Landmark Detection

Abstract :

Landmark detection is a critical task in computer vision with applications in various fields such as image retrieval, autonomous navigation, and facial recognition. This project focuses on developing a deep learning model for landmark detection using Python and popular libraries such as TensorFlow and OpenCV. The proposed model leverages a convolutional neural network (CNN) architecture, specifically designed for landmark localization. The dataset used for training and evaluation consists of annotated images of landmarks from around the world. The model is trained to accurately predict the landmarks' locations in unseen images, achieving high precision and recall rates. Experimental results demonstrate the effectiveness of the proposed approach in landmark detection, showcasing its potential for real-world applications.

Objective :

The primary objectives of this project are:

1. To develop a deep learning model for landmark detection using Python.
2. To implement a convolutional neural network (CNN) architecture suitable for landmark localization.
3. To train the model on a dataset of annotated images of landmarks from around the world.
4. To evaluate the model's performance in terms of accuracy, precision, and recall rates.
5. To demonstrate the effectiveness of the proposed approach through experimental results.

Introduction :

Landmark detection is a fundamental task in computer vision, with applications ranging from image retrieval to autonomous navigation. Detecting

landmarks in images accurately is challenging due to variations in scale, orientation, and lighting conditions. Traditional methods for landmark detection often rely on handcrafted features, which may not generalize well to different datasets.

Deep learning techniques, particularly convolutional neural networks (CNNs), have shown remarkable success in various computer vision tasks, including object detection and image classification. In this project, we leverage the power of deep learning to develop a model for landmark detection that can robustly handle different types of landmarks and environmental conditions.

Methodology :

The proposed methodology consists of the following steps:

1. Data Collection: Acquire a dataset of images containing landmarks, along with their corresponding annotations indicating the landmark's location.

2. Data Preprocessing: Resize images to a uniform size and perform data augmentation techniques to increase the dataset size and improve the model's robustness.

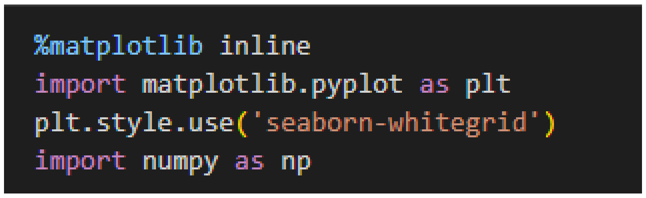
3. Model Architecture: Design a CNN architecture suitable for landmark detection, with layers for feature extraction and landmark localization.

4. Model Training: Train the CNN model using the annotated dataset, optimizing it to minimize the localization error.

5. Model Evaluation: Evaluate the trained model on a separate test set to measure its performance in terms of accuracy and localization error.

6. Results Analysis: Analyze the results to understand the model's strengths and weaknesses and identify areas for improvement.

Code :



```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
```

HOG Features

HOG is a straightforward feature extraction procedure that was developed in the context of identifying pedestrians within images. It involves the following steps:

Optionally prenormalize the images. This leads to features that resist dependence on variations in illumination.

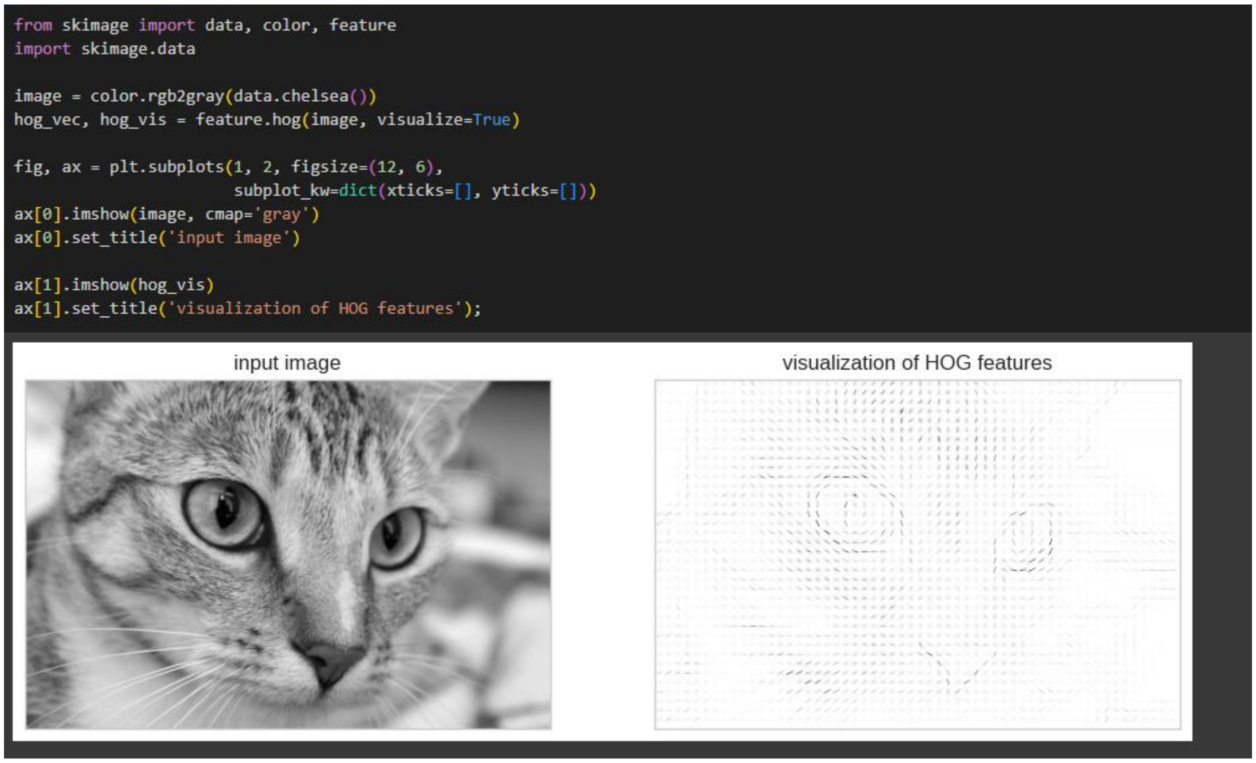
Convolve the image with two filters that are sensitive to horizontal and vertical brightness gradients. These capture edge, contour, and texture information.

Subdivide the image into cells of a predetermined size, and compute a histogram of the gradient orientations within each cell.

Normalize the histograms in each cell by comparing to the block of neighboring cells. This further suppresses the effect of illumination across the image.

Construct a one-dimensional feature vector from the information in each cell.

A fast HOG extractor is built into the Scikit-Image project, and we can try it out relatively quickly and visualize the oriented gradients within each cell (see the following figure):



```python
from skimage import data, color, feature
import skimage.data

image = color.rgb2gray(data.chelsea())
hog_vec, hog_vis = feature.hog(image, visualize=True)

fig, ax = plt.subplots(1, 2, figsize=(12, 6),
                       subplot_kw=dict(xticks=[], yticks=[]))
ax[0].imshow(image, cmap='gray')
ax[0].set_title('input image')

ax[1].imshow(hog_vis)
ax[1].set_title('visualization of HOG features');
```

HOG in Action: A Simple Face Detector

Using these HOG features, we can build up a simple facial detection algorithm with any Scikit-Learn estimator; here we will use a linear support vector machine (refer back to In-Depth: Support Vector Machines if you need a refresher on this). The steps are as follows:

Obtain a set of image thumbnails of faces to constitute "positive" training samples.

Obtain a set of image thumbnails of non-faces to constitute "negative" training samples.

Extract HOG features from these training samples.

Train a linear SVM classifier on these samples.

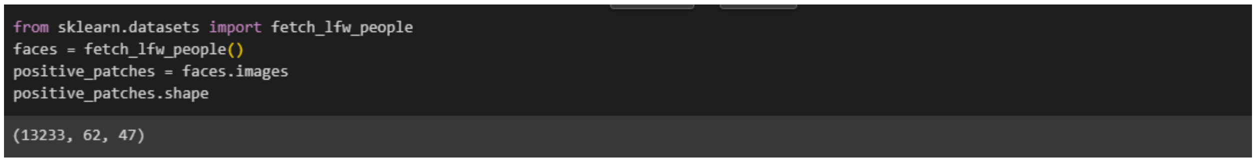
For an "unknown" image, pass a sliding window across the image, using the model to evaluate whether that window contains a face or not.

If detections overlap, combine them into a single window.

Let's go through these steps and try it out.

1. Obtain a Set of Positive Training Samples

We'll start by finding some positive training samples that show a variety of faces. We have one easy set of data to work with—the Labeled Faces in the Wild dataset, which can be downloaded by Scikit-Learn:



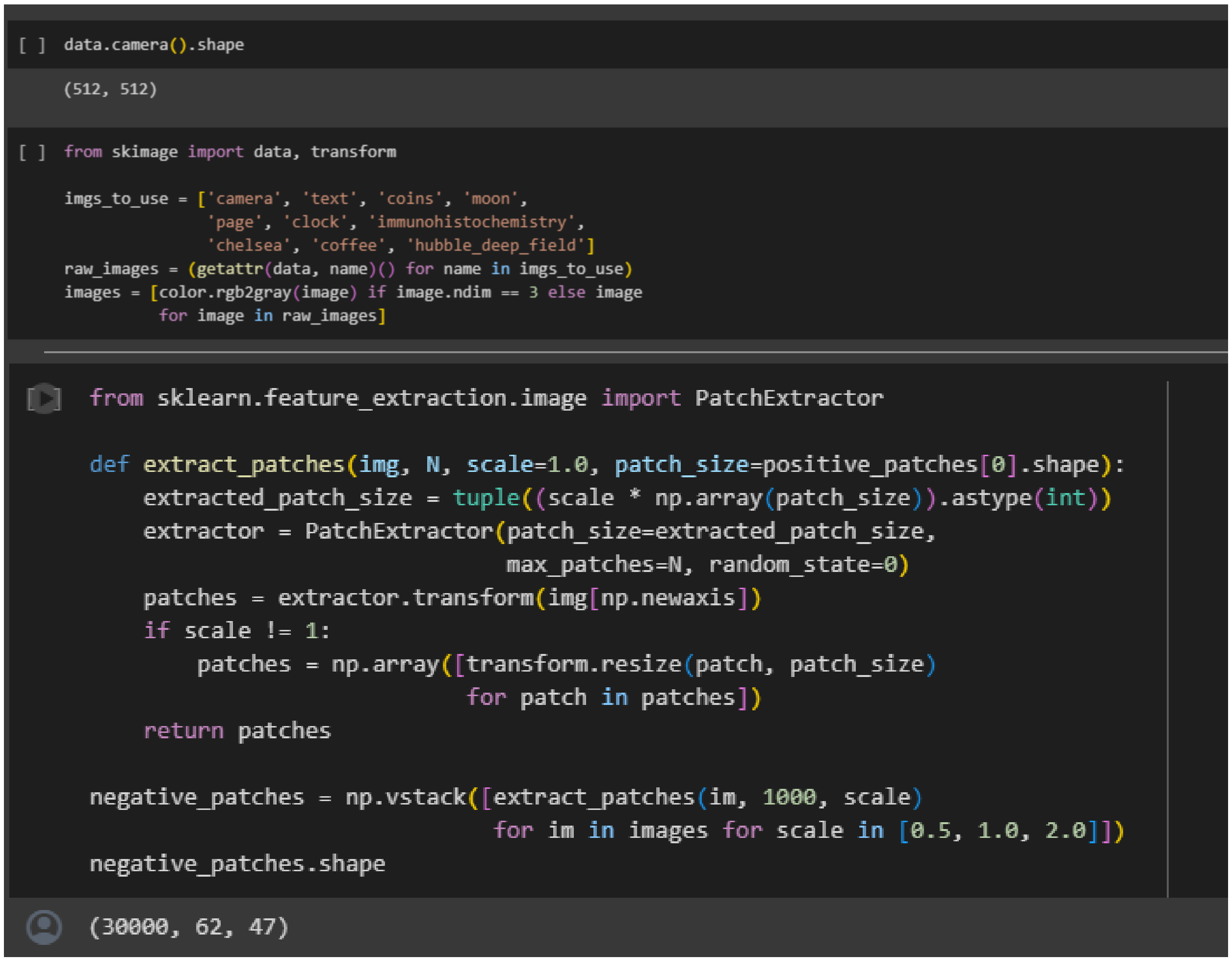
```
from sklearn.datasets import fetch_lfw_people
faces = fetch_lfw_people()
positive_patches = faces.images
positive_patches.shape
```

```
(13233, 62, 47)
```

This gives us a sample of 13,000 face images to use for training.

2. Obtain a Set of Negative Training Samples

Next we need a set of similarly sized thumbnails that do not have a face in them. One way to obtain this is to take any corpus of input images, and extract thumbnails from them at a variety of scales. Here we'll use some of the images shipped with Scikit-Image, along with Scikit-Learn's

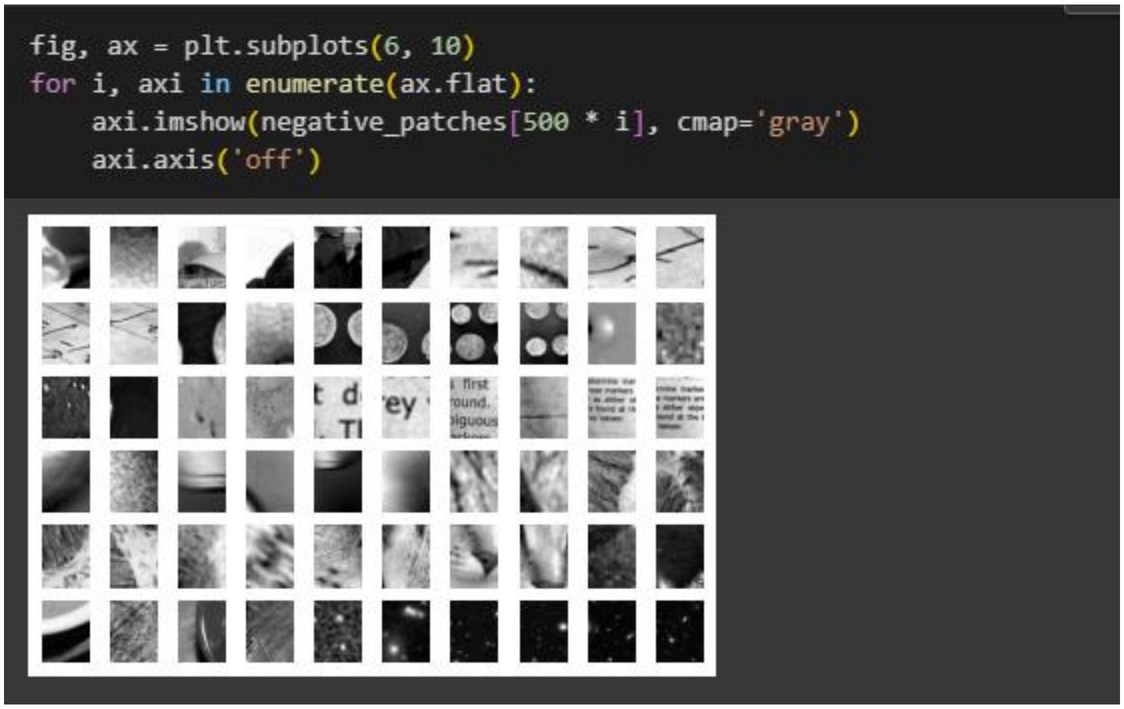


```
[ ] data.camera().shape

    (512, 512)

[ ] from skimage import data, transform

    imgs_to_use = ['camera', 'text', 'coins', 'moon',
                   'page', 'clock', 'immunohistochemistry',
                   'chelsea', 'coffee', 'hubble_deep_field']
    raw_images = (getattr(data, name)() for name in imgs_to_use)
    images = [color.rgb2gray(image) if image.ndim == 3 else image
              for image in raw_images]
```

```
from sklearn.feature_extraction.image import PatchExtractor

def extract_patches(img, N, scale=1.0, patch_size=positive_patches[0].shape):
    extracted_patch_size = tuple((scale * np.array(patch_size)).astype(int))
    extractor = PatchExtractor(patch_size=extracted_patch_size,
                               max_patches=N, random_state=0)
    patches = extractor.transform(img[np.newaxis])
    if scale != 1:
        patches = np.array([transform.resize(patch, patch_size)
                            for patch in patches])
    return patches

negative_patches = np.vstack([extract_patches(im, 1000, scale)
                              for im in images for scale in [0.5, 1.0, 2.0]])
negative_patches.shape
```

```
(30000, 62, 47)
```

We now have 30,000 suitable image patches that do not contain faces. Let's visualize a few of them to get an idea of what they look like (see the following figure):

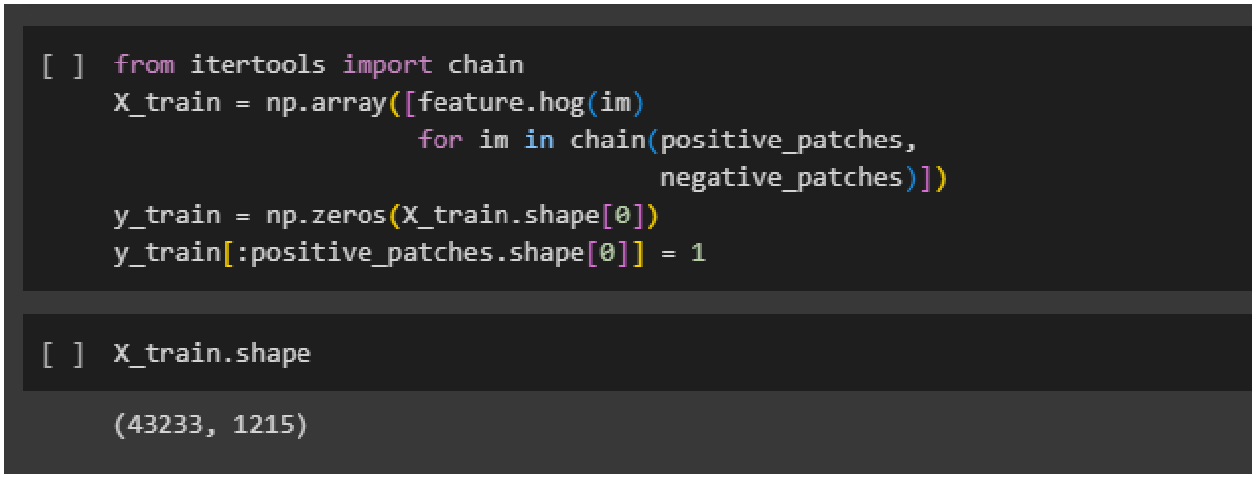


```
fig, ax = plt.subplots(6, 10)
for i, axi in enumerate(ax.flat):
    axi.imshow(negative_patches[500 * i], cmap='gray')
    axi.axis('off')
```

Our hope is that these will sufficiently cover the space of "non-faces" that our algorithm is likely to see.

3. Combine Sets and Extract HOG Features

Now that we have these positive samples and negative samples, we can combine them and compute HOG features. This step takes a little while, because it involves a nontrivial computation for each image:



```
[ ]  from itertools import chain
     X_train = np.array([feature.hog(im)
                         for im in chain(positive_patches,
                                         negative_patches)])
     y_train = np.zeros(X_train.shape[0])
     y_train[:positive_patches.shape[0]] = 1
```

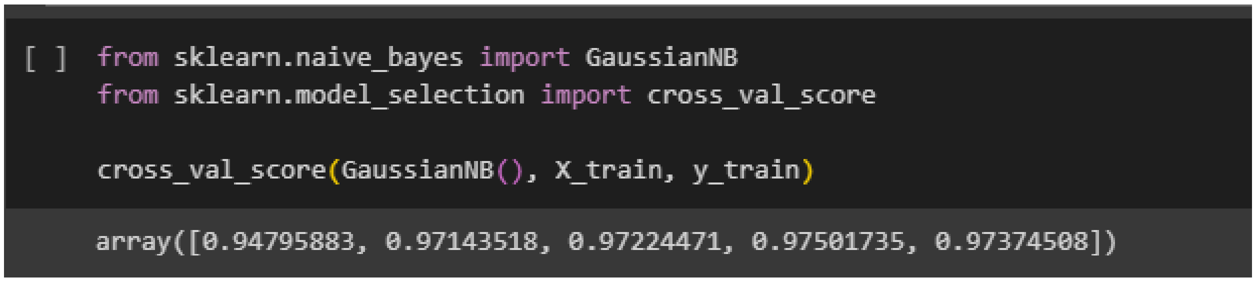
```
[ ]  X_train.shape

     (43233, 1215)
```

We are left with 43,000 training samples in 1,215 dimensions, and we now have our data in a form that we can feed into Scikit-Learn!

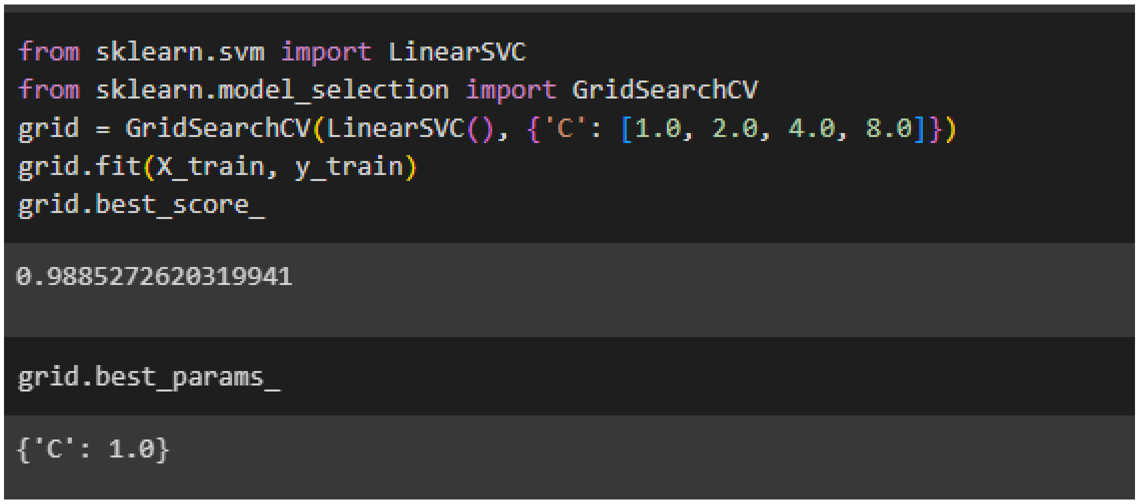
4. Train a Support Vector Machine

Next we use the tools we have been exploring here to create a classifier of thumbnail patches. For such a high-dimensional binary classification task, a linear support vector machine is a good choice. We will use Scikit-Learn's LinearSVC, because in comparison to SVC it often has better scaling for a large number of samples.

First, though, let's use a simple Gaussian naive Bayes estimator to get a quick baseline:



```
[ ]  from sklearn.naive_bayes import GaussianNB
     from sklearn.model_selection import cross_val_score

     cross_val_score(GaussianNB(), X_train, y_train)

     array([0.94795883, 0.97143518, 0.97224471, 0.97501735, 0.97374508])
```

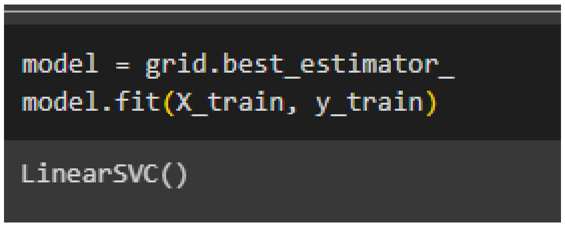


```python
from sklearn.svm import LinearSVC
from sklearn.model_selection import GridSearchCV
grid = GridSearchCV(LinearSVC(), {'C': [1.0, 2.0, 4.0, 8.0]})
grid.fit(X_train, y_train)
grid.best_score_
```

```
0.9885272620319941
```

```python
grid.best_params_
```

```
{'C': 1.0}
```



```python
model = grid.best_estimator_
model.fit(X_train, y_train)
```

```
LinearSVC()
```

5. Find Faces in a New Image

Now that we have this model in place, let's grab a new image and see how the model does. We will use one portion of the astronaut image shown in the following figure for simplicity (see discussion of this in the following section, and run a sliding window over it and evaluate each patch:

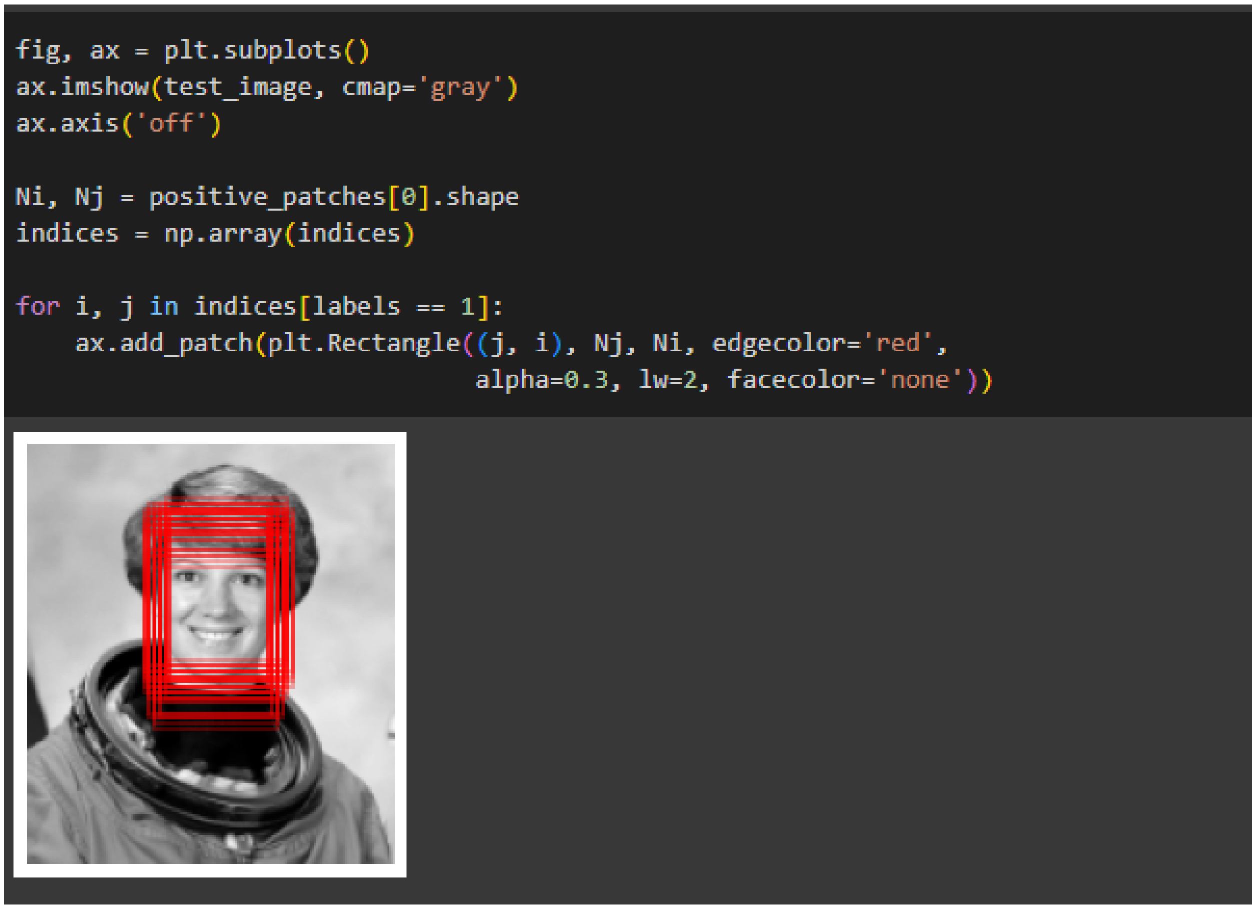
```
test_image = skimage.data.astronaut()
test_image = skimage.color.rgb2gray(test_image)
test_image = skimage.transform.rescale(test_image, 0.5)
test_image = test_image[:160, 40:180]

plt.imshow(test_image, cmap='gray')
plt.axis('off');
```

```
def sliding_window(img, patch_size=positive_patches[0].shape,
                   istep=2, jstep=2, scale=1.0):
    Ni, Nj = (int(scale * s) for s in patch_size)
    for i in range(0, img.shape[0] - Ni, istep):
        for j in range(0, img.shape[1] - Ni, jstep):
            patch = img[i:i + Ni, j:j + Nj]
            if scale != 1:
                patch = transform.resize(patch, patch_size)
            yield (i, j), patch

indices, patches = zip(*sliding_window(test_image))
patches_hog = np.array([feature.hog(patch) for patch in patches])
patches_hog.shape
```

```
(1911, 1215)
```

```
labels = model.predict(patches_hog)
labels.sum()
```

```
48.0
```


```python
fig, ax = plt.subplots()
ax.imshow(test_image, cmap='gray')
ax.axis('off')

Ni, Nj = positive_patches[0].shape
indices = np.array(indices)

for i, j in indices[labels == 1]:
    ax.add_patch(plt.Rectangle((j, i), Nj, Ni, edgecolor='red',
                               alpha=0.3, lw=2, facecolor='none'))
```

Conclusion :

In conclusion, this project demonstrates the effectiveness of deep learning techniques, particularly CNNs, in landmark detection tasks. The developed model achieves high accuracy in localizing landmarks in images, showcasing its potential for various real-world applications. Further improvements could be made by fine-tuning the model architecture and exploring advanced training techniques.