How to exploit knowledge in a classification problem setting

Giovanni Misseri

Master degree in Data Science,
Knowledge and Data Mining course

2018, June

1 Introduction

In this relation I will try to explain and justify what I've done in order to solve a classification problem.

I would like to start talking in general about the objective of the study I'm going to show. The problem I will deal with comes from the medical field and it's about heart disease; in fact I'm going to analyse a dataset born from a study of four different entities: *Cleveland Clinic Foundation; Hungarian Institute of Cardiology, Budapest; V.A. Medical Center, Long Beach, CA; University Hospital, Zurich, Switzerland.* Their aim was collecting evidences in order to create a dataset with whom one should be able to solve the problem of classifying whether a person suffers from heart disease or not.

This dataset has been already used in other works and the one achieving the greatest results has a mean of corrected classified examples of 76%, even though they don't specify in which way they estimated this percentage.

So the setting is the following: we are asked to classify people in ill and not ill and we have a dataset in order to do that. Past works tried to solve the problem using logistic regression and linear discriminant analysis, but I will try

to use a different approach, I will try to classify them using *logic tensor network*.

2 The Data

The dataset on which we will base all our further analysis is a dataset created merging the four databases of the entities cited previously. Let's see in detail how the dataset is composed.

This database contains 76 attributes, but only 14 of them were published; anyway all published experiments used the same subset of 14 of attributes in their analysis I'm going to use. In particular, the Cleveland database is the only one that has been used by researchers up to this date. The "goal" field refers to the presence of heart disease in the patient, measured as an integer valued from 0 (no presence) to 4. Experiments with the Cleveland database have concentrated on simply attempting to distinguish presence (values 1,2,3,4) from absence (value 0). The names and social security numbers of the patients were removed from the database and replaced with dummy values.

The originals datasets were structured in the following way: Cleveland, Hungarian, Switzerland and Long Beach VA contained respectively 303, 294, 123, 200 observation; and for each observation 14 attributes have been collected, in the following schema a short summary of their meaning.

Variables:

- **Age**, age in years
- **Sex**, sex (1 = male; 0 = female)
- **Cp**, chest pain type (1= typical angina; 2= atypical angina; 3= non-anginal pain; 4= asymptomatic)
- **Trestbps**, resting blood pressure
- **Chol**, serum cholestoral
- **Fbs**, fasting blood sugar > 120 mg/dl (1 = true; 0 = false)

- **Restecg**, resting electrocardiographic results (0= normal; 1= having ST-T wave abnormality; 2= showing probable or definite left ventricular hypertrophy
- **Thalach**, maximum heart rate achieved
- **Exang**, exercise induced angina (1 = yes; 0 = no)
- **Oldpeak**, ST depression induced by exercise relative to rest
- **Slope**, the slope of the peak exercise ST segment (1= upsloping; 2= flat; 3= downsloping)
- **Ca**, number of major vessels (0-3) colored by flourosopy
- **Thal**, (3 = normal; 6 = fixed defect; 7 = reversable defect)
- **Num**, diagnosis of heart disease (0= $<$ 50% diameter narrowing; 1= $>$ 50% diameter narrowing)

The variable we will try to predict is **Num**, the variable defining the presence of the heart disease. Let's see now how each dataset is composed with respect to the patient affected by heart disease and with which intensity and the one not affected.

Class distribution						
Database:	0	1	2	3	4	Total
Cleveland:	164	55	36	35	13	303
Hungarian:	188	37	26	28	15	294
Switzerland:	8	48	32	30	5	123
Long Beach VA:	51	56	41	42	10	200

3 Data Preparation

Before any data manipulation, a preliminary exploratory analysis is done. With that we want to spot characteristic trends in the data or typical behaviour that may help us in further analysis. Anyway because of almost all the variables are factor the exploratory analysis we can do is really limited and give us few surplus knowledge of the data with respect to the one we knew from the presentation of the data. What we found at this step of the analysis is the presence of some missing values. I decided to delete three variables, which contained missing values for almost every entry of the dataset, the three variables are *Ca, Thal, Slope*; indeed it seems that only one out of the four datasets composing our dataset has this three variables correctly registered.

Now that we deleted the three useless variables it's time to deal with general missing values and I decided to delete the entries containing missing values just to be sure not to insert bias in the analysis due to the missing data imputation; anyway someone else could go for a sampling imputation of the data, that's also a good choice.

Before we continue the analysis it's important to have in mind the target we want to reach. The aim of this work is checking whether Logic Tensor Network is useful in this setting; but in order to build a Logic Tensor Network we need some logic constrains that rule the data.

This kind of knowledge is typical of some kind of structured problems (read hand written numbers and letters from "codice fiscale" etc) or in general one would need a deep knowledge of the field he is working on; in such a way one would be able to get some rules that influenced the creation of the data and exploiting this knowledge is possible to build a very effective classifier.

In our context anyway we are in neither of the two situations mentioned above, but here is the main aim of this work: checking if, taken a general dataset, is possible to build an effective predicting model (LTN) even without a strong a priori knowledge.

In order to build the LTN anyway some rules are needed, otherwise it would

be a simple Neural Network; so the idea driving further analysis is to derive the rules we lack with rule mining and use that relation to build the classifier.

To apply rule mining we need to have the dataset in a very specific way; in fact the idea in rule mining is that every variable represents the presence of an attribute and exploiting the special configuration of the dataset we can look for specific relation between variables.

In order to apply the a priori algorithm we need to transform every variable of the dataset in factor and then extract as much dummy variable from each attribute as the number of levels for that specific attribute, we need to pose the whole dataset into *one hot encoding*. What we described is exactly what we have done, ending up with a dataset in which every variable takes value zero or one and every variable represents one level of a variable.

Now our data are ready to be processed in order to find rules, but continuing the analysis, I will consider only two levels in the variable *Num*, indeed in future we will care about presence or not of the disease without minding the intensity of the disease. This is done in order to make the classification easier and also to allow a future comparison between this study and the previous one.

4 Rule Mining

In this section I will show how the rule mining has been done. In a first trial of this study I decided to use as data for the Rule Mining process a randomly selected set of 128 units from my dataset, these units than have been removed from the analysis. I made this in order not to make unfair estimation when the model needs to be tested. But in general we suppose to have knowledge about the whole universe generating the data and so to see the difference between a NN approach and a LTN approach is fear to use knowledge coming from the whole dataset. Anyway from a theoretical point of view a randomly selected subset of units coming from the dataset in expectation represents exactly the distribution of the original dataset and if the first one is biased also the subset will be biased, so it makes no difference to apply the rule mining to the whole

dataset or to a subset of it, but in the first case we have more units to train the models.

From a practical point of view the rule obtained from the subset of units and the one coming from the whole dataset are the same; reflecting the probability equivalence we talked about. Using the data the rules come from could lead to biased estimation of performances, but due to possible problems coming from the estimation of the model with few data I will use anyway the whole dataset for training/testing the models.

So as first step we have to import the package needed and the data on which apply the mining.

```
import pandas as pd
import numpy as np
from itertools import combinations

dat=pd.read_csv("../dati_analisi.csv")
del dat["Unnamed: 0"]

dati=np.array(dat)
```

Now we need to set the parameters on which the rule mining will be done, in particular the choice of the minimum support,α and the minimal confidence,β is of fundamental importance. Choosing these parameters is such important because the rules we will find depend on them; putting both of them really low will guarantee us a lot of rules, but none of them can be considered reliable, on the other hand putting those parameters too high will return no rule.

I decided to try to keep α and β as high as possible because my aim is to find few rules but I need to be sure about the confidence of that rules, so I set α and β respectively to 0.4 and 0.87.

Then we start preparing some variables useful for the process of rule mining, like *omega* and *transactions*, which are two vectors containing all integer

numbers from zero to, respectively, the number of variables and the number of observations.

```
alpha = 0.40
beta = 0.87

observation,variables = dati.shape
Omega = range(variables)
transactions = range(observation)
Fi=np.sum(dati,axis=0)
items=list(dat.columns.values)
```

In this passage we prepare a function able to create a dictionary containing all variables or group of variables which together respect the minimum support value that we decided. In order to do that we exploit the fact that a super-group of a rare variable will be rare too, so we explore the minimum possible number of variables combination to get the result.

```
def get_frequent_itemsets(dati,alpha):
    print(dati)
    L = {0:{tuple():observation}}
    Fi = np.sum(dati,axis=0)
    L[1] = {(i,):Fi[i] for i in Omega if Fi[i] >= alpha*observation}
    k = 1
    while L[k]:
        L[k+1] = {}
        for s1 in L[k]:
            for s2 in L[k]:
                s3 = set(s1)-set(s2)
                if len(s3) == 1:
                    s12 = set(s1) | set(s2)
                    s12_is_good = True
                    for i in s12:
                        if tuple(sorted(s12-{i})) not in L[k]:
                            s12_is_good = False
```

```
                        break
                if s12_is_good:
                    s12 = tuple(sorted(s12))
                    Fs12 = np.sum(np.all(dati[:,s12],axis=1))
                    if Fs12 >= alpha*observation:
                        L[k+1][s12]=Fs12
        k += 1
    return L
```

At this point we have the dictionary containing all the set of variables respecting the minimum support, we have to explore whether there are rules between these variables respecting the minimum confidence we imposed above.

In order to do that we build a function that verifies if, considering to have two general variables A B, the number of times A and B appear together in the dataset divided by the number of times A (or B) appears, is bigger or equal with respect to the minimum confidence decided previously.

```
def get_rules(L, beta):
    R = []
    for k in L:
        for s in L[k]:
            for j in range(1,len(s)):
                for sub_s in combinations(set(s),j):
                    if L[len(sub_s)][tuple(sorted(sub_s))]*beta <= L[k][s]:
                        R.append([sub_s,tuple(set(s)-set(sub_s))])
    return R
```

Everything is ready now, so we feed the functions with our data, getting as results the rules, if there is any.

```
L = get_frequent_itemsets(dati,alpha)
for k in L:
    for S in L[k]:
        print([items[i] for i in S])
```

```
R = get_rules(L,beta)
if R:
    print("association rules")
    for r in R:
        print([items[i] for i in r[0]]," -> ",[items[i] for i in r[1]])
else:
    print("no association rules")


.
.
.
#Console Output:
##############################
# ['num0'] -> ['fbs0']
# ['restecg0'] -> ['fbs0']
# ['num_1'] -> ['Sex_1']
##############################
```

Finally we end up with three rules, now we have all we needed to build the Logic Tensor Network and the rules we are going to use are guaranteed to have a confidence level to be true bigger or equal to 0.87, and the rules we found are guaranteed to characterize at least the 40% of the data.

5 Classification Models

Logic Tensor Network wants to classify a set of input building a theory and training it on the train data. So LTN exploit the structure of the problem in order to define a theory, a set of rules that explain the context in which the data are generated or in general their intrinsic characteristics; but in this case we are not dealing with a structured problem, so we don't have a complete knowledge of the "real theory" behind the data.

Because of that we can't build a sufficiently complete theory able to explain

the data, so we will use the rules we have extracted from the data but also the usually cross entropy to train the model.

The idea behind LTN is that we can explain the data with a theory, built in the fuzzy logic context; but given the fact that in the test data we don't have the objective variable we can consider that like a predicate (if the aim of the study is the binary classification), where the predicate is a Neural Network.

So what we will do is to train a NN and analyze its results, then we will build a LTN and its results will be analyzed. After that we will try to compare the results of the two approches (hoping that, has theory suggests, LTN will generalize better to unseen data).

In order to get a fair comparison between LTN and NN a "seed" will be set in the random imputation of the NN weight, in this way we are sure to really start with the same NN.

Instead, to evaluate the goodness of each model, I will use a cross validation approach, and I will take care that both models will be trained and tested on the same data.

Before talking about the practical implementation of the models just one other measure has been take: the data, before being selected for cross validation, will be permutated in order to avoid problems coming from the fact that we are using a dataset coming from four different entities datasets merged (we don't want for example test data coming only from just one of the four datasets composing our dataset).

Described the idea driving these models we can start talking about one of the two, Neural Network.

5.1 Neural Network

In the following lines I will show the NN implementation used for the analysis. First of all I import the data, the package we need and prepare a vector containing the permutated indices that will be used for the cross validation

```
import numpy as np
import pandas as pd
import tensorflow as tf
np.random.seed(123)
a=np.arange(740)
permutated=np.random.permutation(a)
dat=pd.read_csv("../dati_analisi.csv")
del dat["Unnamed: 0"]
```

Now that all the package are imported we can start the definition of the model, and we start defining the general parameter related to the optimization part. We will perform as many steps of descending algorithm as needed to reach a gain on the loss function smaller that our ε. At each step the learning rate used will be 0.1.

We will establish an $\varepsilon <= 10^{-u}$ with u sufficiently big and let the optimization go on until $f(x_{k+1}) - f(x_k) <= \varepsilon$; anyway this can lead to a really long computation and just to understand the differences between the two methods also simply doing 700 steps could work.

By practical experimentation I observed that 700 steps are sufficient to make the network converge, having a more powerful computer would have allowed me to use different stopping rule. Using different stopping rule would be really useful, in fact essentially both methods stop the optimization just looking at the loss function, but in order to avoid overfitting would be nice to have a stopping criterion that mind also the accuracy.

Anyway I tried it on my pc and it struggles also to finish one single model; a cross validation would be unfeasible and it wouldn't be possible to give an estimate on the accuracy for NN and LTN.

After the general parameters there is the model definition, indeed what we built is a NN made by four hidden layers and in each one of that there are 80 nodes.

```
# Parameters
learning_rate = 0.1
```

```
batch_size = 650
display_step = 50
epsilon=0.01

# Network Parameters
n_hidden_1 = 80
n_hidden_2 = 80
n_hidden_3 = 80
n_hidden_4 = 80
num_input = 34
num_classes = 2

# tf Graph input
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])
```

At this point we build the structure of the NN, linking each layer with the next one. All these operations are made inside a for loop in order to register the performance of 10 different NN to have a better estimate about the model performance instead of simply having one train set and on test set.

It's important to notice that at the moment of initializing the weight for all neurons we used a seed, the same seed will be used again defining the LTN. After this passage the NN structure will be ready to be feeded with the data.

```
# Ciclce over the dataset (Cross Validation)
u=np.arange(10)
lis=[]

for i in u:
    dato=dat
    sel=np.repeat(True,740)
    sel[permutated[i*74:(i+1)*74]]=False
    trainx=dato.iloc[sel,0:34]
    trainy=dato.iloc[sel,34:]
    testx=dato.iloc[permutated[i*74:(i+1)*74],0:34]
```

```
testy=dato.iloc[permutated[i*74:(i+1)*74],34:]

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1],
        seed=123+i)),
    'h2': tf.Variable(tf.random_normal([n_hidden_1,
        n_hidden_2],seed=12+2*i)),
    'h3': tf.Variable(tf.random_normal([n_hidden_2, n_hidden_3],
        seed=1+3*i)),
    'h4': tf.Variable(tf.random_normal([n_hidden_3, n_hidden_4],
        seed=1234+4*i)),
    'out': tf.Variable(tf.random_normal([n_hidden_4,
        num_classes],seed=23+5*i))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1],seed=50+6*i)),
    'b2': tf.Variable(tf.random_normal([n_hidden_2], seed=75+7*i)),
    'b3': tf.Variable(tf.random_normal([n_hidden_3], seed= 13+8*i)),
    'b4': tf.Variable(tf.random_normal([n_hidden_4], seed= 44+9*i)),
    'out': tf.Variable(tf.random_normal([num_classes], seed=83+10*i))
}

# Create model
def neural_net(x):
    # Hidden fully connected layers with 80 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_3 = tf.add(tf.matmul(layer_2, weights['h3']), biases['b3'])
    layer_4 = tf.add(tf.matmul(layer_3, weights['h4']), biases['b4'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_4, weights['out']) + biases['out']
    return out_layer

# Construct model
logits = neural_net(X)
```

```
prediction = tf.nn.softmax(logits)
```

What is coming is a fundamental passage, and it's also what distinguish a LTN from a NN; in fact it's time to define a loss function. We make the softmax transformation on the output and define the loss as the cross entropy between the transformed output and the label for each entry.

Defined the loss function we define an optimizer, in this case our choice will be Adam optimizer, which is based on stochastic gradient descent method but implements also ways to control variance problems and momentum terms. In general Adam is very fast and in our context seems to be useful.

```
# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model
correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

Everything is set now, we just need to run the optimization.

```
# Start training
with tf.Session() as sess:
    # Run the initializer
    sess.run(init)
    batch_x = trainx
    batch_y = trainy
    step=0
    losso=sess.run(loss_op, feed_dict={X: batch_x,Y: batch_y})
    k=1000000000
```

```
while k > 0.01:
    # Run optimization op (backprop)
    sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
    lossn=sess.run(loss_op, feed_dict={X: batch_x,Y: batch_y})
    k=np.abs(losso-lossn)
    losso=lossn

    if step % display_step == 0 or step == 1:
        # Calculate batch loss and accuracy
        loss, acc = sess.run([loss_op, accuracy], feed_dict={X:
            batch_x,
                                                             Y:
                                                                 batch_y})
        print("Step " + str(step) + ", Minibatch Loss= " + \
              "{:.4f}".format(loss) + ", Training Accuracy= " + \
              "{:.3f}".format(acc))
    step=step+1
print("Optimization Finished!")

# Calculate accuracy
print("Testing Accuracy:", \
    sess.run(accuracy, feed_dict={X: testx,
                                  Y: testy}))
print("Run {}".format(i))
accu=sess.run(accuracy, feed_dict={X: testx,
                                  Y: testy})
lis.append(accu)
```

5.2 Logic Tensor Network

Now we have the results of 10 NN trained on the described NN's structure. Next step is to train a LTN having the same structure of this NN and even the same starting weights;however, as we said, the only difference between the two models is the loss function definition. For this reason, I will describe only that, because everything else is exactly the same of above.

The only exception is made on the stopping criterion, here epsilon will be posed equal to one; with lower value my pc doesn't converge in feasible time.

As already said to obtain a LTN we insert inside the loss function to minimize some logic constrains, the three rules found during the rule mining. To do that we have to write that rules with respect to the fuzzy logic and which t-norm we want to use. I chose Lukasiewicz t-norm and then the implication $A \Rightarrow B$ is defined as $min(1, 1 - A + B)$.

We need now to define the rules, but whether *fbs0* is just a value in our dataset, *num0* in not available for definition of classification problem, is our target variable; and at this point comes in the LTN theory saying that we can use a NN to define *num0* as a predicate and thanks to the final softmax transformation we can consider the NN output as a fuzzy logic predicate output. So exploiting this fact we build our little theory.

After defining the rules we sum them up and multiply the result by -1; the new loss function is the sum of the softmax cross entropy and the sum of the rules. Conceptually the softmax cross entropy comes in the loss function with positive sign because we want to minimize the case in which our output differs from the given label, on the other hand the rules come with negative sign because we want to maximise the case in which a rule is respected or is generally "more true", in a fuzzy logic conception.

As one can notice there is a parameter c multiplying the rules value, that's useful to modify the impact of the rules on the loss function; clearly as c goes to zero the LTN and the NN becomes more similar. I chose 100 as parameter c because empirically works well.

```
# Define loss and optimizer
# ['num0'] -> ['fbs0']
cond1=tf.minimum(1.0,1-prediction[:,0]+X[:,19])
# ['restecg0'] -> ['fbs0']
cond2=tf.minimum(1.0,1-X[:,21]+X[:,19])
```

```
#['num_1'] -> ['Sex_1']
cond3=tf.minimum(1.0,1-prediction[:,1]+X[:,5])

b= (-1) * tf.reduce_sum([cond1,cond2,cond3])

loss_o = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
loss_op=tf.add(loss_o,100*b)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
```

6 Comparison

Finally the evaluation time comes, in this section indeed I will show the results of both models and I will make a comparison, possibly with some explanation and general interpretation of the analysis' results.

After the two loops shown in the previous section run we end up with two list containing the NN and LTN performances. As said the two models have exactly the same starting point (talking about the NN and the network behind LTN), so we can compare the results of each run to do speculation about how each model performs and possibly why one goes better than the other.

In the following table all the results of the model are reported.

Model Accuracy						
Model:	0	1	2	3	4	5
Neural Network:	0.770	0.729	0.702	0.648	0.676	0.878
Logic Tensor Network:	0.784	0.743	0.770	0.784	0.730	0.703
Difference:	-0.014	-0.014	-0.068	-0.135	-0.054	0.176

Model Accuracy						
Model:	6	7	8	9	Mean	Variance
Neural Network:	0.771	0.716	0.730	0.770	0.739	0.004
Logic Tensor Network:	0.811	0.811	0.730	0.771	0.764	0.001
Difference:	-0.041	-0.095	0	-0.001	-0.024	0.003

In the table above the sequence $0 \dots 9$ represents the result of the i-th model with respect the i-th training set and test set.

In general what's really important is that the mean correct classified percentage of test units in the LTN is 2.4% bigger than NN. This is not such a great result but in general support the decision made in this analysis, this result means that logic can help and increase performance in classification of input even if you start the analysis just with the data or in that situation in which the structure of the problem is not defined a priori.

An other important aspect in my opinion is that this time I used rule mining to get the theory on which train the LTN, but this kind of approach makes really more sense in all that cases in which you perform a study on something known or about a topic on which other studies has already been done; thanks to logic one could exploit the conclusion of more studies in order to include in his prediction knowledge coming from other models.

What I'm saying is that LTN with the very same structure I used could be used in order to do *metanalysis*, with the results of obtaining really strong models in terms of generalizability of model built.

An other useful consequence of the use of conclusion coming from other studies is that we would need fewer data to train our model, fundamental characteristic of rare phenomenon studies.

But going back to the analysis just made, it's interesting to see that one of the rule we found is $[\prime num1\prime] \rightarrow [\prime Sex1\prime]$, having heart disease implies being male. This relation is really interesting because also from here comes a support of what we said few lines above, in fact it's well known that being male is one

of the risk factor of heart disease. This is confirmed by the official "Carta del rischio cardiovascolare" made by Istituto Superiore di Sanità. Also the magazine "Wired" in an article written together with "Zentiva-Sanofi" writes «*gli uomini hanno una probabilità sensibilmente maggiore di ammalarsi di malattie cardiache, tanto che essere maschi è considerato un altro importante fattore di rischio, soprattutto per l'infarto del miocardio.*»

So basically what we have done here could have been done also in a meta-analysis; and maybe thanks to that the LTN generalize well, because the rules we used were strong.

A similar explanation for the second important rule, $[\prime num0\prime] \rightarrow [\prime fbs0\prime]$, can be found. Indeed recalling the meaning of *fbs* we see that people with fasting blood sugar less than 120 mg/dl have *fbs* equal to zero, while the others have value one.

But as we can notice in literature a person with fasting blood sugar bigger than 120 mg/dl are people suffering from prediabetes (if is not actually a diabetic person), and more important, diabetes and heart disease often come together because the two share really a lot of risk factors. The words of the U.S. Department of Health and Human Services confirm our supposition «Having diabetes means that you are more likely to develop heart disease and have a greater chance of a heart attack or a stroke. People with diabetes are also more likely to have certain conditions, or risk factors, that increase the chances of having heart disease or stroke, such as high blood pressure or high cholesterol.»

To conclude the comparison it's interesting to notice that as already said, before this analysis there exist one previous study on this dataset achieving 76% of correct unit classified, here we reached 76.4% transforming all the variables into factorial, loosing information every time we found a continuous variable, so we got a slighlty better result using less information.

7 Final Considerations

Before ending this analysis I would like to talk about the general interpretation and make some final consideration on the model.

The dataset I analyzed was completely composed by factorial data, so in general we could see the space on which our data lays like a n-dimensional ball, n is the dimension of our dataset in terms of variables, where each i-dimension is divided in the number of levels the i-th factorial dimension contains. This means that we can see our space as the union of a lot of n-dimensional subspaces.

In general a trained model in this setting, considering a binary classification problem, can just assign a label to each subspace and then every time a new observation needs to be classified if it falls in that subspace it will be classified with that label.

Clearly the labels to the subspaces are given taking into consideration the data and reducing a certain loss function we will get our classifier; but if data for a certain reason are not representative of the population we want to classify, if the data comes from a biased study our result will be biased, no way to get rid of that.

But applying logic to models, like in LTN, will force some subspaces to have different labels and if the logic we impose is really strongly representative of the population we will get better results in predicting general observation coming from the population, even if our starting data were biased.

In general this is what could have happened in this analysis, we took rules valid in 87% of the cases and we generalized this state also to that data where these rules were false.

To conclude I think that logic is really a useful instrument to increase predicting performances through the use of information coming from other studies or in general t'is a really good technique to include in our estimate the knowledge we have about the problem.