Episode 5: ML Lifecycle Fundamentals

Building Your First Machine Learning Pipeline

AI & ML: From Theory to Practice

Preparing for Coding Exercise 1

Matthew Care 2026

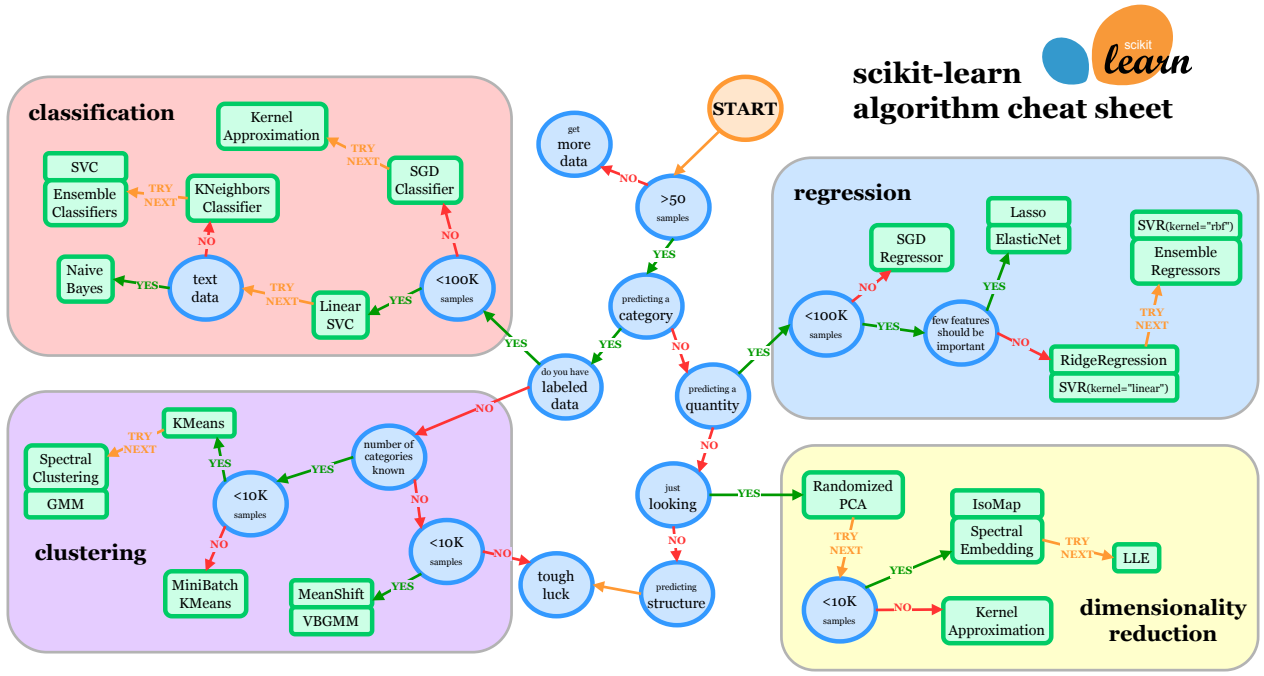
Learning Objectives

By the end of this episode, you will be able to:

- **Navigate** the Python ML ecosystem and choose appropriate tools
- **Understand** dataset structure, features, and data types
- **Select** appropriate models based on problem type
- **Handle** missing data effectively
- **Apply** normalization and encoding techniques
- **Implement** stratified data splitting
- **Explain** cross-validation and why it matters
- **Recognize** common **pitfalls**: bias, imbalanced data, overfitting

The Machine Learning Lifecycle

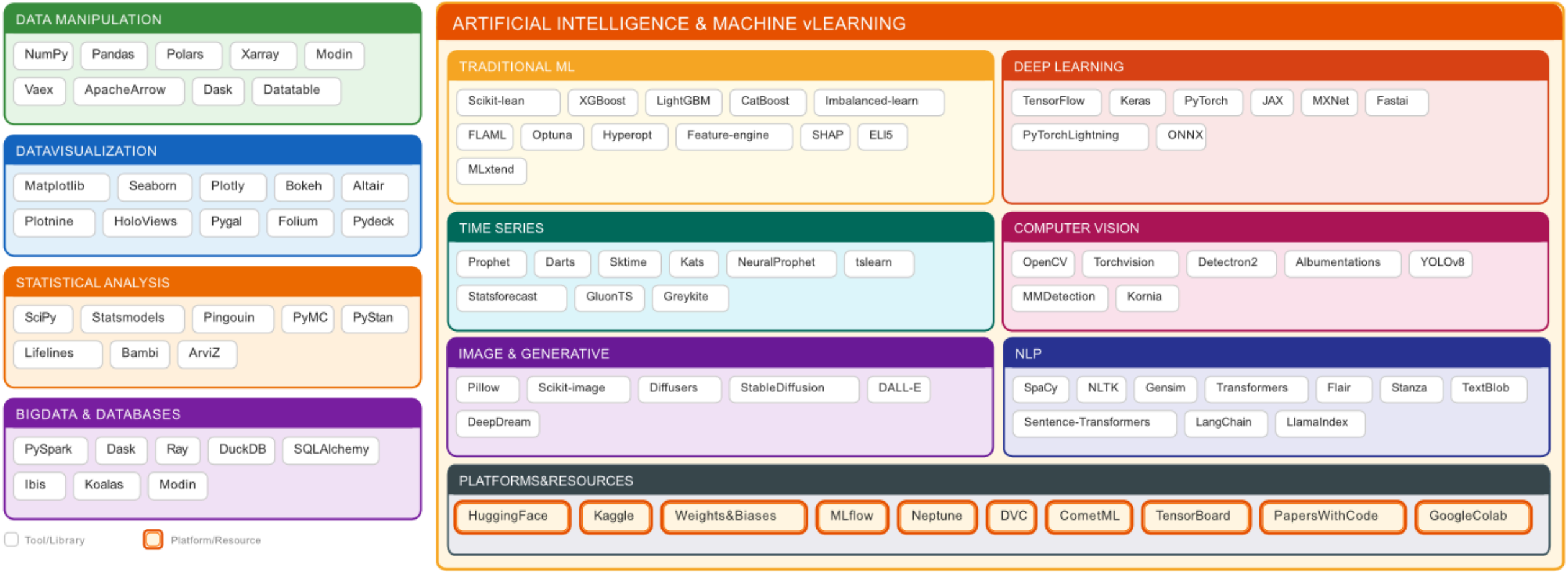
1. **Problem Definition** - What are we trying to predict?
2. **Data Collection** - Gather relevant data
3. **Data Preparation** - Clean, transform, encode
4. **Model Selection** - Choose algorithm(s)
5. **Training** - Fit model to data
6. **Evaluation** - Test performance
7. **Iteration** - Refine and improve



Section 1: The Python ML Ecosystem

A Rich Landscape of Tools

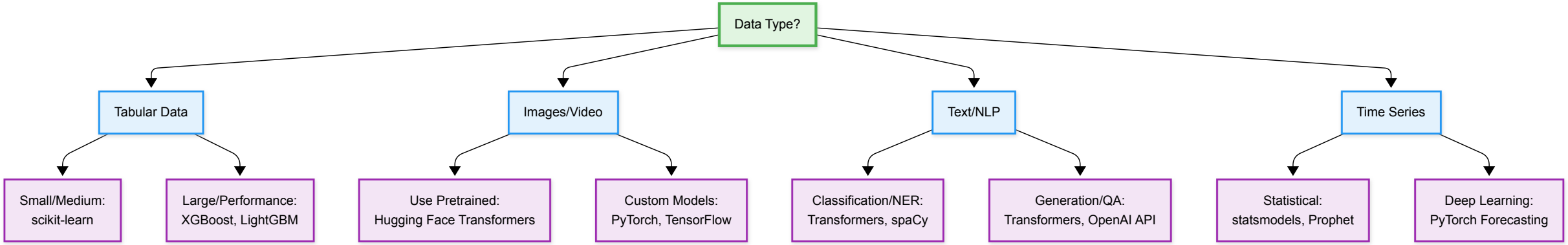
Python Data Science & Machine Learning Ecosystem



The Python ML Ecosystem Overview

Library	Primary Use/Type	When to Use
scikit-learn	Traditional ML	Small/medium tabular data, interpretability needed
statsmodels	Statistical Models	Time series, statistical rigor, interpretability
XGBoost	Gradient Boosting	Large tabular data, maximum performance
LightGBM	Gradient Boosting	Very large datasets, memory efficiency
Transformers	Pretrained Models	Text (NLP), images, multi-modal tasks
spaCy	NLP Pipeline	Production NLP, rule-based + ML systems
TensorFlow/Keras	Deep Learning	Production deployment, quick prototyping
PyTorch	Deep Learning	Research, custom architectures, flexibility
Prophet	Time Series	Business forecasting, seasonal patterns
PyTorch Forecasting	Deep Learning TS	Complex time series, multivariate forecasting
OpenAI API	LLM Access	Text generation, quick prototyping, high quality

Which tool/approach to use



scikit-learn: The Foundation

Why sklearn is your best starting point:

- Consistent API across all algorithms
- Excellent documentation with examples
- Covers 90% of tabular ML needs
- Built-in tools for preprocessing, evaluation, tuning
- Integrates well with pandas, numpy, matplotlib

The sklearn pattern you'll see everywhere:

```
from sklearn.model_name import ModelClass
model = ModelClass(hyperparameters)
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

Hyperparameters: settings of a ML model that are set before training and control the model's learning process and complexity. **Will be covered in detail in Episode 6/CE2**.

Documentation: https://scikit-learn.org/stable/

scikit-learn vs Deep Learning

Use scikit-learn when:

- **Tabular/structured data**
- < 100K samples
- Interpretability matters
- Quick iteration needed
- Limited compute resources

Use Deep Learning when:

- Images, audio, video
- Massive datasets (millions+)
- Complex patterns/relationships
- Pre-trained models available (HF)
- GPU resources available

Rule of thumb: Start with sklearn. Only move to deep learning when sklearn clearly isn't enough.

Key sklearn Modules You'll Use

Data preprocessing

```
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
```

Model selection and evaluation

```
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import accuracy_score, confusion_matrix
```

Classification algorithms

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.svm import SVC
```

Pipelines (combining steps)

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
```

These will be explored in CE1/CE2

Section 2: What is a Dataset?

Understanding the Structure of ML Data

Anatomy of a Tabular Dataset

Feature 1	Feature 2	Feature 3	...	Target
value	value	value	...	label
value	value	value	...	label
...	...	...	...	...

- **Rows** = Samples/Observations/Instances (e.g. patient)
- **Columns** = Features/Variables/Attributes (e.g. genes, clinical-features etc.)
- **Target** = What we're trying to predict (label/response)
- **Features** = Input variables used to make predictions

Feature Types

Numerical Features

- **Continuous**: Height, weight, temperature
- **Discrete**: Count of items, age in years

Can be ordered and measured

Categorical Features

- **Nominal**: Color, country, name
- **Ordinal**: Education level, rating (1-5)

Categories/groups, may or may not have order

Why does this matter? Different feature types require different preprocessing steps. Models need numbers, so categorical data must be encoded.

CE1/2 Dataset: Adult Census Income

Source: Kaggle Datasets

Task: Predict whether income exceeds $50K/year based on census data. 14 features, 1 target.

Feature	Type	Example Values
age	Numerical	39, 50, 38, ...
workclass	Categorical	Private, Self-emp, Gov, ...
education	Ordinal	HS-grad, Bachelors, Masters, ...
occupation	Categorical	Tech-support, Sales, ...
hours-per-week	Numerical	40, 50, 45, ...
...	...	...
income	**Target**	<=50K, >50K

Dataset Characteristics to Assess

Before modelling, always check:

1. **Size**: How many samples? How many features?
2. **Balance**: Are classes equally represented?
3. **Missing values**: Any gaps in the data?
4. **Data types**: Numerical vs categorical?
5. **Distributions**: Normal? Skewed? Outliers?
6. **Correlations**: Related features?

Quick dataset exploration

```
df.shape              # (n_samples, n_features)
df.info()             # Data types and missing values
df.describe()         # Summary statistics
df['target'].value_counts()  # Class distribution
```

Class Imbalance: A Preview

Adult Census Income distribution:

Class	Count	Percentage
<=50K	24,720	~76%
>50K	7,841	~24%

Problem: A model that always predicts "<=50K" gets 76% accuracy but is useless!

We'll address this properly in Episode 6 (CE2) with better metrics.

Section 3: Model Choices by Task

Selecting the Right Algorithm

Classification vs Regression

Classification

Predict a **category/class**

- Is this email spam? (Yes/No)
- What digit is this? (0-9)
- Which species? (A/B/C)
- Which disease sub-type?

Algorithms: Logistic Regression, Random Forest, SVM, Neural Networks

Regression

Predict a **continuous value**

- What's the house price?
- How many sales next month?
- What temperature tomorrow?

Algorithms: Linear Regression, Ridge, Lasso, SVM, Random Forest Regressor

Many algorithms can be used for both classification/regression.

Supervised vs Unsupervised Learning

Supervised Learning

We have labels

- Training data includes "correct answers"
- Model learns input → output mapping
- Classification, Regression

Examples: Spam detection, price prediction, disease subtype etc.

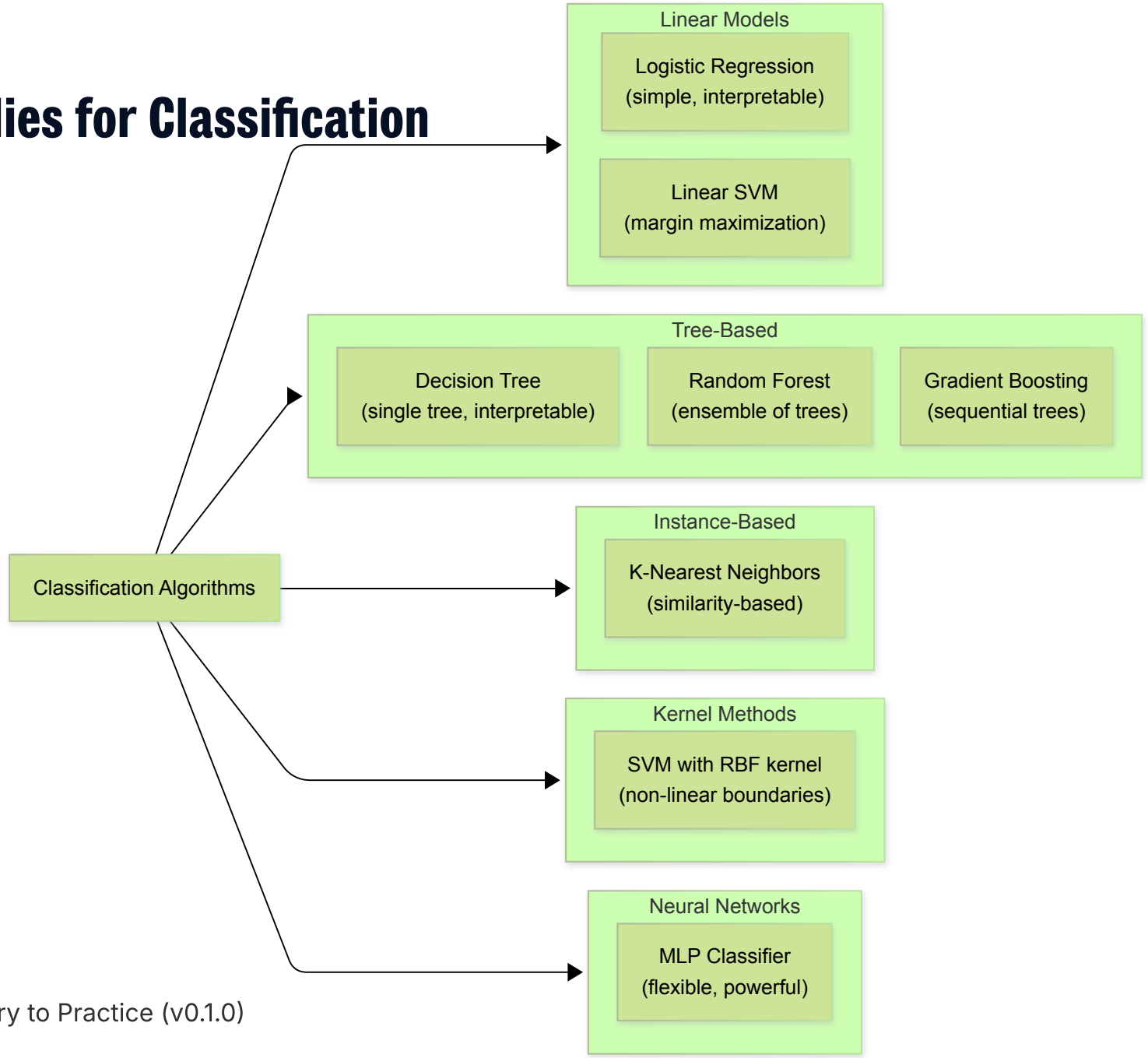
Unsupervised Learning

No labels

- Find structure in unlabelled data
- Clustering, dimensionality reduction
- Anomaly detection

Examples: Customer segmentation, topic modelling, patient stratification etc.

Algorithm Families for Classification



Quick Model Selection Guide

Situation	Consider
Baseline / first attempt	Logistic Regression
Need interpretability	Decision Tree, Logistic Regression
Tabular data, want best accuracy	Gradient Boosting (XGBoost/LightGBM)
Many features, worried about overfitting	Random Forest
Non-linear relationships suspected	SVM (RBF), Random Forest
Very large dataset	Linear models or Neural Networks

Pro tip: Try multiple models and compare. Don't assume one is best without testing! Sklearn makes testing a number of models easy.

Section 4: Handling Missing Data

Strategies for Incomplete Datasets

Types of Missing Data

MCAR

Missing Completely At Random

Missingness has no pattern
(e.g., random sensor failure)

Can safely drop or impute

MAR

Missing At Random

Missingness depends on observed data
(e.g., older respondents skip online-only questions)

Need careful imputation

MNAR (Not at Random)

Missingness depends on the missing value itself
(e.g., high earners don't report income)

Hardest to handle - may need domain knowledge

Strategy 1: Deletion

Drop rows with missing values:

```
df clean = df.dropna()  # Remove rows with any NaN
```

Pros

- Simple and fast
- Preserves data integrity
- Works well if < 5% missing

Cons

- Loses information
- Reduces sample size
- May introduce bias (if not MCAR)

Strategy 2: Imputation

Fill in missing values with estimates:

```
from sklearn.impute import SimpleImputer

# Numerical: mean, median, or constant
imputer = SimpleImputer(strategy='median')
X_imputed = imputer.fit_transform(X)

# Categorical: most frequent or constant
cat_imputer = SimpleImputer(strategy='most_frequent')
X_cat_imputed = cat_imputer.fit_transform(X_cat)
```

Strategy	When to Use
Mean	Normally distributed numerical data
Median	Skewed numerical data, outliers present
Mode (most_frequent)	Categorical data
Constant	Domain-specific default makes sense

Advanced Imputation Methods

Beyond simple imputation:

```
# K-Nearest Neighbors imputation
from sklearn.impute import KNNImputer
imputer = KNNImputer(n_neighbors=5)

# Iterative imputation (experimental!)
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
imputer = IterativeImputer(max iter=10)
```

These methods use relationships between features to make smarter imputations. KNN finds similar samples; Iterative uses models to predict missing values.

See: https://scikit-learn.org/stable/modules/impute.html

Handling Missing Values in Practice

Adult Census dataset example:

```
# Check missing values
df.isnull().sum()  # or df.isna().sum()

# In this dataset, "?" represents missing values
df.replace('?', np.nan, inplace=True)

# Strategy: Drop rows with missing values (small %)
df_clean = df.dropna()

# Alternative: Impute with most frequent
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='most frequent')
```

Section 5: Data Normalization

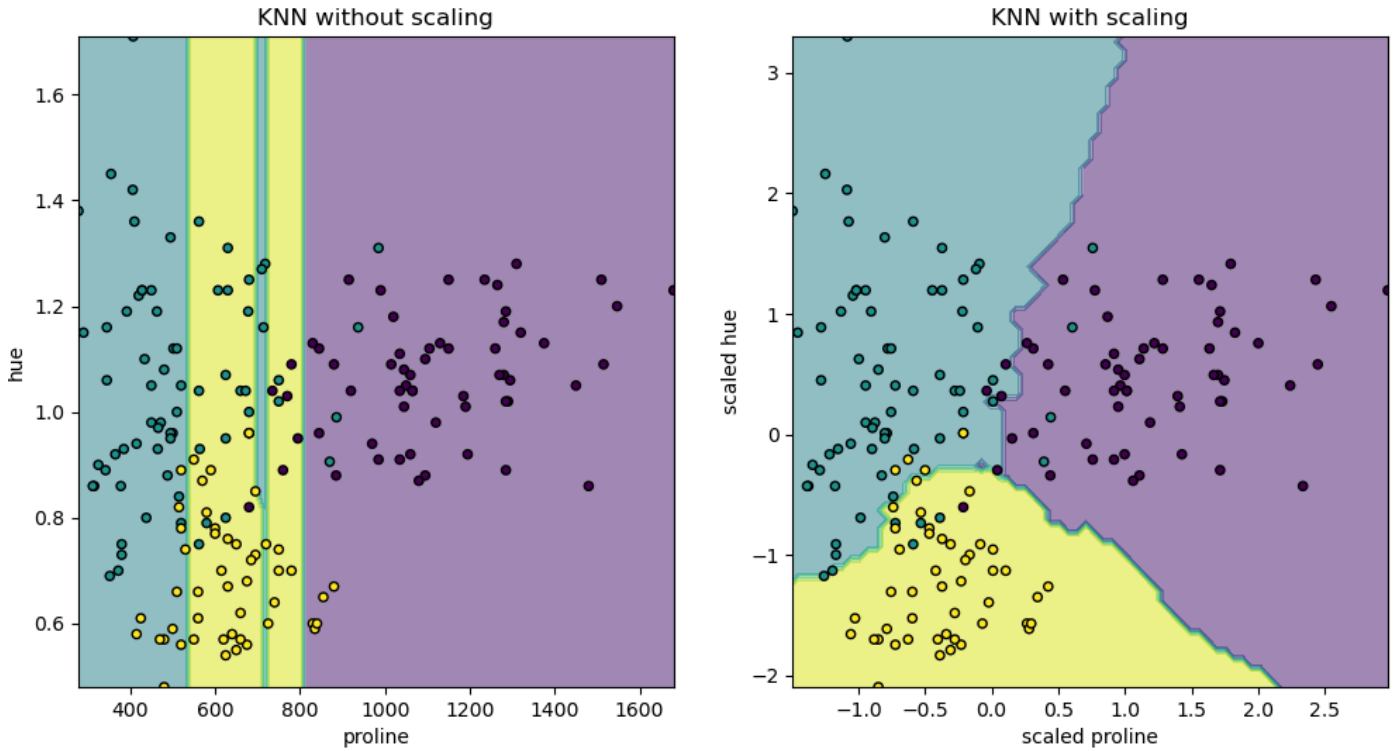
Scaling Features for Better Performance

Why Normalize?

Problem: Features have different scales

Feature	Range	Impact
Age	17-90	Moderate
Capital-gain	0-99,999	Dominates!
Hours-per-week	1-99	Small

Without normalization, algorithms like SVM, kNN, and neural networks give excessive weight to features with larger values.



Normalization Methods Overview

Method	Formula	Output Range	Use When
StandardScaler	$\frac{x-\mu}{\sigma}$	Unbounded	Normally distributed data
MinMaxScaler	$\frac{x-min}{max-min}$	[0, 1]	Bounded range needed
RobustScaler	$\frac{x-median}{IQR}$	Unbounded	Outliers present
QuantileTransformer	Rank → distribution	[0, 1] or Normal	Non-normal data

StandardScaler

Z-score normalization: center at 0, scale to unit variance

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_train)

# On new data, use same parameters
X test scaled = scaler.transform(X test)
```

Before:

- Mean = 38.5
- Std = 13.6
- Range = [17, 90]

After:

- Mean ≈ 0
- Std ≈ 1
- Range ≈ [-1.6, 3.8]

RobustScaler

Uses median and IQR (interquartile range) - robust to outliers

```
from sklearn.preprocessing import RobustScaler

scaler = RobustScaler()
X_scaled = scaler.fit_transform(X_train)
```

> **When to use:** Your data has outliers that you don't want to remove but shouldn't dominate scaling.

The Adult Census dataset has features like `capital-gain` with extreme outliers (most values are 0, some are very large).

QuantileTransformer

Forces data to follow a specific distribution (uniform or normal)

```
from sklearn.preprocessing import QuantileTransformer

# Transform to uniform distribution [0, 1]
transformer = QuantileTransformer(output_distribution='uniform')

# Transform to normal distribution
transformer = QuantileTransformer(output_distribution='normal')

X_transformed = transformer.fit_transform(X_train)
```

Warning: This is a non-linear transformation. It changes the shape of your data, not just the scale.

Scaling: Critical Rules

NEVER fit scalers on test data!

```
# WRONG - Data leakage!
scaler.fit(X_test)

# CORRECT - Fit on train, transform both
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Why? The test set simulates unseen data. Using test statistics to scale means your model has "peeked" at the test data.

Section 6: Categorical Encoding

Converting Categories to Numbers

Why Encode?

ML algorithms require numerical input:

```
# This doesn't work!
model.fit([["Private", "Bachelors"], ["Gov", "HS-grad"]], [0, 1])

# ValueError: could not convert string to float: 'Private'
```

Categories must become numbers, but how we do it matters.

Label Encoding

Assign each category a number:

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
df['workclass encoded'] = le.fit transform(df['workclass'])
```

Original	Encoded
Private	0
Self-emp	1
Gov	2
...	...

Problem: Implies ordering! Model may think Gov (2) > Self-emp (1) > Private (0)

One-Hot/Dummy Encoding

Create binary column for each category:

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse_output=False)
encoded = encoder.fit transform(df[['workclass']])
```

Private	Self-emp	Gov	...
1	0	0	...
0	1	0	...
0	0	1	...

No implied ordering! Each category is equally different from others.

However each category becomes it's own binary feature → increased dimensionality

One-Hot Encoding in Practice

```
from sklearn.preprocessing import OneHotEncoder

# Create encoder
encoder = OneHotEncoder(
    sparse_output=False,    # Return dense array
    handle_unknown='ignore' # Handle unseen categories in test
)

# Fit on training data
encoder.fit(X_train[categorical_columns])

# Transform
X_train_encoded = encoder.transform(X_train[categorical_columns])
X_test_encoded = encoder.transform(X_test[categorical_columns])

# Get feature names
encoder.get feature names out(categorical columns)
```

One-Hot vs Label Encoding

Aspect	One-Hot	Label
For nominal data	Yes	No (implies order)
For ordinal data	Loses order info	Yes
High cardinality	Many columns	Single column
Memory usage	Higher	Lower
Tree-based models	Both work	Preferred

Rule of thumb: Use one-hot for **nominal** categories. Use label/ordinal encoding when categories have **meaningful order**.

Handling High Cardinality

Problem: A feature with 1000 categories creates 1000 columns!

Solutions:

- **Group rare categories:** Combine categories with < N samples into "Other"
- **Target encoding:** Replace category with mean of target (careful: leakage risk!)
- **Frequency encoding:** Replace with category count/frequency
- **Embedding:** Learn vector representation (deep learning)

```
# Group rare categories
value_counts = df['category'].value_counts()
rare = value_counts[value_counts < 100].index
df['category'] = df['category'].replace(rare, 'Other')
```

Section 7: Stratification of data

Preserving Class Balance in Train/Test Splits

The Problem with Random Splits

Without stratification, random chance can cause imbalanced splits:

Original Data

- <=50K: 76%
- >50K: 24%

Possible Bad Split

- Train: 80% / 20%
- Test: 65% / 35%

Different distributions!

With small datasets or severe imbalance, random splits can put few/no minority class samples in test set.

Stratified Splitting

Ensures each split has the same class proportion as the original data:

```
from sklearn.model_selection import train_test_split

# Without stratification
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# WITH stratification (recommended)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42,
    stratify=y  # <-- Key parameter!
)
```

Verifying Stratification

```
# Check class proportions after split
print("Original:", y.value_counts(normalize=True))
print("Train:", y_train.value_counts(normalize=True))
print("Test:", y_test.value_counts(normalize=True))
```

Expected output:

```
Original: <=50K    0.76, >50K    0.24
Train:    <=50K    0.76, >50K    0.24  <- Same proportions!
Test:     <=50K    0.76, >50K    0.24  <- Same proportions!
```

Section 8: Cross-Validation

Robust Model Evaluation

The Problem with a Single Split of the data

Your model's performance depends on which samples end up in test:

```
Split 1: Accuracy = 82%
Split 2: Accuracy = 79%
Split 3: Accuracy = 85%
Split 4: Accuracy = 80%
Split 5: Accuracy = 83%
```

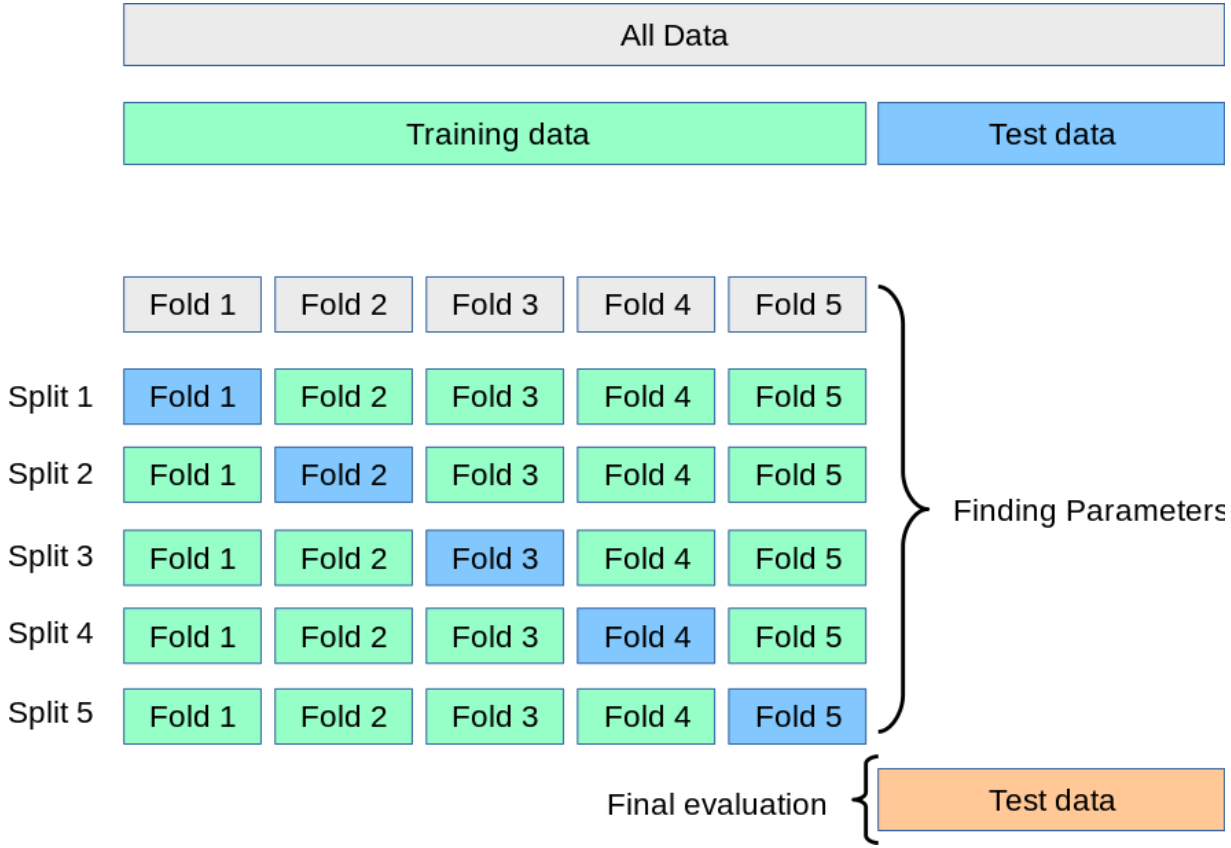
Which is the "true" performance? **All of them!** A single split gives an incomplete picture.

K-Fold Cross-Validation

Divide data into K parts, train on K-1, test on 1, rotate:

Every sample is tested exactly once!

see: https://scikit-learn.org/stable/modules/cross_validation.html



Stratification for Cross-Validation

```python
from sklearn.model_selection import cross_val_score, StratifiedKFold

# Simple cross-validation
scores = cross_val_score(
    model, X, y,
    cv=5,                    # 5-fold
    scoring='accuracy'       # Metric to evaluate
)

print(f"Accuracy: {scores.mean():.3f} ± {scores.std():.3f}")

# Stratified K-Fold for classification
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(model, X, y, cv=skf)
```

Each fold maintains class proportions

Always use StratifiedKFold for classification. Regular KFold doesn't preserve class balance.

Repeated Stratified K-Fold

Even more robust: repeat K-fold multiple times with different shuffles

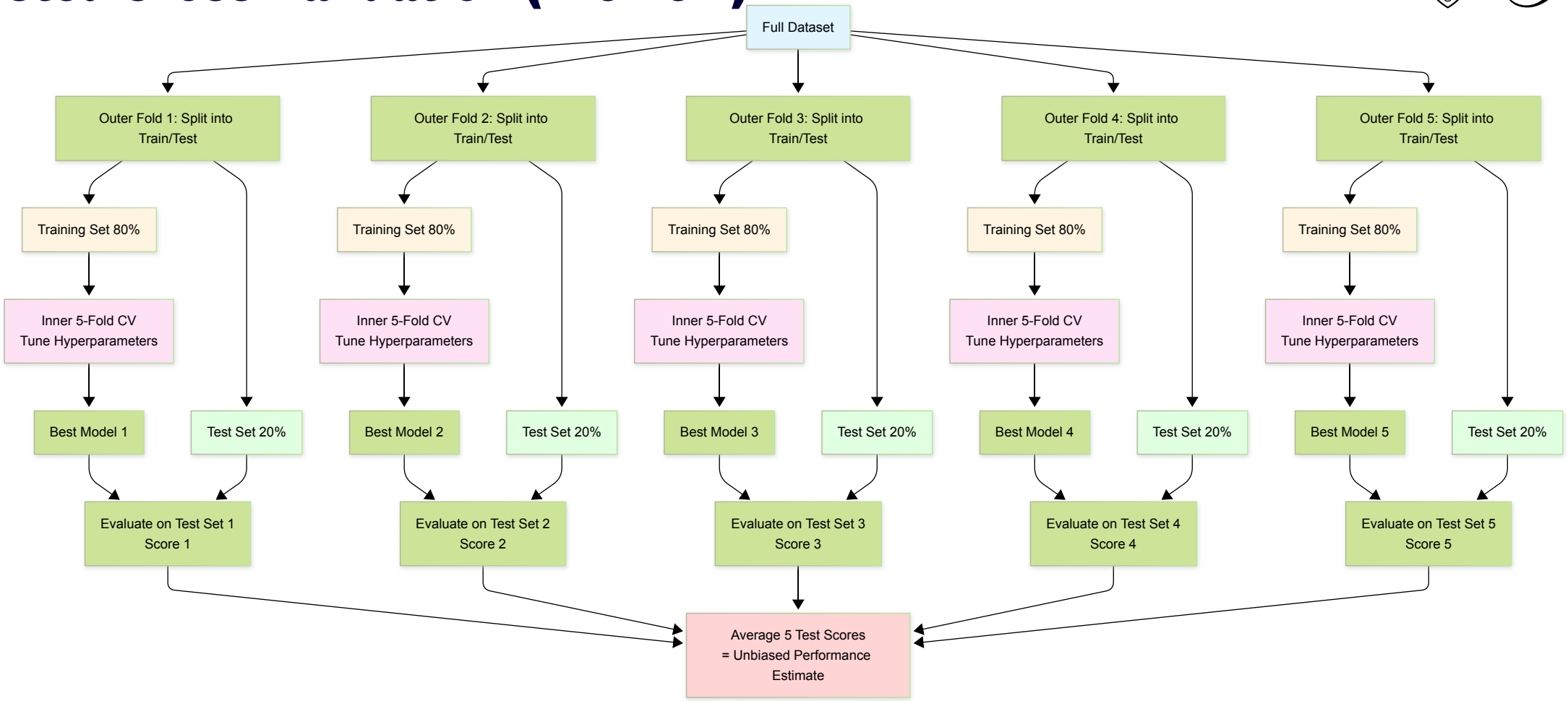
```
from sklearn.model_selection import RepeatedStratifiedKFold

rskf = RepeatedStratifiedKFold(
    n_splits=5,      # 5-fold
    n_repeats=3,     # Repeat 3 times
    random_state=42
)

scores = cross_val_score(model, X, y, cv=rskf)
# Returns 15 scores (5 folds × 3 repeats)
```

Used in CE1: Gives multiple performance estimates for robust comparison between models.

Cross-Validation vs Hold-Out

Aspect	Hold-Out (Single Split)	Cross-Validation
Speed	Fast	Slower (K× training)
Variance	High (depends on split)	Low (averaged)
Data efficiency	Less (test data not used for training)	More (all data used for training)
When to use	Large datasets, quick experiments	Smaller datasets, final evaluation

Nested Cross-Validation (Preview)



Inner loop finds best hyperparameters. Outer loop gives unbiased performance estimate.

Outside scope of this course!.

Section 9: Common Pitfalls

Mistakes to Avoid

Pitfall 1: Data Leakage (VERY important)

Information from test set "leaks" into training

Common causes:

- Fitting scalers/**feature-selection** on full dataset (including test)
- Feature engineering using future information
- Not using pipelines for cross-validation

```
# WRONG
scaler.fit(X)  # Includes test data!

# CORRECT
scaler.fit(X_train)  # Only training data
```

This is perhaps the most important take-away message of the entire course. Very easy to generate very performant models **built on entirely random data** - showing the dangers of data leakage

Pitfall 2: Accuracy with Imbalanced Data

Accuracy can be misleading:

Model	Accuracy	Reality
Always predict "<=50K"	76%	Useless!
Actual good model	82%	Only 6% "better"?

Solution: Use metrics designed for imbalance:

- **Balanced Accuracy**
- **F1 Score**
- **Matthews Correlation Coefficient (MCC)**
- **Precision/Recall**

Covered in depth in Episode 6/CE2

Pitfall 3: Overfitting

Model memorizes training data instead of learning patterns

```
Training Accuracy: 99%
Test Accuracy:     75%
           ↑
     Huge gap = Overfitting!
```

Signs:

- Large gap between train and test performance
- Performance degrades on new data
- Model is very complex

Solutions:

- More training data
- Regularization
- Simpler model
- Cross-validation for early detection

Pitfall 4: Not Stratifying Splits

Random splits can distort class proportions:

```
# Dangerous for imbalanced data
train_test_split(X, y, test_size=0.2)

# Safe - always stratify for classification
train test split(X, y, test size=0.2, stratify=y)
```

Especially important when:

- Classes are imbalanced
- Dataset is small
- Minority class has few samples

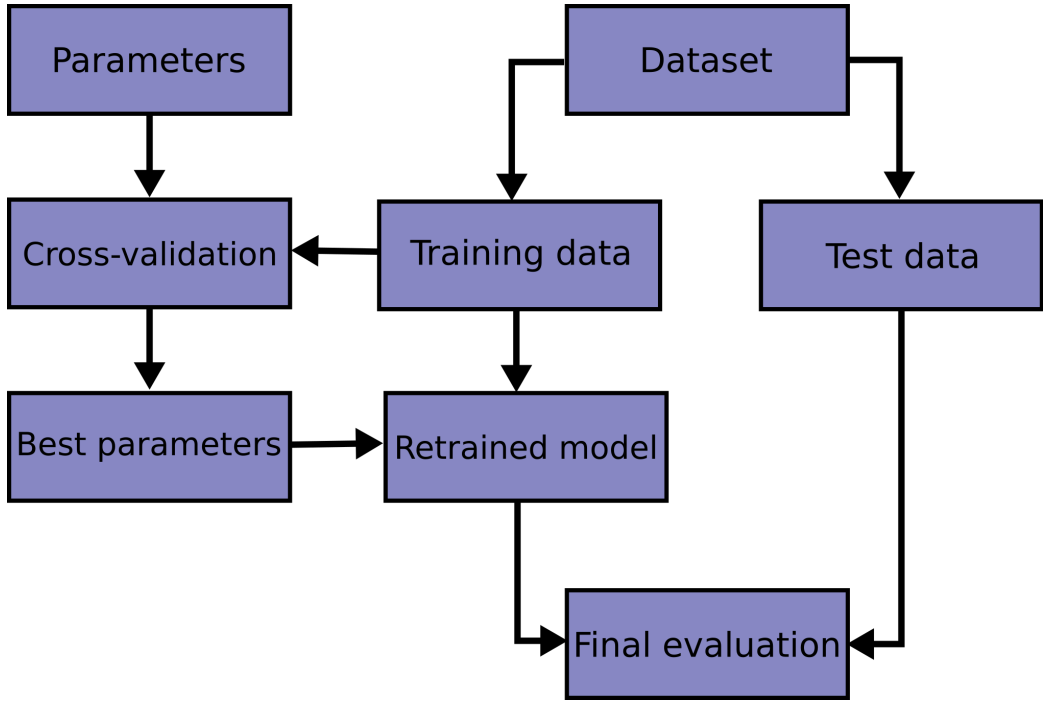
Pitfall 5: Ignoring the Test Set Rule

The test set should be used ONCE for final evaluation

Anti-pattern:

1. Train model, test → 80%
2. Tune hyperparameters, test → 82%
3. Try different model, test → 81%
4. More tuning, test → 84%
5. Report 84% accuracy ← WRONG!

Problem: You've overfit to the test set!



Solution: Use validation set or CV for tuning. Test only at the very end!

Pitfall Summary Checklist

Before claiming your model works:

- [ ] Scalers/encoders fitted on training data only
- [ ] Stratified splitting used for classification
- [ ] Cross-validation for performance estimates
- [ ] Appropriate metrics for imbalanced data
- [ ] Gap between train and test performance is reasonable
- [ ] Test set used only once for final evaluation

Section 10: Preparing for Coding Exercise 1

What's Coming Next

CE1: Adult Census Income Classification

What you'll implement:

1. Load and explore the Adult Census dataset
2. Handle missing values (impute)
3. Stratified train/test split
4. Scale numerical features (RobustScaler)
5. Encode categorical features (OneHotEncoder)
6. Train multiple classifiers
7. Evaluate with cross-validation
8. Compare models and select the best

Models You'll Train

Model	Type	Key Characteristic
DummyClassifier	Baseline	Always predicts most frequent class
LogisticRegression	Linear	Simple, interpretable
RandomForestClassifier	Ensemble	Robust, handles many features
GradientBoostingClassifier	Ensemble	Often best for tabular data
SVC	Kernel	Good with proper tuning

The Dummy baseline is crucial! If your model can't beat it, something is wrong.

What CE1 Deliberately Doesn't Cover

We use accuracy in CE1 to demonstrate its limitations

In Episode 6 and Coding Exercise 2 we'll cover:

- Handling class imbalance
- Better metrics (MCC, F1)
- Hyperparameter tuning
- Pipelines for leakage prevention
- Threshold optimization

> CE1 teaches the basics. CE2 teaches how to do it properly.

Key Takeaways

1. **sklearn** is your foundation for classical ML
2. **Datasets** have structure: features, targets, types
3. **Model choice** depends on data type and problem
4. **Missing data** must be addressed (drop or impute)
5. **Normalization** is critical for many algorithms
6. **Encoding** converts categories to numbers
7. **Stratification** preserves class balance
8. **Cross-validation** gives reliable performance estimates
9. **Watch for pitfalls:** leakage, imbalance, overfitting

Resources and Documentation

Official Documentation:

- scikit-learn User Guide
- scikit-learn API Reference

Tutorials:

- sklearn Getting Started
- pandas User Guide

Cheat Sheets:

- sklearn Algorithm Cheat Sheet

Questions Before CE1?

Up Next: Coding Exercise 1

You'll apply everything from this episode to build your first ML workflow!

- Work through at your own pace
- Run each cell in turn and **understand** the output
- Experiment with different parameters
- Compare your results to expected outputs

Remember: Making mistakes is learning. If something doesn't work, debug it!

Feel free to ask questions as you proceed.