Week 4: Testing Classifiers

Hans Georg Schaathun

25th April 2017

Time	Topic	Reading
8.15-	Recap of tutorials last week	
	Lecture: Testing and Error Estimation	Marsland Section 2.2
9.00-	Tutorial 4.1	
11.45-	Lunch break	
12.15-	Recap/discussion of Tutorial 6	
	Lecture: Linear and non-linear classifiers	Marsland Chapter 3.4-3.5 and introduction to Chapter 4
13.00-	Tutorial 4.2	

This PDF document is available in an HTML version at `http://www.hg.schaathun.net/FPIA/week04.html`.

1 Tutorial 4.1: Testing and Error Estimation

- **Reading:** Stephen Marsland: Chapter 2.2
- Look up *estimation* and *confidence intervals* in statistics if you do not remember what it is and how to do it (e.g. Johnson and Bhattacharyya).

1.1 Problem 1: Splitting the data set

From previous exercises, you should have a function which loads the data set and returns a list of pairs, where each pair contains a class label and a feature vector. We need a function to split it into a training set and a test set.

Ideally we should split randomly, but randomness is tricky. We will learn it next week; for now we split it naïvly and deterministically. Still, we need to make sure that both classes are represented in both sets. There are many ways to do this, but let's take the opportunity to practice list processing in Haskell.

1.1.1 Step 1: Splitting the classes

Write a function which takes a list of class label/feature vector pairs, and returns a list of lists, where each list includes pairs with the same class label.

1. We start with a function `getByClass` to get all the objects of a given class. Add the following declaration to your `ANNData` module.

```
getByClass :: Double -> [(Double,[Double])]
                     -> [(Double,[Double])]
```

   The first argument is a class c, and the second is a data set as returned by `getData` in the previous tutorial. The return value is a list of those objects from the input which belong to class c.

2. Use list comprehension to add a definition for `getByClass`.

3. Secondly, we add a function to sort the entire data set by class. Add the following type declaration to `ANNData`:

```
splitClasses :: [Double] -> [(Double,[Double])]
                         -> [[(Double,[Double])]]
```

   The first argument is the list of class labels in use. The second is a data set as returned from `getData`. The output is a list of lists, where each constituent list is the return value of a call to `getByClass`

4. Use `map` and `getByClass` to define `splitClasses`.

Optional improvement The above solution is not optimised for speed.

1. How many times does the `splitClasses` have to read through the `dl` list?

2. How can process the input list in a single pass?

1.1.2 Step 2: Training and Test sets

Having a list of lists of objects as returned from `splitClasses`, we need to take a fraction p of the elements for training and the remaining $1-p$ of elements fort testing. from each list for training and the remainder for testing. We are looking for the function with the following type.

```
mkTestTrainSets :: Double -> [[(Double,[Double])]]
    -> ([(Double,[Double])],[(Double,[Double])])
```

The first argument is the percentage p. The second argument is the list of lists as produced by the function in Step 1.

1. The `mkTestTrainSets` function takes a list of lists as input. Let's start with a helper function which takes just one lists and splits it in two:

```
mkTestTrainSets' :: Double -> [(Double,[Double])]
                 -> ([(Double,[Double])],[(Double,[Double])])
```

   Add the declaration to `ANNData`

2. Write p and v for the inputs, and (v_1, v_2) for the output. Implement `mkTestTrainSets'` such that v_1 contains a fraction p of the elements from v and v_2 contains the rest. You need the following steps:

   a) find the length l of v

   b) calculate the number of elements $p \cdot l$ for v_1

   c) calculate the number of elements $(1 - p)l$ for v_2

   d) split the input list v into v_1 and v_2

   In order to multiply an integer (l) with a float (p) you need to convert the integer using the `fromIntegral` function. You find the list functions you need in the list on page 127 of Simon Thompson's book.

3. We can now complete `mkTestTrainSets` as follows.

```
mkTestTrainSets :: Double -> [[(Double,[Double])]]
                -> ([(Double,[Double])],[(Double,[Double])])
mkTestTrainSets _ [] = ([],[])
mkTestTrainSets f (d:dl) = prepend e l
              where l = mkTestTrainSets f dl
                    e = mkTestTrainSets' f d
                    prepend (x,y) (x',y') = (x++x',y++y')
```

4. Discuss: What exactly does the local `prepend` function above do?

5. Discuss: Could this function have been written differently?

1.1.3 Step 3: Testing it

1. Load your `ANNData` module in GHCi.

2. Run the following test:

```
s1 <- getData "wdbc.data"
let s2 = splitClasses [0.0,1.0] s1
mkTestTrainSets 0.2 s2
```

3. Discuss: Is the output as expected? What should you expect?

1.2 Problem 2: Testing on a single item

Given a test set (as obtained in Problem 1) and a trained perceptron (as obtained in Tutorial 4), we are going to test the perceptron. Let's do a simple test for now, where we do not distinguish between false positives and false negatives.

1. Add the following declaration to your `Perceptron` module:

   ```
   testNeuron' :: Neuron -> (Double,[Double])] -> Bool
   ```

   The first input is a (trained) neuron and the second is a label/feature vector pair. The output is True if the given feature vector is correctly classified by the neuron.

2. Implement the `testNeuron'` function. You can use the following skeleton as a basis

   ```
   testNeuron' n (t,xs) =
                       where y =
   ```

   You need to calculate the output `y` from the neuron and compare it to the target value `t`.

3. Discuss: How can you test this function? If possible, make a test.

1.3 Problem 3: Testing on a data set

Now we have a function to test a single neuron. How do you test the neuron on every item in the test set?

1. Add the following type declaration to your `Perceptron` module:

   ```
   testNeuron :: Neuron -> [(Double,[Double]) -> [Bool]
   ```

   The return value is a list of boolean values, where True indicates an error and False indicates a correct classification.

2. Add a definition for the `testNeuron` function, by applying `testNeuron'` to every object in the input list.

3. Discuss: How can you test this function? If possible, make a test.

1.4 Problem 4: Putting it together

Having solved Problem 2 as well as Tutorial 4, we are able to both train and test a perceptron. Now we need to put it all into one executable program.

1. Create a Main module which loads the breast cancer data trains a perceptron on part of the data, and tests it on the remaining data. You may use this code:

```haskell
module Main where

-- Import your own modules:
import ANNData      -- load CSV data
import Perceptron   -- neuron

p = 0.1


main :: IO ()
main = do
    dl <- getData "wdbc.data"
    let (testS,trainS) = mkTestTrainSets p (splitClasses [0,1] dl)
    let dlen = length $ snd $ head testS
    print $ "Input length: " ++ show dlen
    let untrainedNeuron = initNeuron dlen
    let trainTarget = map fst trainS
    let trainInput = map snd trainS
    let trainedNeuron = train 1000 0.25 trainInput trainTarget untrain
    print "Untrained network, test data:"
    testPrint untrainedNeuron testS
    print "Untrained network, training data:"
    testPrint untrainedNeuron trainS
    print "Trained network, test data:"
    testPrint trainedNeuron testS
    print "Trained network, training data:"
    testPrint trainedNeuron trainS

testPrint n s = do
    let result = testNeuron n s
    let right = length [ x | x <- result, x ]
    let wrong = length [ x | x <- result, not x ]
    print ( "Correct: " ++ show right   )
    print ( "Errors:  " ++ show wrong )
```

2. Add comments in the `Main` module to explain what each line does.
3. Discuss: What is the constant `p` used for?
4. Compile and test the program.
5. Discuss the output of the program. How do you interpret the different numbers?
6. Discuss. Are the error counts reasonable or not?

It is quite possible that you get a lot of classification errors. Don't worry if you do. It is likely

because the single neuron is a bit too simple. Next week we will discuss how to build networks of multiple neurons.

1.4.1 Problem 5: scaling of features

A common problem in machine learning is that features with very high magnitude may dominate over others, so that the classifier is not properly trained in all dimensions.

1. Look at the breast cancer data set. What is the range of the features in different columns?
2. We have prepared a scaled version of the data set. All the features have been brought into the [0,1] range by linear scaling. Download the file and put it together with the other files.
3. Replace `wdbc.data` with `wdbc.csv` in your `Main` module.
4. Recompile and test the program.
5. Compare your error rates to those of your class mates.
6. Discuss: How does scaling affect the error rates?

If the error rates are still bad, don't worry. We will extend the single neuron perceptron to a neural network in the next couple of tutorials.

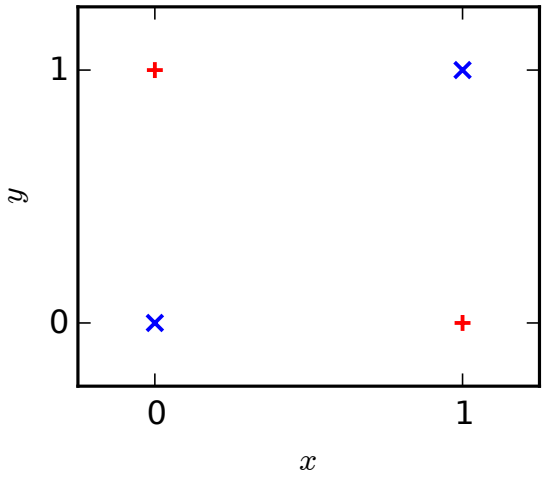
1.5 Problem 6: Statistical analysis

Given the number of errors e and the number of tests n, we can calculate the error rate e/n. We are interested in the *probability* of error, when the classifier is used on a random, unknown object. Answer the following:

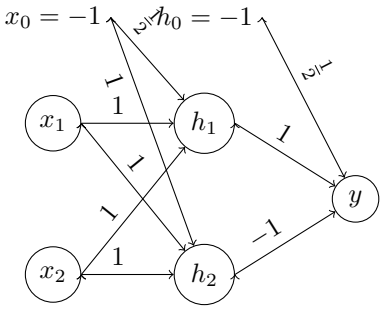
1. Discuss: What is the relationship between the error rate and the probability of error?
2. Discuss: What is the probability distribution of the number of errors e?
3. Calculate a 95% confidence interval for the error probability.

2 Tutorial 4.2: Linear and non-linear classifiers

Reading: Stephen Marsland: Chapter 3.4-3.5 and the introduction for Chapter 4.



(a) The problem



(b) The solution

Figure 1: The XOR classification problem.

2.1 Problem 1: A Neural Network for the XOR problem

2.1.1 Step 1: A new module

1. Review your solution to Problem 2 of Tutorial 3.2.
2. Create a new module `ANN`, where you import your existing `Perceptron` module.

```
import Perceptron
```

3. Define a data type `Network` which consist of several layers. Add it to the `ANN` module.

2.1.2 Step 2: A network for XOR

Recall the XOR classification problem in Figure 1a. We can use the solution (Figure 1b) to test our neural network and the `recall` function before we have to start thinking about training.

1. Define a neural network `xorNetwork` with the weights given in Figure 1b. Use the `Network` type which we defined above.
2. Show the network in GHCi, to check that you have not made a syntax error or similar.

```
xorNetwork
```

3. Now we need to implement a recall function. Add the type declaration to your `ANN` module.

```
recallNetwork :: Network -> [Double] -> [Double]
```

4. Implement `recallNetwork`, using `recallLayer` from Problem 2 of Tutorial 5.
5. Discuss: Which is the output layer? Is it first or last element in the list which makes up the `Network`?
6. Finally, test your definitions with the following evaluations:

```
recallNetwork xorNetwork [0,0]
recallNetwork xorNetwork [0,1]
recallNetwork xorNetwork [1,0]
recallNetwork xorNetwork [1,1]
```

7. Discuss: Is the output as expected?

2.1.3 Step 3: Bug search

There are two common sources of errors in this network/implementation.

1. What is the value of the threshold function at 0? I.e. does the neuron fire when the sum is exactly 0. In floating point problems, this hardly matters, but with the binary xor problem it does. If your test fails, try to change the threshold function.
2. What is the sign of the quasi-input x_0? Very often we use -1, but some authors use +1.