assignement05

March 24, 2023

1

#

Assignment 5

Martim Silva 51304 and Alexandre Sobreira 59451

[20]:
```
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from sklearn.model_selection import KFold, GridSearchCV, train_test_split,␣
 ↪cross_val_predict, cross_val_score
from sklearn.metrics import confusion_matrix, matthews_corrcoef, accuracy_score
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from IPython.display import display
from pomegranate import *
```

[21]:
```
def mdl_stats(gs, X_test, y_test):
    preds = gs.best_estimator_.predict(X_test)
    stats = pd.DataFrame()
    stats['Accuracy'] = [accuracy_score(y_test, preds)]
    stats['Matthews Corr. Coef.'] = [matthews_corrcoef(y_test, preds)]
    cm = confusion_matrix(y_test, preds)
    return stats, pd.DataFrame(cm)
```

<img src="Bayesian Network _Causal Net..png">

Figure 1: Bayesian Network / Causal Net. Variables A, B, C and D are represented as nodes, and a truth table is given alongside them.

1.1 Bayesian Network Creation

1.1.1 Tables

- The first step will be to create the probability tables.
- THe type of probability must be explicitly spelled followed by a list of the paretns in the same orderas the columns take in the tables provided (image above)

- Table A will be the only table with discrete ditribution given that it doesnt have any conditional probabilities.
- All the others will be conditional probability tables.
  * Table c is dependent on both table a and b.
  * Table b is dependent on table a
  * Table d is dependent on table b

[22]:
```
a_table = DiscreteDistribution({"T": .3, "F": .7})
b_table = ConditionalProbabilityTable(
                                        [["F", "T", .4],
                                         ["F", "F", .6],
                                         ["T", "T", .8],
                                         ["T", "F", .2]], [a_table])
c_table = ConditionalProbabilityTable([["F", "F", "T", .1],
                                       ["F", "F", "F", .9],
                                       ["F", "T", "T", .7],
                                      ["F", "T", "F", .3],
                                      ["T", "F", "T", .5],
                                      ["T", "F", "F", .5],
                                      ["T", "T", "T", .99],
                                      ["T", "T", "F", .01]], [a_table, b_table])
d_table = ConditionalProbabilityTable([["F", "T", .55],
                                      ["F", "F", .45],
                                      ["T", "T", .2],
                                      ["T", "F", .8]], [b_table])
```

1.1.2 Nodes

- Our network will have 4 nodes each one with a probability table associated

[23]:
```
a_node = Node(a_table, name = "A")
b_node = Node(b_table, name = "B")
c_node = Node(c_table, name = "C")
d_node = Node(d_table, name = "D")
```

- this nodes are very similar to the hidden markov models “states”

1.1.3 Model

- Finaly the model (Bayesian Network) is created adding the states (nodes) and the edges (conections)

[24]:
```
model = BayesianNetwork()
model.add_states(a_node, b_node, c_node, d_node)
model.add_edge(a_node, b_node)
model.add_edge(a_node, c_node)
model.add_edge(b_node, c_node)
model.add_edge(b_node, d_node)
model.bake()
```

1.2 Problem 1

- Using the predict_proba function we will obtain the probabilities of each variable in the graph given evidence
  - This calculates the marginal probability distributions for each state given the evidence provided through loopy belief propagation. Loopy belief propagation is an approximate algorithm which is exact for certain graph structures.

1.2.1 a)

$ P(A=T|C=T,D=T) $

[25]:
```
model.predict_proba({"C":"T", "D":"T"})
```

[25]:
```
array([{
            "class" : "Distribution",
            "dtype" : "str",
            "name" : "DiscreteDistribution",
            "parameters" : [
                {
                    "T" : 0.5054138717420109,
                    "F" : 0.49458612825798914
                }
            ],
            "frozen" : false
        }                                    ,
        {
            "class" : "Distribution",
            "dtype" : "str",
            "name" : "DiscreteDistribution",
            "parameters" : [
                {
                    "T" : 0.6483149049313832,
                    "F" : 0.3516850950686168
                }
            ],
            "frozen" : false
        }                                    , 'T', 'T'], dtype=object)
```

- In this output we can see: 1st the probability of A being T or F and 2nd the probability of B being T or F, all given the evidence C=T,D=T
  - The answer to this querie is: $ P(A=T|C=T,D=T) = .505 $

1.2.2 b)

$P(A = T|D = F)$

[26]:
```
model.predict_proba({"D":"F"})
```

[26]:

```
array([{
            "class" : "Distribution",
            "dtype" : "str",
            "name" : "DiscreteDistribution",
            "parameters" : [
                {
                    "T" : 0.34651898734177244,
                    "F" : 0.6534810126582277
                }
            ],
            "frozen" : false
        }                                       ,
        {
            "class" : "Distribution",
            "dtype" : "str",
            "name" : "DiscreteDistribution",
            "parameters" : [
                {
                    "T" : 0.6582278481012659,
                    "F" : 0.34177215189873406
                }
            ],
            "frozen" : false
        }                                       ,
        {
            "class" : "Distribution",
            "dtype" : "str",
            "name" : "DiscreteDistribution",
            "parameters" : [
                {
                    "T" : 0.6084545745874058,
                    "F" : 0.3915454254125941
                }
            ],
            "frozen" : false
        }                                       , 'F'], dtype=object)
```


- In this output we can see: 1st the probability of A being T or F, 2nd the probability of B being T or F, 3th the probability of C being T or F, all given the evidence D=F
  - The answer to this querie is: $ P(A=T|D=F) = .347 $

1.2.3 c)

$P(B = T|C = T)$

[27]:
```
model.predict_proba({"C":"T"})
```

[27]:

```
array([{
            "class" : "Distribution",
            "dtype" : "str",
            "name" : "DiscreteDistribution",
            "parameters" : [
                {
                    "T" : 0.5253830090147861,
                    "F" : 0.4746169909852139
                }
            ],
            "frozen" : false
        }                                       ,
        {
            "class" : "Distribution",
            "dtype" : "str",
            "name" : "DiscreteDistribution",
            "parameters" : [
                {
                    "T" : 0.8100843263425553,
                    "F" : 0.1899156736574448
                }
            ],
            "frozen" : false
        }                                       , 'T',
        {
            "class" : "Distribution",
            "dtype" : "str",
            "name" : "DiscreteDistribution",
            "parameters" : [
                {
                    "T" : 0.2664704857800823,
                    "F" : 0.7335295142199177
                }
            ],
            "frozen" : false
        }                                       ], dtype=object)
```


- In this output we can see: 1st the probability of A being T or F, 2nd the probability of B being T or F, 3th the probability of D being T or F, all given the evidence C=T
  - The answer to this querie is: $ P(B=T|C=T) = .810 $

1.2.4 d)

$ P(B=T|A=T, C=T) $

[28]:
```
model.predict_proba({"A":"T", "C":"T"})
```

[28]:

```
array(['T', {
                    "class" : "Distribution",
                    "dtype" : "str",
                    "name" : "DiscreteDistribution",
                    "parameters" : [
                        {
                            "T" : 0.8878923766816139,
                            "F" : 0.11210762331838604
                        }
                    ],
                    "frozen" : false
                }                                    , 'T',
            {
                "class" : "Distribution",
                "dtype" : "str",
                "name" : "DiscreteDistribution",
                "parameters" : [
                    {
                        "T" : 0.23923766816143516,
                        "F" : 0.7607623318385648
                    }
                ],
                "frozen" : false
            }                                    ], dtype=object)
```


- In this output we can see: 1st the probability of B being T or F, 2nd the probability of D being T or F, all given the evidence A=T, C=T
  - The answer to this querie is: $ P(B=T|A=T, C=T) = .888 $

1.2.5 e)

$ P(C=T|A=F, B=F, D=F) $

[29]:
```
model.predict_proba({"A":"F", "B":"F", "D":"F"})
```

[29]:

```
array(['F', 'F', {
                        "class" : "Distribution",
                        "dtype" : "str",
                        "name" : "DiscreteDistribution",
                        "parameters" : [
                            {
                                "T" : 0.10000000000000016,
                                "F" : 0.8999999999999999
                            }
                        ],
                        "frozen" : false
                    }                                    , 'F'],
      dtype=object)
```

- In this output we can see: The probability of C being T or F given the evidence A=F, B=F, D=F.
  - The answer to this querie is: $ P(C=T|A=F, B=F, D=F) = .1 $

1.3 Problem 2

1.3.1 Data

[7]:
```
# Load data
iris = load_iris()
y_iris = iris.target
X_iris = iris.data
# Training test split
X_train, X_test, y_train, y_test = train_test_split(X_iris, y_iris, test_size=0.
 ↪25, random_state=10)
```

[52]:
```
y_iris_test = pd.DataFrame(y_iris)
counts = y_iris_test[0].value_counts()
print(counts)
```

```
0    50
1    50
2    50
Name: 0, dtype: int64
```

- Perfectly balanced dataset

1.3.2 a)

NB

[32]:
```
# Creating dictionary with the testing parameters
var_smoothing_p = [1e-9, 1e-7, 1e-5, 1e-3, 1e-1, 0.2, 0.4, 1, 2, 4]
param_grid = {"var_smoothing": var_smoothing_p}

# Define the model and do the grid search
gnb = GaussianNB()
gs_gnb = GridSearchCV(estimator = gnb, param_grid = param_grid, cv = 5)
gs_gnb = gs_gnb.fit(X_train, y_train)

print("Best parameters for Gaussian Naive Bayes model:", gs_gnb.best_params_)
```

```
Best parameters for Gaussian Naive Bayes model: {'var_smoothing': 0.4}
```

[33]:
```
# Retrieving the best estimator model prediction statistics
stats_gnb, cm_gnb = mdl_stats(gs_gnb, X_test, y_test)
print("Statistics for the best parameter combination of Gaussian Naive Bayes␣
 ↪model:")
display(stats_gnb)
print("And the confusion matrix is: ")
display(cm_gnb)
```

```
Statistics for the best parameter combination of Gaussian Naive Bayes model:

   Accuracy  Matthews Corr. Coef.
0  0.947368              0.920335

And the confusion matrix is:

    0   1   2
0  11   0   0
1   0  14   1
2   0   1  11
```

1.3.3 Comments

- The Naive Bayes algorithm is known to be considered a good baseline comparison model given that is not prone to overfiting.
- The Gaussian Naive Bayes Classifier was used assuming that the independent variables follow a gaussian distribution, invoking the central limit theorem.
- For the Gaussian Naive Bayes classifier the only parameter that can be explored for optimizing the quality of the predictions is the smoothing factor for which several values including the default were checked using Scikit-learn's GridSearch function.
  - The parameter was 'var_smoothing': 0.4.
- Since the data used was the Iris data set where features are quantitative and the target variable has more than 2 possible values, the appropriate metrics (Accuracy, Matthews Correlation Coeficient and the Confusion Matrix) were used. Given that the dataset is perfectly balanced, the accuracy value can be seen as a good measure to access how well the model predicted the species of the flower given petal and sepal length and width.
- This model did very well with only 2 misclassifications, and accuracy of .947 aswell as a MCC of .920.

1.3.4 b)

RF

[53]:
```
n_estimators_p = [100, 150, 200]
criterion_p = ["gini", "entropy"]
max_depth_p = [1, 5, 10]
min_samples_leaf_p = [4, 6, 8]
max_features_p = ["log2", "sqrt", None]
param_grid = {"n_estimators": n_estimators_p, "criterion": criterion_p,␣
 ↪"max_depth": max_depth_p, "min_samples_leaf": min_samples_leaf_p,␣
 ↪"max_features": max_features_p}

# Define the model and do the grid search
rf = RandomForestClassifier(random_state=44)
gs_rf = GridSearchCV(estimator = rf, param_grid = param_grid)
gs_rf = gs_rf.fit(X_train, y_train)

print("Best parameters for Random Forest model:", gs_rf.best_params_)
```

```
Best parameters for Random Forest model: {'criterion': 'gini', 'max_depth': 1,
'max_features': None, 'min_samples_leaf': 4, 'n_estimators': 100}
```

[54]:
```
# Retrieving the best estimator model prediction statistics
stats_rf, cm_rf = mdl_stats(gs_rf, X_test, y_test)
print("Statistics for the best parameter combination of Random Forest model:")
display(stats_rf)
print("And the confusion matrix is: ")
display(cm_rf)
```

```
Statistics for the best parameter combination of Random Forest model:
```

	Accuracy	Matthews Corr. Coef.
0	0.947368	0.924688

```
And the confusion matrix is:
```

	0	1	2
0	11	0	0
1	0	13	2
2	0	0	12

Comments

- For the Random Forest there are many more parameters to explore, it uses Decision Trees (which are a customizable part of the model too) to draw, each one, an output that is then combined with the outputs of the other trees into a single final value. Other parameters were tested aside the ones pertaining to a Decision Tree by itself for example the number of trees to use in the first place (100 Decision Trees).
  - The best combination of parameters produced from applying GridSearch with 5-fold cross validation on the training data for the Random Forest was using Trees with the Gini criterion, a maximum depth limit of 5, a maximum number of features to analyse for deciding on the splitting in the trees of the logarithm of the number of features and a minimum number of samples at leaf nodes of 6 being allowed.
- The results of testing the model were very similar to those in the Naive Bayes case with 2 misclassifications and equal rounded values for accuracy and MCC.

1.3.5 Discussion

- Can be seen that the Naive bayes algorithm that is considered a simple model with only one parameter possible to tune, provided very similar accuracy and mcc values to the Random Forest algorithm which is a much more complex algorithm with many parameters possible to be tuned.
  - This shows that the Naive Bayes is simple but at the same time very usefull for example to be used as a baseline classifier.