Computer Science – Research and Development manuscript No.
(will be inserted by the editor)

An object-oriented library for systematic training and comparison of classifiers for computer-assisted tumor diagnosis from MRSI measurements

Frederik O. Kaster · Bernd Merkel · Oliver Nix · Fred A. Hamprecht

Received: date / Accepted: date

Abstract We present an object-oriented library for the systematic training, testing and benchmarking of classification algorithms for computer-assisted diagnosis tasks, with a focus on tumor probability estimation from magnetic resonance spectroscopy imaging (MRSI) measurements. In connection with a graphical user interface for data annotation, it allows clinical end users to flexibly adapt these classifiers towards changed classification tasks, to benchmark various classifiers and preprocessing steps and to perform quality control of the results. This poses an advantage over previous classification software solutions, which required expert knowledge in pattern recognition techniques in order to adapt them to changes in the data acquisition protocols. This software will constitute a major part of the MRSI analysis functionality of RONDO, an integrated software platform for cancer diagnosis and therapy planning which is under current development.

This research was supported by the Helmholtz International Graduate School for Cancer Research and by the German Federal Ministry for Education and Research within the DOT-MOBI project (grant no. 01IB08002).

F.O. Kaster · F.A. Hamprecht
University of Heidelberg, Heidelberg Collaboratory for Image Processing, Speyerer Straße 6, D-69115 Heidelberg
Tel.: +49-6221-545274
Fax: +49-6221-545276
E-mail: frederik.kaster@iwr.uni-heidelberg.de

F.O. Kaster · O. Nix
German Cancer Research Center, Im Neuenheimer Feld 280, D-69120 Heidelberg

B. Merkel
Fraunhofer MeVis Institute for Medical Image Computing, Universitätsallee 29, D-28359 Bremen

Keywords Magnetic resonance spectroscopy imaging · Computer-assisted diagnostics · Statistical classification · Automated quality control

1 Introduction

1.1 Computer-assisted tumor diagnostics based on MRSI measurements

Imaging methods for the *in vivo* diagnostics of tumors fall into three categories based on the different physical mechanisms they exploit: In computer tomography (CT), X-rays are transmitted through the body, which are attenuated differently in different tissue types. In nuclear medicine methods such as positron emission tomography (PET) or single photon emission computed tomography (SPECT), one detects the radiation of radioactive nuclides, which are selectively accumulated in the tumor region. Finally, magnetic resonance imaging (MRI) exploits the fact that various nuclei (namely protons) have a different energy when aligned in the direction of an external magnetic field than when they are aligned opposite to it. By injecting a radiofrequency wave into the imaged body, one can perturb some protons out of their equilibrium state into a higher-energy state: the radiofrequency signal which they emit upon relaxation is then measured, and its amplitude is proportional to the concentration of the protons in the imaged region. This measurement process can be performed in a spatially resolved fashion, so that a three-dimensional image is formed.

Standard MRI produces a scalar image based on the total signal of all protons, irrespective of the chemical compound to which they belong: typically, the protons in water molecules and in lipids make the highest contribution due to the large concentration of these molecules. However, the protons in different compounds can be distinguished by their

resonance frequencies in the magnetic field (the so-called *chemical shift*), and it is possible to resolve the overall signal not only spatially, but also spectrally: this leads to magnetic resonance spectroscopy imaging (MRSI) or chemical shift imaging (CSI), for which a complex spectrum is obtained at each image voxel instead of a single scalar value as in MRI (de Graaf, 2008). Hence it is possible to measure the local abundance of various biochemical molecules noninvasively, and thereby gain information about the chemical make-up of the body at different locations: besides water and lipids, most major metabolites can be identified in the MRSI spectra, e.g. the most common amino acids (glutamate, alanine, ...), the reactants and products of glycolysis (glucose, ATP, pyruvate, lactate), precursors of membrane biosynthesis (choline, myo-inositol, ethanolamine), energy carriers (creatine) and tissue-specific marker metabolites (citrate for the prostate, N-acetylaspartate or NAA for the brain). As a downside, these metabolites occur in much lower concentrations than water, hence the spatial resolution must be far coarser than in MRI: only by collecting signal from a volume of typically 0.2–2 cm^3, a sufficient signal-to-noise ratio can be achieved.

MRSI provides valuable information for the noninvasive diagnosis of various human diseases, e.g. infantile brain damage (Xu and Vigneron, 2010), multiple sclerosis (Sajja et al, 2009), hepatitis (Cho et al, 2001) or several psychiatric disorders (Dager et al, 2008). The most important medical application field lies in tumor diagnostics, especially in the diagnosis and staging of brain, prostate and breast cancer as well as the monitoring of therapy response (Gillies and Morse, 2005). In tumors, healthy cells are destroyed and the signals of the biomarkers characteristic for healthy tissue (e.g. citrate for the prostate, NAA for the brain) are decreased. On the other hand, biomarkers for pathological metabolic processes often occur in increased concentrations: choline (excessive cell proliferation), lactate (anaerobic glycolysis), mobile lipids (impaired lipid metabolism). The top right and bottom right spectra in fig. 1 are typical examples of spectra occurring in healthy brain tissue and in brain tumor, respectively.

While MRSI has proved its efficacy for radiological diagnostics, it is a fairly new technique that yet has to gain ground in routine radiology and in the training curricula of radiologists. Furthermore, the visual assessment is harder and more time-consuming than for MRI: while most medical imaging modalities provide two- or three-dimensional data, MRSI provides four-dimensional data due to the additional spectral dimension. Automated decision-support systems may assist the radiologists by visualizing the most relevant information in form of easily interpretable *nosologic images* (de Edelenyi et al, 2000): from each spectrum, a scalar classification score is extracted that discriminates well between healthy and tumorous tissue, and all scores are displayed as a color map. Ideally the scores can even be interpreted as the probability that the respective spectrum corresponds to a tumor. While such a decision support system may not completely obviate the need of manual inspection of the spectra, it can at least guide the radiologist towards suspicious regions that should be examined more closely, and facilitate the comparison with other imaging modalities.

Methods for computing the classification scores fall into two categories: quantitation-based approaches (Poullet et al, 2008) and pattern recognition-based approaches (Hagberg, 1998). Quantitation approaches exploit the fact that MRSI signals are physically interpretable as superpositions of metabolite spectra; they can hence be used to quantify the local relative concentrations of these metabolites by fitting measured or simulated basis spectra to the spectrum in every voxel. The fitting parameters (amplitudes, frequency shifts, ...) may be regarded as a low-dimensional representation of the signal. Classification scores are then usually computed from amplitude ratios of relevant metabolites: for instance, the choline/creatine and choline/NAA ratios are frequently employed for the diagnosis of brain tumors (Martínez-Bisbal and Celda, 2009).

Pattern recognition approaches forego an explicit data model: instead, the MRSI signal is preprocessed to a (still high-dimensional) feature vector, and the mapping of feature vectors to classification scores is learned from manually annotated training vectors (the so-called *supervised learning* setting). Because of this need for manually annotated examples, pattern recognition techniques require higher effort from human experts than quantitation-based techniques. Furthermore, they have to be retrained if the experimental measurement conditions change (e.g. different magnetic field strength, different imaged organ or different measurement protocol). However, comparative studies of quantitation and pattern recognition methods for prostate tumor detection showed superior performance of the latter ones, as they are more robust against measurement artifacts and noise (Kelm et al, 2007). Given a sufficiently large and diverse training data set, one can even use pattern recognition to distinguish between different tumor types, e.g. astrocytomas and glioblastomas (Tate et al, 2006).

MRSI data often have quality defects that render malignancy assessment difficult or even impossible: low signal-to-noise ratio, line widening because of shimming errors, head movement effects, lipid contamination, signal bleeding, ghosting etc. (Kreis, 2004). If these defects become sufficiently grave, even pattern recognition methods cannot tolerate them, and the resulting classification scores will be clinically meaningless and should not be used for diagnosis. Fig. 1 shows example spectra of good, poor, and very poor (not evaluable) quality for healthy, undecided and tumorous tissue. One can deal with this problem by augmenting the classification score for the malignancy (also called

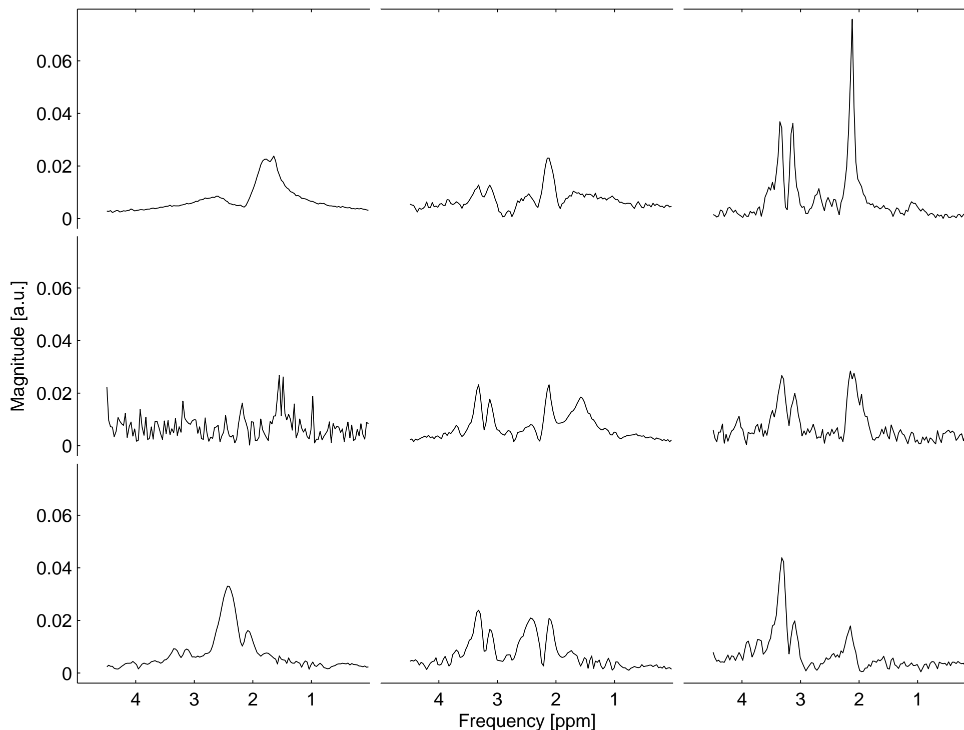

Fig. 1 Exemplary MRSI magnitude spectra of the brain, showing different voxel classes and signal qualities. All spectra have been water-suppressed and L_1 normalized (i.e. divided by the sum of all channel entries), and they are displayed on a common scale. Note the three distinct metabolite peaks, which are characteristic for brain MRSI: Choline (3.2 ppm), creatine (3.0 ppm) and N-acetylaspartate (NAA, 2.0 ppm). NAA is a marker for functional neurons, hence it has a high concentration in healthy tissue, and a low concentration in tumor tissue. On the other hand, choline is a marker for membrane biogenesis and has a higher concentration in tumor tissue than in healthy tissue. Left column: Spectra that are not evaluable owing to poor SNR or the presence of artifacts. Middle column: Spectra with poor signal quality, which however have sufficient quality so that the voxel class may be ascertained. Right column: Spectra with good signal quality. Top row: Spectra from healthy brain tissue. Middle row: Spectra of undecided voxel class. Bottom row: Spectra from tumor tissue. Note that the voxel class is only meaningful for the middle and the right column, and that the spectra in the left column were randomly assigned to the different rows.

voxel class) with a second score for the signal quality: If this score is high, the users know that the spectrum has high quality and that the voxel class score is reliable, while for a low score they know that the voxel class score is unreliable and the spectrum should be ignored. This may also save the users' time, as poor-quality spectra need not be examined in detail. Pattern recognition approaches have been successfully employed for signal quality prediction, with similar performance to expert radiologists (Menze et al, 2008).

1.2 Comparison to existing software

Most existing software products for MRSI classification incorporate quantitation-based algorithms: for instance, they are typically included in the software packages supplied by MR scanner manufacturers. Furthermore, there are several stand-alone software products such as LCModel (Provencher, 2001), jMRUI (Stefan et al, 2009) or MIDAS (Maudsley et al, 2006).

In contrast, the application of pattern recognition-based methods still has to gain ground in clinical routine: We believe that this may be partially due to differences in the flexibility with which both categories of algorithms can be adjusted to different experimental conditions (e.g. changes in scanner hardware and in measurement protocols) or to a different imaged organ. For quantitation-based methods one must only update the metabolite basis spectra to a given experimental setting, which can be achieved by quantum-mechanical simulation, e.g. with the GAMMA library (Smith et al, 1994). For pattern recognition-based methods on the other hand, one has to provide manual labels of spectra from many different patients with a histologically confirmed tumor, which is time-consuming and requires the effort of one or several medical experts. Since there exist many different techniques whose relative and absolute performance on a given task cannot be predicted beforehand, for every change in conditions a benchmarking experiment as in (Menze et al, 2006) or (García-Gomez et al, 2009) should also be conducted to select the best classifier and monitor the classification quality.

While we cannot obviate the need for classifier retraining, benchmarking and quality assessment, we have designed

an object-oriented C++ library and a graphical user interface which assists this task better than existing software. Our work is an extension of the CLARET software (Kelm et al, 2006): While the original prototype of this software was written in MATLAB, we improved upon a C++ reimplementation for the MeVisLab[1] environment. Most of the functionality described in this paper does not exist in the original CLARET version and is hence novel: mainly the possibility to manually define labels and to train, test, evaluate and compare various classifiers and preprocessing schemes.

There are two other alternative software products which employ pattern recognition methods for the analysis of MRSI spectra: HealthAgents by González-Vélez et al (2009) and SpectraClassifier by Ortega-Martorell et al (2010). What sets our software apart from these two systems, is the capability to statistically compare various different classifiers and to select the best one. SpectraClassifier provides statistical analysis functionalities for the trained classifiers, but linear discriminant analysis is the only available classification method. On the other hand, HealthAgent supports different classification algorithms but does not provide statistical evaluation functionality. Our manual annotation interface is also unique: this enables the users to adapt the classifiers flexibly to their (possible customized) measurement protocols.

Extensibility was an important design criterion for our library: by providing abstract interfaces for classifiers, data preprocessing procedures and evaluation statistics, users may plug in their own classes with moderate effort. Hereby it follows similar ideas as general purpose classification frameworks such as Weka[2], TunedIT[3] or RapidMiner[4]. However, it is much more focused in scope and tailored towards medical diagnostic applications. Furthermore, a similar plug-in concept for the analysis of MRSI data was used by Neuter et al (2007), but with a focus on quantitation techniques as opposed to pattern recognition techniques, and also lacking statistical evaluation functionalities.

2 Software architecture

2.1 Overview and design principles

Our software is designed for the following use case: the users label several data volumes with respect to voxel class (tumor vs. healthy) and signal quality and save the results (fig. 2). They specify several classifiers to be compared, the free classifier-specific parameters to be adjusted in parameter optimization (see fig. 3) and preprocessing steps for the data. A training and test suite is then defined, which may contain the voxel class classification task, the signal quality classification task, or both. The users may partition all data volumes explicitly into a separate training and testing set, otherwise a cross-validation scheme is employed: the data is partitioned into several folds, and the classifiers are iteratively trained on all but one folds, and tested on the remaining fold. The latter option is advisable if only few data are available; it has the additional advantage that means and variances for the classifier results may be estimated.

Every classifier is assigned to a preprocessing pipeline, which transforms the observed spectra into training and test features. Some elements of this pipeline may be shared across several classifiers, while others are specific for one classifier. Input data (spectra and labels) are passed, preprocessed and partitioned into cross validation folds if no explicit test data are provided. The parameters of every classifier are optimized either on the designated training data or on the first fold by maximizing an estimate for the generalization error. The classifiers are then trained with the final parameter values, and performance statistics are computed by comparing the prediction results on the current test data with the actual test labels. Statistical tests are conducted to decide whether the classifiers differ significantly in performance. Typically not only two, but multiple classifiers are compared against each other, which must be considered when judging significance. Finally the classifiers are retrained on the total data for predicting the class of unlabeled examples. The user may perform quality control in order to assess if the performance statistics are sufficient for employment in the clinic (fig. 4). The trained classifiers may then be loaded and applied to new data sets, for which no manual labels are available (fig. 5).

Our main design criteria were extensibility, maintainability and exception safety. Extensibility was achieved by providing abstract base classes for classifiers, preprocessing procedures and evaluation statistics, so that it is easily possible to add e.g. new classification methods by deriving from the appropriate class. For maintainability, dedicated manager objects handle the data flow between the different modules of the software and maintain the mutual consistency of their internal states upon changes made by the user. Strong exception safety guarantees are necessitated by the quality requirements for medical software; it was achieved by the techniques described in (Stroustrup, 2001).

2.2 The classification functionality

The design of the classification functionality of our library follows the main aim of separating between classifier-specific functionality (which must be provided by the user when introducing a new classifier) and common functionality that is used by all classifiers and does not need to be changed: the

[1] http://www.mevislab.de
[2] http://www.cs.waikato.ac.nz/ml/weka/
[3] http://tunedit.org/
[4] http://www.rapid-i.com

Fig. 2 User interface for the labeling functionality of our data, showing an exemplary data set acquired at a 3 Tesla Siemens Trio scanner. Top left: Corresponding morphological data set in sagittal view (T_2-weighted turbo spin-echo sequence in this case). We can place a marker (blue) to select a voxel of interest. Middle left: Magnitude spectrum of the selected voxel, which is typical for a cerebral tumor. Top right: Selected voxel (framed in red) together with the axial slice in which it is contained. The user-defined labels are overlayed over a synopsis of all spectra in the slice. The label shape encodes the signal quality (dot / asterisc / cross for "not evaluable" / "poor" / "good"), while the label color encodes the voxel class (green / yellow / red for "healthy" / "undecided" / "tumor"). The labels may also be annotated by free-text strings. Bottom panel: User interface with controls for label definition, text annotation and data import / export.

`ClassifierManager` class is responsible for the former, while the classes derived from the abstract `Classifier` basis class are responsible for the latter. Simple extensibility and avoiding code repetition were therefore the two main design principles.

A `ClassifierManager` object corresponds to each classification task, e.g. classification with respect to signal quality and with respect to voxel class (see fig. 6). It controls all classifiers which are trained and benchmarked for this task, and ensures that operations such as training, testing, and the averaging of performance statistics over cross-validation folds as well as saving and loading are performed for each classifier. It also partitions the training features and labels into several cross-validation folds, if the users do not define a designated test data set.

A `Classifier` object encapsulates an algorithm for mapping feature vectors to discrete labels after training. Alternatively, the output can also be a continuous score that gives information about the confidence that a spectrum corresponds to a tumor. We implemented bindings for several linear and nonlinear classifiers, which previously had been found to be well-suited for the classification of MRSI spectra (Menze et al, 2006): support vector machines (SVMs) with a linear and a radial basis function (RBF) kernel, random forests (RF), ridge regression (RR) and principal components regression (PCR); see (Hastie et al, 2009) for a description of these methods. The actual classification algorithms are provided by external libraries such as LIBSVM (Chang and Lin, 2001) and VIGRA (Köthe, 2000).

Both binary classification (with two labels) as well as multi-class classification (with more than two labels) are supported. Some classifiers (e.g. random forests) natively support multi-class classification, while for other classifiers

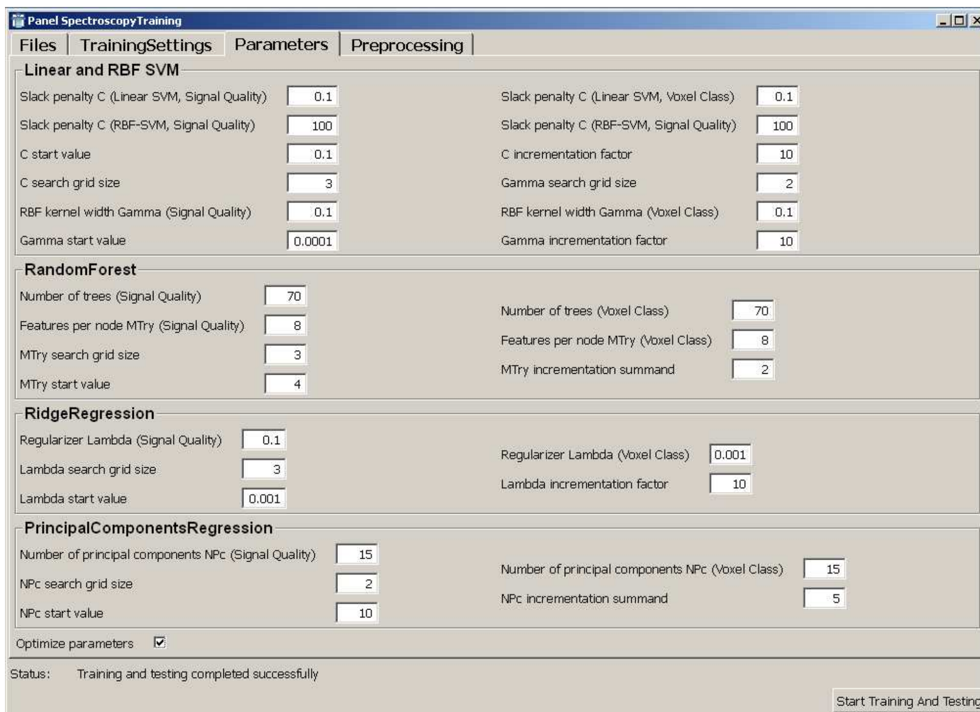

Fig. 3 Part of the user interface for classifier training and testing. In this panel, the search grids for automated parameter tuning of the different classifiers may be defined (default values, starting values, incrementation step sizes and numbers of steps).

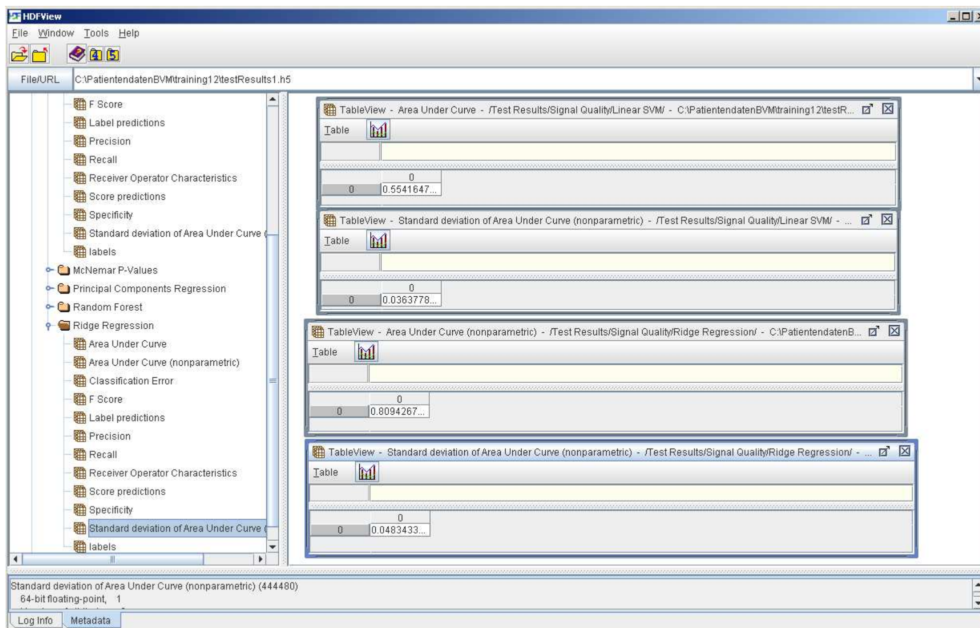

Fig. 4 Evaluation results for an exemplary training and testing suite. The upper two windows on the right-hand side show the estimated area under curve value for a linear support vector machine classifier and its estimated standard deviation (0.554 ± 0.036), while the lower two windows show the same values for a ridge-regression classifier (0.809 ± 0.048). This would allow a clinical user to draw the conclusion that only the latter one of these classifiers differs significantly from random guessing, and may sensibly be used for diagnostics. The poor quality of these classifiers is due to the fact that only a very small training set was used for the purpose of illustrating the user interface design (2 patients).

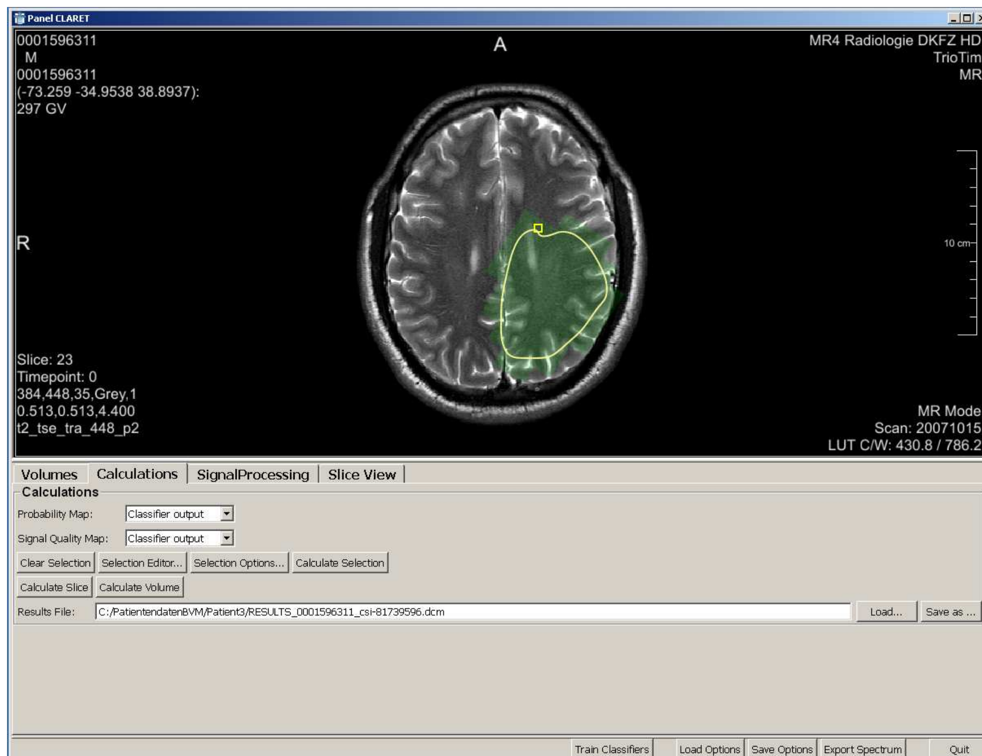

Fig. 5 Exemplary application of a trained classifier for the computer-assisted diagnosis of a new data set. The classifier predictions for both voxel class and signal quality are depicted for a user-defined region of interest: the voxel class is encoded by the color (green for "healthy", yellow for "undecided", red for "tumor"), while the signal quality is encoded by the transparency (opaque for a good signal, invisible for a spectrum which is not evaluable). As an alternative to the classifier predictions, it is possible to display precomputed color maps as well as color maps based on the parametric quantitation of relevant metabolites.

(e.g. ridge regression and principal components regression[5]), it can be achieved via a *one-vs.-all* encoding scheme[6], in which each class is classified against all other classes in turn, and the class with the largest score is selected for the prediction (Rifkin and Klautau, 2004). This multi-class functionality allows the future extension of our library to the task of discriminating different tumor types against each other.

Furthermore, every classifier encapsulates an instance of the `ClassifierParameterManager` class controlling the parameter combinations that are tested during parameter optimization. Most classifiers have one or more internal parameters that ought to be optimized for each data set in order to achieve optimal predictive performance (see sec. 2.4). This is done by maximizing an estimate of the *generalization error* (i.e. the performance of the classifier on new test data that were not encountered during the training process) over a prescribed search grid, using the data from one of the cross-validation folds (or the whole training data, if no cross-validation is used). This generalization error could be estimated by dividing the training data into another training and test fold, training the classifier on the training part of the training data and testing it on the testing part of the training data[7]. However, this would be time-consuming. However, there exists considerable theoretical as well as empirical evidence (Golub et al, 1979; Breiman, 1996) that efficiently computable approximations for the generalization error may be sufficient for parameter adjustment: these are provided by the function `estimatePerformanceCvFold()`. For SVMs, this is an internal cross-validation estimate as described in (Lin et al, 2007), for random forests, the *out-of-bag error* and for regression-based classifiers the *generalized cross-validation* (Hastie et al, 2009). The optimal parameters are selected by the function `optimizeParametersCvFold()` based on the data from one specific cross-validation fold.

This part of the library may be easily extended by adding new classifiers, as long as they fit into the supervised classification settings (i.e. based on labeled training vectors, a function for mapping these vectors to the discrete labels is

[5] To be precise, these two classifiers are actually regression methods and can be used for binary classification by assigning the label +1 and -1 to all positive and negative class examples and training a regressor. The `transformLabelsToBinary()` function maps the original labels to these two numbers.

[6] The virtual `isOnlyBinary()` function allows one to specify the affiliation of a classifier to these two categories.

[7] Note that the actual test data must not be used during parameter tuning.

learnt). Artificial neural networks, boosted ensemble classifiers or Gaussian process classification are examples for alternative classification algorithms that could be added in this way. For this, one only needs to derive from the `Classifier` abstract base class and to provide implementations for its abstract methods (including the definition of the `Preprocessor` subclass with which this classifier type is associated). For parameter tuning, one also has to supply an estimate of the classifier accuracy: This may always be computed via cross-validation, but preferably this estimate should arise as a by-product of the training or be fast to compute (same as e.g. the out-of-bag error for the random forest or the generalized cross-validation). Furthermore we assume the existence of a continuous classification score, which ideally can be interpreted as a tumor probability. However, for classifiers without such a probabilistic interpretation it is sufficient to reuse the 0/1 label values as scores: as long as higher scores correspond to a higher likelihood for the positive (tumor) class, they can take any values. We only use the single-voxel spectra for classification, hence our architecture does not allow classifiers that make explicit use of spatial context information (so-called *probabilistic graphical models*).

2.3 The preprocessing functionality

Preprocessing (fig. 7) is the extraction of a feature vector from the raw MRSI spectra with the aim of improved classification performance. While classification makes use of both the label and the feature information (supervised process), preprocessing only uses the feature information (unsupervised process). `Preprocessor` objects may act both on the total data (`transformTotal()`) and of the data of a single cross-validation fold (`transformCvFold()`): the distinction may be relevant since some preprocessing steps (e.g. singular value decomposition) depend on the actual training data used.

The main goal governing the design of the preprocessing functionality was training speed: data preprocessing steps which are common to multiple classifiers should only be performed once. Hence the different preprocessing steps are packaged into modules (deriving from the `Preprocessor` abstract base class) and arranged into cascades. A common `PreprocessorManager` ensures that every preprocessing step is only performed once. Hiding the preprocessing functionality from the library users was an additional criterion: Every subclass of `Classifier` is statically associated with a specific `Preprocessor` subclass and is responsible for registering this subclass with the `PreprocessorManager` and passing the data to be preprocessed.

First, since we are only interested in the metabolite signals, the nuisance signal caused by water molecules has to be suppressed, using e.g. a Hankel singular value decomposition filter (Zhu et al, 1997). Then the spectra are transformed from the time domain into the Fourier domain by means of the FFTW library (Frigo and Johnson, 2005). The subsequent steps may be adjusted by the user, and typically depend on the classifier:

Common MRSI preprocessing steps used by all classifiers are the rebinning of spectral vectors, the extraction of parts of the spectrum and L_1 *normalization* (i.e. the spectral vector is normalized such that the sum of all component magnitudes in a prescribed interval equals one): these are performed by the class `MrsiPreprocessor`.[8] Other preprocessing steps are only relevant for some of the classifiers, e.g. the `RegressionPreprocessor` performs a singular value decomposition of the data which speeds up subsequent ridge regression or PCR. SVMs perform better when the features have zero mean and unit variance: this can be achieved by the `WhiteningPreprocessor`.

Two features of our software implementation support this modular structure: The `PreprocessorManager` incorporates a class factory, which ensures that only one instance of each preprocessor class is created: this allows to share preprocessors across various classifiers and prevents duplicate preprocessing steps (such as e.g. performing the singular value decomposition twice on the same data). Furthermore, preprocessors are typically arranged in a tree structure (via the `predecessor` and `successors` references) and every classifier is assigned to one vertex of this tree, which ensures that all preprocessing steps on the path from the root to this vertex are applied in order (creating a pipeline of preprocessing steps). Once the data encapsulated inside one module changes, all successors are invalidated.

When new classifiers are added to the library, the preprocessing part may easily extended with new preprocessor modules as long as they fit into the unsupervised setting (i.e. they only make use of the features, but not of the labels). Besides implementing the abstract methods of the `Preprocessor` base class, the association between the classifier and the preprocessor must be included in the classifier definition by implementing its `getPreprocessorStub()` method: then the classifier object ensures that the new preprocessor is correctly registered with the preprocessor manager object. As a limitation, the new preprocessor has to be appended as a new leaf (or a new root node) to the preprocessor tree: the intermediate results from other preprocessing steps can only be reused if the order of these steps is not changed.

2.4 The parameter tuning functionality

All classifiers have adjustable parameters, which are encapsulated in the `ClassifierParameter` class (fig. 8). The

[8] More sophisticated steps such as the extraction of wavelet features might be added as well.

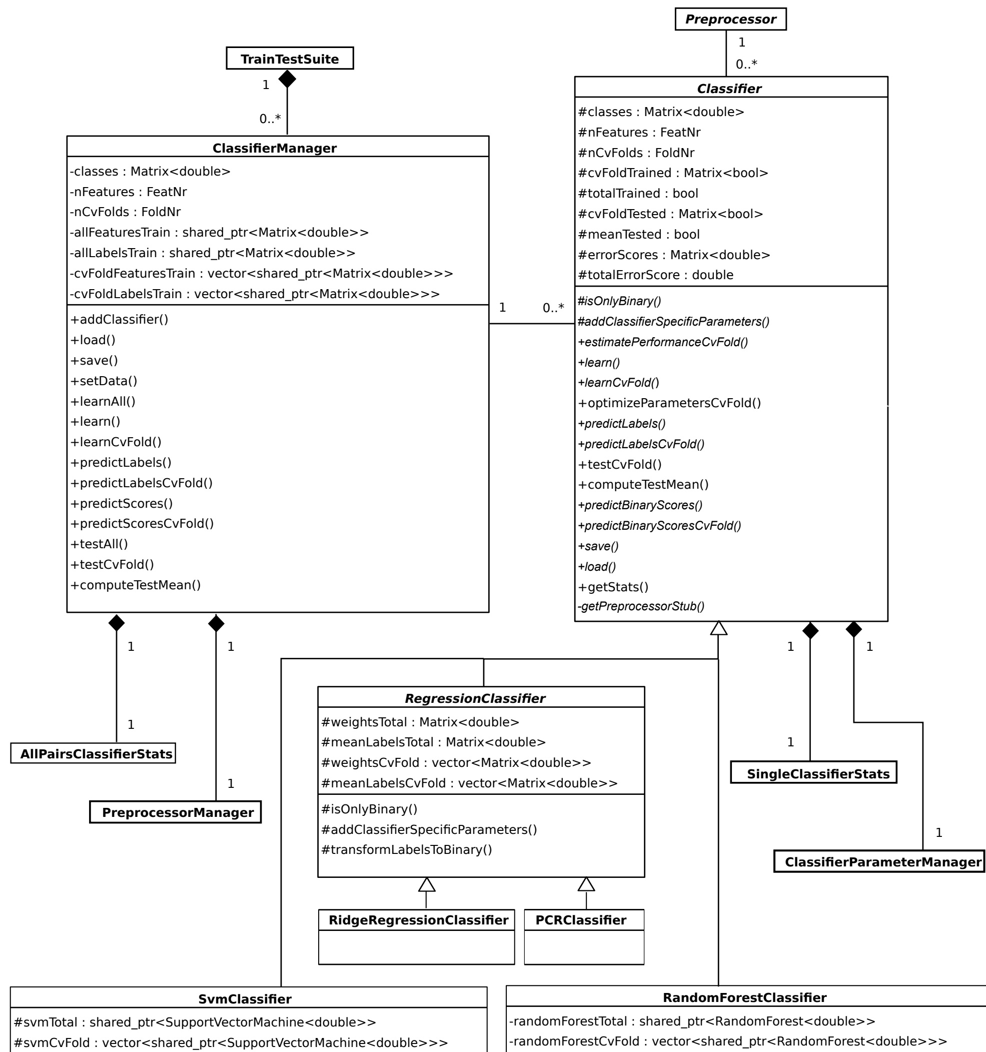

Fig. 6 Simplified UML diagram of the classification functionality of our library: detailed explanations can be found in section 2.2. The connections to the classes `TrainTestSuite` (see fig. 10), `Preprocessor` / `PreprocessorManager` (fig. 7), `ClassifierParameterManager` (fig. 8) and `SingleClassifierStats` / `AllPairClassifierStats` (fig. 9) are shown. In this diagram, as in the following ones, abstract methods are printed in italics: to save space, we do not show the implementations of these abstract methods if they are provided in the leaves of the inheritance tree. The depiction here is simplified: we actually follow the non-virtual interface principle and give protected visibility to all abstract methods, which are then encapsulated by non-virtual public methods.

design of the parameter handling functionality was guided by the main rationale of handling parameters of different datatypes in a uniform way. Furthermore we aimed to enable automated parameter adjustment over a search grid (which may have linear or logarithmic spacing depending on the range of reasonable parameter values), by hiding the details of the search mechanism from the class users.

Some parameters should be optimized for the specific classification task, as described in section 2.2: for the classifiers supplied by us, these are the slack penalty C for SVMs, the kernel width γ for SVMs with an RBF kernel, the random subspace dimension m_{try} for random forests, the number of principal components n_{PC} for PCR and the regularization parameter λ for ridge regression. They are represented as a `TypedOptimizableClassifierParameter`: besides the actual value, these objects also contain the search grid of the parameters, namely the starting and end value, the incrementation step and whether the value should be incremented additively or multiplicatively (encoded in the field `incrInLogSpace`. Multiplicative updates are appropriate for parameters that can span a large range of reasonable values.

There are also parameters which may not be optimized: these are encapsulated as a `TypedClassifierParameter`, which only contains the actual value. A good example would be the number of trees of a random forest classifier, since the generalization error typically saturates as more trees are added.

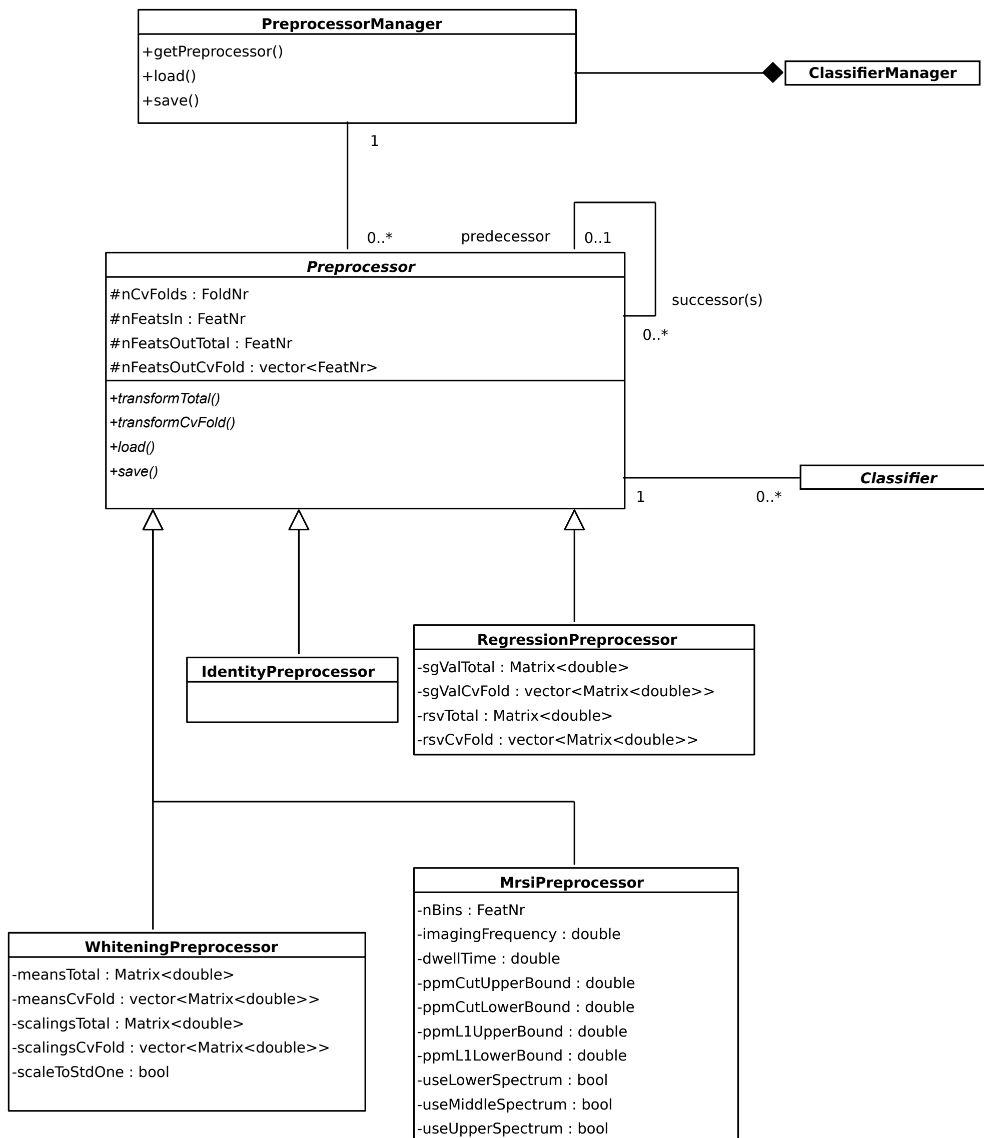

Fig. 7 Simplified UML diagram of the preprocessing functionality; see section 2.3 for details. The connections to the classes `Classifier` and `ClassifierManager` (fig. 6) are shown.

While all currently used parameters are either integers or floating-point numbers, one can define parameters of arbitrary type: however, one has to define how this data type can be written to or retrieved from a file or another I/O medium by implementing the corresponding I/O callbacks (see section 2.6 for detailed explanation). For optimizable parameters, it must also be defined what it means to increase the parameter by a fixed value (by overloading the `operator++()` member function). As a limitation, we assume that all parameters may be varied completely independently and cannot encode constraints coupling the values of multiple parameters.

One should note that the parameter optimization process followed by our library is exactly the way a human expert would do it: in the absence of universal theoretical criteria about the choice of good parameters, they have to be tuned empirically so that a low generalization error is achieved. [9] However, this is the most time-consuming part of adapting a classifier to a new experiment, which is now completely automated by our software.

[9] If sufficient data were available, it would be preferable to perform this parameter tuning on a separate tuning data set that is not used in the training and testing of the classifier. Since typically clinics only have access to few validated MRSI data, this approach may not be practicable, and the cross-validation scheme used in our library is the best alternative to deal with scarce data.

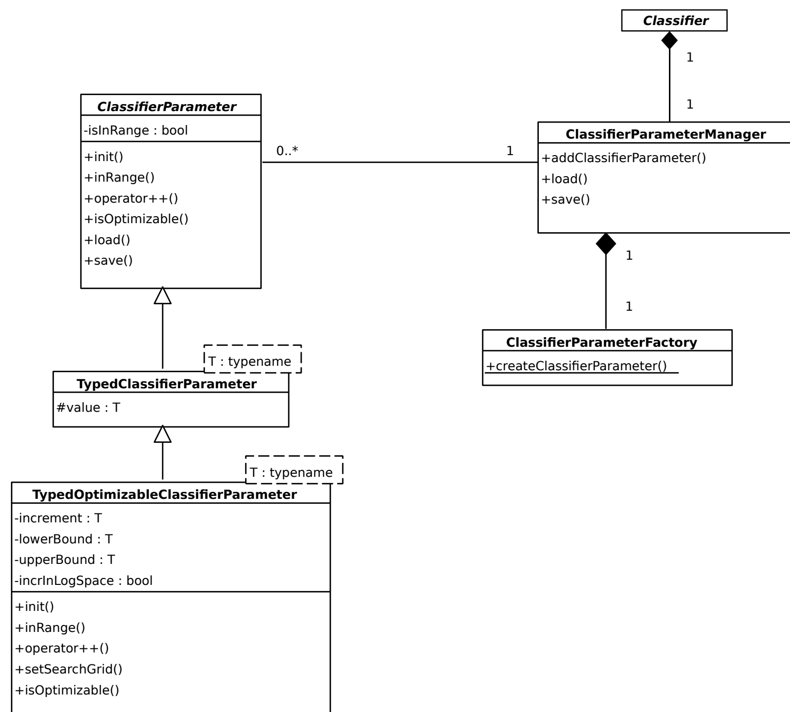

Fig. 8 Simplified UML diagram of the parameter tuning functionality; see section 2.4 for details. The connection to the class `Classifier` (fig. 6) is shown.

2.5 The statistics functionality

The computation of evaluation statistics is crucial for the automated quality control of trained classifiers (fig. 9). We designed this part of the library with the following aims in mind: Needless recomputation of intermediate values should be avoided; thus we compute the binary confusion matrix only once and then cache it within a `StatsDataManager` object, which can be queried for computing the different statistics derived from it (e.g. `Precision` and `Recall`). The library can be simply extended by new statistics characterizing a single classifier. Dedicated manager classes (such as `SingleFoldStats`, `SingleClassifierStats` as well as `PairClassifierStats` and `AllPairsClassifierStats`) are each responsible for a well-defined statistical evaluation task: namely, characterizing a classifier for a single cross-validation fold, characterizing a classifier over all folds, characterizing a single pair of classifiers and characterizing all existing pairs of classifiers. They ensure that this computation is performed in a consistent way for all classifiers, so that code redundancy is avoided.

The class `SingleClassifierStats` manages all statistics pertaining to one single classifier: it is composed of objects of type `SingleFoldStats`, which in turn manage all statistics either of a single cross-validation fold (`cvData`), or the mean and standard deviation values computed over all folds (`meanData`). A `StatsDataManager` is a helper class which caches several intermediate results required for the computation of the different `Statistics`.

There are different variants of how these statistics may be computed in a multi-class classification setting: some of them (e.g. the `MisclassificationRate`) can handle multiple classes natively; these statistics form the derived class `AllVsAllStat`. Other statistics (e.g. `Precision`, `Recall` or `FScore`) were originally designed for a binary classification setting. For the latter kind, one must report multiple values, namely one for each class when discriminated against all others (one-vs.-all encoding), and they inherit from the `OneVsAllStat` class. The `AreaUnderCurve` (AUC) value of the receiver operating characteristic (ROC) curve (Fawcett, 2006) is a specialty: while it is also computed in a one-vs.-all fashion, the underlying ROC curves are stored as well. Standard deviation estimates are mostly available only for the `meanData` averaged over several cross-validation folds, with the exception of the AUC values for which nonparametric bootstrap estimates can be easily computed (Bandos et al, 2007).

Besides the statistical characterization of single classifiers, it is also relevant to compare pairs of classifiers in order to assess which one of them is best for the current task, and whether the differences are statistically significant. The `AllPairsClassifierStats` class manages the statistics characterizing the differences in misclassification rate between all pairs of classifiers, each of which is represented by a single `PairClassifierStats` instance. We report *p*-

values computed by statistical hypothesis tests with the null hypothesis that there is no difference between classifier performances. We provide implementations for two tests: McNemar's test (Dietterich, 1998) is used when the data are provided as a separate training and test set, while a recently proposed conservative t-test variant (Grandvalet and Bengio, 2006) is used if the users provide only a training data set, which is then internally partitioned into cross-validation folds. The latter test assumes that there is an upper border on the correlation of misclassification rates across different cross-validation folds, which is stored in the variable `maxCorrelationGrandvalet`[10].

If we have more than two classifiers, we must adjust the p-values for the effect of multiple comparisons: In the case of five classifiers with equal performance, we have ten pairwise comparisons and a significant difference ($p_{\text{raw}} < 0.001$) is expected to occur with a probability of $1 - 0.999^{10} \approx 0.01$. After computing all "raw" p-values, we correct them using Holm's step-down or Hochberg's step-up method (Demšar, 2006) and store all results as `PValue` structures.

If there is need to extend the statistics functionality, it is simple to add any statistic characterizing a single classifier that can be computed from the true labels and the predicted labels and scores, as these values may be queried from the `StatsDataManager` object. To our knowledge, this comprises all statistics which are commonly used for judging the quality of general classification algorithms. As a limitation, the evaluation statistics cannot use any information about the spatial distribution of the labels: hence it is impossible to compute e.g. the Hausdorff distance between the true and the predicted tumor segmentation. Among the statistical significance tests (like `McNemarPairClassifierStat`), one can add any technique that only requires the mean values of the statistic to be compared from each cross-validation fold. The current design is not prepared for new methods of multi-comparison adjustment beyond Holm's or Hochberg's method: for every method acting only on p-values and computing an adjusted p-value, this would be possible, but requires moderate redesign of this part of the library. We also have hard-wired the assumption that the mean and variance of these evaluation shall be estimated using a cross-validation scheme. The number of cross-validation folds can be specified at the `ClassifierManager` level: It is theoretically possible to run a leave-one-out validation scheme with this machinery, but that would lead to prohibitive computation times.

[10] Note that a classical t-test may not be used, since the variance of misclassification rates is estimated from cross-validation and hence systematically underestimated. (Bengio and Grandvalet, 2004) showed that unbiased estimation of the variances is not possible; but the procedure used here provides an upper bound on the p-value if the assumptions are fulfilled.

2.6 The input / output functionality

We designed the input / output functionality in order to keep it separated from the modules responsible for the internal computations: hence we pass function objects to the classifier, preprocessor etc. objects, which can then be invoked to serialize all types of the data that is encapsulated by these objects. Similar function objects are used for streaming relevant information outside and listening for user signals at check points.

For persistence, classifiers, preprocessors, statistics and all other classes with intrinsic state can be saved and reloaded in a hierarchical data format, and the data input/output can be customized by passing user-defined input and output function objects derived from the base classes `LoadFunctor` and `SaveFunctor` (see fig. 10). For these function objects, the user must define how to enter and leave a new hierarchy level (`initGroup()` and `exitGroup()`) and how to serialize each supported data type (`save()` and `load()`): for the latter purpose, the function objects must implement all required instantiations of the `LoadFunctorInterface` or `SaveFunctorInterface` interface template. We exemplarily provide support for HDF5[11] as the main storage format (XML would be an obvious alternative). For integration into a user interface, other function objects may be passed that can either report progress information, e.g. for updating a progress bar (`StreamProgressFunctor`), or report status information (`StreamStatusFunctor`) or listen for abort requests (`AbortCheckFunctor`) at regular check points. A `ProgressStatusAbortFunctor` bundles these three different functions. The `TrainTestSuite` manages the actions of the library at the highest level: the library users mainly interact with this class by adding classifier manager objects, passing data and retrieving evaluation results.

The I/O functionality can simply be extended to other input and output streams, as long as the data can be stored in a key-value form with string keys, and as long as a hierarchical structure with group denoted by a name string can be imposed. Instead of only listening for abort signals, the `AbortCheckFunctor` could in principle handle more general user requests: but aborting a time-consuming training process is presumably the main requirement for user interaction capabilities.

2.7 User interaction and graphical user interface

In order to further aid the clinical users in spectrum annotation, we developed a graphical user interface in MeVisLab that displays MRSI spectra from a selected slice in the context of its neighbor spectra, which can then be labeled on an ordinal scale by voxel class and signal quality and

[11] http://www.hdfgroup.org/HDF5/

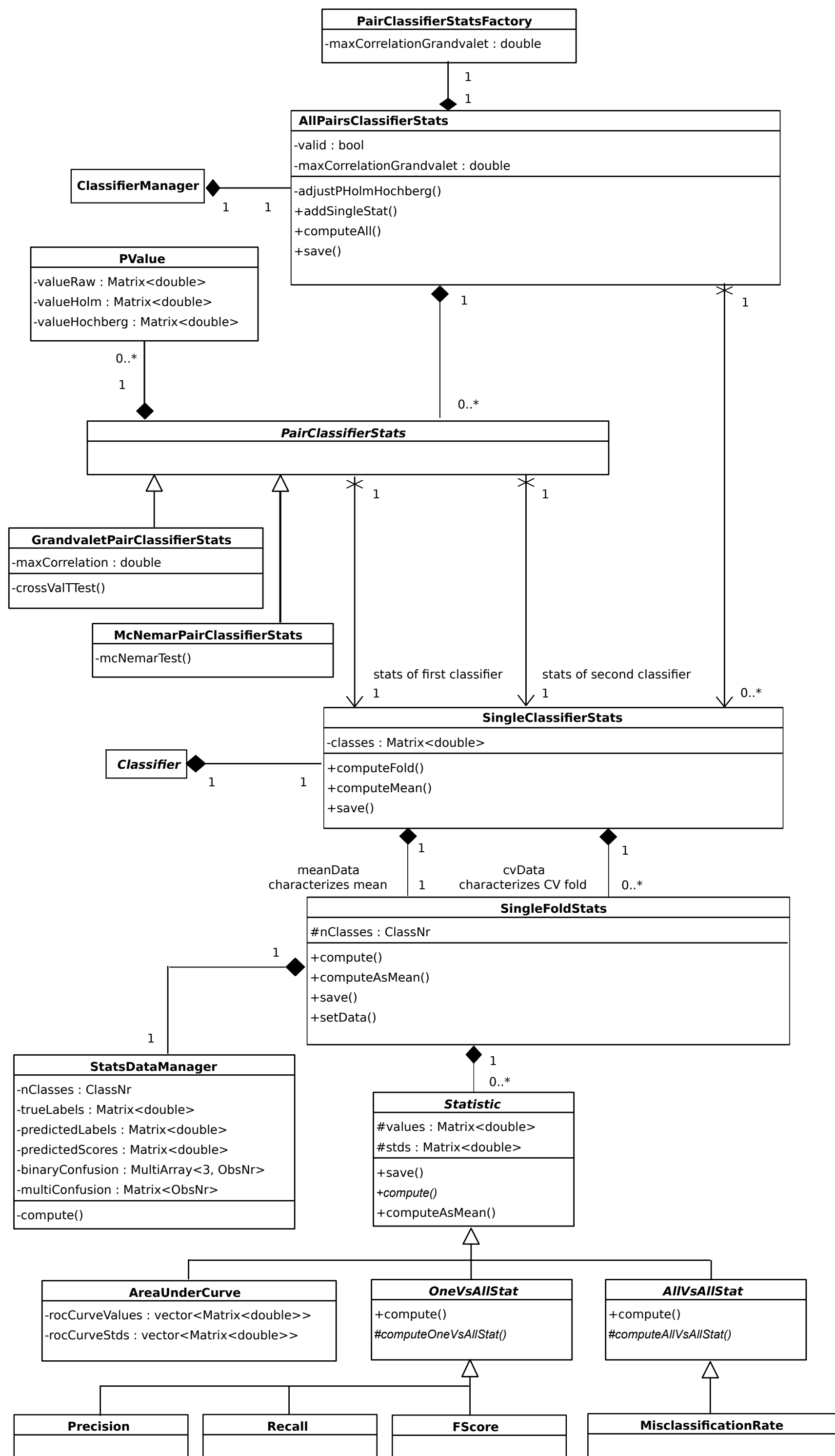

Fig. 9 Simplified UML diagram of the statistical evaluation functionality; see section 2.5 for details. The connections to the classes `Classifier` and `ClassifierManager` (fig. 6) are shown.

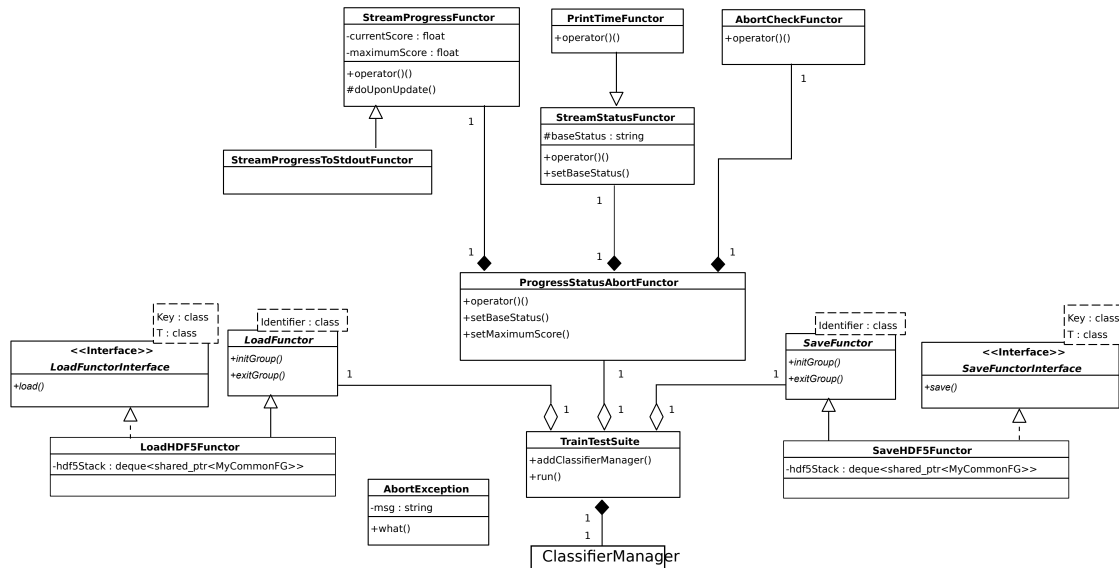

Fig. 10 Simplified UML diagram of the data input / output functionality; see section 2.6 for details. The connection to the class `ClassifierManager` (fig. 6) is shown.

imported into the classification library (fig. 2). Since clinical end users only interact with this user interface, they can start a training and testing experiment and evaluate the results without expert knowledge on pattern recognition techniques: they only have to provide their domain knowledge about the clinical interpretation of MRSI data. To this purpose, our graphical user interface displays the MRSI spectra of the different voxels both in their spatial context (upper right of fig. 2) and as enlarged single spectra (middle left of this figure). It is known that the ability to view MRSI spectra in their surroundings and to incorporate the information from the neighboring voxels is one of the main reasons why human experts still perform better at classifying these spectra than automated methods (Zechmann et al, 2010). Simultaneously one can display a morphological MR image that is registered to the MRSI grid, which can give additional valuable information for the labeling process of the raters. Labels are provided on two axes (signal quality and voxel class / malignancy) that are encoded by marker shape and color; furthermore it is possible to add free-text annotations to interesting spectra.

After saving the label information in a human-readable text format, clinical users only have to provide the information which label files (and associated files with MRSI data) shall be used for training and testing. (As stated in section 2.6, it is not required to specify dedicated testing files; in this case, all data are used in turn for both training and testing via a hold-out scheme.) An expert mode provides the opportunity to select which classifiers to train and test and to set the classifier parameters manually (fig. 3). We also propose default values for these parameters, which gave the best or close to the best accuracy on different prostate data sets acquired at 1.5 Tesla (table 1): these values can at least serve as plausible starting values for the parameter fine tuning on new classification tasks. Alternatively a search grid of parameter values may be specified, so that the best value is detected automatically: this allows to improve the classifier accuracy in some cases, while still requiring little understanding about the detailed effects of the different parameters on the side of the users.

Besides the weights of the trained classifiers, the training and testing procedures also generates test statistics that are estimated from the cross-validation schemes and saved in the HDF5 file format. By inspecting these files, one can get a detailed overview over the accuracy and reliability of the different classifiers and compare whether they yield significantly different results (fig. 4).

Finally, the trained classifiers can be applied to predict the labels of new MRSI spectra for which no manual labels are available. For a user-selected region of interest, this information can be displayed in the CLARET software as an easily interpretable nosologic map overlayed over the morphological MR image (fig. 5). The voxel class is encoded in the color (green for healthy tissue, red for tumor, yellow for undecided cases), while the signal quality is encoded in the alpha channel (for poor spectra the nosologic map is transparent, whereas for very good spectra it is nearly opaque).

3 Case studies

3.1 Exemplary application to 1.5 Tesla data of the prostate

As a case study, we recapitulate the results from a previous study, in which we validated our library on 1.5 Tesla MRSI data of prostate carcinomas (Kaster et al, 2009). We used two different data sets for the training of signal quality and of voxel class classifiers: For signal quality classification, we provided 36864 training spectra and 45312 test spectra and extracted 101 magnitude channels as features during preprocessing (data set 1). For joint signal quality and voxel class classification, we provided 19456 training spectra from 24 patients, from which however only the 2746 spectra with "good" signal quality were used for learning the voxel class classifiers: we extracted 41 magnitude channels as features (data set 2). Since relatively few spectra were available for the voxel class classification task, we opted for an eight-fold cross-validation scheme rather than partitioning the data into a separate training and test set. No preprocessing steps besides rebinning and selection of the appropriate part of the spectrum were used.

Parameter (classifier)	Search grid values	Final values for DS1 (SQ) / DS2 (SQ) / DS2 (VC)
C (SVM)	$10^{-2}, 10^{-1}, \ldots, 10^{3}$	10^{1} / 10^{2} / 10^{2}
m_{try} (RF)	$4, 6, \ldots, 16$	16 / 14 / 16
λ (RR)	$10^{-3}, 10^{-2}, \ldots, 10^{2}$	10^{-1} / 10^{-1} / 10^{-2}
n_{PC} (PCR)	$10, 15, \ldots, 40$	40 / 35 / 25

Table 1 Search grid for automated classifier parameter selection and final values for signal quality (SQ) classification based on data set 1 (DS1) and signal quality and voxel class (VC) classification based on data set 2 (DS2).

As classifiers, we trained support vector machines with linear kernel, random forests, principal component regression and ridge regression, as the training of support vector machines with an RBF kernel was found to be too time-consuming. We used the automated parameter search capabilities of our library to find the optimal values that were finally used (see table 1).

	SVM	RF	RR	PCR
Precision	0.815	0.869	0.921	0.922
Recall	0.913	0.913	0.797	0.802
Specificity	0.972	0.982	0.991	0.991
F-score	0.861	0.891	0.855	0.857
CCR	0.965	0.973	0.968	0.968
AUC	0.989(14)	0.993(14)	0.990(14)	0.990(14)

Table 2 Evaluation statistics for signal quality classifiers based on data set 1. The standard deviation of the area under curve value (in parentheses) is estimated as by Bandos et al (2007). Note that the recall is also known as the "sensitivity".

	SVM	RF	RR	PCR
Precision	0.73(11)	0.832(57)	0.79(12)	0.79(12)
Recall	0.57(18)	0.58(17)	0.42(17)	0.43(17)
Specificity	0.964(23)	0.9820(62)	0.980(18)	0.979(19)
F-score	0.621(14)	0.67(13)	0.53(15)	0.54(16)
CCR	0.905(37)	0.922(32)	0.899(37)	0.899(38)
AUC	0.891(54)	0.946(57)	0.890(54)	0.890(54)

Table 3 Average evaluation statistics for signal quality classifiers based on data set 2 (with standard deviations in parentheses). While the standard deviation reported for the area under curve value is estimated as by Bandos et al (2007) to facilitate the comparison with table 2, the other standard deviation estimates are computed from the cross-validation.

	SVM	RF	RR	PCR
Precision	0.908(76)	0.864(27)	0.966(39)	0.900(14)
Recall	0.69(17)	0.753(16)	0.50(21)	0.50(21)
Specificity	0.983(23)	0.9771(87)	0.9966(39)	0.9928(78)
F-score	0.76(12)	0.79(11)	0.63(22)	0.63(21)
CCR	0.932(42)	0.937(42)	0.909(59)	0.909(62)
AUC	0.97(15)	0.98(15)	0.96(15)	0.95(15)

Table 4 Average evaluation statistics for signal quality classifiers based on data set 2 (see table 4 for further explanations).

With these input data, we achieved state-of-the art classification performance: For signal quality prediction on data set 1, the different classifiers achieved correct classification rates (CCR) of 96.5 % – 97.3 % and area under the ROC curve values of 98.9 % – 99.3 % (see table 2). On data set 2, we obtained correct classification rates of 89.9 % – 92.2 % and area under curve values of 89.0 % – 94.6 % for the signal quality prediction task (table 3), and correct classification rates of 90.9 % – 93.7 % as well as area under curve values of 95 % – 98 % for the voxel class prediction task (table 4).

The automated parameter tuning functionality is especially relevant for the use of support vector machines, since wrong values of the parameter C may lead to a considerably degraded accuracy. If e.g. the starting value of 0.01 for C had been used for the signal quality classification of data set 1, the correct classification rate would have dropped to 92.5 % (which means that the number of wrongly classified spectra would have doubled). The other classifiers that are currently available in our library are more robust with respect to the values of their associated parameters.

While these absolute quality measures are highly relevant for the clinical practitioners, a research clinician may also be interested in the question which classifier to use for this particular task (and whether there is any difference between the different classifiers at all). This question could be answered with the statistical hypothesis testing capabilities of our library, since p-values from McNemar's test (for data set 1) and the t-test variant (for data set 2) characterizing the differences in the correct classification rates of various clas-

sifiers were automatically computed and corrected for multiple comparisons (both Holm's step-down and Hochberg's step-up method yielded qualitatively the same results). For the signal quality classifiers trained on data set 1, random forests differed with high significance from all other classifiers ($p < 10^{-6}$). Support vector machines differed from principal components regression significantly ($p < 10^{-3}$), and ridge regression showed a barely significantly difference to both principal components regression and support vector machines ($p < 10^{-2}$), while all other differences were non-significant. For data set 2, all differences were non-significant owing to the small number of data examples.

3.2 Extending the functionality with a *k* nearest neighbors classifier

As an exemplary case of how the functionality of our library may be extended, we discuss the addition of a new classifier method in detail, namely the *k* nearest neighbors (kNN) method as one of the simplest classifiers (Hastie et al, 2009). Every test spectrum is assigned the majority label of its *k* closest neighbors among the training spectra (with respect to the Euclidean distance).[12] This classifier is represented by a `NearestNeighborClassifier` class derived from the abstract `Classifier` base class:

```
class EXPORT_CLASSTRAIN
NearestNeighborClassifier  : public Classifier {
private:
  // All training spectra
  vigra::Matrix<double> trainingSpectra;
  // All training labels
  vigra::Matrix<double> trainingLabels;
  // Training spectra for the different cross-validation folds
  std::vector<vigra::Matrix<double> > trainingSpectraCvFolds;
  // Training labels for the different cross-validation folds
  std::vector<vigra::Matrix<double> > trainingLabelsCvFolds;
  // Name strings associated with the kNN classifier
  static const std::string knn_name;
  static const std::string k_name;
  static const std::string cv_error_name;
  static const std::string training_spectra_name;
  static const std::string training_labels_name;
protected:
  // Can be used for native multi-class classification
  virtual bool isOnlyBinary() const {
    return false;
  }
public:
  // Stub constructor
  NearestNeighborClassifier() : Classifier(),
    trainingSpectra(), trainingLabels(),
    trainingSpectraCvFolds(), trainingLabelsCvFolds(){
  }
  // Read-only access to classifier name string
  virtual std::string getClassifierName() const {
    return knn_name;
  }
  // Read-only access to error score name string
  virtual std::string getErrorScoreName() const {
    return cv_error_name;
  }
protected:
  /* The following virtual functions are discussed separately */
  ...
};
```

The only adjustable parameter is the number of nearest neighbors k. By default, the odd values $1, 3, \ldots, 15$ shall be considered while optimizing over this parameter: they may also be adjusted afterwards by the library user. The last argument of the `addClassifierParameter` specifies that this parameter shall be incremented additively rather than multiplicatively.

```
void
NearestNeighborClassifier::
addClassifierSpecificParameters(){
  unsigned kValue=5;
  unsigned kLower=1;
  unsigned kUpper=15;
  unsigned kIncr=2;
  parameters->addClassifierParameter(k_name,kValue,kIncr,
                                     kLower,kUpper,false);
}
```

In this application case, the different spectral features correspond to MRSI channels and can assumed to be commensurable: hence no preprocessing except for the general MRSI preprocessing steps is required, and the associated preprocessor is an instance of the `IdentityPreprocessor` class, which leaves the features unchanged. In cases where one cannot assume the features to be commensurable, one should rather associate this classifier with a preprocessor of type `WhiteningPreprocessor` which brings all features to the same scale.

```
shared_ptr<Preprocessor>
NearestNeighborClassifier::getPreprocessorStubSpecific() const {
  shared_ptr<Preprocessor> output(new IdentityPreprocessor());
  return output;
}
```

For didactic reasons, we provide a simple, but admittedly inefficient implementation. The training process consists simply of storing the training features and labels:

```
double
NearestNeighborClassifier::
estimatePerformanceCvFoldSpecific( FoldNr iF,
                                   const Matrix<double>& features,
                                   const Matrix<double>& labels ){
  double output = learnCvFoldSpecific(iF,features,labels);
  cvFoldTrained(iF,0)=true;
  return output;
}

double
NearestNeighborClassifier::
learnSpecific( const Matrix<double>& features,
               const Matrix<double>& labels ){
  trainingSpectra = features;
  trainingLabels = labels;
  return estimateByInternalVal( features, labels );
}

double
NearestNeighborClassifier::
learnCvFoldSpecific(FoldNr iFold,const Matrix<double>&
                    features, const Matrix<double>& labels){
  trainingSpectraCvFolds[iFold] = features;
  trainingLabelsCvFolds[iFold] = labels;
  return estimateByInternalVal( features, labels );
}
```

The automated parameter optimization requires an estimate for the generalization error, which must be obtained from one single cross-validation fold: if the data has for example been split into a training and a testing fold, only the training fold may be used for this estimation. Otherwise one would incur a bias for the test error that is computed on the separate testing data set. Unlike many other classifiers (e.g. random forests), the kNN classifier does not automatically generate a generalization error estimate during training: hence one must resort to an internal validation step, in which the training data is split into an internal "training" and "testing" subset:

```
struct
NearestNeighborClassifier::
Comparison {
  operator()(const pair<double,double>& p1,
             const pair<double,double>& p2){
```

[12] For binary classification, ties can easily be avoided by restricting k to odd values. However, if the user chooses an even k, we err on the safe side and classify the spectrum as tumorous in case of a tie.

```
    return p1.first < p2.first;
  }
};

double
NearestNeighborClassifier::
estimateByInternalVal(const Matrix<double>& features,
                      const Matrix<double>& labels){
  unsigned k = parameters->getValue<unsigned>(k_name);
  // randomly group into two folds
  vector<int> folds( features.shape(0) );
  for( int i=0; i<features.shape(0); ++i ){
    folds[i] = rand() % 2;
  }
  unsigned correct = 0;
  unsigned wrong = 0;
  for(int i=0; i<features.shape(0); ++i ){
    if( folds[i]==0 ){ // 1 : test spectra, 0 : training spectra
      continue;
    }
    priority_queue<pair<double,double>, vector<pair<double,double> >,
      Comparison> currBest;
    unsigned nFound = 0;
    for(int j=0; j<features.shape(0); ++j){
      if( folds[j]==1 ){
        continue;
      }
      Matrix<double> tempVec = features.rowVector(i);
      tempVec -= features.rowVector(j);
      double newDist = tempVec.squaredNorm();
      if( nFound++ < k ){ // first k spectra automatically pushed
        currBest.push(pair<double,double>(newDist,labels(j,0)));
      } else {
        if( newDist < currBest.top().first ){
          currBest.pop();
          currBest.push( pair<double,double>(newDist,labels(j,0)));
        }
      }
    }
    double maxLabel = retrieveMajority(currBest);
    if( maxLabel==labels(i,0) ){
      correct++;
    } else {
      wrong++;
    }
  }
  return double(wrong)/(correct+wrong);
}
```

`retrieveMajority()` is a helper function to retrieve the most common label from the priority queue. Note that the implementation is deliberately simple for didactical reasons and has not been optimized for efficiency: in production code, one would store the training spectra in a balanced data structure like the box-decomposition trees (Arya et al, 1998) used in the ANN library[13] for faster retrieval. A similar implementation is used to predict the values of new test examples:

```
void
NearestNeighborClassifier::
predictLabelsAndScores(const Matrix<double>& featuresTrain,
                       const Matrix<double>& labelsTrain,
                       const Matrix<double>& featuresTest,
                       Matrix<double>& labelsTest,
                       Matrix<double>& scoresTest) const {
  unsigned k = parameters->getValue<unsigned>(k_name);
  labelsTest = Matrix<double>(featuresTest.shape(0),1);
  scoresTest = Matrix<double>(featuresTest.shape(0),classes.size(),0.);
  for(int i=0; i<featuresTest.shape(0); ++i){
    priority_queue<pair<double,double>, vector<pair<double,double> >,
      Comparison> currBest;
    unsigned nFound = 0;
    for(int j=0; j<featuresTrain.shape(0); ++j){
      Matrix<double> tempVec = featuresTest.rowVector(i);
      tempVec -= featuresTrain.rowVector(j);
      double newDist = tempVec.squaredNorm();
      if( nFound++ < k ){
        currBest.push(pair<double,double>(newDist,labelsTrain(j,0)));
      } else {
        if( newDist < currBest.top().first ){
          currBest.pop();
          currBest.push(pair<double,double>(newDist,labelsTrain(j,0)));
        }
      }
    }
    labelsTest(i,0) = retrieveMajority(currBest);
    while( !currBest.empty() ){
      scoresTest(i,classIndices.find(currBest.top().second)->second)+=1./k;
      currBest.pop();
    }
  }
}
```

This helper routine considerably simplifies the definition of the virtual prediction functions:

```
void
NearestNeighborClassifier::
predictBinaryScoresSpecific(const Matrix<double>& features,
                            Matrix<double>& scores) const {
  Matrix<double> labels;
  predictLabelsAndScores(trainingSpectra,trainingLabels,
                         features,labels,scores);
}

void
NearestNeighborClassifier::
predictBinaryScoresCvFoldSpecific(FoldNr iFold,
                                  const Matrix<double> &features,
                                  Matrix<double> &scores)const {
  Matrix<double> labels;
  predictLabelsAndScores(trainingSpectraCvFolds[iFold],
                         trainingLabelsCvFolds[iFold],
                         features,labels,scores);
}

void
NearestNeighborClassifier::
predictLabelsSpecific(const Matrix<double>& features,
                      Matrix<double>& labels) const {
  Matrix<double> scores;
  predictLabelsAndScores(trainingSpectra, trainingLabels,
                         features, labels, scores);
}

void
NearestNeighborClassifier::
predictLabelsCvFoldSpecific(FoldNr iFold, const Matrix<double>&
                            features, Matrix<double> &labels) const{
  Matrix<double> scores;
  predictLabelsAndScores(trainingSpectraCvFolds[iFold],
                         trainingLabelsCvFolds[iFold],
                         features,labels,scores);
}
```

Concerning serialization and derialization, this classifier is only responsible for its internal data. In contrast, the serialization of the parameter k is handled by the associated `ParameterManager` object, while the evaluation statistics are serialized by the `ClassifierManager`.

```
void
NearestNeighborClassifier::
saveSpecific( shared_ptr<SaveFunctor<string> > saver) const {
  shared_ptr<SaveFunctorInterface<string, Matrix<double> > > matSaver =
    dynamic_pointer_cast<SaveFunctorInterface<string,Matrix<double> > >(
    saver);
  CSI_VERIFY( matSaver );
  matSaver->save(training_spectra_name, trainingSpectra);
  matSaver->save(training_labels_name, trainingLabels);
  for( FoldNr iF=0; iF<nCvFolds; ++iF ){
    ostringstream currMatName;
    currMatName << getFoldName() << iF << " " << training_spectra_name;
    matSaver->save(currMatName.str(), trainingSpectraCvFolds[iF]);
    currMatName.str() = "";
    currMatName << getFoldName() << iF << " " << training_labels_name;
    matSaver->save(currMatName.str(), trainingLabelsCvFolds[iF]);
  }
}

void
NearestNeighborClassifier::
loadSpecific( shared_ptr<LoadFunctor<string> >loader){
  shared_ptr<LoadFunctorInterface<string, Matrix<double> > > matLoader =
    dynamic_pointer_cast<LoadFunctorInterface<string,Matrix<double> > >(
    loader);
  CSI_VERIFY( matLoader );
  matLoader->load(training_spectra_name, trainingSpectra);
  matLoader->load(training_labels_name, trainingLabels);
  trainingSpectraCvFolds.resize(nCvFolds);
  for( FoldNr iF=0; iF<nCvFolds;++iF ){
    ostringstream currMatName;
    currMatName << getFoldName() << iF << " " << training_spectra_name;
    matLoader->load(currMatName.str(), trainingSpectraCvFolds[iF]);
    currMatName.str() = "";
    currMatName << getFoldName() << iF << " " << training_labels_name;
    matLoader->load(currMatName.str(), trainingLabelsCvFolds[iF]);
  }
}
```

On the signal quality task for data set 1 (see section 3.1), this classifier gives a correct classification rate of ca. 95 % across all tested values for the parameter k.

4 Discussion and Outlook

To our best knowledge, this is the first C++ library specifically designed for medical applications which allows principled comparison of classifier performance and significance

[13] http://www.cs.umd.edu/~mount/ANN/

testing. We believe that this will help automated quality assessment and the conduction of clinical studies. While the absolute performance statistics of the single classifiers are most relevant for practical quality control in the clinic, the relative comparisons between different classifiers are interesting from a research-oriented point of view: for instance, they may answer the question which out-of-the-box classification techniques work best for the specific task of MRSI analysis, and can check whether newly proposed classification techniques give a significant advantage over established methods. Since quantitation-based classifiers may easily be incorporated into our framework, it will be possible to study the relative merits of quantitation-based techniques as opposed to pattern recognition-based techniques on a large set of patient data.

The design of our library is deliberately restricted to single-voxel classifiers that predict the malignancy or signal quality of each voxel only based on the appearance of the spectrum inside this voxel, without considering the context of the surrounding spectra. The reason for this limitation is that automatic single-voxel classification is a mature technology whose efficacy has been proved in several independent studies, e.g. those by Tate et al (2006), García-Gomez et al (2009) or Menze et al (2006). In contrast, classification with spatial context information has not yet been studied thoroughly: the two-dimensional conditional random field approach by Görlitz et al (2007) is the only one in this direction to our knowledge. In that article, the authors achieve a promising, but moderate improvement in prediction accuracy over single-voxel classification on a simulated data set (98.7 % compared to 98.2 %). However, it is yet far from clear which kinds of spatial context information may be beneficial for MRSI classification (2D neighborhoods, 3D neighborhoods, long-range context, comparison with coregistered MRI), and this question would have to be solved before a generic interface for such classifiers could be designed.

As next steps, the visualization and data reporting functionalities will be enhanced in order to improve usability: especially a more interpretable visualization of the statistical results may considerably benefit the medical users (for instance, plots of ROC curves could be provided, or the meaning of the AUC scores could be explained verbally). The clinical validation on 3 Tesla MRSI measurements of brain and prostate carcinomas is scheduled for the immediate future. Furthermore this software will eventually be integrated into the RONDO software platform for integrated tumor diagnostic and radiotherapy planning[14], where it is planned to be a major workhorse for MRSI analysis. This will provide a good test for the usefulness of pattern recognition techniques in a clinical routine setting.

[14] http://www.projekt-dot-mobi.de

Acknowledgements We acknowledge fruitful discussions with Ralf Floca and Ullrich Köthe, as well as support for the software development by Stephan Kassemeyer. A major share of the MeVisLab / C++ reimplementation of CLARET was done by B. Michael Kelm. The graphical user interface was initiated by Markus Harz. We thank the three anonymous reviewers for their helpful comments.

References

Arya S, Mount D, Netanyahu N, et al (1998) An Optimal Algorithm for Approximate Nearest Neighbor Searching. J ACM 45:891–923

Bandos A, Rockette H, Gur D (2007) Exact Bootstrap Variances of the Area Under ROC Curve. Comm Stat Theor Meth 36:2443–2461

Bengio Y, Grandvalet Y (2004) No Unbiased Estimator of the Variance of K-Fold Cross-Validation. J Mach Learn Res 5:1089–1105

Breiman L (1996) Out-of-Bag Estimation. Tech. rep., UC Berkeley

Chang C, Lin C (2001) LIBSVM: a library for support vector machines. Software available at http://www.csie.ntu.tw/ cjlin/libsvm

Cho S, Kim M, Kim H, et al (2001) Chronic hepatitis: in vivo proton MR spectroscopic evaluation of the liver and correlation with histopathologic findings. Radiology 221(3):740–746

Dager S, Oskin N, Richards T, Posse P (2008) Research Applications of Magnetic Resonance Spectroscopy (MRS) to Investigate Psychiatric Disorders. Top Magn Reson Imaging 19(2):81–96

Demšar J (2006) Statistical Comparisons of Classifiers over Multiple Data Sets. J Mach Learn Res 7:1–30

Dietterich T (1998) Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms. Neur Comput 10:1895–1923

de Edelenyi FS, Rubin C, Estève F, et al (2000) A new approach for analyzing proton magnetic resonance spectroscopic images of brain tumors: nosologic images. Nature Medicine 6:1287–1289

Fawcett T (2006) An introduction to ROC analysis. Patt Recog Lett 27(8):861–874

Frigo M, Johnson S (2005) The Design and Implementation of FFTW3. Proc IEEE 93(2):216–231

García-Gomez J, Luts J, Julià-Sapé M, et al (2009) Multiproject-multicenter evaluation of automatic brain tumor classification by magnetic resonance spectroscopy. Magn Reson Mater Phy 22:5–18

Gillies R, Morse D (2005) In Vivo Magnetic Resonance Spectroscopy in Cancer. Ann Rev Biomed Eng 7:287–326

Golub G, Heath M, Wahba G (1979) Generalized Cross-Validation as a Method for Choosing a Good Ridge Parameter. Technometrics 21(2):215–223

González-Vélez H, Mier M, Julià-Sapé M, et al (2009) HealthAgents: distributed multi-agent brain tumor diagnosis and prognosis. Appl Intell 30:191–202

Görlitz L, Menze B, Weber M, et al (2007) Semi-Supervised Tumor Detection in Magnetic Resonance Spectroscopic Images Using Discriminative Random Fields. In: Proceedings DAGM 2007, Lecture Notes in Computer Science, vol 4713/2007, pp 224–233

de Graaf R (2008) In Vivo NMR Spectroscopy: Principles and Techniques. Wiley, New York

Grandvalet Y, Bengio Y (2006) Hypothesis Testing for Cross-Validation. Tech. Rep. TR 1285, Département d'Informatique et Recherche Opérationelle, University of Montréal

Hagberg G (1998) From magnetic resonance spectroscopy to classification of tumors: A review of pattern recognition methods. NMR Biomed 11(4–5):148–156

Hastie T, Tibshirani R, Friedman J (2009) The Elements of Statistical Learning. Springer, New York

Kaster F, Kelm B, Zechmann C, et al (2009) Classification of Spectroscopic Images in the DIROlab Environment. In: World Congress on Medical Physics and Biomedical Engineering, IFMBE Proc, vol 25/V, pp 252–255

Kelm B, Menze B, Neff T, et al (2006) CLARET: a tool for fully automated evaluation of MRSI with pattern recognition methods. In: Handels H, Ehrhardt J, Horsch A, et al (eds) Bildverarbeitung für die Medizin 2006 – Algorithmen, Systeme, Anwendungen, pp 51–55

Kelm B, Menze B, Zechmann C, et al (2007) Automated Estimation of Tumor Probability in Prostate Magnetic Resonance Spectroscopic Imaging: Pattern Recognition vs Quantification. Magn Reson Med 57:150–159

Köthe U (2000) Generische Programmierung für die Bildverarbeitung. PhD thesis, University of Hamburg, software available at http://hci.iwr.uni-heidelberg.de/vigra/

Kreis R (2004) Issues of spectral quality in clinical ^{1}H magnetic resonance spectroscopy and a gallery of artifacts. NMR Biomed 17:361–381

Lin H, Lin C, Weng R (2007) A note on Platt's probabilistic outputs for support vector machines. Mach Learn 68:267–276

Martínez-Bisbal M, Celda B (2009) Proton magnetic resonance spectroscopy imaging in the study of human brain cancer. Q J Nucl Med Mol Imaging 53(6):618–630

Maudsley A, Darkazanli A, Alger J, et al (2006) Comprehensive processing, display and analysis for *in vivo* MR spectroscopic imaging. NMR Biomed 19:492–503

Menze B, Lichy M, Bachert P, et al (2006) Optimal classification of long echo time in vivo magnetic resonance spectra in the detection of recurrent brain tumors. NMR Biomed 19:599–609

Menze B, Kelm B, Weber M, et al (2008) Mimicking the Human Expert: Pattern Recognition for an Automated Assessment of Data Quality in MR Spectroscopic Images. Magn Reson Med 59:1457–1466

Neuter BD, Luts J, Vanhamme L, et al (2007) Java-based framework for processing and displaying short-echo-time magnetic resonance spectroscopy signals. Comput Methods Programs Biomed 85:129–137

Ortega-Martorell S, Olier I, Julià-Sapé M, et al (2010) SpectraClassifier 1.0: a user friendly, automated MRS-based classifier-development system. BMC Bioinformatics 11:106

Poullet J, Sima D, Van Huffel S (2008) MRS signal quantitation: A review of time- and frequency-domain methods. J Magn Reson 195(2):134–144

Provencher S (2001) Automatic quantitation of localized in vivo ^{1}H spectra with LCModel. NMR Biomed 14(4):260–264

Rifkin R, Klautau A (2004) In Defense of One-Vs-All Classification. J Mach Learn Res 5:101–141

Sajja B, Wolinsky J, Narayana P (2009) Proton Magnetic Resonance Spectroscopy in Multiple Sclerosis. Neuroimaging Clin N Am 19(1):45–58

Smith S, Levante T, Meier B, et al (1994) Computer Simulations in Magnetic Resonance. An Object-Oriented Programming Approach. J Magn Reson A106(1):75–105

Stefan D, Cesare FD, Andrasescu A, et al (2009) Quantitation of magnetic resonance spectroscopy signals: the jMRUI software package. Meas Sci Technol 20:104,035

Stroustrup B (2001) Exception Safety: Concepts and Techniques. In: Dony C, Knudsen J, Romanovsky A, et al (eds) Advances in Exception Handling Techniques, Springer, New York, pp 60–76

Tate A, Underwood J, Acosta D, et al (2006) Development of a decision support system for diagnosis and grading of brain tumours using *in vivo* magnetic resonance single voxel spectra. NMR Biomed 19(4):411–434

Xu D, Vigneron D (2010) Magnetic Resonance Spectroscopy Imaging of the Newborn Brain – A Technical Review. Seminars in Perinatology 34(1):20–27

Zechmann C, Menze B, Kelm B, Zamecnik P, Ikinger U, Waldherr R, Delorme S, Hamprecht F, Bachert P (2010) How much spatial context do we need? Automated versus manual pattern recognition of 3D MRSI data of prostate cancer patients. NMR Biomed (submitted)

Zhu G, Smith D, Hua Y (1997) Post-acquisition solvent suppression by singular-value decomposition. J Magn Res 124:286–289