CellPhe user guide

CellPhe is a pattern recognition toolkit for the unbiased characterisation of cellular phenotypes within time-lapse videos. The toolkit is available on GitHub as an R package together with a user-friendly interactive GUI. This manual aims to guide users through the complete CellPhe workflow with a reproducible worked example.

1 CellPhe R package

1.1 Installing CellPhe

After downloading the CellPhe code from GitHub ( )
as a zip file using the green Code tab, the uncompressed file should be renamed as just CellPhe. Install dependencies in R:

```
install.packages(c("ptinpoly", "tiff", "smotefamily", " RImageJROI"))
library(ptinpoly)
library(tiff)
library(smotefamily)
library(RImageJROI)
install.packages(c("randomForest", "e1071", "factoextra", "tree"))
library(randomForest)
library(e1071)
library(factoextra)
library(tree)
```

In a Terminal window, in the directory above the one where the (uncompressed) CellPhe code is, type:

```
$ R CMD build CellPhe
$ R CMD INSTALL CellPhe_0.0.0.9000.tar.gz
```

Alternatively, the R package can be installed directly in R by running the following lines of code:

```
install.packages("devtools")
library(devtools)
install_github("uoy-research/CellPhe")
library(CellPhe)
```

To follow the worked example included in this manual, ensure that the example data sets are within your current R working directory.
The interactive CellPhe GUI can be accessed here:

1.2 Extraction of time series variables

The function `copyFeatures()` copies the tracking information (frame and cell identifiers) and any chosen features extracted by the tracking software from an input csv file. Currently CellPhe accepts input data from either PhaseFocus or TrackMate software, using `source = "Phase"` or `source = "Trackmate"`. For example, for PhaseFocus generated data, volume and sphericity features are copied as these rely on phase information. Only cells that are tracked for a minimum of `minframes` are included. Output is a dataframe with each row corresponding to a cell tracked on one frame. The first three columns give the `FrameID`, `CellID` and the name of the corresponding ROI file containing the boundary information (as output by either PhaseFocus or TrackMate) which are followed by a

column for each copied feature. Note that ROIs must contain full boundary coordinates. Files output by TrackMate only give the coordinates of the vertices of a polygon, but interpolation can be performed to extract full coordinates using ImageJ. An ImageJ macro, with instructions for use, is provided within the CellPhe GitHub repository.

The output from `copyFeatures()` is then read into the function `extractFeatures()` which calculates a further 72 features related to size, shape, texture and movement for each cell on every non-missing frame, as well as the cell density around each cell on each frame. If the frame rate is input by the user then velocity will be calculated using this information (`framerate`), otherwise a default value of 1 will be used, which does not affect the discriminatory power of the variable.

The cell time series output by `extractFeatures()` can be saved as an RDS (R Data File) and used as input to the CellPhe GUI for interactive data exploration. Alternatively, the time series can be input to the function `varsFromTimeSeries()` which calculates variables that summarise the cells' behaviour over time, providing both summary statistics and indicators of time-series behaviour at different levels of detail obtained via wavelet analysis. The output is a dataframe that can be used in multivariate analysis, e.g. for classification or clustering.

The example below uses data from a PhaseFocus experiment:

```
library('CellPhe')

## define path to input feature table
trial_name <- "05062019_B3_3"
basedir <- "data"
input_feature_table <-  sprintf("%s/%s_Phase-FullFeatureTable.csv", basedir, trial_name)

## set the minimum number of frames a tracked cell should appear on a
min_frames <- 50
## get the frame and cell identifiers and ROI filenames and copy any required features
feature_table <- copyFeatures(input_feature_table, min_frames, source = "Phase")

## define the paths to ROI files and images
roi_folder <- sprintf("%s/%s_Phase", basedir, trial_name)
image_folder <- sprintf("%s/%s_imagedata", basedir, trial_name)

## calculate time series for all features
new_features <- extractFeatures(feature_table, roi_folder, image_folder, framerate = 0.0028)

## save RDS file for use in CellPhe GUI
saveRDS(new_features, file = "my_data.rds")

## calculate variables to characterise each feature's time series
tsvariables <- varsFromTimeSeries(new_features)
```

1.3 Segmentation error prediction and removal

An optional step within the CellPhe workflow is the prediction of segmentation errors within a data set. This requires ground truth data sets of correctly segmented cells and those identified as segmentation errors. For accurate results, it will likely be necessary to compile new ground truth data sets for each cell type you work with due to heterogeneity between cell types. Ground truth data sets can be established by inspecting cells within image visualisation software such as ImageJ, noting cell IDs of correctly segmented cells and segmentation errors, and subsetting your feature table of extracted time series variables to only include these cells. Note that once ground truth data is available for a particular cell type, this can be re-used for further experiments involving the same cell type, including for example, different drug treatments. Ensure that you have one output file for correctly segmented cells and one for

segmentation errors.

You will then be ready to predict segmentation errors in R. The CellPhe R package includes two functions for this: `predictSegErrors()` and `predictSegErrors_Ensemble()`. Both work by training a number of decision trees that are then used to predict whether or not a new set of cells contain any segmentation errors. Final classifications are made via a voting system, where a cell is classified as segmentation error if more than a defined proportion of decision trees predict it as such. `predictSegErrors_Ensemble()` adds further stringency to the prediction of segmentation errors by calling `predictSegErrors()` multiple times and a cell is given a final classification of segmentation error if it receives a vote for this class in at least half of the repeated runs. Both functions output a list of the cells predicted as segmentation errors.

```
## read in ground truth feature tables of correctly segmented cells and segmentation errors

correctsegs = read.csv("CorrectSegs.csv", header = TRUE)
segerrors = read.csv("SegErrors.csv", header = TRUE)
```

```
## read in data sets for segmentation error prediction

UntreatedTraining = read.csv("UntreatedTraining.csv", header = TRUE)
TreatedTraining = read.csv("TreatedTraining.csv", header = TRUE)
UntreatedTest = read.csv("UntreatedTest.csv", header = TRUE)
TreatedTest = read.csv("TreatedTest.csv", header = TRUE)

## add a column of true class labels to each data set

UntreatedTraining<-addGroupInfo(UntreatedTraining, "Untreated")
TreatedTraining<-addGroupInfo(TreatedTraining, "Treated")
UntreatedTest<-addGroupInfo(UntreatedTest, "Untreated")
TreatedTest<-addGroupInfo(TreatedTest, "Treated")

## form training and test sets

Training = rbind(UntreatedTraining, TreatedTraining)
Test = rbind(UntreatedTest, TreatedTest)
```

The `predictSegErrors_Ensemble()` function can then be used for segmentation error prediction. Default parameters are set as follows:

- `num`, number of decision trees to be trained = 50
- `K`, number of repeated runs of `predictSegErrors()` to be performed = 10
- `proportion`, proportion of votes needed for a final classification of segmentation error to be made = 0.7

but these can be customised to suit your own requirements.

The `removePredictedSegErrors()` function can be used to remove any identified segmentation errors prior to downstream analysis.

```
## identify segmentation errors within training and test sets

segErrors_Training<-predictSegErrors_Ensemble(segerrors, correctsegs, 50, 10, Training[,-1],
Training[,2], 0.7)
```

```
segErrors_Test<-predictSegErrors_Ensemble(segerrors, correctsegs, 50, 10, Test[,-1],
Test[,2], 0.7)

## remove predicted segmentation errors from training and test sets

removePredictedSegErrors(Training, 2, segErrors_Training)
removePredictedSegErrors(Test, 2, segErrors_Test)
```

1.4 Calculating separation scores

Separation scores can be calculated to identify discriminatory variables for feature selection. The higher the separation score, the better a variable is at discriminating between the two cell populations. The `calculateSeparationScores()` function can be used to obtain a table of separation scores. The output is a data frame, where the first column lists the variable indices, the second column lists the variable names and the third lists the separation scores. A threshold can be set to only display separation scores above a defined threshold, and the calculateOptimalThresh argument can be set to `TRUE` to determine the optimal separation threshold as described within the CellPhe paper. Note that the default threshold is 0 unless otherwise defined.

```
## subset training set into untreated and treated sets

UntreatedTraining<-subset(Training, Training$Group == "Untreated")
TreatedTraining<-subset(Training, Training$Group == "Treated")
```

To obtain the full list or separation scores:

```
## calculate separation scores with default parameters

sepscores<-calculateSeparationScores(UntreatedTraining[,-c(1,2)], TreatedTraining[,-c(1,2)])
```

or to only store a list of separation scores above the optimal separation threshold:

```
## calculate separation scores with calculateOptimalThresh = TRUE

sepscores<-calculateSeparationScores(UntreatedTraining[,-c(1,2)], TreatedTraining[,-c(1,2)],
calculateOptimalThresh = TRUE)
```

Separation scores can then be used for feature selection by subsetting the full training and test sets so that only variables with separation scores above the desired threshold are retained.

```
## Subset training and test sets to only include features with separation scores above the
## selected threshold

Training = cbind(Training[ ,c(1,2)], Training[ ,sepscores[,2]])
Test = cbind(Test[ ,c(1,2)], Test[ ,sepscores[,2]])
```

1.5 Cell population ensemble classification

CellPhe's `cellPopulationClassification()` function can be used for training and testing of an ensemble of classifiers, namely Linear Discriminant Analysis (LDA), Random Forest (RF) and Support Vector Machines (SVM). The function outputs a table of predicted classes, with the first, second and third columns corresponding to predictions from LDA, RF and SVM respectively. The fourth column lists final predicted class labels for each cell based on a

majority vote system.

R's `table()` function can be used to obtain a confusion matrix of ensemble classification results so that classification accuracy scores can be calculated.

```
## Perform ensemble classification

classifications<-cellPopulationClassification(Training[,-c(1,2)], Test[,-c(1,2)],
as.factor(Training[,1]))

## Display confusion matrix of results

table(Real = Test[,1], Predicted = classifications[,4])
```

2 CellPhe GUI

The interactive CellPhe GUI provides a user-friendly platform for data exploration, cell type classification and identification of heterogeneous clusters. The GUI can be used to visualise cell time series and extract time series variables from feature tables in the same way as the `varsFromTimeSeries()` function, output files can then be explored in real-time or saved for future use. Discriminatory variables can be identified with ease through calculation of separation scores, beeswarm plots, PCA, UMAP and t-SNE. Furthermore, all plots are interactive, facilitating identification and isolation of single cells for further investigation. Training and test sets can be uploaded for ensemble classification, with options for supervised and unsupervised classification. Classification models can be validated within the GUI through confusion matrices, accuracy metrics and ROC curves. The CellPhe GUI also facilitates hierarchical and k-means clustering for identification of heterogeneous cell subsets. Clustering outputs are interactive, allowing the user to determine the optimal number of clusters within their data and explore feature importance.