Report: Comparison of Dimensionality Reduction and Classifier Performance on Breast Cancer Dataset

Dataset Features

1. **Number of Instances (Samples):**
   - **569 samples**
2. **Number of Features:**
   - **30 numeric features (all float type), describing the characteristics of the cell nuclei.**
3. **Target Variable:**
   - **Binary classification:**
     - **0: Malignant (Cancerous)**
     - **1: Benign (Non - Cancerous)**

Objective:

The goal of this experiment was to apply three dimensionality reduction techniques—Self-Organizing Maps (SOM), Restricted Boltzmann Machines (RBM), and Autoencoders—and compare their performance against the original dataset using three classifiers: XGBoost, LightGBM, and Cat Boost. Performance was measured in terms of classification accuracy and execution time.

Code:

```
import numpy as np

import pandas as pd

import time

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy_score

from sklearn.datasets import load_breast_cancer

from sklearn.utils import resample



# Classifier libraries
```

```
import xgboost as xgb
from lightgbm import LGBMClassifier
from catboost import CatBoostClassifier

# PyTorch libraries for RBM and Autoencoder
import torch
from torch import nn
from torch.utils.data import DataLoader

# SOM implementation
from minisom import MiniSom

# ---------------------------------------------------------------------------
# Step 1: Load and Preprocess Data
# ---------------------------------------------------------------------------
# Load the Breast Cancer Dataset
data = load_breast_cancer()
X = data.data  # Features
y = data.target  # Target (0 = Malignant, 1 = Benign)

# Optional: Upsample the dataset to ~700 rows
X, y = resample(X, y, n_samples=700, random_state=42)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```python
# Convert data to PyTorch tensors
X_train_tensor = torch.FloatTensor(X_train)
X_test_tensor = torch.FloatTensor(X_test)

# ---------------------------------------------------------------------------
# Step 2: Dimensionality Reduction
# ---------------------------------------------------------------------------

print("Applying SOM...")
# 1. Self-Organizing Map (SOM)
som = MiniSom(x=10, y=10, input_len=X_train.shape[1], sigma=1.0, learning_rate=0.5)
som.random_weights_init(X_train)
som.train_random(X_train, 500)

# Project data into reduced dimensions
X_train_som = np.array([som.winner(x) for x in X_train])
X_test_som = np.array([som.winner(x) for x in X_test])

print("Applying RBM...")
# 2. Restricted Boltzmann Machine (RBM)
class RBM(nn.Module):
  def __init__(self, n_visible, n_hidden):
    super(RBM, self).__init__()
    self.W = nn.Parameter(torch.randn(n_hidden, n_visible) * 0.1)
    self.h_bias = nn.Parameter(torch.zeros(n_hidden))
    self.v_bias = nn.Parameter(torch.zeros(n_visible))

  def forward(self, v):
    h_prob = torch.sigmoid(torch.matmul(v, self.W.t()) + self.h_bias)
```

```
        return h_prob

# Initialize and train RBM
rbm = RBM(n_visible=X_train.shape[1], n_hidden=10)
optimizer = torch.optim.Adam(rbm.parameters(), lr=0.01)

for epoch in range(10):
    for batch in DataLoader(X_train_tensor, batch_size=16):
        batch = batch.float()
        h_sample = rbm.forward(batch)
        reconstructed = torch.sigmoid(torch.matmul(h_sample, rbm.W) + rbm.v_bias)
        loss = torch.mean((batch - reconstructed) ** 2)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# Transform data using the trained RBM
X_train_rbm = rbm.forward(X_train_tensor).detach().numpy()
X_test_rbm = rbm.forward(X_test_tensor).detach().numpy()

print("Applying Autoencoder...")
# 3. Autoencoder
class Autoencoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Linear(input_size, hidden_size)
        self.decoder = nn.Linear(hidden_size, input_size)

    def forward(self, x):
```

```
        x = torch.relu(self.encoder(x))
        x = self.decoder(x)
        return x

# Train Autoencoder
autoencoder = Autoencoder(input_size=X_train.shape[1], hidden_size=10)
optimizer = torch.optim.Adam(autoencoder.parameters(), lr=0.01)
criterion = nn.MSELoss()

for epoch in range(10):
    for batch in DataLoader(X_train_tensor, batch_size=16):
        batch = batch.float()
        reconstructed = autoencoder(batch)
        loss = criterion(reconstructed, batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

X_train_auto = autoencoder.encoder(X_train_tensor).detach().numpy()
X_test_auto = autoencoder.encoder(X_test_tensor).detach().numpy()

# ---------------------------------------------------------------------------
# Step 3: Train Classifiers
# ---------------------------------------------------------------------------
datasets = {
    "Original": (X_train, X_test),
    "SOM": (X_train_som, X_test_som),
    "RBM": (X_train_rbm, X_test_rbm),
    "Autoencoder": (X_train_auto, X_test_auto),
```

```
}

results = []

for name, (train_data, test_data) in datasets.items():
    for clf_name, clf in [
        ("XGBoost", xgb.XGBClassifier(use_label_encoder=False, eval_metric="logloss")),
        ("LightGBM", LGBMClassifier()),
        ("CatBoost", CatBoostClassifier(verbose=0)),
    ]:
        start_time = time.time()
        clf.fit(train_data, y_train)
        predictions = clf.predict(test_data)
        accuracy = accuracy_score(y_test, predictions)
        elapsed_time = time.time() - start_time
        results.append((name, clf_name, accuracy, elapsed_time))

# ---------------------------------------------------------------------------
# Step 4: Report Results
# ---------------------------------------------------------------------------
results_df = pd.DataFrame(
    results, columns=["Dataset", "Classifier", "Accuracy", "Time (s)"]
)
print("\n=== Final Results ===")
print(results_df)
```

Result

	Dataset	Classifier	Accuracy	Time (s)
0	Original	XGBoost	0.980952	0.096054
1	Original	LightGBM	0.976190	0.216141
2	Original	CatBoost	0.985714	3.448707
3	SOM	XGBoost	0.966667	0.063982
4	SOM	LightGBM	0.957143	0.027295
5	SOM	CatBoost	0.957143	1.013347
6	RBM	XGBoost	0.976190	0.046851
7	RBM	LightGBM	0.961905	0.062937
8	RBM	CatBoost	0.966667	1.964469
9	Autoencoder	XGBoost	0.966667	0.046875
10	Autoencoder	LightGBM	0.976190	0.068369
11	Autoencoder	CatBoost	0.971429	1.919503

Observations:

1. **Accuracy:**
    - Original Dataset achieved the highest accuracy for all classifiers, with CatBoost performing best at 98.57%.
    - **Dimensionality-Reduced Datasets:**
        - RBM and Autoencoder consistently outperformed SOM in terms of accuracy.
        - Autoencoder-based reduction achieved competitive accuracy, closely matching the original dataset.
2. **Execution Time:**
    - XGBoost and LightGBM demonstrated faster training times compared to CatBoost across all datasets.
    - SOM was the fastest dimensionality reduction technique due to its simplicity but slightly lagged in classification accuracy.
    - Autoencoders and RBMs showed moderate training times, balancing complexity and accuracy effectively.
3. **Dimensionality Reduction Techniques:**
    - **SOM:** Efficient but limited in maintaining feature importance, resulting in lower accuracy compared to RBM and Autoencoder.
    - **RBM:** Achieved higher accuracy with moderately faster training times.

- **Autoencoder:** Delivered a balance between accuracy and training time, making it a suitable choice for classification tasks.

Conclusion:

- For high accuracy and acceptable training time, the **original dataset** with CatBoost performed best, albeit at a higher computational cost.
- Among dimensionality reduction techniques, **Autoencoder** emerged as the most effective, providing a trade-off between accuracy and speed.
- For scenarios prioritizing speed, **LightGBM with SOM** provided the quickest solution, although with slightly lower accuracy.

This experiment highlights the trade-offs between dimensionality reduction methods and classifiers, emphasizing the importance of selecting techniques based on task-specific requirements such as accuracy and computational efficiency.