DATA501 project - `dbscanDATA501`

Draft package - Test plan

Hans Eliezer - 300365985

1 Preparation

1.1 Installing the packages

It would be easiest to install my `dbscanDATA501` package directly from the GitHub repository using `devtools`'s `install_github()`. The link is pasted below:

```
devtools::install_github('https://github.com/hanseliezer/dbscanDATA501')
```

There are two other well-known implementations of DBSCAN available in R in packages `fpc` and `dbscan`. These will be useful to compare results. You can install these two using `install.packages()`:

```
install.packages(c('fpc' 'dbscan'))
```

Finally, the clustering summary statistics shown in this package also requires one additional package `clValid`:

```
install.packages('clValid')
```

1.2 Loading package and dataset

The package can now be loaded as usual:

```
library(dbscanDATA501)
```

For this test, you can just use the `iris` dataset. It has the original class labels, which is not needed for a clustering task, so it can be excluded:

```
data(iris)
iris_X <- iris[, 1:4]
```

Though feel free to try out other datasets. The only requirement is that every column must be of `numeric` class.

2 Using `dbscan()` properly

2.1 Right parameters

The primary functionality for this package is the `dbscan()` function. For the first few tests, you can try if the function works as intended when all the parameters are supplied as expected:

- `data` must not be `NULL` or have 0 rows.
- `min_pts` and `eps` should be numeric/integers.
- Default distance `metric` is `euclidean`. Another options are `manhattan` and `precomputed`: `precomputed` accepts `data` in the form of a pre-calculated distance matrix.
- Default `normalise` is `TRUE`. This will normalise the dataset prior to distance matrix calculation and algorithm fitting (skipped if `metric` is `precomputed`).
- Default `border_points` is `TRUE`. This will include "border points" as part of a cluster; this is equivalent to the original described DBSCAN algorithm. Whereas `border_points = FALSE` excludes them, which is equivalent to a later proposed 'hierarchical DBSCAN'.

The following are some example tests where everything should work as intended/no errors generated (remember to use the sliced `iris` rather than the original one):

```
test_1 <- dbscan(iris_X, 0.2, 5)
test_2 <- dbscan(iris_X, 10, 4, metric='manhattan')
test_3 <- dbscan(iris_X, 9L, 2L, normalise=FALSE)
test_4 <- dbscan(iris_X, 0.1, 10, metric='euclidean', border_pts=FALSE)
```

2.2 Right results

The DBSCAN algorithm should be fully deterministic when given exactly the same dataset, which means the generated cluster/cluster labels should be exactly equal between different implementations given the same parameters. You can use `fpc` and `dbscan`'s `dbscan()` functions to cluster the `iris_X` dataset to get their clusters, and compare them with this package's clusters. Note that `fpc` and `dbscan`'s `dbscan()` does *not* normalise the dataset prior to fitting, so you should do that beforehand:

```
iris_X_scl <- scale(iris_X)
```

Also note that only `dbscan`'s `dbscan()` has the `borderPoints` parameter which is equal to this package's `border_pts`, while `fpc`'s does not: `fpc` will always include border points.

Some example tests are as follows. You might want to explicitly include the package's name when calling `dbscan()` so you don't confuse yourself when recalling which result came from which package.

```
dbscan_DATA501 <- dbscanDATA501::dbscan(iris_X, 0.4, 5)
dbscan_dbscan <- dbscan::dbscan(iris_X_scl, 0.4, 5)
dbscan_fpc <- fpc::dbscan(iris_X_scl, 0.4, 5)

all(dbscan_DATA501$cluster_labs == dbscan_dbscan$cluster)
```

```
## [1] TRUE
```

```
all(dbscan_DATA501$cluster_labs == dbscan_fpc$cluster)
```

```
## [1] TRUE
```

3 Clustering summary with `summary()`

`summary()` receives the `dbscanDATA501` object created by `dbscan()`, calculates and displays a few summary statistics such as how many clusters were created, as well as 'internal validation' metrics which basically tells us how "good" the created clusters are. I have included four different metrics called from three different packages: `cluster`, `fpc` and `clValid`. For testing, it would be useful to confirm that `summary()` successfully calls these packages and generated scores for all of them (the actual values don't really matter). `summary()`'s output should look something like this:

```
summary(dbscan_DATA501)
```

```
## DBSCAN result summary:
##
## Parameters:
## eps: 0.4
## min_pts: 5
## metric: euclidean
## normalise: TRUE
## border_pts: TRUE
##
## Running time (s): 0.00391
## Number of generated clusters (excl. noise): 6
##
## Clustering quality metrics:
## Connectivity (lower better): 97.67222
## Mean silhouette width (nearer to 1 better): 0.03232
## Dunn index (higher better): 0.16169
## CDbw (higher better): 4.19324
##
## NOTE: Caution must be taken when interpreting connectivity, mean silhouette width
##
##       and Dunn index for non-globular clusters.
```

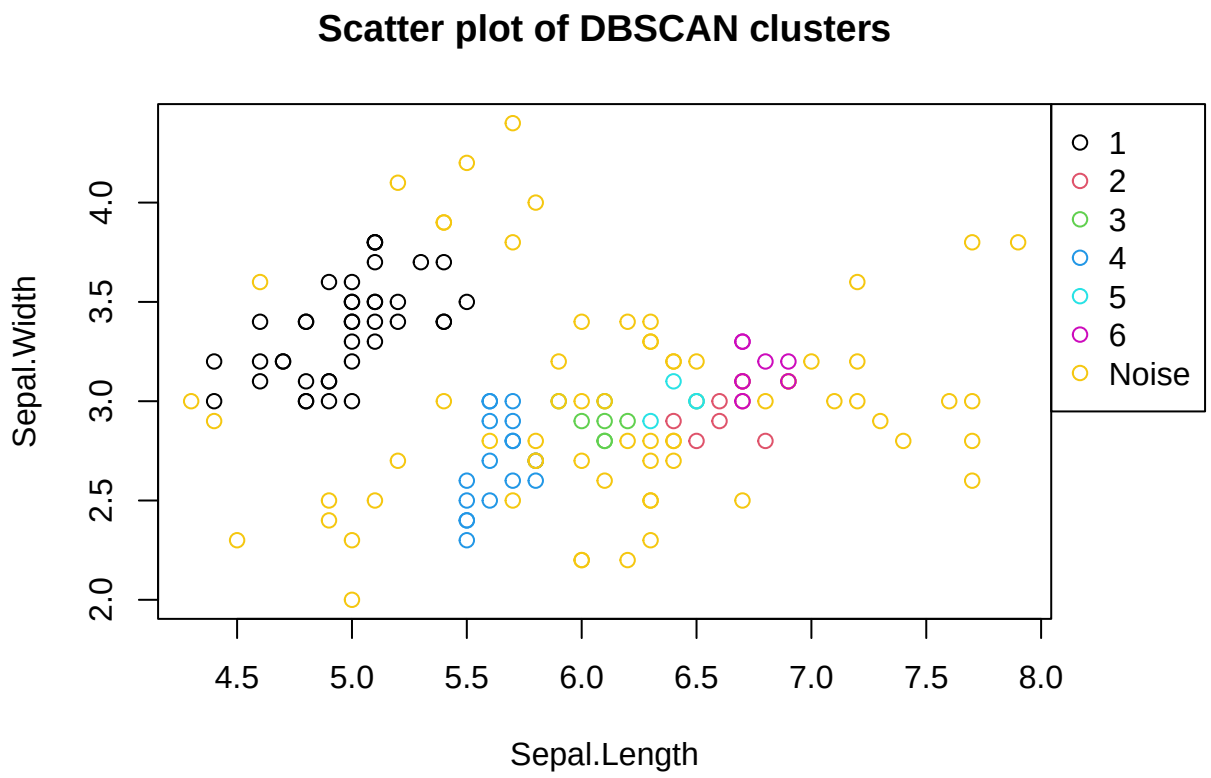
Note that if you are testing using other datasets, there will be cases where the resulting clusters break the calculations of these metrics in some way (e.g., `cluster`'s `silhouette()` returns `NA` if there is either only 1 cluster, or if the number of clusters equal the number of points). In these cases, the `summary()` function will simply display their result as `NaN`.

4 Plotting with `plot()`

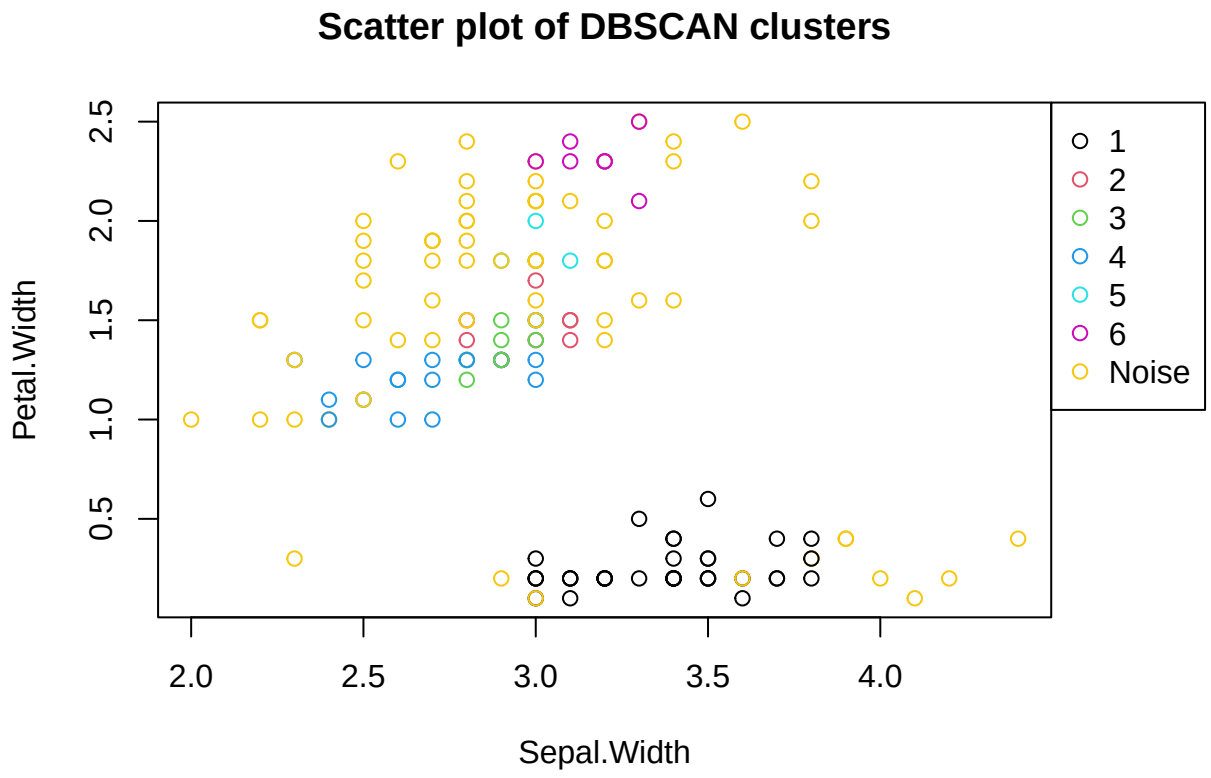
`plot()` receives the `dbscanDATA501` object created by `dbscan()` and displays a scatter plot of the dataset's observations, using two features from the original dataset and adding the generated clusters labels to group the points. The default selection is to use the first column as the x-axis, the second column as the y-axis, and the clusters to be displayed as different colours. There are three possible ways to display the clusters: `colour`, `shape` and `both` (so clusters have different colour *and* shape).

For testing, it would be useful to confirm that a plot is indeed generated, to try out plotting different columns present in your dataset, and to plot it using the three different kinds of grouping.

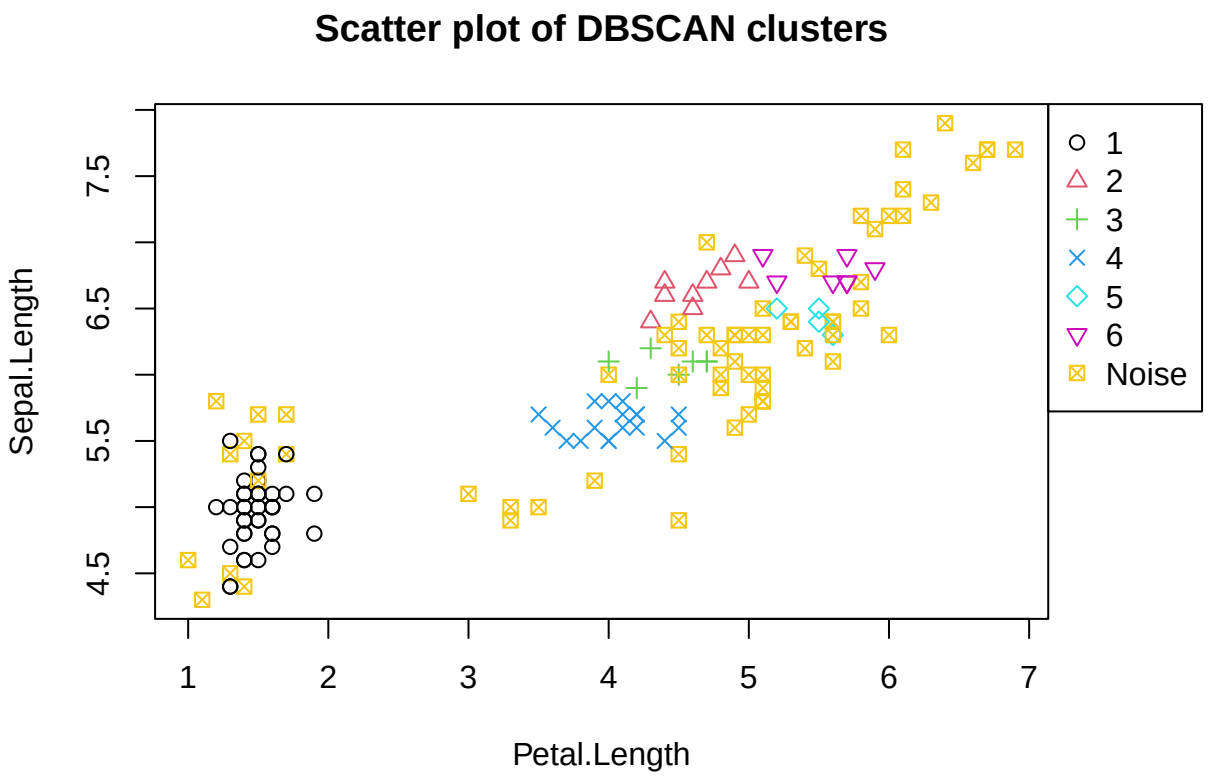
```
plot(dbscan_DATA501)
```




```
plot(dbscan_DATA501, ax1=2, ax2=4)
```




```
plot(dbscan_DATA501, ax1=3, ax2=1, kind='both')
```



5 Breaking `dbscan()`, `summary()` and `plot()`

Obviously there are many, many ways you can try to break the function, the most straightforward would be to supply the wrong type of parameters:

```
dbscan(iris_X, "9", 5)
```

```
## Error in dbscan_input_checks(data, eps, min_pts, metric, normalise, border_pts): eps must be i
```

```
dbscan(NULL, 0.3, 9)
```

```
## Error in dbscan_input_checks(data, eps, min_pts, metric, normalise, border_pts): Please supply
```

```
dbscan(iris_X)
```

```
## Error in dbscan(iris_X): argument "eps" is missing, with no default
```

```
dbscan(iris_X, 9, 10, metric="nonsense")
```

```
## Error in dbscan_input_checks(data, eps, min_pts, metric, normalise, border_pts): Options for d
```

```
dbscan(iris_X, 1.1, 2, normalise=iris_X)
```

```
## Error in dbscan_input_checks(data, eps, min_pts, metric, normalise, border_pts): normalise mus
```

```
plot(dbscan_DATA501, ax1=17, ax2=22)
```

```
## Error in scatter_input_checks(obj, ax1, ax2, kind): If x is an index referring to a column, it
```

```
plot(dbscan_DATA501, ax1=1, ax2=2, kind="empty")
```

```
## Error in scatter_input_checks(obj, ax1, ax2, kind): Options for 'kind' are 'colour', 'shape',
```

For `summary()`, the error trapping present in the function means that no matter the clustering and metric calculation results, the function will still print an output to the console. But you are welcome to try and see if it is still possible to make `summary()` throw an error.