The FieldTrip Multivariate Module

Marcel van Gerven

November 26, 2010

1   Module layout

The FieldTrip Multivariate Module (FMM) is a generic machine learning toolbox with additional support for the analysis of neuroimaging datasets and is written in object-oriented Matlab. It is especially suitable for applications in Brain-Computer Interfacing (BCI) and multivariate pattern analysis (MVPA).
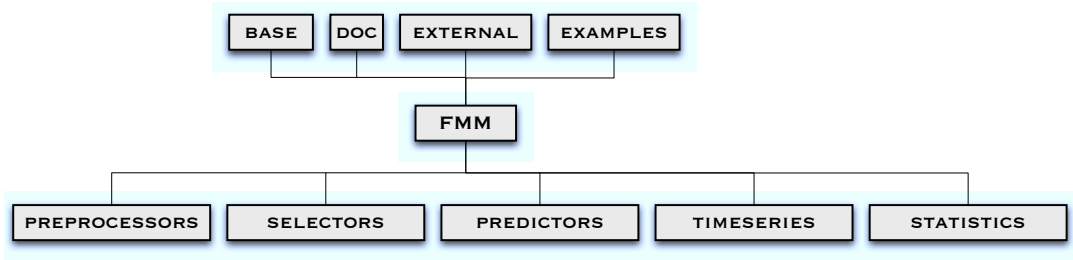
Figure 1: FMM's directory structure

The organization of FMM's directory structure is shown in Figure 1. The role of each directory is as follows:

- `base`: abstract classes that should not be used directly

- `preprocessors`: preprocessing wrapper classes

- `selectors`: feature selection wrapper classes

- `predictors`: prediction wrapper classes

- `timeseries`: timeseries analysis wrapper classes

- `statistics`: crossvalidation, performance measures, significance tests and parameter optimization

- `doc`: FMM documentation

- `external`: machine learning toolboxes used by wrapper classes

- `examples`: example scripts and datasets

The basic philosophy of FMM is that new methods can be added as toolboxes which are called through generic wrapper code. This makes it easy to extend the toolbox to fit your own needs. The currently used external toolboxes are listed in Table 1. Different multivariate methods can be concatenated together in a processing pipeline, constituting a multivariate analysis `ft_mv_analysis`,

Table 1: External toolboxes.

name	function	author
bahramisharif	elastic net	Ali Bahramisharif
farquhar	kernel methods	Jason Faquhar
gerven	Bayesian methods	Marcel van Gerven
glmnet	Fortran version of elastic net	Jerome Friedman
herman	common spatial patterns	Pawel Herman
L1General	L1 regularization	Mark Schmidt
murphy	LDS/HMM toolbox	Kevin Murphy
utilities	global functions	Marcel van Gerven

or MVA for short. Finally, an `ft_mv_analysis` object can either be used standalone or be evaluated on data using a `ft_mv_crossvalidator` object. This automates the whole process of analyzing (neuroimaging) data. Figure 1 shows a diagram of FMM classes, to which will be referred later on.
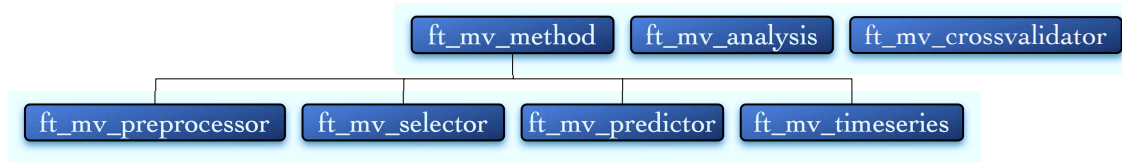
Figure 2: FMM's class diagram

2   Data handling

In FMM, a dataset is represented as a multidimensional array, where the first dimension indexes datapoints and the remaining dimensions are arbitrarily chosen. Multiple datasets are represented as a cell-array of multidimensional arrays. An exception to this rule holds for `ft_mv_timeseries` objects, where the first dimension indexes time and multiple timeseries are represented as a cell-array. Missing values are represented using `NaN`.

3   Multivariate analysis

A multivariate analysis is a Matlab object which can be constructed as follows:

```
obj = ft_mv_analysis({method1,...,methodN})
```

where the argument is a cell-array of multivariate methods. By calling

```
obj.train(X,Y)
```

we can train our MVA on input `X` and output `Y`. Finally, we can test on new data `U` by calling

```
V = obj.test(U)
```

such that V contains the predictions. The MVA object will sequentially call each ft_mv_method in the pipeline where output of the previous method will be input to the next method. The obj.name will return the MVA as a string and the method obj.model will return the associated model, in this case the model of the last method in the pipeline (more on this later).

4 Multivariate methods

Multivariate methods are organized into different subclasses according to the way they transform input data X to output data Y. Currently supported methods are *preprocessors*, which preprocess the data, *selectors*, which perform some kind of feature selection, *predictors*, which make predictions for new data, and *timeseries* methods, which perform timeseries analysis on the data. Many methods handle missing data, multiple outputs, multiple datasets and online (sample by sample) learning. If a method does not handle any of these situations then it will throw an error. Handling of some of these situations can be done by invoking a wrapper function.[1]

All mvmethod classes are structured as follows (here we show an example of a predictor):

```
classdef mymethod < ft_mv_predictor

   properties % insert class-specific properties here
   end

   methods

       function obj = mymethod(varargin)
         obj = obj@ft_mv_predictor(varargin{:});
       end

       function obj = train(obj,X,Y)
          % estimate mvmethod parameters
       end

       function Y = test(obj,X)
         % compute output
       end

       function X = invert(obj,Y)
       % optional function which inverts the mapping performed by this class

       end

       function [m,d] = model(obj)
         % return the model parameters as a cell-array m
         % desc contains a description of each element of m
       end
   end

end
```

In the following subsection we list the currently supported methods. More complete descriptions can be found in the help of the respective methods and examples of their use can be found in Section 8.

---

[1]E.g, multiple outputs can be handled by invoking an ft_mv_noutput object and multiple datasets can be handled by invoking the ft_mv_ndata object.

4.1 Preprocessors

Preprocessors are used to preprocess the input data `X` in such a way that it will be more suitable for making predictions. Table 2 lists the currently supported preprocessors and their functionality.

Table 2: Preprocessors

preprocessor	functionality
`ft_mv_standardizer`	Spheres and centers the data such that it has mean zero and a standard deviation of one
`ft_mv_whitener`	standardizes and whitens the data
`ft_mv_csp`	common spatial pattern algorithm
`ft_mv_pca`	principal component analysis

4.2 Selectors

Selectors are used to perform feature selection on the input data `X`. Table 3 lists the currently supported selectors and their functionality.

Table 3: Selectors

selector	functionality
`ft_mv_filterer`	Performs feature selection by computing a univariate measure for each feature and taking the best N features based on a crossvalidator
`ft_mv_searchlight`	Performs a searchlight analysis

4.3 Predictors

Predictors are used to predict new outputs `Y` based on input data `X`. Table 4 lists the currently supported predictors and their functionality.

Table 4: Predictors

predictor	functionality
`ft_mv_blogreg`	Bayesian logistic regression with a multivariate Laplace prior
`ft_mv_naive`	Naive Bayes classifier with normally distributed feature values
`ft_mv_svm`	Support vector machine; inherits from `ft_mv_kernelmethod`
`ft_mv_rkls`	Regularized kernel least squares; inherits from `ft_mv_kernelmethod`
`ft_mv_klr`	Kernel logistic regression; inherits from `ft_mv_kernelmethod`
`ft_mv_ensemble`	Ensemble method class that uses multiple mvmethods in parallel to achieve some goal
`ft_mv_logreg`	Logistic regression with L1 and/or L2 regularization
`ft_mv_linreg`	Linear regression with L1 and/or L2 regularization
`ft_mv_glmnet`	Wrapper class for efficient implementation of elastic net linear and logistic regression
`ft_mv_rfda`	Regularized Fisher discriminant analysis
`ft_mv_rls`	Recursive least squares regression

4.4 Timeseries

Timeseries methods are used to predict new outputs `Y` based on input data `X` whose first dimension indexes time. Table 5 lists the currently supported timeseries analysis methods and their functionality.

Table 5: Timeseries analysis methods

timeseries	functionality
ft_mv_hmm	Hidden Markov model
ft_mv_lds	Linear dynamical system

4.5   Classification, regression and MIMO models

Note that the above predictors can perform classification tasks, regression tasks or represent multiple-input multiple-output (MIMO) models. In case of K-class classification, it is assumed that the output vector Y consists of labels $1 : K$. For example, $Y = [2\,1\,1\,2\,1\,2]'$ assigns examples 2, 3 and 5 to class 1 and the remaining examples to class 2. In case of regression, Y just consists of the observed real output per example. In case of MIMO models , the same behaviour holds, but Y consists of multiple columns where each column stands for a separate variable.

4.6   Ensemble methods

Ensemble methods evoke methods in parallel and combine their outputs in some prespecified way. In FMM this is realized through the `ensemble` object. For example, we can apply a naive Bayes classifier and a support vector machine through ensemble methods as follows:

```
m = ft_mv_ensemble('mvas',{ft_mv_naive ft_mv_svm},'combfun',myfun);
m = m.train(X,Y);
...
```

Here, the user-specified function `myfun` specifies how the ouputs of naive Bayes and the SVM should be combined. For example, `myfun` could look implement a majority vote as follows

```
function y = myfun(x)

  y = zeros(size(x{1}));
  for k=1:length(x)
      [temp,pcls] = max(x{k},[],2);
      for p=1:length(pcls)
         y(p,pcls(p)) = y(p,pcls(p)) + 1;
       end
   end

  % resolve ties
  for p=1:size(y,1)
   m = find(ismember(y(p,:),max(y(p,:))));
   y(p,:) = 0;
   if length(m) > 1
    m = m(ceil(rand*length(m)));
   end
   y(p,m) = 1;
  end
```

4.7   Transfer learning

Transfer learning is the the notion that if we have multiple datasets (e.g., subjects, sessions) then we can learn better models for each dataset by being informed by the other datasets. For instance, if multiple datasets are given to `ft_mv_blogreg` then the same features are coupled between the different datasets. This leads to dataset-specific models that are easier to compare [3]. For instance, in the following example we apply this method to two datasets, both generated by adding random noise to the original data.

```
>> load 69digits;
>> X1 = X + 0.05*randn(size(X));
>> X2 = X + 0.1*randn(size(X));
>> a = ft_mv_test('mva',{ft_mv_blogreg},'X',X1,'Y',Y);
>> disp(a)
    0.6600
>> a = ft_mv_test('mva',{ft_mv_blogreg},'X',X2,'Y',Y);
>> disp(a)
    0.5800
>> a = ft_mv_test('mva',{ft_mv_blogreg('taskcoupling',100)},'X',{X1 X2},'Y',Y);
>> disp(a)
    [0.6900]
    [0.5900]
```

Hence, if we apply transfer learning then the results per dataset are influenced by the other
datasets. The taskcoupling parameter effectively couples the models:

```
>> disp(corr(d.model{1,1},d.model{1,2}))
    0.9999
```

Weaker coupling will cause weaker correlations between the models.

5 Statistics

We can either use a MVA to perform online state estimation or to perform an offline analysis of
neuroimaging data. The former is used in BCI applications (prediction) whereas the latter is used
in offline analysis of BCI data or MVPA (model inference). Online state estimation is handled in
the next section. Here, we describe statistics for models learned on offline data.

5.1 Cross-validation

Suppose we have acquired an offline dataset where subjects had to attend to the left or right visual
field. In a MVPA approach, we want to get an estimate of how well our MVA can predict the at-
tended location in individual trials and which features (brain regions, channels, latencies, frequen-
cies, etc) contributed to this prediction. This can be assessed using the ft_mv_crossvalidator
object. It splits the data into separate folds and learns for each fold a model on the remaining
folds. For example, in case of ten-fold cross-validation we will obtain ten different models which
are evaluated on ten different parts of the data. It is quite easy to perform such an analysis using
FFM, as shown below.

```
% create crossvalidator object which
% standardizes the data and applies an svm
cv = ft_mv_crossvalidator('mva',{ft_mv_standardizer ft_mv_svm});

% train cv on input data X and output data Y
cv = cv.train(X,Y);

% display classification accuracy
cv.metric = 'accuracy';
disp(cv.performance);
>> 0.80 % 80% correctly classified

% display outcome of mcnemar test
cv.sigtest = 'mcnemar';
disp(cv.significance);
>> 1e-4 % p-value; null-hypothesis rejected
```

In the example, a ten-fold cross-validation is performed by standardizing the data and applying a support vector machine. Subsequently, the classification accuracy (proportion of correctly classified trials) is computed. Finally, an approximate binomial test (McNemar test) is computed which compares the predictions with that of a naive classifier that assigns all outcomes to the majority class. The different performance measures and significance tests can be examined by consulting the help for `ft_mv_performance` and `ft_mv_significance`.

One may change `nfolds` to any number of folds, with `nfolds = inf` implementing leave-one-out cross-validation. Specifying a proportion for `nfolds` will use that proportion for training and the remaining data for testing. Finally, specifying `nfolds = 'online'` will simulate an online session by taking all trials in sequence where we train on trials $1, \ldots, n$ and test on trial $n + 1$ for $n = 0, \ldots, N - 1$.

We have not yet mentioned how to determine which features were responsible for the predictions. This is realized throught the `model` field of the cross-validator. It returns the average model that is produced by the last method in the specified MVA; in our case, the model of a SVM. It is up to each multivariate method to specify how its model is defined. For example, for the above example, assuming we used only five features, we have:


```
disp(cv.model);
>> [ 0.0176 -0.0305 0.0445 0.0904 0.0710]

disp(cv.description)
>> 'primal form parameters; positive values indicate condition 2'
```


Supported performance measures and statistical tests are described in Tables 6 and 7.

Table 6: Performance measures.

measure	output
accuracy	proportion of correctly classified cases
contingency	contingency table with true class as rows and predicted class as colums
cfmatrix	confusion matrix with true class as rows and predicted class as colums
correlation	correlation between real and predicted output

Table 7: Statistical tests.

test	output
mcnemar	approximate binomial test

5.2 Grid search

For many of the described methods there are free parameters which need to be optimized. For instance, the C parameter of the SVM or the regularization parameters $L1$ and $L2$ for the elastic net. In those cases, it is useful to employ the `ft_mv_gridsearch`. This object can be used to optimize free parameters of a multivariate analysis in a fully automated way. The optimizer is called as folllows:

```
ft_mv_gridsearch('mva',mymva,'validator',ft_mv_crossvalidator...
('metric',mymetric),'mvidx',myidx,'vars',myvars,'vals',myvals)
```

where `mymva` is the employed MVA, `mymetric` the employed performance measure to test which configuration is optimal, `myidx` the index of the method in `mymva` that is to be optimized, `myvars`

the variable of mymvamyidx that needs to be optimized and `myvals` the values which that variable may assume. This sounds complicated but it is not. Let's clarify with some examples. Suppose we wish to optimize the C parameter of an SVM in the range `logspace(-3,3,7)`. Then, we can use

```
% crossvalidator with 80% of the data and accuracy as the metric
cv = ft_mv_crossvalidator('nfolds',0.8,'metric','accuracy');

opt = ft_mv_gridsearch('verbose',true,'mva',svm,'validator',cv,...
'vars','C','vals',logspace(-3,3,7))

% standard behavior from here on
opt = opt.train(X,Y)
...
```

6   Online state estimation

If we use the toolbox for prediction then we need to work under strict time constraints. During training we want an optimal MVA to be learned quickly and be able to update our MVA with new incoming data. During testing we want to obtain an MVA output fast such that the online system does not stall. In the following example, we show an example of how online training and testing is realized.

```
% create a naive Bayes classifier
clf = ft_mv_naive

% train on initial data
clf = clf.train(X1,Y1)

% get output for new data; produces a posterior over classes
out1 = clf.test(x1);

% train some more (update the trained classifier)
clf = clf.train(X2,Y2);

% get output for new data; produces a posterior over classes
out2 = clf.test(x2);
```

In the above example we used a naive Bayes classifier but the MVA can be arbitrarily complex. For example,

```
clf = ft_mv_analysis({ft_mv_standardizer ...
        ft_mv_filterer('maxfeatures',10) ft_mv_svm})
```

creates a MVA which first standardizes the data, then performs feature selection and finally applies a support vector machine.

7   Parallelization

Some of the described methods can be resource intensive. Most notably, cross-validation, optimization, feature selection and the application of ensemble methods since they all require iterating over a collection of multivariate analyses. These methods all support parallel computing as implemented through FieldTrip's peer distributed computing module.

8 Examples

In the following examples, we will use the `69digits` dataset which is included in this module. It is a subset of the data used in [1, 2]. It consists of the BOLD measurements for 50 handwritten sixes and 50 handwritten nines as shown in a 3T MRI scanner. The dataset consists of the BOLD data `X`, the labels `Y` (1 stands for 6 and 2 stands for 9) and `images` (the presented handwritten digits). The variables `funloc` an `masks` contain the functional localizers and the associated mask in native space. The data pertains to the primary mask but subsets of features may be selected by using the other masks. In order to test a multivariate analysis we can make use of `ft_mv_test`. It accepts an MVA and tests it by default on the example dataset using ten-fold cross-validation.

A simple example

We start with a simple example to get some baseline performance:

```
>> [a,b,c] = ft_mv_test('mva',{ft_mv_standardizer ft_mv_svm});
using default dataset 69digits
initializing random number generator with seed 1
validating 1 dataset(s)
input 1 consists of 100 examples and 569 features
output 1 consists of 100 examples and 1 features
validating using 10-fold cross-validation
using 10 folds for 1 datasets
dataset 1: validating fold 1 of 10 using 90 training samples and 10 test samples
dataset 1: validating fold 2 of 10 using 90 training samples and 10 test samples
dataset 1: validating fold 3 of 10 using 90 training samples and 10 test samples
dataset 1: validating fold 4 of 10 using 90 training samples and 10 test samples
dataset 1: validating fold 5 of 10 using 90 training samples and 10 test samples
dataset 1: validating fold 6 of 10 using 90 training samples and 10 test samples
dataset 1: validating fold 7 of 10 using 90 training samples and 10 test samples
dataset 1: validating fold 8 of 10 using 90 training samples and 10 test samples
dataset 1: validating fold 9 of 10 using 90 training samples and 10 test samples
dataset 1: validating fold 10 of 10 using 90 training samples and 10 test samples

>> disp(a)
    0.8300

>> disp(b)
     1.1050e-05
```

Hence, baseline performance with a non-optimized SVM is 78% of the trials classified correctly (output a) which is signficant as the null-hypothesis is rejected (output b). Output c returns the employed cross-validator.

Using the optimizer

In case we want to optimize the SVM then we can use `ft_mv_optimizer`:

```
>> [a,b] = ft_mv_test('mva',{ft_mv_standardizer
      ft_mv_gridsearch('mva',{ft_mv_svm},'vars','C','vals',logspace(-3,3,7))});
```

Here we may obtain performance increases by searching over values for the C parameter. Computing time in this case is about 300 times longer. The reason for this is that for each value in the search (seven in this case) it performs an inner ten-fold cross-validation within each outer

cross-validation run. That is, we need to perform $7 \cdot 10 \cdot 10 = 700$ instead of 10 SVM estimations. There are various ways to speed this up. For one, we can use less outer and inner folds for cross-validation (or even just a percentage of the data). Furthermore, we can parallelize the outer and inner cross-validation as well as the optimization. Note however that if we parallelize everything then the cluster may be too busy with broadcasting events instead of doing the actual computations! The optimal combination often is more an art than a science.

Using a predictor to combine ensemble method output

It can be of interest to use the output of several predictors as input to another predictor. This is realized as follows:

```
m = ft_mv_ensemble('mvas',{ft_mv_naive ft_mv_svm},'combfun',@(x)(cell2mat(x)));
a = ft_mv_test('mva',{ft_mv_standardizer m ft_mv_naive});
```

Using the searchlight method

In this example, we show how we can compute the performance of individual spheres in a searchlight method using the default dataset.

```
load 69digits;
```

We first need to specify a cross-validator with which to test performance. We compute accuracy and a McNemar significance test by training on 80% of the data and testing on 20% of the data:

```
cv = ft_mv_crossvalidator('nfolds',0.8,'mva',{ft_mv_standardizer ft_mv_naive},...
          'metric','accuracy','sigtest','mcnemar');
```

Now we may specify the searchlight. We need to specify the sphere radius and the step size. Furthermore, we may use a logical mask which specifies how where the used data features are in native space (e.g., when using a region of interest as in case of the default dataset). Finally, we need to specify the original dimensions of the input data (in this case the size of the mask). The searchlight will use the cross-validator to test performance in each sphere:

```
sl = ft_mv_searchlight('radius',3,'step',3,'mask',zmask,'validator',cv,'...
          indims',size(zmask),'verbose',true);
sl = sl.train(X,Y);
>> sl = sl.train(X,Y);
estimating 39 spheres with radius 3 with 3 steps for a...
volume of size [110  110   37]
estimating sphere 1 of 39
estimating sphere 2 of 39
estimating sphere 3 of 39
estimating sphere 4 of 39
...
average sphere volume: 52.2308
performance for sphere 1 of 39: 0.736842 (p-value: 0.342782)
performance for sphere 2 of 39: 0.842105 (p-value: 0.148915)
performance for sphere 3 of 39: 0.789474 (p-value: 0.267257)
performance for sphere 4 of 39: 0.684211 (p-value: 0.546494)
performance for sphere 5 of 39: 0.421053 (p-value: 0.75183)
performance for sphere 6 of 39: 0.736842 (p-value: 0.386476)
performance for sphere 7 of 39: 0.578947 (p-value: 1)
performance for sphere 8 of 39: 0.736842 (p-value: 0.342782)
performance for sphere 9 of 39: 0.578947 (p-value: 1)
performance for sphere 10 of 39: 0.684211 (p-value: 0.504985)
...
```

In order to check how well each data feature does in discriminating the classes, we may call `sl.model`. This projects back the average sphere results in native space.

Using the searchlight method for feature selection

work in progress.

[1] M. A. J. van Gerven, B. Cseke, F. P. de Lange, and T. Heskes. Efficient Bayesian multivariate fMRI analysis using a sparsifying spatio-temporal prior. *NeuroImage*, 50(1):150–161, 2010.

[2] M. A. J. van Gerven, F. P. de Lange, and T. Heskes. A hierarchical generative model for percept reconstruction. In *Human Brain Mapping*, 2010.

[3] M. A. J. van Gerven and I. Simanova. Concept classification with Bayesian multi-task learning. In *NAACL-HLT workshop on Computational Neurolinguistics*, 2010.