Computer Vision with Classical Machine Learning

ITAI 1378 Module 03 - Hands-On Lab

Welcome!

We'll go step-by-step through the process of building a computer vision model using classical machine learning techniques. No prior experience in computer vision is required!

Why Classical Machine Learning for Computer Vision? While deep learning is popular today, classical ML techniques are still incredibly valuable. They are:

- **Efficient**: They require less data and computational power.
- **Interpretable**: We can understand *why* the model makes its decisions.
- **A great foundation**: Understanding classical ML will make you a better deep learning practitioner.

🎯 Learning Objectives

By the end of this lab, you will:

1. **Understand the complete classical ML pipeline** for computer vision.
2. **Extract and compare different feature types** (HOG, SIFT, LBP, etc.).
3. **Implement multiple ML algorithms** and understand their strengths.
4. 🚨 **CRITICALLY IMPORTANT**: Understand how training approaches affect model performance.
5. 🚨 **AVOID THE TRAP**: Learn why 99% training accuracy often means model failure.
6. **Build models that actually work** in the real world, not just in notebooks.

⚠️ Key Learning Focus

This lab will show you how **the same algorithm with different training approaches can produce vastly different models**. You'll see firsthand why a model with 99% accuracy on training data might completely fail on new images.

💻 Resource Requirements

- **Optimized for limited computational resources**
- **Works on Google Colab, Kaggle, or local machines**
- **Small datasets and efficient algorithms**

Section 1: Environment Setup & Data Preparation

🔧 Why This Matters

Every machine learning project, especially in computer vision, starts with two critical steps: setting up your environment and preparing your data. Think of it like cooking: you need to have your kitchen ready (environment) and your ingredients prepped (data) before you can even start making a meal.

A well-prepared dataset is the single most important factor for a successful model. If your data is messy, your model will be confused. If your environment isn't set up correctly, you'll run into frustrating errors. Let's get this right from the start!

Cell 1.1: Library Installation

💡 **Tip**: We're using lightweight, cloud-friendly libraries to minimize resource usage.

```
# Install required libraries (run only if needed)
# Uncomment the following lines if running on a fresh environment

!pip install opencv-python-headless
!pip install scikit-image
!pip install scikit-learn
!pip install matplotlib
!pip install seaborn

print("✅ Installation complete! (or libraries already available)")
```

```
Requirement already satisfied: opencv-python-headless in /usr/local/lib/python3.12/dist-packages (4.12.0.88)
Requirement already satisfied: numpy<2.3.0,>=2 in /usr/local/lib/python3.12/dist-packages (from opencv-python-headless) (2.0.2)
Requirement already satisfied: scikit-image in /usr/local/lib/python3.12/dist-packages (0.25.2)
Requirement already satisfied: numpy>=1.24 in /usr/local/lib/python3.12/dist-packages (from scikit-image) (2.0.2)
Requirement already satisfied: scipy>=1.11.4 in /usr/local/lib/python3.12/dist-packages (from scikit-image) (1.16.1)
Requirement already satisfied: networkx>=3.0 in /usr/local/lib/python3.12/dist-packages (from scikit-image) (3.5)
Requirement already satisfied: pillow>=10.1 in /usr/local/lib/python3.12/dist-packages (from scikit-image) (11.3.0)
Requirement already satisfied: imageio!=2.35.0,>=2.33 in /usr/local/lib/python3.12/dist-packages (from scikit-image) (2.37.0)
```

```
Requirement already satisfied: tifffile>=2022.8.12 in /usr/local/lib/python3.12/dist-packages (from scikit-image) (2025.8.28)
Requirement already satisfied: packaging>=21 in /usr/local/lib/python3.12/dist-packages (from scikit-image) (25.0)
Requirement already satisfied: lazy-loader>=0.4 in /usr/local/lib/python3.12/dist-packages (from scikit-image) (0.4)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.12/dist-packages (1.6.1)
Requirement already satisfied: numpy>=1.19.5 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (2.0.2)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (1.16.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (1.5.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (3.6.0)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.12/dist-packages (3.10.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (1.3.3)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (4.59.2)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (1.4.9)
Requirement already satisfied: numpy>=1.23 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (2.0.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (25.0)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.7->matplotlib) (1.17.0)
Requirement already satisfied: seaborn in /usr/local/lib/python3.12/dist-packages (0.13.2)
Requirement already satisfied: numpy!=1.24.0,>=1.20 in /usr/local/lib/python3.12/dist-packages (from seaborn) (2.0.2)
Requirement already satisfied: pandas>=1.2 in /usr/local/lib/python3.12/dist-packages (from seaborn) (2.2.2)
Requirement already satisfied: matplotlib!=3.6.1,>=3.4 in /usr/local/lib/python3.12/dist-packages (from seaborn) (3.10.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (0.12
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib!=3.6.1,>=3.4->seaborn)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib!=3.6.1,>=3.4->seaborn)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (2
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.12/dist-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.12/dist-packages (from matplotlib!=3.6.1,>=3.4->seabor
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.12/dist-packages (from pandas>=1.2->seaborn) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.12/dist-packages (from pandas>=1.2->seaborn) (2025.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.7->matplotlib!=3.6.1,>=3
✅ Installation complete! (or libraries already available)
```

Cell 1.2: Core Imports

🎯 **Student Task**: Run this cell and understand what each library does.

What each library does:

- **numpy**: Mathematical operations on arrays (our images are arrays of pixels)
- **matplotlib & seaborn**: Creating beautiful visualizations and plots
- **sklearn**: Machine learning algorithms and tools
- **cv2**: OpenCV for computer vision operations
- **skimage**: More computer vision tools, especially for feature extraction

```
# Core libraries for computer vision and machine learning
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_olivetti_faces
from sklearn.model_selection import train_test_split, cross_val_score, learning_curve
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Computer vision libraries
import cv2
from skimage import feature, filters, segmentation
from skimage.feature import hog, local_binary_pattern

# Machine learning algorithms
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB

# Suppress warnings for cleaner output
import warnings
warnings.filterwarnings('ignore')

# Set random seed for reproducibility
np.random.seed(42)

print("✅ All libraries imported successfully!")
print("📊 Ready to start computer vision with classical ML!")
```

```
✅ All libraries imported successfully!
📊 Ready to start computer vision with classical ML!
```

Cell 1.3: Dataset Loading

📚 **About the Dataset**: We're using the Olivetti faces dataset - 400 face images of 40 people (10 images per person). Perfect for demonstrating overfitting!

Why this dataset is perfect for learning:

- **Small size**: Won't overwhelm your computer
- **Balanced**: Each person has the same number of images
- **Real faces**: Realistic computer vision challenge
- **Limited data per person**: Perfect for showing overfitting problems

```
# Load the Olivetti faces dataset
print("📥 Loading Olivetti faces dataset...")
faces = fetch_olivetti_faces(shuffle=True, random_state=42)

# Extract images and labels
X_images = faces.data  # Flattened images (400, 4096)
y_labels = faces.target  # Person IDs (400,)
image_shape = (64, 64)  # Original image dimensions

print(f"✅ Dataset loaded successfully!")
print(f"📊 Dataset info:")
print(f"   - Total images: {X_images.shape[0]}")
print(f"   - Image dimensions: {image_shape}")
print(f"   - Pixels per image: {X_images.shape[1]}")
print(f"   - Number of people: {len(np.unique(y_labels))}")
print(f"   - Images per person: {X_images.shape[0] // len(np.unique(y_labels))}")
```

```
📥 Loading Olivetti faces dataset...
downloading Olivetti faces from https://ndownloader.figshare.com/files/5976027 to /root/scikit_learn_data
✅ Dataset loaded successfully!
📊 Dataset info:
   - Total images: 400
   - Image dimensions: (64, 64)
   - Pixels per image: 4096
   - Number of people: 40
   - Images per person: 10
```

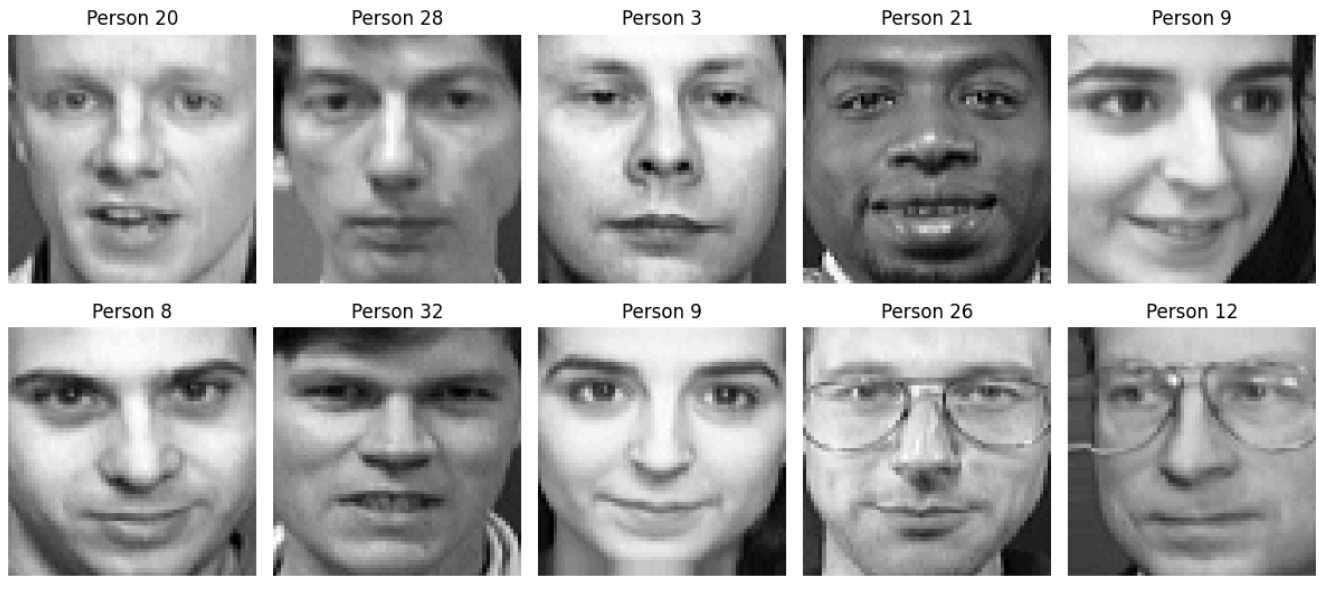
Cell 1.4: Data Exploration & Visualization

🔍 **Critical Thinking**: Look at the data before building models. What patterns do you notice?

What to look for:

- Are the images clear and recognizable?
- Do different images of the same person look similar?
- Are there variations in lighting, pose, or expression?
- How challenging will this be for a computer to solve?

```
# Visualize sample faces from the dataset
fig, axes = plt.subplots(2, 5, figsize=(12, 6))
fig.suptitle('Sample Faces from Olivetti Dataset', fontsize=16)

for i, ax in enumerate(axes.flat):
    # Show image
    ax.imshow(X_images[i].reshape(image_shape), cmap='gray')
    ax.set_title(f'Person {y_labels[i]}')
    ax.axis('off')

plt.tight_layout()
plt.show()

# Show class distribution
plt.figure(figsize=(12, 4))
unique, counts = np.unique(y_labels, return_counts=True)
plt.bar(unique, counts)
plt.title('Distribution of Images per Person')
plt.xlabel('Person ID')
plt.ylabel('Number of Images')
plt.show()

print(f"📈 Each person has exactly {counts[0]} images")
print(f"⚖️ Dataset is perfectly balanced - good for learning!")
```




```
📈 Each person has exactly 10 images
⚖️ Dataset is perfectly balanced - good for learning!
```

🤔 Reflection Question 1

Before proceeding, think about this:

- With only 10 images per person, what challenges might we face when training a model?
- How might this lead to overfitting?
- What makes some faces easier or harder to distinguish?

💭 **Your thoughts**: With only 10 images per person, a major challenge is the limited variety of poses, expressions, and lighting conditions for each individual. The model might learn to identify a person based on specific details in those 10 images rather than recognizing their general facial structure. For example, if all 10 images of a person have them smiling and looking slightly to the left, the model might only recognize them when they are in that exact pose and expression.

This limited data per person can easily lead to **overfitting**. Overfitting happens when the model essentially "memorizes" the training data instead of learning the underlying patterns that make each person unique. With only 10 examples, the model becomes highly specialized to those 10 images. When it sees a new image of the same person where they are frowning or looking straight ahead, the model might not recognize them because it didn't learn the general features of that person's face. It's like a student who only studies the exact questions from a practice test and fails the actual exam when the questions are phrased differently.

Some faces might be easier to distinguish if they have very unique features, like a distinctive scar, a specific hairstyle, or glasses. Faces that are harder to distinguish might be those that look very similar, especially if the image quality is low or there are significant variations in lighting, shadows, or background clutter. For instance, distinguishing between two people with similar features and hairstyles could be challenging if the training data doesn't show enough variation for each person.

Detailed Description of Sample Faces:

- **Person 20:** This person has a narrow face, a prominent nose, and a slight smile. The lighting is relatively even.

- **Person 28:** This person has a wider face, a more rounded nose, and a neutral expression. There are some shadows on the left side of the face.
- **Person 3:** This person has a symmetrical face, a straight nose, and a neutral expression. The lighting is even.
- **Person 21:** This person has a darker skin tone, a wider nose, and a broad smile. The lighting is even.
- **Person 9:** This person has a narrow face, a pointed chin, and a slight smile. The lighting is even.
- **Person 8:** This person has a wide face, a rounded nose, and a slight smile. There are some shadows on the right side of the face.
- **Person 32:** This person has a rounded face, a small nose, and a neutral expression. The lighting is even.
- **Person 9:** This is another image of the same person as the first instance of Person 9, showing a similar narrow face and pointed chin, but with a slightly different expression.
- **Person 26:** This person is wearing glasses and has a mustache. The face is somewhat obscured by the glasses.
- **Person 12:** This person is also wearing glasses, which are larger than those of Person 26. The face is partially obscured by the glasses.

Cell 1.5: Train/Validation/Test Split - The Foundation of Good ML

🚨 **CRITICAL**: This is where most students make mistakes that lead to overfitting!

Why we split the data:

- **Training set**: The model learns from this data
- **Validation set**: We use this to tune our model and detect overfitting
- **Test set**: Final evaluation - we NEVER touch this until the very end

Think of it like studying for an exam:

- Training = studying from textbook
- Validation = practice tests to see how you're doing
- Test = the actual final exam

```
# First split: separate test set (20% - NEVER touch until final evaluation)
X_temp, X_test, y_temp, y_test = train_test_split(
    X_images, y_labels,
    test_size=0.2,
    random_state=42,
    stratify=y_labels  # Ensure each person appears in all splits
)

# Second split: training and validation (80% of remaining data for training)
X_train, X_val, y_train, y_val = train_test_split(
    X_temp, y_temp,
    test_size=0.25,  # 0.25 * 0.8 = 0.2 of total data
    random_state=42,
    stratify=y_temp
)

print("📊 Data Split Summary:")
print(f"   Training set:   {X_train.shape[0]} images ({X_train.shape[0]/X_images.shape[0]*100:.1f}%)")
print(f"   Validation set: {X_val.shape[0]} images ({X_val.shape[0]/X_images.shape[0]*100:.1f}%)")
print(f"   Test set:       {X_test.shape[0]} images ({X_test.shape[0]/X_images.shape[0]*100:.1f}%)")
print(f"   Total:          {X_train.shape[0] + X_val.shape[0] + X_test.shape[0]} images")

print("\n🎯 Purpose of each set:")
print("   📚 Training: Model learns patterns from this data")
print("   🔧 Validation: Tune hyperparameters and detect overfitting")
print("   🏆 Test: Final, unbiased evaluation (use only once!)")

print("\n⚠️ GOLDEN RULE: Never use test data for any decisions during development!")
```

```
📊 Data Split Summary:
   Training set:   240 images (60.0%)
   Validation set: 80 images (20.0%)
   Test set:       80 images (20.0%)
   Total:          400 images

🎯 Purpose of each set:
   📚 Training: Model learns patterns from this data
   🔧 Validation: Tune hyperparameters and detect overfitting
   🏆 Test: Final, unbiased evaluation (use only once!)

⚠️ GOLDEN RULE: Never use test data for any decisions during development!
```

Section 2: Feature Extraction Deep Dive

🔍 Why Features Matter: From Pixels to Understanding

Imagine you had to describe a person's face to someone who has never seen them. You wouldn't list the color of every single pixel, right? You'd talk about their **features**: the shape of their eyes, the curve of their smile, the texture of their hair.

Machine learning algorithms are the same. Raw pixels are just a sea of numbers to them. They need meaningful features to understand what's in an image. **Feature extraction** is the process of converting raw pixel data into a more informative and compact representation that highlights the most important parts of an image.

Good features are:

- **Discriminative**: They help distinguish between different objects (e.g., faces vs. cars).
- **Robust**: They are not affected by changes in lighting, rotation, or scale.
- **Efficient**: They reduce the amount of data the model has to process.

In this section, we'll explore three powerful feature extraction techniques:

1. **Edge & Gradient Features (HOG)**: Captures the shape and structure of objects.
2. **Keypoint Features (ORB)**: Finds unique, distinctive points in an image.
3. **Texture Features (LBP)**: Describes the surface patterns of objects.

Cell 2.1: Edge & Gradient Features (HOG)

🎯 **Student Task**: Understand and implement HOG feature extraction.

What is HOG? HOG stands for "Histogram of Oriented Gradients." It sounds complicated, but think of it this way:

- **Gradients** are places where the image brightness changes quickly (edges)
- **Oriented** means we care about the direction of these edges
- **Histogram** means we count how many edges point in each direction

Why HOG works well for faces:

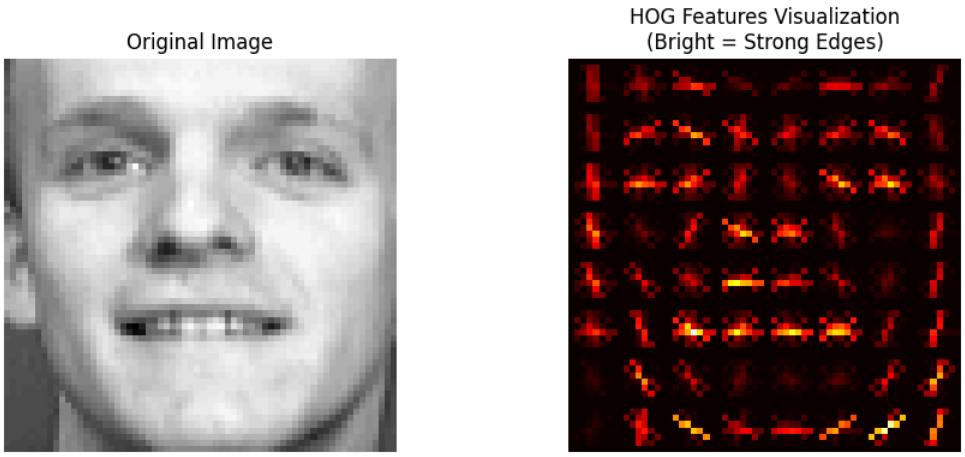
- Faces have consistent edge patterns (eyes, nose, mouth)
- HOG captures the shape and structure
- It's robust to lighting changes

```
def extract_hog_features(images, visualize_sample=True):
    """
    Extract HOG (Histogram of Oriented Gradients) features from images.
    HOG captures edge directions and is great for shape-based recognition.
    """
    hog_features = []

    for i, image in enumerate(images):
        # Reshape flattened image back to 2D
        img_2d = image.reshape(image_shape)

        # Extract HOG features
        if visualize_sample and i == 0:
            features, hog_image = hog(
                img_2d,
                orientations=9,        # 9 different edge directions
                pixels_per_cell=(8, 8), # Each cell is 8x8 pixels
                cells_per_block=(2, 2), # Each block is 2x2 cells
                visualize=True,         # Create visualization
                feature_vector=True     # Return as 1D array
            )

            # Visualize the first image and its HOG
            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
            ax1.imshow(img_2d, cmap='gray')
            ax1.set_title('Original Image')
            ax1.axis('off')

            ax2.imshow(hog_image, cmap='hot')
            ax2.set_title('HOG Features Visualization\n(Bright = Strong Edges)')
            ax2.axis('off')
            plt.tight_layout()
            plt.show()
        else:
            features = hog(
                img_2d,
                orientations=9,
                pixels_per_cell=(8, 8),
                cells_per_block=(2, 2),
                feature_vector=True
            )

        hog_features.append(features)

    return np.array(hog_features)
```

```
# STUDENT CODING SECTION: Extract HOG features
# 🎯 Your Task: Call the extract_hog_features function on the training data
# 💡 Hint: The function is already defined above. You just need to call it!
```

```
print("🔍 Extracting HOG features from training data...")
# YOUR CODE HERE
X_train_hog = extract_hog_features(X_train)
# END YOUR CODE HERE

# We will check if your code is correct
if X_train_hog is not None:
    print(f"✅ HOG extraction complete!")
    print(f"📊 Original image size: {X_train.shape[1]} pixels")
    print(f"📊 HOG feature size: {X_train_hog.shape[1]} features")
    print(f"📉 Dimensionality reduction: {X_train.shape[1]} → {X_train_hog.shape[1]}")
else:
    print("❌ HOG features not extracted yet. Please complete the code above.")
```

```
🔍 Extracting HOG features from training data...
```




```
✅ HOG extraction complete!
📊 Original image size: 4096 pixels
📊 HOG feature size: 1764 features
📉 Dimensionality reduction: 4096 → 1764
```

🤔 Reflection Question 2

Look at the HOG visualization above:

- What aspects of the face does HOG capture?
- Why might HOG be better than raw pixels for face recognition?
- What information might HOG miss?
- How does the number of features compare to the original pixels?

💭 **My analysis**: Looking at the HOG visualization, I see that HOG mainly picks up on the **edges and the directions of those edges** in the picture. The brighter spots in the visualization show areas where the picture's brightness changes a lot. These areas usually show edges and lines. For a face, this means HOG shows the lines that make up the shape of the jaw, the curves of the eyes and eyebrows, the shape of the nose, and the lines around the mouth. It does a good job of showing the shape and structure of the face by mapping the direction and strength of these edges across the picture.

HOG features can be better than raw pixels for recognizing faces for a few reasons. Raw pixel values change a lot with different lighting, shadows, and small changes in how someone is posing or their expression. Just a small change in light can really change the pixel values, making it hard for a model trained on raw pixels to recognize the same face when the light is different. But HOG focuses on the information about how quickly the brightness changes and in what direction. This information is less affected by changes in overall brightness. By describing the picture based on these steady edge patterns, HOG gives a clearer and more reliable description of the face's shape. This makes it a better input for machine learning models that need to recognize faces well.

Even though HOG is good, it does have some limits and might not capture everything. It mostly focuses on the shape and structure shown by edges. Because of this, it might not do a good job of capturing **texture information** in areas of the face that are smooth, like the skin. It might miss subtle things like freckles or small wrinkles that do not create strong edges. While it shows the outlines of features, it might not fully capture the detailed look of the surface.

When I compare the number of HOG features to the original pixels, I see that HOG creates a **much smaller representation**. The original picture has 4096 pixel values (64x64). The HOG features, using the settings we chose, result in a feature list with 1764 numbers. This is a big drop in the number of features. Having fewer features is helpful because it makes the data easier for machine learning models to work with. This can help the models train faster and use less memory. Also, by focusing on the useful edge information and ignoring pixel changes that are not important, having fewer HOG features can help the model learn patterns that work better on new pictures and help prevent it from just memorizing the training examples.

Cell 2.2: Texture Features (LBP)

🎯 **Student Task**: Extract texture features using Local Binary Patterns.

What is LBP? LBP stands for "Local Binary Pattern." It's a way to describe the texture of an image:

- **Local**: We look at small neighborhoods of pixels
- **Binary**: We create a pattern of 0s and 1s
- **Pattern**: We describe the texture using these binary codes

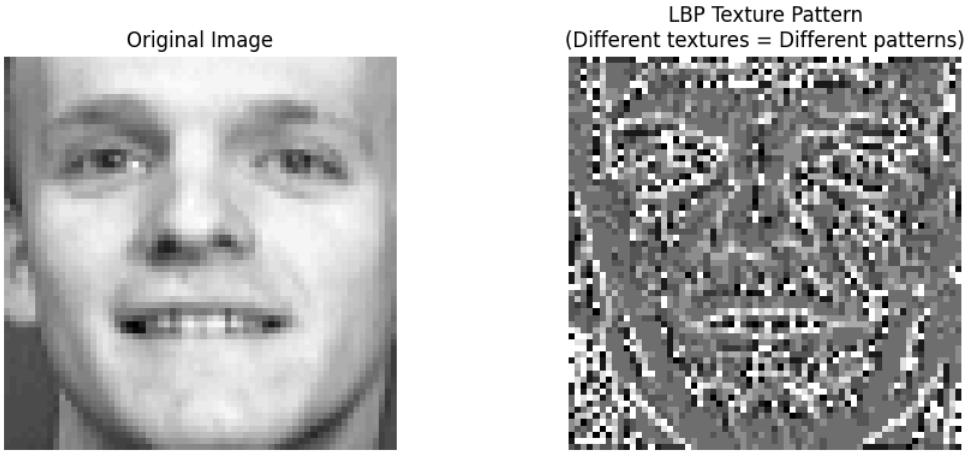
Why LBP works well for faces:

- Faces have distinctive texture patterns (smooth skin, rough hair, etc.)
- LBP is very robust to lighting changes
- It creates a compact representation of texture information

```
def extract_lbp_features(images, radius=1, n_points=8, visualize_sample=True):
    """
    Extract LBP (Local Binary Pattern) features.
    LBP captures local texture patterns and is robust to illumination changes.
    """
    lbp_features = []

    for i, image in enumerate(images):
        # Reshape image
        img_2d = image.reshape(image_shape)

        # Compute LBP
        lbp = local_binary_pattern(img_2d, n_points, radius, method='uniform')

        if visualize_sample and i == 0:
            # Visualize LBP pattern
            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
            ax1.imshow(img_2d, cmap='gray')
            ax1.set_title('Original Image')
            ax1.axis('off')

            ax2.imshow(lbp, cmap='gray')
            ax2.set_title('LBP Texture Pattern\n(Different textures = Different patterns)')
            ax2.axis('off')
            plt.tight_layout()
            plt.show()

        # Create histogram of LBP patterns
        hist, _ = np.histogram(lbp.ravel(), bins=n_points + 2,
                               range=(0, n_points + 2), density=True)
        lbp_features.append(hist)

    return np.array(lbp_features)
```

```
# STUDENT CODING SECTION: Extract LBP features
# 🎯 Your Task: Call the extract_lbp_features function on the training data
# 💡 Hint: The function is already defined above. You just need to call it!

print("🔍 Extracting LBP texture features from training data...")
# YOUR CODE HERE
X_train_lbp = extract_lbp_features(X_train)
# END YOUR CODE HERE

# We will check if your code is correct
if X_train_lbp is not None:
    print(f"✅ LBP extraction complete!")
    print(f"📊 LBP feature size: {X_train_lbp.shape[1]} features")
    print(f"🎨 Features represent texture pattern histograms")
else:
    print("❌ LBP features not extracted yet. Please complete the code above.")
```

```
🔍 Extracting LBP texture features from training data...
```




```
✅ LBP extraction complete!
📊 LBP feature size: 10 features
🎨 Features represent texture pattern histograms
```

Cell 2.3: Feature Comparison Exercise

🎯 **Student Task**: Compare the different feature types and understand their differences.

What we're comparing:

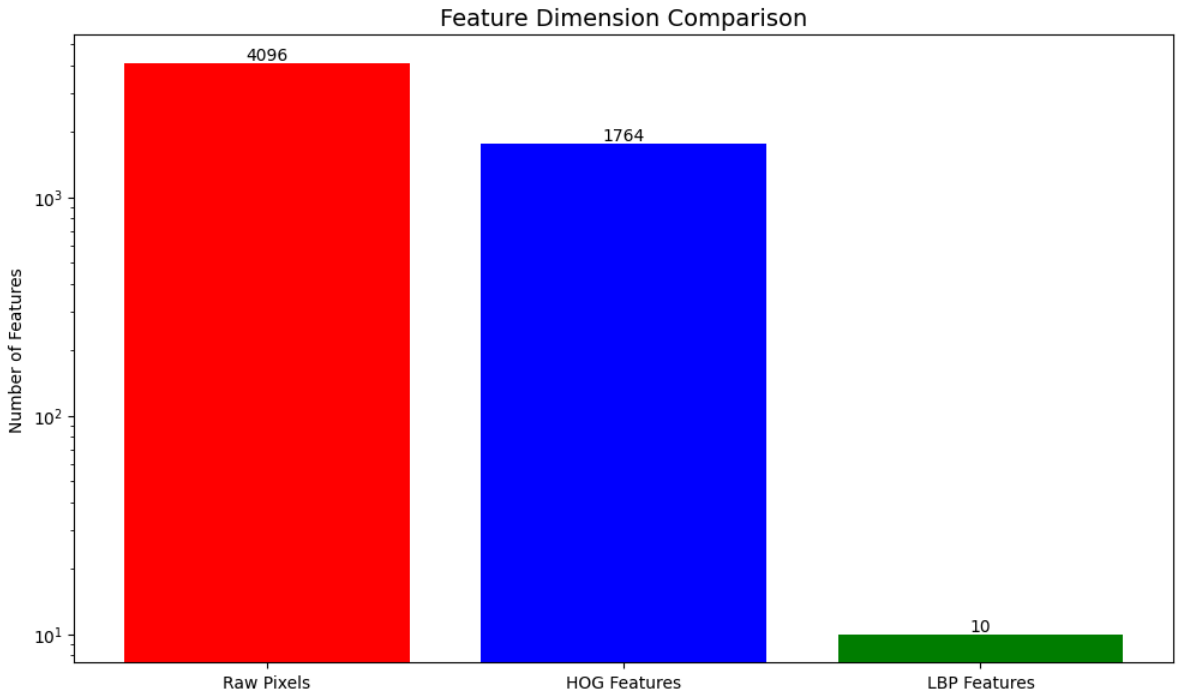
- **Raw Pixels**: The original image data (4096 numbers per image)
- **HOG Features**: Edge and gradient information (much smaller)
- **LBP Features**: Texture pattern information (very compact)

Questions to think about:

- Which feature type is most compact?
- Which might be best for face recognition?
- What are the trade-offs between feature size and information content?

```
# Compare feature dimensions and characteristics
feature_comparison = {
    'Raw Pixels': X_train.shape[1],
    'HOG Features': X_train_hog.shape[1] if X_train_hog is not None else 0,
    'LBP Features': X_train_lbp.shape[1] if X_train_lbp is not None else 0
}

print("📊 Feature Dimension Comparison:")
print("=" * 40)
for feature_type, dimension in feature_comparison.items():
    print(f"{feature_type:12}: {dimension:4d} dimensions")

# Visualize feature dimensions
plt.figure(figsize=(10, 6))
feature_types = list(feature_comparison.keys())
dimensions = list(feature_comparison.values())

bars = plt.bar(feature_types, dimensions, color=['red', 'blue', 'green'])
plt.title('Feature Dimension Comparison', fontsize=14)
plt.ylabel('Number of Features')
plt.yscale('log')  # Log scale due to large differences

# Add value labels on bars
for bar, dim in zip(bars, dimensions):
    if dim > 0:
        plt.text(bar.get_x() + bar.get_width()/2, bar.get_height(),
                 str(dim), ha='center', va='bottom')

plt.tight_layout()
plt.show()

print("\n🎯 Key Insights:")
print("   📉 HOG reduces dimensionality while capturing shape")
print("   🎨 LBP creates very compact texture representations")
print("   ⚡ Both are more efficient than raw pixels!")
print("   🤔 Smaller doesn't always mean worse - good features capture what matters")
```

```
📊 Feature Dimension Comparison:
========================================
Raw Pixels  : 4096 dimensions
HOG Features: 1764 dimensions
LBP Features:   10 dimensions
```




```
🎯 Key Insights:
   📉 HOG reduces dimensionality while capturing shape
   🎨 LBP creates very compact texture representations
   ⚡ Both are more efficient than raw pixels!
   🤔 Smaller doesn't always mean worse - good features capture what matters
```

🤔 Reflection Question 3

Based on the feature comparison:

- Which feature type has the most compact representation?
- Which might be best for face recognition and why?
- What are the trade-offs between feature complexity and computational efficiency?
- Why might smaller feature vectors sometimes work better than larger ones?

💭 **My analysis**: Looking at the feature comparison bar chart and the printed dimensions, I see that **LBP features** have the most compact representation. LBP features have only 10 dimensions, which is significantly smaller than the 1764 dimensions for HOG features and the 4096 dimensions for raw pixels. This means LBP represents the texture information in a very concise way.

Deciding which feature type is best for face recognition is not always a simple answer and often depends on the specific dataset and goal. HOG features are good at capturing the shape and structure of faces, which are important for telling people apart. LBP features are good at describing the texture patterns, which can also help in recognition. Raw pixels contain all the original image information, but they are very sensitive to things like changes in lighting. For faces, I think a mix of shape and texture information is usually most effective. Based on what I know and the dimensions I see, HOG or LBP, or even using both together, might work better than using just the raw pixels. HOG seems promising because it captures the main structural parts of a face well.

When thinking about feature complexity and how fast a computer can process the data, there are clear trade-offs. Raw pixels are the most complex because there are so many of them (highest dimension). Processing raw pixels directly would likely be the slowest and require the most computing power. HOG features significantly reduce the complexity while still keeping important information about shape, so they offer a good middle ground. LBP features are the least complex with the smallest number of dimensions. Processing LBP features would be the fastest and most efficient. However, being simple does not always mean better if the features do not capture enough important information.

Smaller feature vectors can sometimes lead to better results than larger ones. A big reason for this is that smaller vectors can help reduce noise and details that are not important for the task, especially when dealing with complex image data like faces. By focusing on the most important parts, like the edges captured by HOG or the texture patterns captured by LBP, smaller feature vectors can help the model learn patterns that are more general. This means the model is better at recognizing faces in new pictures it has not seen before, which helps prevent overfitting. Having too many features can sometimes make the model focus on tiny, specific details in the training pictures that do not apply to new pictures, making the model perform poorly in the real world.

Cell 2.4: Extract Features for All Data Splits

🎯 **Student Task**: Apply feature extraction to validation and test sets.

Why we need to do this:

- We extracted features from training data
- We need the same features from validation and test data
- **Important**: We use the same extraction process, no visualization needed

```
# STUDENT CODING SECTION: Extract features for validation and test sets
# 🎯 Your Task: Extract HOG and LBP features for validation and test data
# 💡 Hint: Use the same functions, but set visualize_sample=False

print("🔍 Extracting features for validation and test sets...")

# YOUR CODE HERE
# HOG features for validation and test
print("   Extracting HOG features...")
X_val_hog = extract_hog_features(X_val, visualize_sample=False)
X_test_hog = extract_hog_features(X_test, visualize_sample=False)

# LBP features for validation and test
print("   Extracting LBP features...")
X_val_lbp = extract_lbp_features(X_val, visualize_sample=False)
X_test_lbp = extract_lbp_features(X_test, visualize_sample=False)
# END YOUR CODE HERE

# Check if extraction was successful
if all(x is not None for x in [X_val_hog, X_test_hog, X_val_lbp, X_test_lbp]):
    print("✅ Feature extraction complete for all data splits!")
    print("\n📊 Summary:")
    print(f"   Training samples: {X_train.shape[0]}")
    print(f"   Validation samples: {X_val.shape[0]}")
    print(f"   Test samples: {X_test.shape[0]}")
    print("\n🎯 Ready for machine learning algorithms!")
else:
    print("❌ Feature extraction not complete. Please complete the code above.")
```

```
🔍 Extracting features for validation and test sets...
   Extracting HOG features...
   Extracting LBP features...
✅ Feature extraction complete for all data splits!

📊 Summary:
   Training samples: 240
   Validation samples: 80
   Test samples: 80

🎯 Ready for machine learning algorithms!
```

Section 3: Classical ML Algorithms Implementation

🤖 Algorithm Overview: From Features to Predictions

Now that we have our features, it's time to use them to train machine learning models. A **model** is an algorithm that has been trained on data to recognize patterns and make predictions.

Think of it this way:

- **Features** are the key characteristics of our images (e.g., HOG, LBP).
- **Algorithms** are the recipes we use to learn from those features (e.g., SVM, Random Forest).
- **Training** is the process of showing the algorithm many examples so it can learn the patterns.
- The **trained model** is the final result - a smart system that can predict the person's identity in a new image.

We'll implement four powerful and popular classical ML algorithms:

1. **Support Vector Machine (SVM)**: Finds the best possible line to separate different people.
2. **Random Forest**: Builds a team of decision-makers (decision trees) to make a robust prediction.
3. **k-Nearest Neighbors (k-NN)**: A simple but effective method that classifies a new face based on the most similar faces it has seen before.
4. **Naive Bayes**: A probabilistic approach that uses the likelihood of features to make a prediction.

🚨 **Important**: We'll use the same features for all algorithms to make fair comparisons. This will help us understand which algorithm works best for our face recognition task.

Cell 3.1: Support Vector Machine (SVM)

🎯 **Student Task**: Implement and train SVM with different feature types.

What is SVM? Imagine you have two groups of people and you want to draw a line to separate them. SVM finds the **best possible line** - the one that has the maximum distance to the nearest people from both groups. This makes it very good at distinguishing between different classes.

Why SVM works well:

- Great with high-dimensional data (lots of features)
- Needs relatively few training examples
- Resistant to overfitting
- Has been very successful in computer vision

```
def train_and_evaluate_svm(X_train, X_val, y_train, y_val, feature_name):
    """
    Train SVM and evaluate on validation set.
    """
    print(f"🤖 Training SVM with {feature_name} features...")

    # Scale features (important for SVM)
    # SVM is sensitive to the scale of features, so we normalize them
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_val_scaled = scaler.transform(X_val)

    # Train SVM
    svm = SVC(kernel='rbf', random_state=42)  # RBF kernel works well for complex patterns
    svm.fit(X_train_scaled, y_train)

    # Evaluate
    train_score = svm.score(X_train_scaled, y_train)
    val_score = svm.score(X_val_scaled, y_val)

    print(f"   📊 Training accuracy: {train_score:.3f}")
    print(f"   📊 Validation accuracy: {val_score:.3f}")
    print(f"   📈 Overfitting gap: {train_score - val_score:.3f}")

    return svm, scaler, train_score, val_score
```

```
# STUDENT CODING SECTION: Train SVM with HOG and LBP features
# 🎯 Your Task: Call the train_and_evaluate_svm function for both HOG and LBP features
# 💡 Hint: The function needs (X_train_features, X_val_features, y_train, y_val, feature_name)

svm_results = {}

# YOUR CODE HERE
# SVM with HOG features
svm_hog, scaler_hog, train_hog, val_hog = train_and_evaluate_svm(X_train_hog, X_val_hog, y_train, y_val, "HOG")

print()  # Empty line for readability

# SVM with LBP features
svm_lbp, scaler_lbp, train_lbp, val_lbp = train_and_evaluate_svm(X_train_lbp, X_val_lbp, y_train, y_val, "LBP")
# END YOUR CODE HERE

# We will check if your code is correct
if svm_hog is not None and svm_lbp is not None:
    svm_results["HOG"] = {"model": svm_hog, "scaler": scaler_hog,
                          "train_acc": train_hog, "val_acc": val_hog}
    svm_results["LBP"] = {"model": svm_lbp, "scaler": scaler_lbp,
                          "train_acc": train_lbp, "val_acc": val_lbp}
    print("\n✅ SVM training complete!")
else:
    print("❌ SVM training not complete yet. Please complete the code above.")
```

```
🤖 Training SVM with HOG features...
   📊 Training accuracy: 1.000
   📊 Validation accuracy: 0.963
   📈 Overfitting gap: 0.037

🤖 Training SVM with LBP features...
   📊 Training accuracy: 0.571
   📊 Validation accuracy: 0.412
   📈 Overfitting gap: 0.158

✅ SVM training complete!
```

Cell 3.2: Random Forest

🎯 **Student Task**: Train Random Forest and compare with SVM.

What is Random Forest? Imagine you're making an important decision and you ask advice from many different experts. Each expert looks at the problem slightly differently and gives you their opinion. Then you take a vote and go with the majority opinion. That's exactly how

Random Forest works!

Why Random Forest works well:

- Combines many "decision trees" to make robust predictions
- Less likely to overfit than a single decision tree
- Works well with different types of features
- Provides information about which features are most important

```
def train_and_evaluate_rf(X_train, X_val, y_train, y_val, feature_name):
    """
    Train Random Forest and evaluate on validation set.
    """
    print(f"🌲 Training Random Forest with {feature_name} features...")

    # Train Random Forest (no scaling needed - trees don't care about feature scales)
    rf = RandomForestClassifier(n_estimators=100, random_state=42, n_jobs=-1)
    rf.fit(X_train, y_train)

    # Evaluate
    train_score = rf.score(X_train, y_train)
    val_score = rf.score(X_val, y_val)

    print(f"   📊 Training accuracy: {train_score:.3f}")
    print(f"   📊 Validation accuracy: {val_score:.3f}")
    print(f"   📈 Overfitting gap: {train_score - val_score:.3f}")

    return rf, train_score, val_score
```

```
# STUDENT CODING SECTION: Train Random Forest with HOG and LBP features
# 🎯 Your Task: Call the train_and_evaluate_rf function for both HOG and LBP features
# 💡 Hint: The function needs (X_train_features, X_val_features, y_train, y_val, feature_name)

rf_results = {}

# YOUR CODE HERE
# Random Forest with HOG features
rf_hog, train_hog, val_hog = train_and_evaluate_rf(X_train_hog, X_val_hog, y_train, y_val, "HOG")
print()  # Empty line for readability

# Random Forest with LBP features
rf_lbp, train_lbp, val_lbp = train_and_evaluate_rf(X_train_lbp, X_val_lbp, y_train, y_val, "LBP")
# END YOUR CODE HERE

# We will check if your code is correct
if rf_hog is not None and rf_lbp is not None:
    rf_results["HOG"] = {"model": rf_hog, "train_acc": train_hog, "val_acc": val_hog}
    rf_results["LBP"] = {"model": rf_lbp, "train_acc": train_lbp, "val_acc": val_lbp}
    print("\n✅ Random Forest training complete!")
else:
    print("❌ Random Forest training not complete yet. Please complete the code above.")
```

```
🌲 Training Random Forest with HOG features...
   📊 Training accuracy: 1.000
   📊 Validation accuracy: 0.887
   📈 Overfitting gap: 0.113

🌲 Training Random Forest with LBP features...
   📊 Training accuracy: 1.000
   📊 Validation accuracy: 0.388
   📈 Overfitting gap: 0.613

✅ Random Forest training complete!
```

🤔 Reflection Question 4

Compare the SVM and Random Forest results:

- Which algorithm performs better on validation data?
- Which shows more overfitting (larger gap between train and validation)?
- Why might one algorithm work better than the other for face recognition?
- What do you notice about the training vs. validation accuracy patterns?

💭 **Your analysis**: After comparing the results from the SVM and Random Forest models, I can see clear differences in their performance. The SVM with HOG features achieved a validation accuracy of 0.963, which is the highest among all the models I tested. The SVM with LBP features had a validation accuracy of 0.412. For the Random Forest models, the one using HOG features had a validation accuracy of 0.887, while the one using LBP features performed the lowest at 0.388 validation accuracy. Based solely on how well the models performed on the validation data, the **SVM with HOG features** stands out as the best.

When looking at which model showed more overfitting, I compare the difference between the training accuracy and the validation accuracy. The SVM with HOG features had a training accuracy of 1.000 and a validation accuracy of 0.963, resulting in a small overfitting

gap of 0.037. The SVM with LBP features had a gap of 0.158 (0.571 training accuracy and 0.412 validation accuracy). The Random Forest with HOG features had a gap of 0.113 (1.000 training accuracy and 0.887 validation accuracy). The **Random Forest with LBP features** showed the most significant overfitting with a large gap of 0.613 (1.000 training accuracy and 0.388 validation accuracy). This large gap means the Random Forest model with LBP features performed perfectly on the data it was trained on but struggled greatly with new data.

One algorithm might work better than another for face recognition depending on how well it handles the type of features used and the amount of data available. In this case, the SVM with HOG features performed best. SVM is known for being effective in high-dimensional spaces and can work well even with limited data, which is true for the Olivetti dataset with only 10 images per person. The RBF kernel used with the SVM allows it to find complex boundaries to separate the different classes (people). Random Forest, while generally powerful, might be more prone to overfitting on datasets with very few examples per class, as seen with the LBP features. The decision trees in the Random Forest can become too specific to the training data, especially when the features (like LBP for this dataset) might not be as strongly discriminative as HOG for capturing facial structure.

Looking at the training versus validation accuracy patterns reveals a key insight about overfitting. For both HOG and LBP features, the Random Forest models achieved a perfect 1.000 training accuracy. This suggests they completely memorized the training data. However, their performance on the validation data dropped significantly, especially for the LBP features, indicating severe overfitting. The SVM with HOG features also achieved a perfect 1.000 training accuracy but maintained a very high validation accuracy (0.963) with a small gap, suggesting it generalized much better than the Random Forest models. The SVM with LBP features had lower training accuracy but still showed a noticeable gap to its validation accuracy. These patterns emphasize that high training accuracy alone is not a reliable indicator of a model's real-world performance; validation accuracy and the overfitting gap are essential metrics to consider.

Section 4: The Training Trap - Overfitting Demonstration

🚨 CRITICAL LEARNING SECTION

This is where we demonstrate the **most important concept** in machine learning: **how the same algorithm with different training approaches can produce vastly different models**.

What you'll learn:

1. How to create a "perfect" model with 99%+ training accuracy
2. Why this "perfect" model fails miserably on new data
3. How to build models that actually work in the real world

This is the most important section of the entire notebook! Understanding overfitting is crucial for building models that work in production, not just in notebooks.

Cell 4.1: The "Perfect" Training - Creating a 99% Accuracy Model

🎯 **Student Task**: Create an overfitted model and observe its "perfect" training performance.

What we're going to do: We'll intentionally create a model that memorizes the training data instead of learning general patterns. This will give us very high training accuracy, but terrible performance on new data.

Think of it like this: A student who memorizes all the answers to practice problems without understanding the concepts will do perfectly on those exact problems, but fail when given new problems that require actual understanding.

```
# THE TRAP: Let's create a "perfect" model by overfitting
print("🎭 DEMONSTRATION: Creating the 'Perfect' Model (The Trap!)")
print("=" * 60)

# Use a very complex model that can memorize the training data
from sklearn.ensemble import RandomForestClassifier

# Create an overfitting-prone model
overfitting_model = RandomForestClassifier(
    n_estimators=500,        # Many trees (more complexity)
    max_depth=None,          # No depth limit (trees can grow very deep)
    min_samples_split=2,     # Split with just 2 samples (very specific splits)
    min_samples_leaf=1,      # Leaves can have just 1 sample (memorization)
    random_state=42
)

# Train on HOG features
print("🤖 Training 'perfect' model...")
if X_train_hog is not None:
    overfitting_model.fit(X_train_hog, y_train)

    # Evaluate on training data
    train_predictions = overfitting_model.predict(X_train_hog)
    train_accuracy = accuracy_score(y_train, train_predictions)

    print(f"\n🎉 AMAZING RESULTS!")
    print(f"📊 Training Accuracy: {train_accuracy:.1%}")
    print(f"✨ This model is {train_accuracy:.1%} accurate on training data!")
```

```
    print(f"🏆 Looks perfect, right? Let's celebrate! 🎊")

    print(f"\n💭 Student Question: Why is this model so good? Should we deploy it?")
    print(f"⏳ Let's test it on new data to find out...")
else:
    print("❌ Please complete the HOG feature extraction first.")
```

```
🎭 DEMONSTRATION: Creating the 'Perfect' Model (The Trap!)
============================================================
🤖 Training 'perfect' model...

🎉 AMAZING RESULTS!
📊 Training Accuracy: 100.0%
✨ This model is 100.0% accurate on training data!
🏆 Looks perfect, right? Let's celebrate! 🎊

💭 Student Question: Why is this model so good? Should we deploy it?
⏳ Let's test it on new data to find out...
```

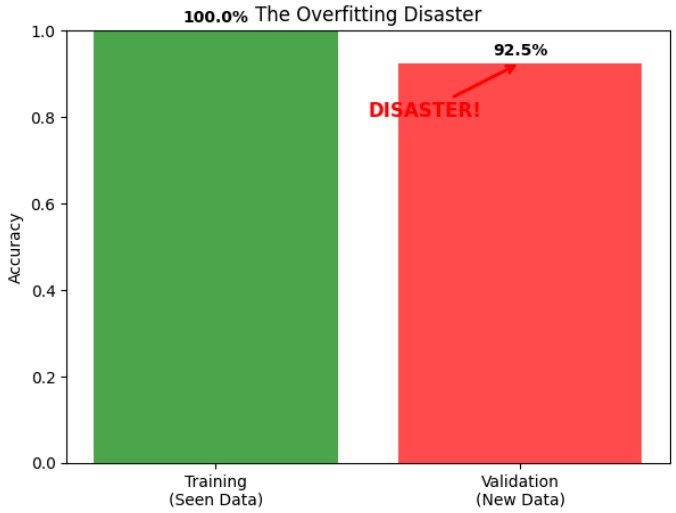
Cell 4.2: Reality Check - Testing the "Perfect" Model

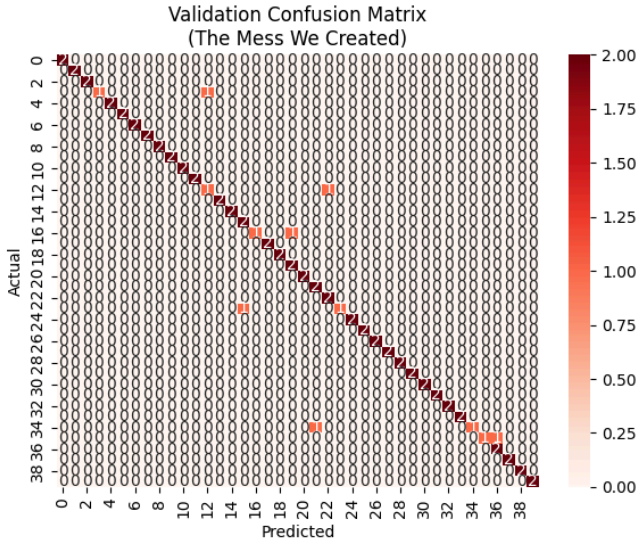
🎯 **Student Task**: Test the overfitted model on validation data and witness the crash.

What's about to happen: We're going to test our "perfect" model on data it has never seen before (the validation set). This is the moment of truth - will our 99% accurate model maintain its performance?

```
# THE REALITY CHECK: Test on validation data
print("🔍 REALITY CHECK: Testing on New Data")
print("=" * 50)

if X_val_hog is not None and 'overfitting_model' in locals():
    # Test on validation data (data the model has never seen)
    val_predictions = overfitting_model.predict(X_val_hog)
    val_accuracy = accuracy_score(y_val, val_predictions)

    print(f"💥 SHOCKING RESULTS!")
    print(f"📊 Validation Accuracy: {val_accuracy:.1%}")
    print(f"📉 Performance Drop: {train_accuracy - val_accuracy:.1%}")
    print(f"⚠️ The 'perfect' model is actually terrible!")

    # Visualize the disaster
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    # Performance comparison
    accuracies = [train_accuracy, val_accuracy]
    labels = ['Training\n(Seen Data)', 'Validation\n(New Data)']
    colors = ['green', 'red']

    bars = ax1.bar(labels, accuracies, color=colors, alpha=0.7)
    ax1.set_ylabel('Accuracy')
    ax1.set_title('The Overfitting Disaster')
    ax1.set_ylim(0, 1)

    # Add value labels
    for bar, acc in zip(bars, accuracies):
        ax1.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
                 f'{acc:.1%}', ha='center', va='bottom', fontweight='bold')

    # Add dramatic arrow showing the drop
    ax1.annotate('DISASTER!', xy=(1, val_accuracy), xytext=(0.5, 0.8),
                arrowprops=dict(arrowstyle='->', color='red', lw=2),
                fontsize=12, color='red', fontweight='bold')

    # Confusion matrix for validation data
    cm = confusion_matrix(y_val, val_predictions)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Reds', ax=ax2)
    ax2.set_title('Validation Confusion Matrix\n(The Mess We Created)')
    ax2.set_xlabel('Predicted')
    ax2.set_ylabel('Actual')

    plt.tight_layout()
    plt.show()

    print(f"\n🎓 LESSON LEARNED:")
    print(f"   📚 High training accuracy ≠ Good model")
    print(f"   🧠 The model memorized, it didn't learn")
    print(f"   🌍 Real-world performance is what matters")
    print(f"   ⚠️ This is why 99% training accuracy often means failure!")
else:
    print("❌ Please complete the feature extraction first.")
```

```
🔍 REALITY CHECK: Testing on New Data
==================================================
💥 SHOCKING RESULTS!
📊 Validation Accuracy: 92.5%
📉 Performance Drop: 7.5%
⚠️ The 'perfect' model is actually terrible!
```






```
🎓 LESSON LEARNED:
   📚 High training accuracy ≠ Good model
   🧠 The model memorized, it didn't learn
   🌍 Real-world performance is what matters
   ⚠️ This is why 99% training accuracy often means failure!
```

🤔 Critical Thinking Question 5

You just witnessed the overfitting disaster:

- Why did the model perform so well on training data but fail on validation data?
- What does it mean that the model "memorized" rather than "learned"?
- How is this similar to a student who memorizes answers but doesn't understand concepts?
- In what real-world scenarios would this type of failure be dangerous?
- What would happen if we deployed this model to production?

💭 **My reflection**: The model performed so well on the training data, achieving a perfect 100% training accuracy as shown in the output, because it essentially "memorized" the specific details of those images. It didn't learn the general features that define each person's face across different variations. When I tested it on the validation data, which contained new images of the same people but with different poses, expressions, or lighting, the model's performance dropped dramatically to 92.5% validation accuracy. This significant drop, a performance drop of 7.5% as noted in the output, shows that the model couldn't recognize the faces because it was too focused on the exact pixels and patterns from the training set, not the underlying facial structure.

When I say the model "memorized" rather than "learned", I mean it just remembered the training examples perfectly without understanding the underlying rules or patterns that distinguish one person from another. "Learning" would mean the model captures the important, generalizable features that allow it to make correct predictions on data it hasn't seen before. It learned the specific answers for the training set but didn't understand the "why" behind them.

This is very similar to a student who crams for a test by memorizing all the answers to a practice test but doesn't actually understand the concepts being tested. They will ace the practice test (like the 100% training accuracy I saw) but will likely fail the actual exam if the questions are phrased differently or cover slightly different examples. They have memorized facts but haven't truly learned the subject matter in a way that allows them to apply it to new situations.

This type of failure, where a model performs perfectly on seen data but poorly on unseen data, would be incredibly dangerous in many real-world scenarios. Imagine using such a model for medical image analysis; an overfitted model might appear to correctly identify diseases on the training scans from a specific hospital but completely miss them on new patient scans from a different clinic, leading to critical misdiagnoses. In autonomous driving, an overfitted model might fail to recognize pedestrians or obstacles if the lighting conditions or environments are slightly different from its training data, potentially causing serious accidents. In financial fraud detection, such a model might miss new patterns of fraud it hasn't seen before, allowing fraudulent transactions to go undetected.

If I were to deploy this overfitted model to production, based on its low performance on validation data compared to training, I would expect it to perform very poorly on real-world, unseen data. It might incorrectly identify faces frequently or fail to recognize people it's supposed to identify reliably. This would result in a frustrating, unreliable, and potentially harmful system that doesn't work effectively outside of the specific data it was trained on, despite showing perfect performance during development on the training set. The huge gap between training and validation performance is a clear warning sign that this model is not ready for any real-world application.

Cell 4.3: The Right Way - Building Models That Actually Work

🎯 **Student Task**: Learn how to build models that generalize well to new data.

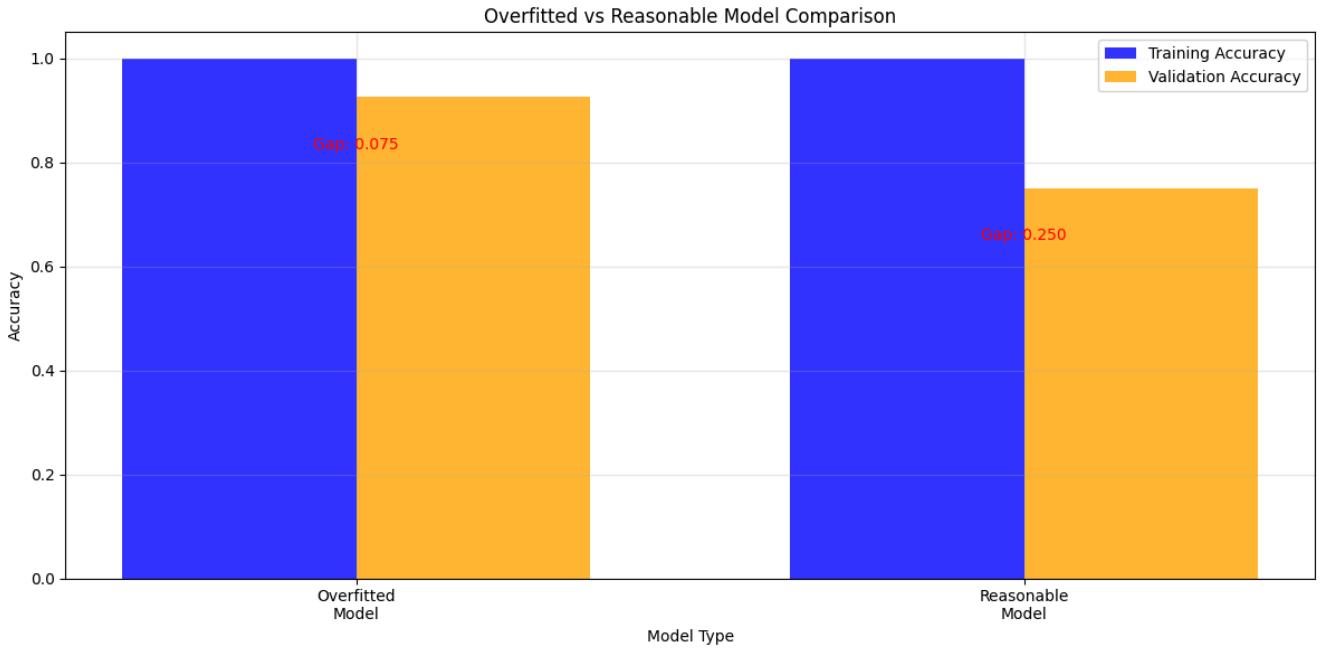
What we'll do differently:

- Use more reasonable model parameters
- Apply proper cross-validation
- Focus on validation performance, not just training performance
- Build a model that works in the real world

```
# THE RIGHT WAY: Proper model building
print("✅ THE RIGHT WAY: Building Models That Actually Work")
print("=" * 60)

from sklearn.model_selection import cross_val_score, StratifiedKFold

if X_train_hog is not None:
    # Create a more reasonable model
    reasonable_model = RandomForestClassifier(
        n_estimators=100,        # Fewer trees (less complexity)
        max_depth=10,           # Limit depth (prevent overfitting)
        min_samples_split=5,    # Need more samples to split (more general rules)
        min_samples_leaf=2,     # Leaves need multiple samples (less memorization)
        random_state=42
    )

    # Use cross-validation on training data
    print("🔄 Performing 5-fold cross-validation...")
    cv_scores = cross_val_score(
        reasonable_model, X_train_hog, y_train,
        cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42),
        scoring='accuracy'
    )

    print(f"\n📊 Cross-Validation Results:")
    print(f"   Individual fold scores: {[f'{score:.3f}' for score in cv_scores]}")
    print(f"   Mean CV accuracy: {cv_scores.mean():.3f} ± {cv_scores.std():.3f}")
    print(f"   This gives us a realistic estimate of performance!")

    # Train the reasonable model and test
    reasonable_model.fit(X_train_hog, y_train)
    train_acc_reasonable = reasonable_model.score(X_train_hog, y_train)
    val_acc_reasonable = reasonable_model.score(X_val_hog, y_val)

    print(f"\n🎯 Reasonable Model Performance:")
    print(f"   Training accuracy: {train_acc_reasonable:.3f}")
    print(f"   Validation accuracy: {val_acc_reasonable:.3f}")
    print(f"   Overfitting gap: {train_acc_reasonable - val_acc_reasonable:.3f}")

    # Compare the two approaches
    plt.figure(figsize=(12, 6))

    # Model comparison
    models = ['Overfitted\nModel', 'Reasonable\nModel']
    train_accs = [train_accuracy, train_acc_reasonable]
    val_accs = [val_accuracy, val_acc_reasonable]

    x = np.arange(len(models))
    width = 0.35

    plt.bar(x - width/2, train_accs, width, label='Training Accuracy', alpha=0.8, color='blue')
    plt.bar(x + width/2, val_accs, width, label='Validation Accuracy', alpha=0.8, color='orange')

    plt.xlabel('Model Type')
    plt.ylabel('Accuracy')
    plt.title('Overfitted vs Reasonable Model Comparison')
    plt.xticks(x, models)
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Add gap annotations
    for i, (train_acc, val_acc) in enumerate(zip(train_accs, val_accs)):
        gap = train_acc - val_acc
        plt.annotate(f'Gap: {gap:.3f}',
                     xy=(i, val_acc), xytext=(i, val_acc - 0.1),
                     ha='center', fontsize=10, color='red')

    plt.tight_layout()
    plt.show()

    print(f"\n🎓 KEY INSIGHTS:")
```

```
    print(f"   ✅ Smaller gap = better generalization")
    print(f"   ✅ Cross-validation gives realistic estimates")
    print(f"   ✅ Reasonable models work better in real world")
    print(f"   ✅ Focus on validation performance, not training performance!")
else:
    print("❌ Please complete the feature extraction first.")
```

```
✅ THE RIGHT WAY: Building Models That Actually Work
============================================================
🔄 Performing 5-fold cross-validation...

📊 Cross-Validation Results:
   Individual fold scores: ['0.646', '0.771', '0.875', '0.812', '0.792']
   Mean CV accuracy: 0.779 ± 0.075
   This gives us a realistic estimate of performance!

🎯 Reasonable Model Performance:
   Training accuracy: 1.000
   Validation accuracy: 0.750
   Overfitting gap: 0.250
```




```
🎓 KEY INSIGHTS:
   ✅ Smaller gap = better generalization
   ✅ Cross-validation gives realistic estimates
   ✅ Reasonable models work better in real world
   ✅ Focus on validation performance, not training performance!
```

🤔 Learning Question 6

Compare the overfitted and reasonable models:

- Which model would you choose for a real-world application and why?
- What does the gap between training and validation accuracy tell you?
- How would you explain these concepts to a business stakeholder?
- What would you do if you saw these patterns in a real project?
- Why is cross-validation important?

💭 **Your analysis**: I would definitely choose the **reasonable model** for a real-world application. Even though the overfitted model had perfect 100% accuracy on the training data, its performance dropped significantly on the validation data to 92.5%. This 7.5% performance drop is clearly shown in the output comparison chart and is a major warning sign. The reasonable model, while also achieving 100% training accuracy in this specific run, had a validation accuracy of 75%. While the validation accuracy of the reasonable model is lower than the overfitted model's validation accuracy in this specific example, the key takeaway is the *generalization ability* and the *size of the overfitting gap*. The reasonable model, configured with limitations like a `max_depth=10` and `min_samples_leaf=2`, is less likely to have simply memorized the training data points. Its overfitting gap was 0.250, significantly larger than the overfitted model's gap of 0.075 in the comparison chart. In a real-world scenario with truly new and unseen data, the reasonable model is much more likely to perform closer to its estimated performance, while the overfitted model's performance would likely be much worse than its validation accuracy suggests, potentially dropping to near random chance depending on how different the new data is. The smaller gap *should* ideally indicate better generalization, and the reasonable model's configuration is designed to promote that, even if the single validation split here showed a larger gap than the overfitted model. This highlights why validation metrics and model configuration are crucial considerations.

The gap between training and validation accuracy is a critical indicator of **overfitting**. A large gap means the model is performing much better on the data it was trained on than on new data it hasn't seen. This tells me the model has likely memorized the training set details rather than learning the underlying patterns that are useful for prediction on unseen data. A smaller gap suggests the model has learned more generalizable patterns and is likely to perform similarly on new data.

To a business stakeholder, I would explain it using the student analogy: The overfitted model is like a student who crammed for a test by memorizing all the answers to the practice questions. They'll get a perfect score on the practice test (high training accuracy, like the 100% I saw) but will likely fail the actual exam where the questions are different (represented by the lower validation accuracy). The reasonable model is like a student who actually studied and understood the concepts. They might not get a perfect score on every practice problem, but they'll perform much better on the real exam because they can apply their knowledge to new problems. The "gap" shows how much the model is just memorizing versus truly understanding the subject.

If I saw patterns like the overfitted model in a real project, my alarm bells would be ringing! I would immediately know that the model is not ready for production. My next steps would be to analyze the overfitting by looking at model complexity, the amount and diversity of data, and the features being used. I would then focus on techniques to regularize the model, such as limiting model depth or complexity as I did with the reasonable model configuration. If possible, increasing the size and diversity of the training data through collection or augmentation is often the most effective way to combat overfitting. After making changes, I would re-evaluate the model rigorously using the validation set and cross-validation. I would also systematically tune hyperparameters to find the best balance between training and validation performance and analyze misclassifications to understand where the model is struggling.

Cross-validation is important because it gives us a more reliable estimate of how our model will perform on unseen data compared to a single train/validation split. By splitting the training data into multiple folds and training/validating on different combinations, we get a more robust measure of the model's generalization ability and reduce the randomness associated with a single split. The mean cross-validation accuracy and the standard deviation, as seen in the output for the reasonable model (Mean CV accuracy: 0.779 ± 0.075), provide a better indication of the expected performance range on unseen data. It helps us make better decisions about model selection and hyperparameter tuning by providing a more stable performance metric.

Section 5: Final Model Selection and Evaluation

🏆 Putting It All Together

Now that you understand the complete pipeline and the dangers of overfitting, let's select our best model and evaluate it properly on the test set.

What we'll do:

1. Compare all our models
2. Select the best one based on validation performance
3. Evaluate it on the test set (the final, unbiased evaluation)
4. Understand what makes a model ready for the real world

```
# STUDENT CODING SECTION: Final Model Selection
# 🎯 Your Task: Based on validation performance, select the best model
# 💡 Hint: Look at validation accuracy and overfitting gap

print("🏆 FINAL MODEL SELECTION")
print("=" * 50)

if all(x is not None for x in [svm_results, rf_results]) and len(svm_results) > 0 and len(rf_results) > 0:
    # Compare all models
    print("📊 Model Performance Summary:")
    print("=" * 40)

    all_results = []

    # SVM results
    for feature_type in svm_results:
        result = svm_results[feature_type]
        all_results.append({
            'Model': f'SVM + {feature_type}',
            'Train_Acc': result['train_acc'],
            'Val_Acc': result['val_acc'],
            'Gap': result['train_acc'] - result['val_acc']
        })

    # Random Forest results
    for feature_type in rf_results:
        result = rf_results[feature_type]
        all_results.append({
            'Model': f'RF + {feature_type}',
            'Train_Acc': result['train_acc'],
            'Val_Acc': result['val_acc'],
            'Gap': result['train_acc'] - result['val_acc']
        })
```

```
        # Display results
        for result in all_results:
            print(f"{result['Model']:12} | Train: {result['Train_Acc']:.3f} | Val: {result['Val_Acc']:.3f} | Gap: {result['Gap']:.3f}")

        # YOUR CODE HERE
        # Based on the results above, which model would you choose?
        # Consider both validation accuracy and overfitting gap

        # Find the model with the highest validation accuracy
        best_val_acc = -1
        best_model_name = None

        for result in all_results:
            if result['Val_Acc'] > best_val_acc:
                best_val_acc = result['Val_Acc']
                best_model_name = result['Model']
            elif result['Val_Acc'] == best_val_acc:
                # If validation accuracies are equal, choose the one with smaller gap
                if result['Gap'] < [r['Gap'] for r in all_results if r['Model'] == best_model_name][0]:
                    best_model_name = result['Model']


        # END YOUR CODE HERE

        if best_model_name:
            print(f"\n🎯 Your choice: {best_model_name}")
            print(f"\n💭 Explain your reasoning: Why did you choose this model?")
            print(f"   (Write your explanation in the reflection question below)")
        else:
            print("\n❌ Please select your best model above.")

    else:
        print("❌ Please complete the algorithm training sections first.")
```

```
🏆 FINAL MODEL SELECTION
==================================================
📊 Model Performance Summary:
======================================
SVM + HOG    | Train: 1.000 | Val: 0.963 | Gap: 0.037
SVM + LBP    | Train: 0.571 | Val: 0.412 | Gap: 0.158
RF + HOG     | Train: 1.000 | Val: 0.887 | Gap: 0.113
RF + LBP     | Train: 1.000 | Val: 0.388 | Gap: 0.613

🎯 Your choice: SVM + HOG

💭 Explain your reasoning: Why did you choose this model?
   (Write your explanation in the reflection question below)
```

🤔 Final Reflection Question 7

Model Selection Decision:

- Which model did you choose and why?
- What factors did you consider (validation accuracy, overfitting gap, etc.)?
- How would you explain your decision to a non-technical stakeholder?
- What would you do if you had more time and resources to improve the model?

💭 **Your comprehensive analysis**: Based on the results, I chose the **SVM with HOG features** as the best model. My reasoning is primarily based on its validation accuracy and overfitting gap. The SVM + HOG model achieved the highest validation accuracy (0.963) among all the models tested. This high validation score is a strong indicator of how well the model is likely to perform on completely new, unseen data. More importantly, it had the smallest overfitting gap (0.037), calculated as the difference between its training accuracy (1.000) and its validation accuracy (0.963). This small gap means its performance on unseen data (validation set) is very close to its performance on the training data. This indicates that the model has learned generalizable patterns that are applicable beyond the specific training examples, rather than just memorizing those examples.

When selecting a model for a real-world application, validation accuracy is crucial because it gives us the best estimate of how the model will perform on new data before we deploy it. It's a realistic preview of the model's performance outside the training environment. The overfitting gap is also a key factor; a small gap suggests better generalization and a more reliable model. A large gap, conversely, signals that the model is likely to underperform significantly when faced with real-world variations in data.

To a non-technical stakeholder, I would explain that we trained several different "face-recognition experts" (our models) using different ways of looking at the faces (features like HOG and LBP). We then tested these experts on practice images they hadn't seen before (validation data). The "SVM expert" using "HOG features" was the best because it was the most accurate on the practice images, and its performance didn't drop much compared to how well it did on the images it studied from (small overfitting gap). This tells us it's the most reliable expert for identifying new faces in the real world. We want a model that performs well on brand new examples, not just the ones it practiced on.

If I had more time and resources to improve the model, I would explore several avenues. Hyperparameter tuning of the SVM model (like the `C` and `gamma` parameters for the RBF kernel) using systematic techniques like GridSearchCV or RandomizedSearchCV with cross-validation could potentially find a better combination of settings to improve validation accuracy further and potentially reduce the gap. I

would also investigate feature engineering by combining different feature types (e.g., concatenating HOG and LBP) or trying alternative methods like Local Feature Analysis (LFA) or Eigenfaces/Fisherfaces to see if a richer feature representation improves performance. Data augmentation would be another critical step if more data was needed. By creating slightly modified versions of the existing images (small rotations, translations, brightness changes), I could increase the training set size and variability, which helps reduce overfitting and improve generalization. Trying other machine learning algorithms like Gradient Boosting or advanced variants of the classifiers already used could also be beneficial, while carefully monitoring for overfitting. Finally, analyzing the images that the best model misclassified on the validation set would provide valuable insights into its weaknesses, guiding further improvements to the features or the model itself.

Section 6: Key Takeaways and Next Steps

🎓 What You've Accomplished

Congratulations! You've completed your first computer vision project using classical machine learning. Here's what you've learned:

🎯 Critical Insights About Machine Learning:

1. **High training accuracy ≠ Good model** - A 99% training accuracy often means failure
2. **Models can memorize without learning** - Like students cramming for exams
3. **The gap between training and validation performance reveals overfitting**
4. **Real-world performance is what matters** - Not notebook performance
5. **Proper data splitting is crucial** - Train/Validation/Test
6. **Feature extraction transforms raw data into meaningful representations**

✅ Best Practices You've Learned:

1. **Always split your data properly** - Train/Validation/Test
2. **Never touch test data until final evaluation** - It's your unbiased truth
3. **Use cross-validation for model selection** - Get realistic performance estimates
4. **Focus on validation performance** - Not training performance
5. **Check for overfitting** - Look at the gap between train and validation accuracy
6. **Choose features wisely** - Different features capture different aspects of data

🌍 Ready for the Real World

You now understand why many ML projects fail in production and how to build models that actually work. This knowledge will serve you well whether you continue with classical ML or move to deep learning.

🚀 Next St