# Assignment 6.1 - Trees

Please submit your solution of this notebook in the Whiteboard at the corresponding Assignment entry as .ipynb-file and as .pdf.

Please do **NOT** rename the file!

### State both names of your group members here:

[Rashid Harvey and S M Shameem Ahmed]

In [ ]:

---

## Grading Info/Details - Assignment 6.1:

The assignment will be graded semi-automatically, which means that your code will be tested against a set of predefined test cases and qualitatively assessed by a human. This will speed up the grading process for us.

- For passing the test scripts:
  - Please make sure to **NOT** alter predefined class or function names, as this would lead to failing of the test scripts.
  - Please do **NOT** rename the files before uploading to the Whiteboard!
- **(RESULT)** tags indicate checkpoints that will be specifically assessed by a human.
- You will pass the assignment if you pass the majority of test cases and we can at least confirm effort regarding the **(RESULT)**-tagged checkpoints per task.

---

In [ ]:

## Task 6.1.1 - Regression Trees

- Implement the Regression Tree Class from scratch using only `NumPy` . **(RESULT)**
- Run your implementation on the synthetic regression dataset provided. **(RESULT)**

In [3]:
```
import numpy as np

def generate_regression_data(n_samples=1000, n_features=8, noise=0.1, random_state=
    """Generate synthetic regression data similar to California housing."""
    np.random.seed(random_state)
```

```
    X = np.random.randn(n_samples, n_features)

    # Create target with non-linear relationships
    y = (2.5 * X[:, 0] +                    # Linear relationship
         1.8 * X[:, 1] ** 2 +               # Quadratic (non-linear)
         -1.2 * X[:, 2] * X[:, 3] +         # Interaction between features
         0.5 * np.sin(5 * X[:, 4]) +        # Sinusoidal (periodic pattern)
         0.8 * X[:, 5] +                    # Linear
         -0.3 * X[:, 6] ** 3 +              # Cubic (strong non-linearity)
         1.5 * X[:, 7])                     # Linear

    # Add noise
    y += noise * np.random.randn(n_samples)

    # Scale to reasonable range
    y = (y - y.min()) / (y.max() - y.min()) * 4 + 1

    return X, y
```

In [4]:

```
class RegressionTree:
    """A decision tree for regression using numpy."""

    def __init__(self):
        pass

    def fit(self, X, y):
        """Build the regression tree."""
        self.tree = self._build_tree(X, y, depth=0)

    def _build_tree(self, X, y, depth):
        """Recursively build the tree."""
        if (
            # depth >= 5 or
            len(y) <= 10):
            return np.mean(y)

        best_q = float('inf')
        best_j = None
        best_z = None

        for j in range(len(X[0])):
            _sorted = sorted(X[:,j])
            # print(X[:,j].shape)

            sorted_X = X[X[:,j].argsort()]

            for x,x_ in zip(_sorted, _sorted[1:]):
                z = (x + x_) / 2
                y_1 = y[np.where(sorted_X[:,j] <= z)]
                y_2 = y[np.where(sorted_X[:,j] > z)]
                c_1 = np.mean(y_1)
                c_2 = np.mean(y_2)
                y_1 -= c_1
                y_2 -= c_2
                q = np.sum(y_1**2) + np.sum(y_2**2)
```

```
                # print(np.sum(y_1**2), np.sum(y_2**2), q)
                if q < best_q:
                    best_q = q
                    best_j = j
                    best_z = z
        # print("Best feature:", best_j, "Best split:", best_z, "Best error:", best_

        _where_1 = np.where(X[:,best_j] <= best_z)
        _where_2 = np.where(X[:,best_j] > best_z)

        return (
            (best_j, best_z),
            self._build_tree(X[_where_1], y[_where_1], depth + 1),
            self._build_tree(X[_where_2], y[_where_2], depth + 1)
        )


    def predict(self, X):
        """Make predictions for X."""
        if not hasattr(self, 'tree'):
            raise Exception("The tree has not been trained yet. Call 'fit' first.")

        return np.array([self._predict_single(x, self.tree) for x in X])

    def _predict_single(self, x, tree):
        """Recursively predict a single instance."""
        if not isinstance(tree, tuple):
            return tree

        (j, z), t1, t2 = tree

        if x[j] <= z:
            return self._predict_single(x, t1)
        else:
            return self._predict_single(x, t2)
```

In [5]:
```
X, y = generate_regression_data(n_samples=1000, n_features=8, noise=0.1, random_sta

test_samples = 1000
test_X, test_y = generate_regression_data(n_samples=test_samples, n_features=8, noi

# print(X.shape, Y.shape)

tree = RegressionTree()

tree.fit(X, y)

print("Finished training.")

number_of_splits = 10
n = test_samples // number_of_splits

mses = np.zeros(number_of_splits)
```

```
for i in range(number_of_splits):
    predictions = tree.predict(test_X[i*n:(i+1)*n])
    q = np.mean((predictions - test_y[i*n:(i+1)*n]) ** 2)
    mses[i] = q
    # print(f"Test MSE ({i}): {mse}")

print("Average Test MSE:", np.mean(mses))
print("Test MSE Std Dev:", np.std(mses))
```

```
Finished training.
Average Test MSE: 0.5002588188605095
Test MSE Std Dev: 0.057910239338356054
```

# Task 6.1.2 - Bagging

- Implement Bagging using only `NumPy`. **(RESULT)**
- Compare the results between the bagged run of your `RegressionTree` class on the synthetic dataset. **(RESULT)**

In [9]:
```
class BaggingRegressor:
    """Bagging ensemble for regression trees."""

    def __init__(self, m=20, B=100):
        self.m = m
        self.B = B

    def fit(self, X, y):
        """Fit the bagging ensemble."""
        self.trees = []
        for _ in range(self.B):
            indices = np.random.choice(len(X), size=self.m, replace=True)
            X_b = X[indices]
            y_b = y[indices]
            tree = RegressionTree()
            tree.fit(X_b, y_b)
            self.trees.append(tree)

    def predict(self, X):
        """Make predictions by averaging all trees."""
        predictions = np.zeros((len(X), self.B))
        for b, tree in enumerate(self.trees):
            predictions[:, b] = tree.predict(X)
        return np.mean(predictions, axis=1)
```

In [10]:
```
X, y = generate_regression_data(n_samples=1000, n_features=8, noise=0.1, random_sta

test_samples = 1000
test_X, test_y = generate_regression_data(n_samples=test_samples, n_features=8, noi

# print(X.shape, Y.shape)

tree = BaggingRegressor(m=40, B=100)
```

```
tree.fit(X, y)

print("Finished training.")

number_of_splits = 10
n = test_samples // number_of_splits

mses = np.zeros(number_of_splits)

for i in range(number_of_splits):
    predictions = tree.predict(test_X[i*n:(i+1)*n])
    q = np.mean((predictions - test_y[i*n:(i+1)*n]) ** 2)
    mses[i] = q
    # print(f"Test MSE ({i}): {mse}")

print("Average Test MSE:", np.mean(mses))
print("Test MSE Std Dev:", np.std(mses))
```

```
Finished training.
Average Test MSE: 0.49242280935031585
Test MSE Std Dev: 0.04738304434502272
```

The regression tree and the bagging perform quite bad due to the badly-generalizing stopping condition.

Also, z is often not in the middle of the feature space and therefore the tree is not very dense. Maybe changing the loss to Q_1^2 + Q_2^2 could improve results.

However, we see that the bagging improves both the error as well as the standard deviation of the error.

# Congratz, you made it! :)