In [1]:
```
import os
import time
import numpy as np
```

In [2]:
```
def load_embeddings(embeddings_path):
    embeddings = {}
    with open(embeddings_path, 'r') as file:
        for line in file:
            splits = line.split()
            word = splits[0]
            coords = np.asarray(splits[1:], dtype='float32')
            embeddings[word] = coords
    return embeddings
```

In [3]:
```
word_embeddings = load_embeddings('./glove.6B.300d.txt')
```

In [4]:
```
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.vgg16 import preprocess_input
```

In [5]:
```
def get_word_embedding(category, word_embeddings, wordvec_size=300):
    # Splitting multiword categories (e.g. potted_plants)
    vector = np.zeros(shape=wordvec_size)
    num_of_categories = 0
    for cat in category.split("_"):
        if cat in word_embeddings:
            vector += word_embeddings[cat]
            num_of_categories += 1
    if num_of_categories > 0:
        return vector / num_of_categories
    return vector

def preprocess_images(dataset_dir, word_embeddings, wordvec_size=300):
    image_list = []
    labels_list = []
    paths_list = []
    categories_list = []
    categories = os.listdir(dataset_dir)

    for category in categories:
        word_embedding = get_word_embedding(category, word_embeddings, wordvec_
        image_names = os.listdir(os.path.join(dataset_dir, category))

        for img_name in image_names:
            full_path = os.path.join(dataset_dir, category, img_name)

            img_pil = image.load_img(full_path, target_size=(224, 224))
            img_raw = image.img_to_array(img_pil)
            img = preprocess_input(img_raw) # VGG16 image preprocessing

            image_list.append(img)
            labels_list.append(word_embedding)
            paths_list.append(full_path)
            categories_list.append(category)

    return np.array(image_list), np.array(labels_list), paths_list, categories
```

In [6]:
```
images_vgg16, image_embeddings, image_paths, categories_list = \
    preprocess_images('./dataset', word_embeddings)
```

In [7]:
```
images_vgg16.shape
```

Out[7]:
```
(1000, 224, 224, 3)
```

In [8]:
```
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers
```

```
from tensorflow.keras.layers import Dense, BatchNormalization, Activation, Drop
from tensorflow.keras.losses import cosine_similarity
```

In [9]:
```
def hybrid_model_backbone(intermediate_dim=2000, word_embedding_dim=300):
    vgg16 = VGG16(input_shape=images_vgg16.shape[1:])
    x = vgg16.get_layer('fc2').output

    for layer in vgg16.layers:
        layer.trainable = False

    x = Dense(intermediate_dim, name="dense1")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)
    x = Dropout(0.5)(x)

    x = Dense(word_embedding_dim, name="dense2")(x)
    outputs = BatchNormalization()(x)

    model = Model(inputs=[vgg16.input], outputs=outputs)
    # https://faroit.com/keras-docs/2.0.8/optimizers/
    sgd = optimizers.SGD(decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(optimizer=sgd, loss=cosine_similarity)
    return model
```

In [10]:
```
hybrid_model = hybrid_model_backbone()
```

In [11]:
```
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.models import load_model
```

In [12]:
```
import json
import pandas as pd
```

In [13]:
```
def get_model_trained(model, X, Y, model_path, train_model=False):
    num_of_instances = X.shape[0]
    train_validation_size = 0.9

    X, Y = shuffle(X, Y, random_state=7)
    df = pd.DataFrame(data=X.reshape(num_of_instances, -1))

    # Split dataset for training_validation and testing
    X_train_validation, X_test, Y_train_validation, Y_test = \
        train_test_split(df, Y, train_size=train_validation_size, random_state=
    test_data = (X_test, Y_test)

    # On Thinkpad 480 (i7-8650u, 32GB of RAM, Nidia MX150) it takes 1h and 52mi
    if train_model:
        epochs, batch_size = (50, 32)
        validation_size = 0.2

        # Convert remaining data back to numpy array
        num_of_train_validation_instances = int(train_validation_size*num_of_in
        X_train_validation_shape = (num_of_train_validation_instances,)+X.shape
        X_train_validation = X_train_validation.values.reshape(X_train_validati

        # Split dataset for training and validation
        X_train, X_validation, Y_train, Y_validation = \
            train_test_split(X_train_validation, Y_train_validation,
                             test_size=validation_size, random_state=7)

        # Train model (save history and checkpoint object)
        checkpointer = ModelCheckpoint(
            filepath='best.hdf5', verbose=1, save_best_only=True)
        history = model.fit(X_train, Y_train,
                            validation_data=(X_validation, Y_validation),
                            epochs=epochs,
                            batch_size=batch_size,
                            callbacks=[checkpointer])
```

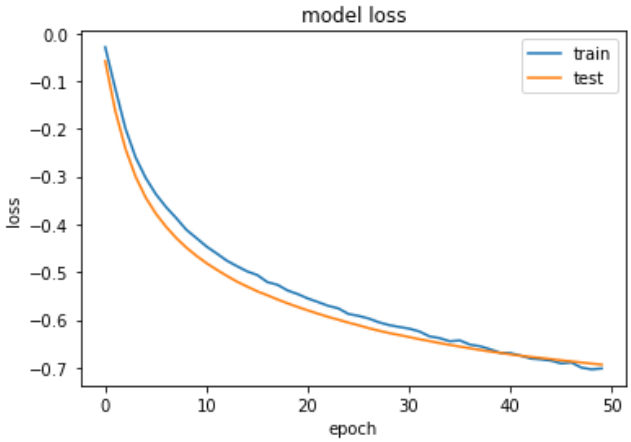
```
        model.save(model_path)
        with open("history_50_epochs.json", 'w') as f:
            json.dump(history.history, f)
        return model, history.history, test_data

    # Assuming that there is a model in a given path.
    else:
        history = None
        with open("history.json", 'r') as f:
            history = json.loads(f.read())
        return load_model(model_path), history, test_data
```

In [14]:
```
hybrid_model, history, test_data = get_model_trained(hybrid_model,
                                                     images_vgg16,
                                                     image_embeddings,
                                                     "./model_with_history_50_e
                                                     train_model=True)
```

```
Epoch 1/50
23/23 [==============================] - ETA: 0s - loss: -0.0290
Epoch 00001: val_loss improved from inf to -0.05834, saving model to best.hdf5
23/23 [==============================] - 127s 6s/step - loss: -0.0290 - val_lo
ss: -0.0583
Epoch 2/50
23/23 [==============================] - ETA: 0s - loss: -0.1162
Epoch 00002: val_loss improved from -0.05834 to -0.16283, saving model to bes
t.hdf5
23/23 [==============================] - 129s 6s/step - loss: -0.1162 - val_lo
ss: -0.1628
Epoch 3/50
23/23 [==============================] - ETA: 0s - loss: -0.1994
Epoch 00003: val_loss improved from -0.16283 to -0.24165, saving model to bes
t.hdf5
23/23 [==============================] - 133s 6s/step - loss: -0.1994 - val_lo
ss: -0.2417
Epoch 4/50
23/23 [==============================] - ETA: 0s - loss: -0.2598
```

In [15]:
```
import matplotlib.pyplot as plt
plt.plot(history['loss'])
plt.plot(history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```



In [16]:
```
from annoy import AnnoyIndex
```

In [17]:
```
def get_annoy_index(vectors_list, number_of_trees=20, dims=300):
```

```
    annoy_index = AnnoyIndex(dims, 'angular')
    for i, vec in enumerate(vectors_list):
        annoy_index.add_item(i, vec)
    annoy_index.build(number_of_trees)
    return annoy_index

def generate_word_annoy_index(word_embeddings):
    word_index = dict(enumerate(word_embeddings))
    word_embeddings_list = [word_embeddings[word] for word in word_index.values
    annoy_index = get_annoy_index(word_embeddings_list)
    return annoy_index, word_index
```

In [18]:
```
# This might take minute or two (since it is a 400k corpus of 300d words).
word_annoy_index, word_index = generate_word_annoy_index(word_embeddings)
```

In [ ]:
```
```

In [19]:
```
def generate_image_embeddings(image_paths, model):
    images = np.zeros(shape=images_vgg16.shape)
    image_path_index = dict(enumerate(image_paths))

    for ind, path in image_path_index.items():
        img_pil = image.load_img(path, target_size=images_vgg16.shape[1:-1])
        img_raw = image.img_to_array(img_pil)
        images[ind, :, :, :] = img_raw

    image_embeddings_list = model.predict(preprocess_input(images))
    return image_embeddings_list, image_path_index

def load_image_embeddings(vectors_filename, mapping_filename):
    image_embeddings = np.load("%s.npy" % vectors_filename)
    with open("%s.json" % mapping_filename) as f:
        image_path_index = json.load(f)
        return image_embeddings, {int(k): v for k, v in image_path_index.items(

def save_image_embeddings(image_embeddings_filename,
                          image_embeddings,
                          mapping_filename,
                          image_path_index):
    np.save("%s.npy" % image_embeddings_filename, image_embeddings)
    with open("%s.json" % mapping_filename, 'w') as index_file:
        json.dump(image_path_index, index_file)
```

In [20]:
```
def nearest_neighbors(vector, annoy_index, item_index, k=10):
    distances = annoy_index.get_nns_by_vector(vector, k, include_distances=True
    return [[a, item_index[a], distances[1][i]] for i, a in enumerate(distances
```

In [ ]:
```
```

In [21]:
```
def get_hybrid_embeddings(generate_embeddings=False):
    image_hybrid_embeddings_filepath = "./image_embeddings_with_history_50_epoc
    image_path_indexes_filepath = "./image_path_indexes_with_history_50_epochs"
    # Generating embedding might take 5-10min.
    if generate_embeddings:
        image_hybrid_embeddings, image_path_index = \
            generate_image_embeddings(image_paths, hybrid_model)
        save_image_embeddings(image_hybrid_embeddings_filepath,
                              image_hybrid_embeddings,
                              image_path_indexes_filepath,
                              image_path_index)
        return image_hybrid_embeddings, image_path_index
    else:
        image_hybrid_embeddings, image_path_index = \
            load_image_embeddings(image_hybrid_embeddings_filepath,
                                  image_path_indexes_filepath)
    return image_hybrid_embeddings, image_path_index
```

In [22]:
```
image_hybrid_embeddings, image_path_index = get_hybrid_embeddings(generate_embe
image_annoy_index = get_annoy_index(image_hybrid_embeddings, number_of_trees=10
```

Utility functions

In [23]:
```
from IPython.display import Image
from IPython.display import HTML, display
import urllib.request
```

```
# Assuming there are global variables (i.e. cells above did run):
#   word_embeddings, word_annoy_index, word_index,
#   hybrid_model, image_annoy_index, image_path_index
```

In [ ]:
```
```

In [24]:
```
def display_similar_images(mean_embedding, image_path=None):
    closest_images = nearest_neighbors(mean_embedding, image_annoy_index, image

    html_str = "<script>$('div.cell.selected').next().height(100);</script>\n"
    if image_path is not None:
        html_str += "<h1>Our input image</h1>"
        html_str += "<img src='%s'>" % image_path
        html_str += "<h1>Similar images</h1>"
    html_str += "<table>"

    for i in range(0, 5):
        left_cell = closest_images[2*i][1]
        html_str += "<tr><td><img src='%s'></td>" % left_cell
        right_cell = closest_images[2*i+1][1]
        html_str += "<td><img src='%s'></td></tr>" % right_cell
    html_str += "</table>\n"

    display(HTML(html_str))

def search_by_text(text):
    mean_embedding = np.mean([word_embeddings[word] for word in text.split()],
    display_similar_images(mean_embedding)

def get_image_labels(image_path, display_image=False, k=10):
    if display_image:
        display(Image(filename=image_path))

    images = np.zeros(shape=(1,)+images_vgg16.shape[1:])
    img = image.load_img(image_path, target_size=images_vgg16.shape[1:-1])
    x_raw = image.img_to_array(img)
    images[0] = np.expand_dims(x_raw, axis=0)

    inputs = preprocess_input(images)

    image_features = hybrid_model.predict(inputs)[0]
    closest_labels = nearest_neighbors(image_features, word_annoy_index, word_i
    return closest_labels

def search_similar_images(image_location, from_url=False):
    image_path = image_location
    if from_url:
        filename = image_location.split("/")[-1]
        urllib.request.urlretrieve(image_location, filename)
        image_path = filename

    closest_labels = [l for _, l, _ in get_image_labels(image_path)]
    mean_embedding = np.mean([word_embeddings[label] for label in closest_label
    display_similar_images(mean_embedding, image_path)
```

Evaluation

In [25]:
```
from termcolor import colored
```

In [26]:
```
def pretty_print_label_found(label, label_set, found = True):
    str = "label '{}' {} found in {}"
    is_found_str = "is" if found else "not"
    color = "green" if found else "red"
    print(colored(str.format(label, is_found_str, label_set), color))

def hit_at_k_metric(k, categories_list, image_paths, test_image_indices, verbos
    count_hits = 0
    test_size = len(test_image_indices)

    for ind in test_image_indices:
        test_image_path = image_paths[ind]
        prediction = get_image_labels(test_image_path, False, k)
        predicted_image_labels = [x for _, x, _ in prediction]

        label = categories_list[ind]
        hit_found = False
        for ground_truth_value in label.split("_"):
            if ground_truth_value in predicted_image_labels:
                if verbose:
                    pretty_print_label_found(ground_truth_value, predicted_imag
                hit_found = True
                break
        if verbose and not hit_found:
            pretty_print_label_found(label, predicted_image_labels, found=False
        count_hits += hit_found

    print("hit@{} = {}/{}".format(k, count_hits, test_size))
    return count_hits/test_size
```

In [27]:
```
hit_at_k_metric(5, categories_list, image_paths, test_data[0].index, verbose=Tr
```

```
label 'bus' is found in ['bus', 'drivers', 'passenger', 'accident', 'driving']
label 'bird' not found in ['chickens', 'fowl', 'geese', 'mosquito', 'insect']
label 'sofa' is found in ['sofa', 'lounging', 'patio', 'pajamas', 'slippers']
label 'potted_plant' not found in ['aquatic', 'seedlings', 'fruits', 'cultivat
ed', 'wetland']
label 'car' is found in ['car', 'vehicle', 'truck', 'cars', 'driving']
label 'aeroplane' not found in ['nieuport', 'herreshoff', 'wide-body', 'indige
nously', 'alouette']
label 'potted_plant' not found in ['shrubs', 'flowering', 'grasses', 'orchids
', 'shrub']
label 'dining' is found in ['dining', 'tables', 'kitchen', 'rooms', 'chairs']
label 'bicycle' not found in ['helmet', 'lockers', 'belts', 'bins', 'luggage']
label 'horse' not found in ['thoroughbred', 'breeders', 'colt', 'derby', 'prea
kness']
label 'person' not found in ['patient', 'fit', 'instance', 'fits', 'guess']
label 'car' is found in ['car', 'vehicle', 'cars', 'truck', 'driving']
label 'sheep' not found in ['dolly', 'infected', 'dove', 'grey', 'spotted']
label 'dog' is found in ['cat', 'dog', 'pet', 'dogs', 'birds']
label 'motorbike' not found in ['motorbikes', 'skateboarders', 'careered', 'ka
```

In [28]:
```
hit_at_k_metric(10, categories_list, image_paths, test_data[0].index)
```

```
hit@10 = 51/100
```

Out[28]:
```
0.51
```

In [29]:
```
hit_at_k_metric(20, categories_list, image_paths, test_data[0].index)
```

```
hit@20 = 61/100
```

Out[29]:
```
0.61
```

In [30]:
```
def hierarchical_precision_at_k_metric(k, categories_list, image_paths, test_im
    count_hits = 0
    total_matches = 0
    test_size = len(test_image_indices)

    for ind in test_image_indices:
        # Hybrid model's k predictions
        test_image_path = image_paths[ind]
        prediction = get_image_labels(test_image_path, False, k)
        predicted_image_labels = [x for _, x, _ in prediction]

        # Ground truth list (k nearest labels in GloVe)
        label_embedding = get_word_embedding(categories_list[ind], word_embeddi
        nearest_k = nearest_neighbors(label_embedding, word_annoy_index, word_i
        ground_truth_list = [x for _, x, _ in nearest_k]

        # Calculate mean precision
        common = set(predicted_image_labels).intersection(ground_truth_list)
        if common:
            total_matches += len(common)
            count_hits += 1
    print("non-empty intersections: {}/{}\n".format(count_hits, test_size))
    precision = total_matches/(test_size * k)
    print("PRECISION = 1/N * SUM(intersection_size_for_ith_image / k)")
    print("          = SUM(intersection_size_for_ith_image)/(N*k)")
    print("          = {}".format(precision))
    print("where N={}, k={}, SUM(intersection_size_for_ith_image)={}".format(te
    return precision
```

In [31]:
```
hierarchical_precision_at_k_metric(
    5, categories_list, image_paths, test_data[0].index)
```

```
non-empty intersections: 52/100

PRECISION = 1/N * SUM(intersection_size_for_ith_image / k)
          = SUM(intersection_size_for_ith_image)/(N*k)
          = 0.252
where N=100, k=5, SUM(intersection_size_for_ith_image)=126
```

Out[31]: 0.252

In [32]:
```
hierarchical_precision_at_k_metric(
    10, categories_list, image_paths, test_data[0].index)
```

```
non-empty intersections: 68/100

PRECISION = 1/N * SUM(intersection_size_for_ith_image / k)
          = SUM(intersection_size_for_ith_image)/(N*k)
          = 0.269
where N=100, k=10, SUM(intersection_size_for_ith_image)=269
```

Out[32]: 0.269

In [33]:
```
hierarchical_precision_at_k_metric(
    20, categories_list, image_paths, test_data[0].index)
```

```
non-empty intersections: 85/100

PRECISION = 1/N * SUM(intersection_size_for_ith_image / k)
          = SUM(intersection_size_for_ith_image)/(N*k)
          = 0.312
where N=100, k=20, SUM(intersection_size_for_ith_image)=624
```

Out[33]: 0.312

DEMO

Identifying labels on some pictures from the dataset:

In [34]:
```
get_image_labels(image_paths[222], display_image=True)
```

Out[34]:
```
[[569, 'car', 0.5527702569961548],
 [1907, 'vehicle', 0.80091792345047],
 [2575, 'truck', 0.8080704212188721],
 [1277, 'cars', 0.813259482383728],
 [2031, 'driving', 0.92554771900177],
 [3069, 'drivers', 0.9319164156913757],
 [7213, 'motorcycle', 0.9379380345344543],
 [14015, 'suv', 0.941043496131897],
 [12238, 'jeep', 0.9664146900177002],
 [7319, 'mercedes', 0.9684042930603027]]
```

In [35]:
```
get_image_labels(image_paths[368], display_image=True)
```

Out[35]:
```
[[4802, 'cnn', 1.0367529392242432],
 [172, 'news', 1.0832966566085815],
 [1170, 'web', 1.10819411277771],
 [925, 'internet', 1.1133296489715576],
 [3874, 'programming', 1.1191654205322266],
 [1292, 'online', 1.137448787689209],
 [9986, 'websites', 1.1395326852798462],
 [1077, 'talk', 1.1419448852539062],
 [2400, 'closely', 1.1424530744552612],
 [8927, 'advertisements', 1.1439480781555176]]
```

Searching for specific category/label:

In [36]:
```
search_by_text("aeroplane")
```





Searching for categories that are not in image dataset:

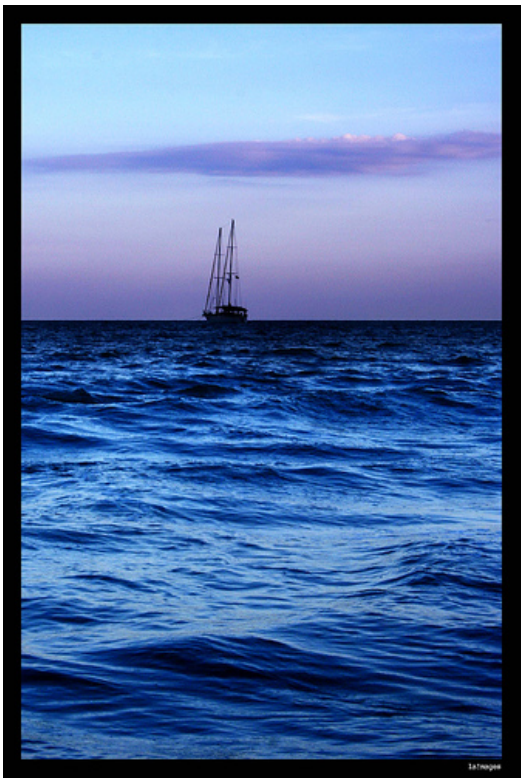
In [37]:
```
search_by_text("avian")
```







In [38]:
```
search_by_text("ocean")
```





In [39]:
```
search_by_text("street")
```




Achieving more complex queries by combining words:

In [40]:
```
search_by_text("horse jockey")
```


S


4
©2007 SARAH K. ANDREW

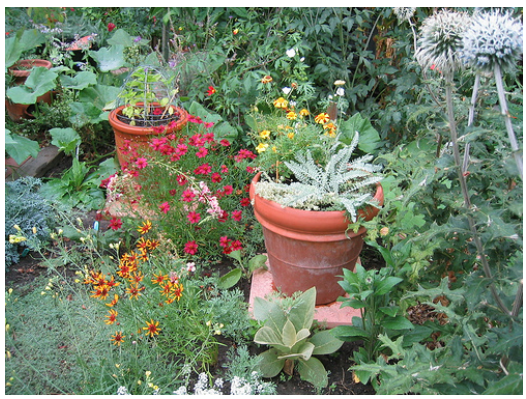
In [41]:
```
search_by_text("bird near water")
```

Since we are able to identify labels on an image, we can perform search by image:

In [42]:
```
search_similar_images(image_paths[test_data[0].index[3]])
```

Our input image

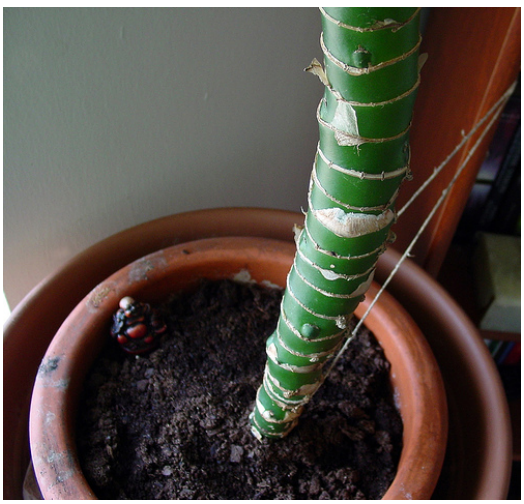
Similar images



In [43]: search_similar_images(image_paths[test_data[0].index[6]])

Our input image

Similar images

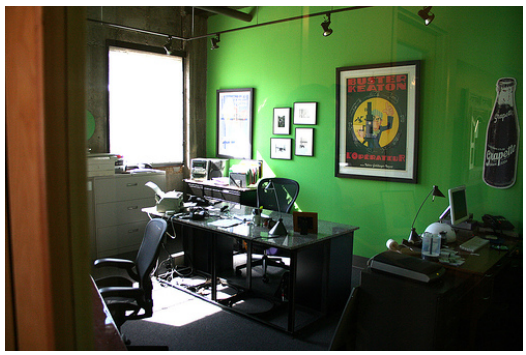


The same goes for images outside of the dataset

In [44]:
```
search_similar_images('./demo_images/modern-interior-design-home-preview.jpg')
```

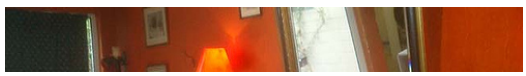
Our input image

Similar images









In [45]:
```
search_similar_images('./demo_images/death-railway-4008940_960_720.jpg')
```

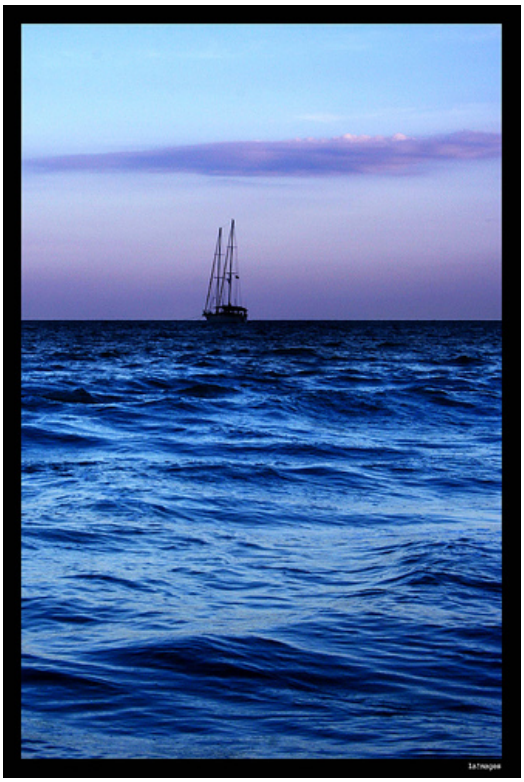
Our input image

Similar images

In [46]:
```
search_similar_images('./demo_images/moor-moorland-grasses-wetland-nature-conse
```

Our input image

Similar images



In [ ]: