DAML HW 5

Cheyenne Bennmarie

Mayuresh Budukh

Edwin Jeyakumar

Hridayraj K Modi

Yuanhui Jiang

Abhishek Raghuvir

In [ ]:

```
!pip install xgboost
from IPython import get_ipython

get_ipython().magic('reset -sf')

import os
import pandas as pd
import numpy as np
import xgboost as xgb
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import ElasticNet
from sklearn.metrics import mean_squared_error
from sklearn import tree
```

Q1

In [ ]:

```
df = pd.read_csv('housing.csv')
# Function to simulate data for model a) and b)
def simulate_data(n_samples):
    # Simulate uncorrelated standard normal variables
    x1 = np.random.normal(0, 1, n_samples)
    x2 = np.random.normal(0, 1, n_samples)
    # Simulate y for model a)
    y_a = 1.5 * x1 + 2 * x2 + np.random.normal(0, 1, n_samples)
    # Simulate y for model b)
    y_b = np.where(x1 > 0, 1.5 * x1 + 2 * x2 + np.random.normal(0, 1, n_samples), 0)
    return x1, x2, y_a, y_b

# Function to calculate out-of-sample mean squared error
def calculate_mse(model, X_train, X_test, y_train, y_test):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    return mse

# Function to perform simulation exercise
def simulation(n_simulations, n_samples_train, n_samples_test):
    results_a = {'OLS': [], 'RandomForest': [], 'XGBoost': []}
    results_b = {'OLS': [], 'RandomForest': [], 'XGBoost': []}

    for _ in range(n_simulations):
        # Simulate data
        x1, x2, y_a, y_b = simulate_data(n_samples_train + n_samples_test)
        # Split data into training and testing samples
        X_train, X_test = np.column_stack((x1[:n_samples_train], x2[:n_samples_train])), np.column_stack((x1[n_samples_train:], x
        y_train_a, y_test_a = y_a[:n_samples_train], y_a[n_samples_train:]
        y_train_b, y_test_b = y_b[:n_samples_train], y_b[n_samples_train:]

        # Model a)
        # OLS
        mse_ols_a = calculate_mse(LinearRegression(), X_train, X_test, y_train_a, y_test_a)
        results_a['OLS'].append(mse_ols_a)

        # Random Forest
        mse_rf_a = calculate_mse(RandomForestRegressor(n_estimators=250, max_depth=10), X_train, X_test, y_train_a, y_test_a)
        results_a['RandomForest'].append(mse_rf_a)

        # XGBoost
        dtrain_a = xgb.DMatrix(X_train, label=y_train_a)
        dtest_a = xgb.DMatrix(X_test)
        params_a = {'learning_rate': 0.3, 'gamma': 0, 'max_depth': 6}
        bst_a = xgb.train(params_a, dtrain_a, num_boost_round=20)
        y_pred_xgb_a = bst_a.predict(dtest_a)
        mse_xgb_a = mean_squared_error(y_test_a, y_pred_xgb_a)
        results_a['XGBoost'].append(mse_xgb_a)

        # Model b)
        # OLS
        mse_ols_b = calculate_mse(LinearRegression(), X_train, X_test, y_train_b, y_test_b)
        results_b['OLS'].append(mse_ols_b)

        # Random Forest
        mse_rf_b = calculate_mse(RandomForestRegressor(n_estimators=250, max_depth=10), X_train, X_test, y_train_b, y_test_b)
        results_b['RandomForest'].append(mse_rf_b)

        # XGBoost
        dtrain_b = xgb.DMatrix(X_train, label=y_train_b)
        dtest_b = xgb.DMatrix(X_test)
        params_b = {'learning_rate': 0.3, 'gamma': 0, 'max_depth': 6}
        bst_b = xgb.train(params_b, dtrain_b, num_boost_round=20)
        y_pred_xgb_b = bst_b.predict(dtest_b)
        mse_xgb_b = mean_squared_error(y_test_b, y_pred_xgb_b)
        results_b['XGBoost'].append(mse_xgb_b)

    return results_a, results_b

# Perform simulation exercise
n_simulations = 500
n_samples_train = 1000
n_samples_test = 500
results_a, results_b = simulation(n_simulations, n_samples_train, n_samples_test)

# Plot histograms
import matplotlib.pyplot as plt

methods = ['OLS', 'RandomForest', 'XGBoost']
for method in methods:
    plt.figure(figsize=(12, 6))
    plt.hist(results_a[method], bins=30, alpha=0.5, label='Model a)')
    plt.hist(results_b[method], bins=30, alpha=0.5, label='Model b)')
    plt.title(f'Out-of-Sample MSE Histogram for {method}')
    plt.xlabel('Mean Squared Error')
    plt.ylabel('Frequency')
    plt.legend(
)
    plt.show()
```

Upon examination of the histograms, it is evident that the performance of all three methods—OLS regression, Random Forest, and XGBoost—is quite similar. This result was anticipated due to the linear relationship between the predictors (designated as Predictor 1 and Predictor 2) and the target variable (designated as Target). While OLS regression, being a linear model, might display slightly superior performance, Random Forest and XGBoost also yield competitive results owing to their ability to capture nonlinear relationships. The histograms illustrate relatively low MSE values across all methods, with OLS regression potentially exhibiting a marginally lower mean MSE. However, there is notable overlap in the MSE distributions among the methods.

Conversely, the histograms for model (b) present a distinct scenario. Here, the relationship between the predictors and the target variable manifests as nonlinear, characterized by a conditional statement. OLS regression, which assumes a linear relationship between predictors and the target, appears to struggle in accurately capturing this nonlinear relationship. As expected, Random Forest and XGBoost surpass OLS regression, benefitting from their adeptness in capturing nonlinear relationships effectively. The histograms for model (b) depict higher MSE values compared to model (a), particularly for OLS regression. This outcome implies a diminished performance of OLS regression in capturing the nonlinear relationship inherent in model (b).

2.

Attached to this problem set is a dataset which deals with Boston real estate prices. The dataset can be found on the UCI Machine Learning Depository: https://archive.ics.uci.edu/ml/index.php (https://archive.ics.uci.edu/ml/index.php). Our goal in this exercise is to predict house prices in Boston (medv) given 11 explanatory variables (columns 1 through 11). Use the first 400 observations as your training sample and observations 401-506 as your test sample.

(a) Use random forest with n_estimators=250 and max_depth=10. Once you run the random forest, use Python's rf.predict function to obtain predicted values for the test sample. What is the MSE of the prediction? Compare this to the benchmark MSE generated by a model that has as its predicted house value the mean house value in the test sample. As in the class notes, also report the Pseudo-R2 implied by these MSEs.

In [46]:

```
df = pd.read_csv('housing.csv')
df.head(6)
```

Out[46]:

	crim	zn	indus	chas	rm	age	dis	rad	tax	ptratio	lstat	medv	nox
0	0.00632	18.0	2.31	0	6.575	65.2	4.0900	1	296	15.3	4.98	24.0	0.538
1	0.02731	0.0	7.07	0	6.421	78.9	4.9671	2	242	17.8	9.14	21.6	0.469
2	0.02729	0.0	7.07	0	7.185	61.1	4.9671	2	242	17.8	4.03	34.7	0.469
3	0.03237	0.0	2.18	0	6.998	45.8	6.0622	3	222	18.7	2.94	33.4	0.458
4	0.06905	0.0	2.18	0	7.147	54.2	6.0622	3	222	18.7	5.33	36.2	0.458
5	0.02985	0.0	2.18	0	6.430	58.7	6.0622	3	222	18.7	5.21	28.7	0.458

In [49]:

```
houses_train = df.iloc[0:399]
predictors_train = houses_train.drop(columns = ["medv", "nox"])
labels_train = houses_train.medv

# Test sample (select the rest)
houses_test = df.iloc[400:]
predictors_test = houses_test.drop(columns = ["medv", "nox"])
labels_test = houses_test.medv

rf = RandomForestRegressor(n_estimators=250 ,
                           max_depth = 10,
                           max_features = 11,
                           random_state = 97)

rf.fit(predictors_train, labels_train)
```

Out[49]:

```
RandomForestRegressor(max_depth=10, max_features=11, n_estimators=250,
                      random_state=97)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [ ]:

```
predictions = rf.predict(predictors_test)
```

In [ ]:

```
importances = rf.feature_importances_

# Sort importances (high to low)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")
for f in range(predictors_train.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

# Plot the impurity-based feature importances of the forest
features_names = predictors_train.columns  # To print the name of the feature

# Since we have too many features, we can just plot top 10 (vertically)
top_10_indices = indices[0:10]
plt.figure()
plt.title("Feature importances (Top 10)")
plt.barh(range(10), importances[top_10_indices],
        color="r", align="center")
plt.yticks(range(10), features_names[top_10_indices])
plt.ylim([-1, 10])
plt.show()
```

In [ ]:

```
# Extract a single tree
estimator = rf.estimators_[5]

# Feature names
fn = list(predictors_train.columns)

# Export a figure using plot_tree
fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (24,10), dpi=800)
tree.plot_tree(estimator,\
               feature_names = fn,\
               filled = True)

fig.savefig('tree.png')
```

In [ ]:

```
# Compute the average of squared errors
MSE_RF = np.mean(np.square((predictions - labels_test)))
print( 'Mean squared error is', MSE_RF )

# Compute pseudo r2
MSE_Baseline = np.mean(np.square((labels_test - np.mean(labels_train))))
PseudoR2 = 1 - MSE_RF / MSE_Baseline
print("Pseudo R2 for random forest is ", round(PseudoR2*100, 2), "%")

# Compare with linear regression
lm = LinearRegression().fit(predictors_train, labels_train)
lm_prediction = lm.predict(predictors_test)

# Compute the average of squared errors
MSE_lm = np.mean(np.square((lm_prediction - labels_test)))

# Compute pseudo r2
PseudoR2 = 1 - MSE_lm / MSE_Baseline
print("Pseudo R2 for linear regression is ", round(PseudoR2*100, 2), "%")
```

(b) Repeat the same exercise as above using XGBoost with learning_rate=0.1, gamma=0, max_depth=6. Use 10 folds and 200 rounds for the cross-validation procedure. Make sure that the output of the cross-validation procedure does not appear in your final write-up

In [ ]:

```
xgb_regressor = xgb.XGBRegressor(
    booster = "gbtree",             # Which booster to use
    objective = "reg:squarederror", # Specify the learning task
    n_estimators = 200,             # Number of trees in random forest to fit
    reg_lambda = 10,                 # L2 regularization term
    gamma = 0,                      # Minimum loss reduction
    max_depth = 6,                  # Maximum tree depth
    learning_rate = 0.1             # Learning rate, eta
)
xgb_parm = xgb_regressor.get_xgb_params()

# Hyper parameter tuning for XGBOOST using cross-validation (maximal # of trees)
# XGBoost uses Dmatrices
xgb_train = xgb.DMatrix(predictors_train, label = labels_train)

# Cross-validation
xgb_cvresult = xgb.cv(xgb_parm, xgb_train,
                      num_boost_round = 200,
                      metrics = "rmse",
                      nfold = 10,
                      stratified=False,
                      seed=1311,
                      early_stopping_rounds=25)

# Print the optimal # of trees
print('Best number of trees = {}'.format(xgb_cvresult.shape[0]))

# Update parameters (# of trees)
xgb_regressor.set_params(n_estimators = xgb_cvresult.shape[0])

# Train the model
xgb_regressor.fit(predictors_train, labels_train)

# Test the model
xgb_prediction = xgb_regressor.predict(predictors_test)

# Compute the average of squared errors
MSE_xgb = np.mean(np.square((xgb_prediction - labels_test)))

# Compute pseudo r2
PseudoR2 = 1 - MSE_xgb / MSE_Baseline
print("Pseudo R2 for XGBoost is ", round(PseudoR2*100, 2), "%")

# get feature importances (high to low)
importances = xgb_regressor.feature_importances_

# Sort importances (high to low)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")
for f in range(predictors_train.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

# Plot the impurity-based feature importances of the forest
features_names = predictors_train.columns  # To print the name of the feature

# Since we have too many features, we can just plot top 10 (vertically)
top_10_indices = indices[0:10]
plt.figure()
plt.title("Feature importances (Top 10)")
plt.barh(range(10), importances[top_10_indices],
        color="r", align="center")
plt.yticks(range(10), features_names[top_10_indices])
plt.ylim([-1, 10])
plt.show()
```

(c) Repeat the exercise in part (a) using elastic net regression (l1_ratio=0.5). Use a cross-validation procedure to find an optimal lambda (alpha). For that exercise, split the training sample into quarters (i.e., the 4-fold cross-validation). Comment on the performance of the linear model relative to decision trees. In particular, get the MSE for the test sample and compute the Pseudo-R2 relative to the benchmark MSE from a).

In [ ]:

```
elastic_net = ElasticNet(l1_ratio=0.5)

                               # split into quarters


quarters = [0, 100, 200, 300, len(predictors_train)]



mse_scores = []
for i in range(4):
    start_index = quarters[i]
    end_index = quarters[i + 1]


    train_q_df = df.iloc[start_index:end_index]
    train_q_predictors = train_q_df.drop(columns=["medv", "nox"])
    train_q_labels = train_q_df["medv"]


    elastic_net.fit(train_q_predictors, train_q_labels)  #model



    predictions = elastic_net.predict(predictors_test)



    mse = mean_squared_error(labels_test, predictions)
    mse_scores.append(mse)


avg_mse = np.mean(mse_scores)
print("Average Mean Squared Error:", avg_mse)
elastic_pseudoR2 = 1 - avg_mse / MSE_Baseline
print('Pseudo R2 for elastic net:', round(elastic_pseudoR2*100, 2), "%")
```

It seems like the decision trees have higher accuracy over linear model in this case. Since the linear model is just in a straight line , it will have more error square around it, the decsiion trees capture each data point with more specific relationships

(d) Try to figure out what the main sources of the discrepancy between Elastic Net and the decision trees are. That is, what is the non-linearity?

The main sourses of disprepany between elastic net and decision tree may be the separation of the data into quarters, there is more of a clustering in each grup that is not going to be as broad compared to whole data. The decision trees are going based non linear since there are different relationships between each of our variables for prediction, and themselves, and the medv predictor,also ,our decision tree is incorporating what it notices in the data so it may be less sensitive to the outlier/ extreme values and not reflect that in model.

In [ ]:

```

```

In [ ]:

```

```