ITAI 1371 Module 11: Hyperparameter Tuning & AutoML

Welcome to Module 11 of ITAI 1371! In this notebook, we'll dive deep into **hyperparameter tuning** and explore the exciting world of **AutoML** using classic machine learning methods. Our goal is to understand how tuning hyperparameters affects model performance and how automated tools can help us find the best models.

We'll work with the **Wine Quality dataset**, transforming it into a binary classification problem (good vs bad wine). Throughout the lab, you'll find detailed explanations, visualizations, and exercises to build your skills.

Part 1 - Pre-coded Sections with Explanations

1. Header & Introduction

In this section, we'll outline the goals and structure of today's lab.

Goals for this module:

- Understand what hyperparameters are and why they matter.
- Learn the difference between parameters and hyperparameters.
- Explore the bias-variance tradeoff and how hyperparameters influence it.
- Use cross-validation to evaluate models properly.
- Perform manual hyperparameter tuning and automated methods like Grid Search and Random Search.
- Try out AutoML using AutoGluon and compare results.

Dataset:

- Wine Quality dataset (binary classification: good vs bad wine).

Let's get started!

Deep Dive: Hyperparameters in Different Algorithms

Different machine learning algorithms have different hyperparameters. Understanding what each hyperparameter controls is crucial for effective tuning. Let's explore hyperparameters across several common algorithms:

Random Forest:

- `n_estimators`: Number of trees in the forest. More trees generally improve performance but increase computation time.
- `max_depth`: Maximum depth of each tree. Controls model complexity and overfitting.
- `min_samples_split`: Minimum samples required to split a node. Higher values prevent overfitting.
- `max_features`: Number of features to consider for each split. Adds randomness to reduce overfitting.

Support Vector Machine (SVM):

- `C`: Regularization parameter. Lower values mean stronger regularization.
- `kernel`: Type of kernel function (linear, rbf, poly). Determines decision boundary shape.
- `gamma`: Kernel coefficient for RBF. Controls influence of single training examples.

Logistic Regression:

- `C`: Inverse regularization strength. Smaller values mean stronger regularization.
- `penalty`: Type of regularization (l1, l2, elasticnet).
- `solver`: Algorithm for optimization (lbfgs, saga, etc.).

```
# Example: Viewing default hyperparameters
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression

# Create models with default hyperparameters
rf = RandomForestClassifier()
svm = SVC()
```

```
lr = LogisticRegression()

print("Random Forest Default Hyperparameters:")
print(f"  n_estimators: {rf.n_estimators}")
print(f"  max_depth: {rf.max_depth}")
print(f"  min_samples_split: {rf.min_samples_split}")
print(f"  max_features: {rf.max_features}")

print("\nSVM Default Hyperparameters:")
print(f"  C: {svm.C}")
print(f"  kernel: {svm.kernel}")
print(f"  gamma: {svm.gamma}")

print("\nLogistic Regression Default Hyperparameters:")
print(f"  C: {lr.C}")
print(f"  penalty: {lr.penalty}")
print(f"  solver: {lr.solver}")
```

```
Random Forest Default Hyperparameters:
  n_estimators: 100
  max_depth: None
  min_samples_split: 2
  max_features: sqrt

SVM Default Hyperparameters:
  C: 1.0
  kernel: rbf
  gamma: scale

Logistic Regression Default Hyperparameters:
  C: 1.0
  penalty: l2
  solver: lbfgs
```

🤔 Knowledge Check: Hyperparameter Understanding

Question 1 (Conceptual): If you wanted to reduce overfitting in a Random Forest model, which hyperparameters would you adjust and in what direction?

Your answer: [Write your answer here]

Question 2 (Reflective): 💭 Why do you think scikit-learn chose these specific default values? What assumptions might they be making about typical datasets?

Your answer: [Write your answer here]

Conceptual Question:

- What do you think hyperparameters are and why might tuning them be important for machine learning models?

Deep Dive: Hyperparameters in Different Algorithms

Different machine learning algorithms have different hyperparameters. Understanding what each hyperparameter controls is crucial for effective tuning. Let's explore hyperparameters across several common algorithms:

Random Forest:

- `n_estimators`: Number of trees in the forest. More trees generally improve performance but increase computation time.
- `max_depth`: Maximum depth of each tree. Controls model complexity and overfitting.
- `min_samples_split`: Minimum samples required to split a node. Higher values prevent overfitting.
- `max_features`: Number of features to consider for each split. Adds randomness to reduce overfitting.

Support Vector Machine (SVM):

- `C`: Regularization parameter. Lower values mean stronger regularization.
- `kernel`: Type of kernel function (linear, rbf, poly). Determines decision boundary shape.
- `gamma`: Kernel coefficient for RBF. Controls influence of single training examples.

Logistic Regression:

- `C`: Inverse regularization strength. Smaller values mean stronger regularization.
- `penalty`: Type of regularization (l1, l2, elasticnet).

- `solver`: Algorithm for optimization (lbfgs, saga, etc.).

```
# Example: Viewing default hyperparameters
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression

# Create models with default hyperparameters
rf = RandomForestClassifier()
svm = SVC()
lr = LogisticRegression()

print("Random Forest Default Hyperparameters:")
print(f"  n_estimators: {rf.n_estimators}")
print(f"  max_depth: {rf.max_depth}")
print(f"  min_samples_split: {rf.min_samples_split}")
print(f"  max_features: {rf.max_features}")

print("\nSVM Default Hyperparameters:")
print(f"  C: {svm.C}")
print(f"  kernel: {svm.kernel}")
print(f"  gamma: {svm.gamma}")

print("\nLogistic Regression Default Hyperparameters:")
print(f"  C: {lr.C}")
print(f"  penalty: {lr.penalty}")
print(f"  solver: {lr.solver}")
```

```
Random Forest Default Hyperparameters:
  n_estimators: 100
  max_depth: None
  min_samples_split: 2
  max_features: sqrt

SVM Default Hyperparameters:
  C: 1.0
  kernel: rbf
  gamma: scale

Logistic Regression Default Hyperparameters:
  C: 1.0
  penalty: l2
  solver: lbfgs
```

🤔 Knowledge Check: Hyperparameter Understanding

Question 1 (Conceptual): If you wanted to reduce overfitting in a Random Forest model, which hyperparameters would you adjust and in what direction?

Your answer: [Write your answer here]

Question 2 (Reflective): 💭 Why do you think scikit-learn chose these specific default values? What assumptions might they be making about typical datasets?

Your answer: [Write your answer here]

---

2. Setup & Data Loading

Let's load the Wine Quality dataset and prepare it for binary classification.

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

import warnings
warnings.filterwarnings('ignore')
```

```
# Load the Wine Quality dataset
# Dataset link: https://archive.ics.uci.edu/ml/datasets/Wine+Quality
# We'll use the red wine dataset

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv'
wine_df = pd.read_csv(url, sep=';')
wine_df.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6

Dataset Description:

- 1599 samples
- 11 physicochemical features (e.g., acidity, sugar, pH)
- Target: quality score (0-10 scale)

Our goal: convert quality into a **binary classification** problem:

- Quality >= 7 -> Good wine (label 1)
- Quality < 7 -> Bad wine (label 0)

```
# Create binary target
wine_df['good_quality'] = (wine_df['quality'] >= 7).astype(int)

# Drop original quality column
wine_df = wine_df.drop('quality', axis=1)

# Check distribution
wine_df['good_quality'].value_counts(normalize=True)
```

good_quality	proportion
0	0.86429
1	0.13571

dtype: float64

The dataset is imbalanced, with fewer good quality wines.

Split dataset into train and test sets

```
X = wine_df.drop('good_quality', axis=1)
y = wine_df['good_quality']

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

print(f"Training samples: {X_train.shape[0]}")
print(f"Testing samples: {X_test.shape[0]}")
```

```
Training samples: 1279
Testing samples: 320
```

Feature Scaling

Many ML algorithms perform better with scaled features.

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

```
X_test_scaled = scaler.transform(X_test)
```

---

3. Understanding Hyperparameters (with examples)

Let's clarify what hyperparameters are by looking at two classic models: Logistic Regression and Random Forest.

Deep Dive: Hyperparameters in Different Algorithms

Different machine learning algorithms have different hyperparameters. Understanding what each hyperparameter controls is crucial for effective tuning. Let's explore hyperparameters across several common algorithms:

Random Forest:

- `n_estimators`: Number of trees in the forest. More trees generally improve performance but increase computation time.
- `max_depth`: Maximum depth of each tree. Controls model complexity and overfitting.
- `min_samples_split`: Minimum samples required to split a node. Higher values prevent overfitting.
- `max_features`: Number of features to consider for each split. Adds randomness to reduce overfitting.

Support Vector Machine (SVM):

- `C`: Regularization parameter. Lower values mean stronger regularization.
- `kernel`: Type of kernel function (linear, rbf, poly). Determines decision boundary shape.
- `gamma`: Kernel coefficient for RBF. Controls influence of single training examples.

Logistic Regression:

- `C`: Inverse regularization strength. Smaller values mean stronger regularization.
- `penalty`: Type of regularization (l1, l2, elasticnet).
- `solver`: Algorithm for optimization (lbfgs, saga, etc.).

```
# Example: Viewing default hyperparameters
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression

# Create models with default hyperparameters
rf = RandomForestClassifier()
svm = SVC()
lr = LogisticRegression()

print("Random Forest Default Hyperparameters:")
print(f"  n_estimators: {rf.n_estimators}")
print(f"  max_depth: {rf.max_depth}")
print(f"  min_samples_split: {rf.min_samples_split}")
print(f"  max_features: {rf.max_features}")

print("\nSVM Default Hyperparameters:")
print(f"  C: {svm.C}")
print(f"  kernel: {svm.kernel}")
print(f"  gamma: {svm.gamma}")

print("\nLogistic Regression Default Hyperparameters:")
print(f"  C: {lr.C}")
print(f"  penalty: {lr.penalty}")
print(f"  solver: {lr.solver}")
```

```
Random Forest Default Hyperparameters:
  n_estimators: 100
  max_depth: None
  min_samples_split: 2
  max_features: sqrt

SVM Default Hyperparameters:
  C: 1.0
  kernel: rbf
  gamma: scale

Logistic Regression Default Hyperparameters:
  C: 1.0
  penalty: l2
  solver: lbfgs
```

🤔 Knowledge Check: Hyperparameter Understanding

Question 1 (Conceptual): If you wanted to reduce overfitting in a Random Forest model, which hyperparameters would you adjust and in what direction?

Your answer: [Write your answer here]

Question 2 (Reflective): 💭 Why do you think scikit-learn chose these specific default values? What assumptions might they be making about typical datasets?

Your answer: [Write your answer here]

Logistic Regression hyperparameters:

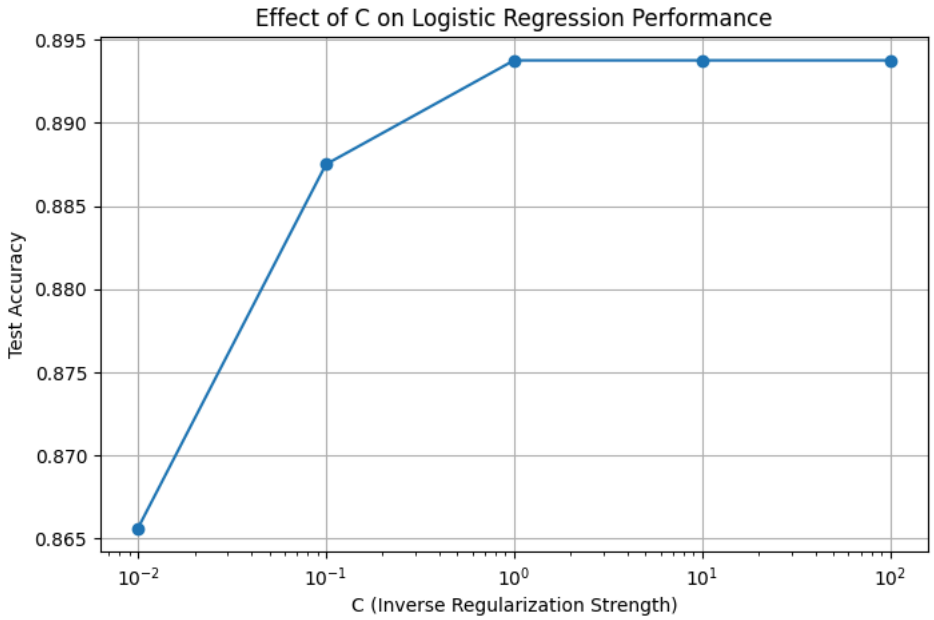
- `C`: inverse of regularization strength
- `max_iter`: max number of iterations

Random Forest hyperparameters:

- `n_estimators`: number of trees
- `max_depth`: max depth of trees
- `max_features`: number of features considered for best split

Let's see their effect on model performance.

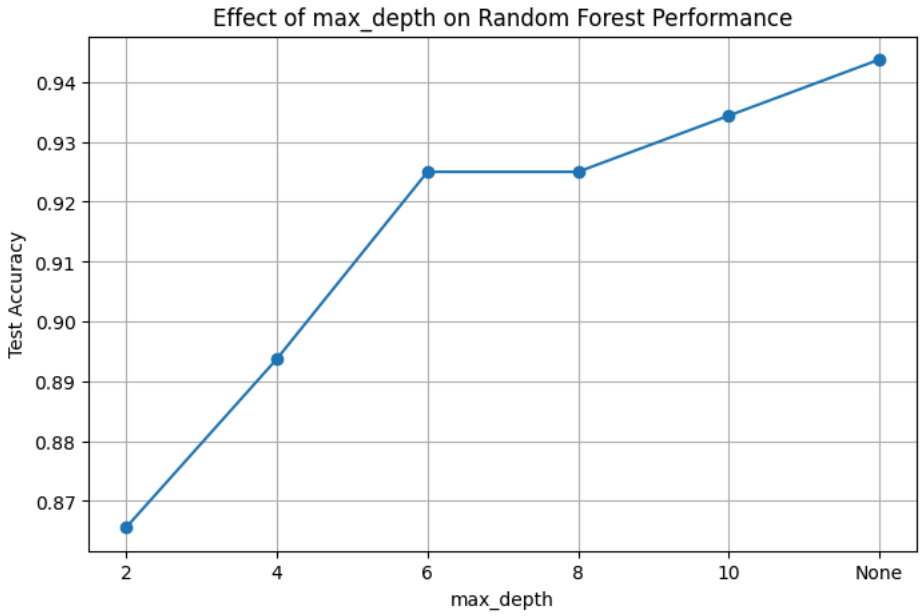
```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Logistic Regression with different C values
C_values = [0.01, 0.1, 1, 10, 100]
logreg_scores = []
for c in C_values:
    model = LogisticRegression(C=c, max_iter=200, random_state=42)
    model.fit(X_train_scaled, y_train)
    preds = model.predict(X_test_scaled)
    acc = accuracy_score(y_test, preds)
    logreg_scores.append(acc)

# Plot results
plt.figure(figsize=(8,5))
plt.plot(C_values, logreg_scores, marker='o')
plt.xscale('log')
plt.xlabel('C (Inverse Regularization Strength)')
plt.ylabel('Test Accuracy')
plt.title('Effect of C on Logistic Regression Performance')
plt.grid(True)
plt.show()
```



Interpretation:

- Smaller `C` means stronger regularization (simpler model).
- Larger `C` means less regularization (more complex model).
- Accuracy changes with these hyperparameters, demonstrating their importance.

```
# Random Forest with different max_depth values
depth_values = [2, 4, 6, 8, 10, None]
rf_scores = []
for depth in depth_values:
    model = RandomForestClassifier(n_estimators=100, max_depth=depth, random_state=42)
    model.fit(X_train, y_train)  # no scaling needed for RF
    preds = model.predict(X_test)
    acc = accuracy_score(y_test, preds)
    rf_scores.append(acc)

# Plot results
plt.figure(figsize=(8,5))
labels = ['2', '4', '6', '8', '10', 'None']
plt.plot(labels, rf_scores, marker='o')
plt.xlabel('max_depth')
plt.ylabel('Test Accuracy')
plt.title('Effect of max_depth on Random Forest Performance')
plt.grid(True)
plt.show()
```



Conceptual Question:

- How might changing the `max_depth` affect underfitting or overfitting? Why?

---

4. Parameters vs Hyperparameters (detailed explanation)

Parameters:

- Learned from the data during training.
- Example: Coefficients in logistic regression, split thresholds in decision trees.

Hyperparameters:

- Set before training.
- Control the training process and model complexity.
- Examples: learning rate, number of trees, max depth.

Understanding the difference helps us tune models effectively.

Reflective Question:

- Why can't we learn hyperparameters directly from the training data like parameters?
- How might hyperparameters impact the final model's ability to generalize?

---

5. Bias-Variance Tradeoff (with visualizations)

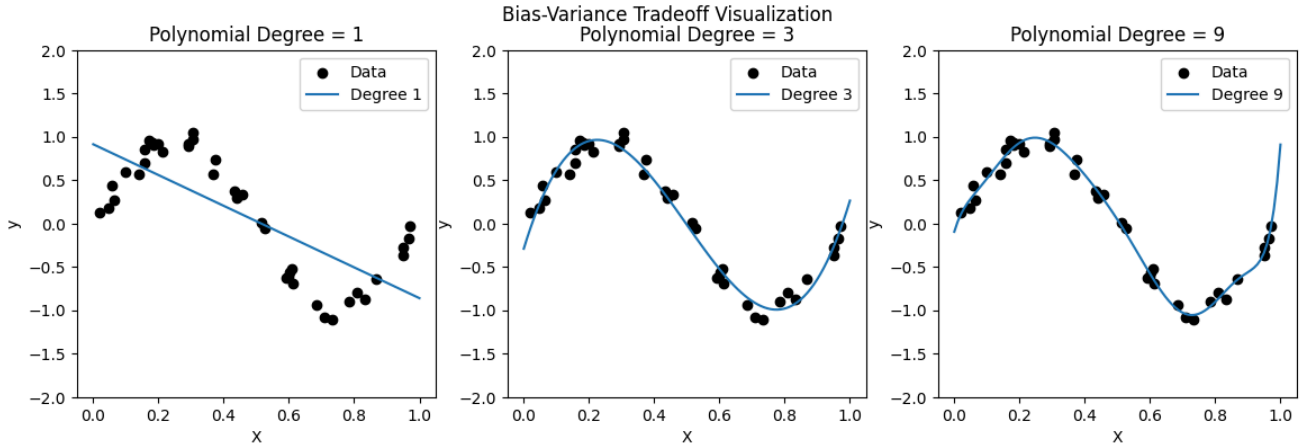
The bias-variance tradeoff is a fundamental concept:

- **Bias:** Error from erroneous assumptions in the model.
- **Variance:** Error from sensitivity to small fluctuations in the training set.

Hyperparameters often control this tradeoff.

```
# Visualizing bias-variance tradeoff with polynomial regression on synthetic data
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline

np.random.seed(42)
X_synthetic = np.sort(np.random.rand(40))
y_synthetic = np.sin(2 * np.pi * X_synthetic) + np.random.randn(40) * 0.1
```

```
X_plot = np.linspace(0, 1, 100)

degrees = [1, 3, 9]
plt.figure(figsize=(14, 4))
for i, degree in enumerate(degrees, 1):
    model = make_pipeline(PolynomialFeatures(degree), LinearRegression())
    model.fit(X_synthetic[:, np.newaxis], y_synthetic)
    y_plot = model.predict(X_plot[:, np.newaxis])

    plt.subplot(1, 3, i)
    plt.scatter(X_synthetic, y_synthetic, color='black', label='Data')
    plt.plot(X_plot, y_plot, label=f'Degree {degree}')
    plt.ylim(-2, 2)
    plt.legend()
    plt.title(f'Polynomial Degree = {degree}')
    plt.xlabel('X')
    plt.ylabel('y')

plt.suptitle('Bias-Variance Tradeoff Visualization')
plt.show()
```



Explanation:

- Degree 1: High bias, underfitting (too simple).
- Degree 9: High variance, overfitting (too complex).
- Degree 3: Good balance.

Hyperparameters like model complexity control this tradeoff.

---

6. Cross-Validation (demonstration)

Cross-validation helps us estimate model performance reliably by splitting the data multiple times.

```
from sklearn.model_selection import cross_val_score

model = RandomForestClassifier(n_estimators=100, max_depth=6, random_state=42)
scores = cross_val_score(model, X_train, y_train, cv=5, scoring='accuracy')

print(f"Cross-validation accuracies: {scores}")
print(f"Mean CV accuracy: {scores.mean():.4f}")
```

```
Cross-validation accuracies: [0.87890625 0.87109375 0.87890625 0.89453125 0.88627451]
Mean CV accuracy: 0.8819
```

Conceptual Question:

- Why is cross-validation considered a better performance estimator than a single train-test split?

---

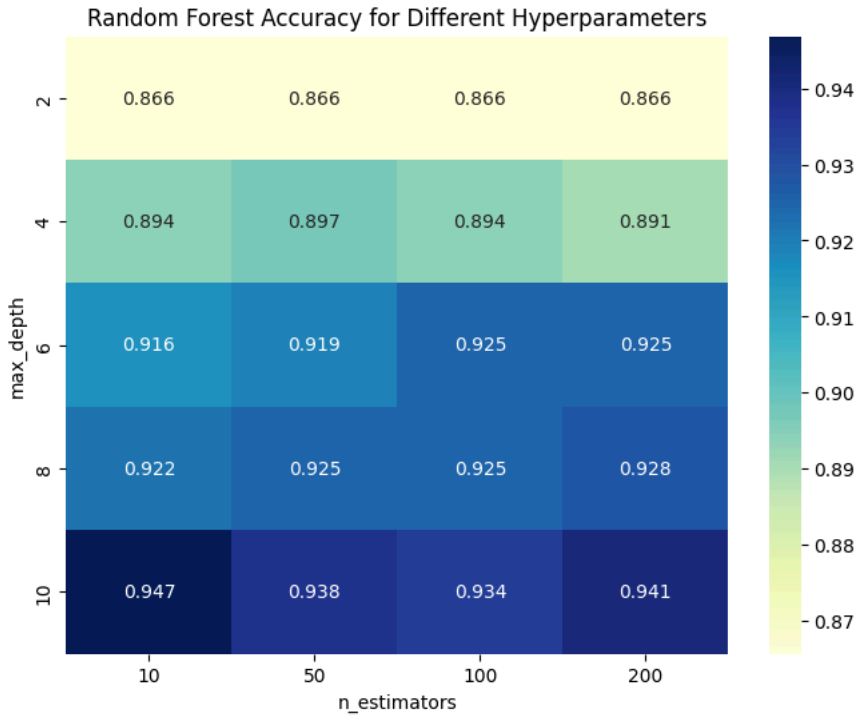
7. Manual Hyperparameter Tuning (complete example)

Let's manually tune hyperparameters for a Random Forest and see how it affects performance.

```
max_depth_values = [2, 4, 6, 8, 10]
n_estimators_values = [10, 50, 100, 200]

results = []
for depth in max_depth_values:
    for n_est in n_estimators_values:
        model = RandomForestClassifier(max_depth=depth, n_estimators=n_est, random_state=42)
        model.fit(X_train, y_train)
        preds = model.predict(X_test)
        acc = accuracy_score(y_test, preds)
        results.append({'max_depth': depth, 'n_estimators': n_est, 'accuracy': acc})

results_df = pd.DataFrame(results)
results_df
```

	max_depth	n_estimators	accuracy
0	2	10	0.865625
1	2	50	0.865625
2	2	100	0.865625
3	2	200	0.865625
4	4	10	0.893750
5	4	50	0.896875
6	4	100	0.893750
7	4	200	0.890625
8	6	10	0.915625
9	6	50	0.918750
10	6	100	0.925000
11	6	200	0.925000
12	8	10	0.921875
13	8	50	0.925000
14	8	100	0.925000
15	8	200	0.928125
16	10	10	0.946875
17	10	50	0.937500
18	10	100	0.934375
19	10	200	0.940625

```
# Visualize results as heatmap
pivot_table = results_df.pivot(index='max_depth', columns='n_estimators', values='accuracy')

plt.figure(figsize=(8,6))
sns.heatmap(pivot_table, annot=True, fmt='.3f', cmap='YlGnBu')
plt.title('Random Forest Accuracy for Different Hyperparameters')
plt.ylabel('max_depth')
plt.xlabel('n_estimators')
plt.show()
```



Reflective Question:

- Which hyperparameter seemed to have a bigger impact on accuracy? Why might that be?
- How would you decide which hyperparameters to tune next based on these results?

---

8. Grid Search (complete implementation)

Grid Search automates the manual hyperparameter tuning by exhaustively searching over a parameter grid.

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'max_depth': [4, 6, 8],
    'n_estimators': [50, 100, 150],
    'max_features': ['auto', 'sqrt']
}

rf = RandomForestClassifier(random_state=42)
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)
grid_search.fit(X_train, y_train)

print(f"Best parameters found: {grid_search.best_params_}")
print(f"Best cross-validation accuracy: {grid_search.best_score_:.4f}")
```

```
Best parameters found: {'max_depth': 8, 'max_features': 'sqrt', 'n_estimators': 50}
Best cross-validation accuracy: 0.8913
```

```
# Evaluate best model on test set
best_rf = grid_search.best_estimator_
test_preds = best_rf.predict(X_test)
print("Test Accuracy:", accuracy_score(y_test, test_preds))
print("Classification Report:")
print(classification_report(y_test, test_preds))
```

```
Test Accuracy: 0.925
Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.99      0.96       277
           1       0.88      0.51      0.65        43

    accuracy                           0.93       320
```

```
       macro avg       0.90      0.75      0.80       320
    weighted avg       0.92      0.93      0.92       320
```

Conceptual Question:

- What are some advantages and disadvantages of Grid Search?
- How does it compare to manual tuning?

Part 2 - Student Coding Exercises

9. Random Search (student implements with tips)

Unlike Grid Search, Random Search samples hyperparameter combinations randomly.

Exercise: Implement RandomizedSearchCV on the Random Forest model to find good hyperparameters.

Hints:

- Use `RandomizedSearchCV` from `sklearn.model_selection`
- Define parameter distributions for `max_depth`, `n_estimators`, and `max_features`
- Use 20 iterations (`n_iter=20`)
- Use 5-fold cross-validation
- Evaluate and print best params and accuracy

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

# YOUR CODE BELOW

# Define parameter distributions
param_dist = {
    'max_depth': randint(2, 15),
    'n_estimators': randint(10, 200),
    'max_features': ['auto', 'sqrt', 'log2', None]
}

# Initialize model
rf = RandomForestClassifier(random_state=42)

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=rf,
    param_distributions=param_dist,
    n_iter=20,
    cv=5,
    scoring='accuracy',
    random_state=42,
    n_jobs=-1
)

# Fit to training data
random_search.fit(X_train, y_train)

# Print best parameters and accuracy
print(f"Best parameters found: {random_search.best_params_}")
print(f"Best cross-validation accuracy: {random_search.best_score_:.4f}")

# Evaluate on test set
best_rf_random = random_search.best_estimator_
test_preds_random = best_rf_random.predict(X_test)
print(f"Test Accuracy: {accuracy_score(y_test, test_preds_random):.4f}")
```

```
Best parameters found: {'max_depth': 12, 'max_features': None, 'n_estimators': 198}
Best cross-validation accuracy: 0.8945
Test Accuracy: 0.9344
```

Reflective Question:

- How did the results from Random Search compare to Grid Search?

- Why might Random Search be more efficient than Grid Search in some cases?

---

10. Comparing Methods (student analysis)

Exercise:

- Compare the performances of manual tuning, grid search, and random search.
- Create a summary DataFrame with method, best hyperparameters, and test accuracy.
- Discuss advantages and disadvantages of each approach.

```
# YOUR CODE BELOW

summary = pd.DataFrame({
    'Method': ['Manual Tuning', 'Grid Search', 'Random Search'],
    'Best Hyperparameters': [
        results_df.loc[results_df['accuracy'].idxmax(), ['max_depth', 'n_estimators']].to_dict(),
        grid_search.best_params_,
        random_search.best_params_
    ],
    'Test Accuracy': [
        results_df['accuracy'].max(),
        accuracy_score(y_test, best_rf.predict(X_test)),
        accuracy_score(y_test, best_rf_random.predict(X_test))
    ]
})

summary
```

	Method	Best Hyperparameters	Test Accuracy
0	Manual Tuning	{'max_depth': 10.0, 'n_estimators': 10.0}	0.946875
1	Grid Search	{'max_depth': 8, 'max_features': 'sqrt', 'n_estimators': 50}	0.925000
2	Random Search	{'max_depth': 12, 'max_features': None, 'n_estimators': 198}	0.934375

Reflective Question:

- Which method would you choose for a real-world project and why?
- What factors (time, computational resources, dataset size) might influence your choice?

---

11. AutoML with AutoGluon (guided then student exercise)

AutoML tools like AutoGluon automate the entire model building and tuning process.

Let's first install and import AutoGluon, then run an AutoML experiment.

```
# Uncomment the next line to install AutoGluon if it's not installed
!pip install autogluon.tabular -q

from autogluon.tabular import TabularPredictor

# Prepare DataFrame for AutoGluon
train_data = X_train.copy()
train_data['good_quality'] = y_train.values

test_data = X_test.copy()
test_data['good_quality'] = y_test.values
```

Guided Exercise: Train an AutoGluon predictor and evaluate performance.

Use `time_limit=60` seconds for quick demonstration.

Observe the leaderboard and test accuracy.

```
# YOUR CODE BELOW

predictor = TabularPredictor(label='good_quality', eval_metric='accuracy').fit(
```

```
    train_data=train_data,
    time_limit=60,
    verbosity=2
)

# Show leaderboard
leaderboard = predictor.leaderboard(silent=True)
leaderboard
```

```
# Evaluate on test data
y_pred_automl = predictor.predict(test_data.drop(columns=['good_quality']))
```

```
No path specified. Models will be saved in: "AutogluonModels/ag-20251121_003917"
AutoGluon Test Accuracy: 0.9213
Verbosity: 2 (Standard Logging)
=================== System Info ===================
AutoGluon Version:  1.4.0
Python Version:     3.12.12
Operating System:   Linux
Platform Machine:   x86_64
Platform Version:   #1 SMP PREEMPT_DYNAMIC Thu Oct  2 10:42:05 UTC 2025
CPU Count:          2
Memory Avail:       10.55 GB / 12.67 GB (83.3%)
Disk Space Avail:   69.42 GB / 107.72 GB (64.4%)
===================================================
No presets specified! To achieve strong results with AutoGluon, it is recommended to use the available presets. Defaulting to `'
	Recommended Presets (For more details refer to https://auto.gluon.ai/stable/tutorials/tabular/tabular-essentials.html#preset
	presets='extreme' : New in v1.4: Massively better than 'best' on datasets <30000 samples by using new models meta-learned on
```

Student Exercise:

- Try increasing the `time_limit` to 180 seconds and retrain.
- Observe how the leaderboard and accuracy change.
- Experiment with excluding some model types by setting `excluded_model_types` in `fit()`.
- Report your findings.