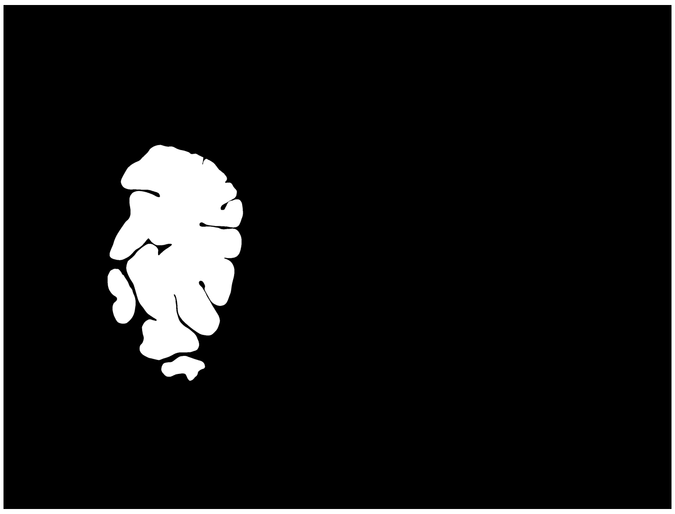
Figure 1.1: Mask of a cross section

1 Approach

1.1 The Data

The data is not given as a classic table and therefore not directly useable for classification. It consists of two sets of images. The first one contains the original images of each cross section as RGB images (fig. **??**). The second one has the masks (fig. 1.1), a gray scale image with only black or white pixels. A pixel is white if it belongs to the cross section and black if it does not. There are 762 images with a resolution of 3272×2469.

As stated above there are two classes in total, so that the classification task is a binary one. Before a classifier can be trained, the images have to be converted into a attribute value format and saved in a file. This could be a simple csv file or any other format that can handle table like data. In this case the libsvm format was chosen, because it is supported by many svm libraries, e.g. scikit-learn. Every row represents one instance, which is in this case one pixel and its features. Currently the only features available are the rgb values of a pixel from the original image and the class label from the mask. Listing 1 shows the libsvm format schematically.

Listing 1: libsvm format

```
<class_label> <feature_id>:value ... <feature_id>:value
<class_label> <feature_id>:value ... <feature_id>:value ...
```

1.2 Data Preparation

In the first preprocessing step the libsvm data is created. The easiest method saves for each pixel the rgb values and the class label. This is done for every image resulting in 762 files. Every file has 8.078.568 instances, so there are 6.155.868.816 pixels in total. A first trivial approach took about 3 days to generate all attribute value files. The RGB values can be read directly from the original images. The class label is set to 1 of the corresponding pixelvalue in the mask is greater than 0. If this is not the case it is set to 0. This was done with a python script, which was executed on Judge, a supercomputer of the JSC with 2472 nodes and 412 graphic processors. The python script is started with a msub script reserving 1 one core with 1 node. To speed up the calculation time, the python script was changed so that it expects a range as parameters that says which image files it should convert. With this several jobs can be started with a different range argument resulting in a faster computation.

Building a model on the whole set would take a huge amount of time and is therefore unreasonable for creating a classifier. Instead of using the complete data set samples have to be used. As figure 1.1 shows there are more instances of class 0 (black pixels) than of class 1 (white pixels). To weight both classes equally, the sampled set is balanced.

The first approach is a balanced random sampling method. For a given percentage p and a data set with N instances the method draws $p \cdot N$ samples from a file. If the sample is balanced, $p \cdot N \cdot 0.5$ instances of each class are drawn randomly. This can easily be parallelized since the files can be sampled independently.

An alternative method is to resize the images to a much smaller resolution (e.g. 256×256). This is done for a number of images that are uniformly distributed among the different layers of the brain. An image has several different sections like the cross section, the brain in the background or the ice. If random sampling is used, it may happen that some of these sections are not represented by the sample. The lower the percentage p is, the higher the chance gets that some sections is missed.

Because of this later samples were made by resizing the images.

This yields a first sample with three features. To improve the classifier the number of features has to be increased.

1.2.1 HSV color space

An easy method to increase the number of features is to use another color space in addition to the given RGB values. In this case the HSV color space was chosen, because it is similiar to the human color vision. Furthermore it is used for object detection because methods used on grayscale images can easily be used on images in the HSV space by using them on each color component.

The HSV color space has three components hue, saturation and value. The hue (H) is the angle on the chromatic circle and is therefore between 0 and 360. The second and third components saturation (S) and lightness (V) are given in percentage, so their value is between 0 and 1 (or $0 - 100\%$). If the saturation is 1, the color is clear. For a low saturation the color becomes gray. If the lightness is 0, the color is black.

The values for a pixel can easily be transformed from RGB to HSV with the following formula:

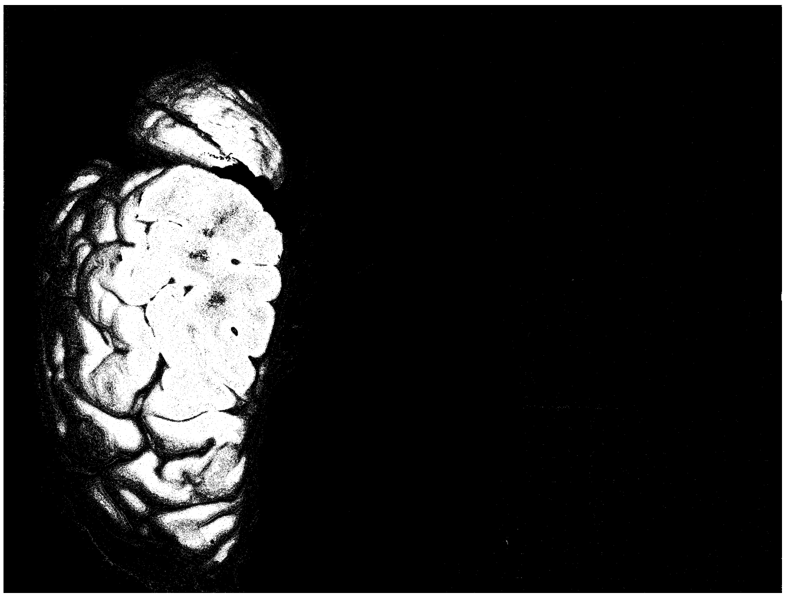

Figure 1.2: predicted image using svm with rbf kernel

$$
\begin{aligned}
Max &= max(R, G, B) \\
Min &= min(R, G, B) \\
H &= \begin{cases} 0, & \text{if Max =Min} \Leftrightarrow R = G = B \\ 60^\circ \cdot \left(0 + \frac{G-B}{Max-Min}\right) & \text{if Max =R} \\ 60^\circ \cdot \left(2 + \frac{B-R}{Max-Min}\right) & \text{if Max =G} \\ 60^\circ \cdot \left(4 + \frac{R-G}{Max-Min}\right) & \text{if Max =B} \end{cases} \\
S &= \begin{cases} 0 & \text{if Max =0} \Leftrightarrow \text{R=G=B=0} \\ \frac{Max-Min}{Max} & else \end{cases} \\
V &= Max
\end{aligned}
$$

Since this computation can be done independently for every pixel, the features can easily be computed in parallel and be added to existing data. While computing the HSV values is simple, they also add only few additional information. Because of this the increase of accuracy is only marginal.

1.2.2 Local Features

Despite adding the HSV color space, there is still no information on the neighbourhood of a pixel. A first level approach to get information on the neighbourhood of a pixel are statistical values like mean or standard deviation. The statistical values are calculated

Listing 2: Calculate standard deviation

```python
def calc_std(img,size,use_mean=False):
  """Calculate the local standard deviation for each pixel of an image.
    Arguments:
        img : rgb image
        size : size of the window of neighbouring pixels
        use_mean : boolean, if False the standard deviation of R,G and B
            value are calculated and return as a 3 dimensional array.
                   if True the standard deviation of the mean is returned.
    Return:
      array_like, 3d or 2d
    """
    if not use_mean:
        std = np.zeros(img.shape)
        for i in range(img.shape[2]):
            std[:,:,i] = filters.generic_filter(img[:,:,i],np.std,size)
    else:
        mean_img = np.mean(img,axis=2)
        std = filters.generic_filter(mean_img,np.std,size)
    return std
```

on an $N \times N$ array around a pixel and assigned to the centered pixel. This is done for every pixel of the image. Listing 2 shows how the standard deviation is calculated for an image. This is done with the numpy and scipy modules, which are optimized so that the calculation time is not too long.

The *generic_filter* function generates a window with the given size for every pixel and calls the given function (e.g. numpy.std). However, the parameter of the called function is an one dimensional array. If the function needs a two dimensional one, it has to know the original shape of the window and reshape it. For most statistical measures this is not needed. The standard deviation is an indication for the homogenity of the pixel's neighbourhood.

By using the local standard deviation as a feature some first information of the neighbourhood is available. This can be further improved by other features.

1.2.3 Image Segmentation

Watershed is an image segmentation algorithm that is used on grayscale images. The gray level of a pixel is interpreted as its altitude and the image is seen as a ground. Water, that is filled into this ground from different markers, flows to a local minimum.

The algorithm floods basins from markers, which were set by the user. When two basins from different markers meet a watershed is drawn. The markers can be set in an easy way by using thresholds. While there are more complex ways using image processing methods, the threshold approach is used to keep the feature generation as simple as possible.

This yields a binary feature that is either one or zero and is used for training the

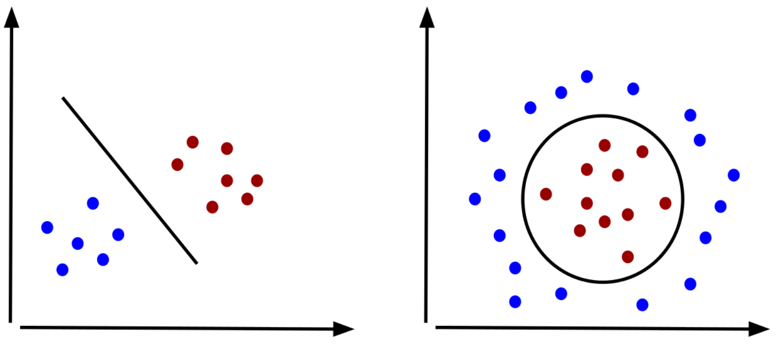
Figure 1.3: Classification using Support Vector Machines. Linear expample on the left side. Non linear example on the right side.

classifier.

1.3 Data Modeling

1.3.1 Support Vector Machines

Support vector machines (SVMs) are one of the preferred classification methods lately. They have a high accuracy, but their training time is quite long. A model can easily be described by the found support vectors. Fig 1.3 shows a basic visualization of an SVM. On the left side a line is drawn that seperates both classes. This is the decision boundary. The right side is not linear separable. A kernel is used to push the data into a higher dimension, so that the classes are linear seperable again.

A kernel defines the used scalar product. The most popular kernels are the following:

- linear: $K(x, y) = \sum_{i=0}^{n} x_i \cdot y_i$
- polynomial: $K(x, y) = (c + \sum_{i=0}^{n} x_i \cdot y_i)^d$
- rbf: $\exp(-\frac{||x-y||^2}{2\sigma^2})$

Unlike other classification methods (e.g. neural networks, naive bayes) SVMs need only a small training set in theory. Additional instances only affect the classifier if they contain new support vectors.

More information on SVMs in general can be found at [?].

Because support vector machines maximize the margin between the decision boundary and the support vectors, they do not overfit as easily as other classifiers like decision trees for example. However, solving the quadratic problem can have a complexity between $O(n^2)$ and $O(n^3)$ depending on the used algorithm and implementation. As figure 1.4 shows training an svm takes much more time to be trained than other classifiers. This is especially true if a non linear kernel is used, e.g. rbf kernel. A linear svm is much faster but this goes along with a drop in accuracy. To get both a feasible training time and a good accuracy, a kernel approximation is used together with a linear SVM. A kernel

Classifier	Accuracy	training time in s	test time in s
SVC (sklearn), rbf, 20k instances	0.8752580408	39.545465	109.808209
SVC (sklearn), rbf, gamma=0.1, C=1, 20k instances	0.9579808675	45.242423	66.846724
libSVM twister, 20k instances	0.9592930497	445.867	-
libSVM twister, 40k instances	0.963317509	934.912	-
libSVM twister, 100k instances	0.9665214475	3077.239	-

Figure 1.4: SVMs from scikit-learn and Twister with rgb as features

Classifier	Accuracy	training time in s	test time in s
GaussianNB	0.9475908597	0.042224	0.075098
KNN	0.9677196684	1.57	5.29
Decision Tree	0.9569552165	0.563767	0.057934
RandomForest	0.9637635858	1.698716	0.399092

Figure 1.5: Different classifiers used with rgb features.

approximation is used before training a SVM (see listing 3). It adds additional random features to the data. For the rbf kernel this is done by a Monte Carlo approximation of its Fourier transform. More information on random fourier features and random binning features can be found at [?].

A second approach is the Nystroem method [?] which can approximate every kernel and not only the rbf one. It uses a subsample of the data set to approximate a kernel. As [?] shows the nyström method can achieve a better generalization in some cases. For the brain analytics use case however this could not be noted.

1.3.2 Other Classifiers

There are some other classifiers besides support vector machines. While the focus is on SVMs and they were optimized the most, some models with other classifiers were tested too. Naive Bayes is one of them. Due to its simplicity it is quite fast. Unfortunatly the accuracy score is not good since Naive Bayes assumes independency of features which is obiously not the cause for RGB values of an image.

While KNearestNeighbour considers feature dependencies and therefore has a higher accuracy, it's training time is also higher than the one of naive bayes. Apart from that

Listing 3: kernel approximation using scikti learn

```
from sklearn.kernel_approximation import RBFSampler
rbf = RBFSampler(gamma=2)
X_features = rbf.fit_transform(X_train)
X_test_features = rbf.transform(X_test)
```

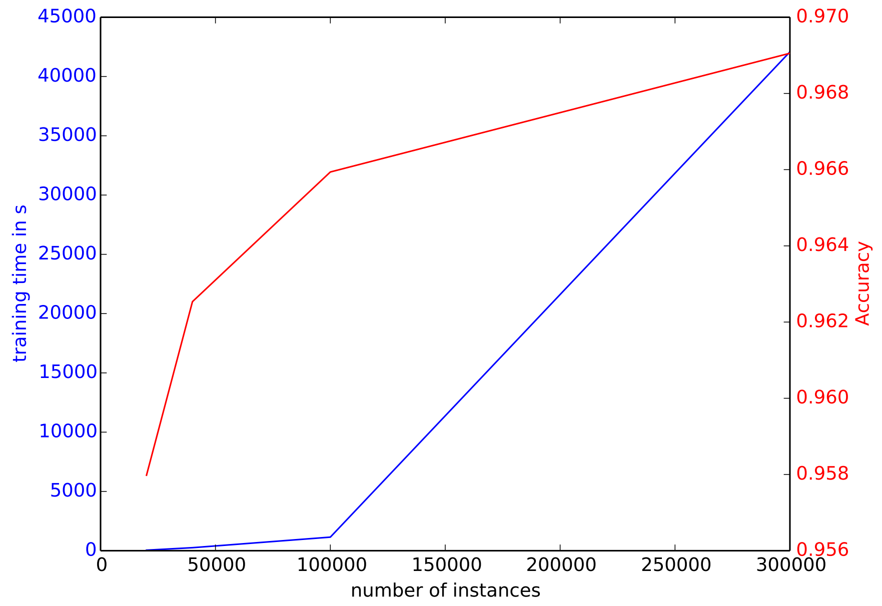

Figure 1.6: SVC training time with increasing sample

figure 1.5 shows that the prediction takes much longer since the distances to all instances have to be calculated.

Decision trees have a better accuracy than naive bayes as well but not as good as KNN. While their training time is larger than the one of naive bayes, it is way faster than KNN. For the random forest classifier, which uses ten randomized decision trees, the accuracy as well as the training and testing time increase.

1.3.3 Multiple Models

With only one model for all the different cross section the prediction is not very good for some cross sections. While it is not reasonable to use the x and y coordinates of a pixel as features because the brains position changes between the images, it is possible to use the z axis as a feature. The z position of an image is contained in its file name, since all the images are numbered. Naturally, for this to work future cross sections have to be numbered in the same way. The alternative to using the z position as a feature with one model is to divide the brain in several layers and create a model for every layer.

This has several advantages in comparison to using one model. By increasing the number of models each single model is simpler. It also decreases the training time. This has two reasons. The first is that the amount of training data for each model is less because the data is sliced into different layers. In addition the training of the models can easily be parallelized because they are completely independent from each other.

Using multiple models can also be useful if the color between the different layers varies slightly.

1.3.4 Grid Search

While scikit-learn already implements a function that performs grid search on a given set of parameters and a classifier, this function can not be used if kernel approximation is used as well. The parameters that have to be optimized are the penalty parameter C and kernel coefficient γ. If kernel approximation is used, the linear SVM does not expect an argument γ. Instead γ is set in the RBFSampler. Therefor the scikit-learn function can not be used and a modified version was implemented. The new version generates a new data set with the additional features and the current γ. Thereafter a linear SVM is trained with the given C and cross validated. All results are saved in a list and returned. To speed up the grid search, it is parallized with IPython using the LoadBalancedView. Unlike the DirectView interface this one does not allow direct access to the individual engines. Instead of that the IPython scheduler assigns the tasks to the engines minimizing the idle time of the engines. Because of this the grid search can not only be used on a single machine but on clusters, if the IPython cluster is setup.

1.4 Data Post Processing

After the model is created an image can be predicted. During the preprocessing the image is flattened and several features are added, so that every row contains one pixel. Since the order of the pixel is contained, the predicted labels can be reshaped to form a mask for the given image. This gives a much better impression of the performance of the model than the accuracy score because the data is not balanced. If the mask of the image is given, the predicted mask and the original mask can easily be compared by opening both masks or calling the *compare_to_mask* function. The latter calculates the accuracy, f-score and the confusion matrix. Furthermore it also visualizes the confusion matrix as an image.

1.5 Deployement

Here

This notebook shows the different steps that are needed to classify the brain cross sections. It starts with the data preprocessing and ends with a final model. Several different functions are used by the imported custom modules. This is done, so that the notebook stays clear. If you want to take a detailed look at the used functions, feel free to use the different source files.

At the beginning the client is set up, so that the engines can be accessed and used.

Listing 4: Client Setup

```
# Import the client and the imp module to load source files on all engines
from IPython.parallel import Client
c = Client() dview = c[:]
dview.block = True
with dview.sync_imports():
```

```
import imp
```

The training data can be sampled. In this example this is done by rescaling some images to a size of 256x256.

Listing 5: Generating samples

```
# resizing the images
import glob masks = glob.glob("../classification/data/rescaled256x256/masks/MSA*.
    tif")
images = glob.glob("../classification/data/rescaled256x256/images/MSA*.tif")
from PIL import Image
size=(256,256)
resized_images = [Image.open(i).resize(size) for i in images]
resized_masks = [Image.open(m).resize(size) for m in images]
```

The following code block shows how to generate libsvm formated data from an original image and the hand labeled mask. This is done on multiple engines started by ipython. In this example the used features are:

$R, G, B, std(R), std(G), std(B), segmentation_bit, H, S, V$

The window size for the image is 16x16 pixel. The thresholds for watershed segmentation are set as p=0.2 and q=0.8. Custom features can be added easily. A feature that needs the original image needs to be set in the new_features list. If only the pixel is needed, the online_feature list is sufficient.

Because callables are passed, the user can add any new function. The only condition is that the first argument is the image/pixel. If the function has more arguments, the *partial* function can be used to create a new one with only one argument.

Listing 6: Converting and Adding Features

```
# load data conversion and feature extration modules locally and on the engines
dc = imp.load_source("data_conversion","../parallel/data_generation/data_conversion
    .py")
fe = imp.load_source("feature_extraction","../classification/feature_extraction.py"
    )
# use the direct view to load the modules
dview.execute('dc = imp.load_source("data_conversion","../parallel/data_generation/
    data_conversion.py")')
dview.execute('fe = imp.load_source("feature_extraction","../classification/
    feature_extraction.py")')
# set masks and images, that are to be converted, as lists
# be aware that masks[0] must belong to images[0]. This is done by sorting the
    lists
import glob
masks = sorted(glob.glob("../classification/data/rescaled256x256/masks/MSA*.tif"))
images = sorted(glob.glob("../classification/data/rescaled256x256/images/MSA*.tif")
    )
# scatter the images and mask to the engines
dview.scatter("images",images)
dview.scatter("masks",masks)
with dview.sync_imports():
  from functools import partial
# add features and convert the images
out_dir = "../classification/data/rescaled256x256/test"
dview['out_dir'] = out_dir
```

```
cmd = 'dc.convert_and_save(images,masks,out_dir,new_features=[partial(fe.calc_std,
    size=16),partial(fe.findSegmentation,p=0.2)],online_features=[fe.add_hsv])'
dview.execute(cmd)
```

The data can be combined into training and testing sets. The data is shuffled and the sets are class balanced by default. The *combine_files* method can be used to generate training sets for the different layers, if multiple models are used.

Listing 7: Create training and testing sets

```
sampling = imp.load_source("sample_files","../data_generation/random_sampling/
    sample_files.py")
sampling.OUT = "../classification/data/rescaled256x256/test"
sampling.combine_files("../classification/data/rescaled256x256/test/","ALL.svm")
sampling.split_train_test("../classification/data/rescaled256x256/test/","ALL.svm",
    test_size=0.5,shuffle=True)
```

To build a classifier, first the training and test data have to be loaded. In this case it is a single cross section, but any combination of the different images can be used. This can be done with *sample.combine_files*.

The data is normalized and additional features are added. Instead of the RBFSampler the Nystroem method can be used to generate different features or approximate another kernel.

At the end the classifier is trained with parameters optimized with a grid search. Instead of building one classifier, grid search may be used to test different parameters on the current data set. After this the model can easily be saved with the pickle module which is part of the python standard library. It can also be used to save the RBFSampler which is needed if new data should be predicted.

Listing 8: Build a SVM classifier

```
from skimage.io import imshow,imread
import numpy as np
from sklearn.kernel_approximation import RBFSampler
from sklearn.svm import LinearSVC
from sklearn.datasets import load_svmlight_file
# load training set
X_train,y_train = load_svmlight_file("../classification/data/rescaled256x256/MSA_03
    -2009_dXXXX-XX-XX_s0110.svm")
X_train = X_train.toarray()
# normalize data
max_val = X_train.max(axis=0)
min_val = X_train.min(axis=0)
X_train = (X_train-min_val)/(max_val-min_val)
# load test image
X_test,y_test = load_svmlight_file("../classification/data/segmentation/MSA_03-2009
    _dXXXX-XX-XX_s0200.svm")
X_test = X_test.toarray()
X_test = (X_test-min_val)/(max_val-min_val)
# generate additional features
rbf = RBFSampler(gamma=2)
X_features = rbf.fit_transform(X_train)
X_test_features = rbf.transform(X_test)
# train linear SVM on highdimensional data
linearSVM = LinearSVC(dual=False,C=1000000)
```

```python
linearSVM.fit(X_features,y_train)
# save model
import pickle with open("SVMClassifier.pkl","w") as f:
    pickle.dump(linearSVM,f)
```

If the classifier is not trained in the current session, it can be loaded with the pickle module. After adding the random features the class can be predicted usign the *predict* method of the classifier. The predicted labels can be reshaped, so that the new mask can be printed.

Listing 9: Classify test image

```python
# load model
import pickle
with open("SVMClassifier.pkl","r") as f:
    linearSVM = pickle.load(f)
# load hand labeled mask
mask = imread("../classification/data/segmentation/masks/MSA_03-2009_dXXXX-XX-
    XX_s0200.tif")
labels = linearSVM.predict(X_test_features)
labels = labels.reshape(mask.shape)
%matplotlib inline
imshow(labels)
```

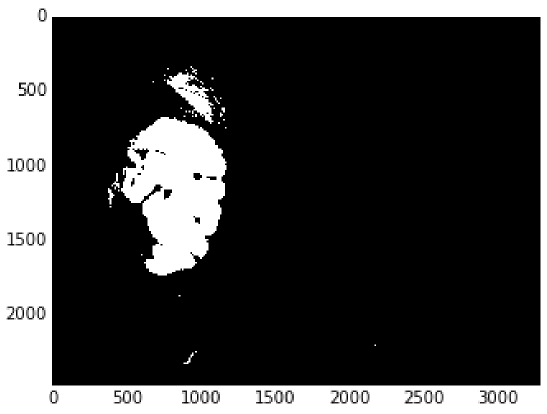

Figure 1.7: predicted mask

If the data is also labeled by hand, the quality of the model can be measured. The following function calculates the accuracy as well as the f-score. It also returns the confusion matrix.

Because the classified data is an image, the confusion matrix is visualized as an image. All true positives are colored white and all true negatives black. With 100% accuracy this would result in the hand labeled mask. Since this is most often not the case, all false positives are marked red and all false negatives blue.

This gives a good first impression on the quality of the classifier. If the predicted labels are post processed by hand, it furthermore gives an indication which regions cause problems.

Listing 10: Evaluation

```
from sklearn.metrics import confusion_matrix,f1_score,accuracy_score
from collections import namedtuple
AccuracyMetrics = namedtuple('AccuracyMetrics',['accuracy','fscore','
    confusion_matrix','image'])
def compare_to_mask(pred,mask):
    size = pred.shape
  img = np.zeros((mask.shape[0],mask.shape[1],3),dtype='uint8')
    tp = np.dstack(((pred>0).reshape(size),(mask>0).reshape(size)))
    fp = np.dstack(((pred>0).reshape(size),(mask==0).reshape(size)))
    fn = np.dstack(((pred==0).reshape(size),(mask>0).reshape(size)))
    img[np.all(tp,axis=2),:] = 255
    img[np.all(fp,axis=2),0] = 255
    img[np.all(fn,axis=2),2] = 255

    acc = accuracy_score(mask.flatten(),pred.flatten(),normalize=True)
    fscore = f1_score(mask.flatten(),pred.flatten())
    cm = confusion_matrix(mask.flatten(),pred.flatten())

    return AccuracyMetrics(acc,fscore,cm,img)

metrics = compare_to_mask(labels,y_test.reshape(labels.shape))
imshow(metrics[3])
```

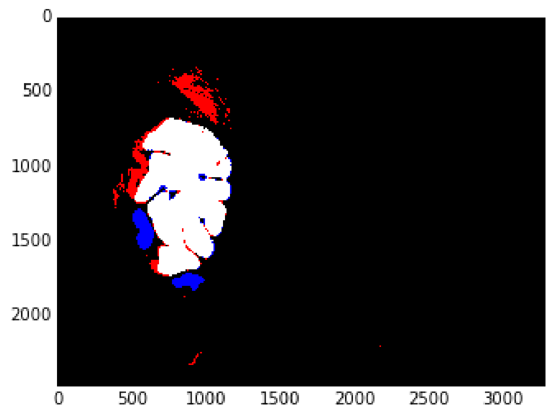

Figure 1.8: Confusion matrix visualized