Programming Exercises for Python

August 24, 2025

[1]:
```
# Title: Programming Exercises for Python
# Name:  Julia Hu
# Email: hslhu@outlook.com
```

1 Data wrangling

1.0.1 Create a DataFrame named df with 6 nrows with the following columns:

A: random floating point value
B: randomly assigned categorical values from ["test", "train"]
C: random integer values, constructed from an numpy.array
D: random integer values, constructed from a Series
E: monthly dates "2021-01-01", "2021-02-01", "2021-03-01" ...

[2]:
```
# load the packages
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

[3]:
```
# create a data frame
df = pd.DataFrame({        # set a data frame
        'A' : np.random.rand(6),    # Random 6 float number between 0 and 1
        'B' : pd.Series(np.random.choice(["test", "train"],6),␣
 ↪dtype="category"),  # Random values in the given sequence
        'C' : np.random.randint(1, 100, 6),     # random 6 integer between 1-100
        'D' : np.random.randint(1, 100, 6),    # random 6 integer between 1-100␣
 ↪in a series
        'E' : pd.date_range(start = "2021-01-01",periods = 6, freq = 'MS')  #␣
 ↪construct date range
    })
df
```

[3]:

	A	B	C	D	E
0	0.396275	test	83	74	2021-01-01
1	0.370099	test	90	6	2021-02-01
2	0.305675	test	72	39	2021-03-01
3	0.293076	test	7	23	2021-04-01

```
4  0.238416  test  66  71 2021-05-01
5  0.765061  test  64  79 2021-06-01
```

[4]:
```
# check the data type
print(df.dtypes)
```

```
A           float64
B          category
C             int32
D             int32
E    datetime64[ns]
dtype: object
```

1.0.2 Convert numeric columns into a numpy.matrix and compute the row sums.

[5]:
```
# select numeric data
numeric_columns = df.select_dtypes(include=['number']) # select type
numeric_columns = numeric_columns.values     # convert to matrix
```

[6]:
```
# compute the row sums
row_sum = np.sum(numeric_columns, axis=1) # sum over a given axis
df['Row_Sum'] = row_sum     # assign to a new column
df
```

[6]:

	A	B	C	D	E	Row_Sum
0	0.396275	test	83	74	2021-01-01	157.396275
1	0.370099	test	90	6	2021-02-01	96.370099
2	0.305675	test	72	39	2021-03-01	111.305675
3	0.293076	test	7	23	2021-04-01	30.293076
4	0.238416	test	66	71	2021-05-01	137.238416
5	0.765061	test	64	79	2021-06-01	143.765061

1.0.3 Sort df by column C.

[7]:
```
# sort columns C
df_sorted = df.sort_values(by='C') # sort number
df_sorted
```

[7]:

	A	B	C	D	E	Row_Sum
3	0.293076	test	7	23	2021-04-01	30.293076
5	0.765061	test	64	79	2021-06-01	143.765061
4	0.238416	test	66	71	2021-05-01	137.238416
2	0.305675	test	72	39	2021-03-01	111.305675
0	0.396275	test	83	74	2021-01-01	157.396275
1	0.370099	test	90	6	2021-02-01	96.370099

1.0.4 Filter df for entries for which B has value train and C has values greater than 0.

[8]:
```
# filter the data
filtered_df = df[(df['B'] == 'train') & (df['C'] > 0)] # filter the data
filtered_df
```

[8]:
```
Empty DataFrame
Columns: [A, B, C, D, E, Row_Sum]
Index: []
```

1.0.5 Change the value in the 4th column and 2nd row to 10.

[9]:
```
# change the value
df.iloc[1, 3] = 10  # index start with 0
df
```

[9]:

	A	B	C	D	E	Row_Sum
0	0.396275	test	83	74	2021-01-01	157.396275
1	0.370099	test	90	10	2021-02-01	96.370099
2	0.305675	test	72	39	2021-03-01	111.305675
3	0.293076	test	7	23	2021-04-01	30.293076
4	0.238416	test	66	71	2021-05-01	137.238416
5	0.765061	test	64	79	2021-06-01	143.765061

1.0.6 Create a column F where half the values are NaN.

[10]:
```
# create a new column with same length
column_f = np.random.rand(len(df))
```

[11]:
```
slice_nan = np.random.choice(len(column_f), size=len(df) // 2, replace=False) #␣
 ↪random select half of the data
column_f[slice_nan] = np.nan     # filled with nan
df['F'] = column_f      # create F column
df
```

[11]:

	A	B	C	D	E	Row_Sum	F
0	0.396275	test	83	74	2021-01-01	157.396275	0.717083
1	0.370099	test	90	10	2021-02-01	96.370099	0.546147
2	0.305675	test	72	39	2021-03-01	111.305675	0.019948
3	0.293076	test	7	23	2021-04-01	30.293076	NaN
4	0.238416	test	66	71	2021-05-01	137.238416	NaN
5	0.765061	test	64	79	2021-06-01	143.765061	NaN

1.0.7 Deal with missing values in two different ways:

1. remove entries with missing data
2. fill missing values with 0

[12]:
```
# deal with the data
df_without_missing = df.dropna() # drop the missing value
```

[13]:
```
categorical_cols = df.select_dtypes(include=['category']).columns
categorical_cols
```

[13]:
```
Index(['B'], dtype='object')
```

[14]:
```
df_without_B = df.drop('B', axis=1)
df_filled_with_zero = df_without_B.fillna(0)  # filled with zero
```

1.0.8 Convert column A into a cumulative sum.

[15]:
```
# calculate column A
df['A'] = df['A'].cumsum() # cumulative sum for A
df
```

[15]:

	A	B	C	D	E	Row_Sum	F
0	0.396275	test	83	74	2021-01-01	157.396275	0.717083
1	0.766373	test	90	10	2021-02-01	96.370099	0.546147
2	1.072048	test	72	39	2021-03-01	111.305675	0.019948
3	1.365124	test	7	23	2021-04-01	30.293076	NaN
4	1.603540	test	66	71	2021-05-01	137.238416	NaN
5	2.368600	test	64	79	2021-06-01	143.765061	NaN

1.0.9 Subtract column A from column B.

Ans: Since column B is a character column and column A is a numeric column, they cannot be subtracted from each other.

1.0.10 Plot the numeric columns as a line plot, ensuring that the plot has proper labels.

[16]:
```
# prepare the data
numeric_columns = df.select_dtypes(include=['number']) # select numeric columns
numeric_columns
```

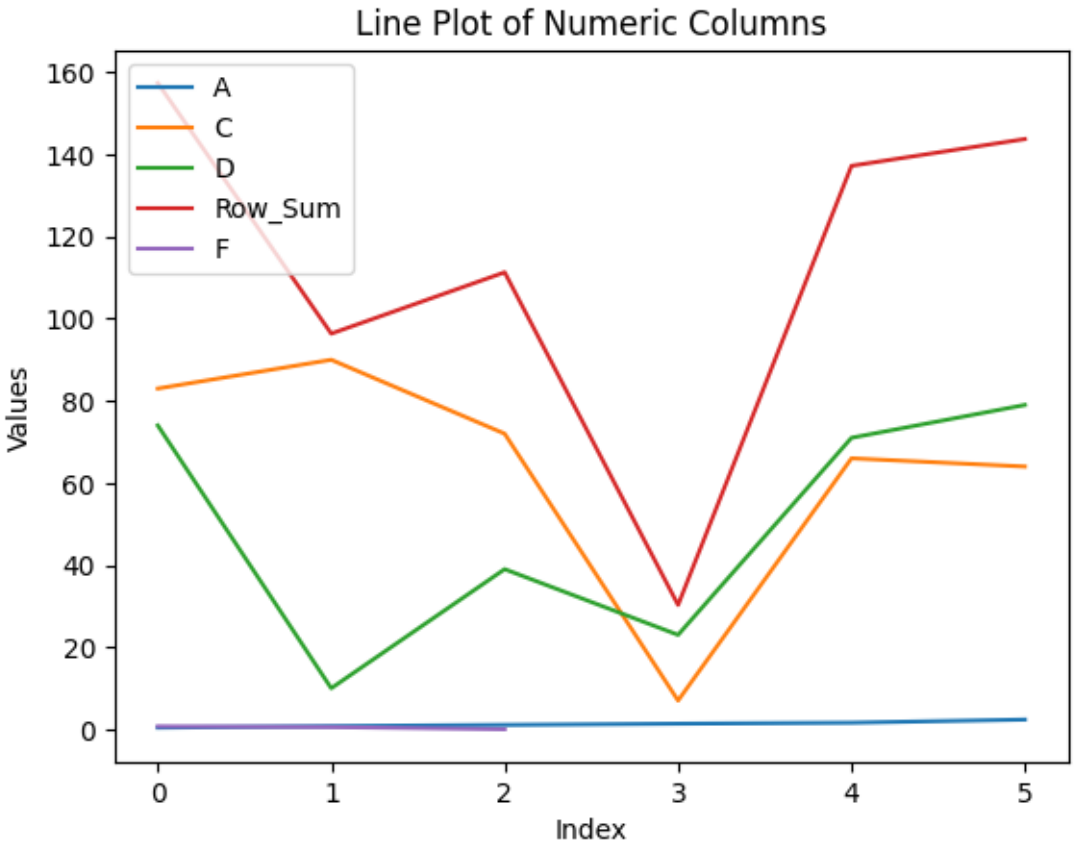
[16]:

	A	C	D	Row_Sum	F
0	0.396275	83	74	157.396275	0.717083
1	0.766373	90	10	96.370099	0.546147
2	1.072048	72	39	111.305675	0.019948
3	1.365124	7	23	30.293076	NaN
4	1.603540	66	71	137.238416	NaN
5	2.368600	64	79	143.765061	NaN

[17]:
```
#plot the line plot
numeric_columns.plot() # create line plot
```

```
# adding title and index
plt.title('Line Plot of Numeric Columns') # add title
plt.xlabel('Index')    # x label as index
plt.ylabel('Values')   # y as value
plt.legend()     # set legend

# show the plot
plt.show()
```



1.0.11 Compute the mean values of each column for groups train and test.

[18]:
```
# groupby the column B and compute the mean value
grouped_mean = df.groupby('B',observed=True).mean()
grouped_mean
```

[18]:

B	A	C	D	E	Row_Sum	F
test	1.261993	63.666667	49.333333	2021-03-17 04:00:00	112.7281	0.427726

1.0.12 Convert the following DataFrame from a into b (long to wide). Additionally, convert from b into a (wide to long).

```
a = pd.DataFrame({  "value": [1, 2, 3, 4, 5, 6],
                    "group": ["a", "a", "a", "b", "b", "b"]})

b = pd.DataFrame({  "a": [1, 2, 3],
                    "b": [4, 5, 6]})
```

[19]:
```
# create a

a = pd.DataFrame({"value": [1, 2, 3, 4, 5, 6],
                  "group": ["a", "a", "a", "b", "b", "b"]})
a
```

[19]:

	value	group
0	1	a
1	2	a
2	3	a
3	4	b
4	5	b
5	6	b

[20]:
```
# create b
b = pd.DataFrame(
        {"a": [1, 2, 3],
         "b": [4, 5, 6]})
b
```

[20]:

	a	b
0	1	4
1	2	5
2	3	6

[21]:
```
# set index
a['index_col'] = a.groupby('group').cumcount()  # create a column and count for␣
 ↪each group
a
```

[21]:

	value	group	index_col
0	1	a	0
1	2	a	1
2	3	a	2
3	4	b	0
4	5	b	1
5	6	b	2

[22]:
```
# pivot data from long to wide data
a_to_b = a.pivot(index="index_col",columns="group",values="value") # reshape␣
 ↪data
a_to_b
```

[22]:
group	a	b
index_col		
0	1	4
1	2	5
2	3	6

[23]:
```
# clean the format
a_to_b = a_to_b.reset_index(drop=True,names=None) # drop the index
a_to_b.index.name = None # clean the index name
a_to_b
```

[23]:
group	a	b
0	1	4
1	2	5
2	3	6

[24]:
```
# melt data from wide to long formate
b_to_a = pd.melt(b, var_name='group', value_name='value')  # reshape the data
b_to_a
```

[24]:
	group	value
0	a	1
1	a	2
2	a	3
3	b	4
4	b	5
5	b	6

2 Supervised learning

2.0.1 Load the iris dataset by

```
import sklearn as sk
import sklearn.datasets
iris = sk.datasets.load_iris()
```

[25]:
```
# load the package and data
import sklearn as sk
import sklearn.datasets
iris = sk.datasets.load_iris()
```

[26]:
```
# check the data
dir(iris) # check dir of the data
```

[26]:
```
['DESCR',
 'data',
 'data_module',
 'feature_names',
 'filename',
 'frame',
 'target',
 'target_names']
```

[27]:
```
# read the data as the data frame
iris_df = pd.DataFrame(
    data = iris.data,  # set iris data
    columns = iris.feature_names) # set column names

iris_df.head(3)
```

[27]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2

[28]:
```
# create target in the data
iris_df['target'] = iris.target   # 0 = Iris Setosa, 1 = Iris Versicolour, 2 =␣
 ↪Iris Virginica
iris_df.head(3)
```

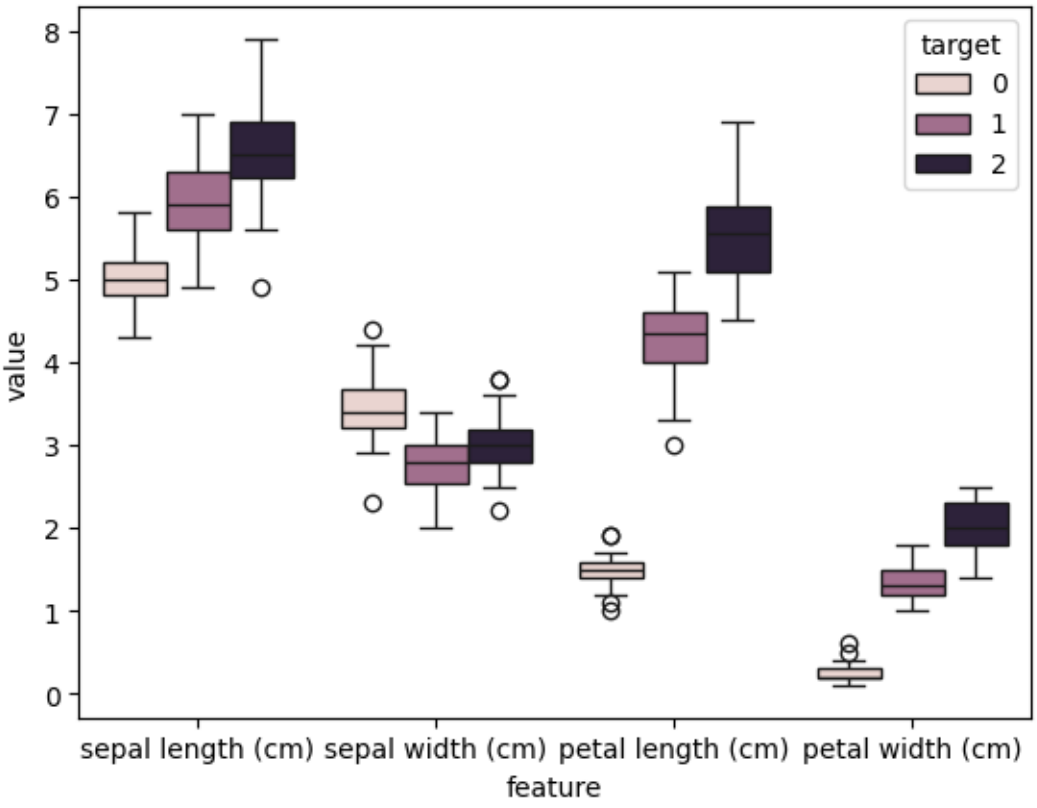
[28]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	\
0	5.1	3.5	1.4	0.2	
1	4.9	3.0	1.4	0.2	
2	4.7	3.2	1.3	0.2	

	target
0	0
1	0
2	0

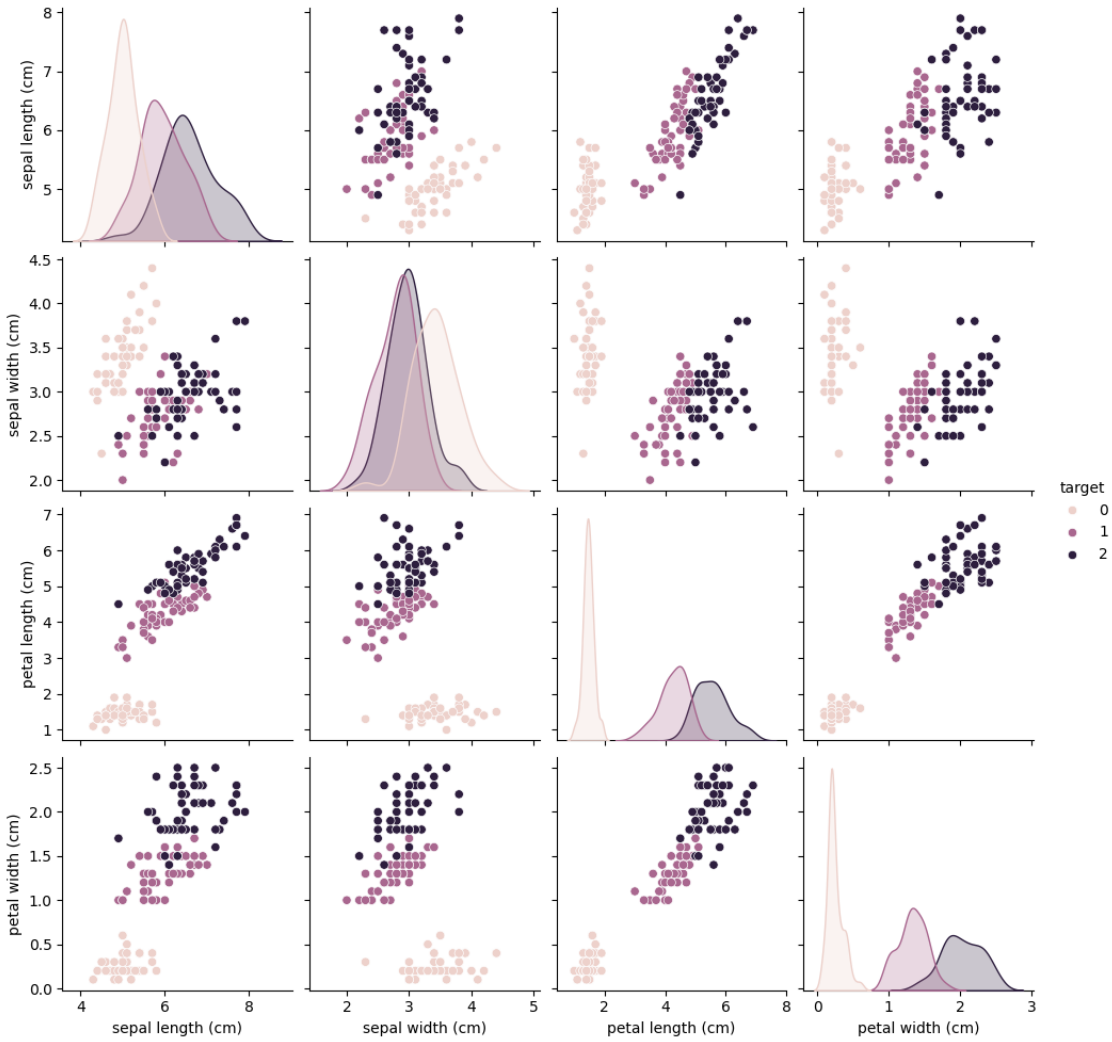
2.0.2 Visualize the data matrix.

[29]:
```
# Box plot
iris_melt = pd.melt(iris_df, id_vars='target', var_name='feature',␣
 ↪value_name='value') # convert data from wide to long format for plot
sns.boxplot(x='feature', y='value', hue='target', data=iris_melt)  # plot box␣
 ↪plot
plt.show()
```



[30]:
```
# plot the data
sns.pairplot(iris_df, hue = "target")
```

[30]: <seaborn.axisgrid.PairGrid at 0x1f97fe38ad0>



2.0.3 Train a random forest classifier to predict the target values and report its performance using an appropriate evaluation metric.

[31]:
```
# load the package
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
```

[32]:
```
# split into training and testingd data
feature_train,feature_test, target_train,target_test = train_test_split(
    iris_df.drop(['target'], axis=1),
    iris.target,
    test_size=0.3,  # 30% of the data as test data
    random_state = 42
)
```

[33]:
```
#load random forest model
model = RandomForestClassifier(
    n_estimators=140 , # 140 trees
    max_depth = 20,
    random_state=42
)
model.fit(feature_train,target_train) # fit the model
model.score(feature_test,target_test) # accuracy score
```

[33]: 1.0

[34]:
```
# Evaluate
y_predicted = model.predict(feature_test)
accuracy = accuracy_score(target_test, y_predicted)
print(f"Model Accuracy: {accuracy:.4f}\n")
```

```
Model Accuracy: 1.0000
```

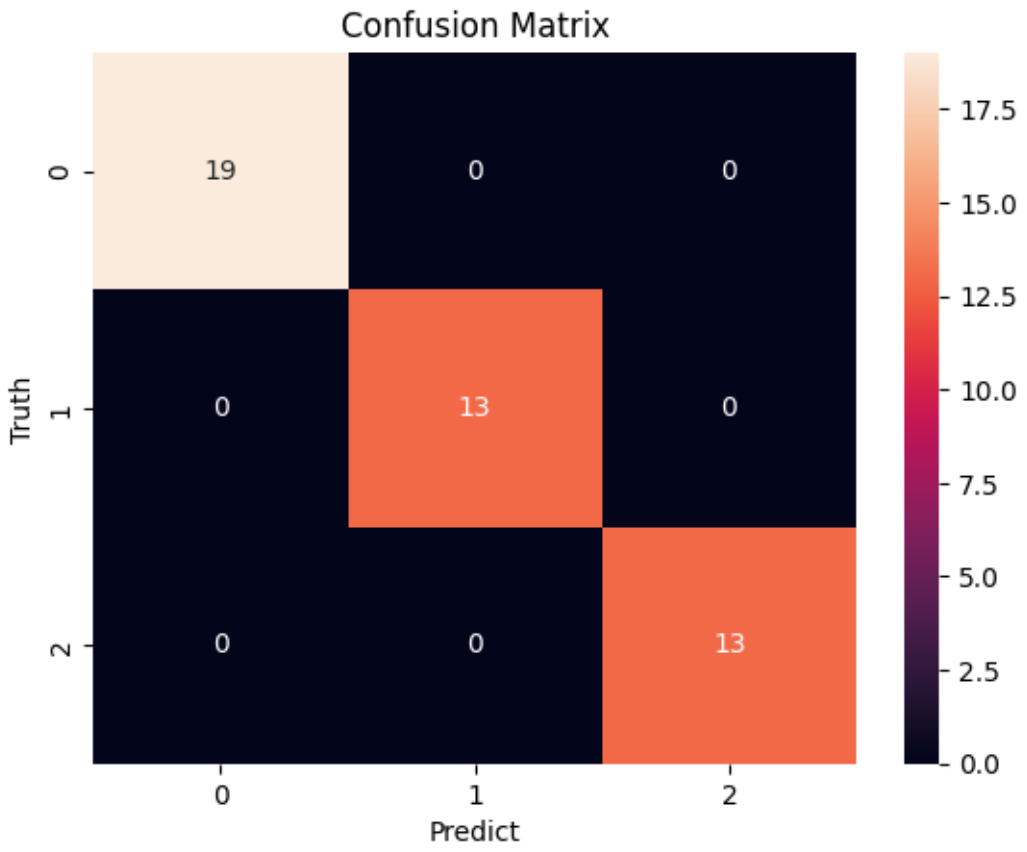
[35]:
```
# bulid confusion matrix with test data and predicted data
cm = confusion_matrix(target_test,y_predicted)
cm
```

[35]:
```
array([[19,  0,  0],
       [ 0, 13,  0],
       [ 0,  0, 13]])
```

[36]:
```
# visulized the result
```

[37]:
```
# plot heatmap
sns.heatmap(cm , annot=True)
plt.xlabel("Predict")   # x label
plt.ylabel('Truth')   # y label
plt.title("Confusion Matrix") #title

plt.show()
```



2.0.4 Explain how key parameters of the random forest classifier would influence its peformance.

The first key parameter is `n_estimators`, which determines the number of decision trees in the forest. More trees usually make the model better by balancing out noise, but after a point, adding more doesn't help much and just takes more time.

The second key parameter is `max_depth`, which controls the maximum depth of each decision tree. Deeper trees can capture more complex patterns but risk overfitting (learning noise or irrelevant details); shallower trees may underfit by failing to capture important trends in the data.

3 Text mining

3.0.1 Using Biopython, collect medline abstracts on "medulloblastoma" published in 2012. Save the data to disk as a CSV table.

[38]:
```
# load the package
from Bio import Entrez
import xml.etree.ElementTree as ET
```

[40]:
```
# search the data
Entrez.email = "hslhu@outlook.com"  # provide the email address

# Search the data
stream = Entrez.esearch(
    db="pubmed",
    term="medulloblastoma AND 2012[PDAT]",
    usehistory="y",
    RetMax = 500,
    retmode="xml") # retrieved 500 data

# search the data with the keyword
search_results = Entrez.read(stream) # parses the XML file
stream.close()
```

[41]:
```
# count the result
acc_list = search_results["IdList"] # get the id list
count = int(search_results["Count"]) # count the list
len(acc_list)
```

[41]: 445

[42]:
```
# get session history feature
webenv = search_results["WebEnv"]
query_key = search_results["QueryKey"]
```

[43]:
```
# Searching for and downloading sequences using the history
batch_size = 100
output = open("medulloblastoma_2012.txt", "w", encoding='utf-8')

for start in range(0, count, batch_size):
    end = min(count, start + batch_size)
    print("Going to download record %i to %i" % (start + 1, end))
    stream = Entrez.efetch(
        db="pubmed",  # database
        rettype="medline",  # retrieval type
        retmode="text", # retrieval mode
        retstart=start,
        retmax=batch_size,
        webenv=search_results["WebEnv"],
        query_key=search_results["QueryKey"],
    )
    data = stream.read()
    stream.close()
    output.write(data)
output.close()
```

```
Going to download record 1 to 100
```

```
Going to download record 101 to 200
Going to download record 201 to 300
Going to download record 301 to 400
Going to download record 401 to 445
```

[44]:
```python
from Bio import Medline
# create the data
records = [] # create empty list
with open("medulloblastoma_2012.txt", encoding='utf-8') as stream:
    for record in Medline.parse(stream):
        pmid = record.get("PMID", "N/A")
        title = record.get("TI", "N/A")
        Author = record.get("AU","N/A")
        abstract = record.get("AB", "N/A")

        records.append({
            'PMID': pmid,
            'Title': title,
            'Author': Author,
            'Abstract': abstract
        })
```

[45]:
```python
# clean the data
df_record = pd.DataFrame(records) # convert the data as data frame

# remove sigal quotation from the author
df_record['Author'] = df_record['Author'].apply(str).str.replace("'", '',␣
 ↪regex=False)

df_record.head(5)
```

[45]:

	PMID	Title	Author	Abstract
0	24049850	Small-molecule antagonists of Gli function.	[Ardecky R, Magnuson GK, Zou J, Ganji SR, Brow…	As cancer treatments have shifted toward targe…
1	24273611	Role of Epidermal Growth Factor-Triggered PI3K…	[Dudu V, Able RA Jr, Rotari V, Kong Q, Vazquez M]	
2	23864912	Update on molecular and genetic alterations in…	[Kool M, Korshunov A, Pfister SM]	
3	23691470	Adult medulloblastoma associated with syringom…	[Wang CC]	
4	23430850	Onset of adreno-leukodystrophy after medullobl…	[Deib G, Poretti A, Meoded A, Cohen KJ, Raymon…	

```
1  Medulloblastoma (MB) is the most common brain …
2  Medulloblastoma encompasses a group of aggress…
3  The association between cerebellar medulloblas…
4  X-linked adreno-leukodystrophy (ALD) is a pero…
```

[46]:
```
# write the data to csv file
df_record.to_csv('medulloblastoma_2012.csv', index=False,␣
↪sep='\t',encoding='utf-8')
```

4 Neural network

Implement a full connected feedforward network from scratch using only the numpy library with the following layers: one input, two hidden, and one output. Neurons in the first hidden layer should use the sigmoid transfer function; those in the second hidden layer should use a ReLU transfer function. The network should be trained using backpropagation of errors.

4.1 Build the model

[49]:
```
# Initialized parameters
def initialize_parameters(input_size, hidden_size1, hidden_size2, output_size):
    parameters = {
        "W1": np.random.randn(hidden_size1, input_size) * 0.01,  # first layer␣
↪weight
        "b1": np.zeros((hidden_size1, 1)),                       # first layer␣
↪bias
        "W2": np.random.randn(hidden_size2, hidden_size1) * 0.01, # second␣
↪layer weight
        "b2": np.zeros((hidden_size2, 1)),                       # second layer␣
↪bias
        "W3": np.random.randn(output_size, hidden_size2) * 0.01,  # output␣
↪layer weight
        "b3": np.zeros((output_size, 1))                         # output layer␣
↪bias
    }
    return parameters
```

[50]:
```
# define active function

def relu(Z): # ReLU active function
    return np.maximum(0, Z)

def relu_derivative(Z):  #derivative of ReLU active function
    return (Z > 0).astype(int)

def sigmoid(Z): # sigmoid active function
    return 1 / (1 + np.exp(-Z))
```

```
def sigmoid_derivative(A2): # derivative of sigmoid function
    return A2 * (1 - A2)
```

[51]:
```
# define forward propagation
def forward_propagation(X, parameters):
    W1, b1, W2, b2, W3, b3 = parameters["W1"], parameters["b1"],␣
 ↪parameters["W2"], parameters["b2"],parameters["W3"], parameters["b3"]

    # first input layer
    Z1 = np.dot(W1, X) + b1  # pass data to the first linear funtion

    # hidden layer 1 & 2
    A1 = relu(Z1)     # pass the results to the first active functuon
    Z2 = np.dot(W2, A1) + b2  # Pass the result to the second linear function
    A2 = sigmoid(Z2)  # pass the results to the second active function

    # output layer
    Z3 = np.dot(W3, A2) + b3 # pass the result to the last linear function

    cache = {"Z1": Z1, "A1": A1, "Z2": Z2, "A2": A2, "Z3":Z3}
    return Z3, cache
```

[52]:
```
# calculate the loss function MSE
def compute_cost(Y, Z3):
    m = Y.shape[1]  # number of examples
    cost = np.sum((Z3 - Y) ** 2) / (2 * m)  # MSE function
    return np.squeeze(cost)  # remove any extra dimensions, returning the cost␣
 ↪as a scalar
```

[53]:
```
def backward_propagation(X, Y, parameters, cache):
    m = X.shape[1]
    W1, W2, W3 = parameters["W1"], parameters["W2"], parameters["W3"]
    A1, A2, Z1, Z2, Z3 = cache["A1"], cache["A2"], cache["Z1"], cache["Z2"],␣
 ↪cache["Z3"]


    dZ3 = Z3 - Y    # output layer (dZ3) as the difference between predicted␣
 ↪outputs (Z3) and true labels (Y).

    dW3 = np.dot(dZ3, A2.T) / m
    db3 = np.sum(dZ3, axis=1, keepdims=True) / m

    dZ2 = np.dot(W3.T, dZ3) * sigmoid_derivative(A2)
    dW2 = np.dot(dZ2, A1.T) / m
    db2 = np.sum(dZ2, axis=1, keepdims=True) / m
```

```
    dZ1 = np.dot(W2.T, dZ2) * relu_derivative(Z1)
    dW1 = np.dot(dZ1, X.T) / m
    db1 = np.sum(dZ1, axis=1, keepdims=True) / m

    grads = {"dW1": dW1, "db1": db1, "dW2": dW2, "db2": db2, "dW3": dW3, "db3":␣
 ↪db3}
    return grads
```

[54]:
```
# update the parameters
def update_parameters(parameters, grads, learning_rate):
    for key in parameters.keys(): # for each key in parameters,
        parameters[key] -= learning_rate * grads["d" + key] # update the grads␣
 ↪result by * learning rate
    return parameters
```

[55]:
```
# traning the model
def train_neural_network(X, Y, input_size, hidden_size1, hidden_size2,␣
 ↪output_size, epochs=1000, learning_rate=0.01):
    parameters = initialize_parameters(input_size, hidden_size1, hidden_size2,␣
 ↪output_size)

    for i in range(epochs):
        Z3, cache = forward_propagation(X, parameters) # get the output layer␣
 ↪data
        cost = compute_cost(Y, Z3)  # calculate the loss
        grads = backward_propagation(X, Y, parameters, cache) # gradient descent
        parameters = update_parameters(parameters, grads, learning_rate) #␣
 ↪update parameters

        if i % 100 == 0:
            print(f"Epoch {i}: Cost = {cost}")

    return parameters
```

5 Prepare the data

[56]:
```
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# using california housing data
california = fetch_california_housing()
```

```
X = california.data  # features
y = california.target.reshape(-1, 1)  # price

# split to training and testing dataset with 20% tese data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# standard the scaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# transposed the data
X_train = X_train.T
X_test = X_test.T
y_train = y_train.T
y_test = y_test.T
```

[57]:
```
# fit the data in model
trained_parameters = train_neural_network(X_train, y_train,
                                          input_size=8, hidden_size1=16,␣
 ↪hidden_size2=10, output_size=1,
                                          epochs=5000, learning_rate=0.1)

# After testing, using 5000 epochs with a 0.1 learning rate yielded the best␣
 ↪result.
```

```
Epoch 0: Cost = 2.8254472224923135
Epoch 100: Cost = 0.6674016700316427
Epoch 200: Cost = 0.6502319834246496
Epoch 300: Cost = 0.48264698993268124
Epoch 400: Cost = 0.33298369241892
Epoch 500: Cost = 0.28925321740648696
Epoch 600: Cost = 0.26331044288229183
Epoch 700: Cost = 0.2427943571380752
Epoch 800: Cost = 0.23054151418367014
Epoch 900: Cost = 0.22418002441285081
Epoch 1000: Cost = 0.21920568226522866
Epoch 1100: Cost = 0.21599302117082092
Epoch 1200: Cost = 0.21347047550190731
Epoch 1300: Cost = 0.21119597758307043
Epoch 1400: Cost = 0.20901401068894748
Epoch 1500: Cost = 0.20691819915425952
Epoch 1600: Cost = 0.20487507060300267
Epoch 1700: Cost = 0.2027676736110235
Epoch 1800: Cost = 0.20068111830355548
Epoch 1900: Cost = 0.1988559359156338
Epoch 2000: Cost = 0.1971775022178321
```

```
Epoch 2100: Cost = 0.19561864049808103
Epoch 2200: Cost = 0.19414475967670433
Epoch 2300: Cost = 0.19277718794353493
Epoch 2400: Cost = 0.19150131604347598
Epoch 2500: Cost = 0.1902977737402879
Epoch 2600: Cost = 0.18917159756582289
Epoch 2700: Cost = 0.18813268660827423
Epoch 2800: Cost = 0.18718983584447887
Epoch 2900: Cost = 0.1863238080576259
Epoch 3000: Cost = 0.18552108684669913
Epoch 3100: Cost = 0.18478781226401608
Epoch 3200: Cost = 0.1841163569750945
Epoch 3300: Cost = 0.18347833151848125
Epoch 3400: Cost = 0.18289202246662703
Epoch 3500: Cost = 0.18233887207830188
Epoch 3600: Cost = 0.18179970627533795
Epoch 3700: Cost = 0.18126903688730495
Epoch 3800: Cost = 0.1807660255527435
Epoch 3900: Cost = 0.18028717991359433
Epoch 4000: Cost = 0.17983259506735227
Epoch 4100: Cost = 0.1793915091022361
Epoch 4200: Cost = 0.17896935637555506
Epoch 4300: Cost = 0.17858450831034986
Epoch 4400: Cost = 0.1782167178761463
Epoch 4500: Cost = 0.17787673943090299
Epoch 4600: Cost = 0.1775477997312949
Epoch 4700: Cost = 0.17721771625650948
Epoch 4800: Cost = 0.17688465974846027
Epoch 4900: Cost = 0.17653674437936337
```

[58]:
```python
# predict the result
def predict(X, parameters):
    Z3, _ = forward_propagation(X, parameters)
    return Z3
```

[59]:
```python
# measure the difference between predicted values and true values
def calculate_rmse(y_true, y_pred):

    return np.sqrt(np.mean((y_true - y_pred)**2))
```

[60]:
```python
# Average absolute difference, less sensitive to outliers
def calculate_mae(y_true, y_pred):
    return np.mean(np.abs(y_true - y_pred))
```

[61]:
```python
def calculate_r2(y_true, y_pred):
    ss_total = np.sum((y_true - np.mean(y_true)) **2)
    ss_residual = np.sum((y_true - y_pred)** 2)
    return 1 - (ss_residual / ss_total)
```

[62]:
```
# calculate prediction errors
y_pred_test = predict(X_test, trained_parameters) # predict housing prices␣
 ↪using test data
test_rmse = calculate_rmse(y_test, y_pred_test)  # calculate rmse between␣
 ↪predictions and true values
print(f"RMSE: {test_rmse:.4f}")
```

```
RMSE: 0.6042
```

[63]:
```
# evaluate the model prediction
error = (test_rmse / np.median(y)) * 100
print(f"The percentage of error: {error:.2f}")
# The model shows a 33% prediction error relative to the median true values
```

```
The percentage of error: 33.62
```