flowMagic: automated gating of bivariate flow cytometry data

Sebastiano Montante

January 25, 2024

Abstract

This pdf describes the usage of the main flowMagic functions and reports examples of typical flowMagic scripts. Each function usually contains many parameters, only few of them are mentioned in this document. A detailed documentation of each parameter for each function can be found on the flowMagic_manual.pdf document in the pkg_manual github directory.

1 Introduction

2 Installation

The easiest way to install flowMagic is directly from github:

```
install_github("semontante/flowMagic")
```

The user can also download the package locally and install it from the package folder:

```
install("path/to/local_dir")
```

3 Input

There are 2 types of input: ungated data to analyze and the trained model to use for gating. The ungated data is within a directory containing the bivariate marker expression of the images under analysis. In particular, the marker expression must be reported in a csv file whose first and second column report the expression of, respectively, the first and second marker. Each row refers to the expression of a single event/cell. Here's an example of a correctly formatted csv file:

"PE-CF594-A"	"FITC-A"
2.985	3.137
2.433	2.752
2.942	2.914
2.874	3.121
1.805	2.119
1.925	2.712
0.871	2.56
2.782	2.836
2.645	2.265

The trained model can be either a templates model or a general purpose model. The template model is needed to automately gate the data based on the patterns defined by the user. The training data to generate this model is extracted by the user from the data to analyze containing the same combination of markers. The general purpose model is trained on a large FCM dataset containing different combination of markers extracted from different projects unrelated to the dataset under analysis. The flowMagic package includes a general purpose model in the inst/data folder trained on the Project Discovery data. The users can train their own general purpose model if they manage to generate a large dataset of FCM data with different combination of markers. Both the template model and the general purpose model require the training data in csv format with 3 columns for each csv file to train. Each csv file represents the bivariate marker expression (first and second column) associated with the gates (the third column called also classes column). Each numerical label indicates a different gate. Note that the 0 label always indicate the background events. In other words, they are events with no gate.

PE-CF594-A	FITC-A	Classes
2.985	3.137	1
2.433	2.752	2
2.942	2.914	2
2.874	3.121	1
1.805	2.119	0
1.925	2.712	0
0.871	2.56	1
2.782	2.836	1
2.645	2.265	1

The next sections show an example of automated gating using the template model and an example of gating using the general purpose model.

4 Automated gating using the template model

First, it is necessary to import the ungated data using the import_test_set_csv function. The user needs to indicate the path to a directory containing all unlabeled csv files under analysis. If there are more than two columns, the function will import only the first two columns. The user can also parallelize the importing process using more cores.

```
list_test_data<-import_test_set_csv(path_data = "path/to/data", n_cores=8)
```

list_test_data is a list in which each element is a dataframe of two columns containing the bivariate marker expression of each CSV file. To generate the template model it is necessary to import the reference data to use for training. The user needs to provide the path to the directory containing the labeled csv files. The CSV files need to contain 3 columns with the third column reporting the labels of each event.

```
list_data_ref<-import_reference_csv(path_results =  "path/to/data",n_cores = 8)
```

list_data_ref is also a list of dataframes. This list is the input of the pre-processing function needed to prepare the data for training. The get_train_data function generates the correctly formatted training set from the raw reference data, extracting the density features needed for the training. The function includes several parameters. For example, the user can choose the number of cores to speed up the pre-processing or they can choose to perform a downsampling of the data. By default, the function considers 90% of the input data. The data is also normalized by default, the user can also choose to disable normalization. If there are bivariate plots with extreme outliers, like sparse events in an angle of the plot, disabling normalization may improve accuracy.

```
# normalized data, no downsampling
ref_train<-get_train_data(paths_file = list_data_ref,n_cores = 8)

# normalized data, yes downsampling
ref_train<-get_train_data(paths_file = list_data_ref,prop_down = 0.90, n_cores = 8) #
    consider only 90\% of the events for each plot.

ref_train<-get_train_data(paths_file = list_data_ref,n_points_per_plot = 500, n_cores =
    8) # consider only 500 points for each plot.

# no normalized data, no downsampling
ref_train<-get_train_data(paths_file = list_data_ref,n_cores = 8, normalize_data=F)
```

Then, the user needs to select the indices of train set and validation set for the cross-validation method to perform during training. There are multiple possible methods, see the appropriate function documentation for the complete list of the methods. Below it is shown an example that performs the leave-out-out cross-validation.

```
list_inds_cross_val<-get_indices_cross_val(df_train = ref_train,
train_inds = "leave_one_out",val_inds = "leave_one_out")
```

After this, the training can begin. The user can choose the model to use for training. Each model has different parameters with their own default values that the users can change. The random forest with the default number of trees (n_trees=10) is used by default:

```
ref_model_info<-magicTrain(df_train = ref_train,train_model = "rf",
list_index_train = list_inds_cross_val$inds_train,list_index_val =
    list_inds_cross_val$inds_val)
```

In case of one template or 2 templates, it is suggested to use the out-of-bag cross validation which is the default cross-validation method in these cases. Note that the user does not need to input the cross-validation indices when using the out-of-bag method.

```
ref_model_info<-magicTrain(df_train = ref_train,train_model = "rf",
method_control="oob")
```

Finally, the prediction step can be performed. The user needs to provide the unlabeled data imported at the beginning and the trained model. If the user provides also the pre-processed data used for training (the previous ref_train variable), the prediciton function can also calculate the template-target distance for further analysis.

```
list_dfs_pred<-magicPred_all(list_test_data = list_test_data, ref_model_info =
    ref_model_info,ref_data_train = ref_train)
```

The list_dfs_pred variable is a nested list. Each element of the list contains the prediction information related to one plot. Each prediction information consists of a list of several dataframes and other outputs referring to different step of the prediction process for one plot. The most important dataframe is the dataframe reporting the predicted labels associated with each events of the input original data.

```
# Selecting the final dataframe of the first gated plot.
# The third column contains the predicted labels.
df_temp<-list_dfs_pred[[1]]$final_df
```

See the appropriate function documentation for details on the other outputs. If the user provided the pre-processed training set used for training, the vec_dist slot will contain the vector of target-template distances for each plot used as template. The other dataframes of each nested element refer to the gating of the downsampled data (in case downsampling is applied during the prediction step) or the normalized data. No downsampling is applied by default when using the template model. The downsampling is applied by default only when using the general purpose model.

5 Automated gating using the general purpose model

To apply the general purpose model, the user can use the same prediction function described in the previous section. The only difference is related to the arguments used. There are two models to use in this case. Model A predicts the number of gates in the plot, while Model B predicts the gates boundaries based on the predicted number of gates. There is a different Model B for each possible number of gates (e.g., Model B.2 predictes the two gates boundaries, Model B.3 predicts 3 gates boundaries). The magic_model argument require the list of Model B for each number of gates, while the magic_model_n_gates requires Model A. Based on the value predicted by Model A, the appropriate Model B.X is used from the provided list of Model B provided. The flowMagic R packgage already provides these two types of models.

```
out_pred<-magicPred_all(test_data = list_test_data,magic_model = list_magic_models,
    magic_model_n_gates = random_forest_model_pred_n_gates_index, n_cores = 8)
```

The users can also generate their own general purpose model using the flowMagic training function as described in next section. It is also possible to force the function to predict a pre-defined number of gates. It is sufficient to replace the number of gates model with an integer indicating the number of gates. The appropriate Model B will be selected from the list of Model B provided.

```
out_pred<-magicPred_all(test_data = list_test_data,magic_model = list_magic_models,
    magic_model_n_gates = 3,n_cores = 8) # to predict boundaries associated to 3 gates.

out_pred<-magicPred_all(test_data = list_test_data,magic_model = list_magic_models,
    magic_model_n_gates = 4,n_cores = 8) # to predict boundaries associated to 4 gates.
```

Finally, it is also possible to provide a single model predicting directly the gate boundaries.

```
out_pred<-magicPred_all(test_data = list_test_data,magic_model = single_model,
    magic_model_n_gates = NULL,n_cores = 8)
```

By default, when applying the general purpose model, the data is downsampled to 500 points to speed up execution. In addition, the number of events considered need to be consistent with the downsampling performed during the training of the general purpose model. The polygons calculated in the downsampled data will be overlapped on the original data to get the true number of events for each gate. See the function documentation for the details of each argument.

5.1 Example of full scripts using either template or general purpose model

Example of complete script using the template model.

Example of complete script using the general purpose model.

5.2 General purpose model training (only expert users)

This section is related to users with experience in training machine learning algorithms. Users with no machine learning experience can skip this section.

In order to generate the general purpose model it is required to import the dataset to use for training. The get_train_data function generates the correctly formatted training set from the raw reference data, extracting the density features needed for the training. The input can either be the list of labeled dataframes (with third column indicating the label assigned to each event) or the paths that lead to the labeled data. Since the data required to train the general purpose model is usually very large, providing the paths may be the best option instead of importing the raw data into memory. The paths format can either be a vector of paths pointing directly to the labeled dataframe or a two-columns dataframe with first column indicating the path to the expression data and the second column indicating the path to the labels of each event. Note that the data is normalized by default. Since the data for the general purpose model is usually very large in size, it is suggested to perform a 500 points dowsampling or similar to avoid overcoming the machine memory and speed the training. This is also the default option.

```
df_train<-get_train_data(df_paths =  df_paths,n_cores = 1,n_points_per_plot = 500) #
    using dataframes of paths and 500 points downsampling.

df_train<-get_train_data(paths_file =  vec_paths,n_cores = 1, n_points_per_plot = 500) #
     using vector of paths to labeled dataframes and 500 points downsampling.
```

As mentioned before, the general purpose model is composed by two models: Model A and Model B. In order to generate the Model A, the users need to provide the data containing plots with different number of gates. Then they will need to extract the cross-validation indices selecting the number of random plots in the training set for each number of gates and the number of random plots in the validation set for each number of gates. The n_folds argument indicates the number of times this process is repeated. Finally, the users will need to execute the training indicating the number of gates as response variable.

```
# generate cross-validation indices using 1000 training plots and 50 validation plots
    for 50 repetitions.
list_inds_cross_val<-get_indices_cross_val(df_train = df_train,n_cores = 4,train_inds =
    "rand_set_n_gates_info",n_train_plots = 1000,n_folds = 50,val_inds = "
    rand_set_n_gates_info",n_val_plots = 50)

# generate general purpose model using random forest.
random_forest_model<-magicTrain(df_train = df_train,n_cores = 4,train_model = "rf",
    list_index_train = list_inds_cross_val$inds_train,
list_index_val = list_inds_cross_val$inds_val,n_tree = 10,type_y = "n_gates_info")
```

In order to generate Model B for the specific number of gates, the users need to provide the data related to plots having only a specific number of gates.

```
# Selecting only plots with 2 gates.
df_train_ngates_selected<-df_train[df_train$n_gates_info==2,]
row.names(df_train_ngates_selected)<-NULL
```

Then, the cross-validation indices need to be extracted indicating the number of random plots in the training set and the number of plots in the validation set. Finally, the users will need to execute the training indicating the gates assignment (the classes column) as response variable.

```
# get cross-validation indices using 300 training plots and 5 validation plots for 50
    repetitions.
list_inds_cross_val<-get_indices_cross_val(df_train = df_train_ngates_selected,n_cores =
     8, train_inds = "rand_set_num",n_train_plots = 300,n_folds = 100, seed = 50,val_inds
     = "rand_set_num",n_val_plots = 5)

# generate general purpose model for the gates boundaries of the specific number of
    classes.
```

```
random_forest_model<-magicTrain(df_train = df_train_ngates_selected, n_cores = 4,
    train_model = "rf", list_index_train = list_inds_cross_val$inds_train, list_index_val
    = list_inds_cross_val$inds_val, n_tree = 10, type_y = "classes")
```

The users can also train a model considering the gates boundaries for all number of gates. In this case, they need to provide the original complete dataframe containing the plots with all number of gates. The users will need to extract the cross-validation indices selecting the number of random plots in the training set for each number of gates and the number of random plots in the validation set for each number of gates.

```
# generate cross-validation indices using 1000 training plots and 50 validation plots
    for 50 repetitions.
list_inds_cross_val<-get_indices_cross_val(df_train = df_train, n_cores = 4, train_inds =
    "rand_set_n_gates_info", n_train_plots = 1000, n_folds = 50, val_inds = "
    rand_set_n_gates_info", n_val_plots = 50)

# generate general purpose model for the gates boundaries for all number of gates.
random_forest_model<-magicTrain(df_train = df_train, n_cores = 4, train_model = "rf",
    list_index_train = list_inds_cross_val$inds_train, list_index_val =
    list_inds_cross_val$inds_val, n_tree = 10, type_y = "classes")
```

6 Visualization options

It is possible to visualize the gated plot using the visualization framework of the flowMagic package. The magicPlot() function can be used to plot the scatter plot of the gated plot of interest. It is possible to visualize the gates assigment on a standard scatter plot (with the events colored based on their gates) or it is possible to visualize the polygons on a bivariate density scatter plot.

```
magicPlot(df = df_temp, type = "ML") # gates assignment visualization

magicPlot(df = df_temp, type = "dens") # Bivariate density plot with polygons
    visualization.
```

7 Automatically replicate pre-defined hierarchy: GatingSet and FCS files

Note: the functions described in this section have been tested with flowCore version 1.11.20 and flowWorkspace 0.5.40. Newer packages version may not be compatible with these functions.

flowMagic is also compatible with GatingSet and FCS files. This format is useful in case the user does not want to generate the csv files for each gating step. In this mode of execution, flowMagic automatically trains on each gating step of a predefined hierarchy and automatilly gate the FCS files of interest based on the combinations of markers indicated in the gating hierarchy. The gating hierarchy information is contained within the GatingSet R object generated by the user using the flowWorkspace package. For example, the users can use flowDensity to gate a specific number of FCS files and then they store the gated results in a GatingSet object. Alternatevely, the users can manually gate the FCS files using the FlowJo software and then they can export a WSP file that flowMagic will automatically convert in a GatingSet object. This GatingSet object can be used to automatically generate the template model for each gating step of the hierarchy. **Note: this mode of execution is compatible only with the template model. The general purpose model cannot be used in this mode**.

First, the GatingSet needs to be imported.

```
gs_sample_gated<-import_gating_info(path="path/to/data")

gh_sample_gated_1<-gs_sample_gated[[1]] # first template

gh_sample_gated_2<-gs_sample_gated[[2]] # second template.
```

We need to import also the CSV files to analyze. Note that the FCS files need to have the same channel names. By default the first FCS is chosen as reference to check that all channel names are the same. FCS with channel names different from the chosen reference are excluded.

```
fs<-import_test_set_fcs(path= "/home/rstudio/final_data_test/HIPC_test_data/
    Myeloid_panel/test_fcs", n_samples = 1, ref_f_n = 1)
```

Next, we ge thet list of training sets for each gating step of the hierarchy.

```
# import gating hierarchy from the GatingHierarchy object which contains the gating
    hierarchy information of the sample.
out<-get_hierarchy_all_pops(gh=gh_sample_gated_1,export_visnet = F)

3 Extract all training data.
list_all_train_sets_1<-get_local_train_sets(gh=gh_sample_gated_1,hierarchical_tree=
    out$hierarchical_tree,
info_hierarchy=out)

list_all_train_sets_2<-get_local_train_sets(gh=gh_sample_gated_2,hierarchical_tree=
    out$hierarchical_tree,
info_hierarchy=out)
```

We prepare also the data to analyze.

```
list_all_test_sets<-get_test_sets(fs,gh=gh_sample_gated_1)
```

Finally, the user can execute the training of each training set and the prediction based on the hierarchy structure.

```
out_train<-magicTrain_hierarchy(list_train_sets = list_all_train_sets_1,n_cores = 8)
list_models<-out_train$list_models_sets_all_levels

list_gated_data<-magicPred_hierarchy(list_test_sets=list_all_test_sets,list_models_local
    = list_models,df_tree = out$df_tree,n_cores = 1)
```

The variable list_gated_data contains the labeled dataset for each gating step.

```
# extract gated results of the Singlets populations for first example
df_gated_1<-list_gated_data[[1]]$`level:6`$Singlets$gated_data

# Visualize gated population.
magicPlot(df=df_gated_1,type = "ML")
```