CS 6375 HW 4:

Sample_1_MLC_2022

Parker Whitehead

Part-1: Build solver

We will load the data and use it for building our solver

Download data

```
!wget https://www.ics.uci.edu/~dechter/uaicompetition/2022/TuningBenchmarks/MLC.zip
!unzip /content/MLC.zip
```

```
--2022-12-05 09:30:00--  https://www.ics.uci.edu/~dechter/uaicompetition/2022/TuningBenchmarks/MLC.zip
Resolving www.ics.uci.edu (www.ics.uci.edu)... 128.195.1.88
Connecting to www.ics.uci.edu (www.ics.uci.edu)|128.195.1.88|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 51653514 (49M) [application/zip]
Saving to: ‘MLC.zip.1’

MLC.zip.1           100%[===================>]  49.26M  16.8MB/s    in 2.9s

2022-12-05 09:30:04 (16.8 MB/s) - ‘MLC.zip.1’ saved [51653514/51653514]

Archive:  /content/MLC.zip
replace MLC/Sample_1_MLC_2022.data? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

Import libraries

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.multioutput import MultiOutputClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
```

Load data

The data consists of three sets of variables:

1. Evidence (observed) variables - X
2. Hidden variables - H
3. Query variables - Y

The data loader class *Data* reads the data and partitions it accordingly.

Helper functions:

`convertToXY()` : This function returns (X, Y) from the .data file

```
class Data:
  #fpath: File path of the .data file

  #self.evid_var_ids: Contains the indices of the observed variables
  #self.query_var_ids: Contains the indices of the query variables
  #self.hidden_var_ids: Contains the indices of the hidden variables

  #self.evid_assignments: Assignments to evid variables
  #self.query_assignments: Assignments to query variables
  #self.weights: Pr(e, q)
  def __init__(self, fpath):

    f = open(fpath, "r")

    self.nvars = int(f.readline()) #1
    line = np.asarray(f.readline().split(), dtype=np.int32)#2
    self.evid_var_ids = line[1:]
    evid_indices = range(1, self.evid_var_ids.shape[0]*2, 2)

    line = np.asarray(f.readline().split(), dtype=np.int32) #3
    self.query_var_ids = line[1:]
    query_indices = range(self.evid_var_ids.shape[0]*2+1, (self.evid_var_ids.shape[0]+self.query_var_ids.shape[0])*2, 2)

    line = np.asarray(f.readline().split(), dtype=np.int32)#4
    self.hidden_var_ids = line[1:]

    line = f.readline()#5
    self.nproblems = int(f.readline())#6

    self.evid_assignments = []
    self.query_assignments = []
    self.weights = []
    for i in range(self.nproblems):
      line = np.asarray(f.readline().split(), dtype=float)
      self.evid_assignments.append(np.asarray(line[evid_indices], dtype=np.int32))
      self.query_assignments.append(np.asarray(line[query_indices], dtype=np.int32))
      self.weights.append(line[-1])
    self.evid_assignments = np.asarray(self.evid_assignments)
    self.query_assignments = np.asarray(self.query_assignments)
    self.weights = np.asarray(self.weights)

  def convertToXY(self):
    return (self.evid_assignments, self.query_assignments)

  def convertResults(self, query_predictions):
    out = np.zeros((query_predictions.shape[0], 1+2*self.query_var_ids.shape[0]), dtype=int)
    out[:, 2::2] = query_predictions[:, :]
    out[:, 1::2] = self.query_var_ids
    out[:, 0] = self.query_var_ids.shape[0]
    return out

data_directory = '/content/MLC/'
dname = 'Sample_2_MLC_2022'
```

```
f =open(data_directory+dname+'.data','r')
nvars = int(f.readline())
line = np.asarray(f.readline().split(), dtype=np.int32)
```

```
f =open(data_directory+dname+'.data','r')
x=f.readlines()
```

```
len(x)
```

```
    10006
```

```
evid_var_ids = line[1:]
evid_var_ids.shape
```

```
    (400,)
```

```
data = Data(data_directory+dname+'.data')
```

```
#Getting Evidence and Query data into X, y

X, y = data.convertToXY()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
```

Defining UAI Helper Classes

These classes utilize the fact that we are aware of the Markov Network that generates our dataset. As such, we can calculate the probability of any given q based on all of our known X values, using the Markov Network.

```
import numpy as np
from dataclasses import dataclass
```

```
@dataclass
class UAIFunction:
    # Number of variables participating in the function
    size: int
    # List of variables s.t. that the index is the order of the variables.
    indexes: list()
    # List of values for each permutation of variable values.
    values: list()

class UAIHelper:
    def __init__(self, fpath):
        with open(fpath,'r') as f:
            # Ignore type; will always be markov, for our case.
            f.readline()
            self.num_vars = int(f.readline())
            # according to prof, variables will always be binary, so we can ignore 3rd line.
            f.readline()

            # Ignore whitespace line.
            f.readline()
            self.num_functions = int(f.readline())
            # Each index contains a set!
            self.var_function_participation = np.array([set() for _ in range(self.num_vars)])
            self.functions = np.array([UAIFunction(0,[],[]) for _ in range(self.num_functions)]).astype

            for i in range(self.num_functions):
                line = f.readline().replace('\n',' ').split(' ')
                vars = int(line[0])
                indexes = []
                for j in range(vars):
                    indexes.append(int(line[1+j]))
                    self.var_function_participation[indexes[j]].add(i)
                self.functions[i].size = vars
                self.functions[i].indexes = indexes

            f.readline()

            for i in range(self.num_functions):
                line = f.readline().replace('\n',' ').split(' ')
                num_values = int(line[0])
                values = []
                for j in range(num_values):
                    values.append(float(line[1+j]))
                self.functions[i].values = values

    def createDataMatrix(self, X, Q_size, X_indexes, Q_indexes):

        # create set of all included X indexes.
        all_x_indexes = set()
        for x_i in X_indexes:
            all_x_indexes.add(x_i)

        # create array of var name to index
        name_to_index = np.zeros(self.num_vars).astype(int)
        for index, value in enumerate(X_indexes):
            name_to_index[value] = index

        # create new matrix of size (x.shape[0], x.shape[1]+q.shape[1])
        new_X = np.zeros((X.shape[0],X.shape[1]+Q_size),dtype=float)
        # for each row:
        for i in range(X.shape[0]):
            # fill in all begining entries with x.
            for j in range(X.shape[1]):
                new_X[i,j] = X[i,j]

            # for each q in Q:
            for j in range(Q_size):
                q_index = Q_indexes[j]
                # for every function q is a part of:
                weight_true = 1
                weight_false = 1
                for function_index in self.var_function_participation[q_index]:
                    pass
                    # calculate weight multiplication sum of true & false
                    current_function = self.functions[function_index]

                    variable_table_index = 0
                    q_index_modifier = 0
                    q_index_modifier = 0

                    # Find the index of the false state & the modifier to get the true state.
                    for counter, f_variable_index in enumerate(current_function.indexes):
                        # if all variable participants present, use; otherwise, ignore function.
                        if f_variable_index not in all_x_indexes and f_variable_index != q_index:
                            continue # if you don't have sufficient metrix to calculate, ignore functio
                        if f_variable_index != q_index and X[i,name_to_index[int(f_variable_index)]] ==
                            variable_table_index += len(current_function.values) / 2**(counter+1)
                        elif f_variable_index == q_index:
                            q_index_modifier = len(current_function.values) / 2**(counter+1)

                    # then calculate the true and false weights for q.
                    weight_true*=current_function.values[int(variable_table_index + q_index_modifier)]
                    weight_false*=current_function.values[int(variable_table_index)]
                # Calculate weight_true / (weight_true + weight_false)
                weight_final = weight_true / (weight_true + weight_false)
                # Add this to the matrix.
                new_X[i,j + X.shape[1]] = weight_final

        # return new matrix and use!
        return new_X
```

Load UAI Helper and Create new input matrix for LogReg

```
uai_helper = UAIHelper(data_directory+dname+'.uai')

new_X_train = uai_helper.createDataMatrix(X_train, len(data.query_var_ids), data.evid_var_ids, data.query_var_ids)
```

Train solver: Logistic Regression

[ ] ↳ *1 cell hidden*

Create new_X_test

[ ] ↳ *1 cell hidden*

Predict Query Assignments

[ ] ↳ *5 cells hidden*

Part-2: Test solver

Once we have trained the solver, we want to test how good it is.

For a given evidence $E = e$, let $Q = \hat{q}$ denote the solver prediction and $Q = q$ denote the ground truth value.

$$Err = log\frac{\prod_{i\in Data} Pr(e^{(i)}, q^{(i)})}{\prod_{i\in Data} Pr(e^{(i)}, \hat{q}^{(i)})}$$

Let $MaxErr$ denote the Err for a trivial solver. Then,

$$Score = max(0, 100(1 - \frac{Err}{MaxErr}))$$

Using Random Forests as the trivial solver

[ ] ↳ *2 cells hidden*

Load Variable Elimination Code

[ ] ↳ *4 cells hidden*

Read the Markov network

[ ] ↳ *5 cells hidden*

Compute $log_{10} Pr(X, y)$

```
def computeLogProb(X, y):
  out = np.zeros(X.shape[0])
  for i in range(X.shape[0]):
    for j in range(X.shape[1]):
      mn.setEvidence(data.evid_var_ids[j], X[i][j])
    for j in range(y.shape[1]):
      mn.setEvidence(data.query_var_ids[j], y[i][j])
    btp = BTP(mn, order)
    out[i] = np.log10(btp.getPR())
  return out
```

Compute error and score

```
def computeErr(true_ll, pred_ll):
  return np.sum(true_ll)-np.sum(pred_ll)
```

```
def computeScore(err, max_err):
  return np.max((0, 100*(1.0-err/max_err)))
```

```
y_pred = np.loadtxt(data_directory+dname+'.pred', dtype=int, delimiter=' ')[:, 1:][:, 1::2]
ntest = 10
lprob_true = computeLogProb(X_test[:ntest, :], y_test[:ntest, :])
lprob_pred = computeLogProb(X_test[:ntest, :], y_pred[:ntest, :])
lprob_trivial = computeLogProb(X_test[:ntest, :], y_trivial[:ntest, :])

err = computeErr(lprob_true, lprob_pred)
maxErr = computeErr(lprob_true, lprob_trivial)
```

```
print(err, maxErr)

     0.0006200212214935164 50.04612987032942
```

```
print("Score:", computeScore(err, maxErr))

     Score: 99.99876110056242
```

```
print(err, maxErr)

     0.0006200212214935164 50.04612987032942
```

```
!pip freeze

     absl-py==1.3.0
     aeppl==0.0.33
     aesara==2.7.9
     aiohttp==3.8.3
     aiosignal==1.3.1
     alabaster==0.7.12
     albumentations==1.2.1
     altair==4.2.0
     appdirs==1.4.4
     arviz==0.12.1
     astor==0.8.1
     astropy==4.3.1
     astunparse==1.6.3
     async-timeout==4.0.2
     asynctest==0.13.0
     atari-py==0.2.9
     atomicwrites==1.4.1
     attrs==22.1.0
     audioread==3.0.0
     autograd==1.5
     Babel==2.11.0
     backcall==0.2.0
     beautifulsoup4==4.6.3
     bleach==5.0.1
```

```
blis==0.7.9
bokeh==2.3.3
branca==0.6.0
bs4==0.0.1
CacheControl==0.12.11
cached-property==1.5.2
cachetools==5.2.0
catalogue==2.0.8
certifi==2022.9.24
cffi==1.15.1
cftime==1.6.2
chardet==3.0.4
charset-normalizer==2.1.1
click==7.1.2
clikit==0.6.2
cloudpickle==1.5.0
cmake==3.22.6
cmdstanpy==1.0.8
colorcet==3.0.1
colorlover==0.3.0
community==1.0.0b1
confection==0.0.3
cons==0.4.5
contextlib2==0.5.5
convertdate==2.4.0
crashtest==0.3.1
crcmod==1.7
cufflinks==0.17.3
cvxopt==1.3.0
cvxpy==1.2.2
cycler==0.11.0
```

✓ 1s completed at 3:58 AM