

Report

February 12, 2026

1 Giovanni Billo - Computer Vision and Pattern Recognition Report

1.1 Project 2: Bag of words classifier

This project consists of the implementation and evaluation of an image classification pipeline based on the Bag of (Visual) Words (BoVW) framework (Csurka et al., 2004[]). Images are represented through histograms of local visual descriptors, which are then used as input features for different classifiers.

Several encoding strategies and classification models are explored and compared systematically.

2 Data

The dataset used in this project is the scene classification dataset introduced by Lazebnik et al., [2006]. It contains 15 scene categories with 1500 images in the training set and 2985 images in the test set.

3 Preprocessing

Initial experiments were conducted on a reduced subset of the data (500 training images and 100 test images) due to hardware and time constraints.

```
[1]: # 1. build a visual vocabulary:

def sample_from_dict(d, sample=10):
    keys = random.sample(list(d), sample)
    values = [d[k] for k in keys]
    return dict(zip(keys, values))

import os
import cv2 as cv
import numpy as np
import random
import matplotlib.pyplot as plt

def extract_label(img_path):
    # .../train/<label>/<file>.jpg  -> <label>
```

```

    return os.path.basename(os.path.dirname(img_path))

def load_images_from_folder(folder):
    # returns: dict[path] = {"img": img, "label": label}
    data = {}
    for label in os.listdir(folder):
        folder_path = os.path.join(folder, label)
        if not os.path.isdir(folder_path):
            continue

        for filename in os.listdir(folder_path):
            img_path = os.path.join(folder_path, filename)
            img = cv.imread(img_path)
            if img is None:
                continue

            data[img_path] = {"img": img, "label": label}

    return data

def sample_data(data, sample=10, seed=None):
    if seed is not None:
        random.seed(seed)

    keys = list(data.keys())
    if sample > len(keys):
        raise ValueError(f"sample={sample} > dataset size={len(keys)}")

    sampled_keys = random.sample(keys, sample)

    # Build aligned outputs
    sampled_images = {k: data[k]["img"] for k in sampled_keys}
    sampled_labels = [data[k]["label"] for k in sampled_keys]

    return sampled_images, sampled_labels, sampled_keys

# =====
# Example usage
# =====
base_drive_path = "data"
train_folder = os.path.join(base_drive_path, "test")
test_folder = os.path.join(base_drive_path, "train")

train_data = load_images_from_folder(train_folder)
test_data = load_images_from_folder(test_folder)

```

```

print(f"Loaded {len(train_data)} training images.")
print(f"Loaded {len(test_data)} testing images.")

# Sample 15 training images + aligned labels
train_images, train_labels, train_paths = sample_data(train_data, sample=500, seed=0)
test_images, test_labels, test_paths = sample_data(test_data, sample=100, seed=0)

print("Sample size:", len(train_images))
print("First 5 labels:", train_labels[:5])
print("First 5 paths:", train_paths[:5])

```

Loaded 2985 training images.
 Loaded 1500 testing images.
 Sample size: 500
 First 5 labels: ['InsideCity', 'Bedroom', 'Industrial', 'Mountain', 'Street']
 First 5 paths: ['data/test/InsideCity/image_0025.jpg',
 'data/test/Bedroom/image_0024.jpg', 'data/test/Industrial/image_0276.jpg',
 'data/test/Mountain/image_0025.jpg', 'data/test/Street/image_0112.jpg']

Images were loaded as NumPy arrays and local features were extracted independently for training and test sets using the SIFT algorithm by Lowe, [2004].

The resulting descriptors were stacked into a single matrix for clustering.

[2]: #2. sample many (10K to 100K) SIFT descriptors from the images of the training set
 #(you either use a detector or sample on a grid in the scale-space)

```

def extract_descriptors(images):
    # Initialize SIFT detector
    sift = cv.SIFT_create()

    # List of image paths
    # Dictionary to store results
    descriptor_dict = {}
    all_descriptors = np.empty((1, 128))
    for key, value in images.items():
        keypoints, descriptors = sift.detectAndCompute(value, None)
        # Store results
        descriptor_dict[key] = descriptors
        all_descriptors = np.concatenate((all_descriptors, descriptors), axis=0)
    return descriptor_dict, all_descriptors

_, train_descriptors = extract_descriptors(train_images)
_, test_descriptors = extract_descriptors(test_images)
print(train_descriptors.shape)
print(test_descriptors.shape)

```

```
(263641, 128)  
(46940, 128)
```

An initial value of $k = 36$ visual words was chosen, corresponding to the number of clusters in the descriptor space and, consequently, to the number of bins in the image histograms.

```
[3]: # 3. cluster them using k-means (the choice of the number of clusters is  
# up to you, and you should experiment with different values, but you  
# could start with a few dozens);  
  
from sklearn.cluster import KMeans  
  
def get_visual_words(descriptors, k=36):  
    kmeans = KMeans(n_clusters=k, random_state=0, n_init="auto").  
    ↪fit(descriptors)  
    visual_words = kmeans.cluster_centers_  
    return visual_words  
  
# • collect (and save for future use) the clusters' centroids which represent  
↪the $k$ 128-dimensional visual words.  
  
visual_words = get_visual_words(train_descriptors, k =36)  
visual_words.shape
```

[3]: (36, 128)

Each visual word is a 128-dimensional SIFT centroid, yielding a $(k, 128)$ visual vocabulary.

4 Image representation

Given the visual vocabulary, each local descriptor was assigned to its closest visual word according to the Euclidean distance. For each image, a k -dimensional histogram was constructed by counting the occurrences of visual words across all its descriptors. This histogram provides a compact representation of the image. The resulting histograms were normalized to ensure comparability across images with different numbers of detected keypoints.

Note that the function given below also enables options for normalization and soft-assignment, which will be useful later on in the report.

```
[4]: # 2. Represent each image of the training set as a normalized histogram having  
↪k bins, each corresponding to a visual word;  
# A possibility is to perform a rather dense sampling in space and scale;  
↪another possibility is to use the the SIFT detector to find the points in  
↪scale-space where the descriptor is computed. In any case, each computed  
↪descriptor will increase the value of the bin corresponding to the closest  
↪visual word.  
  
from sklearn.metrics.pairwise import euclidean_distances
```

```

def gaussian_kernel(dist, sigma):
    return 1/(np.sqrt(2*np.pi)*sigma) * np.exp(-(dist**2) / (2 * sigma**2))

def extract_labels(img_path):
    return img_path.split('/')[-1][2]

def get_histograms(images, dictionary=visual_words, normalize=True,
                   ↪soft_assignment=False):
    sift = cv.SIFT_create()
    histograms = {}
    if soft_assignment:
        for idx in images:
            _, img_des = sift.detectAndCompute(images[idx], None)
            img_histogram = np.zeros(dictionary.shape[0])
            for des in img_des:
                dist = euclidean_distances(dictionary, des.reshape(1, -1))
                sigma = dist.mean()
                assignment = gaussian_kernel(dist, sigma)
                # assignment = 1/(np.sqrt(2*np.pi)*sigma) * np.exp(-(dist**2) / ↪(2 * sigma**2))
                img_histogram += assignment.squeeze()
            if normalize:
                hist_sum = np.sum(img_histogram)
                histograms[idx] = img_histogram/hist_sum
            else:
                histograms[idx] = img_histogram
    else:
        for idx in images:
            _, img_des = sift.detectAndCompute(images[idx], None)
            img_histogram = np.zeros(dictionary.shape[0])
            for des in img_des:
                dist = euclidean_distances(dictionary, des.reshape(1, -1))
                assignment = np.argmin(dist)
                img_histogram[assignment] += 1
            # print(f"Histogram produced for {idx}: {histogram}")
            if normalize:
                hist_sum = np.sum(img_histogram)
                histograms[idx] = img_histogram/hist_sum
            else:
                histograms[idx] = img_histogram

    img_labels, img_histograms = np.array(list(histograms.keys())), np.
    ↪array(list(histograms.values()))
    img_labels = np.array([extract_labels(img_labels[i]) for i in
    ↪range(img_labels.shape[0])])
    return img_labels, img_histograms

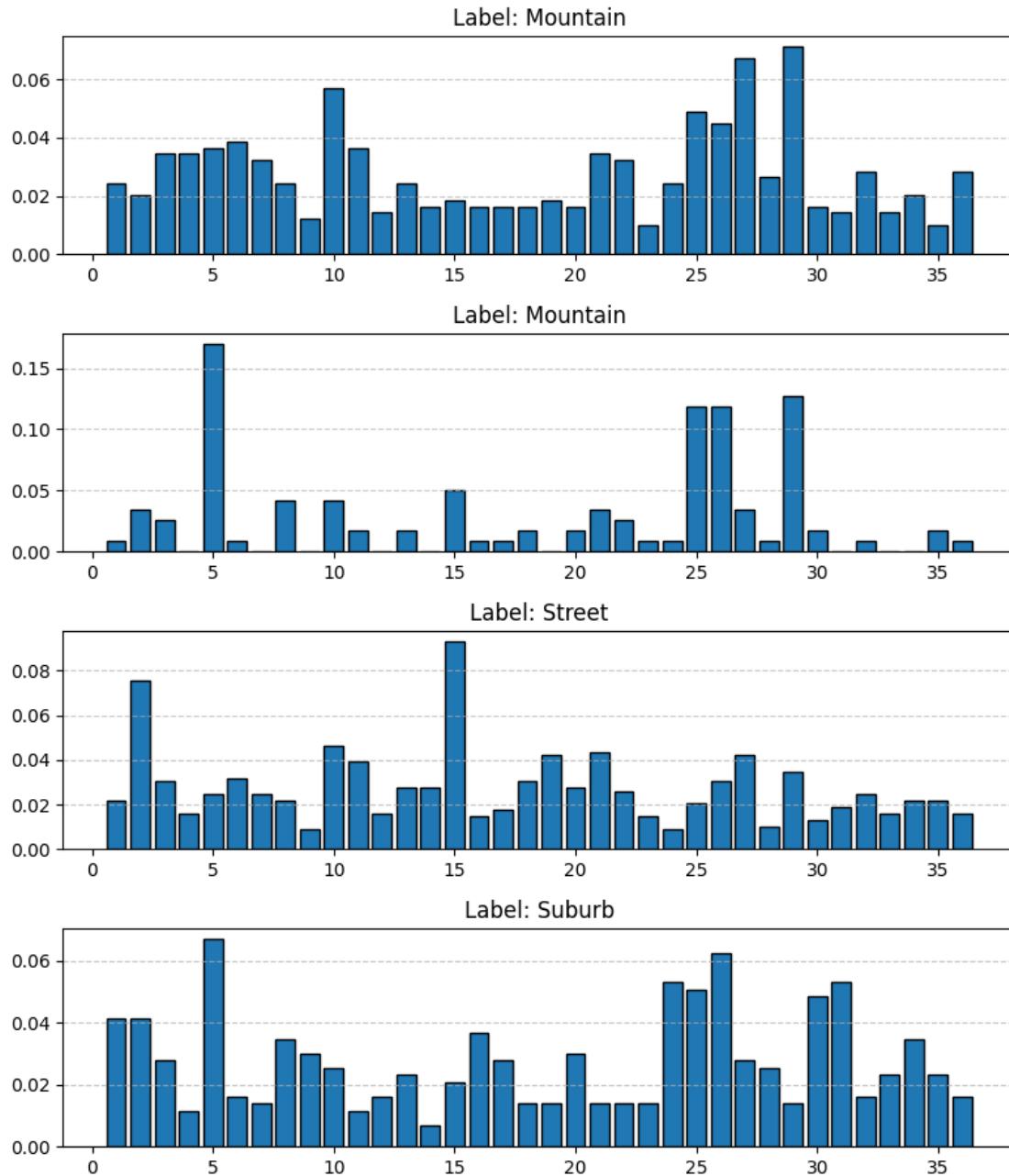
```

```
[5]: train_labels, train_histograms = get_histograms(train_images, visual_words, ▾  
    ↴normalize=True)  
test_labels, test_histograms = get_histograms(test_images, visual_words, ▾  
    ↴normalize=True)
```

Qualitatively, images belonging to different categories exhibit distinct distributions of visual words, as can be seen from the sample below.

```
[8]: import numpy as np  
import matplotlib.pyplot as plt  
  
# Number of samples to visualize  
n_samples = 4  
  
# Randomly choose indices  
idxs = np.random.choice(test_histograms.shape[0], size=n_samples, replace=False)  
  
fig, axs = plt.subplots(n_samples, figsize=(8, 10))  
fig.suptitle('Visualizing some test images')  
  
for i, idx in enumerate(idxs):  
    frequencies = test_histograms[idx]  
    label = train_labels[idx]  
  
    # Feature numbers (1 to 36)  
    features = np.arange(1, len(frequencies) + 1)  
  
    axs[i].bar(features, frequencies, edgecolor='black')  
    axs[i].set_title(f'Label: {label}')  
    axs[i].grid(axis='y', linestyle='--', alpha=0.7)  
  
plt.tight_layout(rect=[0, 0, 1, 0.96])  
plt.savefig('Test_images_visualization.png')  
plt.show()
```

Visualizing some test images



Thus, images can now be compactly represented as an histogram only, but this more compact representation still exhibits differences across categories, and can now be used for classification.

5 Classification & model selection

Different classification strategies were employed, from very simple classifiers with a standard histogram representation to more complex ones.

In this initial setting, the model was purely evaluated based on the confusion matrix and the score admittedly a harsh metric

The first candidate was the K-Nearest Neighbour Classifier

```
[9]: # 3. Employ a nearest neighbor classifier and evaluate its performance:  
# • compute the normalized histogram for the test image to be classified;  
# • assign to the image the class corresponding to the training image  
# having the closest histogram.  
# • repeat for all the test images and build a confusion matrix  
  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.preprocessing import OrdinalEncoder  
from sklearn.metrics import ConfusionMatrixDisplay  
  
def plot_confusion_matrix(y_test, y_hat, normalized=True, model_name=""):  
    if normalized:  
        fig, axs = plt.subplots(1, 2, figsize=(12, 5))  
    else:  
        fig, axs = plt.subplots(1, 1, figsize=(6, 5))  
        axs = [axs]  
  
    # --- Raw confusion matrix ---  
    ConfusionMatrixDisplay.from_predictions(  
        y_test,  
        y_hat,  
        ax=axs[0],  
        xticks_rotation='vertical'  
    )  
    axs[0].set_title(f"{model_name} - Confusion Matrix")  
  
    # --- Normalized confusion matrix ---  
    if normalized:  
        ConfusionMatrixDisplay.from_predictions(  
            y_test,  
            y_hat,  
            normalize="true",  
            values_format=".1f",  
            ax=axs[1],  
            xticks_rotation='vertical'  
        )  
        axs[1].set_title(f"{model_name} - Normalized Confusion Matrix")  
  
    plt.tight_layout()
```

```

plt.show()

def fit_and_evaluate_classifier(clf, x_train = train_histograms, ▾
    ↪y_train=train_labels, x_test=test_histograms, y_test=test_labels):
    print(clf)
    clf.fit(x_train, y_train)
    model_name = clf.__class__.__name__
    y_hat = clf.predict(x_test)

    plot_confusion_matrix(y_test = y_test, y_hat = y_hat, normalized=True, ▾
    ↪model_name=model_name)

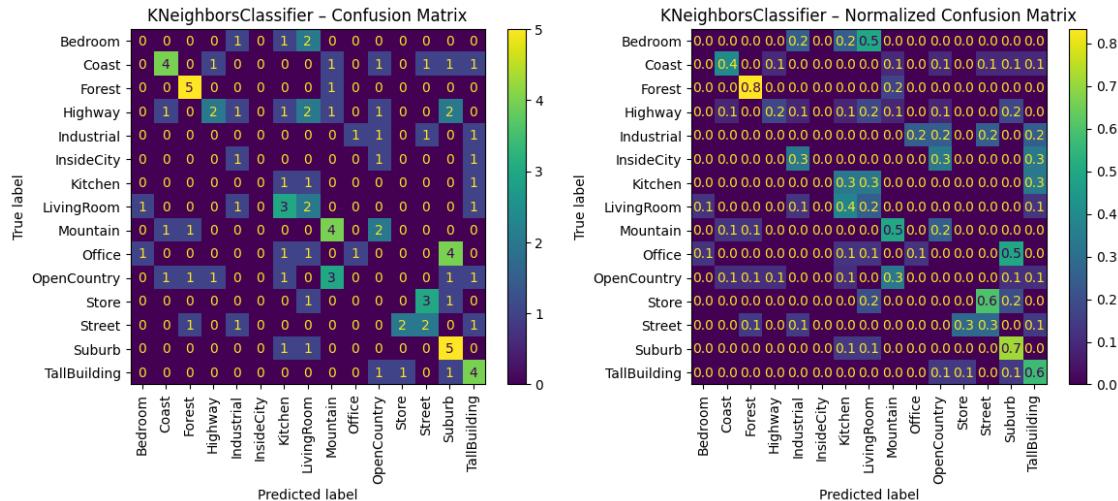
    train_score = clf.score(X=x_train, y= y_train)
    test_score = clf.score(X=x_test, y= y_test)

    print(f"Train score for {model_name}:", train_score)
    print(f"Test score for {model_name}:", test_score)

```

[10]: `neigh_clf = KNeighborsClassifier(n_neighbors=1)
fit_and_evaluate_classifier(clf=neigh_clf)`

`KNeighborsClassifier(n_neighbors=1)`



Train score for KNeighborsClassifier: 1.0

Test score for KNeighborsClassifier: 0.3

While there are few classes where the number of correct predictions is acceptable, overall this is the model that performs the worst.

The second family of candidates for classification were SVMs trained on the image histograms

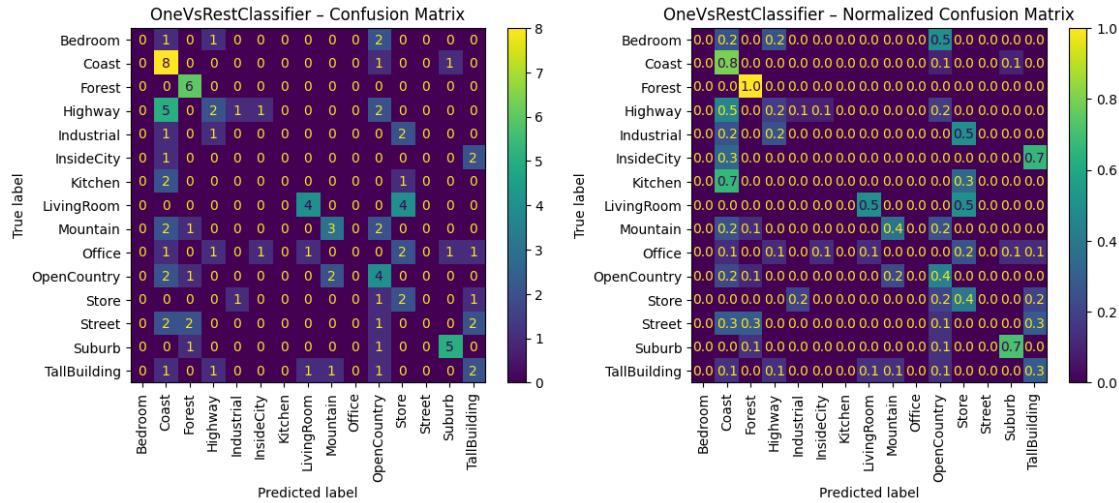
```
[11]: # 4. Train a multiclass linear Support Vector Machine, using the one-vs-rest
# approach (you will need to train 15 binary classifiers having the normalized
# histograms as the input vectors and positive labels for the "one" class and
# negative for the "rest.")

# 5. Evaluate the multiclass SVM:
# • compute the normalized histogram for the test image to be classified;
# • compute the real-valued output of each of the SVMs, using that histogram as
#   input;
# • assign to the image the class corresponding to the SVM having the
# greatest real-valued output.
# • repeat for all the test images and build a confusion matrix.

from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC, SVC
```

linear_svm = OneVsRestClassifier(LinearSVC())
fit_and_evaluate_classifier(clf=linear_svm)

```
OneVsRestClassifier(estimator=LinearSVC())
```



Train score for OneVsRestClassifier: 0.42

Test score for OneVsRestClassifier: 0.36

The second variant would be using the generalized gaussian kernel, which can accept a preferred distance metric, not only euclidean.

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{d(\mathbf{x}, \mathbf{y})^p}{2\sigma^p}\right)$$

where: - $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ are feature vectors, - $d(\mathbf{x}, \mathbf{y})$ is a chosen distance metric, - $\sigma > 0$ is the kernel

bandwidth, $-p > 0$ controls the shape of the kernel
 (for $p = 2$ and Euclidean distance, this reduces to the standard RBF kernel).

Especially when comparing histograms, choosing a more specific and meaningful distance, like chi-squared or Earth Mover's Distance, might bring benefits compared to just using the standard euclidean distance.

$$d_{\chi^2}(\mathbf{x}, \mathbf{y}) = \frac{1}{2} \sum_{i=1}^d \frac{(x_i - y_i)^2}{x_i + y_i + \varepsilon}$$

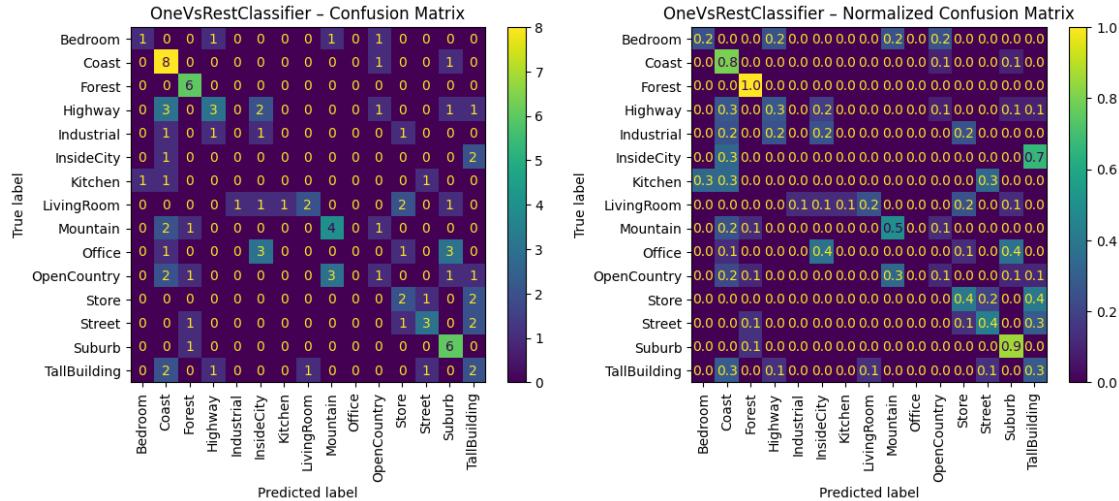
[12]: # 6. Optionally, you could train the SVM using a generalized Gaussian kernel
 # (Lecture 7) based on the χ^2 distance;

```
import numpy as np
from sklearn.metrics.pairwise import chi2_kernel

def generalized_gaussian_kernel(X, Y, A=2.0):
    gamma = 1.0 / A
    return chi2_kernel(X, Y, gamma=gamma)
```

[13]: ggk_clf = OneVsRestClassifier(SVC(kernel=generalized_gaussian_kernel))
fit_and_evaluate_classifier(clf=ggk_clf)

OneVsRestClassifier(estimator=SVC(kernel=<function generalized_gaussian_kernel at 0x7f164d3e7320>))



Train score for OneVsRestClassifier: 0.73

Test score for OneVsRestClassifier: 0.38

Another option to improve SVM-based classification borrows from Information-Theory and specifically error-correcting output codes: the main idea here is to represent each class as a unique binary

code, keeping their specific representations in a “codebook”, and train the classifier on those.

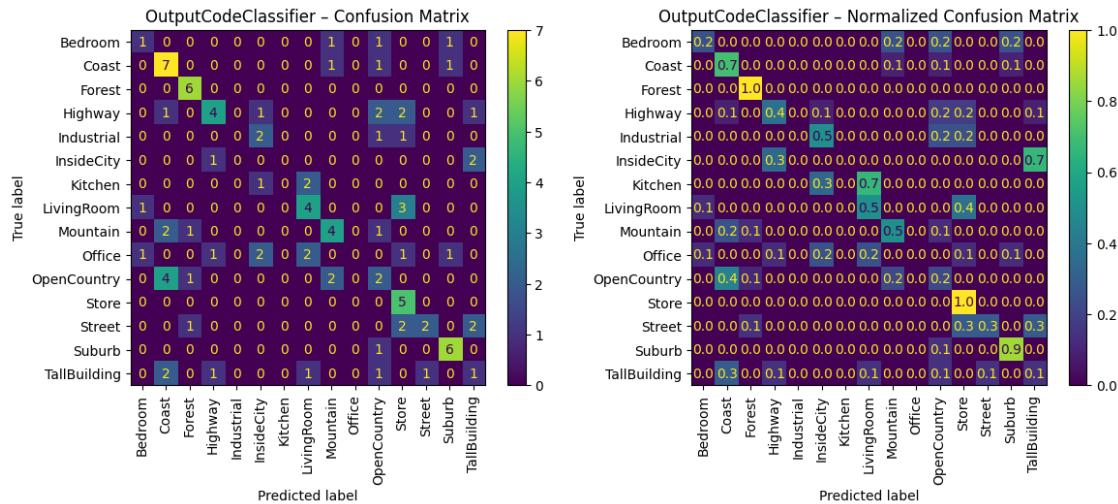
```
[14]: # 7. optionally, you could implement the multiclass SVM using the Error
      ↵Correcting Output Code approach [Dietterich and Bakiri, 1994, James and
      ↵Hastie, 1998];

from sklearn.multiclass import OutputCodeClassifier

# Create an SVM classifier
svm = SVC(kernel='rbf')
# Create an ECOC classifier
# ecoc = OutputCodeClassifier(estimator=sum, code_size=2, random_state=42)
ecoc = OutputCodeClassifier(estimator=OneVsRestClassifier(SVC()), code_size=2,
                           ↵random_state=42)

fit_and_evaluate_classifier(clf=ecoc)
```

```
OutputCodeClassifier(code_size=2,
                     estimator=OneVsRestClassifier(estimator=SVC()),
                     random_state=42)
```



Train score for OutputCodeClassifier: 0.744

Test score for OutputCodeClassifier: 0.42

The histogram representations can also be changed: by using a gaussian kernel instead of a hard assignment based on distances, a single descriptor might in fact contribute to multiple bins, leading to more nuanced results.

```
[19]: # 8. optionally, you could use soft assignment to assign descriptors to
      ↵histogram
```

```

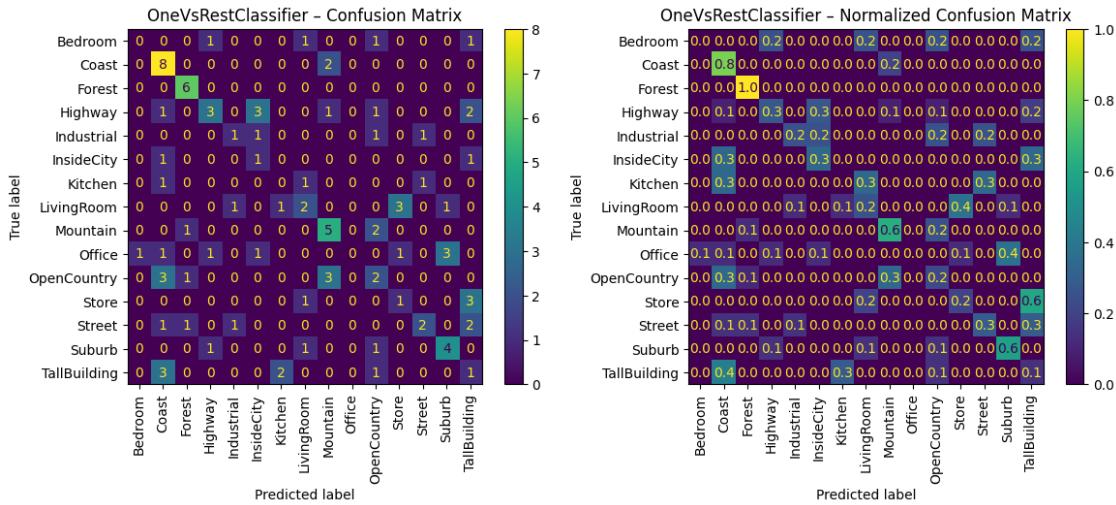
# bins. Each descriptor will contribute to multiple bins, using a distancebased
# weighting scheme (see [Van Gemert et al., 2008]);

_, soft_assignment_histograms_train = get_histograms(train_images,
    ↪visual_words, normalize=True, soft_assignment=True)
_, soft_assignment_histograms_test = get_histograms(test_images, visual_words,
    ↪normalize=True, soft_assignment=True)

soft_assignment_classifier = OneVsRestClassifier(SVC())
fit_and_evaluate_classifier(clf=soft_assignment_classifier,
    ↪x_train=soft_assignment_histograms_train,
    ↪x_test=soft_assignment_histograms_test)

```

OneVsRestClassifier(estimator=SVC())



Train score for OneVsRestClassifier: 0.716

Test score for OneVsRestClassifier: 0.36

Finally, by following Lazebnik et al., 2006, it's possible to take into account multiple levels of the image via gridding, effectively computing multiple histograms for the same image. In this case, the representation for each image will be an even larger tensor than before, albeit much sparser.

[15]: # 9. optionally, you could add some spatial information by taking inspiration
from the spatial pyramid feature representation of [Lazebnik et al., 2006]
(a simple approach could be that of building an extended descriptor, by
stacking the histogram of the whole image, the histograms of the 4 quadrants
↪and so on, up to a desired level).
this entails
computing L different resolutions of the image
splitting the image into $2^d l$ grids.

```

# finding the histogram of each image at each resolution
# find how many points from X and Y are contained in the same grid

def sequential_split(img, n_grids, verbose=False):
    # first split rows
    rows = np.array_split(img, n_grids, axis=0)

    # then split columns for each row block
    subarrays = [np.array_split(r, n_grids, axis=1) for r in rows]
    if verbose:
        print("Size of first subarray:", subarrays[0][0].shape)

    return subarrays

```

```

[16]: def get_multiresolution_histograms(images, resolution_levels=2, ndim_img = 2, ↴
                                         verbose=False):
    multi_resolution_histograms = {}

    sift = cv.SIFT_create()
    for img_path, img in images.items():

        resolutions = [l for l in range(resolution_levels)]
        d = ndim_img
        resized = img
        if verbose:
            print("Original Image size:", img.shape)
        orig_width, orig_height = img.shape[0], img.shape[1]

        # resolution_collector = np.zeros((len(resolutions), n_grids, ↴
        # visual_words.shape[0]))
        max_grids = d*(resolutions[-1] + 1)
        if verbose:
            print("maximum number of grids obtainable:", max_grids)

        resolution_collector = np.zeros((len(resolutions), max_grids, ↴
                                         visual_words.shape[0]))
        for l in resolutions: # missing first level! (full histogram)
            new_width = int(np.round(orig_width/(l + 1)))
            new_height = int(np.round(orig_height/(l + 1)))
            if verbose:
                print("new dims:", new_width, new_height)
            # resized = cv.resize(resized, None, fx = 0.5, fy = 0.5)
            resized = cv.resize(resized, (new_width, new_height))
            if verbose:
                print(f'Resized(resolution level {l}):', resized.shape)

        n_grids = 2**((d*l)

```

```

    if verbose:
        print("number of extracted grids:", n_grids)
    grids = sequential_split(img, n_grids)[0]
    # for g in grids:
    #     sift = cv.create_SIFT()
    #     keypoints, descriptors = cv.DetectAndCompute(g, None)
    # ...
    # resolution_collector = np.zeros((len(resolutions), n_grids, visual_words.shape[0]))
    # grid_collector = np.zeros((n_grids, visual_words.shape[0]))

    for i, g in enumerate(grids):

        keypoints, descriptors = sift.detectAndCompute(g, None)
        grid_histogram = np.zeros(visual_words.shape[0])

        if descriptors is not None: # probably could be amended by just
            # sampling and not actually
            for des in descriptors:
                dist = euclidean_distances(visual_words, des.reshape(1, -1))
                assignment = np.argmin(dist)
                grid_histogram[assignment] += 1
        if verbose:
            print(f"{i}-th grid histogram:", grid_histogram)

        resolution_collector[1, i, :] = grid_histogram
        if verbose:
            print(f"resolution_collector for level {1}:",
            resolution_collector)

        multi_resolution_histograms[img_path] = resolution_collector
        img_labels, img_histograms = np.array(list(multi_resolution_histograms.keys())),
        np.array(list(multi_resolution_histograms.values()))
        img_labels = np.array([extract_labels(img_labels[i]) for i in
        range(img_labels.shape[0])])

    return img_labels, img_histograms

```

After summing twice, over grids and over resolutions, we get, once again, one k -dimensional histogram for each image. This time, however, it's the sum of different grids and each should be weighted accordingly: matches found between images at a finer scale should have greater weight than those found at a larger scale.

```
[18]: def histogram_intersection(hists_A, hists_B, verbose=False):
    if verbose:
        print("Input histograms shape:", hists.shape)
```

```

intersection_over_grids = np.minimum(
    hists_A[:, None, :, :, :],
    hists_B[None, :, :, :, :]
).sum(axis=3)

if verbose:
    print("Intersection shape:", intersection_over_grids.shape)

return intersection_over_grids


def pyramid_match_kernel(hists_A, hists_B, L, verbose=False):
    """
    returns an n_images x n_images matrix.
    """

N = hists_A.shape[0]
M = hists_B.shape[0]

if verbose:
    print("Number of images:", N)
    print("Histogram tensor shape:", hists.shape)

K = np.zeros((N, M))

I = histogram_intersection(hists_A, hists_B, verbose=verbose)

if verbose:
    print("shape of histogram intersections:", I.shape)

I_0 = I[:, :, 0, :]
first_term = 1 / (2 * L) * I_0
K = first_term

for l in range(1, L):
    scale_factor = 1 / (2 * (L - l + 1))

    if verbose:
        print(f"Level {l}, scale factor {scale_factor}")

    K += scale_factor * I[:, :, l, :]

final_kernel = K.sum(axis=-1)

if verbose:
    print("Final kernel shape:", final_kernel.shape)

```

```

    assert final_kernel.shape == (N, M), (
        f"Assertion Failed: shape of pyramid match kernel "
        f"should be {(N, M)} but is {final_kernel.shape}"
    )

    return final_kernel

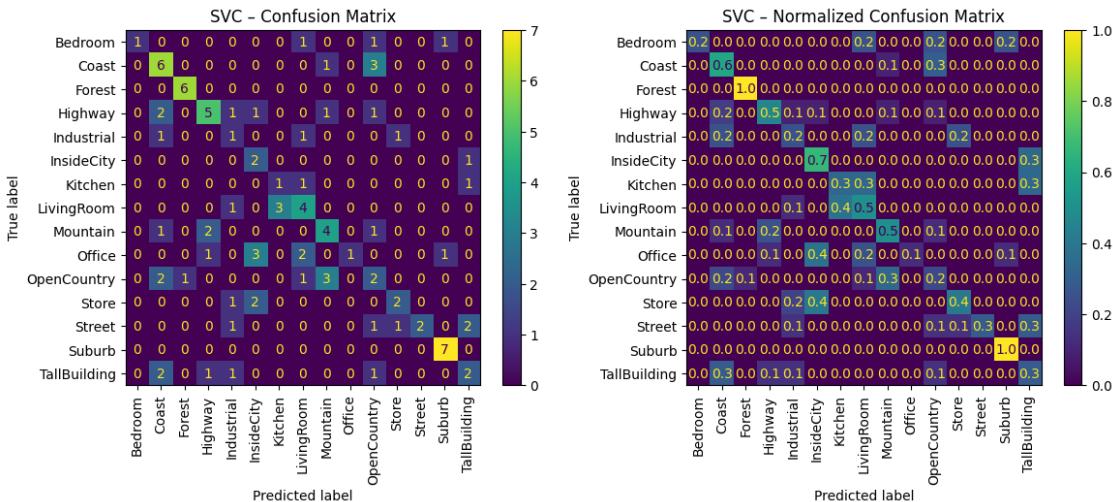
```

[19]: pmkernel_train_labels, pmkernel_train_hists =
 ↪get_multiresolution_histograms(train_images, resolution_levels=2)
pmkernel_test_labels, pmkernel_test_hists =
 ↪get_multiresolution_histograms(test_images, resolution_levels=2)

pm_train_kernel =
 ↪pyramid_match_kernel(pmkernel_train_hists, pmkernel_train_hists, L=2)
pm_test_kernel = pyramid_match_kernel(pmkernel_train_hists, □
 ↪pmkernel_test_hists, L=2)

[20]: pmkernel_classifier = SVC(kernel="precomputed")
fit_and_evaluate_classifier(
 clf=pmkernel_classifier,
 x_train=pm_train_kernel, # (n_train, n_train)
 y_train=train_labels,
 x_test=pm_test_kernel.T, # MUST be (n_test, n_train)
 y_test=test_labels
)

SVC(kernel='precomputed')



Train score for SVC: 1.0

Test score for SVC: 0.46

This simple experimental setting, although useful didactically, did not scale well, especially in the case of the Pyramid Match Kernel, restricting evaluation to only a fraction of the available data. Furthermore, I had hard limits on my hardware for loading and computing all the descriptors and kernel matrices necessary.

6 Advanced experimental setting

To enable a more ordered and systematic comparison between models and hyperparameters, the initial prototype code was restructured into a cluster-friendly pipeline, also leveraging the help of ChatGPT.

Specific names and parameter attributes were set for each of the classifiers illustrated above and afterwards multiple experiments were executed on the Orfeo computing cluster, allowing the use of the full dataset and a more exhaustive exploration of hyperparameters.

Using *scikit-learn*'s pipeline and model selection utilities, the following parameters were varied:

- Number of neighbours for the kNN classifier.
- Number of visual words: 32, 64, 128, 256, 512.
- Number of Pyramid Match Kernel levels L , which controls the number of increasingly finer spatial grids (2^{dL} in d dimensions).
- Kernel types: linear, radial basis function (RBF), polynomial
- Gaussian kernel bandwidth A .
- SVM regularization parameter C .

All experiments were performed using 5-fold cross-validation with the same seed. The multi-seed option, influencing the sampling of descriptors, was considered, but afterwards abandoned due to time constraints. The entirety of the Dataset was used this time; however a maximum limit of 200.000 descriptors was chosen, which were then subsequently batched by `MiniBatchKMeans` in batches of 4096 elements when creating the visual vocabulary.

Furthermore, since in the original data it was observed that the test split contained more images than the training split, the two were swapped in an attempt to improve training performance.

7 Results

The Pyramid Match Kernel and ECOC-based SVMs were the most computationally expensive methods. Special care was taken to avoid excessive memory usage when computing the $N \times N$ kernel matrix for the Pyramid Match Kernel. These models required approximately 8 minutes per run, while simpler classifiers typically completed within 1 minute.

Models were evaluated using:

- classification accuracy
- Roc-curves and AUC metrics.

Regarding the ROC curve and AUC metrics, as we are dealing with multiclass classification, a different approach has to be used when interpreting the curve, as these metrics were originally built for binary classification. Following [Hand et al., 2001](#) we turn the multiclass classification into a series of one-vs-all problems and then average these scores.

In addition, confusion matrices were plotted to analyze per-class performance.

Performance improves marginally as we consider more descriptors. However, the runtime quickly becomes unfeasible (1.5+ hours).

Below a summary table of the best hyperparameters found for each model variant via Cross-Validation.

Model ID	Model Name	Best CV			Other Parameters
		Score	k	C	
1	BoVW Hard kNN	0.380	64	—	n_neighbors = 5
2	BoVW Hard Linear SVC	0.472	512	10	—
3	BoVW Hard Chi ² SVC	0.574	512	10	A = 0.5
4	BoVW Hard ECOC (RBF)	0.499	512	10	kernel = rbf
5	BoVW Soft SVC (Polynomial)	0.538	256	0.1	kernel = poly
6	Pyramid Match Kernel (PMK)	0.634	512	1.0	L = 3

Across cross-validation folds, overall performance gain remained moderate.

Here we see that all SVM models decisively outperform a KNN classifier. There seem to be some benefits from adopting non-linear kernels, as in the case of the generalized gaussian kernel with chi-squared distance, which is only beaten by the histogram intersection kernel by a slight margin: This suggests that incorporating spatial information and appropriate histogram similarity measures is crucial for this task, although not sufficient.

On the contrary, adding soft assignment to the categories does not seem to bring any benefit. For most models, increasing visual vocabulary size decisively improved the performance.

```
[26]: def display_model_confusion_matrices(model):
    raw = Path(model) / "confusion_raw.png"
    # raw = os.path.join(path, "confusion_raw.png")
    norm = Path(model) / "confusion_normalized.png"
    # norm = os.path.join(path, "confusion_normalized.png")

    display(Markdown(f"### {model}"))

    html = "<div style='display: grid; grid-template-columns: 1fr 1fr; gap: 16px; '>"
    if raw.exists():
        html += f"""
        <div style='text-align: center; '>
            <img src='{raw}' style='max-width: 100%;' />
            <div><b>Raw</b></div>
        </div>
        """
    if norm.exists():
        html += f"""
        <div style='text-align: center; '>
            <img src='{norm}' style='max-width: 100%;' />
            <div><b>Normalized</b></div>
        </div>
        """

    display(Markdown(html))
```

```

        
        <div><b>Normalized</b></div>
    </div>
    """
    html += "</div>"

display(HTML(html))

```

[27]: `display_model_confusion_matrices("images/1_BoVW_Hard_kNN")`

7.0.1 images/1_BoVW_Hard_kNN

<IPython.core.display.HTML object>

Unsurprisingly, the KNN model fails at consistently recognizing most classes.

[29]: `display_model_confusion_matrices("images/2_BoVW_Hard_LinearSVC")`

7.0.2 images/2_BoVW_Hard_LinearSVC

<IPython.core.display.HTML object>

The situation already improves with a linear SVM, even if not by a large margin.

[30]: `display_model_confusion_matrices("images/3_BoVW_Hard_Chi2SVC")`

7.0.3 images/3_BoVW_Hard_Chi2SVC

<IPython.core.display.HTML object>

Going from a linear to a non-linear kernel improves classification by a lot.

[31]: `display_model_confusion_matrices("images/5_BoVW_Soft_SVC")`

7.0.4 images/5_BoVW_Soft_SVC

<IPython.core.display.HTML object>

Instead, adopting soft assignment doesn't seem to bring any particular improvement to classification.

[35]: `display_model_confusion_matrices("images/6_PMK")`

7.0.5 images/6_PMK

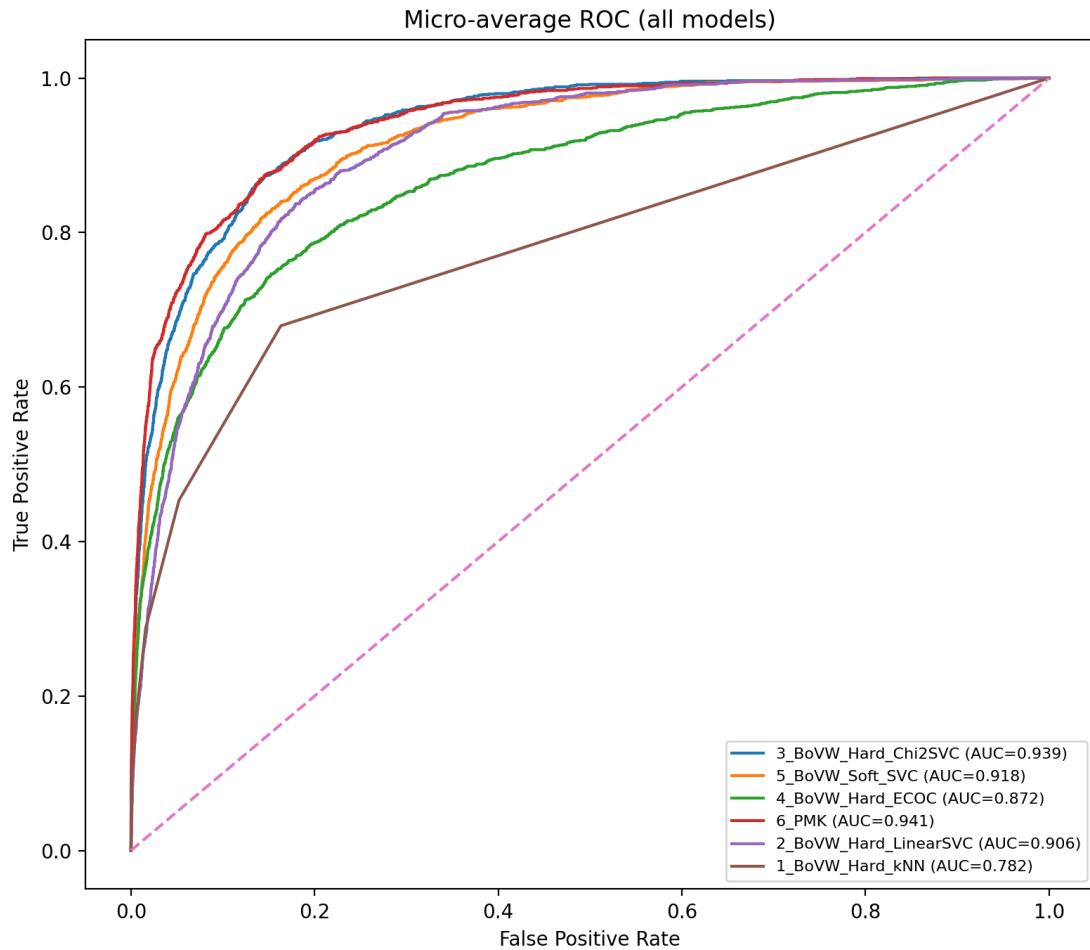
<IPython.core.display.HTML object>

The Pyramid Match Kernel is by far the strongest model: its diagonal is solid, even though some classes like Bedroom are still misclassified often. In general, across models, there are clearly classes which are more easily recognized (e.g Suburb, Forest) than others (Bedroom, Kitchen). By observing where the instances of these hard classes are misclassified, it seems all models particularly

struggle in discerning different parts of a house/building from each other: even for PMK, the categories **Office**, **Bedroom**, **Kitchen** have the lowest values, and a large part of the missclassification is between e.g **Kitchen** and **LivingRoom** or **Bedroom** and **LivingRoom**.

In order to get a more immediate visual comparison between all of these models, the ROC (Receiver Operating Characteristic) curve can be employed.

```
[25]: from IPython.display import Image, display
display(Image(filename="FINAL/roc_all_models.png"))
```

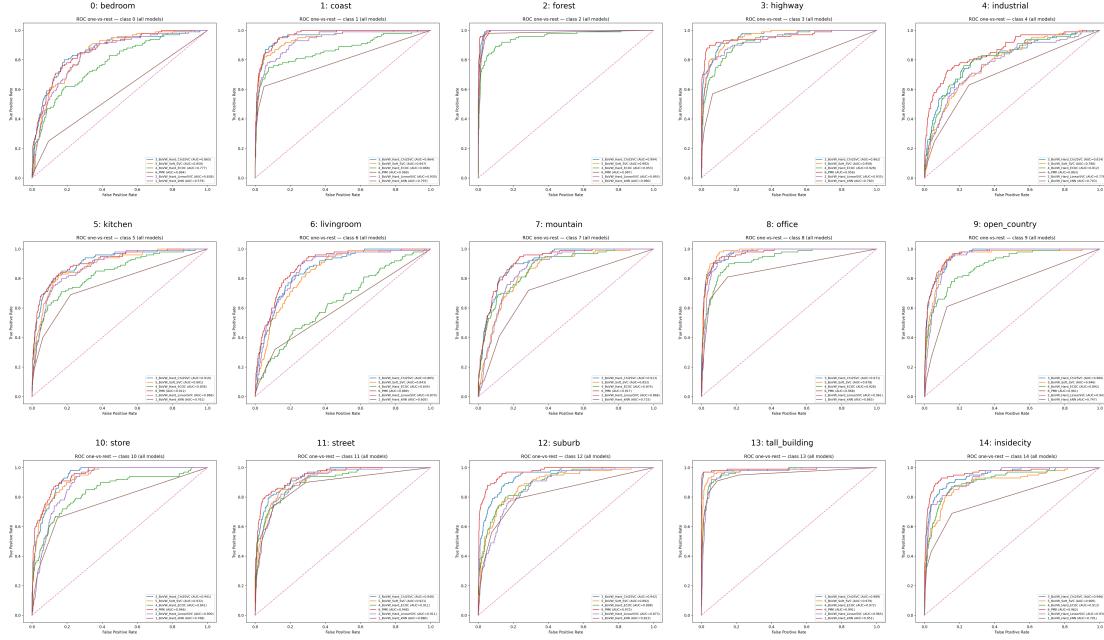


Usually, the higher the area under the curve (AUC), the better the classifier. A curve reaching up to 1 would mean a perfect classifier, while the dashed curve in the middle is the baseline random classifier.

However, some caution has to be employed when utilizing this metric, as it was originally built only for binary classification: what is represented here is therefore an average among all the classes, which is far from a guarantee that the model will perform even decently in some of them.

See the plots below for an example of this: while for some classes (e.g `Forest` we get remarkably close to the perfect classifier, for other the performance is unsatisfactory, especially for those classes which come from a similar environment (e.g `Living Room`, `Bedroom`, `Kitchen` being all rooms of the house) which the model struggles to tell apart.

```
[34]: display(Image(filename="FINAL/roc_per_class_grid.png"))
```



These single plots can be found in the `roc_per_class` folder inside `images`.

8 Outlook

In this brief experiment, it was shown how the ‘bag-of-visual-words’ approach can be beneficial when training models for classification in computer vision: the increases expression of SVMs, paired with more information (e.g PMK) has beneficial effects on classification accuracy. While some interesting results were achieved, further extensions are needed:

- focused effort on disentangling house rooms: as mentioned above, all models across the spectrum struggle to distinguish the different rooms of the house between each other. Special care and work would have to be put into improving just this specific aspect.
- more exhaustive hyperparameter search: due to time constraints, not all the tunable parameters were always varied in the models, focusing most of the time on those of outmost importance (e.g C in the Pyramid Match Kernel was set to 1.0). Further work might explore these untuned parameters to see if they make any difference.
- Further optimization in pytorch: especially with the PMK, large tensor operations become heavy quickly. Some GPU optimization could benefit the runtime of the whole experiment set and promote exploration.
- Further metric exploration. Here, only the accuracy and ROC/AUC metric were employed. However, there is a vast discussion on the validity of these metrics, and many interesting proposals for new and alternative ones.

9 References

- Dietterich, T. G., & Bakiri, G. (1994). Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2, 263–286.
- Lazebnik, S., Schmid, C., & Ponce, J. (2006). Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)* (Vol. 2, pp. 2169–2178).
- Van Gemert, J. C., Geusebroek, J.-M., Veenman, C. J., & Smeulders, A. W. (2008). Kernel codebooks for scene categorization. In *European Conference on Computer Vision (ECCV)* (pp. 696–709). Springer.
- Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2), 91–110.
- Hand, D.J., Till, R.J. A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems. *Machine Learning* 45, 171–186 (2001).