Introduction
Preparations for the Second Week

**Course web-page:** `https://github.com/BMCV/mobi-fs5-python`

If working in the BioQuant computer room, please **use Chrome instead of Firefox** as the web browser, because the installed version of Firefox is known to have issues with GitHub. Using Firefox is fine when working on your personal computer.

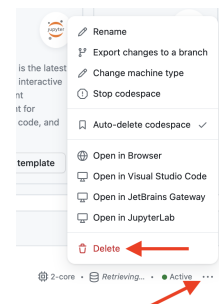## 1 Setting up your GitHub repository

1. Open the course web-page in any web-browser of your choice.

2. Click on the link in "Create a new GitHub repository by using this link".

3. This will load another web-page entitled "Create a new repository". Leave everything on default and confirm the creation of the repository by clicking the "Create repository" button.

4. You should see a "Generating your repository" message for a few seconds and then be presented with an **overview of your repository**.

2 Firing up a GitHub Codespace

1. Open the **overview of your repository** on GitHub. This is the web-page that you landed on after completing Task 1.

2. Click on the green "Code'" button, then select the "Codespaces" tab.

3. Click the green "Create codespace on master" button. This will load **VS Code** (Visual Studio Code) inside of your web-browser. Wait until everything is loaded, it may take about one minute.
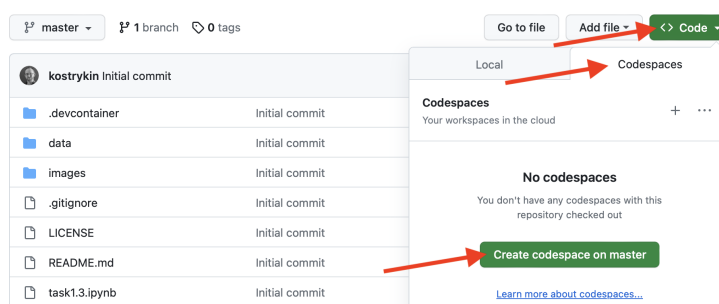
Creating a codespace.   Deleting a codespace.

Note: If, at any time, **VS Code** behaves weirdly (e.g., complaining about missing extensions, not loading notebooks, not finding kernels, or similar), try to reload the **VS Code** window. To do that, press ⌈Ctrl⌋ + ⌈P⌋ (or ⌘ + ⌈P⌋ if you are on macOS) and type "> Reload window ⏎ " (don't miss the ">" at the front). Your work progress will be preserved. If this does not solve the issue, make sure your work is **committed and pushed** (see Task 4), then go to `https://github.com/codespaces` and **delete** the codespace. Then, re-create the codespace by following the steps 1–3 described above.

Introduction

3 Working with VS Code and Jupyter notebooks

The left panel of **VS Code** shows an overview of **your local repository**. Right now it is identical to your GitHub repository. The assignments of this course are organized into several Jupyter notebooks. These are the `*.ipynb` files that you can see in your repository. By progressing from task to task, you will work with different notebooks (open a notebook by double clicking it).

In Jupyter notebooks, code cells can be run in an *arbitrary order*. This is very helpful for experimenting and trying out new things. Still, an assignment in this course is *only* considered "finished" when the results can be reproduced by re-running all code cells *from top to bottom* by clicking the "Run all" button.

4 Preserving your work progress with Git

When finished, close your notebook. Changes are saved automatically within **your local repository**, but remember, that those changes will be lost when you close the codespace, unless you push them to your GitHub repository.

To do that, click on the "Terminal" tab at the bottom of **VS Code**, type the following Git command, and press ⏎ to execute it:

```
git commit --all -m "Finish task 1.3"
```

The text "`Finish task 1.3`" is the **commit message**, which is arbitrary. Describe what changes you have done since your previous commit.

If you wanted, you could revert to any previous commit at a later time, and choosing an expressive message is convenient for finding the commit which you will be looking for. There also are some conventions for how a **commit message** should be formatted: It should tell in an "imperative mood" and as concisely as possible, what *the committed changes* are supposed to do[1].

If done correctly, the output of the above command should be something like "1 file changed, 12 insertions(+), 1 deletion(-)", but the exact numbers may vary.

Finally, type "`git push`" and press ⏎ to **push the committed changes** to your GitHub repository. If done correctly, there should be multiple lines of output, concluding with a line similar to "`1d2e403..a387645 master —> master`".

5 Create a cheat sheet for the second week

Now you already know how to edit, commit, and push a Jupyter notebook.

1. In preparation of the second week, create a cheat sheet of the most important takeaways from what you have learned in Datacamp. To do so, open the notebook `cheatsheet.ipynb` and complete it.

2. Commit and push your changes, then close **VS Code** (the browser tab/window).

---

[1]From the official Git documentation: `https://git.kernel.org/pub/scm/git/git.git/tree/Documentation/SubmittingPatches?h=v2.36.1#n181`

Lab Session 1
Threshold-based segmentation, Sobel filter

1 Intensity thresholding and Dice coefficient

Above, we described how to use Codespaces, work with Jupyter notebooks, and commit and push your changes to your GitHub repository. Now proceed with the tasks below.

1. Open VS Code in a GitHub Codespace (see Task 2 of the Introduction) using the repository which you created before (see Task 1 of the Introduction).

2. Open the Jupyter notebook `segm_threshold.ipynb` and extend it as follows:

    (a) Use `plt.imread('data/NIH3T3/im/dna-0.png')` to load an image.

    (b) Use `plt.figure()` (or, e.g., `plt.figure(figsize=(15,8))` if you want to specify the size of the figure) to add a *figure* to a code cell of the notebook. Then, *within the same code cell*, use `imshow` to display the image within the figure. In addition, use `colorbar()` after the `imshow`-instruction to include a legend of the color encoding.

    (c) Recall that given a threshold T, each pixel x, y of the image with an intensity value $g(x, y)$ greater or equal the threshold T is assigned the value 1 and each pixel with a value less than T is assigned the value 0:

    $$g_{\text{threshold}}(x, y) = \begin{cases} 1 & \text{if } g(x, y) \geq T, \\ 0 & \text{otherwise} \end{cases}$$

    Using the color legend from task (b), what might be a good choice for the threshold T to reproduce the binary image in Figure 1 below?
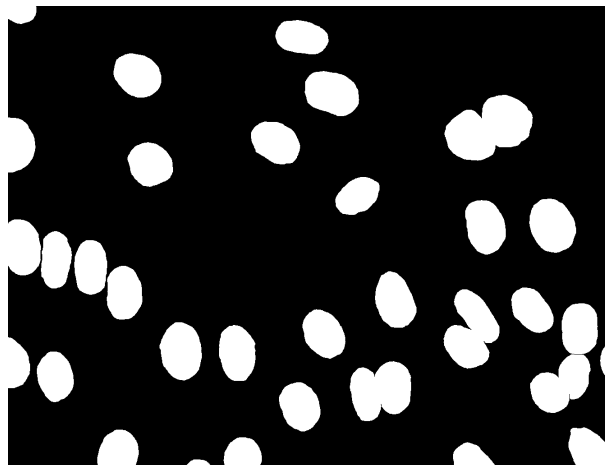

Figure 1: Segmentation ground truth for the image `data/NIH3T3/im/dna-0.png`.

Lab Session 1

(d) Binarize the loaded image by applying **intensity thresholding**. Adjust the threshold such that your result looks similar to Figure 1 – **Hints:**

    i. When working with objects of the type `np.ndarray` (e.g., images), mathematical operations are *propagated* to the intensity values of the image. For example, if g is the image `img` and T is the intensity threshold `thres`, then the formal expression $g_{\text{threshold}}$ corresponds to "`img >= thres`" in Python.

    ii. Exploiting propagation of mathematical expressions is *faster* than writing loops. More importantly, it is also *less cumbersome* and *less prone to programming mistakes*. Always avoid writing loops when propagation of mathematical expressions can be used instead!

(e) Evaluate your segmentation result by quantitatively comparing it to the ground truth in Figure 1. Use the **Dice coefficient** for the quantitative comparison:

$$\text{Dice}(G, H) = \frac{2\,|G \cap H|}{|G| + |H|}$$

where G and H are the two *sets of image points* corresponding to the foreground of the segmentation result (i.e. the binary image produced by intensity thresholding) and the foreground of the segmentation ground truth, respectively. Compute the Dice coefficient for your segmentation result – **Hints:**

    i. In Python, sets of image points are conveniently represented as *binary* images (a pixel value is set to `True` if the set contains that pixel and to `False` otherwise). An image `img` is binary if `img.dtype` is `bool`.

    ii. If `G` is the *binary* image representing the set G, then `G.sum()` yields the *cardinality* $|G|$ of that set.

    iii. If `G` and `H` are the two *binary* images representing the sets G and H, then "`G * H`" yields the binary image representing $G \cap H$.

    iv. Use "`plt.imread('data/NIH3T3/gt/0.png')`" to load the ground truth.

3. Commit and push your changes. Remember to do this regularly!

Lab Session 1

$\frac{1}{8}$	+1	0	-1
	+2	0	-2
	+1	0	-1

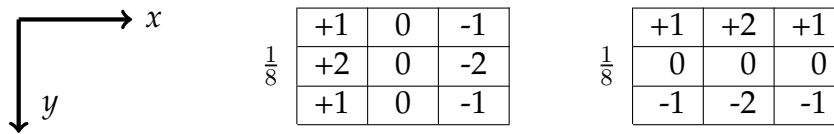| $\frac{1}{8}$	+1	+2	+1
	0	0	0
	-1	-2	-1

Figure 2: Sobel derivative operators

2 Sobel filter (BONUS)

Open the notebook `sobel.ipynb`. Extend it as follows:

1. Use `plt.imread` and `plt.imshow` to load and show the image `data/lena.png`.

2. Compute the partial derivatives g_x and g_y by *convolving* the image $g(x, y)$ with Sobel derivative operators (see Figure 2). To this end, implement the re-usable functions `sobel_h` and `sobel_v`. The functions are supposed to compute and return the *convolution* of the input image `img` with the horizontal and vertical Sobel derivative operators, respectively.

3. Test your implementations for `sobel_h` and `sobel_v` by including images of the computed partial derivatives into your notebook (also include color legends).

4. Compute the *magnitude* of the image gradient

$$\left\| \nabla g(x, y) \right\| = \sqrt{g_x^2(x, y) + g_y^2(x, y)}$$

and include the resulting image into the notebook. *Exploit propagation of mathematical expressions instead of writing loops!*

1 Otsu thresholding

Open the notebook `otsu.ipynb` and extend it as follows:

1. Load the image from: `imgf = plt.imread('data/NIH3T3/im/dna-0.png')`.
   The intensities of this image range from 0 to 1.

2. Quantize the image into 256 bins:

   ```
   img8 = (imgf * 255).round().astype(np.uint8)
   ```

3. Compute the histogram `h` of `img8` such that, for example, `h[0]` corresponds to the number of image pixels with intensity 0 and `h[255]` corresponds to the number of image pixels with intensity 255 – **Hint:** "`img8 == i`" yields a binary image, which corresponds to the set of all points with intensity `i`.

   Note: Do *not* use pre-defined functions like `hist` or `plt.hist`, but implement the computation of the histogram *yourself*. To accomplish that, the occurrences of the different intensity values in the image `img8` must be *counted*. For counting, `h` should be initialized as a *flat* array with 256 entries (e.g., "`np.zeros(256)`").

   After counting, `h` corresponds to the histogram of `img8`, and the code below can be used to show the histogram:

   ```
   plt.figure(figsize=(14,5))
   plt.bar(range(256), h, width=1)
   ```

4. Compute the optimal intensity threshold T by implementing the **method of Otsu**,

$$\min_{T=1\ldots255} n_1(T) \cdot \sigma_1^2(T) + n_2(T) \cdot \sigma_2^2(T), \tag{1}$$

   where n_i is the number of pixels and σ_i^2 is the intensity variance within the i-th class. Assume that the two classes are given by the intensities $[0, T-1]$ and $[T, 255]$. **Also consider the following hints**:

   (a) The empirical variance of `values` is obtained by `var(values)`.

   (b) The intensity variance σ_i^2 can be computed either using *(i)* the image intensities `img8` directly or *(ii)* the histogram `h` and the formulas from the lecture.

5. Compute the segmentation of the loaded image using **Otsu thresholding** (i.e. use the optimal intensity threshold T to perform intensity thresholding).

6. Evaluate the segmentation result using the Dice coefficient.

Lab Session 2

2 Batch processing

Open the notebook `otsu_batch.ipynb` and extend it as follows:

1. In the previous task, you have implemented the method of Otsu and performed Otsu thresholding for a single image. Now, make this code *re-usable* by putting it into a *function* which you can use anytime later. For your convenience, a skeleton of the code you need to write to implement the function is already added to the notebook.

2. Implement a re-usable function to compute the Dice coefficient using the skeleton in the notebook.

3. Test your implementations by using the two functions to perform Otsu thresholding of the image `'data/NIH3T3/im/dna-0.png'` and compute the corresponding Dice coefficient. *The Dice coefficient should be the same as in Task 1!* If it is not, look for a mistake you have made and fix it.

   Note: The function "`compute_dice`" takes two parameters, which must correspond to *binary* images. If you encounter an `AssertionError` in this function, one of the images used as the parameters of the function is not binary. You can use the method "`.astype(bool)`" to obtain a binary representation of an image.

4. Write a `for`-loop which iterates the sequence `i` = $28, 29, 33, 44, 46, 49$ and

   (a) loads the `i`-th image via `plt.imread(f'data/NIH3T3/im/dna-{i}.png')`,

   (b) loads the corresponding ground truth from `f'data/NIH3T3/gt/{i}.png'`,

   (c) performs Otsu thresholding,

   (d) computes the Dice coefficient of the segmentation result.

   The Dice coefficient should be printed for each image. In addition, compute and print the mean Dice coefficient for all images.

Open the notebook `fourier.ipynb` and extend it as follows:

1. Explore the Fourier transform by following these steps:

    (a) Load the first image[1] as given in the notebook, resize it to (256, 256) pixels using the `resize()` function, and show the result.

    (b) Apply the *fast Fourier transform* (FFT) to the image:

    ```
    cell_ft = np.fft.fft2(cell_img)
    ```

    Be careful to use the `fft2` function (not just `fft` without the "2") since we use a *two*-dimensional image. The result should be a *complex* array, so verify this by checking `cell_ft.dtype` of the result of the FFT. – **Hint:** The FFT is an algorithm that efficiently calculates the discrete Fourier transform (DFT).

    (c) The result of the FFT is in the Cartesian form ($x = a + bi$, where "i" is the imaginary number). You can access the real part a by `.real` and the imaginary part b by `.imag`. Note that in Python the imaginary number i is `j`. To interpret the image in the Fourier domain, take a look at the amplitude r and phase ϕ considering the *polar* form ($x = re^{i\phi}$). Extract the amplitude r the phase ϕ using following code:

    ```
    amplitude = np.abs(cell_ft)
    phase = np.angle(cell_ft)
    ```

    Display them side by side using the following code:

    ```
    plt.figure()
    plt.subplot(1, 2, 1)
    plt.imshow(np.log(amplitude), 'gray')
    plt.subplot(1, 2, 2)
    plt.imshow(phase, 'gray')
    ```

    Hints: The function `plt.subplot` adds a new subplot with the three parameters *(i)* total number of rows, *(ii)* total number of columns, and *(iii)* at which position to place the image. Furthermore, show the *logarithm* of the amplitude for better visibility of the huge range of values.

    (d) Now, shift the zero components to the center of the spectral image using `np.fft.fftshift(cell_ft)`. Extract and show amplitude and phase again.

    (e) Restore the image using the *inverse* fast Fourier transform (IFFT) by using the function `np.fft.ifft2` and show the restored image. The IFFT will also return a complex array. However, since the original image contains only *real* numbers, the restored image can be accessed using the real values of the array. Please note, if you did a shift in the Fourier domain, you have to reverse this first with `np.fft.ifftshift` before applying the IFFT.

---

[1]Source: `https://www.cellpose.org/dataset`

Lab Session 3

2. Study the role of the amplitude and the phase:

   (a) Load the image `data/brain_ct.png`[2].

   (b) Apply the FFT, do a shift, then extract and display phase and amplitude.

   (c) Compose a merged image by "restoring" it from the amplitude of the brain image and the phase of the cell image. To this end, first convert the amplitude and phase to a Cartesian complex array using the pre-implemented function `to_complex_array`. Be careful to *consistently* either use only the shifted or the non-shifted amplitude and phase, when merging. Reverse the shift if you used the shifted values and do the IFFT. Show the result.

   (d) Repeat the previous step vice versa (use the amplitude of the cell image and the phase of the brain image).

   (e) Conclude, *what carries more information*, the phase or the amplitude?

3. Apply a low-pass and a high-pass filter using the convolution theorem:

   (a) Create a mask to use as low-pass filter on the shifted amplitudes. To this end, first create an array of the same `shape` as the image, filled with zeros. Insert a rectangle of filter window size with the value one to the mask at the center of the image. Display your mask to check if only the center contains ones. Your mask should look like the following image:
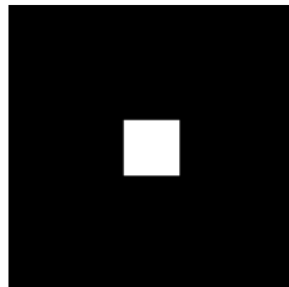
   (b) Also create a high-pass filter mask – **Hint:** You can use "one minus the low-pass mask" to achieve this.

   (c) Now apply your masks by multiplying them with the shifted amplitude of the brain image. Display the shifted amplitude without any filter, the low pass-filtered, and the high pass-filtered image side by side. Scale the amplitudes logarithmically.

   (d) Merge the filtered amplitudes with the phase of the brain image to a complex array. Reverse the shift and apply the IFFT. Show the original image, the low pass-filtered, and the high pass-filtered merged images side by side.

   (e) Repeat the above steps using different filter sizes and observe the differences.

---

[2]Source: `https://brain-development.org/ixi-dataset`

Lab Session 3

4. Implement your own DFT function and compare the result as well as the run time to the implementation from `numpy`:

(a) Implement a re-usable function that computes the DFT, i.e.

$$\texttt{dft}[u,v] = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \texttt{img}[x,y] e^{-2j\pi(\frac{ux}{M} + \frac{vy}{N})} \tag{1}$$

Hints: First, create an array filled with complex values using

```
dft = np.zeros(img.shape, dtype=complex)
```

Then, extract the dimensions of the image by `M,N = img.shape`. The formula (1) calculates one entry of the `dft`-array, so you have to use `for`-loops to fill the whole array (which you initialized with zeros).

(b) Implement a re-usable function that computes the IDFT, i.e.

$$\texttt{idft}[u,v] = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \texttt{dft}[x,y] e^{2j\pi(\frac{ux}{M} + \frac{vy}{N})} \tag{2}$$

(c) Try your implementation using the brain image and display amplitude and phase. – **Hint:** You can still use the `fftshift` function after using your DFT.

(d) Reconstruct the image using your IDFT implementation and display the result.

(e) The FFT is an efficient algorithm to compute the DFT. Compare your results to those obtained using the FFT from `numpy`, which you used for the previous tasks. Apply the FFT as well as your DFT implementation to the brain image. Then, compare the results using `np.allclose`. – **Note:** You must use the parameter `norm='forward'` for the `np.fft.fft2` function to obtain identical results. The reason is that there are slightly different definitions[3] of the DFT and using `norm='forward'` enforces the definition from Eq. (1).

(f) Compute the mean difference between the results of the two algorithms by `np.mean(np.abs(own_dft - np_fft))`. Afterwards, round the DFT and the FFT arrays using `np.round(array, 5)`, where 5 is the number of decimals to maintain. Compare the arrays using `(own_dft == np_fft).all()`. This statement returns `True` if all values of the arrays match. Select the number of decimals in such a way that you round as few as possible while the arrays are still matching.

(g) Compare the run times of your DFT implementation to the FFT implementation from `numpy` using the `%timeit` command.

---

[3]`https://numpy.org/doc/stable/reference/routines.fft.html#implementation-details`

Lab Session 4
Deconvolution

Open the notebook `deconvolution.ipynb` and extend it as follows:

1. Load the image in the notebook and apply the given uniform point spread function (PSF) by using the function `scipy.signal.convolve2d(img, psf, mode='same', boundary='symm')` which is already imported as "conv". Display the resulting image.

2. The notebook also contains a Gaussian PSF. Apply the Gaussian PSF to the original image and observe the differences in comparison to using the uniform PSF.

3. Implement the Richardson-Lucy (R-L) deconvolution as a re-usable function (for your convenience, the notebook already contains a skeleton which you can use). The R-L deconvolution is an iterative process with the step

$$h^{(t+1)} = h^{(t)} \cdot \left(\frac{g}{h^{(t)} * P} * P^* \right), \tag{1}$$

   where g is the input image to the algorithm, h is the deconvolved image, which is initialized as an array of the same `shape` as the image, filled with constant value 0.5. P is the PSF and P^* is the flipped PSF. You can flip it using the `np.flip` function. The operator "$*$" corresponds to the convolution operation for which you can use the "conv" function from Task 1. again.

   Hint: The indices t and $t + 1$ written in parentheses and superscript in Eq. (1) are not exponents, but numbers of iterations, i.e. $h^{(t)}$ denotes h at iteration t and $h^{(t+1)}$ denotes h at iteration $t + 1$.

4. Use your R-L implementation to restore the original image, which was blurred by the first PSF. Display the original, the blurred, and the restored image side by side. Try different values for the number of iterations for the R-L algorithm.

5. Add white noise to the blurred image generated in Task 2 as an additive component (the white noise is already generated in the notebook). To ensure that the image intensities remain in the interval $[0, 1]$ remember to perform intensity clipping after adding the noise. Repeat the R-L deconvolution for the noisy image and display the result as before. Try a different noise level by changing the variable `reduce_factor` (larger values correspond to lower noise levels).

6. Compare the result of your R-L implementation to that of the Wiener deconvolution, which you can use by `wiener(img_psf, psf, balance=2, clip=True)`. The corresponding function `wiener` is pre-implemented. Display the original image beside the R-L reconstruction and the Wiener reconstruction. – **Hint:** You can try different values for the parameter `balance` to achieve better results.

7. Repeat the previous tasks using a smaller value for the size of the PSF (e.g., 5).

8. Try using a "wrong" PSF for the R-L deconvolution and observe the impact. Display both, the R-L reconstruction using the *uniform* PSF for the image blurred with the *Gaussian* PSF, and the other way round.

1   Patch-based segmentation

In this assignment, you will implement segmentation using a machine learning approach. To this end, an image is first subdivided into a equisized grid of patches. Then, a classifier is used to decide for each image patch, whether that image patch corresponds to an object or the image background. More formally, the **decision function**

$$f : \mathbb{R}^{n^2} \rightarrow \{-1, +1\}$$

maps the image intensities from an $n \times n$ image region to a **label**, which represents image background (-1) or image foreground ($+1$), respectively. We implement the function f using a **support vector machine** (SVM) based on Gaussian kernels.

One part of the assignment is the **training** of the SVM using pre-annotated ground truth image data. The SVM is trained by supplying a **data matrix** $X \in \mathbb{R}^{m \times n^2}$ and a corresponding ground truth vector of labels $y \in \{-1, +1\}^m$,

$$X = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}, \qquad y = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix},$$

where each row of the data matrix corresponds to the image intensities from an image patch (squeezed into a row vector). The image patch x_k is associated with the corresponding ground truth label y_k by the index $k \in [m]$.

In the other part of the assignment, the SVM is queried to compute the **prediction** $f(x)$ for image patches $x \in \mathbb{R}^{n^2}$. Instead of computing the prediction for only a single image patch, it is convenient to predict the labels

$$F(X) = \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_m) \end{bmatrix},$$

for multiple image patches at once (where the image patches are arranged in a data matrix $X \in \mathbb{R}^{m \times n^2}$). We use the SVM implementation from scikit-learn[1] and patches of the size 32×32 pixels (i.e. $n = 32$).

Open the notebook `svm_segm.ipynb` and extend it as follows:

1. Implement the function `create_data_matrix`, which takes an image as the parameter, creates the corresponding data matrix X for that image, and returns the data matrix – **Hint:** Loops *can* be avoided by using the function `view_as_blocks`[2] from the `skimage.util` module.

2. Implement the function `create_gt_labels_vector`, which takes a binary image as the parameter (the ground truth segmentation), creates the vector of labels y, and returns that vector. The implementation is analogous to `create_data_matrix`:

---

[1] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
[2] https://scikit-image.org/docs/dev/api/skimage.util.html#skimage.util.view_as_blocks

Bonus Tasks

If the area of the ground truth image corresponding to an image patch contains more than 50% foreground, assign the label $+1$ to that image patch. If that area contains no foreground, assign the label -1. Otherwise, assign the label 0 (see below).

3. Create the SVM classifier "`clf`" using the following code:

```
clf = make_pipeline(StandardScaler(), \
                    SVC(class_weight='balanced', gamma=0.1))
```

Now it is time to train the classifier:

(a) Using the function `create_data_matrix` implemented before, generate the data matrices for the two images `data/NIH3T3/im/dna-33.png` and `dna-44.png`.

(b) Use `create_gt_labels_vector` to generate the label vectors based on the corresponding ground truth images `data/NIH3T3/gt/33.png` and `44.png`.

(c) Stack the two data matrices and the two corresponding label vectors, but only keeping those rows of the data matrices and the label vectors, where the corresponding label is -1 or $+1$ (i.e. discard rows where the label is 0) – **Hint:** You can use `np.concatenate`[3] to perform the stacking.

(d) Use "`clf.fit(X, Y)`" to train the classifier, where `X` is the stacked data matrix and `Y` is the stacked label vector. The duration of the training depends on your computer, but should be complete within a few seconds.

4. Implement the function `predict_image`, which takes an image as the parameter and performs segmentation using the trained classifier `clf`. To this end, the function

(a) uses `create_data_matrix` to obtain the data matrix `X` for the given image,

(b) uses `clf.predict(X)` to obtain the predicted label vector $F(X)$,

(c) creates a binary image `result`, where patches corresponding to the predicted label $+1$ are set to `True` and to `False` otherwise, and returns it.

Afterwards, test your implementation for the image `dna-0.png`. You can use the pre-implemented function `blend_result` to visualize the segmentation results:

```
plt.figure()
img = plt.imread(f'data/NIH3T3/im/dna-0.png')
seg = predict_image(img)
imshow(blend_result(img, seg))
```

---

[3]`https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html`

Bonus Tasks

5. Write a `for`-loop which iterates the sequence $i = 28, 29, 33, 44, 46, 49$ and

   (a) loads the i-th image via `plt.imread(f'data/NIH3T3/im/dna-{i}.png')`,

   (b) loads the corresponding ground truth from `f'data/NIH3T3/gt/{i}.png'`,

   (c) performs patch-based segmentation by using the function `predict_image`,

   (d) computes the Dice coefficient of the segmentation result.

   The Dice coefficient should be printed for each image. In addition, compute and print the mean Dice coefficient for all images.

6. Compare the results to those from Otsu thresholding.

2 Improving the segmentation performance

Improve the segmentation performance. Here are some ideas:

1. The segmentation accuracy of the segmentation approach you have implemented above is limited by the size of the patches. Although larger patches are capable of capturing more information, they also lead to a loss of accuracy since the image is subdivided more coarsely. This can be improved by using *overlapping* patches, which are commonly known as a *sliding window*. You can use `view_as_windows`[4] from the `skimage.util` module instead of `view_as_blocks`.

2. Incorporate pre-processing of the image data (e.g., classify filter responses instead of the raw image intensities, or use principal component analysis to reduce the dimensionality of the data under classification).

3. Incorporate other image features (e.g., Histograms of Gaussians[5]).

4. Be creative and try out different possibilities.

---

[4]`https://scikit-image.org/docs/dev/api/skimage.util.html#skimage.util.view_as_windows`
[5]`https://scikit-image.org/docs/dev/api/skimage.feature.html#skimage.feature.hog`