```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


data = pd.read_csv("/content/database.csv")
data.columns
```

```
Index(['Date', 'Time', 'Latitude', 'Longitude', 'Type', 'Depth', 'Depth Error',
       'Depth Seismic Stations', 'Magnitude', 'Magnitude Type',
       'Magnitude Error', 'Magnitude Seismic Stations', 'Azimuthal Gap',
       'Horizontal Distance', 'Horizontal Error', 'Root Mean Square', 'ID',
       'Source', 'Location Source', 'Magnitude Source', 'Status'],
      dtype='object')
```

```
data = data[['Date', 'Time', 'Latitude', 'Longitude', 'Depth', 'Magnitude']]
data.head()
```

	Date	Time	Latitude	Longitude	Depth	Magnitude
0	01/02/1965	13:44:18	19.246	145.616	131.6	6.0
1	01/04/1965	11:29:49	1.863	127.352	80.0	5.8
2	01/05/1965	18:05:58	-20.579	-173.972	20.0	6.2
3	01/08/1965	18:49:43	-59.076	-23.557	15.0	5.8

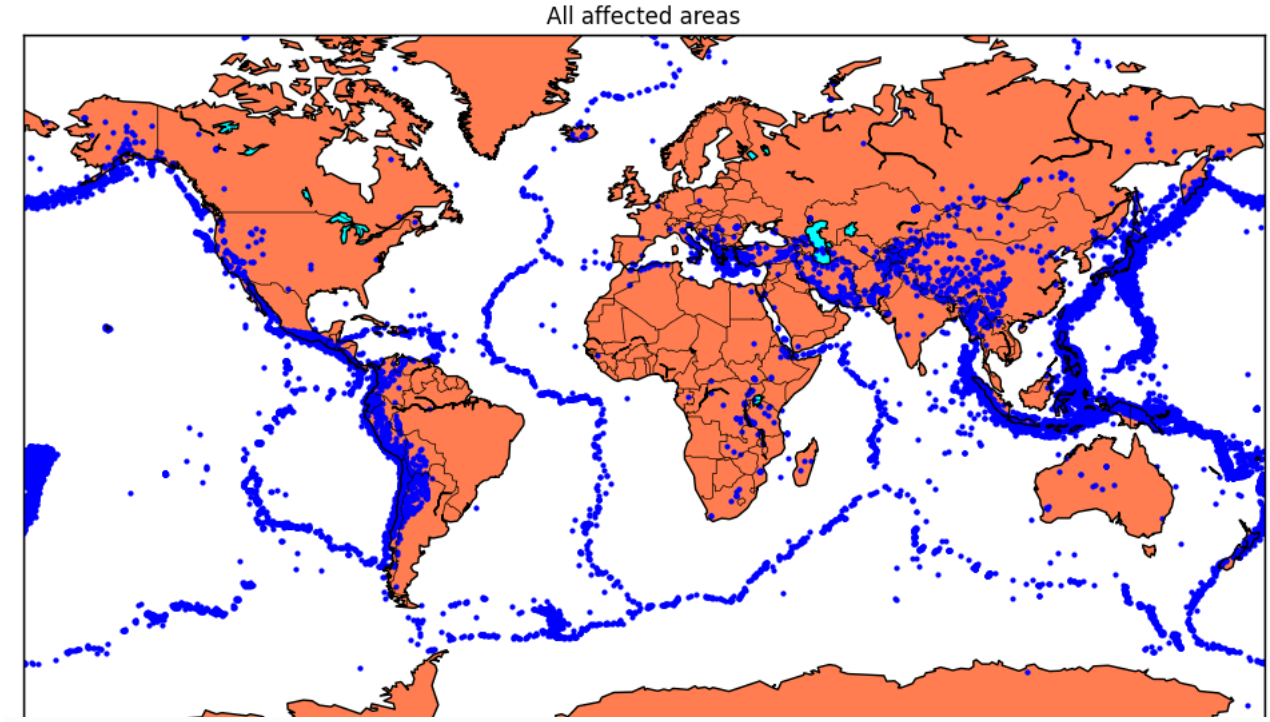
Next steps: Generate code with data | View recommended plots | New interactive sheet

```
import datetime
import time

timestamp = []
for d, t in zip(data['Date'], data['Time']):
    try:
        ts = datetime.datetime.strptime(d+' '+t, '%m/%d/%Y %H:%M:%S')
        timestamp.append(time.mktime(ts.timetuple()))
    except ValueError:
        # print('ValueError')
        timestamp.append('ValueError')
timeStamp = pd.Series(timestamp)
data['Timestamp'] = timeStamp.values
final_data = data.drop(['Date', 'Time'], axis=1)
final_data = final_data[final_data.Timestamp != 'ValueError']
final_data.head()
```

	Latitude	Longitude	Depth	Magnitude	Timestamp
0	19.246	145.616	131.6	6.0	-157630542.0
1	1.863	127.352	80.0	5.8	-157465811.0
2	-20.579	-173.972	20.0	6.2	-157355642.0
3	-59.076	-23.557	15.0	5.8	-157093817.0

Next steps: Generate code with final_data | View recommended plots | New interactive sheet

```
from mpl_toolkits.basemap import Basemap

m = Basemap(projection='mill',llcrnrlat=-80,urcrnrlat=80, llcrnrlon=-180,urcrnrlon=180,lat_ts=20,resolution='c')

longitudes = data["Longitude"].tolist()
latitudes = data["Latitude"].tolist()
#m = Basemap(width=12000000,height=9000000,projection='lcc',
            #resolution=None,lat_1=80.,lat_2=55,lat_0=80,lon_0=-107.)
x,y = m(longitudes,latitudes)

fig = plt.figure(figsize=(12,10))
plt.title("All affected areas")
m.plot(x, y, "o", markersize = 2, color = 'blue')
m.drawcoastlines()
m.fillcontinents(color='coral',lake_color='aqua')
m.drawmapboundary()
m.drawcountries()
plt.show()
```




```
X = final_data[['Timestamp', 'Latitude', 'Longitude']]
y = final_data[['Magnitude', 'Depth']]
from sklearn.model_selection import train_test_split


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
print(X_train.shape, X_test.shape, y_train.shape, X_test.shape)
```

```
(18727, 3) (4682, 3) (18727, 2) (4682, 3)
```

```
from keras.models import Sequential
from keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import GridSearchCV

def create_model(neurons=16, activation='relu', optimizer='adam', loss='sparse_categorical_crossentropy' ):
    model = Sequential()
    model.add(Dense(neurons, activation=activation, input_shape=(3,)))
    model.add(Dense(neurons, activation=activation))
    model.add(Dense(2, activation='softmax'))

    model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

    return model


model = KerasClassifier(model=create_model, verbose=0)

# neurons = [16, 64, 128, 256]
neurons = [16]
# batch_size = [10, 20, 50, 100]
batch_size = [10]
epochs = [10]
# activation = ['relu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear', 'exponential']
activation = ['sigmoid', 'relu']
# optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']
optimizer = ['SGD', 'Adadelta']
loss = ['squared_hinge']

param_grid = dict(neurons=neurons, batch_size=batch_size, epochs=epochs, activation=activation, optimizer=optimizer, loss=loss)


grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, y_train)
print(model.get_params().keys())

print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
```

```
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Show hidden output

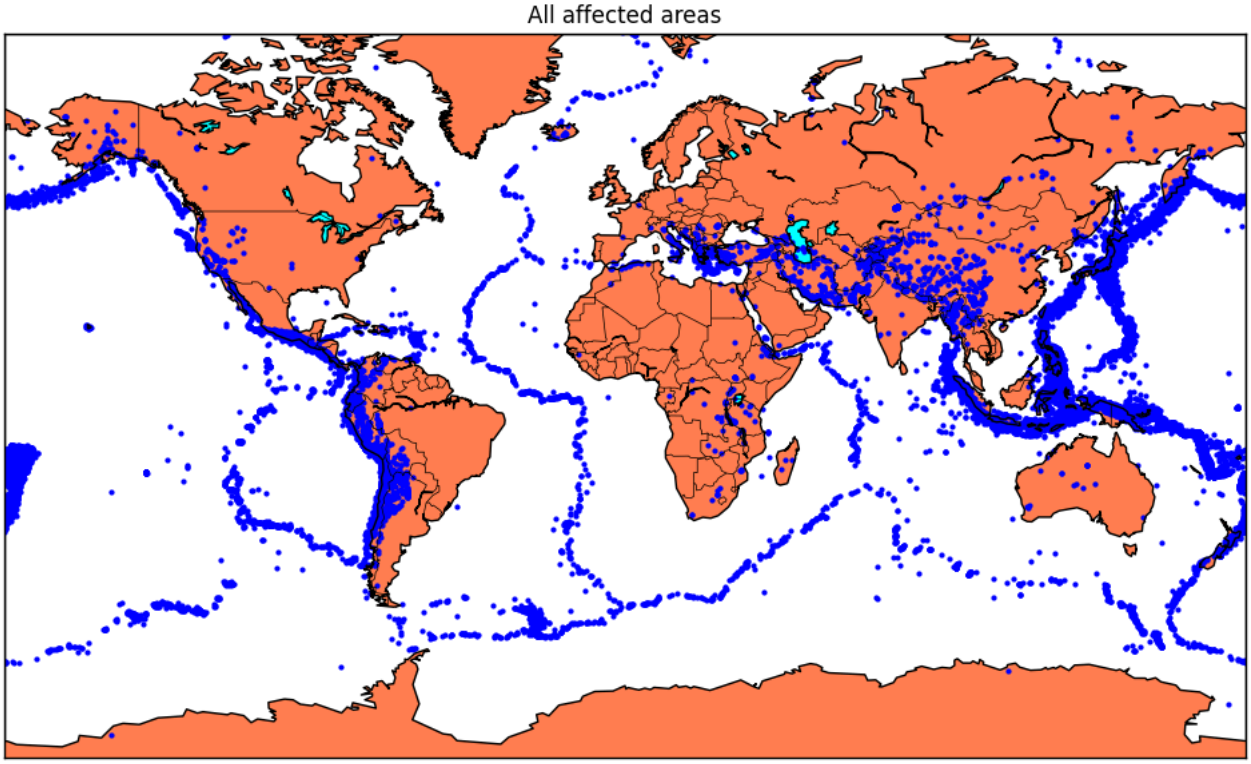
Next steps: Explain error

```
model = Sequential()
model.add(Dense(16, activation='relu', input_shape=(3,)))
model.add(Dense(16, activation='relu'))
model.add(Dense(2, activation='softmax'))

model.compile(optimizer='SGD', loss='squared_hinge', metrics=['accuracy'])
model.fit(X_train, y_train, batch_size=10, epochs=20, verbose=1, validation_data=(X_test, y_test))

[test_loss, test_acc] = model.evaluate(X_test, y_test)
print("Evaluation result on Test Data : Loss = {}, accuracy = {}".format(test_loss, test_acc))


import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
import datetime
import time
import warnings
warnings.filterwarnings('ignore')

data = pd.read_csv("/content/database.csv")
print(f"Original data shape: {data.shape}")
print(f"Columns: {data.columns.tolist()}")

data = data[['Date', 'Time', 'Latitude', 'Longitude', 'Depth', 'Magnitude']]
print(f"Data after column selection: {data.shape}")
print(data.head())

# Convert Date and Time to timestamp
timestamp = []
for d, t in zip(data['Date'], data['Time']):
    try:
        ts = datetime.datetime.strptime(d+' '+t, '%m/%d/%Y %H:%M:%S')
        timestamp.append(time.mktime(ts.timetuple()))
    except ValueError:
        timestamp.append(np.nan)

data['Timestamp'] = timestamp


final_data = data.drop(['Date', 'Time'], axis=1)
final_data = final_data.dropna()
print(f"Data after cleaning: {final_data.shape}")
print(final_data.head())


try:
    from mpl_toolkits.basemap import Basemap
    print("\nCreating visualization...")

    m = Basemap(projection='mill', llcrnrlat=-80, urcrnrlat=80,
                llcrnrlon=-180, urcrnrlon=180, lat_ts=20, resolution='c')

    longitudes = final_data["Longitude"].tolist()
    latitudes = final_data["Latitude"].tolist()

    x, y = m(longitudes, latitudes)

    fig = plt.figure(figsize=(12, 10))
    plt.title("All affected areas")
    m.plot(x, y, "o", markersize=2, color='blue')
    m.drawcoastlines()
    m.fillcontinents(color='coral', lake_color='aqua')
    m.drawmapboundary()
    m.drawcountries()
    plt.show()

except ImportError:
    print("Basemap not available. Skipping visualization.")
    # Alternative simple plot
    plt.figure(figsize=(12, 8))
    plt.scatter(final_data['Longitude'], final_data['Latitude'],
```

```
            c=final_data['Magnitude'], cmap='viridis', alpha=0.6)
        plt.colorbar(label='Magnitude')
        plt.xlabel('Longitude')
        plt.ylabel('Latitude')
        plt.title('Earthquake Locations colored by Magnitude')
        plt.show()

    # Prepare features and targets for REGRESSION (not classification)
    print("\nPreparing features and targets...")
    X = final_data[['Timestamp', 'Latitude', 'Longitude']].values
    y = final_data[['Magnitude', 'Depth']].values

    print(f"Features shape: {X.shape}")
    print(f"Targets shape: {y.shape}")

    # Scale features
    scaler_X = StandardScaler()
    scaler_y = StandardScaler()
    X_scaled = scaler_X.fit_transform(X)
    y_scaled = scaler_y.fit_transform(y)

    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(
        X_scaled, y_scaled, test_size=0.2, random_state=42
    )

    print(f"Training shapes: X_train={X_train.shape}, y_train={y_train.shape}")
    print(f"Testing shapes: X_test={X_test.shape}, y_test={y_test.shape}")

    # Neural Network Model (using TensorFlow/Keras)
    try:
        import tensorflow as tf
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Dense, Dropout
        from tensorflow.keras.optimizers import Adam

        print("\nBuilding neural network model...")

        def create_regression_model(neurons=64, activation='relu', optimizer='adam',
                                    learning_rate=0.001, dropout_rate=0.2):
            model = Sequential([
                Dense(neurons, activation=activation, input_shape=(3,)),
                Dropout(dropout_rate),
                Dense(neurons//2, activation=activation),
                Dropout(dropout_rate),
                Dense(neurons//4, activation=activation),
                Dense(2, activation='linear')
            ])

            if optimizer == 'adam':
                opt = Adam(learning_rate=learning_rate)
            else:
                opt = optimizer

            model.compile(optimizer=opt, loss='mse', metrics=['mae'])
            return model


        model = create_regression_model(neurons=128, activation='relu',
                                        learning_rate=0.001, dropout_rate=0.3)

        print("Model architecture:")
        model.summary()


        print("\nTraining model...")
        history = model.fit(X_train, y_train,
                            batch_size=32,
                            epochs=50,
                            verbose=1,
                            validation_data=(X_test, y_test),
                            callbacks=[tf.keras.callbacks.EarlyStopping(patience=10)])


        print("\nEvaluating model...")
        train_loss, train_mae = model.evaluate(X_train, y_train, verbose=0)
        test_loss, test_mae = model.evaluate(X_test, y_test, verbose=0)

        print(f"Training - Loss: {train_loss:.4f}, MAE: {train_mae:.4f}")
        print(f"Testing - Loss: {test_loss:.4f}, MAE: {test_mae:.4f}")


        y_pred = model.predict(X_test)
```

```
        y_pred = model.predict(X_test)


        y_test_original = scaler_y.inverse_transform(y_test)
        y_pred_original = scaler_y.inverse_transform(y_pred)


        mag_mse = mean_squared_error(y_test_original[:, 0], y_pred_original[:, 0])
        mag_r2 = r2_score(y_test_original[:, 0], y_pred_original[:, 0])
        depth_mse = mean_squared_error(y_test_original[:, 1], y_pred_original[:, 1])
        depth_r2 = r2_score(y_test_original[:, 1], y_pred_original[:, 1])

        print(f"\nMagnitude Prediction - MSE: {mag_mse:.4f}, R²: {mag_r2:.4f}")
        print(f"Depth Prediction - MSE: {depth_mse:.4f}, R²: {depth_r2:.4f}")


        plt.figure(figsize=(12, 4))

        plt.subplot(1, 2, 1)
        plt.plot(history.history['loss'], label='Training Loss')
        plt.plot(history.history['val_loss'], label='Validation Loss')
        plt.title('Model Loss')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.legend()

        plt.subplot(1, 2, 2)
        plt.plot(history.history['mae'], label='Training MAE')
        plt.plot(history.history['val_mae'], label='Validation MAE')
        plt.title('Model MAE')
        plt.xlabel('Epoch')
        plt.ylabel('MAE')
        plt.legend()

        plt.tight_layout()
        plt.show()


        plt.figure(figsize=(12, 5))

        plt.subplot(1, 2, 1)
        plt.scatter(y_test_original[:, 0], y_pred_original[:, 0], alpha=0.5)
        plt.plot([y_test_original[:, 0].min(), y_test_original[:, 0].max()],
                 [y_test_original[:, 0].min(), y_test_original[:, 0].max()], 'r--', lw=2)
        plt.xlabel('Actual Magnitude')
        plt.ylabel('Predicted Magnitude')
        plt.title(f'Magnitude Prediction (R² = {mag_r2:.3f})')

        plt.subplot(1, 2, 2)
        plt.scatter(y_test_original[:, 1], y_pred_original[:, 1], alpha=0.5)
        plt.plot([y_test_original[:, 1].min(), y_test_original[:, 1].max()],
                 [y_test_original[:, 1].min(), y_test_original[:, 1].max()], 'r--', lw=2)
        plt.xlabel('Actual Depth')
        plt.ylabel('Predicted Depth')
        plt.title(f'Depth Prediction (R² = {depth_r2:.3f})')

        plt.tight_layout()
        plt.show()

    except ImportError:
        print("TensorFlow not available. Please install TensorFlow to run the neural network model.")
        print("Alternative: Use scikit-learn models")

        from sklearn.ensemble import RandomForestRegressor
        from sklearn.linear_model import LinearRegression


        print("Using Random Forest as alternative...")
        rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
        rf_model.fit(X_train, y_train)

        y_pred_rf = rf_model.predict(X_test)


        mag_mse = mean_squared_error(y_test[:, 0], y_pred_rf[:, 0])
        mag_r2 = r2_score(y_test[:, 0], y_pred_rf[:, 0])
        depth_mse = mean_squared_error(y_test[:, 1], y_pred_rf[:, 1])
        depth_r2 = r2_score(y_test[:, 1], y_pred_rf[:, 1])

        print(f"Random Forest Results:")
        print(f"Magnitude - MSE: {mag_mse:.4f}, R²: {mag_r2:.4f}")
        print(f"Depth - MSE: {depth_mse:.4f}, R²: {depth_r2:.4f}")
```

```
Loading data...
Original data shape: (23412, 21)
Columns: ['Date', 'Time', 'Latitude', 'Longitude', 'Type', 'Depth', 'Depth Error', 'Depth Seismic Stations', 'Magnitude', 'Magnitude
Data after column selection: (23412, 6)
         Date      Time  Latitude  Longitude  Depth  Magnitude
0  01/02/1965  13:44:18    19.246    145.616  131.6        6.0
1  01/04/1965  11:29:49     1.863    127.352   80.0        5.8
2  01/05/1965  18:05:58   -20.579   -173.972   20.0        6.2
3  01/08/1965  18:49:43   -59.076    -23.557   15.0        5.8
4  01/09/1965  13:32:50    11.938    126.427   15.0        5.8

Converting timestamps...
Data after cleaning: (23409, 5)
   Latitude  Longitude  Depth  Magnitude    Timestamp
0    19.246    145.616  131.6        6.0 -157630542.0
1     1.863    127.352   80.0        5.8 -157465811.0
2   -20.579   -173.972   20.0        6.2 -157355642.0
3   -59.076    -23.557   15.0        5.8 -157093817.0
4    11.938    126.427   15.0        5.8 -157026430.0

Creating visualization...
```




```
Preparing features and targets...
Features shape: (23409, 3)
Targets shape: (23409, 2)
Training shapes: X_train=(18727, 3), y_train=(18727, 2)
Testing shapes: X_test=(4682, 3), y_test=(4682, 2)

Building neural network model...
Model architecture:
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	512
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8,256
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 32)	2,080
dense_3 (Dense)	(None, 2)	66

```
Total params: 10,914 (42.63 KB)
Trainable params: 10,914 (42.63 KB)
Non-trainable params: 0 (0.00 B)

Training model...
Epoch 1/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.9790 - mae: 0.6530 - val_loss: 0.9697 - val_mae: 0.6465
Epoch 2/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.9832 - mae: 0.6500 - val_loss: 0.9564 - val_mae: 0.6358
Epoch 3/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 0.9239 - mae: 0.6336 - val loss: 0.9510 - val mae: 0.6392
```

```
Epoch 4/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.9247 - mae: 0.6374 - val_loss: 0.9444 - val_mae: 0.6332
Epoch 5/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 0.9068 - mae: 0.6274 - val_loss: 0.9367 - val_mae: 0.6362
Epoch 6/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.8908 - mae: 0.6247 - val_loss: 0.9276 - val_mae: 0.6351
Epoch 7/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - loss: 0.8958 - mae: 0.6264 - val_loss: 0.9342 - val_mae: 0.6433
Epoch 8/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.9005 - mae: 0.6258 - val_loss: 0.9272 - val_mae: 0.6224
Epoch 9/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 0.9225 - mae: 0.6321 - val_loss: 0.9140 - val_mae: 0.6278
Epoch 10/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - loss: 0.9109 - mae: 0.6297 - val_loss: 0.9121 - val_mae: 0.6248
Epoch 11/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - loss: 0.8959 - mae: 0.6223 - val_loss: 0.9208 - val_mae: 0.6156
Epoch 12/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - loss: 0.9045 - mae: 0.6256 - val_loss: 0.9149 - val_mae: 0.6200
Epoch 13/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.8806 - mae: 0.6172 - val_loss: 0.9029 - val_mae: 0.6228
Epoch 14/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.8885 - mae: 0.6214 - val_loss: 0.9007 - val_mae: 0.6191
Epoch 15/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 8s 13ms/step - loss: 0.8825 - mae: 0.6163 - val_loss: 0.8960 - val_mae: 0.6183
Epoch 16/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.8697 - mae: 0.6134 - val_loss: 0.8930 - val_mae: 0.6160
Epoch 17/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - loss: 0.8864 - mae: 0.6179 - val_loss: 0.8959 - val_mae: 0.6198
Epoch 18/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 0.8711 - mae: 0.6126 - val_loss: 0.8944 - val_mae: 0.6153
Epoch 19/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - loss: 0.8891 - mae: 0.6187 - val_loss: 0.8926 - val_mae: 0.6142
Epoch 20/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.8820 - mae: 0.6209 - val_loss: 0.8841 - val_mae: 0.6150
Epoch 21/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - loss: 0.8800 - mae: 0.6146 - val_loss: 0.8891 - val_mae: 0.6106
Epoch 22/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - loss: 0.8590 - mae: 0.6060 - val_loss: 0.8866 - val_mae: 0.6199
Epoch 23/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - loss: 0.8595 - mae: 0.6088 - val_loss: 0.8855 - val_mae: 0.6179
Epoch 24/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.8845 - mae: 0.6169 - val_loss: 0.8715 - val_mae: 0.6093
Epoch 25/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.8692 - mae: 0.6146 - val_loss: 0.8911 - val_mae: 0.6115
Epoch 26/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step - loss: 0.8814 - mae: 0.6169 - val_loss: 0.8772 - val_mae: 0.6042
Epoch 27/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 12s 13ms/step - loss: 0.8735 - mae: 0.6098 - val_loss: 0.8732 - val_mae: 0.6052
Epoch 28/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - loss: 0.8562 - mae: 0.6050 - val_loss: 0.8703 - val_mae: 0.6089
Epoch 29/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - loss: 0.8765 - mae: 0.6120 - val_loss: 0.8886 - val_mae: 0.6033
Epoch 30/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 7s 12ms/step - loss: 0.8580 - mae: 0.6069 - val_loss: 0.8635 - val_mae: 0.6126
Epoch 31/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 9s 10ms/step - loss: 0.8590 - mae: 0.6094 - val_loss: 0.8660 - val_mae: 0.6039
Epoch 32/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.8693 - mae: 0.6072 - val_loss: 0.8636 - val_mae: 0.6091
Epoch 33/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.8657 - mae: 0.6094 - val_loss: 0.8613 - val_mae: 0.6062
Epoch 34/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.8435 - mae: 0.6009 - val_loss: 0.8693 - val_mae: 0.6128
Epoch 35/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 6s 6ms/step - loss: 0.8365 - mae: 0.6000 - val_loss: 0.8696 - val_mae: 0.6042
Epoch 36/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.8359 - mae: 0.5985 - val_loss: 0.8712 - val_mae: 0.6092
Epoch 37/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.8471 - mae: 0.6052 - val_loss: 0.8528 - val_mae: 0.5887
Epoch 38/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.8574 - mae: 0.6042 - val_loss: 0.8628 - val_mae: 0.5993
Epoch 39/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.8582 - mae: 0.6039 - val_loss: 0.8486 - val_mae: 0.6011
Epoch 40/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 0.8479 - mae: 0.6033 - val_loss: 0.8658 - val_mae: 0.6022
Epoch 41/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - loss: 0.8378 - mae: 0.5972 - val_loss: 0.8433 - val_mae: 0.6006
Epoch 42/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 0.8322 - mae: 0.5956 - val_loss: 0.8471 - val_mae: 0.6007
Epoch 43/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.8231 - mae: 0.5966 - val_loss: 0.8542 - val_mae: 0.5993
Epoch 44/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.8409 - mae: 0.5984 - val_loss: 0.8510 - val_mae: 0.5957
Epoch 45/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - loss: 0.8509 - mae: 0.6026 - val_loss: 0.8455 - val_mae: 0.5945
Epoch 46/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.8567 - mae: 0.6086 - val_loss: 0.8392 - val_mae: 0.5970
Epoch 47/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.8411 - mae: 0.6020 - val_loss: 0.8444 - val_mae: 0.6090
Epoch 48/50
586/586 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.8379 - mae: 0.6010 - val_loss: 0.8374 - val_mae: 0.6006
```