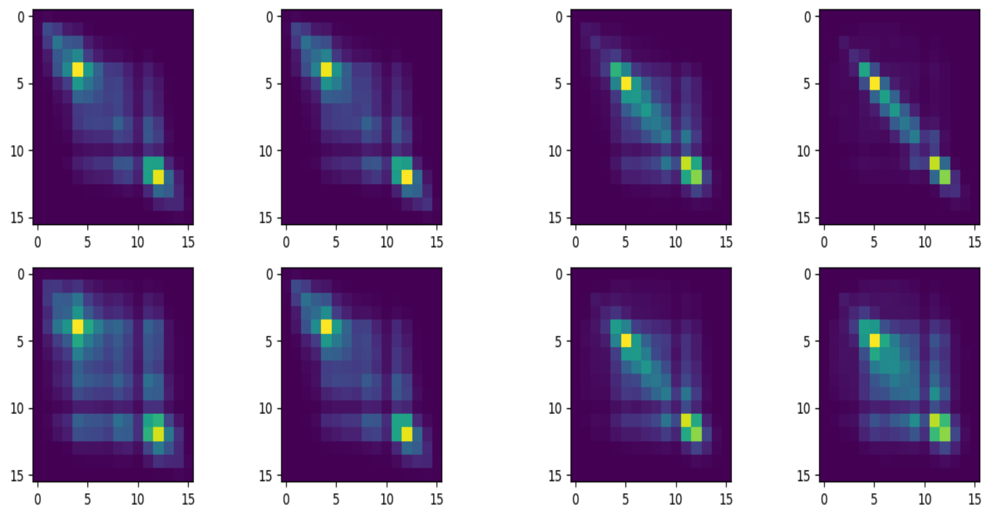
IN5520 mandatory 2

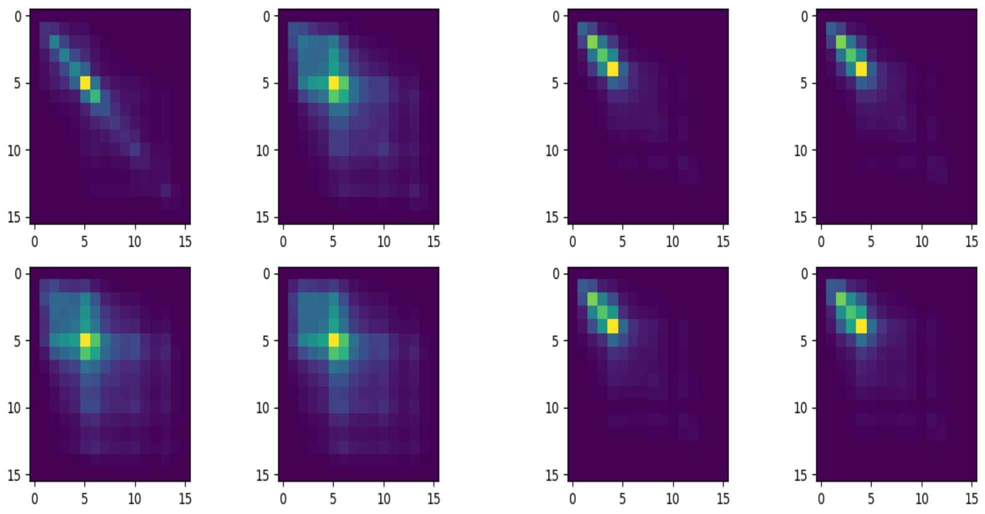
Erlend Kristensen

19.10.2022

Choosing GLCM

The training data used has 4 different textures, where we have a calculated GLCM direction for each of those textures. We analyze these GLCMs (shown bellow) to figure out which direction is most suitable for our textures.





The images are all the GLCM images for texture 1-4 in that order. When analyzing we see that in texture 4, every GLCM picture is the same no matter which direction we choose. Looking at all the textures in combination, we see that the top right GLCM image is very different in each texture, so we choose this direction to work with, which is Angle 0, $\Delta x = 1, \Delta y = 0$. The other options will result in texture 1 and 2 being too similar (see top two pictures).

Subdividing the GLCM matrices

We then want to subdivide our GLCM matrices to get new features (which can and will be used for classification later on). We do this by dividing each GLCM, which is 16×16, into 4 new 8×8. Then, for each of these new matrices, we get the feature as:

$$FQ_1 = \frac{\sum_{i=1}^{8} \sum_{j=1}^{8} P(i,j)}{\sum_{i=1}^{G} \sum_{j=1}^{G} P(i,j)}$$

where the rest of the features are calculated the same, just using the indices to the corresponding part of the GLCM.

Since $\sum_{i=1}^{G}\sum_{j=1}^{G} P(i,j) = 1$, we don't need to calculate this.

When we analyzed the GLCM matrices, we saw that each one was very unique/different from the others. Therefore, i think that based simply of visuals, that 4 quadrants is enough, and that we don't need any further subdivision of the quadrants.

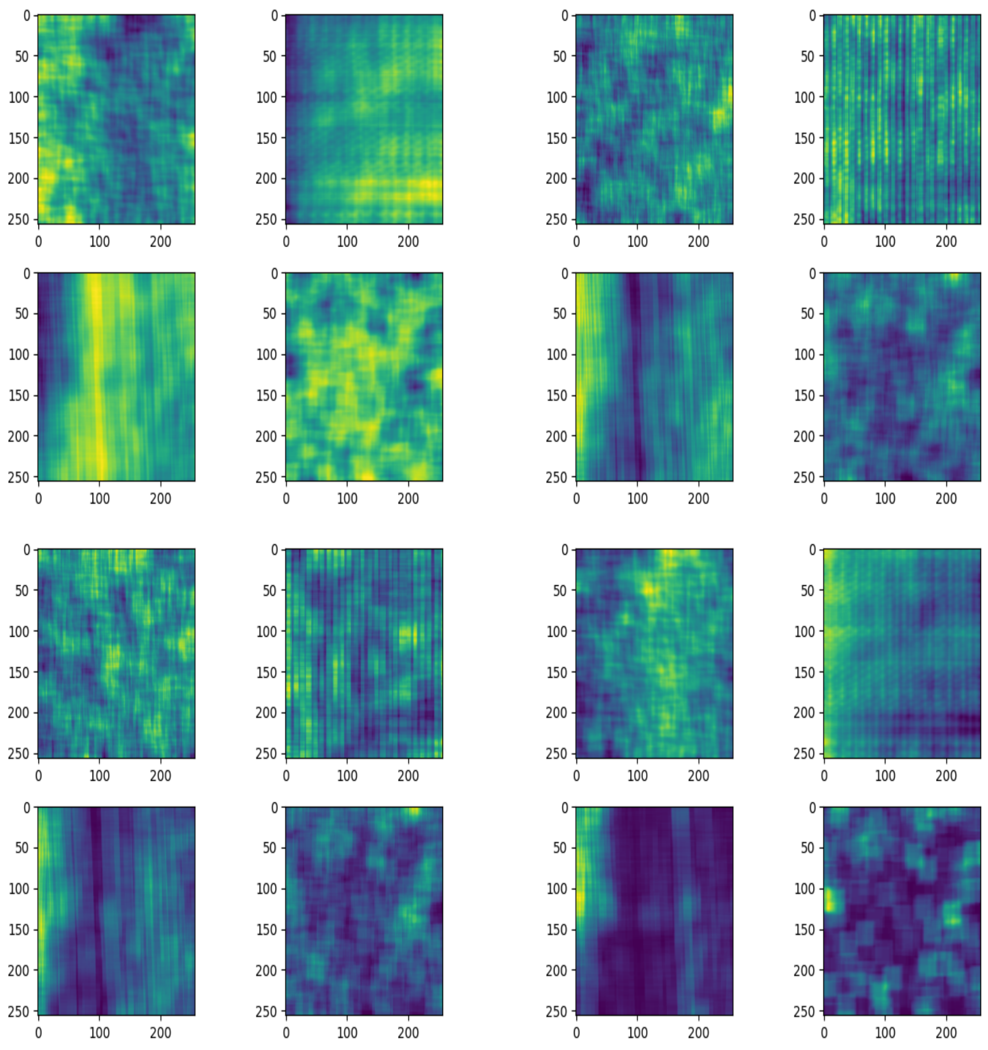
When testing the feature values for the different GLCM matrices, we got the output:

```
[0.44053309  0.09730392  0.06081495  0.31779259]
[0.42009804  0.07071844  0.03501072  0.37784161]
[0.64474571  0.11894914  0.08148744  0.0932598 ]
[0.86672794  0.03900123  0.0178845   0.03975184]
```

As we can see, the features are able to seperate well between the textures, but 1 and 2 are a bit alike. However, i think that as we see in feature 2 and 3 on these textures, it seems the difference is large enough to be able to seperate between them. Therefore i don't think any further subdividing of the features is needed. Also, since some of the quadrants are very similar, i think we need all four to be able to seperate between the textures. However, if we were to pick away one of them, it would be number 2 since the values for each texture are very similar, but i still mean we should keep all 4, and will do that further on.

Testing the features

We then test the features by gliding a 31×31 window across and extracting the features using 16×16 GLCMs for every pixel. We do this on every texture and get the following feature images:



Looking at the feature images above, there is no clear indicaion that we can use a few of them and not all, therefore i choose to use all features for our classification.

Multivariate Gaussian classifier

We then want to implement our multivariate Gaussian classifier. This classifier works by calculating a probability for each class given the features we have, and then selecting the class which has the highest probability. The functions used for this classifier is:

$$p(\omega_k|\mathbf{x}) = p(\mathbf{x}|\omega_k)p(\omega_k) = \frac{1}{(2*\pi)^{\frac{1}{2}}|\Sigma_k|^{\frac{1}{2}}} * exp\left[-\frac{1}{2}(\mathbf{x}-\hat{\mu}_{\mathbf{k}})^T \hat{\Sigma_k^{-1}}(\mathbf{x}-\hat{\mu}_{\mathbf{k}})\right] \tag{1}$$

Which we do for all x (features) in the given image we want to classify. Here ω_k represents the k th class, and we also have:

$$\hat{\mu}_{\mathbf{k}} = \frac{1}{M_k}\sum_{m=1}^{M_k}\mathbf{x}_m$$
$$\hat{\Sigma}_k = \frac{1}{M_k}\sum_{m=1}^{M_k}(\mathbf{x}_m-\hat{\mu}_{\mathbf{k}})^T(\mathbf{x}_m-\hat{\mu}_{\mathbf{k}})$$

(Note: in the lecture notes it sais $\hat{\Sigma}_k = \frac{1}{M_k}\sum_{m=1}^{M_k}(\mathbf{x}_m-\hat{\mu}_{\mathbf{k}})(\mathbf{x}_m-\hat{\mu}_{\mathbf{k}})^T$, but this order of vector multiplication will result in a value and not a matrix. Therefore we have to swap the order and do it the way i wrote.)

The code for the classifier:

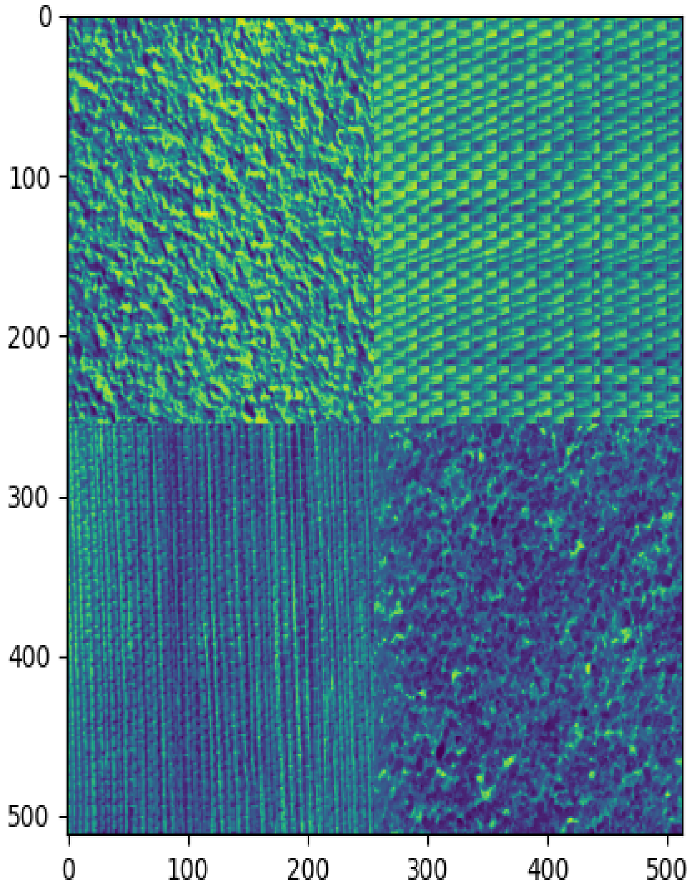
```
class GausianClassifier:

    def fit(self, mask, features):
        self.features = features
        self.mask = mask
        num_features = len(features)
        num_classes = len(np.unique(mask)) - 1
        num_pixels = np.zeros(num_classes)

        mu = np.zeros((num_classes, num_features))
        mu = np.matrix(mu)
```

```
self.num_classes = num_classes
self.num_features = num_features

#Calculating mu
for i in range(mask.shape[0]):
    for j in range(mask.shape[1]):
        correct_class = int(mask[i,j])
        if correct_class != 0:
            num_pixels[correct_class - 1] += 1
            mu[correct_class - 1] += features[:,i,j]

#normalizing mu
for i in range(num_classes):
    mu[i] /= num_pixels[i]

self.mu = mu

#Making sigma
sigma = np.zeros((num_classes, num_features,
num_features))
for i in range(mask.shape[0]):
    for j in range(mask.shape[1]):
        correct_class = int(mask[i,j])
        if correct_class != 0:
            sigma[correct_class - 1] +=
            (features[:,i,j] - mu[correct_class -
            1]).T*(features[:,i,j] -
            mu[correct_class - 1])

#normalizing sigma
for i in range(num_classes):
    sigma[i] /= num_pixels[i]
```

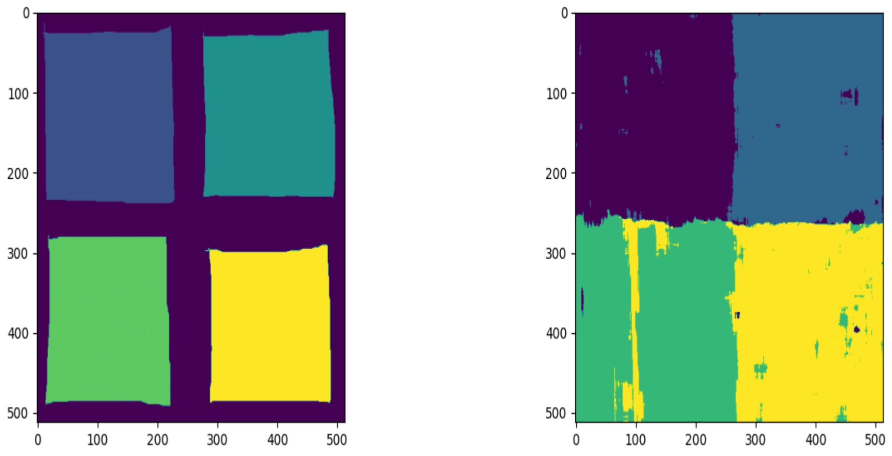
```
        self.sigma = sigma

        self.num_pixels = num_pixels



    def predict(self):
        mask = self.mask
        pred_matrix = np.zeros_like(self.mask)
        prob_arr = np.zeros(self.num_classes)

        #Probability for each class
        p_w = np.ones(self.num_classes)
        p_w *= self.num_pixels/np.sum(self.num_pixels)

        for i in range(mask.shape[0]):
            for j in range(mask.shape[1]):
                x = self.features[:,i,j]
                for k in range(self.num_classes):

                    first_part = 1/(np.sqrt(2*np.pi) *
                    np.sqrt(np.linalg.det(self.sigma[k])))

                    second_part = np.exp(-0.5 * (x -
                    self.mu[k]) @
                    np.linalg.pinv(self.sigma[k]) @ (x -
                    self.mu[k]).T)

                    prob_arr[k]=first_part*second_part*p_w[k]
                pred_matrix[i,j] = np.argmax(prob_arr) + 1
        return pred_matrix
```

Training with our multivariate Gaussian classifier

We then train a mask on our classifier, and calculate the resulting confusion matrix for the given image bellow:



Here is the mask and the resulting classification image:

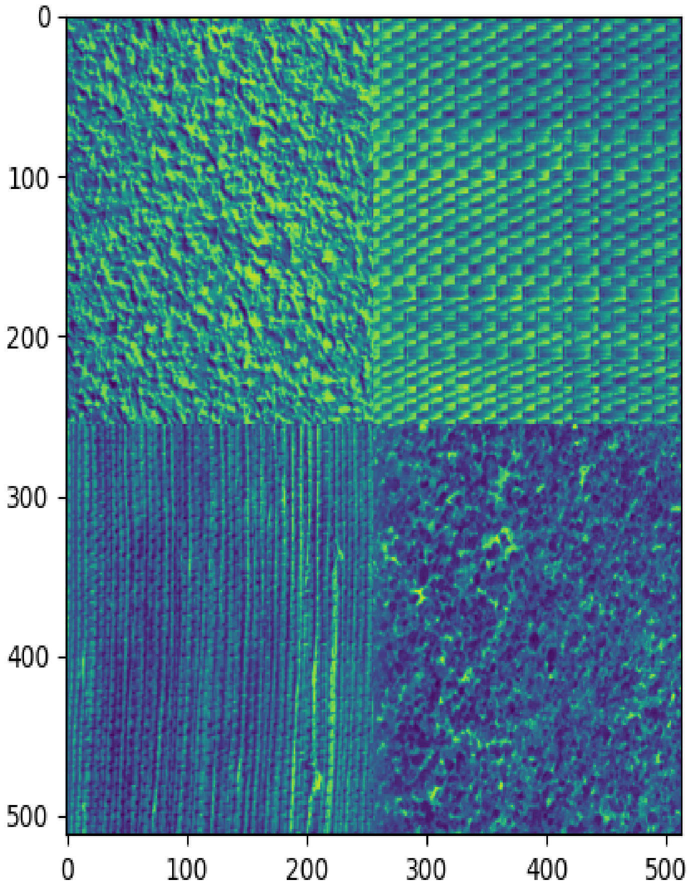


with the resulting confusion matrix:

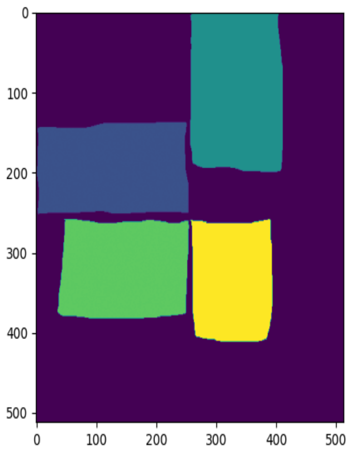
```
[[9.90554810e+01 7.91931152e-01 1.46484375e-01 6.10351562e-03]
 [4.10156250e+00 9.58984375e+01 0.00000000e+00 0.00000000e+00]
 [2.08282471e+00 1.69372559e-01 8.99658203e+01 7.78198242e+00]
 [1.02081299e+00 2.51922607e+00 6.10656738e+00 9.03533936e+01]]
 overall accuracy = 0.9381828308105469
```

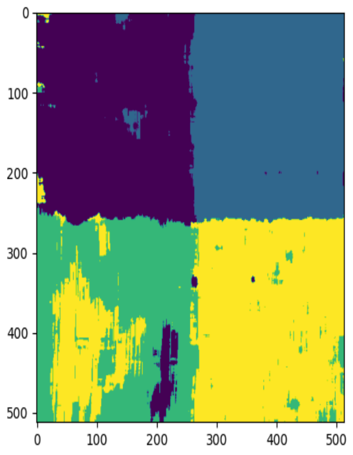
As we can see from both the images and the confusion matrix, the classifier does a pretty good job at predicting the correct classes. The worst result, as we see both on the image and the confusion matrix, is a lot of pixels from class 3 is labeles as class 4, precicely 7.78%. However, the true positives are all around 90% or higher, which is a good result when dealing with multiple classes. Interestingly, these results showed us that class 1 and 2, which we earlier said were the most alike, were actually the ones with best results. However, we do see that they also overlap a bit, which was to be expected.

We then perform the same test on more masks and images:



Here is the mask and the resulting classification image:

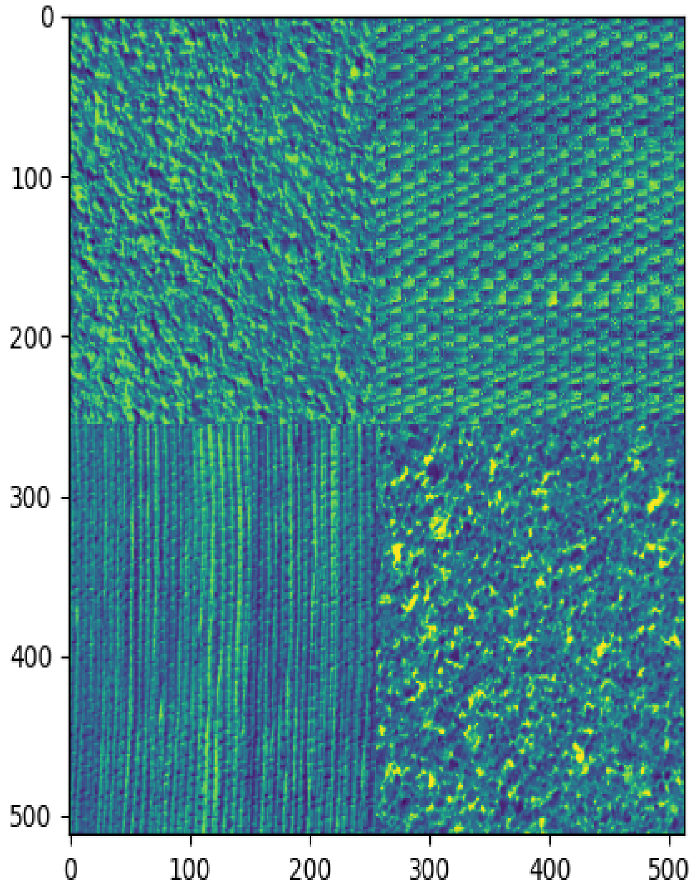




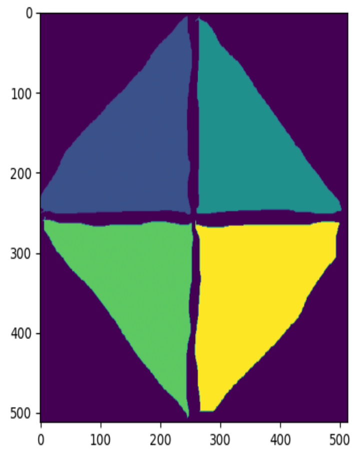
with the resulting confusion matrix:

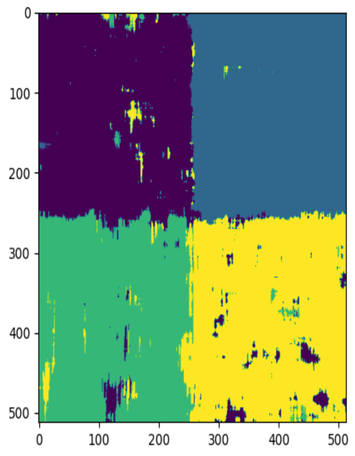
```
[[9.67163086e+01  1.77001953e+00  4.15039062e−01  1.09863281e+00]
 [1.79138184e+00  9.81964111e+01  0.00000000e+00  1.22070312e−02]
 [6.05163574e+00  0.00000000e+00  6.84585571e+01  2.54898071e+01]
 [3.66210938e−01  1.21765137e+00  6.06842041e+00  9.23477173e+01]]
 overall  accuracy  =  0.8892974853515625
```

and then with:



Here is the mask and the resulting classification image:





with the resulting confusion matrix:

```
[[95.69244385  0.62866211  1.69525146  1.98364258]
 [ 0.54473877 98.64501953  0.          0.8102417 ]
 [ 3.76586914  0.         91.07666016  5.1574707 ]
 [ 4.65698242  0.66070557  1.3961792  93.28613281]]
 overall accuracy = 0.9467506408691406
```

As we see from the overall accuracy, the 2nd test gave the worst result. This can be explained due to the fact that it had less training data in the mask, but also more unevenly distributed and spread out data for each class, compared to the other two masks. The best perfoming mask was the last one, where we see each class gets about the same amount of data.

All in all we can say that the classifier with our chosen features worked out well. It clearly captures where each texture is, and has a high enough accuracy to be content with.