Acute Nephritis Detector Using Logistic Regression*

Christian Sámano[A01423363]

Tecnológico de Monterrey, Querétaro, Querétaro, México
https://tec.mx/es

Abstract. The following report will present the performance analysis, the construction and the coding of a logistic regression trained to predict Acute Nephritis. The goal of the project is to classify the patients based on their probability of either having or not this disease. This probability is obtained with some data given by the patient, such as: temperature, pain feeling, etc.

Keywords: Logistic · Regression · Nephritis · Renal · Pelvis · Classification.

1 Introduction

Kidneys are part of the urinary tract, they are in charge of filtering the blood and also form around half of the urine by pulling water and waste from the blood. Some symptoms that can appear when kidneys are damage are: frequent need for urination, fever, rib pain and nausea. Nephritis of Renal Pelvis is an urinary tract infection that affects women more than it does to men. This infection causes kidney inflation that later can result on kidney failure if the infection is not properly identified and treated. Even Though this is a very common and painful disease it is treatable, there is the importance of a correct diagnosis at the right time.

2 Data set

The data set which I will be working with was downloaded from the UCI Repository and was created by Jacek Czerniak, Ph.D., Assistant Professor at the Polish Academy of Sciences. The set takes the possible symptoms as the attributes used to predict if the patient have one or both of the 2 possible infections which are: Inflammation of urinary bladder and Nephritis of renal pelvis origin. For simplification purposes I will be working only with the Nephritis of Renal Pelvis Origin and the model will be trained just to predict this infection. The attributes are labeled as: Temperature of patient, Occurrence of nausea, Lumbar pain, Urine pushing (continuous need for urination), Micturition pains and Burning of urethra, itch, swelling of urethra outlet. The data set consists of 120 instances that represent 120 potential patients.

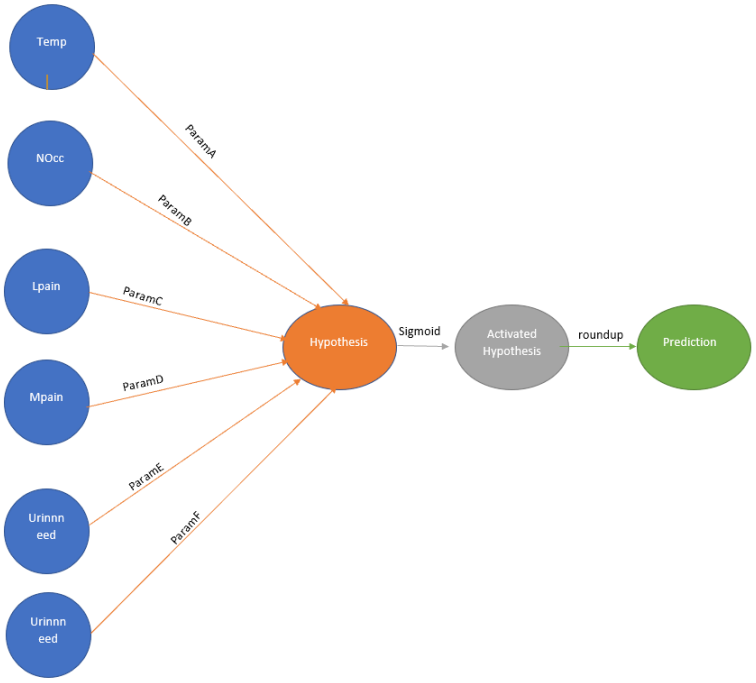
*Supported by TEC de Monterrey.

Data set sample

Temperature	Occurence of nausea	Lumbar pain	Continuous need for urination	Micturition pains	Urethra discomfort	Urinary bladder inflamation	Nephritis of renal pelvis origin
35.5	0	1	0	0	0	0	0
35.9	0	0	1	1	1	1	0
35.9	0	1	0	0	0	0	0
36	0	0	1	1	1	1	0
36	0	1	0	0	0	0	0
36	0	1	0	0	0	0	0
36.2	0	0	1	1	1	1	0
36.2	0	1	0	0	0	0	0
36.3	0	0	1	1	1	1	0
36.6	0	0	1	1	1	1	0
36.6	0	0	1	1	1	1	0

3 Model proposal

Model Diagram Proposal



4 Implementation

4.1 Scaling

Most of the attributes are binary columns, which are either 1 or 0 values, however there is just 1 attribute that has higher values. The temperature column is a float value in between 35 and 42 Celcius, in order to avoid the temperature attribute to keep all the weight or importance in the model prediction, I needed to scale

this column so its values remain in between 0 and 1. The scaling was done using the MinMaxScaler function from the sklearn library. What MinMax scaling does is that it takes the original value, it subtracts the minimum value of that column and divides that value by the difference of the maximum and minimum values.

Scaling function

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

4.2 Hypothesis

The hypothesis is the model prediction for the given attributes of that instance. As we can see in the model proposal, the hypothesis of this regression is obtained by the sum of all the products obtained by the multiplication of each attribute times its respective parameter, plus a bias. Until this point, the value obtained is the same one that could be obtained by a linear regression. The difference is that here we are trying to classify the patients in either 2 categories: presenting origins of nephritis or not presenting origins of nephritis. In order to obtain that probability, we must pass the prediction value through an activation function that in this case is the Sigmoid function.

Hypothesis implementation

```
def hyp(params, samples, b):
    global acum, h, x, count_pred
    acum = []
    count_pred = []
    x=0
    for i in range(len(samples)):
        h = 0
        count_pred = np.append(count_pred, x)
        x = x + 1
        for j in range (len(params)):
            h = h + (params[j] * float(samples[i][j]))
        h = h + b
        acum = np.append(acum, h)
```

4.3 Loss function

The loss function is used to obtain the error of the model predictions with respect to the original values. This errors are gonna be used in the gradient descend function in order to update the parameters. There are plenty of types of loss functions, there is MSE, MAE, huber loss, cross entropy, etc. The one used for this model is cross entropy The formula that was used was:

Cross entropy formula

$$\text{Loss} = -\frac{1}{\substack{\text{output}\\\text{size}}} \sum_{i=1}^{\substack{\text{output}\\\text{size}}} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i)$$

For the implementation, the formula was divided into two cases in order to avoid domain error in the logarithm.

Loss Function implementation

```
def cross_en(instances, hypothesis, realval):
    global uni_cost, cost, unit_error
    cost = 0
    unit_cost = 0
    unit_error = []
    for i in range(instances):
        if  (realval[i]) == 1:
            unit_cost = (-float(realval[i]) * math.log(float(hypothesis[i])))
            cost = cost + (-float(realval[i]) * math.log(float(hypothesis[i])))
        if  (realval[i]) == 0:
            unit_cost = (-(1 - float(realval[i])) * math.log(1 - float(hypothesis[i])))
            cost = cost + (-(1 - float(realval[i])) * math.log(1 - float(hypothesis[i])))
        unit_error.append(unit_cost)
    cost = cost / instances
```

4.4 Gradient descend

Gradient descend is the method used to update the parameter through out all the epochs. The formula used was:

Loss Function implementation

Repeat until convergence

{

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

}

The way gradient descend is by getting the derivative of the cross entropy function and measuring its gradient. It modifies the parameters in order to reach

the local or global minima of the loss function. The formula works in a different way for the attributes' parameters and for the bias. To solve this problem, there were declared two different functions, one for the parameters and other one for the bias.

Parameter gradient descend

```
def params_gd(params, alpha, instances, hypothesis, realval, features):
    global suma
    for i in range(len(params)):
        suma = 0
        for j in range(instances):
            suma = suma + (((float(hypothesis[j]) - float(realval[j]))) * float(features[j][i]))
        suma = (suma * alpha) / instances
        params[i] = params[i] - suma
```

Bias gradient descend

```
def bias_gd(b, alpha, instances, hypothesis, realval):
    global bias_suma, bias
    bias_suma = 0
    bias = b
    for j in range(instances):
        bias_suma = bias_suma + (((float(hypothesis[j]) - float(realval[j]))) * 1)
    bias_suma = (bias_suma * alpha) / instances
    bias = bias - bias_suma
    return b
```

4.5 Rounding up

As it was mentioned before, what the hypothesis of the model is, is just the probability of presenting or not presenting origins of Renal Pelvis Nephritis. However, what we really want to know is if the potential patient does present origins of the infection. To solve this problem was just needed a series of conditions where it was stated that if the probability was above 0.5, the patient does present the infection and if the probability was below 0.5 the patient does not present the infection.

Round up implementation

```
def roundup(values):
    for i in range(len(values)):
        if values[i] > 0.5:
            values[i] = 1.0
        else:
            values[i] = 0.0
```

4.6 Training

The training process refers to the repetition of the sequence stated in the previous points. The amount of times the process will be repeated would be determined by the error we get out of the loss function, or the amount of epochs we want

to repeat the cycle. For the training process, the data set needs to be divided so that the model does not know the actual outputs of all the instances. In this model the 80 percent of the data set was designated for the training process while the other 20 percent was for the test process.

Training cycle

```
def train():
    global epochs, count_train, error_train, activ_hyp_train, flag
    flag = 1
    epochs = 0
    count_train = []
    error_train = []
    activ_hyp_train = []
    while (epochs != 10000 and flag != 0):
        instances = len(df_x_train)
        hyp(params, df_x_train, bias)
        activ(acum)
        activ_hyp_train = activ_hyp
        cross_en(instances, activ_hyp_train, df_y_train)
        params_gd(params, alpha, instances, activ_hyp_train, df_y_train, df_x_train)
        "print(params)"
        bias_gd(bias, alpha, instances, activ_hyp_train, df_y_train)
        count_train = np.append(count_train, epochs)
        epochs = epochs + 1
        "print(cost)"
        if cost > 0.3:
            flag = 1
        else:
            flag = 0
        error_train = np.append(error_train, cost)
    roundup(activ_hyp_train)
    Traindone = Label(window, text= "Train Done! ").place(x = 400, y = 20)
```

4.7 Testing

The testing process is the same as the training process, but using the remaining 20 percent of the data set. Moreover, the testing process will start with the previous values obtained in the training process instead of starting with random values. Depending on how the error behaves we can see is our model is overfitting, underfitting or well fitted.

Testing cycle

```
def test():
    global epochs, count_test, error_test, activ_hyp_test, flag
    flag = 1
    epochs = 0
    count_test = []
    error_test = []
    activ_hyp_test = []
    while (epochs != 10000 and flag != 0 ):
        instances = len(df_x_test)
        hyp(params, df_x_test, bias)
        activ(acum)
        activ_hyp_test = activ_hyp
        cross_en(instances, activ_hyp_test, df_y_test)
        params_gd(params, alpha, instances, activ_hyp_test, df_y_test, df_x_test)
        "print(params)"
        bias_gd(bias, alpha, instances, activ_hyp_test, df_y_test)
        count_test = np.append(count_test, epochs)
        epochs = epochs + 1
        "print(cost)"
        if cost > 0.25:
            flag = 1
        else:
            flag = 0
        error_test = np.append(error_test, cost)
    roundup(activ_hyp_test)
    Testdone = Label(window, text= "Test Done! ").place(x = 400, y = 60)
```

4.8 Predicting

Once the model is well trained and well tested, we can pass to the prediction part. In my implementation I have decided to include 2 formats to predict. One format was done by predicting the outputs of the same data set, but randomly shuffled to see how the model behaves and to be sure that the good performance of the model was not coincidence. The other format is the one that allows the user to introduce its symptoms so that the model can tell if that patient does or does not present origins of Renal Pelvis Nephritis.

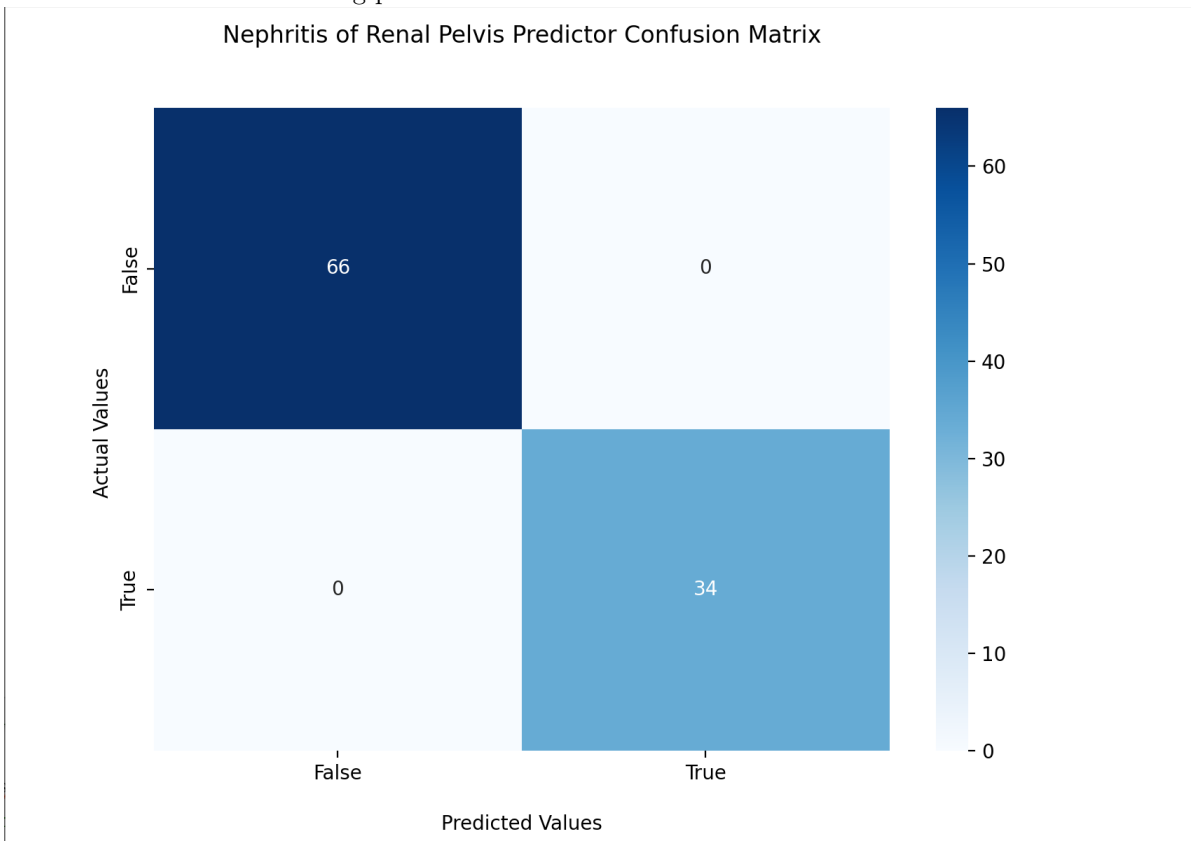
Prediction implementation

```
def predict():

    global activ_hyp_pred, activ_hyp_pred2, predictors, error_pred, instances, count_pred
    predictors2 = [0,0,0,0,0,0]
    predictors = []
    activ_hyp_pred = []
    activ_hyp_pred2 = []
    error_pred = 0
    instances = len(shuffle_y)
    predictors = shuffle_x
    hyp(params, predictors, bias)
    activ(acum)
    activ_hyp_pred2 = activ_hyp
    cross_en(instances, activ_hyp_pred2, shuffle_y)
    error_pred = unit_error

    print(bias)
    roundup(activ_hyp_pred2)


    predictors2[0] = txtTemp.get()
    predictors2[1] = txtNOcc.get()
    predictors2[2] = txtLpain.get()
    predictors2[3] = txtUrinneed.get()
    predictors2[4] = txtMpain.get()
    predictors2[5] = txtUdisc.get()

    predictors2[0] = (float(predictors2[0]) - 35.5) / (41.5 - 35.5)

    error_pred = 0
    print(predictors2)
    activ_hyp_pred = hyp2(params, predictors2, bias)


    Predres = Label(window, text= "Result: " + str(activ_hyp_pred)).place(x = 400, y = 180)
```

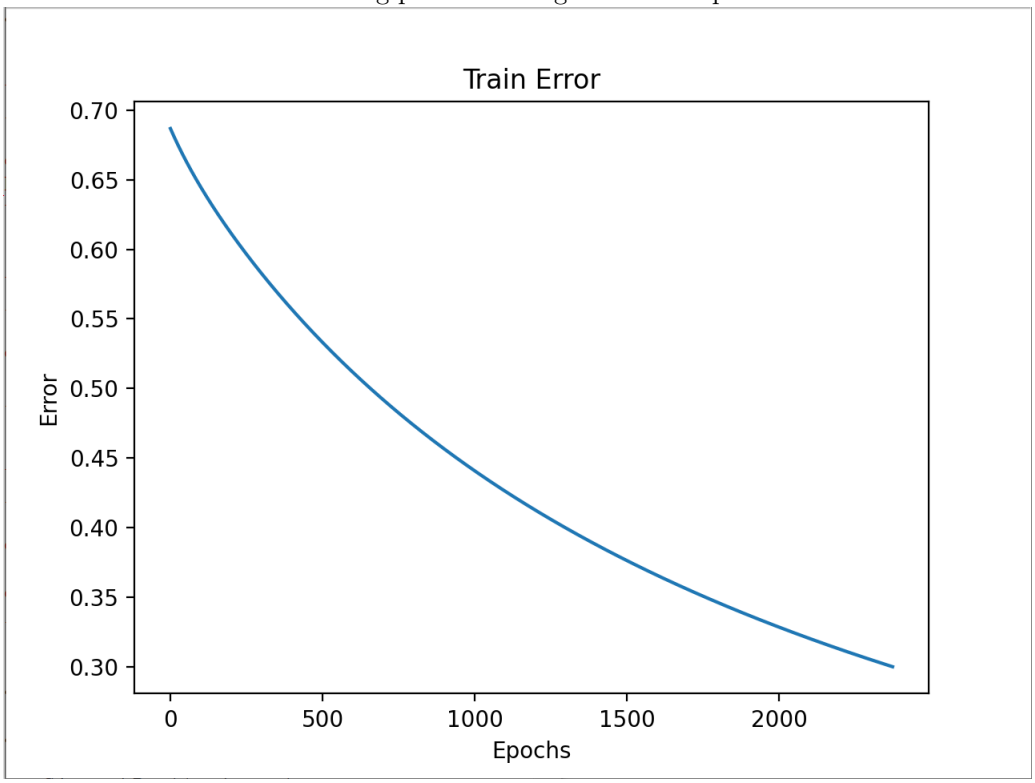
5 Results

5.1 Training

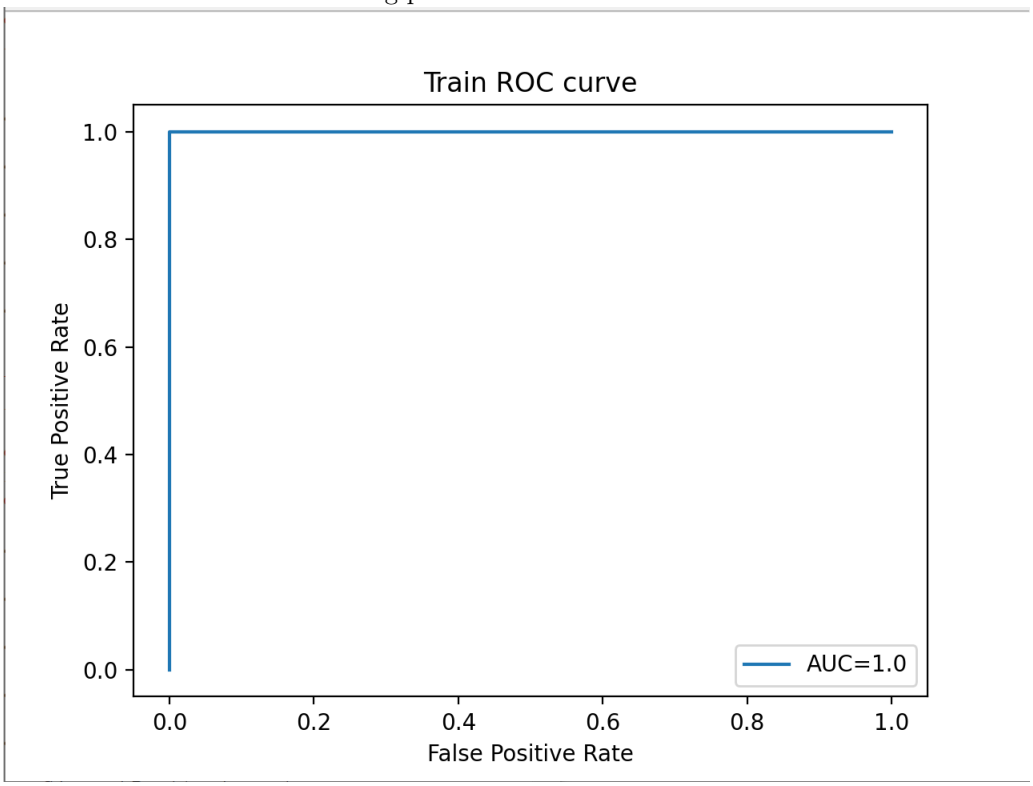
Training process confusion matrix



Error of the training process through out 2000 epochs

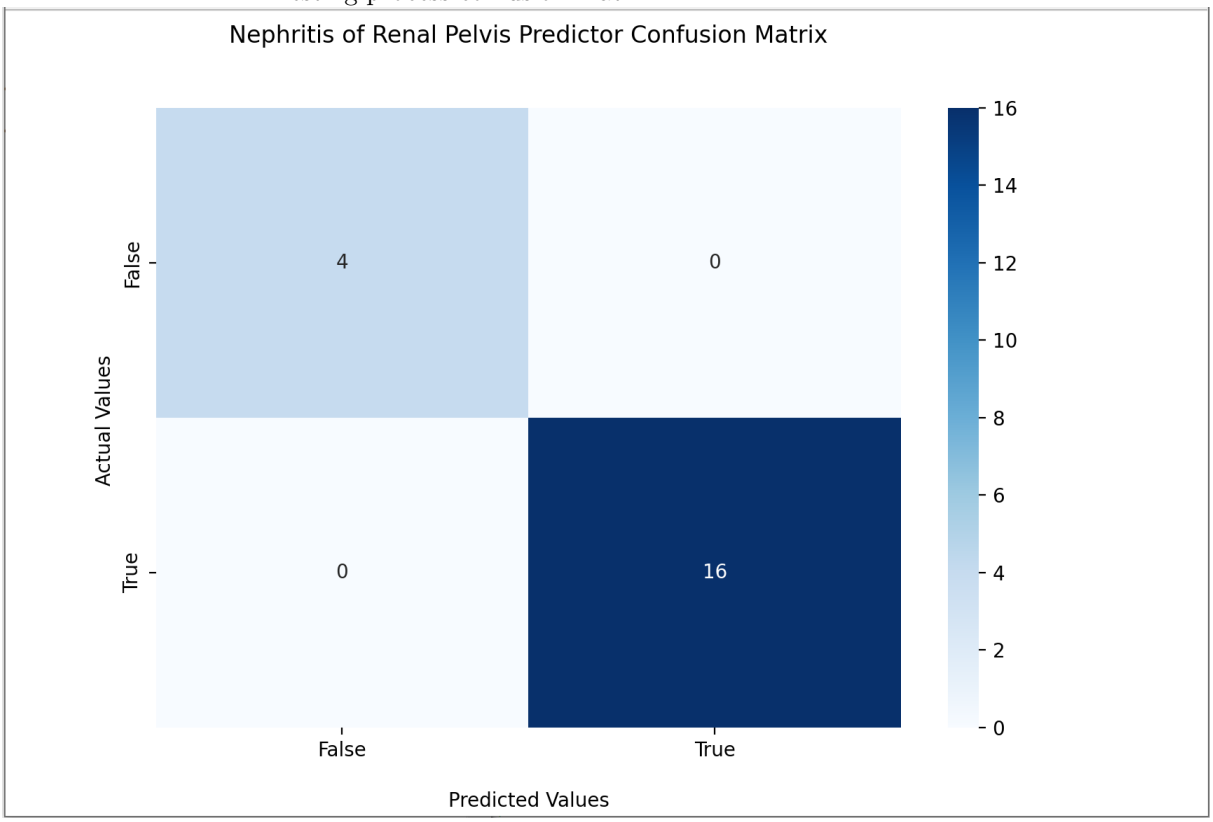


Training process ROC curve

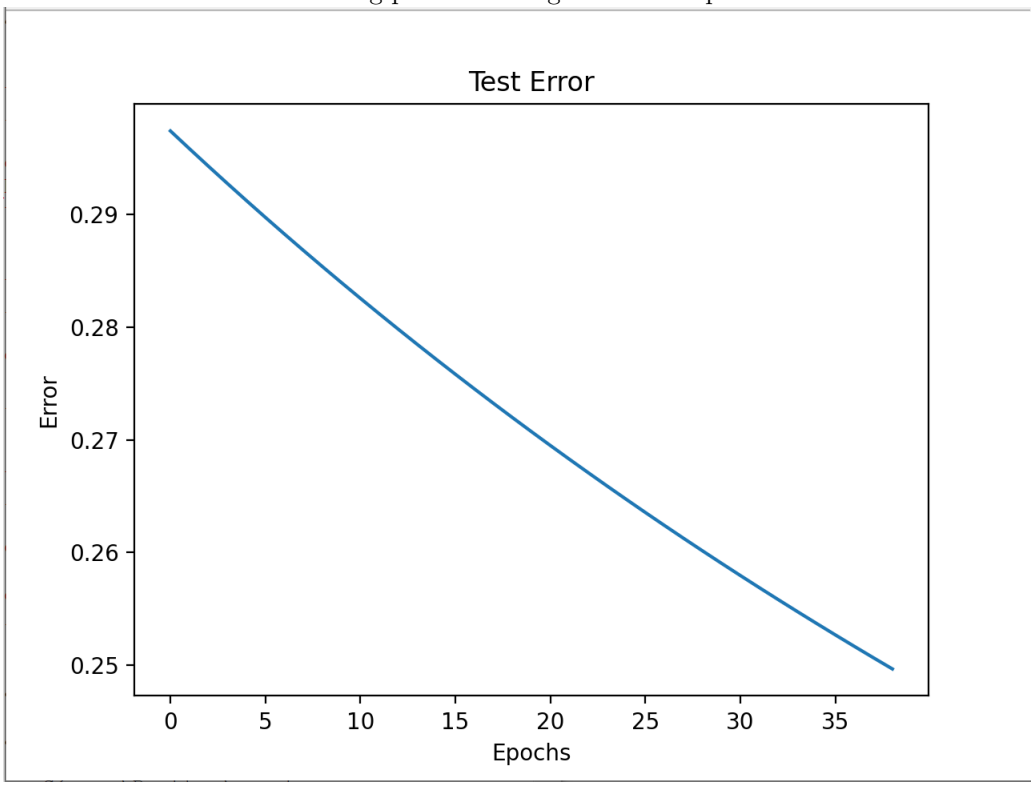


5.2 Testing

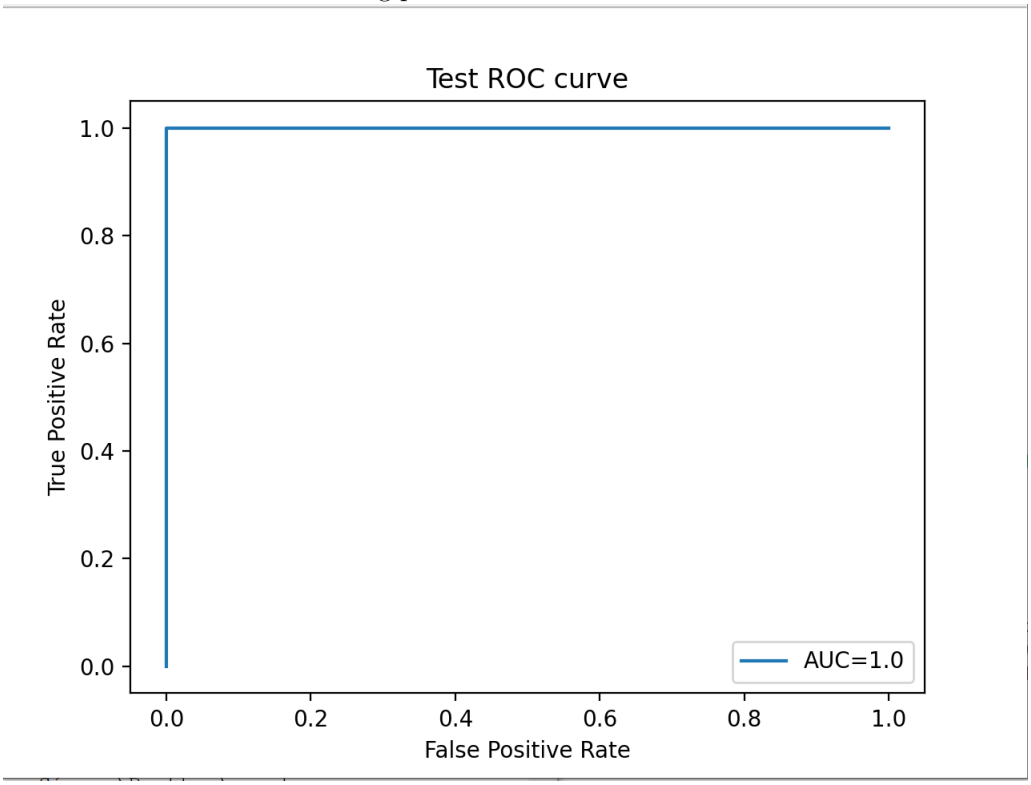
Testing process confusion matrix



Error of the testing process through out 2000 epochs

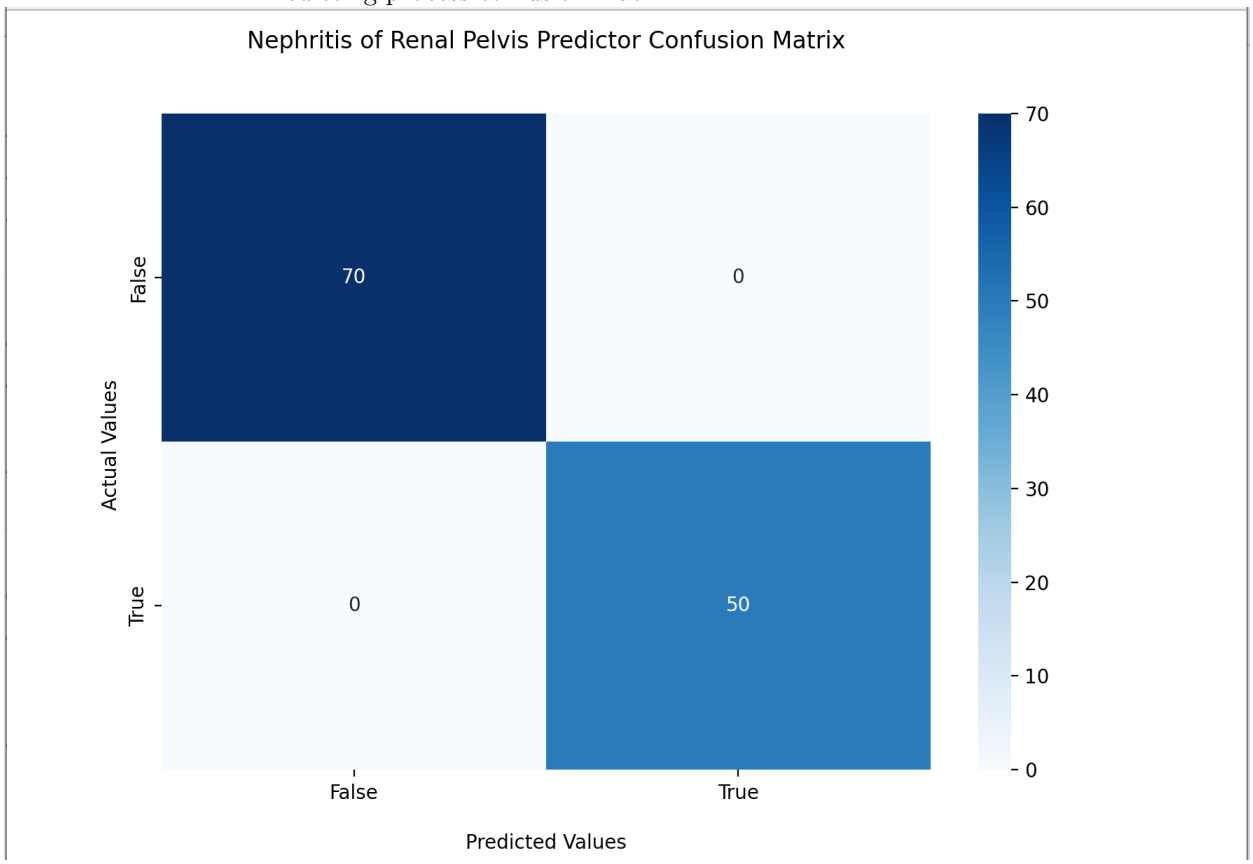


Testing process ROC curve

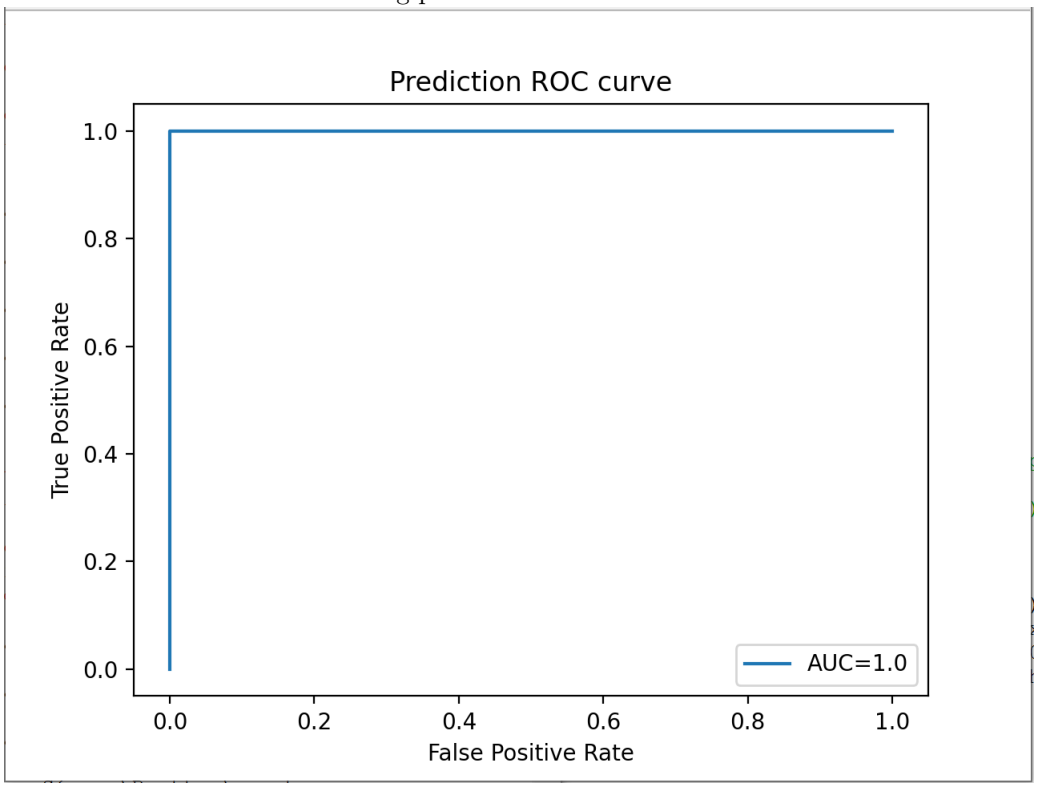


5.3 Predicting

Predicting process confusion matrix



Predicting process ROC curve



6 Conclusion

Even Though the model results might show a good performance, I might not recommend the use of this model in a real life application due to the fact that the model was trained and tested with a very small data set. There are different techniques that can be done in order to get a more reliable model using small data sets, such as cross validation. However, that techniques were not implemented for this delivery and I would prefer to compare the model performance against another ones or with bigger data sets.

7 References

Lee, B. (2019). Acute Nephritis. 10/05/2022, de healthline Sitio web: https://www.healthline.com/health/kidne
health-warning-signs
https://archive.ics.uci.edu/ml/datasets/Acute+Inflammations