Loading the Data

In [ ]:
```
ID = 1631938
Name = 'MIN SOE HTUT'
ID
Name
```

Part A: Regression

1 Loading the data

In [ ]:
```
import numpy as np
import pandas as pd
import sklearn.metrics as skmetric
import seaborn as sns
df = pd.read_csv('https://raw.githubusercontent.com/martianunlimited/compx310_
datasets/main/housing.csv')
df
```

2 Preparing the full data for the regression algorithms:replace any missing values & turn categorical values ('ocean_proximity') into numeric ones

In [ ]:
```
# Replace missing values with median
df = df.fillna(df.select_dtypes(include=['number']).median())

# Convert categorical features to numerical using one-hot encoding
df = pd.get_dummies(df, columns=['ocean_proximity'])
df.info()
```

3 Define X as all the numerical features in the dataframe except median_house_value and y to be the target value median_house_value. Split the data into 80% train and 20% test data

In [ ]:
```
from sklearn.model_selection import train_test_split

# Define features and target
X = df.drop('median_house_value', axis=1)
y = df['median_house_value']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rando
m_state=ID)

X_train.shape, X_test.shape
```

4 Define a function called run_reg that takes a regressor and X_train, X_test, y_train, y_test as arguments

In [ ]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error
from sklearn.linear_model import SGDRegressor

# Define the run_reg function
def run_reg(regressor, X_train, X_test, y_train, y_test):

 # Train the regressor using the train data
 regressor.fit(X_train, y_train)

 # Compute predictions for the test data
 y_pred = regressor.predict(X_test)

 # Set predictions smaller than 15000 to 15000
 y_pred[y_pred < 15000] = 15000

 # Set predictions larger than 500000 to 500000
 y_pred[y_pred>500000]=500000

 # Compute the MAE for the test data
 mae = mean_absolute_error(y_test, y_pred)

 # Scatterplot of true test targets vs. predictions

 plt.scatter(y_test, y_pred)
 plt.xlabel("True Test Targets")
 plt.ylabel("Predictions")
 plt.title(f'MAE = {mae:0.4}')
 plt.show()
 return mae
```

5 Call run_reg each with the following SGDRegressor with the following learning rates [0.00000001, 0.0001]

In [ ]:
```
# Define the regressors with different learning rates
regressor_a = SGDRegressor(learning_rate='constant', eta0=0.00000001, random_s
tate=ID)
regressor_b = SGDRegressor(learning_rate='constant', eta0=0.0001, random_state
=ID)

# Run the regressors
mae_a_original = run_reg(regressor_a, X_train, X_test, y_train, y_test)
mae_b_original = run_reg(regressor_b, X_train, X_test, y_train, y_test)

print(f"MAE for regressor a with original data: {mae_a_original}")
print(f"MAE for regressor b with original data: {mae_b_original}")
```

6 Scale the features using standard scaler

In [ ]:
```
from sklearn.preprocessing import StandardScaler

# Load the scaled data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

7 Rerun 5) with X_train_scaled and X_test_scaled instead of X_train and X_test

In [ ]:
```
mae_a_scaled = run_reg(regressor_a, X_train_scaled, X_test_scaled, y_train, y_
test)
mae_b_scaled = run_reg(regressor_b, X_train_scaled, X_test_scaled, y_train, y_
test)

print(f"MAE for regressor a with scaled data: {mae_a_scaled}")
print(f"MAE for regressor b with scaled data: {mae_b_scaled}")
```

Comment on the difference in the results in steps A.5 vs A.7

In A.5 the MAE was 174,144.26 suggesting larger prediction errors which is evident from the more scattered points in the corresponding plot.

In A.7 the MAE reduced substantially to 49,862.04 indicating much closer predictions to the actual values as shown by the tighter clustering around the diagonal in the second plot.

Part B: Learning Curves

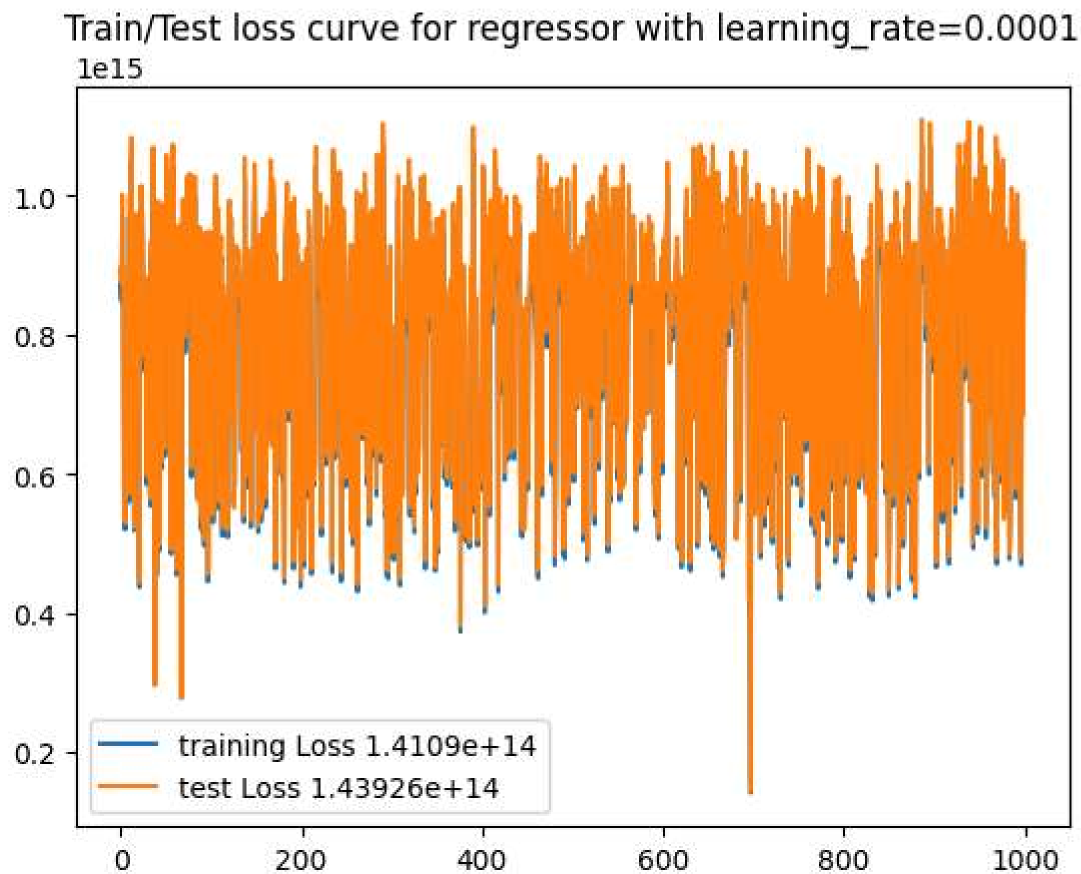
1) Skeleton Code

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import SGDRegressor
def run_training_curve(X_train,X_test,y_train,y_test,learning_rate=0.000000001,no_epoch=1000):
    sgd=SGDRegressor(random_state=1234567,verbose=0,learning_rate='constant',eta0=learning_rate) # Don't change the random_state, this so that we more likely to have meaningful result
    train_loss_list=[]
    test_loss_list=[]
    for epoch in range(no_epoch):
        sgd.partial_fit(X_train,y_train)
        y_pred_train=sgd.predict(X_train)
        y_pred_test=sgd.predict(X_test)
        train_loss_list.append(np.sqrt(np.mean((y_pred_train-y_train)**2)))
        test_loss_list.append(np.sqrt(np.mean((y_pred_test-y_test)**2)))
    plt.plot(train_loss_list,label=f'training Loss {np.min(train_loss_list[500:]):.6}')
    plt.plot(test_loss_list,label=f'test Loss {np.min(test_loss_list[500:]):.6}')
    plt.legend()
    plt.title(f"Train/Test loss curve for regressor with learning_rate={learning_rate}")
    plt.show()
```

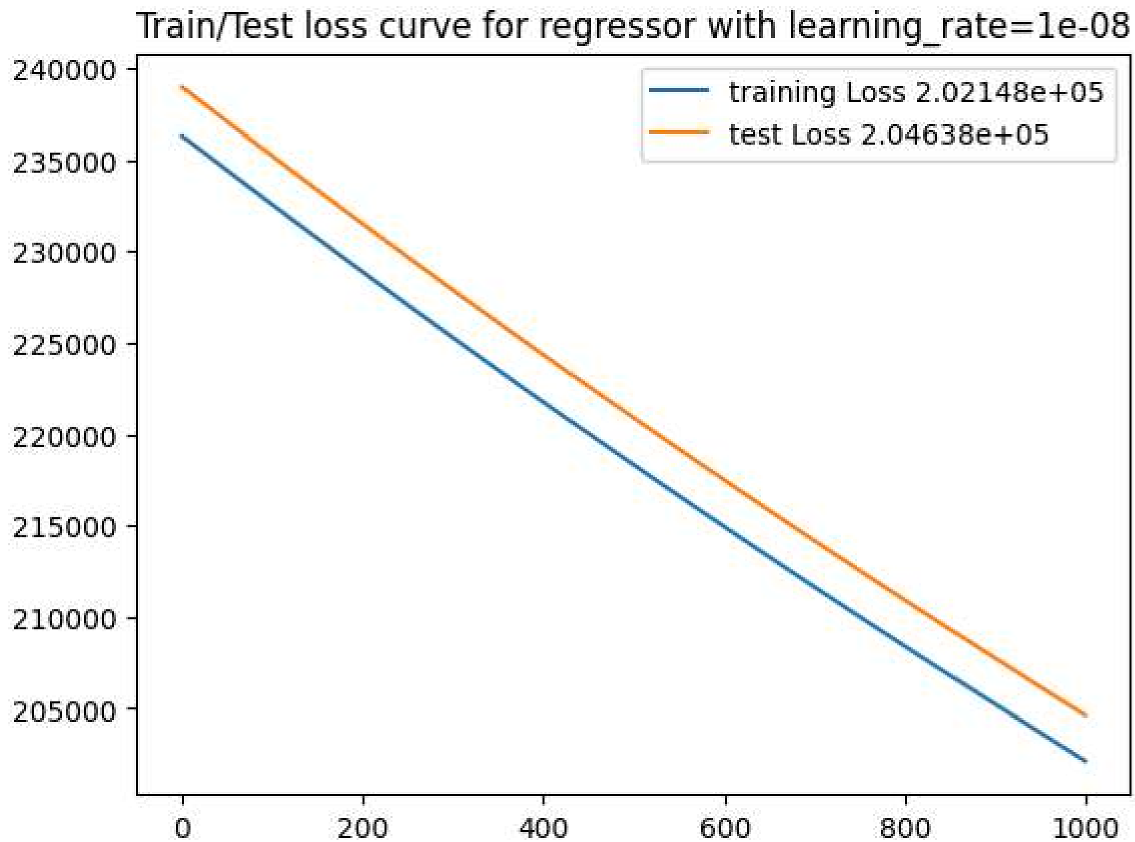
2) Plot the learning curves from cases A to E

In [ ]:
```
# Case A
run_training_curve(X_train, X_test, y_train, y_test, learning_rate=0.00000001)
```

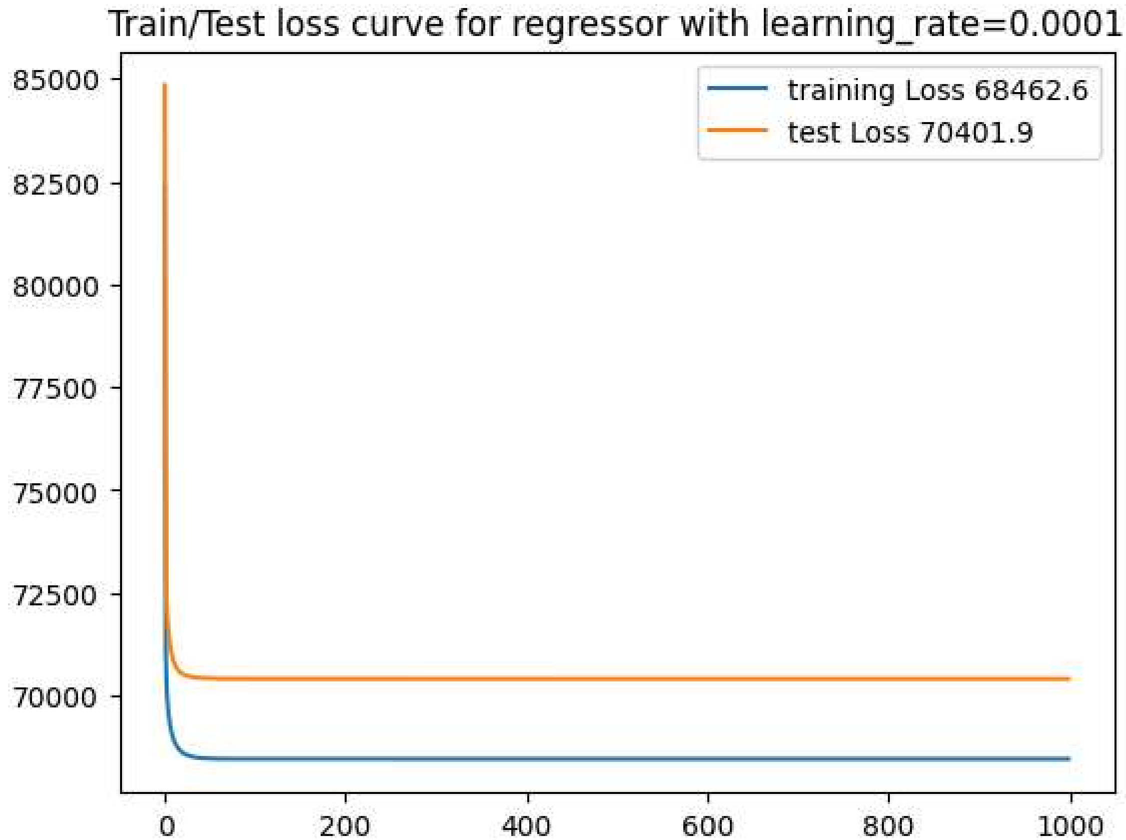
In [ ]:
```
# Case B
run_training_curve(X_train, X_test, y_train, y_test, learning_rate=0.0001)
```



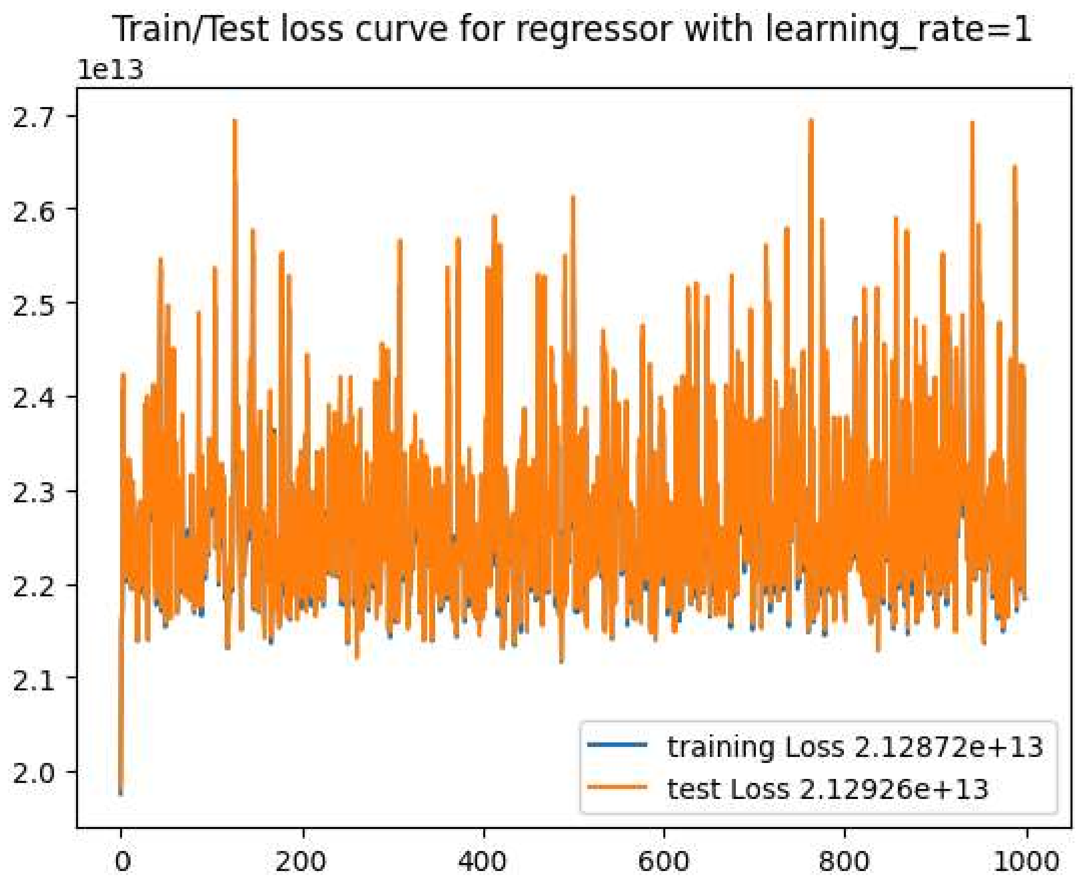
In [ ]:
```
# Case C
run_training_curve(X_train_scaled, X_test_scaled, y_train, y_test, learning_ra
te=0.00000001)
```



In [ ]:
```
# Case D
run_training_curve(X_train_scaled, X_test_scaled, y_train, y_test, learning_ra
te=0.0001)
```



In [ ]:
```
# Case E
run_training_curve(X_train_scaled, X_test_scaled, y_train, y_test, learning_ra
te=1)
```



Comment about the learning curves obtained in step B.2

In Case B (unscaled, 0.0001 learning rate) and Case E (scaled, 1 learning rate), the high learning rates cause the loss to fluctuate, making the training unstable. This prevents the model from converging, indicating that the learning rate needs to be lowered for stable and effective training.

In Case A (unscaled, 1e-8 learning rate), the very low learning rate leads to a slow but steady decrease in loss. The model improves gradually but requires many more epochs to converge. Slightly increasing the learning rate or adding more epochs could speed up convergence.

In Case C (scaled data with a 1e-8 learning rate), the very low learning rate results in slow but stable convergence. The model's progress is steady, but it may take many more epochs to reach an optimal solution. Increasing the learning rate slightly could accelerate convergence without sacrificing stability.

In Case D (scaled data with a 0.0001 learning rate), the learning rate is well-chosen, leading to fast and stable convergence. The model quickly reaches a good solution with fewer epochs, showing that scaling combined with an appropriate learning rate allows for efficient training.

Part C - Simple neural networks (MLP)

1) Load MNIST Dataset

In [ ]:
```
from keras.datasets import mnist

# Load the MNIST dataset
(X_trainval,y_trainval),(X_test,y_test) = mnist.load_data()
```

2) split X_trainval and y_trainval into a 75%/25% training and validation split stratified on y_trainval

In [ ]:
```
from sklearn.model_selection import train_test_split

# Split into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test
_size=0.25, stratify=y_trainval)
```

3) normalize X_train, X_val and X_test

In [ ]:
```
# Normalize the data
X_train=X_train/255
X_val=X_val/255
X_test=X_test/255
```

4) import the keras library and define your model using model=keras.models.Sequential()

In [ ]:
```
import tensorflow as tf
from tensorflow import keras

# Define the model using Keras Sequential API
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),     # Flatten the input
    keras.layers.Dense(50, activation='relu'),      # First hidden layer with
50 units
    keras.layers.Dense(50, activation='relu'),      # Second hidden layer with
50 units
    keras.layers.Dense(50, activation='relu'),      # Third hidden layer with
50 units
    keras.layers.Dense(10, activation='softmax')    # Output layer with 10 uni
ts (one for each digit)
])
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/reshaping/flatten.p
y:37: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a lay
er. When using Sequential models, prefer using an `Input(shape)` object as th
e first layer in the model instead.
  super().__init__(**kwargs)
```

5) Set your optimizer as ADAM, and your loss function to sparse_categorical_crossentropy and metrics=['accuracy']

In [ ]:
```
# Compile the model with Adam optimizer and sparse categorical crossentropy lo
ss
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metric
s=['accuracy'])
```

6) view your model using model.summary()

In [ ]:
```
# View the model summary
model.summary()
```

Model: "sequential_8"

Layer (type)	Output Shape	
flatten_1 (Flatten)	(None, 784)	
dense_11 (Dense)	(None, 50)	
dense_12 (Dense)	(None, 50)	
dense_13 (Dense)	(None, 50)	
dense_14 (Dense)	(None, 10)	

Total params: 44,860 (175.23 KB)

Trainable params: 44,860 (175.23 KB)

Non-trainable params: 0 (0.00 B)

7) Fit your model using X_train for 30 epoch & to set validation_data=(X_val,y_val).

In [75]:
```
# Train the model on the training data for 30 epochs, with validation data
history = model.fit(X_train, y_train, epochs=30, validation_data=(X_val, y_va
l), verbose=2)
```

```
Epoch 1/30
1407/1407 - 8s - 6ms/step - accuracy: 0.8986 - loss: 0.3430 - val_accuracy:
0.9383 - val_loss: 0.2030
Epoch 2/30
1407/1407 - 9s - 6ms/step - accuracy: 0.9530 - loss: 0.1574 - val_accuracy:
0.9541 - val_loss: 0.1479
Epoch 3/30
1407/1407 - 5s - 3ms/step - accuracy: 0.9641 - loss: 0.1154 - val_accuracy:
0.9585 - val_loss: 0.1324
Epoch 4/30
1407/1407 - 4s - 3ms/step - accuracy: 0.9710 - loss: 0.0933 - val_accuracy:
0.9617 - val_loss: 0.1292
Epoch 5/30
1407/1407 - 5s - 4ms/step - accuracy: 0.9751 - loss: 0.0785 - val_accuracy:
0.9634 - val_loss: 0.1236
Epoch 6/30
1407/1407 - 5s - 3ms/step - accuracy: 0.9787 - loss: 0.0665 - val_accuracy:
0.9677 - val_loss: 0.1112
Epoch 7/30
1407/1407 - 5s - 3ms/step - accuracy: 0.9820 - loss: 0.0562 - val_accuracy:
0.9652 - val_loss: 0.1258
Epoch 8/30
1407/1407 - 6s - 4ms/step - accuracy: 0.9829 - loss: 0.0504 - val_accuracy:
0.9609 - val_loss: 0.1449
Epoch 9/30
1407/1407 - 4s - 3ms/step - accuracy: 0.9848 - loss: 0.0464 - val_accuracy:
0.9668 - val_loss: 0.1220
Epoch 10/30
1407/1407 - 5s - 3ms/step - accuracy: 0.9873 - loss: 0.0382 - val_accuracy:
0.9681 - val_loss: 0.1225
Epoch 11/30
1407/1407 - 6s - 5ms/step - accuracy: 0.9878 - loss: 0.0374 - val_accuracy:
0.9640 - val_loss: 0.1426
Epoch 12/30
1407/1407 - 5s - 4ms/step - accuracy: 0.9891 - loss: 0.0324 - val_accuracy:
0.9666 - val_loss: 0.1364
Epoch 13/30
1407/1407 - 7s - 5ms/step - accuracy: 0.9899 - loss: 0.0308 - val_accuracy:
0.9704 - val_loss: 0.1303
Epoch 14/30
1407/1407 - 4s - 3ms/step - accuracy: 0.9902 - loss: 0.0282 - val_accuracy:
0.9698 - val_loss: 0.1327
Epoch 15/30
1407/1407 - 6s - 4ms/step - accuracy: 0.9907 - loss: 0.0283 - val_accuracy:
0.9670 - val_loss: 0.1441
Epoch 16/30
1407/1407 - 5s - 4ms/step - accuracy: 0.9914 - loss: 0.0251 - val_accuracy:
0.9687 - val_loss: 0.1418
Epoch 17/30
1407/1407 - 4s - 3ms/step - accuracy: 0.9928 - loss: 0.0217 - val_accuracy:
0.9681 - val_loss: 0.1462
Epoch 18/30
1407/1407 - 7s - 5ms/step - accuracy: 0.9930 - loss: 0.0215 - val_accuracy:
0.9721 - val_loss: 0.1417
Epoch 19/30
1407/1407 - 9s - 6ms/step - accuracy: 0.9932 - loss: 0.0197 - val_accuracy:
0.9697 - val_loss: 0.1562
```

```
Epoch 20/30
1407/1407 - 5s - 4ms/step - accuracy: 0.9929 - loss: 0.0201 - val_accuracy:
0.9709 - val_loss: 0.1450
Epoch 21/30
1407/1407 - 9s - 6ms/step - accuracy: 0.9938 - loss: 0.0183 - val_accuracy:
0.9713 - val_loss: 0.1519
Epoch 22/30
1407/1407 - 6s - 5ms/step - accuracy: 0.9938 - loss: 0.0173 - val_accuracy:
0.9696 - val_loss: 0.1652
Epoch 23/30
1407/1407 - 11s - 8ms/step - accuracy: 0.9939 - loss: 0.0172 - val_accuracy:
0.9695 - val_loss: 0.1693
Epoch 24/30
1407/1407 - 4s - 3ms/step - accuracy: 0.9946 - loss: 0.0172 - val_accuracy:
0.9681 - val_loss: 0.1657
Epoch 25/30
1407/1407 - 4s - 3ms/step - accuracy: 0.9951 - loss: 0.0142 - val_accuracy:
0.9693 - val_loss: 0.1827
Epoch 26/30
1407/1407 - 6s - 5ms/step - accuracy: 0.9945 - loss: 0.0175 - val_accuracy:
0.9605 - val_loss: 0.2195
Epoch 27/30
1407/1407 - 4s - 3ms/step - accuracy: 0.9950 - loss: 0.0152 - val_accuracy:
0.9658 - val_loss: 0.1828
Epoch 28/30
1407/1407 - 6s - 4ms/step - accuracy: 0.9953 - loss: 0.0139 - val_accuracy:
0.9721 - val_loss: 0.1674
Epoch 29/30
1407/1407 - 5s - 4ms/step - accuracy: 0.9950 - loss: 0.0150 - val_accuracy:
0.9649 - val_loss: 0.2045
Epoch 30/30
1407/1407 - 10s - 7ms/step - accuracy: 0.9957 - loss: 0.0141 - val_accuracy:
0.9715 - val_loss: 0.1643
```

8) prediction of X_test, and visualize the confusion matrix

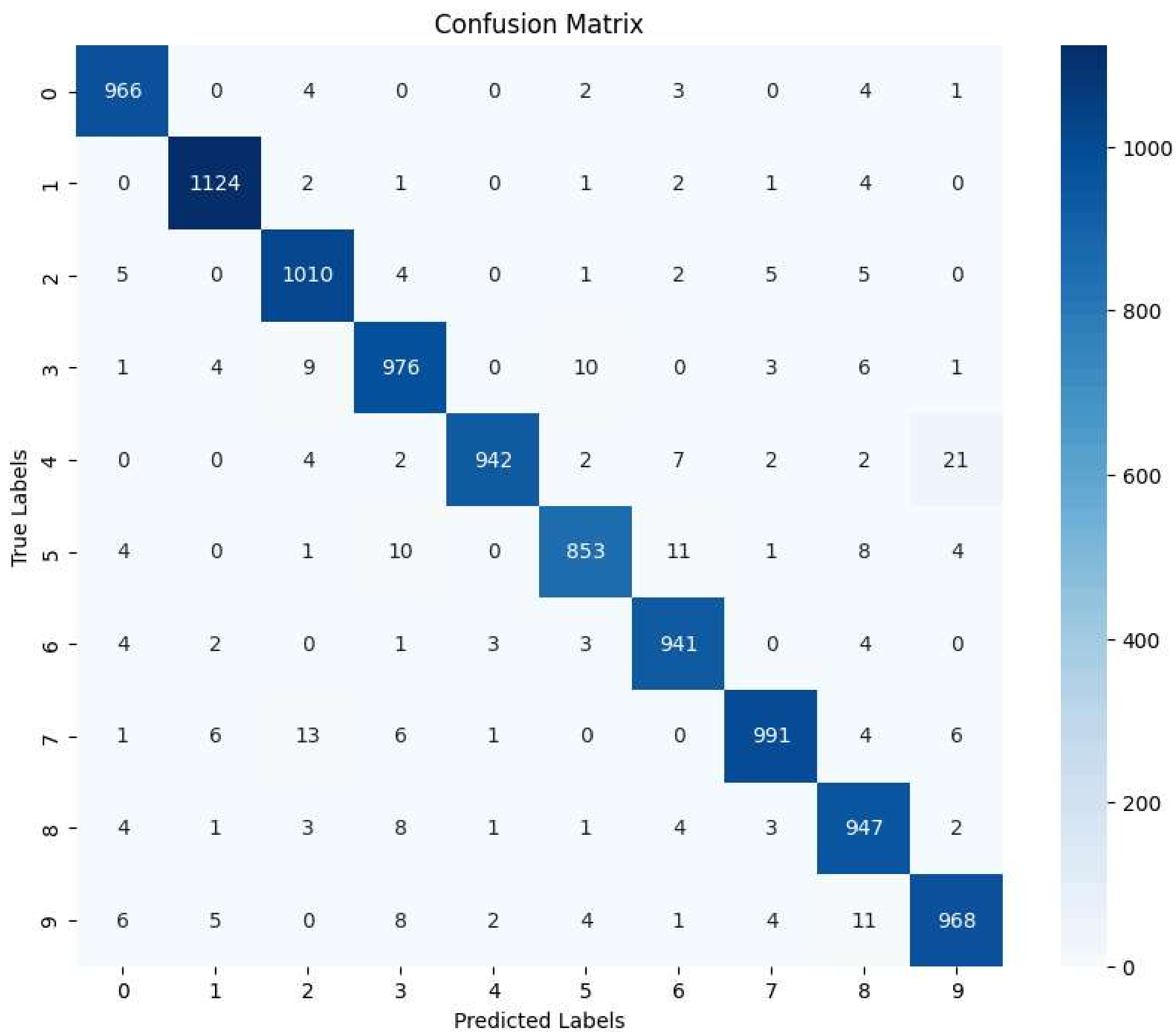
In [76]:
```
import seaborn as sns
from sklearn.metrics import confusion_matrix
import numpy as np

# Predict on the test set
y_pred_prob = model.predict(X_test)
y_pred = np.argmax(y_pred_prob, axis=1)

# Compute the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
```



Comment on the training and validation loss & accuracies and comment on the confusion matrix

The training and validation loss curves typically show a decreasing trend as the model learns indicating effective training.If the validation loss starts to increase while the training loss continues to decrease it is overfitting.A gap between training and validation accuracy curves could indicate the model is performing well on the training data but less so on unseen data, highlighting the importance of monitoring these metrics throughout the training process. The confusion matrix helps to identify where the model might be struggling especially between similar-looking digits.

Part D: Transfer learning

1) set batch_size to 256

In [77]:
```
# Define the batch size
batch_size = 256
```

2) load the data, and scaling the features by dividing it by 255.

In [78]:
```
from tensorflow.keras.datasets import cifar100

# Load the CIFAR-100 dataset
(X_trainval, y_trainval), (X_test, y_test) = cifar100.load_data()

# Scale the pixel values to the range [0, 1]
X_trainval = X_trainval / 255
X_test = X_test / 255

# Check the shapes of the data
print(X_trainval.shape)
print(X_test.shape)
```

```
(50000, 32, 32, 3)
(10000, 32, 32, 3)
```

3) split X_trainval into : 80% train and 20% validation

In [79]:
```
from sklearn.model_selection import train_test_split

# Split the data into 80% training and 20% validation sets with stratification
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=0.2, stratify=y_trainval, random_state=ID)

# Check the shapes after splitting
print(X_train.shape)
print(X_val.shape)
```

```
(40000, 32, 32, 3)
(10000, 32, 32, 3)
```

4) Select one of the smaller pre-trained models from [https://keras.io/api/applications/ (https://keras.io/api/applications/)](https://keras.io/api/applications/).

In [80]:
```
from tensorflow.keras.applications import MobileNetV2
```

5) Load the model without its final layer, include pre-trained weights, "freeze" the model & combine it with a final Dense(100) layer for predicting the 100 classes, using softmax activation.

In [81]:
```
from tensorflow.keras import layers, models
from tensorflow.keras.layers import Resizing
from tensorflow.keras.applications import MobileNetV2

# Load the pre-trained MobileNetV2 model without the final layer, include pre-
trained weights
base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(2
24, 224, 3))

# Freeze the base model layers to prevent their weights from being updated dur
ing initial training
base_model.trainable = False

# Create the model with a resizing layer and custom layers on top
model = models.Sequential([
    Resizing(224, 224, input_shape=(32, 32, 3)),
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(100, activation='softmax')
])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applica
tions/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no
_top.h5
9406464/9406464 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step

/usr/local/lib/python3.10/dist-packages/keras/src/layers/preprocessing/tf_dat
a_layer.py:19: UserWarning: Do not pass an `input_shape`/`input_dim` argument
to a layer. When using Sequential models, prefer using an `Input(shape)` obje
ct as the first layer in the model instead.
  super().__init__(**kwargs)
```

6) Compile the model, and produce display model.summary()

In [82]:
```
# Compile the model with Adam optimizer and sparse categorical crossentropy lo
ss
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metric
s=['accuracy'])

# Display the model summary to verify the architecture
model.summary()
```

Model: "sequential_9"

Layer (type)	Output Shape	
resizing_5 (Resizing)	(None, 224, 224, 3)	
mobilenetv2_1.00_224 (Functional)	(None, 7, 7, 1280)	2
global_average_pooling2d_7 (GlobalAveragePooling2D)	(None, 1280)	
dense_15 (Dense)	(None, 100)	

Total params: 2,386,084 (9.10 MB)

Trainable params: 128,100 (500.39 KB)

Non-trainable params: 2,257,984 (8.61 MB)

7) Train the frozen model for 20 epochs.

In [2]:
```
# Train the model for 20 epochs with frozen MobileNet layers
history_frozen = model.fit(X_train, y_train, epochs=20, validation_data=(X_va
l, y_val), batch_size=batch_size)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-70beffaeba7b> in <cell line: 2>()
      1 # Train the model for 20 epochs with frozen MobileNet layers
----> 2 history_frozen = model.fit(X_train, y_train, epochs=20, validation_da
ta=(X_val, y_val), batch_size=batch_size)

NameError: name 'model' is not defined
```

8) Unfreeze the model, and train for a further 30 iterations.

In [1]:
```
test_accuracies_frozen = history_frozen.history['test_accuracy']
test_accuracies_fine = history_fine.history['test_accuracy']

print(test_accuracies_frozen)
print(test_accuracies_fine)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-d967fc3c0724> in <cell line: 1>()
----> 1 test_accuracies_frozen = history_frozen.history['test_accuracy']
      2 test_accuracies_fine = history_fine.history['test_accuracy']
      3 
      4 print(test_accuracies_frozen)
      5 print(test_accuracies_fine)

NameError: name 'history_frozen' is not defined
```

9) Plot the accuracy learning curves for train, validation, and test, for both the pretraining stage

In [3]:
```
# Unfreeze the MobileNet base model layers for fine-tuning
model.layers[1].trainable = True  # Unfreeze the MobileNet model

# Recompile the model with a lower learning rate for fine-tuning
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss
='sparse_categorical_crossentropy', metrics=['accuracy'])

# Fine-tune the model for 30 more epochs with the custom callback
history_fine = model.fit(X_train, y_train, epochs=30, validation_data=(X_val,
y_val), batch_size=batch_size, callbacks=[test_acc_callback])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-a4343e3e8e62> in <cell line: 2>()
      1 # Unfreeze the MobileNet base model layers for fine-tuning
----> 2 model.layers[1].trainable = True  # Unfreeze the MobileNet model
      3 
      4 # Recompile the model with a lower learning rate for fine-tuning
      5 model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.000
1), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

NameError: name 'model' is not defined
```

Part E

1-4. Predict, Identify Worst Misclassifications, and Plot

In [4]:
```
# Get predictions
preds = model.predict(X_test)

# For each class, identify the worst misclassification
worst_misclassifications = []

for true_class in range(100):
    X_class = X_test[y_test.flatten() == true_class]
    y_class = preds[y_test.flatten() == true_class]
    true_scores = y_class[:, true_class]
    diff = y_class.max(axis=1) - true_scores
    worst_idx = diff.argmax()
    worst_misclassifications.append((true_class, worst_idx))

# Plot the worst misclassifications
for true_class, worst_idx in worst_misclassifications:
    plt.imshow(X_test[y_test.flatten() == true_class][worst_idx])
    predicted_class = np.argmax(preds[y_test.flatten() == true_class][worst_id
x])
    plt.title(f"True: {true_class}, Predicted: {predicted_class}")
    plt.show()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-d5d3e3dd4a4a> in <cell line: 2>()
      1 # Get predictions
----> 2 preds = model.predict(X_test)
      3
      4 # For each class, identify the worst misclassification
      5 worst_misclassifications = []

NameError: name 'model' is not defined
```

Comment on Mispredictions

The mispredictions generally occur for classes that are visually similar, indicating where the model struggles to distinguish features. Understanding these weaknesses can help in refining the model or data preprocessing to improve accuracy.