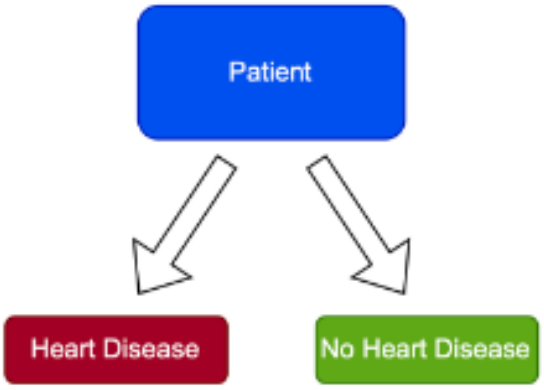
10.01 : End-to-End Machine Learning

```
### 이번 DataCamp 강의를 수강하면서 배운점
- 특히 docker의 개념을 아예 몰라서 애를 먹었던 경험이 있었는데, 이번 기회에 어떠한 느낌으로 작용하는지 알 수 있었다.
### 강의 내용중 궁금했던점
- 사실 EDA나 data전처리 부분은 예전의 기초수업에서도 들어본 적 있을 정도로 기초적인 내용인 것 같은데, 이정도의 기술만으로도 노하우를 발휘하면 충분히 business에서 정말 통하는지 궁금하다.
### 함께 이야기 나누며 얻은 것
-
```

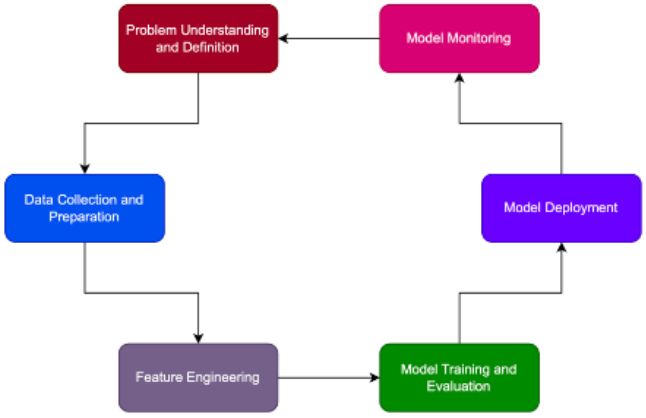
The model's role

- Models can **inform**, but should not **make** decisions



The machine learning lifecycle

The machine learning lifecycle



Case study : CardioCare clinic

- **End user**: CardioCare clinic.
- **Requirement**: A machine learning model that accurately and reliably predicts heart disease risk using patient health data.
- **Performance**: Must match or exceed a human cardiologist's expertise and generalize well to unseen data, providing timely predictions, even without engineer oversight.
- **Security**: The model design must ensure sensitive data is handled securely and privately.
- **Monitoring & Retraining**: Continuous monitoring with retraining as needed.
- **Interpretability**: The model should be interpretable, allowing cardiologists to understand and potentially override its predictions.

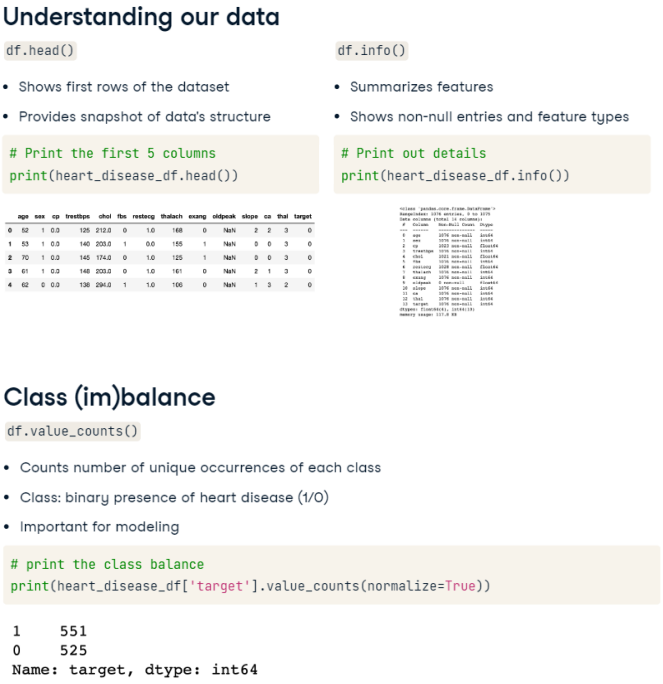
Data collection

- **Data collection**: Gather relevant patient health data (e.g., age, cholesterol, blood pressure).
- **Data sources**: Could include electronic health records or public health databases.
- **Understanding the data**: It’s crucial to grasp the context and potential biases.

- **Bias considerations**: Look for issues like error-prone self-reported measurements.
- These steps are key to ensuring the success of the machine learning project.

Exploratory Data Analysis

- **Exploratory Data Analysis (EDA)**: A process to examine and analyze the dataset for insights, patterns, and data characteristics.
- **Dataset focus**: Heart disease data from CardioCare.
- **Visualization**: Key aspects, such as the proportion of missing values, will be visualized.
- **Importance of EDA**: Critical for identifying issues and ensuring model performance isn't affected downstream(후속단계).



Missing values

- Can lead to errors
- Unrepresentative, biased results

Use `df.isnull()`

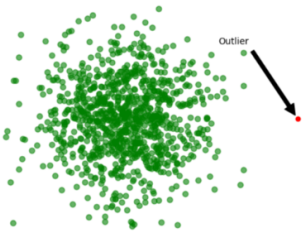
- Checks for null/empty/missing values
- Applied to column or collection of columns

Usage

```
# check whether all values in a column are null
print(heart_disease_df['oldpeak'].isnull().all())
```

Outliers

- Anomalous values
  - Measurement errors
  - Data entry errors
  - Rare events
- Can skew model performance
  - Model learns based on extreme values
  - Doesn't capture general data trend
- Sometimes can be useful:
  - Rare values
  - Detection: use boxplot, or IQR



Goals of EDA

- Understand the data and uncover patterns (e.g., do men have higher heart disease rates than women?).
- **Outlier detection**: Identify data points that fall outside acceptable ranges.
- **Hypothesis design**: Validate and check assumptions (e.g., does reality match expectations?).
- **Influence on ML**: EDA informs the choice of ML algorithms, feature selection, and the need for feature engineering, which are critical for project success.

Data preparation

Identifying and carrying out the data-cleaning steps derived from EDA.

Data preparation steps

Dataset has:

- Missing values
- Outliers
- Imbalances
- Empty columns
- Duplicates

Data preparation:

- Based on insights from EDA
- Critical for model performance downstream

Null / empty values

- Drop missing or sparse rows/columns
- Null values can break model
- Use `df.drop()` for columns
- Use `df.dropna(how='all')` for rows

```
# count missing values
print(df['oldpeak'].isnull().sum())

# Drop empty column(s) and row(s)
columns_dropped = heart_disease_df.drop(['oldpeak'], axis='columns')
rows_and_columns_dropped = columns_dropped.dropna(how='all')
```

Dealing with null / empty values

- Data cleaning / dropping values depends on EDA findings

- If given column has too many missing values:
  - Drop column

- If target column has missing values:
  - Drop rows with missing targets
  - Or treat as separate category

Imputation

What to do when there are only a few missing values?

- Imputation:
  - Fill missing values with substitutes
- Strategies
  - Fill with mean or median
  - Use constant or previous value

```
# Calculate the mean cholestrol value
mean_value = heart_disease_df['chol'].mean()

# Fill missing cholestrol values with the mean
heart_disease_df['chol'].fillna(mean_value, inplace=True)
```

Advanced imputation

Advanced techniques:

- K-nearest neighbors
- SMOTE (synthetic minority oversampling technique)

```
from sklearn.impute import KNNImputer

# Initialize KNNImputer
imputer = KNNImputer(n_neighbors=2, weights="uniform")

# Perform the imputation on your DataFrame
df_imputed['oldpeak'] = imputer.fit_transform(df['oldpeak'])
```

Dropping duplicates

- Data must be clean, concise, and rich
- Redundancies are unhelpful
- Duplicates can bias or confuse model
- Look at unique identifiers as a criteria for dropping records / rows.

```
# Drop duplicate rows
heart_disease_duplicates_dropped = heart_disease_column_dropped.drop_duplicates()
```

Feature engineering and selection

Feature engineering

Creating features

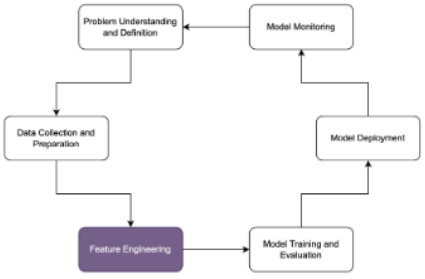
- Simplifies problem
- Improves model efficiency

Techniques

- Modify pre-existing features
- Design new features

Benefits

- Easier deployment, maintenance, training
- Interpretability gain



One of common technique is Normalization, Standaraization

Normalization

- Scales numeric features to [0, 1]
- Helpful when features have different scales/ranges.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import Normalizer

# Split the data
X_train, X_test = train_test_split(df, test_size=0.2, random_state=42)
# Createnormalizer object, fit on training data, normalize, and transform test set
norm = Normalizer()
X_train_norm = norm.fit_transform(X_train)
X_test_norm = norm.transform(X_test)
```

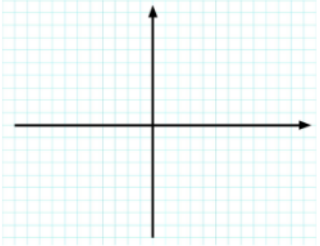
Standardization

- Scales data to have mean = 0, variance = 1
- Beneficial for algorithms that assume similar mean and variance

```
from sklearn.preprocessing import StandardScaler

# Split the data
X_train, X_test = train_test_split(df, test_size=0.2, random_state=42)
# Create a scaler object and fit training data to standardize it
sc = StandardScaler()
X_train_stzd = sc.fit_transform(X_train)
# Only standardize the test data
X_test_stzd = sc.transform(X_test)
```

What constitutes a good feature?

- Use relevant features
- Weather on the day of patient appointment should have no bearing on diagnosis
- Use dissimilar (orthogonal) features
- Two features of age in months and age in years would not be helpful



sklearn.feature_selection

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectFromModel
from sklearn.model_selection import train_test_split

# Splitting data into train and test subsets first to avoid data leakage
X_train, X_test, y_train, y_test = train_test_split(
    heart_disease_df_X, heart_disease_df_y, test_size=0.2, random_state=42)
```

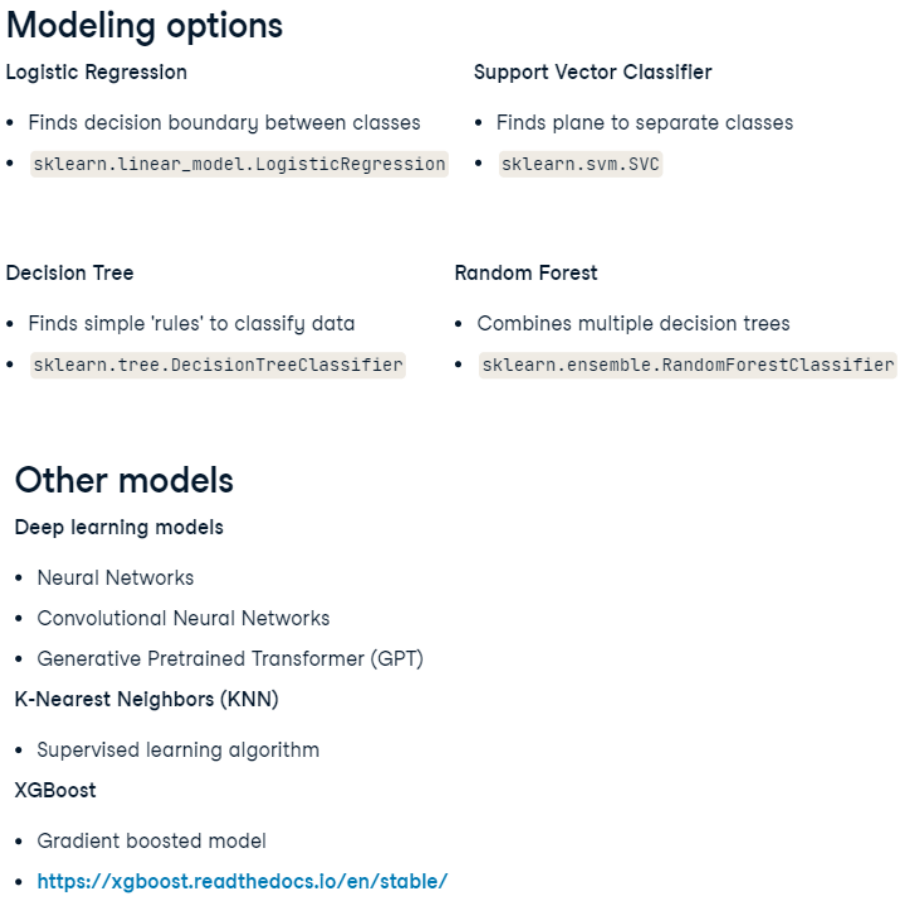
sklearn.feature_selection (cont.)

```
# Define and fit the random forest model
rf = RandomForestClassifier(n_jobs=-1, class_weight='balanced', max_depth=5)
rf.fit(X_train, y_train)

# Define and run feature selection
model = SelectFromModel(rf, prefit=True)
features_bool = model.get_support()
features = heart_disease_df.columns[features_bool]
```

- **SelectFromModel**: A method in `sklearn.feature_selection` for feature selection.
- **Random Forest classifier**: Used to estimate feature importance by eliminating irrelevant features.
- **Prefit=True**: Indicates that the model has already been trained.
- **Random Forest parameters**:
  - `n_jobs=-1`: Utilizes all available processors.

    - `class_weight='balanced'`: Balances class frequencies.
    - `max_depth=5`: Limits the depth of the trees.
- **Dot-fit function**: Used to fit the model and identify critical features based on feature `X` predicting target `Y`.
- **Model.get_support()**: Returns a Boolean array indicating which features are important (True) or not (False).

Modeling options



Training a model

```
# Importing necessary libraries
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Split the data into training and testing sets (80:20)
X_train, X_test, y_train, y_test = train_test_split(features, heart_disease_y,
    test_size=0.2, random_state=42)

# Define the models
logistic_model = LogisticRegression(max_iter=200)

# Train the model
logistic_model.fit(X_train, y_train)
```

Getting model predictions

```
# Jane Doe's health data, for example: [age, cholesterol level, blood pressure, etc.]
jane_doe_data = [45, 230, 120, ...]

# Reshape the data to 2D, because scikit-learn expects a 2D array-like input
jane_doe_data = jane_doe_data.reshape(1, -1)

# Use the model to predict Jane's heart disease diagnosis probabilities
jane_doe_probabilities = logistic_model.predict_proba(jane_doe_data)
jane_doe_prediction = logistic_model.predict(jane_doe_data)
```

Getting model predictions (cont.)

```
# Print the probabilities
print(f"Jane Doe's predicted probabilities: {jane_doe_probabilities[0]}")
print(f"Jane Doe's predicted health condition: {jane_doe_prediction[0]}")
```

```
Jane Doe's predicted health condition probabilities: [0.2 0.8]
```

Logging experiments on MLFlow

- **Experiments**: Common practice in ML to test and validate hypotheses, involving tweaking parameters, feature selection, or algorithms.
- **Challenge**: Keeping track of experiments can become disorganized and unreproducible, especially in clinical settings where reproducibility is crucial.
- **MLflow**: An open-source platform that manages the ML lifecycle, helping track and compare experiment results, package code for reproducibility, and share and deploy models.

MLFlow

Without MLflow...

- Many untracked, disorganized experiment runs
- Dissimilar, or incomparable runs
- Unreproducible, lost runs

With MLflow...

- Tracked, organized experiment runs
- Comparison between standardized runs
- Reproducible runs
- Share, deploy models

Creating experiments

mlflow.set_experiment()

- Sets experiment name
- Provides workspace for experiment runs

Usage:

```
import mlflow

# Set an experiment name, which is a workspace for your runs
mlflow.set_experiment("Heart Disease Classification")
```

Running experiments

```
# Start a new run in this experiment
with mlflow.start_run():
    # Train a model, get the prediction accuracy
    logistic_model = LogisticRegression()
    # Log parameters, eg:
    mlflow.log_param("n_estimators", logistic_model.n_estimators)
    # Log metrics (accuracy in this case)
    mlflow.log_metric("accuracy", logistic_model.accuracy)
    # Print out metrics
    print("Model accuracy: %.3f" % accuracy)
```

```
Model accuracy: 0.96
```

Retrieving experiments

`mlflow.get_run(run_id)`

- Metadata for specific run

`mlflow.search_runs()`

- Returns DataFrame of metrics for multiple runs

Usage:

```
# Fetch the run data and print params
run_data = mlflow.get_run(run_id)
print(run_data.data.params)
print(run_data.data.metrics)

# Search all runs in experiment
exp_id = run_data.info.experiment_id
runs_df = mlflow.search_runs(exp_id)
```

```
{'epochs': '20', 'accuracy': 0.95}
```

MLFlow UI

- **MLflow's power**: Acts as a central hub for managing ML experiments.
- **Workflow benefits**: Organizes and streamlines the ML workflow, making it more manageable and effective.
- **Tracking experiments**: Helps identify successful model configurations, making it easier to build on them and improve.

Model evaluation and visualization

Accuracy

- Correct accuracy metrics are vital to robust model evaluation
- Easy to misinterpret or obscure results

Standard accuracy:

- Standard accuracy = num correct answers / num answers
- Standard accuracy can be unhelpful

Example:

```
# achieves ~99% accuracy for imbalanced dataset of 99 positive and 1 negative
for patient_datapoint in heart_disease_dataset:
    model.prediction(patient_datapoint) = 'positive'
```

Confusion matrix

True positives (TP)

- Model prediction = actual classification = positive
- The model predicted heart disease, the patient had heart disease

False positives (FP)

- Model prediction = positive, actual classification = negative
- The model predicted heart disease, the patient did not have heart disease

False negatives (FN)

- Model prediction = negative, actual classification = positive
- The model predicted no heart disease, the patient had heart disease

True negatives (TN)

- Model prediction = actual classification = negative
- The model predicted no heart disease, the patient did not have heart disease

Balanced accuracy

- Better metric than plain accuracy for most binary classification models
- Provides weighted average across both classes
- Balanced accuracy = (TP + TN) / 2

```python
from sklearn.metrics import balanced_accuracy_score

# Assume y_test is the true labels and y_pred are the predicted labels
y_pred = model.predict(X_test)
bal_accuracy = balanced_accuracy_score(y_test, y_pred)
print(f"Balanced Accuracy: {bal_accuracy:.2f}")
```

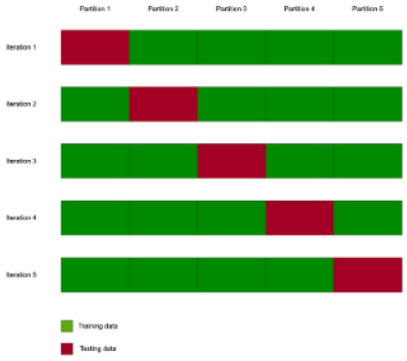
Cross validation

Cross-validation

- Resampling procedure
- Ensures robustness of results

k-fold cross-validation

- Param 'k' = number of splits for dataset
- Resample new train/test split for each modeling run



- **Cross-validation**: A method to estimate model performance by averaging scores across different dataset splits, preventing dependency on a single split.
- **k-fold cross-validation**: A resampling procedure that splits data into *k* groups.
- **Parameter k**: Refers to the number of groups the data will be divided into.
- **Small dataset**: k-fold cross-validation is ideal for smaller datasets, like the heart disease dataset.
- **k=5 example**: The data is split into five equal parts, with four parts used for training and one part for testing in each iteration.

Cross validation usage

- Straightforward implementation of k-fold cross validation using sklearn
- Model-agnostic scoring

Usage:

```
from sklearn.model_selection import cross_val_score, KFold

# split the data into 10 equal parts
kfold = KFold(n_splits=5, shuffle=True, random_state=42)

# get the cross validation accuracy for a given model
cv_results = cross_val_score(model, heart_disease_X,
heart_disease_y, cv=kfold, scoring='balanced_accuracy')
```

Hyperparameter tuning

Hyperparameter:

- Global model parameter (doesn't change during training)
- Adjust to improve model performance

```
# Hyperparameters to test
C_values = [0.001, 0.01, 0.1, 1, 10, 100, 1000]

# Manually iterate over the hyperparameters
for C in C_values:
    model = LogisticRegression(max_iter=200, C=C)
    model.fit(X_train, y_train)
    accuracy = cross_val_score(model, X, y, cv=kfold, scoring='balanced_accuracy')
    print(f"C = {C}: Bal Acc: {accuracy.mean(): 4f} (+/- {accuracy.std(): 4f})")
```

Hyperparameter tuning example

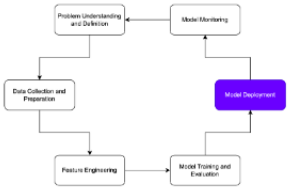
Example output for hyperparameter tuning:

```
C = 0.001: Bal Acc: 0.6200 (+/- 0.0215)
C = 0.01: Bal Acc: 0.7325 (+/- 0.0234)
C = 0.1: Bal Acc: 0.7923 (+/- 0.0202)
C = 1: Bal Acc: 0.8050 (+/- 0.0191)
C = 10: Bal Acc: 0.8034 (+/- 0.0185)
C = 100: Bal Acc: 0.8021 (+/- 0.0187)
C = 1000: Bal Acc: 0.8017 (+/- 0.0188)
```

Testing a model

Testing

- Testing:
  - Model does not crash
  - Returning reasonable outputs at inference time



- **Pre-deployment testing**: Write tests to flag anomalous or unexpected events, ensuring the model performs as expected before clinical use.
- **Automation**: These tests can run automatically during deployment.
- **Unittest library**: Python's built-in `unittest` library is used to write unit tests for the system and model.
- **Test cases**: Define specific types of tests (e.g., inference tests).
- **Test methods**: Include tests for scenarios like model failures.

Unittest usage

```python
import unittest

class TestModelInference(unittest.TestCase):
    def setUp(self):
        self.model = fitted_model
        self.X_test = X_test
    def test_prediction_output_shape(self):
        y_pred = self.model.predict(self.X_test)
        self.assertEqual(y_pred.shape[0], self.X_test.shape[0])

if __name__ == '__main__':
    unittest.main()
```

Unittest usage (cont.)

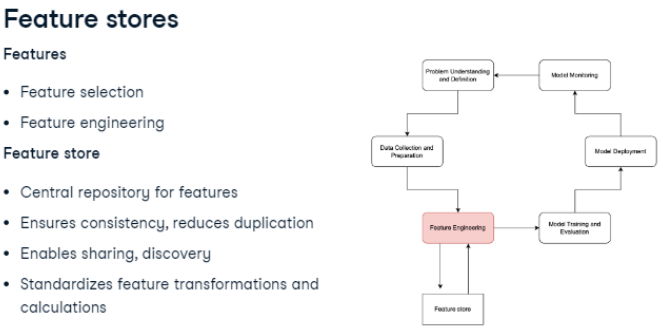
```python
    def test_input_values(self):
        print("Running test_input_values test case")

        # Get inputs (each row in testing set)
        for input in X_test:
            for value in input:
                # if value is cholestrol, for example:
                self.assertIn(value, [0, 500])
```

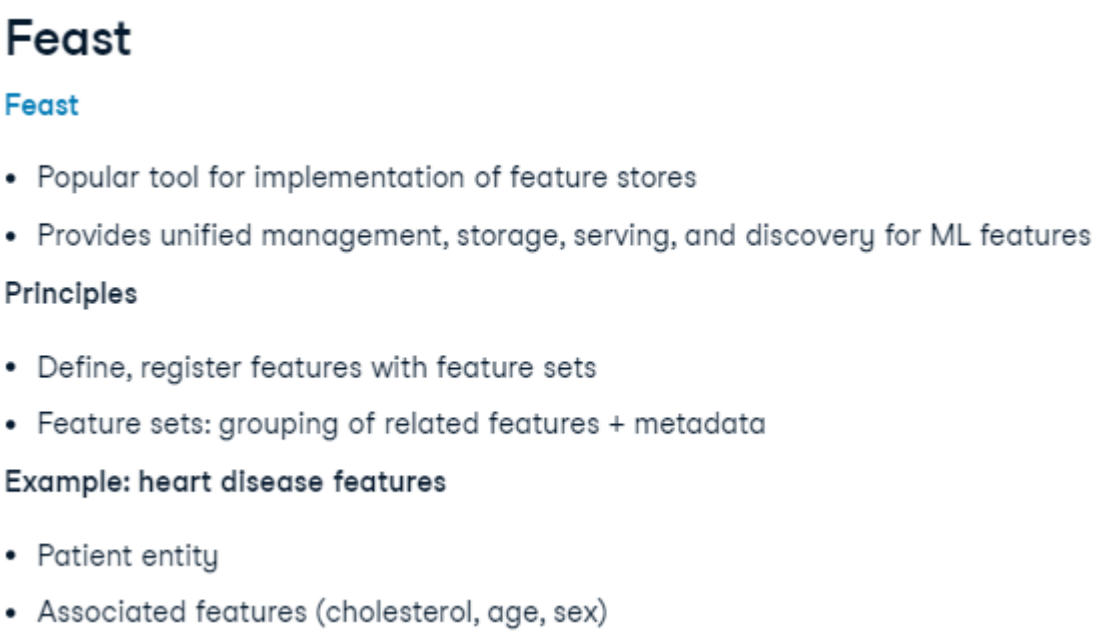
Testing do's and dont's

Best-practices

- DON'T...
  - Write too many tests
  - Write redundant tests
  - Write tests for highly reliable components.
- DO...
  - Write tests to increase reliability.
  - Write tests to check/manage expectations.
  - Write tests for new functionality.

Architectural components in end-to-end machine learning frameworks





- **Feast platform**: An open-source tool for implementing feature stores, developed by Gojek.
- **Purpose**: Provides a unified way to manage and store ML features.
- **Feature set**: A grouping of related features and their metadata.
- **Example**: In the heart disease dataset, a **patient entity** could have related features such as cholesterol, age, and sex.
- **Patient entity**: Represents a patient in the CardioCare clinic with various attributes or features.

Feast feature stores part 1

```
from feast import Field, Entity, ValueType, FeatureStore
from feast.data_source import FileSource

# Define the entity, which in this case is a patient, and features
patient = Entity(name="patient", join_keys=["patient_id"])
chol = Field(name="chol", dtype=Float32)
age = Field(name="age", dtype=Int32)
...
# Define the data source
data_source = FileSource(
    path="/path_to_heart_disease_dataset.csv",
    event_timestamp_column="event_timestamp",
    created_timestamp_column="created")
```

```
# Create a feature view of the data
heart_disease_fv = FeatureView(name="heart_disease", entities=[patient],
    schema=[cholesterol, ...], ttl=timedelta(days=1), input=data_source,)

# Create a FeatureStore object
store = FeatureStore(repo_path=".")

# Register the FeatureView
store.apply([patient, heart_disease_fv])
```

Model registries

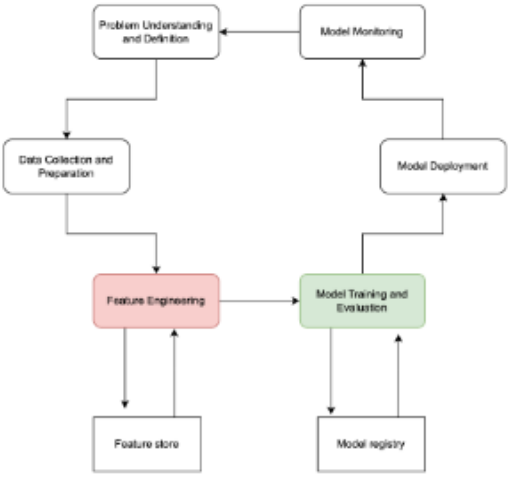
Model registries

Model registry

- Version control systems
- Keep track of different versions of model
- Annotate models
- Track performance over time

Benefits

- Organization
- Transparency
- Reproducibility



- **Model outputs**: Just as managing features is important, managing and storing model outputs is critical for timely, accurate, and persistent patient diagnoses.
- **Model registries**: Act as version control systems for ML models, helping track and manage different model versions.
- **Features of model registries**:
  - Annotate models with metadata.
  - Compare different models and track performance over time.
- **Benefits**: Improves organization, transparency, and reproducibility in ML workflows.
- **MLflow**: A model registry example that tracks ML experiments, logs performance metrics, and stores trained model artifacts for comparison.

Packaging and containerization

- **Deployment phase**: Involves packaging the model and its dependencies into a standalone unit for easy execution in various environments.
- **Containerization**: This practice allows the model to run consistently across different setups.
- **Docker**: A platform that simplifies creating and deploying applications using containers.
- **Containers**: Package an application along with its libraries and dependencies into one unit.
- **Platform-agnostic**: Containers run consistently on any machine, regardless of its settings or environment.

Docker usage

Docker usage part 1

Dockerfile: instructions for building container

```
# Use an official Python runtime as a parent image
FROM Python:3.7

# Set the working directory in the container to /app
WORKDIR /ML_pipeline

# Copy the current directory contents into the container at /app
ADD . /ML_pipeline

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
```

```
# ... continued
# Make port 80 available to the world outside this container
EXPOSE 80
```

```
# Define environment variable
ENV NAME World
```

```
# Run app.py when the container launches
CMD ["Python", "ML_pipeline.py"]
```

Build the defined image:

```
docker build -t heart_disease_model .
```

- **Dockerfile**: A text document containing commands to create a Docker image, like a recipe for the application.
- **Docker build**: A command that automates the process of building an image by running the commands in the Dockerfile.
- **Base image**: The starting point, a read-only template, for creating the container. Example: Python 3.7 base image for Python applications.
- **Steps in Dockerfile**:
  - Specify the **base image**.
  - Copy necessary files (e.g., Python script, `requirements.txt`) into the Docker image.

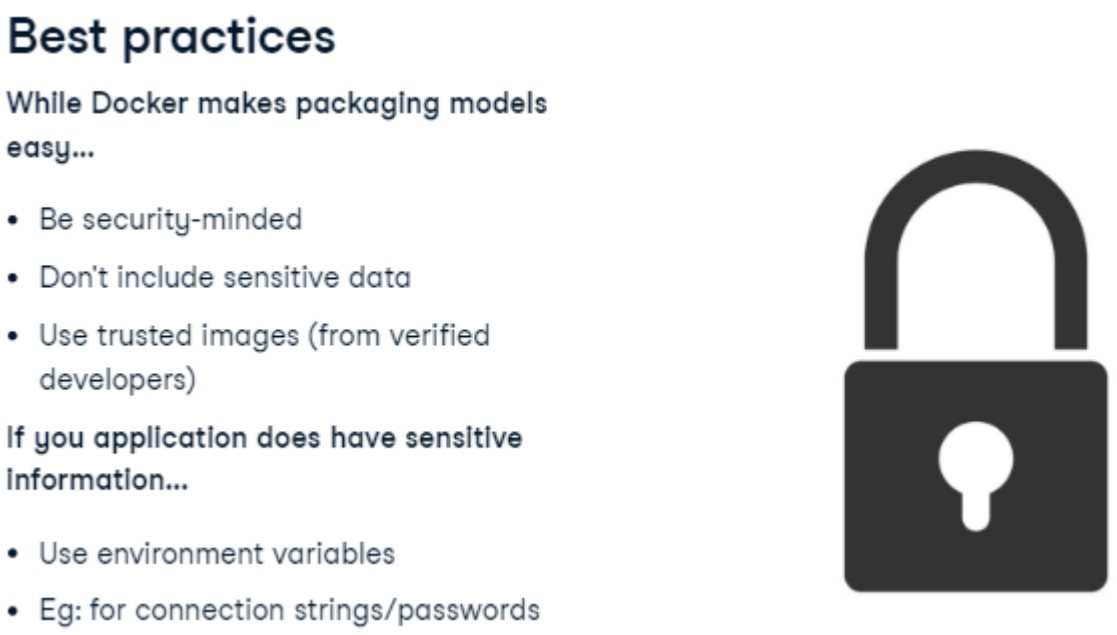
    - Install dependencies by running `pip install` for the listed packages in `requirements.txt`.
- **Port specification**: Define the port the container will run on.
- **Environment variables**: Set any necessary or sensitive information (e.g., API keys, database credentials).
- **Run command**: Specify the command to be executed when the container is launched from the Docker image.
- **Building the Docker image**:
    - Run `docker build -t heart_disease_model .` in the command line.
    - **heart_disease_model** is the image name.
    - The dot (`.`) tells Docker to search the current directory for the Dockerfile.
- Make sure to run the command in the root directory containing the **ML_pipeline.py** script and Dockerfile.

Tagging containers

Tagging:

```
docker tag heart_disease_model:latest heart_disease_model:1.0
```

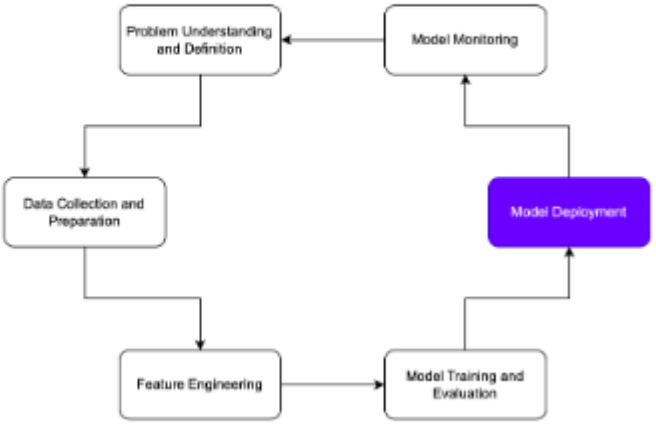
- Makes images / containers easier to identify and manage.
- Helps in maintaining a detailed and robust model registry.
- After tagging, we are ready to deploy!



- **Security considerations**: Crucial when using Docker, especially in healthcare settings.
- **Avoid sensitive data**: Do not include sensitive data, such as patient information, in Docker images.
- **Use trusted base images**: Only rely on verified and secure base images to prevent vulnerabilities.
- **Environment variables**: Define sensitive information like connection strings or passwords via environment variables in the Dockerfile to keep them secure.

Continuous integration and continuous deployment (CI/CD)

CI/CD in the ML lifecycle



CI/CD principles

Continuous Integration(CI)

- Regularly merging to central repository
- Often involves automatic testing for identifying bugs

Continuous Deployment (CD)

- Automatically deploying updates in codebase to production
- Often combined with CI

- **Continuous Integration (CI)**: Regularly merges code changes into a central repository, often paired with automated testing to catch bugs early.
- **DevOps**: Streamlines the production process and integrates with CI for efficiency.
- **Continuous Deployment (CD)**: Automatically deploys updates or changes to production following CI.
- **Combined benefits**: CI/CD practices accelerate development, reduce production risk, and ensure smoother workflows.

CI/CD in machine learning

CI/CD is critical for production / iteration

- E.g.: automate including new patient data
- Helps to avoid data drift

CI/CD in ML:

- Regularly retrain models
- Testing performance
- Automated, rule-based deployment

CI/CD with AWS Elastic Beanstalk

AWS Elastic Beanstalk (EB):

- Fully managed service for deployment and scaling of applications + services
- Install EB

```
eb init

eb create heart_disease_env

eb deploy

eb open
```

- **Alternatives to AWS Elastic Beanstalk**:
  - **Azure Machine Learning**: Offers tools for real-time model scoring, managing compute resources, and monitoring model performance.
  - **Google Cloud App Engine**: A similar solution for deploying ML models on Google Cloud.
  - **Kubernetes**: An open-source container orchestration system that automates deployment, scaling, and management of containerized applications. Works across multiple cloud platforms (GCP, Azure, AWS) and provides flexibility, though it has a steeper learning curve.

- **Key takeaway**: There are many CI/CD solutions beyond AWS, each suited to different cloud environments and needs. Explore alternatives for flexibility and control in your deployments.

Monitoring and visualization

- **Monitoring and Visualization Overview** (00:00 - 00:10):
  - Introduces the importance of **monitoring and visualization** for maintaining and improving machine learning models after deployment.
- **What's Next?** (00:10 - 00:46):

- After deploying and making predictions, **monitoring** is crucial to ensure that the model's performance doesn't degrade over time.
- **Logging** and **visualizing performance metrics** at inference and beyond helps identify issues.
- Monitoring sets up a **feedback loop**, which will be covered in future videos.
- **Logging with Python** (00:46 - 01:10):

Logging with python

```python
import logging
import matplotlib.pyplot as plt

# Setting up basic logging configuration
logging.basicConfig(filename='predictions.log', level=logging.INFO)

# Make predictions on the test set and log the results
for i in range(X_test.shape[0]):
    instance = X_test[i,:].reshape(1, -1)
    prediction = model.predict(instance)
    logging.info(f'Inst. {i} - PredClass: {prediction[0]}, RealClass: {y_test[i]}')
```

Logging with python (cont.)

```python
# Function to visualize the predictions from log
with open(logfile, 'r') as f:
    lines = f.readlines()
    predicted_classes = [int(line.split("Predicted Class: ")[1].split(",")[0]) \
        for line in lines]

    # Perform data analysis, visualization, etc.
    ...
```

- Use Python logging to trace model performance

- **Logging** is vital in ML workflows, not just for debugging but for tracking model predictions and performance over time.
- Python's built-in **logging library** helps log events like model predictions, enabling the identification of trends and anomalies.
- Logs are processed and visualized later to ensure the model functions as expected.
- Continuous monitoring of these logs helps address potential issues.
- **Visualization** (01:22 - 01:49):

Visualization

- Inspect performance over time
- Transform raw data of inputs / predictions into insights

```python
import matplotlib.pyplot as plt

# Sample data: Random accuracy values for 12 months
months = ["Jan", "Feb", "Mar", ...]
accuracies = [0.86, 0.91, 0.74, ...]
plt.plot(months, accuracies, '-o')
plt.title("Model Accuracy Over Months")
plt.xlabel("Months")
plt.ylabel("Accuracy")
plt.show()
```

- **Visualization** transforms raw data into intuitive insights by graphically representing model inputs and outputs.
- Common visualization methods include **line plots** to show accuracy over time, using libraries like **matplotlib**.

- **Importance of Logging** (02:01 - 02:34):

Logging

- Recording of events
  - Tracking variable values, Function calls
  - Information that informs execution + performance
- Monitoring helps track:
  - Usage, Performance, Errors/anomalies

```
2023-08-04 09:15:20 [INFO] Model version 1.2.7 started
2023-08-04 09:15:45 [INFO] Preprocessing input data for prediction
2023-08-04 09:15:47 [DEBUG] Input data shape: (1, 12)
2023-08-04 09:15:48 [INFO] Making prediction
2023-08-04 09:15:50 [DEBUG] Output prediction: [0.78]
...
```

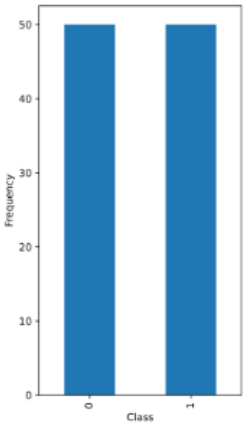
- **Logging** records events in the program, such as tracking variable values and function calls, which helps understand execution flow and system performance.
- Logs help track model usage, performance, and identify anomalies or errors, especially for deployed models in production.
- **Visualization Examples** (02:34 - 03:24):

Visualization examples

- Helpful metric for our model: balanced accuracy over time
- Spot trends, see if performance degrades
- See if retraining is necessary
- Choose helpful metrics for our use-case

Example:

- Balanced accuracy changes relative to expected, real-world rate
- Potentially indicative of problem
- Choose and evaluate

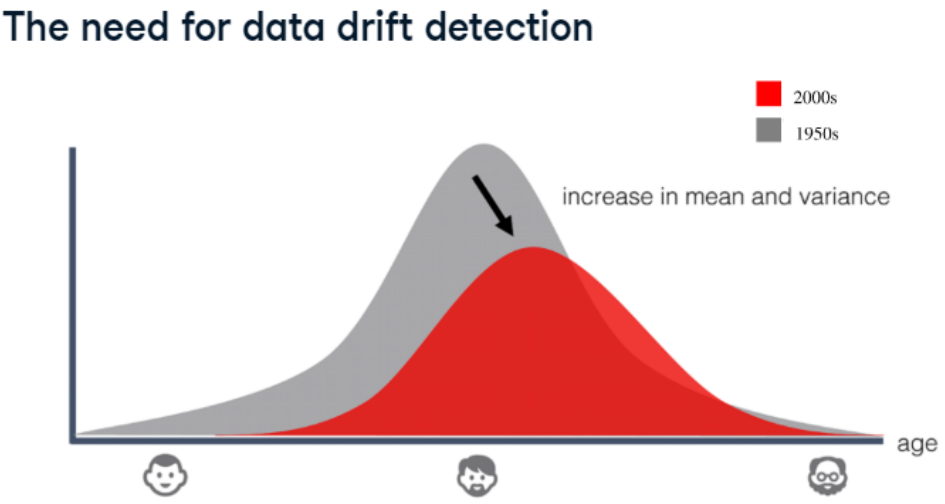


- For the **heart disease binary classification model**, balanced accuracy is visualized over time.
- Visualizing performance helps detect trends, patterns, and anomalies, such as **model drift** or when **retraining** is necessary.
- Collaborating with domain experts (e.g., cardiologists) ensures that monitored metrics align with real-world clinical performance.

Data drift

The Need for Data Drift Detection

00:17 - 01:10



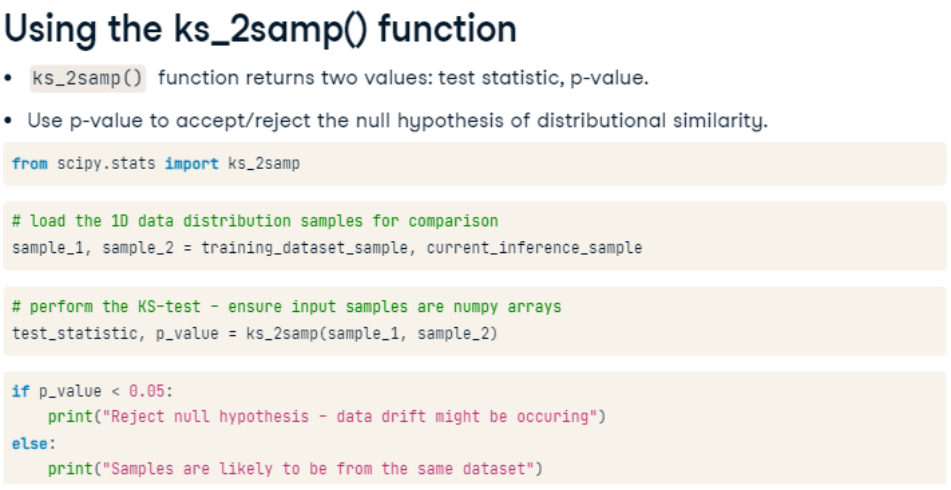
- **Data drift** occurs when the statistical properties of the model's input features change over time due to shifts in the population or data collection process.
- Example: Heart disease rates have changed over time, affecting the accuracy of models trained on older data.
- While data drift doesn't always degrade performance, ensuring models are trained on **recent, relevant data** is crucial for accuracy.

The Kolmogorov-Smirnov Test

01:10 - 01:41

- **Kolmogorov-Smirnov test** (KS test) is a statistical method to detect data drift by comparing distributions of two datasets.
- It highlights significant differences that may indicate drift.

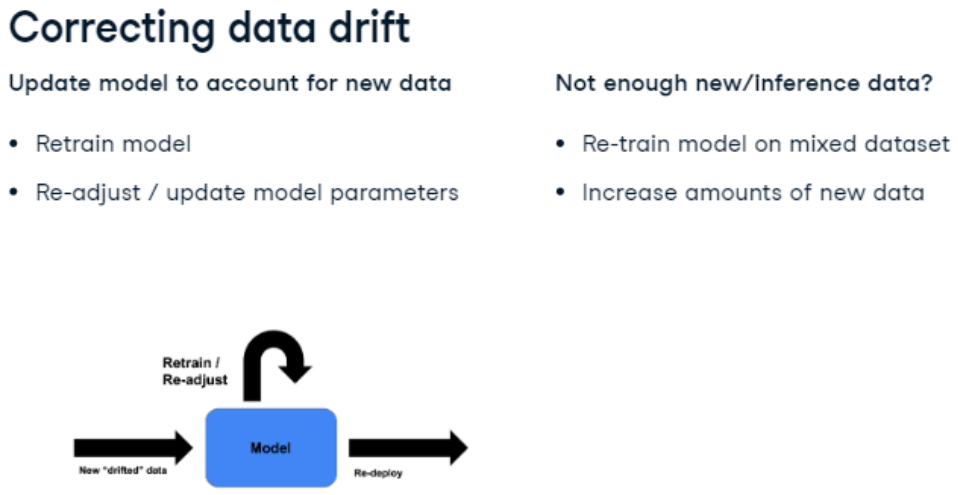
Using the ks_2samp() Function



01:41 - 02:18

- The **scipy ks_2samp** function performs the KS test.
- Outputs include a **test statistic** (showing distribution difference) and a **p-value** (assessing the likelihood of such differences under the assumption of same distribution).
- If the p-value is less than 0.05, data drift is suspected.

Correcting Data Drift



02:18 - 02:51

- **Correcting data drift** involves retraining models to reflect the new data properties.
- If new data is insufficient for training, models can be periodically retrained on a **mixed dataset** of old and new data, gradually increasing the amount of new data.

Further Resources for Detecting and Rectifying Data Drift

02:51 - 03:37

- **Population Stability Index (PSI)** is another method for detecting drift in categorical variables.
- Tools like **Evidently** and **NannyML** help monitor and correct data drift, providing solutions from validation to post-deployment.

Feedback loop, re-training, labeling

Feedback Loop

Feedback loop

- Model output considered as system input:
  - Using metrics/predictions to inform system evolution
  - Can use model monitoring
- Integral part of ML:
  - Allows for rapid learning and adjustment
  - Better adapt to change

00:16 - 01:01

- A **feedback loop** feeds model outputs or performance data back into the system to adjust its future behavior.
- Outputs are continuously used to help the model adapt to changes, trends, or user behaviors.
- The **feedback loop** allows for rapid learning and adjustment, making the model more responsive to changing conditions.

Feedback Loop Implementation

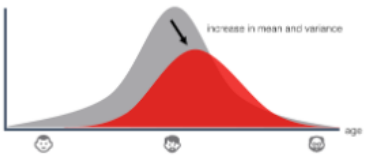
Feedback loop implementation

Data drift detection

- Input data distribution changes over time
- Feedback loop: retrain on newer data

Online learning

- Periodically retrain based on changing data
- Beyond data drift: adapts to changes in data structure

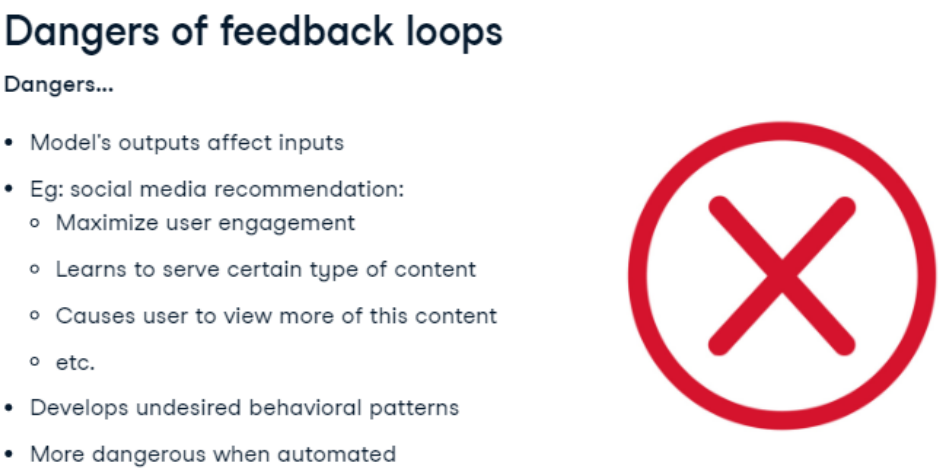


01:01 - 02:07

- A feedback loop can be **manual, semi-automated, or fully automated**.
- **Data drift detection** can be part of a feedback loop, where the model is retrained on newer labeled data to stay current.
- **New labels** can be obtained via manual labeling by experts, crowd-sourcing, or supervised learning.
- **Online learning** allows for periodic retraining based on changing data trends, not limited to data drift but also new data categories or variable relationships.
- Feedback loops can be implemented in different ways based on the specific use case.

Dangers of Feedback Loops

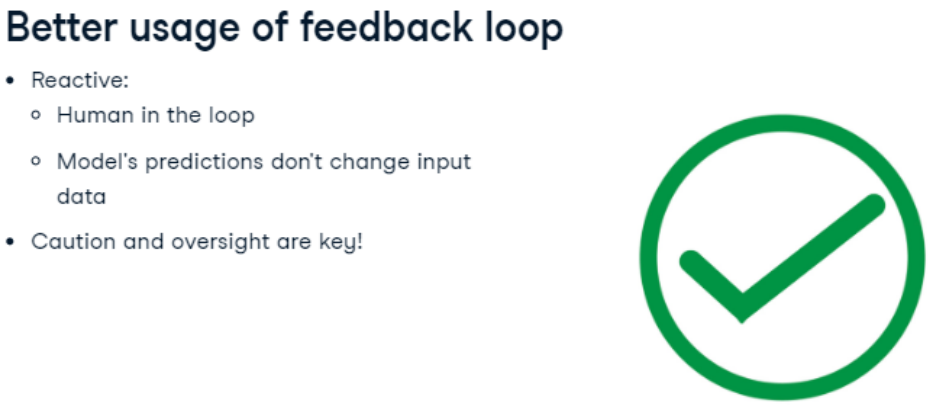
02:07 - 02:49



- Feedback loops can be harmful if model outputs influence the inputs, creating a **negative feedback loop**.
- Example: Social media recommendation systems, where repeated recommendations lead to echo chambers or misinformation.
- **ML engineers** must be cautious when automating feedback loops to avoid unintended consequences.

Better Usage of Feedback Loop

02:49 - 03:20



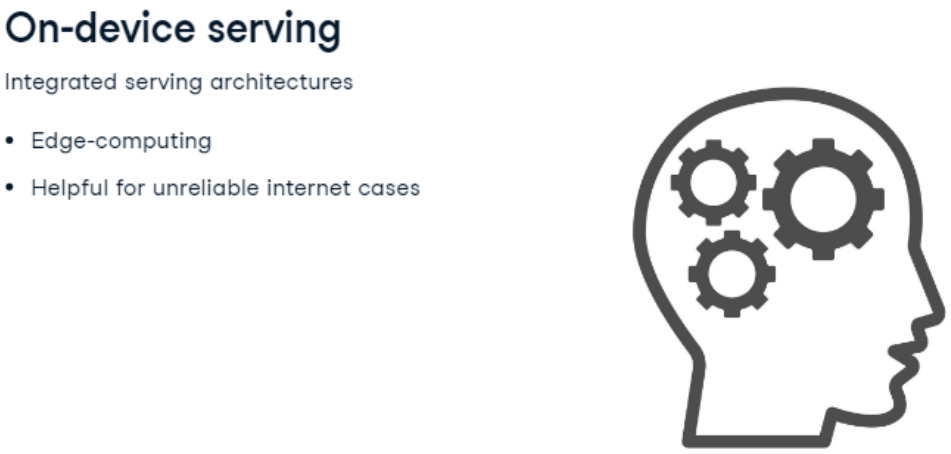
- For the **heart disease model**, the feedback loop is **reactive**, and model predictions are unlikely to affect the input data.
- A **heart disease diagnosis** might encourage healthier behavior in patients, which could be seen as a positive effect.
- It's important to be cautious when setting up feedback loops and ensure they align with **human values**.

Serving the model

Model-as-a-Service (00:07 - 00:56)

- **Model-as-a-service** architecture assumes stakeholders will access the model over the internet through a secure portal.
- Users submit queries and data to receive predictions online.
- However, internet access may not always be available, especially in rural or highly secure environments where sensitive data, like patient information, cannot be passed online.

On-Device Serving (00:56 - 01:20)



- In certain cases, **on-device serving** is more suitable, where the model is integrated into the application or device itself.
- Common in **edge computing**, where devices may lack a reliable network connection.

Pros and Cons of On-Device Serving (01:20 - 02:47)

- **Benefits**:
  - Faster response times since it doesn't rely on external servers, ideal for real-time predictions.
  - No need for internet access, reducing the risk of data breaches and enabling offline functionality.
  - Useful in **remote areas** or disconnected environments.
- **Challenges**:
  - Limited memory and processing power of edge devices require model optimization, potentially compromising accuracy for speed.

- Lacks cloud scalability, leading to issues with diverse devices and OS versions.
- Updating models becomes harder without a central server, and usage statistics and performance metrics are harder to collect.

Implementation Strategies (02:47 - 03:25)

- Techniques for **on-device serving** include:
  - **Pruning** models to make them lighter and faster.
  - **Transfer learning**, where pre-trained models are fine-tuned for specific tasks.
  - Specialized ML frameworks for edge deployment like **TensorFlow Lite**, **Core ML** (Apple), and **ONNX Runtime**.
- Research these techniques further as needed.