LogisticRegression - Results - Parml03

May 29, 2020

Logistic Regression Hyperparameter Tuning using FPGAs

Sotiris Pelekis 03400069

Mary Kaiktzoglou 03400052

Alexandros Vythoulkas 03400045

Spyros Pouros 03400071

Thanos Pantos 03400026

1 Run

This notebook shows how to train and apply many accelerated sklearn models, with a k-fold cross validation and hyperparameter tuning step.

[1]:
```
from inaccel.sklearn.linear_model import LogisticRegression
from sklearn.datasets import fetch_openml
from sklearn.linear_model import LogisticRegression as LogisticRegressionCPU
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from time import time
import matplotlib.pyplot as plt
import numpy as np
import random
import warnings

warnings.filterwarnings('ignore')
```

For this example we'll be using the Modified NIST (*National Institute of Standards and Technology*) Digits dataset which is a set of **handwritten digits** derived from the NIST Special Database 19 and converted to a 28x28 pixel grayscale image format. Further information on the dataset's contents and conversion process is available at https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#mnist8m. Begin by featching the data from openml and inspecting the schema.

```
[2]: X, y = fetch_openml('mnist_784', return_X_y = True)

     features = StandardScaler()
     X = features.fit_transform(X)

     label = LabelEncoder()
     y = label.fit_transform(y)

     print("Data shape:\tLabels: " + str(y.shape) + "\tFeatures: " + str(X.shape))
```

```
Data shape:     Labels: (70000,)        Features: (70000, 784)
```

Grid of Hyperparameters

```
[20]: param_grid = {
              'max_iter': (50, 100),
              'l1_ratio': (0.3, 0.9),
      }
```

FPGA Accelerated GridSearchCV

```
[21]: lr = LogisticRegression(n_accel=4)

      grid_search = GridSearchCV(lr, param_grid, scoring='accuracy', verbose=1, cv=3,␣
       ↪n_jobs=4, pre_dispatch = '4 * n_jobs')
```

```
[22]: start_time = time()
      grid_search.fit(X, y)
      elapsed_time = int((time() - start_time) * 100) / 100
```

```
Fitting 3 folds for each of 4 candidates, totalling 12 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  10 out of  12 | elapsed:   41.5s remaining:    8.3s
[Parallel(n_jobs=4)]: Done  12 out of  12 | elapsed:   42.7s finished
```

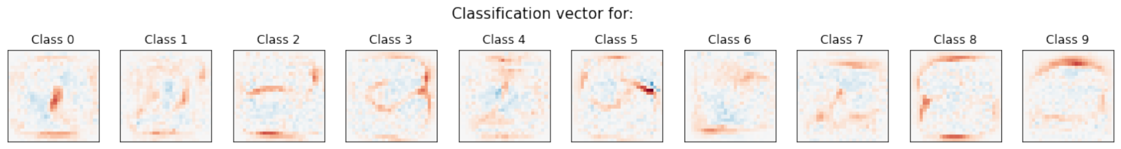
```
[23]: print("done in {0}s".format(elapsed_time))
      print("Best score: {0}".format(grid_search.best_score_))

      print("Best parameters set:")
      best_parameters = grid_search.best_estimator_.get_params()
      for param_name in sorted(list(param_grid.keys())):
          print("\t{0}: {1}".format(param_name, best_parameters[param_name]))
```

```
done in 50.88s
Best score: 0.9166714724143418
Best parameters set:
        l1_ratio: 0.3
        max_iter: 100
```

1.0.1 Evaluating Logistic Regression

In the next cell we validate the trained model using the produced coefficients to visualize its classes.

[24]:
```
coef = grid_search.best_estimator_.coef_.copy()
plt.figure(figsize=(20, 20))
plt.suptitle('Classification vector for:', y=0.91, fontsize = 15)
scale = np.abs(coef).max()
for i in range(len(np.unique(y))):
    fig = plt.subplot(10, 10, i + 1)
    fig.imshow(coef[i].reshape(28, 28), cmap=plt.cm.RdBu, vmin=-scale,␣
 ↪vmax=scale)
    fig.set_xticks(())
    fig.set_yticks(())
    fig.set_title('Class %i' % i)
```



CPU GridSearchCV

[25]:
```
lrCPU = LogisticRegressionCPU(intercept_scaling=0)

grid_searchCPU = GridSearchCV(lrCPU, param_grid, scoring='accuracy', verbose=1,␣
 ↪cv=3, n_jobs=4, pre_dispatch = '4 * n_jobs')
```

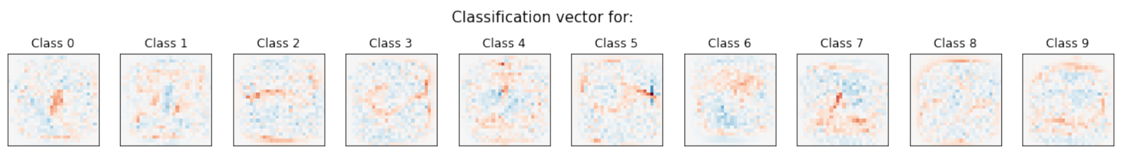
[26]:
```
start_timeCPU = time()
grid_searchCPU.fit(X, y)
elapsed_timeCPU = int((time() - start_timeCPU) * 100) / 100
```

```
Fitting 3 folds for each of 4 candidates, totalling 12 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  10 out of  12 | elapsed:  2.6min remaining:   31.3s
[Parallel(n_jobs=4)]: Done  12 out of  12 | elapsed:  2.8min finished
```

```
[27]: print("done in {0}s".format(elapsed_timeCPU))
      print("Best score: {0}".format(grid_searchCPU.best_score_))
      print("Best parameters set:")
      best_parameters = grid_searchCPU.best_estimator_.get_params()
      for param_name in sorted(list(param_grid.keys())):
          print("\t{0}: {1}".format(param_name, best_parameters[param_name]))
```

```
done in 191.62s
Best score: 0.916428611802047
Best parameters set:
        l1_ratio: 0.3
        max_iter: 50
```

```
[28]: coef = grid_searchCPU.best_estimator_.coef_.copy()
      plt.figure(figsize=(20, 20))
      plt.suptitle('Classification vector for:', y=0.91, fontsize = 15)
      scale = np.abs(coef).max()
      for i in range(len(np.unique(y))):
          fig = plt.subplot(10, 10, i + 1)
          fig.imshow(coef[i].reshape(28, 28), cmap=plt.cm.RdBu, vmin=-scale,␣
       ↪vmax=scale)
          fig.set_xticks(())
          fig.set_yticks(())
          fig.set_title('Class %i' % i)
```



Speedup Calculation

```
[29]: speedup = int(elapsed_timeCPU / elapsed_time * 100) / 100
      print("Speedup: " + str(speedup))
```

```
Speedup: 3.76
```

2 Results

2.1 Task 1

The speedup of the initial hyperparameter tuning which consisted only of two hyperparameter combination was 4.39 which is a considerable improvement.

2.2 Task 2

After changing the hyperparameter grid to contain 4 combinations of hyperparameters we can observe that the FPGA keeps scaling (at a slightly lower magnitude of 3.75 however). Nevertheless, the difference is too small to draw comparative conclusions given that cpu times often can fluctuate due to the different processes that may run ther.

2.3 Task 3

We reran both FPGA and CPU trainings so as to be able to compare the same hyperparameter selection problem as described above.

2.4 Task 4

In terms of accuracy, the l1_ratio hyperparameter has to take value 0.3 while selection of n_iter does not seem to affect that much given that the cpu and fpga provide different results (100, 50 respectively). Of course this is not something that depends on the hardware. Most probably it is due to the fact that no specific random state has been chosen for the Gridsearch CV-folds that can slightly alter the algorithm results.