ML Algorithm Templates

Deep Learning & Classical ML for Staff/Principal Interviews

DEEP LEARNING FUNDAMENTALS

1. Neural Network (MLP)

Use when: Tabular data, feature learning, general classification/regression
Key concepts: Forward/backward prop, activation functions, loss functions

```
import numpy as np

class NeuralNetwork:
    def __init__(self, layers):
        """layers: [input_dim, hidden1, hidden2, ...,
 output_dim]"""
        self.weights = []
        self.biases = []

        # Xavier initialization
        for i in range(len(layers)-1):
            w = np.random.randn(layers[i], layers[i
+1]) * np.sqrt(2.0/layers[i])
            b = np.zeros((1, layers[i+1]))
            self.weights.append(w)
            self.biases.append(b)

    def relu(self, z):
        return np.maximum(0, z)

    def relu_derivative(self, z):
        return (z > 0).astype(float)

    def softmax(self, z):
        exp_z = np.exp(z - np.max(z, axis=1, keepdims=
True))
        return exp_z / np.sum(exp_z, axis=1, keepdims=
True)

    def forward(self, X):
        """Forward propagation"""
        self.activations = [X]
        self.z_values = []

        A = X
        for i in range(len(self.weights)-1):
            Z = np.dot(A, self.weights[i]) + self.
biases[i]
            A = self.relu(Z)
            self.z_values.append(Z)
            self.activations.append(A)

        # Output layer (linear for regression, softmax
 for classification)
        Z = np.dot(A, self.weights[-1]) + self.biases
[-1]
        A = self.softmax(Z)  # Change to sigmoid for
binary classification
        self.z_values.append(Z)
        self.activations.append(A)

        return A

    def backward(self, X, y, lr=0.01):
        """Backpropagation with gradient descent"""
        m = X.shape[0]

        # Output layer gradient
        dZ = self.activations[-1] - y  # Cross-entropy
 + softmax
        dW = (1/m) * np.dot(self.activations[-2].T, dZ
)
        db = (1/m) * np.sum(dZ, axis=0, keepdims=True)

        self.weights[-1] -= lr * dW
        self.biases[-1] -= lr * db

        # Hidden layers
        for i in range(len(self.weights)-2, -1, -1):
            dZ = np.dot(dZ, self.weights[i+1].T) *
self.relu_derivative(self.z_values[i])
            dW = (1/m) * np.dot(self.activations[i].T,
 dZ)
            db = (1/m) * np.sum(dZ, axis=0, keepdims=
True)

            self.weights[i] -= lr * dW
            self.biases[i] -= lr * db

    def train(self, X, y, epochs=100, lr=0.01,
batch_size=32):
        """Mini-batch gradient descent"""
        for epoch in range(epochs):
            # Shuffle data
            indices = np.random.permutation(X.shape
[0])
            X_shuffled = X[indices]
            y_shuffled = y[indices]

            # Mini-batch training
            for i in range(0, X.shape[0], batch_size):
                X_batch = X_shuffled[i:i+batch_size]
                y_batch = y_shuffled[i:i+batch_size]

                self.forward(X_batch)
                self.backward(X_batch, y_batch, lr)

            # Calculate loss every 10 epochs
            if epoch % 10 == 0:
                y_pred = self.forward(X)
                loss = -np.mean(np.sum(y * np.log(
y_pred + 1e-8), axis=1))
                print(f"Epoch {epoch}, Loss: {loss:.4f
}")

# Example usage
nn = NeuralNetwork([784, 128, 64, 10])  # MNIST-like
# X_train: (n_samples, 784), y_train: (n_samples, 10)
    one-hot
# nn.train(X_train, y_train, epochs=100, lr=0.01)
```

Key Activation Functions:

```
# ReLU (most common for hidden layers)
relu = lambda x: np.maximum(0, x)
relu_grad = lambda x: (x > 0).astype(float)

# Leaky ReLU (prevents dying neurons)
leaky_relu = lambda x, alpha=0.01: np.where(x > 0, x,
    alpha * x)

# Sigmoid (binary classification output)
sigmoid = lambda x: 1 / (1 + np.exp(-x))
sigmoid_grad = lambda x: sigmoid(x) * (1 - sigmoid(x))

# Tanh (zero-centered, good for RNNs)
tanh = lambda x: np.tanh(x)
tanh_grad = lambda x: 1 - np.tanh(x)**2

# Softmax (multi-class output)
def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True
    ))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True
    )
```

Loss Functions:

```
# Mean Squared Error (regression)
mse = lambda y_true, y_pred: np.mean((y_true - y_pred)
    **2)
mse_grad = lambda y_true, y_pred: 2 * (y_pred - y_true
    ) / len(y_true)

# Binary Cross-Entropy (binary classification)
def binary_crossentropy(y_true, y_pred):
    y_pred = np.clip(y_pred, 1e-7, 1 - 1e-7)
    return -np.mean(y_true * np.log(y_pred) + (1 -
    y_true) * np.log(1 - y_pred))

# Categorical Cross-Entropy (multi-class)
def categorical_crossentropy(y_true, y_pred):
    y_pred = np.clip(y_pred, 1e-7, 1 - 1e-7)
    return -np.mean(np.sum(y_true * np.log(y_pred),
    axis=1))
```

2. Convolutional Neural Network (CNN)

Use when: Images, spatial data, local patterns (CV tasks)
Key concepts: Convolution, pooling, feature maps, receptive field

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class CNN(nn.Module):
    """Classic CNN architecture (LeNet/AlexNet style)"
    ""
    def __init__(self, num_classes=10):
        super(CNN, self).__init__()

        # Convolutional layers
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3,
    padding=1)  # 3 input channels (RGB)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3,
    padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3,
     padding=1)
        self.bn3 = nn.BatchNorm2d(128)

        # Pooling
        self.pool = nn.MaxPool2d(kernel_size=2, stride
    =2)

        # Fully connected layers
        # Input size calculation: 128 channels * (H/8)
     * (W/8)
        # For 32x32 input: 128 * 4 * 4 = 2048
        self.fc1 = nn.Linear(128 * 4 * 4, 256)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(256, num_classes)

    def forward(self, x):
```

```python
        # Conv block 1
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
      # 32x32 -> 16x16

        # Conv block 2
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
      # 16x16 -> 8x8

        # Conv block 3
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
      # 8x8 -> 4x4

        # Flatten
        x = x.view(x.size(0), -1)  # (batch, 128*4*4)

        # FC layers
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x

# Advanced: ResNet Block (residual connections)
class ResidualBlock(nn.Module):
    """ResNet building block with skip connections"""
    def __init__(self, in_channels, out_channels,
     stride=1):
        super(ResidualBlock, self).__init__()

        self.conv1 = nn.Conv2d(in_channels,
     out_channels, kernel_size=3,
                               stride=stride, padding
     =1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels,
     out_channels, kernel_size=3,
                               stride=1, padding=1,
     bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Skip connection (identity or projection)
        self.skip = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.skip = nn.Sequential(
                nn.Conv2d(in_channels, out_channels,
     kernel_size=1,
                         stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        residual = self.skip(x)

        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += residual  # Skip connection
        out = F.relu(out)

        return out

# Usage
model = CNN(num_classes=10)
# Input: (batch, 3, 32, 32) -> Output: (batch, 10)
```

CNN Key Concepts:

```python
# Convolution output size calculation
def conv_output_size(input_size, kernel_size, stride
    =1, padding=0):
    return (input_size - kernel_size + 2*padding) //
     stride + 1

# Example: 32x32 input, 3x3 kernel, stride=1, padding
    =1
# Output: (32 - 3 + 2*1) / 1 + 1 = 32 (same size)

# Receptive field calculation
def receptive_field(layers):
    """Calculate receptive field of stacked conv
     layers"""
    rf = 1
    for kernel_size, stride in layers:
        rf = rf + (kernel_size - 1) * stride
    return rf

# Data augmentation for images
import torchvision.transforms as transforms

train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ColorJitter(brightness=0.2, contrast
     =0.2),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
     0.5))
])
```

3. Recurrent Neural Network (RNN/LSTM/GRU)

Use when: Sequential data, time series, NLP, variable-length inputs
Key concepts: Hidden state, temporal dependencies, vanishing gradient

```python
import torch
import torch.nn as nn

class LSTM_Model(nn.Module):
    """LSTM for sequence classification/generation"""
    def __init__(self, vocab_size, embedding_dim,
     hidden_dim, num_layers, num_classes):
        super(LSTM_Model, self).__init__()

        # Embedding layer
        self.embedding = nn.Embedding(vocab_size,
     embedding_dim)

        # LSTM layer
        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            batch_first=True,  # Input: (batch,
     seq_len, features)
            dropout=0.3 if num_layers > 1 else 0,
            bidirectional=False
        )

        # Fully connected output
        self.fc = nn.Linear(hidden_dim, num_classes)

        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

    def forward(self, x, hidden=None):
        """
        x: (batch, seq_len) - token indices
        """
        # Embedding
        embedded = self.embedding(x)  # (batch,
    seq_len, embedding_dim)

        # LSTM
        if hidden is None:
            # Initialize hidden state and cell state
            h0 = torch.zeros(self.num_layers, x.size
    (0), self.hidden_dim).to(x.device)
            c0 = torch.zeros(self.num_layers, x.size
    (0), self.hidden_dim).to(x.device)
            hidden = (h0, c0)

        lstm_out, hidden = self.lstm(embedded, hidden)
      # lstm_out: (batch, seq_len, hidden_dim)

        # Use last timestep for classification
        last_output = lstm_out[:, -1, :]  # (batch,
    hidden_dim)

        # Or use all timesteps for sequence labeling
        # all_outputs = lstm_out  # (batch, seq_len,
    hidden_dim)

        output = self.fc(last_output)  # (batch,
    num_classes)

        return output, hidden

# Bidirectional LSTM (better for NLP tasks)
class BiLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim,
    hidden_dim, num_layers, num_classes):
        super(BiLSTM, self).__init__()

        self.embedding = nn.Embedding(vocab_size,
    embedding_dim)
        self.lstm = nn.LSTM(
            embedding_dim, hidden_dim, num_layers,
            batch_first=True, dropout=0.3,
    bidirectional=True
        )
        # *2 because bidirectional
        self.fc = nn.Linear(hidden_dim * 2,
    num_classes)

    def forward(self, x):
        embedded = self.embedding(x)
        lstm_out, _ = self.lstm(embedded)
        # Concatenate forward and backward hidden
    states
        last_output = lstm_out[:, -1, :]
        output = self.fc(last_output)
        return output

# GRU (simpler, often works as well as LSTM)
class GRU_Model(nn.Module):
    def __init__(self, input_size, hidden_size,
    num_layers, output_size):
        super(GRU_Model, self).__init__()

        self.gru = nn.GRU(
            input_size, hidden_size, num_layers,
            batch_first=True, dropout=0.3
        )
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        gru_out, _ = self.gru(x)
        output = self.fc(gru_out[:, -1, :])
        return output
```

```python
# Sequence-to-Sequence with attention (for translation
    , summarization)
class Seq2SeqAttention(nn.Module):
    def __init__(self, vocab_size, embedding_dim,
    hidden_dim):
        super(Seq2SeqAttention, self).__init__()

        self.embedding = nn.Embedding(vocab_size,
    embedding_dim)

        # Encoder
        self.encoder = nn.LSTM(embedding_dim,
    hidden_dim, batch_first=True)

        # Decoder
        self.decoder = nn.LSTM(embedding_dim,
    hidden_dim, batch_first=True)

        # Attention
        self.attention = nn.Linear(hidden_dim * 2, 1)

        # Output
        self.fc = nn.Linear(hidden_dim, vocab_size)

    def forward(self, src, trg):
        # Encode
        src_embedded = self.embedding(src)
        encoder_outputs, (hidden, cell) = self.encoder
    (src_embedded)

        # Decode with attention
        trg_embedded = self.embedding(trg)
        outputs = []

        for t in range(trg.size(1)):
            # Attention weights
            decoder_hidden = hidden[-1].unsqueeze(1).
    repeat(1, encoder_outputs.size(1), 1)
            energy = torch.tanh(self.attention(torch.
    cat([decoder_hidden, encoder_outputs], dim=2)))
            attention_weights = torch.softmax(energy.
    squeeze(2), dim=1)

            # Context vector
            context = torch.bmm(attention_weights.
    unsqueeze(1), encoder_outputs)

            # Decoder step
            decoder_input = trg_embedded[:, t:t+1, :]
            decoder_output, (hidden, cell) = self.
    decoder(decoder_input, (hidden, cell))

            # Prediction
            output = self.fc(decoder_output)
            outputs.append(output)

        return torch.cat(outputs, dim=1)

# Usage
model = LSTM_Model(vocab_size=10000, embedding_dim
    =128, hidden_dim=256,
                  num_layers=2, num_classes=5)
```

4. Transformer & Attention

Use when: NLP, long sequences, parallel processing (BERT, GPT, ViT)
Key concepts: Self-attention, positional encoding, multi-head attention

```python
import torch
import torch.nn as nn
import math

class MultiHeadAttention(nn.Module):
    """Multi-head self-attention mechanism"""
    def __init__(self, d_model, num_heads, dropout
    =0.1):
        super(MultiHeadAttention, self).__init__()
        assert d_model % num_heads == 0

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads  # Dimension
    per head

        # Linear projections for Q, K, V
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

        self.dropout = nn.Dropout(dropout)
        self.scale = math.sqrt(self.d_k)

    def forward(self, query, key, value, mask=None):
        batch_size = query.size(0)

        # Linear projections and split into heads
        Q = self.W_q(query).view(batch_size, -1, self.
    num_heads, self.d_k).transpose(1, 2)
        K = self.W_k(key).view(batch_size, -1, self.
    num_heads, self.d_k).transpose(1, 2)
        V = self.W_v(value).view(batch_size, -1, self.
    num_heads, self.d_k).transpose(1, 2)
        # Q, K, V: (batch, num_heads, seq_len, d_k)

        # Scaled dot-product attention
        scores = torch.matmul(Q, K.transpose(-2, -1))
    / self.scale  # (batch, num_heads, seq_len,
    seq_len)

        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1
    e9)

        attention_weights = torch.softmax(scores, dim
    =-1)
        attention_weights = self.dropout(
    attention_weights)

        # Apply attention to values
        context = torch.matmul(attention_weights, V)
    # (batch, num_heads, seq_len, d_k)

        # Concatenate heads
        context = context.transpose(1, 2).contiguous()
    .view(batch_size, -1, self.d_model)

        # Final linear projection
        output = self.W_o(context)

        return output, attention_weights

class PositionalEncoding(nn.Module):
    """Sinusoidal positional encoding"""
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__init__()

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=
    torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model,
    2).float() * (-math.log(10000.0) / d_model))

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        pe = pe.unsqueeze(0)  # (1, max_len, d_model)
        self.register_buffer('pe', pe)

    def forward(self, x):
        """x: (batch, seq_len, d_model)"""
        return x + self.pe[:, :x.size(1), :]

class TransformerEncoderLayer(nn.Module):
    """Single Transformer encoder layer"""
    def __init__(self, d_model, num_heads, d_ff,
    dropout=0.1):
        super(TransformerEncoderLayer, self).__init__
    ()

        # Multi-head attention
        self.attention = MultiHeadAttention(d_model,
    num_heads, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)

        # Feed-forward network
        self.ff = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model)
        )
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Multi-head attention with residual
    connection
        attn_output, _ = self.attention(x, x, x, mask)
        x = self.norm1(x + self.dropout1(attn_output))

        # Feed-forward with residual connection
        ff_output = self.ff(x)
        x = self.norm2(x + self.dropout2(ff_output))

        return x

class TransformerClassifier(nn.Module):
    """Transformer for sequence classification (BERT-
    style)"""
    def __init__(self, vocab_size, d_model, num_heads,
     num_layers, d_ff, num_classes, max_len=512,
    dropout=0.1):
        super(TransformerClassifier, self).__init__()

        # Embeddings
        self.embedding = nn.Embedding(vocab_size,
    d_model)
        self.pos_encoding = PositionalEncoding(d_model
    , max_len)
        self.dropout = nn.Dropout(dropout)

        # Transformer encoder layers
        self.encoder_layers = nn.ModuleList([
            TransformerEncoderLayer(d_model, num_heads
    , d_ff, dropout)
            for _ in range(num_layers)
        ])
```

```
        # Classification head
        self.fc = nn.Linear(d_model, num_classes)

    def forward(self, x, mask=None):
        """
        x: (batch, seq_len) - token indices
        mask: (batch, 1, seq_len, seq_len) - attention
     mask
        """
        # Embedding + positional encoding
        x = self.embedding(x) * math.sqrt(self.
    embedding.embedding_dim)
        x = self.pos_encoding(x)
        x = self.dropout(x)

        # Transformer encoder layers
        for layer in self.encoder_layers:
            x = layer(x, mask)

        # Use [CLS] token (first token) for
    classification
        cls_output = x[:, 0, :]
        output = self.fc(cls_output)

        return output

# Vision Transformer (ViT) for images
class VisionTransformer(nn.Module):
    """ViT: Treat image patches as sequence tokens"""
    def __init__(self, img_size=224, patch_size=16,
    num_classes=1000,
                 d_model=768, num_heads=12, num_layers
    =12, d_ff=3072):
        super(VisionTransformer, self).__init__()

        self.patch_size = patch_size
        num_patches = (img_size // patch_size) ** 2

        # Patch embedding (flatten patches and linear
    projection)
        self.patch_embed = nn.Conv2d(3, d_model,
    kernel_size=patch_size, stride=patch_size)

        # Learnable [CLS] token and positional
    embeddings
        self.cls_token = nn.Parameter(torch.randn(1,
    1, d_model))
        self.pos_embed = nn.Parameter(torch.randn(1,
    num_patches + 1, d_model))

        # Transformer encoder
        self.encoder_layers = nn.ModuleList([
            TransformerEncoderLayer(d_model, num_heads
    , d_ff)
            for _ in range(num_layers)
        ])

        self.norm = nn.LayerNorm(d_model)
        self.fc = nn.Linear(d_model, num_classes)

    def forward(self, x):
        """x: (batch, 3, H, W)"""
        batch_size = x.size(0)

        # Patch embedding
        x = self.patch_embed(x)  # (batch, d_model, H/
    P, W/P)
        x = x.flatten(2).transpose(1, 2)  # (batch,
    num_patches, d_model)

        # Add [CLS] token
        cls_tokens = self.cls_token.expand(batch_size,
     -1, -1)
        x = torch.cat([cls_tokens, x], dim=1)  # (
    batch, num_patches+1, d_model)

        # Add positional encoding
        x = x + self.pos_embed

        # Transformer encoder
        for layer in self.encoder_layers:
            x = layer(x)

        x = self.norm(x)

        # Classification using [CLS] token
        cls_output = x[:, 0]
        output = self.fc(cls_output)

        return output

# Usage
model = TransformerClassifier(
    vocab_size=30000, d_model=512, num_heads=8,
    num_layers=6, d_ff=2048, num_classes=2
)
```

5. Autoencoders & VAE

Use when: Dimensionality reduction, anomaly detection, generation
Key concepts: Encoding, latent space, reconstruction, KL divergence

```
import torch
import torch.nn as nn

class Autoencoder(nn.Module):
    """Basic autoencoder for dimensionality reduction"
    ""
    def __init__(self, input_dim, encoding_dim):
        super(Autoencoder, self).__init__()

        # Encoder
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, encoding_dim)
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(encoding_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, input_dim),
            nn.Sigmoid()  # For normalized input in
    [0, 1]
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

    def encode(self, x):
        return self.encoder(x)

# Variational Autoencoder (VAE)
class VAE(nn.Module):
    """VAE for generative modeling"""
    def __init__(self, input_dim, latent_dim):
        super(VAE, self).__init__()

        # Encoder
        self.fc1 = nn.Linear(input_dim, 256)
        self.fc2_mu = nn.Linear(256, latent_dim)
    # Mean of latent distribution
        self.fc2_logvar = nn.Linear(256, latent_dim)
    # Log variance

        # Decoder
        self.fc3 = nn.Linear(latent_dim, 256)
        self.fc4 = nn.Linear(256, input_dim)

    def encode(self, x):
        h = torch.relu(self.fc1(x))
        return self.fc2_mu(h), self.fc2_logvar(h)

    def reparameterize(self, mu, logvar):
        """Reparameterization trick: z = mu + sigma *
    epsilon"""
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z):
        h = torch.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h))

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        reconstructed = self.decode(z)
        return reconstructed, mu, logvar

def vae_loss(reconstructed, x, mu, logvar):
    """VAE loss = Reconstruction loss + KL divergence"
    ""
    # Reconstruction loss (Binary Cross-Entropy)
    recon_loss = nn.functional.binary_cross_entropy(
    reconstructed, x, reduction='sum')

    # KL divergence: -0.5 * sum(1 + log(sigma^2) - mu
    ^2 - sigma^2)
    kl_div = -0.5 * torch.sum(1 + logvar - mu.pow(2) -
     logvar.exp())

    return recon_loss + kl_div

# Convolutional Autoencoder for images
class ConvAutoencoder(nn.Module):
    def __init__(self):
        super(ConvAutoencoder, self).__init__()

        # Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=2, padding=1),
    # 28x28 -> 14x14
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
    # 14x14 -> 7x7
            nn.ReLU(),
            nn.Conv2d(32, 64, 7)  # 7x7 -> 1x1
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 7),  # 1x1 -> 7
    x7
```

```
            nn.ReLU(),
            nn.ConvTranspose2d(32, 16, 3, stride=2,
    padding=1, output_padding=1),  # 7x7 -> 14x14
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 3, stride=2,
    padding=1, output_padding=1),   # 14x14 -> 28x28
            nn.Sigmoid()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

6. Generative Adversarial Network (GAN)

Use when: Image generation, data augmentation, style transfer
Key concepts: Generator, discriminator, adversarial training, Nash equilibrium

```
import torch
import torch.nn as nn

class Generator(nn.Module):
    """GAN generator: noise -> fake images"""
    def __init__(self, latent_dim, img_channels,
    img_size):
        super(Generator, self).__init__()

        self.init_size = img_size // 4  # Initial
    spatial size
        self.l1 = nn.Linear(latent_dim, 128 * self.
    init_size ** 2)

        self.conv_blocks = nn.Sequential(
            nn.BatchNorm2d(128),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 128, 3, stride=1, padding
    =1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 64, 3, stride=1, padding=1)
    ,
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2),
            nn.Conv2d(64, img_channels, 3, stride=1,
    padding=1),
            nn.Tanh()  # Output in [-1, 1]
        )

    def forward(self, z):
        """z: (batch, latent_dim) - random noise"""
        out = self.l1(z)
        out = out.view(out.size(0), 128, self.
    init_size, self.init_size)
        img = self.conv_blocks(out)
        return img

class Discriminator(nn.Module):
    """GAN discriminator: images -> real/fake
    probability"""
    def __init__(self, img_channels, img_size):
        super(Discriminator, self).__init__()

        def discriminator_block(in_channels,
    out_channels, bn=True):
            layers = [nn.Conv2d(in_channels,
    out_channels, 3, 2, 1)]
            if bn:
                layers.append(nn.BatchNorm2d(
    out_channels))
            layers.append(nn.LeakyReLU(0.2))
            layers.append(nn.Dropout2d(0.25))
            return layers

        self.model = nn.Sequential(
            *discriminator_block(img_channels, 16, bn=
    False),
            *discriminator_block(16, 32),
            *discriminator_block(32, 64),
            *discriminator_block(64, 128),
        )

        # Output: probability of real
        ds_size = img_size // 2 ** 4
        self.adv_layer = nn.Sequential(
            nn.Linear(128 * ds_size ** 2, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        out = self.model(img)
        out = out.view(out.size(0), -1)
        validity = self.adv_layer(out)
        return validity

# Training GAN
def train_gan(generator, discriminator, dataloader,
    num_epochs, latent_dim):
    device = torch.device("cuda" if torch.cuda.
    is_available() else "cpu")

    adversarial_loss = nn.BCELoss()

    optimizer_G = torch.optim.Adam(generator.
    parameters(), lr=0.0002, betas=(0.5, 0.999))
    optimizer_D = torch.optim.Adam(discriminator.
    parameters(), lr=0.0002, betas=(0.5, 0.999))

    for epoch in range(num_epochs):
        for i, (imgs, _) in enumerate(dataloader):
            batch_size = imgs.size(0)

            # Labels
            real_labels = torch.ones(batch_size, 1).to
    (device)
            fake_labels = torch.zeros(batch_size, 1).
    to(device)

            # Real images
            real_imgs = imgs.to(device)

            # ---------------------
            #  Train Discriminator
            # ---------------------
            optimizer_D.zero_grad()

            # Real images
            real_loss = adversarial_loss(discriminator
    (real_imgs), real_labels)

            # Fake images
            z = torch.randn(batch_size, latent_dim).to
    (device)
            fake_imgs = generator(z)
            fake_loss = adversarial_loss(discriminator
    (fake_imgs.detach()), fake_labels)

            # Total discriminator loss
            d_loss = (real_loss + fake_loss) / 2
            d_loss.backward()
            optimizer_D.step()

            # -----------------
            #  Train Generator
            # -----------------
            optimizer_G.zero_grad()

            # Generate fake images
            z = torch.randn(batch_size, latent_dim).to
    (device)
            gen_imgs = generator(z)

            # Generator loss (fool discriminator)
            g_loss = adversarial_loss(discriminator(
    gen_imgs), real_labels)

            g_loss.backward()
            optimizer_G.step()

        print(f"Epoch [{epoch}/{num_epochs}] D_loss: {
    d_loss.item():.4f}, G_loss: {g_loss.item():.4f}")

# Conditional GAN (cGAN) - generate images from class
    labels
class ConditionalGenerator(nn.Module):
    def __init__(self, latent_dim, num_classes,
    img_channels, img_size):
        super(ConditionalGenerator, self).__init__()

        self.label_emb = nn.Embedding(num_classes,
    latent_dim)

        # Rest similar to Generator, but input is
    latent_dim + latent_dim (concatenated)
        self.init_size = img_size // 4
        self.l1 = nn.Linear(latent_dim * 2, 128 * self
    .init_size ** 2)
        # ... (same conv_blocks as Generator)

    def forward(self, z, labels):
        # Concatenate noise and label embedding
        gen_input = torch.cat([z, self.label_emb(
    labels)], dim=1)
        out = self.l1(gen_input)
        # ... (same as Generator)
        return img

# DCGAN (Deep Convolutional GAN) is the standard
    architecture shown above
```

CLASSICAL MACHINE LEARNING

7. Linear & Logistic Regression

Use when: Baseline model, linear relationships, interpretability
Key concepts: Gradient descent, regularization (L1/L2), feature scaling

```
import numpy as np

class LinearRegression:
    """Linear regression with gradient descent"""
```

```
    def __init__(self, lr=0.01, n_iters=1000,
    regularization=None, lambda_=0.01):
        self.lr = lr
        self.n_iters = n_iters
        self.regularization = regularization  # None,
    'l1', or 'l2'
        self.lambda_ = lambda_
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        n_samples, n_features = X.shape

        # Initialize parameters
        self.weights = np.zeros(n_features)
        self.bias = 0

        # Gradient descent
        for _ in range(self.n_iters):
            # Forward pass
            y_pred = np.dot(X, self.weights) + self.
    bias

            # Compute gradients
            dw = (1/n_samples) * np.dot(X.T, (y_pred -
     y))
            db = (1/n_samples) * np.sum(y_pred - y)

            # Add regularization
            if self.regularization == 'l2':
                dw += (self.lambda_ / n_samples) *
    self.weights
            elif self.regularization == 'l1':
                dw += (self.lambda_ / n_samples) * np.
    sign(self.weights)

            # Update parameters
            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    def predict(self, X):
        return np.dot(X, self.weights) + self.bias

class LogisticRegression:
    """Logistic regression for binary classification""
    "
    def __init__(self, lr=0.01, n_iters=1000):
        self.lr = lr
        self.n_iters = n_iters
        self.weights = None
        self.bias = None

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-np.clip(z, -500, 500))
    )  # Clip to prevent overflow

    def fit(self, X, y):
        n_samples, n_features = X.shape

        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iters):
            # Forward pass
            linear_pred = np.dot(X, self.weights) +
    self.bias
            predictions = self.sigmoid(linear_pred)

            # Gradients
            dw = (1/n_samples) * np.dot(X.T, (
    predictions - y))
            db = (1/n_samples) * np.sum(predictions -
    y)

            # Update
            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    def predict(self, X):
        linear_pred = np.dot(X, self.weights) + self.
    bias
        y_pred = self.sigmoid(linear_pred)
        return (y_pred >= 0.5).astype(int)

    def predict_proba(self, X):
        linear_pred = np.dot(X, self.weights) + self.
    bias
        return self.sigmoid(linear_pred)

# Closed-form solution (Normal Equation) for linear
    regression
def normal_equation(X, y):
    """Analytical solution: w = (X^T X)^-1 X^T y"""
    # Add bias term
    X_b = np.c_[np.ones((X.shape[0], 1)), X]
    # Solve
    theta = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).
    dot(y)
    return theta[1:], theta[0]  # weights, bias
```

8. Decision Trees & Random Forest

Use when: Non-linear relationships, interpretability, feature interactions
Key concepts: Gini impurity, information gain, ensemble methods

```
import numpy as np
from collections import Counter

class Node:
    def __init__(self, feature=None, threshold=None,
    left=None, right=None, value=None):
        self.feature = feature      # Feature index to
     split on
        self.threshold = threshold  # Threshold value
        self.left = left           # Left child
        self.right = right         # Right child
        self.value = value         # Leaf node value (
    class or mean)

class DecisionTree:
    """Decision tree for classification or regression"
    ""
    def __init__(self, max_depth=10, min_samples_split
    =2, task='classification'):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.task = task
        self.root = None

    def gini_impurity(self, y):
        """Gini impurity for classification"""
        counter = Counter(y)
        impurity = 1.0
        for count in counter.values():
            prob = count / len(y)
            impurity -= prob ** 2
        return impurity

    def variance(self, y):
        """Variance for regression"""
        if len(y) == 0:
            return 0
        return np.var(y)

    def information_gain(self, y, y_left, y_right):
        """Calculate information gain from split"""
        if self.task == 'classification':
            parent_impurity = self.gini_impurity(y)
            n = len(y)
            n_left, n_right = len(y_left), len(y_right
)
            child_impurity = (n_left/n) * self.
gini_impurity(y_left) + (n_right/n) * self.
gini_impurity(y_right)
        else:
            parent_impurity = self.variance(y)
            n = len(y)
            n_left, n_right = len(y_left), len(y_right
)
            child_impurity = (n_left/n) * self.
variance(y_left) + (n_right/n) * self.variance(
y_right)

        return parent_impurity - child_impurity

    def best_split(self, X, y):
        """Find the best feature and threshold to
split on"""
        best_gain = -1
        best_feature = None
        best_threshold = None

        n_features = X.shape[1]

        for feature in range(n_features):
            thresholds = np.unique(X[:, feature])

            for threshold in thresholds:
                # Split
                left_mask = X[:, feature] <= threshold
                right_mask = ~left_mask

                if np.sum(left_mask) == 0 or np.sum(
right_mask) == 0:
                    continue

                y_left = y[left_mask]
                y_right = y[right_mask]

                # Calculate information gain
                gain = self.information_gain(y, y_left
, y_right)

                if gain > best_gain:
                    best_gain = gain
                    best_feature = feature
                    best_threshold = threshold

        return best_feature, best_threshold

    def build_tree(self, X, y, depth=0):
        """Recursively build the decision tree"""
        n_samples, n_features = X.shape
        n_labels = len(np.unique(y))

        # Stopping criteria
        if depth >= self.max_depth or n_labels == 1 or
 n_samples < self.min_samples_split:
            if self.task == 'classification':
                leaf_value = Counter(y).most_common(1)
[0][0]
```

```
            else:
                leaf_value = np.mean(y)
            return Node(value=leaf_value)

        # Find best split
        best_feature, best_threshold = self.best_split
    (X, y)

        if best_feature is None:
            if self.task == 'classification':
                leaf_value = Counter(y).most_common(1)
    [0][0]
            else:
                leaf_value = np.mean(y)
            return Node(value=leaf_value)

        # Split dataset
        left_mask = X[:, best_feature] <=
    best_threshold
        right_mask = ~left_mask

        # Recursively build left and right subtrees
        left = self.build_tree(X[left_mask], y[
    left_mask], depth + 1)
        right = self.build_tree(X[right_mask], y[
    right_mask], depth + 1)

        return Node(feature=best_feature, threshold=
    best_threshold, left=left, right=right)

    def fit(self, X, y):
        self.root = self.build_tree(X, y)

    def predict_sample(self, x, node):
        """Predict a single sample"""
        if node.value is not None:
            return node.value

        if x[node.feature] <= node.threshold:
            return self.predict_sample(x, node.left)
        else:
            return self.predict_sample(x, node.right)

    def predict(self, X):
        return np.array([self.predict_sample(x, self.
    root) for x in X])

class RandomForest:
    """Random Forest ensemble"""
    def __init__(self, n_trees=10, max_depth=10,
    min_samples_split=2, task='classification'):
        self.n_trees = n_trees
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.task = task
        self.trees = []

    def bootstrap_sample(self, X, y):
        """Create bootstrap sample"""
        n_samples = X.shape[0]
        indices = np.random.choice(n_samples, size=
    n_samples, replace=True)
        return X[indices], y[indices]

    def fit(self, X, y):
        self.trees = []
        for _ in range(self.n_trees):
            tree = DecisionTree(max_depth=self.
    max_depth,
                              min_samples_split=self.
    min_samples_split,
                              task=self.task)
            X_sample, y_sample = self.bootstrap_sample
    (X, y)
            tree.fit(X_sample, y_sample)
            self.trees.append(tree)

    def predict(self, X):
        # Get predictions from all trees
        tree_preds = np.array([tree.predict(X) for
    tree in self.trees])

        # Majority vote (classification) or average (
    regression)
        if self.task == 'classification':
            # Transpose to get predictions per sample,
     then mode
            return np.array([Counter(tree_preds[:, i])
    .most_common(1)[0][0]
                           for i in range(X.shape[0])
    ])
        else:
            return np.mean(tree_preds, axis=0)
```

9. Gradient Boosting (XGBoost/LightGBM)

Use when: Tabular data, kaggle competitions, high accuracy needed
Key concepts: Boosting, residual learning, regularization

```
import numpy as np

class GradientBoostingRegressor:
    """Gradient Boosting for regression"""
    def __init__(self, n_estimators=100, learning_rate
    =0.1, max_depth=3):
        self.n_estimators = n_estimators
        self.learning_rate = learning_rate
        self.max_depth = max_depth
        self.trees = []
        self.initial_prediction = None

    def fit(self, X, y):
        # Initialize with mean
        self.initial_prediction = np.mean(y)
        predictions = np.full(len(y), self.
    initial_prediction)

        for _ in range(self.n_estimators):
            # Compute residuals (negative gradient for
     MSE)
            residuals = y - predictions

            # Fit tree to residuals
            tree = DecisionTree(max_depth=self.
    max_depth, task='regression')
            tree.fit(X, residuals)

            # Update predictions
            update = tree.predict(X)
            predictions += self.learning_rate * update

            self.trees.append(tree)

    def predict(self, X):
        predictions = np.full(len(X), self.
    initial_prediction)

        for tree in self.trees:
            predictions += self.learning_rate * tree.
    predict(X)

        return predictions

# Using XGBoost (industry standard)
"""
import xgboost as xgb

# Classification
model = xgb.XGBClassifier(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=6,
    subsample=0.8,           # Row sampling
    colsample_bytree=0.8,    # Column sampling
    reg_alpha=0.1,           # L1 regularization
    reg_lambda=1.0,          # L2 regularization
    eval_metric='logloss'
)

model.fit(X_train, y_train,
          eval_set=[(X_val, y_val)],
          early_stopping_rounds=10,
          verbose=True)

# Feature importance
import matplotlib.pyplot as plt
xgb.plot_importance(model)
plt.show()

# Regression
model = xgb.XGBRegressor(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=6,
    objective='reg:squarederror'
)
"""

# Using LightGBM (faster for large datasets)
"""
import lightgbm as lgb

# Create dataset
train_data = lgb.Dataset(X_train, label=y_train)
val_data = lgb.Dataset(X_val, label=y_val, reference=
    train_data)

# Parameters
params = {
    'objective': 'binary',
    'metric': 'binary_logloss',
    'boosting_type': 'gbdt',
    'num_leaves': 31,
    'learning_rate': 0.05,
    'feature_fraction': 0.9,
    'bagging_fraction': 0.8,
    'bagging_freq': 5,
    'verbose': 0
}

# Train
model = lgb.train(
    params,
    train_data,
    num_boost_round=100,
    valid_sets=[train_data, val_data],
    early_stopping_rounds=10
)
```

```
# Predict
y_pred = model.predict(X_test)
"""
```

10. Support Vector Machine (SVM)

Use when: Small/medium datasets, high-dimensional data, clear margin
Key concepts: Maximum margin, kernel trick, soft margin

```
import numpy as np

class SVM:
    """Support Vector Machine with linear kernel"""
    def __init__(self, lr=0.001, lambda_param=0.01,
 n_iters=1000):
        self.lr = lr
        self.lambda_param = lambda_param  #
 Regularization
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self, X, y):
        n_samples, n_features = X.shape

        # Convert labels to {-1, 1}
        y_ = np.where(y <= 0, -1, 1)

        # Initialize weights
        self.w = np.zeros(n_features)
        self.b = 0

        # Gradient descent
        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                condition = y_[idx] * (np.dot(x_i,
 self.w) - self.b) >= 1

                if condition:
                    # Correctly classified, only
 update regularization
                    self.w -= self.lr * (2 * self.
 lambda_param * self.w)
                else:
                    # Misclassified, update with hinge
 loss gradient
                    self.w -= self.lr * (2 * self.
 lambda_param * self.w - np.dot(x_i, y_[idx]))
                    self.b -= self.lr * y_[idx]

    def predict(self, X):
        linear_output = np.dot(X, self.w) - self.b
        return np.sign(linear_output)

# Kernel SVM (using scikit-learn)
"""
from sklearn.svm import SVC

# Linear kernel
svm_linear = SVC(kernel='linear', C=1.0)

# RBF (Gaussian) kernel - most common
svm_rbf = SVC(kernel='rbf', C=1.0, gamma='scale')

# Polynomial kernel
svm_poly = SVC(kernel='poly', degree=3, C=1.0)

# Custom kernel
def custom_kernel(X1, X2):
    return np.dot(X1, X2.T)

svm_custom = SVC(kernel=custom_kernel)

# Fit
svm_rbf.fit(X_train, y_train)

# Predict
y_pred = svm_rbf.predict(X_test)

# Get support vectors
support_vectors = svm_rbf.support_vectors_
"""
```

11. K-Nearest Neighbors (KNN)

Use when: Small datasets, non-parametric, anomaly detection
Key concepts: Distance metrics, lazy learning, curse of dimensionality

```
import numpy as np
from collections import Counter

class KNN:
    """K-Nearest Neighbors classifier"""
    def __init__(self, k=3, distance_metric='euclidean
 '):
        self.k = k
        self.distance_metric = distance_metric
        self.X_train = None
        self.y_train = None

    def fit(self, X, y):
        """Store training data (lazy learning)"""
        self.X_train = X
        self.y_train = y

    def euclidean_distance(self, x1, x2):
        return np.sqrt(np.sum((x1 - x2) ** 2))

    def manhattan_distance(self, x1, x2):
        return np.sum(np.abs(x1 - x2))

    def cosine_similarity(self, x1, x2):
        dot_product = np.dot(x1, x2)
        norm_product = np.linalg.norm(x1) * np.linalg.
 norm(x2)
        return 1 - (dot_product / norm_product)  #
 Convert to distance

    def predict(self, X):
        predictions = [self._predict_single(x) for x
 in X]
        return np.array(predictions)

    def _predict_single(self, x):
        # Compute distances to all training samples
        if self.distance_metric == 'euclidean':
            distances = [self.euclidean_distance(x,
 x_train) for x_train in self.X_train]
        elif self.distance_metric == 'manhattan':
            distances = [self.manhattan_distance(x,
 x_train) for x_train in self.X_train]
        elif self.distance_metric == 'cosine':
            distances = [self.cosine_similarity(x,
 x_train) for x_train in self.X_train]

        # Get k nearest neighbors
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = self.y_train[k_indices]

        # Majority vote
        most_common = Counter(k_nearest_labels).
 most_common(1)
        return most_common[0][0]

# Optimized KNN with ball tree or KD tree
"""
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(
    n_neighbors=5,
    weights='distance',  # Weight by inverse distance
    algorithm='ball_tree',  # 'ball_tree', 'kd_tree',
 'brute'
    metric='euclidean'
)

knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
"""
```

12. Naive Bayes

Use when: Text classification, spam detection, high-dimensional data
Key concepts: Bayes theorem, conditional independence, prior/posterior

```
import numpy as np

class GaussianNaiveBayes:
    """Naive Bayes with Gaussian distribution
 assumption"""
    def __init__(self):
        self.classes = None
        self.mean = {}
        self.var = {}
        self.priors = {}

    def fit(self, X, y):
        self.classes = np.unique(y)

        for c in self.classes:
            X_c = X[y == c]

            # Calculate mean and variance for each
 feature
            self.mean[c] = np.mean(X_c, axis=0)
            self.var[c] = np.var(X_c, axis=0)

            # Calculate prior probability
            self.priors[c] = X_c.shape[0] / X.shape[0]

    def gaussian_probability(self, x, mean, var):
        """P(x | class) assuming Gaussian distribution
 """
        eps = 1e-6  # Avoid division by zero
        coeff = 1 / np.sqrt(2 * np.pi * var + eps)
        exponent = np.exp(- (x - mean) ** 2 / (2 * var
 + eps))
        return coeff * exponent

    def predict(self, X):
        predictions = []

        for x in X:
            posteriors = []

            for c in self.classes:
```

```python
                # Log probabilities to avoid underflow
                prior = np.log(self.priors[c])

                # Product of conditional probabilities
 (sum in log space)
                conditional = np.sum(np.log(self.
    gaussian_probability(x, self.mean[c], self.var[c])
    ))

                posterior = prior + conditional
                posteriors.append(posterior)

            # Return class with highest posterior
            predictions.append(self.classes[np.argmax(
    posteriors)])

        return np.array(predictions)

# Multinomial Naive Bayes for text
class MultinomialNaiveBayes:
    """Naive Bayes for count/frequency features (text)
    """
    def __init__(self, alpha=1.0):
        self.alpha = alpha  # Laplace smoothing
        self.classes = None
        self.class_priors = {}
        self.feature_probs = {}

    def fit(self, X, y):
        self.classes = np.unique(y)
        n_samples = X.shape[0]

        for c in self.classes:
            X_c = X[y == c]

            # Class prior
            self.class_priors[c] = X_c.shape[0] /
    n_samples

            # Feature probabilities with Laplace
    smoothing
            feature_counts = np.sum(X_c, axis=0) +
    self.alpha
            total_count = np.sum(feature_counts)
            self.feature_probs[c] = feature_counts /
    total_count

    def predict(self, X):
        predictions = []

        for x in X:
            posteriors = []

            for c in self.classes:
                # Log prior
                log_prior = np.log(self.class_priors[c
    ])

                # Log likelihood (sum of log
    probabilities)
                log_likelihood = np.sum(x * np.log(
    self.feature_probs[c]))

                posterior = log_prior + log_likelihood
                posteriors.append(posterior)

            predictions.append(self.classes[np.argmax(
    posteriors)])

        return np.array(predictions)

# Usage for text classification
"""
from sklearn.feature_extraction.text import
    CountVectorizer
from sklearn.naive_bayes import MultinomialNB

# Vectorize text
vectorizer = CountVectorizer()
X_train_counts = vectorizer.fit_transform(train_texts)

# Train
nb = MultinomialNB(alpha=1.0)
nb.fit(X_train_counts, y_train)

# Predict
X_test_counts = vectorizer.transform(test_texts)
y_pred = nb.predict(X_test_counts)
"""
```

13. K-Means Clustering

Use when: Unsupervised learning, customer segmentation, data exploration
Key concepts: Centroids, inertia, elbow method

```python
import numpy as np

class KMeans:
    """K-Means clustering algorithm"""
    def __init__(self, n_clusters=3, max_iters=100,
     tol=1e-4):
        self.n_clusters = n_clusters
        self.max_iters = max_iters
        self.tol = tol  # Convergence tolerance
        self.centroids = None
        self.labels = None

    def fit(self, X):
        n_samples, n_features = X.shape

        # Initialize centroids randomly
        random_indices = np.random.choice(n_samples,
     self.n_clusters, replace=False)
        self.centroids = X[random_indices]

        for _ in range(self.max_iters):
            # Assign samples to nearest centroid
            self.labels = self._assign_clusters(X)

            # Calculate new centroids
            new_centroids = self._calculate_centroids(
     X)

            # Check convergence
            if np.all(np.abs(new_centroids - self.
     centroids) < self.tol):
                break

            self.centroids = new_centroids

        return self

    def _assign_clusters(self, X):
        """Assign each sample to nearest centroid"""
        distances = np.zeros((X.shape[0], self.
     n_clusters))

        for i, centroid in enumerate(self.centroids):
            distances[:, i] = np.linalg.norm(X -
     centroid, axis=1)

        return np.argmin(distances, axis=1)

    def _calculate_centroids(self, X):
        """Calculate mean of samples in each cluster""
"
        centroids = np.zeros((self.n_clusters, X.shape
[1]))

        for i in range(self.n_clusters):
            cluster_samples = X[self.labels == i]
            if len(cluster_samples) > 0:
                centroids[i] = np.mean(cluster_samples
, axis=0)
            else:
                # If cluster is empty, reinitialize
 randomly
                centroids[i] = X[np.random.choice(X.
 shape[0])]

        return centroids

    def predict(self, X):
        """Assign new samples to nearest centroid"""
        return self._assign_clusters(X)

    def inertia(self, X):
        """Sum of squared distances to nearest
 centroid"""
        distances = np.min([np.linalg.norm(X - c, axis
=1) for c in self.centroids], axis=0)
        return np.sum(distances ** 2)

# Elbow method to find optimal K
def find_optimal_k(X, max_k=10):
    inertias = []

    for k in range(1, max_k + 1):
        kmeans = KMeans(n_clusters=k)
        kmeans.fit(X)
        inertias.append(kmeans.inertia(X))

    # Plot elbow curve
    import matplotlib.pyplot as plt
    plt.plot(range(1, max_k + 1), inertias, 'bo-')
    plt.xlabel('Number of clusters')
    plt.ylabel('Inertia')
    plt.title('Elbow Method')
    plt.show()

# K-Means++ initialization (better than random)
"""
from sklearn.cluster import KMeans

kmeans = KMeans(
    n_clusters=5,
    init='k-means++',  # Smart initialization
    n_init=10,         # Run 10 times with different
     centroids
    max_iter=300,
    random_state=42
)

kmeans.fit(X)
labels = kmeans.labels_
centroids = kmeans.cluster_centers_
"""
```

14. Principal Component Analysis (PCA)

Use when: Dimensionality reduction, visualization, feature extraction
Key concepts: Eigenvalues, eigenvectors, variance explained

```python
import numpy as np

class PCA:
    """Principal Component Analysis"""
    def __init__(self, n_components):
        self.n_components = n_components
        self.components = None
        self.mean = None
        self.explained_variance = None

    def fit(self, X):
        """Fit PCA on training data"""
        # Center the data
        self.mean = np.mean(X, axis=0)
        X_centered = X - self.mean

        # Covariance matrix
        cov = np.cov(X_centered.T)

        # Eigenvalues and eigenvectors
        eigenvalues, eigenvectors = np.linalg.eig(cov)

        # Sort by eigenvalues (descending)
        indices = np.argsort(eigenvalues)[::-1]
        eigenvalues = eigenvalues[indices]
        eigenvectors = eigenvectors[:, indices]

        # Store principal components
        self.components = eigenvectors[:, :self.n_components]

        # Explained variance ratio
        total_var = np.sum(eigenvalues)
        self.explained_variance = eigenvalues[:self.n_components] / total_var

        return self

    def transform(self, X):
        """Project data onto principal components"""
        X_centered = X - self.mean
        return np.dot(X_centered, self.components)

    def fit_transform(self, X):
        """Fit and transform in one step"""
        self.fit(X)
        return self.transform(X)

    def inverse_transform(self, X_transformed):
        """Reconstruct original data from transformed"""
        return np.dot(X_transformed, self.components.T) + self.mean

# Using sklearn PCA
"""
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Fit PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Explained variance
print(f"Explained variance ratio: {pca.explained_variance_ratio_}")
print(f"Total variance explained: {np.sum(pca.explained_variance_ratio_):.2%}")

# Scree plot
plt.plot(range(1, len(pca.explained_variance_ratio_) + 1),
         pca.explained_variance_ratio_, 'bo-')
plt.xlabel('Principal Component')
plt.ylabel('Variance Explained')
plt.show()

# Determine optimal number of components (e.g., 95% variance)
pca_full = PCA()
pca_full.fit(X)
cumsum = np.cumsum(pca_full.explained_variance_ratio_)
n_components_95 = np.argmax(cumsum >= 0.95) + 1
"""
```

ML OPTIMIZATION & TRAINING

15. Optimization Algorithms

Key concepts: SGD, momentum, Adam, learning rate schedules

```python
import numpy as np

# Stochastic Gradient Descent (SGD)
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for param, grad in zip(params, grads):
            param -= self.lr * grad

# SGD with Momentum
class Momentum:
    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.velocity = None

    def update(self, params, grads):
        if self.velocity is None:
            self.velocity = [np.zeros_like(p) for p in params]

        for i, (param, grad) in enumerate(zip(params, grads)):
            self.velocity[i] = self.momentum * self.velocity[i] - self.lr * grad
            param += self.velocity[i]

# RMSprop
class RMSprop:
    def __init__(self, lr=0.001, decay=0.9, epsilon=1e-8):
        self.lr = lr
        self.decay = decay
        self.epsilon = epsilon
        self.cache = None

    def update(self, params, grads):
        if self.cache is None:
            self.cache = [np.zeros_like(p) for p in params]

        for i, (param, grad) in enumerate(zip(params, grads)):
            self.cache[i] = self.decay * self.cache[i] + (1 - self.decay) * grad**2
            param -= self.lr * grad / (np.sqrt(self.cache[i]) + self.epsilon)

# Adam (Adaptive Moment Estimation)
class Adam:
    def __init__(self, lr=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = None  # First moment
        self.v = None  # Second moment
        self.t = 0     # Timestep

    def update(self, params, grads):
        if self.m is None:
            self.m = [np.zeros_like(p) for p in params]
            self.v = [np.zeros_like(p) for p in params]

        self.t += 1

        for i, (param, grad) in enumerate(zip(params, grads)):
            # Update biased first moment
            self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1) * grad

            # Update biased second moment
            self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2) * (grad**2)

            # Bias correction
            m_hat = self.m[i] / (1 - self.beta1**self.t)
            v_hat = self.v[i] / (1 - self.beta2**self.t)

            # Update parameters
            param -= self.lr * m_hat / (np.sqrt(v_hat) + self.epsilon)

# AdamW (Adam with weight decay)
class AdamW:
    def __init__(self, lr=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8, weight_decay=0.01):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.weight_decay = weight_decay
        self.m = None
        self.v = None
        self.t = 0

    def update(self, params, grads):
        if self.m is None:
            self.m = [np.zeros_like(p) for p in params]
            self.v = [np.zeros_like(p) for p in params]
```

```
        self.t += 1

        for i, (param, grad) in enumerate(zip(params, grads)):
            # Weight decay
            param -= self.lr * self.weight_decay * param

            # Adam update (same as above)
            self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1) * grad
            self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2) * (grad**2)

            m_hat = self.m[i] / (1 - self.beta1**self.t)
            v_hat = self.v[i] / (1 - self.beta2**self.t)

            param -= self.lr * m_hat / (np.sqrt(v_hat) + self.epsilon)

# Learning Rate Schedules
class StepLR:
    """Decay LR by gamma every step_size epochs"""
    def __init__(self, optimizer, step_size, gamma=0.1):
        self.optimizer = optimizer
        self.step_size = step_size
        self.gamma = gamma
        self.epoch = 0

    def step(self):
        self.epoch += 1
        if self.epoch % self.step_size == 0:
            self.optimizer.lr *= self.gamma

class CosineAnnealingLR:
    """Cosine annealing schedule"""
    def __init__(self, optimizer, T_max, eta_min=0):
        self.optimizer = optimizer
        self.T_max = T_max
        self.eta_min = eta_min
        self.base_lr = optimizer.lr
        self.epoch = 0

    def step(self):
        self.epoch += 1
        self.optimizer.lr = self.eta_min + (self.base_lr - self.eta_min) * \
                            (1 + np.cos(np.pi * self.epoch / self.T_max)) / 2

# PyTorch example
"""
import torch.optim as optim

# Adam optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001, betas=(0.9, 0.999), weight_decay=0.01)

# Learning rate scheduler
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=100)

# Training loop
for epoch in range(num_epochs):
    for batch in dataloader:
        optimizer.zero_grad()
        loss = compute_loss(batch)
        loss.backward()
        optimizer.step()

    scheduler.step()  # Update learning rate
"""
```

16. Regularization Techniques

Key concepts: L1/L2, dropout, batch normalization, early stopping

```
import torch
import torch.nn as nn

# Dropout
class DropoutExample(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(DropoutExample, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.dropout1 = nn.Dropout(0.5)  # Drop 50% during training
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.dropout2 = nn.Dropout(0.3)
        self.fc3 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.dropout1(x)  # Only active during training
        x = torch.relu(self.fc2(x))
        x = self.dropout2(x)
        x = self.fc3(x)
        return x

# Batch Normalization
class BatchNormExample(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(BatchNormExample, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.bn1 = nn.BatchNorm1d(hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.bn2 = nn.BatchNorm1d(hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = self.bn1(x)  # Normalize before activation
        x = torch.relu(x)
        x = self.fc2(x)
        x = self.bn2(x)
        x = torch.relu(x)
        x = self.fc3(x)
        return x

# Layer Normalization (better for RNNs/Transformers)
class LayerNormExample(nn.Module):
    def __init__(self, d_model):
        super(LayerNormExample, self).__init__()
        self.layer_norm = nn.LayerNorm(d_model)

    def forward(self, x):
        return self.layer_norm(x)

# L1/L2 Regularization
def l1_regularization(model, lambda_l1):
    """Add L1 penalty to loss"""
    l1_loss = 0
    for param in model.parameters():
        l1_loss += torch.sum(torch.abs(param))
    return lambda_l1 * l1_loss

def l2_regularization(model, lambda_l2):
    """Add L2 penalty to loss (weight decay)"""
    l2_loss = 0
    for param in model.parameters():
        l2_loss += torch.sum(param ** 2)
    return lambda_l2 * l2_loss

# Training with regularization
"""
for batch in dataloader:
    optimizer.zero_grad()

    # Forward pass
    outputs = model(inputs)
    loss = criterion(outputs, targets)

    # Add regularization
    loss += l1_regularization(model, lambda_l1=0.01)
    loss += l2_regularization(model, lambda_l2=0.001)

    loss.backward()
    optimizer.step()

# Or use weight_decay in optimizer (L2 only)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=0.01)
"""

# Early Stopping
class EarlyStopping:
    def __init__(self, patience=7, min_delta=0):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss):
        if self.best_loss is None:
            self.best_loss = val_loss
        elif val_loss > self.best_loss - self.min_delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_loss = val_loss
            self.counter = 0

# Usage
"""
early_stopping = EarlyStopping(patience=10)

for epoch in range(num_epochs):
    train_loss = train_one_epoch()
    val_loss = validate()

    early_stopping(val_loss)
    if early_stopping.early_stop:
        print("Early stopping")
        break
"""

# Gradient Clipping (prevent exploding gradients in RNNs)
"""
# Clip by norm
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

```
# Clip by value
torch.nn.utils.clip_grad_value_(model.parameters(),
    clip_value=0.5)
"""

# Data Augmentation (regularization via more training
    data)
"""
from torchvision import transforms

train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.RandomCrop(32, padding=4),
    transforms.ColorJitter(brightness=0.2, contrast
    =0.2),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
"""
```

EVALUATION & METRICS

17. Classification Metrics

```
import numpy as np

def accuracy(y_true, y_pred):
    return np.sum(y_true == y_pred) / len(y_true)

def precision(y_true, y_pred, pos_label=1):
    """TP / (TP + FP)"""
    tp = np.sum((y_true == pos_label) & (y_pred ==
    pos_label))
    fp = np.sum((y_true != pos_label) & (y_pred ==
    pos_label))
    return tp / (tp + fp) if (tp + fp) > 0 else 0

def recall(y_true, y_pred, pos_label=1):
    """TP / (TP + FN)"""
    tp = np.sum((y_true == pos_label) & (y_pred ==
    pos_label))
    fn = np.sum((y_true == pos_label) & (y_pred !=
    pos_label))
    return tp / (tp + fn) if (tp + fn) > 0 else 0

def f1_score(y_true, y_pred, pos_label=1):
    """Harmonic mean of precision and recall"""
    p = precision(y_true, y_pred, pos_label)
    r = recall(y_true, y_pred, pos_label)
    return 2 * p * r / (p + r) if (p + r) > 0 else 0

def confusion_matrix(y_true, y_pred, num_classes):
    """Confusion matrix"""
    cm = np.zeros((num_classes, num_classes), dtype=
    int)
    for true, pred in zip(y_true, y_pred):
        cm[true][pred] += 1
    return cm

def roc_auc(y_true, y_scores):
    """ROC AUC score"""
    # Sort by scores
    indices = np.argsort(y_scores)[::-1]
    y_true_sorted = y_true[indices]

    # Calculate TPR and FPR at different thresholds
    tpr = np.cumsum(y_true_sorted) / np.sum(
    y_true_sorted)
    fpr = np.cumsum(1 - y_true_sorted) / np.sum(1 -
    y_true_sorted)

    # Calculate AUC using trapezoidal rule
    auc = np.trapz(tpr, fpr)
    return auc

# Using sklearn
"""
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score,
    confusion_matrix, classification_report,
    roc_auc_score, roc_curve
)

# Basic metrics
acc = accuracy_score(y_true, y_pred)
prec = precision_score(y_true, y_pred, average='macro
    ')  # 'micro', 'weighted', 'binary'
rec = recall_score(y_true, y_pred, average='macro')
f1 = f1_score(y_true, y_pred, average='macro')

# Confusion matrix
cm = confusion_matrix(y_true, y_pred)

# Full report
print(classification_report(y_true, y_pred))

# ROC curve
fpr, tpr, thresholds = roc_curve(y_true, y_scores)
auc = roc_auc_score(y_true, y_scores)

import matplotlib.pyplot as plt
plt.plot(fpr, tpr, label=f'AUC = {auc:.2f}')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.show()
"""
```

18. Regression Metrics

```
import numpy as np

def mse(y_true, y_pred):
    """Mean Squared Error"""
    return np.mean((y_true - y_pred) ** 2)

def rmse(y_true, y_pred):
    """Root Mean Squared Error"""
    return np.sqrt(mse(y_true, y_pred))

def mae(y_true, y_pred):
    """Mean Absolute Error"""
    return np.mean(np.abs(y_true - y_pred))

def r2_score(y_true, y_pred):
    """R-squared (coefficient of determination)"""
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    return 1 - (ss_res / ss_tot)

def mape(y_true, y_pred):
    """Mean Absolute Percentage Error"""
    return np.mean(np.abs((y_true - y_pred) / y_true))
     * 100

# Using sklearn
"""
from sklearn.metrics import mean_squared_error,
    mean_absolute_error, r2_score

mse = mean_squared_error(y_true, y_pred)
rmse = mean_squared_error(y_true, y_pred, squared=
    False)
mae = mean_absolute_error(y_true, y_pred)
r2 = r2_score(y_true, y_pred)
"""
```

STAFF/PRINCIPAL: CAUSAL INFERENCE

19. Uplift Modeling & Causal Inference

Use when: Treatment effects, marketing campaigns, policy evaluation
Key concepts: Causation vs correlation, treatment effect, persuadables

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier

class UpliftTwoModel:
    """Two-Model Approach (T-Learner) for uplift
    modeling"""
    def __init__(self):
        self.model_treatment = RandomForestClassifier(
    n_estimators=100, random_state=42)
        self.model_control = RandomForestClassifier(
    n_estimators=100, random_state=42)

    def fit(self, X, treatment, y):
        """
        X: Features (n_samples, n_features)
        treatment: Treatment indicator (0 or 1)
        y: Outcome (0 or 1 for binary, continuous for
    regression)
        """
        # Split data by treatment
        treatment_mask = treatment == 1
        control_mask = treatment == 0

        X_treatment = X[treatment_mask]
        y_treatment = y[treatment_mask]
        X_control = X[control_mask]
        y_control = y[control_mask]

        # Train separate models
        self.model_treatment.fit(X_treatment,
    y_treatment)
        self.model_control.fit(X_control, y_control)

        return self

    def predict_uplift(self, X):
        """
        Predict uplift: E[Y|X,T=1] - E[Y|X,T=0]
        Returns: uplift scores for each sample
        """
        p_treatment = self.model_treatment.
    predict_proba(X)[:, 1]
        p_control = self.model_control.predict_proba(X
    )[:, 1]
```

```
        uplift = p_treatment - p_control
        return uplift

    def segment_users(self, X, threshold=0):
        """
        Segment users by uplift:
        - Persuadables: positive uplift
        - Lost causes: negative uplift
        - Sure things/Do-not-disturbs: uplift near
    zero
        """
        uplift = self.predict_uplift(X)

        segments = np.zeros(len(uplift), dtype=int)
        segments[uplift > threshold] = 1  #
    Persuadables
        segments[uplift < -threshold] = -1  # Lost
    causes

        return segments, uplift

class UpliftSingleModel:
    """Single-Model Approach (S-Learner) for uplift
    modeling"""
    def __init__(self):
        self.model = RandomForestClassifier(
    n_estimators=100, random_state=42)

    def fit(self, X, treatment, y):
        """Train single model with treatment as a
    feature"""
        # Add treatment as feature
        X_with_treatment = np.column_stack([X,
    treatment])
        self.model.fit(X_with_treatment, y)
        return self

    def predict_uplift(self, X):
        """Predict uplift by comparing T=1 vs T=0
    predictions"""
        # Predict with treatment
        X_treatment = np.column_stack([X, np.ones(len(
    X))])
        p_treatment = self.model.predict_proba(
    X_treatment)[:, 1]

        # Predict without treatment
        X_control = np.column_stack([X, np.zeros(len(X
    ))])
        p_control = self.model.predict_proba(X_control
    )[:, 1]

        uplift = p_treatment - p_control
        return uplift

# Uplift Evaluation Metrics
def uplift_curve(y_true, treatment, uplift_scores,
    n_bins=10):
    """
    Calculate uplift curve (cumulative gain)

    Returns:
        pcts: Percentage of population targeted
        uplifts: Cumulative uplift at each percentage
    """
    # Sort by uplift score (descending)
    indices = np.argsort(uplift_scores)[::-1]
    y_sorted = y_true[indices]
    t_sorted = treatment[indices]

    n = len(y_sorted)
    bin_size = n // n_bins

    pcts = []
    uplifts = []

    for i in range(1, n_bins + 1):
        # Calculate uplift for top i bins
        idx = i * bin_size

        y_t = y_sorted[:idx][t_sorted[:idx] == 1]
        y_c = y_sorted[:idx][t_sorted[:idx] == 0]

        if len(y_t) > 0 and len(y_c) > 0:
            uplift = np.mean(y_t) - np.mean(y_c)
        else:
            uplift = 0

        pcts.append(i / n_bins * 100)
        uplifts.append(uplift)

    return pcts, uplifts

def qini_coefficient(y_true, treatment, uplift_scores)
    :
    """
    Qini coefficient: Area between uplift curve and
     random curve
    Higher is better (max = 1.0)
    """
    # Sort by uplift score
    indices = np.argsort(uplift_scores)[::-1]
    y_sorted = y_true[indices]
    t_sorted = treatment[indices]

    n = len(y_sorted)
    cumulative_qini = 0

    n_t_cumsum = 0
    n_c_cumsum = 0
    y_t_cumsum = 0
    y_c_cumsum = 0

    for i in range(n):
        if t_sorted[i] == 1:
            n_t_cumsum += 1
            y_t_cumsum += y_sorted[i]
        else:
            n_c_cumsum += 1
            y_c_cumsum += y_sorted[i]

        # Qini at this point
        if n_t_cumsum > 0 and n_c_cumsum > 0:
            qini = y_t_cumsum - (n_t_cumsum /
     n_c_cumsum) * y_c_cumsum
            cumulative_qini += qini

    # Normalize by maximum possible Qini
    max_qini = n * np.mean(y_sorted[t_sorted == 1]) if
      np.sum(t_sorted) > 0 else 1

    return cumulative_qini / max_qini if max_qini != 0
      else 0

# Visualization
def plot_uplift_curve(y_true, treatment, uplift_scores
    ):
    """Plot uplift curve to visualize model
     performance"""
    import matplotlib.pyplot as plt

    pcts, uplifts = uplift_curve(y_true, treatment,
     uplift_scores, n_bins=20)

    plt.figure(figsize=(10, 6))
    plt.plot(pcts, uplifts, marker='o', label='Uplift
     Model')
    plt.axhline(y=0, color='r', linestyle='--', label=
     'Random (No Uplift)')
    plt.xlabel('Percentage of Population Targeted (%)'
     )
    plt.ylabel('Incremental Uplift')
    plt.title('Uplift Curve')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

# Example Usage
"""
# Simulate data
np.random.seed(42)
n = 10000

# Features
X = np.random.randn(n, 5)

# Treatment (random assignment)
treatment = np.random.binomial(1, 0.5, n)

# True uplift effect (depends on features)
true_uplift = 0.1 * X[:, 0] + 0.05 * X[:, 1]
base_prob = 1 / (1 + np.exp(-X[:, 2]))

# Outcome
y_control = np.random.binomial(1, base_prob)
y_treatment = np.random.binomial(1, base_prob +
    true_uplift)
y = np.where(treatment == 1, y_treatment, y_control)

# Train uplift model
model = UpliftTwoModel()
model.fit(X, treatment, y)

# Predict uplift
X_test = np.random.randn(1000, 5)
uplift_scores = model.predict_uplift(X_test)

# Segment users
segments, uplift = model.segment_users(X_test,
    threshold=0.01)

print(f"Persuadables: {np.sum(segments == 1)}")
print(f"Neutral: {np.sum(segments == 0)}")
print(f"Lost causes: {np.sum(segments == -1)}")

# Evaluate
qini = qini_coefficient(y, treatment, model.
    predict_uplift(X))
print(f"Qini coefficient: {qini:.4f}")

# Plot
plot_uplift_curve(y, treatment, model.predict_uplift(X
    ))
"""

# Business Impact Calculation
def calculate_roi(uplift_scores, treatment_cost,
    conversion_value,
                  percentile=0.2):
    """
    Calculate ROI of targeting top percentile by
     uplift
```

```
    Args:
        uplift_scores: Predicted uplift for each user
        treatment_cost: Cost per treatment (e.g., \$10
      coupon)
        conversion_value: Value per conversion (e.g.,
     \$100 purchase)
        percentile: Top X% to target (default 20%)

    Returns:
        roi: Return on investment
    """
    # Target top percentile
    threshold = np.percentile(uplift_scores, 100 * (1
     - percentile))
    targeted = uplift_scores >= threshold

    n_targeted = np.sum(targeted)

    # Expected incremental conversions
    incremental_conversions = np.sum(uplift_scores[
     targeted])

    # Calculate ROI
    revenue = incremental_conversions *
     conversion_value
    cost = n_targeted * treatment_cost
    roi = (revenue - cost) / cost if cost > 0 else 0

    return roi, n_targeted, incremental_conversions

"""
# Example: Marketing campaign ROI
roi, n_users, conversions = calculate_roi(
    uplift_scores=uplift_scores,
    treatment_cost=10,      # \$10 coupon
    conversion_value=100,   # \$100 purchase value
    percentile=0.2          # Target top 20%
)

print(f"ROI: {roi:.2%}")
print(f"Users targeted: {n_users:,}")
print(f"Incremental conversions: {conversions:.1f}")
"""
```

Key Interview Points:

- **Correlation vs Causation:** $P(Y|X)$ vs $P(Y|do(X))$ - observational vs interventional
- **Two approaches:** T-Learner (two models) vs S-Learner (single model with treatment feature)
- **User segments:** Persuadables (positive uplift), Sure things (convert anyway), Lost causes (negative uplift), Do-not-disturbs (won't convert)
- **Evaluation:** Uplift curves, Qini coefficient (not just AUC!)
- **Business impact:** ROI calculation, incremental value over random targeting
- **When to use:** Marketing campaigns, medical treatments, policy interventions

INTERVIEW PATTERN RECOGNITION

Problem to Algorithm Mapping

Computer Vision:

- Image classification → CNN (ResNet, EfficientNet)
- Object detection → R-CNN, YOLO, SSD
- Semantic segmentation → U-Net, FCN
- Face recognition → Siamese networks, ArcFace
- Style transfer → GAN, Neural Style Transfer
- Image generation → VAE, GAN, Diffusion models

Natural Language Processing:

- Text classification → BERT, RoBERTa, DistilBERT
- Sequence labeling (NER, POS) → BiLSTM-CRF, BERT
- Machine translation → Transformer, Seq2Seq with attention
- Question answering → BERT, GPT, T5
- Text generation → GPT, T5, BART
- Sentiment analysis → LSTM, BERT

Tabular Data:

- Classification/Regression → XGBoost, LightGBM, CatBoost
- High interpretability needed → Decision Trees, Linear models
- Small dataset → Random Forest, SVM
- Large dataset → Neural networks, gradient boosting

Time Series:

- Forecasting → LSTM, GRU, Prophet, ARIMA
- Anomaly detection → Autoencoders, Isolation Forest
- Sequence classification → 1D CNN, LSTM, Transformer

Recommendation Systems:

- Collaborative filtering → Matrix factorization, NCF
- Content-based → Cosine similarity, embeddings
- Hybrid → Deep learning with side information

Model Selection Checklist

Data considerations:

- Structured/tabular → XGBoost, Random Forest
- Images → CNN, Vision Transformer
- Text → Transformer, BERT, GPT
- Time series/sequential → LSTM, GRU, Transformer
- Small dataset (¡ 10k) → Classical ML, transfer learning
- Large dataset (¿ 1M) → Deep learning
- High-dimensional → PCA, autoencoders

Performance requirements:

- Low latency → Smaller models, model compression, caching
- High throughput → Batch processing, model parallelism
- Resource-constrained → MobileNet, DistilBERT, quantization
- Accuracy critical → Ensemble methods, larger models

Interpretability:

- High interpretability → Linear models, decision trees, SHAP
- Black box acceptable → Deep learning, ensembles