Graphical Models

In this exercise, you will construct several graphical models for the MNIST dataset, and perform inference on them to determine the most likely class for each example. You are provided with a modular graphical model implementation (`graphical.py`). It lets you specify the graph (Variables and Factors) in an object oriented fashion and does inference automatically. Because the implementation is generic (it can handle any directed tree), it can be quite slow for large networks.

The data is stored in the file `mnist.mat`. The handwritten digits are cropped to 20x20 pixels. The data is accessed through the method `utils.getData()` and returns three matrices: the input `X`, the labels `T`, and some additional data `Z` that will be used in the second part of the exercise.

Example of Execution

You are provided with a simple example where the most likely class is inferred based on the number of activated pixels in the top part of the 20x20 image (first 10 rows), and the number of activated pixels (called levels) in the bottom part of the image (last 10 rows). The corresponding graphical model is depicted in the diagram below. The letter V denotes the variables, and the letter F denotes the factors.

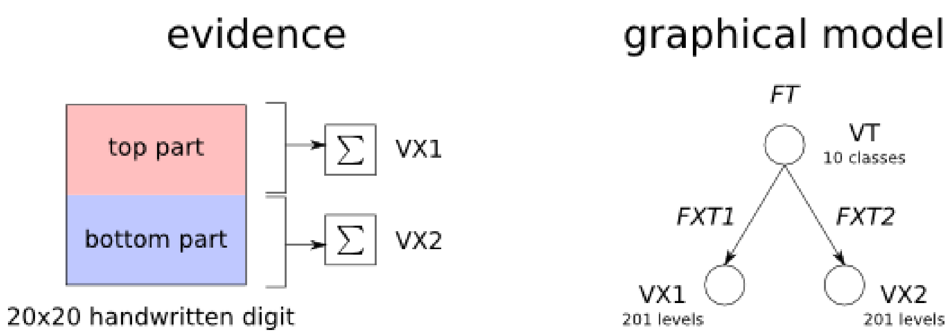

Figure 1: scenario1

The sum operator counts the number of white pixels in the corresponding region of the image. Note that this model looses a lot of information (all details within the top and bottom part of the image), and thus, the predictive accuracy is expected to be low (here, ~30%).

```
In [1]: import utils
        import numpy
        from graphical import *

        X,T,_ = utils.getData()

        nbclasses = 10
        nblevels  = 201

        # ========================================
        # BUILD THE MODEL
        # ========================================

        # ----------------------------------------
        # Compute the evidence for VX1 and VX2
        # ----------------------------------------
        Xtop = X[:,:10,:].sum(axis=2).sum(axis=1)
```

```python
Xbot = X[:,10:,:].sum(axis=2).sum(axis=1)

# -----------------------------------------
# Define the variable nodes
# -----------------------------------------
VT  = VariableNode("VT",nbclasses)
VX1 = VariableNode("VX1",nblevels)
VX2 = VariableNode("VX2",nblevels)

# -----------------------------------------
# Compute class factors
# -----------------------------------------
nbexamples = numpy.zeros([nbclasses])
for cl in range(nbclasses):
    nbexamples[cl] = (T==cl).sum()

PT = (nbexamples+1) / (nbexamples+1).sum() # adding 1 avoids log(0)
FT = FactorNode("FT",numpy.log(PT),[VT])

# -----------------------------------------
# Compute class-level factors (top)
# -----------------------------------------
nbexamples = numpy.zeros([nbclasses,nblevels])
for cl in range(nbclasses):
    x = Xtop[T==cl]
    for lv in range(nblevels):
        nbexamples[cl,lv] = (x==lv).sum()

PXT1 = (nbexamples+1) / (nbexamples+1).sum(axis=1)[:,numpy.newaxis] # adding 1 avoids log(0)
FXT1 = FactorNode("FXT",numpy.log(PXT1),[VT,VX1])

# -----------------------------------------
# Compute class-level factors (bottom)
# -----------------------------------------
nbexamples = numpy.zeros([nbclasses,nblevels])
for cl in range(nbclasses):
    x = Xbot[T==cl]
    for lv in range(nblevels):
        nbexamples[cl,lv] = (x==lv).sum()

PXT2 = (nbexamples+1) / (nbexamples+1).sum(axis=1)[:,numpy.newaxis] # adding 1 avoids log(0)
FXT2 = FactorNode("FXT",numpy.log(PXT2),[VT,VX2])

# =========================================
# INFER CLASSES FOR TEST DATA
# =========================================
def predict(x):
    VX1.evidence = x[:10,:].sum()
    VX2.evidence = x[10:,:].sum()
    VT.initiateMessagePassing(None)
    return numpy.argmax(VT.computeMarginal())

print('Accuracy: %.3f'%utils.getAccuracy(predict,debug=False))
```

```
it: 000  acc: 0.000
it: 025  acc: 0.462
it: 050  acc: 0.373
it: 075  acc: 0.395
it: 100  acc: 0.386
it: 125  acc: 0.381
it: 150  acc: 0.377
it: 175  acc: 0.347
it: 200  acc: 0.338
it: 225  acc: 0.341
it: 250  acc: 0.339
it: 275  acc: 0.326
it: 300  acc: 0.322
it: 325  acc: 0.319
it: 350  acc: 0.336
it: 375  acc: 0.330
it: 400  acc: 0.342
it: 425  acc: 0.338
it: 450  acc: 0.330
it: 475  acc: 0.328
it: 500  acc: 0.319
it: 525  acc: 0.312
it: 550  acc: 0.310
it: 575  acc: 0.314
it: 600  acc: 0.314
it: 625  acc: 0.315
it: 650  acc: 0.316
it: 675  acc: 0.320
it: 700  acc: 0.324
it: 725  acc: 0.321
it: 750  acc: 0.322
it: 775  acc: 0.322
it: 800  acc: 0.323
it: 825  acc: 0.326
it: 850  acc: 0.321
it: 875  acc: 0.321
it: 900  acc: 0.324
it: 925  acc: 0.325
it: 950  acc: 0.328
it: 975  acc: 0.326
Accuracy: 0.325
```

Shallow Model (25 P)

We would like to modify the model above in the following way: We define 400 input nodes (as many nodes as pixels of the 20x20 image) with two possible states (black or white). Each input node is connected to the class node. Given a particular class is observed, the input nodes are assumed to be independent. A diagram of the proposed model is given below:

Tasks:

- **Implement the graphical model shown above. Set the factors to their most likely value given the data (X,T). Use the same variable names as in the diagram above. (20 P)**
- **Print the classification accuracy of the graphical model you have implemented. (5 P)**

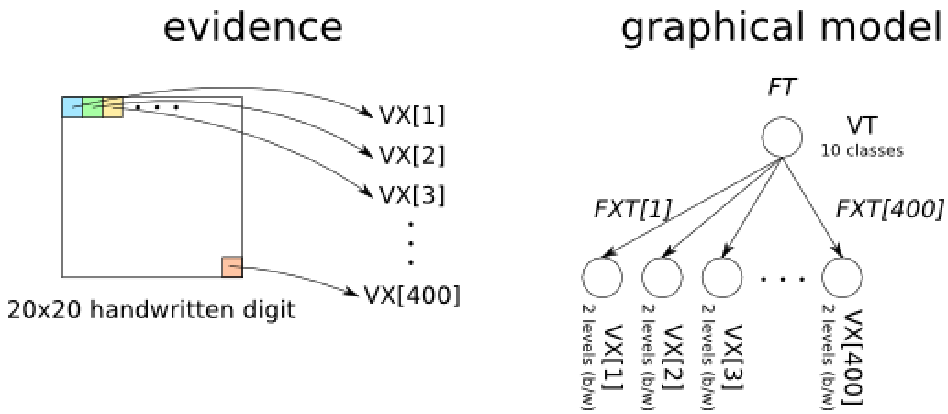

Figure 2: scenario2

In [2]:
```
# REPLACE BY YOUR CODE
import solution
solution.shallow()
# --------------------
```

```
it: 000  acc: 1.000
it: 025  acc: 0.885
it: 050  acc: 0.843
it: 075  acc: 0.868
it: 100  acc: 0.871
it: 125  acc: 0.857
it: 150  acc: 0.854
it: 175  acc: 0.858
it: 200  acc: 0.851
it: 225  acc: 0.858
it: 250  acc: 0.857
it: 275  acc: 0.855
it: 300  acc: 0.847
it: 325  acc: 0.840
it: 350  acc: 0.843
it: 375  acc: 0.843
it: 400  acc: 0.840
it: 425  acc: 0.833
it: 450  acc: 0.836
it: 475  acc: 0.838
it: 500  acc: 0.836
it: 525  acc: 0.833
it: 550  acc: 0.829
it: 575  acc: 0.832
it: 600  acc: 0.829
it: 625  acc: 0.824
it: 650  acc: 0.829
it: 675  acc: 0.828
it: 700  acc: 0.827
it: 725  acc: 0.824
it: 750  acc: 0.823
it: 775  acc: 0.822
it: 800  acc: 0.819
```

```
it: 825  acc: 0.823
it: 850  acc: 0.826
it: 875  acc: 0.826
it: 900  acc: 0.827
it: 925  acc: 0.828
it: 950  acc: 0.831
it: 975  acc: 0.828
Accuracy: 0.829
```

Hierarchical Model (25 P)

We now would like to construct a more complex architecture consisting of two layers. There are 400 input nodes that are separated into 16 groups representing local regions of the image of size 5x5. As in the previous model, each input node has 2 possible states (black or white). Each input node is only connected to its associated group node that has 12 possible states (called subclasses). The state of these group nodes is available for the training data and is returned by the method `utils.getData()`, and can therefore be used to set the factors of the hierarchical model. All group nodes are connected to the top-level class node. In this hierarchical model, the group nodes are independent given the class is known, and the pixel values within a patch are independent given that the state of the associated group node is known. However, the pixels within the same group are no longer independent given the class only. These correlations caused by the unknown state of the group node confer added representational power to the model. A diagram of the model is given below:

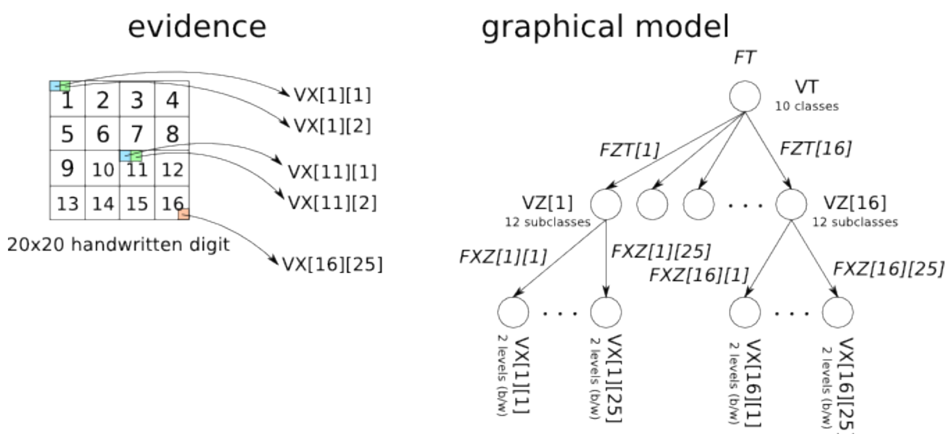

Figure 3: scenario3

Tasks:

- **Implement the graphical model shown above. Set the factors to their most likely value given the data (X,T,Z). Use the same variable names as in the diagram above. (20 P)**
- **Print the classification accuracy of the graphical model you have implemented. (5 P)**

```
In [3]: # REPLACE BY YOUR CODE
        import solution
        solution.hierarchical()
        # --------------------
```

```
it: 000  acc: 1.000
it: 025  acc: 0.923
it: 050  acc: 0.922
it: 075  acc: 0.921
it: 100  acc: 0.911
```

```
it: 125  acc: 0.905
it: 150  acc: 0.894
it: 175  acc: 0.898
it: 200  acc: 0.896
it: 225  acc: 0.903
it: 250  acc: 0.896
it: 275  acc: 0.899
it: 300  acc: 0.900
it: 325  acc: 0.883
it: 350  acc: 0.883
it: 375  acc: 0.883
it: 400  acc: 0.885
it: 425  acc: 0.873
it: 450  acc: 0.876
it: 475  acc: 0.878
it: 500  acc: 0.880
it: 525  acc: 0.876
it: 550  acc: 0.875
it: 575  acc: 0.877
it: 600  acc: 0.877
it: 625  acc: 0.879
it: 650  acc: 0.880
it: 675  acc: 0.882
it: 700  acc: 0.882
it: 725  acc: 0.877
it: 750  acc: 0.875
it: 775  acc: 0.879
it: 800  acc: 0.875
it: 825  acc: 0.877
it: 850  acc: 0.878
it: 875  acc: 0.878
it: 900  acc: 0.880
it: 925  acc: 0.881
it: 950  acc: 0.883
it: 975  acc: 0.880
Accuracy: 0.880
```