Bonus Slide

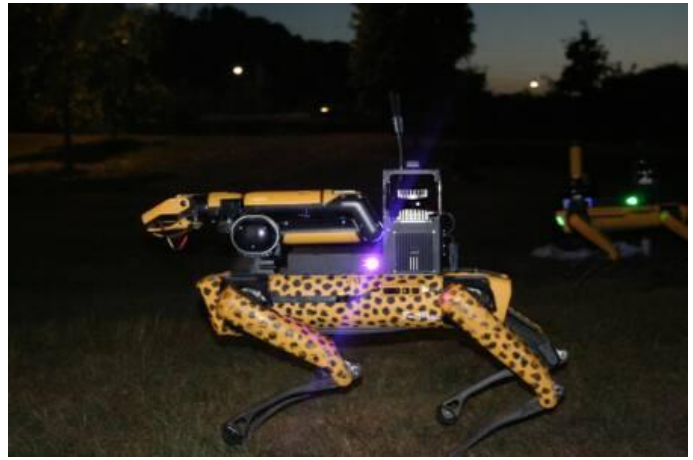
Robots use Sensors and Actuators

Robots are machines that can be programmed to complete complex tasks, often in the physical world. How is this possible?

Robots use **sensors** to gather data from the real world. Examples include cameras (visual data), microphones (auditory data), thermometers (temperature data).

They then use their machine components to create **physical changes**. For example, a self-driving car might tell the engine to slow down, or a robot arm might reposition itself and grasp the fingers.





CMU Robots: https://www.ri.cmu.edu/research/robots/

Unit 2: Modern CS Techniques

Machine Learning Libraries

15-110 – Wednesday 10/28

Quizlet6

Learning Goals

- Describe how **training**, **validation**, and **testing** are used to build a model and measure its performance

- Use **machine learning libraries** to learn models from datasets.

From Training to Testing

Last time, we talked about training a model, the training set, learning algorithms, etc. But that is not the end of the story.

Once we've trained a model using machine learning, we want to evaluate that model to see how well it actually works.

We can do this by **testing** the model using test data held in reserve.

Testing Machine Learning Models

Training Data, Validation Data, Testing Data

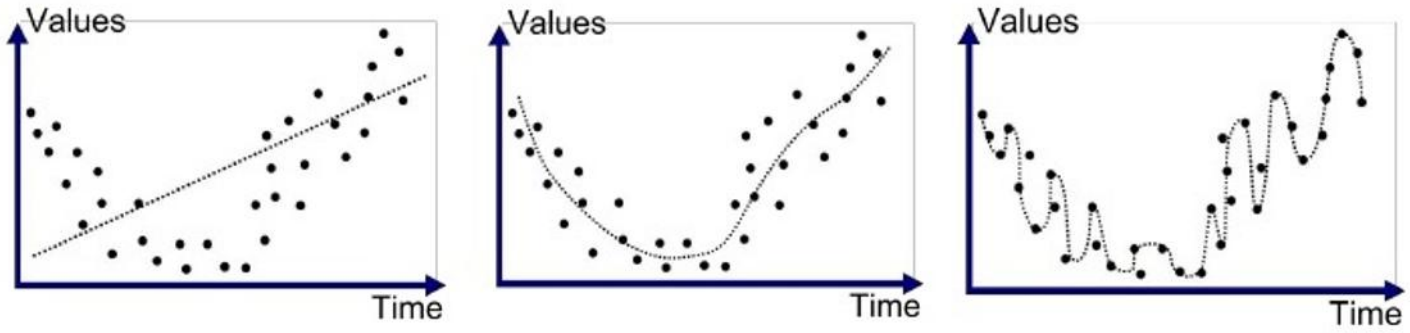
Once we've trained a model, we can use that model to make predictions about data. That means we don't want the model to **only** work on the data we provided originally; we want it to work on future data too.

We can separate our data into three sets: **training**, **validation**, and **testing**. Each subset of data plays a particular role in making the model work as well as it possibly can.

Too Much Data Can Cause Overfitting

The **training data** is normally composed of the majority (maybe 70%) of the available dataset. This data is run through the machine learning algorithm to produce the model. The more training data there is, the more **accurate** the algorithm's model becomes.

This can go wrong if the algorithm over-optimizes the model. For example, it might identify a pattern that only exists within the training data, not in the general population – the pattern might just be noise. This is called **overfitting**. Overfitting can result in a model performing very well on training data, but poorly on test data.



Validation Data Identifies Overfitting

To detect and remove parameters in the model that cause overfitting, you can use **validation data**. This is a subset of the data (maybe 15%) that is **not** used when training the model. Instead, it will be used to **validate** the model during training.

When the programmer runs the algorithm, they can choose certain hyperparameters to customize how it will work. The programmer can then evaluate the produced model on the validation data to see how accurate it is. By trying different combinations of hyperparameters and checking the performance on the validation data, they can try to produce a model that performs well on *all* data, not just the training data.

This is similar to how a teacher may provide students a set of practice problems to learn on, then provide a practice quiz with different problems where the students can test their knowledge.

Testing Data Provides Final Results

When the programmer thinks they've achieved an optimal model, the **testing data** is used to determine how accurate that model actually is. This is a portion of the data (maybe 15%) that was set aside at the beginning and never used during the training or validation process.

Unlike the validation data, which is evaluated multiple times, the model is run on the test data only **once**. We measure how close the predicted results are to the actual results. That score is the accuracy of the model.

You cannot train on your testing data if you want a fair test of the model!!!

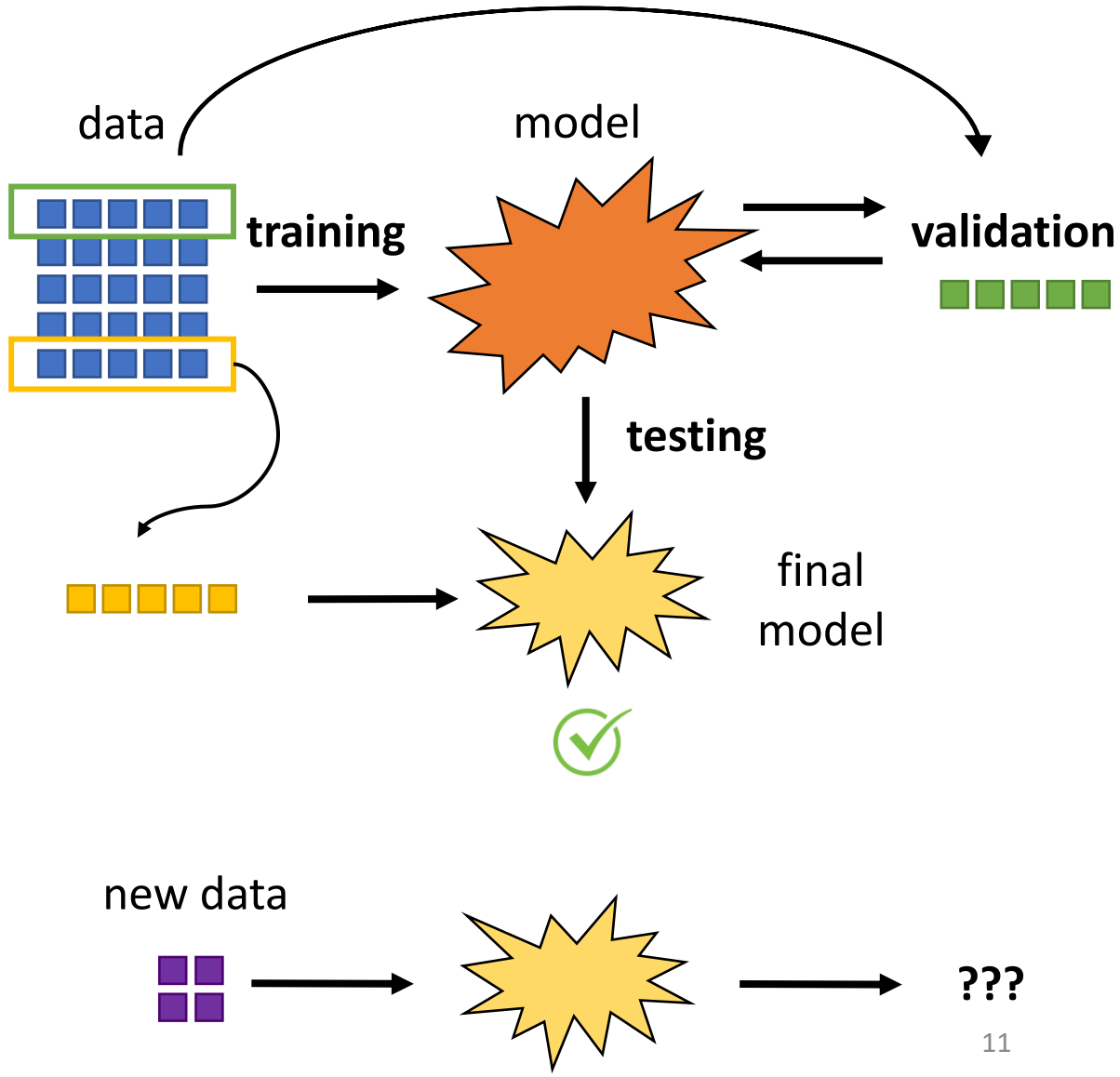
Example: Bad Training Process

What happens if we use our validation and testing data to train as well?

The algorithm will get the opportunity to observe patterns in the additional data. It will optimize the model to include those patterns, even through validation.

When the model is tested, it will of course be accurate because the model was optimized to notice the correct patterns.

But if we try to use the model on new, unlabeled data later on, the patterns may no longer be valid. We don't know for sure because all the labeled data was used for training.



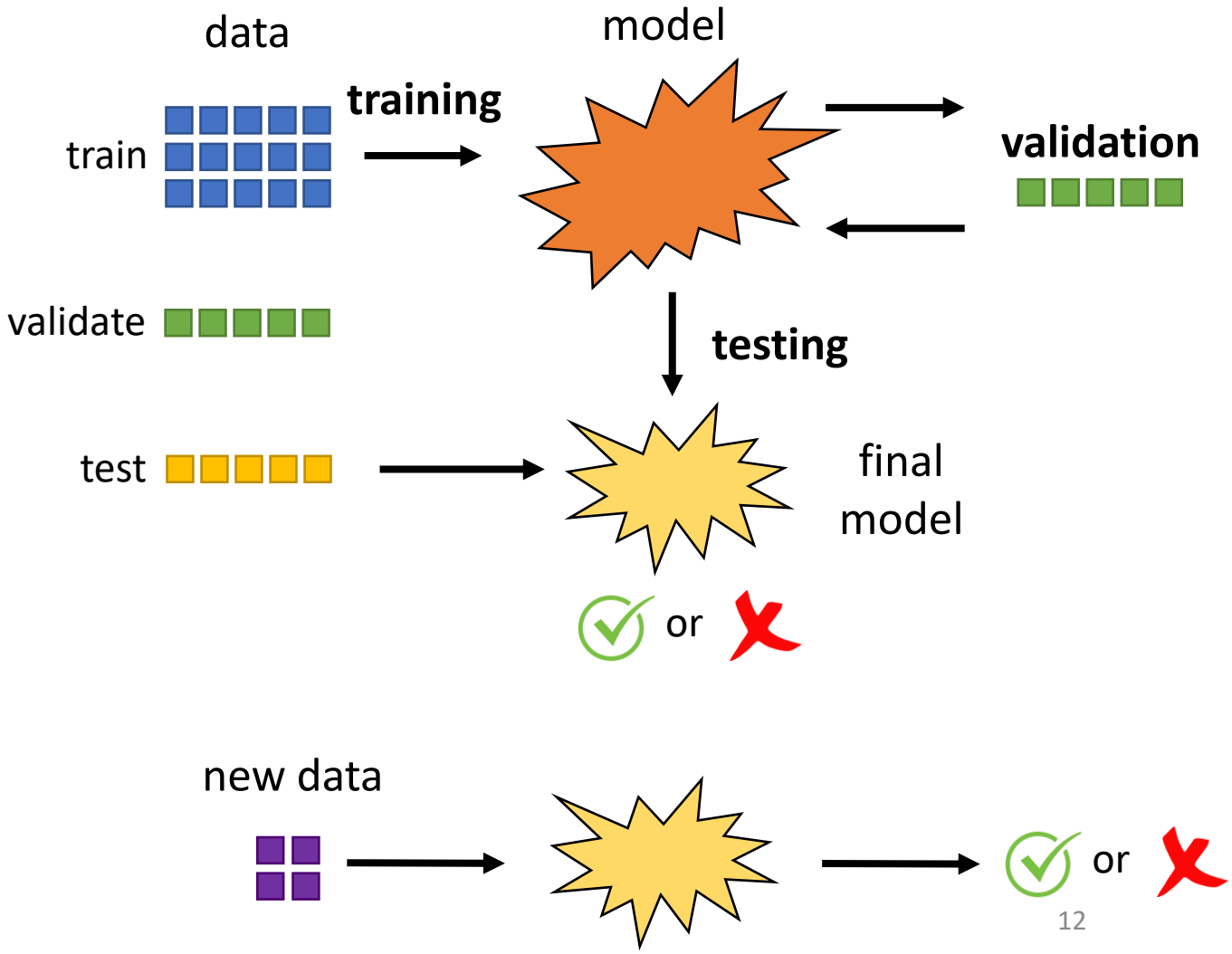
Example: Good Training Process

A better process: split the data into training, validation, and testing sets.

We'll train on **only** the training set and repeatedly test on the validation set. This should produce a model that doesn't overfit too much.

When we're done, we'll test on the test set **once**. That produces our final result. It might be good, or it might be bad; it depends on how the model turned out.

However, the new data should have about the same accuracy as the test data, since the model never saw the test data before. There's less uncertainty.



The **numpy** library

What is **numpy** and why do we use it?



As we work with training and testing data sets, lists are slow and clunky for numeric work.

NumPy is a library that allows us to work with a structure called arrays that are fast, memory-efficient, and built for mathematical calculations. They are especially useful when we work with multiple dimensions of data.

You first have to install `numpy` and import it.

Some common numpy examples

```
import numpy as np

# --- Creating arrays from lists---

np.array([1, 2, 3])                 # array([1, 2, 3])

np.array([[1, 2], [3, 4]])          # 2D array from nested list

np.arange(0, 10, 2)                 # array([0, 2, 4, 6, 8])
                                    # (like range())

# --- Array attributes ---

arr = np.array([[1, 2, 3], [4, 5, 6]])

arr.shape    # (2, 3)  -> 2 rows, 3 columns

arr.size     # 6  -> total number of elements


# --- Indexing & slicing ---

arr = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])

arr[1, 2]     # 60   -> row 1, column 2

arr[0]        # array([10, 20, 30])  -> entire row 0

arr[:, 0]     # array([10, 40, 70])  -> entire column 0
```

```
# --- Useful functions ---

nums = np.array([4, 9, 16, 25])

nums.sum()     # 54

nums.mean()    # 13.5

nums.max()     # 25

nums.min()     # 4


# also works along an axis for 2D arrays:

grid = np.array([[1, 2], [3, 4]])

grid.sum(axis=0)   # array([4, 6])  -> column sums

grid.sum(axis=1)   # array([3, 7])  -> row sums
```

Machine Learning with `scikit-learn`

Machine Learning Libraries



You can implement a machine learning algorithm yourself, but it isn't usually necessary. There are lots of great **external libraries** that have already implemented machine learning algorithms for you. All you have to do is plug in data!

We'll use the library **scikit-learn** here.

You first have to install `scikit-learn` and import it.

Real Example: Predicting Types of Wheat Seeds

Researchers examined 210 kernels belonging to three varieties of wheat: Kama, Rosa and Canadian. These are actual wheat cultivars grown for research/agricultural purposes

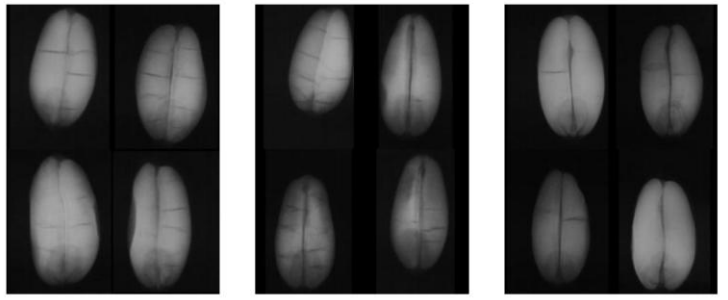
- Kama and Rosa are winter wheat varieties common in parts of Europe
- Canadian refers to a hard red spring wheat type.

High quality visualization of the kernel structure was detected using an X-ray technique and features were established based on the scans.

Studies were conducted using combine harvested wheat grain originating from experimental fields, explored at the Institute of Agrophysics of the Polish Academy of Sciences in Lublin.



Fig. 1. Wheat crop seeds (a) Canadian (b) Rosa (c) Kama



X-ray images of three species of wheats

Charytanowicz, M., Niewczas, J., Kulczycki, P., Kowalski, P. A., Łukasik, S., & Żak, S. (2010). Complete gradient clustering algorithm for features analysis of X-ray images. In *Advances in Intelligent and Soft Computing*. UCI Machine Learning Repository. https://doi.org/10.24432/C5H30K

Meme time:



https://www.youtube.com/shorts/PCwinj3KqX0

Dataset

Let's take a look at **`wheat-seeds.csv`**. What do these numbers mean?

15.26	14.84	0.871	5.763	3.312	2.221	5.22	1
14.88	14.57	0.8811	5.554	3.333	1.018	4.956	1
14.29	14.09	0.905	5.291	3.337	2.699	4.825	1
13.84	13.94	0.8955	5.324	3.379	2.259	4.805	1
16.14	14.99	0.9034	5.658	3.562	1.355	5.175	1
14.38	14.21	0.8951	5.386	3.312	2.462	4.956	1
14.69	14.49	0.8799	5.563	3.259	3.586	5.219	1
14.11	14.1	0.8911	5.42	3.302	2.7	5	1
16.63	15.46	0.8747	6.053	3.465	2.04	5.877	1
16.44	15.25	0.888	5.884	3.505	1.969	5.533	1
15.26	14.85	0.8696	5.714	3.242	4.543	5.314	1

Features: Area, perimeter, compactness, length of kernel, width of kernel, asymmetry coefficient, and length of kernel groove

- All continuous and real-valued

Target Label: 1 (Kama), 2 (Rosa), 3 (Canadian)

Our steps to train an ML Model

1. Load the Data
2. Train / Test Split
3. Train the Model
4. Prediction Example
5. Evaluate the Model
    1. Accuracy Evaluation
    2. Confusion Matrix

Loading and Preparing the Data

We'll load **`wheat-seeds.csv`** into a pandas DataFrame and take a quick look at its structure.

Next, we'll define our features (X) and target label (y):

Features (X) are the columns:
`area, perimeter, compactness, kernel_length, kernel_width, asymmetry_coefficient, kernel_groove_length`

Target label (y): variety
`0 = Kama, 1 = Rosa, 2 = Canadian`

Train / Test Split

Before training, we need to split our data into two groups:
training data and **testing** data.

`scikit-learn`'s function `train_test_split` handles this automatically:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

We use an **80/20 split**:
80% for training, 20% for testing

Ensures the split is reproducible,
running the code again gives you the same
train/test split every time

Train the Model

Since we're predicting a category (not a continuous value), we could use **logistic regression.**

The `.fit()` method trains the starter model on our training data.

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(max_iter=200)
model.fit(X_train, y_train)
```



limits how many rounds of adjustments the model can make while trying to find the best fit

Prediction Example

Once trained, the model can predict on new, unseen data using `model.predict()`

Just pass in a sample with the same features the model was trained on.

```
# One kernel's measurements
sample = [[15.26, 14.84, 0.871, 5.763, 3.312, 2.221, 5.22]]

prediction = model.predict(sample)
print("Predicted variety:", target_names[prediction[0]])
```

Accuracy Evaluation

The `.score()` method returns the model's **accuracy** on the test data:

Accuracy = (correct predictions) / (total predictions)

To test our model, we call `.score()` using the test data we set aside earlier. The result is a value between 0 and 1.

```
accuracy = model.score(X_test, y_test)
```

We get **90.47%**, quite good!

Reminder: the test data must be set aside *before* training. Training on test data gives a misleadingly high score.

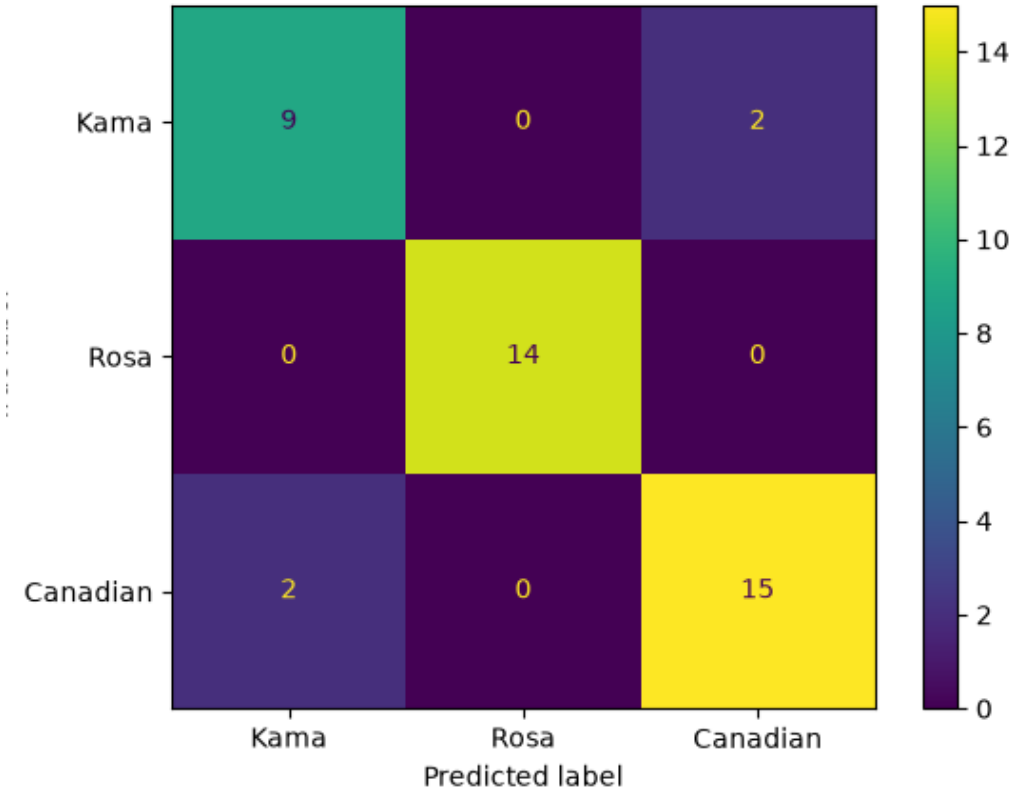
Confusion Matrix

A confusion matrix shows how many predictions were correct vs. incorrect, **broken down by class.**

Rows represent the actual (true) labels; columns represent the predicted labels.

The diagonal shows correct predictions; everything off the diagonal shows where the model confused one class for another.

```
ConfusionMatrixDisplay.from_estimator(
    model, X_test, y_test,
    display_labels=target_names
)
```



Sidebar: Many more ways to evaluate

Accuracy isn't the only way to measure a model's performance. Metrics like **precision, recall, F1-score,** and **ROC-AUC** can tell a more complete story.

These matter most when classes are imbalanced (e.g., 95% of samples are in one category) or when different types of mistakes carry different costs.

We won't cover these in detail here, that's a topic for a more advanced class.

Learning Goals

- Describe how **training**, **validation**, and **testing** are used to build a model and measure its performance

- Use **machine learning libraries** to learn models from datasets.