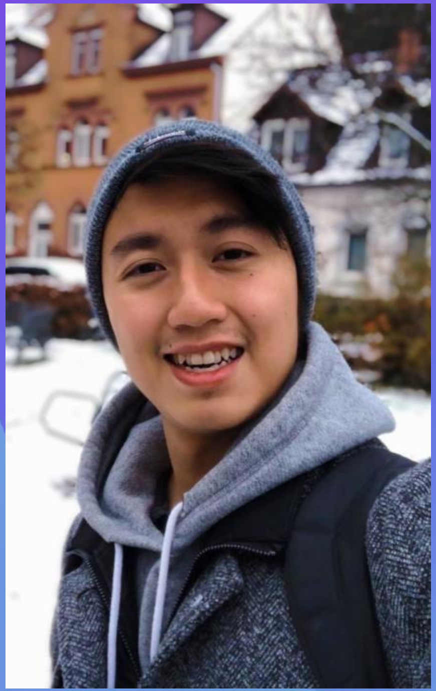
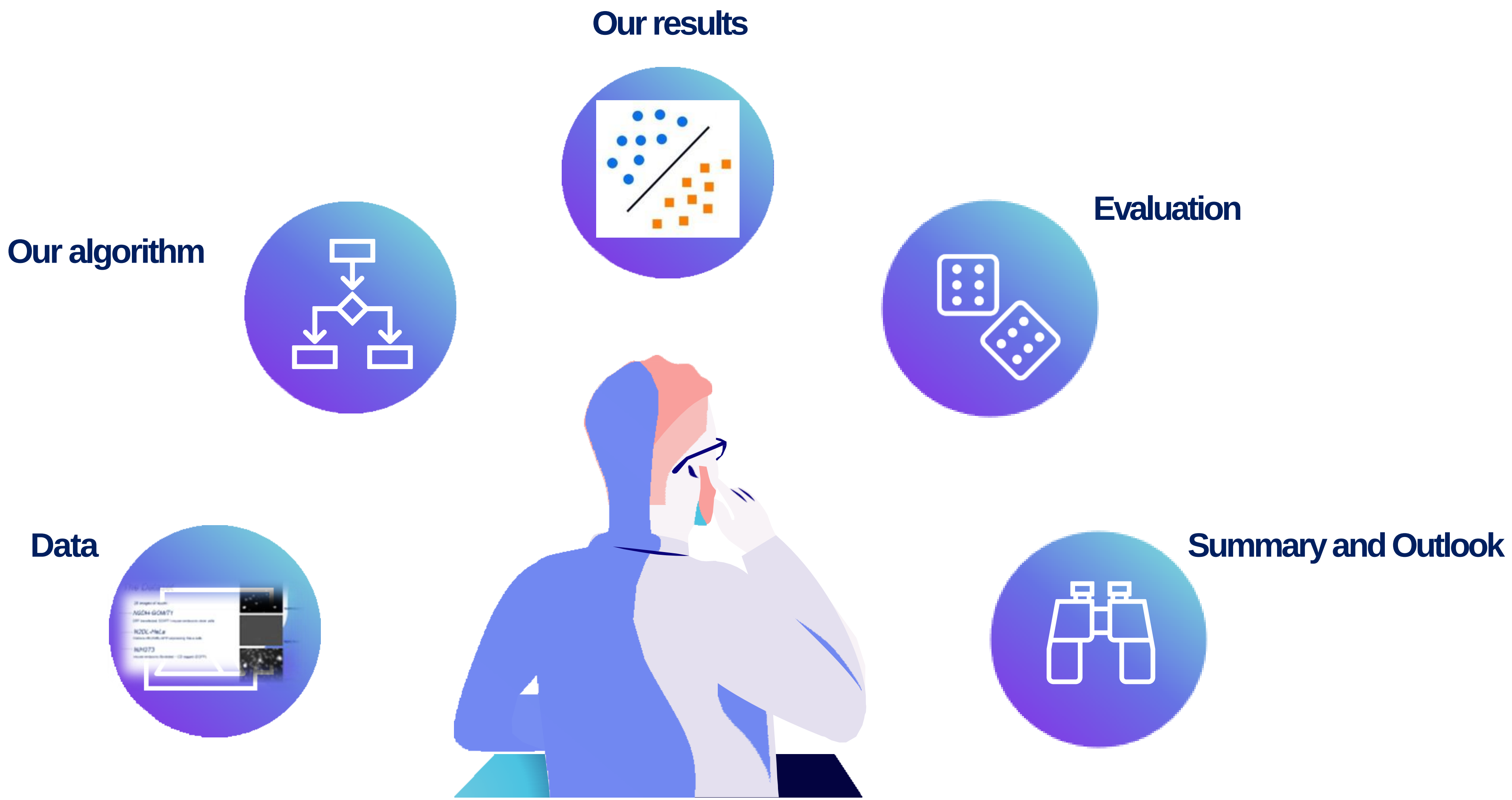
Cell nuclei segmentation: support vector machine

Final presentation by:
Michelle Emmert, Juan Hamdan, Laura Sanchis, und Gloria Timm



Our results
Our algorithm
Evaluation
Data
Summary and Outlook

The Dataset

28 images of nuclei :

N2DH-GOWT1

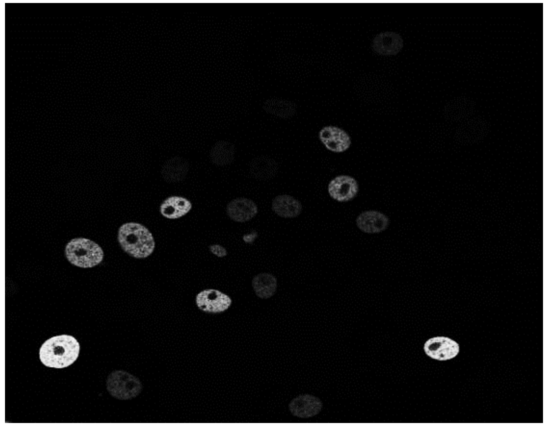
GFP transfected GOWT1 mouse embryonic stem cells

N2DL-HeLa

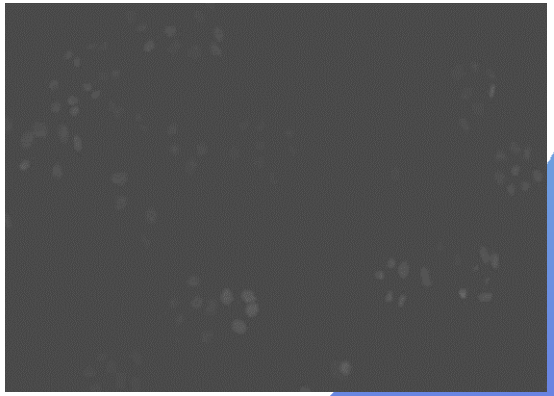
Histone 2B (H2B)-GFP expressing HeLa cells

NIH3T3

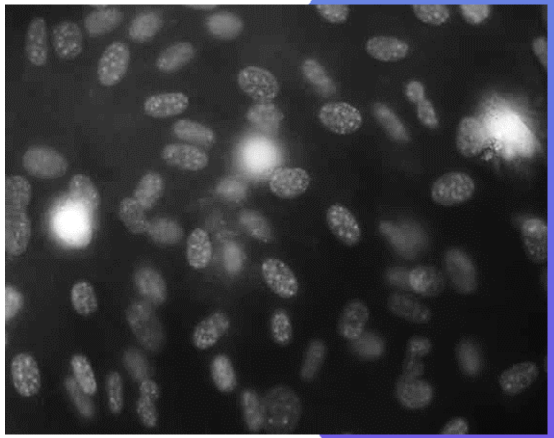
mouse embyonic fibroblast – CD tagged (EGFP)



N2DH-GOWT1



N2DL-HeLa

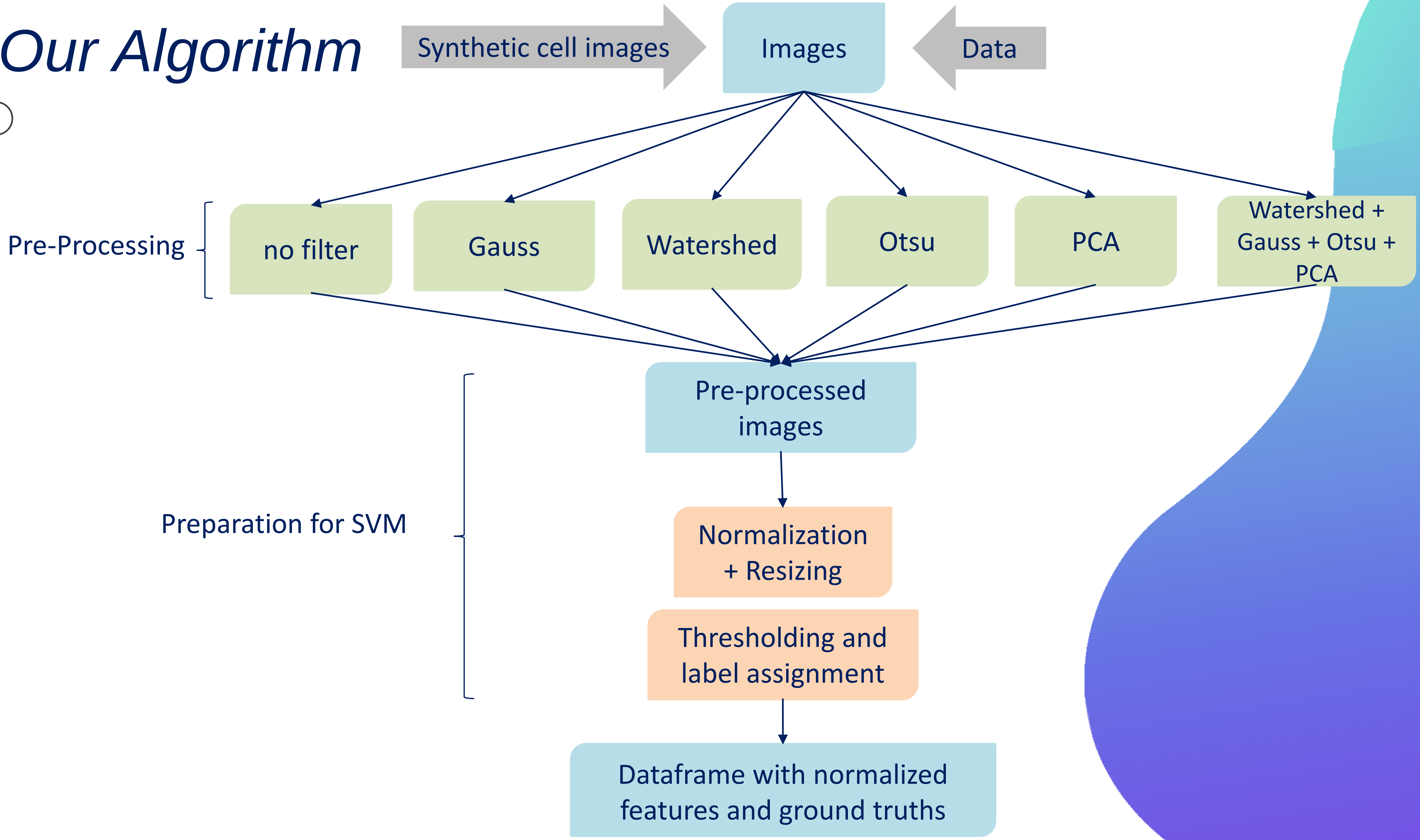


NIH3T3

(1) Osuna, E. et al. 2007. Large-Scale Automated Analysis of Location Patterns in Randomly Tagged 3T3Cells

(2) Maska, M. et al. 2014. A benchmark for comparisonof cell tracking algorithms

Our Algorithm



Our Algorithm

Dataframe with normalized features and ground truths

Segmentation

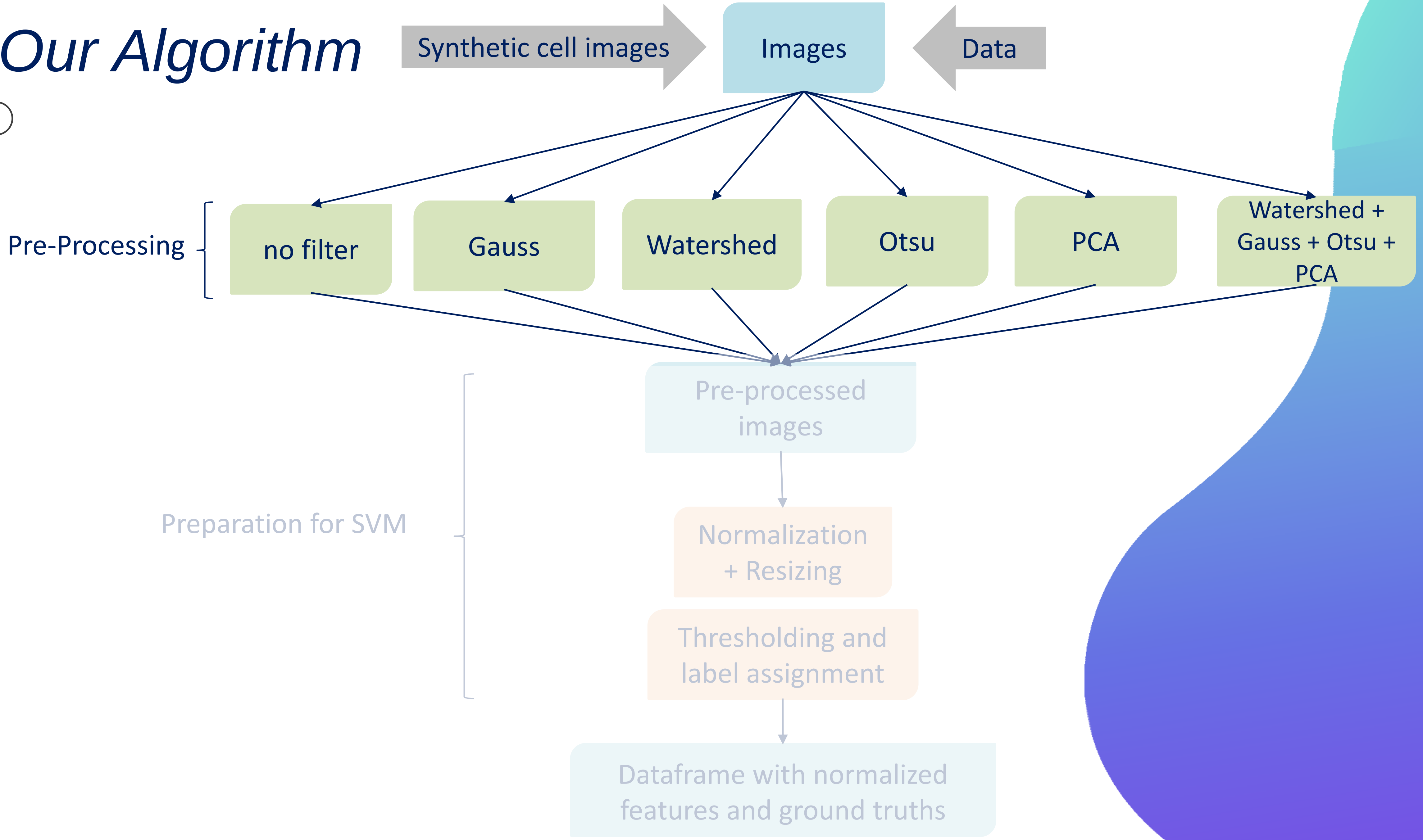
SVM

Segmented images

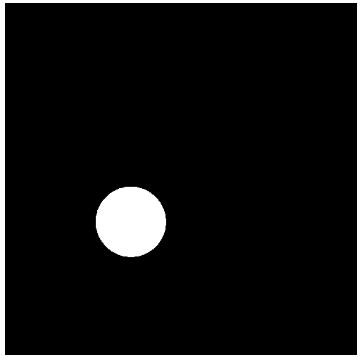
Evaluation

Dice score

Our Algorithm

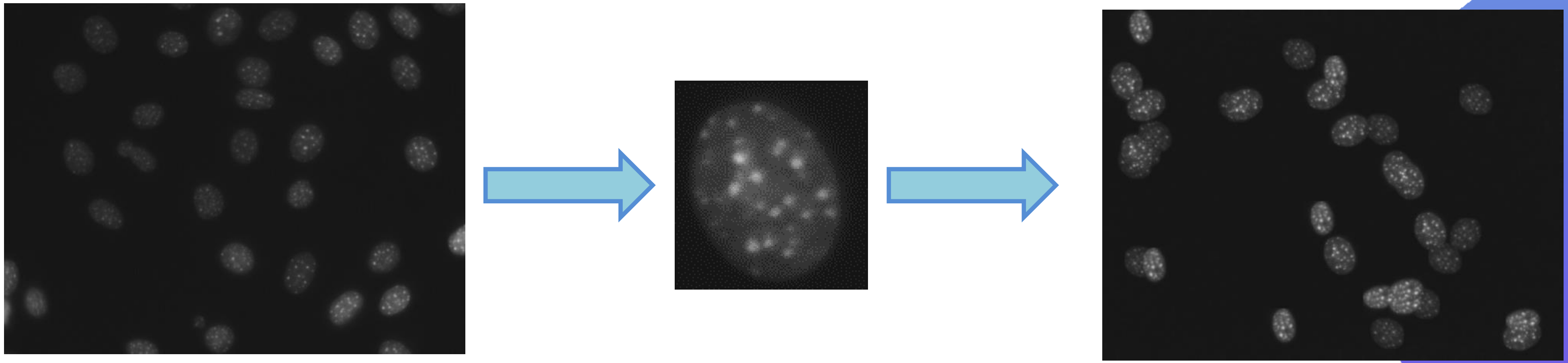


Synthetic images

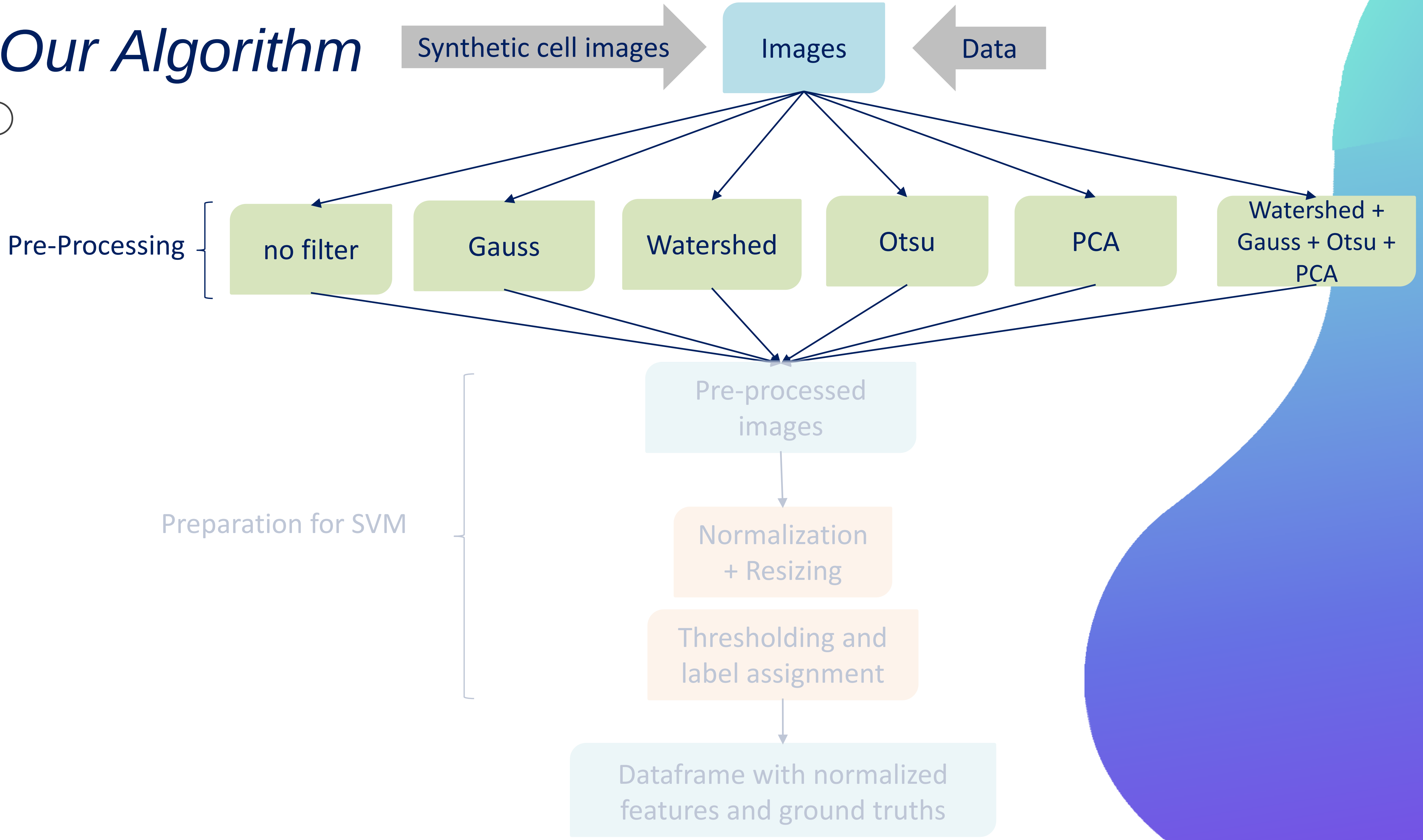


Goals:
- test the Dice score code
- enrich and enlarge our training data set for the SVM

Result:
- did not improve Dice score
- advantage: being able to segment almost all original images



Our Algorithm



Pre-processing Methods

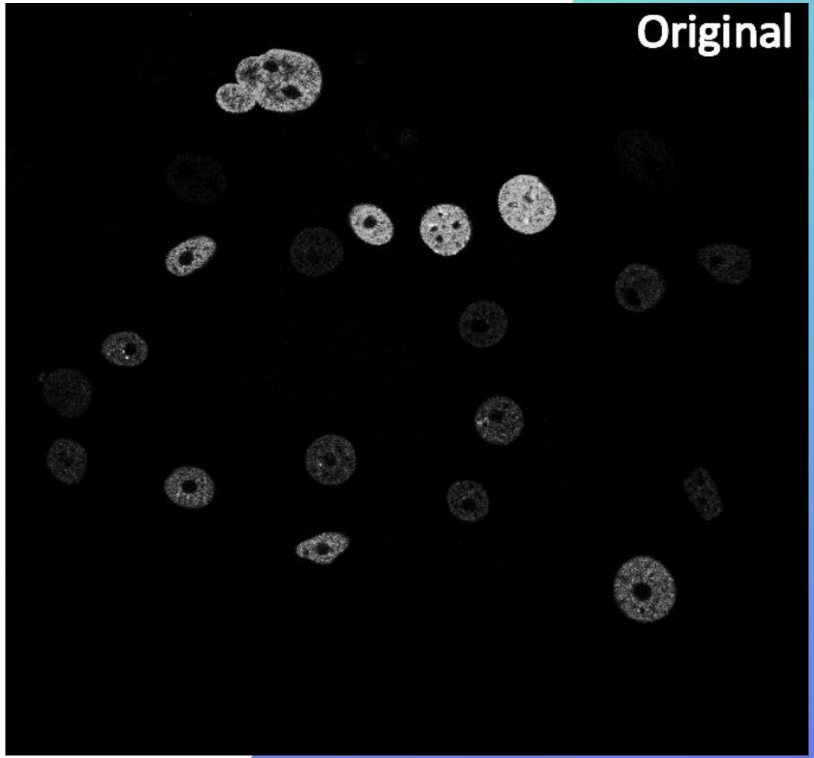
Goal:

improve Dice Score of segmentation method through better image quality

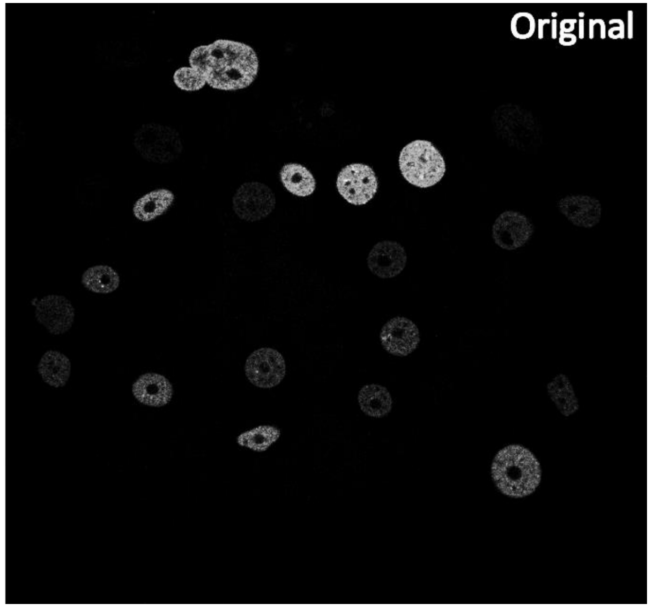
Desired effect:

1. average local pixel intensity values
2. separate nuclei which appear fused
3. increase contrast
4. enhance edges

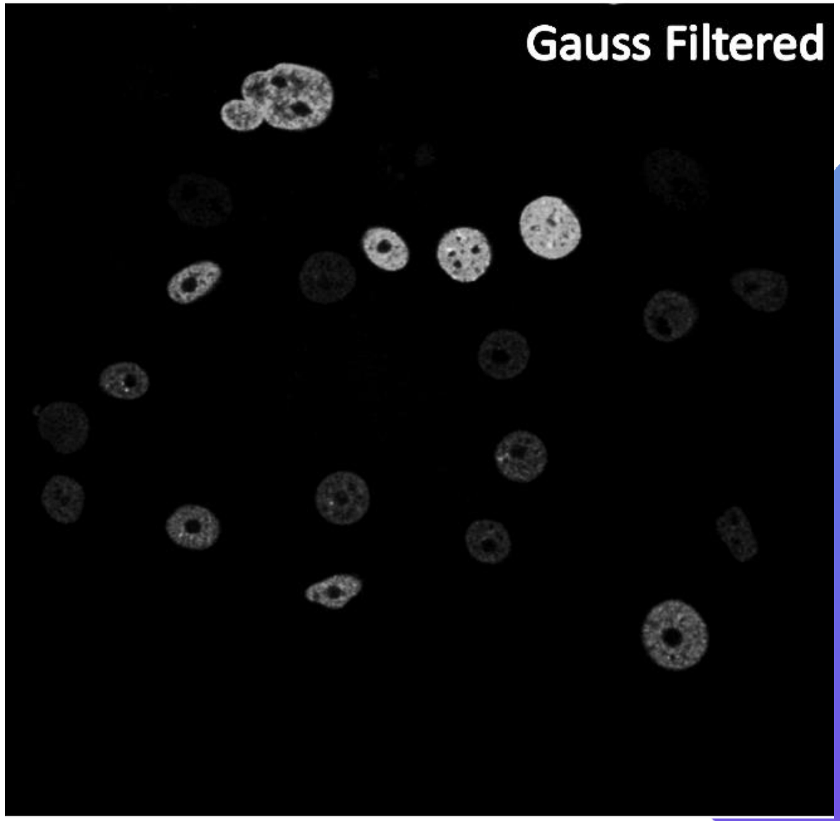
through noise reduction, super-pixel segmentation , thresholding and data selection.



Pre-processing Methods



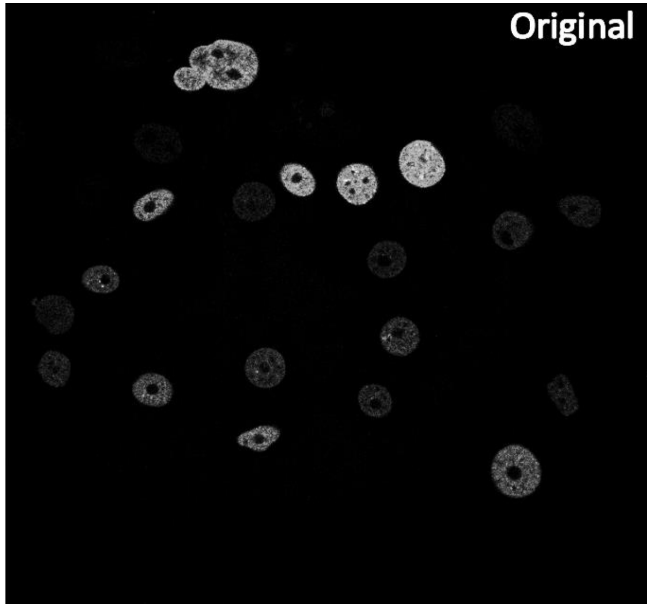
Gaussian Filtering



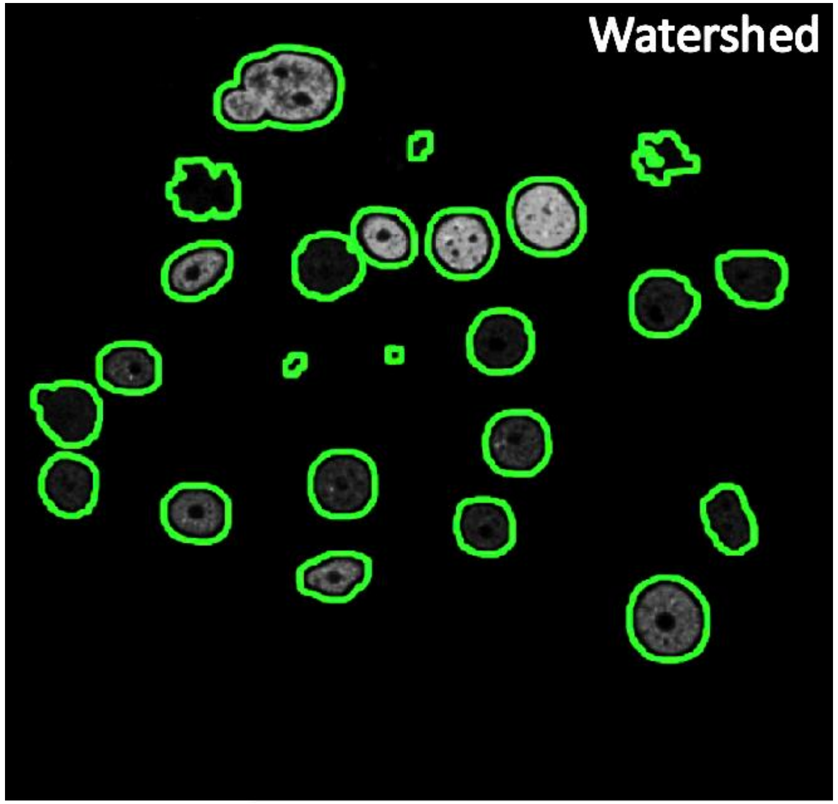
Desired effect:

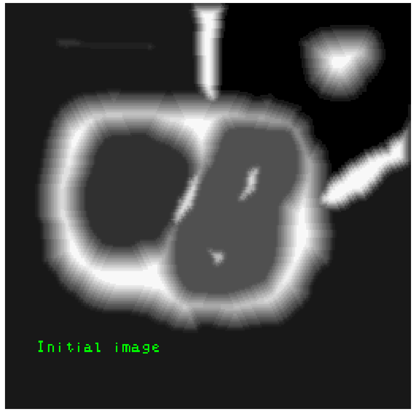
noise reduction

Pre-processing Methods



Watershed Filtering





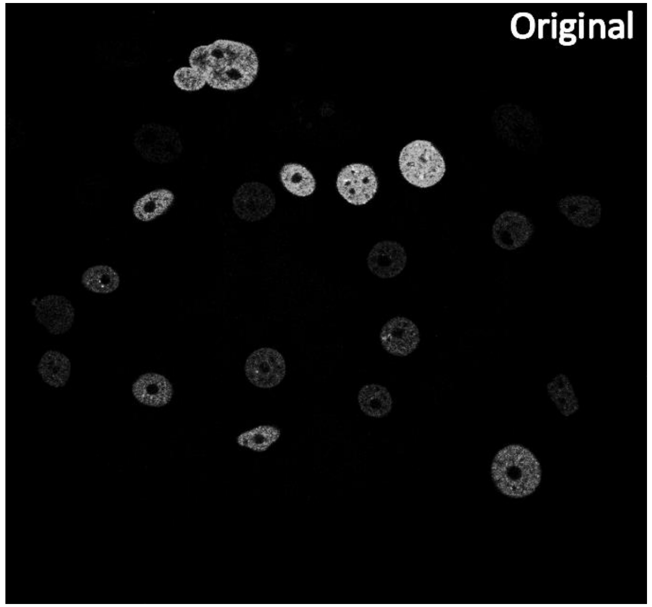
http://www.cmm.mines-paristech.fr/~beucher/wtshed.html

Desired effect:

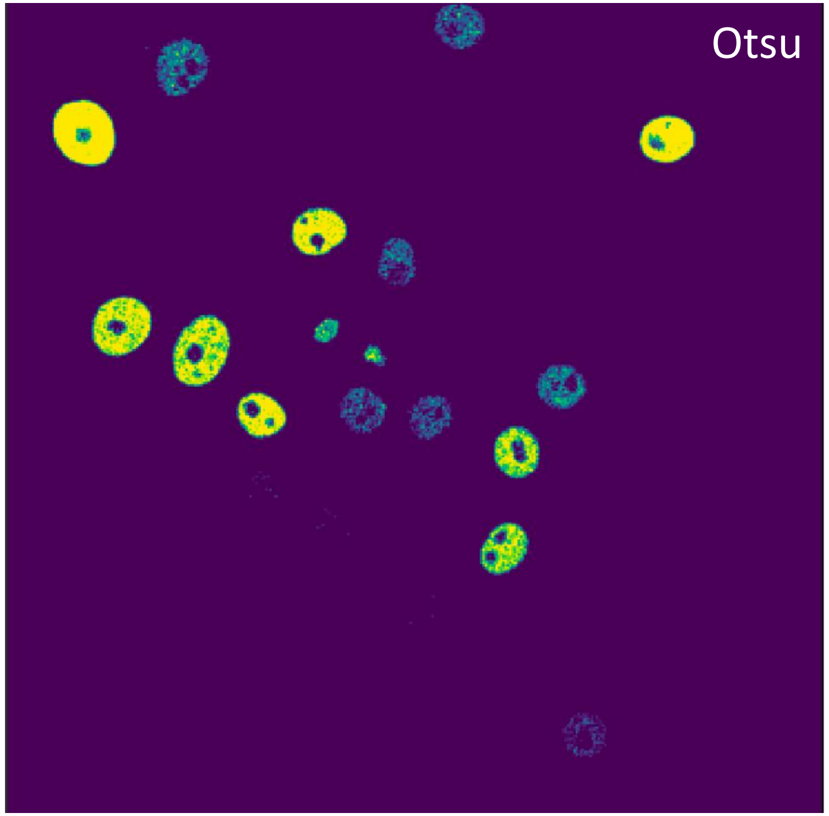
edge enhancement

separate fused nuclei

Pre-processing Methods



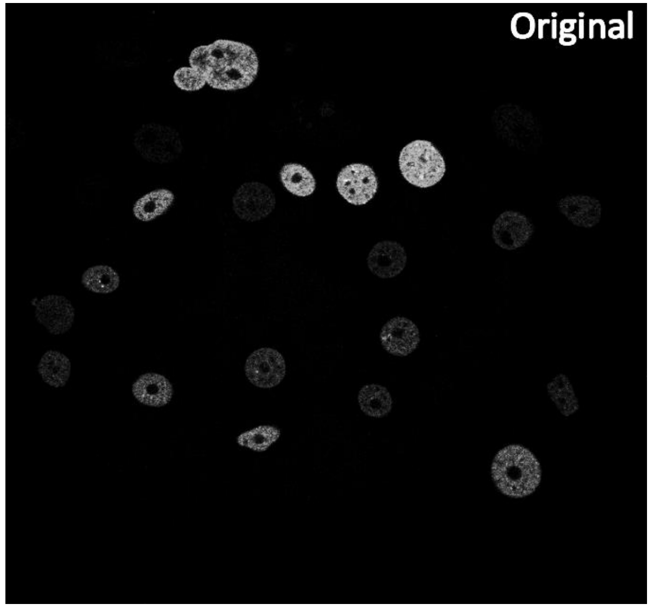
Otsu



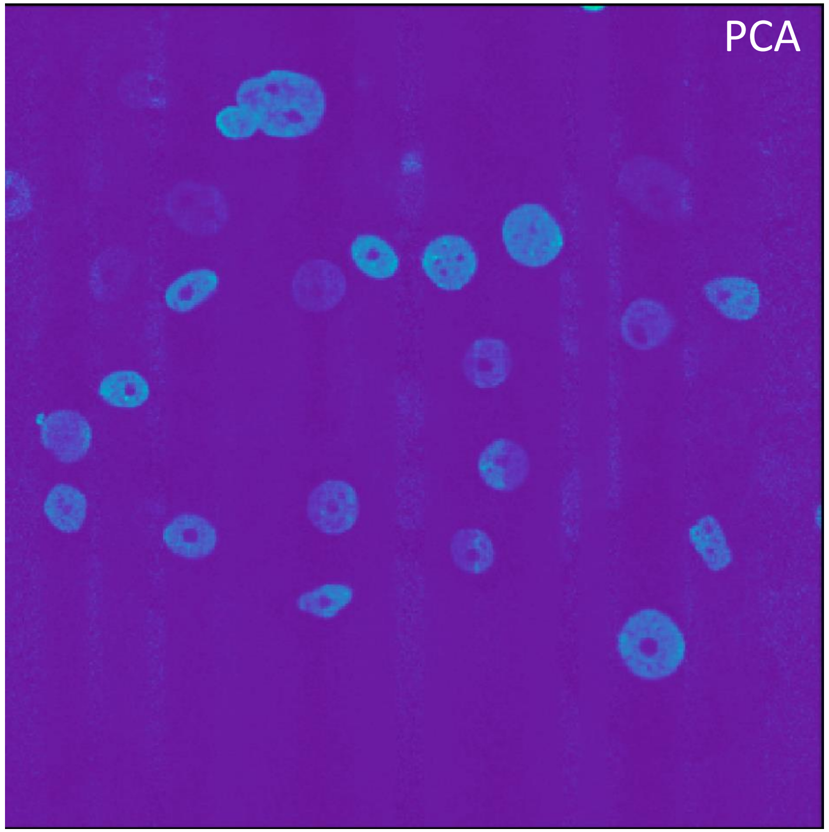
Desired effect:

increase contrast and enhance edges

Pre-processing Methods



PCA

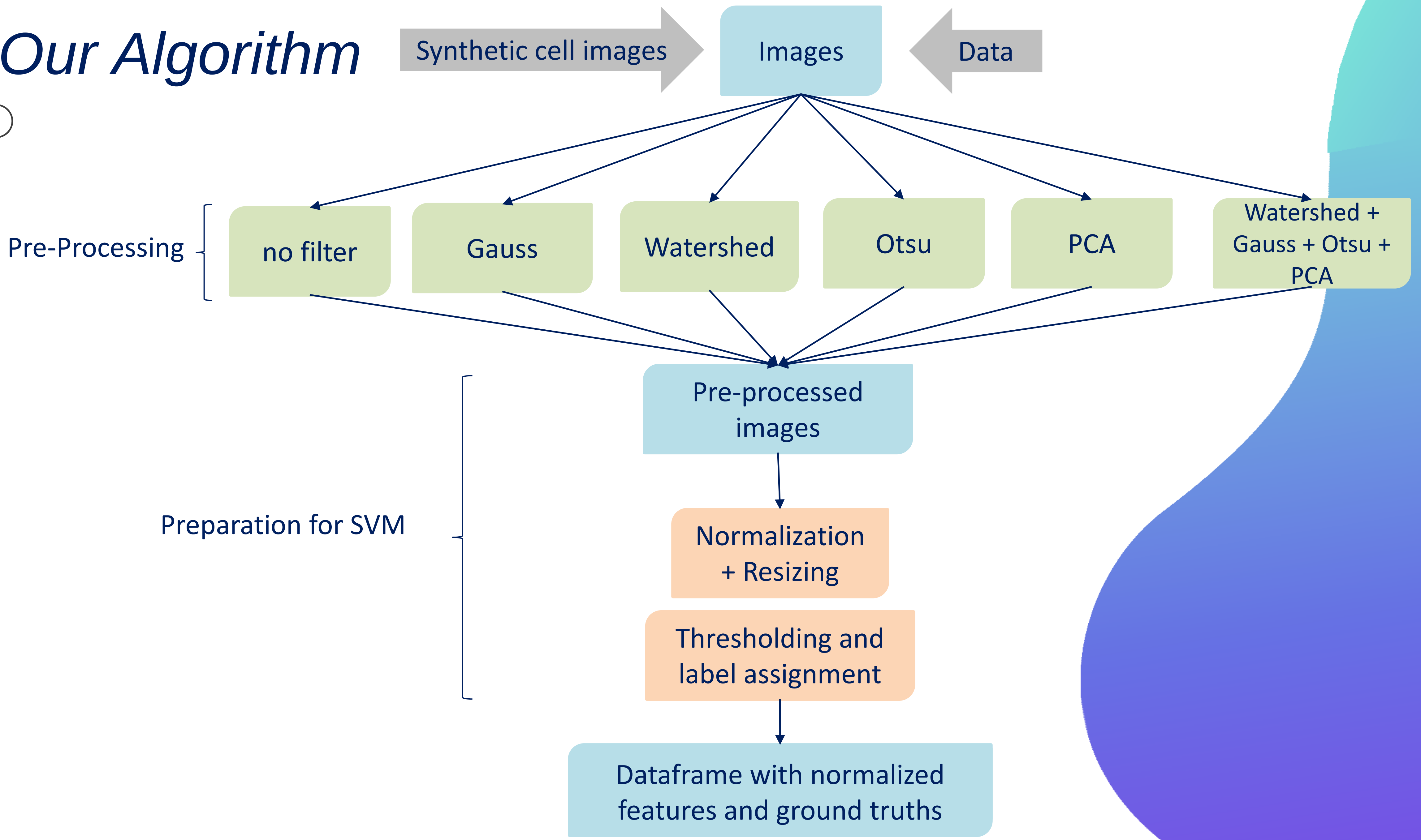


Desired effect:

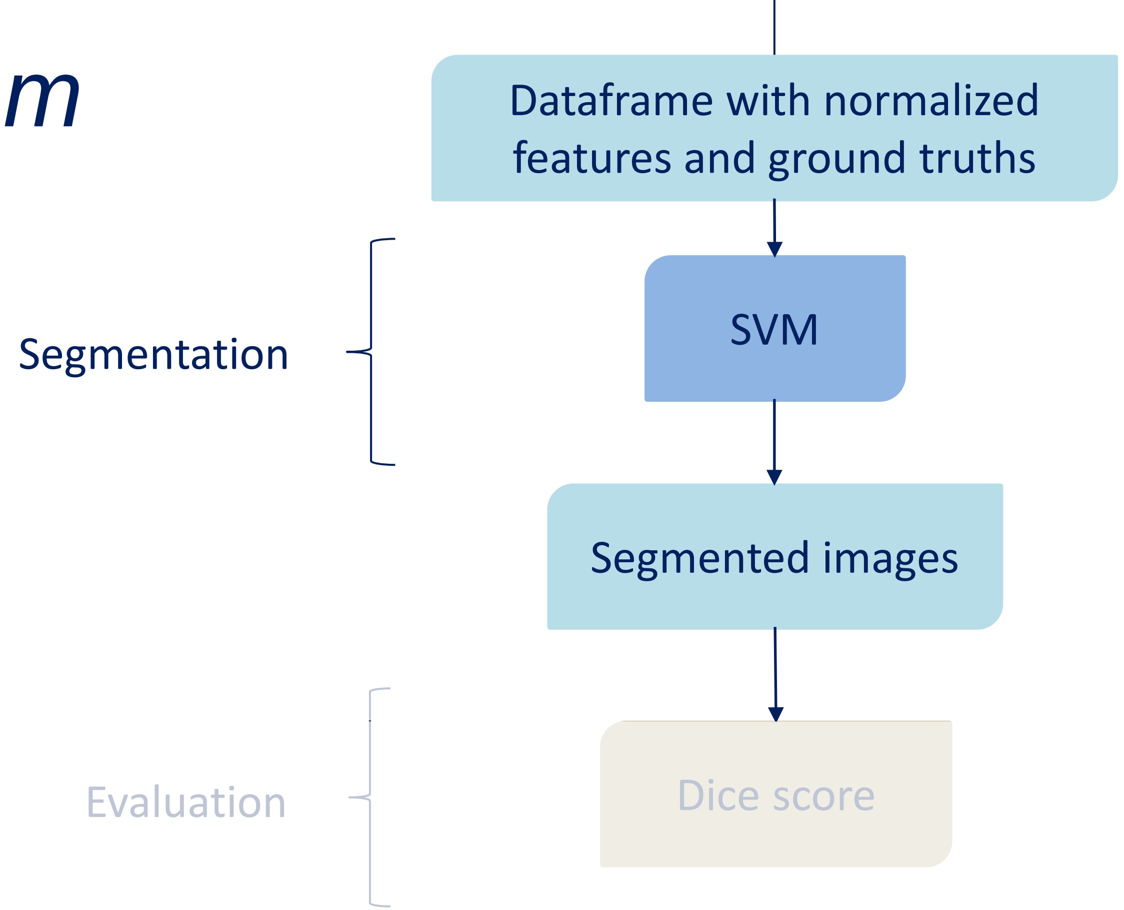
reduce noise and increase contrast

not used to for dimension reduction!

Our Algorithm



Our Algorithm



Support vector machine

calculate **cost**

↓

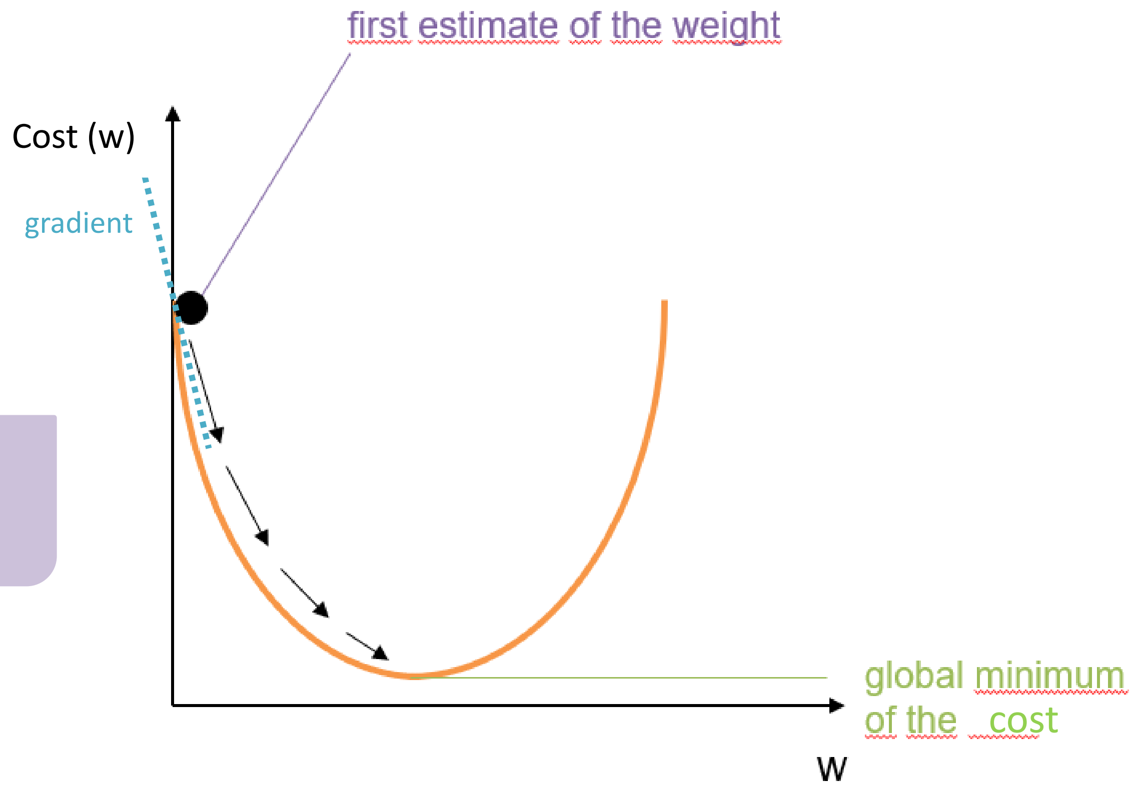
calculate **cost-gradient**

↓

minimize gradient using **stochastic gradient decent**

↓

train model



desired output:
find a hyperplane that separates samples belonging to two classes with the largest margin possible, while keeping the misclassification low
→ minimize loss function

Support vector machine

calculate **cost**

$$\text{cost}\,(\mathrm{w}) = \frac{1}{2}\,||\mathrm{w}||^2 + C\left[\frac{1}{N}\,\sum_i^n max\,(0, 1 - \; y_i \; * (w \; * \; x_i + b))\right]$$

```
def compute_cost(weights, features, labels, soft_margin_factor):
    """
    This function calculates the cost for a given feature, label and the weights vector.
    First, the hinge loss is calculated, a loss function that determines the importance of
    misclassified pixels. The smaller the soft margin factor (smf) is, the smaller the hinge loss is, which means the
    SVM margin will be softer.
    The cost function depends on the weights vector and the hinge loss.
    :param soft_margin_factor: determines the importance of misclassified pixels.
    :param weights: Weights vector
    :param features: Feature vector, has different columns for the different features for each pixel. Every pixel is
    a row.
    :param labels: Labels vector, is either 1 or -1 for each pixel (each row).
    :return: The cost of the current weights vector at classifying the pixels.
    """
    # calculate hinge loss
    number_pixels = features.shape[0]
    distances_to_hyperplane = 1 - labels * (np.dot(features, weights))
    distances_to_hyperplane = np.maximum(0, distances_to_hyperplane)
    hinge_loss = soft_margin_factor * (np.sum(distances_to_hyperplane) / number_pixels)
    # calculate cost
    cost = 1 / 2 * np.dot(weights, weights) + hinge_loss
    return cost
```

Support vector machine

calculate **cost**

$$\text{cost}\,(\mathrm{w}) = \frac{1}{2}\,||\mathrm{w}||^2 + C\left[\frac{1}{N}\sum_i^n max\,(0, 1 - y_i * (w * x_i + b))\right]$$

```python
def compute_cost(weights, features, labels, soft_margin_factor):
    """
    This function calculates the cost for a given feature, label and the weights vector.
    First, the hinge loss is calculated, a loss function that determines the importance of
    misclassified pixels. The smaller the soft margin factor (smf) is, the smaller the hinge loss is, which means the
    SVM margin will be softer.
    The cost function depends on the weights vector and the hinge loss.
    :param soft_margin_factor: determines the importance of misclassified pixels.
    :param weights: Weights vector
    :param features: Feature vector, has different columns for the different features for each pixel. Every pixel is
    a row.
    :param labels: Labels vector, is either 1 or -1 for each pixel (each row).
    :return: The cost of the current weights vector at classifying the pixels.
    """
    # calculate hinge loss
    number_pixels = features.shape[0]
    distances_to_hyperplane = 1 - labels * (np.dot(features, weights))
    distances_to_hyperplane = np.maximum(0, distances_to_hyperplane)
    hinge_loss = soft_margin_factor * (np.sum(distances_to_hyperplane) / number_pixels)
    # calculate cost
    cost = 1 / 2 * np.dot(weights, weights) + hinge_loss
    return cost
```

Support vector machine

calculate **cost**

$$\text{cost}\,(\mathrm{w}) = \frac{1}{2}\,||\mathrm{w}||^2 + C\left[\frac{1}{N}\,\sum_i^n max\,(0, 1 - \,y_i\, * (w\, * \,x_i + b))\right]$$

```python
def compute_cost(weights, features, labels, soft_margin_factor):
    """
    This function calculates the cost for a given feature, label and the weights vector.
    First, the hinge loss is calculated, a loss function that determines the importance of
    misclassified pixels. The smaller the soft margin factor (smf) is, the smaller the hinge loss is, which means the
    SVM margin will be softer.
    The cost function depends on the weights vector and the hinge loss.
    :param soft_margin_factor: determines the importance of misclassified pixels.
    :param weights: Weights vector
    :param features: Feature vector, has different columns for the different features for each pixel. Every pixel is
    a row.
    :param labels: Labels vector, is either 1 or -1 for each pixel (each row).
    :return: The cost of the current weights vector at classifying the pixels.
    """
    # calculate hinge loss
    number_pixels = features.shape[0]
    distances_to_hyperplane = 1 - labels * (np.dot(features, weights))
    distances_to_hyperplane = np.maximum(0, distances_to_hyperplane)
    hinge_loss = soft_margin_factor * (np.sum(distances_to_hyperplane) / number_pixels)
    # calculate cost
    cost = 1 / 2 * np.dot(weights, weights) + hinge_loss
    return cost
```

Support vector machine

calculate **cost**

$$\text{cost}\,(\mathrm{w}) = \frac{1}{2}\,||\mathrm{w}||^2 + C\left[\frac{1}{N}\sum_i^n max\,(0, 1 - \, y_i \, * (w \, * \, x_i + b))\right]$$

```python
def compute_cost(weights, features, labels, soft_margin_factor):
    """
    This function calculates the cost for a given feature, label and the weights vector.
    First, the hinge loss is calculated, a loss function that determines the importance of
    misclassified pixels. The smaller the soft margin factor (smf) is, the smaller the hinge loss is, which means the
    SVM margin will be softer.
    The cost function depends on the weights vector and the hinge loss.
    :param soft_margin_factor: determines the importance of misclassified pixels.
    :param weights: Weights vector
    :param features: Feature vector, has different columns for the different features for each pixel. Every pixel is
    a row.
    :param labels: Labels vector, is either 1 or -1 for each pixel (each row).
    :return: The cost of the current weights vector at classifying the pixels.
    """
    # calculate hinge loss
    number_pixels = features.shape[0]
    distances_to_hyperplane = 1 - labels * (np.dot(features, weights))
    distances_to_hyperplane = np.maximum(0, distances_to_hyperplane)
    hinge_loss = soft_margin_factor * (np.sum(distances_to_hyperplane) / number_pixels)
    # calculate cost
    cost = 1 / 2 * np.dot(weights, weights) + hinge_loss
    return cost
```

Support vector machine

calculate **cost**

↓

calculate **cost-gradient**

$$\mathrm{cost}\,(\mathrm{w}) = \frac{1}{2}\,||\mathrm{w}||^2 + C\left[\frac{1}{N}\sum_i^n max\,(0, 1 - \, y_i \, * (w \, * \, x_i + b))\right]$$

$$\nabla_w\, \mathrm{cost}\,(\mathrm{w}) = \frac{1}{N}\sum_i^n \begin{cases} w & if\ max\,(0, 1 - \, y_i \, * (w \, * \, x_i)) = 0 \\ w \, - C\ y_i x_i & otherwise \end{cases}$$

Support vector machine

calculate **cost-gradient**

$$\nabla_w \text{ cost }(\text{w}) = \frac{1}{N}\sum_i^n \begin{cases} w & if\ max\,(0, 1 - \ y_i\ * (w\ *\ x_i)) = 0 \\ w\ - C\ y_i x_i & otherwise \end{cases}$$

```python
def calculate_cost_gradient(weights, features, labels, soft_margin_factor):
    """
    This function calculates the gradient of the cost function. It computes its derivative, in order to know in which
    direction the gradient descent has to go to find the minimum of the cost function.
    :param soft_margin_factor: determines the importance of misclassified pixels.
    :param weights: weights vector.
    :param features: a vector with all our pixels as rows.
    :param labels: label of the pixels, either +1 or -1.
    :return: gradient of the cost function with that weight vector.
    """
    # In order to iterate correctly, we turn our vectors into arrays.
    labels = np.array([labels])
    features = np.array([features])

    # Calculate distance to hyperplane, to classify the pixels. This dot product can be changed depending on the
    # chosen kernel.
    distance_to_hyperplane = 1 - (labels * np.dot(features, weights))

    # Create an empty gradient vector, to fill with the gradient of the current pixel.
    gradient_cost = np.zeros(len(weights))

    for index_pixel, distance_pixel in enumerate(distance_to_hyperplane):
        # For correctly classified pixels, the current weight vector is maintained
        if max(0, distance_pixel) == 0:
            gradient_pixel = weights
        # For incorrectly classified pixels, the weight vector is corrected in the direction contrary to the gradient.
        else:
            gradient_pixel = weights - (soft_margin_factor * labels[index_pixel] * features[index_pixel])
        gradient_cost += gradient_pixel

    return gradient_cost
```

Support vector machine

calculate **cost-gradient**

$$\nabla_w \text{ cost (w)} = \frac{1}{N}\sum_{i}^{n}\begin{cases} w & \text{if } max\,(0, 1 - y_i * (w * x_i)) = 0 \\ w - C\, y_i x_i & \text{otherwise} \end{cases}$$

```python
def calculate_cost_gradient(weights, features, labels, soft_margin_factor):
    """
```

```python
    distance_to_hyperplane = 1 - (labels * np.dot(features, weights))

    # Create an empty gradient vector, to fill with the gradient of the current pixel.
    gradient_cost = np.zeros(len(weights))

    for index_pixel, distance_pixel in enumerate(distance_to_hyperplane):
        # For correctly classified pixels, the current weight vector is maintained
        if max(0, distance_pixel) == 0:
            gradient_pixel = weights
        # For incorrectly classified pixels, the weight vector is corrected in the direction contrary to the gradient.
        else:
            gradient_pixel = weights - (soft_margin_factor * labels[index_pixel] * features[index_pixel])
        gradient_cost += gradient_pixel

    return gradient_cost
```


```python
        else:
            gradient_pixel = weights - (soft_margin_factor * labels[index_pixel] * features[index_pixel])
        gradient_cost += gradient_pixel

    return gradient_cost
```

Support vector machine

calculate **cost-gradient**

$$\nabla_w \text{ cost (w)} = \frac{1}{N} \sum_{i}^{n} \begin{cases} w & if\ max\ (0, 1 - \ y_i\ * (w\ *\ x_i)) = 0 \\ w\ - C\ y_i x_i & otherwise \end{cases}$$

```python
def calculate_cost_gradient(weights, features, labels, soft_margin_factor):
    """
        distance_to_hyperplane = 1 - (labels * np.dot(features, weights))

        # Create an empty gradient vector, to fill with the gradient of the current pixel.
        gradient_cost = np.zeros(len(weights))

        for index_pixel, distance_pixel in enumerate(distance_to_hyperplane):
            # For correctly classified pixels, the current weight vector is maintained
            if max(0, distance_pixel) == 0:
                gradient_pixel = weights
            # For incorrectly classified pixels, the weight vector is corrected in the direction contrary to the gradient.
            else:
                gradient_pixel = weights - (soft_margin_factor * labels[index_pixel] * features[index_pixel])
            gradient_cost += gradient_pixel

        return gradient_cost
```


```python
        else:
            gradient_pixel = weights - (soft_margin_factor * labels[index_pixel] * features[index_pixel])
        gradient_cost += gradient_pixel

    return gradient_cost
```

Support vector machine

calculate **cost**

$$\text{cost}\,(\text{w}) = \frac{1}{2}\,||\text{w}||^2 + C\left[\frac{1}{N}\sum_i^n max\,(0, 1 - \,y_i\, * (w\, *\, x_i + b))\right]$$

calculate **cost-gradient**

$$\nabla_w\,\text{cost}\,(\text{w}) = \frac{1}{N}\sum_i^n \begin{cases} w & if\ max\,(0, 1 - \,y_i\, * (w\, *\, x_i)) = 0 \\ w\, - C\, y_i x_i & otherwise \end{cases}$$

minimize gradient using **stochastic gradient decent**

Support vector machine

minimize gradient using **stochastic gradient decent**

```python
def sgd(features, labels, soft_margin_factor, learning_rate, max_epochs):
    """
    This function calculates the stochastic gradient descent to minimize our cost function.
    :param features: all pixels as rows, with n columns, which stand for n features.
    :param labels: all pixels as rows, only one column per pixel, either +1 or -1.
    :param soft_margin_factor:
    :param learning_rate:
    :param max_epochs: maximum number of cycles to try to find the minimum of the cost function.
    :return: the weight vector found at the minimum of the cost function, and the history of how the costs have
    evolved during the calculations.
    """
    # Create an empty weight vector that is the same size as the number of columns (features) of a single pixel.
    weights = np.zeros(features.shape[1])
    # Define the first cost for our function and an empty history cost list.
    prev_cost = float("inf")
    history_cost = []
    patience = 0

    # Stochastic gradient descent
    for epoch in range(0, max_epochs):
        # shuffle to prevent repeating update cycles
        features_shuffled, labels_shuffled = shuffle(features, labels)
        for pixel_index, pixel_value in enumerate(features_shuffled):
            gradient = calculate_cost_gradient(weights, pixel_value, labels_shuffled[pixel_index], soft_margin_factor)
            weights = weights - (learning_rate * gradient)

        # Calculate cost to evaluate the advance.
        cost = compute_cost(weights, features, labels, soft_margin_factor)
        history_cost.append(cost)
        if epoch % 20 == 0 or epoch == max_epochs - 1:
            print("Epoch is: {} and Cost is: {}".format(epoch, cost))

        # Stoppage criterion
        if prev_cost < cost:
            if patience == 10:
                return weights, history_cost
            else:
                patience += 1
        else:
            patience = 0
        prev_cost = cost

    return weights, history_cost
```

Support vector machine

minimize gradient using **stochastic gradient decent**

```
def sgd(features, labels, soft_margin_factor, learning_rate, max_epochs):
    """
    This function calculates the stochastic gradient descent to minimize our cost function.
        for pixel_index, pixel_value in enumerate(features_shuffled):
            gradient = calculate_cost_gradient(weights, pixel_value, labels_shuffled[pixel_index], soft_margin_factor)
            weights = weights - (learning_rate * gradient)

        # Calculate cost to evaluate the advance.
        cost = compute_cost(weights, features, labels, soft_margin_factor)
        history_cost.append(cost)
        if epoch % 20 == 0 or epoch == max_epochs - 1:
            print("Epoch is: {} and Cost is: {}".format(epoch, cost))

        # Stoppage criterion
        if prev_cost < cost:
            if patience == 10:
                return weights, history_cost
            else:
                patience += 1
        else:
            patience = 0
        prev_cost = cost

    return weights, history_cost
            patience = 0
        prev_cost = cost

    return weights, history_cost
```

Support vector machine

minimize gradient using **stochastic gradient decent**

```python
def sgd(features, labels, soft_margin_factor, learning_rate, max_epochs):
    """
    This function calculates the stochastic gradient descent to minimize our cost function.
```

```python
        for pixel_index, pixel_value in enumerate(features_shuffled):
            gradient = calculate_cost_gradient(weights, pixel_value, labels_shuffled[pixel_index], soft_margin_factor)
            weights = weights - (learning_rate * gradient)

        # Calculate cost to evaluate the advance.
        cost = compute_cost(weights, features, labels, soft_margin_factor)
        history_cost.append(cost)
        if epoch % 20 == 0 or epoch == max_epochs - 1:
            print("Epoch is: {} and Cost is: {}".format(epoch, cost))

        # Stoppage criterion
        if prev_cost < cost:
            if patience == 10:
                return weights, history_cost
            else:
                patience += 1
        else:
            patience = 0
        prev_cost = cost

    return weights, history_cost
```

```python
                patience = 0
            prev_cost = cost

    return weights, history_cost
```

Support vector machine

minimize gradient using **stochastic gradient decent**

```
def sgd(features, labels, soft_margin_factor, learning_rate, max_epochs):
    """
    This function calculates the stochastic gradient descent to minimize our cost function.
        for pixel_index, pixel_value in enumerate(features_shuffled):
            gradient = calculate_cost_gradient(weights, pixel_value, labels_shuffled[pixel_index], soft_margin_factor)
            weights = weights - (learning_rate * gradient)

        # Calculate cost to evaluate the advance.
        cost = compute_cost(weights, features, labels, soft_margin_factor)
        history_cost.append(cost)
        if epoch % 20 == 0 or epoch == max_epochs - 1:
            print("Epoch is: {} and Cost is: {}".format(epoch, cost))

        # Stoppage criterion
        if prev_cost < cost:
            if patience == 10:
                return weights, history_cost
            else:
                patience += 1
        else:
            patience = 0
        prev_cost = cost

    return weights, history_cost
            patience = 0
        prev_cost = cost

    return weights, history_cost
```

Support vector machine

calculate **cost**

calculate **cost-gradient**

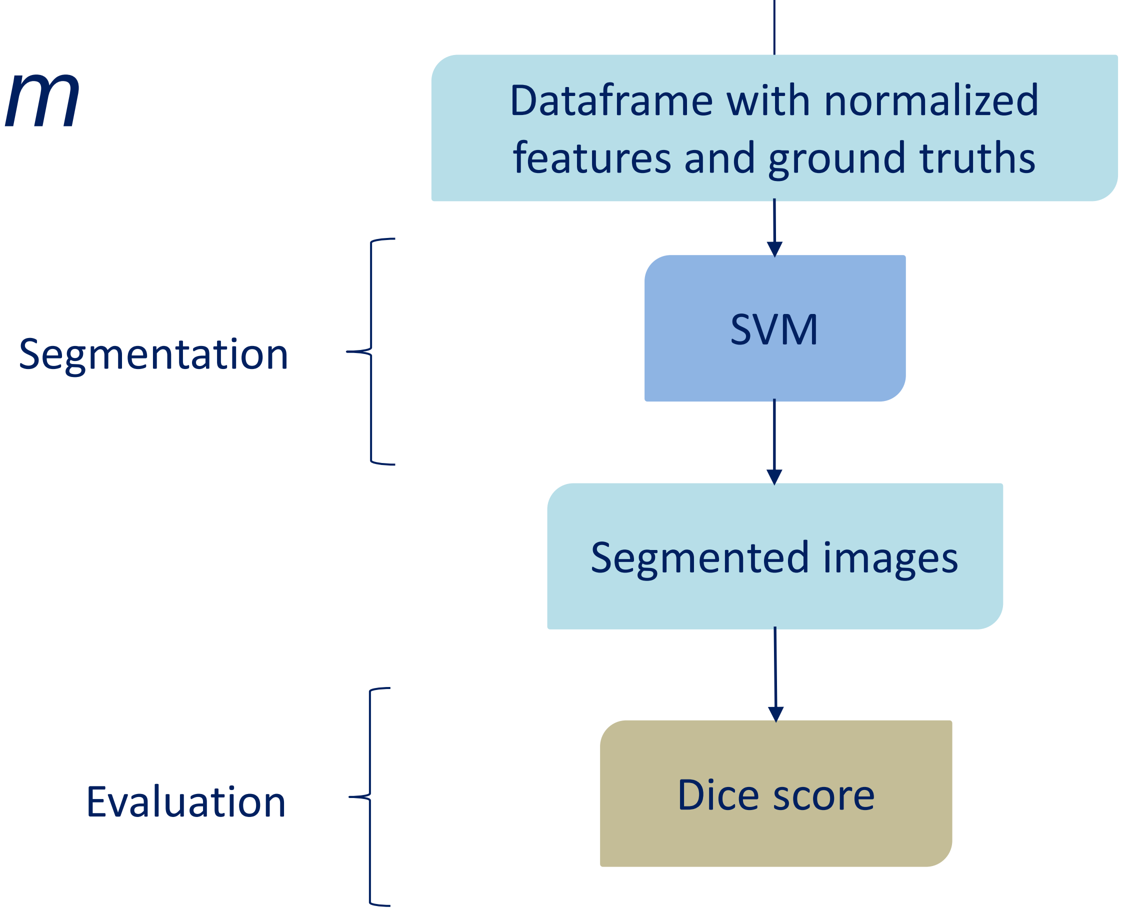
minimize gradient using **stochastic gradient decent**

train model

$$\mathrm{cost}\ (\mathrm{w}) = \frac{1}{2}\,||\mathrm{w}||^2 + C\left[\frac{1}{N}\sum_i^n max\,(0, 1 - \ y_i\ * (w\ *\ x_i + b))\right]$$

$$\nabla_w\ \mathrm{cost}\ (\mathrm{w})\ = \frac{1}{N}\sum_i^n \begin{cases} w & if\ max\,(0, 1 - \ y_i\ * (w\ *\ x_i)) = 0 \\ w\ - C\ y_i x_i & otherwise \end{cases}$$

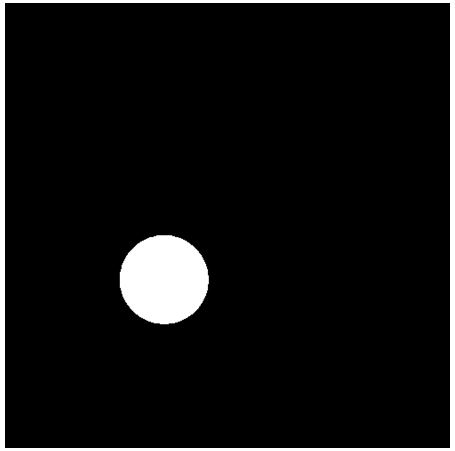
Our Algorithm



Evaluation using the dice score

```python
def dice_score(pred, gt):
    """
    This function calculates the similiarity between two arrays.
    :param pred: an array of predicted labels
    :param gt: the ground truth of this array
    :return: a value between 0 and 1, describing the similiarity between those arrays. 1 is the dice score of similiar arrays.
    """
    dice = np.sum(pred[gt == pred]) * 2.0 / (np.sum(gt) + np.sum(pred))
    print(dice)
```

$$\text{Dice} = \frac{2 * Intersection}{Union + Intersection} = \mathrm{F}_1 = \frac{1}{\frac{1}{Prec} + \frac{1}{Recall}} = \frac{2\,TP}{2\,TP + FP + FN}$$



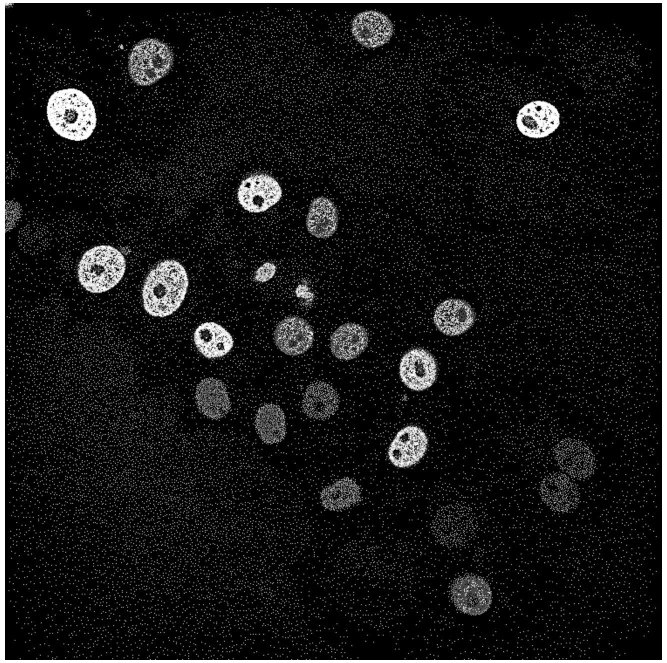
Unittesting with synthetic masks

and

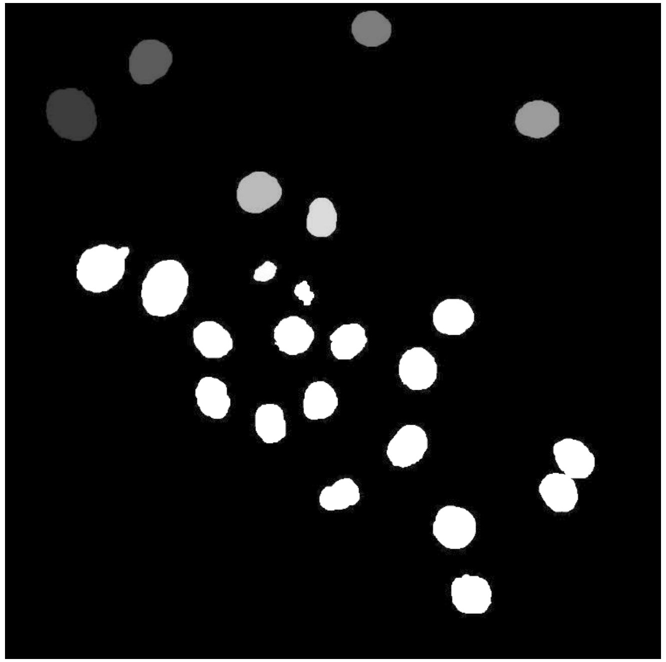
manually created arrays

Results: N2DH-GOW1

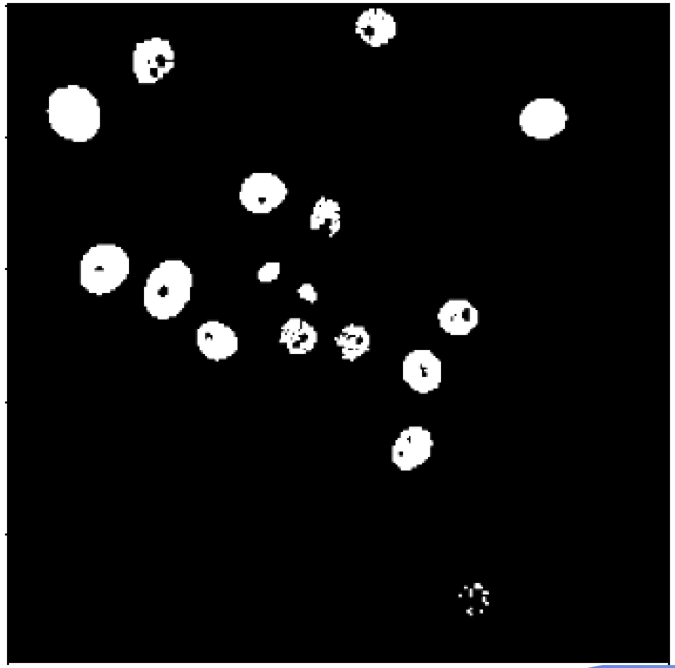
original image:
t21.tif



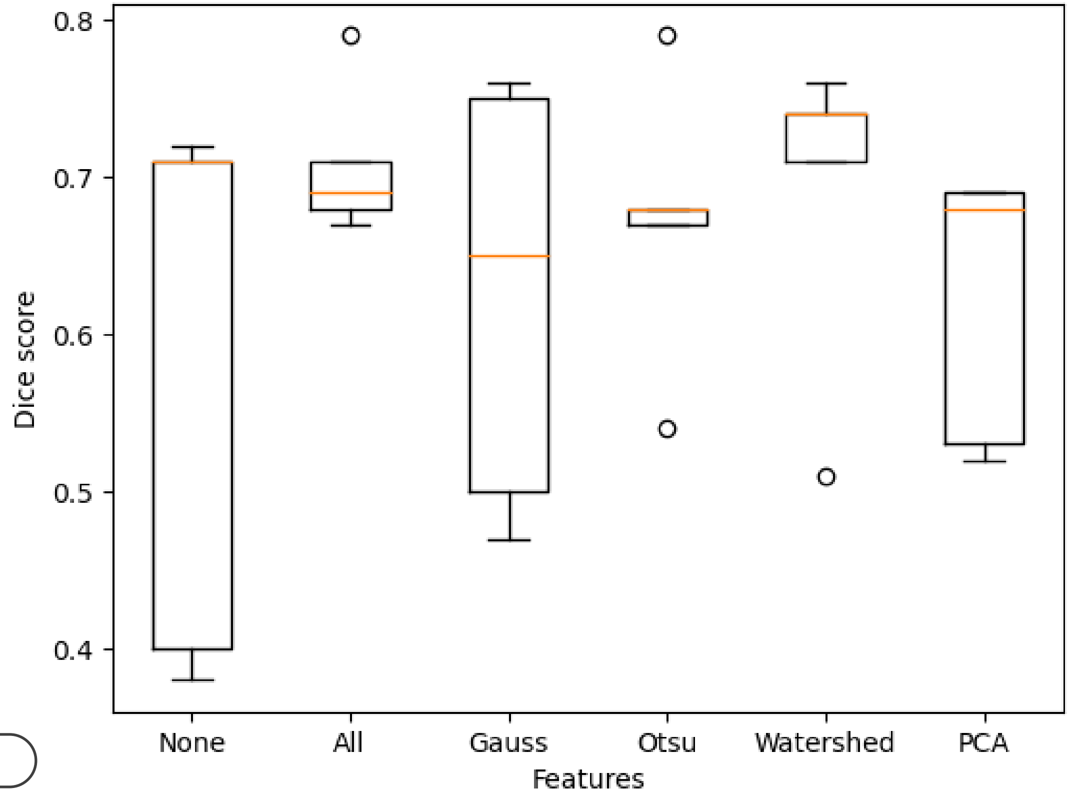
ground truth:
man_seg21.tif



segmented image:
t21_seg.tif
dice score: 0.67



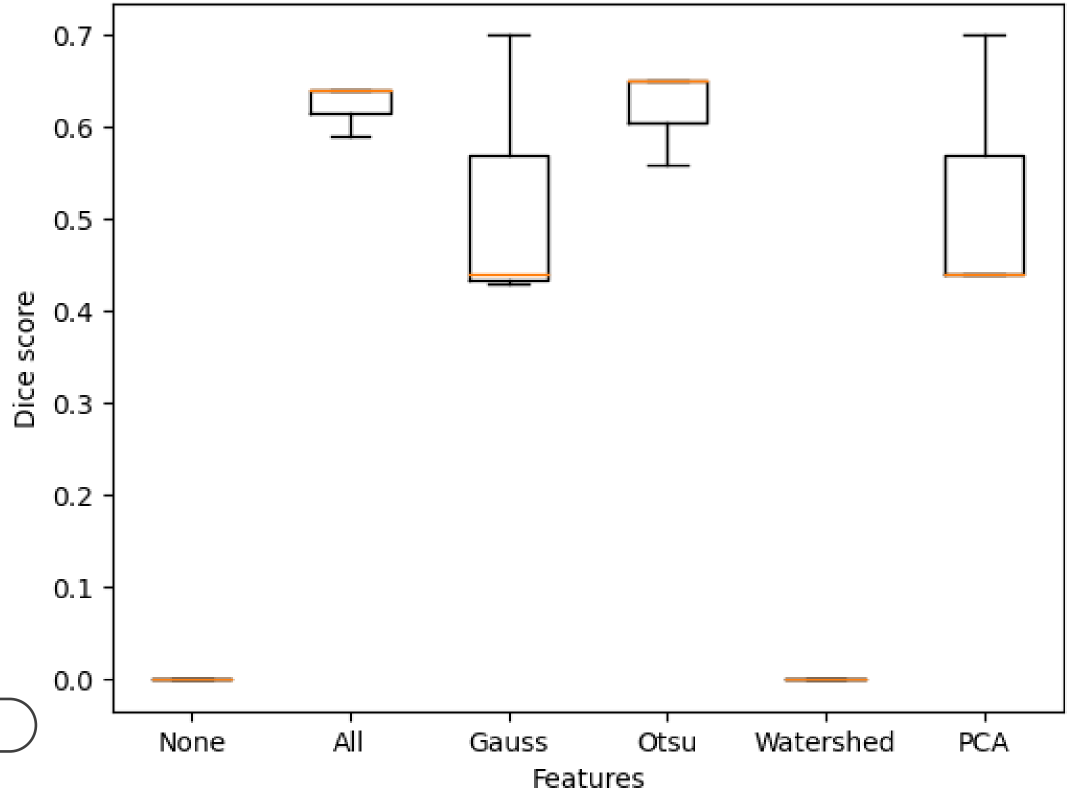
Results: N2DH-GOW1



Features	Dice Score (DS)
no filters (NF)	good DS, but highest variance
all filters	worse than NF, but lower variance
Gauss, Otsu, PCA	worse than NF, but lower variance
Watershed	best DS

- average dice score with all filters: 0.71

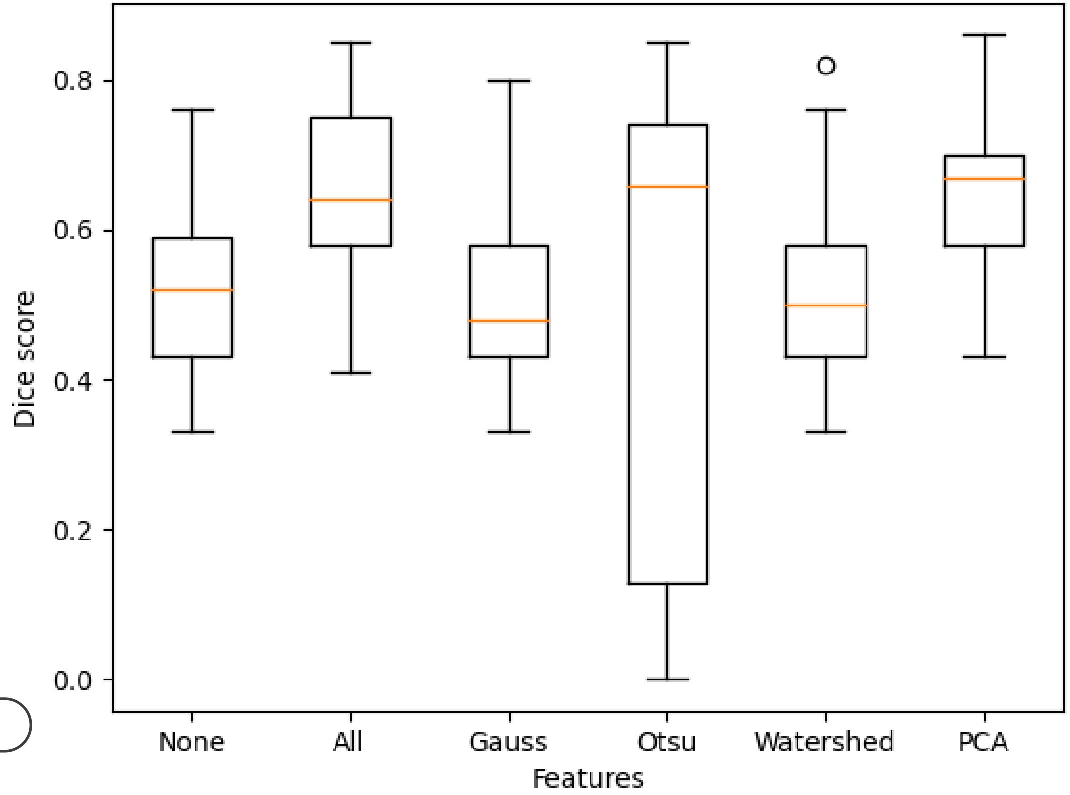
Results: N2DL-HeLa



Features	Dice Score (DS)
no filters (NF), Watershed	no segmentation
all filters	best DS
Otsu	biggest DS improvement
Gauss, PCA	not as relevant

- average Dice score with all filters: 0.63

Results: NIH3T3



Features	Dice Score (DS)
no filters (NF)	DS over 0.5
all filters, PCA	highest DS
Gauss, Watershed	similar to NF
Otsu	good DS, but high variance

- average Dice score with all filters: 0.65

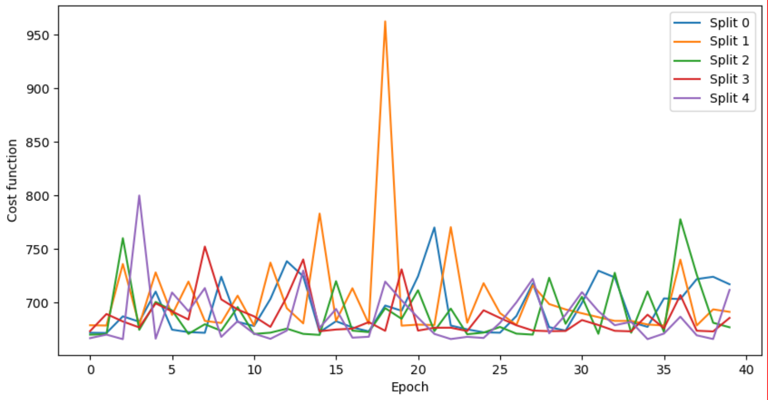
Optimal settings for our SVM

- Learning rate: $1 \cdot 10^{-7}$
- Regularization parameter: 10.000
- Maximum epochs: 40

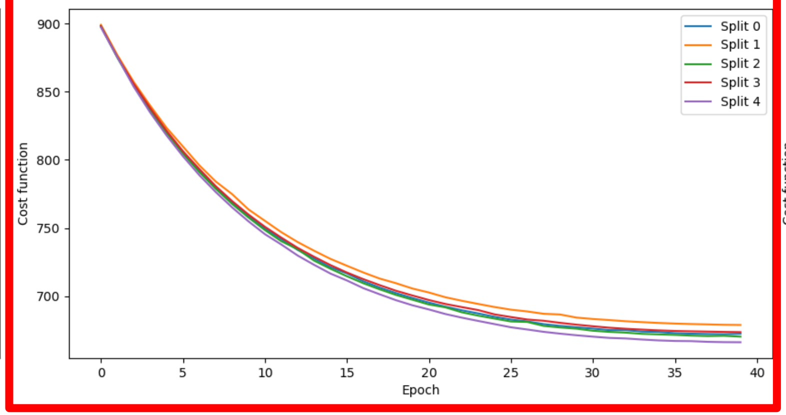
➢ *Why?*

  - Different learning rates
    - Regularization parameter: 10.000
    - Maximum epochs: 40

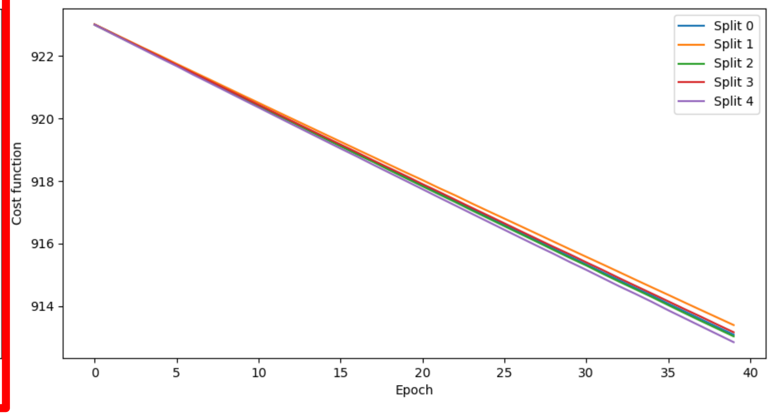
Learning rate: $1 \cdot 10^{-5}$



Learning rate: $1 \cdot 10^{-7}$



Learning rate: $1 \cdot 10^{-9}$



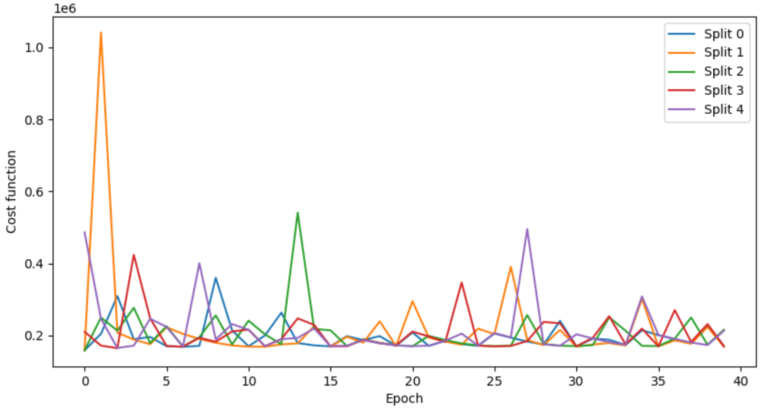
Optimal settings for our SVM

- Learning rate: $1 \cdot 10^{-7}$
- Regularization parameter: 10.000
- Maximum epochs: 40

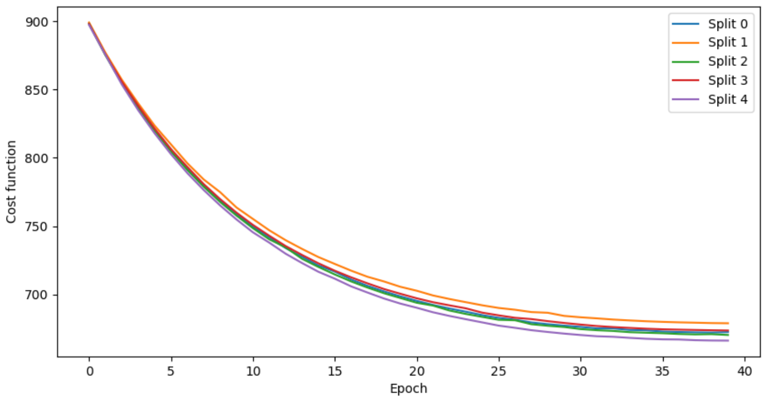
➢ *Why?*

  - Learning rate: $1 \cdot 10^{-7}$
  - Different regularization parameters
  - Maximum epochs: 40

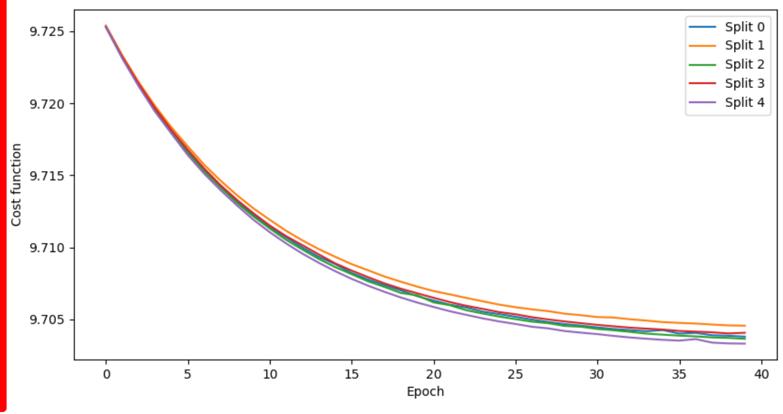
Regularization parameter: 10.000.000



Regularization parameter: 10.000



Regularization parameter: 100



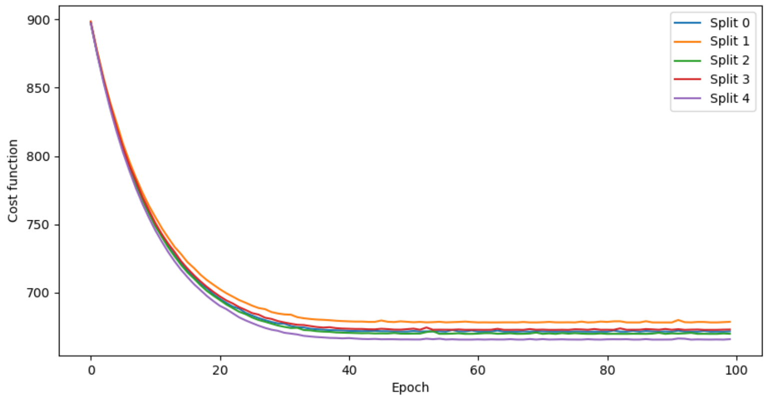
Optimal settings for our SVM

- Learning rate: $1 \cdot 10^{-7}$
- Regularization parameter: 10.000
- Maximum epochs: 40

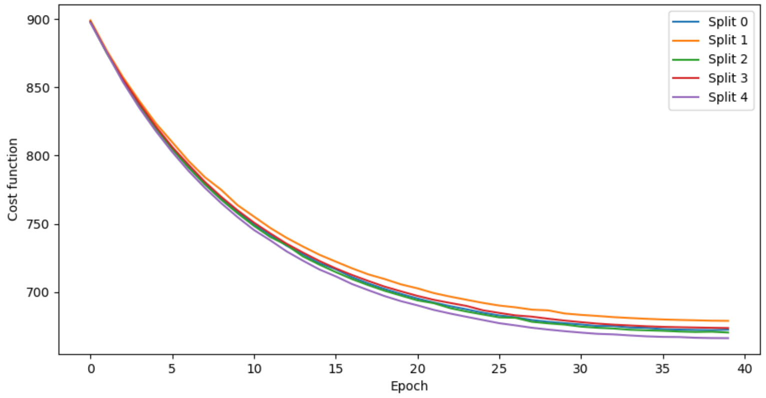
➢ *Why?*

  - Learning rate: $1 \cdot 10^{-7}$
  - Regularization parameter: 10.000
  - Different maximum epochs

Maximum epochs: 100



Maximum epochs: 40



What we’ve learned

- pre-processing methods depend on specific challenges of the data

- SVM with a Linear kernel is easier to code and takes less time to run

- SVM is a powerful tool for segmentation

But:

- needs a lot of computational power and thus has long runtime

Outlook

SVMs include many possibilities for expansion

- Non-linear kernels for better segementation results e.g. the RBF kernel
- include other features e.g. intensity values of the neighborhood
- test correlation of features to reduce runtime
- generate weights vector from changing starting points

Thank you for listening!