Learning Bayesian Networks and EM algorithm: An application to kinect data

Jingwen Yang
Anne-Claire Fouchier

1. Introduction

a. Dataset

The static body positions in MSRC-12 Kinect gesture data set of Microsoft Research Cambridge are used in this assignment, which is only a small part of it. The data are in the following body positions: crouch, right arm extended and both arms lifted.

b. Data structure and functions

The dataset consists of 2045 instances of body positions for 4 classes:

- “arms lifted”,
- “right arm extended to one side”,
- “crouched” and
- “right arm extended to the front”.

The body positions are encoded using a 20 x 3 matrix where each row is the position in space (x,y,z) of each of the 20 joints. The order in which the different joints appear in the vector are shown in Figure 1 for one instance of class both arms lifted.

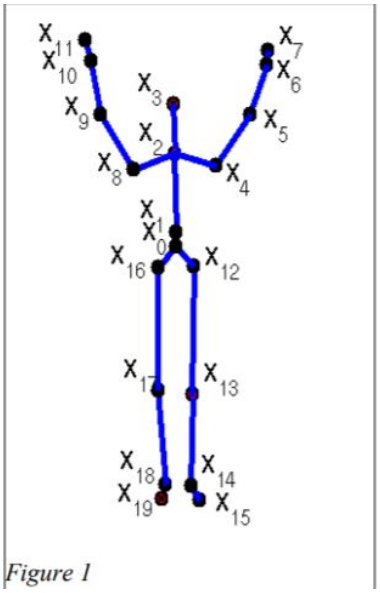

Figure 1

c. Models:Naive Bayes

i. Training

In the Naive Bayes (NB) model, each of the 60 variables(20 joints with each 3 coordinates) that define the body positions are considered independent given the class. Each variable under the circumstance of a specific class is going to be modeled using a Gaussian distribution. Therefore, in total there should be 20*3*4*2(joints*coordinates*classes*[1mean, 1sigma]) variables saved in the NB model, estimating by MLE the values for the mean and variance for each variable and class.

ii. Classifying

In the classifier, the output is the posterior probability for each instance belonging to each class:

$$P(C=k\,|\,instance) \propto P(C=k)\prod_{j=1}^{20} p(x_i,y_i,z_i|x_{p(i)},y_{p(i)},z_{p(i)},C=k) \quad (1)$$

where

$$p(x_i,y_i,z_i|x_{p(i)},y_{p(i)},z_{p(i)},C=k) = p(x_i|x_{p(i)},y_{p(i)},z_{p(i)},C=k)\times$$
$$p(y_i|x_{p(i)},y_{p(i)},z_{p(i)},C=k)\times$$
$$p(z_i|x_{p(i)},y_{p(i)},z_{p(i)},C=k)$$

Since Naïve Bayes supposes each variable is independent, we just discard the parent node in the conditional probability(crossed by the red arrow). Then, for each variable, we compare its observation to the gaussian distribution for each class, and output the probability for each class, sum them up together to get the probability for the whole body. We apply the implementation in logarithm space to get rid of underflow numerical issues.

d. Models:Linear Gaussian Model

i. Training

The Linear Gaussian Model (LGM) assumes each variable is dependent on their parent joints, which is given by a connection graph **(`nui_skeleton_conn`).** Therefore, instead of learning the means and sigmas in NB, here we need to learn the betas and sigmas to depict the relationship between each joint and their parents. Therefore in total, we should have 20*3*4*5(20joints*3coordinates*4classes*[4betas,1sigma]) variables saved in the LGM model. Except for the first joint which does not have parents. It should have mean and sigma calculated from NB, and stored accordingly. Our implementation stores the mean as betas.

ii. Classifying

$$P(C=k \mid instance) \propto P(C=k) \prod_{j=1}^{20} p\left(x_i, y_i, z_i \mid x_{p(i)}, y_{p(i)}, z_{p(i)}, C=k\right) \quad (1)$$

where

$$p\left(x_i, y_i, z_i \mid x_{p(i)}, y_{p(i)}, z_{p(i)}, C=k\right) = p\left(x_i \mid x_{p(i)}, y_{p(i)}, z_{p(i)}, C=k\right) \times p\left(y_i \mid x_{p(i)}, y_{p(i)}, z_{p(i)}, C=k\right) \times p\left(z_i \mid x_{p(i)}, y_{p(i)}, z_{p(i)}, C=k\right)$$

Same as in the NB model, but we calculate the means based on betas and parent variables according to formula below:

$$p\left(x_i \mid x_{p(i)}, y_{p(i)}, z_{p(i)}, C\right) = Normal\left(\beta_{01} + \beta_{11} x_{p(i)} + \beta_{21} y_{p(i)} + \beta_{31} z_{p(i)}; \sigma^2\right)$$
$$p\left(y_i \mid x_{p(i)}, y_{p(i)}, z_{p(i)}, C\right) = Normal\left(\beta_{02} + \beta_{12} x_{p(i)} + \beta_{22} y_{p(i)} + \beta_{32} z_{p(i)}; \sigma^2\right)$$
$$p\left(z_i \mid x_{p(i)}, y_{p(i)}, z_{p(i)}, C\right) = Normal\left(\beta_{03} + \beta_{13} x_{p(i)} + \beta_{23} y_{p(i)} + \beta_{33} z_{p(i)}; \sigma^2\right)$$

2. Code description

a. [Function] Fit_gaussian(X,W)

Here, we simply calculate the mean and standard deviation of the input observations of one single variable. Be careful with the std, it should not be variance.

b. [Function] Fit_linear_gaussian(Y,X,W)

$$\begin{pmatrix} E_D[Y] \\ E_D[X_1 Y] \\ \vdots \\ E_D[X_K Y] \end{pmatrix} = \begin{pmatrix} 1 & E_D[X_1] & \cdots & E_D[X_K] \\ E_D[X_1] & E_D[X_1 X_1] & \cdots & E_D[X_1 X_K] \\ \vdots & \vdots & \ddots & \vdots \\ E_D[X_K] & E_D[X_K X_1] & \cdots & E_D[X_K X_K] \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_K \end{pmatrix}$$

Solve with linear algebra $\mathrm{b} = \mathrm{A}\,\mathrm{x} \qquad \mathrm{x} = \mathrm{A}^{-1} b$

$$\sigma^2 = Cov_D[Y;Y] - \sum_i \sum_j \beta_i \beta_j Cov_D[X_i; X_j]$$

$$\mathrm{Cov}_D[X;Y] = E_D[XY] - E_D[X] E_D[Y]$$

Here we are filling the equation above to get the betas and sigma of one variable given its parents. The Y is the observations for the variable, X is the parent matrix, which should be careful of the order of row and column, as it is different from the order organized in the dataset.

c. [Class] Model

We introduce two classes : Model and JointPart.

The class Model contains model.connectivity, model.class_priors and model.jointparts.

To create a Model, three inputs are necessary :

- a dataset (of size numberOfJoints x localisation x numberOfInstances)
- A label vector (of size numberOfInstances)
- Eventually a graph. It is a connectivity graph between each joint. If none is given, the model will compute Naïve Bayes, otherwise, Linear Gaussian.

Model.connectivity stores the connectivity graph.

Model.class_priors stores the prior estimates for each class (see set_priors).

Model.jointparts stores a list of objects of class Joint (see below). model.jointparts[i] contains the estimated parameters for the i-th joint.

The model is learned at initialization, i.e. each variable is initialized and computed/filled at initialization, with the following functions.

i. Set_priors

For each class, it calculates the proportion of instances belonging to that class over the total number of instances.

ii. Set_jointparts

Set_jointparts takes the dataset, the labels and the graph as input. It creates a list of objects of class JointPart (see below).

iii. isNaiveBayes

isNaiveBayes takes the model as input and returns True if model.connectivity equals to None, False otherwise. The reason behind this function is that the Model class, for generality purpose, can either compute Naïve Bayes or Linear Gaussian, depending on the input. As the model is learned at initialization, a user might want to check what operation a particular model performs.

d. [Class] JointPart

The class JointPart contains jointpart.joint_nb, jointpart.means, jointpart.betas and jointpart.sigma.

To create a Model, three inputs are necessary :

- the dataset
- the label vector
- eventually the graph
- the joint number, i.e, its identification

jointpart.joint_nb, stores the joint number, which will be of use in determining the appropriate dataset to use for the joint and eventually its parent.

Jointpart.sigma stores the sigmas output by fit_gaussian or fit_linear_gausian. It is of size : numberOfLocation x numberOfClasses.

Jointpart.means stores the means output by fit_gaussian. It is of size : numberOfLocation x numberOfClasses.
Jointpart.betas stores the betas output by fit_linear_gausian. It is of size : numberOfLocation*4 x numberOfClasses.

Either fit_gaussian or fit_linear_gausian are used, depending on the model, so either jointpart.means or jointpart.betas are filled.

A joint's values are computed and filled at initialization with the following functions.

i. fill_Naive_Bayes

fill_Naive_Bayes is used if there is no connectivity graph. It computes fit_gaussian for each location of the joint for each class, and stores the values in jointpart.means and jointpart.sigma.

ii. fill_Linear_Gaussian

fill_Linear_Gaussian is used if there is a connectivity graph. It computes fit_linear_gaussian for each location of the joint for each class, and stores the values in jointpart.betas and jointpart.sigma. If the joint does not have a parent, then it computes fit_gaussian

e. [Function] classify_instances(instances, model)

The input 'instances' are instances to be classified. Model is either NB or LGM in this case. The output should be a matrix of probabilities of each class for each instance. We call the compute_logprobs function to get the log probabilities for each instance, for calculation efficiency. The function outputs the exponential of the log probabilities in order to get the expected form.

f. [Function] compute_logprobs(instance,model)

The input instance is a single instance of the testing dataset, the model is either NB or LGM. The output should be a vector of log likelihood of the instance for each class. First we initialize the log likelihood by the priors saved in the model. In the case of NB, we simply pass the observations, means, sigmas of each joint to the function log_normpdf to get the logarithm of their probabilities. By summing up the probabilities for variables X,Y,Z for all the 20 joints, we get the final log probability for each class.

g. [Function] log_normpdf(x,mu,sigma)

The input x is a vector of the location variables (x,y,z) of an instance, mu is a vector of the means of (x,y,z) in one specific class, sigma is a vector of standard deviation of (x,y,z) in one specific class. It computes the natural log of the normal probability density function using the functions from 'scipy'.

3. Description of the metrics and validation of model

a. Measures

Two measures were used to evaluate our models: accuracy and a confusion matrix. Accuracy is used to calculate the ratio of correctly predicted instances over the entire set. However, accuracy is a global measure. With a confusion matrix, we can see which classes are badly predicted.

b. Cross-validation

A 10-fold cross-validation was performed to extract the model giving back the best accuracy. The data is shuffled to limit the chances of not having enough instances of a certain class. In turn, a model is trained on 90% of the data and validated on 10%. The model is stored in a list of models as well as the accuracy. The function returns the list of 10 models and the list of the 10 corresponding accuracies.

c. Evaluation and validation of our models

i. Evaluation of fit_linear_gaussian

The function was evaluated with the provided code and passed the test.

ii. Evaluation of the model using the validation dataset

The provided validation dataset was used to validate our models. A Naïve Bayes model and a Linear Gaussian model were created on the training dataset and tested on the test dataset.
The accuracy of our Naïve Bayes model was the same as the provided accuracy, but the accuracy of our Linear Gaussian model was of 1, which is better than the one provided in accur_lg.
Then, sigmas and means of the Naïve Bayes model were compared with the means and sigmas stored in model_nb, and sigmas and betas of the Linear Gaussian model were

compared with the betas and sigmas stored in model_lg. Their differences were compared to 1e-10. With this, we know that our models were similar to the provided models at least to 1e-10.

This evaluation validated our model and allowed us to move on to evaluate their performances.

4. Performance evaluation

a. Naive Bayes

i. On the whole dataset

The whole dataset was trained in a Naïve Bayes model. Then, using this model, we classified each instance of the dataset. With Naïve Bayes, we get an accuracy of about 0.95.

```
sklearn.metrics.confusion_matrix(labels, y_pred)
```

```
array([[480,   0,   3,  25],
       [  0, 500,   0,   0],
       [  0,   0, 474,  48],
       [  0,   0,  21, 494]])
```

The confusion matrix highlights the discrepancy. All the instances of class 2 are classified correctly. However, 3 instances of class 1 are predicted as belonging to class 3, and 25 to class 8. Also, 48 instances of class 3 are predicted as belonging to class 8, and 21 classes of class 8 are predicted as class 3.
Only class 2 seems well-enough defined.

ii. Using cross-validation and a test set

The cross-validation function is described above. For this test, the provided dataset was used in turn as a training and validation set. Then the best model is extracted and the validation dataset is used as the test set.

As the dataset is shuffled, when doing the cross-validation twice, the function did not output the same model, yielding different accuracies on the validation sets.

```
#testing the best NB model

model_nb=models_nb[np.argmax(accuracies_nb)]
prob_nb_val=classify_instances(data_small, model_nb)
y_pred_nb_val=predict_label(prob_nb_val, np.unique(labels))
accuracy(y_pred_nb_val, labels_small)
```

```
0.975
```

```
sklearn.metrics.confusion_matrix(labels_small, y_pred_nb_val)
```

```
array([[30,  0,  0,  0],
       [ 0, 30,  0,  0],
       [ 0,  0, 30,  0],
       [ 0,  0,  3, 27]])
```

Case 1

```
#testing the best NB model

model_nb=models_nb[np.argmax(accuracies_nb)]
prob_nb_val=classify_instances(data_small, model_nb)
y_pred_nb_val=predict_label(prob_nb_val, np.unique(labels))
accuracy(y_pred_nb_val, labels_small)
```

```
0.9833333333333333
```

```
sklearn.metrics.confusion_matrix(labels_small, y_pred_nb_val)
```

```
array([[29,  0,  0,  1],
       [ 0, 30,  0,  0],
       [ 0,  0, 30,  0],
       [ 0,  0,  1, 29]])
```

Case 2

In both cases, cross-validation helped extract a better model than the one previously learned from the whole dataset.

However, as shown below, the second model classifies all the instances in the whole dataset with less accuracy than the first model. However, both models yield better accuracy than the model trained on the whole dataset. It also has to me noted that none of these Naïve Bayes models were able to beat any of the Linear Gaussian models (next section).

```
PNB=classify_instances(data, model_nb)
YPREDNB=predict_label(PNB, np.unique(labels))
accuracy(YPREDNB, labels)
```

```
0.9867970660146699
```

```
sklearn.metrics.confusion_matrix(labels, YPREDNB)
```

```
array([[506,   0,   0,   2],
       [  0, 500,   0,   0],
       [  0,   0, 517,   5],
       [  0,   0,  20, 495]])
```

Case 1

```
PNB=classify_instances(data, model_nb)
YPREDNB=predict_label(PNB, np.unique(labels))
accuracy(YPREDNB, labels)
```

```
0.9770171149144254
```

```
sklearn.metrics.confusion_matrix(labels, YPREDNB)
```

```
array([[479,   0,   4,  25],
       [  0, 500,   0,   0],
       [  0,   0, 516,   6],
       [  0,   0,  12, 503]])
```

Case 2

b. Linear gaussian model

i. On the whole dataset

The whole dataset was trained in a Linear Gaussian model. Then, using this model, we classified each instance of the dataset. Here, we get an accuracy of 0.9921760391198045.

```
sklearn.metrics.confusion_matrix(labels, lg_y_pred)
```

```
array([[493,   0,   0,  15],
       [  0, 500,   0,   0],
       [  0,   0, 521,   1],
       [  0,   0,   0, 515]])
```

The confusion matrix above highlights the discrepancies. All the instances of class 2 are classified correctly, as with the Naïve Bayes model. However, 15 instances of class 1 are predicted as belonging to class 8. Only 1 instance of class 3 is predicted as belonging to class 8, and all class 8 instances are predicted correctly.

ii. Using cross-validation and a test set

The cross-validation function was performed as for Naïve Bayes.

```
#testing the best  model

model_lg=models_lg[np.argmax(accuracies_lg)]
prob_lg_val=classify_instances(data_small, model_lg)
y_pred_lg_val=predict_label(prob_lg_val, np.unique(labels))
accuracy(y_pred_lg_val, labels_small)
```

```
1.0
```

```
sklearn.metrics.confusion_matrix(labels_small, y_pred_lg_val)
```

```
array([[30,  0,  0,  0],
       [ 0, 30,  0,  0],
       [ 0,  0, 30,  0],
       [ 0,  0,  0, 30]])
```

Case 1 and 2

In both cases, we get perfect accuracy on the test set .

However, as shown below, on a bigger dataset to classify, the datasets do not yield perfect accuracy, but it is better than the one yielded by the model learned on the whole dataset. Also, as mentioned in the previous section, all our Linear Gaussian models are beating the Naïve Bayes models.

```
PLG=classify_instances(data, model_lg)
YPREDLG=predict_label(PLG, np.unique(labels))
accuracy(YPREDLG, labels)
```

```
0.9965770171149144
```

```
sklearn.metrics.confusion_matrix(labels, YPREDLG)
```

```
array([[501,   0,   3,   4],
       [  0, 500,   0,   0],
       [  0,   0, 522,   0],
       [  0,   0,   0, 515]])
```

Case 1

```
PLG=classify_instances(data, model_lg)
YPREDLG=predict_label(PLG, np.unique(labels))
accuracy(YPREDLG, labels)
```

```
0.993643031784841
```

```
sklearn.metrics.confusion_matrix(labels, YPREDLG)
```

```
array([[501,   0,   0,   7],
       [  0, 500,   0,   0],
       [  0,   0, 520,   2],
       [  0,   0,   4, 511]])
```

Case 2

5. Conclusion

In this lab, we have shown that Linear Gaussian models get better accuracy than Naïve Bayes models. Linear Gaussian takes parent variables into consideration. This information is essential to yield better results and make better predictions. However, Naïve Bayes is a simpler model that can be used safely if no information regarding the relationship between the variables is known. Finally, cross-validation is a good tool to extract a better model.