Ejercícios SVM y Random Forest

Autores: Daniel Ferreira Zanchetta y Laís Silva Almeida Zanchetta

Nota introductoria: Este informe está dividido en dos partes, donde en la primera parte comentamos sobre los resultados de los distintos modelos SVM y, en la segunda parte, de los distintos modelos de Random Forest. En ambas técnicas elegimos utilizar el dataset de *Musk2*.data para clasificación, donde para pre-processing también dividimos en dos partes:

> **A)** Eliminación de las dos primeras variables; eliminación de duplicados; normalización de variables numéricas y de la variable Class; división de 70% de muestreo para training y 30% para testes;
>
> **B)** Mismas características del uno pero aplicando el PCA. Al aplicar el PCA hemos llegado al número de 25 componentes principales, con los que hemos utilizado para los modelos.

En las siguientes secciones abordamos un poco más nuestros modelos y resultados. En la tabla llamamos de "***P.P. A***" el pre-processing A arriba, y "***P.P. B***" el pre-processing B.

1) SVM

Con SVM hemos construido diversos modelos para 3 de los principales tipos de kernel: Linear, Radial y Polynomial.

En un primer momento, luego al empezar con el ejercicio, estuvimos construyendo modelos por separado para probar distintos valores de hyperparameters. Sin embargo, al estudiar más sobre las funciones del paquete *e1071*, hemos descubierto la función tune(), que ofrece la ejecución de modelos (SVM incluido), y aplica por defecto *10-fold Cross-Validation*. Aunque el número de k-fold puede ser "tuneado", creemos que para nuestros objetivos 10-fold ya es suficiente.

Con el tune() hemos podido definir ranges de hyperparameters para probar los distintos valores, y el tune al final devuelve cual es el mejor de los modelos, basado en aquella combinación de hyperparameters que tuvo el error y dispersión más pequeño.

La única desventaja que hemos visto al usar tune() ha sido el tiempo de ejecución. Algunas de nuestras ejecuciones han tardado horas. Para una de ellas, con el kernel radial, hemos sido obligados a parar la ejecución para continuar con el ejercicio pues ya llevaba más de 4 horas y 30 minutos ejecutándose.

Entendemos que el propio *SVM* (cuando usado el propio svm()) ya es un método muy complejo para las maquinas, independiente del tipo de kernel, lo que puede tardar des de muchos minutos o hasta muchas horas en ejecutarse. Sin embargo, la función tune() añade más tiempo al proceso debido a toda la complejidad que tiene por detrás. Ejecuciones con el kernel radial, en nuestra experiencia con el ejercicio, han sido aquellas que han tardado más tiempo.

Observaciones generales

- Consideramos que el mejor modelo será aquel que minimice el error de teste;
- El pre-processing aplicado es esencial para el rendimiento de un modelo SVM. Sin embargo, con nuestra experiencia en este ejercicio, no está directamente relacionada a una mejor calidad de los resultados. Quizás no sea siempre así, sin embargo en nuestro

caso hemos observado una mejora de los errores de teste con el ***pre-processing A*** que con el B;

- No todos los hyperparameters son aplicados a todas los tipos de kernel. Abajo son los hyperparameters que hemos usado para cada uno:

    - Linear: cost (C)
    - Radial: cost (C) y Gamma (γ)
        - Formula radial kernel: $$K(x_i, x_{i'}) = \exp(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2).$$ (Ref.: *An Introduction to Statistical Learning*)
    - Polynomial: cost (C), degree (d) y coef0
        - Formula polynomial kernel: $$K(x_i, x_{i'}) = (1 + \sum_{j=1}^{p} x_{ij} x_{i'j})^d$$ Ref.: *An Introduction to Statistical Learning*)

Cada parámetro es necesario en sus respectivos kernels y sirven para ajustar la variancia y bias. Por ejemplo, degree controla la flexibilidad de la decisión boundary.

Tabla comparativa de resultados

Para la tabla abajo, los IDs 1 y 2 han sido los únicos casos en que hemos hecho sin CV. Para los demás, todos han sido con CV.

ID	Tipo de Kernel	Hyperparameters	Tiempo de Ejecución	Test Error (P.P. A)	Test Error (P.P. B)
1	**Linear**	**Cost** = 10	Con P.P. A (solo) – 8 minutos	3.93%	-
2		**Cost** = 0.1	Con P.P. A (solo) – 8 minutos	5.30%	-
3		**Cost** = c(0.001, 0.01, 1, 1.5, 5, 10, 100) **Cost** = 1.5 *ha sido el óptimo en "P.P. A" y "P.P. B"*	Con P.P. A) 1h10min Con P.P. B) 24min	4.69%	8.28%
4	**Radial**	**Cost** = 1 **Gamma** = 2	Con P.P. A) 4min 45seg Con P.P. B) 7min 30seg	8.93%	8.88%
5		**Cost** = 0.1 **Gamma** = 0.25	Con P.P. A) 4min Con P.P. B) 3min 40seg	14.24%	13.18%
6		**Cost** = 0.1 **Gamma** = 0.125	Con P.P. A) 20seg Con P.P. B) 20seg	14.24%	7.17%
7	**Polynomial**	**Cost** = c(0.001, 0.01, 1, 1.5, 5, 10,	Con P.P. A) 3min 50seg	0.15%	1.76%

ID	Tipo de Kernel	Hyperparameters	Tiempo de Ejecución	Test Error (P.P. A)	Test Error (P.P. B)
		100)), **degree = 3**, coef0 = 1 **Cost óptimo** = 10 para P.P. A y P.P. B	Con P.P. B) 3min 45seg		
8		**Cost** = c(0.001, 0.01, 1, 1.5, 5, 10, 100)), **degree = 4**, coef0 = 1 **Cost optimo** = 5 para P.P. A **Cost optimo** = 1.5 para P.P. B	Con P.P. A) 4min Con P.P. B) 3min 15seg	0.10%	1.61%
9		**Cost** = c(0.001, 0.01, 1, 1.5, 5, 10, 100)), **degree = 5**, coef0 = 1 **Cost optimo = 1** para P.P. A y P.P. B	Con P.P. A) 3min 45seg Con P.P. B) 3min 40seg	0.35%	1.61%

Nos hacía mucha ilusión obtener los resultados para el código a continuación, porque probaba muchas combinaciones de cost y gamma para el kernel "radial". No obstante, hemos tenido que abortar la ejecución para continuar avanzando con el ejercicio, que ya llevaba 5 horas con los datos de "P.P. A" y 4 horas y 30 minutos con los datos de "P.P B".

tune.out <- tune(svm,class~., data = train_musk, type = "C-classification", kernel = "radial",ranges=list(cost=c(0.001, 0.01, 1, 1.5, 5, 10, 100),gamma=c(0.125, 0.25,0.5,1,2,4,8)))

1.1) Modelos

Abajo pasamos el código de los modelos que nos ha salido mejor, referentes a los IDs 7 y 8 para el P.P. A:

```
#It appears that the degree parameter controls the flexibility of the dec
ision boundary. Higher degree kernels yield a more flexible decision boun
dary.
#degree: parameter needed for kernel of type polynomial (default: 3) - pr
obar con 1, 2, 3 y 5
#coef0: parameter needed for kernels of type polynomial and sigmoid (defa
ult: 0)
start.time <- Sys.time()
tune.out.poly3 <- tune(svm,class~., data = train_musk, type = "C-
classification", kernel = "polynomial",ranges=list(cost=c(0.001, 0.01, 1,
 1.5, 5, 10, 100)),degree = 3, coef0=1)
(end.time <- Sys.time() - start.time)

(bestmodel.poly3 <- tune.out.poly3$best.model)
```

```
pred <- predict(bestmodel.poly3, newdata = test_musk,type="class", decisi
on.values = TRUE)
cm <- table(Truth=test_musk$class, Pred=pred)
(error <- (1-sum(diag(cm))/sum(cm))*100) #Error
#
start.time <- Sys.time()
tune.out.poly4 <- tune(svm,class~., data = train_musk, type = "C-
classification", kernel = "polynomial",ranges=list(cost=c(0.001, 0.01, 1,
 1.5, 5, 10, 100)),degree = 4, coef0=1)
(end.time <- Sys.time() - start.time)

(bestmodel.poly4 <- tune.out.poly4$best.model)

pred <- predict(bestmodel.poly4, newdata = test_musk,type="class", decisi
on.values = TRUE)
cm <- table(Truth=test_musk$class, Pred=pred)
(error <- (1-sum(diag(cm))/sum(cm))*100) #Error
```

1.2) Conclusiones

Teniendo en cuenta la experiencia que hemos tenido en la construcción y ejecución de estos modelos de SVM, llegamos a conclusión que utilizando todas las dimensiones, y no solo los componentes principales, nos da mejores resultados. En particular creemos que el ID 8 ha sido el mejor resultado obtenido por algunas razones:

- Por ser un polynomial kernel de degree 4, hace con que el algoritmo tenga una "decisión boundary" más flexible;
- Siendo un kernel no-lineal, esto hace con que el ajuste pase en un espacio dimensional más amplio involucrando polinomios de grado 4;
- El coste optimo para el modelo siendo 5 también ayuda a que tenga más varianza y bias más pequeño.

Si fuera posible sacar un plot de este modelo, imaginamos que podríamos ver las márgenes, decisión boundaries y observaciones muy clasificadas.

2) Random Forest

Para realizar la parte de Random Forest, también utilizamos el principio de construir y probar diferentes modelos, divididos en 3 categorías:

Tipo de Modelo 1) Usamos distintas combinaciones de número de árboles;

Tipo de Modelo 2) Realizamos upsample de la clase menos representada (que es clase Musk = "1"), con el objetivo era de encontrar el número de árboles que un error real más pequeños. Elegimos focalizarnos en la clase Musk pues entendemos ser más importante a nivel de negocios saber si estamos clasificando bien una observación como Musk;

Tipo de Modelo 3) Optimización del Random Forest guiado por el OOB, para encontrar el número de árboles que nos trae en OOB error rate más pequeño.

Para el Random Forest no hemos hecho la medición del tiempo, porque creíamos que no hacía falta. Todos los modelos han sido muy rápidos en ejecutarse.

ID	Tipo de Modelo	Parámetro	P.P. A		P.P. B	
			OOB Error Rate	Error Real	OOB Error Rate	Error Real
1	1	**Ntree** = 100	2.43%	2.52%	4.85%	4.34%
2		**Ntree** = 200	2.58%	2.63%	4.83%	4.19%
3	2	**Ntree** = 100 **Sampsize=**c("1"=735, "0"= 650)	4.96%	4.69%	4.79%	4.79%
4		**Ntree** = 250 **Sampsize=**c("1"=735, "0"= 650)	4.35%	4.34%	4.59%	4.44%
5	3	**Sampsize=**c("1"=735, "0"= 650)	**Ntree con mejor OOB** = 158 4.66%	4.69%	**Ntree con mejor OOB** = 1000 4.44%	4.44%

2.1) <u>Conclusiones</u>

Aunque sea visible que los dos primeros modelos, en P.P. A, tengan los mejores resultados, creemos que los resultados obtenidos con los tipos de modelos 2 y 3 son los que debíamos centrarnos. Dicho esto, el modelo del ID 4 para el P.P. A ha sido lo que ha tenido error real más bajo, de 4.34%. Entre todos los modelos probados, este ha sido aquel que mejor ha clasificado en la clase Musk.

Abajo pasamos los modelos del ID 1, ID 2 e ID 4:

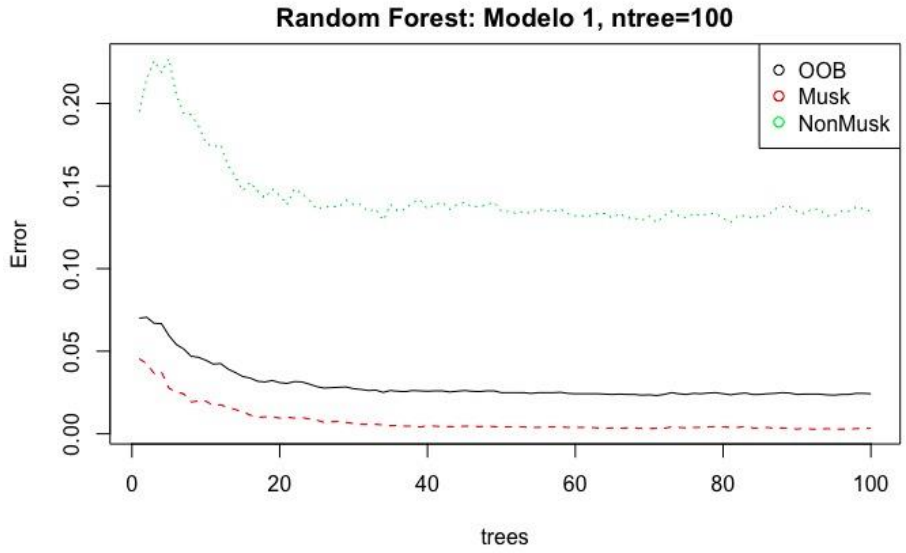


Figura 1: Plot Modelo ID 1

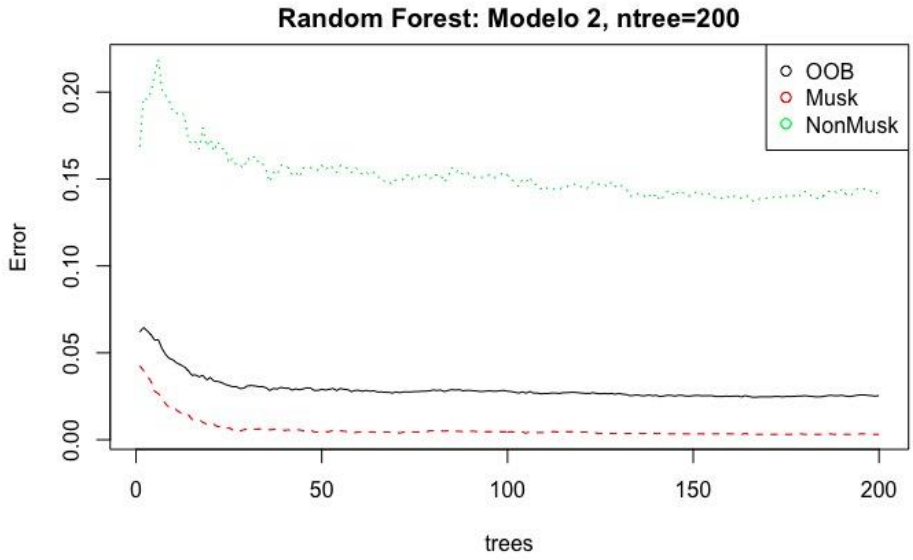


Figura 2: Plot Modelo ID 2

Un punto que hemos dado cuenta es que no haría falta un número tan grande de árboles, porque ya por la vigésima árbol el error ya se estabiliza bastante.

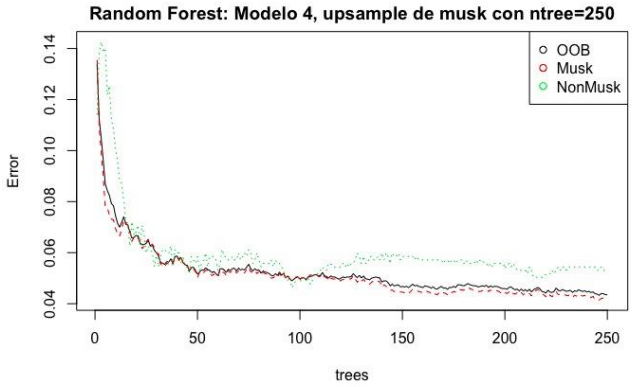
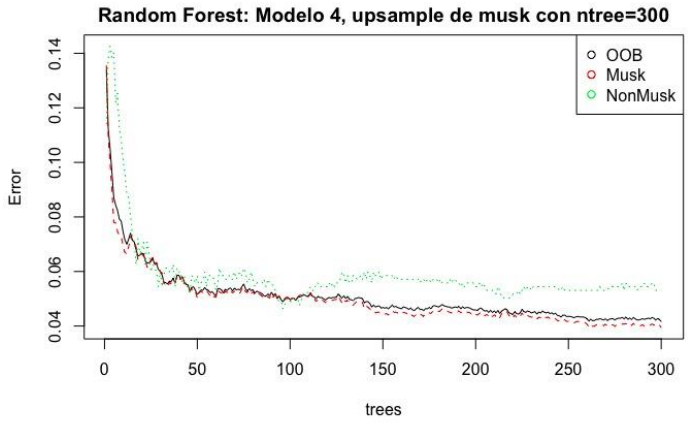


Figura 3: Plot ID 4

Mientras mirábamos el resultado del plot arriba nos ha parecido ver que el número de error aun podía mejorar un poco más. Luego, entonces, probamos aumentar el número de árboles del ID 4, para 300. Hemos descubierto un Error rate mejor aún:

- 4.16% OOB
- 4.19% Error Real



Más allá de 300 árboles, el error empeora.

Como una última observación, basado en nuestros estudios, creemos que el P.P B (con PCA) no ha tenido buenos resultados pues los modelos Random Forest están utilizando para predicción muchas de las variables que con PCA acabamos quitando. Abajo, por ejemplo, ponemos un plot de las variables más importantes del último modelo analizado: