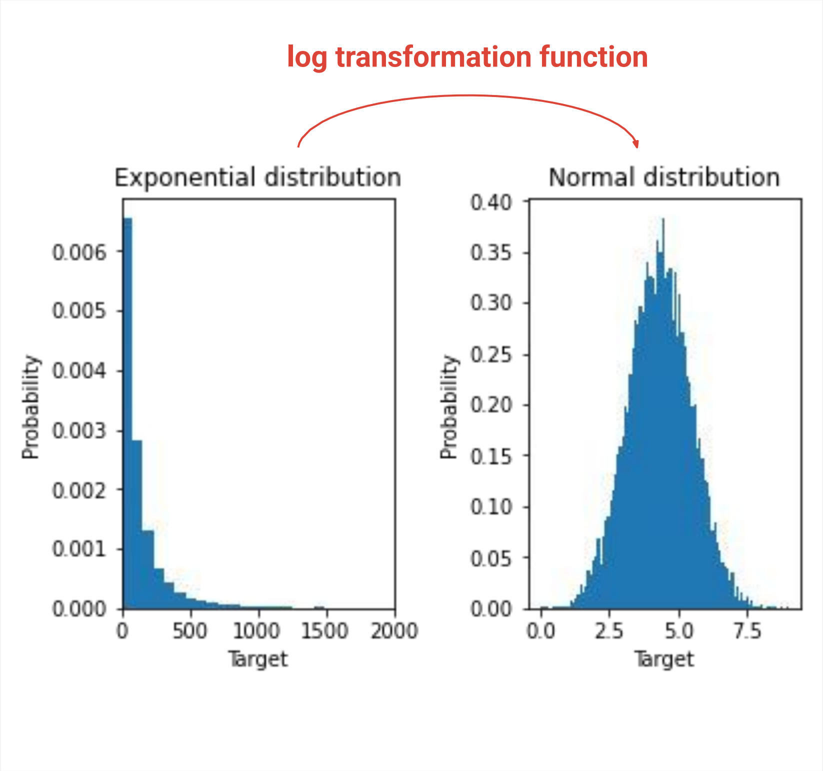
Image Embeddings enable Similarity Search



Transformations

- **Transformations for data compatibility**
  - Convert non-numeric features into numeric
  - Resize inputs to a fixed size

- **Transformations to improve model performance**
  - Many models perform badly if numerical features do not follow a normal (Gaussian) distribution
  - Tokenization or lower-casing of text features
  - Allowing linear models to introduce non-linearities into the feature space



Background on Transformations

- https://developers.google.com/machine-learning/data-prep/transform/introduction

Transformations

Type of Transformation
Scaling to Minimum And Maximum values
Scaling To Median And Quantiles
Gaussian Transformation
Logarithmic Transformation
Reciprocal Transformation
Square Root Transformation
Exponential Transformation
Box Cox Transformation

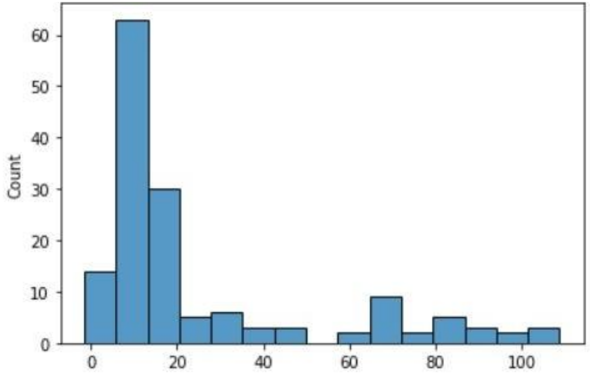
ML Algorithms that may need Transformations
Linear regression
Logistic regression
K Nearest neighbours
Neural networks
Support vector machines with radial bias kernel functions
Principal components analysis
Linear discriminant analysis

Note: tree-based models do not need transformations

Reading on Types of Transformations

- https://towardsdatascience.com/how-to-differentiate-between-scaling-normalization-and-log-transformations-69873d365a94

Transformations - MinMax Normalization example in Pandas

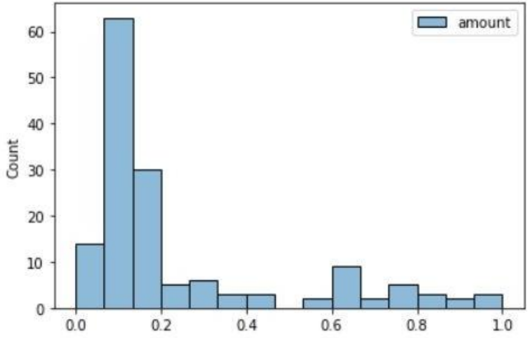



```
columns = ['amount']
df_exp = pd.DataFrame(data = array, columns = columns)
```

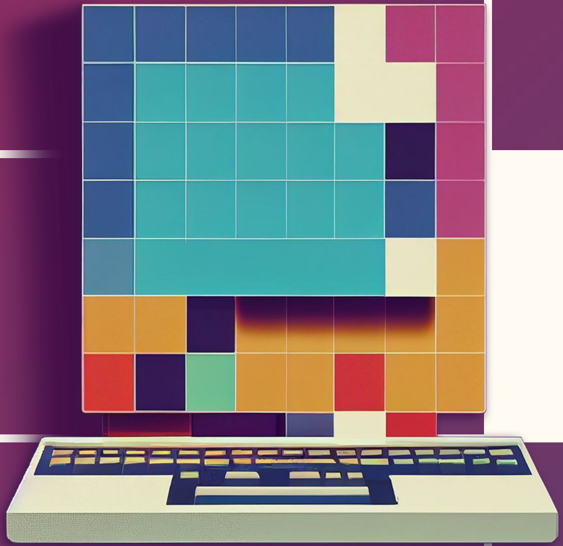
```
# Min-Max Normalization in Pandas
df_norm = (df_exp-df_exp.min())/(df_exp.max()-df_exp.min())
df_norm.head()
```

```
sns.histplot(df_norm)
```

```
<AxesSubplot:ylabel='Count'>
```



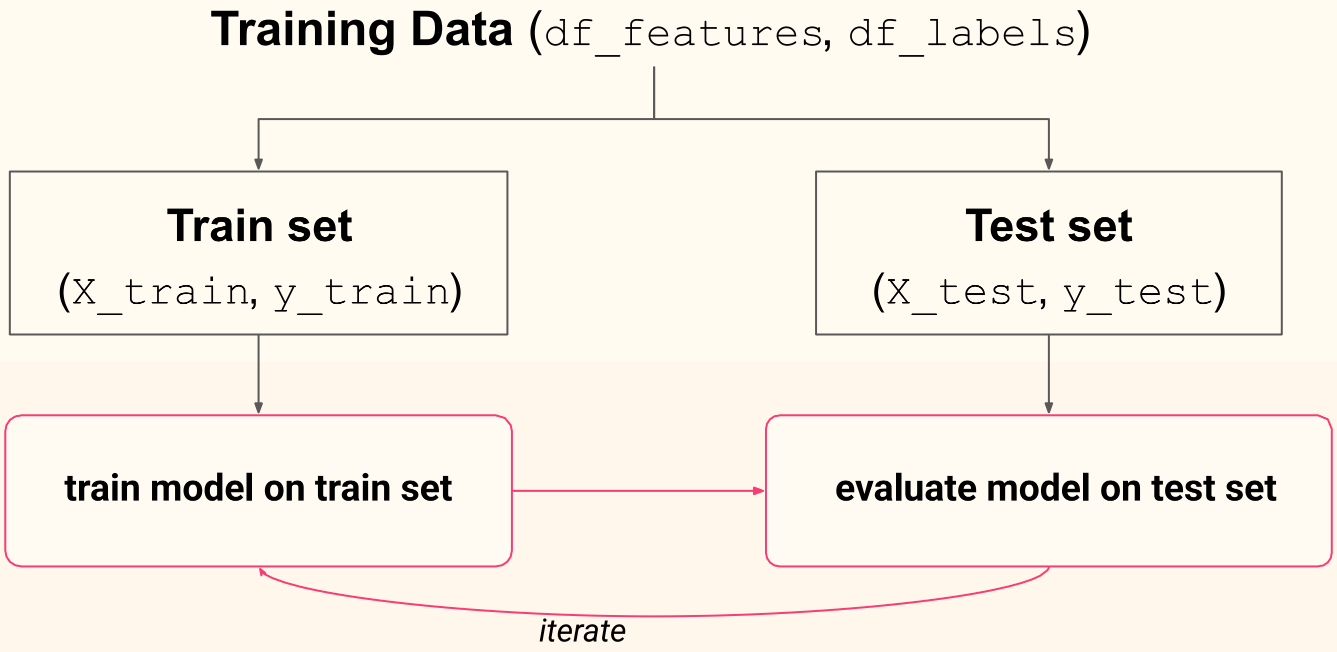
Scikit-Learn has native support for many transformations



Model Training

Splitting Pandas DataFrames into Train/Test Sets & Features(X) and Labels(y)

```
from sklearn.model_selection  import train_test_split
X_train,X_test,y_train,y_test = train_test_split(df_features, df_labels, test_size=0.2)
```



Model training/eval - we follow this pattern in this Module

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import xgboost as xgb
```

```
X_train,X_test,y_train,y_test =
    train_test_split(features, labels, test_size=0.2)
```

Get train and test data sets as features (X) and labels (y)

```
model = xgb.XGBClassifier()
```

Use XGBoost as modelling algorithm

```
model.fit(X_train,y_train)
```

Train supervised ML classifier with features and labels from train set

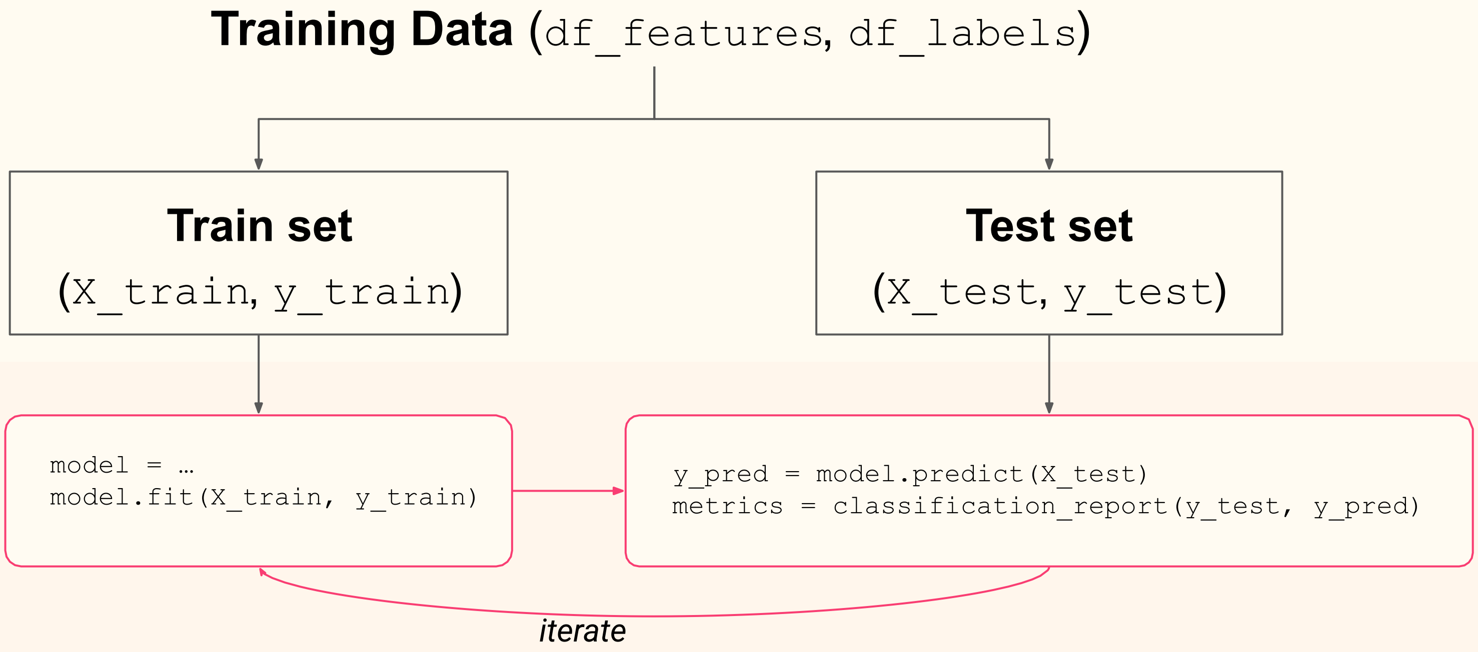
```
y_pred = model.predict(X_test)
```

Generate predictions with model on test features (X_test)

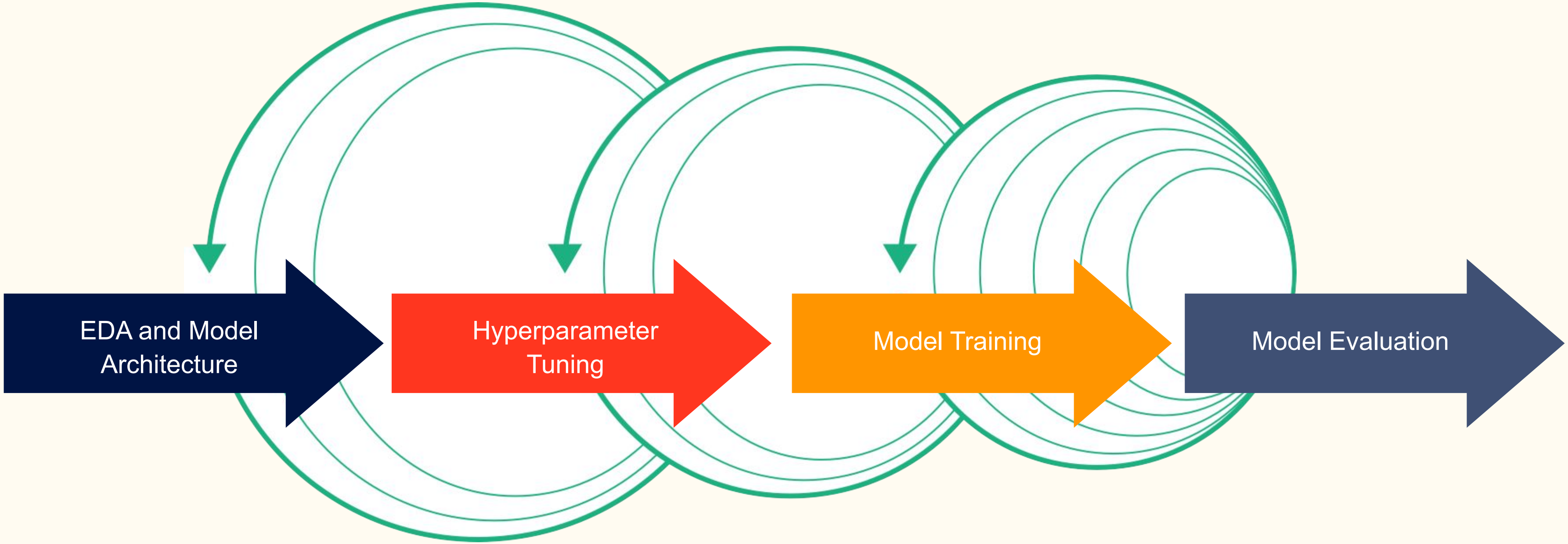
```
report_dict = classification_report(
                  y_test, y_pred, output_dict=True)
```

Evaluate model performance by comparing predictions (y_pred) and labels (y_test) for the test set

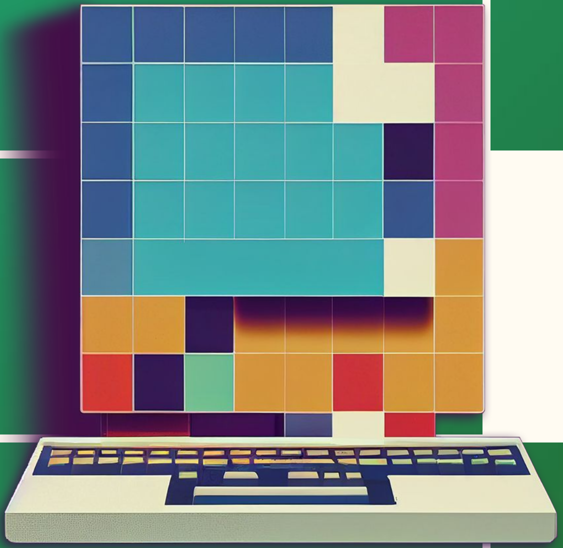
Iterative model improvement with train/test sets



Model training is an iterative process



Model training is out-of-scope for this course.
Here's 10 recommended free ML courses - take one if you are feel you need it!



ML Pipelines

ML Pipeline Stages

Collect raw data	Process data to create features	Store features	Use features for training and inference

- Discover data sources, securely connect to heterogeneous data sources
- Manage dependencies such as connectors and drivers
- Manage connection information securely: network endpoint, database/table names, authentication credentials such as API keys or credentials (username/password)

ML Pipeline Stages

Collect raw data

Process data to create features

Store features

Use features for training and inference

- Pipelines that extract, transform and load the data (ETL)
- Clean, validate, data to make it usable for creating features
- Data de-duplication, pseudononymization, data wrangling
- Feature extraction, aggregations, dimensionality reduction

ML Pipeline Stages

Collect raw data

Process data to create features

Store features

Use features for training and inference

- Need scalable storage that is securely accessible
- Use a feature store to store the features so that they can be reused across different models
- Use a feature store to store the features so that they can be used for both training and inference
- Use a feature store to compute statistics over the features for easy EDA

ML Pipeline Stages

Collect raw data

Process data to create features

Store features

Use features for training and inference

- Select features from the feature store to use to train a model
    - Filter features using time and/or filters (for example, training data for users located in the European Union)
- Select batches of feature data for inference (batch predictions) using the trained model
- The feature store provides low latency access to pre-computed features for online models (models that run 24x7)

There are many different types of ML pipelines

- End-to-End ML Pipelines
  - Go from raw data to models to predictions in a single program
- Feature Pipelines
  - Go from raw data to features in a single program
- Training Pipelines
  - Go from features to a trained model in a single program
- Batch Inference Pipelines
  - A batch program that takes unseen input features and produces predictions using a model
- Online Inference Pipelines
  - An online program that takes unseen input features and produces

There are many different types of ML pipelines

- End-to-End ML Pipelines
  - Go from raw data to models to predictions in a single program
- Feature Pipelines
  - Go from raw data to features in a single program
- Training Pipelines
  - Go from features to a trained model in a single program
- Batch Inference Pipelines
  - A batch program that takes unseen input features and produces predictions using a model
- Online Inference Pipelines
  - An online program that takes unseen input features and produces predictions using a model

There are many different types of ML pipelines

- End-to-End ML Pipelines
  - Go from raw data to models to predictions in a single program
- Feature Pipelines
  - Go from raw data to features in a single program
- Training Pipelines
  - Go from features to a trained model in a single program
- Batch Inference Pipelines
  - A batch program that takes unseen input features and produces predictions using a model
- Online Inference Pipelines
  - An online program that takes unseen input features and produces predictions using a model

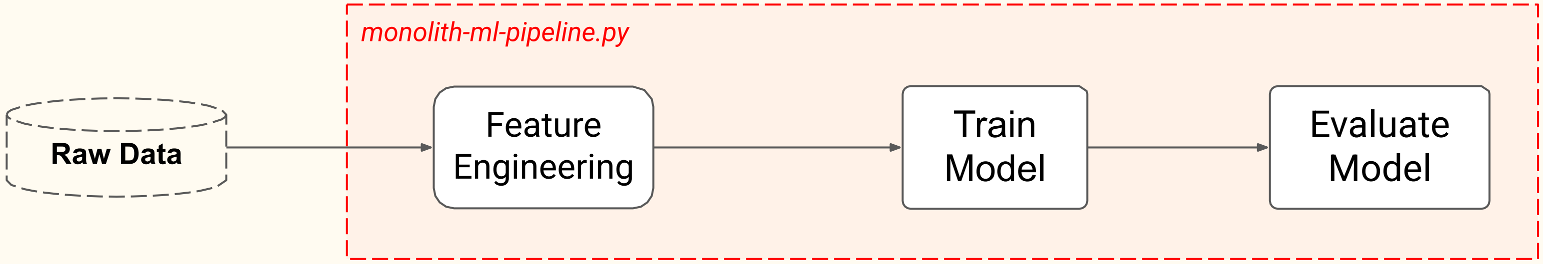
There are many different types of ML pipelines

- End-to-End ML Pipelines
  - Go from raw data to models to predictions in a single program
- Feature Pipelines
  - Go from raw data to features in a single program
- Training Pipelines
  - Go from features to a trained model in a single program
- Batch Inference Pipelines
  - A batch program that takes unseen input features and produces predictions using a model
- Online Inference Pipelines
  - An online program that takes unseen input features and produces predictions using a model

There are many different types of ML pipelines

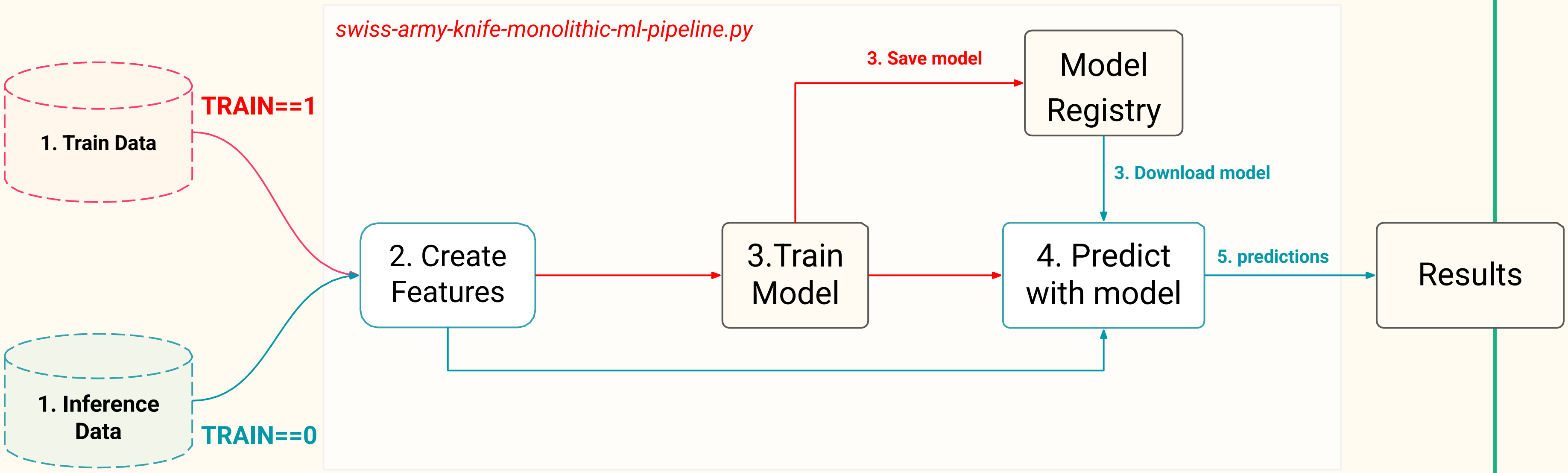
- End-to-End ML Pipelines
  - Go from raw data to models to predictions in a single program
- Feature Pipelines
  - Go from raw data to features in a single program
- Training Pipelines
  - Go from features to a trained model in a single program
- Batch Inference Pipelines
  - A batch program that takes unseen input features and produces predictions using a model
- Online Inference Pipelines
  - An online program that takes unseen input features and produces predictions using a model

- A pipeline is a program that takes an input and produces an output
- End-to-end ML Pipelines are a single pipeline that transforms raw data into features and trains and scores the model in one single program



ANTI-PATTERN! An end-to-end Batch Training/Inference 2-in-1 pipeline

TRAIN==0 is Inference; TRAIN==1 is Training



Problems with End-to-end ML Pipelines

- They are often not **modular** - their components (such as feature engineering) are not modular. Notebooks are often not modular.
- They are **difficult to test** - production software needs automated tests to ensure features and models are of high quality.
- They **tightly couple the execution of feature engineering, model training, and inference steps** - running them in the same pipeline program at the same time.
- They **do not promote reuse** of features/models/code. The code for computing features (feature logic) cannot be easily disentangled from its pipeline jungle.
- They are difficult to **document**.

Problems with End-to-end ML Pipelines

- They are often not **modular** - their components (such as feature engineering) are not modular. Notebooks are often not modular.

- They are **difficult to test** - production software needs automated tests to ensure features and models are of high quality.

- They **tightly couple the execution of feature engineering, model training, and inference steps** - running them in the same pipeline program at the same time.

- They **do not promote reuse** of features/models/code. The code for computing features (feature logic) cannot be easily disentangled from its pipeline jungle.

- They are difficult to **document**.

Problems with End-to-end ML Pipelines

- They are often not **modular** - their components (such as feature engineering) are not modular. Notebooks are often not modular.

- They are **difficult to test** - production software needs automated tests to ensure features and models are of high quality.

- They **tightly couple the execution of feature engineering, model training, and inference steps** - running them in the same pipeline program at the same time.

- They **do not promote reuse** of features/models/code. The code for computing features (feature logic) cannot be easily disentangled from its pipeline jungle.

- They are difficult to **document**.

Problems with End-to-end ML Pipelines

- They are often not **modular** - their components (such as feature engineering) are not modular. Notebooks are often not modular.
- They are **difficult to test** - production software needs automated tests to ensure features and models are of high quality.
- They **tightly couple the execution of feature engineering, model training, and inference steps** - running them in the same pipeline program at the same time.
- They **do not promote reuse** of features/models/code. The code for computing features (feature logic) cannot be easily disentangled from its pipeline jungle.
- They are difficult to **document**.

Problems with End-to-end ML Pipelines

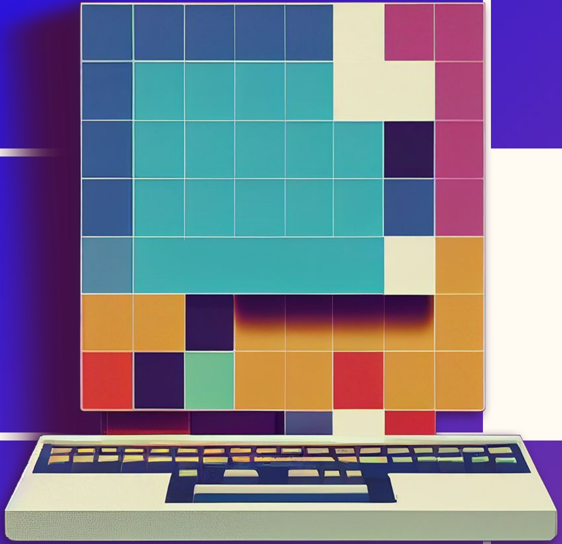
- They are often not **modular** - their components (such as feature engineering) are not modular. Notebooks are often not modular.

- They are **difficult to test** - production software needs automated tests to ensure features and models are of high quality.

- They **tightly couple the execution of feature engineering, model training, and inference steps** - running them in the same pipeline program at the same time.

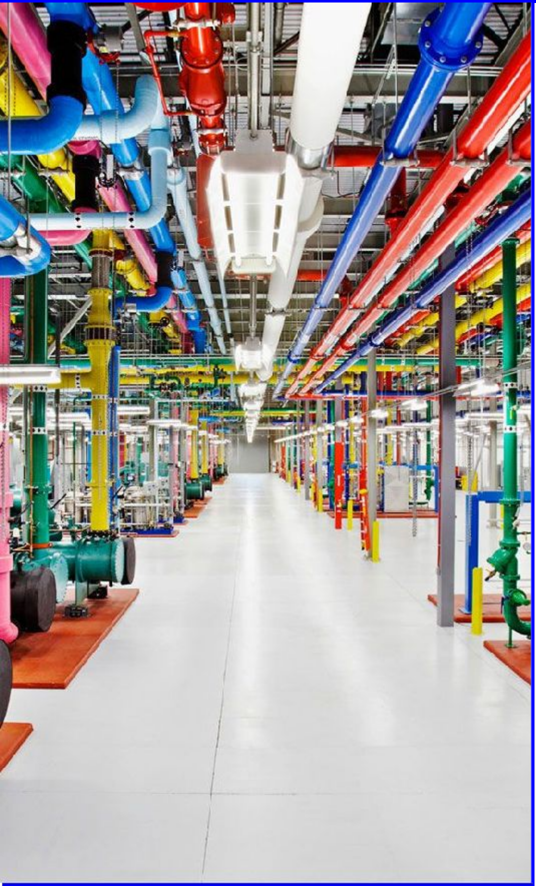
- They **do not promote reuse** of features/models/code. The code for computing features (feature logic) cannot be easily disentangled from its pipeline jungle.

- They are difficult to **document**.



Modular Pipelines for Machine Learning

Modular ML Pipelines



- **Modularity involves structuring your code** such that its functionality is separated into independent classes and/or functions that can be more easily reused and tested.
- **Modules should be placed in accessible classes or functions**, keeping them small and easy to understand and document.
- **Modules enable code to be more easily reused** in different pipelines.
- **Modules enable code to be more easily independently tested,** enabling the easier and earlier discovery of bugs.

*Modular **water pipes** in a Google Datacenter. Instead of one giant water pipe (our monolithic notebook), separate water pipes reduce the blast radius if one fails. Color coding makes it easier to debug problems in a damaged water pipe.*

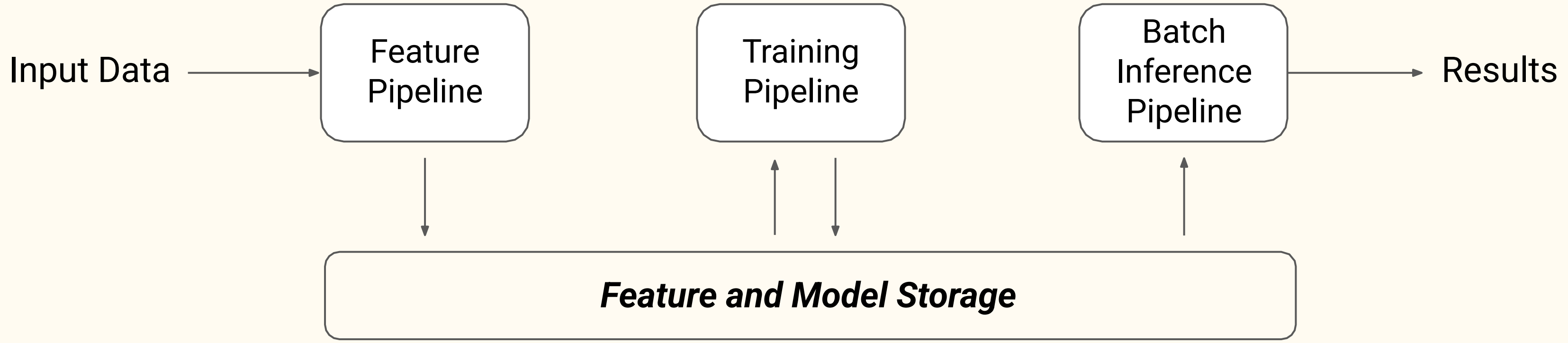
Feature, Training and Inference Pipelines

Refactor your monolithic end-to-end pipeline into

a. A **feature pipeline** to create features from new live data or to backfill features from historical data

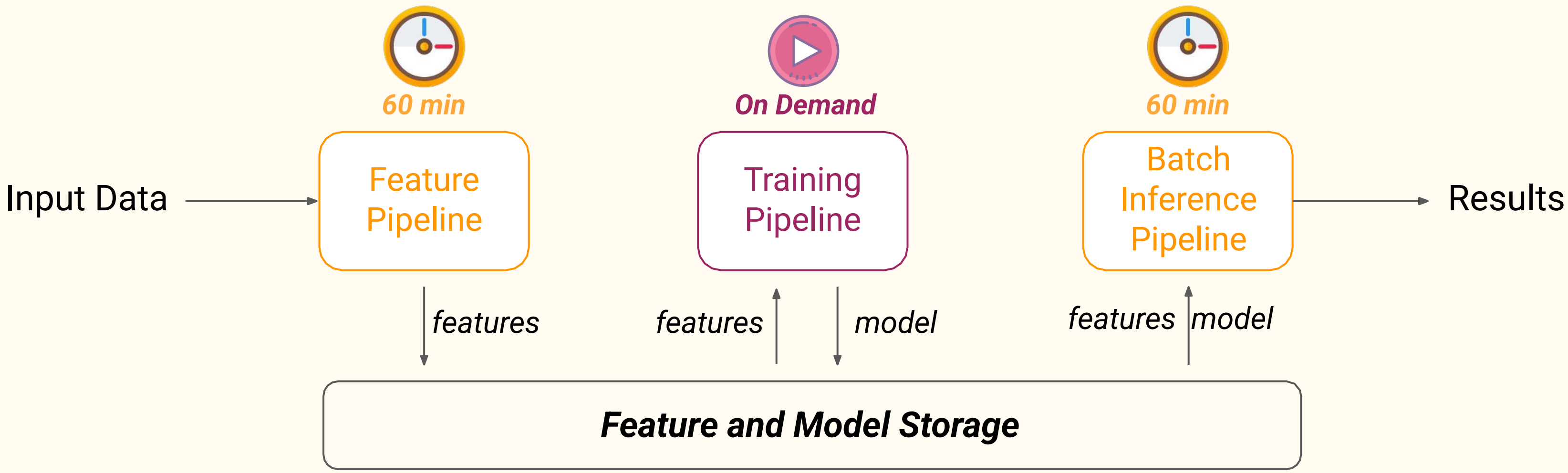
b. A **training pipeline** takes input features and produces a model as output

c. An **inference pipeline** (either batch or online) that takes unseen feature data and produces predictions



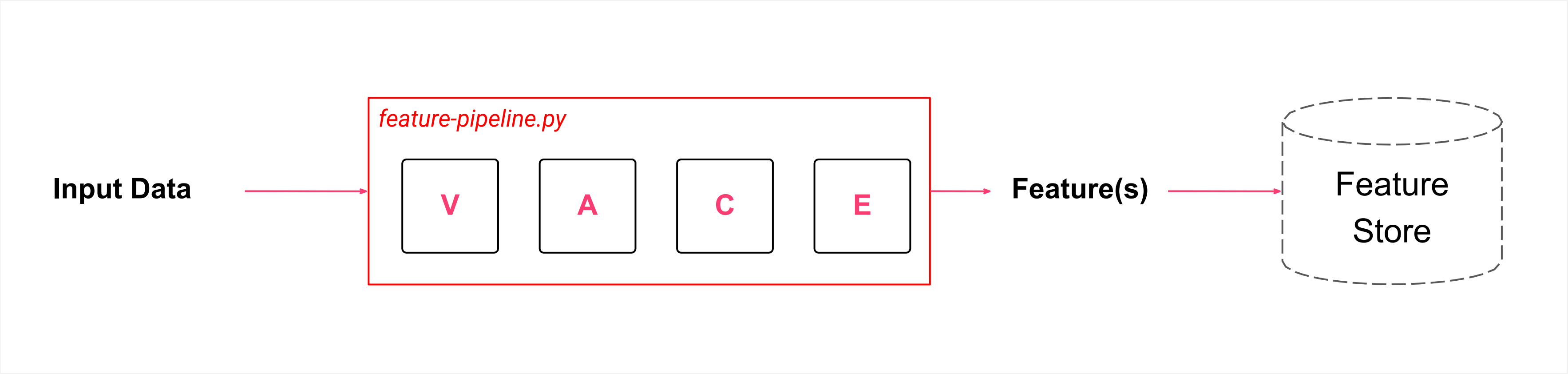
When to run our Pipelines

- **Feature pipelines** are run on a schedule or when new data is available
- **Training pipelines** are run when we need a new model (e.g., old model is stale)
- **Inference pipelines** are run when we need predictions on new data



Feature pipelines

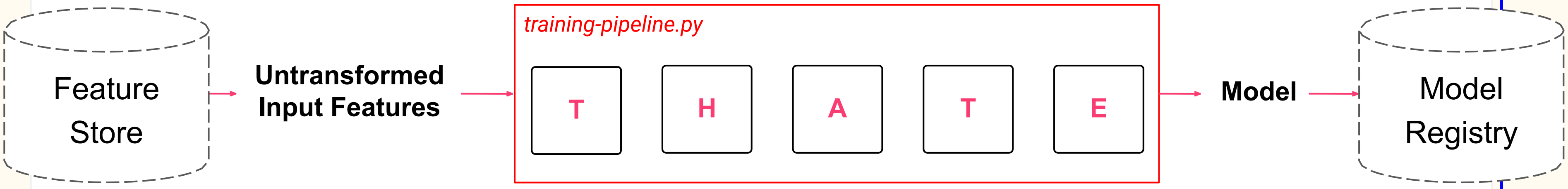
A **feature pipeline** is a program that orchestrates the execution of a dataflow graph of data validation, aggregation, dimensionality reduction, binning, crossing, and other feature engineering steps on input data to create and/or update feature values.



VAC = Validate, Aggregate, Compress (dimensionality reduction), Extract (Binning, Crosses, etc)

Training Pipeline

A **training pipeline** is a program that takes input features, optionally transforms them, and passes them to a model training function. The model training code can optionally include the definition of hyperparameters and a model architecture (e.g., for deep learning), and produces a model as output. The model is typically stored in a model registry.



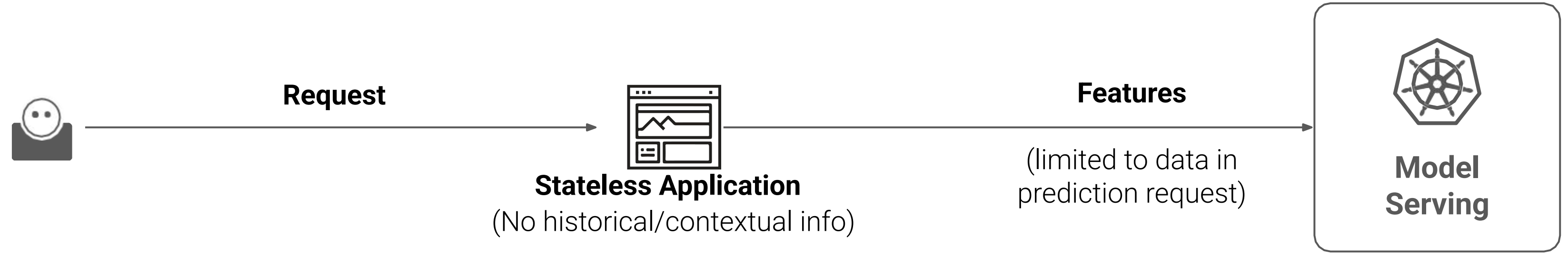
T-HATE =
Transform features, Hyperparameter tuning, model Architecture, Train model (fit to data), Evaluate your model.

Online Models - where do they get the features from?

A stateless ecommerce web application does not know the number of times you visited the website this month

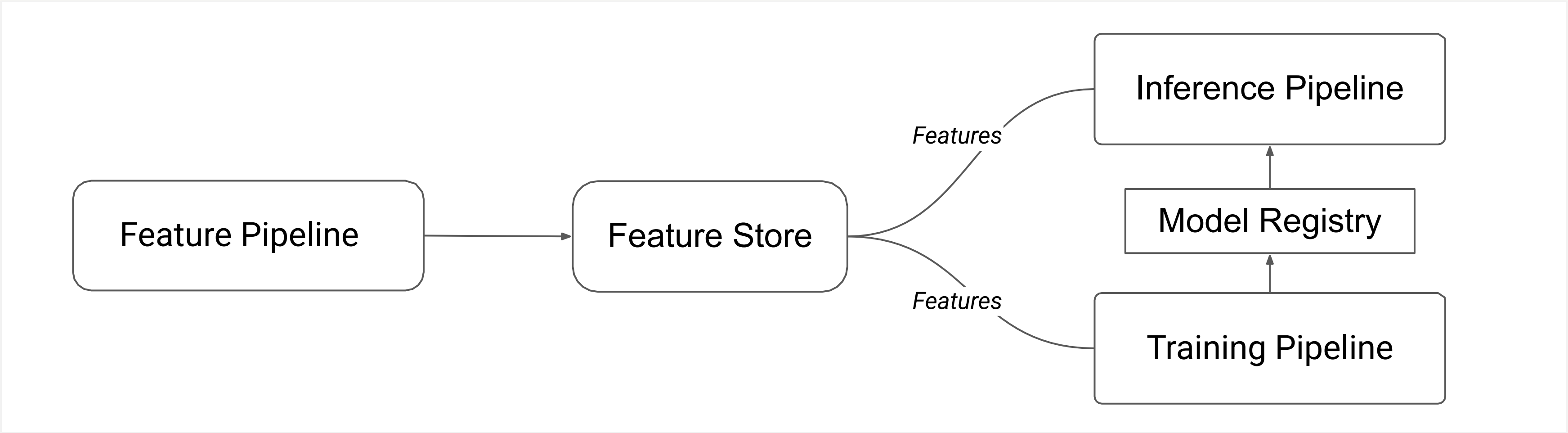
A credit card fraud identification service does not know your credit history, recent purchases, credit card details.

The feature store provides historical and contextual features as precomputed features for online models.



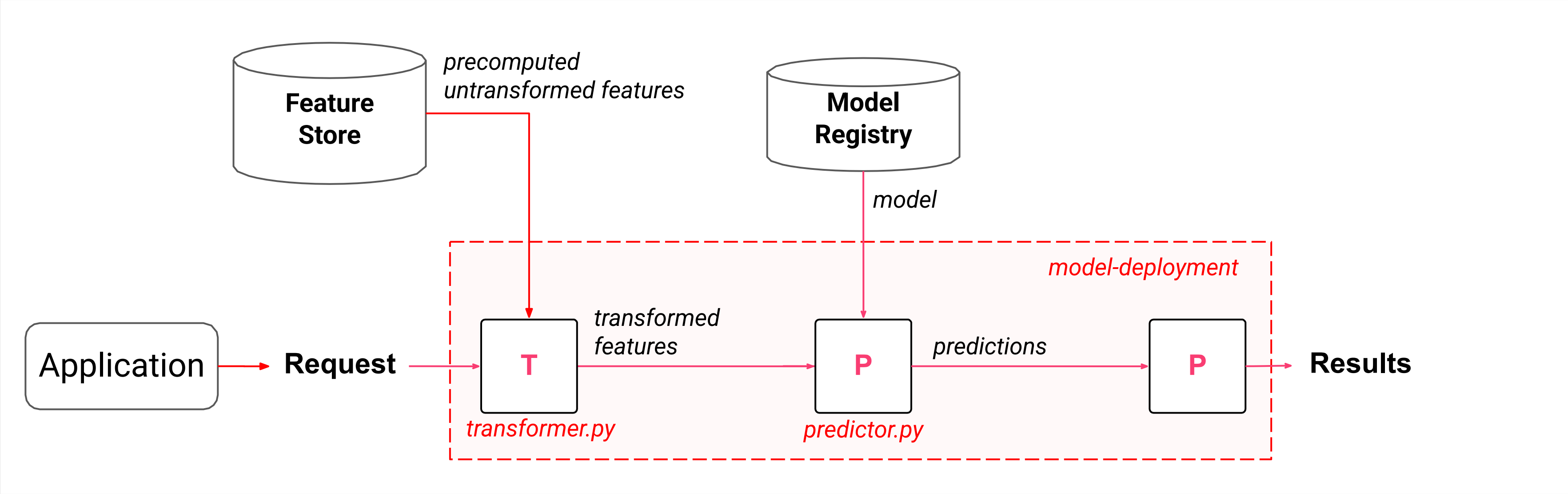
How can you use features that are known at training time but are unavailable or difficult to obtain once a trained model is deployed to production?

The Feature Store provides Online Historical and Contextual Features



No Training/Inference Skew, as features computed in same feature pipeline

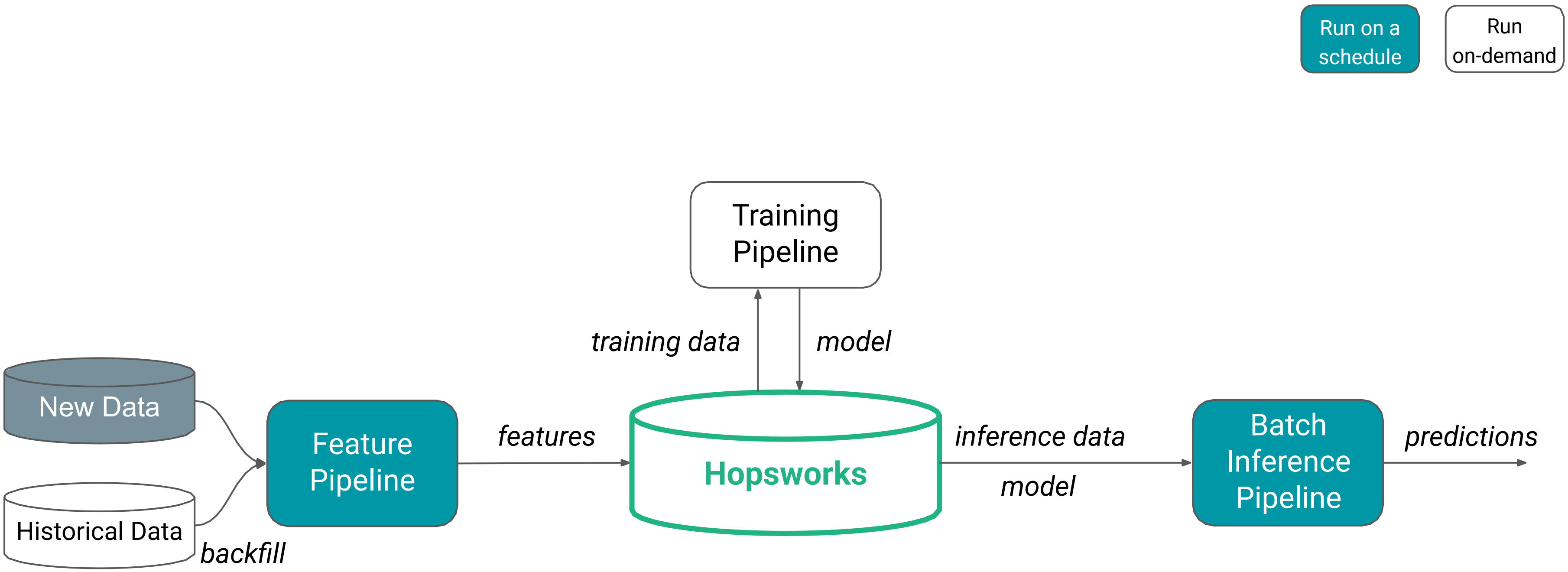
Online Inference Pipeline



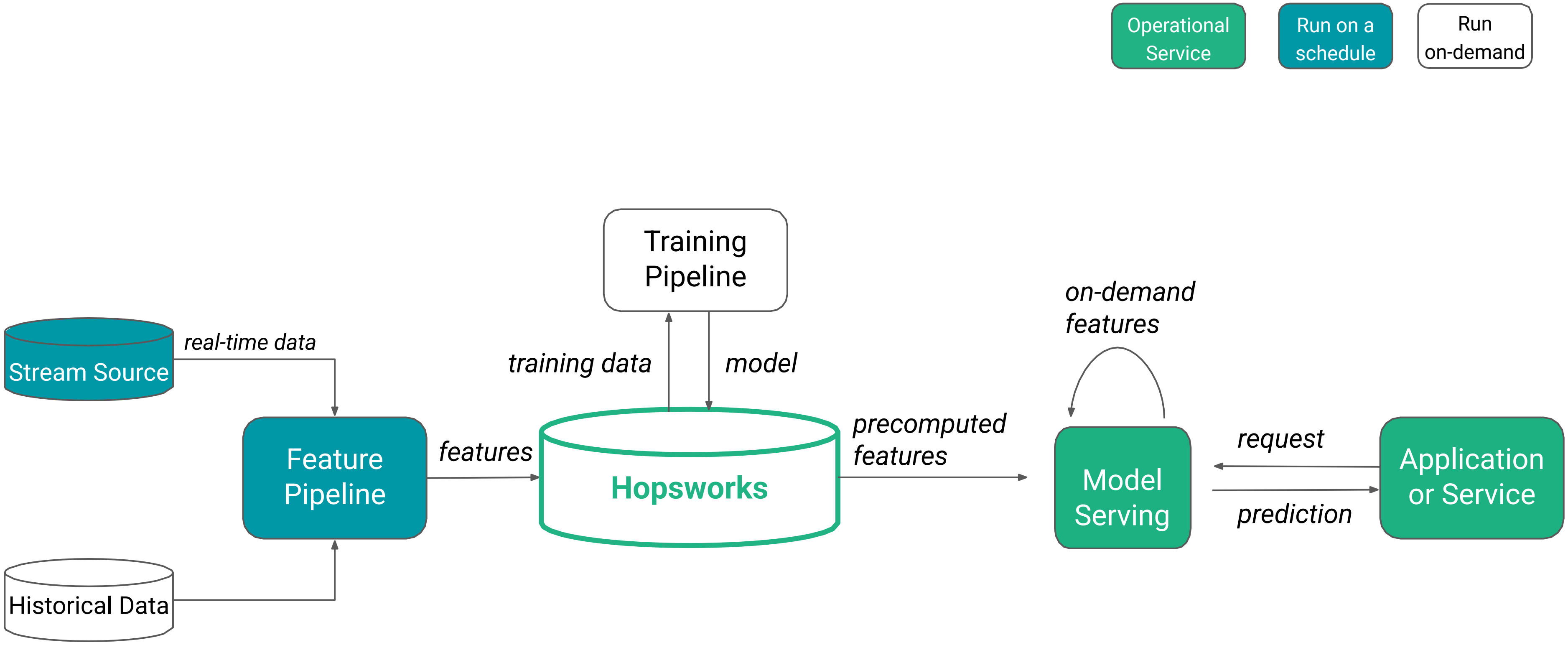
TPP = Transform the input request into features, Predict using input features and the model, Post-process predictions, before output results.

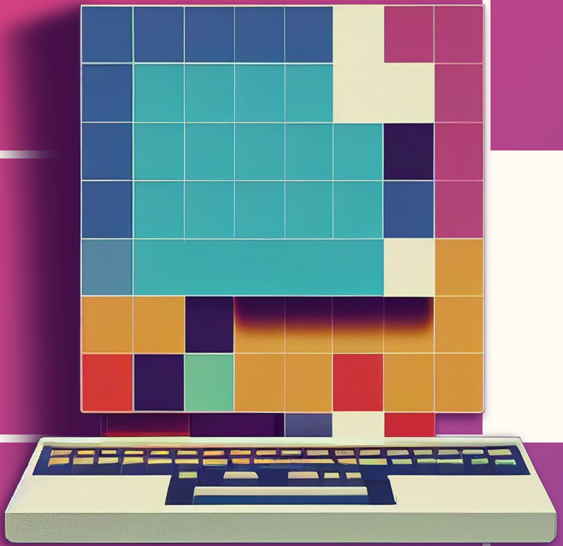
Analytical and Operational ML Systems with Modular Pipelines

An Analytical ML System with feature, training, and batch inference pipelines



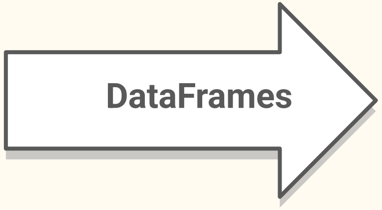
An Operational ML System with feature, training, and online inference pipelines

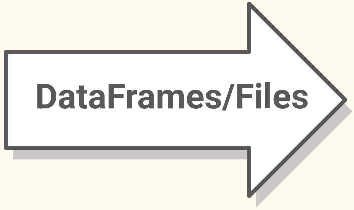




Hopsworks Feature store

Writing/Reading DataFrames to/from Hopsworks Feature Store

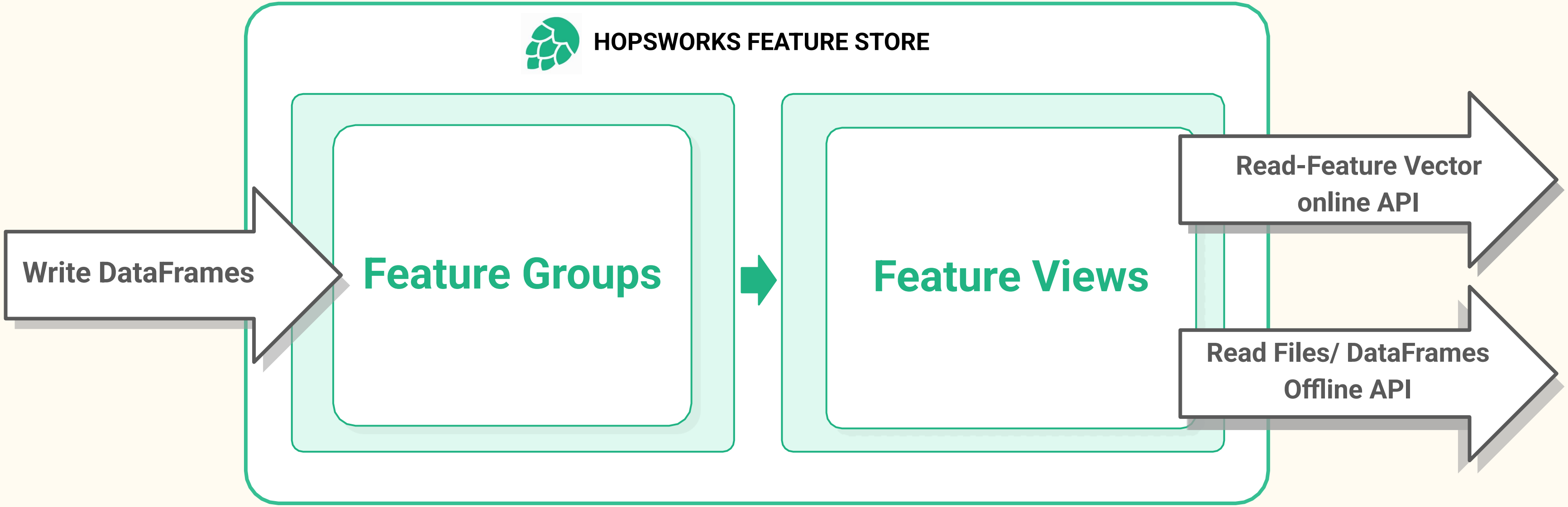




- Aggregations
- Dimensionality Reductions
- Validations

- Normalization
  One-hot encoding

Writing/Reading DataFrames to/from Hopsworks Feature Store



Create a Hopsworks Feature Group and write a Pandas DataFrame to it

```
import pandas as pd

data = {
    'credit_card_number': ['1111 2222 3333 4444', '1111 2222 3333 4444','1111 2222 3333 4444',
                           '1111 2222 3333 4444'],
    'trans_datetime': ['2022-01-01 08:44', '2022-01-01 19:44', '2022-01-01 20:44', '2022-01-01 20:55'],
    'amount': [142.34, 12.34, 66.29, 112.33],
    'location': ['Sao Paolo', 'Rio De Janeiro', 'Stockholm', 'Stockholm'],
    'fraud': [False, False, True, True]
}

df = pd.DataFrame.from_dict(data)
df['trans_datetime']= pd.to_datetime(df['trans_datetime'])
df
```

Create a DataFrame

```
fg = fs.get_or_create_feature_group(
        name="credit_card_transactions",
        version=1,
        description="Credit Card Transaction data",
        primary_key=['credit_card_number'],
        event_time='trans_datetime'
)
```

Create a Feature Group in Hopsworks. Note, the schema (columns) has not been defined yet.

```
fg.insert(df)
```

Write the DataFrame to the Feature Group. This call blocks until the DataFrame has been ingested.

The Feature Group takes its schema from the schema of the first DataFrame written to it.

Create a Hopsworks Feature View and read Training Data as DataFrames from it

```
query = fg.select(["amount", "location", "fraud"])
```

Select features from feature groups for your model

```
fv = fs.create_feature_view(name="credit_card_transactions",
                            version=1,
                            description="Features from the credit_card_transactions FG",
                            labels=["fraud"],
                            query=query)
```

Create a Feature View

```
X_train, X_test, y_train, y_test = fv.train_test_split(0.5)
X_train
```

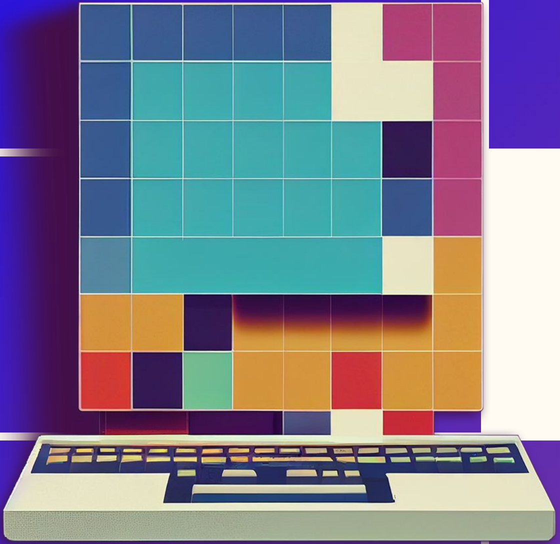
Read training data as Pandas DataFrames, randomly split into train/test sets of features and labels

	amount	location
0	66.29	Stockholm
4	84.00	San Francisco
5	183.00	Dublin
6	55.67	San Francisco

```
y_train
```

Let's look at the train set labels

	fraud
0	True
4	False
5	False
6	False



Pipeline Orchestration with Github Actions

Developing and Running Serverless Feature Pipelines

- **Python programs as Feature/Training/Inference Pipelines**
  - Feature and Inference pipelines are programs that are run on a schedule
    - For example, once per hour/day/week/month
  - Training pipelines are run on-demand (e.g., using Jupyter or Colab notebooks)

- **Github Actions**
  - orchestrate the execution of workflows (Jupyter Notebooks as Python programs)
  - Run your feature pipeline workflow “everyday at 04:00”

.github/workflows/my-workflow.yml

```
Steps:
install Python 3.8
pip install requirements.txt
cron schedule "11 11 * * * "
scripts/run-feature-pipe.sh
```

runs →

scripts/run-feature-pipe.sh

```
#!/bin/bash
jupyter nbconvert
--to notebook --execute
feature-pipeline.ipynb
```

runs →

feature-pipeline.ipynb

```
df = …//create features
write df to feature store
```

Lab 1:
Iris Flower Dataset