In [2]:

*# Program 1*

**class** Graph:

**def** init (self,adjac\_lis): self.adjac\_lis **=** adjac\_lis

**def** get\_neighbours(self,v):

**return** self.adjac\_lis[v]

**def** h(self,n):

H **=** {'A':1,'B':1, 'C':1,'D':1}

**return** H[n]

**def** a\_star\_algorithm(self,start,stop): open\_lst **=** set([start])

closed\_lst **=** set([]) dist **=** {}

dist[start] **=** 0 prenode **=** {}

prenode[start] **=** start

**while** len(open\_lst) **>** 0: n **= None**

**for** v **in** open\_lst:

**if** n **== None or** dist[v] **+** self.h(v) **<** dist[n] **+** self.h(n): n **=** v;

**if** n **== None**:

print("path doesnot exist")

**return None if** n **==** stop:

reconst\_path **=** []

**while** prenode[n] **!=** n:

reconst\_path.append(n) n **=** prenode[n]

reconst\_path.append(start) reconst\_path.reverse()

print("path found: {".format(reconst\_path))

**return** reconst\_path

**for** (m, weight) **in** self.get\_neighbours(n):

**if** m **not in** open\_lst **and** m **not in** closed\_lst: open\_lst.add(m)

prenode[m] **=** n

dist[m] **=** dist[n] **+** weight

**else**:

**if** dist[m] **>** dist[n] **+** weight: dist[m] **=** dist[n] **+** weight prenode[m] **=** n

**if** m **in** closed\_lst:

closed\_lst.remove(m) open\_lst.add(m)

open\_lst.remove(n) closed\_lst.add(n)

print("Path does not exist")

**return None**

adjac\_lis **=**{'A':[('B',1),('C',3),('D',7)],'B':[('D',5)],'C':[('D',12)]}

graph1**=**Graph(adjac\_lis)

graph1.a\_star\_algorithm('A', 'D')

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

path found:['A', 'B', 'D'] Out[2]:

['A', 'B', 'D']

In [3]:

1. **def** recAOStar(n):
2. **global** finalPath
3. print("Expanding Node:",n)
4. and\_nodes **=** []
5. or\_nodes **=**[]
6. **if**(n **in** allNodes):
7. **if** 'AND' **in** allNodes[n]:
8. and\_nodes **=** allNodes[n]['AND']
9. **if** 'OR' **in** allNodes[n]:
10. or\_nodes **=** allNodes[n]['OR']
11. **if** len(and\_nodes)**==**0 **and** len(or\_nodes)**==**0:

# return

13

1. solvable **= False**
2. marked **=**{}

16

1. **while not** solvable:
2. **if** len(marked)**==**len(and\_nodes)**+**len(or\_nodes):
3. min\_cost\_least,min\_cost\_group\_least **=** least\_cost\_group(and\_nodes,or\_nodes,{})
4. solvable **= True**
5. change\_heuristic(n,min\_cost\_least)
6. optimal\_child\_group[n] **=** min\_cost\_group\_least

# continue

1. min\_cost,min\_cost\_group **=** least\_cost\_group(and\_nodes,or\_nodes,marked)
2. is\_expanded **= False**
3. **if** len(min\_cost\_group)**>**1:
4. **if**(min\_cost\_group[0] **in** allNodes):
5. is\_expanded **= True**
6. recAOStar(min\_cost\_group[0])
7. **if**(min\_cost\_group[1] **in** allNodes):
8. is\_expanded **= True**
9. recAOStar(min\_cost\_group[1])

# else:

1. **if**(min\_cost\_group **in** allNodes):
2. is\_expanded **= True**
3. recAOStar(min\_cost\_group)
4. **if** is\_expanded:
5. min\_cost\_verify, min\_cost\_group\_verify **=** least\_cost\_group(and\_nodes, or\_nodes, {})
6. **if** min\_cost\_group **==** min\_cost\_group\_verify:
7. solvable **= True**
8. change\_heuristic(n, min\_cost\_verify)
9. optimal\_child\_group[n] **=** min\_cost\_group

# else:

1. solvable **= True**
2. change\_heuristic(n, min\_cost)
3. optimal\_child\_group[n] **=** min\_cost\_group
4. marked[min\_cost\_group]**=**1
5. **return** heuristic(n)

49

1. **def** least\_cost\_group(and\_nodes, or\_nodes, marked):
2. node\_wise\_cost **=** {}
3. **for** node\_pair **in** and\_nodes:
4. **if not** node\_pair[0] **+** node\_pair[1] **in** marked:
5. cost **=** 0
6. cost **=** cost **+** heuristic(node\_pair[0]) **+** heuristic(node\_pair[1]) **+** 2
7. node\_wise\_cost[node\_pair[0] **+** node\_pair[1]] **=** cost
8. **for** node **in** or\_nodes:
9. **if not** node **in** marked:
10. cost **=** 0
11. cost **=** cost **+** heuristic(node) **+** 1
12. node\_wise\_cost[node] **=** cost

62 min\_cost **=** 999999

1. min\_cost\_group **= None**
2. **for** costKey **in** node\_wise\_cost:
3. **if** node\_wise\_cost[costKey] **<** min\_cost:
4. min\_cost **=** node\_wise\_cost[costKey]
5. min\_cost\_group **=** costKey
6. **return** [min\_cost, min\_cost\_group] 69
7. **def** heuristic(n):
8. **return** H\_dist[n]

72

1. **def** change\_heuristic(n, cost):
2. H\_dist[n] **=** cost

# return

76

1. **def** print\_path(node):
2. print(optimal\_child\_group[node], end**=**"")
3. node **=** optimal\_child\_group[node]
4. **if** len(node) **>** 1:
5. **if** node[0] **in** optimal\_child\_group:
6. print("->", end**=**"")
7. print\_path(node[0])
8. **if** node[1] **in** optimal\_child\_group:
9. print("->", end**=**"")
10. print\_path(node[1])

# else:

1. **if** node **in** optimal\_child\_group:
2. print("->", end**=**"")
3. print\_path(node)

Expanding Node: B Expanding Node: C Expanding Node: D

|  |  |
| --- | --- |
| 91 | H\_dist **=** { |
| 92 | 'A': **-**1, |
| 93 | 'B': 4, |
| 94 | 'C': 2, |
| 95 | 'D': 3, |
| 96 | 'E': 6, |
| 97 | 'F': 8, |
| 98 | 'G': 2, |
| 99 | 'H': 0, |
| 100 | 'I': 0, |
| 101 | 'J': 0 |
| 102 | } |
| 103 | allNodes **=** { |
| 104 | 'A': {'AND': [('C', 'D')], 'OR': ['B']}, |
| 105 | 'B': {'OR': ['E', 'F']}, |
| 106 | 'C': {'OR': ['G'], 'AND': [('H', 'I')]}, |
| 107 | 'D': {'OR': ['J']} |
| 108 | } |
| 109 | optimal\_child\_group **=** {} |
| 110 | optimal\_cost **=** recAOStar('A') |
| 111 | print('Nodes which gives optimal cost are') |
| 112 | print\_path('A') |
| E1x1p3a | npdriinngt(N'o\dneO:ptAimal Cost is :: ', optimal\_cost) |

Nodes which gives optimal cost are CD->HI->J

Optimal Cost is :: 5

|  |  |
| --- | --- |
| 1 | *# Program 3* |
| 2 | **import** csv |
| 3 |  |
| 4 | **with** open("prog3.csv") **as** f: |
| 5 | csv\_file **=** csv.reader(f) |
| 6 | data **=** list(csv\_file) |
| 7 |  |
| 8 | specific **=** data[0][:**-**1] |
| 9 | general **=** [['?' **for** i **in** range(len(specific))] **for** j **in** range(len(specific))] |
| 10 |  |
| 11 | **for** i **in** data: |
| 12 | **if** i[**-**1] **==** "Yes": |
| 13 | **for** j **in** range(len(specific)): |
| 14 | **if** i[j] **!=** specific[j]: |
| 15 | specific[j] **=** "?" |
| 16 | general[j][j] **=** "?" |
| 17 |  |
| 18 | **elif** i[**-**1] **==** "No": |
| 19 | **for** j **in** range(len(specific)): |
| 20 | **if** i[j] **!=** specific[j]: |
| 21 | general[j][j] **=** specific[j] |
| 22 | **else**: |
| 23 | general[j][j] **=** "?" |
| 24 |  |
| 25 | print("\nStep " **+** str(data.index(i)**+**1) **+** " of Candidate Elimination Algorithm") |
| 26 | print(specific) |
| 27 | print(general) |
| 28 | gh **=** [] |
| 29 | **for** i **in** general: |
| 30 | **for** j **in** i: |
| 31 | **if** j **!=** '?': |
| 32 | gh.append(i) |
| 33 | **break** |
| 34 | print("\nFinal Specific hypothesis:\n", specific) |
| 35 | print("\nFinal General hypothesis:\n", gh) |

Step 1 of Candidate Elimination Algorithm

['sunny', 'warm', 'normal', 'strong', 'warm', 'same']

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Step 2 of Candidate Elimination Algorithm

['sunny', 'warm', '?', 'strong', 'warm', 'same']

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Step 3 of Candidate Elimination Algorithm

['sunny', 'warm', '?', 'strong', 'warm', 'same']

[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',

'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'same']]

Step 4 of Candidate Elimination Algorithm ['sunny', 'warm', '?', 'strong', '?', '?']

[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',

'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final Specific hypothesis:

['sunny', 'warm', '?', 'strong', '?', '?']

Final General hypothesis:

[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?']]

|  |  |
| --- | --- |
| 1 | *# Program 4* |
| 2 | **import** pandas **as** pd |
| 3 | **from** pprint **import** pprint |
| 4 | **from** sklearn.feature\_selection **import** mutual\_info\_classif |
| 5 | **from** collections **import** Counter |
| 6 |  |
| 7 | **def** id3(df, target\_attribute, attribute\_names, default\_class**=None**): |
| 8 | cnt**=**Counter(x **for** x **in** df[target\_attribute]) |
| 9 | **if** len(cnt)**==**1: |
| 10 | **return** next(iter(cnt)) |
| 11 |  |
| 12 | **elif** df.empty **or** (**not** attribute\_names): |
| 13 | **return** default\_class |
| 14 |  |
| 15 | **else**: |
| 16 | gainz **=** mutual\_info\_classif(df[attribute\_names],df[target\_attribute],discrete\_features**=True**) |
| 17 | index\_of\_max**=**gainz.tolist().index(max(gainz)) |
| 18 | best\_attr**=**attribute\_names[index\_of\_max] |
| 19 | tree**=**{best\_attr:{}} |
| 20 | remaining\_attribute\_names**=**[i **for** i **in** attribute\_names **if** i**!=**best\_attr] |
| 21 |  |
| 22 | **for** attr\_val, data\_subset **in** df.groupby(best\_attr): |
| 23 | subtree**=**id3(data\_subset, target\_attribute, remaining\_attribute\_names,default\_class) |
| 24 | tree[best\_attr][attr\_val]**=**subtree |
| 25 |  |
| 26 | **return** tree |
| 27 | df**=**pd.read\_csv("prog4.csv") |
| 28 |  |
| 29 | attribute\_names**=**df.columns.tolist() |
| 30 | print("List of attribut name") |
| 31 |  |
| 32 | attribute\_names.remove("PlayTennis") |
| 33 |  |
| 34 | **for** colname **in** df.select\_dtypes("object"): |
| 35 | df[colname], \_ **=** df[colname].factorize() |
| 36 |  |
| 37 | print(df) |
| 38 |  |
| 39 | tree**=** id3(df,"PlayTennis", attribute\_names) |
| 40 | print("The tree structure") |
| 41 | pprint(tree) |

List of attribut name

outlook temp humidity windy PlayTennis

1. 0 0 0 False 0
2. 0 0 0 True 0
3. 1 0 0 False 1
4. 2 1 0 False 1
5. 2 2 1 False 1
6. 2 2 1 True 0
7. 1 2 1 True 1
8. 0 1 0 False 0
9. 0 2 1 False 1
10. 2 1 1 False 1
11. 0 1 1 True 1
12. 1 1 0 True 1
13. 1 0 1 False 1
14. 2 1 0 True 0

The tree structure

{'outlook': {0: {'humidity': {0: 0, 1: 1}},

1: 1,

2: {'windy': {False: 1, True: 0}}}}

|  |  |
| --- | --- |
| 1 | *# Program 5* |
| 2 | **import** numpy **as** np |
| 3 |  |
| 4 | X **=** np.array(([2, 9], [1, 5], [3, 6]), dtype**=**float) |
| 5 | y **=** np.array(([92], [86], [89]), dtype**=**float) |
| 6 |  |
| 7 | *# scale units* |
| 8 | X **=** X**/**np.amax(X, axis**=**0) |
| 9 | y **=** y**/**100 |
| 10 |  |
| 11 | **class** Neural\_Network(object): |
| 12 | **def** init (self): |
| 13 |  |
| 14 | self.inputSize **=** 2 |
| 15 | self.outputSize **=** 2 |
| 16 | self.hiddenSize **=** 4 |
| 17 | self.W1 **=** np.random.randn(self.inputSize, self.hiddenSize) |
| 18 | self.W2 **=** np.random.randn(self.hiddenSize, self.outputSize) |
| 19 |  |
| 20 | **def** forward(self, X): |
| 21 | self.z **=** np.dot(X, self.W1) |
| 22 | self.z2 **=** self.sigmoid(self.z) |
| 23 | self.z3 **=** np.dot(self.z2, self.W2) |
| 24 | o **=** self.sigmoid(self.z3) |
| 25 | **return** o |
| 26 |  |
| 27 | **def** sigmoid(self, s): |
| 28 | **return** 1**/**(1**+**np.exp(**-**s)) |
| 29 |  |
| 30 | **def** sigmoidPrime(self, s): |
| 31 | **return** s **\*** (1 **-** s) |
| 32 |  |
| 33 | **def** backward(self, X, y, o): |
| 34 | self.o\_error **=** y **-** o |
| 35 | self.o\_delta **=** self.o\_error**\***self.sigmoidPrime(o) |
| 36 | self.z2\_error **=** self.o\_delta.dot(self.W2.T) |
| 37 | self.z2\_delta **=** self.z2\_error**\***self.sigmoidPrime(self.z2) |
| 38 | self.W1 **+=** X.T.dot(self.z2\_delta) |
| 39 | self.W2 **+=** self.z2.T.dot(self.o\_delta) |
| 40 |  |
| 41 | **def** train (self, X, y): |
| 42 | o **=** self.forward(X) |
| 43 | self.backward(X, y, o) |
| 44 |  |
| 45 | NN **=** Neural\_Network() |
| 46 | print ("\nInput: \n" **+** str(X)) |
| 47 | print ("\nActual Output: \n" **+** str(y)) |
| 48 | print ("\nPredicted Output: \n" **+** str(NN.forward(X))) |
| 49 | print ("\nLoss: \n" **+** str(np.mean(np.square(y **-** NN.forward(X))))) |

Input:

[[0.66666667 1. ]

[0.33333333 0.55555556]

[1. 0.66666667]]

Actual Output:

[[0.92]

[0.86]

[0.89]]

Predicted Output:

[[0.49211237 0.37794696]

[0.50715326 0.34596148]

[0.48104659 0.40879093]]

Loss:

0.21074179886944763

|  |  |
| --- | --- |
| 1 | *# Program 6* |
| 2 | *# Program 6* |
| 3 | **import** pandas **as** pd |
| 4 | **from** sklearn.preprocessing **import** LabelEncoder |
| 5 | **from** sklearn.naive\_bayes **import** GaussianNB |
| 6 |  |
| 7 | data **=** pd.read\_csv('prog6.csv') |
| 8 | print("The first 5 Values of data is :\n", data.head()) |
| 9 | X **=** data.iloc[:, :**-**1] |
| 10 | print("\nThe First 5 values of the train data is\n", X.head()) |
| 11 | y **=** data.iloc[:, **-**1] |
| 12 | print("\nThe First 5 values of train output is\n", y.head()) |
| 13 |  |
| 14 | le\_outlook **=** LabelEncoder() |
| 15 | X.Outlook **=** le\_outlook.fit\_transform(X.Outlook) |
| 16 | le\_Temperature **=** LabelEncoder() |
| 17 | X.Temperature **=** le\_Temperature.fit\_transform(X.Temperature) |
| 18 | le\_Humidity **=** LabelEncoder() |
| 19 | X.Humidity **=** le\_Humidity.fit\_transform(X.Humidity) |
| 20 | le\_Windy **=** LabelEncoder() |
| 21 | X.Windy **=** le\_Windy.fit\_transform(X.Windy) |
| 22 |  |
| 23 | print("\nNow the Train output is\n", X.head()) |
| 24 |  |
| 25 | le\_PlayTennis **=** LabelEncoder() |
| 26 | y **=** le\_PlayTennis.fit\_transform(y) |
| 27 | print("\nNow the Train output is\n",y) |
| 28 |  |
| 29 | **from** sklearn.model\_selection **import** train\_test\_split |
| 30 | X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(X,y, test\_size **=** 0.20) |
| 31 | classifier **=** GaussianNB() |
| 32 | classifier.fit(X\_train, y\_train) |
| 33 |  |
| 34 | **from** sklearn.metrics **import** accuracy\_score |
| 35 | print("Accuracy is:", accuracy\_score(classifier.predict(X\_test), y\_test)) |

The first 5 Values of data is :

Outlook Temperature Humidity Windy PlayTennis

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | Sunny | Hot | High | False | No |
| 1 | Sunny | Hot | High | True | No |
| 2 | Overcast | Hot | High | False | Yes |
| 3 | Rainy | Mild | High | False | Yes |
| 4 | Rainy | Cool | Normal | False | Yes |

The First 5 values of the train data is Outlook Temperature Humidity Windy

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | Sunny | Hot | High | False |
| 1 | Sunny | Hot | High | True |
| 2 | Overcast | Hot | High | False |
| 3 | Rainy | Mild | High | False |
| 4 | Rainy | Cool | Normal | False |

The First 5 values of train output is

1. No
2. No
3. Yes
4. Yes
5. Yes

Name: PlayTennis, dtype: object

Now the Train output is

Outlook Temperature Humidity Windy

0 2 1 0 0

1 2 1 0 1

2 0 1 0 0

3 1 2 0 0

4 1 0 1 0

Now the Train output is

[0 0 1 1 1 0 1 0 1 1 1 1 1 0]

Accuracy is: 0.6666666666666666

|  |  |
| --- | --- |
| 1 | *# Program 7* |
| 2 | **from** sklearn **import** datasets |
| 3 | **from** sklearn **import** metrics |
| 4 | **from** sklearn.cluster **import** KMeans |
| 5 | **from** sklearn.model\_selection **import** train\_test\_split |
| 6 |  |
| 7 | iris **=** datasets.load\_iris() |
| 8 | X\_train,X\_test,y\_train,y\_test **=** train\_test\_split(iris.data,iris.target) |
| 9 | model **=**KMeans(n\_clusters**=**3) |
| 10 | model.fit(X\_train,y\_train) |
| 11 | model.score |
| 12 | print('K-Mean: ',metrics.accuracy\_score(y\_test,model.predict(X\_test))) |
| 13 |  |
| 14 | **from** sklearn.mixture **import** GaussianMixture |
| 15 | model2 **=** GaussianMixture(n\_components**=**3) |
| 16 | model2.fit(X\_train,y\_train) |
| 17 | model2.score |
| 18 | print('EM Algorithm:',metrics.accuracy\_score(y\_test,model2.predict(X\_test))) |

K-Mean: 0.02631578947368421

EM Algorithm: 0.9736842105263158

In [13]:

|  |  |
| --- | --- |
| 1 | *# Program 8* |
| 2 | **from** sklearn.datasets **import** load\_iris |
| 3 | iris **=** load\_iris() |
| 4 |  |
| 5 | **from** sklearn.model\_selection **import** train\_test\_split |
| 6 | x\_train, x\_test, y\_train, y\_test **=** train\_test\_split(iris.data,iris.target,random\_state**=**0) |
| 7 |  |
| 8 | **from** sklearn.neighbors **import** KNeighborsClassifier |
| 9 | knn **=** KNeighborsClassifier(n\_neighbors **=** 5) |
| 10 | knn.fit(x\_train,y\_train) |
| 11 |  |
| 12 | **for** i,item **in** enumerate(x\_test): |
| 13 | prediction **=** knn.predict([item]) |
| 14 | print("Actual : ", iris['target\_names'][y\_test[i]]) |
| 15 | print("Prediction : ", iris['target\_names'][prediction], " \n") |
| 16 | print("Classification Accuracy : ",knn.score(x\_test,y\_test)) |

Actual : virginica



Prediction : ['virginica']

Actual : versicolor

Prediction : ['versicolor']

Actual : setosa

Prediction : ['setosa']

Actual : virginica

Prediction : ['virginica']

Actual : setosa

Prediction : ['setosa']

Actual : virginica

Prediction : ['virginica'] Actual : setosa

|  |  |
| --- | --- |
| 1 | *# Program 9* |
| 2 | **from** math **import** ceil |
| 3 | **import** numpy **as** np |
| 4 | **from** scipy **import** linalg |
| 5 |  |
| 6 | **def** lowess(x, y, f, iterations): |
| 7 | n **=** len(x) |
| 8 | r **=** int(ceil(f **\*** n)) |
| 9 | h **=** [np.sort(np.abs(x **-** x[i]))[r] **for** i **in** range(n)] |
| 10 | w **=** np.clip(np.abs((x[:, **None**] **-** x[**None**, :]) **/** h), 0.0, 1.0) |
| 11 | w **=** (1 **-** w **\*\*** 3) **\*\*** 3 |
| 12 | yest **=** np.zeros(n) |
| 13 | delta **=** np.ones(n) |
| 14 | **for** iteration **in** range(iterations): |
| 15 | **for** i **in** range(n): |
| 16 | weights **=** delta **\*** w[:, i] |
| 17 | b **=** np.array([np.sum(weights **\*** y), np.sum(weights **\*** y **\*** x)]) |
| 18 | A **=** np.array([[np.sum(weights), np.sum(weights **\*** x)],[np.sum(weights **\*** x), np.sum(weights **\*** x **\*** x)]]) |
| 19 | beta **=** linalg.solve(A, b) |
| 20 | yest[i] **=** beta[0] **+** beta[1] **\*** x[i] |
| 21 |  |
| 22 | residuals **=** y **-** yest |
| 23 | s **=** np.median(np.abs(residuals)) |
| 24 | delta **=** np.clip(residuals **/** (6.0 **\*** s), **-**1, 1) |
| 25 | delta **=** (1 **-** delta **\*\*** 2) **\*\*** 2 |
| 26 |  |
| 27 | **return** yest |
| 28 |  |
| 29 | **import** math |
| 30 | n **=** 100 |
| 31 | x **=** np.linspace(0, 2 **\*** math.pi, n) |
| 32 | y **=** np.sin(x) **+** 0.3 **\*** np.random.randn(n) |
| 33 | f **=**0.25 |
| 34 | iterations**=**3 |
| 35 | yest **=** lowess(x, y, f, iterations) |
| 36 |  |
| 37 | **import** matplotlib.pyplot **as** plt |
| 38 | plt.plot(x,y,"r.", color**=**"green") |
| 39 | plt.plot(x,yest,"b-",color**=**"red") |

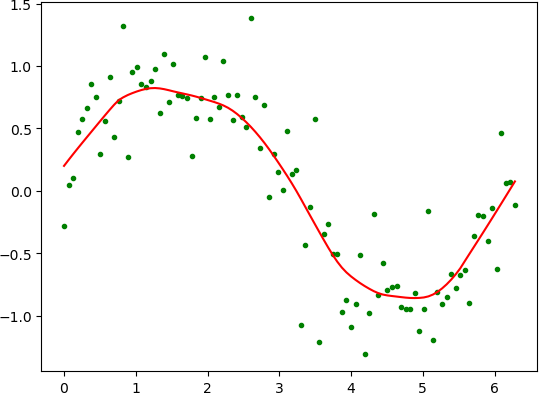
C:\Users\prsur\AppData\Local\Temp\ipykernel\_90004\1619400893.py:38: UserWarning: color is redundantly defined by the 'colo r' keyword argument and the fmt string "r." (-> color='r'). The keyword argument will take precedence.

plt.plot(x,y,"r.", color="green")

C:\Users\prsur\AppData\Local\Temp\ipykernel\_90004\1619400893.py:39: UserWarning: color is redundantly defined by the 'colo r' keyword argument and the fmt string "b-" (-> color='b'). The keyword argument will take precedence.

plt.plot(x,yest,"b-",color="red") Out[14]:

[<matplotlib.lines.Line2D at 0x1d320484370>]



In [ ]:

1