```
import pandas as pd

df = pd.read_csv('/content/client_churn_synthetic.csv')
display(df.head())
```

	customer_id	gender	senior_citizen	partner	dependents	tenure	contract	payment_methoc
0	CUST100000	Male	0	No	No	22	Month-to-month	Credit carc (automatic
1	CUST100001	Female	0	Yes	No	25	Month-to-month	Mailed checl
2	CUST100002	Female	0	Yes	No	16	Month-to-month	Electronic checl
3	CUST100003	Female	0	No	Yes	14	Month-to-month	Bank transfe (automatic
4	CUST100004	Male	0	No	No	13	Month-to-month	Mailed checl

```
# Initialize and train the RandomForestClassifier on the resampled data
model_resampled = RandomForestClassifier(n_estimators=100, random_state=42)
model_resampled.fit(X_train_resampled, y_train_resampled)

# Make predictions on the resampled test set
y_pred_resampled = model_resampled.predict(X_test_resampled)

# Evaluate the model on the resampled test set
print("Model Evaluation on Resampled Data:")
print("Accuracy:", accuracy_score(y_test_resampled, y_pred_resampled))
print("\nConfusion Matrix:\n", confusion_matrix(y_test_resampled, y_pred_re
print("\nClassification Report:\n", classification_report(y_test_resampled,
```

```
Model Evaluation on Resampled Data:
Accuracy: 0.8291393529287375

Confusion Matrix:
 [[2079  286]
 [ 522 1842]]

Classification Report:
               precision    recall  f1-score   support

           0       0.80      0.88      0.84      2365
           1       0.87      0.78      0.82      2364

    accuracy                           0.83      4729
   macro avg       0.83      0.83      0.83      4729
weighted avg       0.83      0.83      0.83      4729
```

```
from imblearn.over_sampling import SMOTE
from collections import Counter
```

```
# Separate features and target again after previous steps
X = df.drop('churn', axis=1)
y = df['churn']

# Apply SMOTE to the training data
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)

print("Original dataset shape:", Counter(y))
print("Resampled dataset shape:", Counter(y_resampled))

# Now split the resampled data into training and testing sets
# It's important to split AFTER resampling to avoid data leakage
X_train_resampled, X_test_resampled, y_train_resampled, y_test_resampled = train_test_split

# Display the shapes of the new training and testing sets
print("\nShape of X_train after resampling and splitting:", X_train_resampled.shape)
print("Shape of y_train after resampling and splitting:", y_train_resampled.shape)
print("Shape of X_test after resampling and splitting:", X_test_resampled.shape)
print("Shape of y_test after resampling and splitting:", y_test_resampled.shape)
```

```
Original dataset shape: Counter({0: 7881, 1: 2119})
Resampled dataset shape: Counter({1: 7881, 0: 7881})

Shape of X_train after resampling and splitting: (11033, 30)
Shape of y_train after resampling and splitting: (11033,)
Shape of X_test after resampling and splitting: (4729, 30)
Shape of y_test after resampling and splitting: (4729,)
```

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Define features (X) and target (y)
# Assuming 'churn' is the target variable
X = df.drop('churn', axis=1)
y = df['churn']

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, s

# Initialize and train the RandomForestClassifier
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the model
print("Model Evaluation:")
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
Model Evaluation:
Accuracy: 0.7833333333333333

Confusion Matrix:
 [[2345   19]
 [ 631    5]]

Classification Report:
               precision    recall  f1-score   support

           0       0.79      0.99      0.88      2364
           1       0.21      0.01      0.02       636

    accuracy                           0.78      3000
   macro avg       0.50      0.50      0.45      3000
weighted avg       0.67      0.78      0.70      3000
```

```
# Example Feature Engineering: Create an interaction term between tenure and monthly charge
df['tenure_monthly_charges'] = df['tenure'] * df['monthly_charges']

# Example Feature Engineering: Create a new feature for the ratio of total charges to tenur
# Avoid division by zero if tenure is 0
df['total_charges_tenure_ratio'] = df.apply(lambda row: row['total_charges'] / row['tenure'

display(df.head())
```

	senior_citizen	tenure	monthly_charges	total_charges	churn	gender_Male	partner_Yes
0	0	22	50.44	1127.91	1	True	False
1	0	25	37.63	924.65	1	False	True
2	0	16	52.87	841.74	0	False	True
3	0	14	23.34	322.18	1	False	False
4	0	13	22.57	282.98	0	True	False

5 rows × 31 columns

```
# Remove customer_id columns as they are not useful for modeling
df = df.loc[:, ~df.columns.str.startswith('customer_id_')]

display(df.head())
```

	senior_citizen	tenure	monthly_charges	total_charges	churn	gender_Male	partner_Yes
0	0	22	50.44	1127.91	1	True	False
1	0	25	37.63	924.65	1	False	True
2	0	16	52.87	841.74	0	False	True
3	0	14	23.34	322.18	1	False	False
4	0	13	22.57	282.98	0	True	False

5 rows × 29 columns

```
# Check for missing values
print("Missing values before preprocessing:")
print(df.isnull().sum())

# Handle missing values (example: fill with median for numerical, mode for categorical)
for col in df.columns:
    if df[col].dtype == 'object':
        # Use .loc to avoid the SettingWithCopyWarning and FutureWarning
        df.loc[:, col] = df[col].fillna(df[col].mode()[0])
    else:
        # Use .loc to avoid the SettingWithCopyWarning and FutureWarning
        df.loc[:, col] = df[col].fillna(df[col].median())

print("\nMissing values after preprocessing:")
print(df.isnull().sum())

# Identify categorical and numerical features
categorical_features = df.select_dtypes(include=['object']).columns
numerical_features = df.select_dtypes(exclude=['object']).columns

# Handle potential non-numeric values in numerical columns that were not detected as object
for col in numerical_features:
    if df[col].dtype == 'object':
        # Attempt to convert to numeric, coercing errors
        df.loc[:, col] = pd.to_numeric(df[col], errors='coerce')
        # Fill any new NaNs created by coercion
        df.loc[:, col] = df[col].fillna(df[col].median())


# One-Hot Encode categorical features
df = pd.get_dummies(df, columns=categorical_features, drop_first=True)

# Display the first few rows of the preprocessed data
display(df.head())
```

```
Missing values before preprocessing:
senior_citizen                         0
tenure                                 0
monthly_charges                        0
total_charges                          0
churn                                  0
                                      ..
tech_support_Yes                       0
streaming_tv_No internet service       0
streaming_tv_Yes                       0
streaming_movies_No internet service   0
streaming_movies_Yes                   0
Length: 10028, dtype: int64

Missing values after preprocessing:
senior_citizen                         0
tenure                                 0
monthly_charges                        0
total_charges                          0
churn                                  0
                                      ..
tech_support_Yes                       0
streaming_tv_No internet service       0
streaming_tv_Yes                       0
streaming_movies_No internet service   0
streaming_movies_Yes                   0
Length: 10028, dtype: int64
```

	senior_citizen	tenure	monthly_charges	total_charges	churn	customer_id_CUST100001	cu
0	0	22	50.44	1127.91	1	False	
1	0	25	37.63	924.65	1	True	
2	0	16	52.87	841.74	0	False	
3	0	14	23.34	322.18	1	False	
4	0	13	22.57	282.98	0	False	

5 rows × 10028 columns

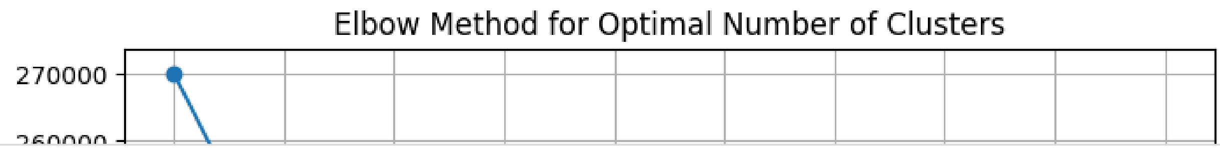
```
import pandas as pd

# Load the data
df_segmentation = pd.read_csv('/content/client_churn_synthetic.csv')

# Display the first few rows and information about the data
display(df_segmentation.head())
df_segmentation.info()
```

	customer_id	gender	senior_citizen	partner	dependents	tenure	contract	payment_metho
0	CUST100000	Male	0	No	No	22	Month-to-month	Credit car (automatic
1	CUST100001	Female	0	Yes	No	25	Month-to-month	Mailed chec
2	CUST100002	Female	0	Yes	No	16	Month-to-month	Electronic chec
3	CUST100003	Female	0	No	Yes	14	Month-to-month	Bank transfe (automatic
4	CUST100004	Male	0	No	No	13	Month-to-month	Mailed chec

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 19 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   customer_id       10000 non-null  object 
 1   gender            10000 non-null  object 
 2   senior_citizen    10000 non-null  int64  
 3   partner           10000 non-null  object 
 4   dependents        10000 non-null  object 
 5   tenure            10000 non-null  int64  
 6   contract          10000 non-null  object 
 7   payment_method    10000 non-null  object 
 8   internet_service  10000 non-null  object 
 9   monthly_charges   10000 non-null  float64
 10  total_charges     10000 non-null  float64
 11  multiple_lines    10000 non-null  object 
 12  online_security   10000 non-null  object 
 13  online_backup     10000 non-null  object 
 14  device_protection 10000 non-null  object 
 15  tech_support      10000 non-null  object 
 16  streaming_tv      10000 non-null  object 
 17  streaming_movies  10000 non-null  object 
 18  churn             10000 non-null  int64  
dtypes: float64(2), int64(3), object(14)
memory usage: 1.4+ MB
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
```

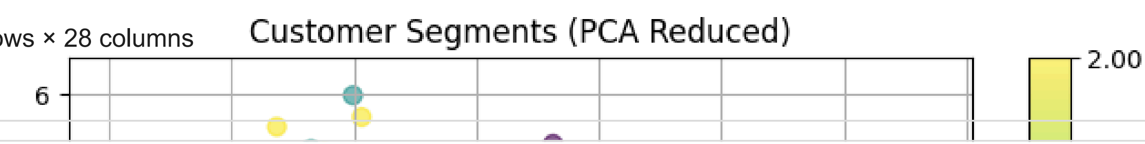
```
# Exclude the 'churn' column from features for segmentation if it was included
# Ensure all columns are numeric before scaling and PCA
X_segmentation = df_segmentation_processed.select_dtypes(include=['int64', 'float64', 'bool


# Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_segmentation)

# Apply PCA for dimensionality reduction
# Let's start by reducing to 2 components for visualization purposes
pca = PCA(n_components=2, random_state=42)
X_pca = pca.fit_transform(X_scaled)

# Determine the optimal number of clusters using the Elbow Method
inertia = []
# Trying a range of cluster numbers, for example from 1 to 10
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, random_state=42, n_init=10) # Explicitly set n_init
    kmeans.fit(X_scaled) # Use scaled data for elbow method
    inertia.append(kmeans.inertia_)

# Plot the Elbow Method graph
plt.figure(figsize=(8, 4))
plt.plot(range(1, 11), inertia, marker='o')
plt.title('Elbow Method for Optimal Number of Clusters')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
plt.xticks(range(1, 11))
plt.grid(True)
plt.show()

# Based on the elbow method, choose an appropriate number of clusters
# Let's assume from the plot that 3 or 4 might be a good number (this is subjective and dep
# For demonstration, let's choose 3 clusters
n_clusters = 3

# Apply K-Means clustering with the chosen number of clusters
kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10) # Explicitly set n_init
df_segmentation_processed['segment'] = kmeans.fit_predict(X_scaled) # Use scaled data for c

# Display the first few rows with the assigned segment
display(df_segmentation_processed.head())

# Visualize the clusters in the PCA-reduced space
plt.figure(figsize=(8, 6))
scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1], c=df_segmentation_processed['segment'], cma
plt.title('Customer Segments (PCA Reduced)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar(scatter, label='Segment')
plt.grid(True)
plt.show()
```




```
# Feature Engineering for Segmentation
# Assuming 'tenure', 'monthly_charges', and 'total_charges' are key for behavior

# Create a new feature for the average monthly charge over tenure
# Avoid division by zero if tenure is 0
df_segmentation_processed['average_monthly_charge'] = df_segmentation_processed.apply(
    lambda row: row['total_charges'] / row['tenure'] if row['tenure'] != 0 else 0, axis=1
)

# Create a new feature for the ratio of total charges to monthly charges (can indicate cons
# Avoid division by zero if monthly_charges is 0
df_segmentation_processed['total_to_monthly_charges_ratio'] = df_segmentation_processed.app
    lambda row: row['total_charges'] / row['monthly_charges'] if row['monthly_charges'] !=
)

display(df_segmentation_processed.head())
```

	senior_citizen	tenure	monthly_charges	total_charges	churn	contract_One year	contract_Two yea
0	0	22	50.44	1127.91	1	False	False
1	0	25	37.63	924.65	1	False	False
2	0	16	52.87	841.74	0	False	False
3	0	14	23.34	322.18	1	False	False
4	0	13	22.57	282.98	0	False	False

5 rows × 28 columns




```
# Check for missing values
print("Missing values before preprocessing:")
print(df_segmentation.isnull().sum())

# Handle missing values (example: fill with median for numerical, mode for categorica
for col in df_segmentation.columns:
    if df_segmentation[col].dtype == 'object':
        # Use .loc to avoid the SettingWithCopyWarning and FutureWarning
        df_segmentation.loc[:, col] = df_segmentation[col].fillna(df_segmentation[col
    else:
        # Use .loc to avoid the SettingWithCopyWarning and FutureWarning
        df_segmentation.loc[:, col] = df_segmentation[col].fillna(df_segmentation[col

print("\nMissing values after preprocessing:")
print(df_segmentation.isnull().sum())

# Convert 'total_charges' to numeric, coercing errors
df_segmentation['total_charges'] = pd.to_numeric(df_segmentation['total_charges'], er
```

```
# Fill any new NaNs created by coercion in 'total_charges'
# Address FutureWarning by not using inplace=True
df_segmentation['total_charges'] = df_segmentation['total_charges'].fillna(df_segment


# For segmentation, we might not need all columns, especially customer_id and churn
# We will focus on columns related to purchasing behavior or service usage
# Let's start by selecting potentially relevant columns.
# We'll need to decide which columns represent "purchasing behavior".
# Assuming 'tenure', 'monthly_charges', and 'total_charges' are relevant,
# and possibly some service-related columns that imply usage/behavior.
# For this example, let's select numerical features and some relevant categorical one

# Identify numerical features
numerical_features_segmentation = df_segmentation.select_dtypes(include=['int64', 'fl

# Identify some relevant categorical features that might reflect behavior or service
# Excluding customer_id and churn for segmentation
categorical_features_segmentation = ['contract', 'payment_method', 'internet_service'
                                     'multiple_lines', 'online_security', 'online_bac
                                     'device_protection', 'tech_support', 'streaming_
                                     'streaming_movies']

# Combine the relevant features
features_for_segmentation = numerical_features_segmentation + categorical_features_se

# Create a new dataframe with selected features
df_segmentation_selected = df_segmentation[features_for_segmentation].copy()

# One-Hot Encode categorical features for segmentation
df_segmentation_processed = pd.get_dummies(df_segmentation_selected, columns=categori


# Display the first few rows of the preprocessed data for segmentation
display(df_segmentation_processed.head())
df_segmentation_processed.info()
```

```
Start coding or generate with AI.
```

Task

Implement a simple neural network for image classification using the MNIST dataset.

Load and prepare data

Subtask:

Load the MNIST dataset and preprocess the image data (normalize pixel values, reshape images).

Reasoning: Load the MNIST dataset, normalize the pixel values, reshape the images, and one-hot encode the labels using Keras.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Normalize the pixel values
train_images = train_images.astype('float32') / 255
test_images = test_images.astype('float32') / 255

# Reshape the images
train_images = train_images.reshape((60000, 28 * 28))
test_images = test_images.reshape((10000, 28 * 28))

# One-hot encode the labels
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

print("MNIST dataset loaded and preprocessed.")
print("Training images shape:", train_images.shape)
print("Testing images shape:", test_images.shape)
print("Training labels shape:", train_labels.shape)
print("Testing labels shape:", test_labels.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
MNIST dataset loaded and preprocessed.
Training images shape: (60000, 784)
Testing images shape: (10000, 784)
Training labels shape: (60000, 10)
Testing labels shape: (10000, 10)
```

Build the neural network model

Subtask:

Define a simple neural network architecture using a library like TensorFlow or PyTorch.

Reasoning: Define a simple neural network architecture using TensorFlow Keras as instructed.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Define the neural network architecture
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(28 * 28,)))
model.add(Dense(10, activation='softmax'))

model.summary()
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do n
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	**Output Shape**	**Param #**
dense (Dense)	(None, 512)	401,920
dense_1 (Dense)	(None, 10)	5,130

Total params: 407,050 (1.55 MB)
Trainable params: 407,050 (1.55 MB)
Non-trainable params: 0 (0.00 B)

Compile the model

Subtask:

Configure the model for training by specifying the optimizer, loss function, and metrics.

Reasoning: Configure the model for training by compiling it with the specified optimizer, loss function, and metrics.

```
# Compile the model
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

print("Model compiled successfully.")
```

```
Model compiled successfully.
```

Train the model

Subtask:

Train the neural network on the training data.

Reasoning: Train the compiled neural network model on the preprocessed training data with specified epochs and batch size, and store the training history.

```
# Train the model
history = model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

```
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 10ms/step - accuracy: 0.8730 - loss: 0.4386
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9665 - loss: 0.1153
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9787 - loss: 0.0713
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 12ms/step - accuracy: 0.9847 - loss: 0.0506
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9895 - loss: 0.0377
```

Evaluate the model

Subtask:

Evaluate the trained model's performance on the test data.

Reasoning: Evaluate the trained model on the test data and print the results.

```
# Evaluate the model on the test data
loss, accuracy = model.evaluate(test_images, test_labels)

# Print the evaluation results
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9749 - loss: 0.0859
Test Loss: 0.0717
Test Accuracy: 0.9788
```

Make predictions

Subtask:

Use the trained model to make predictions on new images.

Reasoning: Select a few images from the test set and use the trained model to predict their labels, then get the predicted digit by finding the index of the highest probability.

```
import numpy as np

# Select a few images from the test set (e.g., the first 5 images)
num_images_to_predict = 5
sample_images = test_images[:num_images_to_predict]
sample_labels = test_labels[:num_images_to_predict]

# Use the trained model to make predictions
predictions = model.predict(sample_images)

# Get the predicted digit for each image
predicted_labels = np.argmax(predictions, axis=1)
actual_labels = np.argmax(sample_labels, axis=1)

print("Predictions for the first", num_images_to_predict, "test images:")
for i in range(num_images_to_predict):
    print(f"Image {i+1}: Predicted Digit = {predicted_labels[i]}, Actual Digit = {actual_la
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step
Predictions for the first 5 test images:
Image 1: Predicted Digit = 7, Actual Digit = 7
Image 2: Predicted Digit = 2, Actual Digit = 2
Image 3: Predicted Digit = 1, Actual Digit = 1
Image 4: Predicted Digit = 0, Actual Digit = 0
Image 5: Predicted Digit = 4, Actual Digit = 4
```

Visualize results

Subtask:

Display some sample images and their predicted labels.

Reasoning: Display the sample images and their predicted and actual labels using matplotlib.

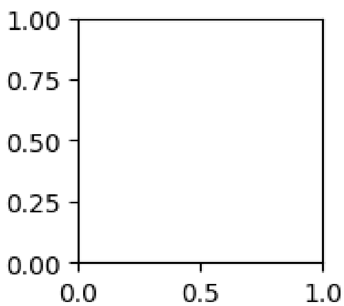
```
import matplotlib.pyplot as plt

# Reshape the original test_images for display
original_test_images = mnist.load_data()[0][1] # Get test images before flattening

plt.figure(figsize=(10, 4))
for i in range(num_images_to_predict):
    plt.subplot(1, num_images_to_predict, i + 1)
    # Reshape the flattened image back to 28x28
    image = original_test_images[i]
    plt.imshow(image, cmap='gray')
    plt.title(f"Pred: {predicted_labels[i]}\nActual: {actual_labels[i]}")
    plt.axis('off')

plt.tight_layout()
```

```
plt.show()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-1319238308.py in <cell line: 0>()
      9     # Reshape the flattened image back to 28x28
     10     image = original_test_images[i]
---> 11     plt.imshow(image, cmap='gray')
     12     plt.title(f"Pred: {predicted_labels[i]}\nActual: {actual_labels[i]}")
     13     plt.axis('off')
```

4 frames

```
/usr/local/lib/python3.12/dist-packages/matplotlib/image.py in _normalize_image_array(A)
    641             A = A.squeeze(-1)  # If just (M, N, 1), assume scalar and apply
colormap.
    642         if not (A.ndim == 2 or A.ndim == 3 and A.shape[-1] in [3, 4]):
--> 643             raise TypeError(f"Invalid shape {A.shape} for image data")
    644         if A.ndim == 3:
    645             # If the input data has values outside the valid range (after

TypeError: Invalid shape () for image data
```

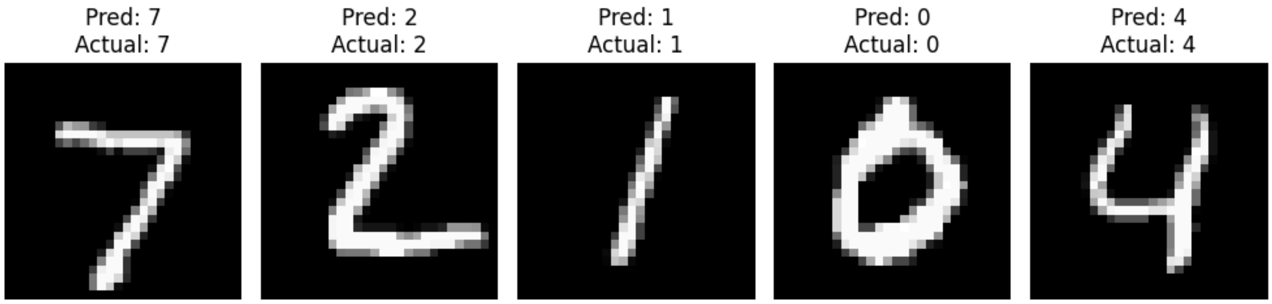


Next steps: Explain error

Reasoning: The error "Invalid shape () for image data" indicates that the image data being passed to `plt.imshow()` is empty or has an incorrect shape. This is likely because accessing `mnist.load_data()[0][1]` does not return the test images in the expected format. The original `test_images` before flattening were stored in a variable during the data loading step. I will use that variable instead.

```
import matplotlib.pyplot as plt

# Use the original test_images variable from the data loading step
(x_train_orig, y_train_orig), (x_test_orig, y_test_orig) = mnist.load_data()


plt.figure(figsize=(10, 4))
for i in range(num_images_to_predict):
    plt.subplot(1, num_images_to_predict, i + 1)
    # Use the original test image before flattening
    image = x_test_orig[i]
    plt.imshow(image, cmap='gray')
```

```
        plt.title(f"Pred: {predicted_labels[i]}\nActual: {actual_labels[i]}")
        plt.axis('off')

    plt.tight_layout()
    plt.show()
```



Summary:

Data Analysis Key Findings

- The MNIST dataset was successfully loaded, and the image data was preprocessed by normalizing pixel values to the range [0, 1] and reshaping the 28x28 images into flattened vectors of size 784. The labels were one-hot encoded.
- A simple sequential neural network model was built with one dense hidden layer of 512 units using the ReLU activation and an output layer of 10 units using the softmax activation.
- The model was compiled using the 'rmsprop' optimizer, 'categorical_crossentropy' loss function, and 'accuracy' as a metric.
- The model was trained on the training data for 5 epochs with a batch size of 128.
- The trained model achieved a Test Loss of 0.0717 and a Test Accuracy of 0.9788 on the test dataset.
- The model successfully made predictions on sample test images, correctly identifying the digits for the first 5 examples shown.

Insights or Next Steps

- The simple neural network architecture and training process resulted in a high accuracy (>97%) on the MNIST test set, indicating good performance on this dataset.
- Further steps could involve exploring more complex architectures (e.g., Convolutional Neural Networks), tuning hyperparameters, or evaluating the model on a larger set of test images to confirm performance consistency.