Assignment 5: Scene Recognition with Bag of Words

In [4]:
```
import numpy as np
import os
import glob
import matplotlib.pyplot as plt
import seaborn as sn
from sklearn.cluster import KMeans
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import LinearSVC
from sklearn.metrics import confusion_matrix, silhouette_samples, silhouette_s
core
from tqdm import tqdm
```

```
C:\Users\mrric\Anaconda3\lib\site-packages\statsmodels\tools\_testing.py:19:
FutureWarning: pandas.util.testing is deprecated. Use the functions in the pu
blic API at pandas.testing instead.
  import pandas.util.testing as tm
```

Question 4: bags of SIFT descriptors

Question 4a: clustering SIFT descriptors with K-means

In [35]:
```
def load(ds_path):
    """ Load from the training/testing dataset.

    Parameters
    ----------
    ds_path: path to the training/testing dataset.
             e.g., sift/train or sift/test

    Returns
    -------
    image_paths: a (n_sample, 1) array that contains the paths to the descript
ors.
    labels: class labels corresponding to each image
    """
    # Grab a list of paths that matches the pathname
    files = glob.glob(os.path.join(ds_path, "*", "*.txt"))
    n_files = len(files)
    image_paths = np.asarray(files)

    # Get class labels
    classes = glob.glob(os.path.join(ds_path, "*"))
    labels = np.zeros(n_files)
    labels_text = {}

    for i, path in enumerate(image_paths):
        folder, fn = os.path.split(path)
        labels[i] = np.argwhere(np.core.defchararray.equal(classes, folder))[0
,0]
        labels_text[int(labels[i])] = path.split("\\")[1]

    # Randomize the order
    idx = np.random.choice(n_files, size=n_files, replace=False)
    image_paths = image_paths[idx]
    labels = labels[idx]
    return image_paths, labels, labels_text

def sample_descriptors(image_paths):
    """ Sample SIFT descriptors, cluster them using k-means, and return the fi
tted k-means model.
    NOTE: We don't necessarily need to use the entire training dataset. You ca
n use the function
    sample_images() to sample a subset of images, and pass them into this func
tion.

    Parameters
    ----------
    image_paths: an (n_image, 1) array of image paths.

    Returns
    -------
    descriptors: a (n_image * n_each, 128) array of sampled descriptors
    """
    n_image = len(image_paths)

    # Since want to sample tens of thousands of SIFT descriptors from differen
t images, we
```

```
    # calculate the number of SIFT descriptors we need to sample from each ima
ge.
    n_each = int(np.ceil(10000 / n_image))

    # Initialize an array of features, which will store the sampled descriptor
s
    # 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:]

        # Randomly sample n_each descriptors from sift_descriptor and store th
em into descriptors
        n,d = sift_descriptors.shape
        indices = np.random.choice(n, n_each)
        descriptors[i:i+n_each,:] = sift_descriptors[indices]

    return descriptors

def build_vocabulary(descriptors):
    # Perfom k-means clustering to cluster sampled sift descriptors into vocab
_size regions.
    print("Fitting K-means clustering")
    silhouette_scores = []
    for c in np.arange(10, 101, 10):
        kmeans = KMeans(n_clusters=c, random_state=0).fit(descriptors)
        labels = kmeans.predict(descriptors)

        # The silhouette_score gives the average value for all the samples
        # This gives a perspective into the density and separation of the form
ed clusters
        ss = silhouette_score(descriptors, labels)
        silhouette_scores.append(ss)

    plt.plot(np.arange(10, 101, 10), silhouette_scores)
    plt.ylabel('silhouette_score')
    plt.xlabel('n_clusters')
    plt.show()

    # Return fitted model with best clustering from silhouette score plot
    return KMeans(n_clusters=90, random_state=0).fit(descriptors)
```

In [18]:
```
print('Getting paths and labels for all train and test data')
train_image_paths, train_labels, train_labels_text = load("sift/train")
test_image_paths, test_labels, test_labels_text = load("sift/test")

print('Labels:')
print(test_labels_text)
```

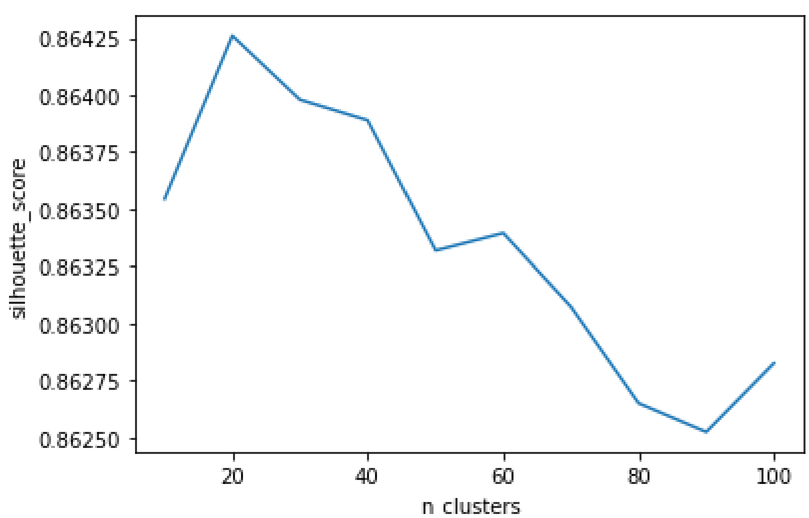
```
Getting paths and labels for all train and test data
Labels:
{0: 'Bedroom', 1: 'Coast', 2: 'Forest', 3: 'Highway', 4: 'Industrial', 5: 'In
sideCity', 6: 'Kitchen', 7: 'LivingRoom', 8: 'Mountain', 9: 'Office', 10: 'Op
enCountry', 11: 'Store', 12: 'Street', 13: 'Suburb', 14: 'TallBuilding'}
```

In [19]:
```
print('Extracting SIFT features')
descriptors = sample_descriptors(train_image_paths)
```

```
Extracting SIFT features
```

In [36]:
```
kmeans = build_vocabulary(descriptors)
```

```
Fitting K-means clustering
```



Using the elbow method to find a suitable number of clusters, it is evident that 90 clusters is a good fit for our data. We can see the the silhouette score is rather high with less than 90 clusters, and increases with more than 90 clusters.

Question 4b: representing images as bags of SIFT feature histograms

In [21]:
```
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 histogra
m.
    """
    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)

        # Assign each feature to the closest cluster center
        # Again, each feature consists of the (x, y) location and the 128-dime
nsional sift descriptor
        # You can access the sift descriptors part by features[:, 2:]
        sift_descriptors = features[:, 2:]
        predictions = kmeans.predict(sift_descriptors)

        # Build a histogram normalized by the number of descriptors
        hist, bins = np.histogram(predictions, bins=np.arange(vocab_size+1), d
ensity=True)
        image_feats[i,:] = hist

    return image_feats
```

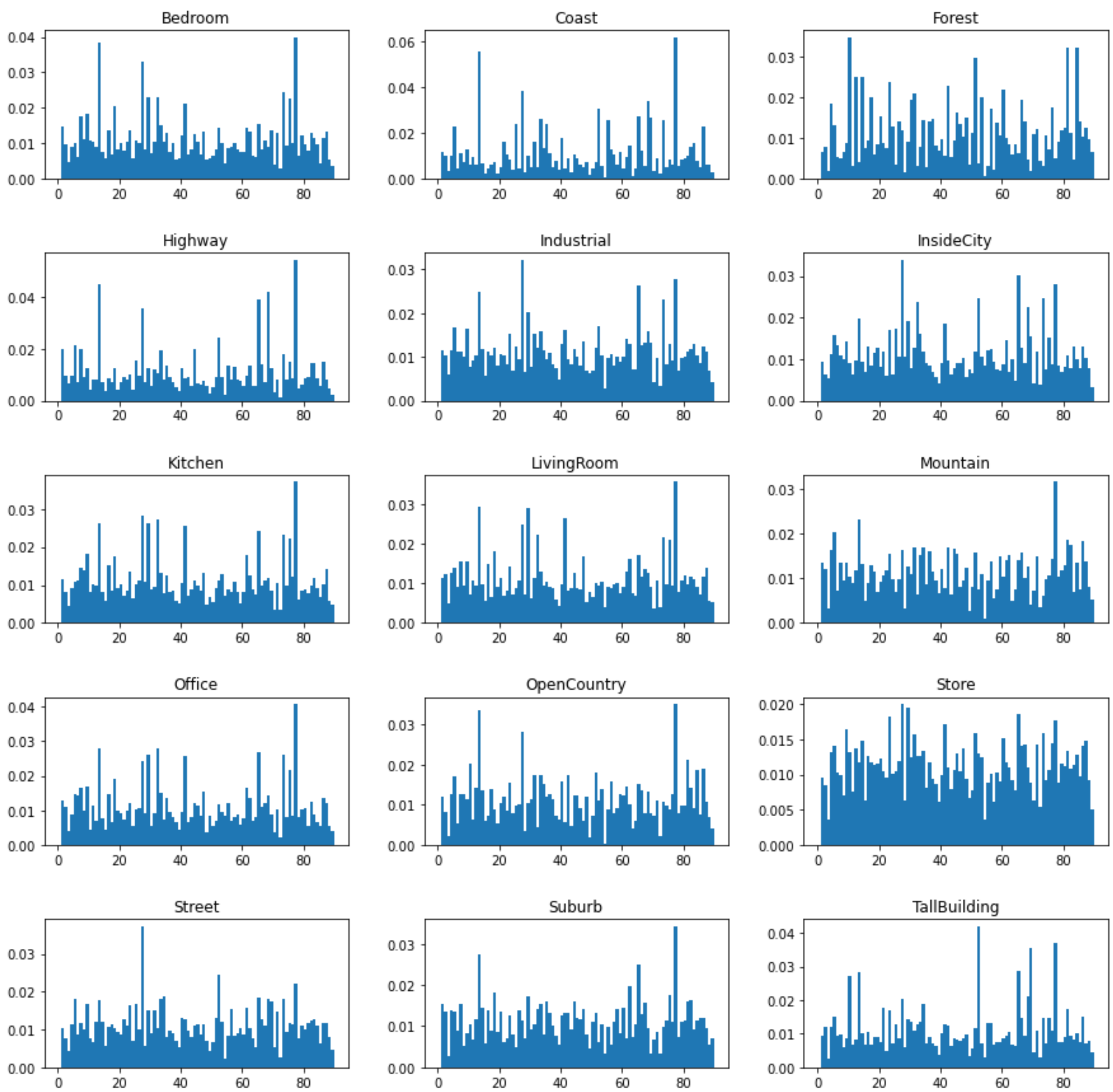
In [37]:
```
train_image_feats = get_bags_of_sifts(train_image_paths, kmeans)
test_image_feats = get_bags_of_sifts(test_image_paths, kmeans)
```

Question 4c: average histogram for each scene category

While most of the category histograms are distinct, there are some that are surprisingly alike. For example, kitchens and offices have a similar keypoint distribution, with similar values in similar bins. This is also true for mountains and open country, kitchens and living rooms, and industrial settings and inside city scenes. We can predict that the classifiers will not perform as well to differentiate between these pairs of classes. For example, the model may predict a picture to be of a living room when it is actually a picture of a kitchen.

In [38]:
```
vocab_size = kmeans.cluster_centers_.shape[0]
category_feats = np.zeros((15, vocab_size))
bins = np.arange(vocab_size+1)

for row, label in zip(train_image_feats, train_labels):
    category_feats[int(label),:] = category_feats[int(label),:] + row

fig, axs = plt.subplots(5,3, figsize=(15, 15), facecolor='w', edgecolor='k')
fig.subplots_adjust(hspace =0.5, wspace=0.25)
axs = axs.ravel()

for i, category in enumerate(category_feats):
    category_feats[i] = category / list(train_labels).count(i)
    axs[i].hist(bins[:-1], bins, weights=category_feats[i])
    axs[i].set_title(test_labels_text[i])
```

```
90
```



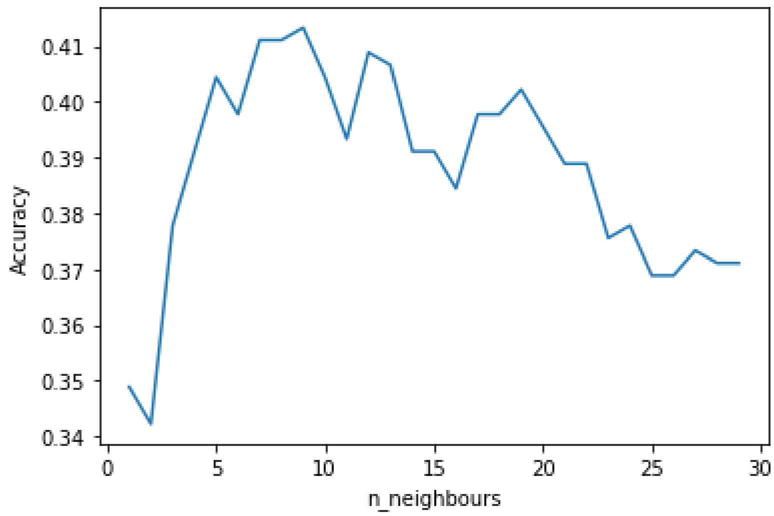
Question 5: scene recognition with KNN

In [39]:
```
def nearest_neighbor_classify(train_image_feats, train_labels, test_image_feat
s):
    """ This function will predict the category for every test image by findin
g the
    training image with most similar features. Instead of 1 nearest neighbor,
 you can
    vote based on k nearest neighbors which will increase performance (althoug
h you need
    to pick a reasonable value for k).

    Parameters
    ----------
    train_image_feats: is an N x d matrix, where d is the dimensionality of th
e 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 yo
u'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.
    """
    # Keep track of best accuracy and model
    best_acc = (1,0)
    best_model = None
    accuracies = []

    # Fit KNN classifiers on range of n_neighbours
    for nn in range(1,30):
        model = KNeighborsClassifier(n_neighbors=nn).fit(train_image_feats, tr
ain_labels)
        predicted_labels = model.predict(test_image_feats)
        acc = model.score(test_image_feats, test_labels)

        # Save model if new acc is better than current best
        accuracies.append(acc)
        if acc > best_acc[1]:
            best_acc = (nn, acc)
            best_model = model

    plt.plot(range(1,30), accuracies)
    plt.ylabel('Accuracy')
    plt.xlabel('n_neighbours')
    plt.show()

    print("Model with best test accuracy:")
    print("{} neighbours, {} accuracy".format(best_acc[0], best_acc[1]))

    print("Normalized confusion matrix (true labels vs. predicted labels):")
    cf_matrix = confusion_matrix(test_labels, best_model.predict(test_image_fe
ats), labels=list(range(0,15)))
    cm = cf_matrix.astype('float') / cf_matrix.sum(axis=1)[:, np.newaxis]
```

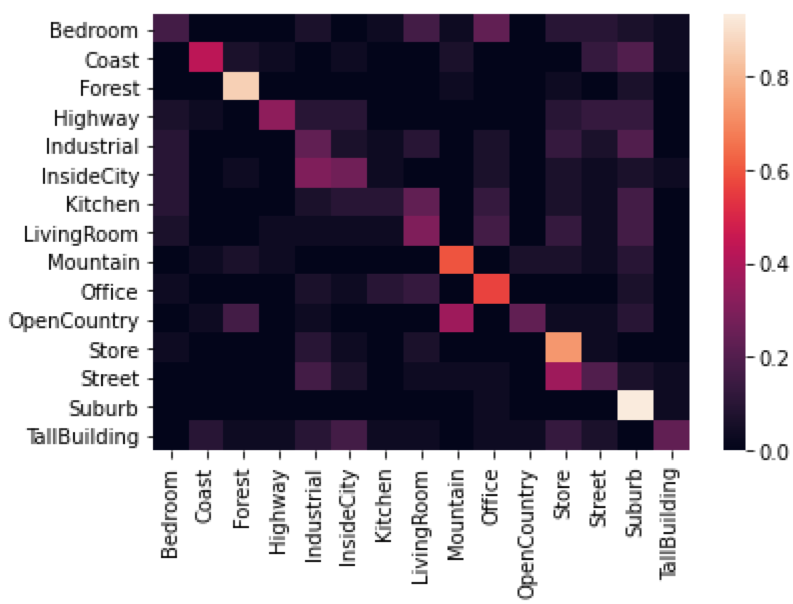
```
    axis_labels = test_labels_text.values()
    sn.heatmap(cm, xticklabels=axis_labels, yticklabels=axis_labels)
    return predicted_labels
```

In [40]:
```
print('Using nearest neighbor classifier to predict test set categories')
pred_labels_knn = nearest_neighbor_classify(train_image_feats, train_labels, t
est_image_feats)
```

```
Using nearest neighbor classifier to predict test set categories
```




```
Model with best test accuracy:
9 neighbours, 0.41333333333333333 accuracy
Normalized confusion matrix (true labels vs. predicted labels):
```



From the plot above, we can see that there is a significant increase in accuracy when we increase the number of neighbours from 1 to 9. This improvement is due to the model being less prone to overfitting, since it uses more neighbours. The model performs decently with 9-17 neighbours, worsening dramatically after 19 neighbours.

From the confusion matrix, we can confirm our suspicions from before. When the true label is "industrial", there is a high chance the classifier predicts "inside city". When the true label is "mountain", there is a high chance the classifier predicts "open country". There are also some surprising results, between "store" and "street" scenes. I am not sure why the classifier did so poorly differentiating these two classes, as their histograms (above) are rather different.

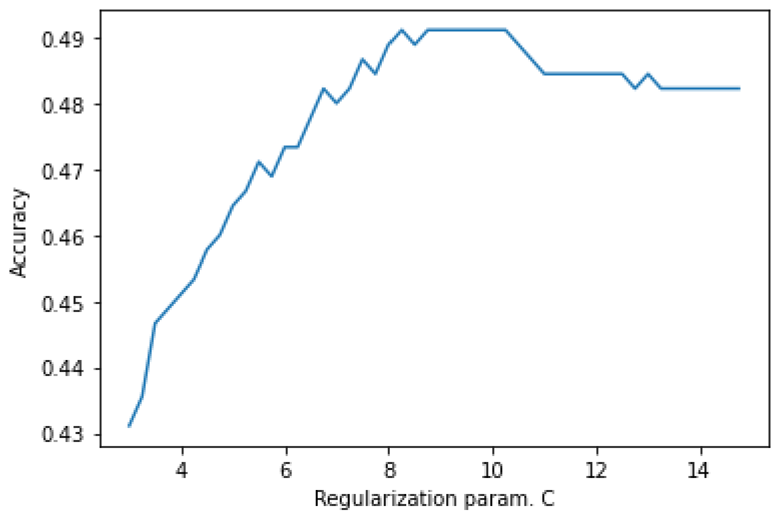
Question 6: scene recognition with 1-vs-all linear SVMs

In [43]:
```
def svm_classify(train_image_feats, train_labels, test_image_feats):
    """ This function will train a linear SVM for every category (i.e. one vs
 all) and then use the
    learned linear classifiers to predict the category of very test image. Eve
ry test feature will
    be evaluated with all 15 SVMs and the most confident SVM will "win". Confi
dence, or distance
    from the margin, is W*X + B where '*' is the inner product or dot product
 and W and B are the
    learned hyperplane parameters.

    Parameters
    ----------
    train_image_feats:  is an N x d matrix, where d is the dimensionality of t
he 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 yo
u'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.
    """
    # Keep track of best accuracy and model
    best_acc = (0,0)
    best_model = None
    accuracies = []

    # Fit linear-SVM on range of regularization param. c
    for c in np.arange(3, 15, 0.25):
        model = LinearSVC(C=c).fit(train_image_feats, train_labels)
        predicted_labels = model.predict(test_image_feats)
        acc = model.score(test_image_feats, test_labels)

        # Save model if new acc is better than current best
        accuracies.append(acc)
        if acc > best_acc[1]:
            best_acc = (c, acc)
            best_model = model

    plt.plot(np.arange(3, 15, 0.25), accuracies)
    plt.ylabel('Accuracy')
    plt.xlabel('Regularization param. C')
    plt.show()

    print("Model with best test accuracy:")
    print("C={}, {} accuracy".format(best_acc[0], best_acc[1]))

    print("Normalized confusion matrix (true labels vs. predicted labels):")
    cf_matrix = confusion_matrix(test_labels, best_model.predict(test_image_fe
ats), labels=list(range(0,15)))
    cm = cf_matrix.astype('float') / cf_matrix.sum(axis=1)[:, np.newaxis]
```

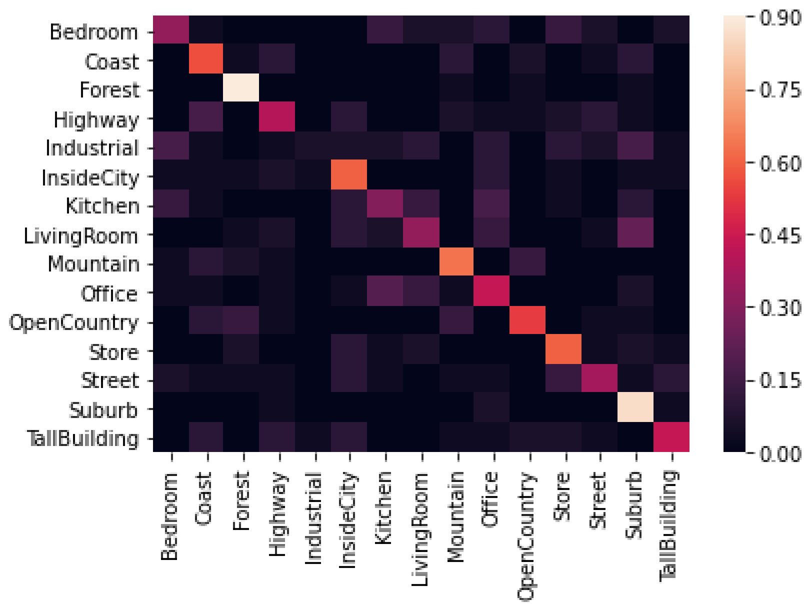
```
    axis_labels = test_labels_text.values()
    sn.heatmap(cm, xticklabels=axis_labels, yticklabels=axis_labels)
    return predicted_labels
```

In [44]:
```
print('Using support vector machine to predict test set categories')
pred_labels_svm = svm_classify(train_image_feats, train_labels, test_image_feats)
```

```
Using support vector machine to predict test set categories
```




```
Model with best test accuracy:
C=8.25, 0.4911111111111111 accuracy
Normalized confusion matrix (true labels vs. predicted labels):
```



From the plot above, we can see that performance of the classifier peaks when the regulurization parameter C equals 8.25. Regularization adjusts how robust the SVM is to variance in the data, where a low number results in a more flexible model, while a higher number results in a more robust model. 8.25 is a (relatively) large number, so we can infer that our data has a significant amount of variance, and increasing the regularization parameter helps to prevent the model from overfitting. Increasing C past 8.25 results in a decrease in accuracy, suggesting that there is too much regularization, and the model is underfitting.

From the confusion matrix, we can see that this classifier clearly does a much better job at separating and classifying the scenes than the KNN classifier. However, the model still has trouble differentiating kitchens with offices and living rooms, as we predicted from looking at the histograms above. It is also interesting to note that the classifier does very poorly on industrial scenes, having nearly no correct predictions. On the otherhand, the model correctly classifies nearly all of the forest scenes.