R-session-databasics

Chao, Jon, Yann

5 November 2019

Introduction

In this session we will outline a basic environmental data work-flow. Our goal is to highlight common data tasks, and typical ways to solve them in R.

When working with environmental data, there are usually a few steps that come up each time. These are:

- **reading**. Typically data is read from text files, but can also come from the internet or in other format
- **processing**. The data we read is usually a little untidy , for example we may need to subset to correct dates.
- **plotting**. Plotting data is always worth doing as early as possible. Use histograms or simple line plots as your first steps in visualising data.

To do these efficiently in R is mainly about learning which functions to use, and how to apply these functions.

In this notebook we will work through each step in turn with example data. We will once again work with data downloaded from SMARTSMEAR, in this case we will use flux data measured using the eddy co-variance technique at SMEARII research station. We will use Temperature which is measured at 16.8m Height, Gross Primary Production (GPP) which is derived from measurements of CO2 exchange, and Evapotranspiration (ET) which is derived from measurements of H2O exchange.

Before we start there is one other thing we should mention.In this session we will assume that terms like *function*, *argument* are familiar to you. If they are not then go back to R1-introduction.ipynb, and check the definition. If you cannot find the definition in there then complain to your instructors to update the intro! Alright, let's get started.

#1. Reading

Our first task is to read in our GPP, ET and temperature data. Reading data takes data from storage (typically your computer's hard disk) and places it somewhere (in RAM) that is can be operated on by R. We have already downloaded our data as two seperate text files from SMARTSMEAR, and stored these files in the */data* directory (folder) on github: https://github.com/OptPhotLab/EnvDataSciNotebooks/tree/master/data (You can inspect the data files by clicking the github link, but opening the individual files on github could slow your computer down!)

There are a few different functions for reading data in R, these include:

- read.csv
- read.table
- read.delim
- read.csv2

We can use **help** to inspect these functions, see what arguments they have, and how to set these arguments so that you can read your data/file in a proper way.

```r
#help(read.csv)
```

Let's use *read.csv* to read in our GPP dataset.

```r
gpp<-read.csv('../data/gppsmeardata_20160101120000.csv',header = T,sep = ',',dec='.')
```

The double dots **..** in the path tell R to go up a level in the directory (folder) hierarchy. The full path (location) of the ET data is:

../data/ET smeardata_20160101120000.csv

go ahead and read the ET data:

```
# name the output data ET
ET<-read.csv('../data/ET smeardata_20160101120000.csv',header = T,sep = ',',dec='.')
```

It is as simple as that!

We have read our data into the memory, the next step is processing. But just before we move on we can use the *head* function to inspect the first few lines of our data object:

```
head(ET)
```

```
##   Year Month Day Hour Minute Second HYY_EDDY233.ET_gapf
## 1 2016     1   1    0      0      0               0.102
## 2 2016     1   1    0     30      0               0.035
## 3 2016     1   1    1      0      0              -0.042
## 4 2016     1   1    1     30      0               0.043
## 5 2016     1   1    2      0      0               0.027
## 6 2016     1   1    2     30      0               0.077
```

can you also remember how to check the type of our objects?

```
str(ET)
```

```
## 'data.frame':    17616 obs. of  7 variables:
##  $ Year               : int  2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 ...
##  $ Month              : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Day                : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Hour               : int  0 0 1 1 2 2 3 3 4 4 ...
##  $ Minute             : int  0 30 0 30 0 30 0 30 0 30 ...
##  $ Second             : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ HYY_EDDY233.ET_gapf: num  0.102 0.035 -0.042 0.043 0.027 0.077 0.121 0.055 0.165 0.141 ...
```

#2. Processing

Before we can make any graphs or perform any stats we usually have to tidy our data and there are a bunch of techniques in R that can help out with this. Let's check out a few of them that make life easier.

##2.1. Combining

We read in *two* different data files which have same first six columns. We can make life easier by combining these into a single dataframe using *merge()* function.

Use the *by* argument to set which variables are shared.

```
gpp.ET<-merge(gpp,ET,by=c("Year","Month","Day","Hour","Minute", "Second"),all = T)
```

Use *head* to check the combination worked:

```
head(gpp.ET)
```

```
##   Year Month Day Hour Minute Second HYY_EDDY233.GPP HYY_EDDY233.ET_gapf
## 1 2016     1   1    0      0      0           0.430               0.102
## 2 2016     1   1    0     30      0           0.318               0.035
## 3 2016     1   1    1      0      0          -0.219              -0.042
## 4 2016     1   1    1     30      0           0.220               0.043
## 5 2016     1   1   10      0      0           0.268               0.095
## 6 2016     1   1   10     30      0           0.145               0.112
```

Now, let's read the third file: temperature into R.

'../data/T168_20160101120000.csv'

```
temp<-read.csv('../data/T168_20160101120000.csv',header = T,sep = ',',dec='.')
```

we will combine temperature (temp) data with gpp and ET.

Try yourself here and name the combined file as *merge.all*:

```
merge.all<-merge(gpp.ET,temp,by=c("Year","Month","Day","Hour","Minute", "Second"),all = T)
```

Note, merge() can be only used to combine two dataframe into one file. Can we combine multiple files into one using one function at one time? The answer is Yes.

Reduce() function is one of the solutions.

```
#Note x or y is index, they can also be anything else, such as m and n.
reduce.all<-
  Reduce(function(x,y) merge(x,y,by=c("Year","Month","Day",
                                    "Hour","Minute","Second"),all = T),
         list(gpp,ET,temp))
```

check the str of *reduce.all* that should be exactly same with *merge.all*

```
str(reduce.all)
```

```
## 'data.frame':    17616 obs. of  9 variables:
##  $ Year                : int  2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 ...
##  $ Month               : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Day                 : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Hour                : int  0 0 1 1 10 10 11 11 12 12 ...
##  $ Minute              : int  0 30 0 30 0 30 0 30 0 30 ...
##  $ Second              : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ HYY_EDDY233.GPP     : num  0.43 0.318 -0.219 0.22 0.268 0.145 -0.09 -0.063 -0.023 -0.046 ...
##  $ HYY_EDDY233.ET_gapf: num  0.102 0.035 -0.042 0.043 0.095 0.112 0.035 0.101 0.075 0.11 ...
##  $ HYY_META.T168      : num  -6.53 -6.52 -6.59 -6.55 -5.96 ...
```

##2.2 Rename the column names

We can check the colunm names using *names(data)*

```
names(reduce.all)
```

```
## [1] "Year"                "Month"               "Day"
## [4] "Hour"                "Minute"              "Second"
## [7] "HYY_EDDY233.GPP"     "HYY_EDDY233.ET_gapf" "HYY_META.T168"
```

we can also use *names(data)* to change the column names

```
names(reduce.all)<-c("Year","Month","Day","Hour","Minute","Second",
                     "HYY_EDDY233.GPP","HYY_EDDY233.ET_gapf","HYY_META.T168")
```

But usually we only want to change few column names:

```
names(reduce.all)[8]<-'ET'
```

```
names(reduce.all)[c(7,9)]<-c('GPP','T168')
```

check the updated column names

```
names(reduce.all)
```

```
## [1] "Year"   "Month"  "Day"    "Hour"   "Minute" "Second" "GPP"    "ET"
## [9] "T168"
```

##2.3 Subsetting

###Method 1

Often we download much more data than we need. Subsetting using the *subset* function is a useway to restrict our datasets to the bits we are actually interested in.

subset accepts column names as a second argument. You can use subset to extract data for the month of September from *reduce.all* like this:

```
reduce.all.sep <- subset(reduce.all, Month==9)
```

Can you create a new dataframe containing data measured at midday (when hour is between 10 and 15 o'clock) only?

Name this dataframe *reduce.all.midday*

```
reduce.all.midday<- subset(reduce.all, Hour>10&Hour<15)
```

Use *head* to check the dates are correct:

```
head(reduce.all.midday)
```

```
##    Year Month Day Hour Minute Second    GPP    ET       T168
## 7  2016     1   1   11      0      0 -0.090 0.035 -6.034000
## 8  2016     1   1   11     30      0 -0.063 0.101 -6.196667
## 9  2016     1   1   12      0      0 -0.023 0.075 -6.292000
## 10 2016     1   1   12     30      0 -0.046 0.110 -6.459000
## 11 2016     1   1   13      0      0  0.279 0.080 -6.583667
## 12 2016     1   1   13     30      0  0.375 0.110 -6.671667
```

###Method 2.

We can also subset the data inside the dataframe.

data[row.index,column.index]

Example 1, we want to select first 7 rows

```
exam1<-reduce.all[c(1:7),]
exam1
```

```
##   Year Month Day Hour Minute Second    GPP     ET       T168
## 1 2016     1   1    0      0      0  0.430  0.102 -6.527667
## 2 2016     1   1    0     30      0  0.318  0.035 -6.520333
## 3 2016     1   1    1      0      0 -0.219 -0.042 -6.594666
## 4 2016     1   1    1     30      0  0.220  0.043 -6.551667
## 5 2016     1   1   10      0      0  0.268  0.095 -5.964667
## 6 2016     1   1   10     30      0  0.145  0.112 -5.964334
## 7 2016     1   1   11      0      0 -0.090  0.035 -6.034000
```

Example 2, we want to select data *reduce.all* when row numbers are 2, 50, and 100:102 and keep first 7 columns

```
exam2<-reduce.all[c(2,50,100:102),1:7]
exam2
```

```
##     Year Month Day Hour Minute Second   GPP
## 2   2016     1   1    0     30      0 0.318
## 50  2016     1  10    0     30      0 0.000
## 100 2016     1  11    1     30      0 0.000
## 101 2016     1  11   10      0      0 0.194
## 102 2016     1   11    10      30       0 1.351
```

we can also select data by specific conditions.

Example 3, We want to select the rows when the Hour is between 10 and 15 o'clock (midday data) and keep all the columns.

```
reduce.all.midday<-reduce.all[reduce.all$Hour>10&reduce.all$Hour<15,]
head(reduce.all.midday)
```

```
##    Year Month Day Hour Minute Second    GPP    ET      T168
## 7  2016     1   1   11      0      0 -0.090 0.035 -6.034000
## 8  2016     1   1   11     30      0 -0.063 0.101 -6.196667
## 9  2016     1   1   12      0      0 -0.023 0.075 -6.292000
## 10 2016     1   1   12     30      0 -0.046 0.110 -6.459000
## 11 2016     1   1   13      0      0  0.279 0.080 -6.583667
## 12 2016     1   1   13     30      0  0.375 0.110 -6.671667
```

##2.3 reordering

Did you notice something odd? The days are not in ascending order. We can sort this out using *order()* function

```
reduce.all.midday <-
  reduce.all.midday[order(reduce.all.midday$Year,reduce.all.midday$Month,reduce.all.midday$Day), ]
```

Let's check if this has worked out as expected:

Now we have a single dataframe with data at our desired midday time-step we can start with our visualisations.

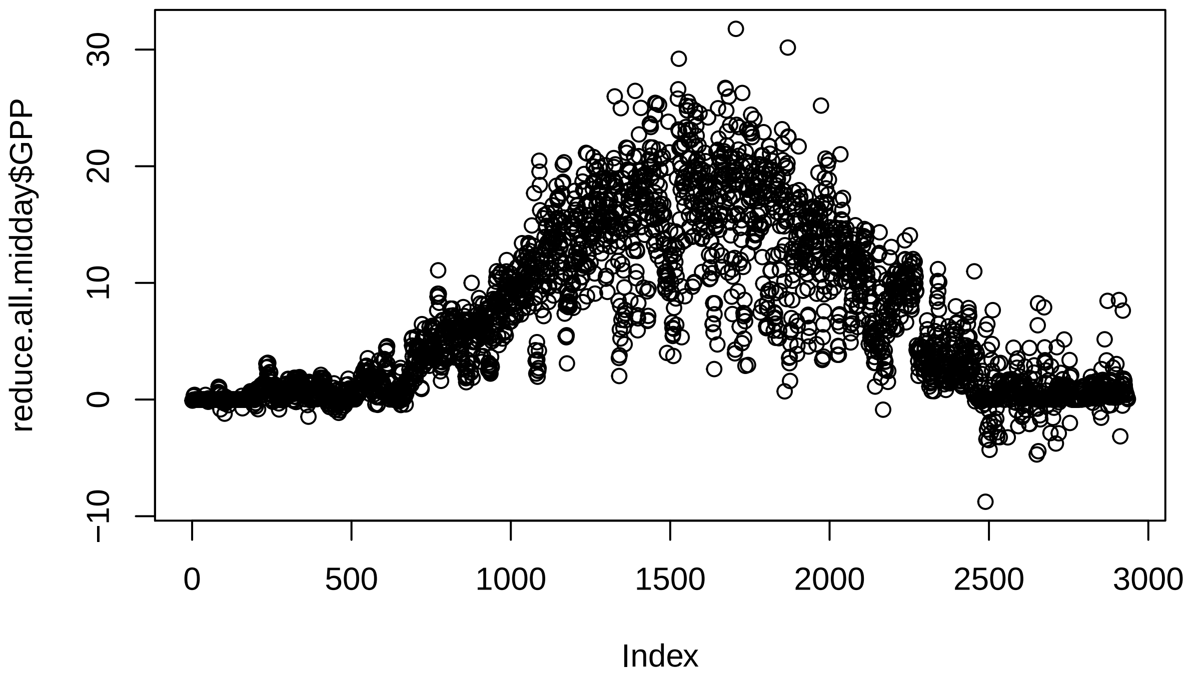
#3. Plotting using basic plot() function in r

##3.1 Line plot

The simplest plot of them all is the dot (or line) plot. The *plot* command is your friend here!

Let's see what our GPP data looks like:

```
plot(reduce.all.midday$GPP)
```

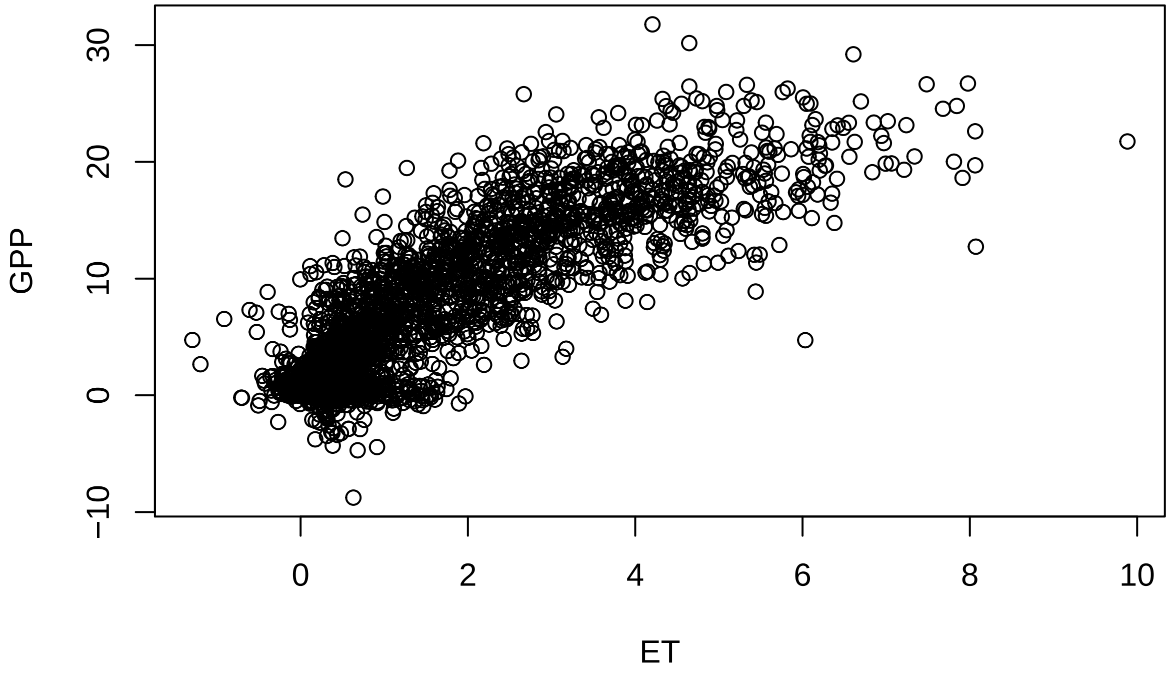


##3.2 Scatter plot

We can also use *plot* to plot the relationship between variables by making scatter plots. Use the ~ operator to achieve this e.g. *plot(A~B,data=data.AB)*, where *A* and *B* are our variables and *data.AB* is our dataframe that contains our variables.

Try to make a scatter plot between GPP and ET for our midday data:

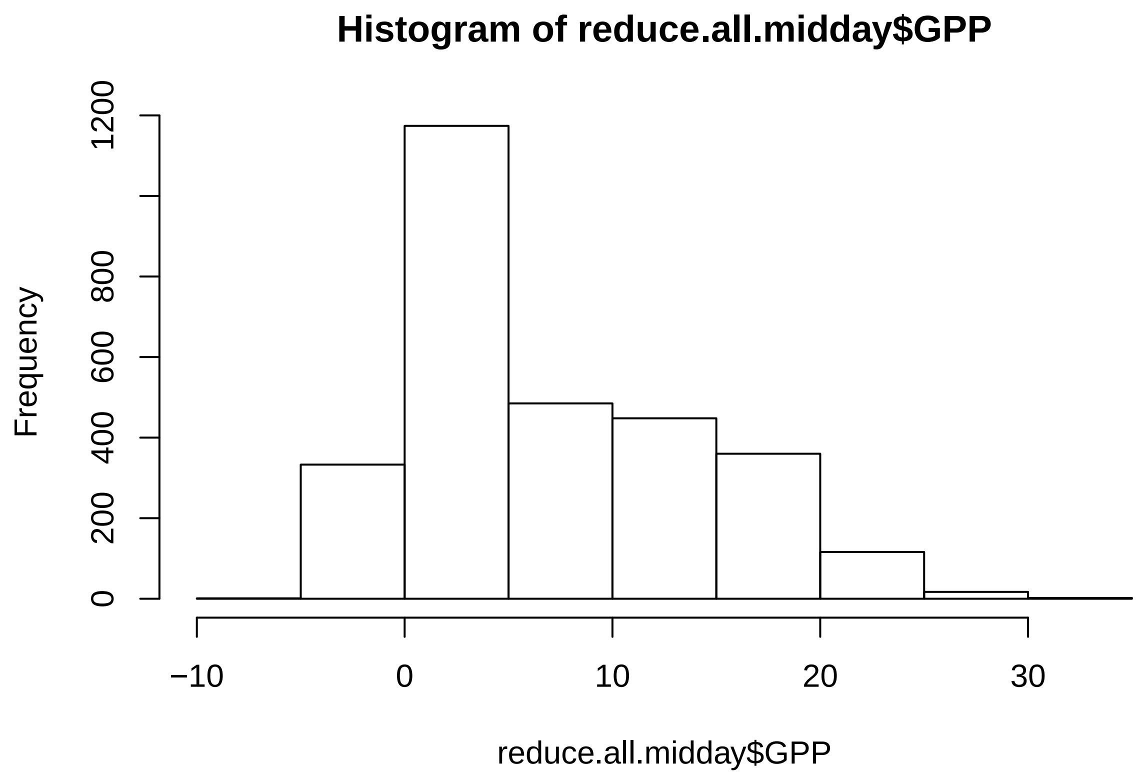
```
plot(GPP~ET,data = reduce.all.midday)
```



##3.3 histogram

Checking the distribution of your data is usually a very good idea! **hist** is used to draw histograms. How is our midday GPP distributed?

```
hist(reduce.all.midday$GPP)
```

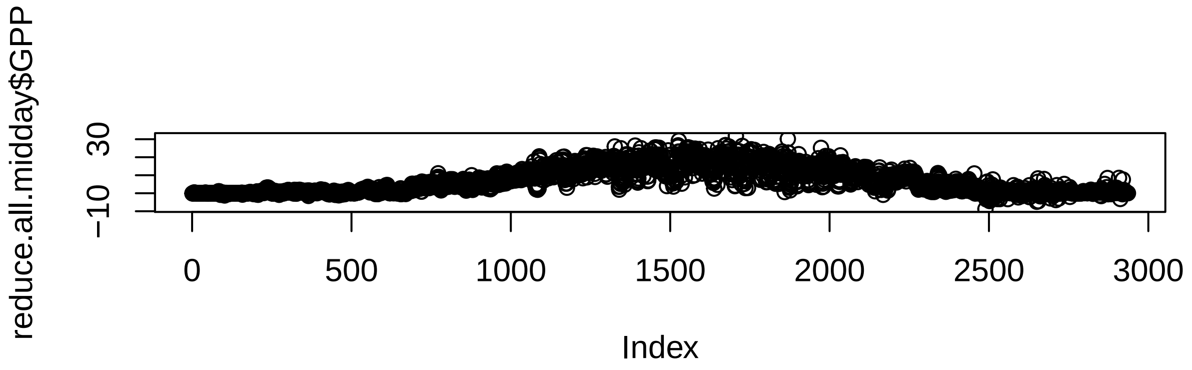


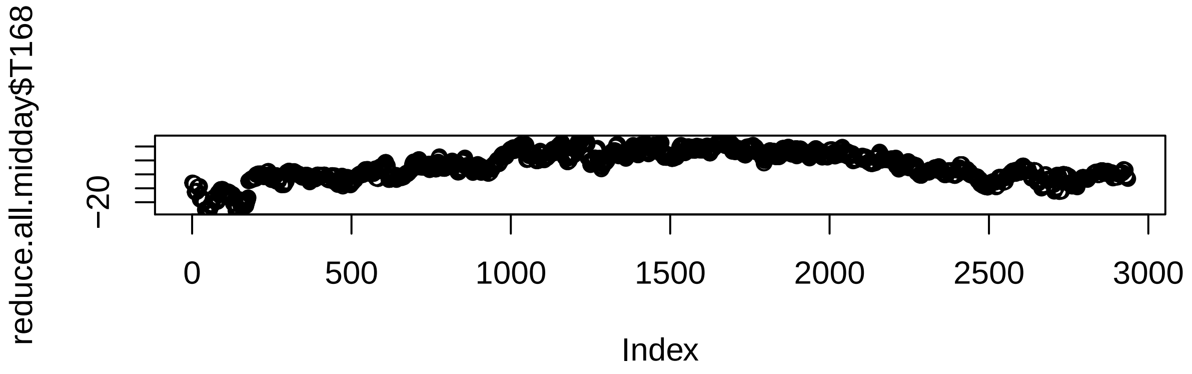
##3.4 Panels

Subplots (multiple plots in the same window) in R are achieved with the panels or *par* command. Specify the number of rows and columns as a two element vector and pass it using the *mfrow* key word as an argument to *par* e.g. par(mfrow =c(num.row,num.col)), then use repeated calls to *plot* in the usual way.

Can you complete the box below to draw ET and GPP in the same window but as separate subplots?

```
# first swap num.row and num.col for integers *par(mfrow =c(num.row,num.col))*
# then call plot() for each plot instance
par(mfrow=c(2,1))
plot(reduce.all.midday$GPP)
plot(reduce.all.midday$T168)
```



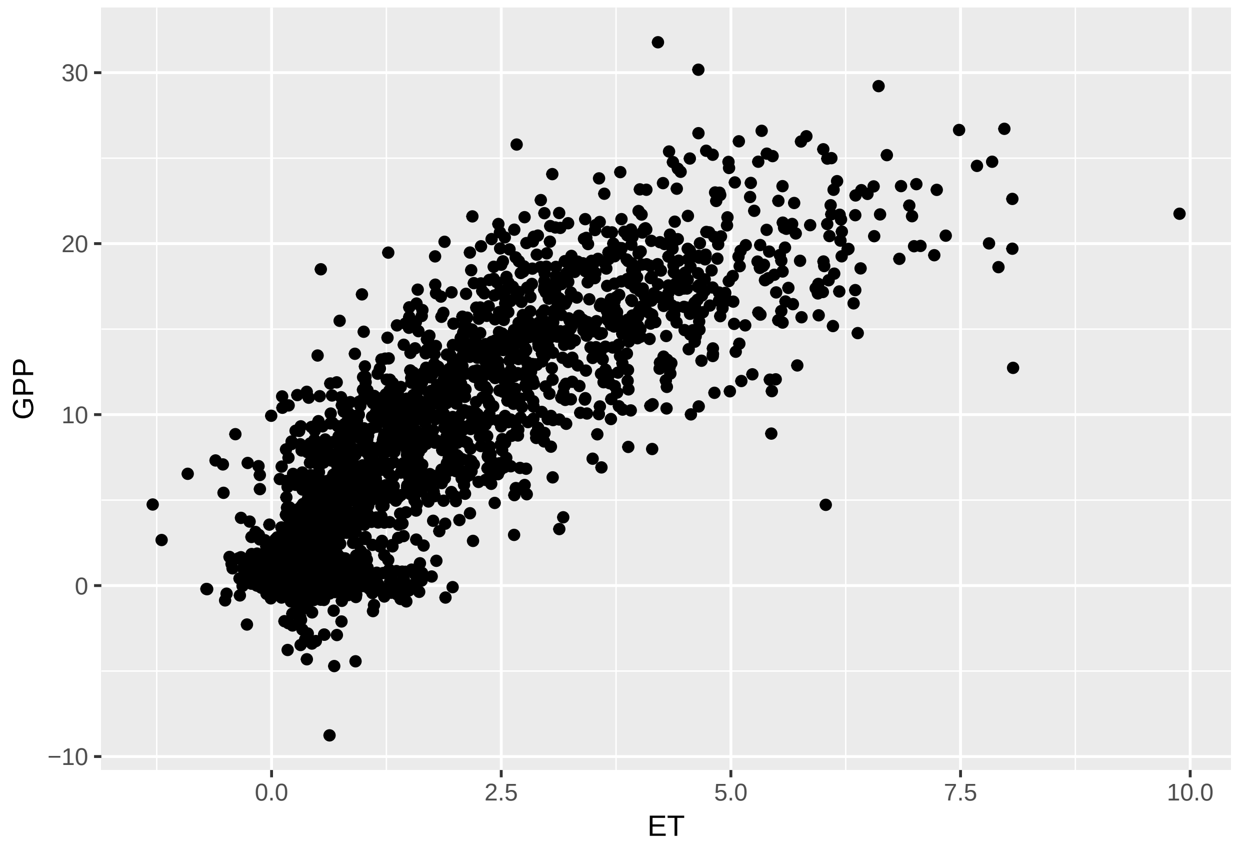


#3. Plotting using ggplot() function in ggplot2 packages

ggplot(data,aes(x=A,y=B))+ geom_point()+ geom_line()

Let's plot the scatter plot of GPP vs. ET. We can also assign the plot to a variable name, and later using ggsave() to save your graph.

```r
library(ggplot2)
p<-
  ggplot(reduce.all.midday,aes(x=ET,y=GPP))+
    geom_point()
p
```



save your graph

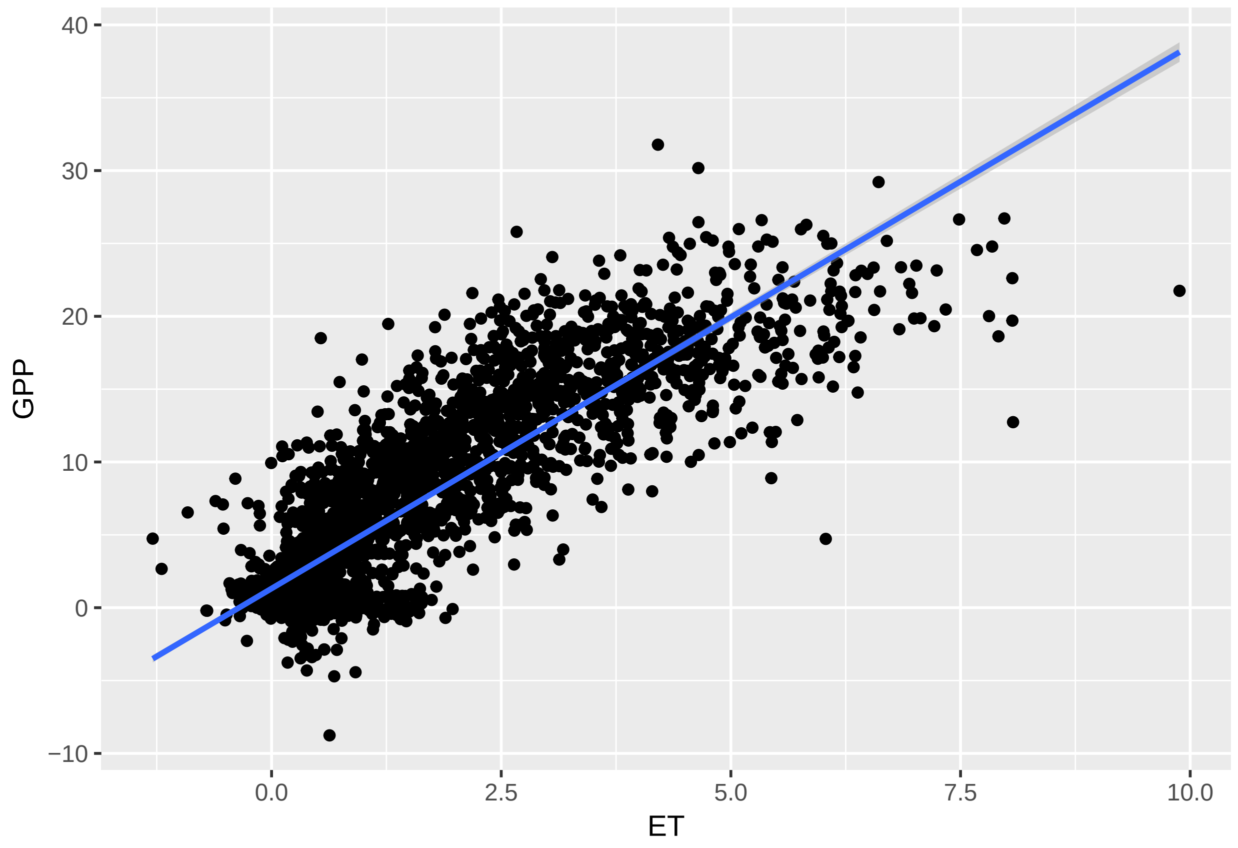
```
ggsave(p,filename = '../data/GPP_ET.png',dpi = 200,width=5,height=5)
```

Now let's add a regression line to the scatter plot and name the plot as *p2*

```
p2<-p+geom_smooth(method = 'lm')
```

Can you print the figure p2?

```
p2
```



Can you save the figure p2 into local folder?

```
ggsave(p2,filename = '../data/GPP_ET_lm.png',dpi = 200,width=5,height=5)
```