wine_classifier_pseudo

December 1, 2024

1 Wine Classifier (Using Pseudo labeling)

Now that we've seen how unsupervised learning and supervised learning can be used to classify on the wines dataset. Lets try something different. We will combine these two techniques. We will assume our data to be unlabeled, and use unsupervised learning to generate labels for the data. Now that we have our "labeled" data, we will train a supervised learning model on this data.

After this, we can compare the performance of unsupervised learning, supervised learning, and supervised learning with pseudo-labeling.

[18]:
```python
# import libraries

import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.datasets import load_wine
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import torch
import torch.nn as nn
import torch.optim as optim

device = ("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
```

```
Using device: cuda
```

Similar to what we did for purely unsupervised learning, we will first drop the labels

[6]:
```python
# Load the wine dataset
wine = load_wine(as_frame=True)
wine_df = wine.frame
wine_df.head()

# Drop target column
wine_df_unlabeled = wine_df.drop(['target'], axis=1)
wine_df_unlabeled.head()
```

```
# Normalize features
scaler = StandardScaler()
scaled_wine_df_unlabeled = scaler.fit_transform(wine_df_unlabeled)
```

Since we already know the number of optimal clusters to be 3 from our earlier attempt, we will skip the elbow method.

[11]:
```
X = scaled_wine_df_unlabeled
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X)
print("Cluster centroids")
print(kmeans.cluster_centers_)
print()
print("Sample labels")
print(kmeans.labels_)
```

```
Cluster centroids
[[-0.92607185 -0.39404154 -0.49451676  0.17060184 -0.49171185 -0.07598265
   0.02081257 -0.03353357  0.0582655  -0.90191402  0.46180361  0.27076419
  -0.75384618]
 [ 0.16490746  0.87154706  0.18689833  0.52436746 -0.07547277 -0.97933029
  -1.21524764  0.72606354 -0.77970639  0.94153874 -1.16478865 -1.29241163
  -0.40708796]
 [ 0.83523208 -0.30380968  0.36470604 -0.61019129  0.5775868   0.88523736
   0.97781956 -0.56208965  0.58028658  0.17106348  0.47398365  0.77924711
   1.12518529]]

Sample labels
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 2
 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 0 0 2 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

c:\Users\2004e\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1429:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=1.
  warnings.warn(
```

From our knowledge of the data set, we can map the cluster labels to the corresponding target values.

[12]:
```
 # Assigning clusters to wine targets
label_to_class_map = {0: 1, 1: 2, 2: 0}
```

Now, we can create our own target column based on the labels we have obtained.

```
[16]: pseudo_labeled_wine_df = wine_df_unlabeled.copy()
      pseudo_labeled_wine_df['target'] = pd.Series(kmeans.labels_).map(lambda l:␣
       ↪label_to_class_map[l])
      pseudo_labeled_wine_df.head()
```

```
[16]:    alcohol  malic_acid   ash  alcalinity_of_ash  magnesium  total_phenols  \
      0    14.23        1.71  2.43               15.6      127.0           2.80
      1    13.20        1.78  2.14               11.2      100.0           2.65
      2    13.16        2.36  2.67               18.6      101.0           2.80
      3    14.37        1.95  2.50               16.8      113.0           3.85
      4    13.24        2.59  2.87               21.0      118.0           2.80

         flavanoids  nonflavanoid_phenols  proanthocyanins  color_intensity   hue  \
      0        3.06                  0.28             2.29             5.64  1.04
      1        2.76                  0.26             1.28             4.38  1.05
      2        3.24                  0.30             2.81             5.68  1.03
      3        3.49                  0.24             2.18             7.80  0.86
      4        2.69                  0.39             1.82             4.32  1.04

         od280/od315_of_diluted_wines  proline  target
      0                          3.92   1065.0       0
      1                          3.40   1050.0       0
      2                          3.17   1185.0       0
      3                          3.45   1480.0       0
      4                          2.93    735.0       0
```

Now that we have our "labeled" data, we can perform supervised learning using neural networks on it. The method will be the same as before.

```
[33]: X = pseudo_labeled_wine_df.loc[:, :'proline'].values
      y = pseudo_labeled_wine_df.target.values
      y_true = wine.target.values

      # Normalize features
      scaler = StandardScaler()
      X_scaled = scaler.fit_transform(X)



      # Split into training and test sets
      X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,␣
       ↪random_state=42)

      # We want to test against the true labels

      _, y_test = train_test_split(y_true, test_size=0.2, random_state=42)

      # Convert to pytorch tensors
```

```
X_train = torch.FloatTensor(X_train)
X_test = torch.FloatTensor(X_test)
y_train = torch.LongTensor(y_train)
y_test = torch.LongTensor(y_test)

# Define model class
class WineClassifier(nn.Module):
    def __init__(self, input_size=13, hidden_size=8, num_classes=3):
        super(WineClassifier, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        return x

model = WineClassifier()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Alright, let us now train our model.

[31]:
```
# Define training loop

num_epochs = 100
train_losses = []
train_accuracies = []
test_accuracies = []

for epoch in range(num_epochs):
    # Training
    model.train()                          # Kind of put the model into␣
 ↪training mode
    optimizer.zero_grad()                  # Reset gradient values for␣
 ↪parameters
    output = model(X_train)                # Do a forward pass
    loss = criterion(output, y_train)      # Compute loss
    loss.backward()                        # Do a backward pass
    optimizer.step()                       # Update weights

    # Calculate training accuracy
    _, y_pred = torch.max(output, 1)
    train_accuracy = (y_pred == y_train).sum().item() / y_train.size(0)
    # Calculate test accuracy
    model.eval()
    with torch.no_grad():
        test_outputs = model(X_test)
        _, test_predicted = torch.max(test_outputs.data, 1)
        test_accuracy = (test_predicted == y_test).sum().item() / y_test.size(0)

    # Store metrics
    train_losses.append(loss.item())
    train_accuracies.append(train_accuracy)
    test_accuracies.append(test_accuracy)

    # Print progress every 10 epochs
    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
        print(f'Train Accuracy: {train_accuracy:.4f}, Test Accuracy:␣
↪{test_accuracy:.4f}')
```

```
Epoch [10/100], Loss: 0.6007
Train Accuracy: 0.9437, Test Accuracy: 0.9167
Epoch [20/100], Loss: 0.2630
Train Accuracy: 0.9718, Test Accuracy: 0.9444
Epoch [30/100], Loss: 0.1199
Train Accuracy: 0.9930, Test Accuracy: 1.0000
Epoch [40/100], Loss: 0.0664
Train Accuracy: 0.9930, Test Accuracy: 1.0000
Epoch [50/100], Loss: 0.0439
Train Accuracy: 0.9930, Test Accuracy: 1.0000
Epoch [60/100], Loss: 0.0330
Train Accuracy: 0.9930, Test Accuracy: 1.0000
Epoch [70/100], Loss: 0.0268
Train Accuracy: 0.9930, Test Accuracy: 1.0000
Epoch [80/100], Loss: 0.0226
Train Accuracy: 0.9930, Test Accuracy: 1.0000
Epoch [90/100], Loss: 0.0196
Train Accuracy: 0.9930, Test Accuracy: 1.0000
Epoch [100/100], Loss: 0.0172
Train Accuracy: 0.9930, Test Accuracy: 1.0000
```

Lastly, lets get some metrics to compare our new model based on pseudo-labeled data, to our previous models.

[34]:
```
# Final evaluation
model.eval()
with torch.no_grad():
    final_test_outputs = model(X_test)
    _, final_predicted = torch.max(final_test_outputs.data, 1)
    final_accuracy = (final_predicted == y_test).sum().item() / y_test.size(0)
    print(f'\nFinal Test Accuracy: {final_accuracy:.4f}')
```

```
Final Test Accuracy: 0.6111
```

We have obtained results that suggest our new model accuracy is better than random classification (0.33 for 3 classes), but worse than a purely supervised learning approach. There are also a few drawbacks when it comes to generalizing this approach. Our current dataset was quite small, and thus we cannot say the same will work for much larger data sets. Additionally, we had a low number of features, and our data happened to be separated well enough for clustering to work.

The main potential I see for this approach is when working with data that does not have "true" labels. When the labels are subjective, such as emotion, mood etc. These can be derived from natural groupings of the data, and a model can be trained on these labels to predict the same for unseen data. In such cases, the decision boundaries are expected to be fuzzy, and imperfections can be forgiven more easily.