Problem 1:

Once again consider the plane-strain compression problem shown in "data/plane-strain.png". In this problem you are given node features for 100 parts. These node features have been extracted by processing each part shape using a neural network. You will train a neural network to von Mises stress at each node given its 60 features. Then you will analyze R^2 for the training and testing data, both for the full dataset and for individual shapes within each dataset.

Summary of deliverables

- Neural network model definition
- Training function
- Training loss curve
- Overall R^2 on training and testing data
- Predicted-vs-actual plots for training and testing data
- Histograms of R^2 distributions on training and testing shapes
- Median R^2 values across training and testing shapes

In [75]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score

import torch
from torch import nn, optim

def plot_shape(dataset, index, model=None, lims=None):
    x = dataset["coordinates"][index][:,0]
    y = dataset["coordinates"][index][:,1]

    if model is None:
        c = dataset["stress"][index]
    else:
        c = model(torch.tensor(dataset["features"][index])).detach().numpy().flatte

    if lims is None:
        lims = [min(c),max(c)]

    plt.scatter(x,y,s=5,c=c,cmap="jet",vmin=lims[0],vmax=lims[1])
    plt.colorbar(orientation="horizontal", shrink=.75, pad=0,ticks=lims)
    plt.axis("off")
    plt.axis("equal")

def plot_shape_comparison(dataset, index, model, title=""):
    plt.figure(figsize=[6,3.2], dpi=120)
    plt.subplot(1,2,1)
    plot_shape(dataset,index)
    plt.title("Ground Truth",fontsize=9,y=.96)
```

```
    plt.subplot(1,2,2)
    c = dataset["stress"][index]
    plot_shape(dataset, index, model, lims = [min(c), max(c)])
    plt.title("Prediction",fontsize=9,y=.96)
    plt.suptitle(title)
    plt.show()

def load_dataset(path):
    dataset = np.load(path)
    coordinates = []
    features = []
    stress = []
    N = np.max(dataset[:,0].astype(int)) + 1
    split = int(N*.8)
    for i in range(N):
        idx = dataset[:,0].astype(int) == i
        data = dataset[idx,:]
        coordinates.append(data[:,1:3])
        features.append(data[:,3:-1])
        stress.append(data[:,-1])
    dataset_train = dict(coordinates=coordinates[:split], features=features[:split]
    dataset_test = dict(coordinates=coordinates[split:], features=features[split:],
    X_train, X_test = np.concatenate(features[:split], axis=0), np.concatenate(feat
    y_train, y_test = np.concatenate(stress[:split], axis=0), np.concatenate(stress
    return dataset_train, dataset_test, X_train, X_test, y_train, y_test

def get_shape(dataset,index):
    X = torch.tensor(dataset["features"][index])
    Y = torch.tensor(dataset["stress"][index].reshape(-1,1))
    return X, Y

def plot_r2_distribution(r2s, title=""):
    plt.figure(dpi=120,figsize=(6,4))
    plt.hist(r2s, bins=10)
    plt.xlabel("$R^2$")
    plt.ylabel("Number of shapes")
    plt.title(title)
    plt.show()
```

Loading the data

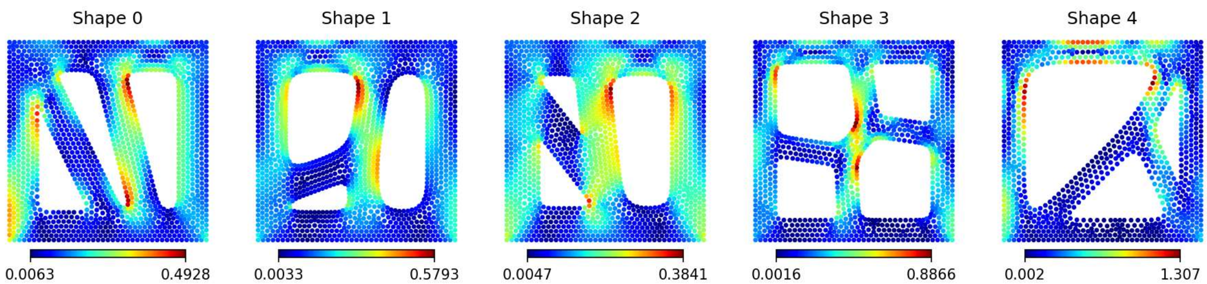
First, complete the code below to load the data and plot the von Mises stress fields for a few shapes.
You'll need to input the path of the data file, the rest is done for you.

All training node features and outputs are in `X_train` and `y_train`, respectively.
Testing nodes are in `X_test`, `y_test`.

`dataset_train` and `dataset_test` contain more detailed information such as node coordinates, and they are separated by shape.
Get features and outputs for a shape by calling `get_shape(dataset,index)`.

`N_train` and `N_test` are the number of training and testing shapes in each of these datasets.

In [76]:
```
data_path = r"C:\Users\zsqu4\Desktop\ML HW\ML-for-engineers\HW9\data\stress_nodal_f
dataset_train, dataset_test, X_train, X_test, y_train, y_test = load_dataset(data_p
N_train = len(dataset_train["stress"])
N_test = len(dataset_test["stress"])

plt.figure(figsize=[15,3.2], dpi=150)
for i in range(5):
    plt.subplot(1,5,i+1)
    plot_shape(dataset_train,i)
    plt.title(f"Shape {i}")
plt.show()
```



Neural network to predict stress

Create a PyTorch neural network class `StressPredictor` below. This should be an MLP with 60 inputs (the given features) and 1 output (stress). The hidden layer sizes and activations are up to you.

In [ ]:
```
import torch.nn.functional as F
class StressPredictor(nn.Module):
    # YOUR CODE GOES HERE
    def __init__(self):
        super().__init__()
        # YOUR CODE GOES HERE
        # self.layers = nn.ModuleList()
        # for hidden_size in hidden_layer_sizes:
        #     self.layers.append(nn.Linear(input_size, hidden_size))
        #     input_size = hidden_size
        # self.layers.append(nn.Linear(hidden_layer_sizes[-1], output_size))
        # self.act1 = nn.ReLU()
        input_size = 60
        output_size = 1

        self.model = nn.Sequential(
            nn.Linear(input_size,120),
            nn.Linear(120,240),
            nn.ReLU(),
            nn.Linear(240,120),
            nn.ReLU(),
            nn.Linear(120,output_size)
```

```
        )


    def forward(self, xy):
        # YOUR CODE GOES HERE
        # for layer in self.layers[:-1]:
        #     self.act1(xy)
        # xy = self.layers[-1](xy)
        # return xy
        return self.model(xy)
```

Training function

Below, you should define a function `train(model, dataset, lr, epochs)` that will train `model` on the data in `dataset` with the Adam optimizer for `epochs` epochs with a learning rate of `lr`.

Because there are so many total nodes, you should treat each shape as a batch of nodes -- each epoch of training will require you to loop through each shape in the dataset in a random order, performing a step of gradient descent for each shape encountered. Your function should automatically generate a plot of the loss curve on training data.

- You can use the provided `get_shape` to access feature and output tensors for each shape.
- Use MSE as a your loss function.
- Look into `np.random.permutation()` for generating a random index order

In [186…

```
import numpy as np
import torch
import matplotlib.pyplot as plt

def train(model, dataset, lr, epochs):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    lossfun = torch.nn.MSELoss()
    loss_values = []

    for epoch in range(epochs):
        epoch_loss = 0.0
        indices = np.random.permutation(len(dataset['features']))

        for idx in indices:
            features, stress = get_shape(dataset, idx)

            optimizer.zero_grad()
            pred = model(features)

            loss = lossfun(pred, stress)
            loss.backward()
            optimizer.step()
```

```
            epoch_loss += loss.item()

        avg_epoch_loss = epoch_loss / len(dataset['features'])
        loss_values.append(avg_epoch_loss)

        if (epoch + 1) % 10 == 0:
            print(f'Epoch [{epoch+1}/{epochs}], Loss: {avg_epoch_loss:.4f}')

    # Plotting the loss curve
    plt.plot(range(epochs), loss_values, label='Training Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training Loss Curve')
    plt.legend()
    plt.show()
```

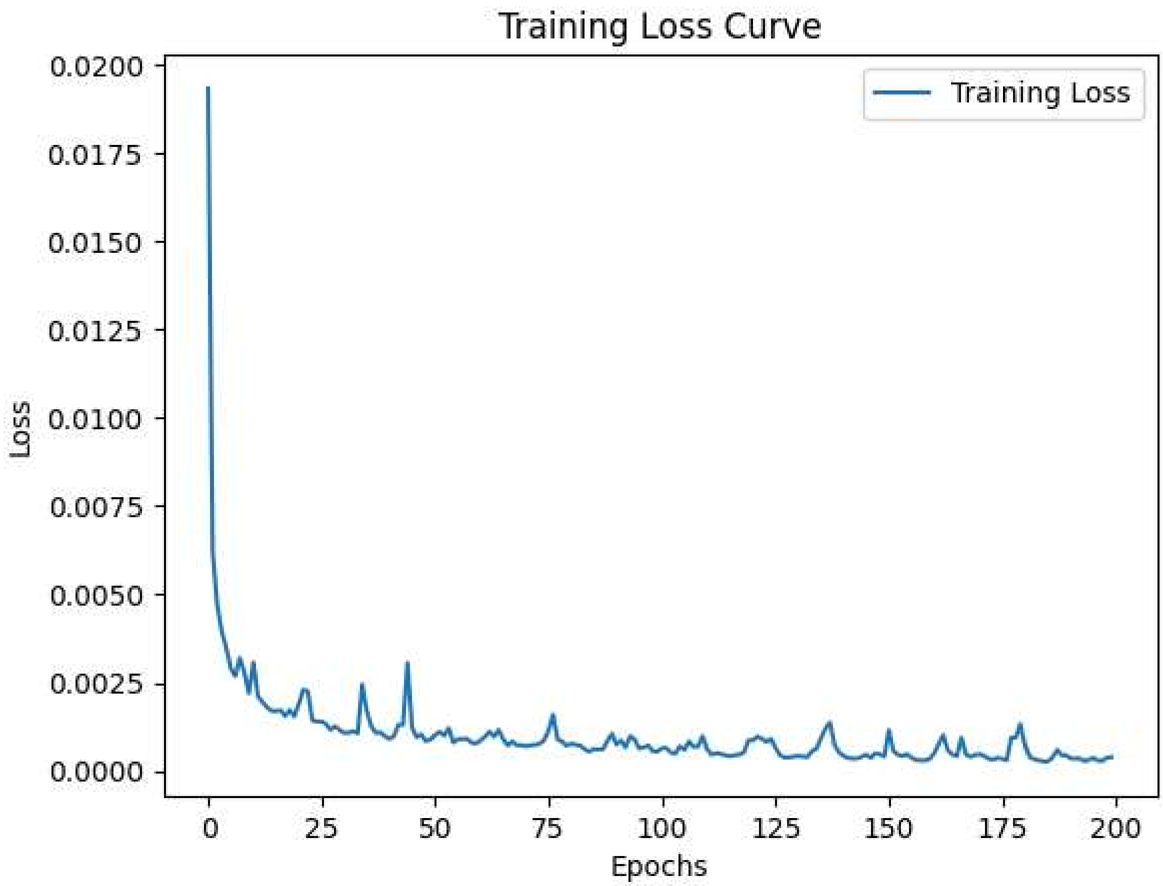
Training your Neural Network

Now, create your neural network model and run your train function on the training dataset `dataset_train`.
Determining the right number of epochs and learning rate are up to you. The training loss curve should be shown.

In [187…

```
# YOUR CODE GOES HERE
lr = 0.0005
epochs = 200
model = StressPredictor()
print(model)
dataset = dataset_train
train(model,dataset,lr,epochs)
```

```
StressPredictor(
  (model): Sequential(
    (0): Linear(in_features=60, out_features=120, bias=True)
    (1): Linear(in_features=120, out_features=240, bias=True)
    (2): ReLU()
    (3): Linear(in_features=240, out_features=120, bias=True)
    (4): ReLU()
    (5): Linear(in_features=120, out_features=1, bias=True)
  )
)
Epoch [10/200], Loss: 0.0022
Epoch [20/200], Loss: 0.0015
Epoch [30/200], Loss: 0.0012
Epoch [40/200], Loss: 0.0010
Epoch [50/200], Loss: 0.0009
Epoch [60/200], Loss: 0.0008
Epoch [70/200], Loss: 0.0007
Epoch [80/200], Loss: 0.0007
Epoch [90/200], Loss: 0.0010
Epoch [100/200], Loss: 0.0005
Epoch [110/200], Loss: 0.0010
Epoch [120/200], Loss: 0.0009
Epoch [130/200], Loss: 0.0004
Epoch [140/200], Loss: 0.0005
Epoch [150/200], Loss: 0.0004
Epoch [160/200], Loss: 0.0003
Epoch [170/200], Loss: 0.0004
Epoch [180/200], Loss: 0.0013
Epoch [190/200], Loss: 0.0004
Epoch [200/200], Loss: 0.0004
```



R^2 Score

Compute the R^2 Score on the training dataset. You will have to convert between tensors and arrays versions to use sklearn functions, or you can write your own function.

In [191…

```
from sklearn.metrics import r2_score
import torch
import numpy as np

model.eval()
predictions = []
true_labels = []

with torch.no_grad():

        features = torch.from_numpy(X_train).float()
        preds = model(features).numpy().flatten()
        print(preds)

r2 = r2_score(y_train, preds)
print(f"R^2 Score: {r2:.4f}")
```

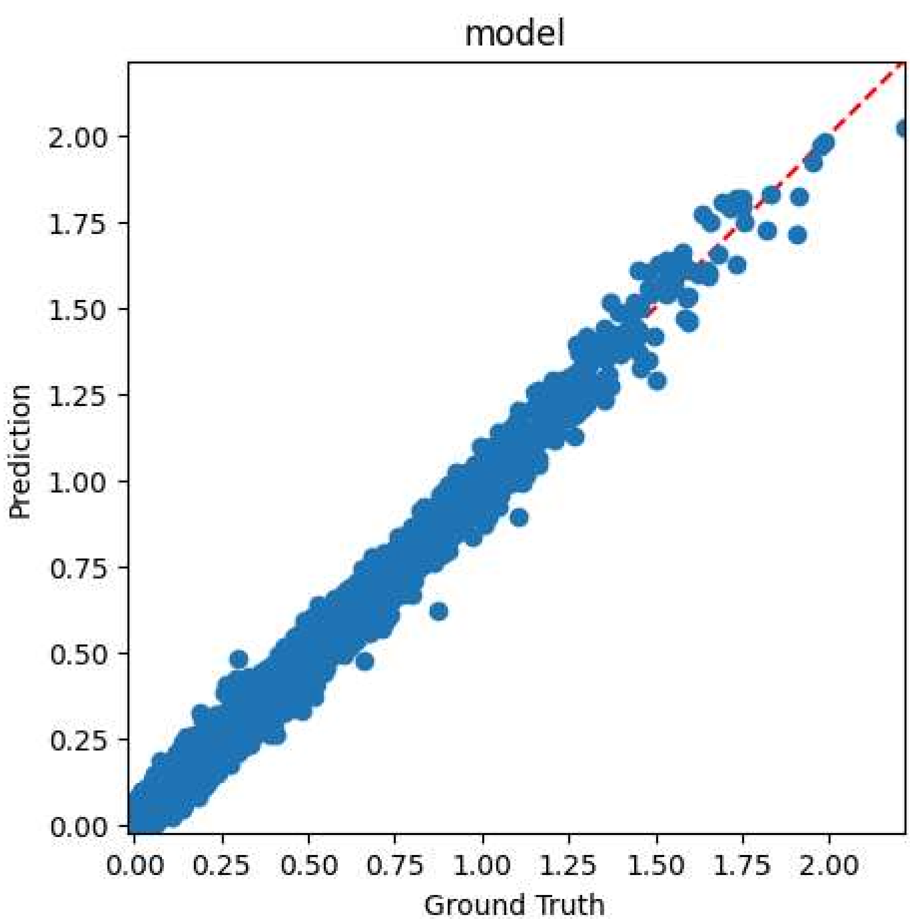
```
[0.10123041 0.09175783 0.19859877 ... 0.18281814 0.07602369 0.15722239]
R^2 Score: 0.9866
```

R^2 Plots

Now, generate predicted-vs-actual plots that display both data and a theoretical best fit line. Make 2 such plots - one for training data and one for testing.

In [192…

```
# YOUR CODE GOES HERE
def plot_r2(gt, pred, title):
    plt.figure(figsize=[5,5])

    plt.plot([-1000,1000], [-1000,1000],"r--")
    plt.plot(gt,pred,'o')
    all = np.concatenate([gt, pred])
    plt.xlim(np.min(all), np.max(all))
    plt.ylim(np.min(all), np.max(all))
    plt.xlabel("Ground Truth")
    plt.ylabel("Prediction")
    plt.title(title)
    plt.show()

plot_r2(y_train,preds,"model")
```



Individual Shape R^2

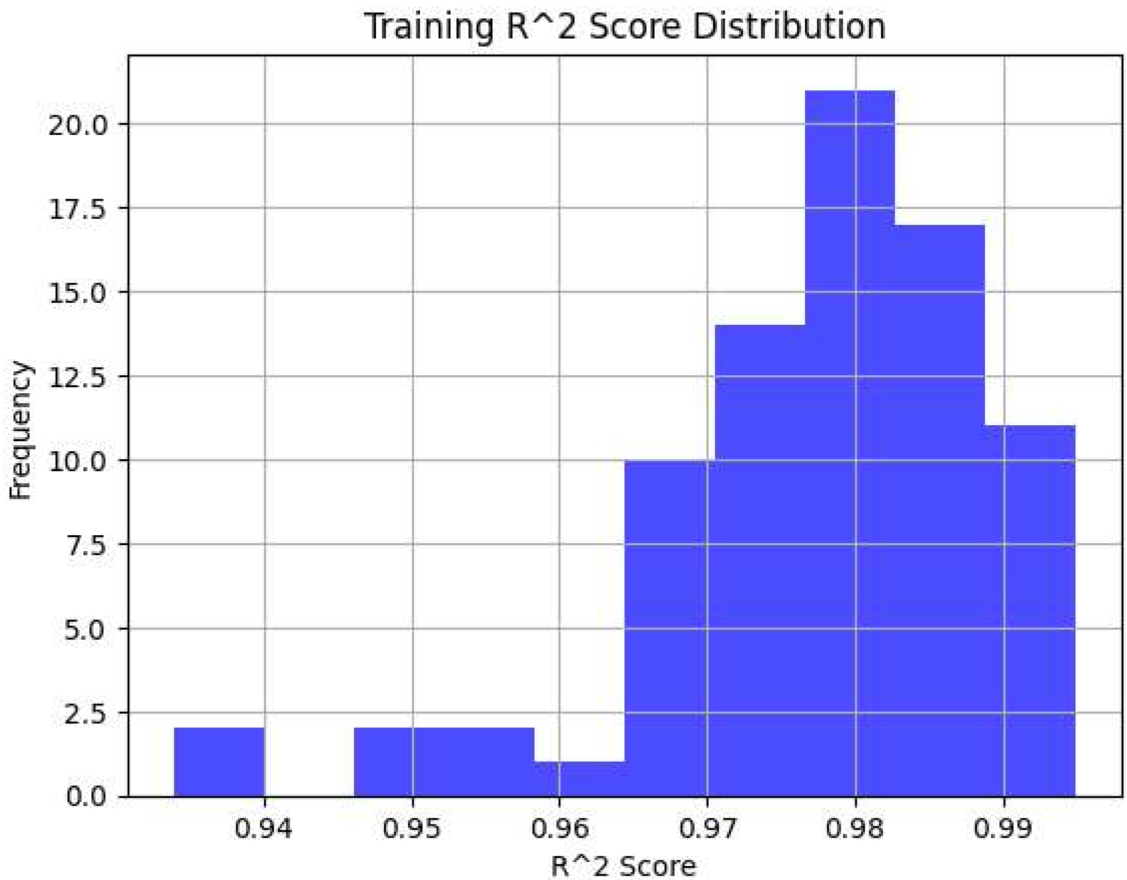
Because we have a unique problem where groups of nodes in a dataset form a single shape, we can compute an R^2 score for an individual shape. For each shape in the training set, compute an R^2 score. Then create a histogram of the values with the function `plot_r2_hist(r2s)`. Repeat for the testing set.

Report the median R^2 score across all training shapes, and the median across all testing shapes.

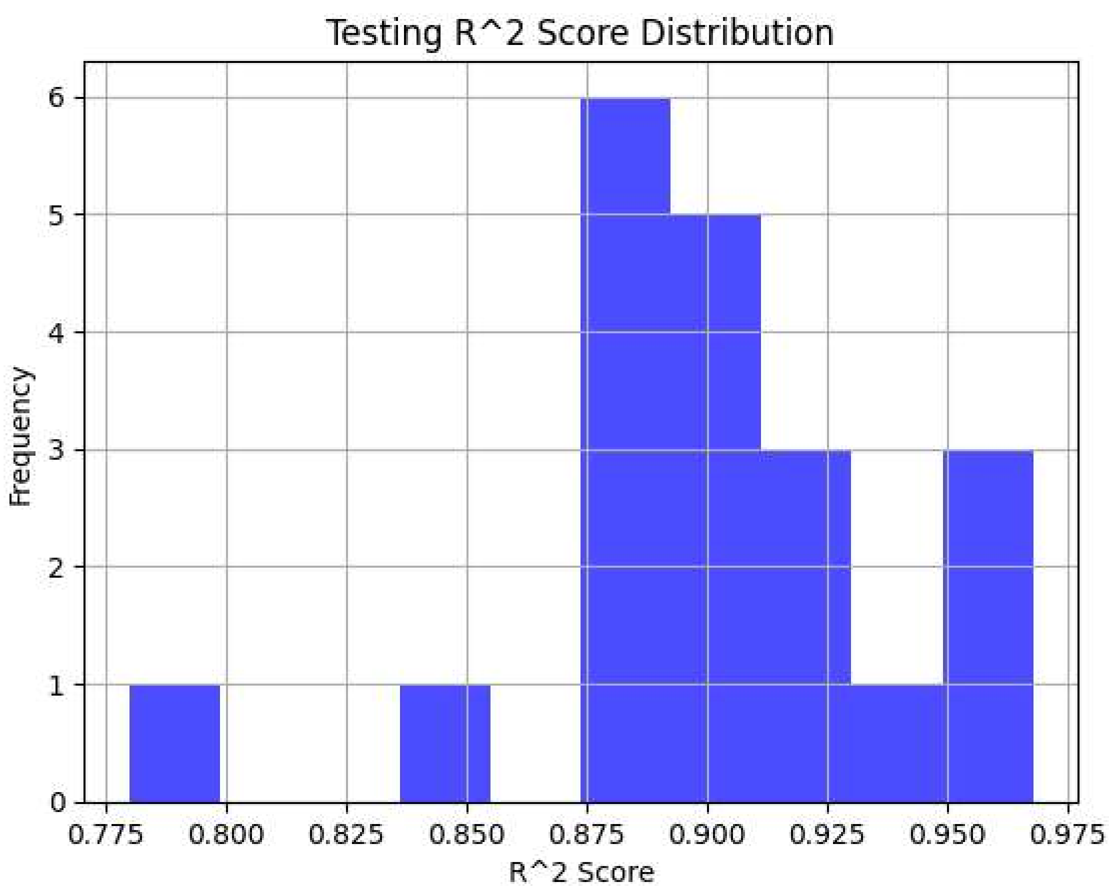
If your test median is below 0.85, try and tune your network size/training hyperparameters until it reaches this threshold.

In [ ]:
```
# YOUR CODE GOES HERE
def evaluate_shapes(model, dataset, dataset_name="Training"):
    model.eval()
    r2_scores = []

    with torch.no_grad():
        for idx in range(len(dataset['features'])):
            features, labels = get_shape(dataset, idx)


            predictions = model(features).squeeze(-1).numpy()
            labels = labels.squeeze(-1).numpy()


            r2 = r2_score(labels, predictions)
            r2_scores.append(r2)


    median_r2 = np.median(r2_scores)
    print(f"{dataset_name} Median R^2 Score: {median_r2:.4f}")


    plt.hist(r2_scores, bins=10, alpha=0.7, color='b')
    plt.xlabel('R^2 Score')
    plt.ylabel('Frequency')
    plt.title(f'{dataset_name} R^2 Score Distribution')
    plt.grid(True)
    plt.show()

    return median_r2

median_r2_train = evaluate_shapes(model, dataset_train, dataset_name="Training")
median_r2_test = evaluate_shapes(model, dataset_test, dataset_name="Testing")
```

```
Training Median R^2 Score: 0.9787
```




```
Testing Median R^2 Score: 0.8966
```



In [ ]:

```

```