Paper: Label-Free Identification of White Blood Cells Using Machine Learning

Wenxuan Zhao

2025-06-30

Table of contents

1 Introduction 1

2 Method 3

2.1 ML Classification 4

2.2 Dataset Overview 4

2.3 Read single cell images of 1 patient (BF channel) 4

2.4 Features by category 10

2.4.1 Intensity Features 10

2.4.2 Granularity Features 11

2.4.3 Radial Distribution Features 16

2.4.4 Gabor Texture Features 22

2.4.5 Haralick Texture Features 22

2.4.6 Zernike Shape Features 26

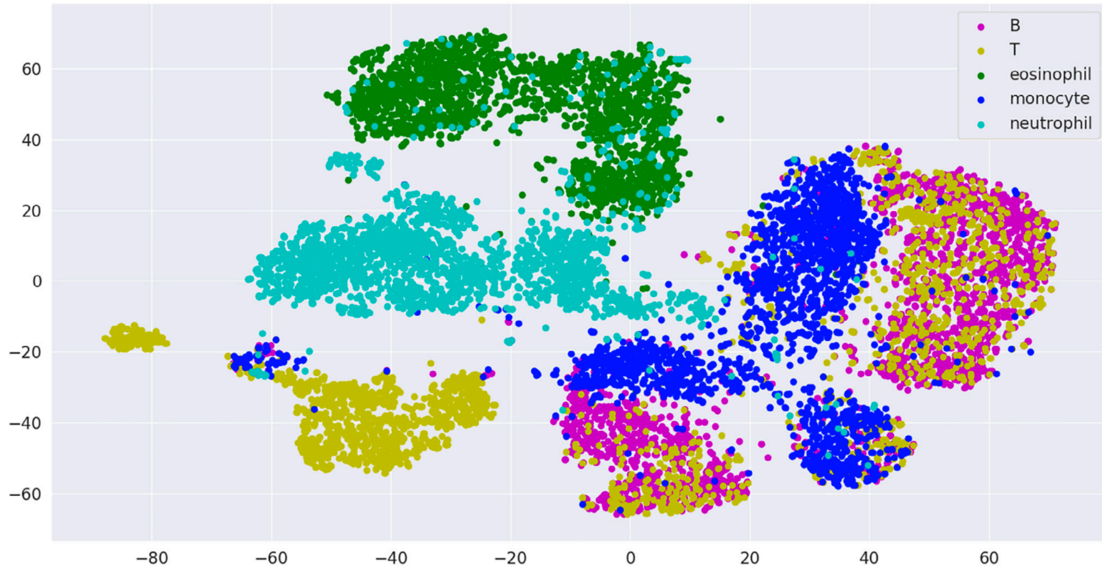
2.4.7 Shape Features 27

2.5 Top 10 Features from classification 27

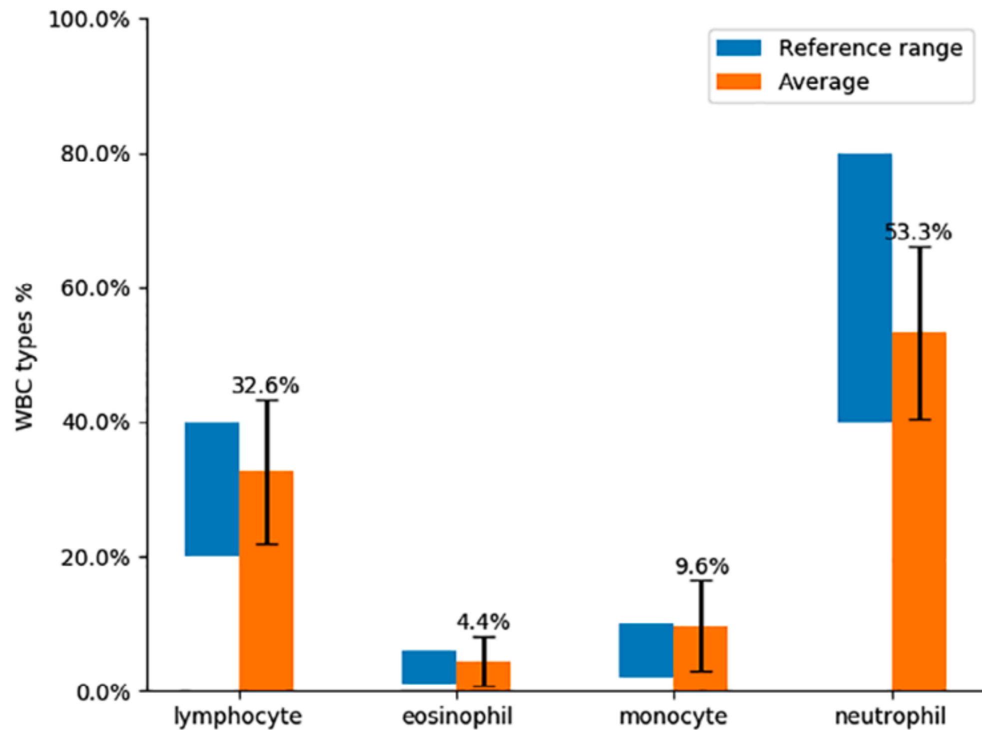
1 Introduction

- **Clinical goal**: White blood cell (WBC) differential counting is an established clinical routine to assess patient immune system status.
- **However**: Fluorescent markers and a flow cytometer are required for the current state-of-the-art method for determining WBC differential counts.
- **Goal**: Develop a method to identify WBCs without the need for fluorescent markers or a flow cytometer: *a novel label-free approach using an imaging flow cytometer and machine learning algorithms, where live, unstained WBCs were classified.*

- **Result**: It achieved an average F1-score of 97% among WBC cell types and two subtypes of WBCs, B and T lymphocytes with an average F1-score of 78%



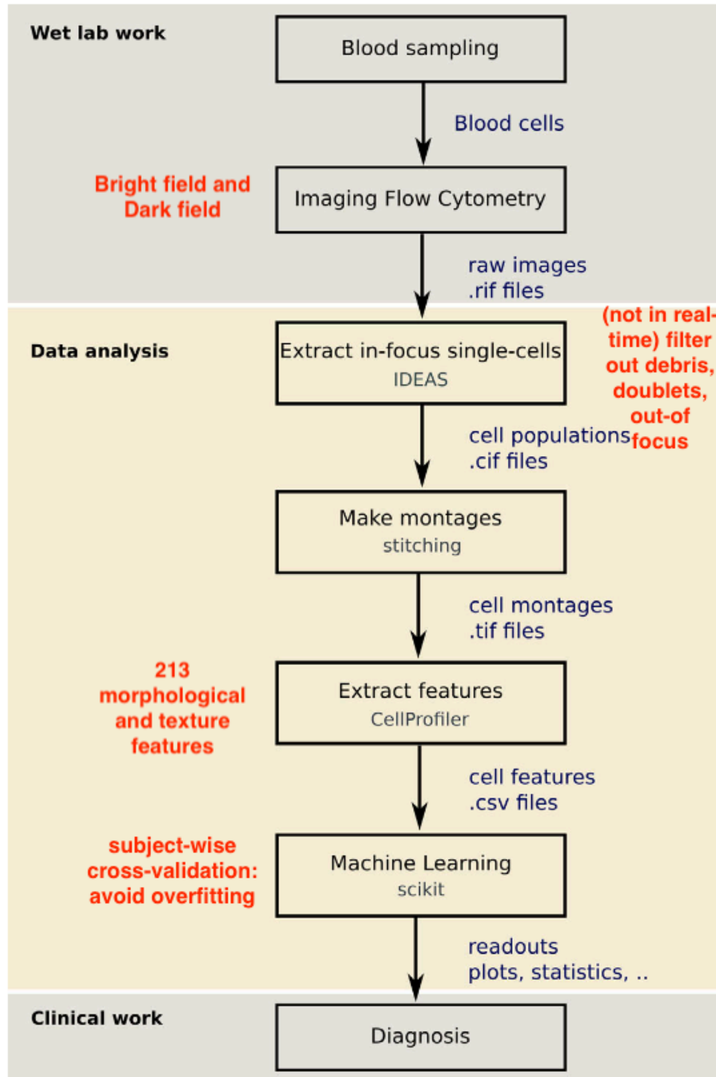
- **Validation**: 85 donors. They computed the average WBC count and compared it with the reference WBC count



- What I love:
  - Great open source dataset: over 50000 single cell images.

– meaningful morphological and texture features are used to classify, instead of using NN extracted features.
– Rigorous Machine learning training and validation.

2 Method



2.1 ML Classification

- The data set was highly imbalanced with typically (40–80%) cells in the most prominent class and only (1–6%) cells in the least prominent class. 10 random undersampling datasets were created. for each run, they determined the best classifiers. After that, they applied majority voting to determine the overall best classifier.
- What is in each run: subject-wise cross-validation: In each iteration, the data set from one blood donor was separated for validation, and the machine-learning classifier is trained on the others. Each run has 13 iterations and F1-score is computed for each iteration and averaged out.
- They used the following machine learning classifiers simultaneously in each iteration:
  - AdaBoost
  - Gradient Boosting (GB)
  - K-Nearest Neighbors (KNN)
  - Random Forest (RF)
  - Support Vector Machine (SVM)
- And the best classifier is: **Gradient Boosting with random undersampling**. They repeat the above process also using unbalanced data.

2.2 Dataset Overview

```
# folder paths and overview
PROJECT_ROOT = Path.cwd()
STAINED_FEATURES_DIR = PROJECT_ROOT / "Stained_CP_Features"
STAINED_POPULATIONS_DIR = PROJECT_ROOT / "Stained_Populations"
PATIENT_ID = "CRF022"
PATIENT_FOLDER = STAINED_POPULATIONS_DIR / PATIENT_ID
print (f"There are {len([p for p in STAINED_POPULATIONS_DIR.iterdir() if p.is_dir()])} patie
print("Each of them has the following cell types:")
print([p.name for p in PATIENT_FOLDER.iterdir() if p.is_dir()])
```

```
There are 13 patients in the Training Set.
Each of them has the following cell types:
['neutrophil', 'monocyte', 'T', 'eosinophil', 'B']
```

2.3 Read single cell images of 1 patient (BF channel)

```python
import glob
import os
import numpy as np

# Set JAVA_HOME for JPype
os.environ['JAVA_HOME'] = '/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home'
import imagej
import scyjava
import numpy as np
ij = imagej.init('sc.fiji:fiji', mode='headless')

# Suppress Bio-Formats logging messages
from scyjava import jimport
Logger = jimport('org.slf4j.Logger')
LoggerFactory = jimport('org.slf4j.LoggerFactory')
Level = jimport('ch.qos.logback.classic.Level')

# Set logging levels to suppress verbose output
try:
    # Get the root logger and set to WARN level
    root_logger = LoggerFactory.getLogger('ROOT')
    if hasattr(root_logger, 'setLevel'):
        root_logger.setLevel(Level.WARN)

    # Specifically suppress FlowSightReader messages
    flowsight_logger = LoggerFactory.getLogger('loci.formats.in.FlowSightReader')
    if hasattr(flowsight_logger, 'setLevel'):
        flowsight_logger.setLevel(Level.ERROR)

    # Suppress other Bio-Formats loggers
    bioformats_logger = LoggerFactory.getLogger('loci.formats')
    if hasattr(bioformats_logger, 'setLevel'):
        bioformats_logger.setLevel(Level.ERROR)

    print("  Bio-Formats logging suppressed")
except Exception as e:
    print(f"Note: Could not configure logging: {e}")
    print("(Verbose messages may still appear)")

# Get all CIF files
cif_files = glob.glob(f"{PATIENT_FOLDER}/*/*.cif")
# Extract intensity images and masks, keeping only pairs with non-empty masks
```

```
intensity_images = []  # List of 2D intensity images
mask_images = []       # List of corresponding binary masks
image_metadata = []    # Track cell type, filename, and series for each image pair

for file_idx, cif_file in enumerate(cif_files):

    # Get cell type from folder name
    cell_type = os.path.basename(os.path.dirname(cif_file))
    filename = os.path.basename(cif_file)

    try:
        # Use Bio-Formats reader directly for multi-series support
        ImageReader = jimport('loci.formats.ImageReader')
        reader = ImageReader()
        reader.setId(cif_file)

        # Get the number of series (individual cells) in this file
        series_count = reader.getSeriesCount()

        # Process pairs of series (intensity + mask)
        cells_added = 0
        for series_idx in range(0, series_count, 2):  # Step by 2 to process pairs
            if series_idx + 1 >= series_count:  # Make sure we have both series
                break

            # Read intensity image (series 0, 2, 4, ...)
            reader.setSeries(series_idx)
            size_x = reader.getSizeX()
            size_y = reader.getSizeY()

            # Read intensity image
            intensity_bytes = reader.openBytes(0)
            expected_pixels = size_x * size_y

            if len(intensity_bytes) == expected_pixels:
                intensity_image = np.frombuffer(intensity_bytes, dtype=np.uint8)
            elif len(intensity_bytes) == expected_pixels * 2:
                intensity_image = np.frombuffer(intensity_bytes, dtype=np.uint16)
            else:
                continue  # Skip if unexpected size

            intensity_image = intensity_image.reshape((size_y, size_x))
```

```
            # Read corresponding mask (series 1, 3, 5, ...)
            reader.setSeries(series_idx + 1)
            mask_bytes = reader.openBytes(0)

            if len(mask_bytes) == expected_pixels:
                mask_image = np.frombuffer(mask_bytes, dtype=np.uint8)
            elif len(mask_bytes) == expected_pixels * 2:
                mask_image = np.frombuffer(mask_bytes, dtype=np.uint16)
            else:
                continue  # Skip if unexpected size

            mask_image = mask_image.reshape((size_y, size_x))

            # Check if mask is empty (min = max = 0)
            if mask_image.min() == 0 and mask_image.max() == 0:
                continue  # Skip this pair - empty mask

            # Add valid pair to our collections
            intensity_images.append(intensity_image)
            mask_images.append(mask_image)

            # Store metadata
            image_metadata.append({
                'cell_type': cell_type,
                'filename': filename,
                'series_idx': series_idx,
                'intensity_shape': intensity_image.shape,
                'mask_shape': mask_image.shape,
                'intensity_min': int(intensity_image.min()),
                'intensity_max': int(intensity_image.max()),
            })

            cells_added += 1

        # Close the reader
        reader.close()
        print(f"   Extracted {cells_added} valid cells (with non-empty masks) from {filename

    except Exception as e:
        print(f"   Error processing {filename}: {e}")
        continue
```

```
print(f"\n Total valid cell pairs extracted: {len(intensity_images)}")
print(f"  Intensity images: {len(intensity_images)}")
print(f"  Mask images: {len(mask_images)}")

# Summary by cell type
from collections import Counter
cell_type_counts = Counter([meta['cell_type'] for meta in image_metadata])
print("\nValid cells per cell type:")
for cell_type, count in cell_type_counts.items():
    print(f"  {cell_type}: {count} cells")


#| fig-width: 12
#| fig-height: 4

import matplotlib.pyplot as plt

# Plot the first 5 intensity images and their corresponding masks
if len(intensity_images) >= 5:
    fig, axes = plt.subplots(2, 5, figsize=(15, 6))

    for i in range(5):
        intensity_img = intensity_images[i]
        mask_img = mask_images[i]
        metadata = image_metadata[i]

        # Display intensity image (top row)
        im1 = axes[0, i].imshow(intensity_img, cmap='gray', interpolation='nearest')
        axes[0, i].set_title(f"Intensity: {metadata['cell_type']}\n{metadata['filename'][:15
        axes[0, i].set_xticks([])
        axes[0, i].set_yticks([])

        # Add intensity info
        info_text = f"Min: {intensity_img.min()}\nMax: {intensity_img.max()}"
        axes[0, i].text(0.02, 0.98, info_text, transform=axes[0, i].transAxes,
                       verticalalignment='top', fontsize=7,
                       bbox=dict(boxstyle='round', facecolor='white', alpha=0.8))

        # Display mask image (bottom row)
        im2 = axes[1, i].imshow(mask_img, cmap='hot', interpolation='nearest')
        axes[1, i].set_title(f"Mask: Series {metadata['series_idx']}", fontsize=9)
        axes[1, i].set_xticks([])
```

```
        axes[1, i].set_yticks([])

        # Add mask info
        mask_info = f"Min: {mask_img.min()}\nMax: {mask_img.max()}"
        axes[1, i].text(0.02, 0.98, mask_info, transform=axes[1, i].transAxes,
                       verticalalignment='top', fontsize=7,
                       bbox=dict(boxstyle='round', facecolor='white', alpha=0.8))

    plt.tight_layout()
    plt.suptitle('First 5 Cell Pairs: Intensity Images (top) and Masks (bottom) in BF channel
    plt.show()

else:
    print("Not enough valid cell pairs loaded to display samples")
```

```
  Bio-Formats logging suppressed
    Extracted 813 valid cells (with non-empty masks) from CRF022_neutrophil_1.cif
    Extracted 880 valid cells (with non-empty masks) from CRF022_neutrophil_2.cif
    Extracted 346 valid cells (with non-empty masks) from CRF022_monocyte_1.cif
    Extracted 900 valid cells (with non-empty masks) from CRF022_T_1.cif
    Extracted 913 valid cells (with non-empty masks) from CRF022_T_2.cif
    Extracted 1 valid cells (with non-empty masks) from CRF022_eosinophil_2.cif
    Extracted 270 valid cells (with non-empty masks) from CRF022_eosinophil_1.cif
    Extracted 197 valid cells (with non-empty masks) from CRF022_B_1.cif
    Extracted 193 valid cells (with non-empty masks) from CRF022_B_2.cif
    Extracted 143 valid cells (with non-empty masks) from CRF022_B_3.cif

  Total valid cell pairs extracted: 4656
  Intensity images: 4656
  Mask images: 4656

Valid cells per cell type:
  neutrophil: 1693 cells
  monocyte: 346 cells
  T: 1813 cells
  eosinophil: 271 cells
  B: 533 cells
```

First 5 Cell Pairs: Intensity Images (top) and Masks (bottom) in BF channel

2.4 Features by category

```
feature_csv = "Stained_CP_Features/CRF022/B/BF_cells_on_grid.txt"
feature_df = pd.read_csv(feature_csv, sep="\t")
feature_list = feature_df.columns.tolist()[2:]
print("All features: ", feature_list)
edge_intensity_features = [f for f in feature_list if "IntensityEdge_" in f]
polarity_features = [f for f in feature_list if "MassDisplacement_" in f]
granularity_features = [f for f in feature_list if "Granularity_" in f]
intensity_features = [f for f in feature_list if "Intensity_" in f and "IntensityEdge_" not i
radial_distribution_features = [f for f in feature_list if "RadialDistribution_" in f]
gabor_texture_features = [f for f in feature_list if "Gabor_" in f]
haralick_texture_features = [f for f in feature_list if "Texture_" in f and "Gabor_" not in f
zernike_shape_features = [f for f in feature_list if "Zernike_" in f]
shape_features = [f for f in feature_list if "AreaShape_" in f and "Zernike_" not in f]
```

```
All features:  ['AreaShape_Area', 'AreaShape_Center_X', 'AreaShape_Center_Y', 'AreaShape_Com
```

2.4.1 Intensity Features

- Edge intensity: the "membrane" intensity – one-pixel rim

```
print(f"Edge intensity features (length: {len(edge_intensity_features)}): {edge_intensity_fea
print(f"Intensity features ({len(intensity_features)}): {intensity_features}")
print(f"Polarity features ({len(polarity_features)}): {polarity_features}")
```

```
Edge intensity features (length: 10): ['Intensity_IntegratedIntensityEdge_BF_image', 'Intens
Intensity features (26): ['Intensity_IntegratedIntensity_BF_image', 'Intensity_IntegratedInte
Polarity features (2): ['Intensity_MassDisplacement_BF_image', 'Intensity_MassDisplacement_DI
```

2.4.2 Granularity Features

- Granularity_N: % intensity removed when bright objects of ~N pixels in diameter are removed
  - the countour of bright objects is defined by the change in pixel values itself. The algorithm treats the image as a 3D landscape where pixel intensity is the "height." A bright region is simply a "hill" rising from the "flat ground" of the darker background. The contour is the base of that hill. **No need for a global threshold.**
- interpretation: elevated granularity at smaller scales implies higher fragmentation (Mitochondrial fragmentation)
- They used 5 different granularities: 1, 2, 3, 4, 5.

```
print(f"Granularity features ({len(granularity_features)}): {granularity_features}")
```

```
Granularity features (10): ['Granularity_1_BF_image', 'Granularity_1_DF_image', 'Granularity_
```

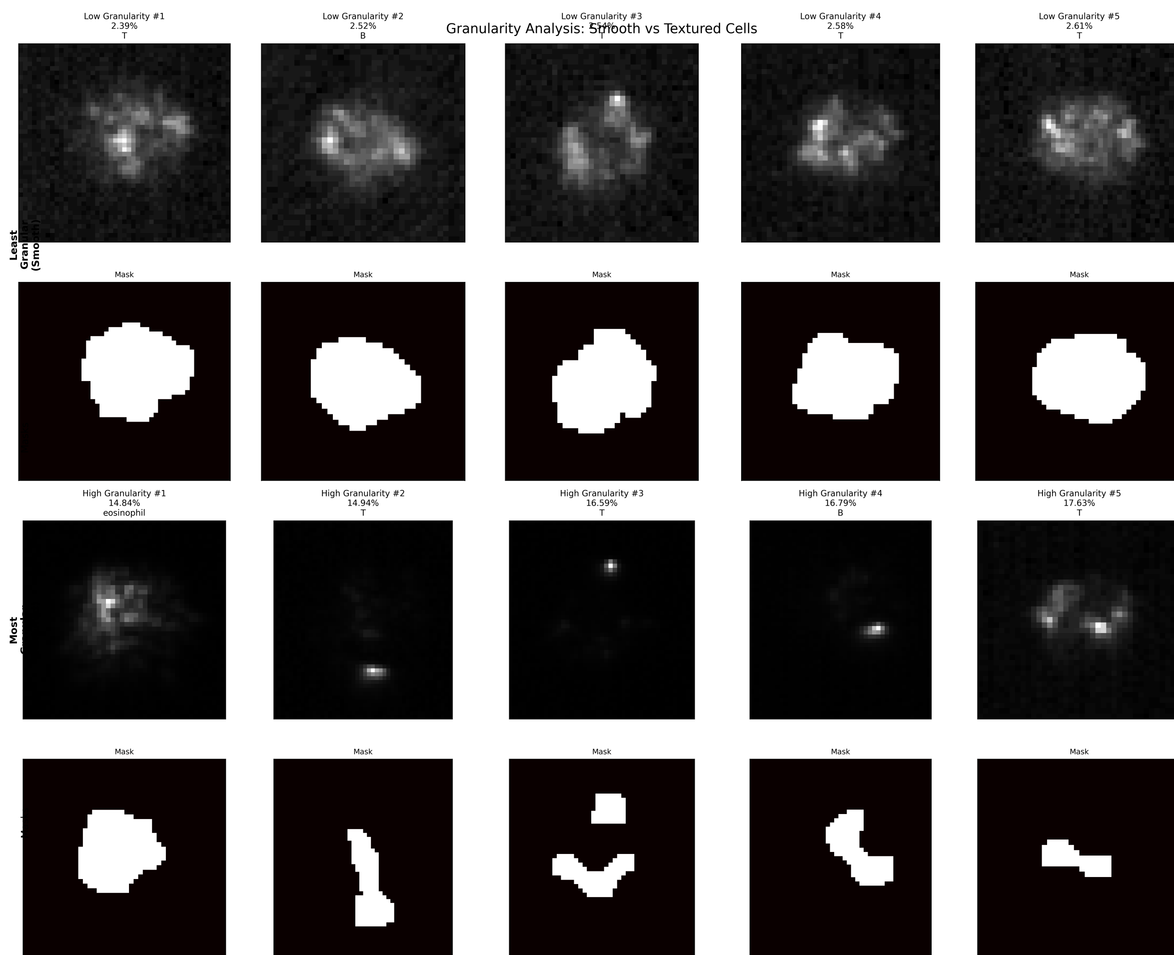
```
def granularity(image, mask, n):
    """
    Calculate the granularity of the image using morphological opening

    Parameters:
    - image: 2D numpy array (intensity image)
    - mask: 2D numpy array (binary mask)
    - n: int (size of structuring element)

    Returns:
    - granularity value: percentage of intensity removed by opening
    """
    from scipy import ndimage
    from skimage import morphology
    import numpy as np

    # Apply mask to get only the cell region
    masked_image = image * (mask > 0)

    # Create circular structuring element of size n
```

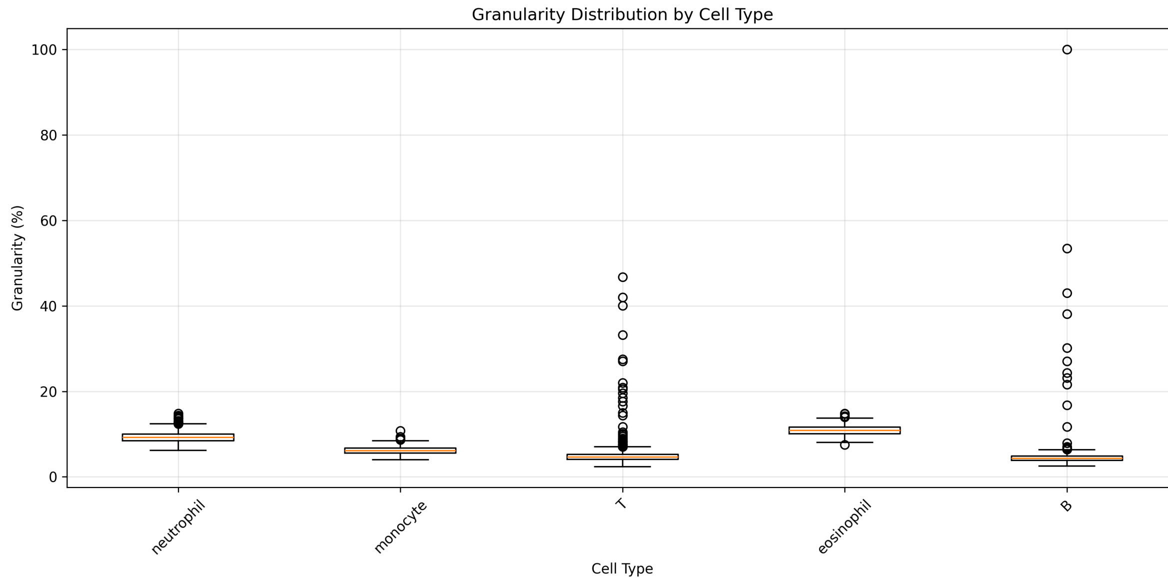
```
    if n == 1:
        # For size 1, use a simple 3x3 cross
        selem = np.array([[0, 1, 0],
                          [1, 1, 1],
                          [0, 1, 0]], dtype=bool)
    else:
        # For larger sizes, use disk structuring element
        selem = morphology.disk(n)

    # Apply morphological opening (erosion followed by dilation) to remove bright objects of
    opened_image = morphology.opening(masked_image, selem)

    # Calculate intensity difference
    intensity_removed = masked_image - opened_image

    # Calculate total intensity in masked region
    total_intensity = np.sum(masked_image)

    # Avoid division by zero
    if total_intensity == 0:
        return 0.0

    # Calculate percentage of intensity removed
    granularity_value = (np.sum(intensity_removed) / total_intensity) * 100

    return granularity_value

# Computer granularity for each image
print("Computing granularity for all images...")
granularity_values = []
for i in range(len(intensity_images)):
    granularity_values.append(granularity(intensity_images[i], mask_images[i], 1))

print(f"Granularity computed for {len(granularity_values)} images")
print(f"Granularity range: {min(granularity_values):.2f}% to {max(granularity_values):.2f}%"

# Find indices of top 5 and bottom 5 granularity values
import numpy as np
granularity_array = np.array(granularity_values)
sorted_indices = np.argsort(granularity_array)

# Get bottom 5 (least granular) and top 5 (most granular)
```

```
bottom_5_indices = sorted_indices[:5]
top_5_indices = sorted_indices[-25:-20]

print(f"\nBottom 5 granularity values: {granularity_array[bottom_5_indices]}")
print(f"Top 5 granularity values: {granularity_array[top_5_indices]}")

# Plot the top 5 and bottom 5 granularity images
fig, axes = plt.subplots(4, 5, figsize=(20, 16))

# Plot bottom 5 (least granular)
for i, idx in enumerate(bottom_5_indices):
    # Intensity image
    axes[0, i].imshow(intensity_images[idx], cmap='gray', interpolation='nearest')
    axes[0, i].set_title(f"Low Granularity #{i+1}\n{granularity_values[idx]:.2f}%\n{image_me
    axes[0, i].set_xticks([])
    axes[0, i].set_yticks([])

    # Mask
    axes[1, i].imshow(mask_images[idx], cmap='hot', interpolation='nearest')
    axes[1, i].set_title(f"Mask", fontsize=9)
    axes[1, i].set_xticks([])
    axes[1, i].set_yticks([])

# Plot top 5 (most granular)
for i, idx in enumerate(top_5_indices):
    # Intensity image
    axes[2, i].imshow(intensity_images[idx], cmap='gray', interpolation='nearest')
    axes[2, i].set_title(f"High Granularity #{i+1}\n{granularity_values[idx]:.2f}%\n{image_me
    axes[2, i].set_xticks([])
    axes[2, i].set_yticks([])

    # Mask
    axes[3, i].imshow(mask_images[idx], cmap='hot', interpolation='nearest')
    axes[3, i].set_title(f"Mask", fontsize=9)
    axes[3, i].set_xticks([])
    axes[3, i].set_yticks([])

# Add row labels
fig.text(0.02, 0.75, 'Least\nGranular\n(Smooth)', rotation=90, fontsize=12, ha='center', va=
fig.text(0.02, 0.55, 'Masks', rotation=90, fontsize=12, ha='center', va='center')
fig.text(0.02, 0.35, 'Most\nGranular\n(Textured)', rotation=90, fontsize=12, ha='center', va=
fig.text(0.02, 0.15, 'Masks', rotation=90, fontsize=12, ha='center', va='center')
```

```
plt.tight_layout()
plt.suptitle('Granularity Analysis: Smooth vs Textured Cells', fontsize=16, y=0.98)
plt.show()

# Show granularity distribution by cell type
import matplotlib.pyplot as plt
from collections import defaultdict

cell_type_granularity = defaultdict(list)
for i, meta in enumerate(image_metadata):
    cell_type_granularity[meta['cell_type']].append(granularity_values[i])

# Box plot of granularity by cell type
plt.figure(figsize=(12, 6))
cell_types = list(cell_type_granularity.keys())
granularity_by_type = [cell_type_granularity[ct] for ct in cell_types]

plt.boxplot(granularity_by_type, labels=cell_types)
plt.ylabel('Granularity (%)')
plt.xlabel('Cell Type')
plt.title('Granularity Distribution by Cell Type')
plt.xticks(rotation=45)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
Computing granularity for all images...
Granularity computed for 4656 images
Granularity range: 2.39% to 100.00%

Bottom 5 granularity values: [2.39103362 2.51593056 2.53533569 2.57548845 2.61414882]
Top 5 granularity values: [14.83766795 14.94359164 16.58677761 16.78799418 17.63140386]
```


Granularity Analysis: Smooth vs Textured Cells
Low Granularity #1
2.39%
T
Low Granularity #2
2.52%
B
Low Granularity #3
T
Low Granularity #4
2.58%
T
Low Granularity #5
2.61%
T
Least
Mask
Mask
Mask
Mask
Mask
High Granularity #1
14.84%
eosinophil
High Granularity #2
14.94%
T
High Granularity #3
16.59%
T
High Granularity #4
16.79%
B
High Granularity #5
17.63%
T
Most
Mask
Mask
Mask
Mask
Mask



2.4.3 Radial Distribution Features

- partition each object into four concentric rings and measure, respectively, the fraction of total intensity in a ring, that fraction normalised by area, and the coefficient of variation within the ring—capturing nucleus-to-cytoplasm or centre-to-edge gradients

```
print(f"Radial distribution features ({len(radial_distribution_features)}): {radial_distribu
```

```
Radial distribution features (24): ['RadialDistribution_FracAtD_BF_image_1of4', 'RadialDistri
```

```
# write me a function to calculate the radial distribution MeanFraction of ring 4 (the outerm
def radial_distribution(image, mask, ring_number=4):
    """
    Calculate the radial distribution mean fraction for a specific ring

    Parameters:
    - image: 2D numpy array (intensity image)
    - mask: 2D numpy array (binary mask)
    - ring_number: int (1-4, where 1 is innermost, 4 is outermost)

    Returns:
    - mean_fraction: fraction of total intensity in the specified ring
    """
```

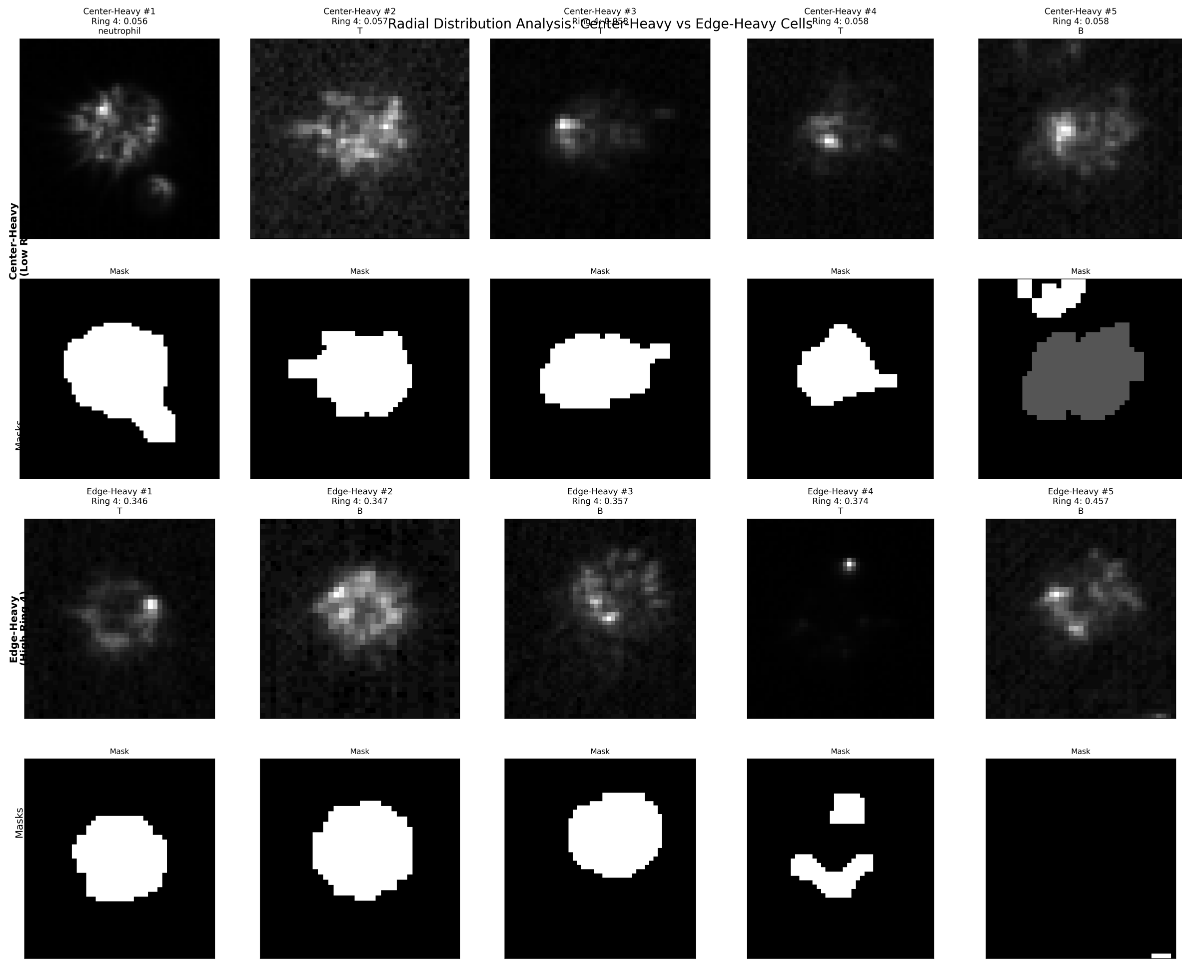
```
import numpy as np
from scipy import ndimage

# Apply mask to get only the cell region
masked_image = image * (mask > 0)

# Find the centroid of the mask
y_coords, x_coords = np.where(mask > 0)
if len(y_coords) == 0:
    return 0.0

centroid_y = np.mean(y_coords)
centroid_x = np.mean(x_coords)

# Create coordinate arrays
y_indices, x_indices = np.indices(mask.shape)

# Calculate distance from centroid to each pixel
distances = np.sqrt((y_indices - centroid_y)**2 + (x_indices - centroid_x)**2)

# Only consider distances within the mask
mask_distances = distances[mask > 0]
max_distance = np.max(mask_distances)

# Divide into 4 equal rings based on distance
ring_thickness = max_distance / 4.0

# Define ring boundaries
ring_min = (ring_number - 1) * ring_thickness
ring_max = ring_number * ring_thickness

# Create ring mask
ring_mask = (distances >= ring_min) & (distances < ring_max) & (mask > 0)

# For the outermost ring (ring 4), include the maximum distance
if ring_number == 4:
    ring_mask = (distances >= ring_min) & (distances <= ring_max) & (mask > 0)

# Calculate intensity in this ring
ring_intensity = np.sum(masked_image[ring_mask])

# Calculate total intensity in the entire cell
```

```
    total_intensity = np.sum(masked_image)

    # Avoid division by zero
    if total_intensity == 0:
        return 0.0

    # Calculate fraction of total intensity in this ring
    mean_fraction = ring_intensity / total_intensity

    return mean_fraction

# compute the radial distribution for each image
print("Computing radial distribution (ring 4) for all images...")
radial_distribution_values = []
for i in range(len(intensity_images)):
    radial_distribution_values.append(radial_distribution(intensity_images[i], mask_images[i

print(f"Radial distribution computed for {len(radial_distribution_values)} images")
print(f"Ring 4 fraction range: {min(radial_distribution_values):.3f} to {max(radial_distribu

# Find indices of top 5 and bottom 5 radial distribution values
radial_array = np.array(radial_distribution_values)
sorted_indices = np.argsort(radial_array)

# Get bottom 5 (center-heavy) and top 5 (edge-heavy)
bottom_5_indices = sorted_indices[35:40]
top_5_indices = sorted_indices[-5:]

print(f"\nBottom 5 ring 4 fractions (center-heavy): {radial_array[bottom_5_indices]}")
print(f"Top 5 ring 4 fractions (edge-heavy): {radial_array[top_5_indices]}")

# Plot the top 5 and bottom 5 radial distribution images
fig, axes = plt.subplots(4, 5, figsize=(20, 16))

# Plot bottom 5 (center-heavy cells)
for i, idx in enumerate(bottom_5_indices):
    # Intensity image
    axes[0, i].imshow(intensity_images[idx], cmap='gray', interpolation='nearest')
    axes[0, i].set_title(f"Center-Heavy #{i+1}\nRing 4: {radial_distribution_values[idx]:.3f
    axes[0, i].set_xticks([])
    axes[0, i].set_yticks([])
```

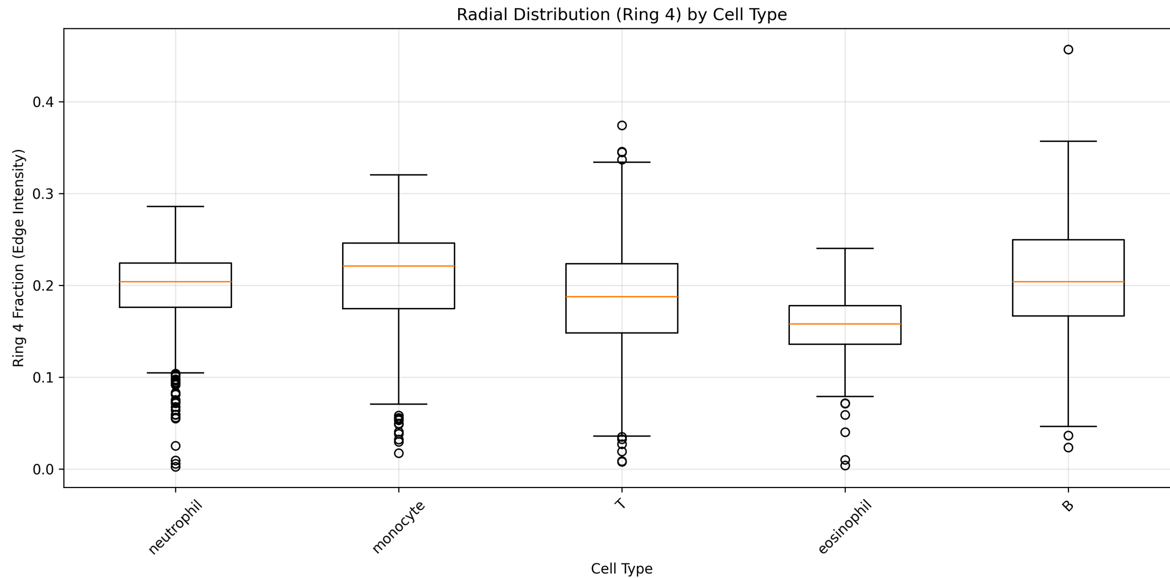
```
    # Mask
    axes[1, i].imshow(mask_images[idx], cmap='gray', interpolation='nearest')
    axes[1, i].set_title(f"Mask", fontsize=9)
    axes[1, i].set_xticks([])
    axes[1, i].set_yticks([])

# Plot top 5 (edge-heavy cells)
for i, idx in enumerate(top_5_indices):
    # Intensity image
    axes[2, i].imshow(intensity_images[idx], cmap='gray', interpolation='nearest')
    axes[2, i].set_title(f"Edge-Heavy #{i+1}\nRing 4: {radial_distribution_values[idx]:.3f}\n
    axes[2, i].set_xticks([])
    axes[2, i].set_yticks([])

    # Mask
    axes[3, i].imshow(mask_images[idx], cmap='gray', interpolation='nearest')
    axes[3, i].set_title(f"Mask", fontsize=9)
    axes[3, i].set_xticks([])
    axes[3, i].set_yticks([])

# Add row labels
fig.text(0.02, 0.75, 'Center-Heavy\n(Low Ring 4)', rotation=90, fontsize=12, ha='center', va=
fig.text(0.02, 0.55, 'Masks', rotation=90, fontsize=12, ha='center', va='center')
fig.text(0.02, 0.35, 'Edge-Heavy\n(High Ring 4)', rotation=90, fontsize=12, ha='center', va=
fig.text(0.02, 0.15, 'Masks', rotation=90, fontsize=12, ha='center', va='center')

plt.tight_layout()
plt.suptitle('Radial Distribution Analysis: Center-Heavy vs Edge-Heavy Cells', fontsize=16, y
plt.show()

# Show radial distribution by cell type
from collections import defaultdict

cell_type_radial = defaultdict(list)
for i, meta in enumerate(image_metadata):
    cell_type_radial[meta['cell_type']].append(radial_distribution_values[i])

# Box plot of radial distribution by cell type
plt.figure(figsize=(12, 6))
cell_types = list(cell_type_radial.keys())
radial_by_type = [cell_type_radial[ct] for ct in cell_types]
```

```
plt.boxplot(radial_by_type, labels=cell_types)
plt.ylabel('Ring 4 Fraction (Edge Intensity)')
plt.xlabel('Cell Type')
plt.title('Radial Distribution (Ring 4) by Cell Type')
plt.xticks(rotation=45)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
Computing radial distribution (ring 4) for all images...
Radial distribution computed for 4656 images
Ring 4 fraction range: 0.003 to 0.457

Bottom 5 ring 4 fractions (center-heavy): [0.05600573 0.05707306 0.05765143 0.05779556 0.0582
Top 5 ring 4 fractions (edge-heavy): [0.34576822 0.34674401 0.35668189 0.37415212 0.45695364]
```



Radial Distribution Analysis: Center-Heavy vs Edge-Heavy Cells
Center-Heavy #1
Ring 4: 0.056
neutrophil
Center-Heavy #2
Ring 4: 0.057
T
Center-Heavy #3
T
Center-Heavy #4
Ring 4: 0.058
T
Center-Heavy #5
Ring 4: 0.058
B
Center-Heavy
Mask
Mask
Mask
Mask
Mask
Edge-Heavy #1
Ring 4: 0.346
T
Edge-Heavy #2
Ring 4: 0.347
B
Edge-Heavy #3
Ring 4: 0.357
B
Edge-Heavy #4
Ring 4: 0.374
T
Edge-Heavy #5
Ring 4: 0.457
B
Edge-Heavy
Masks
Mask
Mask
Mask
Mask
Mask



2.4.4 Gabor Texture Features

```
print(f"Gabor texture features ({len(gabor_texture_features)}): {gabor_texture_features}")
```

```
Gabor texture features (2): ['Texture_Gabor_BF_image_3', 'Texture_Gabor_DF_image_3']
```

2.4.5 Haralick Texture Features

2.4.5.1 Example: `Texture_AngularSecondMoment_3_45`

1. Quantize the intensities into 8 levels
2. Compute the gray-level co-occurrence matrix using the direction: 45 degree (0, 45, 90, 135) and 3 pixels apart (3 or 5) that defines the "neighborhood".

   - rows of the matrix are the intensity levels of the first pixel (reference pixel)
   - columns of the matrix are the intensity levels of the second pixel (neighbor pixel)
   - the value at the (i, j) position C(i, j) is the number of times the pair (i, j) occurs in the image
   - $P(i,j) = \frac{C(i,j)}{\sum_i \sum_j C(i,j)}$ is the normalized co-occurrence matrix

3. Compute the Haralick texture features based on the co-occurrence matrix: AngularSecondMoment, Contrast, Correlation, Entropy, DifferenceVariance, SumAverage, InfoMeas1/2, InverseDifferenceMoment

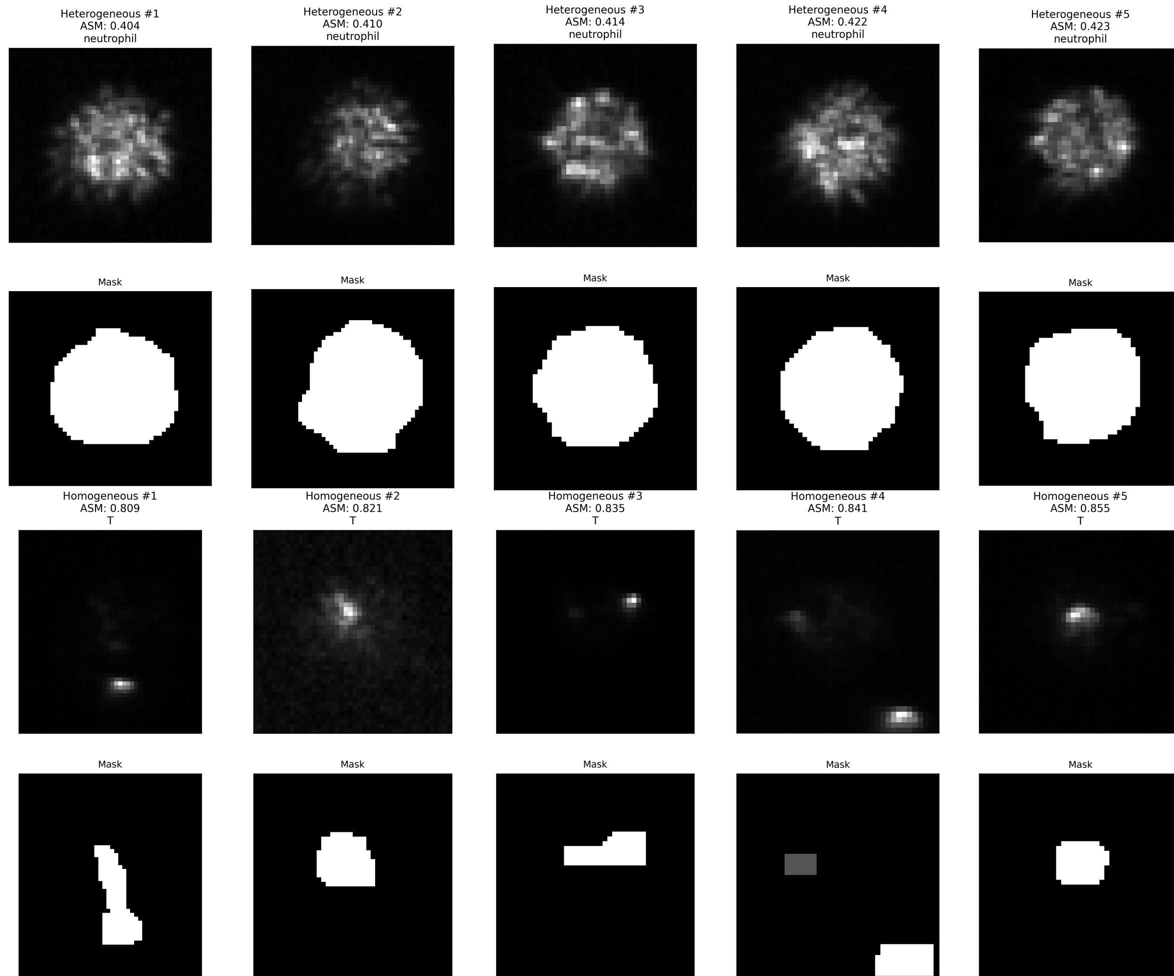
- $\text{ASM} = \sum_i \sum_j P(i,j)^2$
- $\text{Contrast} = \sum_i \sum_j |i-j|^2 P(i,j)$
- $\text{Correlation} = \frac{\sum_i \sum_j (i-\mu_i)(j-\mu_j)P(i,j)}{\sigma_i \sigma_j}$
- $\text{Entropy} = -\sum_i \sum_j P(i,j) \log P(i,j)$
- $\text{DifferenceVariance} = \sum_i \sum_j (i-j)^2 P(i,j)$
- $\text{SumAverage} = \sum_i \sum_j (i+j) P(i,j)$

```
def haralick_texture(image, mask, direction, distance, metric="ASM"):
    """
    Calculate Haralick texture features for a given image and mask.

    Parameters:
    - image: 2D numpy array (intensity image)
    - mask: 2D numpy array (binary mask)
    - direction: int (0, 45, 90, or 135 degrees)
    - distance: int (pixel distance for co-occurrence)
    - metric: str (e.g., "ASM", "Contrast", "Correlation")

    Returns:
    - feature_value: calculated Haralick texture feature
    """
    try:
        import numpy as np
        from skimage.feature import graycomatrix, graycoprops
    except ImportError:
        print("Please install scikit-image: uv add scikit-image")
        return None

    # 1. Create a blank image and get pixels from within the mask
    image_masked = np.zeros_like(image, dtype=np.uint8)
    pixels_in_mask = image[mask > 0]

    if len(pixels_in_mask) == 0:
        return 0.0

    # Handle flat images where texture is undefined
    min_val, max_val = np.min(pixels_in_mask), np.max(pixels_in_mask)
    if min_val == max_val:
        return 0.0

    # 2. Quantize the pixels to 8 levels (0-7) and place them in the blank image.
```

```python
    # This leaves the background as 0, but since we ignore it in glcm, it's fine.
    bins = np.linspace(start=min_val, stop=max_val, num=8) # 8 bins

    # Digitize to get levels 0-7, then shift to 1-8 for the masked image
    quantized_pixels = np.digitize(pixels_in_mask, bins, right=True)

    # Ensure values are within the 1-8 range for the GLCM levels
    quantized_pixels[quantized_pixels > 8] = 8
    quantized_pixels[quantized_pixels < 1] = 1

    image_masked[mask > 0] = quantized_pixels

    # 3. Map direction from degrees to radians
    angle_rad = {0: 0, 45: np.pi/4, 90: np.pi/2, 135: 3*np.pi/4}.get(direction)
    if angle_rad is None:
        raise ValueError("Direction must be one of 0, 45, 90, 135")

    # 4. Compute Gray-Level Co-occurrence Matrix (GLCM)
    # levels=9 to account for background (0) and 8 quantized levels (1-8)
    glcm = graycomatrix(
        image_masked,
        distances=[distance],
        angles=[angle_rad],
        levels=9,
        symmetric=True,
        normed=True
    )

    # 5. Map metric name and compute the property
    prop_map = {
        "ASM": "ASM",
        "AngularSecondMoment": "ASM",
        "Contrast": "contrast",
        "Correlation": "correlation",
        "Homogeneity": "homogeneity", # Inverse Difference Moment
        "Energy": "energy" # sqrt(ASM)
    }

    prop = prop_map.get(metric)
    if prop is None:
        raise ValueError(f"Metric '{metric}' not supported. Choose from {list(prop_map.keys()
```

```
    feature_value = graycoprops(glcm, prop)[0, 0]
    return feature_value

# compute the haralick texture for each image
print("Computing haralick texture for all images...")
haralick_texture_values = []
for i in range(len(intensity_images)):
    haralick_texture_values.append(haralick_texture(intensity_images[i], mask_images[i], dire
# plot the top 5 and bottom 5 haralick texture images as well as the masks
# Plot top 5 and bottom 5 haralick texture images
sorted_indices = np.argsort(haralick_texture_values)
bottom_5_indices = sorted_indices[:5]
top_5_indices = sorted_indices[-25:-20]

fig, axes = plt.subplots(4, 5, figsize=(20, 16))

# Plot bottom 5 (heterogeneous)
for i, idx in enumerate(bottom_5_indices):
    # Intensity image
    axes[0, i].imshow(intensity_images[idx], cmap='gray', interpolation='nearest')
    axes[0, i].set_title(f"Heterogeneous #{i+1}\nASM: {haralick_texture_values[idx]:.3f}\n{in
    axes[0, i].set_xticks([])
    axes[0, i].set_yticks([])
    # Mask
    axes[1, i].imshow(mask_images[idx], cmap='gray', interpolation='nearest')
    axes[1, i].set_title(f"Mask", fontsize=9)
    axes[1, i].set_xticks([])
    axes[1, i].set_yticks([])
# Plot top 5 (homogeneous)
for i, idx in enumerate(top_5_indices):
    # Intensity image
    axes[2, i].imshow(intensity_images[idx], cmap='gray', interpolation='nearest')
    axes[2, i].set_title(f"Homogeneous #{i+1}\nASM: {haralick_texture_values[idx]:.3f}\n{imag
    axes[2, i].set_xticks([])
    axes[2, i].set_yticks([])
    # Mask
    axes[3, i].imshow(mask_images[idx], cmap='gray', interpolation='nearest')
    axes[3, i].set_title(f"Mask", fontsize=9)
    axes[3, i].set_xticks([])
    axes[3, i].set_yticks([])
```

```
Computing haralick texture for all images...
```




```
print(f"Haralick texture features ({len(haralick_texture_features)}): {haralick_texture_featu
```

```
Haralick texture features (104): ['Texture_AngularSecondMoment_BF_image_3_0', 'Texture_Angula
```

2.4.6 Zernike Shape Features

```
print(f"Zernike shape features ({len(zernike_shape_features)}): {zernike_shape_features}")
```

```
Zernike shape features (30): ['AreaShape_Zernike_0_0', 'AreaShape_Zernike_1_1', 'AreaShape_Ze
```

2.4.7 Shape Features

```
print(f"Shape features ({len(shape_features)}): {shape_features}")
```

```
Shape features (18): ['AreaShape_Area', 'AreaShape_Center_X', 'AreaShape_Center_Y', 'AreaSha
```

2.5 Top 10 Features from classification

FEATURE	CHANNEL
MAD intensity	Darkfield
Std intensity	Darkfield
Integrated intensity	Darkfield
Lower quartile intensity	Brightfield
Granularity 1	Darkfield
Mean Intensity	Brightfield
Upper quartile intensity	Darkfield
Granularity 1	Brightfield
Std intensity edge	Brightfield
Integrated intensity edge	Darkfield

FEATURE	CHANNEL
Std intensity edge	Brightfield
Lower quartile intensity	Brightfield
MeanFrac Radial Distribution 4of4	Brightfield
Mean intensity	Brightfield
Integrated intensity edge	Darkfield
Granularity 1	Brightfield
FracAtD Radial Distribution 4of4	Brightfield
Granularity 1	Darkfield
DifferenceVariance Texture 3_0	Brightfield
Granularity 3	Brightfield

- MAD intensity: $median(|x_i - median(x)|)$ – deviation from the median - Integrated intensity: $sum(x_i)$ – sum of the intensity values