FIB: Probabilitat i Estadística

Quina shell és la més ràpida?

Rubén Catalán, Ismael El Basli i Muhammad Yasin Khokhar

19 de desembre de 2022

1 Introducció

Com a estudiants d'enginyeria informàtica, estem molt acostumats a treballar dintre d'una shell, que es tracta de l'intèrpret de comandes, un programa que proveeix una interfície d'usuari per accedir als serveis del sistema operatiu. La qüestió és que no només n'existeix una, sinó que hi ha una gran variabilitat en quant a opcions possibles. Això ens va fer preguntar-nos si totes són igual de ràpides o realment hi ha diferències de velocitat considerables. Cal deixar clar que contemplem només les shells de sistemes operatius basats en UNIX, com ara Linux o MacOS. Per exemple, als laboratoris de la FIB treballem amb la *tcsh* shell, mentre que la *bash* shell és la més popular en la majoria dels sistemes Linux i, de fet, normalment és la predeterminada per les contes d'usuari. Llavors, amb quina shell ens quedem?

1.1 Objectius

1. Ser capaços d'argumentar quina shell és la més ràpida de totes o si realment hi ha diferències mínimes en quan a velocitat, tot en base a càlculs estadístics.

2. Veure si hi ha certa relació lineal o no entre el temps d'usuari i el temps de sistema que s'usa per executar els programes que emprarem per evaluar les velocitats de les shells.

3. Poder consolidar tot el que s'ha après a l'assignatura, aplicant-ho a tot el càlcul estadístic que implica aquest petit projecte.

4. Millorar la nostra habilitat utilitzant R, un llenguatge de programació enfocat a l'anàlisi estadístic.

5. Com a objectiu més secundari, que no té gaire a veure amb l'assignatura, aprendre LaTeX, que és un sistema de composició de textos orientat a la creació de documents escrits que presentin una alta qualitat tipogràfica, i amb el que està fet aquest document.

En quant a aquests objectius, després d'haver reflexionat, són totalment factibles i estan a l'abast dels nostres recursos i possibilitats, ja que simplement haurem de fer mesures del temps que triguen certes shells entre l'inici i el final de determinats programes.

2 Variables

L'estudi presenta la variable explicativa S $\equiv$ "Tipus de shell" i el seu univers Ω és:

- **Bourne Shell** (sh): Intèrpret de comandes interactiu. Inclou característiques com el llistat, lectura i protecció de fitxers, creació i eliminació de processos, etc.
- **Almquist shell** (ash): Escrita per Kenneth Almquist com un reemplaç del shell Bourne a les versions BSD de Unix.
- **Bourne-Again shell** (bash): Escrita pel projecte GNU, serveix com a superconjunt d'instruccions basades en l'intèrpret del que origina, Bourne.
- **Debian Almquist Shell** (dash): Conté menys comandes que Bash, però requereix de menys llibreries. Algunes de les funcionalitats faltants són requerides per POSIX.
- **Korn Shell** (ksh): A més d'intèrpret, serveix com a llenguatge de programació, gràcies a les seves funcions per administrar arxius de comandes.
- **Z shell** (zsh): Dissenyada per us interactiu com a subconjunt d'altres shells com sh, bash, ksh, etc.
- **TENEX C shell** (tcsh): Shell amb un llenguatge de comandes similar a C que serveix com a reemplaç de C Shell gràcies a característiques com la autocompleció de noms.
- **Friendly Interactive Shell** (fish): Considerada una shell "exòtica" ja que la seva sintaxis no prové de cap altra shell. Pretén ser una shell "user-friendly".

El motiu pel qual s'han escollit aquestes shells concretes de totes les que existeixen, és perquè són les que s'usen més habitualment i, a més, són les que per opinió popular podrien arribar a ser les més ràpides. És evident que no és factible incloure totes les shells de UNIX, ja que ens extendriem massa i probablement les shells que no contemplem en aquest projecte no arribarien a ser les més ràpides. Per una banda, sis de les shells incloses (sh, ash, bash, dash, ksh, zsh) són compatibles amb Bourne Shell, tcsh és compatible amb la shell de C i fish és una shell una mica més exòtica que hem volgut incloure degut a que hi ha una opinió generalitzada que diu que es bastant ràpida.

Una altra variable explicativa en aquest estudi és P $\equiv$ "Tipus de programa que s'executa" i el seu univers Ω és:

- Ordenació de vectors: Es tracta d'un shell script (.sh) que utilitza un quicksort per ordenar un vector desordenat de 10000 elements. Aquest vector és sempre el mateix, per evitar afavorir a alguna shell. El codi del programa està disponible a l'ANNEX.

- Partida joc EDA: Es tracta de l'execució d'una partida del joc de l'assignatura Estructures de Dades i Algoritmes on nosaltres hem de programar l'intel·ligència d'un jugador. Les partides s'executen amb un jugador anomenat PEplayer, que no fa res de forma aleatòria. A l'ANNEX hi haurà disponible un enllaç per descarregar el codi font del joc i el codi de l'intel·ligència de PEPlayer.

En quant a aquesta variable explicativa, cal destacar certs aspectes. En un principi, preteníem que hi haguessin 4 tipus de programes diferents, els ja mostrats i dos més: comanda *tree /* i un programa de gestió de processos fet en bash script que ja teníem creat. El principal problema que ens va portar això va ser que els temps d'execució d'aquests altres dos programes diferien bastant respecte els altres i això, juntament amb les poques execucions per programa a cada shell, ens va portar problemes a l'hora d'assegurar la premissa de normalitat per realitzar l'anàlisi estadístic. L'objectiu que teníem en ment era que els programes no impliquessin molta desviació cap a un tipus de gestió concreta per part de la shell, ja que l'objectiu principal és intentar saber quina és la shell més ràpida en termes globals. Això és complicat, i més endevant es discutirà.

Ja passant a la variables de resposta, la primera i principal es tracta de T ≡ "Temps que triga una shell en executar un programa, en segons". És evident que aquesta variable és mesurable, ja que disposem de diverses eines que ens permeten mesurar el temps entre l'inici i el final de l'execució d'un programa, que són adequades per llegir els valors en qüestió. En concret, l'instrument que s'usarà pel càlcul dels temps que triguen certes shells a executar determinats programes serà la comanda */usr/bin/time* de UNIX.

Les altres dues variables de resposta són el "Temps d'usuari que consumeix l'execució d'un programa a una shell" i "Temps de sistema que consumeix l'execució d'un programa a una shell". El valor d'aquestes també és fàcilment obtenible amb la comanda */usr/bin/time* mencionada abans. Ens permetran veure si hi ha o no relació entre el temps de sistema i el d'usuari, almenys en els programes que estudiarem.

3 Pla de Recollida i procés de mostreig

Pel que fa al nombre de dades que recollirem, seran 384. Aquesta n ve de 8 shells x 2 programes x 8 execucions x 3 (temps total, de sistema i d'usuari) = 384. Considerem que aquest nombre de dades a recollir és suficient per arribar a complir els nostres objectius. Totes aquestes dades les recollirem en un computador amb arquitectura AMD: Ryzen 5 3500U a 2.1GHz, 8GB de RAM i amb sistema operatiu Debian GNU/Linux 11 bullseye. Totes les execucións es duran a terme a Kitty, que es tracta d'una de les terminals més ràpides actualment. Un punt molt important i que, de fet, és premissa indispensable per l'anàlisi estadístic és l'aleatorietat de les mostres. A la nostre recollida de dades ens assegurem d'això esborrant la memòria cachè abans de cada execució, mitjançant l'execució en mode root de la comanda "sync & & echo 3 >/proc/sys/vm/drop_caches". A més, en els programes que usem no hi ha cap aspecte aleatori que pugui afavorir a una shell en concret en algún cas. De fet, també realitzem totes les execucions amb

l'ordinador carregant-se, ja que el computador es tracta d'un portàtil, i no volem que el nivell de bateria afecti al rendiment, i per conseqüent a les nostres mostres.

4 Resultats

4.1 Temps total per cada shell

La mitjana mostral del temps total d'execució (en segons) de cadascuna de les shells ha sigut:

- sh: 3.56125
- ash: 3.563125
- bash: 3.56
- dash: 3.569375
- ksh: 3.57125
- zsh: 3.536875
- tcsh: 3.541875
- fish: 3.55625

A la Figura **??** es pot observar un boxplot del temps total a cada shell. D'esquerra a dreta: sh, ash, bash, dash, ksh, zsh, tcsh i fish.

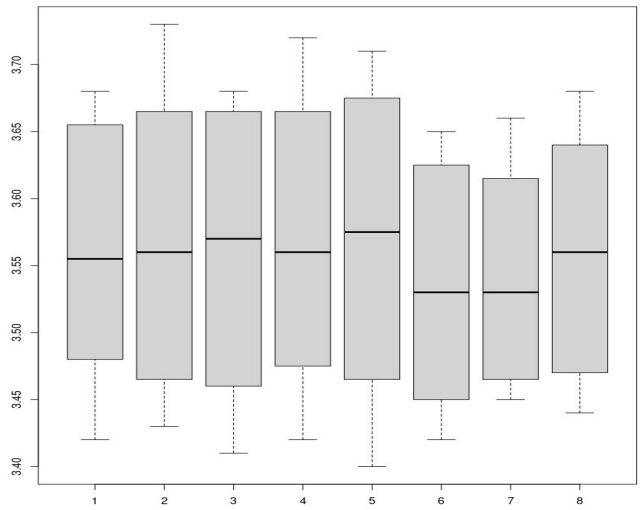
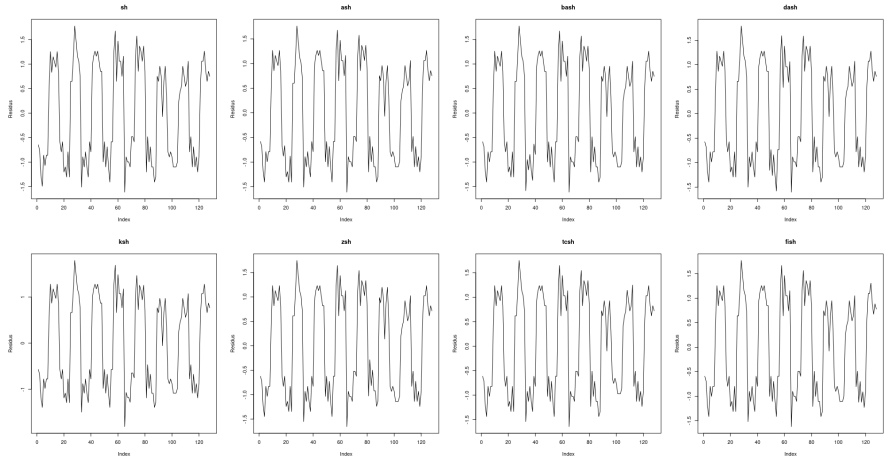


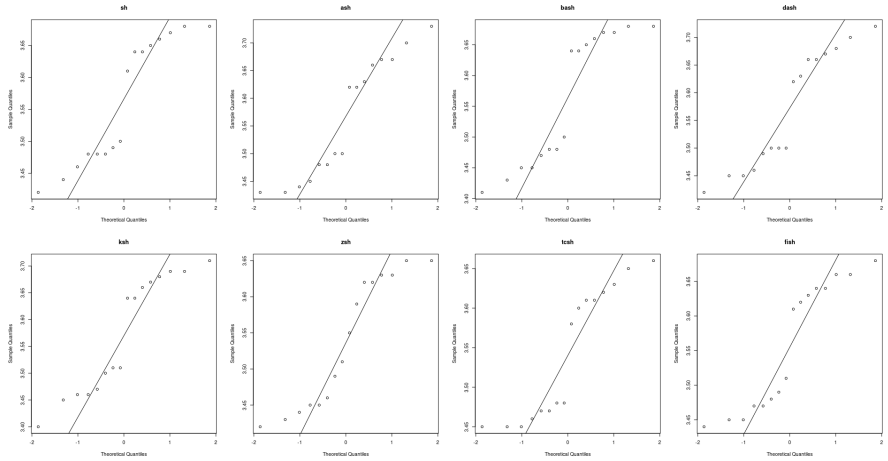
Figura 1: Boxplot del temps total a cada shell

Per poder comparar el paràmetre poblacional μ del temps total d'execució entre totes les shells, que són mostres independents, podem usar el model estadístic: $Y = \mu + \upsilon_k + \epsilon$. El model contempla μ com a mitjana de referència i υ_k com a canvi de la mitjana del grup k.

Per poder usar aquest model hi ha certes premises. Primerament que la mostra sigui aleatòria, cosa que ja s'ha argumentat anteriorment. A més ja es veu com no es segueix cap patró a les dades obtingudes, com es pot veure a la Figura **??**. També es requeriex normalitat, cosa que es compleix per totes les shells, com es pot veure a la Figura **??**. Finalment, també hi ha variabilitat semblant entre els grups.



(a) Independència



(b) Normalitat

Figura 2: Premisses

Es poden arribar a apreciar dos subgrups de punts a la Figura **??**, que corresponen als temps de cadascun dels dos programes pel testeig.

Ara, ja demostrades les premisses del model, podem emprar la comanda lm de R per poder inferir les mitjanes de temps:

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.563125   0.024870 143.270   <2e-16 ***
SHELLbash   -0.003125   0.035171  -0.089    0.929
SHELLdash    0.006250   0.035171   0.178    0.859
SHELLfish   -0.006875   0.035171  -0.195    0.845
SHELLksh     0.008125   0.035171   0.231    0.818
SHELLsh     -0.001875   0.035171  -0.053    0.958
SHELLtcsh   -0.021250   0.035171  -0.604    0.547
SHELLzsh    -0.026250   0.035171  -0.746    0.457
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.09948 on 120 degrees of freedom
Multiple R-squared:  0.01399,    Adjusted R-squared:  -0.04353
F-statistic: 0.2432 on 7 and 120 DF,  p-value: 0.9735
```

Figura 3: Sortida comanda lm

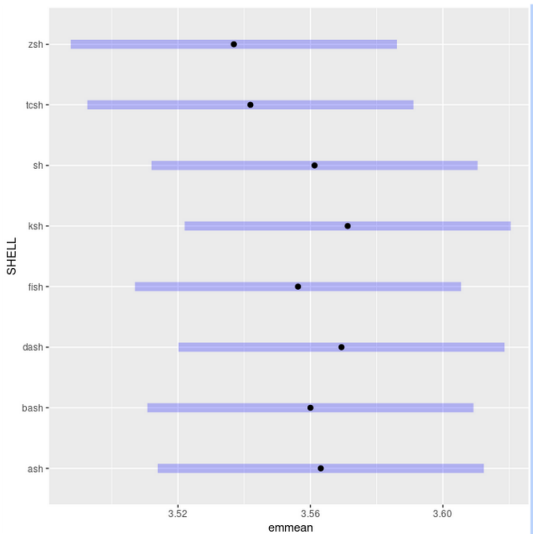
A partida de la sortida de la comanda lm que es pot observar a la Figura **??**, es poden obtenir les estimacions de les 8 μ poblacionals, en segons:

- sh: 3.563125 - 0.001875 = 3.56125
- ash: 3.563125
- bash: 3.563125 - 0.003125 = 3.56
- dash: 3.563125 + 0.006250 = 3.569375
- ksh: 3.563125 + 0.008125 = 3.57125
- zsh: 3.563125 - 0.026250 = 3.536875
- tcsh: 3.563125 - 0.021250 = 3.541875
- fish: 3.563125 - 0.006875 = 3.55625

Per altra banda, també podem obtenir els intervals de confiança, amb un nivell de confiança del 95%, per cadascun dels paràmetres, mitjançant la sortida de la comanda en R confint, com es pot veure a la Figura **??**. Per altra banda, també es poden veure els intervals de confiança de forma gràfica mitjançant la comanda emmeans de R, com es pot veure a la Figura **??**.

	2.5 %	97.5 %
SHELLash	3.513884	3.612366
SHELLbash	3.510759	3.609241
SHELLdash	3.520134	3.618616
SHELLfish	3.507009	3.605491
SHELLksh	3.522009	3.620491
SHELLsh	3.512009	3.610491
SHELLtcsh	3.492634	3.591116
SHELLzsh	3.487634	3.586116

(a) Sortida comanda confint



(b) Sortida comanda emmeans

Figura 4: Intervals de confiança

Després d'haver mostrat tots els resultats es deduïble que, més enllà que una shell té una mitjana menor a les demés, pràcticaments no hi ha diferències. Tot això es discutirà més endevant.

4.2 Relació entre temps d'usuari i de sistema

Per poder veure si existeix algun tipus de relació entre el temps d'usuari i el de sistema, es pot usar el model estadístic lineal $Y = \beta_0 + \beta_1 X + \epsilon$, on Y representaria el temps d'usuari i X el temps de sistema. Igual que a l'apartat anterior, per poder usar aquest model estadístic per analitzar les nostres dades, cal demostrar certes premisses. En aquest cas són 4:

- Mostra aleatòria
- Linealitat entre Y i X: Y i X s'han de poder ajustar a una recta
- Normalitat als residus
- Independència als residus: Els residus no han de seguir cap patró
- Homoscedasticitat als residus: Hi ha d'haver variabilitat semblant entre els grups.

Com que vàrem tenir problemes a l'hora d'assegurar aquestes premisses per poder realitzar l'estudi, ens vam veure obligats a separar l'anàlisi per programa.

4.2.1 Ordenació de vectors

Pel cas de l'ordenació de vectors, a la Figura **??**, de dalt a baix i d'esquerra a dreta s'observa la premissa de homoscedasticitat, normalitat dels residus, linealitat i independència . A més, l'aleatorietat de les mostres es compleix pel que s'ha argumentat en apartats anteriors.

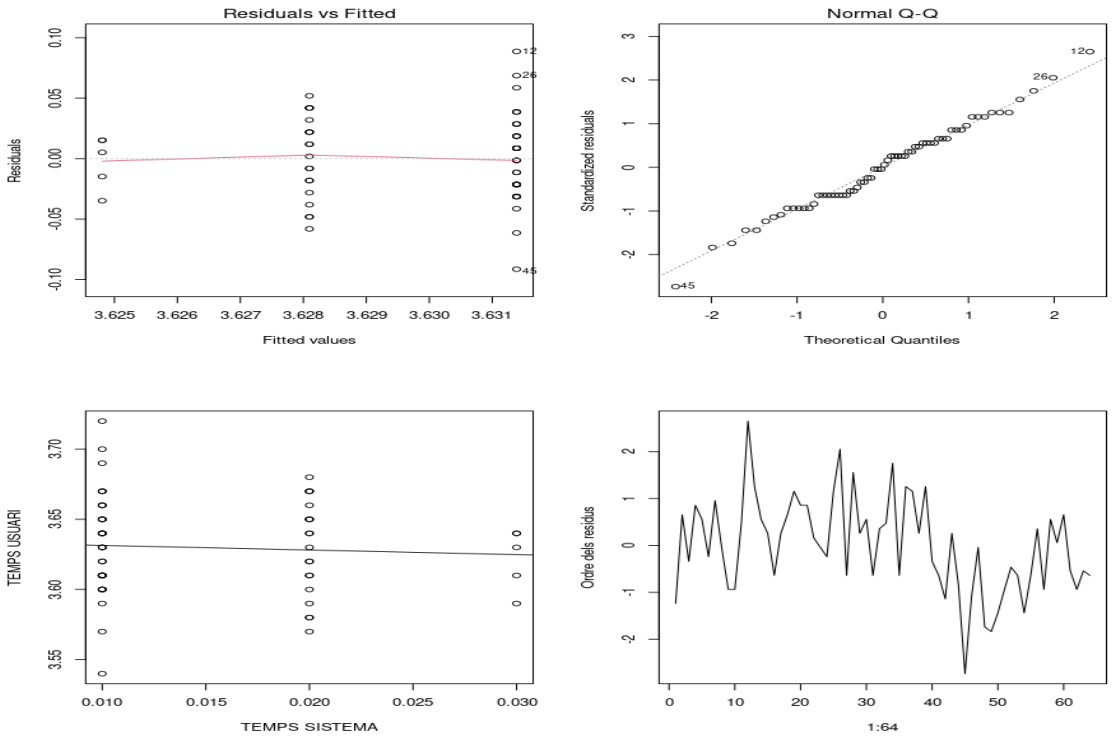


Figura 5: Premisses

Tot i que són poques dades, res s'oposa a validar cap de les 4 premisses. Ara, podem usar la comanda lm per poder veure com és la recta a la que s'ajusten el temps d'usuari i de sistema al programa d'ordenació de vectors, per poder inferir quin tipus de relació hi ha entre ells:

A la Figura **??** es pot veure clarament com l'R-squared, que es tracta del coeficient de determinació, pren un valor molt proper a 0, pel que X no determina per res la variació de Y. És a dir, el temps d'usuari no influeix en com varia el temps de sistema. Per aquest motiu, no té sentit parlar de la recta que relacions ambdues variables que ens dona la comanda lm de R.

```
Coefficients:
                Estimate  Std. Error   t value   Pr(>|t|)
(Intercept)     3.63467   0.01064    341.672  <2e-16 ***
TEMPS_SISTEMA  -0.32892   0.66454     -0.495   0.622
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.03384 on 62 degrees of freedom
Multiple R-squared:  0.003936,    Adjusted R-squared:  -0.01213
F-statistic: 0.245 on 1 and 62 DF,  p-value: 0.6224
```

Figura 6: Sortida comanda lm

4.2.2 Partida EDA

Pel cas de la partida d'EDA, a la Figura **??**, de dalt a baix i d'esquerra a dreta s'observa la premissa de homoscedasticitat, normalitat dels residus, linealitat i independència . A més, l'aleatorietat de les mostres es compleix pel que s'ha argumentat en apartats anteriors.

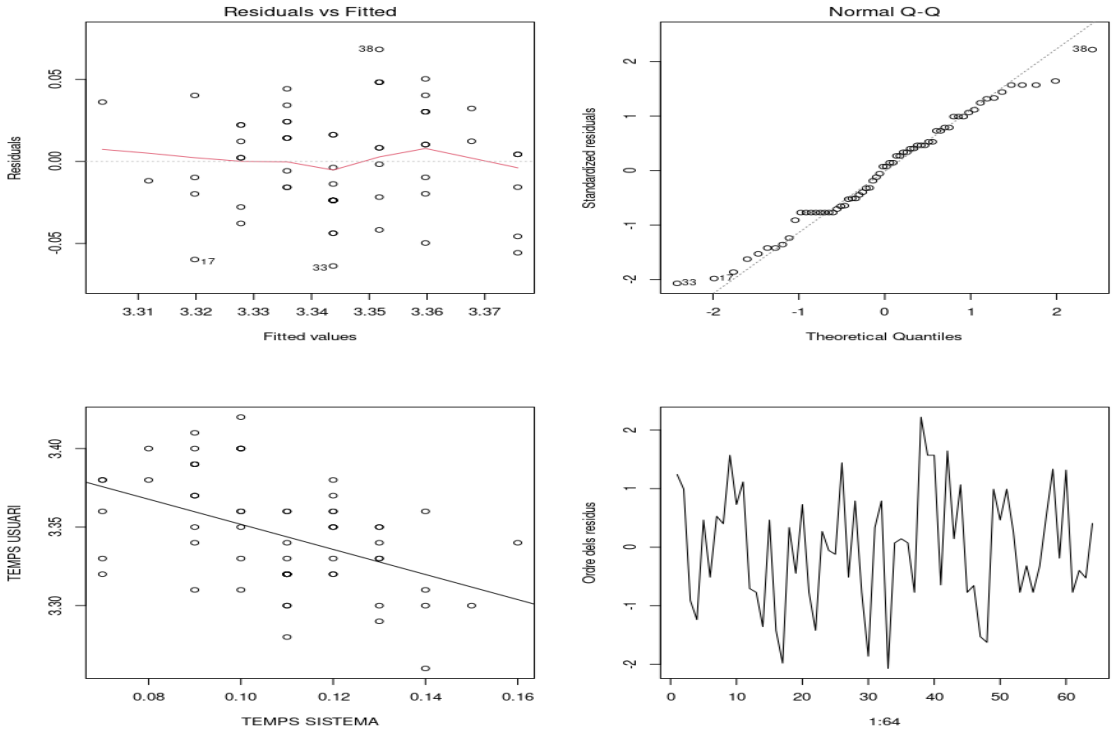


Figura 7: Premisses

Igual que abans, tot i que són poques dades, res s'oposa a validar cap de les 4 premisses.
Ara, podem usar la comanda lm per poder veure com és la recta a la que s'ajusten el temps d'usuari i de sistema al programa d'ordenació de vectors, per poder inferir quin tipus de relació hi ha entre ells:

```
Coefficients:
             Estimate   Std. Error  t value  Pr(>|t|)
(Intercept)   1.01912    0.21943    4.644   1.82e-05 ***
TEMPS_USUARI -0.27225    0.06559   -4.151   0.000103 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.01813 on 62 degrees of freedom
Multiple R-squared:  0.2174,    Adjusted R-squared:  0.2048
F-statistic: 17.23 on 1 and 62 DF,  p-value: 0.000103
```

Figura 8: Sortida comanda lm

A la Figura **??** es pot veure com l'R-squared ara és més gran, de 0.2174, però tot i així continua essent petit. És a dir, a la partida d'EDA el temps de sistema determina una mica més la variabilitat del temps d'usuari, pero continua sent poc. Per tant, per segon cop, no té sentit analitzar la recta que ens proporciona la comanda d'R.

5 Discussió dels resultats

5.1 Conclusió

Pel que fa a la mitjana del temps total que tarden les shells en realitzar totes les execucions, els resultats obtinguts ens mostren com hi ha una shell la mitjana de la qual hem inferit és menor a les demés: Zshell. Tot i així, com que s'obtenen diferències entre shells amb uns p-values inferiors al risc (sota l'hipòtesi que la diferència és 0), llavors les podem considerar igual de ràpides. No hi ha res que s'oposi a dir que són igual de ràpides, almenys amb les dades que hem recollit. De fet, en el nostre cas, els temps entre programes són més diferents que entre shells. En conclusió, les dades mostren que no hi ha diferències entre les shells, més enllà que una d'elles tingui la mitjana mínima, però no significativament inferior a les altres.

Per altra banda, en quant a la relació entre el temps d'usuari i el temps de sistema, els resultats ens diuen que el temps de sistema no determina la variabilitat del temps d'usuari, en cap dels dos programes que hem usat. Aquesta conclusió resulta tenir

molt de sentit, ja que en l'execució del programa de gestió de processos, que al final vam decidir no incloure pel que s'explicarà a l'apartat posterior, es veia com els temps de sistema eren superiors als temps d'usuari, cosa totalment inversa al que s'obté a l'ordenació de vectors i a la partida d'EDA.

5.2 Limitacions i problemes trobats

La major limitació del nostre estudi ve donada pel reduït nombre d'execucions realitzades i el fet de no tenir en compte una amplia variabilitat de programes, per poder assegurar encara més la "justicia"a l'hora de comparar les shells. Una possible millora a realitzar seria fer un únic programa que faci moltes coses diferents i realitzar moltes execucions d'aquest programa a cadascuna de les shells.

Pel que fa als problemes trobats, com s'havia mencionat anteriorment, inicialment voliem realitzar, per cada shell, quatre execucions de quatre programes diferents, els dos que s'usen en aquest treball i uns altres dos:

- tree /: *tree* es tracta d'una comanda UNIX que s'usa per llistar de forma recursiva el contingut d'un directori. Llavors, si fem tree /, per ser / el directori arrel del sistema operatiu, es llistaria tot el contingut del sistema de fichers.

- Gestió de processos: disposàvem d'un altre programa en bash script on el procés pare crea 1000 processos fills que passen a calcular i mostrar per sortida estàndard el nombre de fibonacci amb índex 40. Després d'això el pare espera a la mort dels fills mitjançant la crida a sistema wait per eliminar la seva informació de les estructures de dades del kernel.

L'objectiu que teniem a l'usar aquests altres dos programes era que hi hagués més variabilitat en quant a programes, per no afavorir a una shell que sigui millor en una gestió concreta. Però ens vam trobar amb un greu problema a l'hora de voler fer l'anàlisi del temps total d'execució per part de cada shell: la premissa de normalitat no es complia. Això era degut a que, com que la diferència entre els temps dels dos programes que usem en el treball i aquest altres dos era molt diferent, i s'observaben tres subconjunts molt diferents a les gràfiques. Una opció podria haver sigut analitzar els temps associat a cada subconjunt de programes, però vam preferir prescindir d'aquests dos i realitzar més execucions de la partida d'EDA i l'ordenació de vectors per donar més fiabilitat a l'estudi. De totes, formes, tant la recollida de dades qua vam fer inicialment com el programa de gestió de processos estaran disponibles a **??**.

5.3 Treballs futurs

Com a proposta de treball futur, es podria fer el que s'ha comentat abans, comparar les shells fent un únic programa amb molta variabilitat de còmput, i realitzar moltes execucions d'aquest a cada shell. També es podrien evaluar shells que no s'han contemplat en aquest treball i fins i tot analitzar el seu rendiment en funció de l'arquitectura del computador, o qualsevol altre aspecte que pugui afectar al temps que tarda cada shell.

ANNEX

A continuació es pot observar el codi en bashscript del programa que ordena un vector de 10000 elements. El vector, com s'ha dit anteriorment, és sempre el mateix i de fet es troba present al codi font, però no l'inclourem en aquest document per motius obvis.

```
#!/bin/bash

function partition {
  let i=$2-1
  let pivot=${x[$3]}

 for(( j="$2"; j<"$3"; j++ ))
  do
    if [ "${x[$j]}" -lt "$pivot" ]
    then
      let i=i+1
      let temp=${x[$i]}
      x[$i]=${x[$j]}
      x[$j]=$temp
    fi
  done

  let temp=${x[$((i+1))]}
  x[$((i+1))]=${x[$3]}
  x[$3]=$temp

  k=$((i+1))
}

function quicksort {
  if [ "$2" -lt "$3" ]
  then
    partition x $2 $3
    let pi=k
    quicksort x $2 $((pi-1))
    quicksort x $((pi+1)) $3
  fi
}
echo -n "Sorted array: "
for i in "${x[@]}"
do
  echo -n "$i "
done
echo
```

Listing 1: "Programa en bashscript: Ordenació d'un vector de 10000 elements"

Aquí podem veure el codi en bashscript del programa que crea 1000 processos que calculen i mostren per sortida estàndard el nombre de fibonacci amb índex 40:

```
#!/bin/bash

fibonacci() {
  local n=$1
  local a=0
  local b=1
  for ((i=0; i<n; i++)); do
    local c=$((a + b))
    a=$b
    b=$c
  done
  echo $a
}

for ((i=1;i<=1000;i++)); do
  (
    echo "Proceso $i: $(fibonacci 5000)"
  ) &
done

wait
```

Listing 2: "Programa en bashscript: Gestió de processos"

Finalment, a través d'aquest enllaç es pot accedir a un repositori de GitHub on es pot veure tot el codi del jugador PEplayer per les partides d'EDA, a més del ficher .xlsx amb totes les dades que hem usat, entre altres coses relacionades amb el projecte: `http://www.latex-project.org/`