NaiveBayes - Results - Parml03

May 29, 2020

FPGA Accelerated - Naive Bayes Classification

Sotiris Pelekis 03400069

Mary Kaiktzoglou 03400052

Alexandros Vythoulkas 03400045

Spyros Pouros 03400071

Thanos Pantos 03400026

1 Run

In this section we run all code inside loops in order to get the different results for all required parameters (number of classes, number of features). Results are stored in lists and are displayed later in this presentation.

[2]:
```
from inaccel.sklearn.naive_bayes import GaussianNB
import numpy as np
from sklearn import datasets
from sklearn import metrics
from time import time
```

[15]:
```
samples = 10000
speedups = []
classez = []
featurez = []
i = 0
for features in [500, 1000, 2000]:
    for classes in [10, 35, 60]:
        i += 1
        print("Run {} \n======================\n".format(i))
        X, y = datasets.make_classification(n_samples = samples, n_features =␣
↪features, n_informative = 400, n_redundant = 50,
                                            n_repeated = 50, n_classes =␣
↪classes, class_sep = 10.0, random_state = 0)
```

```
        ### Use only **10%** of the generated samples for the training part and␣
↪the rest for the classification

        # Samples used for training the model
        train_samples = int(0.1 * samples)

        train_labels = y[:train_samples]
        train_features = X[:train_samples]
        print("Train data shape:\n\tLabels: " + str(train_labels.shape) +␣
↪"\n\tFeatures: " + str(train_features.shape))

        test_labels = y[train_samples:]
        test_features = X[train_samples:]
        print("Test data shape:\n\tLabels: " + str(test_labels.shape) +␣
↪"\n\tFeatures: " + str(test_features.shape))

        ### Create a Naive Bayes object and **train** a model

        nb = GaussianNB()

        startTime = time()
        nb_model = nb.fit(train_features, train_labels)
        elapsedTime = int((time() - startTime) * 100) / 100

        print("Naive Bayes training (CPU) took: " + str(elapsedTime) + " sec")

        ### Calculate the predictions using the FPGA resources

        startTime = time()
        predictions = nb_model.predict(test_features)
        elapsedTime = int((time() - startTime) * 100) / 100

        print("Accuracy: " + str(int(metrics.accuracy_score(test_labels,␣
↪predictions) * 10000) / 100) + "%")
        print("Naive Bayes classification (FPGA) took: " + str(elapsedTime) + "␣
↪sec")

        ### Import the Original NaiveBayes class to compare the classification␣
↪part execution time

        from sklearn.naive_bayes import GaussianNB as OriginalNB

        cpuNB = OriginalNB()

        cpu_model = cpuNB.fit(train_features, train_labels)
```

```
        ### Calculate the predictions using the CPU resources

        startTimeCPU = time()
        predictionsCPU = cpu_model.predict(test_features)
        elapsedTimeCPU = int((time() - startTimeCPU) * 100) / 100

        print("Accuracy: " + str(int(metrics.accuracy_score(test_labels,␣
↪predictionsCPU) * 10000) / 100) + "%")
        print("Naive Bayes classification (CPU) took: " + str(elapsedTimeCPU) +␣
↪" sec")

        ### Speedup Calculation

        speedup = int(elapsedTimeCPU / elapsedTime * 100) / 100
        print("Speedup: " + str(speedup))
        classez.append(classes)
        featurez.append(features)
        speedups.append(speedup)
        print("\n")
```

```
Run 1
========================

Train data shape:
        Labels: (1000,)
        Features: (1000, 500)
Test data shape:
        Labels: (9000,)
        Features: (9000, 500)
Naive Bayes training (CPU) took: 0.0 sec
Accuracy: 99.13%
Naive Bayes classification (FPGA) took: 0.04 sec
Accuracy: 99.13%
Naive Bayes classification (CPU) took: 0.3 sec
Speedup: 7.5


Run 2
========================

Train data shape:
        Labels: (1000,)
        Features: (1000, 500)
Test data shape:
        Labels: (9000,)
        Features: (9000, 500)
Naive Bayes training (CPU) took: 0.01 sec
```

```
Accuracy: 98.67%
Naive Bayes classification (FPGA) took: 0.04 sec
Accuracy: 98.67%
Naive Bayes classification (CPU) took: 1.03 sec
Speedup: 25.75


Run 3
========================

Train data shape:
        Labels: (1000,)
        Features: (1000, 500)
Test data shape:
        Labels: (9000,)
        Features: (9000, 500)
Naive Bayes training (CPU) took: 0.01 sec
Accuracy: 98.42%
Naive Bayes classification (FPGA) took: 0.05 sec
Accuracy: 98.42%
Naive Bayes classification (CPU) took: 1.75 sec
Speedup: 35.0


Run 4
========================

Train data shape:
        Labels: (1000,)
        Features: (1000, 1000)
Test data shape:
        Labels: (9000,)
        Features: (9000, 1000)
Naive Bayes training (CPU) took: 0.01 sec
Accuracy: 99.0%
Naive Bayes classification (FPGA) took: 0.06 sec
Accuracy: 99.0%
Naive Bayes classification (CPU) took: 0.68 sec
Speedup: 11.33


Run 5
========================

Train data shape:
        Labels: (1000,)
        Features: (1000, 1000)
Test data shape:
```

```
        Labels: (9000,)
        Features: (9000, 1000)
Naive Bayes training (CPU) took: 0.01 sec
Accuracy: 99.27%
Naive Bayes classification (FPGA) took: 0.07 sec
Accuracy: 99.27%
Naive Bayes classification (CPU) took: 2.35 sec
Speedup: 33.57


Run 6
========================

Train data shape:
        Labels: (1000,)
        Features: (1000, 1000)
Test data shape:
        Labels: (9000,)
        Features: (9000, 1000)
Naive Bayes training (CPU) took: 0.01 sec
Accuracy: 94.84%
Naive Bayes classification (FPGA) took: 0.07 sec
Accuracy: 94.84%
Naive Bayes classification (CPU) took: 4.03 sec
Speedup: 57.57


Run 7
========================

Train data shape:
        Labels: (1000,)
        Features: (1000, 2000)
Test data shape:
        Labels: (9000,)
        Features: (9000, 2000)
Naive Bayes training (CPU) took: 0.02 sec
Accuracy: 99.08%
Naive Bayes classification (FPGA) took: 0.11 sec
Accuracy: 99.08%
Naive Bayes classification (CPU) took: 1.36 sec
Speedup: 12.36


Run 8
========================

Train data shape:
```

```
        Labels: (1000,)
        Features: (1000, 2000)
Test data shape:
        Labels: (9000,)
        Features: (9000, 2000)
Naive Bayes training (CPU) took: 0.02 sec
Accuracy: 98.61%
Naive Bayes classification (FPGA) took: 0.12 sec
Accuracy: 98.61%
Naive Bayes classification (CPU) took: 4.72 sec
Speedup: 39.33


Run 9
=======================

Train data shape:
        Labels: (1000,)
        Features: (1000, 2000)
Test data shape:
        Labels: (9000,)
        Features: (9000, 2000)
Naive Bayes training (CPU) took: 0.02 sec
Accuracy: 87.76%
Naive Bayes classification (FPGA) took: 0.13 sec
Accuracy: 87.76%
Naive Bayes classification (CPU) took: 8.16 sec
Speedup: 62.76
```

2 Results

2.1 Task 1

We inspected the initial results in the original Naive-Bayes file.

2.2 Task 2

Accuracies are exactly the same for both FPGA and CPU at all runs. Results are consistent.

2.3 Task 3

The outputs are displayed above for all runs, using loops for the the use of all requested parameters (classes, features)

2.4 Task 4

The best speedup was achieved for the maximum numbers of classes and features. It seems that inference is a task that gets more and more complicated for the CPU as numbers of classes and features grow larger and larger. However the FPGA, as it is explicitly designed and programmed and optimized for this kind of application can manage the overhead added by big numbers of classes and features more effectively. However, it is important to keep in mind that the FPGA cannot keep scaling beyond its limits (2047 features & 64 classes) while CPU could potentially handle any value of these parameters.

2.5 Task 5

Final results and required charts are depicted below.

[27]:
```
import pandas as pd
results = pd.DataFrame(np.array(list(zip(classez, featurez, speedups))),␣
 ↪columns=["Classes", "Features", "Speedup"])
print("Final results:\n")
print(results)
```

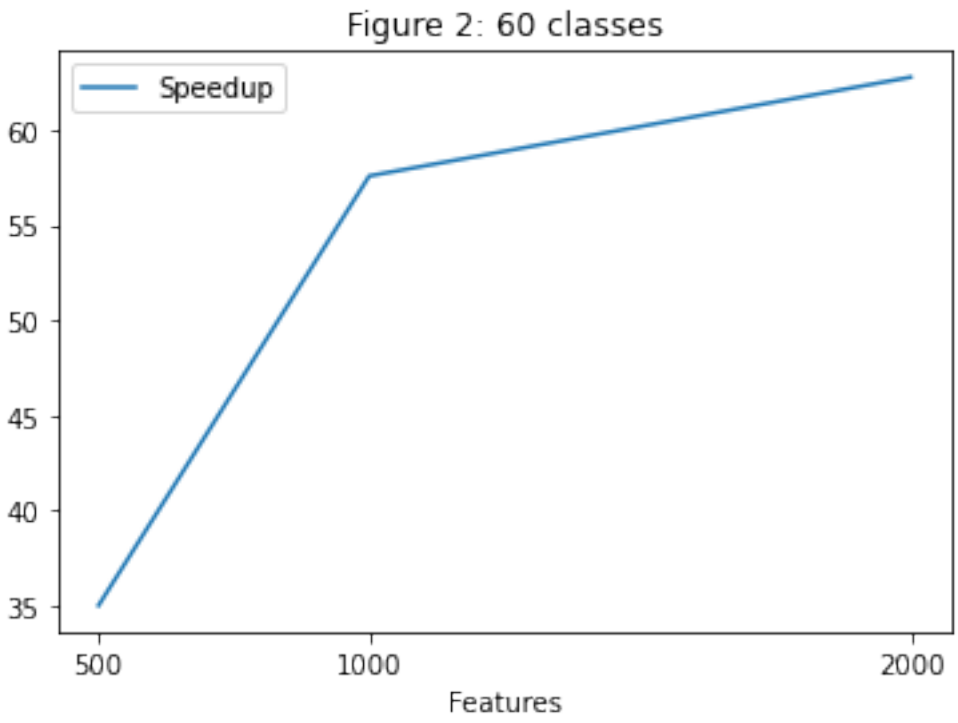
```
Final results:

   Classes  Features  Speedup
0     10.0     500.0     7.50
1     35.0     500.0    25.75
2     60.0     500.0    35.00
3     10.0    1000.0    11.33
4     35.0    1000.0    33.57
5     60.0    1000.0    57.57
6     10.0    2000.0    12.36
7     35.0    2000.0    39.33
8     60.0    2000.0    62.76
```

2.5.1 Results and plots for 60 classes

[28]:
```
results60 = results[results["Classes"]==60]
print(results60)
results60.plot(x="Features", y="Speedup", xticks=[500, 1000, 2000],␣
 ↪title="Figure 2: 60 classes")
```

```
   Classes  Features  Speedup
2     60.0     500.0    35.00
5     60.0    1000.0    57.57
8     60.0    2000.0    62.76
```

[28]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f64bce19250>
```

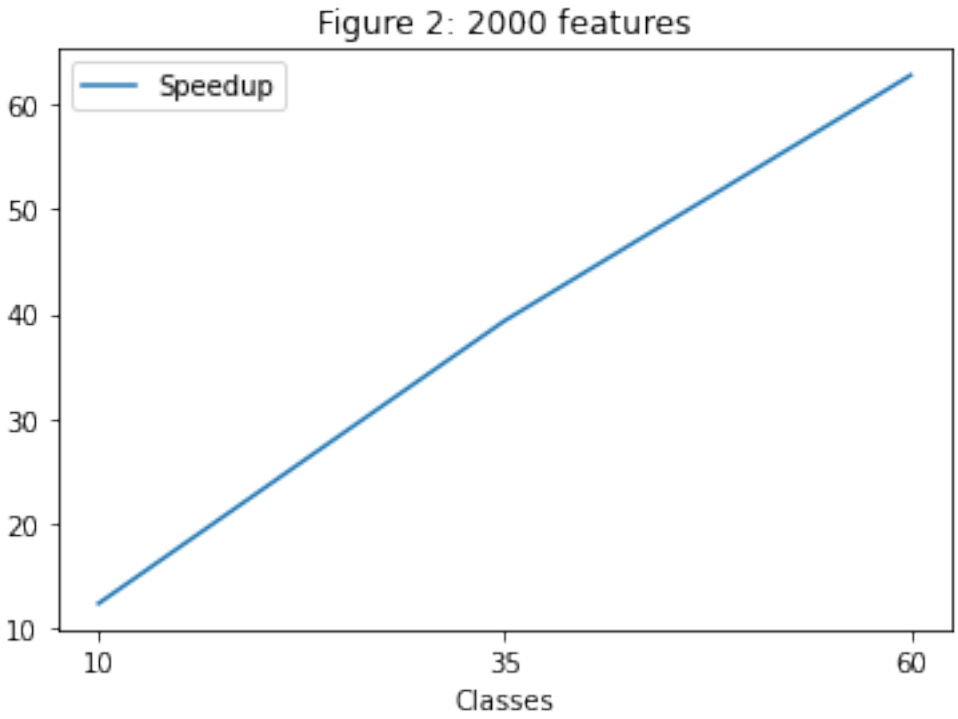


2.5.2 Results and plots for 2000 features

[29]:
```
results2000 = results[results["Features"]==2000]
print(results2000)
results2000.plot(x="Classes", y="Speedup", xticks = [10, 35, 60], title="Figure␣
↪2: 2000 features")
```

```
   Classes  Features  Speedup
6     10.0    2000.0    12.36
7     35.0    2000.0    39.33
8     60.0    2000.0    62.76
```

[29]: `<matplotlib.axes._subplots.AxesSubplot at 0x7f64bcd79210>`



2.6 Task 6

FPGA speedup during inference seems to work both for increasing numbers of classes and features. However, having a lot of classes guarantees a minimum speedup of 35 irrespective of number of features while for 2000 features if the number of classes is small lower speedups can be achieved. So it makes better sense to use this specific FPGA for a big number of classes and then hope for even higher speedups when increasing the number of features. That seems to be justified by the speed of the FPGA to execute operations and comparisons. Many classes mean many likelihood computations and many likelihood comparisons so as to decide the class of each test sample. Due to its hardware nature the FPGA significantly speeds up this process for many classes.