Prac Sheet – Intro Machine Learning

Last modified on Saturday, 30 April 2022 by f.maire@qut.edu.au

The main goal of ML is **good generalization**. In this prac, we illustrate the problem by fitting polynomial functions of various degrees to a set of points. You will implement a training algorithm for those parameterized functions and perform some experiments to investigate how the generalization depends on the size of the training set and the complexity of the learning machine (here the humble polynomial, but your findings will apply to deep neural networks too!).

First, let us explain how the pseudo-inverse allows us to select an optimal weight vector for the parameterized function $P(x,w)$.

The pseudo-inverse of a matrix A (not necessarily invertible) is denoted by A^+. The most interesting property of a pseudo-inverse is that $\|Aw-t\|^2$ is minimum for $w=A^+t$. Let $P(x,w)=\sum_{i=0}^{n} w_i x^i$ be a polynomial, and let $x_0, x_1, \ldots, x_{m-1}$ be m scalar points, and $t_0, t_1, \ldots, t_{m-1}$ be m scalar target values. We wish to find a weight vector $w=(w_0, w_1, \ldots, w_n)^T$ such that $\sum_{i=0}^{m-1} \left(P(x_i,w)-t_i\right)^2$ is minimum. Let $A=\begin{bmatrix} 1 & x_0^1 & \cdots & x_0^n \\ 1 & x_1^1 & \cdots & x_1^n \\ \vdots & \vdots & & \vdots \\ 1 & x_{m-1}^1 & \cdots & x_{m-1}^n \end{bmatrix}$. By definition, the vector w that we are looking for minimizes $\|Aw - t\|^2$, where $t=(t_0, t_1, \ldots, t_{m-1})^T$.

The pseudo-inverse (numpy.linalg.pinv in Python) provides a simple solution to the problem of finding a polynomial approximation to the mapping that generated the data set $\{(x_0,t_0),(x_1,t_1),\ldots,(x_{m-1},t_{m-1})\}$. After creating the vector t and the matrix A, we can compute the optimal coefficient vector for the polynomial with the formula $w=A^+t$.

Although we can use lists to represent vectors and lists of lists to represent matrices, a more efficient data structure is provided by the numpy library. Numpy is traditionally imported with the statement import numpy as np to keep the expressions short.

This library defines a class np.array that allows you to build and manipulate multi-dimensional numerical arrays.

Here are a few examples

```
w = np.array([3,0,5])   # create a vector with 3 entries (initialized with [3,0,5]).
A = np.array([[1,2],[3,4]]) # create a 2 by 2 matrix with first row  [1,2] and second row [3,4].
```

Note that to multipy a 3 by 3 matrix A and a vector w with 3 entries, you must make sure that the shape of w is (3,1).

```
>>> import numpy as np   # this import statement is needed to access the ndarray class
>>> w = np.array([3,0,5])  # create a 1D vector. It is neither a column vector nor a row vector!
>>> w.shape
(3,)  # incompatible with a 3x3 matrix. We  need to reshape w as a column vector
>>> w1=w.reshape((3,1))  # create a column vector view of w (a matrix with 3 rows and 1 column)
```

Warning: w and w1 share the same memory space!

```
>>> w1
array([[3],
       [0],
       [5]])
>>> A = np.diag([3,2,1])  # create a 3 by 3 matrix with diagonal [3,2,1]
>>> A
array([[3, 0, 0],
       [0, 2, 0],
       [0, 0, 1]])
>>> q = A.dot(w1)  # the matrix product of A and w1
>>> q
array([[9],
       [0],
       [5]])
```

Warning: the multiplication operator * works **elementwise**.

```
>>> A  = np.array([[1,0,3],[7,-5,4]])  # 2x3 matrix
>>> B  = np.array([[-2,7,2],[-3,0,4]])  # 2x3 matrix
>>> A*B  # elementwise multiplication
array([[ -2,   0,   6],
       [-21,   0,  16]])
>>> A[0,:]*B[1,:]  # multiplying row 0 of A with row 1 of B elementwise
array([-3,  0, 12])
```

```
>>> A.T  # the transpose method T returns the transpose view of a matrix (shared memory)
array([[ 1,  7],
       [ 0, -5],
       [ 3,  4]])
```

Exercise 1

Implement the Python function;

```
def polynom_1(x,w):
  '''
    Return
        y = w[0]+w[1]  x**1 +  w[2] x**2 + ..+ w[n]  x**n
        where
              n is the index of the last entry of w,
    Params
        w : vector of the polynomial coefficients (n+1 elements)
        x  : scalar
  '''
```

Exercise 2

Implement the Python function;

```
def polynom_2(x,w):
  '''
    Return
        y = w[0]+w[1]  x**1 +  w[2] x**2 + ..+ w[n]  x**n
        where
              n is the index of the last entry of w,
              and y has the same shape as x
    Params
        w : vector of the polynomial coefficients (n+1 elements)
        x  : a numpy array
  '''
```

Exercise 3

Implement the Python function;

```
def  powerCol(x,n):
'''
  Return
    the column vector [1 x x^2 ... x^n]   if x is a scalar
    otherwise (if x is a 1D array) return

        | 1    ..        1     |
        | x0   ..       xm     |
  y =   |      :               |       if x is a vector x = [x0 x1 .. xm]
        |x**n  ..   xm**n  |
'''
```

Exercise 4

Before attempting this exercise, you should look up the function **np.linspace**, **np.sin**, **np.random.randn** from the numpy library.

To plot a function you can use the following code

```
import matplotlib.pyplot as plt
x_plot = np.linspace(-3,3,100)
y_plot = polynom(x_plot,w)
plt.plot(x_plot, np.sin(x_plot),'b-',label="target")
plt.plot(x_plot, y_plot,'r-',label="approx")
plt.legend(loc='upper left')
plt.show()
```

Write a Python script to compute the best polynomial approximation of degree n that fits the data set $\{(x_0,t_0),(x_1,t_1),\ldots,(x_{m-1},t_{m-1})\}$
Use your script to experiment with the quality of the polynomial approximation when n varies, and your data set is made of m = 10 noisy points of the sin function on [-3, +3]
What can you conclude from these experiments?

Exercise 5 (not a programming exercise)

- Recall Bayes rule, P(A | B) = P(B | A) P(A) / P(B)
- Show that we have also P(A | B,C) = P(B | A,C) P(A | C) / P(B | C)

Exercise 6 (not a programming exercise)

- Suppose we want to fit a linear function $f_w(x) = wx$ where x is scalar to a data set $\{(x_1,y_1),\ldots,(x_n,y_n)\}$.
- Compute the derivative of the error $E = \sum_{i=1}^{n} (y_i - w x_i)^2$ with respect to w.
- What is the most probable w?