Episode 6: Metrics, Pipelines & Hyperparameter Tuning

Building Production-Quality ML Workflows

AI & ML: From Theory to Practice

Preparing for Coding Exercise 2

Matthew Care 2026

Learning Objectives

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

- **Explain** why accuracy fails for imbalanced datasets
- **Calculate and interpret** Precision, Recall, F1, and MCC
- **Construct** sklearn Pipelines and ColumnTransformers
- **Prevent** data leakage through proper pipeline usage
- **Understand** hyperparameters vs parameters
- **Apply** Optuna for Bayesian hyperparameter optimization
- **Integrate** pipelines with Optuna for robust tuning
- **Analyze** hyperparameter importance and optimization results

Recap: The Problem We Left Unsolved

From Episode 5, the Adult Census dataset:

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

DummyClassifier accuracy: 76%

If our "real" model gets 82% accuracy, is it actually good?
Or just slightly better than guessing the majority class?

Section 1: Why Accuracy Fails

The Accuracy Paradox

What Accuracy Measures

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}} = \frac{TP + TN}{TP + TN + FP + FN}$$

Seems reasonable, right?

But consider: if 99% of emails are not spam, a model that **predicts "not spam" for everything** gets 99% accuracy.

It catches zero spam. This is useless!

The Accuracy Paradox Illustrated

Scenario: Fraud Detection (1% fraud rate)

Model A: "Never Fraud"

- Accuracy: **99%**
- Catches 0/100 frauds
- Useless for its purpose

Model B: Actual Predictor

- Accuracy: **95%**
- Catches 80/100 frauds
- Actually useful!

The paradox: Lower accuracy can mean better performance when classes are imbalanced.

When Accuracy Works (and Doesn't)

Situation	Accuracy Reliable?
Balanced classes (50/50)	Yes
Slightly imbalanced (60/40)	Mostly
Moderately imbalanced (75/25)	Caution
Highly imbalanced (95/5)	No
Extremely imbalanced (99/1)	Definitely not

Rule of thumb: If the majority class is more than 70% of data, be skeptical of accuracy alone.

Section 2: The Confusion Matrix

Foundation for Better Metrics

The Confusion Matrix

A 2×2 table of prediction outcomes:

	Predicted: Positive	Predicted: Negative
Actual: Positive	True Positive (TP)	False Negative (FN)
Actual: Negative	False Positive (FP)	True Negative (TN)

TP: Correctly predicted positive

TN: Correctly predicted negative

FP: Incorrectly predicted positive (**Type I error**)

FN: Incorrectly predicted negative (**Type II error**)

Confusion Matrix Example

Adult Census Income predictions:

	Predicted: >50K	Predicted: <=50K
Actual: >50K	TP = 1,500	FN = 850
Actual: <=50K	FP = 700	TN = 6,500

Total: 9,550 test samples

Accuracy:

$$\frac{1500+6500}{9550} = 83.8\%$$

But wait...

- We only caught 63.8% (1500/(1500+850)) of high earners
- That's a lot of missed positive cases!

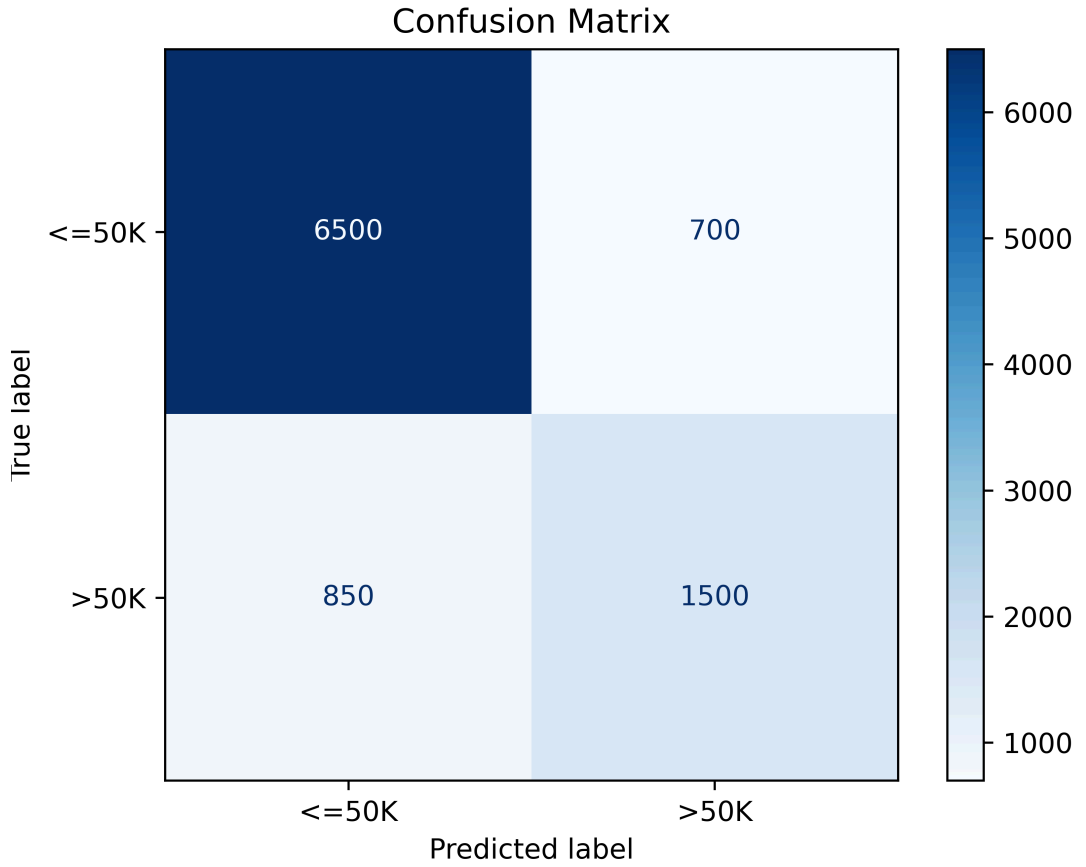
Visualizing the Confusion Matrix

```python
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Generate confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Visualize
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=['<=50K', '>50K'])
disp.plot(cmap='Blues')
plt.title('Confusion Matrix')
plt.show()
```

The visualization makes it easy to see where your model succeeds and fails.



Section 3: Precision and Recall

Measuring What Matters

Precision: How Trustworthy Are Positive Predictions?

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{\text{True Positives}}{\text{All Positive Predictions}}$$

"Of all samples I predicted as positive, how many actually were?"

High Precision Means:

- Few false alarms
- When you predict positive, you're usually right
- Conservative predictions

Precision Matters When:

- False positives are costly
- Spam filter (don't delete real email!)
- Fraud alerts (don't freeze legitimate accounts)

Precision is also called Positive Predictive Value (PPV)

Recall: How Complete Are We?

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{\text{True Positives}}{\text{All Actual Positives}}$$

"Of all actual positive samples, how many did I find?"

High Recall Means:

- Found most positive cases
- Few missed positives
- Thorough detection

Recall Matters When:

- Missing positives is costly
- Cancer screening (don't miss cancer!)
- Security threats (don't miss attacks!)

Recall is also called **Sensitivity** or **True Positive Rate (TPR)**

The Precision-Recall Trade-off

You usually can't maximize both:

More aggressive (lower threshold):

→ More positive predictions
→ Higher Recall (catch more positives)
→ Lower Precision (more false alarms)

More conservative (higher threshold):

→ Fewer positive predictions
→ Lower Recall (miss more positives)
→ Higher Precision (fewer false alarms)

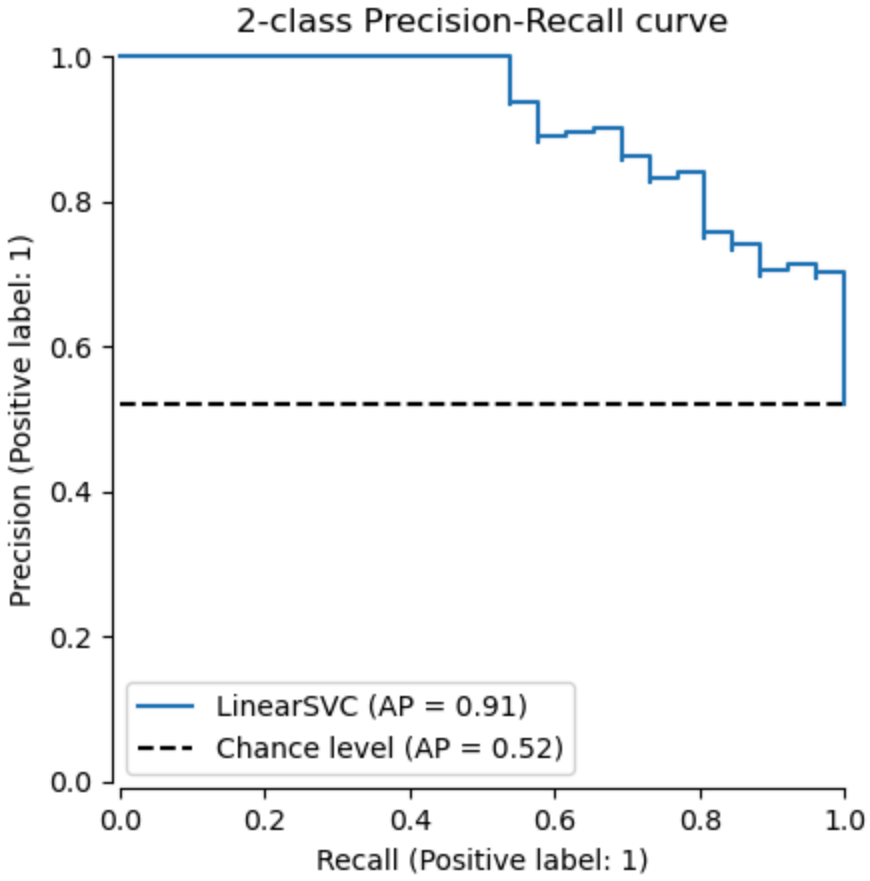
The trade-off is fundamental. You must decide what matters more for your application.

Precision-Recall Trade-off: Visual

Precision-Recall Curve:

- Each point is a different threshold
- Top-right corner is ideal (both high)
- Area under curve (AP/average precision) summarizes overall performance

Source: sklearn documentation



Precision vs Recall: Which to Prioritize?

Application	Priority	Reason
Email spam filter	Precision	Don't delete important emails
Cancer screening	Recall	Don't miss cancer cases
Fraud detection	Depends on cost	Balance false alarms vs missed fraud
Search engines	Precision	Top results should be relevant
Legal document discovery	Recall	Find all relevant documents

Ask: What's the cost of a false positive vs a false negative?

Calculating Precision and Recall in sklearn

```python
from sklearn.metrics import precision_score, recall_score

# Calculate for binary classification
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)

print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")

# For multi-class, specify averaging
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='macro')
```

Section 4: F1 Score

Balancing Precision and Recall

F1 Score: The Harmonic Mean

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}$$

Why harmonic mean, not arithmetic?

- Arithmetic mean: $\frac{0.9+0.1}{2} = 0.5$ (seems okay)
- Harmonic mean: $\frac{2 \cdot 0.9 \cdot 0.1}{0.9+0.1} = 0.18$ (reveals problem!)

> Harmonic mean **penalizes imbalance** between precision and recall.
> Both must be high for F1 to be high.

See https://en.wikipedia.org/wiki/F-score

F1 Score Behavior

Precision	Recall	F1 Score
0.90	0.90	0.90
0.90	0.70	0.79
0.90	0.50	0.64
0.90	0.30	0.45
0.90	0.10	0.18
0.50	0.50	0.50

F1 is high only when **both** precision and recall are high.
It's the geometric balance point.

F1 Score in sklearn

```python
from sklearn.metrics import f1_score

# Binary classification
f1 = f1_score(y_test, y_pred)
print(f"F1 Score: {f1:.4f}")

# Multi-class options
f1_macro = f1_score(y_test, y_pred, average='macro')    # Unweighted mean
f1_weighted = f1_score(y_test, y_pred, average='weighted')  # Weighted by support
f1_micro = f1_score(y_test, y_pred, average='micro')    # Global TP, FP, FN

# Get F1 for each class
f1 per class = f1 score(y test, y pred, average=None)
```

When F1 Falls Short

F1 ignores True Negatives:

$$F_1 = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}$$

Notice: TN doesn't appear!

Problem scenarios:

- When correctly identifying negatives is important
- When comparing models where TN differs significantly
- When you need a truly balanced view of all four cells

Solution: Matthews Correlation Coefficient (MCC)!

Section 5: Matthews Correlation Coefficient

A Balanced Measure for All Four Cells

Matthews Correlation Coefficient (MCC)

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Range: -1 to +1

- **+1:** Perfect prediction
- **0:** No better than random
- **-1:** Perfect inverse prediction

> MCC is a **correlation coefficient** between observed and predicted classes. It's balanced even when classes are very different sizes.

For MCC extension to multi-class problems see this paper. This is implemented in sklearn.

Why MCC is Excellent for Imbalanced Data

Adult Census Example:

Metric	Always Predict <=50K	Actual Model
Accuracy	76%	83%
F1 (>50K)	0%	58%
MCC	**0.00**	**0.52**

MCC correctly shows that always predicting the majority class has **zero predictive power** (MCC = 0), while accuracy makes it look reasonable (76%).

MCC vs Other Metrics

Property	Accuracy	F1	MCC
Uses all 4 cells	Yes	No (no TN)	Yes
Symmetric	Yes	No	Yes
Random = 0 (imbalanced)	No	No	Yes
Handles imbalance	No	Partially	Yes
Range	[0, 1]	[0, 1]	[-1, 1]

Recommendation: Use MCC as your primary metric for imbalanced binary classification. Supplement with precision/recall for interpretability.

MCC in sklearn

```python
from sklearn.metrics import matthews_corrcoef

mcc = matthews_corrcoef(y_test, y_pred)
print(f"MCC: {mcc:.4f}")

# In cross-validation
from sklearn.model_selection import cross_val_score

scores = cross_val_score(
    model, X, y,
    cv=5,
    scoring='matthews_corrcoef'  # Use MCC as scoring
)
print(f"MCC: {scores.mean():.3f} ± {scores.std():.3f}")
```

Section 6: Other Important Metrics

Completing the Metrics Toolkit

Balanced Accuracy

$$\text{Balanced Accuracy} = \frac{1}{2}\left(\frac{TP}{TP+FN} + \frac{TN}{TN+FP}\right) = \frac{\text{Sensitivity} + \text{Specificity}}{2}$$

Average of recall for each class

Metric	Always Predict Majority	Actual Model
Accuracy	76%	83%
Balanced Accuracy	**50%**	**74%**

Balanced accuracy is always 50% for random guessing, regardless of class distribution.

Specificity

$$\text{Specificity} = \frac{TN}{TN + FP} = \frac{\text{True Negatives}}{\text{All Actual Negatives}}$$

"Of all actual negatives, how many did I correctly identify?"

Complement of Recall: Recall for the negative class

Specificity is also called the **True Negative Rate (TNR)**
Specificity + Recall = Complete picture of both classes' identification rates.

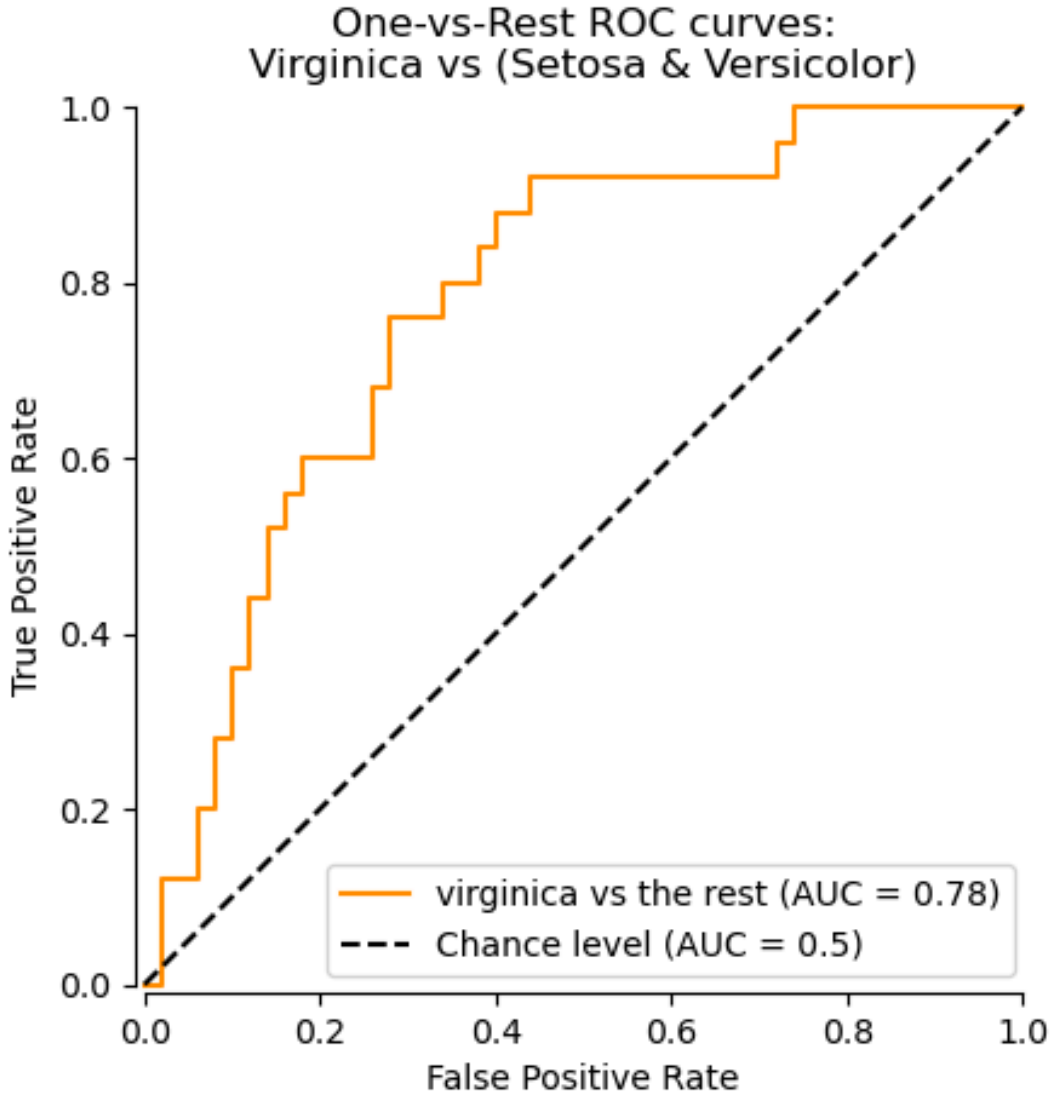
False Positive Rate (FPR) = 1 - Specificity (TNR)

ROC Curve and AUC

Receiver Operating Characteristic curve:

- X-axis: False Positive Rate (1 - Specificity)
- Y-axis: True Positive Rate (Recall)
- Each point = different threshold
- Diagonal = random classifier
- **AUC:** Area Under Curve (0.5-1.0)

Source: sklearn ROC documentation



ROC AUC vs PR AUC

Metric	Best For	Weakness
ROC AUC	Balanced/moderate imbalance	Optimistic for severe imbalance
PR AUC	Severe imbalance	Harder to interpret

ROC AUC:

- Range: 0.5 (random) to 1.0 (perfect)
- Well understood
- Can be overly optimistic when positives are rare

PR AUC (Average Precision):

- Focuses on positive class performance
- More sensitive to improvements in imbalanced settings
- Baseline varies with class distribution

Classification Report: All Metrics at Once

```
from sklearn.metrics import classification_report

print(classification report(y test, y pred, target names=['<=50K', '>50K']))
```

```
              precision    recall  f1-score   support

       <=50K       0.89      0.92      0.91      7500
        >50K       0.68      0.61      0.64      2350

    accuracy                           0.84      9850
   macro avg       0.79      0.77      0.77      9850
weighted avg       0.84      0.84      0.84      9850
```

support is the number of occurrences of each class in y_true.

Metric Selection Summary

Situation	Recommended Metrics
Balanced classes	Accuracy, F1, MCC (all similar)
Moderate imbalance	MCC, Balanced Accuracy, F1
Severe imbalance	MCC, PR AUC, Class-specific F1
False positives costly	Precision, Specificity
False negatives costly	Recall, Sensitivity
Need probability ranking	ROC AUC, PR AUC
Business costs known	Custom cost-benefit metric

Section 7: sklearn Pipelines

Preventing Data Leakage Through Proper Workflow

The Data Leakage Problem

Within CE1:

```
# Scale all data
scaler.fit(X)  # WRONG - Sees test data!
X_scaled = scaler.transform(X)

# Then split
X_train, X_test = train_test_split(X_scaled, ...)

# Or during cross-validation <-- CE1 approach
scaler.fit(X train)  # WRONG - Sees validation folds!
```

The scaler has seen data it shouldn't. Test/validation statistics influence training.

What is a Pipeline?

A Pipeline chains preprocessing and modelling into a single object:

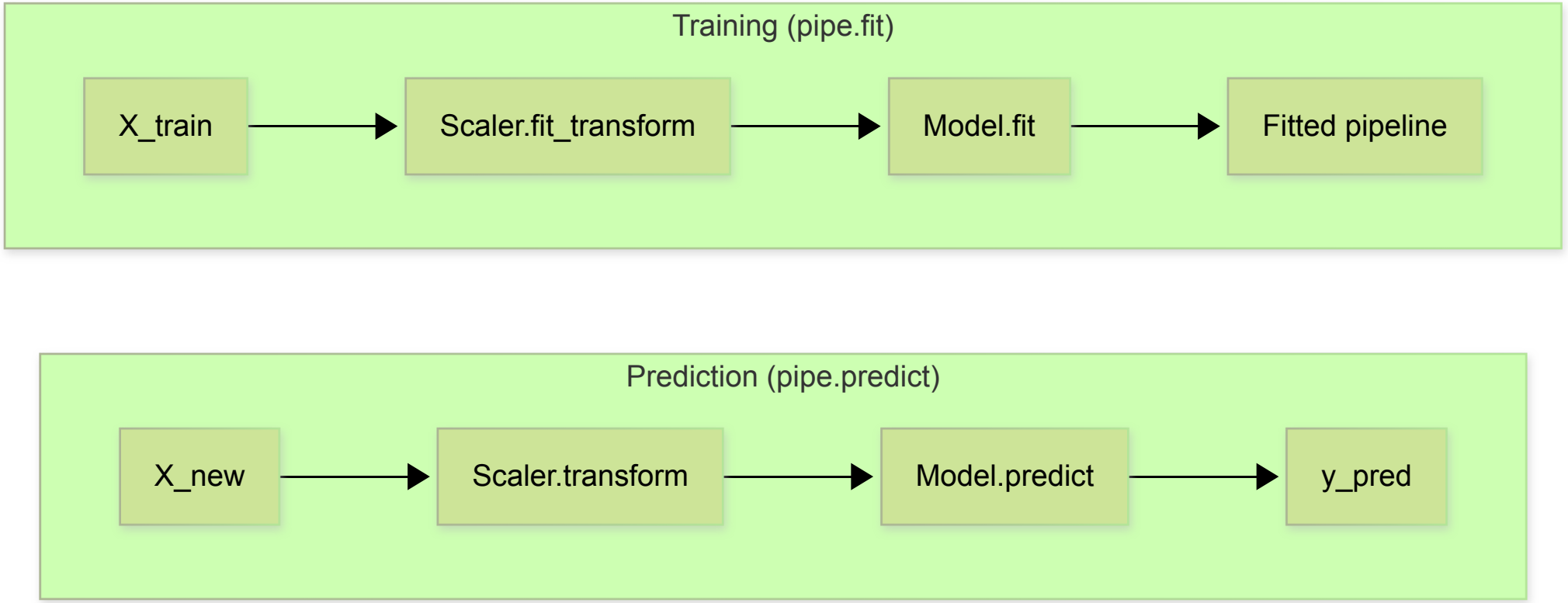
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression())
])

# Fit and predict as if it's a single model
pipe.fit(X_train, y_train)
predictions = pipe.predict(X test)
```

The pipeline ensures that each step is fit only on training data, **even during cross-validation**.

Pipeline Execution Flow



Key difference:

- Training: `fit_transform` (learns parameters)
- Prediction: `transform` only (uses learned parameters)

Pipelines + Cross-Validation

```
from sklearn.model_selection import cross_val_score

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression())
])

# Cross-validation handles everything correctly!
scores = cross val score(pipe, X, y, cv=5, scoring='accuracy')
```

Inside each fold:

1. Split data into train/validation
2. Fit scaler on **train fold only**
3. Transform train and validation with fitted scaler
4. Fit classifier on transformed train
5. Score on transformed validation

ColumnTransformer: Different Preprocessing per Column

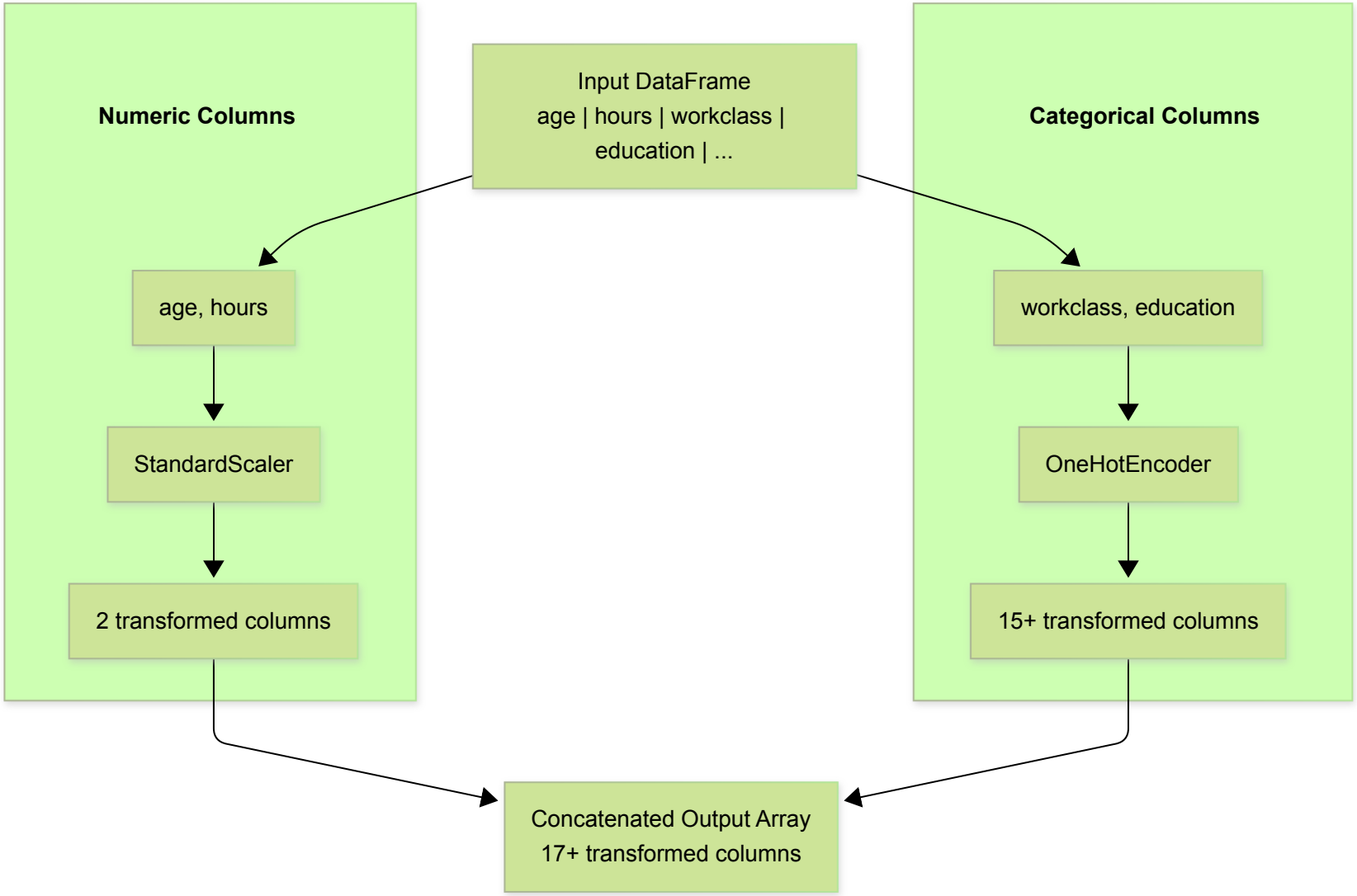
Problem: Numerical and categorical columns need different preprocessing

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), ['age', 'hours-per-week']),
        ('cat', OneHotEncoder(), ['workclass', 'education'])
    ]
)
```

ColumnTransformer applies different transformers to different columns.

ColumnTransformer Visualization



Building a Complete Pipeline

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier

# Define column groups
numeric_features = ['age', 'hours-per-week', 'capital-gain']
categorical_features = ['workclass', 'education', 'occupation']

# Preprocessing for each type
numeric_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('encoder', OneHotEncoder(handle_unknown='ignore'))
])
```

Complete Pipeline Assembly

Wrap `numeric_transformer` and `categorical_transformer` pipelines in `preprocessor` pipeline:

```
# Combine preprocessing steps
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ]
)

# Full pipeline with classifier
full_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Use like any sklearn estimator
full_pipeline.fit(X_train, y_train)
y_pred = full_pipeline.predict(X_test)
score = full_pipeline.score(X_test, y_test)
```

Pipeline Benefits Summary

Correctness

- No data leakage in CV
- Proper fit/transform separation
- Consistent preprocessing

Convenience

- Single object to save/load
- Works with all sklearn tools
- Easy hyperparameter tuning

Best Practice: Always use pipelines in production code. They prevent subtle bugs and simplify deployment.

Section 8: Hyperparameter Tuning

Finding Optimal Model Configurations

Parameters vs Hyperparameters

Parameters

- **Learned from data**
- Set during training (fit)
- Examples:
  - Linear regression coefficients
  - Neural network weights
  - Tree split thresholds

Hyperparameters

- **Set before training**
- Control learning process
- Examples:
  - Learning rate
  - Number of trees
  - Regularization strength
  - Max depth

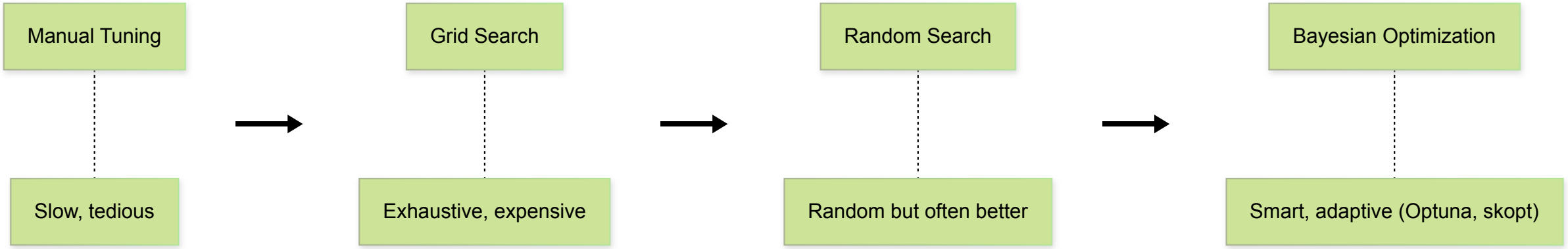
Hyperparameter tuning = finding the best hyperparameter values for your data.

Common Hyperparameters by Algorithm

Algorithm	Key Hyperparameters
Logistic Regression	C (regularization), penalty type
Random Forest	n_estimators, max_depth, min_samples_split
Gradient Boosting	n_estimators, learning_rate, max_depth
SVM	C, kernel, gamma
Neural Networks	layers, units, learning_rate, dropout

Focus on the most impactful hyperparameters first. Documentation usually indicates which matter most.

Tuning Approaches



Method	Trials for 4 params × 5 values	Intelligence
Grid Search	$5^4 = 625$	None
Random Search	~60-100	None
Bayesian (**Optuna**)	~30-50	Learns from previous trials

Grid Search: Exhaustive but Expensive

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'classifier__n_estimators': [50, 100, 200],
    'classifier__max_depth': [5, 10, 20, None],
    'classifier__min_samples_split': [2, 5, 10]
}

grid_search = GridSearchCV(
    pipeline,
    param_grid,
    cv=5,
    scoring='matthews_corrcoef',
    n_jobs=-1
)

grid_search.fit(X_train, y_train)
print(f"Best params: {grid_search.best_params_}")
print(f"Best score: {grid_search.best_score_:.4f}")
```

Random Search: Often Better than Grid

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint, uniform

param_distributions = {
    'classifier__n_estimators': randint(50, 300),
    'classifier__max_depth': randint(3, 30),
    'classifier__min_samples_split': randint(2, 20),
    'classifier__learning_rate': uniform(0.01, 0.3)
}

random_search = RandomizedSearchCV(
    pipeline,
    param_distributions,
    n_iter=50,  # Number of random combinations
    cv=5,
    scoring='matthews_corrcoef',
    random_state=42
)
```

Section 9: Introduction to Optuna

Intelligent Hyperparameter Optimization

What is Optuna?

A next-generation hyperparameter optimization framework:

- **Bayesian optimization** via TPE (Tree-structured Parzen Estimator)
- **Pruning** of unpromising trials (early stopping)
- **Visualization** of optimization results
- **Distributed** optimization support
- **Database** storage for resumable studies

```
pip install optuna
```

Documentation: optuna.org

Optuna Concepts

```
Study
  └── contains many Trials
        └── each Trial tests specific hyperparameters
              └── returns an objective value (e.g., MCC)

Sampler: chooses hyperparameters (TPE by default)
Pruner: stops unpromising trials early
```

Term	Meaning
Study	Complete optimization session
Trial	Single hyperparameter evaluation
Objective	Function to optimize (returns score)
Sampler	Algorithm choosing next parameters

Basic Optuna Example

```python
import optuna

def objective(trial):
    # Suggest hyperparameters
    n_estimators = trial.suggest_int('n_estimators', 50, 300)
    max_depth = trial.suggest_int('max_depth', 3, 20)
    learning_rate = trial.suggest_float('learning_rate', 0.01, 0.3, log=True)

    # Build model with suggested params
    model = GradientBoostingClassifier(
        n_estimators=n_estimators,
        max_depth=max_depth,
        learning_rate=learning_rate
    )

    # Evaluate with cross-validation
    scores = cross_val_score(model, X_train, y_train, cv=5,
                             scoring='matthews_corrcoef')
    return scores.mean()

# Run optimization
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=50)
```

Optuna suggest_* Methods

```
# Integer values
n_estimators = trial.suggest_int('n_estimators', 50, 300)
n_estimators = trial.suggest_int('n_estimators', 50, 300, step=10)

# Float values
learning_rate = trial.suggest_float('learning_rate', 1e-5, 1e-1, log=True)
dropout = trial.suggest_float('dropout', 0.1, 0.5)

# Categorical choices
optimizer = trial.suggest_categorical('optimizer', ['adam', 'sgd', 'rmsprop'])

# Conditional parameters
if optimizer == 'sgd':
    momentum = trial.suggest float('momentum', 0.0, 0.99)
```

Use `log=True` for parameters that span orders of magnitude (like learning rate).

TPE: How Optuna Learns

Tree-structured Parzen Estimator:

1. **Divide trials** into good (above median) and bad (below)
2. **Model distributions** of hyperparameters for each group
3. **Sample new parameters** likely to be good (high in good distribution, low in bad)
4. **Update** as more trials complete

Initial random exploration (`n_startup_trials` ; default==10):

```
Trial 1: random
Trial 2: random
Trial 3: random
...
Trial 10+: informed by previous results
```

Pruning: Stop Bad Trials Early

```python
from optuna.pruners import MedianPruner

study = optuna.create_study(
    direction='maximize',
    pruner=MedianPruner()  # Stop if below median at same step
)

def objective(trial):
    model = SomeIterativeModel(...)

    for epoch in range(100):
        model.train_one_epoch()
        intermediate_score = model.validate()

        # Report and check for pruning
        trial.report(intermediate_score, epoch)
        if trial.should_prune():
            raise optuna.TrialPruned()

    return model.final_score()
```

Early stopping isn't available for most sklearn models, see documentation for details. Models such as XGBoost/LightGBM support early stopping out of the box.

Section 10: Optuna + Pipelines

Integrating Everything Together

Tuning Pipeline Parameters with Optuna

Make **objective function** that wraps an sklearn pipeline

```
def objective(trial):
    # Preprocessing parameters
    scaler_type = trial.suggest_categorical('scaler',
                                            ['standard', 'robust', 'minmax'])

    # Model parameters
    n_estimators = trial.suggest_int('n_estimators', 50, 300)
    max_depth = trial.suggest_int('max_depth', 3, 20)

    # Build pipeline with suggested params
    if scaler_type == 'standard':
        scaler = StandardScaler()
    elif scaler_type == 'robust':
        scaler = RobustScaler()
    else:
        scaler = MinMaxScaler()

    pipeline = Pipeline([
        ('scaler', scaler),
        ('classifier', RandomForestClassifier(
            n_estimators=n_estimators,
            max_depth=max_depth
        ))
    ])

    # Cross-validate and return mean score
    scores = cross_val_score(pipeline, X_train, y_train,
                             cv=5, scoring='matthews_corrcoef')
    return scores.mean()
```

Tuning Multiple Models

Can also choose the **model type** AND their **hyperparameters** at the same time

```
def objective(trial):
    # Choose model type
    model_name = trial.suggest_categorical('model',
                    ['logistic', 'rf', 'gb', 'svm'])

    # Model-specific hyperparameters
    if model_name == 'logistic':
        C = trial.suggest_float('lr_C', 0.001, 100, log=True)
        model = LogisticRegression(C=C, max_iter=1000)

    elif model_name == 'rf':
        n_est = trial.suggest_int('rf_n_estimators', 50, 300)
        max_depth = trial.suggest_int('rf_max_depth', 3, 20)
        model = RandomForestClassifier(n_estimators=n_est,
                                       max_depth=max_depth)
    elif model_name == 'gb':
        # ... Gradient Boosting params
    elif model_name == 'svm':
        # ... SVM params

    pipeline = make_pipeline(preprocessor, model)
    return cross_val_score(pipeline, X_train, y_train,
                           cv=5, scoring='matthews_corrcoef').mean()
```

Sampling Strategies with Optuna

In CE2 (advance/expert versions), we also tune resampling strategies for imbalanced data:

```python
from imblearn.over_sampling import SMOTE, ADASYN
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline as ImbPipeline

def objective(trial):
    sampling = trial.suggest_categorical('sampling',
                ['none', 'smote', 'adasyn', 'undersample'])

    if sampling == 'smote':
        sampler = SMOTE(random_state=42)
    elif sampling == 'adasyn':
        sampler = ADASYN(random_state=42)
    elif sampling == 'undersample':
        sampler = RandomUnderSampler(random_state=42)
    else:
        sampler = None

    # Use imbalanced-learn Pipeline if sampler needed
    if sampler:
        pipeline = ImbPipeline([
            ('preprocess', preprocessor),
            ('sampler', sampler),
            ('classifier', model)
        ])
    else:
        pipeline = Pipeline([...])
```

Retrieving Best Parameters

```
# Run optimization
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)

# Get best results
print(f"Best MCC: {study.best_value:.4f}")
print(f"Best params: {study.best_params}")

# Build final model with best params
best_params = study.best_params
final_model = build_model_from_params(best_params)
final_model.fit(X_train, y_train)

# Evaluate on test set (only once!)
test score = final model.score(X test, y test)
```

Analyzing Hyperparameter Importance

```python
from optuna.importance import get_param_importances

# Get parameter importance
importance = optuna.importance.get_param_importances(study)
print("Parameter importance:")
for param, imp in importance.items():
    print(f"  {param}: {imp:.4f}")
```

Output:

```
Parameter importance:
  learning_rate: 0.3521
  n_estimators: 0.2847
  max_depth: 0.2134
  min_samples_split: 0.0892
```

Use importance to understand which hyperparameters matter. Focus future tuning on important ones.

Optuna Visualization

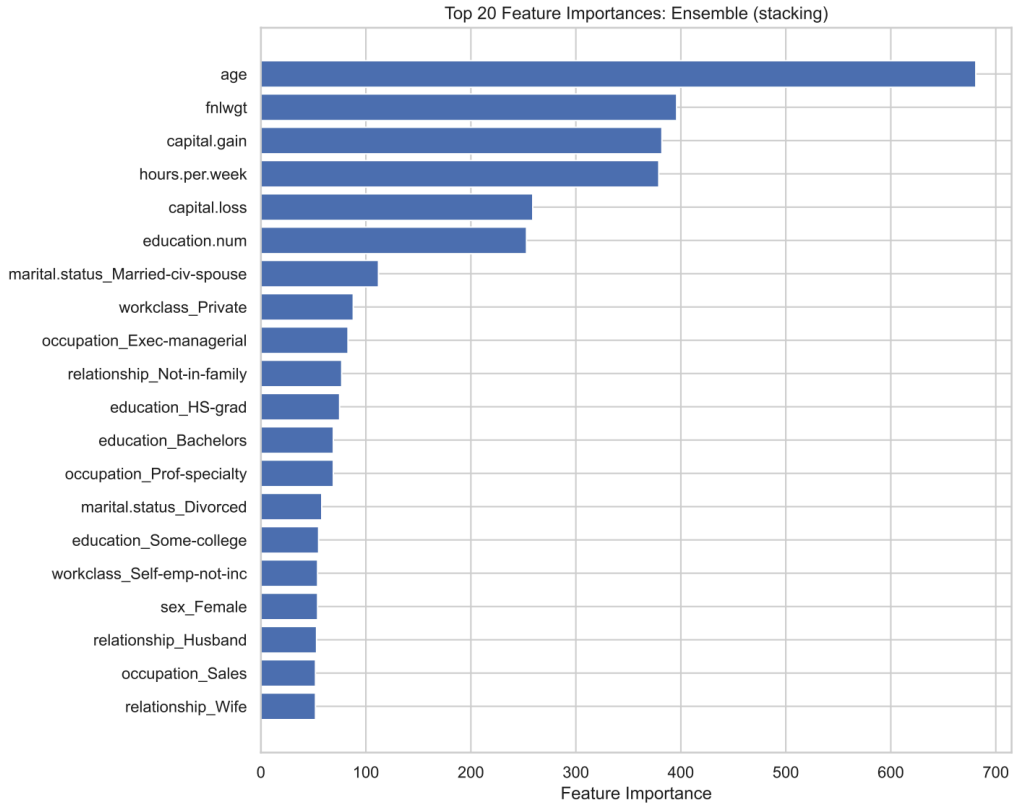
```python
import optuna.visualization as vis

# Optimization history
vis.plot_optimization_history(study)

# Parameter importance
vis.plot_param_importances(study)

# Parallel coordinate plot (parameter relationships)
vis.plot_parallel_coordinate(study)

# Hyperparameter slice plots
vis.plot_slice(study)
```



Section 11: Threshold Optimization

Beyond the Default 0.5

Why Optimize the Threshold?

Default threshold of 0.5 assumes:

- Equal costs for FP and FN
- Balanced class distribution
- Well-calibrated probabilities

Reality:

- Costs are rarely equal
- Classes are often imbalanced
- Probabilities may not be calibrated

> By adjusting the threshold, we can trade off precision vs recall to match our actual priorities.

Threshold Effect on Metrics

```python
# Get probabilities instead of predictions
y_proba = model.predict_proba(X_test)[:, 1]

# Try different thresholds
for threshold in [0.3, 0.4, 0.5, 0.6, 0.7]:
    y_pred = (y_proba >= threshold).astype(int)
    mcc = matthews_corrcoef(y_test, y_pred)
    prec = precision_score(y_test, y_pred)
    rec = recall_score(y_test, y_pred)
    print(f"Threshold {threshold}: MCC={mcc:.3f}, Prec={prec:.3f}, Rec={rec:.3f}")

# Thresholds
Threshold 0.3: MCC=0.485, Prec=0.52, Rec=0.78  # High recall
Threshold 0.5: MCC=0.520, Prec=0.68, Rec=0.61  # Balanced
Threshold 0.7: MCC=0.478, Prec=0.82, Rec=0.42  # High precision
```

Finding Optimal Threshold

```python
import numpy as np
from sklearn.metrics import matthews_corrcoef

thresholds = np.linspace(0.1, 0.9, 81)
best_thresh = 0.5
best_mcc = 0

for thresh in thresholds:
    y_pred = (y_proba >= thresh).astype(int)
    mcc = matthews_corrcoef(y_test, y_pred)
    if mcc > best_mcc:
        best_mcc = mcc
        best_thresh = thresh

print(f"Optimal threshold: {best_thresh:.3f}")
print(f"Best MCC: {best mcc:.4f}")
```

Important: Optimize threshold on validation data, not test data!

Here (and in CE2) **we are** calculating optimized threshold on test data. This would then need to be validated on and additional dataset. Alternatively optimize the threshold using cross-validation.

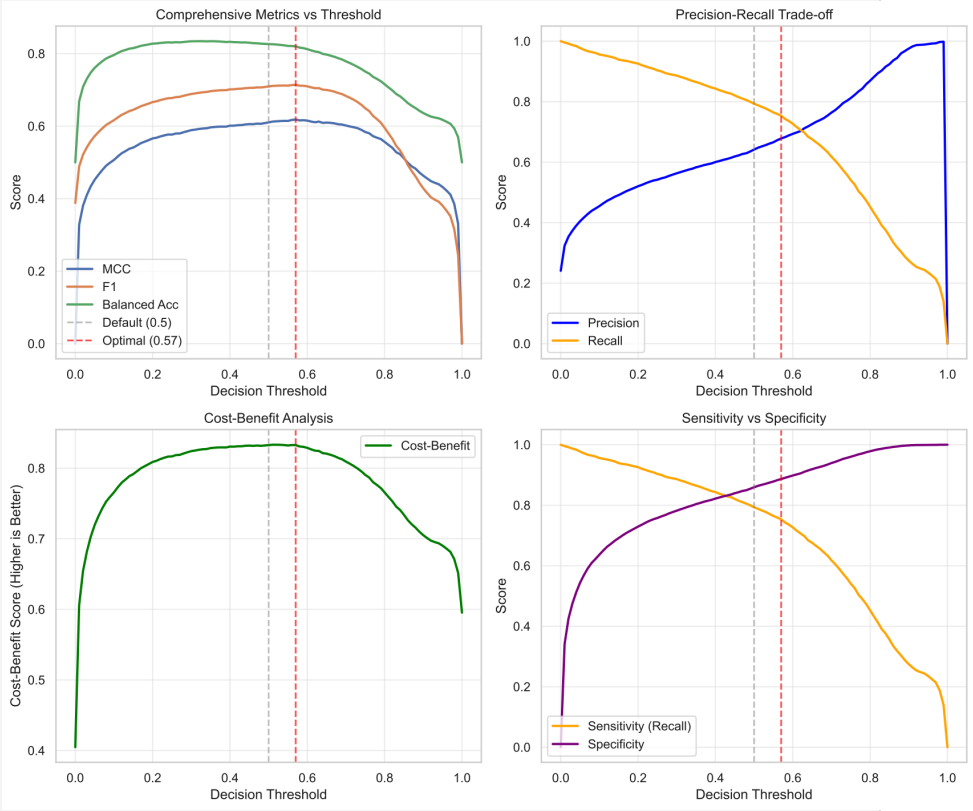
Threshold Visualization

```python
import matplotlib.pyplot as plt

thresholds = np.linspace(0.0, 1.0, 101)
mccs = []
f1s = []
precisions = []
recalls = []

for thresh in thresholds:
    y_pred = (y_proba >= thresh).astype(int)
    mccs.append(matthews_corrcoef(y_test, y_pred))
    f1s.append(f1_score(y_test, y_pred))
    precisions.append(precision_score(y_test, y_pred, zero_division=0))
    recalls.append(recall_score(y_test, y_pred))

plt.figure(figsize=(10, 6))
plt.plot(thresholds, mccs, label='MCC')
plt.plot(thresholds, f1s, label='F1')
plt.plot(thresholds, precisions, label='Precision')
plt.plot(thresholds, recalls, label='Recall')
plt.axvline(0.5, color='gray', linestyle='--', label='Default (0.5)')
plt.xlabel('Threshold')
plt.ylabel('Score')
plt.legend()
plt.title('Metrics vs Decision Threshold')
```



Section 12: Ensemble Methods Preview

Combining Models for Better Performance

Ensemble Principles

Why ensembles work:

- Different models make different errors
- Combining predictions averages out individual mistakes
- Diversity among models is key

Two main approaches:

Method	How It Works
Voting	Average predictions (soft) or majority vote (hard)
Stacking	Train a meta-model on base model predictions

VotingClassifier

Soft voting averages the predicted probabilities, **hard voting** just counts votes.

```
from sklearn.ensemble import VotingClassifier

# Base models
models = [
    ('lr', LogisticRegression()),
    ('rf', RandomForestClassifier()),
    ('gb', GradientBoostingClassifier())
]

# Soft voting (average probabilities)
ensemble = VotingClassifier(estimators=models, voting='soft')

# Hard voting (majority vote)
ensemble = VotingClassifier(estimators=models, voting='hard')

ensemble.fit(X_train, y_train)
predictions = ensemble.predict(X test)
```

Soft voting usually works better because it uses prediction confidence.

StackingClassifier

Stacking trains a meta-model to learn how to best combine base model predictions

```
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression

# Base models
estimators = [
    ('rf', RandomForestClassifier()),
    ('gb', GradientBoostingClassifier()),
    ('svm', SVC(probability=True))
]

# Meta-learner combines base predictions
stacking = StackingClassifier(
    estimators=estimators,
    final_estimator=LogisticRegression(),
    cv=5  # Use CV to prevent overfitting
)

stacking.fit(X train, y train)
```

When Ensembles Help

Ensembles Help When:

- Base models are strong but diverse
- Models make uncorrelated errors
- Sufficient training data
- Acceptable computation cost

Ensembles Don't Help When:

- Base models are all similar
- One model dominates
- Limited computation budget
- Interpretability required

Ensembling weak models rarely helps. Focus on making individual models strong first.

Section 13: Preparing for Coding Exercise 2

What's Coming Next

CE2: Production-Quality Pipeline

Building on CE1, you'll implement:

3 scripts, with increasing complexity

All

1. Full sklearn Pipeline with ColumnTransformer
2. Multiple metrics: MCC, F1, balanced accuracy, ROC AUC
3. Optuna hyperparameter tuning across models

Advanced/Expert

4. Resampling strategies (SMOTE, ADASYN, undersampling)
5. Threshold optimization

Expert

6. Hyperparameter importance analysis
7. Ensemble methods (voting, stacking)
8. Probability calibration analysis
9. Learning curves for overfitting diagnosis

CE2 Configuration Options

```
# Tuning objective - what to optimize
TUNING_OBJECTIVE = "mcc"  # Options: mcc, f1, balanced_accuracy

# Quick mode for testing
QUICK_MODE = True  # Fewer trials, smaller search space

# Ensemble settings
ENSEMBLE_METHOD = "stacking"  # Options: voting, stacking, None
TOP_N_MODELS_FOR_ENSEMBLE = 3

# Threshold optimization
USE OPTIMAL THRESHOLD = True
```

Start with `QUICK_MODE = True` to test the pipeline, then run full optimization.

Most importantly, play/experiment/break. Breaking/fixing is a great way to learn.

Key Differences: CE1 vs CE2

Aspect	CE1	CE2
Preprocessing	Manual	Pipeline
Primary metric	Accuracy	MCC
Hyperparameter tuning	None	Optuna
Class imbalance	Ignored	Resampling strategies
Threshold	Default 0.5	Optimized (*but with test!*)
Model comparison	Simple	Comprehensive multi-metric
Visualization	Basic	Learning curves, threshold plots

Key Takeaways

1. **Accuracy fails** for imbalanced data - use MCC, F1, balanced accuracy
2. **Confusion matrix** is the foundation for all metrics
3. **Precision-Recall trade-off** - you can't maximize both
4. **MCC** is the best single metric for imbalanced binary classification
5. **Pipelines** prevent data leakage and simplify workflows
6. **Optuna** provides intelligent hyperparameter optimization
7. **Threshold optimization** can improve performance beyond 0.5
8. **Ensembles** combine strong, diverse models effectively

Resources and Documentation

Metrics:

- sklearn Metrics Guide
- MCC Wikipedia

Pipelines:

- sklearn Pipeline User Guide
- ColumnTransformer

Optuna:

- Optuna Documentation
- Optuna Tutorial

Imbalanced Learning:

- imbalanced-learn Documentation

Questions Before CE2?

Up Next: Coding Exercise 2

You'll build a complete, production-quality ML pipeline!

- Expect it to take longer than CE1
- Read the detailed documentation in the notebook
- Experiment with different configurations
- Compare metrics to understand their behavior

Remember: This is what real ML engineering looks like. Master these techniques and you can tackle any tabular ML problem.