CPSC 425 - A5

Eloise Peng, 13812169

November 2019

1 Q4

1.1 build_vocabulary()

```python
def build_vocabulary(image_paths, vocab_size):
    """ Sample SIFT descriptors, cluster them using k-means, and return the fitted k-means model.
    NOTE: We don't necessarily need to use the entire training dataset. You can use the function
    sample_images() to sample a subset of images, and pass them into this function.

    Parameters
    ----------
    image_paths: an (n_image, 1) array of image paths.
    vocab_size: the number of clusters desired.

    Returns
    -------
    kmeans: the fitted k-means clustering model.
    """
    n_image = len(image_paths)

    # Since want to sample tens of thousands of SIFT descriptors from different images, we
    # calculate the number of SIFT descriptors we need to sample from each image.
    n_each = int(np.ceil(10000 / n_image))

    # Initialize an array of features, which will store the sampled descriptors
    # keypoints = np.zeros((n_image * n_each, 2))
    descriptors = np.zeros((n_image * n_each, 128))

    for i, path in enumerate(image_paths):
        # Load features from each image
        features = np.loadtxt(path, delimiter=',',dtype=float)
        sift_keypoints = features[:, :2]
        sift_descriptors = features[:, 2:]
        # TODO: Randomly sample n_each descriptors from sift_descriptor and store them into descriptors
        r = np.random.choice(sift_descriptors.shape[0], min(n_each, len(sift_descriptors)), replace=False)
        descriptors = np.vstack((descriptors, sift_descriptors[r,]))

    # TODO: perform k-means clustering to cluster sampled sift descriptors into vocab_size regions.
    # You can use KMeans from sci-kit learn.
    # Reference: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
    kmeans = KMeans(n_clusters=vocab_size, n_jobs=8).fit(descriptors)
    return kmeans
```

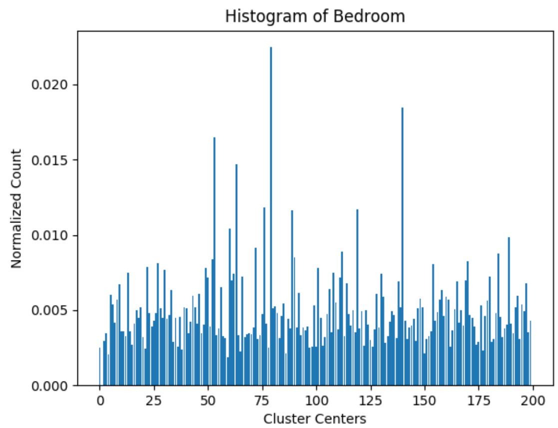
1.2 get_bags_of_sift()

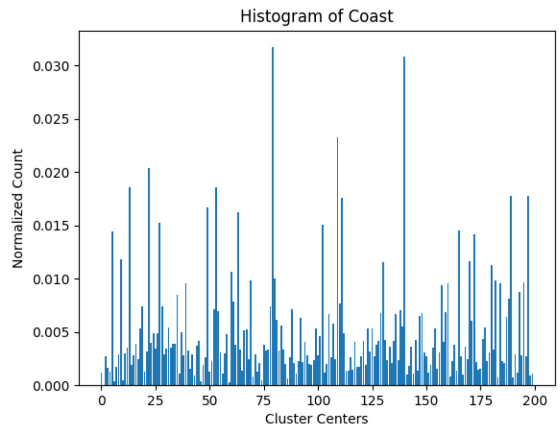
```python
def get_bags_of_sifts(image_paths, kmeans):
    """ Represent each image as bags of SIFT features histogram.

    Parameters
    ----------
    image_paths: an (n_image, 1) array of image paths.
    kmeans: k-means clustering model with vocab_size centroids.

    Returns
    -------
    image_feats: an (n_image, vocab_size) matrix, where each row is a histogram.
    """
    n_image = len(image_paths)
    vocab_size = kmeans.cluster_centers_.shape[0]

    image_feats = np.zeros((n_image, vocab_size))

    for i, path in enumerate(image_paths):
        # Load features from each image
        features = np.loadtxt(path, delimiter=',',dtype=float)

        # TODO: Assign each feature to the closest cluster center
        # Again, each feature consists of the (x, y) location and the 128-dimensional sift descriptor
        # You can access the sift descriptors part by features[:, 2:]
        closest_cluster_center = kmeans.predict(features[:, 2:])
        # TODO: Build a histogram normalized by the number of descriptors
        np.add.at(image_feats[i], closest_cluster_center, 1/features.shape[0])

    return image_feats
```

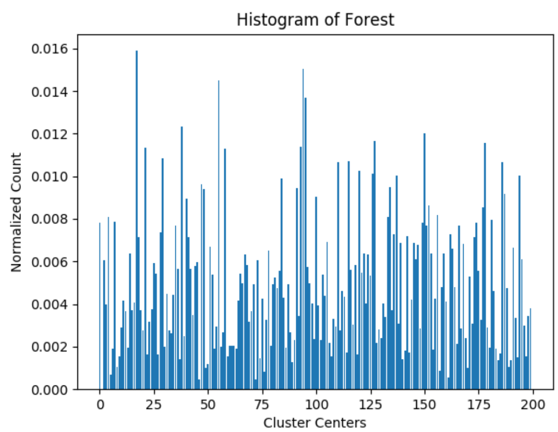
1.3 plot_histograms()

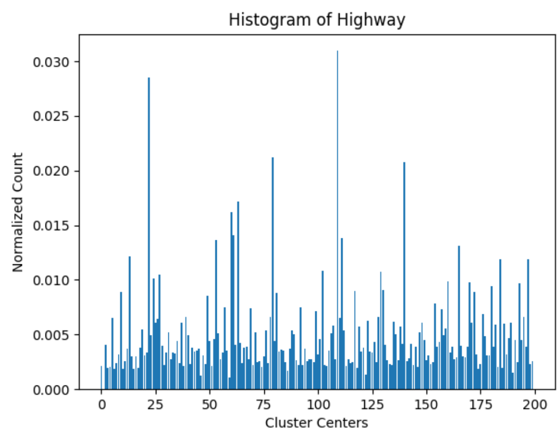
```python
def plot_histograms(image_feats, labels):
    """ image_feats: an (n_image, vocab_size) matrix, where each row is a histogram.
    labels: class labels corresponding to each image

    Parameters
    ----------
    image_feats: an (n_image, vocab_size) matrix, where each row is a histogram.
    labels: class labels corresponding to each image

    Output/Display
    --------
    histograms of each class
    """
    hist = {}
    for i, label in enumerate(labels):
        hist_label = classes_dict[int(label)]
        cur_hist_data = hist.get(hist_label, (np.zeros((1, image_feats.shape[1])), 0))
        hist[hist_label] = (np.add(cur_hist_data[0], image_feats[i]), cur_hist_data[1]+1)

    for label, (f, count) in hist.items():
        plt.clf()
        plt.bar(np.arange(image_feats.shape[1]), (f[0]/count))
        plt.ylabel("Normalized Count")
        plt.xlabel("Cluster Centers")
        plt.title("Histogram of " + label)
        plt.show()
```

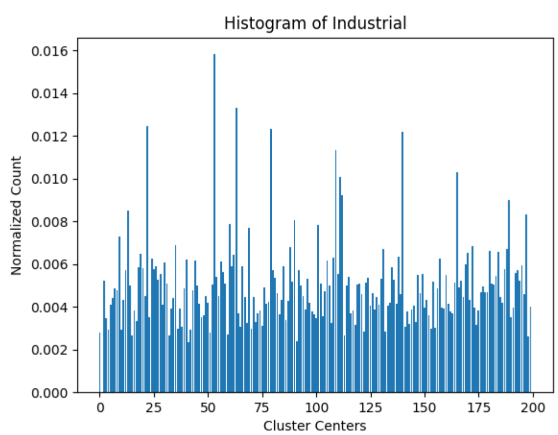
Average BoW histogram plots for 15 classes

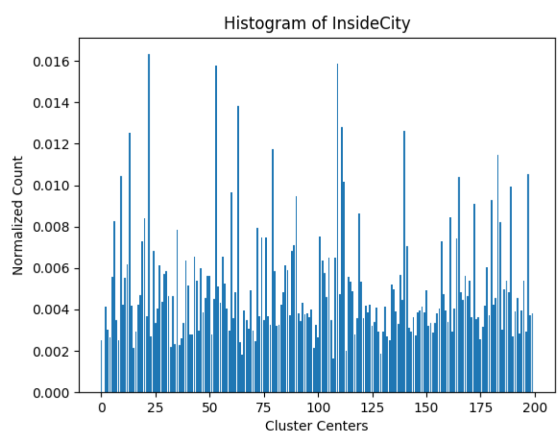


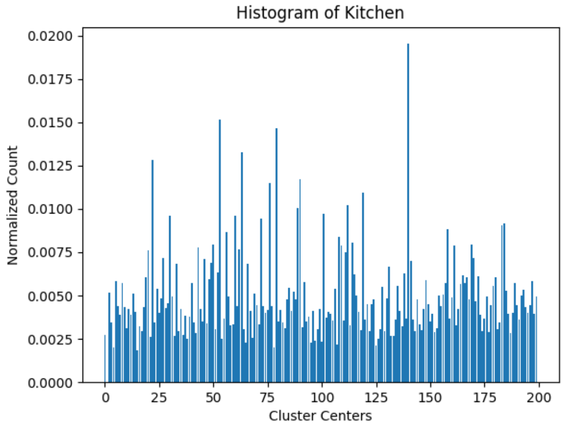


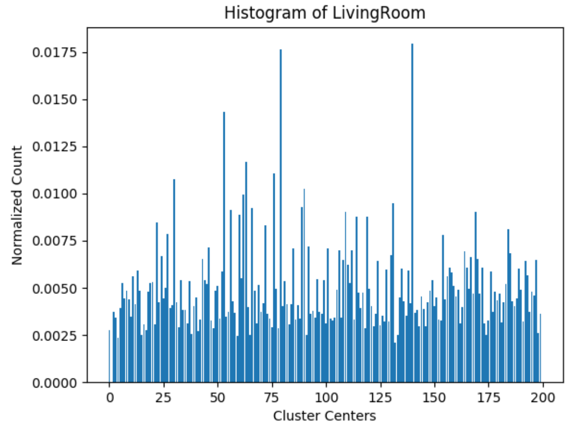


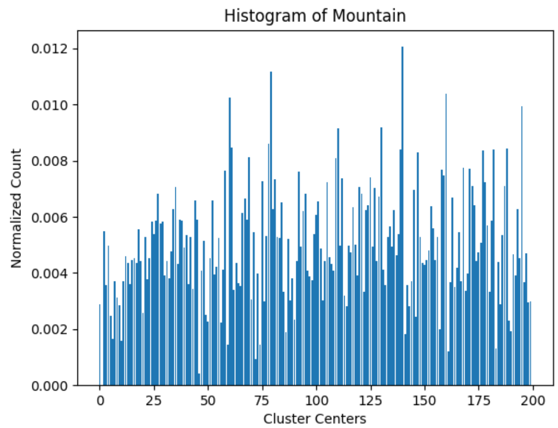


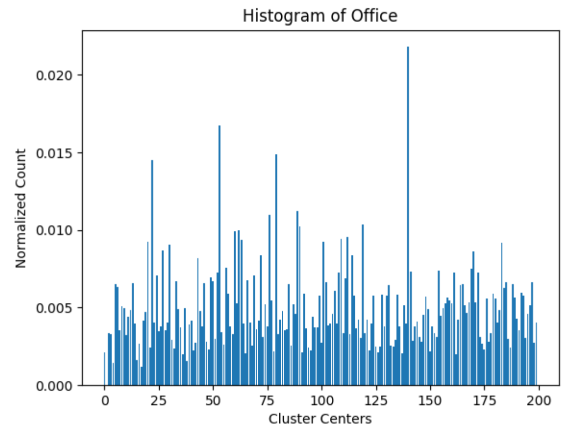


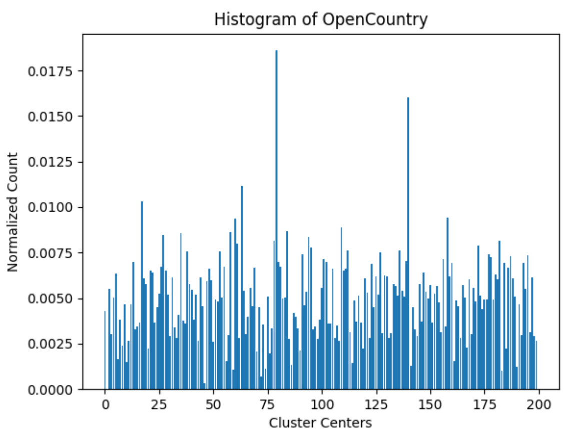


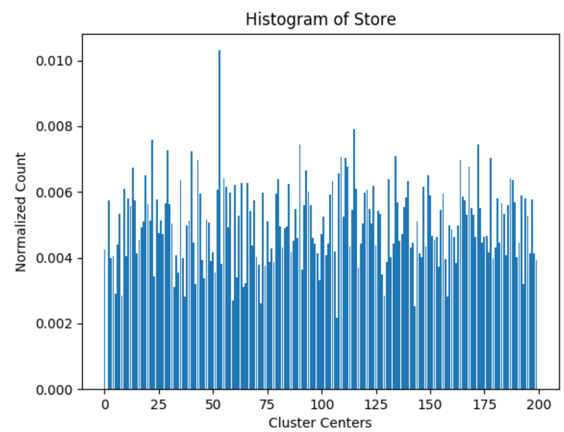

Histogram of Kitchen
Normalized Count
Cluster Centers

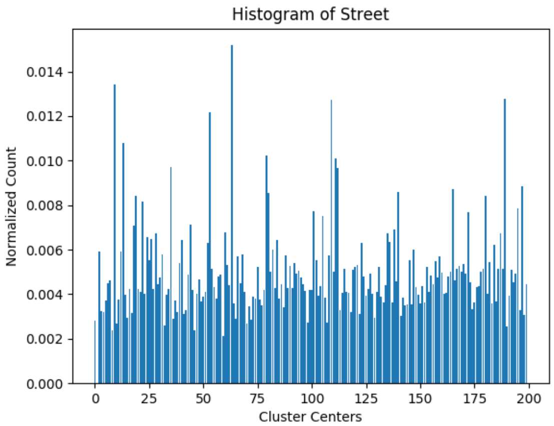

Histogram of LivingRoom
Normalized Count
Cluster Centers

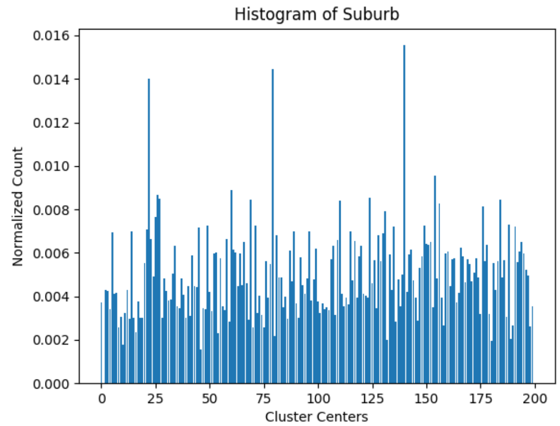

Histogram of Mountain
Normalized Count
Cluster Centers

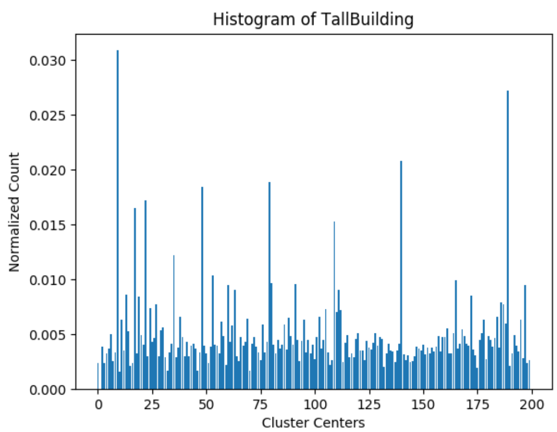

Histogram of Office
Normalized Count
Cluster Centers


Histogram of OpenCountry
Normalized Count
Cluster Centers


Histogram of Store
Normalized Count
Cluster Centers







As we observed in above histograms, some classes have a generally **higher count** on each cluster such as Mountain and Store. And some classes have a **larger variance** on each cluster than the others such as Coast, InsideCity, and TallBuilding. The larger the variance, the easier to distinguish the scene. Thus, I believe the classes are hardest to separate are those with smallest variance, as **Store** and **Suburb**.

2 Q5

2.1 nearest_neighbor_classify()

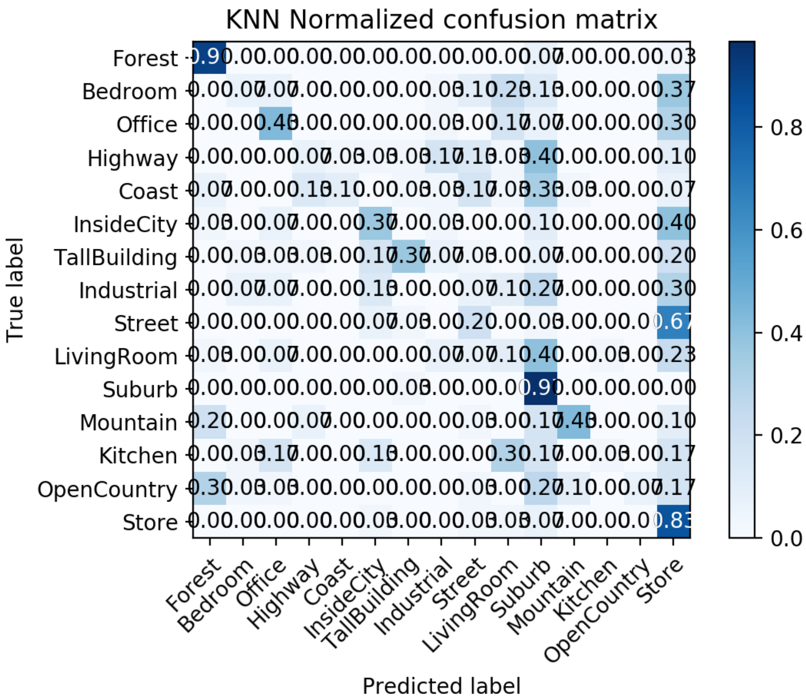
```python
def nearest_neighbor_classify(train_image_feats, train_labels, test_image_feats):

    '''
    Parameters
        ----------
        train_image_feats:  is an N x d matrix, where d is the dimensionality of the feature representation.
        train_labels: is an N x l cell array, where each entry is a string
                      indicating the ground truth one-hot vector for each training image.
        test_image_feats: is an M x d matrix, where d is the dimensionality of the
                          feature representation. You can assume M = N unless you've modified the starter code.

    Returns
        -------
        is an M x l cell array, where each row is a one-hot vector
        indicating the predicted category for each test image.

    Usefull funtion:

        # You can use knn from sci-kit learn.
        # Reference: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
    '''
    # using default k = 5
    neigh = neighbors.KNeighborsClassifier(n_neighbors=7)
    neigh.fit(train_image_feats, train_labels)
    predicted_labels = neigh.predict(test_image_feats)
    return predicted_labels
```

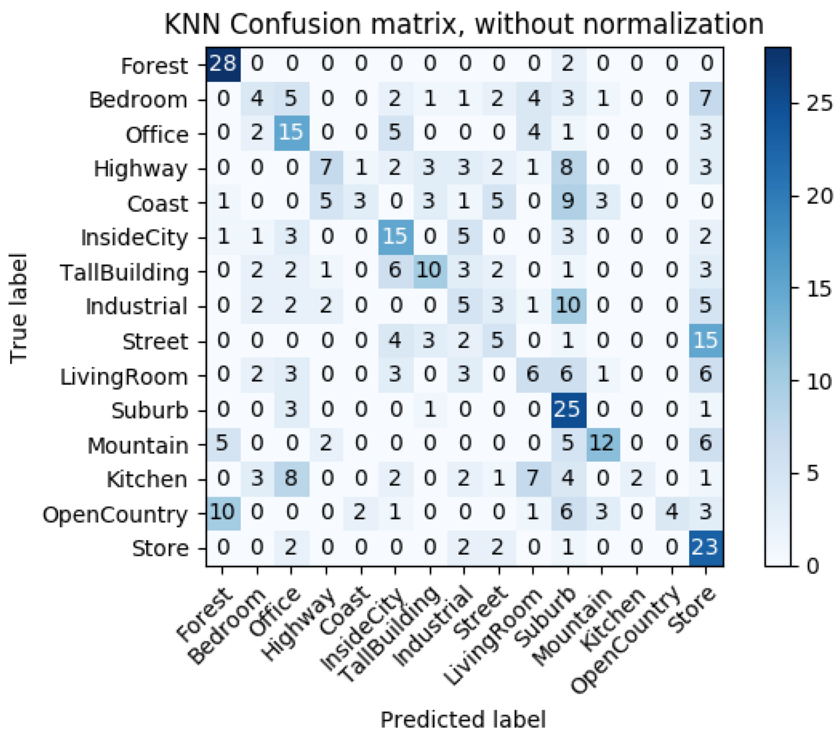
2.2 Accuracy of KNN (with k = 7): 0.3644

tweeking with k(using blue to highlight the local max):
k = 1: accuracy = 0.3333
k = 2: accuracy = 0.3444
k = 3: accuracy = 0.3422
k = 4: accuracy = 0.3533
k = 5: accuracy = 0.3511 (default)
k = 6: accuracy = 0.3511
k = 7: accuracy = 0.3644
k = 8: accuracy = 0.3489
k = 9: accuracy = 0.3556
k = 10: accuracy = 0.3467
k = 15: accuracy = 0.3422
k = 30: accuracy = 0.3289

As shown above, observed that the KNN accuracy varies between 30% to 40%, and it's swinging ups and downs like a multi-degrees graph with the tendency to reach a global max on **k=7** for accuracy of **36.44%**. If we smooth out the noises and we will see a rough bell-curve with center (highest) at **k=7** and yields toward the 2 sides.

2.3 Confusion Matrix





3 Q6

3.1 svm_classify()

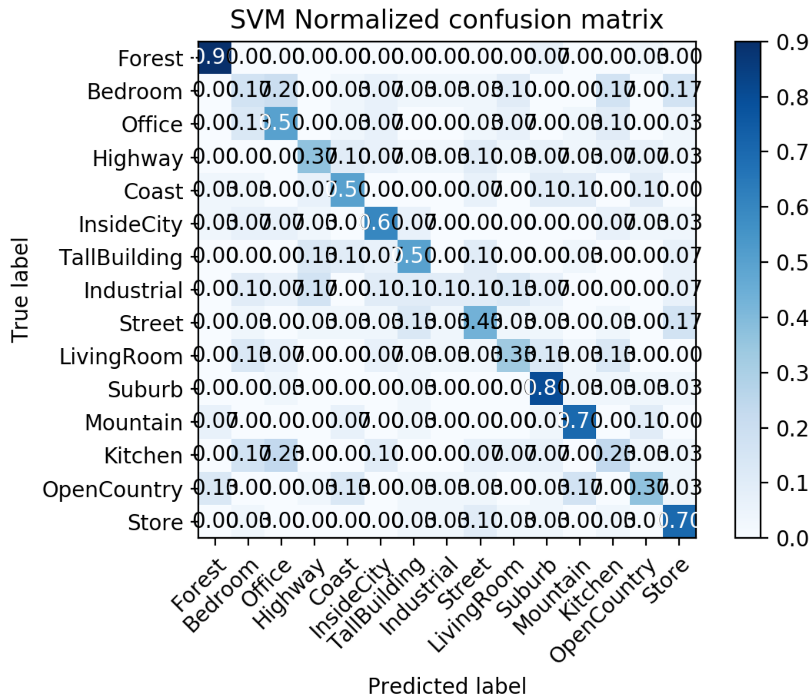
```python
def svm_classify(train_image_feats, train_labels, test_image_feats):

    '''
    Parameters
        ----------
        train_image_feats:  is an N x d matrix, where d is the dimensionality of the feature representation.
        train_labels: is an N x l cell array, where each entry is a string
                      indicating the ground truth one-hot vector for each training image.
        test_image_feats: is an M x d matrix, where d is the dimensionality of the
                          feature representation. You can assume M = N unless you've modified the starter code.

    Returns
        --------
        is an M x l cell array, where each row is a one-hot vector
        indicating the predicted category for each test image.

    Usefull funtion:

        # You can use svm from sci-kit learn.
        # Reference: https://scikit-learn.org/stable/modules/svm.html

    '''
    clf = multiclass.OneVsRestClassifier(svm.LinearSVC(C=20.0))
    clf.fit(train_image_feats, train_labels)
    predicted_labels = clf.predict(test_image_feats)
    return predicted_labels
```

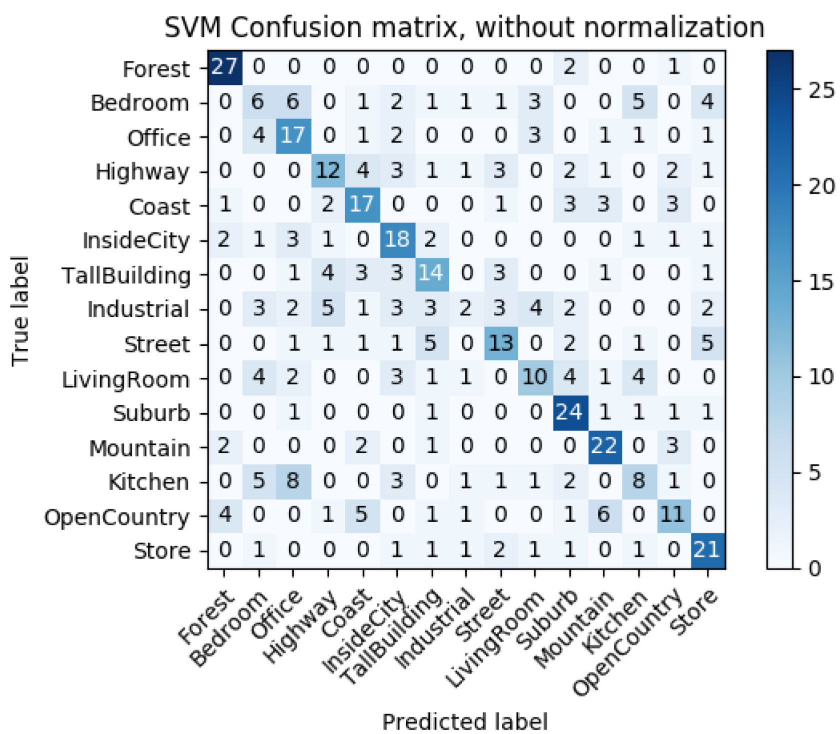
3.2 Accuracy of SVM (with C = 20.0): 0.4933

tweeking with C:
C = 0.1: accuracy = 0.3511
C = 1.0: accuracy = 0.4155 (default)
C = 1.5: accuracy = 0.4356
C = 2.0: accuracy = 0.4489
C = 4.0: accuracy = 0.4644
C = 10.0: accuracy = 0.4844
C = 19.0: accuracy = 0.4933
C = 20.0: accuracy = 0.4933
C = 21.0: accuracy = 0.4933
C = 22.0: accuracy = 0.4911
C = 25.0: accuracy = 0.4844
C = 30.0: accuracy = 0.4822
C = 40.0: accuracy = 0.4800

As shown above, observed that the SVM accuracy varies between 35% to 50%, observed the best performance at **C=19.0/20.0/21.0** for **49.33%** accuracy. It appears to be a bell-curve with the center at **C=20.0** and falls toward the two sides. The performance of using SVM classifier is also generally better than using KNN classifier.

3.3 Confusion Matrix





Calculate accuracies and plot confusion matrics:

```
print('---Evaluation---\n')
# Step 3: Build a confusion matrix and score the recognition system for
#         each of the classifiers.
# TODO: In this step you will be doing evaluation.

# 1) Calculate the total accuracy of your model by counting number
#   of true positives and true negatives over all.

accuracy_knn = np.sum(pred_labels_knn == test_labels) / len(test_labels)
accuracy_svm = np.sum(pred_labels_svm == test_labels) / len(test_labels)

print("KNN Accuracy: ", round(accuracy_knn, 4))
print("SVM Accuracy: ", round(accuracy_svm, 4))

# 2) Build a Confusion matrix and visualize it.
#   You will need to convert the one-hot format labels back
#   to their category name format.
plot_confusion_matrix_u(test_labels, pred_labels_knn, normalize=True, type='KNN')
plot_confusion_matrix_u(test_labels, pred_labels_svm, normalize=True, type='SVM')
```

```
def plot_confusion_matrix_u(test_labels, pred_labels, normalize, type): {
    # Reference: https://scikit-learn.org/stable/auto_examples/model_selection
    #   /plot_confusion_matrix.html#sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py

    plot_confusion_matrix(
        test_labels,
        pred_labels,
        classes=classes_dict,
        normalize=normalize,
        title=type
    )
}
```