cosc325_team_22_final_code

May 1, 2025

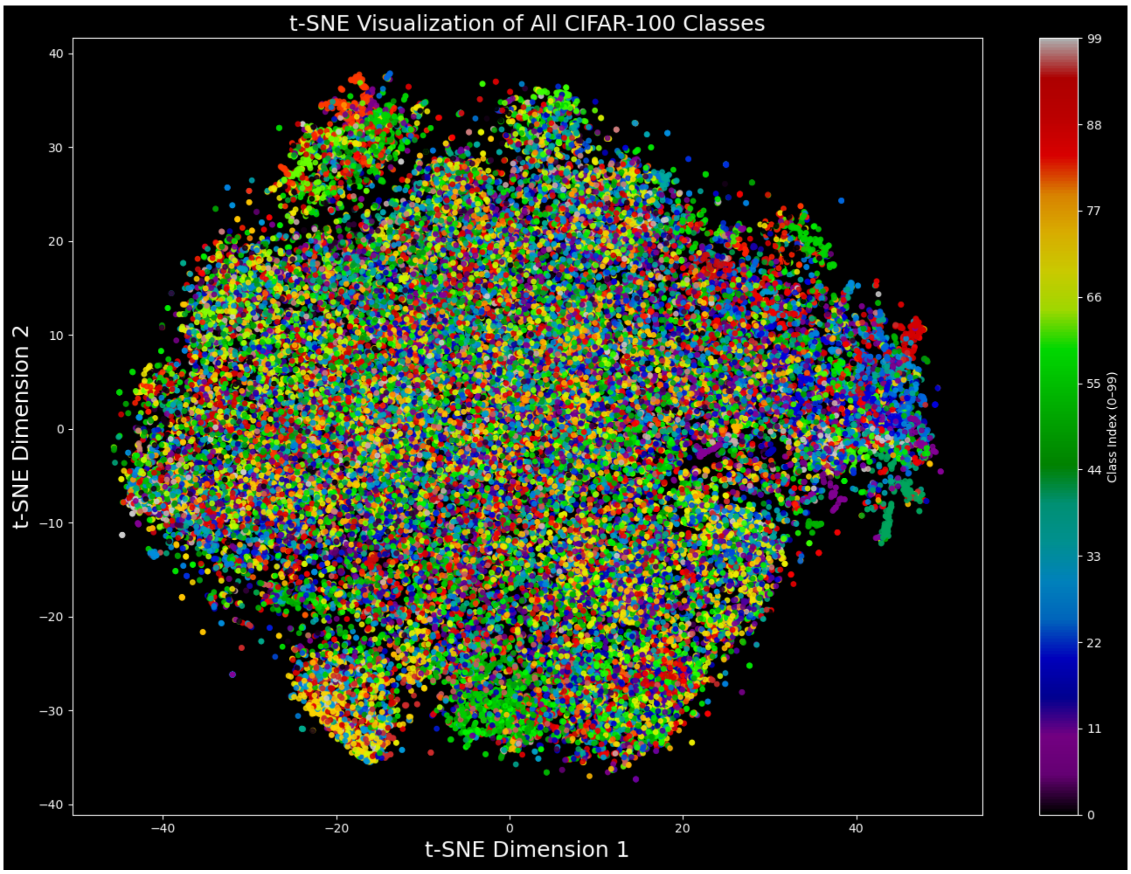
[1]:
```python
"""
Pre-requisites
"""

# Import required libraries
import cv2
from IPython.display import display
from PIL import Image
import PIL
from glob import glob

import matplotlib.image as mpimg
import matplotlib.pyplot as plt

import numpy as np
import os
import pandas as pd
import pickle
import random
import seaborn as sns
from scipy import stats

from skimage.feature import hog
from sklearn.manifold import TSNE
from sklearn.metrics import accuracy_score, confusion_matrix, recall_score,␣
↪ConfusionMatrixDisplay
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.svm import SVC

import time
import torchvision
from torchvision import transforms
from tqdm.notebook import tqdm
```

```python
import xgboost as xgb

# Set seed
SEED = 42
random.seed(SEED)
os.environ['PYTHONHASHSEED'] = str(SEED)
np.random.seed(SEED)

# Load CIFAR-100
cifar100 = torchvision.datasets.CIFAR100(root='./data', train=True,␣
↪download=True)
```

[2]:
```python
"""
Visualize CIFAR-100 Dataset with t-SNE
"""

# Extract images and labels
images, labels = cifar100.data, np.array(cifar100.targets)

# Normalize images to [0, 1]
images = images.astype(np.float32) / 255.0

# Reshape images
images_flat = images.reshape(images.shape[0], -1)

# Scale features
scaler = StandardScaler()
images_scaled = scaler.fit_transform(images_flat)

print("Visualizing t-SNE for all CIFAR-100 classes...Very time consuming")
tsne = TSNE(n_components=2, perplexity=30, learning_rate=200, max_iter=1000,␣
↪random_state=SEED)
tsne_results = tsne.fit_transform(images_scaled)

plt.style.use('dark_background')
colors_100 = plt.get_cmap('nipy_spectral')

# Create the scatter plot
plt.figure(figsize=(14, 10))
scatter = plt.scatter(tsne_results[:, 0], tsne_results[:, 1], c=labels,␣
↪cmap=colors_100, s=15, alpha=0.85)

# Colorbar
cbar = plt.colorbar(scatter, ticks=np.linspace(0, 99, 10))
cbar.set_label('Class Index (0-99)', color='white')
cbar.ax.yaxis.set_tick_params(color='white')
plt.setp(plt.getp(cbar.ax.axes, 'yticklabels'), color='white')
```

```
# Labels
plt.title("t-SNE Visualization of All CIFAR-100 Classes", color='white',␣
↪fontsize=18)
plt.xlabel("t-SNE Dimension 1", color='white', fontsize=18)
plt.ylabel("t-SNE Dimension 2", color='white', fontsize=18)
plt.tick_params(colors='white')
plt.tight_layout()
plt.show()

plt.style.use('default')
```

Visualizing t-SNE for all CIFAR-100 classes…Very time consuming



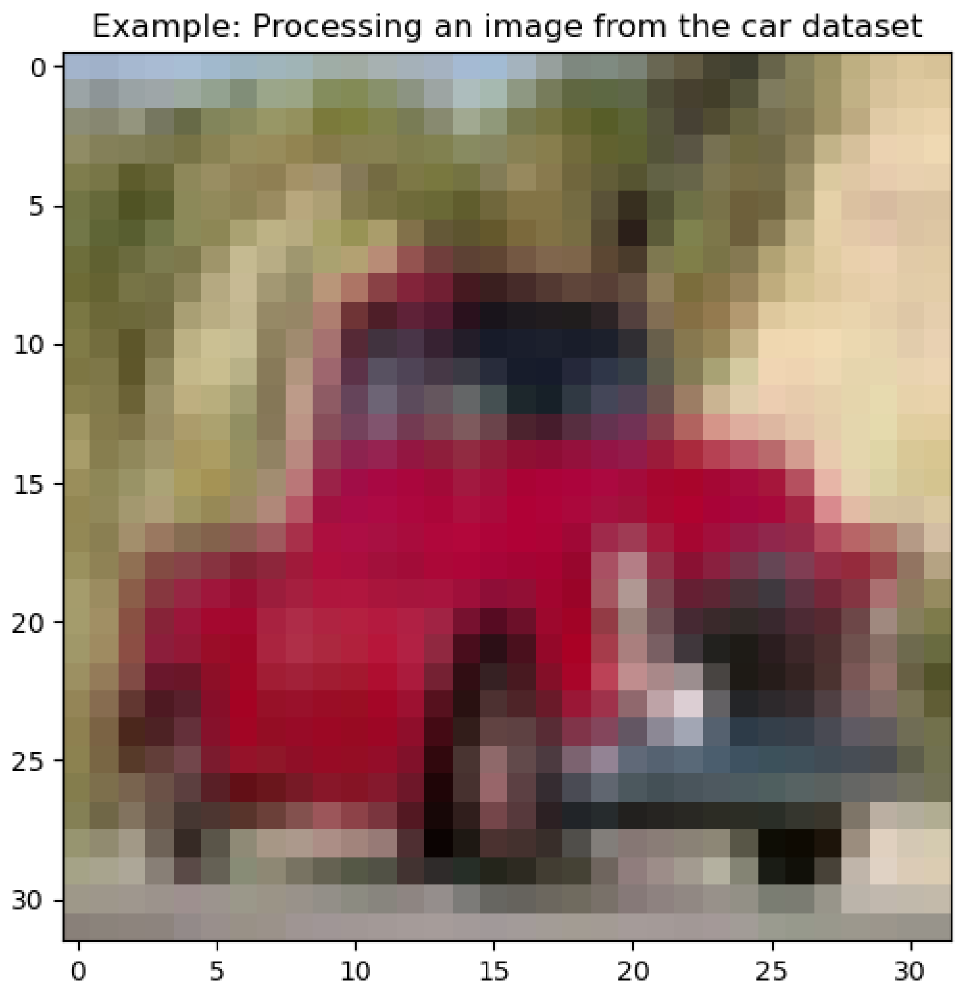
[3]:
```
"""
Pre-processing: Choosing positive and negative samples
"""


# Identify car-related class names
```

```
car_class_names = ['bicycle', 'bus', 'motorcycle', 'pickup_truck', 'streetcar', 'tractor']
car_indices = [cifar100.class_to_idx[name] for name in car_class_names]

# Output directory for car images
output_car_dir = './cifar100_cars'
os.makedirs(output_car_dir, exist_ok=True)

# Save only car images to directory
count = 0
for i in range(len(cifar100)):
    img, label = cifar100[i]
    if label in car_indices:
        img.save(os.path.join(output_car_dir, f"car_{count}.png"))
        count += 1

# Identify non-car class names
non_car_class_names = [class_name for class_name in cifar100.classes if class_name not in car_class_names]
non_car_indices = [cifar100.class_to_idx[name] for name in non_car_class_names]

# Output directory for non-car images
output_non_car_dir = './cifar100_non_cars'
os.makedirs(output_non_car_dir, exist_ok=True)

# Save 3000 random non-car images to directory
num_images = 3000
count_non_car = 0
sampled_images = []
non_car_indices_list = [i for i in range(len(cifar100)) if cifar100.targets[i] in non_car_indices]
random_indices = random.sample(non_car_indices_list, num_images)
for idx in random_indices:
    img, label = cifar100[idx]
    img.save(os.path.join(output_non_car_dir, f"non_car{count_non_car}.png"))
    count_non_car += 1

print(f"Saved {count} car images to {output_car_dir}")
print(f"Saved {count_non_car} non-car images to {output_non_car_dir}\n")

# Get file paths of saved car and non-car images
car_paths = glob(output_car_dir + "/*.png")
non_car_paths = glob(output_non_car_dir + "/*.png")

# Display example image from the car class
example_image = np.asarray(PIL.Image.open(car_paths[894]))
fig = plt.figure(figsize=(12, 6))
```

```
plt.title("Example: Processing an image from the car dataset")
plt.imshow(example_image)
example_image.shape
```

```
Saved 3000 car images to ./cifar100_cars
Saved 3000 non-car images to ./cifar100_non_cars
```

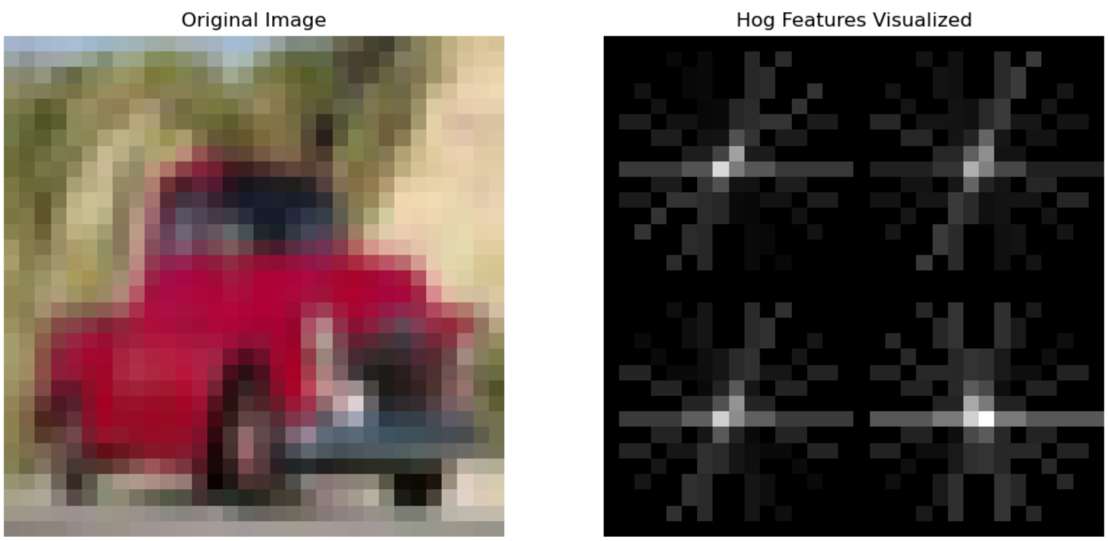
[3]: (32, 32, 3)



[4]:
```
"""
Pre-processing: HOG Visualization
"""

# Use HOG to focus on the shape of an image
```

```python
hog_features, visualized = hog(image=example_image, orientations=9,␣
↪pixels_per_cell=(16, 16), cells_per_block=(2, 2), visualize=True,␣
↪channel_axis=-1)

# HOG Visualization Example
fig = plt.figure(figsize=(12, 6))       # Original Image
fig.add_subplot(1, 2, 1)
plt.imshow(example_image)
plt.title("Original Image")
plt.axis("off")

fig.add_subplot(1, 2, 2)                # HOG Visualized Image
plt.imshow(visualized, cmap="gray")
plt.title("Hog Features Visualized")
plt.axis("off")
plt.show()

# hog_features is a vector
hog_features.shape

pos_images = []
neg_images = []

# Labels: 1 for car, 0 for non-car
pos_labels = np.ones(len(car_paths))
neg_labels = np.zeros(len(non_car_paths))

# Start timer to measure processing time
start = time.time()
print("\nBegin extracting HOG features for positive and negative images (This␣
↪will take a while)")

# Extract HOG features for positive images (cars)
for car_path in car_paths:
    img = np.asarray(PIL.Image.open(car_path))
    # We don't need RGB channels
    img = cv2.cvtColor(cv2.resize(img, (96,64)), cv2.COLOR_RGB2GRAY)
    img = hog(img, orientations=9, pixels_per_cell=(16, 16),cells_per_block=(2,␣
↪2))
    pos_images.append(img)

# Extract HOG features for negative images (non-cars)
for non_car_path in non_car_paths:
    img = np.asarray(PIL.Image.open(non_car_path))
    img = cv2.cvtColor(cv2.resize(img, (96,64)), cv2.COLOR_RGB2GRAY)
    img = hog(img, orientations=9, pixels_per_cell=(16, 16),cells_per_block=(2,␣
↪2))
```

```
    neg_images.append(img)

# Stack positive and negative images
x = np.asarray(pos_images + neg_images)
y = np.asarray(list(pos_labels) + list(neg_labels))

processTime = round(time.time()-start, 2)
print(f"Reading images and extracting features has taken {processTime} seconds")

print("Shape of image set", x.shape)
print("Shape of labels", y.shape)
```




```
Begin extracting HOG features for positive and negative images (This will take a
while)
Reading images and extracting features has taken 62.77 seconds
Shape of image set (6000, 540)
Shape of labels (6000,)
```

[5]:

```
"""
Split data into train and test
"""

print("\nSplit data into training and test sets")
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,␣
 ↪random_state=SEED)
print(f"x_train: {x_train.shape}")
print(f"x_test: {x_test.shape}")
print(f"y_train: {y_train.shape}")
```

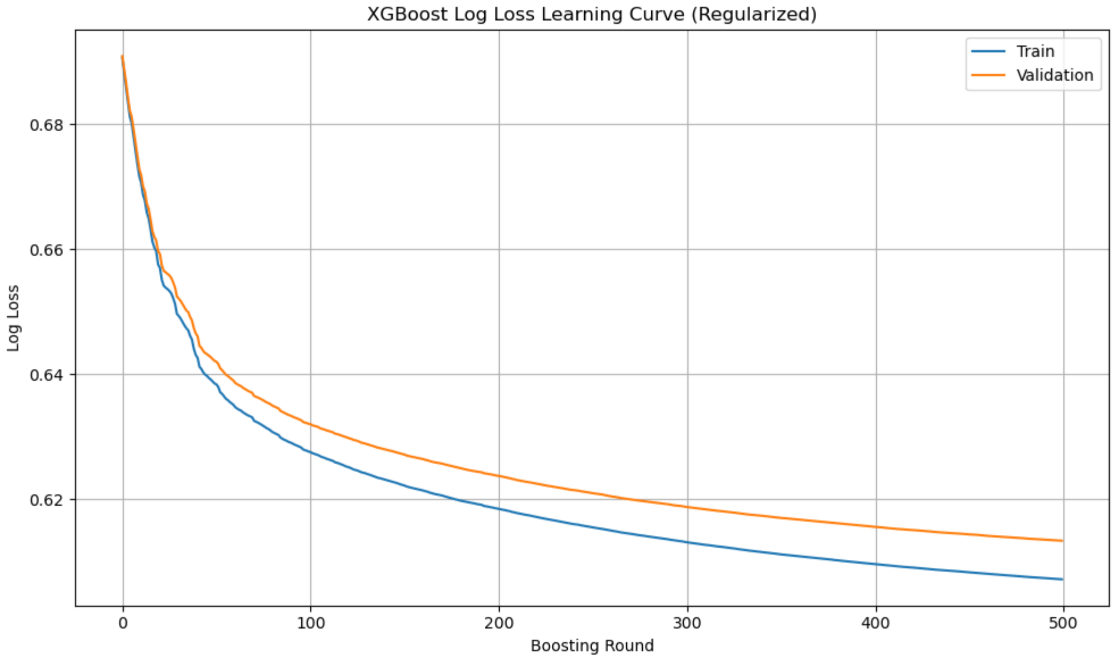
```
print(f"y_test: {y_test.shape}")
```

```
Split data into training and test sets
x_train: (4800, 540)
x_test: (1200, 540)
y_train: (4800,)
y_test: (1200,)
```

[6]:
```
"""
Create XGBoost Model
"""

# Create DMatrix objects for XGBoost
train_data = xgb.DMatrix(x_train, label=y_train)
test_data = xgb.DMatrix(x_test, label=y_test)

# Define tuned XGBoost parameters to reduce overfitting
params = {
    'objective': 'binary:logistic',
    'eval_metric': 'logloss',
    'eta': 0.025,
    'max_depth': 2,
    'min_child_weight': 20,
    'subsample': 0.5,
    'colsample_bytree': 0.5,
    'lambda': 5.0,          # L2 regularization
    'alpha': 2.0,           # L1 regularization
    'seed': SEED,
    'max_delta_step': 1,
    'gamma': 1.0,
    'booster': 'dart',
    'rate_drop': 0.1
}

# Train the model with early stopping
print("\nTraining XGBoost model")
evals = [(train_data, 'train'), (test_data, 'eval')]
eval_result = {}
model = xgb.train(params, train_data, num_boost_round=500, evals=evals,
 ↪early_stopping_rounds=30, evals_result=eval_result, verbose_eval=False)

# Predict probabilities
y_pred_prob = model.predict(test_data)

# Find best threshold: accuracy >= 70% & best recall
best_threshold = 0.0
```

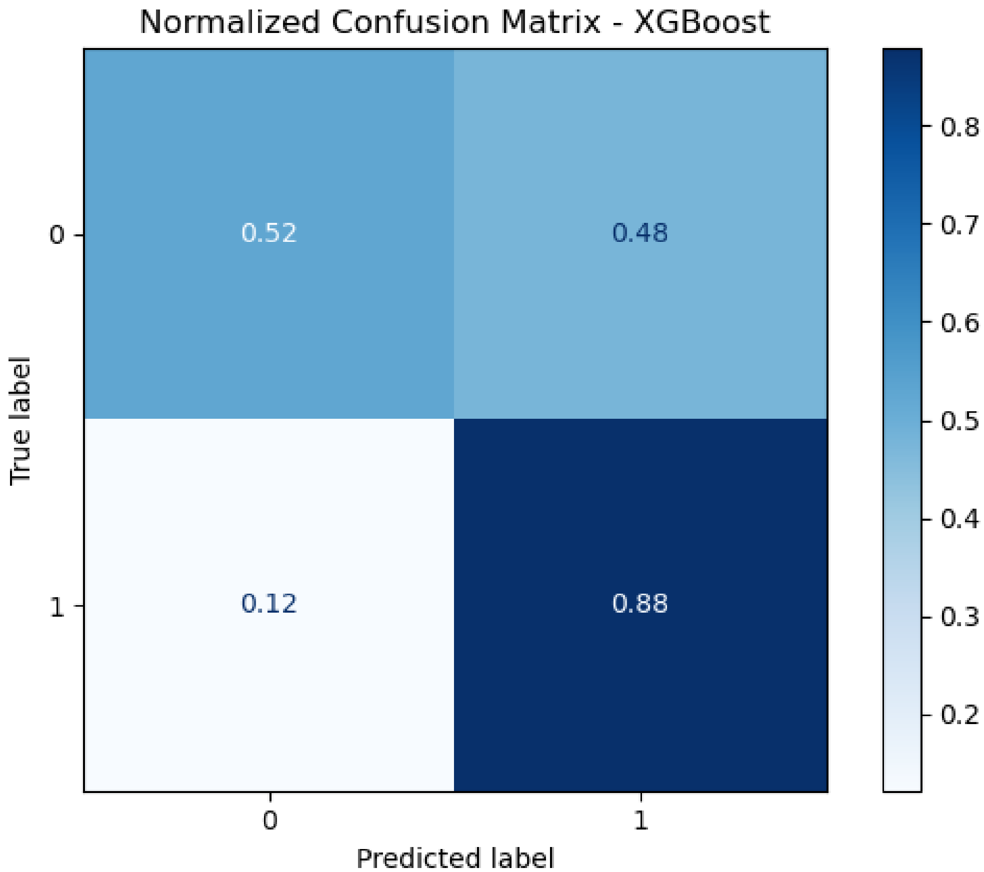
```python
best_recall = 0.0
for threshold in np.arange(0.1, 0.9, 0.01):
    y_pred_temp = (y_pred_prob > threshold).astype(int)
    acc = accuracy_score(y_test, y_pred_temp)
    rec = recall_score(y_test, y_pred_temp)
    if acc >= 0.70 and rec > best_recall:
        best_threshold = threshold
        best_recall = rec

# Predict using best threshold
y_pred = (y_pred_prob > best_threshold).astype(int)

# Evaluate
accuracy = accuracy_score(y_test, y_pred)
print("\nXGBoost Accuracy: {:.2f}%".format(accuracy * 100))
print("Recall: {:.2f}".format(best_recall))
print("Best Threshold: {:.2f}".format(best_threshold))

# Plot learning curves
epochs = len(eval_result['train']['logloss'])
x_axis = range(epochs)

plt.figure(figsize=(10, 6))
plt.plot(x_axis, eval_result['train']['logloss'], label='Train')
plt.plot(x_axis, eval_result['eval']['logloss'], label='Validation')
plt.xlabel('Boosting Round')
plt.ylabel('Log Loss')
plt.title('XGBoost Log Loss Learning Curve (Regularized)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Training XGBoost model

XGBoost Accuracy: 70.50%
Recall: 0.88
Best Threshold: 0.47
```



[7]:
```python
"""
Plot Confusion Matrix
"""

print("\nPlotting Confusion Matrix")
# Generate and display normalized confusion matrix
cm = confusion_matrix(y_test, y_pred, normalize='true')
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(cmap=plt.cm.Blues)
plt.title('Normalized Confusion Matrix - XGBoost')
plt.grid(False)
plt.tight_layout()
plt.show()
```

```
Plotting Confusion Matrix
```



[9]:
```python
"""
5-Fold Cross-Validation
"""

print("\nApplying 5-Fold Validation...Very time consuming")
# Reconstruct full dataset from training and test splits
x_data = np.concatenate((x_train, x_test), axis=0)
y_data = np.concatenate((y_train, y_test), axis=0)

# Set up 5-fold stratified cross-validation
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=SEED)
accuracies = []
recalls = []

for fold, (train_index, test_index) in enumerate(kf.split(x_data, y_data)):
    X_train_fold, X_test_fold = x_data[train_index], x_data[test_index]
    y_train_fold, y_test_fold = y_data[train_index], y_data[test_index]

    dtrain = xgb.DMatrix(X_train_fold, label=y_train_fold)
```

```
    dtest = xgb.DMatrix(X_test_fold, label=y_test_fold)

    eval_result = {}
    model = xgb.train(params, dtrain, num_boost_round=500, evals=[(dtrain,␣
↪'train'), (dtest, 'eval')],
                      early_stopping_rounds=30, evals_result=eval_result,␣
↪verbose_eval=False)

    y_pred_prob = model.predict(dtest)

    # Tune threshold: maximize recall with accuracy >= 70%
    best_threshold = 0.0
    best_recall = 0.0
    for threshold in np.arange(0.1, 0.9, 0.01):
        y_pred_temp = (y_pred_prob > threshold).astype(int)
        acc = accuracy_score(y_test_fold, y_pred_temp)
        rec = recall_score(y_test_fold, y_pred_temp)
        if acc >= 0.70 and rec > best_recall:
            best_threshold = threshold
            best_recall = rec

    # Final prediction with best threshold
    y_pred = (y_pred_prob > best_threshold).astype(int)

    accuracies.append(accuracy_score(y_test_fold, y_pred))
    recalls.append(recall_score(y_test_fold, y_pred))

# Summariy Stats
mean_acc = np.mean(accuracies)
mean_rec = np.mean(recalls)
std_err = stats.sem(accuracies)
conf_int = stats.t.interval(0.95, len(accuracies)-1, loc=mean_acc,␣
↪scale=std_err)

print("\nCross-Validated Results (5 Folds):")
print(f"Average Accuracy: {mean_acc * 100:.2f}%")
print(f"95% Confidence Interval for Accuracy: ({conf_int[0] * 100:.2f}%,␣
↪{conf_int[1] * 100:.2f}%)")
print(f"Average Recall: {mean_rec * 100:.2f}%")
```

```
Applying 5-Fold Validation…Very time consuming

Cross-Validated Results (5 Folds):
Average Accuracy: 70.48%
95% Confidence Interval for Accuracy: (69.96%, 71.01%)
Average Recall: 79.90%
```