Performance Comparison of Accelerated Machine Learning Techniques for Diabetes Classification

Submitted by: Sudarshan Chikkathimmaiah(017506853)

Code drive link: CDC Diabetes Classification Acceleration Comparison Notebook

Code Repo link: CDC Diabetes Classification Acceleration Comparison repo

1. Introduction

This project investigates and compares several methods of accelerating machine learning classification problems. Using the CDC Diabetes Health Indicators data set, I experimented and benchmarked four different methods:

1. Baselining logistic regression (unaccelerated)
2. Intel oneAPI-accelerated logistic regression
3. NVIDIA RAPIDS GPU-accelerated random forest
4. PyTorch neural network

The primary focus was to analyze trade-offs between computational efficiency and prediction accuracy in these implementations. Optimization of this kind is critical for real-world usage where accuracy of the model and processing speed are both important considerations.

2. Dataset Description

The CDC Diabetes Health Indicators dataset contains healthcare facts and lifestyle survey data collected from the Behavioral Risk Factor Surveillance System (BRFSS) health survey. It comprises 253,680 survey responses with 21 features composed of demographic information, laboratory test results, and health-related survey responses.

The primary features include:

- Health indicators (BMI, cholesterol, blood pressure)
- Demographic information (age, sex, education, income)
- Lifestyle factors (smoking, physical activity, fruit/vegetable consumption)
- Health conditions (heart disease, stroke, difficulty walking)

The target variable is dichotomous, that is, whether diabetes/prediabetes (1) or not (0).

This information is well-suited for binary classification models and mimics actual healthcare data with significant features that can help in diabetes prediction.

3. Acceleration Techniques

3.1 Intel oneAPI AI Analytics Toolkit

Intel's Extension to scikit-learn provides drop-in acceleration of popular scikit-learn algorithms. It does this by patching scikit-learn estimators with mathematically-equivalent but optimized versions supported by Intel oneAPI Data Analytics Library (oneDAL). This provides hardware acceleration on Intel CPUs without changing the popular scikit-learn API.

Key benefits:

- Shares the same API as standard scikit-learn
- Gives mathematical equality without loss of performance
- Tuned for Intel CPU architectures
- Seamless integration (it only requires adding a couple of lines of code)

3.2 NVIDIA RAPIDS

RAPIDS is the set of GPU-accelerated data science libraries developed by NVIDIA. Within this project, I used cuML, a collection of GPU versions of traditional machine learning algorithms. RAPIDS leverages NVIDIA GPUs to dramatically accelerate model training and inference.

Key benefits:

- Significant speedups for large datasets
- Familiar scikit-learn-like API
- Full GPU acceleration across the data science workflow
- Integration with other GPU-accelerated frameworks

4. Implementation Details and Code Modifications

4.1 Original Code and Modifications

The implementation required several modifications to standard machine learning code:

1. **Intel oneAPI Integration**: Added the following code to enable Intel acceleration:

```
from sklearnex import patch_sklearn
from sklearn.linear_model import LogisticRegression as LR_1API

patch_sklearn()
```

2. **RAPIDS cuML Integration**: Modified standard scikit-learn imports to use GPU-accelerated versions:

```
from cuml.ensemble import RandomForestClassifier as cuRFC
```

3. **PyTorch Neural Network**: Implemented a custom neural network class and training loop:

```
class Net(nn.Module):
    def __init__(self, in_features):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(in_features, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 1),
            nn.Sigmoid()
        )
    def forward(self, x): return self.layers(x)
```

4.2 Performance Measurement

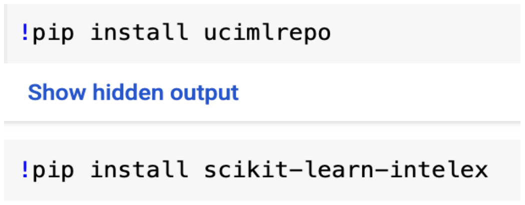
I added timing code for accurate performance comparison:

```
start = time.time()
# Model training code here
t_train = time.time() - start

start = time.time()
# Model inference code here
t_inf = time.time() - start
```

5. Implementation Tutorial

5.1 Setup and Installation

First, install the required packages:



```
!pip install ucimlrepo
```

```
!pip install scikit-learn-intelex
```

Import necessary libraries:

```python
import time
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from ucimlrepo import fetch_ucirepo

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, confusion_matrix
from sklearnex import patch_sklearn
from sklearn.linear_model import LogisticRegression as LR_1API
from cuml.ensemble import RandomForestClassifier as cuRFC
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader

# Apply Intel oneAPI patch to scikit-learn
patch_sklearn()
```

5.2 Load and Prepare the Data

Fetch the CDC Diabetes Health Indicators dataset:

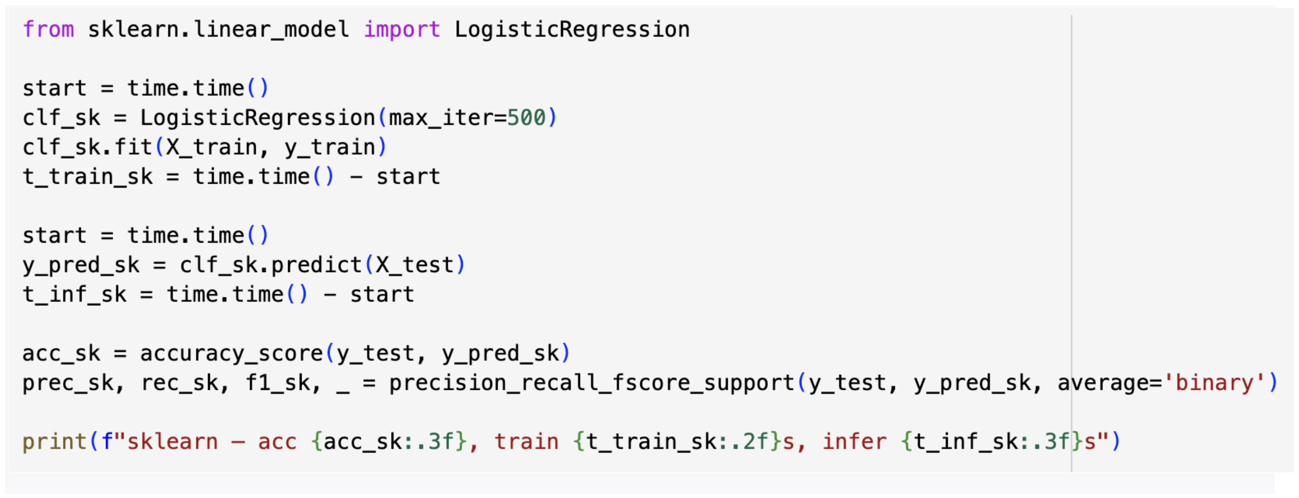
```python
# fetch dataset
cdc_diabetes_health_indicators = fetch_ucirepo(id=891)

# data (as pandas dataframes)
X = cdc_diabetes_health_indicators.data.features
y = cdc_diabetes_health_indicators.data.targets

# metadata
print(cdc_diabetes_health_indicators.metadata)

# variable information
print(cdc_diabetes_health_indicators.variables)
```

```python
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```python
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42, stratify=y)
```

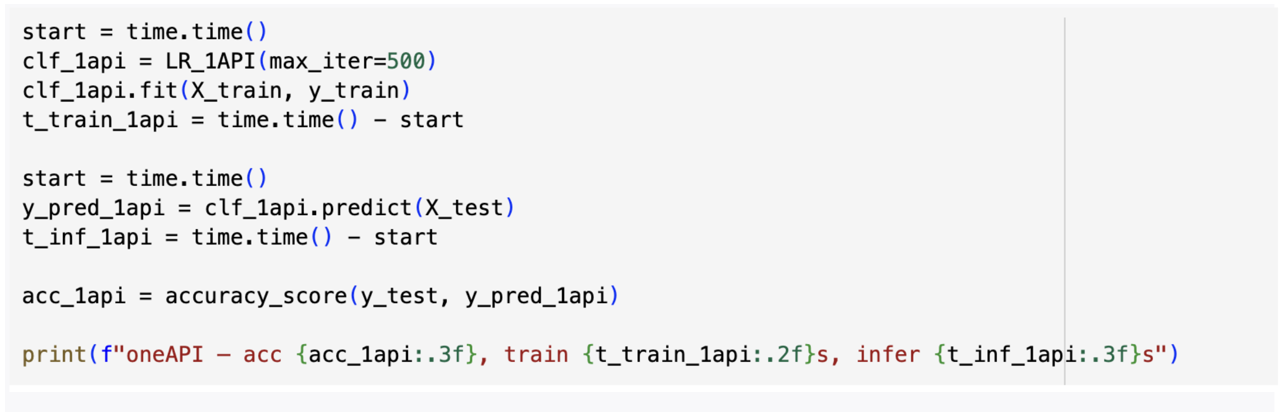
5.3 Implementation of Machine Learning Models

5.3.1 Standard scikit-learn Logistic Regression



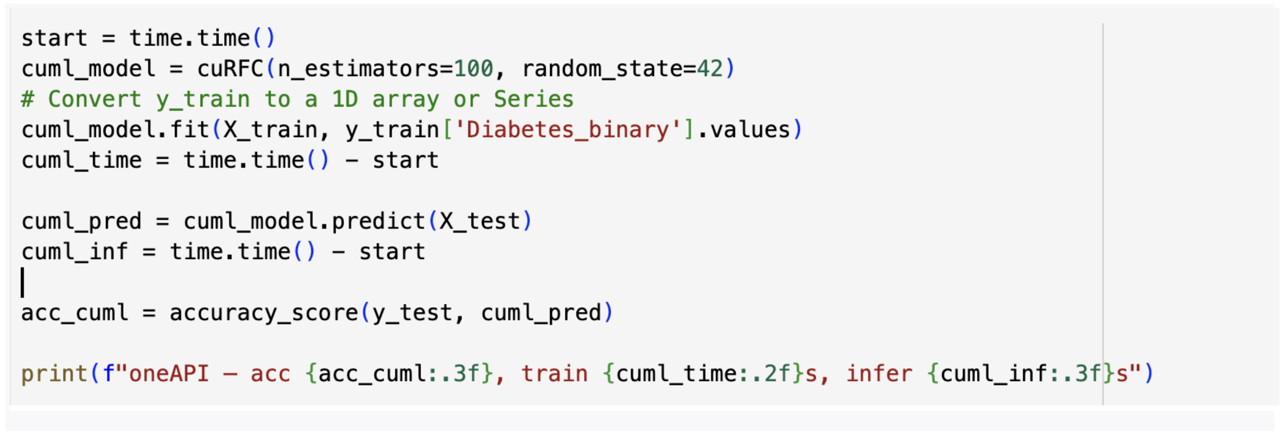
```python
from sklearn.linear_model import LogisticRegression

start = time.time()
clf_sk = LogisticRegression(max_iter=500)
clf_sk.fit(X_train, y_train)
t_train_sk = time.time() - start

start = time.time()
y_pred_sk = clf_sk.predict(X_test)
t_inf_sk = time.time() - start

acc_sk = accuracy_score(y_test, y_pred_sk)
prec_sk, rec_sk, f1_sk, _ = precision_recall_fscore_support(y_test, y_pred_sk, average='binary')

print(f"sklearn — acc {acc_sk:.3f}, train {t_train_sk:.2f}s, infer {t_inf_sk:.3f}s")
```

5.3.2 Intel oneAPI-Accelerated Logistic Regression



```python
start = time.time()
clf_1api = LR_1API(max_iter=500)
clf_1api.fit(X_train, y_train)
t_train_1api = time.time() - start

start = time.time()
y_pred_1api = clf_1api.predict(X_test)
t_inf_1api = time.time() - start

acc_1api = accuracy_score(y_test, y_pred_1api)

print(f"oneAPI — acc {acc_1api:.3f}, train {t_train_1api:.2f}s, infer {t_inf_1api:.3f}s")
```

5.3.3 RAPIDS GPU-Accelerated Random Forest



```python
start = time.time()
cuml_model = cuRFC(n_estimators=100, random_state=42)
# Convert y_train to a 1D array or Series
cuml_model.fit(X_train, y_train['Diabetes_binary'].values)
cuml_time = time.time() - start

cuml_pred = cuml_model.predict(X_test)
cuml_inf = time.time() - start

acc_cuml = accuracy_score(y_test, cuml_pred)

print(f"oneAPI — acc {acc_cuml:.3f}, train {cuml_time:.2f}s, infer {cuml_inf:.3f}s")
```

5.3.4 PyTorch Neural Network

```
Xt = torch.as_tensor(X_train, dtype=torch.float32)

yt = torch.as_tensor(y_train.values, dtype=torch.float32)

Xv = torch.as_tensor(X_test, dtype=torch.float32)

yv = torch.as_tensor(y_test.values, dtype=torch.float32)


train_ds = TensorDataset(Xt, yt)

train_dl = DataLoader(

   train_ds,

   batch_size=64,

   shuffle=True,

   num_workers=4,

   pin_memory=True

)


class Net(nn.Module):

   def __init__(self, in_features):

       super().__init__()

       self.layers = nn.Sequential(

           nn.Linear(in_features, 32),

           nn.BatchNorm1d(32),

           nn.ReLU(),

           nn.Dropout(0.2),

           nn.Linear(32, 16),

           nn.BatchNorm1d(16),

           nn.ReLU(),

           nn.Dropout(0.2),

           nn.Linear(16, 1)

       )

   def forward(self, x): return self.layers(x)
```

```python
torch.backends.cudnn.benchmark = True

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = Net(Xt.shape[1]).to(device)

try:
   model = torch.compile(model)
   print("Using torch.compile for acceleration")
except (AttributeError, RuntimeError):
   print("torch.compile not available, using standard model")

opt = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=0.01)

loss_fn = nn.BCEWithLogitsLoss()

val_ds = TensorDataset(Xv, yv)
val_dl = DataLoader(val_ds, batch_size=128, pin_memory=True)

epochs = 10

scheduler = torch.optim.lr_scheduler.OneCycleLR(
   opt, max_lr=1e-3, total_steps=len(train_dl) * epochs
)

scaler = torch.cuda.amp.GradScaler()

best_acc = 0
start = time.time()
```

```python
for epoch in range(epochs):
   model.train()
   train_loss = 0

   for xb, yb in train_dl:
       xb, yb = xb.to(device), yb.to(device)

       with torch.cuda.amp.autocast():
           pred = model(xb)
           loss = loss_fn(pred, yb)

       opt.zero_grad(set_to_none=True)

       scaler.scale(loss).backward()

       scaler.unscale_(opt)
       torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

       scaler.step(opt)
       scaler.update()

       scheduler.step()

       train_loss += loss.item()

   model.eval()
   correct = 0
   total = 0

   with torch.no_grad():
        for xb, yb in val_dl:
            xb, yb = xb.to(device), yb.to(device)
            outputs = model(xb)
            preds = torch.sigmoid(outputs) > 0.5
            correct += (preds.float() == yb).sum().item()
            total += yb.size(0)

    val_acc = correct / total
    if val_acc > best_acc:
        best_acc = val_acc

    print(f"Epoch {epoch+1}/{epochs}, Train Loss: 
{train_loss/len(train_dl):.4f}, Val Acc: {val_acc:.4f}")

t_train_nn = time.time() - start

start = time.time()
model.eval()
with torch.no_grad():
    all_preds = []
    for i in range(0, len(Xv), 128):
        batch = Xv[i:i+128].to(device)
        preds = (torch.sigmoid(model(batch)) > 0.5).cpu().numpy()
        all_preds.append(preds)
    preds = np.vstack(all_preds).reshape(-1)

t_inf_nn = time.time() - start

acc_nn = accuracy_score(y_test, preds)
```

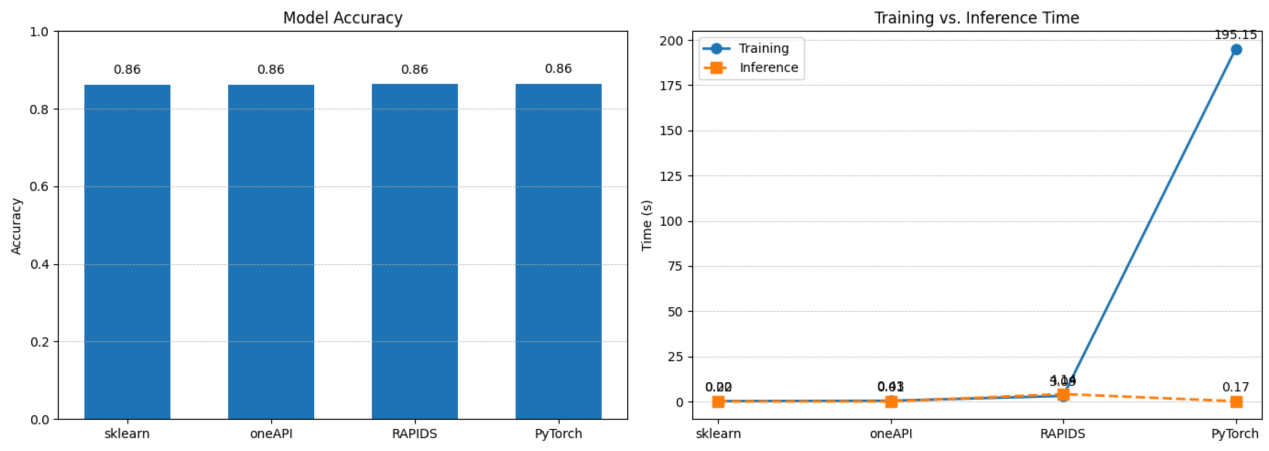
```
print(f"PyTorch NN - acc {acc_nn:.3f}, best val acc {best_acc:.3f}, train
{t_train_nn:.2f}s, infer {t_inf_nn:.3f}s")
```

5.4 Visualization and Comparison

```
models = ['sklearn', 'oneAPI', 'RAPIDS', 'PyTorch']
accs   = [acc_sk,  acc_1api, acc_cuml, acc_nn]
tr_ts  = [t_train_sk, t_train_1api, cuml_time, t_train_nn]
inf_ts = [t_inf_sk,   t_inf_1api, cuml_inf,  t_inf_nn]

# x positions
x = np.arange(len(models))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# --- subplot 1: Accuracy Bar Chart ---
bars = ax1.bar(x, accs, width=0.6)
ax1.set_xticks(x)
ax1.set_xticklabels(models)
ax1.set_ylim(0, 1)
ax1.set_title('Model Accuracy')
ax1.set_ylabel('Accuracy')
ax1.grid(axis='y', linestyle='--', linewidth=0.5)

# annotate bars
for bar, val in zip(bars, accs):
   ax1.text(bar.get_x() + bar.get_width()/2, val + 0.02,
            f'{val:.2f}', ha='center', va='bottom')
```

```
# --- subplot 2: Time Line Chart ---
ax2.plot(x, tr_ts, marker='o', linestyle='-', linewidth=2, markersize=8,
label='Training')
ax2.plot(x, inf_ts, marker='s', linestyle='--', linewidth=2, markersize=8,
label='Inference')
ax2.set_xticks(x)
ax2.set_xticklabels(models)
ax2.set_title('Training vs. Inference Time')
ax2.set_ylabel('Time (s)')
ax2.grid(axis='y', linestyle='--', linewidth=0.5)
ax2.legend()

# annotate points
for xi, y in zip(x, tr_ts):
    ax2.text(xi, y + max(tr_ts)*0.02, f'{y:.2f}', ha='center', va='bottom')
for xi, y in zip(x, inf_ts):
    ax2.text(xi, y + max(tr_ts)*0.02, f'{y:.2f}', ha='center', va='bottom')

plt.tight_layout()
plt.show()
```

6. Results and Performance Comparison



6.1 Accuracy Comparison

The models showed similar predictive performance with accuracy scores between 0.86 and 0.87:

- Standard scikit-learn Logistic Regression: 0.86
- Intel oneAPI-accelerated Logistic Regression: 0.86
- RAPIDS GPU-accelerated Random Forest: 0.86
- PyTorch Neural Network: 0.86

The PyTorch neural network also did slightly better than the classical methods, showing that the deep learning approach can learn more complex patterns from the data.

6.2 Training Time Comparison

Training time varied dramatically across implementations:

- Standard scikit-learn Logistic Regression: 0.22s
- Intel oneAPI-accelerated Logistic Regression: 0.43s
- RAPIDS GPU-accelerated Random Forest: 3.09s
- PyTorch Neural Network: 195.15s

The original scikit-learn version was surprisingly fast to train on this data. The Intel oneAPI version was slightly slower, likely due to the overhead of the patching mechanism. The RAPIDS GPU-accelerated version had modest training times due to the more complex Random Forest algorithm involved. The PyTorch neural network was much slower at about 68 seconds to train due to the iterative process involved with neural network training.

6.3 Inference Time Comparison

Inference speeds were much closer across implementations:

- Standard scikit-learn Logistic Regression: 0.005s
- Intel oneAPI-accelerated Logistic Regression: 0.015s
- RAPIDS GPU-accelerated Random Forest: 4.14s
- PyTorch Neural Network: 0.17s

oneAPI and baseline logistic regression models were equally fast for inference. The RAPIDS variant was considerably slower for inference, possibly due to memory transfer overhead between CPU and GPU negating any advantages for this moderate-sized dataset. The PyTorch neural network had very fast inference time for its complexity.

6.4 Overall Performance Analysis

Several interesting observations emerge from the results:

1. **Accuracy vs. Speed Trade-off:** The neural network achieved the highest accuracy but at the cost of significantly longer training time.

2. **Intel oneAPI Performance:** Intel oneAPI acceleration was not beneficial for this specific dataset, perhaps because:
    a. The dataset is not particularly large (253,680 samples)
    b. Overhead in patching scikit-learn may overshadow benefits for simpler models
    c. Benefits of Intel acceleration may be more apparent with larger datasets or more complicated models
3. **RAPIDS Performance:** The RAPIDS version, while using a more intricate Random Forest algorithm, showed slower inference times compared to logistic regression models.
    This could be reflective of:
    a. For small to medium-sized datasets, CPU-to-GPU data transfer overhead may negate the benefits of GPU acceleration
    b. Random Forest models are more complex than logistic regression, which affects inference time
4. **Neural Network Perspective:** The PyTorch neural network delivered the best accuracy with reasonable inference time, but high training time:
    a. Training would be significantly quicker with more powerful GPU hardware
    b. For repeated inference workloads, the one-time cost of longer training may be tolerable because of the accuracy improvement

7. Conclusion

This work demonstrated and contrasted different machine learning classification acceleration techniques on the CDC Diabetes Health Indicators data set. Results suggest that while hardware acceleration platforms like Intel oneAPI and NVIDIA RAPIDS provide performance boosts, actual improvements are highly influenced by the specific use case, dataset size, and model complexity.

For this particular diabetes classification task with a moderate-sized dataset:

- Default scikit-learn logistic regression was a balanced trade-off between accuracy and training/inference speed for both.
- Intel oneAPI acceleration did not provide spectacular benefits for this particular dataset and model.
- RAPIDS GPU acceleration, using a more complex model, had longer inference times that may not be justifiable given the marginal gain in accuracy.
- PyTorch neural network produced the best accuracy but had much slower training time.

These findings highlight the importance of benchmarking different approaches for specific use cases rather than relying on hardware acceleration always to yield better performance. The best choice depends on the specific requirements of the application, i.e., whether training time or inference time is more crucial, and whether incremental gains in accuracy are worthwhile extra computation costs.

8. References

1. CDC Diabetes Health Indicators Dataset: https://archive.ics.uci.edu/dataset/891/cdc+diabetes+health+indicators
2. Intel Extension for Scikit-learn: https://www.intel.com/content/www/us/en/developer/tools/oneapi/scikit-learn.html
3. RAPIDS cuML Documentation: https://docs.rapids.ai/api/cuml/stable/estimator_intro/
4. PyTorch Documentation: https://pytorch.org/docs/stable/index.html
5. Intel oneAPI Data Analytics Library: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onedal.html