cffi_linear_regression

November 26, 2021

1 Optimisation de code avec cffi, numba, cython

L'idée est de recoder une fonction en C. On prend comme exemple la fonction de prédiction de la régression linéaire de scikit-learn et de prévoir le gain de temps qu'on obtient en recodant la fonction dans un langage plus rapide.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

[2]:
```
memo_time = []
import timeit

def unit(x):
    if x >= 1: return "%1.2f s" % x
    elif x >= 1e-3: return "%1.2f ms" % (x* 1000)
    elif x >= 1e-6: return "%1.2f µs" % (x* 1000**2)
    elif x >= 1e-9: return "%1.2f ns" % (x* 1000**3)
    else:
        return "%1.2g s" % x

def timeexe(legend, code, number=100, repeat=1000):
    rep = timeit.repeat(code, number=number, repeat=repeat, globals=globals())
    ave = sum(rep) / (number * repeat)
    std = (sum((x/number - ave)**2 for x in rep) / repeat)**0.5
    fir = rep[0]/number
    fir3 = sum(rep[:3]) / (3 * number)
    las3 = sum(rep[-3:]) / (3 * number)
    rep.sort()
    mini = rep[len(rep)//20] / number
    maxi = rep[-len(rep)//20] / number
    print("Moyenne: %s Ecart-type %s (with %d runs) in [%s, %s]" % (
                unit(ave), unit(std), number, unit(mini), unit(maxi)))
    return dict(legend=legend, average=ave, deviation=std, first=fir, first3=fir3,
                last3=las3, repeat=repeat, min5=mini, max5=maxi, code=code, run=number)
```

1.1 Régression linéaire

[3]:
```
from sklearn.datasets import load_diabetes
diabetes = load_diabetes()
diabetes_X_train = diabetes.data[:-20]
diabetes_X_test  = diabetes.data[-20:]
diabetes_y_train = diabetes.target[:-20]
diabetes_y_test  = diabetes.target[-20:]
```

[4]:
```
from sklearn.linear_model import LinearRegression
clr = LinearRegression()
clr.fit(diabetes_X_train, diabetes_y_train)
```

[4]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

[5]:
```
clr.coef_
```

[5]:
```
array([ 3.03499549e-01, -2.37639315e+02,  5.10530605e+02,  3.27736980e+02,
       -8.14131709e+02,  4.92814588e+02,  1.02848452e+02,  1.84606489e+02,
        7.43519617e+02,  7.60951722e+01])
```

[6]:
```
clr.intercept_
```

[6]:
```
152.76430691633442
```

[7]:
```
z = diabetes_X_test[0:1,:]
memo_time.append(timeexe("sklearn.predict", "clr.predict(z)"))
```

```
Moyenne: 50.66 µs Ecart-type 13.18 µs (with 100 runs) in [40.06 µs, 75.88 µs]
```

[8]:
```
%timeit clr.predict(z)
```

```
55.5 µs ± 6.86 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

1.1.1 optimisation avec cffi

On s'inspire de l'exemple Purely for performance (API level, out-of-line).

[9]:
```
from cffi import FFI
ffibuilder = FFI()

ffibuilder.cdef("int linreg(int, double *, double *, double, double *);")

ffibuilder.set_source("_linear_regression",
r"""
    static int linreg(int dimension, double * x, double *coef, double intercept,␣
 ↪double * out)
    {
        for(; dimension > 0; --dimension, ++x, ++coef)
            intercept += *x * *coef;
        *out = intercept;
        return 1;
    }
```

```
""")

ffibuilder.compile(verbose=True)
```

```
generating .\_linear_regression.c
(already up-to-date)
the current directory is
'C:\\xavierdupre\\__home_\\GitHub\\ensae_teaching_cs\\_doc\\notebooks\\2a'
running build_ext
building '_linear_regression' extension
C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\bin\HostX86\x64\cl.exe /c
/nologo /Ox /W3 /GL /DNDEBUG /MD -Ic:\python372_x64\include
-Ic:\python372_x64\include "-IC:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\ATLMFC\include" "-IC:\Program
Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\include" "-IC:\Program Files
(x86)\Windows Kits\NETFXSDK\4.6.1\include\um" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\ucrt" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\shared" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\um" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\winrt" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\cppwinrt" /Tc_linear_regression.c
/Fo.\Release\_linear_regression.obj
C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\bin\HostX86\x64\link.exe /nologo
/INCREMENTAL:NO /LTCG /DLL /MANIFEST:EMBED,ID=2 /MANIFESTUAC:NO
/LIBPATH:c:\python372_x64\libs /LIBPATH:c:\python372_x64\PCbuild\amd64
"/LIBPATH:C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\ATLMFC\lib\x64"
"/LIBPATH:C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\lib\x64" "/LIBPATH:C:\Program
Files (x86)\Windows Kits\NETFXSDK\4.6.1\lib\um\x64" "/LIBPATH:C:\Program Files
(x86)\Windows Kits\10\lib\10.0.17763.0\ucrt\x64" "/LIBPATH:C:\Program Files
(x86)\Windows Kits\10\lib\10.0.17763.0\um\x64" /EXPORT:PyInit__linear_regression
.\Release\_linear_regression.obj /OUT:.\_linear_regression.cp37-win_amd64.pyd
/IMPLIB:.\Release\_linear_regression.cp37-win_amd64.lib
```

[9]:
```
'C:\\xavierdupre\\__home_\\GitHub\\ensae_teaching_cs\\_doc\\notebooks\\2a\\_line
ar_regression.cp37-win_amd64.pyd'
```

La fonction compilée est accessible comme suit.

[10]:
```
from _linear_regression import ffi, lib
lib.linreg
```

[10]:
```
<function _linear_regression.CompiledLib.linreg>
```

On s'inspire de l'exemple How to pass a Numpy array into a cffi function and how to get one back out?.

[11]:
```
import numpy
out = numpy.zeros(1)
```

[12]:
```
ptr_coef = clr.coef_.__array_interface__['data'][0]
cptr_coef = ffi.cast ( "double*" , ptr_coef )
```

[13]:
```
x = diabetes_X_test[0:1,:]
ptr_x = x.__array_interface__['data'][0]
cptr_x = ffi.cast ( "double*" , ptr_x )
```

[14]:
```
ptr_out = out.__array_interface__['data'][0]
cptr_out = ffi.cast ( "double*" , ptr_out )
```

[15]:
```
n = len(clr.coef_)
lib.linreg(n, cptr_x, cptr_coef, clr.intercept_, cptr_out)
```

[15]:
```
1
```

[16]:
```
out
```

[16]:
```
array([197.61846908])
```

On vérifie qu'on obtient bien la même chose.

[17]:
```
clr.predict(x)
```

[17]:
```
array([197.61846908])
```

Et on mesure le temps d'exécution :

[18]:
```
memo_time.append(timeexe("cffi-linreg", "lib.linreg(n, cptr_x, cptr_coef, clr.
↪intercept_, cptr_out)"))
```

```
Moyenne: 691.69 ns Ecart-type 301.22 ns (with 100 runs) in [470.00 ns, 1.09 µs]
```

C'est beaucoup plus rapide. Pour être totalement honnête, il faut mesurer les étapes qui consiste à extraire les pointeurs.

[19]:
```
def predict_clr(x, clr):
    out = numpy.zeros(1)
    ptr_coef = clr.coef_.__array_interface__['data'][0]
    cptr_coef = ffi.cast ( "double*" , ptr_coef )
    ptr_x = x.__array_interface__['data'][0]
    cptr_x = ffi.cast ( "double*" , ptr_x )
    ptr_out = out.__array_interface__['data'][0]
    cptr_out = ffi.cast ( "double*" , ptr_out )
    lib.linreg(len(x), cptr_x, cptr_coef, clr.intercept_, cptr_out)
    return out

predict_clr(x, clr)
```

[19]:
```
array([152.74058378])
```

[20]:
```
memo_time.append(timeexe("cffi-linreg-wrapped", "predict_clr(x, clr)"))
```

```
Moyenne: 12.64 µs Ecart-type 5.78 µs (with 100 runs) in [7.15 µs, 22.51 µs]
```

Cela reste plus rapide.

1.1.2 cffi - seconde version

Comme on construit la fonction en dynamique (le code est connu lors de l'exécution), on peut facilement se passer de la boucle et écrire le code sans boucle et avec les coefficients.

[21]:
```
res = " + ".join("{0}*x[{1}]".format(c, i) for i, c in enumerate(clr.coef_))
res
```

[21]:
```
'0.3034995490657413*x[0] + -237.6393153335348*x[1] + 510.53060543622456*x[2] +
327.73698040934715*x[3] + -814.1317093725389*x[4] + 492.8145879837313*x[5] +
102.84845219167991*x[6] + 184.60648905984044*x[7] + 743.5196167505418*x[8] +
76.09517221662408*x[9]'
```

[22]:
```
code = """
    static int linreg_custom(double * x, double * out)
    {{
        out[0] = {0} + {1};
    }}
""".format(clr.intercept_, res)
print(code)
```

```
    static int linreg_custom(double * x, double * out)
    {
        out[0] = 152.76430691633442 + 0.3034995490657413*x[0] +
-237.6393153335348*x[1] + 510.53060543622456*x[2] + 327.73698040934715*x[3] +
-814.1317093725389*x[4] + 492.8145879837313*x[5] + 102.84845219167991*x[6] +
184.60648905984044*x[7] + 743.5196167505418*x[8] + 76.09517221662408*x[9];
    }
```

[23]:
```
from cffi import FFI
ffibuilder = FFI()

ffibuilder.cdef("int linreg_custom(double *, double *);")
ffibuilder.set_source("_linear_regression_custom", code)
ffibuilder.compile(verbose=True)
```

```
generating .\_linear_regression_custom.c
(already up-to-date)
the current directory is
'C:\\xavierdupre\\__home_\\GitHub\\ensae_teaching_cs\\_doc\\notebooks\\2a'
running build_ext
building '_linear_regression_custom' extension
C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\bin\HostX86\x64\cl.exe /c
/nologo /Ox /W3 /GL /DNDEBUG /MD -Ic:\python372_x64\include
-Ic:\python372_x64\include "-IC:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\ATLMFC\include" "-IC:\Program
Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\include" "-IC:\Program Files
(x86)\Windows Kits\NETFXSDK\4.6.1\include\um" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\ucrt" "-IC:\Program Files (x86)\Windows
```

```
Kits\10\include\10.0.17763.0\shared" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\um" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\winrt" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\cppwinrt" /Tc_linear_regression_custom.c
/Fo.\Release\_linear_regression_custom.obj
C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\bin\HostX86\x64\link.exe /nologo
/INCREMENTAL:NO /LTCG /DLL /MANIFEST:EMBED,ID=2 /MANIFESTUAC:NO
/LIBPATH:c:\python372_x64\libs /LIBPATH:c:\python372_x64\PCbuild\amd64
"/LIBPATH:C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\ATLMFC\lib\x64"
"/LIBPATH:C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\lib\x64" "/LIBPATH:C:\Program
Files (x86)\Windows Kits\NETFXSDK\4.6.1\lib\um\x64" "/LIBPATH:C:\Program Files
(x86)\Windows Kits\10\lib\10.0.17763.0\ucrt\x64" "/LIBPATH:C:\Program Files
(x86)\Windows Kits\10\lib\10.0.17763.0\um\x64"
/EXPORT:PyInit__linear_regression_custom .\Release\_linear_regression_custom.obj
/OUT:.\_linear_regression_custom.cp37-win_amd64.pyd
/IMPLIB:.\Release\_linear_regression_custom.cp37-win_amd64.lib
```

[23]:
```
'C:\\xavierdupre\\__home_\\GitHub\\ensae_teaching_cs\\_doc\\notebooks\\2a\\_line
ar_regression_custom.cp37-win_amd64.pyd'
```

[24]:
```
from _linear_regression_custom.lib import linreg_custom
linreg_custom(cptr_x, cptr_out)
out
```

[24]:
```
array([197.61846908])
```

[25]:
```
memo_time.append(timeexe("cffi-linreg-custom", "linreg_custom(cptr_x, cptr_out)"))
```

```
Moyenne: 504.81 ns Ecart-type 282.17 ns (with 100 runs) in [352.00 ns, 836.00
ns]
```

On a gagné un facteur 2.

[26]:
```
def predict_clr_custom(x):
    out = numpy.zeros(1)
    ptr_x = x.__array_interface__['data'][0]
    cptr_x = ffi.cast("double*", ptr_x)
    ptr_out = out.__array_interface__['data'][0]
    cptr_out = ffi.cast("double*", ptr_out)
    linreg_custom(cptr_x, cptr_out)
    return out

predict_clr_custom(x)
```

[26]:
```
array([197.61846908])
```

[27]:
```
memo_time.append(timeexe("cffi-linreg-custom wrapped", "predict_clr_custom(x)"))
```

```
Moyenne: 6.79 µs Ecart-type 2.83 µs (with 100 runs) in [4.98 µs, 12.87 µs]
```

C'est un peu plus rapide.

1.1.3 et en float?

L'ordinateur fait la distinction entre les double code sur 64 bit et les float codé sur 32 bits. La précision est meilleure dans le premier cas et les calculs sont plus rapides dans le second. Dans le cas du machine learning, on préfère la rapidité à une perte précision en précision qui est souvent compensée par l'optimisation inhérente à tout problème de machine learning. Ce qu'on perd sur une observation, on le retrouve sur une autre.

[28]:
```
res = " + ".join("{0}f*x[{1}]".format(c, i) for i, c in enumerate(clr.coef_))
res
```

[28]:
```
'0.3034995490657413f*x[0] + -237.6393153335348f*x[1] + 510.53060543622456f*x[2]
+ 327.73698040934715f*x[3] + -814.1317093725389f*x[4] + 492.8145879837313f*x[5]
+ 102.84845219167991f*x[6] + 184.60648905984044f*x[7] + 743.5196167505418f*x[8]
+ 76.09517221662408f*x[9]'
```

[29]:
```
code = """
    static int linreg_custom_float(float * x, float * out)
    {{
        out[0] = {0}f + {1};
    }}
""".format(clr.intercept_, res)
print(code)
```

```
    static int linreg_custom_float(float * x, float * out)
    {
        out[0] = 152.76430691633442f + 0.3034995490657413f*x[0] +
-237.6393153335348f*x[1] + 510.53060543622456f*x[2] + 327.73698040934715f*x[3] +
-814.1317093725389f*x[4] + 492.8145879837313f*x[5] + 102.84845219167991f*x[6] +
184.60648905984044f*x[7] + 743.5196167505418f*x[8] + 76.09517221662408f*x[9];
    }
```

[30]:
```
from cffi import FFI
ffibuilder = FFI()

ffibuilder.cdef("int linreg_custom_float(float *, float *);")
ffibuilder.set_source("_linear_regression_custom_float", code)
ffibuilder.compile(verbose=True)
```

```
generating .\_linear_regression_custom_float.c
(already up-to-date)
the current directory is
'C:\\xavierdupre\\__home_\\GitHub\\ensae_teaching_cs\\_doc\\notebooks\\2a'
running build_ext
building '_linear_regression_custom_float' extension
C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\bin\HostX86\x64\cl.exe /c
/nologo /Ox /W3 /GL /DNDEBUG /MD -Ic:\python372_x64\include
-Ic:\python372_x64\include "-IC:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\ATLMFC\include" "-IC:\Program
Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\include" "-IC:\Program Files
```

```
(x86)\Windows Kits\NETFXSDK\4.6.1\include\um" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\ucrt" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\shared" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\um" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\winrt" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\cppwinrt" /Tc_linear_regression_custom_float.c
/Fo.\Release\_linear_regression_custom_float.obj
C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\bin\HostX86\x64\link.exe /nologo
/INCREMENTAL:NO /LTCG /DLL /MANIFEST:EMBED,ID=2 /MANIFESTUAC:NO
/LIBPATH:c:\python372_x64\libs /LIBPATH:c:\python372_x64\PCbuild\amd64
"/LIBPATH:C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\ATLMFC\lib\x64"
"/LIBPATH:C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\lib\x64" "/LIBPATH:C:\Program
Files (x86)\Windows Kits\NETFXSDK\4.6.1\lib\um\x64" "/LIBPATH:C:\Program Files
(x86)\Windows Kits\10\lib\10.0.17763.0\ucrt\x64" "/LIBPATH:C:\Program Files
(x86)\Windows Kits\10\lib\10.0.17763.0\um\x64"
/EXPORT:PyInit__linear_regression_custom_float
.\Release\_linear_regression_custom_float.obj
/OUT:.\_linear_regression_custom_float.cp37-win_amd64.pyd
/IMPLIB:.\Release\_linear_regression_custom_float.cp37-win_amd64.lib
```

[30]:
```
'C:\\xavierdupre\\__home_\\GitHub\\ensae_teaching_cs\\_doc\\notebooks\\2a\\_line
ar_regression_custom_float.cp37-win_amd64.pyd'
```

[31]:
```
from _linear_regression_custom_float.lib import linreg_custom_float
```

[32]:
```
def predict_clr_custom_float(x):
    out = numpy.zeros(1, dtype=numpy.float32)
    ptr_x = x.__array_interface__['data'][0]
    cptr_x = ffi.cast ( "float*" , ptr_x )
    ptr_out = out.__array_interface__['data'][0]
    cptr_out = ffi.cast ( "float*" , ptr_out )
    linreg_custom_float(cptr_x, cptr_out)
    return out
```

Avant d'appeler la fonction, on doit transformer le vecteur iniatial en float32.

[33]:
```
x32 = x.astype(numpy.float32)
predict_clr_custom(x32)
```

[33]:
```
array([152.76430692])
```

[34]:
```
memo_time.append(timeexe("cffi-linreg-custom-float wrapped",␣
↪"predict_clr_custom(x32)"))
```

```
Moyenne: 6.21 µs Ecart-type 2.34 µs (with 100 runs) in [5.14 µs, 10.12 µs]
```

La différence n'est pas flagrante. Mesurons le code C uniquement même si la partie Python ne peut pas être complètement évitée.

[35]:
```
out = numpy.zeros(1, dtype=numpy.float32)
ptr_x = x32.__array_interface__['data'][0]
```

```
cptr_x = ffi.cast ( "float*" , ptr_x )
ptr_out = out.__array_interface__['data'][0]
cptr_out = ffi.cast ( "float*" , ptr_out )

memo_time.append(timeexe("cffi-linreg-custom-float32", "linreg_custom_float(cptr_x,␣
↪cptr_out)"))
```

```
Moyenne: 421.78 ns Ecart-type 237.56 ns (with 100 runs) in [354.00 ns, 676.00
ns]
```

La différence n'est pas significative.

1.1.4 SIMD

C'est un ensemble d'instructions processeur pour faire des opérations terme à terme sur 4 float32 aussi rapidement qu'une seule. Le processeur ne peut faire des opérations que les nombres sont copiés dans ses registres. Le programme passe alors son temps à copier des nombres depuis la mémoire vers les registres du processeur puis à faire la copie dans le chemin inverse pour le résultat. Les instructions SIMD font gagner du temps du niveau du calcul. Au lieu de faire 4 opérations de multiplication terme à terme, il n'en fait plus qu'une. Il suffit de savoir comment utiliser ces instructions. Avec Visual Studio, elles sont accessible via ces fonctions Memory and Initialization Using Streaming SIMD Extensions. Le code suivant n'est probablement pas optimal mais il n'est pas trop compliqué à suivre.

[36]:
```
code = """
#include <xmmintrin.h>

static int linreg_custom_float_simd(float * x, float * out)
{
    __m128 c1 = _mm_set_ps(0.3034995490664121f, -237.63931533353392f, 510.
↪5306054362245f, 327.7369804093466f);
    __m128 c2 = _mm_set_ps(-814.1317093725389f, 492.81458798373245f, 102.
↪84845219168025f, 184.60648905984064f);
    __m128 r1 = _mm_set_ss(152.76430691633442f);
    r1 = _mm_add_ss(r1, _mm_mul_ps(c1, _mm_load_ps(x)));
    r1 = _mm_add_ss(r1, _mm_mul_ps(c2, _mm_load_ps(x+4)));
    float r[4];
    _mm_store_ps(r, r1);
    out[0] = r[0] + r[1] + r[2] + r[3] + 743.5196167505419f * x[8] + 76.095172216624f␣
↪* x[9];
    return 1;
}
"""
```

[37]:
```
from cffi import FFI
ffibuilder = FFI()

ffibuilder.cdef("int linreg_custom_float_simd(float *, float *);")
ffibuilder.set_source("_linear_regression_custom_float_simd", code)
ffibuilder.compile(verbose=True)
```

```
generating .\_linear_regression_custom_float_simd.c
(already up-to-date)
the current directory is
```

```
'C:\\xavierdupre\\__home_\\GitHub\\ensae_teaching_cs\\_doc\\notebooks\\2a'
running build_ext
building '_linear_regression_custom_float_simd' extension
C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\bin\HostX86\x64\cl.exe /c
/nologo /Ox /W3 /GL /DNDEBUG /MD -Ic:\python372_x64\include
-Ic:\python372_x64\include "-IC:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\ATLMFC\include" "-IC:\Program
Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\include" "-IC:\Program Files
(x86)\Windows Kits\NETFXSDK\4.6.1\include\um" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\ucrt" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\shared" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\um" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\winrt" "-IC:\Program Files (x86)\Windows
Kits\10\include\10.0.17763.0\cppwinrt" /Tc_linear_regression_custom_float_simd.c
/Fo.\Release\_linear_regression_custom_float_simd.obj
C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\bin\HostX86\x64\link.exe /nologo
/INCREMENTAL:NO /LTCG /DLL /MANIFEST:EMBED,ID=2 /MANIFESTUAC:NO
/LIBPATH:c:\python372_x64\libs /LIBPATH:c:\python372_x64\PCbuild\amd64
"/LIBPATH:C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\ATLMFC\lib\x64"
"/LIBPATH:C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\VC\Tools\MSVC\14.16.27023\lib\x64" "/LIBPATH:C:\Program
Files (x86)\Windows Kits\NETFXSDK\4.6.1\lib\um\x64" "/LIBPATH:C:\Program Files
(x86)\Windows Kits\10\lib\10.0.17763.0\ucrt\x64" "/LIBPATH:C:\Program Files
(x86)\Windows Kits\10\lib\10.0.17763.0\um\x64"
/EXPORT:PyInit__linear_regression_custom_float_simd
.\Release\_linear_regression_custom_float_simd.obj
/OUT:.\_linear_regression_custom_float_simd.cp37-win_amd64.pyd
/IMPLIB:.\Release\_linear_regression_custom_float_simd.cp37-win_amd64.lib
```

[37]:
```
'C:\\xavierdupre\\__home_\\GitHub\\ensae_teaching_cs\\_doc\\notebooks\\2a\\_line
ar_regression_custom_float_simd.cp37-win_amd64.pyd'
```

[38]:
```
from _linear_regression_custom_float_simd.lib import linreg_custom_float_simd
```

[39]:
```
out = numpy.zeros(1, dtype=numpy.float32)
ptr_x = x32.__array_interface__['data'][0]
cptr_x = ffi.cast ( "float*" , ptr_x )
ptr_out = out.__array_interface__['data'][0]
cptr_out = ffi.cast ( "float*" , ptr_out )

linreg_custom_float_simd(cptr_x, cptr_out)
out
```

[39]:
```
array([171.1178], dtype=float32)
```

[40]:
```
memo_time.append(timeexe("cffi-linreg-custom-float32-simd",␣
↪"linreg_custom_float_simd(cptr_x, cptr_out)"))
```

```
Moyenne: 497.72 ns Ecart-type 294.62 ns (with 100 runs) in [343.00 ns, 730.00
ns]
```

C'est légèrement mieux, quelques références :

- aligned_vs_unaligned_load.c : c'est du code mais facile à lire.
- How to Write Fast Numerical Code

Les processeurs évoluent au fil du temps, 4 float, 8 float, SIMD2, FMA4 Intrinsics Added for Visual Studio 2010 SP1, AVX.

1.1.5 Réécriture purement Python

On continue avec uniquement du Python sans numpy.

[41]:
```
coef = clr.coef_
list(coef)
```

[41]:
```
[0.3034995490657413,
 -237.6393153335348,
 510.53060543622456,
 327.73698040934715,
 -814.1317093725389,
 492.8145879837313,
 102.84845219167991,
 184.60648905984044,
 743.5196167505418,
 76.09517221662408]
```

[42]:
```
code = str(clr.intercept_) + "+" + "+".join("x[{0}]*({1})".format(i, c) for i, c in␣
↪enumerate(coef))
code
```

[42]:
```
'152.76430691633442+x[0]*(0.3034995490657413)+x[1]*(-237.6393153335348)+x[2]*(51
0.53060543622456)+x[3]*(327.73698040934715)+x[4]*(-814.1317093725389)+x[5]*(492.
8145879837313)+x[6]*(102.84845219167991)+x[7]*(184.60648905984044)+x[8]*(743.519
6167505418)+x[9]*(76.09517221662408)'
```

[43]:
```
def predict_clr_python(x):
    return 152.764306916+x[0]*0.3034995490664121+x[1]*(-237.63931533353392)+x[2]*510.
↪5306054362245+ \
           x[3]*327.7369804093466+ \
           x[4]*(-814.1317093725389)+x[5]*492.81458798373245+x[6]*102.84845219168025+␣
↪\
           x[7]*184.60648905984064+x[8]*743.5196167505419+x[9]*76.095172216624

predict_clr_python(x[0])
```

[43]:
```
197.61846907469848
```

[44]:
```
z = list(x[0])
memo_time.append(timeexe("python-linreg-custom", "predict_clr_python(z)"))
```

```
Moyenne: 4.11 µs Ecart-type 1.71 µs (with 100 runs) in [2.81 µs, 7.48 µs]
```

De façon assez surprenante, c'est plutôt rapide. Et si on y mettait une boucle.

[45]:
```
def predict_clr_python_loop(x, coef, intercept):
    return intercept + sum(a*b for a, b in zip(x, coef))

predict_clr_python_loop(x[0], list(clr.coef_), clr.intercept_)
```

[45]:
```
197.61846907503283
```

[46]:
```
coef = list(clr.coef_)
intercept = clr.intercept_
memo_time.append(timeexe("python-linreg", "predict_clr_python_loop(z, coef,␣
 ↪intercept)"))
```

```
Moyenne: 6.17 µs Ecart-type 2.48 µs (with 100 runs) in [4.17 µs, 10.51 µs]
```

A peine plus long.

1.1.6 Réécriture avec Python et numpy

[47]:
```
def predict_clr_numpy(x, coef, intercept):
    return intercept + numpy.dot(coef, x).sum()

predict_clr_numpy(x[0], clr.coef_, clr.intercept_)
```

[47]:
```
197.61846907503283
```

[48]:
```
memo_time.append(timeexe("numpy-linreg-numpy", "predict_clr_numpy(z, coef, clr.
 ↪intercept_)"))
```

```
Moyenne: 10.80 µs Ecart-type 3.70 µs (with 100 runs) in [8.25 µs, 18.65 µs]
```

Les dimensions des tableaux sont trop petites pour que le calcul matriciel apporte une différence. On se retrouve dans le cas *cffi* où les échanges Python - C grignotent tout le temps de calcul.

1.1.7 numba

numba essaye de compiler à la volée des bouts de codes écrits en Python. On induque quelle fonction optimiser en faisant précéder la fonction de `@jit`. Toutes les écritures ne fonctionnent, typiquement, certaines listes en compréhension soulèvent une exception. Il faut donc écrire son code en Python d'une façon assez proche de ce qu'il serait en C.

[49]:
```
from numba import jit
```

[50]:
```
@jit
def predict_clr_numba(x, coef, intercept):
    s = intercept
    for i in range(0, len(x)):
        s += coef[i] * x[i]
    return s

predict_clr_numba(z, clr.coef_, clr.intercept_)
```

[50]:
```
197.6184690750328
```

[51]:
```
memo_time.append(timeexe("numba-linreg-notype", "predict_clr_numba(z, clr.coef_, clr.
↪intercept_)"))
```

```
Moyenne: 29.82 µs Ecart-type 10.14 µs (with 100 runs) in [23.41 µs, 48.61 µs]
```

Plutôt rapide !

[52]:
```
@jit('double(double[:], double[:], double)')
def predict_clr_numba_cast(x, coef, intercept):
    s = intercept
    for i in range(0, len(x)):
        s += coef[i] * x[i]
    return s

# La fonction ne fonctionne qu'avec un numpy.array car le langage C est fortement typé.
predict_clr_numba_cast(x[0], clr.coef_, clr.intercept_)
```

[52]:
```
197.6184690750328
```

[53]:
```
memo_time.append(timeexe("numba-linreg-type", "predict_clr_numba_cast(x[0], clr.coef_,␣
↪clr.intercept_)"))
```

```
Moyenne: 1.52 µs Ecart-type 686.75 ns (with 100 runs) in [917.00 ns, 2.67 µs]
```

On voit que plus on donne d'information au compilateur, plus il est capable d'optimiser.

[54]:
```
@jit('float32(float32[:], float32[:], float32)')
def predict_clr_numba_cast_float(x, coef, intercept):
    s = intercept
    for i in range(0, len(x)):
        s += coef[i] * x[i]
    return s

# La fonction ne fonctionne qu'avec un numpy.array car le langage C est fortement typé.
x32 = x[0].astype(numpy.float32)
c32 = clr.coef_.astype(numpy.float32)
i32 = numpy.float32(clr.intercept_)
predict_clr_numba_cast_float(x32, c32, i32)
```

[54]:
```
197.61846923828125
```

[55]:
```
memo_time.append(timeexe("numba-linreg-type-float32",␣
↪"predict_clr_numba_cast_float(x32, c32, i32)"))
```

```
Moyenne: 823.78 ns Ecart-type 468.02 ns (with 100 runs) in [604.00 ns, 1.46 µs]
```

On essaye avec les coefficients dans la fonction.

[56]:
```
@jit('double(double[:])')
def predict_clr_numba_cast_custom(x):
    coef = [ 3.03499549e-01, -2.37639315e+02,  5.10530605e+02,  3.27736980e+02,
            -8.14131709e+02,  4.92814588e+02,  1.02848452e+02,  1.84606489e+02,
             7.43519617e+02,  7.60951722e+01]
    s = 152.76430691633442
```

```
    for i in range(0, len(x)):
        s += coef[i] * x[i]
    return s

predict_clr_numba_cast_custom(x[0])
```

[56]: 197.61846907190048

[57]:
```
memo_time.append(timeexe("numba-linreg-type-custom",␣
↪"predict_clr_numba_cast_custom(x[0])"))
```

```
Moyenne: 1.10 µs Ecart-type 515.34 ns (with 100 runs) in [777.00 ns, 1.84 µs]
```

On se rapproche des temps obtenus avec *cffi* sans *wrapping*, cela signifie que *numba* fait un bien meilleur travail à ce niveau que le wrapper rapidement créé.

[58]:
```
@jit('double(double[:], double[:], double)')
def predict_clr_numba_numpy(x, coef, intercept):
    return intercept + numpy.dot(coef, x).sum()

predict_clr_numba_numpy(x[0], clr.coef_, clr.intercept_)
```

[58]: 197.61846907503283

[59]:
```
memo_time.append(timeexe("numba-linreg-type-numpy", "predict_clr_numba_numpy(x[0], clr.
↪coef_, clr.intercept_)"))
```

```
Moyenne: 8.26 µs Ecart-type 2.50 µs (with 100 runs) in [6.90 µs, 12.80 µs]
```

numba est moins performant quand *numpy* est impliqué car le code de numpy n'est pas réécrit, il est appelé.

1.1.8 cython

cython permet de créer des extensions C de plus grande envergure que *numba*. C'est l'option choisie par scikit-learn. Il vaut mieux connaître le C pour s'en servir et là encore, l'objectif est de réduire les échanges Python / C qui coûtent cher.

[60]:
```
%load_ext cython
```

[61]:
```
%%cython
def predict_clr_cython(x, coef, intercept):
    s = intercept
    for i in range(0, len(x)):
        s += coef[i] * x[i]
    return s
```

[62]:
```
predict_clr_cython(x[0], clr.coef_, clr.intercept_)
```

[62]: 197.6184690750328

[63]:
```
memo_time.append(timeexe("cython-linreg", "predict_clr_cython(x[0], clr.coef_, clr.
↪intercept_)"))
```

```
Moyenne: 4.23 µs Ecart-type 981.04 ns (with 100 runs) in [3.57 µs, 6.49 µs]
```

Cython fait moins bien que *numba* dans notre cas et l'optimisation proposée est assez proche du temps déjà obtenue avec le langage Python seul. Cela est dû au fait que la plupart des objets tels que du code associé aux listes ou aux dictionnaires ont été réécrits en C.

[64]:
```
%%cython
cimport numpy as npc

def predict_clr_cython_type(npc.ndarray[double, ndim=1, mode='c'] x,
                            npc.ndarray[double, ndim=1, mode='c'] coef,
                            double intercept):
    cdef double s = intercept
    for i in range(0, x.shape[0]):
        s += coef[i] * x[i]
    return s
```

[65]:
```
predict_clr_cython_type(x[0], clr.coef_, clr.intercept_)
```

[65]:
```
197.6184690750328
```

[66]:
```
memo_time.append(timeexe(
    "cython-linreg-type", "predict_clr_cython_type(x[0], clr.coef_, clr.intercept_)"))
```

```
Moyenne: 2.10 µs Ecart-type 1.07 µs (with 100 runs) in [1.48 µs, 3.57 µs]
```

Le temps est quasi identique avec un écart type moins grand de façon significative.

1.1.9 Une dernière option : ONNX

ONNX est un format de sérialisation qui permet de décrire un modèle de modèle de machine learning ou de deep learning. Cela permet de dissocer le modèle de la librairie qui a servi à le produire (voir ML.net and ONNX).

[67]:
```
try:
    from skl2onnx import convert_sklearn
    from skl2onnx.common.data_types import FloatTensorType
    import onnxruntime
    import onnx
    ok_onnx = True
    print("onnx, skl2onnx, onnxruntime sont disponibles.")

    def save_model(onnx_model, filename):
        with open(filename, "wb") as f:
            f.write(onnx_model.SerializeToString())
except ImportError as e:
    print("La suite requiert onnx, skl2onnx et onnxruntime.")
    print(e)
    ok_onnx = False
```

```
onnx, skl2onnx, onnxruntime sont disponibles.
```

On convertit le modèle au format ONNX.

[68]:
```
if ok_onnx:
    onnx_model = convert_sklearn(
        clr, 'model', [('input', FloatTensorType([None, clr.coef_.shape[0]]))],
```

```
        target_opset=11)
    onnx_model.ir_version = 6
    save_model(onnx_model, 'model.onnx')

    model_onnx = onnx.load('model.onnx')
    print("Modèle sérialisé au format ONNX")
    print(model_onnx)
else:
    print("onnx, onnxmltools, onnxruntime sont disponibles.")
```

```
Modèle sérialisé au format ONNX
ir_version: 6
producer_name: "skl2onnx"
producer_version: "1.6.995"
domain: "ai.onnx"
model_version: 0
doc_string: ""
graph {
  node {
    input: "input"
    output: "variable"
    name: "LinearRegressor"
    op_type: "LinearRegressor"
    attribute {
      name: "coefficients"
      floats: 0.3034995496273041
      floats: -237.63931274414062
      floats: 510.5306091308594
      floats: 327.7369689941406
      floats: -814.1317138671875
      floats: 492.8145751953125
      floats: 102.84844970703125
      floats: 184.6064910888672
      floats: 743.5195922851562
      floats: 76.09516906738281
      type: FLOATS
    }
    attribute {
      name: "intercepts"
      floats: 152.76431274414062
      type: FLOATS
    }
    domain: "ai.onnx.ml"
  }
  name: "model"
  input {
    name: "input"
    type {
      tensor_type {
        elem_type: 1
        shape {
          dim {
          }
          dim {
```

```
            dim_value: 10
          }
        }
      }
    }
  }
  output {
    name: "variable"
    type {
      tensor_type {
        elem_type: 1
        shape {
          dim {
          }
          dim {
            dim_value: 1
          }
        }
      }
    }
  }
}
opset_import {
  domain: "ai.onnx.ml"
  version: 1
}
```

On calcule les prédictions. Le module {onnxruntime](https://docs.microsoft.com/en-us/python/api/overview/azure/onnx/intro?view=azure-onnx-py) optimise les calculs pour des modèles de deep learning. Cela explique pourquoi tous les calculs sont réalisés avec des réels représentés sur 4 octets numpy.float32.

[69]:
```
if ok_onnx:
    sess = onnxruntime.InferenceSession("model.onnx")
    for i in sess.get_inputs():
        print('Input:', i)
    for o in sess.get_outputs():
        print('Output:', o)

    def predict_onnxrt(x):
        return sess.run(["variable"], {'input': x})

    print("Prediction:", predict_onnxrt(x.astype(numpy.float32)))
```

```
Input: NodeArg(name='input', type='tensor(float)', shape=[None, 10])
Output: NodeArg(name='variable', type='tensor(float)', shape=[None, 1])
Prediction: [array([[197.61847]], dtype=float32)]
```

[70]:
```
if ok_onnx:
    x32 = x.astype(numpy.float32)
    memo_time.append(timeexe("onnxruntime-float32", "predict_onnxrt(x32)"))
    memo_time.append(timeexe("onnxruntime-float64", "predict_onnxrt(x.astype(numpy.
 ↪float32))"))
```

```
Moyenne: 11.59 µs Ecart-type 2.44 µs (with 100 runs) in [10.08 µs, 15.82 µs]
Moyenne: 14.40 µs Ecart-type 3.73 µs (with 100 runs) in [11.83 µs, 22.19 µs]
```

1.1.10 Récapitulatif

[71]:
```
import pandas
df = pandas.DataFrame(data=memo_time)
df = df.set_index("legend").sort_values("average")
df
```

[71]:

legend	average	deviation	first	first3	last3	repeat
cffi-linreg-custom-float32	4.217760e-07	2.375564e-07	0.000006	0.000002	3.546667e-07	1000
cffi-linreg-custom-float32-simd	4.977160e-07	2.946159e-07	0.000006	0.000002	5.716667e-07	1000
cffi-linreg-custom	5.048080e-07	2.821699e-07	0.000002	0.000001	5.383333e-07	1000
cffi-linreg	6.916940e-07	3.012178e-07	0.000003	0.000002	9.816667e-07	1000
numba-linreg-type-float32	8.237830e-07	4.680243e-07	0.000002	0.000002	8.666667e-07	1000
numba-linreg-type-custom	1.102100e-06	5.153437e-07	0.000002	0.000002	8.503333e-07	1000
numba-linreg-type	1.516927e-06	6.867544e-07	0.000003	0.000002	9.143333e-07	1000
cython-linreg-type	2.096415e-06	1.066262e-06	0.000002	0.000002	2.092333e-06	1000
python-linreg-custom	4.106685e-06	1.707472e-06	0.000007	0.000007	2.968333e-06	1000
cython-linreg	4.233643e-06	9.810450e-07	0.000008	0.000007	3.822333e-06	1000
python-linreg	6.167101e-06	2.476066e-06	0.000007	0.000007	5.894667e-06	1000
cffi-linreg-custom-float wrapped	6.212464e-06	2.343711e-06	0.000020	0.000020	6.376667e-06	1000
cffi-linreg-custom wrapped	6.790410e-06	2.825853e-06	0.000017	0.000015	5.495333e-06	1000
numba-linreg-type-numpy	8.255435e-06	2.502778e-06	0.000036	0.000030	7.373333e-06	1000
numpy-linreg-numpy	1.079853e-05	3.698106e-06	0.000029	0.000017	1.400800e-05	1000
onnxruntime-float32	1.159433e-05	2.439559e-06	0.000038	0.000026	1.056167e-05	1000
cffi-linreg-wrapped	1.264076e-05	5.782540e-06	0.000015	0.000016	7.154000e-06	1000
onnxruntime-float64	1.440223e-05	3.733025e-06	0.000016	0.000014	1.388100e-05	1000
numba-linreg-notype	2.982464e-05	1.014061e-05	0.000076	0.000080	2.592767e-05	1000
sklearn.predict	5.066311e-05	1.318277e-05	0.000079			

```
sklearn.predict                    0.000082  4.157900e-05    1000

                                          min5          max5  \
legend
cffi-linreg-custom-float32        3.540000e-07  6.760000e-07
cffi-linreg-custom-float32-simd   3.430000e-07  7.300000e-07
cffi-linreg-custom                3.520000e-07  8.360000e-07
cffi-linreg                       4.700000e-07  1.091000e-06
numba-linreg-type-float32         6.040000e-07  1.456000e-06
numba-linreg-type-custom          7.770000e-07  1.844000e-06
numba-linreg-type                 9.170000e-07  2.669000e-06
cython-linreg-type                1.482000e-06  3.568000e-06
python-linreg-custom              2.808000e-06  7.478000e-06
cython-linreg                     3.573000e-06  6.488000e-06
python-linreg                     4.170000e-06  1.051500e-05
cffi-linreg-custom-float wrapped  5.144000e-06  1.011800e-05
cffi-linreg-custom wrapped        4.985000e-06  1.287000e-05
numba-linreg-type-numpy           6.902000e-06  1.280300e-05
numpy-linreg-numpy                8.247000e-06  1.865300e-05
onnxruntime-float32               1.008100e-05  1.582400e-05
cffi-linreg-wrapped               7.149000e-06  2.251500e-05
onnxruntime-float64               1.183500e-05  2.218600e-05
numba-linreg-notype               2.340700e-05  4.861400e-05
sklearn.predict                   4.005600e-05  7.588400e-05

code  \
legend
cffi-linreg-custom-float32                  linreg_custom_float(cptr_x,
cptr_out)
cffi-linreg-custom-float32-simd         linreg_custom_float_simd(cptr_x,
cptr_out)
cffi-linreg-custom                              linreg_custom(cptr_x,
cptr_out)
cffi-linreg                       lib.linreg(n, cptr_x, cptr_coef,
clr.intercept…
numba-linreg-type-float32               predict_clr_numba_cast_float(x32, c32,
i32)
numba-linreg-type-custom
predict_clr_numba_cast_custom(x[0])
numba-linreg-type                 predict_clr_numba_cast(x[0], clr.coef_,
clr.in…
cython-linreg-type                predict_clr_cython_type(x[0], clr.coef_,
clr.i…
python-linreg-custom
predict_clr_python(z)
cython-linreg                     predict_clr_cython(x[0], clr.coef_,
clr.interc…
python-linreg                           predict_clr_python_loop(z, coef,
intercept)
cffi-linreg-custom-float wrapped
predict_clr_custom(x32)
cffi-linreg-custom wrapped
predict_clr_custom(x)
```

```
numba-linreg-type-numpy           predict_clr_numba_numpy(x[0], clr.coef_,
clr.i…
numpy-linreg-numpy                       predict_clr_numpy(z, coef,
clr.intercept_)
onnxruntime-float32
predict_onnxrt(x32)
cffi-linreg-wrapped                                              predict_clr(x,
clr)
onnxruntime-float64
predict_onnxrt(x.astype(numpy.float32))
numba-linreg-notype                 predict_clr_numba(z, clr.coef_,
clr.intercept_)
sklearn.predict
clr.predict(z)
```

legend	run
cffi-linreg-custom-float32	100
cffi-linreg-custom-float32-simd	100
cffi-linreg-custom	100
cffi-linreg	100
numba-linreg-type-float32	100
numba-linreg-type-custom	100
numba-linreg-type	100
cython-linreg-type	100
python-linreg-custom	100
cython-linreg	100
python-linreg	100
cffi-linreg-custom-float wrapped	100
cffi-linreg-custom wrapped	100
numba-linreg-type-numpy	100
numpy-linreg-numpy	100
onnxruntime-float32	100
cffi-linreg-wrapped	100
onnxruntime-float64	100
numba-linreg-notype	100
sklearn.predict	100

On enlève quelques colonnes et on rappelle :

- **cffi**: signifie optimisé avec cffi
- **custom**: pas de boucle mais la fonction ne peut prédire qu'une seule régression linéaire
- **float32**: utilise des float et non des double
- **linreg**: régression linéaire
- **numba**: optimisation avec numba
- **numpy**: optimisation avec numpy
- **python**: pas de C, que du python
- **simd**: optimisé avec les instructions SIMD
- **sklearn**: fonction sklearn.predict
- **static**: la fonction utilise des variables statiques
- **type**: la fonction est typée et ne fonctionne qu'avec un type précis en entrée.
- **wrapped**: code optimisé mais embabllé dans une fonction Python qui elle ne l'est pas (les containers sont recréés à chaque fois)

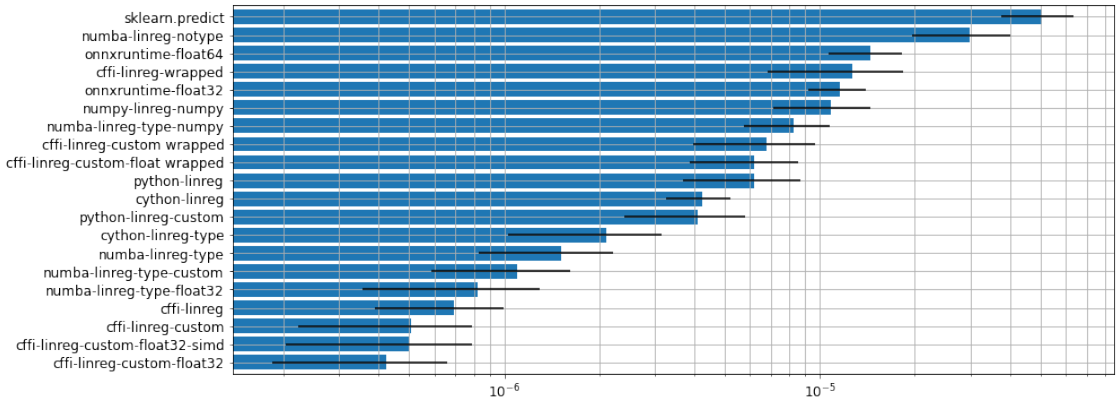
[72]:
```
cols = ["average", "deviation", "min5", "max5", "run", "code"]
df[cols]
```

[72]:

legend	average	deviation	min5	max5	run	code
cffi-linreg-custom-float32	4.217760e-07	2.375564e-07	3.540000e-07	6.760000e-07	100	linreg_custom_float(cptr_x, cptr_out)
cffi-linreg-custom-float32-simd	4.977160e-07	2.946159e-07	3.430000e-07	7.300000e-07	100	
cffi-linreg-custom	5.048080e-07	2.821699e-07	3.520000e-07	8.360000e-07	100	
cffi-linreg	6.916940e-07	3.012178e-07	4.700000e-07	1.091000e-06	100	
numba-linreg-type-float32	8.237830e-07	4.680243e-07	6.040000e-07	1.456000e-06	100	
numba-linreg-type-custom	1.102100e-06	5.153437e-07	7.770000e-07	1.844000e-06	100	
numba-linreg-type	1.516927e-06	6.867544e-07	9.170000e-07	2.669000e-06	100	
cython-linreg-type	2.096415e-06	1.066262e-06	1.482000e-06	3.568000e-06	100	
python-linreg-custom	4.106685e-06	1.707472e-06	2.808000e-06	7.478000e-06	100	
cython-linreg	4.233643e-06	9.810450e-07	3.573000e-06	6.488000e-06	100	
python-linreg	6.167101e-06	2.476066e-06	4.170000e-06	1.051500e-05	100	
cffi-linreg-custom-float wrapped	6.212464e-06	2.343711e-06	5.144000e-06	1.011800e-05	100	
cffi-linreg-custom wrapped	6.790410e-06	2.825853e-06	4.985000e-06	1.287000e-05	100	
numba-linreg-type-numpy	8.255435e-06	2.502778e-06	6.902000e-06	1.280300e-05	100	
numpy-linreg-numpy	1.079853e-05	3.698106e-06	8.247000e-06	1.865300e-05	100	
onnxruntime-float32	1.159433e-05	2.439559e-06	1.008100e-05	1.582400e-05	100	
cffi-linreg-wrapped	1.264076e-05	5.782540e-06	7.149000e-06	2.251500e-05	100	
onnxruntime-float64	1.440223e-05	3.733025e-06	1.183500e-05	2.218600e-05	100	
numba-linreg-notype	2.982464e-05	1.014061e-05	2.340700e-05	4.861400e-05	100	
sklearn.predict	5.066311e-05	1.318277e-05	4.005600e-05	7.588400e-05	100	

```
cffi-linreg-custom-float32-simd          linreg_custom_float_simd(cptr_x,
cptr_out)
cffi-linreg-custom                                  linreg_custom(cptr_x,
cptr_out)
cffi-linreg                         lib.linreg(n, cptr_x, cptr_coef,
clr.intercept…
numba-linreg-type-float32                predict_clr_numba_cast_float(x32, c32,
i32)
numba-linreg-type-custom
predict_clr_numba_cast_custom(x[0])
numba-linreg-type                   predict_clr_numba_cast(x[0], clr.coef_,
clr.in…
cython-linreg-type                  predict_clr_cython_type(x[0], clr.coef_,
clr.i…
python-linreg-custom
predict_clr_python(z)
cython-linreg                       predict_clr_cython(x[0], clr.coef_,
clr.interc…
python-linreg                            predict_clr_python_loop(z, coef,
intercept)
cffi-linreg-custom-float wrapped
predict_clr_custom(x32)
cffi-linreg-custom wrapped
predict_clr_custom(x)
numba-linreg-type-numpy             predict_clr_numba_numpy(x[0], clr.coef_,
clr.i…
numpy-linreg-numpy                        predict_clr_numpy(z, coef,
clr.intercept_)
onnxruntime-float32
predict_onnxrt(x32)
cffi-linreg-wrapped                                              predict_clr(x,
clr)
onnxruntime-float64
predict_onnxrt(x.astype(numpy.float32))
numba-linreg-notype                   predict_clr_numba(z, clr.coef_,
clr.intercept_)
sklearn.predict
clr.predict(z)
```

[73]:
```
%matplotlib inline
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 1, figsize=(14,6))
df[["average", "deviation"]].plot(kind="barh", logx=True, ax=ax, xerr="deviation",
                                  legend=False, fontsize=12, width=0.8)
ax.set_ylabel("")
ax.grid(b=True, which="major")
ax.grid(b=True, which="minor");
```



Il manque à ce comparatif le GPU mais c'est un peu plus complexe à mettre en oeuvre, il faut une carte GPU et la parallélisation n'apporterait pas énormément compte tenu de la faible dimension du problème.

1.1.11 Prédiction one-off et biais de mesure

Le graphique précédent montre que la fonction `predict` de *scikit-learn* est la plus lente. La première raison est que ce code est valable pour toutes les régresssions linéaires alors que toutes les autres fonctions sont spécialisées pour un seul modèle. La seconde raison est que le code de *scikit-learn* est optimisé pour le calcul de plusieurs prédictions à la fois alors que toutes les autres fonctions n'en calcule qu'une seule (scénario dit *one-off*). On compare à ce que donnerait unev version purement python et numpy.

[74]:
```
def predict_clr_python_loop_multi(x, coef, intercept):
    # On s'attend à deux dimension.
    res = numpy.zeros((x.shape[0], 1))
    res[:, 0] = intercept
    for i in range(0, x.shape[0]):
        res[i, 0] += sum(a*b for a, b in zip(x[i, :], coef))
    return res

predict_clr_python_loop_multi(diabetes_X_test[:2], clr.coef_, clr.intercept_)
```

[74]:
```
array([[197.61846908],
       [155.43979328]])
```

[75]:
```
def predict_clr_numpy_loop_multi(x, coef, intercept):
    # On s'attend à deux dimension.
    res = numpy.ones((x.shape[0], 1)) * intercept
    res += x @ coef.reshape((len(coef), 1))
    return res

predict_clr_numpy_loop_multi(diabetes_X_test[:2], clr.coef_, clr.intercept_)
```

[75]:
```
array([[197.61846908],
       [155.43979328]])
```

[76]:
```
def predict_clr_numba_cast_multi(X, coef, intercept):
    return [predict_clr_numba_cast(x, coef, intercept) for x in X]

predict_clr_numba_cast_multi(diabetes_X_test[:2], clr.coef_, clr.intercept_)
```

[76]: [197.6184690750328, 155.43979327521237]

[77]:
```
def predict_clr_cython_type_multi(X, coef, intercept):
    return [predict_clr_cython_type(x, coef, intercept) for x in X]

predict_clr_cython_type_multi(diabetes_X_test[:2], clr.coef_, clr.intercept_)
```

[77]: [197.6184690750328, 155.43979327521237]

[78]:
```
memo = []
batch = [1, 10, 100, 200, 500, 1000, 2000, 3000, 4000, 5000, 10000,
         20000, 50000, 75000, 100000, 150000, 200000, 300000, 400000,
         500000, 600000]
number = 10
for i in batch:
    if i <= diabetes_X_test.shape[0]:
        mx = diabetes_X_test[:i]
    else:
        mxs = [diabetes_X_test] * (i // diabetes_X_test.shape[0] + 1)
        mx = numpy.vstack(mxs)
        mx = mx[:i]

    print("batch", "=", i)
    repeat=20 if i >= 5000 else 100

    memo.append(timeexe("sklearn.predict %d" % i, "clr.predict(mx)",
                        repeat=repeat, number=number))
    memo[-1]["batch"] = i
    memo[-1]["lib"] = "sklearn"

    if i <= 1000:
        # très lent
        memo.append(timeexe("python %d" % i, "predict_clr_python_loop_multi(mx, clr.
↪coef_, clr.intercept_)",
                            repeat=20, number=number))
        memo[-1]["batch"] = i
        memo[-1]["lib"] = "python"

    memo.append(timeexe("numpy %d" % i, "predict_clr_numpy_loop_multi(mx, clr.coef_,␣
↪clr.intercept_)",
                        repeat=repeat, number=number))
    memo[-1]["batch"] = i
    memo[-1]["lib"] = "numpy"

    if i <= 10000:
        # très lent
        memo.append(timeexe("numba %d" % i, "predict_clr_numba_cast_multi(mx, clr.
↪coef_, clr.intercept_)",
                            repeat=repeat, number=number))
        memo[-1]["batch"] = i
        memo[-1]["lib"] = "numba"

    if i <= 1000:
```

```
        # très lent
        memo.append(timeexe("cython %d" % i, "predict_clr_cython_type_multi(mx, clr.
↪coef_, clr.intercept_)",
                            repeat=repeat, number=number))
        memo[-1]["batch"] = i
        memo[-1]["lib"] = "cython"

    if ok_onnx:
        memo.append(timeexe("onnxruntime %d" % i, "predict_onnxrt(mx.astype(numpy.
↪float32))",
                            repeat=repeat, number=number))
        memo[-1]["batch"] = i
        memo[-1]["lib"] = "onnxruntime"
```

```
batch = 1
Moyenne: 72.00 µs Ecart-type 44.02 µs (with 10 runs) in [40.40 µs, 115.59 µs]
Moyenne: 12.45 µs Ecart-type 4.62 µs (with 10 runs) in [9.59 µs, 31.13 µs]
Moyenne: 10.55 µs Ecart-type 4.06 µs (with 10 runs) in [8.49 µs, 18.70 µs]
Moyenne: 3.00 µs Ecart-type 1.25 µs (with 10 runs) in [2.24 µs, 5.39 µs]
Moyenne: 4.67 µs Ecart-type 2.13 µs (with 10 runs) in [2.89 µs, 8.40 µs]
Moyenne: 16.54 µs Ecart-type 9.30 µs (with 10 runs) in [12.06 µs, 38.25 µs]
batch = 10
Moyenne: 47.73 µs Ecart-type 9.68 µs (with 10 runs) in [40.71 µs, 64.48 µs]
Moyenne: 83.77 µs Ecart-type 11.59 µs (with 10 runs) in [78.60 µs, 131.76 µs]
Moyenne: 9.43 µs Ecart-type 2.55 µs (with 10 runs) in [8.30 µs, 16.81 µs]
Moyenne: 10.98 µs Ecart-type 5.03 µs (with 10 runs) in [9.17 µs, 18.89 µs]
Moyenne: 15.14 µs Ecart-type 2.63 µs (with 10 runs) in [14.11 µs, 19.38 µs]
Moyenne: 16.17 µs Ecart-type 11.43 µs (with 10 runs) in [12.20 µs, 27.21 µs]
batch = 100
Moyenne: 59.78 µs Ecart-type 24.40 µs (with 10 runs) in [42.49 µs, 121.46 µs]
Moyenne: 1.14 ms Ecart-type 349.02 µs (with 10 runs) in [812.90 µs, 1.97 ms]
Moyenne: 51.45 µs Ecart-type 223.13 µs (with 10 runs) in [11.86 µs, 101.18 µs]
Moyenne: 135.75 µs Ecart-type 62.65 µs (with 10 runs) in [77.71 µs, 228.08 µs]
Moyenne: 196.45 µs Ecart-type 60.61 µs (with 10 runs) in [136.24 µs, 294.37 µs]
Moyenne: 20.38 µs Ecart-type 8.50 µs (with 10 runs) in [14.05 µs, 42.62 µs]
batch = 200
Moyenne: 82.37 µs Ecart-type 37.51 µs (with 10 runs) in [43.70 µs, 151.49 µs]
Moyenne: 1.86 ms Ecart-type 274.17 µs (with 10 runs) in [1.64 ms, 2.54 ms]
Moyenne: 16.66 µs Ecart-type 7.57 µs (with 10 runs) in [9.82 µs, 33.68 µs]
Moyenne: 187.97 µs Ecart-type 50.61 µs (with 10 runs) in [151.22 µs, 335.14 µs]
Moyenne: 328.20 µs Ecart-type 83.76 µs (with 10 runs) in [268.49 µs, 535.13 µs]
Moyenne: 16.56 µs Ecart-type 5.97 µs (with 10 runs) in [14.77 µs, 26.23 µs]
batch = 500
Moyenne: 57.48 µs Ecart-type 12.61 µs (with 10 runs) in [45.16 µs, 90.60 µs]
Moyenne: 4.48 ms Ecart-type 438.20 µs (with 10 runs) in [4.21 ms, 6.03 ms]
Moyenne: 13.77 µs Ecart-type 3.81 µs (with 10 runs) in [11.01 µs, 21.57 µs]
Moyenne: 522.07 µs Ecart-type 142.02 µs (with 10 runs) in [394.04 µs, 896.03 µs]
Moyenne: 756.15 µs Ecart-type 111.59 µs (with 10 runs) in [687.45 µs, 996.65 µs]
Moyenne: 20.12 µs Ecart-type 7.01 µs (with 10 runs) in [18.00 µs, 23.64 µs]
batch = 1000
Moyenne: 68.58 µs Ecart-type 61.64 µs (with 10 runs) in [52.53 µs, 104.30 µs]
Moyenne: 10.01 ms Ecart-type 1.08 ms (with 10 runs) in [9.01 ms, 12.47 ms]
Moyenne: 19.63 µs Ecart-type 6.73 µs (with 10 runs) in [14.81 µs, 26.22 µs]
```

```
Moyenne: 1.02 ms Ecart-type 226.02 µs (with 10 runs) in [799.87 µs, 1.55 ms]
Moyenne: 1.49 ms Ecart-type 104.30 µs (with 10 runs) in [1.40 ms, 1.73 ms]
Moyenne: 31.15 µs Ecart-type 17.16 µs (with 10 runs) in [25.15 µs, 41.71 µs]
batch = 2000
Moyenne: 84.68 µs Ecart-type 19.03 µs (with 10 runs) in [60.01 µs, 114.15 µs]
Moyenne: 25.52 µs Ecart-type 7.76 µs (with 10 runs) in [17.64 µs, 36.54 µs]
Moyenne: 2.17 ms Ecart-type 674.09 µs (with 10 runs) in [1.64 ms, 3.73 ms]
Moyenne: 41.05 µs Ecart-type 9.31 µs (with 10 runs) in [33.48 µs, 57.88 µs]
batch = 3000
Moyenne: 87.48 µs Ecart-type 21.37 µs (with 10 runs) in [67.16 µs, 117.76 µs]
Moyenne: 32.81 µs Ecart-type 16.12 µs (with 10 runs) in [20.84 µs, 57.16 µs]
Moyenne: 3.02 ms Ecart-type 587.49 µs (with 10 runs) in [2.47 ms, 4.28 ms]
Moyenne: 50.26 µs Ecart-type 10.86 µs (with 10 runs) in [41.81 µs, 71.71 µs]
batch = 4000
Moyenne: 82.40 µs Ecart-type 18.50 µs (with 10 runs) in [72.03 µs, 120.87 µs]
Moyenne: 31.39 µs Ecart-type 7.08 µs (with 10 runs) in [23.94 µs, 48.14 µs]
Moyenne: 3.96 ms Ecart-type 732.60 µs (with 10 runs) in [3.29 ms, 5.64 ms]
Moyenne: 65.28 µs Ecart-type 16.14 µs (with 10 runs) in [49.98 µs, 95.03 µs]
batch = 5000
Moyenne: 147.92 µs Ecart-type 56.17 µs (with 10 runs) in [83.45 µs, 265.47 µs]
Moyenne: 46.21 µs Ecart-type 31.71 µs (with 10 runs) in [32.66 µs, 182.87 µs]
Moyenne: 5.31 ms Ecart-type 1.08 ms (with 10 runs) in [4.16 ms, 7.52 ms]
Moyenne: 87.70 µs Ecart-type 13.90 µs (with 10 runs) in [60.09 µs, 128.92 µs]
batch = 10000
Moyenne: 122.37 µs Ecart-type 20.32 µs (with 10 runs) in [110.79 µs, 192.69 µs]
Moyenne: 43.93 µs Ecart-type 9.07 µs (with 10 runs) in [38.27 µs, 69.81 µs]
Moyenne: 9.84 ms Ecart-type 1.88 ms (with 10 runs) in [8.25 ms, 14.84 ms]
Moyenne: 117.33 µs Ecart-type 26.94 µs (with 10 runs) in [97.26 µs, 175.96 µs]
batch = 20000
Moyenne: 217.04 µs Ecart-type 44.69 µs (with 10 runs) in [177.67 µs, 311.32 µs]
Moyenne: 89.48 µs Ecart-type 14.11 µs (with 10 runs) in [67.01 µs, 108.96 µs]
Moyenne: 268.69 µs Ecart-type 54.83 µs (with 10 runs) in [183.14 µs, 373.70 µs]
batch = 50000
Moyenne: 1.05 ms Ecart-type 183.52 µs (with 10 runs) in [836.30 µs, 1.45 ms]
Moyenne: 261.80 µs Ecart-type 38.00 µs (with 10 runs) in [210.45 µs, 311.47 µs]
Moyenne: 1.95 ms Ecart-type 158.53 µs (with 10 runs) in [1.70 ms, 2.33 ms]
batch = 75000
Moyenne: 1.59 ms Ecart-type 420.96 µs (with 10 runs) in [1.23 ms, 3.23 ms]
Moyenne: 675.15 µs Ecart-type 223.73 µs (with 10 runs) in [425.52 µs, 1.39 ms]
Moyenne: 3.19 ms Ecart-type 377.79 µs (with 10 runs) in [2.56 ms, 3.69 ms]
batch = 100000
Moyenne: 2.09 ms Ecart-type 228.97 µs (with 10 runs) in [1.78 ms, 2.54 ms]
Moyenne: 574.44 µs Ecart-type 54.02 µs (with 10 runs) in [477.36 µs, 646.93 µs]
Moyenne: 4.07 ms Ecart-type 529.66 µs (with 10 runs) in [3.45 ms, 5.23 ms]
batch = 150000
Moyenne: 4.25 ms Ecart-type 301.91 µs (with 10 runs) in [3.94 ms, 5.05 ms]
Moyenne: 3.02 ms Ecart-type 162.20 µs (with 10 runs) in [2.82 ms, 3.42 ms]
Moyenne: 5.73 ms Ecart-type 523.58 µs (with 10 runs) in [5.19 ms, 7.41 ms]
batch = 200000
Moyenne: 6.30 ms Ecart-type 1.02 ms (with 10 runs) in [5.21 ms, 8.84 ms]
Moyenne: 4.40 ms Ecart-type 689.32 µs (with 10 runs) in [3.76 ms, 5.71 ms]
Moyenne: 7.90 ms Ecart-type 1.04 ms (with 10 runs) in [6.84 ms, 10.64 ms]
batch = 300000
Moyenne: 8.74 ms Ecart-type 1.04 ms (with 10 runs) in [7.90 ms, 11.39 ms]
```

```
Moyenne: 6.05 ms Ecart-type 353.81 µs (with 10 runs) in [5.66 ms, 7.22 ms]
Moyenne: 11.43 ms Ecart-type 382.57 µs (with 10 runs) in [10.84 ms, 12.19 ms]
batch = 400000
Moyenne: 11.28 ms Ecart-type 502.22 µs (with 10 runs) in [10.75 ms, 12.68 ms]
Moyenne: 8.12 ms Ecart-type 370.35 µs (with 10 runs) in [7.74 ms, 8.94 ms]
Moyenne: 16.21 ms Ecart-type 1.55 ms (with 10 runs) in [15.15 ms, 22.05 ms]
batch = 500000
Moyenne: 14.79 ms Ecart-type 938.41 µs (with 10 runs) in [13.55 ms, 17.21 ms]
Moyenne: 10.33 ms Ecart-type 521.49 µs (with 10 runs) in [9.70 ms, 11.64 ms]
Moyenne: 20.23 ms Ecart-type 738.32 µs (with 10 runs) in [19.24 ms, 22.12 ms]
batch = 600000
Moyenne: 17.40 ms Ecart-type 1.04 ms (with 10 runs) in [16.05 ms, 20.28 ms]
Moyenne: 12.21 ms Ecart-type 460.13 µs (with 10 runs) in [11.67 ms, 13.48 ms]
Moyenne: 23.65 ms Ecart-type 701.39 µs (with 10 runs) in [22.83 ms, 25.71 ms]
```

[79]:
```
dfb = pandas.DataFrame(memo)[["average", "lib", "batch"]]
piv = dfb.pivot("batch", "lib", "average")
piv
```

[79]:

lib	cython	numba	numpy	onnxruntime	python	sklearn
batch						
1	0.000005	0.000003	0.000011	0.000017	0.000012	0.000072
10	0.000015	0.000011	0.000009	0.000016	0.000084	0.000048
100	0.000196	0.000136	0.000051	0.000020	0.001137	0.000060
200	0.000328	0.000188	0.000017	0.000017	0.001860	0.000082
500	0.000756	0.000522	0.000014	0.000020	0.004476	0.000057
1000	0.001489	0.001021	0.000020	0.000031	0.010010	0.000069
2000	NaN	0.002174	0.000026	0.000041	NaN	0.000085
3000	NaN	0.003015	0.000033	0.000050	NaN	0.000087
4000	NaN	0.003959	0.000031	0.000065	NaN	0.000082
5000	NaN	0.005314	0.000046	0.000088	NaN	0.000148
10000	NaN	0.009835	0.000044	0.000117	NaN	0.000122
20000	NaN	NaN	0.000089	0.000269	NaN	0.000217
50000	NaN	NaN	0.000262	0.001946	NaN	0.001047
75000	NaN	NaN	0.000675	0.003189	NaN	0.001589
100000	NaN	NaN	0.000574	0.004066	NaN	0.002088
150000	NaN	NaN	0.003025	0.005729	NaN	0.004253
200000	NaN	NaN	0.004402	0.007898	NaN	0.006301
300000	NaN	NaN	0.006052	0.011425	NaN	0.008742
400000	NaN	NaN	0.008120	0.016213	NaN	0.011279
500000	NaN	NaN	0.010332	0.020226	NaN	0.014790
600000	NaN	NaN	0.012205	0.023646	NaN	0.017401

[80]:
```
for c in piv.columns:
    piv["ave_" + c] = piv[c] / piv.index
piv
```

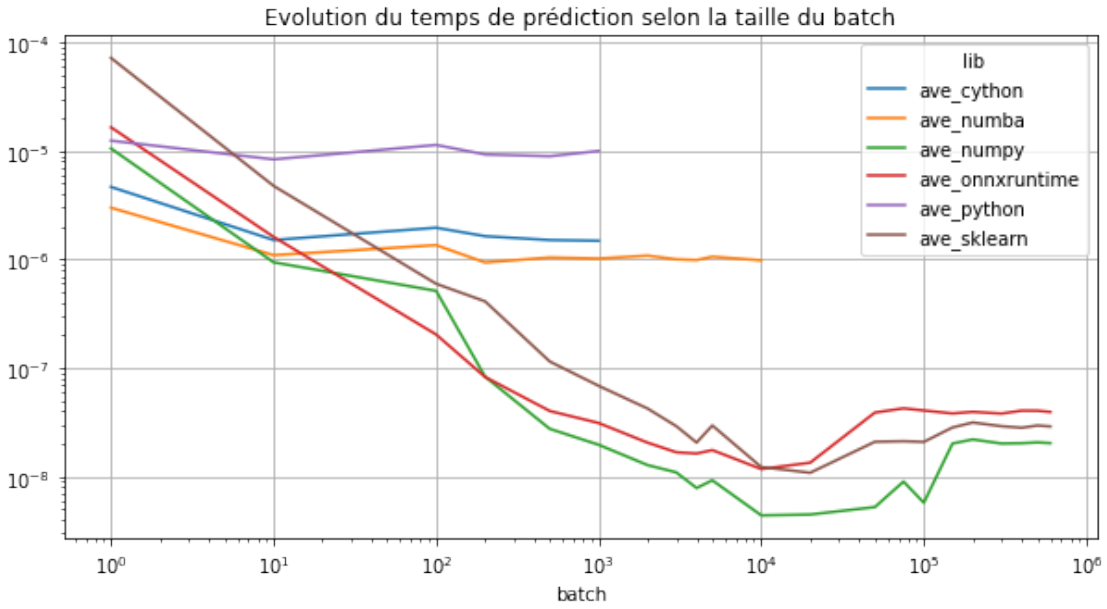
[80]:

lib	cython	numba	numpy	onnxruntime	python	sklearn	\
batch							
1	0.000005	0.000003	0.000011	0.000017	0.000012	0.000072	
10	0.000015	0.000011	0.000009	0.000016	0.000084	0.000048	
100	0.000196	0.000136	0.000051	0.000020	0.001137	0.000060	
200	0.000328	0.000188	0.000017	0.000017	0.001860	0.000082	
500	0.000756	0.000522	0.000014	0.000020	0.004476	0.000057	

```
1000    0.001489  0.001021  0.000020     0.000031  0.010010  0.000069
2000         NaN  0.002174  0.000026     0.000041       NaN  0.000085
3000         NaN  0.003015  0.000033     0.000050       NaN  0.000087
4000         NaN  0.003959  0.000031     0.000065       NaN  0.000082
5000         NaN  0.005314  0.000046     0.000088       NaN  0.000148
10000        NaN  0.009835  0.000044     0.000117       NaN  0.000122
20000        NaN       NaN  0.000089     0.000269       NaN  0.000217
50000        NaN       NaN  0.000262     0.001946       NaN  0.001047
75000        NaN       NaN  0.000675     0.003189       NaN  0.001589
100000       NaN       NaN  0.000574     0.004066       NaN  0.002088
150000       NaN       NaN  0.003025     0.005729       NaN  0.004253
200000       NaN       NaN  0.004402     0.007898       NaN  0.006301
300000       NaN       NaN  0.006052     0.011425       NaN  0.008742
400000       NaN       NaN  0.008120     0.016213       NaN  0.011279
500000       NaN       NaN  0.010332     0.020226       NaN  0.014790
600000       NaN       NaN  0.012205     0.023646       NaN  0.017401

lib     ave_cython     ave_numba     ave_numpy  ave_onnxruntime  ave_python  \
batch
1         0.000005  3.002400e-06  1.055270e-05     1.653790e-05    0.000012
10        0.000002  1.098170e-06  9.431700e-07     1.617430e-06    0.000008
100       0.000002  1.357537e-06  5.144980e-07     2.037730e-07    0.000011
200       0.000002  9.398675e-07  8.331650e-08     8.278650e-08    0.000009
500       0.000002  1.044134e-06  2.754420e-08     4.023220e-08    0.000009
1000      0.000001  1.021122e-06  1.963400e-08     3.115070e-08    0.000010
2000           NaN  1.086933e-06  1.275935e-08     2.052690e-08         NaN
3000           NaN  1.005031e-06  1.093637e-08     1.675440e-08         NaN
4000           NaN  9.897634e-07  7.848175e-09     1.631988e-08         NaN
5000           NaN  1.062815e-06  9.242400e-09     1.753960e-08         NaN
10000          NaN  9.835271e-07  4.392550e-09     1.173325e-08         NaN
20000          NaN           NaN  4.473800e-09     1.343460e-08         NaN
50000          NaN           NaN  5.236060e-09     3.891805e-08         NaN
75000          NaN           NaN  9.002027e-09     4.252482e-08         NaN
100000         NaN           NaN  5.744380e-09     4.065953e-08         NaN
150000         NaN           NaN  2.016368e-08     3.819613e-08         NaN
200000         NaN           NaN  2.200919e-08     3.948752e-08         NaN
300000         NaN           NaN  2.017391e-08     3.808462e-08         NaN
400000         NaN           NaN  2.030041e-08     4.053341e-08         NaN
500000         NaN           NaN  2.066300e-08     4.045292e-08         NaN
600000         NaN           NaN  2.034169e-08     3.940929e-08         NaN

lib      ave_sklearn
batch
1       7.200280e-05
10      4.772530e-06
100     5.977990e-07
200     4.118330e-07
500     1.149560e-07
1000    6.857600e-08
2000    4.233910e-08
3000    2.916010e-08
4000    2.059958e-08
5000    2.958310e-08
```

```
10000   1.223700e-08
20000   1.085213e-08
50000   2.093737e-08
75000   2.118651e-08
100000  2.087921e-08
150000  2.835613e-08
200000  3.150672e-08
300000  2.913874e-08
400000  2.819856e-08
500000  2.957939e-08
600000  2.900188e-08
```

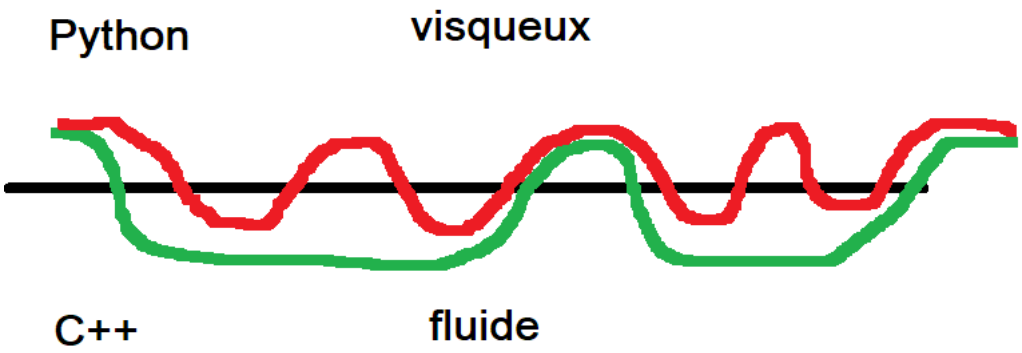
[81]:
```
libs = list(c for c in piv.columns if "ave_" in c)
ax = piv.plot(y=libs, logy=True, logx=True, figsize=(10, 5))
ax.set_title("Evolution du temps de prédiction selon la taille du batch")
ax.grid(True);
```



Le minimum obtenu est pour $10^{-8}s$ soit 10 ns. Cela montre que la comparaisson précédente était incomplète voire biaisée. Tout dépend de l'usage qu'on fait de la fonction de prédiction même s'il sera toujours possible de d'écrire un code spécialisé plus rapide que toute autre fonction générique. En général, plus on reste du côté Python, plus le programme est lent. Le nombre de passage de l'un à l'autre, selon la façon dont il est fait ralenti aussi. En tenant compte de cela, le programme rouge sera plus lent que le vert.

[82]:
```
from pyquickhelper.helpgen import NbImage
NbImage("pycpp.png")
```

[82]:



Ces résultats sont d'une façon générale assez volatile car le temps de calcul est enrobé dans plusieurs fonctions Python qui rendent une mesure précise difficile. Il reste néanmoins une bonne idée des ordres de grandeurs.

1.2 Random Forest

On reproduit les mêmes résultats pour une random forest mais la réécriture n'est plus aussi simple qu'une régression linéaire.

1.2.1 Une prédiction à la fois

[83]:
```
from sklearn.datasets import load_diabetes
diabetes = load_diabetes()
diabetes_X_train = diabetes.data[:-20]
diabetes_X_test  = diabetes.data[-20:]
diabetes_y_train = diabetes.target[:-20]
diabetes_y_test  = diabetes.target[-20:]
```

[84]:
```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=10)
rf.fit(diabetes_X_train, diabetes_y_train)
```

[84]:
```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=10, n_jobs=None, oob_score=False,
                      random_state=None, verbose=0, warm_start=False)
```

[85]:
```
memo_time = []
x = diabetes_X_test[:1]
memo_time.append(timeexe("sklearn-rf", "rf.predict(x)", repeat=100, number=20))
```

```
Moyenne: 696.91 µs Ecart-type 91.96 µs (with 20 runs) in [628.61 µs, 954.61 µs]
```

C'est beaucoup plus long que la régression linéaire. On essaye avec *onnx*.

[86]:
```
if ok_onnx:
    onnxrf_model = convert_sklearn(
        rf, 'model', [('input', FloatTensorType([None, clr.coef_.shape[0]]))],
        target_opset=11)
    onnxrf_model.ir_version = 6
    save_model(onnxrf_model, 'model_rf.onnx')
    model_onnx = onnx.load('model_rf.onnx')
```

[87]:
```
if ok_onnx:
    sess = onnxruntime.InferenceSession("model_rf.onnx")
    for i in sess.get_inputs():
        print('Input:', i)
    for o in sess.get_outputs():
        print('Output:', o)

    def predict_onnxrt_rf(x):
        return sess.run(["variable"], {'input': x})

    print(predict_onnxrt_rf(x.astype(numpy.float32)))
    memo_time.append(timeexe("onnx-rf", "predict_onnxrt_rf(x.astype(numpy.float32))",
                             repeat=100, number=20))
```

```
Input: NodeArg(name='input', type='tensor(float)', shape=[None, 10])
Output: NodeArg(name='variable', type='tensor(float)', shape=[None, 1])
[array([[264.5]], dtype=float32)]
Moyenne: 21.05 µs Ecart-type 11.04 µs (with 20 runs) in [12.03 µs, 32.32 µs]
```

C'est beaucoup plus rapide.

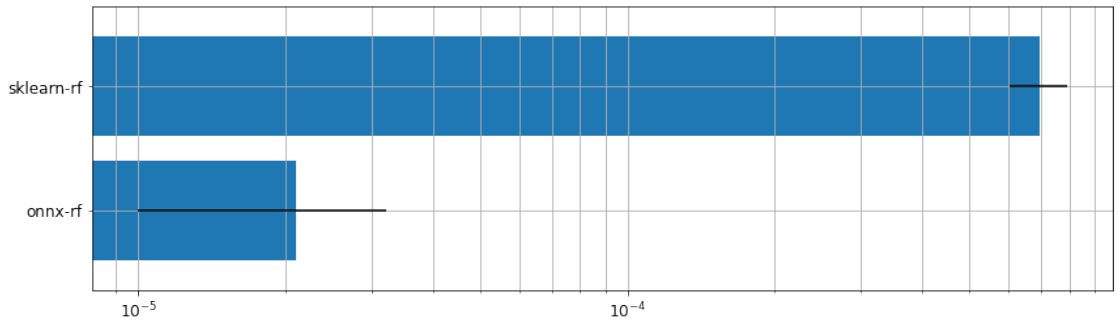
[88]:
```
import pandas
df2 = pandas.DataFrame(data=memo_time)
df2 = df2.set_index("legend").sort_values("average")
df2
```

[88]:

legend	average	deviation	first	first3	last3	repeat	min5	max5	code	run
onnx-rf	0.000021	0.000011	0.000109	0.000060	0.000012	100	0.000012	0.000032	predict_onnxrt_rf(x.astype(numpy.float32))	20
sklearn-rf	0.000697	0.000092	0.001094	0.000838	0.000671	100	0.000629	0.000955	rf.predict(x)	20

[89]:
```
fig, ax = plt.subplots(1, 1, figsize=(14,4))
df2[["average", "deviation"]].plot(kind="barh", logx=True, ax=ax, xerr="deviation",
                                   legend=False, fontsize=12, width=0.8)
ax.set_ylabel("")
ax.grid(b=True, which="major")
```

```
ax.grid(b=True, which="minor");
```



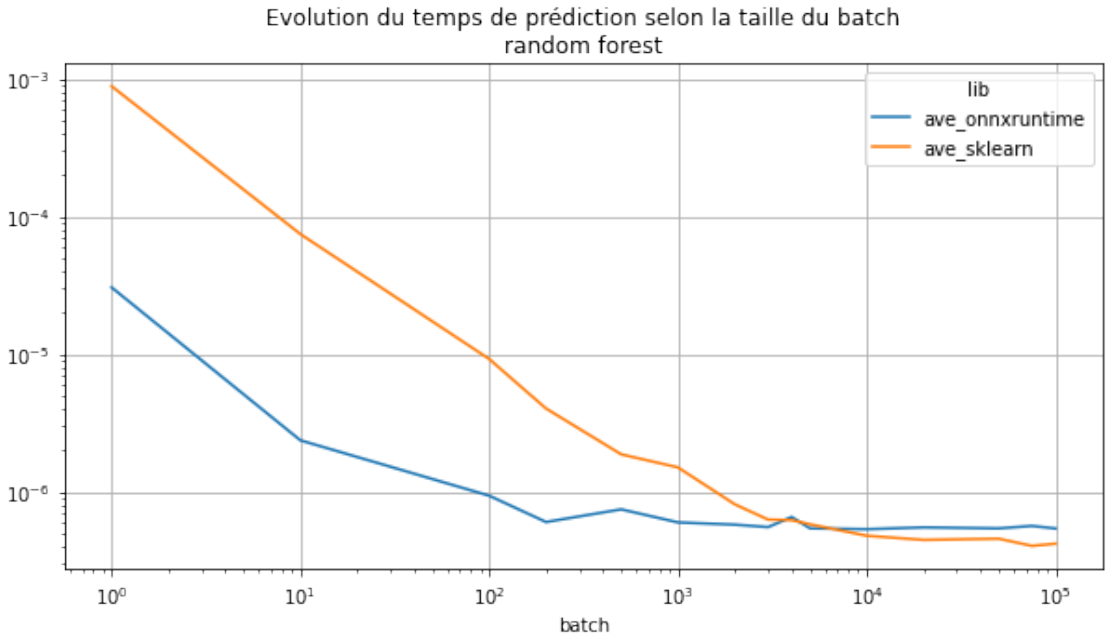
1.2.2 Prédiction en batch

[90]:
```
memo = []
batch = [1, 10, 100, 200, 500, 1000, 2000, 3000, 4000, 5000, 10000,
         20000, 50000, 75000, 100000, 150000, 200000, 300000, 400000,
         500000, 600000]
number = 10
repeat = 10
for i in batch[:15]:
    if i <= diabetes_X_test.shape[0]:
        mx = diabetes_X_test[:i]
    else:
        mxs = [diabetes_X_test] * (i // diabetes_X_test.shape[0] + 1)
        mx = numpy.vstack(mxs)
        mx = mx[:i]

    print("batch", "=", i)

    memo.append(timeexe("sklearn.predict %d" % i, "rf.predict(mx)",
                        repeat=repeat, number=number))
    memo[-1]["batch"] = i
    memo[-1]["lib"] = "sklearn"

    if ok_onnx:
        memo.append(timeexe("onnxruntime %d" % i,
                            "predict_onnxrt_rf(mx.astype(numpy.float32))",
                            repeat=repeat, number=number))
        memo[-1]["batch"] = i
        memo[-1]["lib"] = "onnxruntime"
```

```
batch = 1
Moyenne: 882.72 µs Ecart-type 263.99 µs (with 10 runs) in [683.09 µs, 1.60 ms]
Moyenne: 30.52 µs Ecart-type 17.66 µs (with 10 runs) in [18.42 µs, 81.77 µs]
batch = 10
Moyenne: 744.27 µs Ecart-type 175.60 µs (with 10 runs) in [611.42 µs, 1.25 ms]
Moyenne: 23.66 µs Ecart-type 9.45 µs (with 10 runs) in [17.72 µs, 47.13 µs]
batch = 100
```

```
Moyenne: 921.55 µs Ecart-type 140.71 µs (with 10 runs) in [729.57 µs, 1.20 ms]
Moyenne: 93.72 µs Ecart-type 43.45 µs (with 10 runs) in [67.00 µs, 221.17 µs]
batch = 200
Moyenne: 805.42 µs Ecart-type 71.19 µs (with 10 runs) in [730.12 µs, 975.80 µs]
Moyenne: 120.70 µs Ecart-type 26.89 µs (with 10 runs) in [108.65 µs, 201.05 µs]
batch = 500
Moyenne: 936.14 µs Ecart-type 61.18 µs (with 10 runs) in [851.63 µs, 1.05 ms]
Moyenne: 373.03 µs Ecart-type 95.25 µs (with 10 runs) in [262.67 µs, 555.07 µs]
batch = 1000
Moyenne: 1.50 ms Ecart-type 272.17 µs (with 10 runs) in [1.19 ms, 2.16 ms]
Moyenne: 599.52 µs Ecart-type 73.61 µs (with 10 runs) in [517.51 µs, 737.79 µs]
batch = 2000
Moyenne: 1.62 ms Ecart-type 204.76 µs (with 10 runs) in [1.45 ms, 2.08 ms]
Moyenne: 1.16 ms Ecart-type 139.14 µs (with 10 runs) in [988.10 µs, 1.52 ms]
batch = 3000
Moyenne: 1.89 ms Ecart-type 32.50 µs (with 10 runs) in [1.84 ms, 1.94 ms]
Moyenne: 1.66 ms Ecart-type 156.88 µs (with 10 runs) in [1.47 ms, 1.94 ms]
batch = 4000
Moyenne: 2.49 ms Ecart-type 374.50 µs (with 10 runs) in [2.20 ms, 3.36 ms]
Moyenne: 2.63 ms Ecart-type 298.92 µs (with 10 runs) in [2.15 ms, 3.22 ms]
batch = 5000
Moyenne: 2.90 ms Ecart-type 396.16 µs (with 10 runs) in [2.59 ms, 3.67 ms]
Moyenne: 2.72 ms Ecart-type 211.63 µs (with 10 runs) in [2.45 ms, 3.08 ms]
batch = 10000
Moyenne: 4.80 ms Ecart-type 314.66 µs (with 10 runs) in [4.44 ms, 5.28 ms]
Moyenne: 5.35 ms Ecart-type 314.76 µs (with 10 runs) in [5.00 ms, 6.08 ms]
batch = 20000
Moyenne: 8.95 ms Ecart-type 412.17 µs (with 10 runs) in [8.19 ms, 9.53 ms]
Moyenne: 11.03 ms Ecart-type 966.39 µs (with 10 runs) in [9.94 ms, 13.08 ms]
batch = 50000
Moyenne: 22.83 ms Ecart-type 2.58 ms (with 10 runs) in [20.47 ms, 28.28 ms]
Moyenne: 27.13 ms Ecart-type 2.12 ms (with 10 runs) in [25.96 ms, 33.36 ms]
batch = 75000
Moyenne: 30.37 ms Ecart-type 611.94 µs (with 10 runs) in [29.88 ms, 31.87 ms]
Moyenne: 42.39 ms Ecart-type 2.93 ms (with 10 runs) in [38.85 ms, 46.44 ms]
batch = 100000
Moyenne: 42.00 ms Ecart-type 3.44 ms (with 10 runs) in [39.88 ms, 51.96 ms]
Moyenne: 54.14 ms Ecart-type 2.95 ms (with 10 runs) in [52.11 ms, 61.71 ms]
```

[91]:
```
dfbrf = pandas.DataFrame(memo)[["average", "lib", "batch"]]
pivrf = dfbrf.pivot("batch", "lib", "average")
for c in pivrf.columns:
    pivrf["ave_" + c] = pivrf[c] / pivrf.index
libs = list(c for c in pivrf.columns if "ave_" in c)
ax = pivrf.plot(y=libs, logy=True, logx=True, figsize=(10, 5))
ax.set_title("Evolution du temps de prédiction selon la taille du batch\nrandom␣
 ↪forest")
ax.grid(True);
```


Evolution du temps de prédiction selon la taille du batch
random forest
lib
ave_onnxruntime
ave_sklearn
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
batch

[92]: