13-L1 Problem 1

We return to the plane-strain compression problem where the goal is to predict von Mises stress at a node given a set of its features.

Now, you will look at ensemble methods in sklearn to determine how well they perform in this context.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor

from sklearn.metrics import mean_squared_error
from sklearn.ensemble import StackingRegressor


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.predict(dataset["features"][index])

    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], stress=stre
    dataset_test = dict(coordinates=coordinates[split:], features=features[split:], stress=stres
    X_train, X_test = np.concatenate(features[:split], axis=0), np.concatenate(features[split:],
    y_train, y_test = np.concatenate(stress[:split], axis=0), np.concatenate(stress[split:], axi
    return dataset_train, dataset_test, X_train, X_test, y_train, y_test

def get_shape(dataset,index):
    X = dataset["features"][index]
    y = dataset["stress"][index]
    return X, y
```

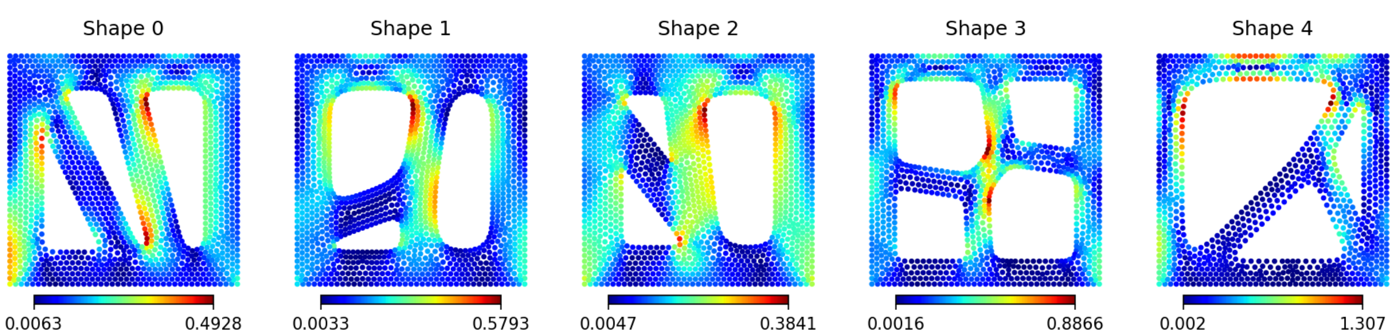
Loading the data

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 [2]:
```
# YOU MAY NEED TO EDIT data_path
data_path = "stress_nodal_features.npy"
dataset_train, dataset_test, X_train, X_test, y_train, y_test = load_dataset(data_path)
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()
```



StackingRegressor

A StackingRegressor consists of N fitted regression models, which it evaluates to make N predictions. Then, these predictions are fed into another regression model, which makes a final prediction of the target.

List of Regressors

First, initialize 4 regression models:

1. Linear Regression
2. Decision Tree regression, max depth 4
3. Decision Tree regression, max depth 8
4. Decision Tree regression, max depth 12

Then, store these in a list called `models`.

In [3]:
```
# YOUR CODE GOES HERE
# Define models, and put in a list called 'models'
models = []

model1 = LinearRegression()
model2 = DecisionTreeRegressor(max_depth=4)
model3 = DecisionTreeRegressor(max_depth=8)
model4 = DecisionTreeRegressor(max_depth=12)

models.append(model1)
models.append(model2)
models.append(model3)
models.append(model4)

named_models = [(f"Model {i+1}", model) for i, model in enumerate(models)]
print(*named_models, sep="\n")
```

```
('Model 1', LinearRegression())
('Model 2', DecisionTreeRegressor(max_depth=4))
('Model 3', DecisionTreeRegressor(max_depth=8))
('Model 4', DecisionTreeRegressor(max_depth=12))
```

Final Regressor

Now make one more regressor, which will take as input the other four predictions, and combine them to make an improved prediction.

This can be another linear regression model. Call it `final_model`.

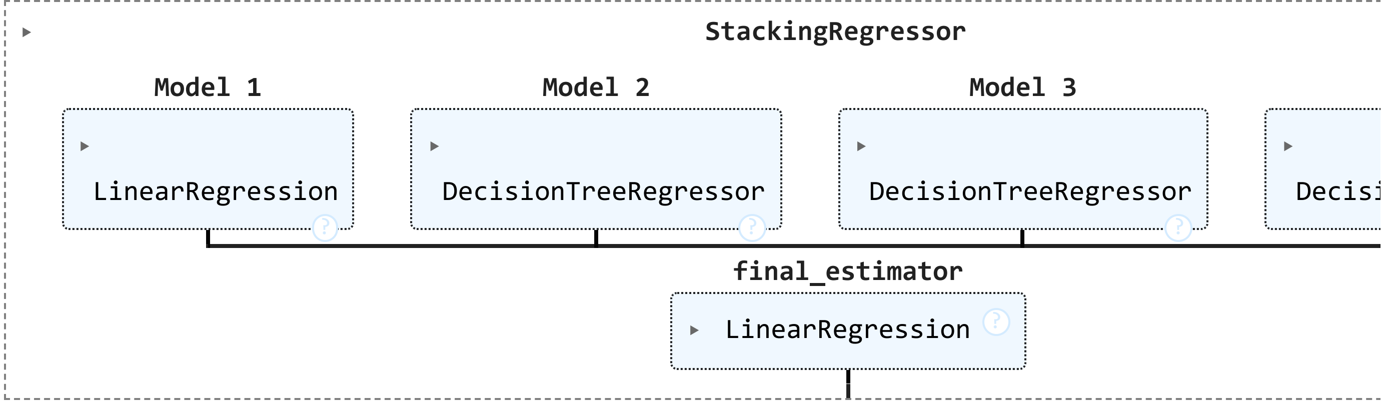
In [4]:
```
# YOUR CODE GOES HERE

final_model = LinearRegression()
```

Creating and training the StackingRegressor

Finally, we can combine all of our models into a StackingRegressor model. We fit this just as we would fit any sklearn model. Because of the size of the dataset, this may take a few minutes.

In [5]:
```
srm = StackingRegressor(named_models, final_model, verbose=True)
srm.fit(X_train, y_train)
```

Out[5]:



Performance of the model

Now we can investigate the performance of the model on test data, compared to constituent models. First, let's look at the performance of each individual model of our model `srm`. These can be accessed via `srm.estimators_`.

In [6]:

```
for i, estimator in enumerate(srm.estimators_):
    print(f"\n{named_models[i][0]}:")
    train_err = mean_squared_error(y_train, estimator.predict(X_train))
    test_err = mean_squared_error(y_test, estimator.predict(X_test))
    print(f"Training MSE: {train_err:.2e}")
    print(f" Testing MSE: {test_err:.2e}")
```

```
Model 1:
Training MSE: 8.11e-03
 Testing MSE: 9.78e-03

Model 2:
Training MSE: 1.26e-02
 Testing MSE: 1.51e-02

Model 3:
Training MSE: 7.56e-03
 Testing MSE: 1.03e-02

Model 4:
Training MSE: 3.75e-03
 Testing MSE: 8.40e-03
```

Stacking Regressor MSE on Test Data

Now compute the MSE of `srm` on training and testing data.

Note how the results, particularly on test data, compare to the individual models.

In [7]:

```
# YOUR CODE GOES HERE
train_err = mean_squared_error(y_train, srm.predict(X_train))
test_err = mean_squared_error(y_test, srm.predict(X_test))

print(f"\nStacking Regressor:")
print(f"Training MSE: {train_err:.2e}")
print(f"Testing MSE: {test_err:.2e}")
```

```
Stacking Regressor:
Training MSE: 4.11e-03
Testing MSE: 6.61e-03
```

M13-L2 Problem 1

Once more, we will study the stress prediction problem, this time using XGBoost, a very powerful boosting method.

In [1]:

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

import xgboost as xgb
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error


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.predict(dataset["features"][index])

    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], stress=stre
    dataset_test = dict(coordinates=coordinates[split:], features=features[split:], stress=stres
```

```
    X_train, X_test = np.concatenate(features[:split], axis=0), np.concatenate(features[split:], 
    y_train, y_test = np.concatenate(stress[:split], axis=0), np.concatenate(stress[split:], axi
    return dataset_train, dataset_test, X_train, X_test, y_train, y_test

def get_shape(dataset,index):
    X = dataset["features"][index]
    y = dataset["stress"][index]
    return X, y

def eval_model(model, verbose=False):
    pred_train = model.predict(X_train)
    pred_test = model.predict(X_test)
    mse_train = mean_squared_error(y_train, pred_train)
    mse_test = mean_squared_error(y_test, pred_test)
    if verbose:
        print(f"Train MSE = {mse_train:.2e}")
        print(f"Test  MSE = {mse_test:.2e}")
    return mse_train, mse_test
```

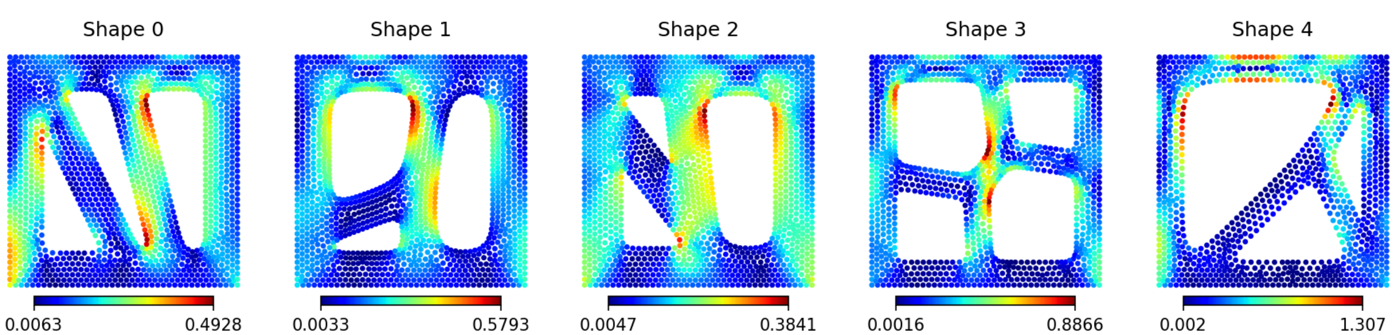
Loading the data

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 [3]:
```
# YOU MAY NEED TO EDIT data_path
data_path = "stress_nodal_features.npy"
dataset_train, dataset_test, X_train, X_test, y_train, y_test = load_dataset(data_path)
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()
```



XGBoost Regressor

XGBoost models, like `XGBRegressor` here, can be used much like sklearn models.

First, define an instance of `XGBRegressor` with the desired parameters; then, fit the model with `model.fit`. You can evaluate a fitted model with `model.predict`.

The provided function `mse_train, mse_test = eval_model(model)` to get MSE values on the train and test datasets.

In [5]:
```
eta = 0.8
depth = 9

params = dict(
    eta = eta,
    max_depth = depth,
)

model = XGBRegressor(objective ='reg:squarederror', seed = 123, n_estimators = 10, **params)
model.fit(X_train, y_train)

mse_train, mse_test = eval_model(model)
print("  eta | depth | Train MSE | Test MSE")
print(f"  {eta:.1f} | {depth:>2d} | {mse_train:.2e} | {mse_test:.2e}")
```

```
 eta | depth | Train MSE | Test MSE
 0.8 |   9 | 2.19e-03 | 6.10e-03
```

Parametric study

Now let's examine the effects of varying the parameters `eta` and `max_depth`, keeping `n_estimators` as 10. For every combination of eta in [0.1, 0.3, 0.5, 0.7] and `max_depth` in [5, 10, 15, 20], train an XGB regressor and report the train and test MSE values.

Which combination has the best performance on testing data?

In [6]:
```
# YOUR CODE GOES HERE
eta = [0.1, 0.3, 0.5, 0.7]
max_depth = [5, 10, 15, 20]

for e in eta:
    for d in max_depth:
        params = dict(eta = e, max_depth = d)
        model = XGBRegressor(objective ='reg:squarederror', seed = 123, n_estimators = 10, **para
        model.fit(X_train, y_train)

        mse_train, mse_test = eval_model(model)
        print("  eta | depth | Train MSE | Test MSE")
        print(f"  {e:.1f} | {d:>2d} | {mse_train:.2e} | {mse_test:.2e}")
```

```
eta | depth | Train MSE | Test MSE
0.1 |  5 | 1.05e-02 | 1.28e-02
eta | depth | Train MSE | Test MSE
0.1 | 10 | 5.93e-03 | 8.93e-03
eta | depth | Train MSE | Test MSE
0.1 | 15 | 3.86e-03 | 7.93e-03
eta | depth | Train MSE | Test MSE
0.1 | 20 | 3.35e-03 | 7.98e-03
eta | depth | Train MSE | Test MSE
0.3 |  5 | 5.43e-03 | 7.05e-03
eta | depth | Train MSE | Test MSE
0.3 | 10 | 1.57e-03 | 4.58e-03
eta | depth | Train MSE | Test MSE
0.3 | 15 | 2.79e-04 | 4.59e-03
eta | depth | Train MSE | Test MSE
0.3 | 20 | 8.02e-05 | 4.73e-03
eta | depth | Train MSE | Test MSE
0.5 |  5 | 4.60e-03 | 6.49e-03
eta | depth | Train MSE | Test MSE
0.5 | 10 | 1.36e-03 | 4.80e-03
eta | depth | Train MSE | Test MSE
0.5 | 15 | 1.47e-04 | 5.01e-03
eta | depth | Train MSE | Test MSE
0.5 | 20 | 8.30e-06 | 5.20e-03
eta | depth | Train MSE | Test MSE
0.7 |  5 | 4.83e-03 | 7.03e-03
eta | depth | Train MSE | Test MSE
0.7 | 10 | 1.44e-03 | 5.54e-03
eta | depth | Train MSE | Test MSE
0.7 | 15 | 1.60e-04 | 5.82e-03
eta | depth | Train MSE | Test MSE
0.7 | 20 | 9.05e-06 | 6.02e-03
```

The best model is eta = 0.3 and depth = 10 for the testing data.