BBM418 - COMPUTER VISION LABORATORY ASSIGNMENT 2

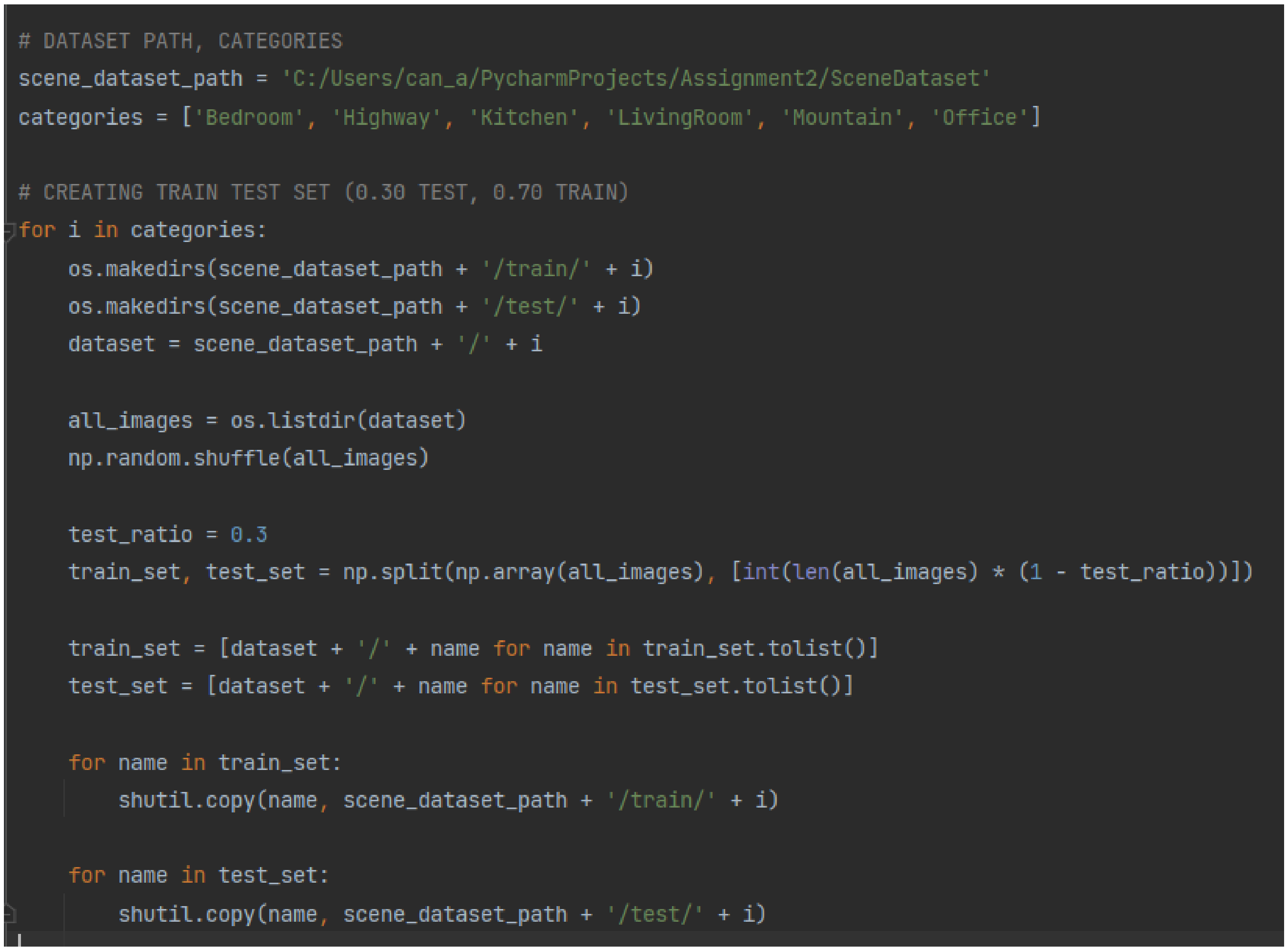
Sadık Can ACAR
21626843

April 15, 2021

1 Train Test Split

During the train-test-split phase, I first created a folder called train and test. Then I shuffled all the images and separated them into train set and test set by 0.7 and 0.3.

1.1 Code Snippet



```python
# DATASET PATH, CATEGORIES
scene_dataset_path = 'C:/Users/can_a/PycharmProjects/Assignment2/SceneDataset'
categories = ['Bedroom', 'Highway', 'Kitchen', 'LivingRoom', 'Mountain', 'Office']

# CREATING TRAIN TEST SET (0.30 TEST, 0.70 TRAIN)
for i in categories:
    os.makedirs(scene_dataset_path + '/train/' + i)
    os.makedirs(scene_dataset_path + '/test/' + i)
    dataset = scene_dataset_path + '/' + i

    all_images = os.listdir(dataset)
    np.random.shuffle(all_images)

    test_ratio = 0.3
    train_set, test_set = np.split(np.array(all_images), [int(len(all_images) * (1 - test_ratio))])

    train_set = [dataset + '/' + name for name in train_set.tolist()]
    test_set = [dataset + '/' + name for name in test_set.tolist()]

    for name in train_set:
        shutil.copy(name, scene_dataset_path + '/train/' + i)

    for name in test_set:
        shutil.copy(name, scene_dataset_path + '/test/' + i)
```

Figure 1: Code Snippet

1.2 Number of Images Per Category of Train and Test Sets

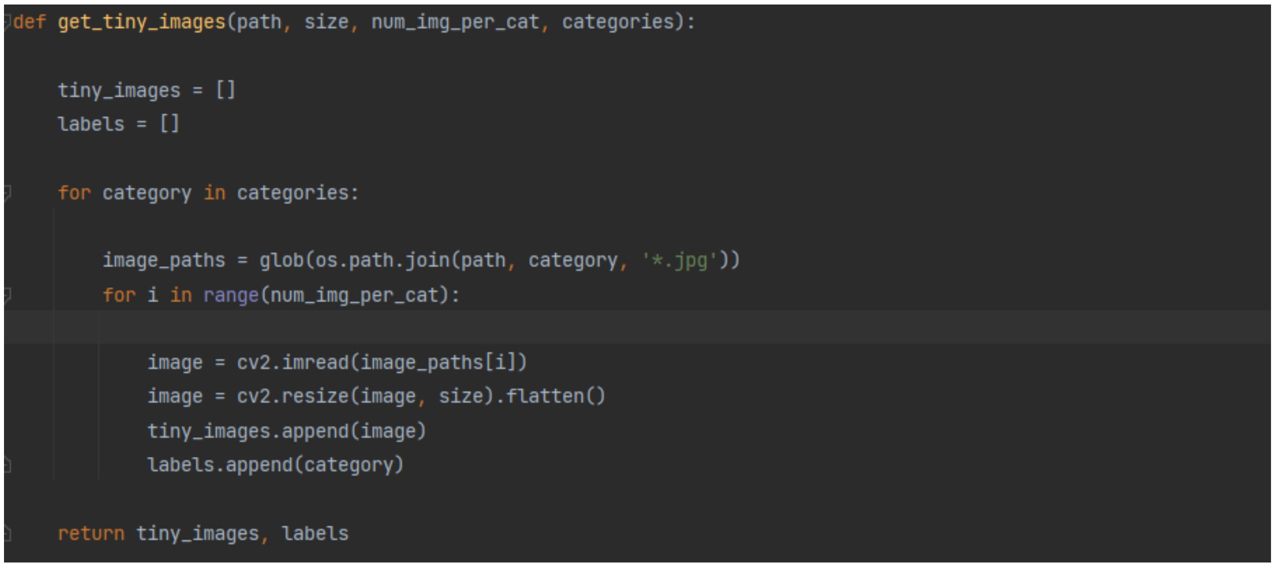
I used 60 images from each category, but the versions after the train-test split are in the table below.

	Train Set	Test Set
Bedroom	151	65
Highway	182	75
Kitchen	147	63
LivingRoom	202	87
Mountain	261	113
Office	150	65

2 Tiny Images Features and Bag of Visual Words

2.1 Tiny Images Features

During the process of getting Tiny Images, I created two arrays to hold images and categories of images. I read 60 images per category, changed their size to (16, 16) and put them in the tiny images array. I also put their categories on the labels array. With the function I wrote, I created these two arrays separately for train set and test set.

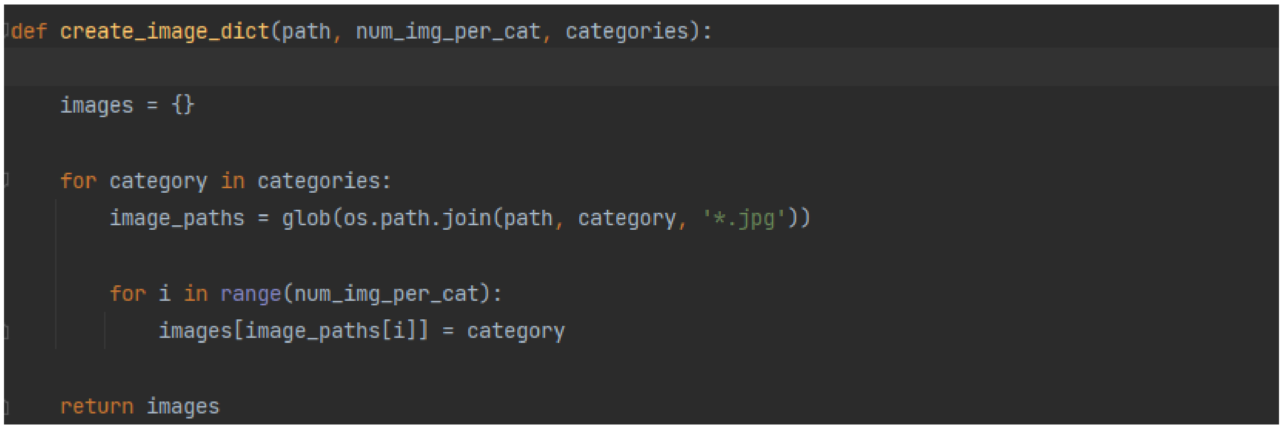


```python
def get_tiny_images(path, size, num_img_per_cat, categories):

    tiny_images = []
    labels = []

    for category in categories:

        image_paths = glob(os.path.join(path, category, '*.jpg'))
        for i in range(num_img_per_cat):

            image = cv2.imread(image_paths[i])
            image = cv2.resize(image, size).flatten()
            tiny_images.append(image)
            labels.append(category)

    return tiny_images, labels
```

Figure 2: Getting Tiny Images

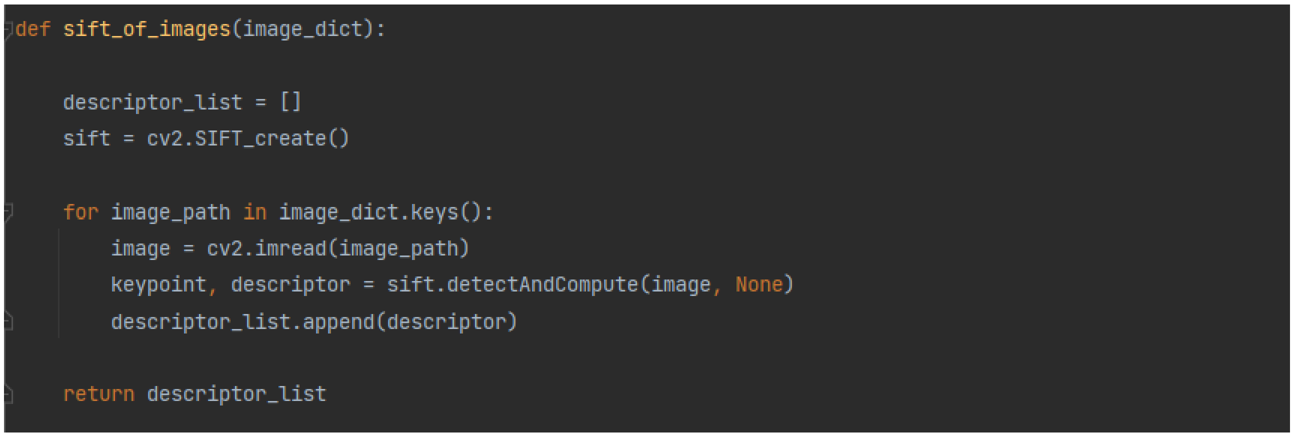
2.2 Bag of Visual Words

In the Bag of Visual Words stage, I first created an image dictionary using train and test set paths, number image per category and categories. Image dictionaries keep the path of the images as the key and the categories as the value. Then I extracted these image dictionaries with sift and created descriptor lists. I obtained vocabulary by applying K-means clustering to the descriptor list I created for the train set. I created image histograms using vocabulary and descriptor lists that I created for this train-test set.



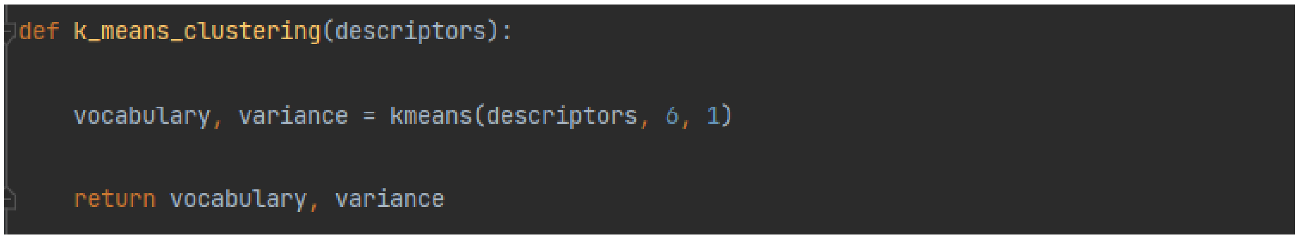
```python
def create_image_dict(path, num_img_per_cat, categories):

    images = {}

    for category in categories:
        image_paths = glob(os.path.join(path, category, '*.jpg'))

        for i in range(num_img_per_cat):
            images[image_paths[i]] = category

    return images
```

Figure 3: Creating Image Dictionary that holds Image Paths and Categories



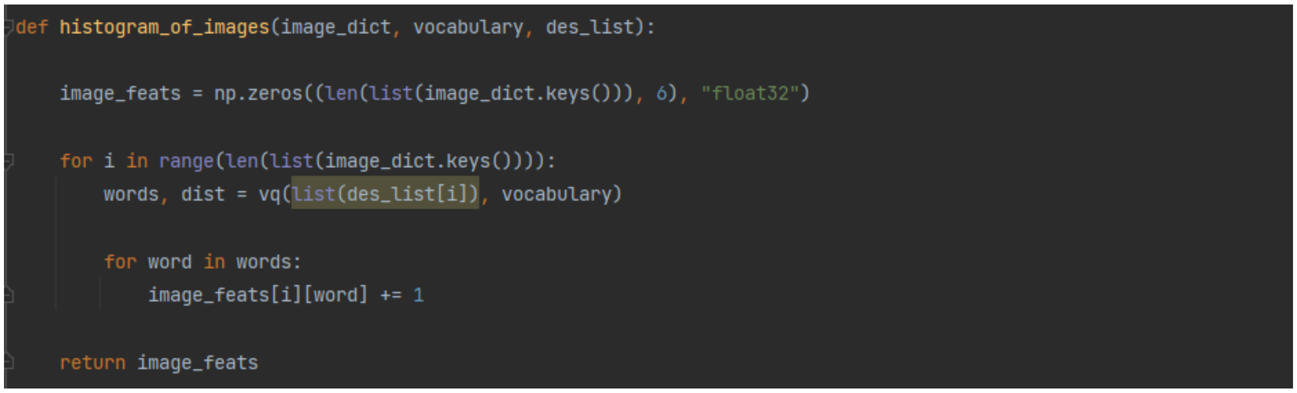
```python
def sift_of_images(image_dict):

    descriptor_list = []
    sift = cv2.SIFT_create()

    for image_path in image_dict.keys():
        image = cv2.imread(image_path)
        keypoint, descriptor = sift.detectAndCompute(image, None)
        descriptor_list.append(descriptor)

    return descriptor_list
```

Figure 4: Creating Descriptor List from Image Dictionaries' Keys



```python
def k_means_clustering(descriptors):

    vocabulary, variance = kmeans(descriptors, 6, 1)

    return vocabulary, variance
```

Figure 5: Creating Vocabulary from Descriptor List with K-means



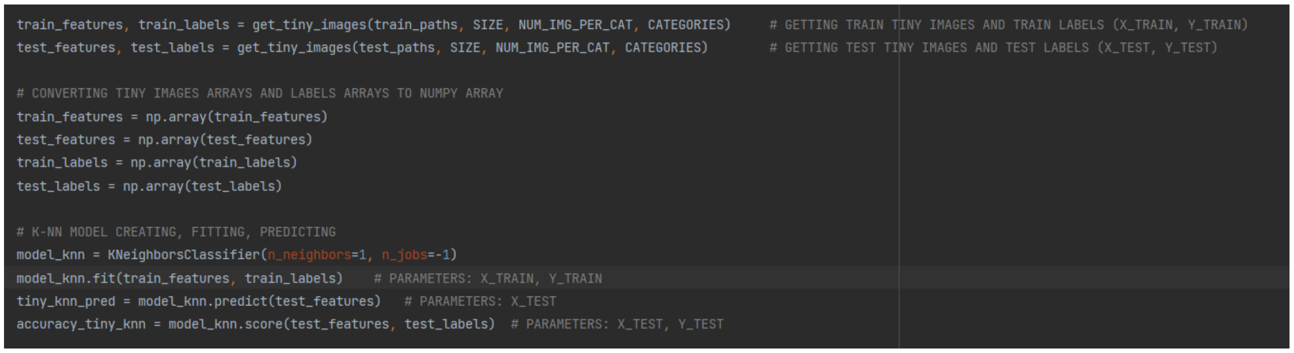
```python
def histogram_of_images(image_dict, vocabulary, des_list):

    image_feats = np.zeros((len(list(image_dict.keys())), 6), "float32")

    for i in range(len(list(image_dict.keys()))):
        words, dist = vq(list(des_list[i]), vocabulary)

        for word in words:
            image_feats[i][word] += 1

    return image_feats
```

Figure 6: Creating Histograms from Vocabulary and Descriptor Lists

3 K-Nearest Neighbors and Linear Support Vector Machine

3.1 Tiny Images Features

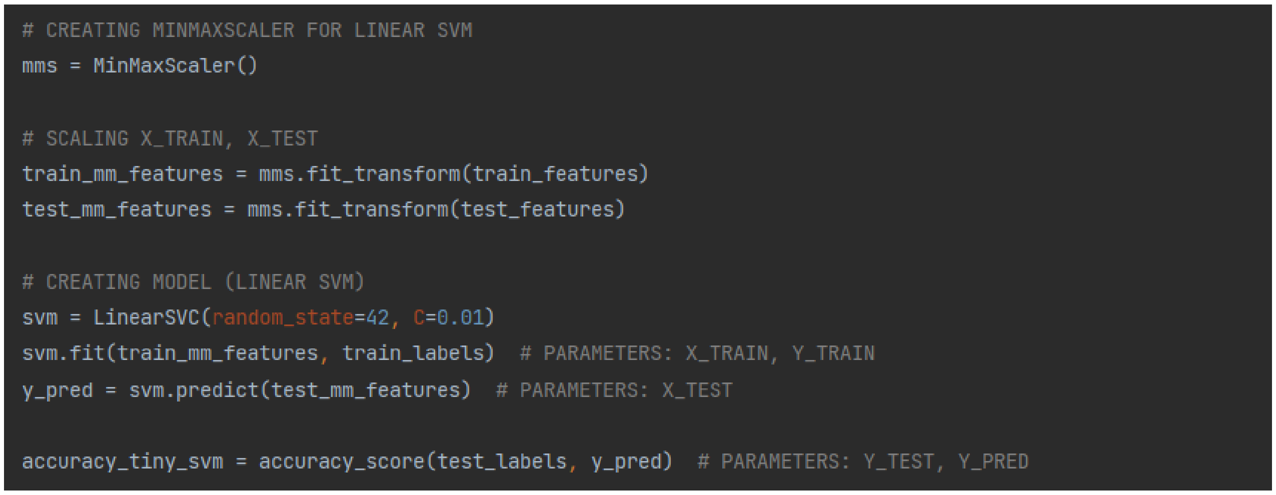
On Tiny Images, when I was implementing K-Nearest Neighbors, I first created my model. Then I fitted my model with train-features (X-train) and train-labels (y-train). I predict my model using test-features (X-test). Finally, I calculated my accuracy score using test-features (X-test) and test-labels (y-test).



```python
train_features, train_labels = get_tiny_images(train_paths, SIZE, NUM_IMG_PER_CAT, CATEGORIES)     # GETTING TRAIN TINY IMAGES AND TRAIN LABELS (X_TRAIN, Y_TRAIN)
test_features, test_labels = get_tiny_images(test_paths, SIZE, NUM_IMG_PER_CAT, CATEGORIES)        # GETTING TEST TINY IMAGES AND TEST LABELS (X_TEST, Y_TEST)

# CONVERTING TINY IMAGES ARRAYS AND LABELS ARRAYS TO NUMPY ARRAY
train_features = np.array(train_features)
test_features = np.array(test_features)
train_labels = np.array(train_labels)
test_labels = np.array(test_labels)

# K-NN MODEL CREATING, FITTING, PREDICTING
model_knn = KNeighborsClassifier(n_neighbors=1, n_jobs=-1)
model_knn.fit(train_features, train_labels)    # PARAMETERS: X_TRAIN, Y_TRAIN
tiny_knn_pred = model_knn.predict(test_features)   # PARAMETERS: X_TEST
accuracy_tiny_knn = model_knn.score(test_features, test_labels)  # PARAMETERS: X_TEST, Y_TEST
```

Figure 7: K-NN on Tiny Images

On Tiny Images, while implementing the Linear Support Vector Machine, I first created a scaler (mms = MinMaxScaler). Afterwards, I normalized my train-features (X-train) and test-features (X-tests). I built my model and fit it with train-mm-features (X-train) and train-labels (y-train). Finally, I predict with test-mm-features (X-test) and calculate my accuracy score with test-labels (y-test) and my predict result (y-pred).

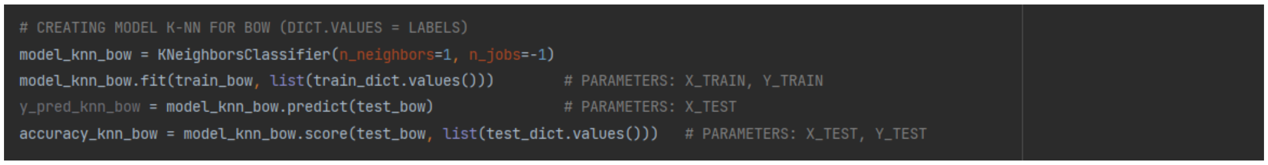


```
# CREATING MINMAXSCALER FOR LINEAR SVM
mms = MinMaxScaler()

# SCALING X_TRAIN, X_TEST
train_mm_features = mms.fit_transform(train_features)
test_mm_features = mms.fit_transform(test_features)

# CREATING MODEL (LINEAR SVM)
svm = LinearSVC(random_state=42, C=0.01)
svm.fit(train_mm_features, train_labels)  # PARAMETERS: X_TRAIN, Y_TRAIN
y_pred = svm.predict(test_mm_features)  # PARAMETERS: X_TEST

accuracy_tiny_svm = accuracy_score(test_labels, y_pred)  # PARAMETERS: Y_TEST, Y_PRED
```

Figure 8: Linear SVM on Tiny Images

3.2 Bag of Visual Words

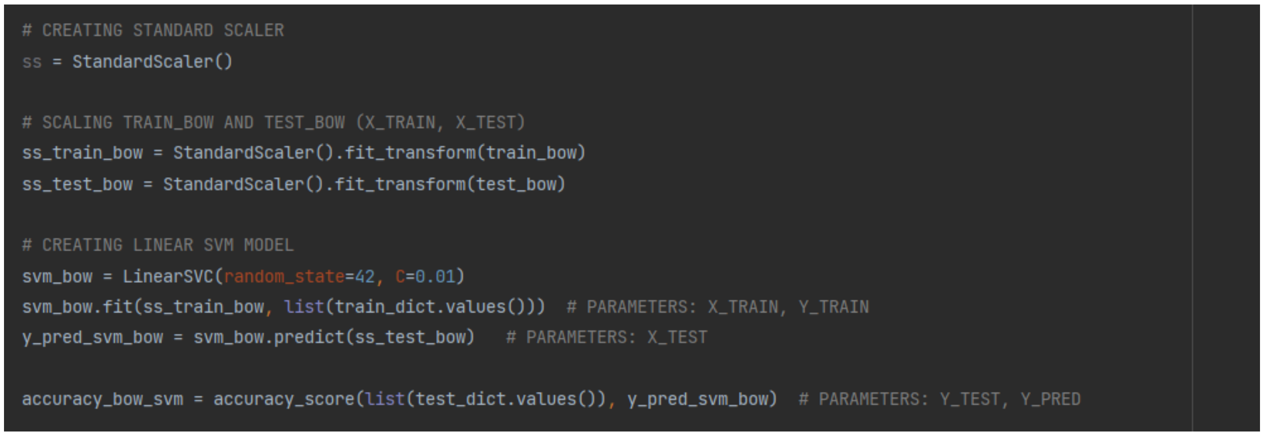
While implementing K-Nearest Neighbors in Bag of Visual Words, I first created my model. Later, I fit it with train-bow (X-train) which hold histograms of train images in it, and train-dict-values (categories) (y-train). I calculated my accuracy score with test-bow (X-test) which hold the histograms of the test images and test-dict-values (categories) (y-test).



```
# CREATING MODEL K-NN FOR BOW (DICT.VALUES = LABELS)
model_knn_bow = KNeighborsClassifier(n_neighbors=1, n_jobs=-1)
model_knn_bow.fit(train_bow, list(train_dict.values()))        # PARAMETERS: X_TRAIN, Y_TRAIN
y_pred_knn_bow = model_knn_bow.predict(test_bow)               # PARAMETERS: X_TEST
accuracy_knn_bow = model_knn_bow.score(test_bow, list(test_dict.values()))   # PARAMETERS: X_TEST, Y_TEST
```

Figure 9: K-NN on Bag of Visual Words

While implementing the Linear Support Vector Machine in Bag of Visual Words, I created a scaler. (ss = StandardScaler) I normalized the arrays that hold the histograms of the train and test images. (ss-train-bow, ss-test-bow) After creating my model, I fit it with ss-train-bow (X-train) and train-dict-values (y-train). I predict with ss-test-bow (X-test). I found the accuracy score with test-dict-values (categories) (y-test) and y-pred-svm-bow (y-pred).



```
# CREATING STANDARD SCALER
ss = StandardScaler()

# SCALING TRAIN_BOW AND TEST_BOW (X_TRAIN, X_TEST)
ss_train_bow = StandardScaler().fit_transform(train_bow)
ss_test_bow = StandardScaler().fit_transform(test_bow)

# CREATING LINEAR SVM MODEL
svm_bow = LinearSVC(random_state=42, C=0.01)
svm_bow.fit(ss_train_bow, list(train_dict.values()))  # PARAMETERS: X_TRAIN, Y_TRAIN
y_pred_svm_bow = svm_bow.predict(ss_test_bow)   # PARAMETERS: X_TEST

accuracy_bow_svm = accuracy_score(list(test_dict.values()), y_pred_svm_bow)  # PARAMETERS: Y_TEST, Y_PRED
```

Figure 10: Linear SVM on Bag of Visual Words

4 Accuracy Scores and Confusion Matrices

When calculating the accuracy scores, I used the accuracy-score metric from sklearn.metrics as it can be displayed in the simplest and most beautiful way. I used the plot-confusion-matrix function from sklearn.metrics while plotting the confusion matrix.

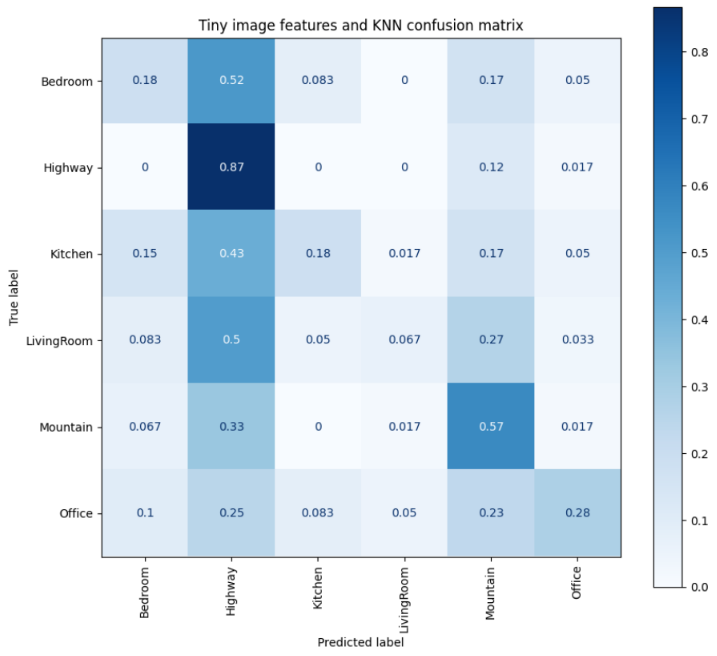


Figure 11: Tiny Images - K-NN -¿ Confusion Matrix

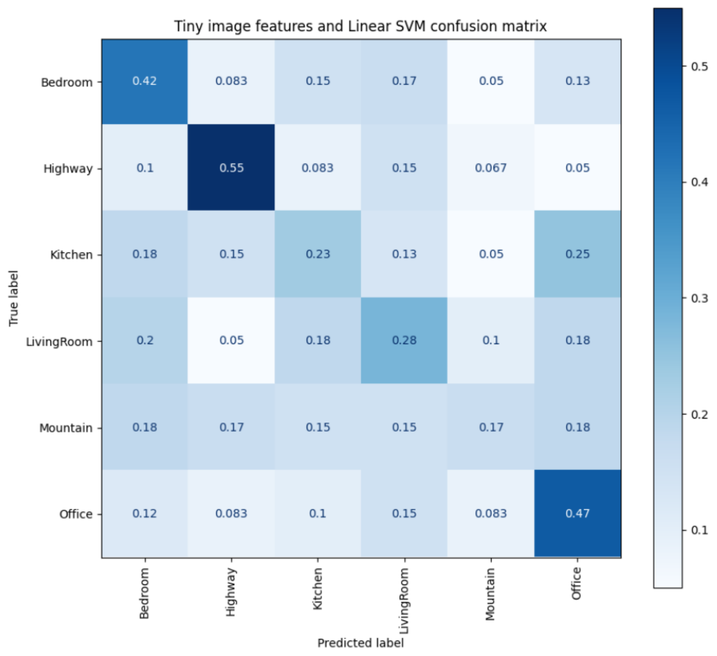


Figure 12: Tiny Images - Linear SVM -¿ Confusion Matrix

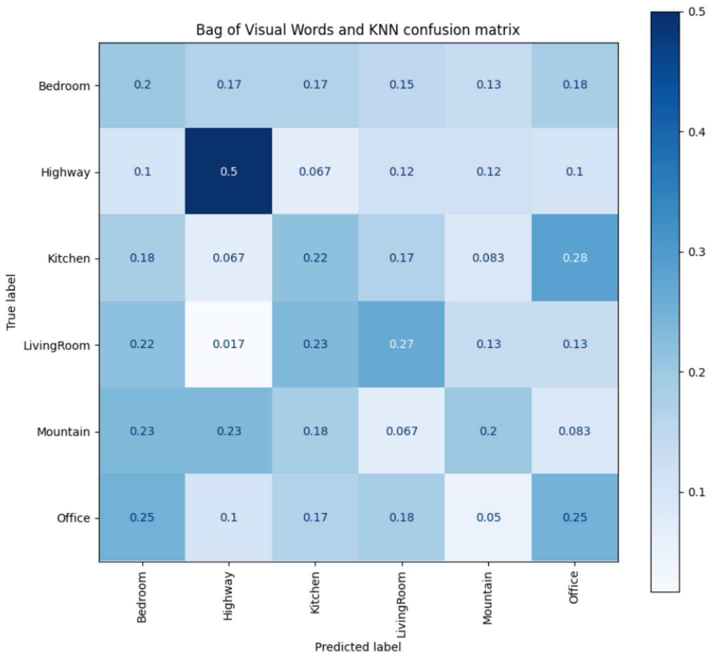


Figure 13: Bag of Visual Words - K-NN - Confusion Matrix

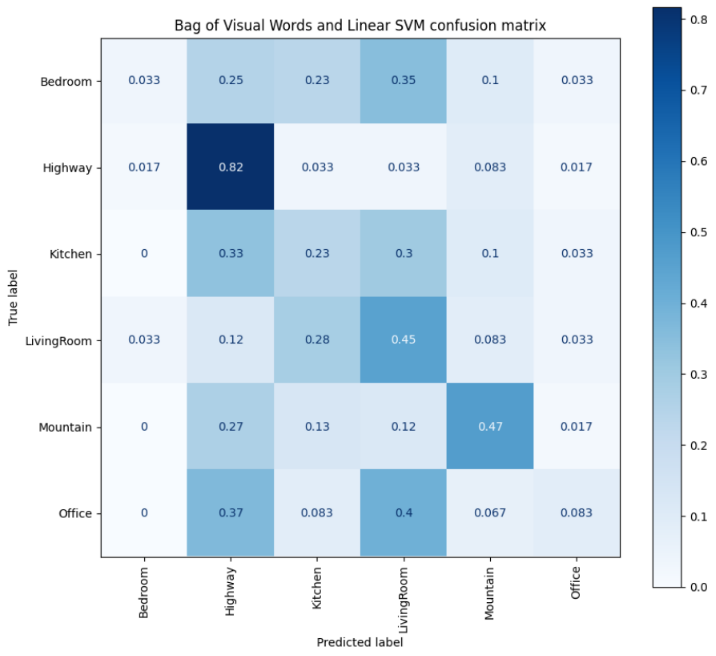


Figure 14: Bag of Visual Words - Linear SVM - Confusion Matrix

5 True-False Predicted Images

Correctly estimated images from each category are below. The results in K-NN on Tiny Image with the highest accuracy score are discussed.

5.1 True Predicted Bedrooms

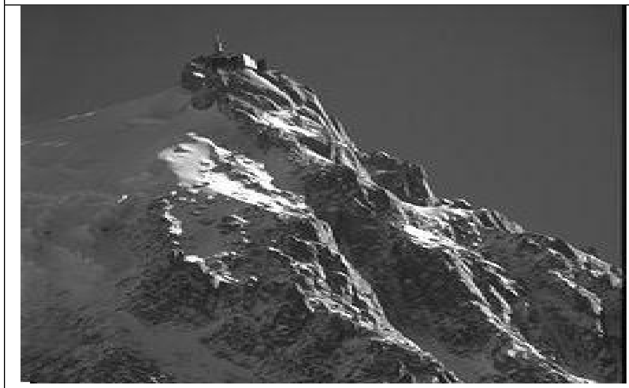
5.2 True Predicted Highways

5.3 True Predicted Kitchen

5.4 True Predicted Living Rooms

Although the results in K-NN on Tiny Image have the highest accuracy score, Living Rooms only have 4 correct predictions.

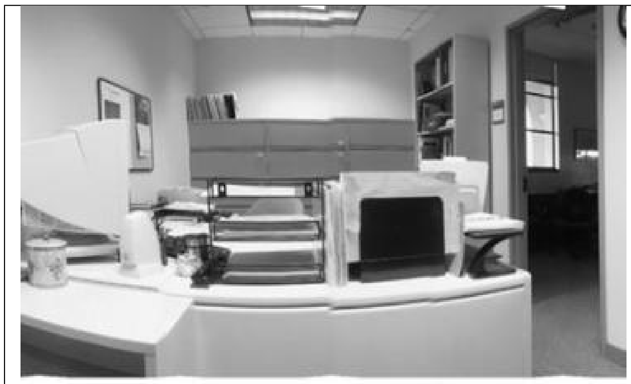
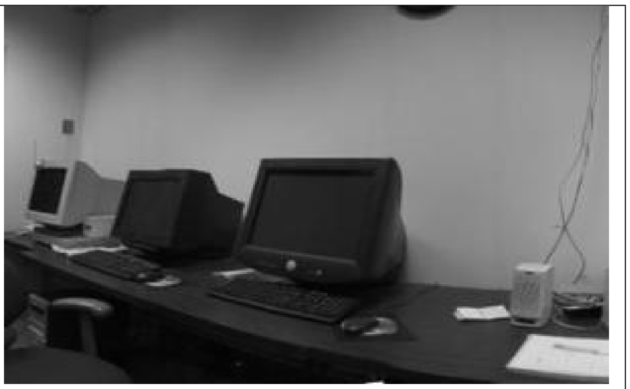
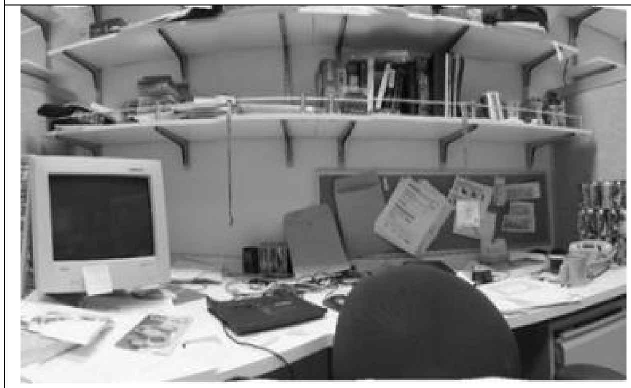
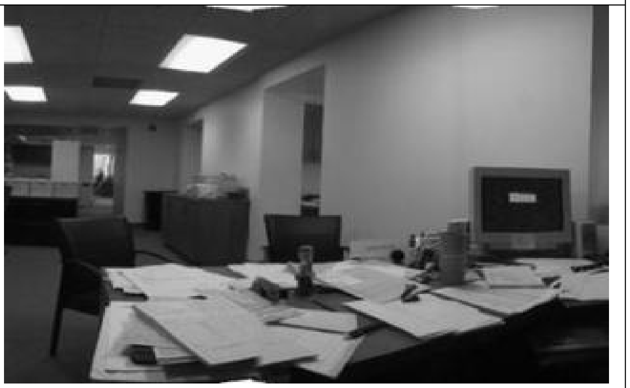
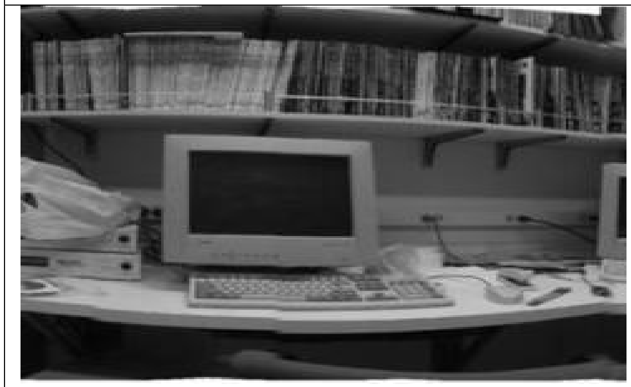
5.5 True Predicted Mountains

5.6 True Predicted Offices

5.7 False Predicted Images

The images here are all taken from the Office category and have been estimated incorrectly.
1. Predicted as Bedroom, 2. Predicted as Highway, 3. Predicted as Kitchen, 4. Predicted as Mountain,
5. Predicted as Mountain
Incorrect estimates may be due to similarities in the images of the category in which it is estimated. Similar protrusions, similar colors, similar articles, similar neighboring features, etc.

6 as a Conclusion

Looking at the accuracy scores and confusion matrices, I decided that the most accurate and well-working algorithm was the K-NN algorithm applied on Tiny Images. In fact, Linear SVM applied on Bag of Visual Words sometimes gave better results than Tiny - K-NN, but I don't think it is a good algorithm because it gives different results every time I run the code. The reason may be that it produces different visual words each time.