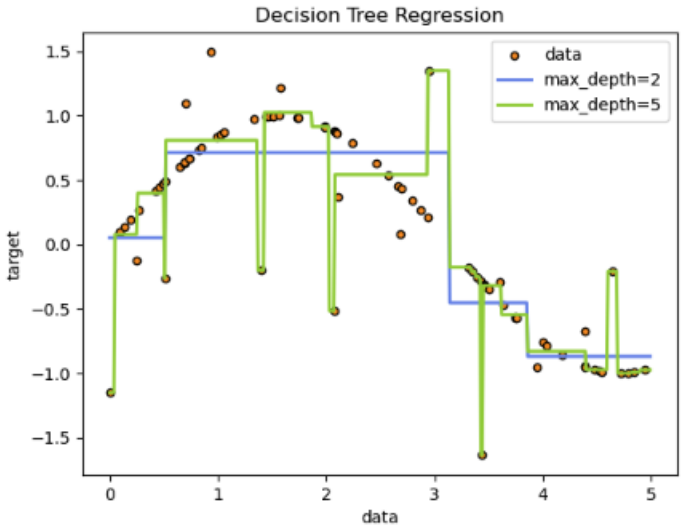
의사결정나무 보고서

> 19011810 이용빈

1. Desicion Tree 결정나무

makedata 함수를 이용하여 임의의 학습 데이터를 생성하였습니다. 생성한 데이터를 회귀 결정 트리 모델을 사용하여 학습하고, 결과를 예측하는 과정으로 이루어집니다.

sklearn에서 구현된 Descision Tree와, 코드로 구현한 Decision Tree의 차이를 비교합니다.



Code Review

아래 코드는 Decision Tree Regression을 구현한 코드입니다. 코드에서는 Decision Tree를 TreeNode class를 이용하여 구현하였습니다. 또한 sklearn에서 제공하는 Decision Tree Regression과 비교하여 코드의 정확도를 비교하였습니다. 이후 코드에 대한 상세 설명을 진행하였습니다.

```
class TNode:
    def __init__(self, depth, X, y):

        self.depth = depth                      # 트리 max depth
        self.X = X                              # train_X (Feature)
        self.y = y                              # train_y (Label)
        self.xi = None                          # 분할 인덱스
        self.left = None                        # 왼쪽 자식 노드
        self.right = None                       # 오른쪽 자식 노드
        self.predictor = None                   # 예측 함수

    def CalculateLoss(self):
        if len(self.y)==0:
            return 0
        else:
            ####### Empty Module.1 #######
            # SSE Loss를 직접 계산하여 반환 # Sum of Squared Error
            return np.sum((self.y - np.mean(self.y))**2)
            ##############################
```

TreeNode: 결정 트리를 구성하는 노드로, class 형식으로 구현하였으며, 내장 함수는 손실함수를 구하는 CalculateLoss가 있습니다. 자료구조의 트리 형태로 데이터의 인덱스를 소분하여 노드 형태로 나누어 저장합니다.

```
def DataSplit(X, y, j, xi):
    ####### Empty Module.2 #######
    # xi보다 작거나 같은 X들의 인덱스 추출
    ids = X[:,j] <= xi
    Xt = [X[i,:] for i in range(len(X)) if ids[i] == True] # ids가 True인 X들만 샘플링
    Xf = [X[i,:] for i in range(len(X)) if ids[i] == False] # ids가 False인 X들만 샘플링

    yt = [y[i] for i in range(len(y)) if ids[i] == True] # ids가 True인 y들만 샘플링
    yf = [y[i] for i in range(len(y)) if ids[i] == False] # ids가 False인 y들만 샘플링
    ################################
    return Xt, yt, Xf, yf
```

데이터 분할 함수로, 입력으로 받은 데이터를 feature와 label로 분리합니다. feature 중 j번째 feature가 xi보다 작거나 같은 데이터와 큰 데이터를 각각 Xt, Xf로 분리하고, label도 같은 방식으로 분리합니다.

```
def CalculateOptimalSplit(node):
    X = node.X
    y = node.y
    best_feature = 0
    best_xi = X[0, best_feature]
    best_split_val = node.CalculateLoss()

    m,n = X.shape

    for j in range(0,n):
        for i in range(0,m):
            xi = X[i,j]
            Xt, yt, Xf, yf = DataSplit(X,y,j,xi)
            tmpt = TNode(0, Xt, yt)
            tmpf = TNode(0, Xf, yf)
            loss_t = tmpt.CalculateLoss()
            loss_f = tmpf.CalculateLoss()
            curr_val = loss_t + loss_f
            ####### Empty Module.3 #######
            if (curr_val < best_split_val):
                best_split_val = curr_val # loss 업데이트
                best_feature = j # best_feature 업데이트
                best_xi = xi              # best_xi 업데이트
            ################################

    return best_feature, best_xi
```

CalculateOptimalSplit 함수는 최적 분할 값을 구하는 함수입니다. 현재 노드의 feature X와 label y를 받아서 best_feature와 best_xi를 최적 분할 지점으로 설정합니다. 이후, 데이터를 best_feature와 best_xi를 기준으로 Xt, Xf로 분할하고, 분할된 데이터를 이용해 TNode class 객체를 만들어 loss_t와 loss_f를 계산합니다. 이 값들을 더해서 curr_val에 저장하고, 이 값이 best_split_val보다 작으면 best_split_val을 curr_val 값으로 업데이트하고, best_feature와 best_xi도 업데이트합니다.

```
def Construct_Subtree(node, max_depth):
    if (node.depth == max_depth or len(node.y) == 1): # node의 깊이가 max_depth에 도달했거나 리프 노드일 때
        ####### Empty Module.4 #######
        node.predictor = np.mean(node.y) # node 내부에 있는 y값들의 평균을 활용하여 예측 수행
        ################################
```

```
    else:
        j, xi = CalculateOptimalSplit(node)
        node.j = j
        node.xi = xi
        Xt, yt, Xf, yf = DataSplit(node.X, node.y, j, xi)

        if (len(yt)>0):
            ####### Empty Module.5 #######
            node.left = TNode(node.depth+1, np.array(Xt), np.array(yt)) # TNode를 활용하여 새로운 왼쪽 자식 노드 구축
            Construct_Subtree(node.left, max_depth) # Construct_Subtree를 활용하여 왼쪽 자식 노드에 대한 Subtree 구축
            ##############################
        if (len(yf)>0):
            ####### Empty Module.6 #######
            node.right = TNode(node.depth+1, np.array(Xf), np.array(yf)) # TNode를 활용하여 새로운 오른쪽 자식 노드 구축
            Construct_Subtree(node.right, max_depth) # Construct_Subtree를 활용하여 오른쪽 자식 노드에 대한 Subtree 구축
            ##############################
            # node를 반환할때 np.array 타입으로 반환해야 한다

    return node
```

Construct_Subtree는 최적 분할을 반복하여 서브 트리를 구축하는 함수입니다. 노드의 깊이가 max_depth에 도달하면 현재 노드에 있는 y값들의 평균을 활용하여 예측 수행합니다. 그렇지 않으면 CalculateOptimalSplit 함수를 이용하여 best_feature와 best_xi를 구하고, 노드의 j와 xi를 업데이트합니다. 그 후, DataSplit 함수를 이용하여 Xt, yt, Xf, yf를 구하고, 왼쪽 자식 노드와 오른쪽 자식 노드를 생성합니다. 이후, 왼쪽 자식 노드와 오른쪽 자식 노드에 대해 Construct_Subtree 함수를 재귀적으로 호출하여 서브 트리를 구축합니다.

```
MSE_scratch = np.mean(np.power(y_hat-y_test, 2)) # MSE 계산 직접 구현한 코드
MSE_scikit = np.mean(np.power(y_hat2-y_test, 2)) # MSE 계산 사이킷런 코드

print("사이킷런 결정트리:loss= {:.3f}".format(MSE_scikit))
print("직접구현 결정트리:loss= {:.3f}".format(MSE_scratch))
```

- 사이킷런 결정트리: `loss= 7.599`
- 직접구현 결정트리: `loss= 7.731`

sklearn의 `Decision Tree Regression`

sklearn에서 제공하는 Decision Tree의 경우, 다양한 하이퍼 파라미터를 설정할 수 있습니다. 이러한 하이퍼 파라미터를 조정함으로써 모델의 성능을 향상시킬 수 있습니다. 하이퍼 파라미터의 종류와 기본값은 다음과 같습니다.

```
{'ccp_alpha': 0.0,
'criterion': 'mse',
'max_depth': None,
'max_features': None,
'max_leaf_nodes': None,
'min_impurity_decrease': 0.0,
'min_samples_leaf': 1,
'min_samples_split': 2,
'min_weight_fraction_leaf': 0.0,
'random_state': None,
'splitter': 'best'}
```

- `ccp_alpha`: 알고리즘의 복잡도를 조정하기 위한 파라미터입니다.
- `criterion`: 분할 품질을 측정하는 데 사용되는 함수입니다. 'mse' 또는 'friedman_mse'를 사용할 수 있습니다.
- `max_depth`: 트리의 최대 깊이를 정합니다. None으로 설정할 경우, 무한대로 트리를 생성합니다.
- `max_features`: 분할 시 고려할 feature의 수를 정합니다. None으로 설정할 경우, 모든 feature를 고려합니다.

- `max_leaf_nodes`: 트리에서 최대 리프 노드 수를 정합니다. None으로 설정할 경우, 무한대로 리프 노드를 생성합니다.
- `min_impurity_decrease`: 불순도를 줄이는 데 대한 최소한의 임계값을 설정합니다.
- `min_samples_leaf`: 리프 노드에 필요한 최소 샘플 수를 정합니다.
- `min_samples_split`: 내부 노드를 분할하기 위해 필요한 최소 샘플 수를 정합니다.
- `min_weight_fraction_leaf`: 리프 노드에 필요한 최소 가중치 비율을 정합니다.
- `random_state`: 비교를 위해 seed 값을 고정합니다.
- `splitter`:분할 전략을 설정합니다. 'best' 또는 'random'을 사용할 수 있습니다.

sklearn의 `Decision Tree Regression`와의 코드 구현의 차이

위 코드에서 구현한 Decision Tree와 sklearn에서 제공하는 Decision Tree의 차이점은 크게 하이퍼 파라미터의 차이와, 학습 방법의 차이 등으로 구분될 수 있습니다.

구현한 Decision Tree는 가능한 모든 분할 지점을 탐색함으로써 더 정확한 분할 기준을 찾을 수 있지만, 이에 따라 모델의 학습 시간이 길어질 수 있습니다. 반면에 sklearn의 Decision Tree는 미리 설정한 하이퍼 파라미터에 따라 탐색 시간을 줄이기 때문에 더 빠른 학습이 가능합니다.

이러한 차이점은 모델의 성능에 영향을 미칠 수 있습니다. 구현한 Decision Tree는 가능한 모든 분할 지점을 탐색함으로써 더 정확한 분할 기준을 찾을 수 있지만, 이에 따라 모델의 학습 시간이 길어지므로 대용량 데이터셋에 대해서는 학습 시간이 길어질 가능성이 있습니다. 반면에 sklearn의 Decision Tree는 미리 설정한 하이퍼 파라미터에 따라 탐색 시간을 줄이기 때문에 대용량 데이터셋에 대해서도 학습이 빠르게 가능합니다.

2. Bagging 배깅

```
# bag 예측기 구성 (500개의 예측기 구성)

n_estimators = 500
bag = np.empty((n_estimators), dtype=object)

for i in range(n_estimators):

    ####### Empty Module.8 #######
    # 복원 추출을 이용하여 랜덤한 훈련 데이터셋 정의 (bootstrap)
    ids = np.random.choice(len(X_train), len(X_train), replace=True)
    # np.random.choice 이용하여 인덱스 먼저 정의, 복원 추출로 진행, size는 len(X_train)만큼 샘플링
    X_boot = X_train[ids] # 인덱스 이용하여 데이터 샘플링
    y_boot = y_train[ids] # 인덱스 이용하여 데이터 샘플링
    ################################

    bag[i] = DecisionTreeRegressor()
    bag[i].fit(X_boot, y_boot)

####### Empty Module.9 #######

# 500개의 예측기를 이용하여 yhatbag 생성 반복문 이용하거나 np의 mean 함수 사용할 것
yhatbag = np.mean([bag[i].predict(X_test) for i in range(n_estimators)], axis=0)
################################

MSE_bagging = np.mean(np.power(yhatbag-y_test, 2))
```

결정트리는 sklearn의 모델을 불러와 사용하였지만, 과정은 모두 코딩으로 구현된 배깅입니다. 먼저, n_estimators 변수를 통해 예측기의 개수를 500으로 설정합니다. 반복문을 통해 n_estimators만큼의 예측기를 생성하고 학습합

니다. 각 예측기는 DecisionTreeRegressor로 초기화되며, 훈련 데이터셋에서 복원 추출(bootstrap)을 통해 랜덤한 샘플을 선택하여 학습합니다. 빈 배열 yhatbag를 생성한 후, 반복문을 통해 각 예측기를 사용하여 X_test에 대한 예측값을 구합니다. np.mean 함수를 사용하여 예측기들의 예측 결과를 평균하여 yhatbag를 계산합니다. MSE_bagging은 y_test와 yhatbag의 평균 제곱 오차(Mean Squared Error)로 계산됩니다.

```
# 결과 출력
print("사이킷런 결정트리:loss= {:.3f}".format(MSE_scikit)) # 사이킷런 결정트리 MSE
print("직접구현 결정트리:loss= {:.3f}".format(MSE_scratch)) # 직접 구현한 결정트리 MSE
print("직접구현-배깅방식:loss= {:.3f}".format(MSE_bagging)) # 직접 구현한 배깅방식 MSE
```

- 사이킷런 결정트리: `loss= 7.599`
- 직접구현 결정트리: `loss= 7.731`
- 직접구현-배깅방식: `loss= 3.198`

sklearn의 `RandomForest`와 코드로 구현한 bagging의 차이

위의 코드에서는 bagging을 적용한 결정 트리 모델을 구현하였습니다. 반면에 sklearn에서는 이와 유사한 알고리즘을 제공하는데, 바로 `RandomForest`입니다. `RandomForest`는 bagging의 일종으로, 여러 개의 결정 트리를 만들어 그 결과를 평균하여 예측하는 알고리즘입니다.

`RandomForest`의 가장 큰 차이점은 개별 결정 트리의 학습 방식입니다. `RandomForest`에서는 개별 결정 트리의 학습 시, 일부 feature들만 랜덤하게 선택하여 사용합니다. 이를 feature sampling이라고 합니다. 이렇게 하면 개별 결정 트리가 서로 다른 feature들을 사용하여 학습하게 되므로, 각 결정 트리는 살짝 다른 결과를 내게 됩니다. 이렇게 다양한 결정 트리의 결과를 평균하여 예측하면, 일반적으로 단일 결정 트리보다 더 좋은 성능을 얻을 수 있습니다.

반면에 구현한 코드에서는, 개별 결정 트리의 학습에는 랜덤성이 적용되지 않습니다. 대신에, 훈련 데이터셋에서 복원 추출(bootstrap)을 통해 랜덤한 샘플을 선택하여 학습합니다. 이렇게 하면 각 결정 트리는 서로 다른 데이터셋을 사용하여 학습하게 되므로, 각 결정 트리는 살짝 다른 결과를 내게 됩니다. 이렇게 다양한 결정 트리의 결과를 평균하여 예측하면, 일반적으로 단일 결정 트리보다 더 좋은 성능을 얻을 수 있습니다. 하지만 `RandomForest`보다는 덜 일반화된 모델을 만들게 됩니다.

또한, 랜덤하게 feature를 선택하는 것보다 데이터를 랜덤하게 샘플링하는 것이 모델의 분산을 줄이는 데 더 효과적이라는 연구 결과도 있습니다. 이러한 이유로, `RandomForest`는 일반적으로 더 선호되는 알고리즘이며, `sklearn`에서도 `RandomForest`를 기본적으로 제공합니다.

Scikit-learn의 의미

과거에 인공지능을 공부할때는 이러한 효율적인 프레임워크가 없었지만, 현재는 인공지능을 공부하는 학생부터, 교수, 연구실 등에서 구현되어 있는 모델을 손쉽게 사용할 수 있도록 sklearn, keras, tensorflow, pytorch등의 다양한 프레임 워크들이 존재하고 우리는 이를 잘 활용하여 우리의 공부, 연구의 핵심적인 부분에 더 집중 할수 있게 되었습니다.

인공지능의 최적화 함수(Optimizer)에는 많은 종류가 있습니다. 우리가 수업시간에서 배운 기본적인 확률적 경사하강법 부터, 나중에는 모멘텀, RMSprop, 가장 많이 사용하는 Adam까지 , 모두 인공지능이 발전하며 모델의 오차를 최소화 하여 최적의 성능을 내기 위해 사용합니다.

제가 수업시간에서 여러가지 기계학습 모델을 만지면서 느낀 큰 두가지 점은 지금까지 배워왔던 모든 모델은 인공지능 기술이 발전한 과정이며, 상대적으로 성능이 안 좋은 모델이라고 생각하기 보다는 방법론적으로 수많은 연구자들이 연구한 성과로서 받아 들여야 한다는 것입니다. 분야가 분야인 많큼 빠르게 발전하고 수많은 방법론이 사용

되며, 의사결정나무가 random forest와 같이 발전한것 처럼 기존의 이론은 새로운 이론의 밑 바탕이 되기에 새로운 가치를 찾을 수 있는 것 같습니다.