Assignment 3 report

June 18, 2018

Assignment 3
Mahsa Daneshmandmehrabani
V00892119
Question 1: Constraint-satisfaction problems
problem of Waltz Filtering as a CSP problem:
Variables: edges of the given polyhedron
Domain: labels of the edges from the set {left arrow, right arrow, positive, negative}
Constaints: junctions (L-junction, Arrow-junction and Fork-junction) I used the constraint module in Python to write this problem as a CSP problem and find the valid solutions. The 3 main methods of this module that I used are **addVariable**, **addConstraint** and **getSolutions**.

addVariable: addVariable(self, variable, domain)
Add a variable to the problem
addConstraint: addConstraint(self, constraint, variables)
Add a constraint to the problem

To define the constraints, I used the **FunctionConstraint** type; constraint which wraps a function defining the constraint logic. Therefore, I defined 3 there functions: L_Junction, Arrow_Junction and Fork_Junction and called them inside addConstraint. These there functions get the edges as their inputs and return one of the possible assignment of labels to edges.

getSolutions: Return all solutions for the given problem

```
In [16]: from constraint import *
         #create a prblem
         problem = Problem()

         #creating variables which are edges with their domain(labels of te edges) which
         #is the set {0,1,2,3}. 0: Left arrow, 1: Right arrow, 2: Negative, 3: Positive
         problem.addVariable("E1", [0,1,2,3])
         problem.addVariable("E2", [0,1,2,3])
         problem.addVariable("E3", [0,1,2,3])
         problem.addVariable("E4", [0,1,2,3])
         problem.addVariable("E5", [0,1,2,3])
         problem.addVariable("E6", [0,1,2,3])
         problem.addVariable("E7", [0,1,2,3])
         problem.addVariable("E8", [0,1,2,3])
         problem.addVariable("E9", [0,1,2,3])
         problem.addVariable("E10", [0,1,2,3])
         problem.addVariable("E11", [0,1,2,3])
```

```
problem.addVariable("E12", [0,1,2,3])
problem.addVariable("E13", [0,1,2,3])
problem.addVariable("E14", [0,1,2,3])
problem.addVariable("E15", [0,1,2,3])

#Define L-Junction constraint function
def L_Junction(x, y):
    #Equivalent to the class L mentioned in the paper
    return x == 1 and y == 3 or
    x == 1 and y == 1 or
    x == 3 and y == 1 or
    x == 0 and y == 2 or
    x == 0 and y == 0 or
    x == 2 and y == 0

#Define Arrow-Junction constraint function
def Arrow_Junction(x, y, z):
    #Equivalent to the class Arrow mentioned in the paper
    return x == 0 and y == 3 and z == 0 or
    x == 2 and y == 3 and z == 2 or
    x == 3 and y == 2 and z == 3

#Define Fork-Junction constraint function
def Fork_Junction(x, y, z):
    #Equivalent to the class Fork mentioned in the paper
    return x == 0 and y == 0 and z == 2 or
    x == 2 and y == 0 and z == 0 or
    x == 0 and y == 2 and z == 0 or
    x == 3 and y == 3 and z == 3 or
    x == 2 and y == 2 and z == 2


#Adding L_Junction constraints:

#E1 and E2 form a L-junction
problem.addConstraint(FunctionConstraint(L_Junction), ["E1", "E2"])
#E5 and E6 form a L-junction
problem.addConstraint(FunctionConstraint(L_Junction), ["E5", "E6"])
#E7 and E8 form a L-junction
problem.addConstraint(FunctionConstraint(L_Junction), ["E7", "E8"])

#Adding Fork_Junction constraints:

#E4, E5 and E14 form a Fork-junction
problem.addConstraint(FunctionConstraint(Fork_Junction), ["E4", "E3", "E14"])
#E9, E11 and E10 form a Fork-junction
problem.addConstraint(FunctionConstraint(Fork_Junction), ["E9", "E11","E10"])
#E12, E13 and E15 form a Fork-junction
```

```
problem.addConstraint(FunctionConstraint(Fork_Junction), ["E12", "E13", "E15"])

#Adding Arrow_Junction constraints:

#E8, E9 and E11 form a Fork-junction
problem.addConstraint(FunctionConstraint(Arrow_Junction), ["E8", "E9", "E1"])
#E2, E10 and E3 form a Fork-junction
problem.addConstraint(FunctionConstraint(Arrow_Junction), ["E2", "E10", "E3"])
#E4, E15 and E5 form a Fork-junction
problem.addConstraint(FunctionConstraint(Arrow_Junction), ["E4", "E15", "E5"])
#E6, E13 and E7 form a Fork-junction
problem.addConstraint(FunctionConstraint(Arrow_Junction), ["E6", "E13", "E7"])
#E12, E14 and E11 form a Fork-junction
problem.addConstraint(FunctionConstraint(Arrow_Junction), ["E12", "E14", "E11"])

#Get the solution with getSolutions
Solutions = problem.getSolutions()

for solution in Solutions:
     #print each solution
     print('solution: ', solution)
```

```
solution:  {'E1': 2, 'E2': 0, 'E10': 3, 'E3': 0, 'E11': 3, 'E9': 3, 'E8': 2, 'E7': 0, 'E12': 3
solution:  {'E1': 0, 'E2': 2, 'E10': 3, 'E3': 2, 'E11': 3, 'E9': 3, 'E8': 0, 'E7': 0, 'E12': 3
solution:  {'E1': 0, 'E2': 0, 'E10': 3, 'E3': 0, 'E11': 3, 'E9': 3, 'E8': 0, 'E7': 2, 'E12': 3
solution:  {'E1': 0, 'E2': 0, 'E10': 3, 'E3': 0, 'E11': 3, 'E9': 3, 'E8': 0, 'E7': 0, 'E12': 3
```

solution 1: {'E1': 2, 'E2': 0, 'E10': 3, 'E3': 0, 'E11': 3, 'E9': 3, 'E8': 2, 'E7': 0, 'E12': 3, 'E14': 2, 'E4': 0, 'E13': 3, 'E15': 3, 'E5': 0, 'E6': 0}

solution 2: {'E1': 0, 'E2': 2, 'E10': 3, 'E3': 2, 'E11': 3, 'E9': 3, 'E8': 0, 'E7': 0, 'E12': 3, 'E14': 2, 'E4': 2, 'E13': 3, 'E15': 3, 'E5': 2, 'E6': 0}

solution 3: {'E1': 0, 'E2': 0, 'E10': 3, 'E3': 0, 'E11': 3, 'E9': 3, 'E8': 0, 'E7': 2, 'E12': 3, 'E14': 2, 'E4': 0, 'E13': 3, 'E15': 3, 'E5': 0, 'E6': 2}

solution 4: {'E1': 0, 'E2': 0, 'E10': 3, 'E3': 0, 'E11': 3, 'E9': 3, 'E8': 0, 'E7': 0, 'E12': 3, 'E14': 2, 'E4': 0, 'E13': 3, 'E15': 3, 'E5': 0, 'E6': 0}

Question 3: Discrete Bayesian Networks I used the pgmpy module in Python to answer this question. In the followin, you can see the code to create the bayesian network for this proble.

```
In [17]: # -*- coding: utf-8 -*-
         """
         Created on Sat Jun 16 14:12:16 2018

         @author: mahsa
         """

         from pgmpy.models import BayesianModel
         from pgmpy.factors.discrete import JointProbabilityDistribution
```

```
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination

# Defining the model structure. We can define the network by just
#passing a list of edges. M: Musican , D: difficulity,
#R: Rating, E: Exam, L: Letter

BN = BayesianModel([('D', 'R'), ('M', 'R'), ('M', 'E'), ('R', 'L')])

# Defining individual CPDs.
cpd_d = TabularCPD(variable='D', variable_card=2, values=[[0.6, 0.4]])
cpd_m = TabularCPD(variable='M', variable_card=2, values=[[0.7, 0.3]])


cpd_r = TabularCPD(variable='R', variable_card=3,
                   values=[[0.3, 0.05, 0.9,  0.5],
                           [0.4, 0.25, 0.08, 0.3],
                           [0.3, 0.7,  0.02, 0.2]],
                   evidence=['M', 'D'],
                   evidence_card=[2, 2])

cpd_l = TabularCPD(variable='L', variable_card=2,
                   values=[[0.1, 0.4, 0.99],
                           [0.9, 0.6, 0.01]],
                   evidence=['R'],
                   evidence_card=[3])

cpd_e = TabularCPD(variable='E', variable_card=2,
                   values=[[0.95, 0.2],
                           [0.05, 0.8]],
                   evidence=['M'],
                   evidence_card=[2])

# Associating the CPDs with the network
BN.add_cpds(cpd_d, cpd_m, cpd_r, cpd_l, cpd_e)

BN.check_model()
```

```
Out[17]: True
```

Part 1: To calculate the join probability (probability that the candidate has strong musicianship, the pieces played are not that difficult, the rating is 2 stars, the exam score is high, and the recommendation letter is weak), we just need extract the probabilities from the CPDs. To do so, I convert the CPDs to numpy arrays that I can access the values easily.

```
In [18]: # Converting the cpd tables to numpy arrays
         d = cpd_d.values
         m = cpd_m.values
```

```
r = cpd_r.values
e = cpd_e.values
l = cpd_l.values

#calcultion for the joint probability
JPD = m[1] * d[0] * r[1][0][1] * e[1][1]* l[0][1]
print('p(M = strong) * p(D = low) * p(R = ** | M = strong, D = low) *
      'p(E = high|M = strong) * p(Letter = weak | R = **) = ',
      m[1],' * ',d[0],' * ',r[1][0][1],' * ',e[1][1],' * ',l[0][1],' = ',JPD)
```

```
p(M = strong) * p(D = low) * p(R = ** | M = strong, D = low) * p(E = high|M = strong) * p(Lette
```

Output:

p(M = strong) * p(D = low) * p(R = 2 star | M = strong, D = low) * p(E = high | M = strong) * p(Letter = weak | R = 2 star) = 0.3 * 0.6 * 0.25 * 0.8 * 0.4 = 0.0144

Part 2: To calculate the probability of George giving a strong recommendation letter(knowing nothing else), I used the Variable Elimination concept.

$$P(L = Strong) = \sum_{M,G,R,E} P(L|R) * P(E|I) * P(R|D,I) * P(D) * P(I)$$

This is done by query in pgmpy. It takes the variable which you want calculate its probability. You can see that this value for value strong (which is mapped to 1) is about %50.2.

```
In [19]: #Probability that george gives recommendation (for each possible value of letter) kno
         infer = VariableElimination(BN)
         print(infer.query(['L']) ['L'])
```

```
 L      phi(L)

 L_0    0.4977

 L_1    0.5023
```

Part 2: To calculate the probability of George giving a strong recommendation letter knowing George is not a strong musician, I used the Variable Elimination concept again. But in this case the variable M with the value weak (which is mapped to 0) is also given to the query method. You can see that this probability for value strong is about %38.

```
In [20]: #Probability that george gives a strong recommendation given he is a weak musician
         print(infer.query(['L'], evidence={'M': 0}) ['L'])
```

```
 L      phi(L)

 L_0    0.6114
```

```
L_1    0.3886
```

Part 3: Approximate inference for joint probability I used the Bayesian Model Samplers in pgmpy to generate random samples from my bayesian network. I used the forward_sample(size=1, return_type='dataframe') command that generates sample(s) from joint distribution of the bayesian network. Then among all these random samples, I count the number of samples in which, Musician = strong, Difficulty = low, Rating = 2 star, Exam = high and Letter = weak. Lets say this number is equal to N_{query} and N is the total number of generated samples. Then the estimated probability is:

$$P(query) = \frac{N_{query}}{N}$$

```
In [21]: from pgmpy.models.BayesianModel import BayesianModel
         from pgmpy.factors.discrete import TabularCPD
         from pgmpy.sampling import BayesianModelSampling

         inference = BayesianModelSampling(BN)
         #forward_sample generates random samples from joint distribution
         samples = inference.forward_sample(size=1000000, return_type = 'dataframe')

         # We are looking for P(M = strong, D = low, R = **, E = high, L = weak) =
         # = P(M = 1, D = 0, R = 1, E = 1, L = 0)

         #Define a counter to count the number of samples that matches our query
         #P(M = 1, D = 0, R = 1, E = 1, L = 0)

         jpd_conter = 0
         #samples.iloc[4][0]
         #samples.shape[0]
         for i in range(samples.shape[0]):
             #checking if it is the query
             if samples.iloc[i][0] == 1 and samples.iloc[i][1] == 1 and
             samples.iloc[i][2] == 0  and samples.iloc[i][3] == 1
             and samples.iloc[i][4] == 0:
                 jpd_conter = jpd_conter + 1

         print(jpd_conter)
         p_extimate = float(jpd_conter / 1000000)
         print(p_extimate)
```

```
4529
0.004529
```

Part 3: Approximate inference for conditional probability

I used the rejection_sample(evidence=None, size=1, return_type='dataframe') method in pgmpy that generates sample(s) from joint distribution of the bayesian network, given the evidence. For the first probability no evidence is needed to provide. For the seconde probability the M variable with value weak (0) is given to the function. The approximate probabilities are different from the actual probabilities since it's a random process and needs many iterations to converge to the actual probabilities.

```
In [6]: from pgmpy.models.BayesianModel import BayesianModel
        from pgmpy.factors.discrete import TabularCPD
        from pgmpy.factors.discrete import State
        from pgmpy.sampling import BayesianModelSampling


        inference = BayesianModelSampling(BN)
        samples = inference.rejection_sample(evidence = None, size=100,
                                             return_type='dataframe')
        #print(samples)

        counter = 0
        for i in range(samples.shape[0]):
            #Checking for M = strong in each saample, if it is strong (1) then
            #increment the counter by 1
            if (samples.iloc[i][4] == 1) :
                counter = counter + 1

        print(counter)
        p = float(counter / 100)
        print('Estimate of probability of Goerge gives a strong letter is: ', p)
```

```
47
An estimate of probability of Goerge gives a strong recommendation letter is:  0.47
```

Output:
An estimate of probability of Goerge gives a strong recommendation letter is: 0.47

```
In [22]: from pgmpy.models.BayesianModel import BayesianModel
         from pgmpy.factors.discrete import TabularCPD
         from pgmpy.factors.discrete import State
         from pgmpy.sampling import BayesianModelSampling


         inference = BayesianModelSampling(BN)
         evidence = [State(var='M', state=0)]
         samples = inference.rejection_sample(evidence = evidence, size=100, return_type='data
         #print(samples)

         counter2 = 0
         for i in range(samples.shape[0]):
```

```
            if (samples.iloc[i][4] == 1) :
                counter2 = counter2 + 1

        print(counter2)
        P = float(counter2 / 100)
        print('An estimate of probability of Goerge gives a strong
              'recommendation letter given he is a weak musician is: ', P)
```

```
38
An estimate of probability of Goerge gives a strong recommendation letter given he is a weak mu
```

Output:

38 An estimate of probability of Goerge gives a strong recommendation letter given he is a weak musician is: 0.38

Question 2: Naive Bayes Text Classification using scikitlearn

Naive Bayes Bernoulli:

I defined two functions **FeatureNeg** and **FeaturePos** that create the binary matrix for negative and positive files. For each file that they open, the function first create an all zero matrix that later its entries will be updated to 1 if the keyword exist in the file contents. Then the function Featurematrix is called to concatenate these two matrices. Then in the main part of the program, after creating an object of the class classifier, these three functions are called on it, then we will get the binary matrix of the data. Then I used **StratifiedKFold** and **BernoulliNB** from the scikit learn module to calculate the accuracy scores of the cross validation.

```
In [190]: # -*- coding: utf-8 -*-
          """
          Created on Sun Jun 17 15:51:02 2018

          @author: mahsa
          """


          # -*- coding: utf-8 -*-
          """
          Created on Mon Jun  4 15:16:01 2018

          @author: mahsa
          """
          import os
          import numpy as np
          from os import listdir
          from os.path import isfile, join
          import glob
          import errno
          from sklearn.naive_bayes import BernoulliNB
          from sklearn.naive_bayes import MultinomialNB
          from sklearn.model_selection import KFold
          from sklearn.model_selection import StratifiedKFold
```

```
from sklearn.metrics import accuracy_score
from collections import Counter



class Classifier():


    def __init__(self):

        #vacaulary containing the words that we are interested to calculate
        #their accurance or absence probabilities
        self.vocabulary = ["awful","bad","boring","dull","effective","enjoyable",
                           "great","hilarious"]




    def FeatureNeg(self, path):

        self.path = path
        files = glob.glob(path)
        self.matrixNeg = np.zeros((1000, 9))
        for row in range(self.matrixNeg.shape[0]):
            #last row of the feature matrix for negative files is the
            #label 0 representing negative class
            self.matrixNeg[row][8] = 0

        for keyword in self.vocabulary:
            for f in files:
                with open(f) as file:
                    self.contents = file.read().strip().split()
                    if keyword in self.contents:
                        i = int(files.index(f))
                        j = int(self.vocabulary.index(keyword))
                        #checks if the keyword exists in the file,
                        #if yes it changes the corresponding entry to 1
                        self.matrixNeg[i][j] = 1

        return self.matrixNeg


    def FeaturePos(self, Path):

        self.Path = Path
        Files = glob.glob(Path)
        self.matrixPos = np.zeros((1000, 9))
```

```
        for r in range(self.matrixPos.shape[0]):
            #last row of the feature matrix for positive files
            #is the label 1 representing positive class
            self.matrixPos[r][8] = 1

        for Keyword in self.vocabulary:
            for f in Files:
                with open(f) as file:
                    self.Contents = file.read().strip().split()
                    if Keyword in self.Contents:
                        i = int(Files.index(f))
                        j = int(self.vocabulary.index(Keyword))
                        self.matrixPos[i][j] = 1

        return self.matrixPos



    def Featurematrix(self, M, N):
        self.Featurematrix = np.concatenate((M, N))
        return self.Featurematrix



if __name__ == '__main__':
    myclassifier = Classifier()


    FeatureNegative = myclassifier.FeatureNeg(r'C:\Users\mahsa\
    OneDrive\Desktop\Assignment 3_AI\txt_sentoken\neg\*.txt')
    FeaturePositive = myclassifier.FeaturePos(r'C:\Users\mahsa\
    OneDrive\Desktop\Assignment 3_AI\txt_sentoken\pos\*.txt')
    #Binary matrix of the data
    BinaryMatrix = myclassifier.Featurematrix(FeatureNegative,
                                              FeaturePositive)
    np.random.shuffle(BinaryMatrix)
    #the last row are the labels so we take it as Y (labels)
    Y = BinaryMatrix[:,8]
    X = np.delete(BinaryMatrix, -1 , axis=1)



    # Bernoulli Naive Bayes
    Accuracy = []
    #creating an object from StratifiedKFold with 10 splits
    kf = StratifiedKFold(n_splits=10)
    StratifiedKFold(n_splits=10, random_state = None, shuffle = False)
```

```
    for train, test in kf.split(X, Y):
        X_train = X[train]
        X_test  = X[test]
        Y_train = Y[train]
        Y_test  = Y[test]
        #create a Bernoulli model
        clf = BernoulliNB()
        clf.fit(X_train, Y_train)
        #predicts the labels of test data
        Y_predict = clf.predict(X_test)
        Accuracy.append(clf.score(X_test, Y_test))

    print('Cross Validation accuracy using Bernoulli Naive Bayes: '
          , Accuracy)
```

```
Cross Validation accuracy using Bernoulli Naive Bayes:  [0.68500000000000005, 0.689999999999999
```

The complete accuracy from Bernouli model:

Cross Validation accuracy using Bernoulli Naive Bayes: [0.68500000000000005, 0.68999999999999995, 0.67000000000000004, 0.67000000000000004, 0.625, 0.64000000000000001, 0.625, 0.75, 0.68500000000000005, 0.69999999999999996]

Naive Bayes Multinomial classifier:

This classifier is also similar to Bernouli classifier, but instead of just checking if the keyword exist in the file or not, it counts the frequency of the keyword in the file. Therefore, I defined **FreqNeg** and **FreqPos** that return the frequency matrices for negative and positive classes. The entires of these matrices are the frequency of one those 8 keywords in the corresponding review.

```
In [189]: # -*- coding: utf-8 -*-
          """
          Created on Sun Jun 17 15:51:02 2018

          @author: mahsa
          """


          # -*- coding: utf-8 -*-
          """
          Created on Mon Jun  4 15:16:01 2018

          @author: mahsa
          """
          import os
          import numpy as np
          from os import listdir
          from os.path import isfile, join
          import glob
```

```
import errno
from sklearn.naive_bayes import BernoulliNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import KFold
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score
from collections import Counter



class Classifier():


    def __init__(self):

        #vacaulary containing the words that we are interested to calculate their ac
        self.vocabulary = ["awful","bad","boring","dull","effective",
                           "enjoyable","great","hilarious"]




    def Featurematrix(self, M, N):
        self.Featurematrix = np.concatenate((M, N))
        return self.Featurematrix



    #Frequency matrix for negative files
    def FreqNeg(self, path):

        self.path = path
        files = glob.glob(path)
        self.matrixNeg = np.zeros((1000, 9))
        for row in range(self.matrixNeg.shape[0]):
            #last row of the feature matrix for negative files
            #is the label 0 representing negative class
            self.matrixNeg[row][8] = 0

        #for keyword in self.vocabulary:

        for f in files:
                with open(f) as file:
                    self.contents = file.read().strip().split()
                    self.c = Counter(self.contents)
                    for keyword in self.vocabulary:
```

```
                        i = int(files.index(f))
                        j = int(self.vocabulary.index(keyword))
                        self.matrixNeg[i][j] = self.c[keyword]

        return self.matrixNeg


    #Frequency matrix for positive files
    def FreqPos(self, Path):

        self.Path = Path
        Files = glob.glob(Path)
        self.matrixPos = np.zeros((1000, 9))
        for r in range(self.matrixPos.shape[0]):
            #last row of the feature matrix for positive files
            #is the label 1 representing positive class
            self.matrixPos[r][8] = 1



        for f in Files:
               for f in Files:
                with open(f) as file:
                    self.contents = file.read().strip().split()
                    self.c = Counter(self.contents)
                    for keyword in self.vocabulary:

                            i = int(Files.index(f))
                            j = int(self.vocabulary.index(keyword))
                            self.matrixPos[i][j] = self.c[keyword]
        return self.matrixPos




if __name__ == '__main__':
    myclassifier = Classifier()



    FreqMatrixNeg = myclassifier.FreqNeg(r'C:\Users\mahsa\OneDrive\
    Desktop\Assignment 3_AI\txt_sentoken\neg\*.txt')
    FreqMatrixPos = myclassifier.FreqPos(r'C:\Users\mahsa\OneDrive\
    Desktop\Assignment 3_AI\txt_sentoken\pos\*.txt')
    #Frequency matrix of the text files
    FrequencyMatrix = myclassifier.Featurematrix(FreqMatrixNeg, FreqMatrixPos)
```

```
np.random.shuffle(FrequencyMatrix)
Y_freq = FrequencyMatrix[:,8]
X_freq = np.delete(FrequencyMatrix, -1 , axis=1)


#Cross Validation using Multinomial Naive Bayes
accuracy = []
Kf = StratifiedKFold(n_splits=10)
StratifiedKFold(n_splits=10, random_state = None, shuffle = False)
for Train, Test in Kf.split(X_freq, Y_freq):
    X_Train = X_freq[Train]
    X_Test  = X_freq[Test]
    Y_Train = Y_freq[Train]
    Y_Test  = Y_freq[Test]

    clf = MultinomialNB()
    clf.fit(X_Train, Y_Train)
    Y_Predict = clf.predict(X_Test)
    accuracy.append(clf.score(X_Test, Y_Test))

print('Cross Validation accuracy using Multinomial Naive Bayes: ', accuracy)
```

```
Cross Validation accuracy using Multinomial Naive Bayes:  [0.625, 0.73499999999999999, 0.65500
```

Cross Validation accuracy using Multinomial Naive Bayes: [0.625, 0.73499999999999999, 0.65500000000000003, 0.68000000000000005, 0.63, 0.625, 0.63, 0.65500000000000003, 0.68999999999999995, 0.65000000000000002]