Appointment Prediction

Sebastian Milewski

2025-03-04

Schedule Optimization Documentation

This document exists as documentation to the Schedule Optimization projected worked on by Sebastian Milewski and Enyu Pan during F24-W25. This specific document was created to explain the predictive part of the schedule. It was created in R-code, for its statistical analysis tools. This half was specifically designed by Sebastian Milewski, with the advice and guidance of Sergio Garcia, and the consultation of Enyu Pan.

Goal of this project

The point of this project was to predict appointment demand for the TLC. This was done to create a Linear regression model to help find patterns in appointments and to help better guess when tutors are needed.

The way I have done this so far is by considering the following factors:

Time Day of the week Month.

This works well for same year comparisons, since the only factor missing in the year is 'term' (like winter, spring/summer, summer).

The following factors should be made to consider:

Location Subject Term Year.

Model Validity

I will go through the 4 main assumptions for SLR and talk about their validity and any related ideas/concerns.

Linearity We first need to note that most, if not all the factors are categorical factors. Thus, We don't 'require' it since we have binary variables.

Normality and Independence of errors This is easier to verify once we have worked through some of the data. I have run different data sets of the WCO data through this. Some of them have showed normal errors, while others do not.

One of the things that make this very hard to challenge is that I haven not built in a day skipper from when we're closed. THis can have weird skews in the data; and will ruin the normality of the errors.

One way to get around that would be to delete times/days where there is 0 appointments - but you might overshoot and start deleting accurate times. I have no actual 'solution' for this problem that is efficient; they would all require some weird work around.

Once you have data, there are many different ways to test this- residual plots and qq plots being the most common and useful.

Equal Variance Again, this is something that can be checked with residual plots and residual qq plots.

Having as many factors as I recommend should help remove as much as the unequal variance as possible. However, I will note that adding a time factor (that is, we could use the index or something similar to note time moving forward) could yield a strong p-value, simply because (in theory) the TLC would get more and more popular over time. Thus, this means that there could be a small inflation factor where variance slowly become more positive (or larger).

Explanation of Code:

Now, I will start going the code by chunks, explaining the larger detailed points.

```
setwd("SET FILE PATH")

##This is an exemplar way to break it down; splitting it into subject and
##Location already.
BigData<- read.csv("FILE NAME")

MediumData <- subset(BigData,Schedule.Title == "Online Tutors - Winter 2024")
Data <- subset(MediumData, Focus == "English")

##To just import the data, you can do:
#Data <- read.csv("FILE NAME")

start <-  "2/18/2024"
end <-  "8/10/2024"
```

Introductory Section: So the start of the code file is the first 2 parts: We import the data; which allows us to access it.

NOTE: YOU NEED TO REPLACE THIS WITH YOUR FILE PATH AND FILE NAME. IT WILL NOT WORK OTHERWISE

Then, we import the data and I subset the data twice: Once for location and once for subject. This was mostly for testing purposes,This could (and should) be redone to be a little cleaner.

Then, we have start and end which are just storage for days; so that we can generate the list of days we want to use to predict. IT HAS TO BE IN THE "DD/MM/YYYY" FORMAT- it might not work elsewise

```
##String equality function
StrEqu <- function(str1, str2){
  ##We start of by setting the default to "false"
  state <- FALSE
  ##First we check if the strings are the same length
  if(!((length(str1)==(length(str2))))){
    return(state)
  }
  else{
    ##Now, we iterate through the strings checking if each element is the same.
```

```
    for(i in 1:length(str1)){
      if (str1[i]==str2[i]){
        ##If they are the same, we set the state to true and continue
        state<- TRUE
      }
      else{
        ##If they are different, we set the state to false, and end the loop.
        state<- FALSE
        break
        }
    }
    ##We then just return the state of equality.
    return(state)
  }
}
```

String Equality I built a string equality to use. First it checks if the strings are of equal length, and then it goes through all the entries checking if they are the same.

```
##Checks if an appointment is active
ongoing<- function(time1, time2, time3){
  ##Time1 is the current time, time2 is the start of an app, time3 is the end

  ##There is a lot of string parsing for this to work properly, we end up
  ##Having to set up each string slowly and then change them individually.

  ##setting up the broken up strings
    time1.1<-unlist(strsplit(time1, split = ''))
    time2.1<-unlist(strsplit(time2, split = ''))
    time3.1<-unlist(strsplit(time3, split = ''))
    ##Grabbing the lengths of each string for easier storage
    len1 <- length(time1.1)
    len2 <- length(time2.1)
    len3 <- length(time3.1)

    time1.2<- c()
    time2.2<- c()
    time3.2<- c()

    ##Changing times from 12 hour to 24 hour for easier comparison. Note that
    ##00 time will be represented by 24

    ##parsing time1
    ##First check out if it is an AM time
    if(StrEqu(time1.1[(len1-1):len1], c("A", "M"))){
      ##If it is an AM time; take time 12AM to 00:00
      if(StrEqu(time1.1[1:2], c("1", "2"))){
        time <- time1.1[1:2]
        newtime <- "0"
        time1.2 <- c(newtime, time1.1[3:(len1-2)])
      }
```

```
  else{
    ##Elsewise; just take off the Am
    time1.2 <- time1.1[1:(len1-2)]
  }
}
else{
  ##Checks if its twelve; if it is just leave it at 12
  if(StrEqu(time1.1[1:2], c("1", "2"))){
  time1.2 <- time1.1[1:(len1-2)]
}
else{
  ##
  time <- time1.1[1]
  newtime <- as.character(as.integer(time)+12)
  time1.2 <- c(newtime, time1.1[2:(len1-2)])
}}
time1.2<- paste(time1.2, collapse = "")


##Parsing time2
##First check out if it is an AM time
if(StrEqu(time2.1[(len2-1):len2], c("A", "M"))){
  ##If it is an AM time; take time 12AM to 00:00
  if(StrEqu(time2.1[1:2], c("1", "2"))){
    time <- time2.1[1:2]
    newtime <- "0"
    time2.2 <- c(newtime, time2.1[3:(len2-2)])
  }
  else{
    ##Elsewise; just take off the Am
    time2.2 <- time2.1[1:(len2-2)]
  }
}
else{
  ##Checks if its twelve; if it is just leave it at 12
  if(StrEqu(time2.1[1:2], c("1", "2"))){
    time2.2 <- time2.1[1:(len2-2)]
  }
  else{
    ##
    time <- time2.1[1]
    newtime <- as.character(as.integer(time)+12)
    time2.2 <- c(newtime, time2.1[2:(len2-2)])
  }}
time2.2<- paste(time2.2, collapse = "")

##parsing time3
if(StrEqu(time3.1[(len3-1):len3], c("A", "M"))){
  ##If it is an AM time; take time 12AM to 00:00
  if(StrEqu(time3.1[1:2], c("1", "2"))){
    time <- time3.1[1:2]
    newtime <- "0"
    time3.2 <- c(newtime, time3.1[3:(len3-2)])
```

```
    }
    else{
      ##Elsewise; just take off the Am
      time3.2 <- time3.1[1:(len3-2)]
    }
  }
  else{
    ##Checks if its twelve; if it is just leave it at 12
    if(StrEqu(time3.1[1:2], c("1", "2"))){
      time3.2 <- time3.1[1:(len3-2)]
    }
    else{
      ##
      time <- time3.1[1]
      newtime <- as.character(as.integer(time)+12)
      time3.2 <- c(newtime, time3.1[2:(len3-2)])
    }}
  time3.2<- paste(time3.2, collapse = "")


  ##end of string parsing


  ##Converting from string to a vector of floating points.
  time1.3 <- as.integer(unlist(strsplit(time1.2, split = ":")))
  time2.3 <- as.integer(unlist(strsplit(time2.2, split = ":")))
  time3.3 <- as.integer(unlist(strsplit(time3.2, split = ":")))

  ##Converting it from 2 separate numbers; we change it to store as purely hour.
  time1.4 <- time1.3[1]+ time1.3[2]/60
  time2.4 <- time2.3[1]+ time2.3[2]/60
  time3.4 <- time3.3[1]+ time3.3[2]/60


  ##Checks if the current time is during the appointment.
  if((time2.4<= time1.4)&(time1.4<time3.4)){
    return(TRUE)
  }
  else{
    return(FALSE)
  }
}
```

Ongoing appointment check This function takes in 3 times. Time 1 and Time 2 are the start and end of appointments, while time 3 is the 'current' time. It then goes through and checks if the appointment is considered ongoing. This is used to check if the time given is during the appointment.

```
##This function takes in a date from WCO and converts it to a date in R.
dateConvert<- function(date){
  ##First we split up the original string
  newdate<-unlist(strsplit(date, split = "/"))
```

```
  ##Now we reorder it to fit R's standards
  return<- c(newdate[3], newdate[1], newdate[2])
  #Recreate it as a string.
  return(paste(return, collapse = "-"))
}
```

Date format converter Takes an input date from WCO and converts it into the standard date in R; but keeps the data type as CHA.

```
##Converts the time given into a 24hour format
TimeSwitch <- function(t){

  ##setting up everything we need, the list of characters, the length, AM vs PM
  char<- unlist(strsplit(t, split = ""))
  last<- length(char)
  daytime<- t[last-1:last]

  ##Now we check if it is AM or PM, and add 12 if necessary.
  comb<- 0
  if(StrEqu(char[last-1:last],"PM")&(StrEqu(char[1:2], "12"))){
    comb <- as.character(as.integer(char[1])+12)
    char <- paste(comb, char[2:last], sep= "")
  }
  newtime<- comb
  return(newtime)
}
```

Time format converter This function changes the time from the WCO format (HH:MM _M) (with it filled with A or P) to a 24 hour format (HH:MM).

```
##Generates a sequence of dates.
dateseq<- function(start, end){
  seq(as.Date(start), as.Date(end), by = "days")
}
```

Date generation This uses the start and end date and generates a vector of dates of every day between.

Note that this does include days we are not open, so eventually we would need to build a function that filters out the days that we are not open.

Counting appointments

```
##Counts the amount of appointments that occur during said time/day
countappointments<- function(date, time){

  count<- 0
  ##We Loop through the appointment Data looking to see if the appointment date
  ##is the same, and then we check if the time is ongoing in the appointments
```

```
  ##that are on the same day.
  for(k in 1:length(Data$Schedule.Title)){
    if(StrEqu(as.character(as.Date(dateConvert((Data$Appointment.Date[k])))),
              as.character(date)) &
       (ongoing(time, Data$Start.Time[k], Data$End.Time[k]))){

      ##If it is, we add 1 to the count
      count<- count + 1
    }
  }
  return(count)
}
```

The way that we count appointments during each slot is that we slowly go through each appointment and check if it is on the day that we are counting and then checking if it is ongoing in the recorded appointment time.

```
createdataarray<- function(dates, times){
  ##We set up our empty lists for our loop
  AppointmentSlot<- c()
  AppointmentTime<- c()
  AppointmentDate<- c()
  Month<- c()
  Amount<- c()
  Weekday<- c()

  ##We double loop day and time, creating new entries for each one, with the
  ##following information in each one:
  for(i in 1:length(dates)){
    for(j in 1:length(times)){
      ##the date and time as one unit as the "name"
      newdate<- paste(dates[i], times[j], sep = " ")
      AppointmentSlot <- c(AppointmentSlot, newdate)
      ##The time of the appoinment
      AppointmentTime <- c(AppointmentTime, times[j])
      ##the Day of the appointment
      AppointmentDate <- c(AppointmentDate, dates[i])
      ##The weekday of the appointment (like Mon, Tues, etc.)
      Weekday<- c(Weekday, weekdays(dates[i]))
      ##The month of the appointment
      Month <- c(Month, format(dates[i], "%m"))
      ##The amount of appointments that were in that time slot in the data.
      Amount<- c(Amount, countappointments(dates[i], times[j]))
    }
  }
  ##we return a data frame of the recorded information.
  return(data.frame(AppointmentSlot, AppointmentTime, Month,
                    Weekday, Amount))
}
```

Creating a new dataframe that makes sure this works well This function first pregenerates empty lists for us to store the new explanatory variables.

Then, what we do is we loop through each date and time adding the appointment slot as the name (time and day), add the date and time separately, also adding what day of the week, what month it is, and then the amount of appointments that occurred during that time.

Ways to work on this

Immediate things to fix: First, building a way to filter focuses (or subjects) would be useful, and it would be good to also have one for location.

On top of this, there needs to be a way to filter days and times when we generate the list of times and days that are allowed in the code. ~~~insert function ~~~. this is because currently, it generates 9pm-9pm everyday for all days. This is a problem since on weekends, we are only open 10am-6pm, and for times between 9-10am and 5-9pm we have different times available for online to in person. Alongside this, for the more fringe campuses like 341 and TMU we would also have that it is only open on certain days so we could limit the days.

This should be done for a couple reasons:

1) Cleans up computations

If we add a lot of extra times and days, we then have that we are lengthening the run time. This would help fix up efficiency and would reduce memory requirements, while also removing points of error as we have less data points to verify.

2) Smooths out modelling

Since the data points we are removing are specifically times that we are closed, we are removing points where there will be 0's and are outside the scope of the model, we are helping remove sources of errors and helps increase the significance of the model.

It also can help remove some unnecessary parameters, which helps simplify the model.

3)Removes some of the need to study interactions

Since we have less parameters, it removes need to study interaction terms for when we have relations for hours beyond in person openings, and would help remove some of the weird interactions over weekends.

Long term edits:

1) The code that counts appointment is $(n \times m)$ length in run time. This is pretty inefficient.

One way that you could implement a faster algorithm would be to use a binary search algorithm to take an appointment to find it's appointment time and day and add it to the tally. this would then run in n*log(m), which for larger data pools would be significantly faster.