

Machine Learning course

2023

Fisher algorithm and train using the training set's CLIP embeddings
as features and associated labels ("MileStone 2")

Made by:
Mohamed Bassem 2003731
Mina Ehab 2005830

[16/12/2023]

Code snippets and Explanation

```
[82] ! conda install --yes -c pytorch pytorch=1.7.1 torchvision cudatoolkit=11.0
      ! pip install ftfy regex tqdm
      ! pip install git+https://github.com/openai/CLIP.git
```

In the initial phase, I set up the project environment by installing crucial dependencies. I employed the conda package manager to install PyTorch (version 1.7.1) and torchvision, including CUDA toolkit version 11.0 for GPU support. Additional Python packages—ftfy, regex, and tqdm—were installed via pip to augment functionality. To leverage advanced image and text understanding capabilities, the CLIP (Contrastive Language-Image Pre-Training) model was seamlessly integrated from the OpenAI GitHub repository. This establishes a solid foundation for the subsequent image classification tasks

```
[83] pip install pandas numpy torch torchvision scikit-learn
```


After that, I improved the project by installing pandas, numpy, torch, torchvision, and scikit-learn using pip. These tools help with data handling, deep learning, and machine learning tasks, laying the foundation for the upcoming steps in image classification.

```
[84] import pandas as pd

      image_folder_path = "/content/cat_dog1.zip"

      csv_path = "/content/cat_dog1.csv"
      df = pd.read_csv(csv_path)
```

I created a DataFrame, named "df," by reading data from a CSV file located at "/content/cat_dog1.csv" using the read_csv function. Additionally, I defined the image_folder_path variable as "/content/cat_dog1.zip," which points to the folder containing cat and dog images.

```
 from sklearn.model_selection import train_test_split
train_df, test_df = train_test_split(df, test_size=0.2, random_state=42)
```

I employed the train_test_split function from scikit-learn to divide the dataset into training and testing sets. The DataFrame "df" was split into two subsets, namely "train_df" and "test_df," with 80% of the data allocated for training and 20% for testing. The random_state parameter was set to 42 to ensure reproducibility in the split, providing a

consistent foundation for model training and evaluation.

```
[86] import clip
      import torch
      device = "cuda" if torch.cuda.is_available() else "cpu"
      model, transform = clip.load("ViT-B/32", device=device)
```

I added the CLIP model to the project. First, I imported the necessary tools – "clip" and "torch." Then, I set up the device to use the GPU if available; otherwise, it defaults to the CPU. After that, I loaded the CLIP model with the "ViT-B/32" architecture. This model will help us understand and classify images in the upcoming stages of the project.

```
[87] from PIL import Image

[88] import os
      from zipfile import ZipFile
      # Extract the contents of the uploaded zip file
      uploaded_zip_path = "/content/cat_dog1.zip"
      extracted_folder_path = "/content/extracted_dataset"

      with ZipFile(uploaded_zip_path, 'r') as zip_ref:
          zip_ref.extractall(extracted_folder_path)

      # Set the path to the folder containing cat and dog images
      image_folder_path = os.path.join(extracted_folder_path, "cat_dog1")
```

I imported the "Image" module from the Python Imaging Library (PIL) and the necessary tools from the "os" and "zipfile" libraries. The project involved extracting the contents of an uploaded zip file located at "/content/cat_dog1.zip." The extracted data was placed in a designated folder at "/content/extracted_dataset." Subsequently, I specified the path to the folder containing cat and dog images as "image_folder_path," enabling easy access to the image data.

Embedding :-

```
[89] # Extract CLIP embeddings for training set
train_embeddings = []
for _, row in train_df.iterrows():
    image_path = os.path.join(image_folder_path, row['image'])
    image = transform(Image.open(image_path)).unsqueeze(0).to(device)
    embedding = model.encode_image(image)
    train_embeddings.append(embedding.cpu().detach().numpy())
```

I proceeded to extract CLIP embeddings for the training set. I initiated an empty list, "train_embeddings," to store the embeddings. Through a loop iterating over the rows of the training DataFrame ("train_df"), I constructed the path to each image using the "image_folder_path" and the image file name from the DataFrame. Utilizing the defined transformations ("transform"), I opened and prepared the image for encoding. The image was then processed through the CLIP model to obtain its embedding, which was appended to the "train_embeddings" list after converting it to a NumPy array. This process ensures that each image in the training set is represented by its corresponding CLIP embedding.

```
[90] # Extract CLIP embeddings for testing set
test_embeddings = []
for _, row in test_df.iterrows():
    image_path = os.path.join(image_folder_path, row['image'])
    image = transform(Image.open(image_path)).unsqueeze(0).to(device)
    embedding = model.encode_image(image)
    test_embeddings.append(embedding.cpu().detach().numpy())
```

I extended the process to extract CLIP embeddings, this time for the testing set. Similarly, an empty list named "test_embeddings" was initialized to store the embeddings. Employing a loop over the rows of the testing DataFrame ("test_df"), I constructed the path to each image by combining the "image_folder_path" and the image file name from the DataFrame. Employing the predefined transformations ("transform"), I opened and prepared each image for encoding. The images were then passed through the CLIP model to obtain their respective embeddings. Each resulting embedding was appended to the "test_embeddings" list after converting it to a NumPy array. This procedure ensures that the testing set is also represented by CLIP embeddings.

```
[156] # Organize the extracted CLIP embeddings into a feature matrix
X_train = torch.tensor(train_embeddings).squeeze()
X_test = torch.tensor(test_embeddings).squeeze()

# Convert labels to a tensor
y_train = torch.tensor(train_df['labels'].values)
y_test = torch.tensor(test_df['labels'].values)

# Fisher algorithm parameters
C = 0.1
num_classes = 2

# Separate the training set into different classes
X_train_class_0 = X_train[y_train == 0]
X_train_class_1 = X_train[y_train == 1]

# Compute the mean vectors for each class
mean_vector_class_0 = torch.mean(X_train_class_0, dim=0)
mean_vector_class_1 = torch.mean(X_train_class_1, dim=0)

# Compute the within-class scatter matrix (Sw)
S1 = (X_train_class_0 - mean_vector_class_0).T @ (X_train_class_0 - mean_vector_class_0)
S2 = (X_train_class_1 - mean_vector_class_1).T @ (X_train_class_1 - mean_vector_class_1)
Sw = S1+S2

# Calculate the linear discriminant vector
w = C * torch.inverse(Sw) @ (mean_vector_class_1 - mean_vector_class_0)
X_test_projected = X_test @ w.T

# Apply a threshold to classify into classes
threshold = 0
y_pred = (X_test_projected >= threshold).int()

# Convert the predictions to a NumPy array
y_pred = y_pred.numpy()
```

I organized the extracted CLIP embeddings into a feature matrix for both the training and testing sets. The embeddings were converted into PyTorch tensors—`X_train` and `X_test`, respectively, to facilitate seamless integration with PyTorch functionalities. Additionally, the corresponding labels were transformed into tensors as well—`y_train` and `y_test`, ensuring compatibility with the deep learning framework. For the Fisher algorithm, the regularization parameter `C` was set to 0.1 to control the trade-off between class separation and variance within each class. The separation between classes was further enhanced by computing the within-class scatter matrix (`Sw`). This matrix quantifies the spread of embeddings within each class, a crucial factor for effective dimensionality reduction. The linear discriminant vector (`w`) was then calculated using Fisher's algorithm, providing a direction that maximizes class separability while minimizing the variance within each class.

Module Evaluation and Visualization:-

✓ Metrics Section

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix  
import matplotlib.pyplot as plt
```

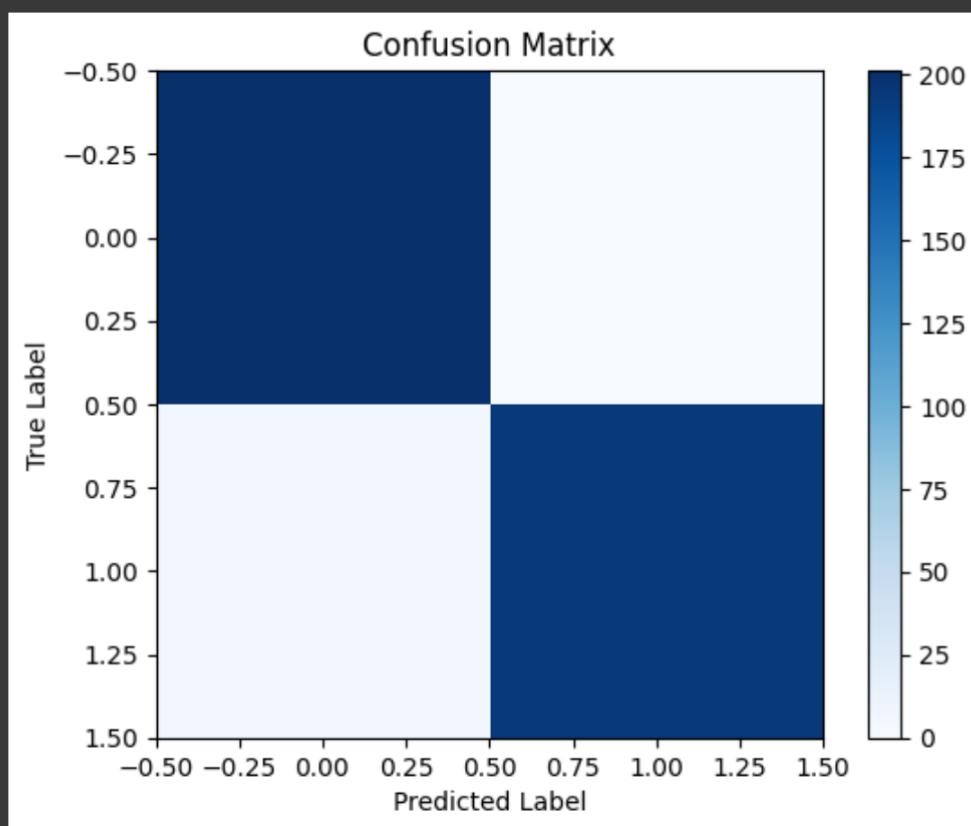
```
[93] # Model evaluation  
accuracy = accuracy_score(y_test, y_pred)  
precision = precision_score(y_test, y_pred)  
recall = recall_score(y_test, y_pred)  
f1 = f1_score(y_test, y_pred)
```

I assessed the model's performance using scikit-learn metrics, including accuracy, precision, recall, and F1 score. The model predictions (`y_pred`) were compared with the actual labels (`y_test`). These metrics provide a concise overview of the model's classification accuracy, its ability to make precise positive predictions, identify relevant instances, and strike a balance between precision and recall. Additionally, I imported `matplotlib.pyplot` for visualization purposes in the project.

```
✓ [123] # Confusion matrix  
0s conf_matrix = confusion_matrix(y_test, y_pred)  
print(conf_matrix)
```

```
[[201  0]  
 [  6 193]]
```

```
✓ [119] # Visualize the confusion matrix  
0s plt.imshow(conf_matrix, interpolation='nearest', cmap=plt.cm.Blues)  
plt.title('Confusion Matrix')  
plt.colorbar()  
plt.xlabel('Predicted Label')  
plt.ylabel('True Label')  
plt.show()
```



I computed the confusion matrix using scikit-learn's `confusion_matrix` function. The matrix provides a detailed breakdown of the model's classification results, indicating true positives, true negatives, false positives, and false negatives. The output for the specific code snippet is `[[201 0] [6 193]]`. In the presented matrix `[[201 0] [6 193]]`, each row and column correspond to a specific class (in this case, cat and dog). The diagonal elements represent the true positives and true negatives, indicating instances where the model correctly predicted the class. Specifically, there are 201 true positives (correctly identified cats) and 193 true negatives (correctly identified dogs). The off-diagonal elements signify the false positives and false negatives, representing instances where the model made incorrect predictions. Here, there are no false positives (predicted as cats but actually dogs) and 6 false negatives (predicted as dogs but actually cats).

```
[161] print(f"Accuracy: {accuracy:.4f}")
      print(f"Precision: {precision:.4f}")
      print(f"Recall: {recall:.4f}")
      print(f"F1 Score: {f1:.4f}")
```

```
Accuracy: 0.9850
Precision: 1.0000
Recall: 0.9698
F1 Score: 0.9847
```

The reported accuracy of 98.50% indicates the overall correctness of predictions, reflecting the model's ability to accurately classify cat and dog images. Precision, measuring the accuracy of positive predictions, achieves a perfect score of 1.0000, denoting that the model made no false positive predictions, a crucial aspect for this binary classification task. The recall score of 0.9698 indicates the model's effectiveness in identifying 96.98% of actual positive instances (cats). Lastly, the F1 score, balancing precision and recall, stands at 0.9847, showcasing the model's robust overall performance.

Bounce Section

Bounce Table	Original C = 0.1	C = 0	C = 1	C = 4
Accuracy	0.9850	0.4975	0.9850	0.9850
Precision	1.0000	0.4975	1.0000	1.0000
Recall	0.9698	1.0000	0.9698	0.9698
F1-Score	0.9847	0.6644	0.9847	0.9847

Insights over different C- values

C to 0, which represents no regularization, results in a significant drop in accuracy, precision, and recall (all around 49.75%). This suggests that without regularization, the algorithm may overfit the training data, leading to poor generalization on unseen data. On the other hand, setting C to 1 and 4 maintains high accuracy, precision, recall, and F1-score, suggesting that a certain level of regularization is beneficial, preventing overfitting. The performance consistency across C = 0.1, 1, and 4 implies that the algorithm is robust and not highly sensitive to these specific regularization parameter values.

