Internship Report

Philipp Glock

8th December 2014

1 Introduction

The division Federated Systems and Data of the Juelich Super Computing Centre provides access to distributed resources like HPC systems and data. This is done by implementing open standards and simplifying usage and administration of these services. Therefor the division is one of the core development partners of the UNICORE Grid middleware and has members in the Research Data Alliance and the Open Grid Forum.

The research group High Productivity Data Processing works on solutions to overcome problems and challenges occurring while handling 'big data' problems. This can be splitted into three categories.

1. **Investigate Generic Data Methods:** How to overcome limitations of processing and analyzing large amounts of data (e.g. in-memory databases, data privacy methods and query processing)
2. **Machine Learning:** Use and extend serial or parallel machine learning techniques, like classification and clustering, to improve performance
3. **Smart Data Analytics Applications:** Find and develop suitable applications for general data analysis

The goal of the internship is to build classification models for data sets that show problems of 'big data' . Support Vector Machines are the focused method, but other methods are considered as well. This report shows the different problems and their solutions of several approaches and implementations.

2 Background

The primary programming language used during this internship is python. Python is a dynamic scripting language and provides many nice modules for scientific use cases like numpy and scipy. There are also some machine learning modules and the scikit-learn module [1] is one of the most used and stable ones. While python alone may not have the best performance, it is easy to improve it by using c or fortran code. This is done by most libraries. The serial support vector machine algorithms of scikit-learn are based on the well known libsvm library for example. This makes python and the scikit-learn module a good choice for classification tasks.

The tasks are executed on a super computer at the Juelich Supercomputing Centre. It uses a PBS based batch system to handle the different jobs of users and distribute the available nodes. A popular method for distributed memory multiprocessing is MPI. It is, however, not being used here.

IPython is an interactive python shell with many improvements. One of them is interactive parallel computing. This allows parallelizing python code in an interactive python session and has a robust error handling.

Listing 1: pbs.controller.template

```
#PBS -l nodes={n/4}:ppn=4
#PBS -l walltime=24:00:00
#PBS -l pvmem=1GB
#PBS -N ipython-engine-glock
#PBS -m abe
#PBS -j oe
#PBS -o ipc.engine-$PBS_JOBID.out

module load python
module load intel
module load mvapich2

cd $PBS_O_WORKDIR
which ipengine
mpiexec -np {n} ipengine --profile=pbs
```

After setting up a profile and creating PBS templates (e.g. listing 1), you can start a cluster easily using:

$$ipcluster\ start\ --profile = pbs\ -n\ 8$$

More information on ipython in general and the setup for parallel usage can be found at [2].

3 Human Brain Project - brain analytics

3.1 The Project

The Human Brain Project aims to acquire a better understanding of the human brain by modeling it as a whole. One sub task of this project is to build a 3d model of the human brain.

The brain analytics data set provides images of different cross sections of a brain and each pixel is labeled whether it belongs to the cross section or not. Labeling a whole brain takes a huge amount of time and resources. So instead of labeling new images by hand, this data is used to build a model that automatically classifies a pixel. If this classification is good enough, the predicted 2d brain parts should be used to rebuild a 3d model of a human brain.

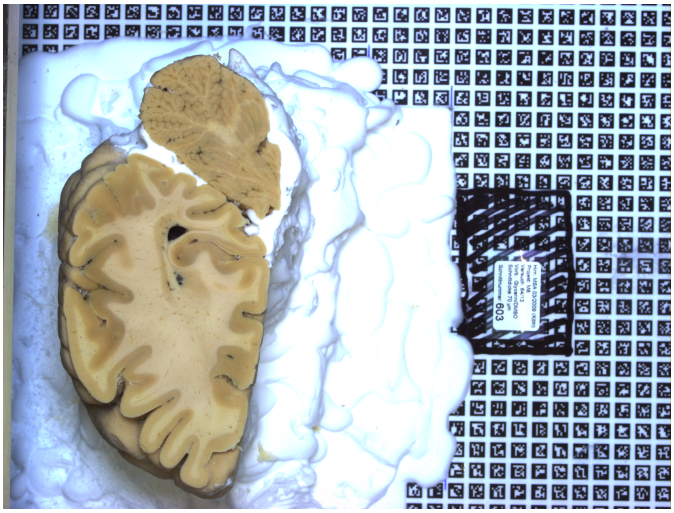

Figure 3.1: An image of a cross section of a human brain

4 Approach

4.1 The Data

The data is not given as a classic table and therefor not directly useable for classification. It consists of two sets of images. The first one contains the original images of each cross section and a corresponding mask. A mask is a grayscale 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, resulting in a huge amount of pixels.

Listing 2: libsvm format

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

The classification problem is a binary one since there are only two classes. Before a classifier can be trained, the images have to be converted into a attribute value format. The libsvm format was chosen, because it is well known and many libraries support it. 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.

4.2 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.

Building a model on this set would take a huge amount of time and is therefor unreasonable for selecting a model. Instead of using the complete data set samples have to be used. As figure 4.1 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. This yields a first sample with three features. To improve the classifier the number of features has to be increased.

4.2.1 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. It represents a color by its hue, saturation and value. The values for a pixel can easily be transformed from RGB to HSV with the following formula:

Figure 4.1: Mask of a cross section

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

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.

4.2.2 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 on an $N \times N$ array around a pixel and assigned to the centered pixel. Listing 3 shows how the standard deviation is calculated for an image. This is done with the numpy and scipy modules. The *generic_filter* function generates a window with the given size for

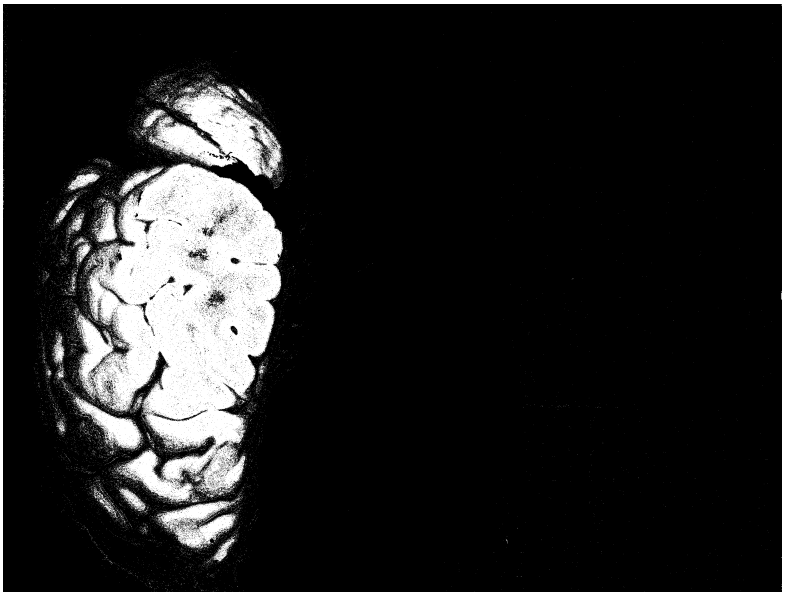

Figure 4.2: predicted image using svm with rbf kernel

every pixel and calls the given function (e.g. numpy.std). The standard deviation is an indication for the homogenity of the pixel's neighbourhood.

Listing 3: Calculate standard deviation

```
1 def calc_std(img,size,use_mean=False):
2   """Calculate the local standard deviation for each pixel of an image.
3     Arguments:
4         img : rgb image
5         size : size of the window of neighbouring pixels
6         use_mean : boolean, if False the standard deviation of R,G and B
              value are calculated and return as a 3 dimensional array.
7                       if True the standard deviation of the mean is returned
                            .
8     Return:
9       array_like, 3d or 2d
10    """
11    if not use_mean:
12        std = np.zeros(img.shape)
13        for i in range(img.shape[2]):
14            std[:,:,i] = filters.generic_filter(img[:,:,i],np.std,size)
15    else:
16        mean_img = np.mean(img,axis=2)
17        std = filters.generic_filter(mean_img,np.std,size)
18    return std
```

By using the local standard deviation as a feature some information of the neighbourhood is used, but this can be improved with other features.

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 4.3: 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 4.4: Different classifiers used with rgb features.

4.2.3 Image Segmentation

A more complex feature can be achieved by performing watershed segmentation. It is used on grayscale images and the pixel values are treated as a elevation. 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 simple

4.3 Data Modeling

4.3.1 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.

SVMs can model nonlinear decision boundaries by using other kernels instead of the linear kernel to increase the dimension of a vector. 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: $\exp(-\frac{||x-y||^2}{2\sigma^2})$

More information on SVMs in general can be found at [3].

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 4.3 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 approximation is used before training a SVM (see listing 4). 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 [4].

Listing 4: 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)
```

A second approach is the Nystroem method [5] which can approximate every kernel and not only the rbf one. It uses a subsample of the data set to approximate a kernel. As [6] shows the nyström method can achieve a better generalization in some cases. For the brain analytics use case however this could not be noted.

4.3.2 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 figure 4.4 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.

4.3.3 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

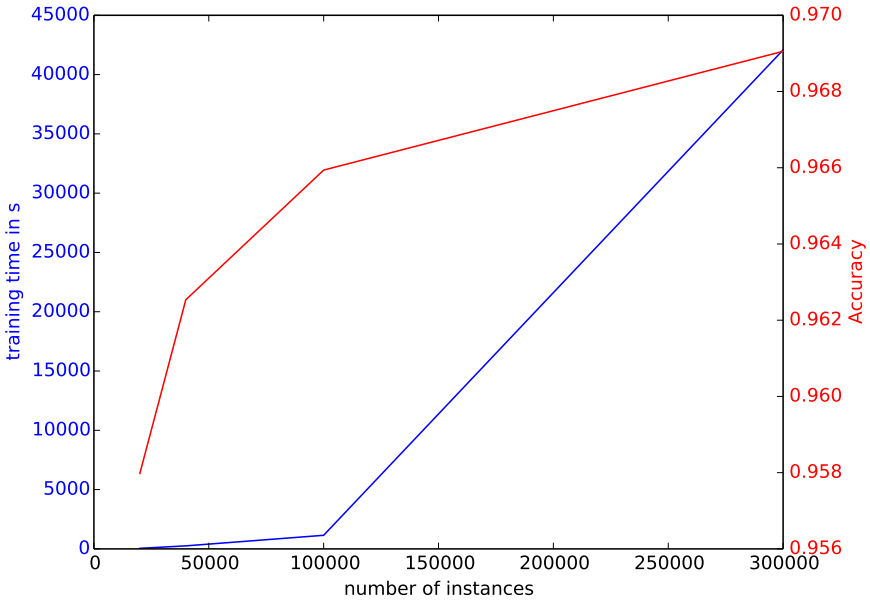

Figure 4.5: SVC training time with increasing sample

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.

- simpler models Ausformulieren?
- less training data for each model
- more specialized models

4.3.4 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.

4.4 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.

4.5 Deployement

4.6 Conclusion

References

[1] http://scikit-learn.org/stable/. [PLACEHOLDEROnline; accessed 20-January-2015].

[2] http://ipython.org/documentation.html. [PLACEHOLDEROnline; accessed 20-January-2015].

[3] J. Han, M. Kamber, and J. Pei. *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2011.

[4] A. Rahimi and B. Recht. Random features for large-scale kernel machines. http://www.eecs.berkeley.edu/~brecht/papers/07.rah.rec.nips.pdf.

[5] Christopher Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. In *Advances in Neural Information Processing Systems 13*, pages 682–688. MIT Press, 2001.

[6] Tianbao Yang, Yu-feng Li, Mehrdad Mahdavi, Rong Jin, and Zhi-Hua Zhou. Nyström method vs random fourier features: A theoretical and empirical comparison. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 476–484. Curran Associates, Inc., 2012.