sheet4

November 10, 2017

Group: LGLLK

1 Sheet 4: Rounding, Overflow, Linear Algebra

In this exercise sheet, we look at various sources of numerical overflow when executing Python and numpy code for large input values, and how to efficiently handle them, for example, by using numpy special functions.

```
In [120]: import numpy,utils
```

1.1 Building a robust "softplus" nonlinear function (40 P)

The softplus function is defined as:

$$\text{softplus}(x) = \log(1 + \exp(x)).$$

It intervenes as elementary computation in certain machine learning models such as neural networks. Plotting it gives the following curve

where the function tends to zero for very negative input values and tends to the identity for very positive input values.

```
In [121]: def softplus(z): return numpy.log(1+numpy.exp(z))
```

We consider an input vector from the module `utils` containing varying values between 1 and 10000. We would like to apply the `softplus` function to all of its element in an element-wise manner.

```
In [122]: X = utils.softplus_inputs
          print(X)
```

```
[-10000, -1000, -100, -10, -1, 0, 1, 10, 100, 1000, 10000]
```

We choose these large values in order to test whether the behavior of the function is correct in all regimes of the function, in particular, for very small or very large values. The code below applies the `softplus` function directly to the vector of inputs and then prints for all cases the input and the corresponding function output:

plot generated using fooplot.com

```
In [123]: Y = softplus(X)
          for x,y in zip(X,Y):
              print('softplus(%11.4f) = %11.4f'%(x,y))
```

```
softplus(-10000.0000) =      0.0000
softplus( -1000.0000) =      0.0000
softplus(  -100.0000) =      0.0000
softplus(   -10.0000) =      0.0000
softplus(    -1.0000) =      0.3133
softplus(     0.0000) =      0.6931
softplus(     1.0000) =      1.3133
softplus(    10.0000) =     10.0000
softplus(   100.0000) =    100.0000
softplus(  1000.0000) =         inf
softplus( 10000.0000) =         inf
```

```
C:\Program Files\Anaconda2\lib\site-packages\ipykernel_launcher.py:1: RuntimeWarning: overflow e
  """Entry point for launching an IPython kernel.
```

For large input values, the softplus function returns `inf` whereas analysis of that function tells us that it should compute the identity. Let's now try to apply the softplus function one element at a time, to see whether the problem comes from numpy arrays:

```
In [124]: for x in X:
              y = softplus(x)
              print('softplus(%11.4f) = %11.4f'%(x,y))
```

```
softplus(-10000.0000) =      0.0000
softplus( -1000.0000) =      0.0000
softplus(  -100.0000) =      0.0000
softplus(   -10.0000) =      0.0000
softplus(    -1.0000) =      0.3133
softplus(     0.0000) =      0.6931
softplus(     1.0000) =      1.3133
softplus(    10.0000) =     10.0000
softplus(   100.0000) =    100.0000
softplus(  1000.0000) =         inf
softplus( 10000.0000) =         inf
```

```
C:\Program Files\Anaconda2\lib\site-packages\ipykernel_launcher.py:1: RuntimeWarning: overflow e
  """Entry point for launching an IPython kernel.
```

Unfortunately, the result is the same. We observe that the function always stops working when its output approaches 1000, even though the input was given in high precision `float64`.

- Create an alternative function for `softplus` that applies to input scalars and that correctly applies to values that can be much larger than 1000 (e.g. billions or more). Your function can be written in Python directly and does not need numpy parallelization.

```
In [125]: def softplus_opt(Z):
              result = []
              for z in Z:
                  if z > 100:
                      result += [z]
                  else:
                      result += [numpy.log(1+numpy.exp(z))]
              return result
```

```
In [126]: Y = softplus_opt(X)
          for x,y in zip(X,Y):
              print('softplus(%11.4f) = %11.4f'%(x,y))
```

```
softplus(-10000.0000) =      0.0000
softplus( -1000.0000) =      0.0000
softplus(  -100.0000) =      0.0000
softplus(   -10.0000) =      0.0000
softplus(    -1.0000) =      0.3133
softplus(     0.0000) =      0.6931
softplus(     1.0000) =      1.3133
softplus(    10.0000) =     10.0000
softplus(   100.0000) =    100.0000
softplus(  1000.0000) =   1000.0000
softplus( 10000.0000) =  10000.0000
```

As we have seen in the previous exercise sheet, the problem of functions that apply to scalars only is that they are less efficient than functions that apply to vectors directly. Therefore, we would like to handle the rounding issue directly at the vector level.

- Create a new softplus function that applies to vectors and that has the desired behavior for large input values. Your function should be fast for large input vectors (i.e. it is not appropriate to use an inner Python loop inside the function).

```
In [127]: # using clip to give the boundary of input vector.
          # Ref in lecture4 "The sigmoid function (4)"

          def softplus_npopt(Z):
              clipZ = numpy.clip(Z, -10000, 100)
              Y = numpy.log(1+numpy.exp(clipZ))
              results = numpy.where(clipZ == 100, Z, Y)
              return results

          Y = softplus_npopt(X)
```

```
In [ ]: for x,y in zip(X,Y):
            print('softplus3(%11.4f) = %11.4f'%(x,y))
```

```
softplus3(-10000.0000) =      0.0000
softplus3( -1000.0000) =      0.0000
softplus3(  -100.0000) =      0.0000
softplus3(   -10.0000) =      0.0000
softplus3(    -1.0000) =      0.3133
softplus3(     0.0000) =      0.6931
softplus3(     1.0000) =      1.3133
softplus3(    10.0000) =     10.0000
softplus3(   100.0000) =    100.0000
softplus3(  1000.0000) =   1000.0000
softplus3( 10000.0000) =  10000.0000
```

1.2 Computing a partition function (30 P)

We consider a discrete probability distribution of type

$$p(\boldsymbol{x};\boldsymbol{w}) = \frac{1}{Z(\boldsymbol{w})}\exp(\boldsymbol{x}^\top \boldsymbol{w})$$

where $\boldsymbol{x} \in \{-1,1\}^{10}$ is an observation, and $\boldsymbol{w} \in \mathbb{R}^{10}$ is a vector of parameters. The term $Z(\boldsymbol{w})$ is called the partition function and is chosen such that the probability distribution sums to 1. That is, the equation:

$$\sum_{\boldsymbol{x}\in\{-1,1\}^{10}} p(\boldsymbol{x};\boldsymbol{w}) = 1$$

must be satisfied. Below is a simple method that computes the log of the partition function $Z(\boldsymbol{w})$ for various choices of parameter vectors. The considered parameters (`w_small`, `w_medium`, and `w_large`) are increasingly large (and thus problematic), and can be found in the file `utils.py`.

```
In [128]: import numpy,utils
          import itertools

          def getlogZ(w):
              Z = 0
              for x in itertools.product([-1, 1], repeat=10):
                  Z += numpy.exp(numpy.dot(x,w))
              return numpy.log(Z)

          print('%11.4f'%getlogZ(utils.w_small))
          print('%11.4f'%getlogZ(utils.w_medium))
          print('%11.4f'%getlogZ(utils.w_big))
```

```
    18.2457
    89.5932
        inf
```

```
C:\Program Files\Anaconda2\lib\site-packages\ipykernel_launcher.py:7: RuntimeWarning: overflow e
  import sys
```

We can observe from these results, that for parameter vectors with large values (e.g. `utils.w_big`), the exponential function overflows, and thus, we do not obtain a correct value for the logarithm of `Z`.

- Implement an improved function that avoids the overflow problem, and evaluate the partition function for the same parameters.

```
In [129]: def getlogZ_opt(w):

              # use the log-sum-exp trick. for the proof see
              # https://www.xarg.org/2016/06/the-log-sum-exp-trick-in-machine-learning/

              argForExp_i1 = []

              for x in itertools.product([-1, 1], repeat=10):
                  argForExp = numpy.dot(x,w)
                  argForExp_i1.append( argForExp )

              offsetForExpArgs = max( argForExp_i1 )

              Zrest = 0.
              for x in itertools.product([-1, 1], repeat=10):
                  Zrest += numpy.exp ( numpy.dot(x,w)-offsetForExpArgs )

              lnZ = offsetForExpArgs + numpy.log(Zrest)

              return lnZ

          print('%11.4f'%getlogZ_opt(utils.w_small))
          print('%11.4f'%getlogZ_opt(utils.w_medium))
          print('%11.4f'%getlogZ_opt(utils.w_big))
```

```
    18.2457
    89.5932
 24921.9913
```

- For the model with parameter `utils.w_big`, evaluate the log-probability of the binary vectors contained in the list `itertools.product([-1, 1], repeat=10)`, and return the indices (starting from 0) of those that have probability greater or equal to 0.001.

```
In [130]: lnZ_wBig = getlogZ_opt( utils.w_big )

          def logP(x):
```

```
            logP = - lnZ_wBig + numpy.dot(x,utils.w_big)
            return logP

        def getIndicesOfProbsLargerOrEqualThan( probLowerBound ):
            iBinVec = 0
            iLargerOrEqual = []
            for binVec in itertools.product([-1, 1], repeat=10):
                logPBinVec = logP(binVec)
                pBinVec = numpy.exp( logPBinVec )
                if( pBinVec >= probLowerBound ):
                    iLargerOrEqual.append( iBinVec )
                iBinVec += 1

            return iLargerOrEqual

        print( getIndicesOfProbsLargerOrEqualThan( 1e-3 ) )
```

```
[81, 83, 85, 87, 209, 211, 213, 215, 337, 339, 341, 343, 465, 467, 469, 471, 597, 599, 725, 727,
```

1.3 Probability of generating data from a Gaussian model (30 P)

Consider a multivariate Gaussian distribution of mean vector `m` and covariance `S`. The probability associated to a vector `x` is given by:

$$p(\boldsymbol{x};(\boldsymbol{m},S)) = \frac{1}{\sqrt{(2\pi)^d \det(S)}} \exp\Big(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{m})^\top S^{-1}(\boldsymbol{x}-\boldsymbol{m})\Big)$$

We consider the calculation of the probability of observing a certain dataset

$$\mathcal{D} = (\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(N)})$$

assuming the data is generated according to a Gaussian distribution of fixed parameters $\boldsymbol{m}$ and S. Such probability density is given by the formula:

$$\log P(\mathcal{D};(\boldsymbol{m},S)) = \log \prod_{i=1}^{N} p(\boldsymbol{x}^{(i)};(\boldsymbol{m},S))$$

The function below implements such function:

```
In [131]: import numpy,numpy.linalg,utils

        def logp(X,m,S):

            # Find the number of dimensions from the data vector
            d = X.shape[1]

            # Invert the covariance matrix
            Sinv = numpy.linalg.inv(S)
```

```
    # Compute the quadratic terms for all data points
    Q = -0.5*(numpy.dot(X-m,Sinv)*(X-m)).sum(axis=1)

    # Raise them quadratic terms to the exponential
    Q = numpy.exp(Q)

    # Divide by the terms in the denominator
    P = Q / numpy.sqrt((2*numpy.pi)**d * numpy.linalg.det(S))

    # Take the product of the probability of each data points
    Pprod = numpy.prod(P)

    # Return the log-probability
    return numpy.log(Pprod)
```

Evaluation of this function for various datasets and parameters provided in the file `utils.py` gives the following probabilities:

```
In [132]: print(logp(utils.X1,utils.m1,utils.S1))
          print(logp(utils.X2,utils.m2,utils.S2))
          print(logp(utils.X3,utils.m3,utils.S3))
```

```
C:\Program Files\Anaconda2\lib\site-packages\ipykernel_launcher.py:24: RuntimeWarning: divide by
```

```
-13.0067700574
-inf
-inf
```

This function is numerically instable for multiple reasons. The product of many probabilities, the inversion of a large covariance matrix, and the computation of its determinant, are all potential causes for overflow. Thus, we would like to find a numerically robust way of performing each of these.

- Implement a numerically stable version of the function `logp`
- Evaluate it on the same datasets and parameters as the function `logp`

```
In [134]: #This function is numerically instable for multiple reasons.
          # 1 The product of many probabilities,
          # 2 the inversion of a large covariance matrix,
          # 3 and the computation of its determinant,
          # are all potential causes for overflow.

          def logp_opt(X,m,S):

              # Find the number of dimensions from the data vector
              d = X.shape[1]
```

```
            # prob2 = inverting large matrix.
            L = numpy.linalg.cholesky(S)
            LT = numpy.transpose( L )
            L_inv = numpy.linalg.inv(L)
            LT_inv = numpy.linalg.inv(LT)
            # prob3 = computation of det.
            detS = numpy.linalg.det( L ) * numpy.linalg.det( LT )

            # Compute the quadratic terms for all data points
            vecDev = X-m
            vecDevMapped1 = numpy.dot( vecDev , LT_inv )
            vecDevMapped2 = numpy.dot( vecDevMapped1 , L_inv )
            prodComponents_i1 = vecDevMapped2 * vecDev
            scalProd = prodComponents_i1.sum(axis=1)
            Q = -0.5*scalProd

            # Raise them quadratic terms to the exponential
            Q = numpy.exp(Q)

            # Divide by the terms in the denominator
            P = Q / numpy.sqrt((2*numpy.pi)**d * detS )

            # prob1 -> solution = replace prod by sum.
            # Take the product of the probability of each data points
            lnP = numpy.sum( numpy.log(P) )

            # Return the log-probability
            return lnP

        print(logp_opt(utils.X1,utils.m1,utils.S1))
        print(logp_opt(utils.X2,utils.m2,utils.S2))
        print(logp_opt(utils.X3,utils.m3,utils.S3))
```

```
-13.0067700574
-1947.97098067
-inf
```

```
C:\Program Files\Anaconda2\lib\site-packages\ipykernel_launcher.py:36: RuntimeWarning: divide by
```