Lec5: Clasificacion

Isidoro Garcia Urquieta

2023

Agenda

1. Teoría de clasificación
   - Clasificación Binaria, Múltiple, Multi variada
   - Matriz de confusión
   - Métricas de éxito (Accurracy, Sensibility, Specificity, AUC ROC)
   - Rebalanceo de clases
2. Algoritmos de Clasificación
   - Logit (Multinomial Logit) (LASSO!)
   - Distributed Multinomial Regression
   - K-nearest neighboors
   - Otros (Trees, Forests, XGBoosting, Support Vector Machines)

Clasificación

Los modelos de clasificación son parte del Aprendizaje Estadístico Supervisado. La diferencia es que en los problemas de regresión y_i es una variable continua y en los modelos de clasificación y_i es **categórica**.
$y_i \in \{1, 2, ..., m\}$ (M categorias).

Al igual que en los problemas de regresión, buscamos *poder de predicción fuera de la muestra (OOS)*. Es decir:

- Dividimos la muestra en entrenamiento y validación;
- Entrenamos el modelo (usamos CV o no..) y
- Validamos las predicciones en la muestra de validación.

El problema es:

$$\text{Dado } X_{ij}^{Validación}, \text{ cual es la clase predecida de } \hat{y}?$$

Donde el error de predicción $(\hat{y}_i \neq y_i)$

Tipos de clasificación

Hay tres tipos de clasificación:

1. **Binaria**: Y sólo toma dos clases (i.e. $\{0, 1\}$. Los ejemplos de esto son muchos: La persona acusada es culpable o inocente? Esta transacción es un fraude?)
2. **Múltiple**: Y toma k clases que son mutuamente excluyentes.
3. **Multi-variada**: Y toma b clases que no son mutuamente excluyentes

Es importante resaltar que en todos estos casos las categorias son **unordered**. Es decir, no hay algo como (no en esta clase) de 'Ordered Random Forest'.

Clasificación Binaria

En la clásificación binaria podemos utilizar los modelos **Logit** y **LASSO Logit** y *K-nearest neighbors (No paramétrico)*.

En general *LASSO Logit* $>$ *LASSO*

Hoy veremos *K-nearest neighbors* para poder hacer la comparativa entre esos dos algoritmos.

Nota: Existen otros algoritmos para hacer clasificación (y regresión) como los Random Forests, XGBoostings y Support Vector Machines. Los veremos de manera directa en otras clases.

Desempeño OOS de un modelo de clasificación

La forma más directa de evaluar el poder predictivo de un modelo de clasificación es la **Matriz de Confusión**:

		Real	
		$y_i = 0$	$y_i = 1$
Predecido	$\widehat{y_i} = 0$	**(A):** V Negativo	**(C):** Falso Negativo
	$\widehat{y_i} = 1$	**(B):** Falso Positivo	**(D):** V Positivo

A partir de ahí, hay varias métricas de éxito:

- **Sensibilidad/Recall** $(\frac{D}{C+D})$: De los $1's$ reales, cuál es la probabilidad de que los encontremos
- **Especificidad** $(\frac{A}{A+B})$: De los $0's$ reales, que tan probable es que los encuentre

Desempeño OOS de un modelo de clasificación

- **Positivos Acertados/Precisión** ($\frac{D}{B+D}$): Que porcentaje de mis $1's$ predecidos son aciertos
- **Negativos Acertados** ($\frac{A}{A+C}$): Qué porcentaje de mis $0's$ predecidos son aciertos
- **Accuracy** ($\frac{A+D}{A+B+C+D}$):

La sensibilidad nos dice que tan bueno es nuestro algoritmo detectando $1's$.

La especificidad nos dice que tan bueno es nuestro algoritmo detectando $0's$.

Estos parecen muchas métricas de éxito. Más aún, Los mayoría de los algoritmo predicen $P(y_i = 1|X)$ y no un valor $\{0, 1\}$. Necesitamos una mejor métrica.

Receiving Operating Characteristic (ROC)

La Curva ROC define la habilidad predictiva de un modelo de clasificación binaria ante *un punto de corte* de clasificación variante.

- Se dibuja en el espacio $\{1 - \textit{especificidad}, \textit{sensibilidad}\}$.
- Se interpreta como la probabilidad de que nuestro algoritmo obtenga la clasificación correcta **balanceando** el error tipo I y error tipo II. (Ojo! diferencias vs Accuracy?)

Pasos:

1. Estimas la probabilidad estimada $\hat{p} \in [0, 1]$
2. Eliges un punto de corte p^* tal que $\tilde{p} = 1\{\hat{p} \geq p^*\}$
3. Calculas la matriz de confusión, la sensibilidad y la especificidad
4. Graficas en el espacio de la curva ROC.
5. Repite el ejercicio para $p^* \in [0, 1]$

Ejemplo: Calculando una ROC

Imaginen que estimamos un Logit LASSO.

```
library(tidyverse)
library(gamlr)

diamonds<-diamonds

table(diamonds$cut)
```

```
##
##      Fair      Good Very Good   Premium     Ideal
##      1610      4906     12082     13791     21551
```

```
# Creamos categorias de premium
diamonds<-
  diamonds %>%
  mutate(premium = if_else(cut == 'Premium', 1, 0),
         id = seq(1,nrow(diamonds)))
```

Ejemplo

Separamos la base en training y validation

```
diamonds_training<-
  diamonds %>%
  slice_sample(prop = 0.7)


diamonds_test<-
  diamonds %>%
  filter(!(id %in% diamonds_training$id))
```

Ejemplo

```
X<-diamonds_training %>% select(carat, depth, price, x,y,z)
y<-diamonds_training$premium

modelo<-gamlr(x = X, y = y, family = 'binomial')

# Pronostico OOS
X_test<-diamonds_test %>% select(carat, depth, price, x,y,z)
diamonds_test$premium_pred <- predict(modelo, newdata = X_test,
                                      type = 'response')[,1]
head(diamonds_test %>% select(id, premium, premium_pred), n = 4)
```

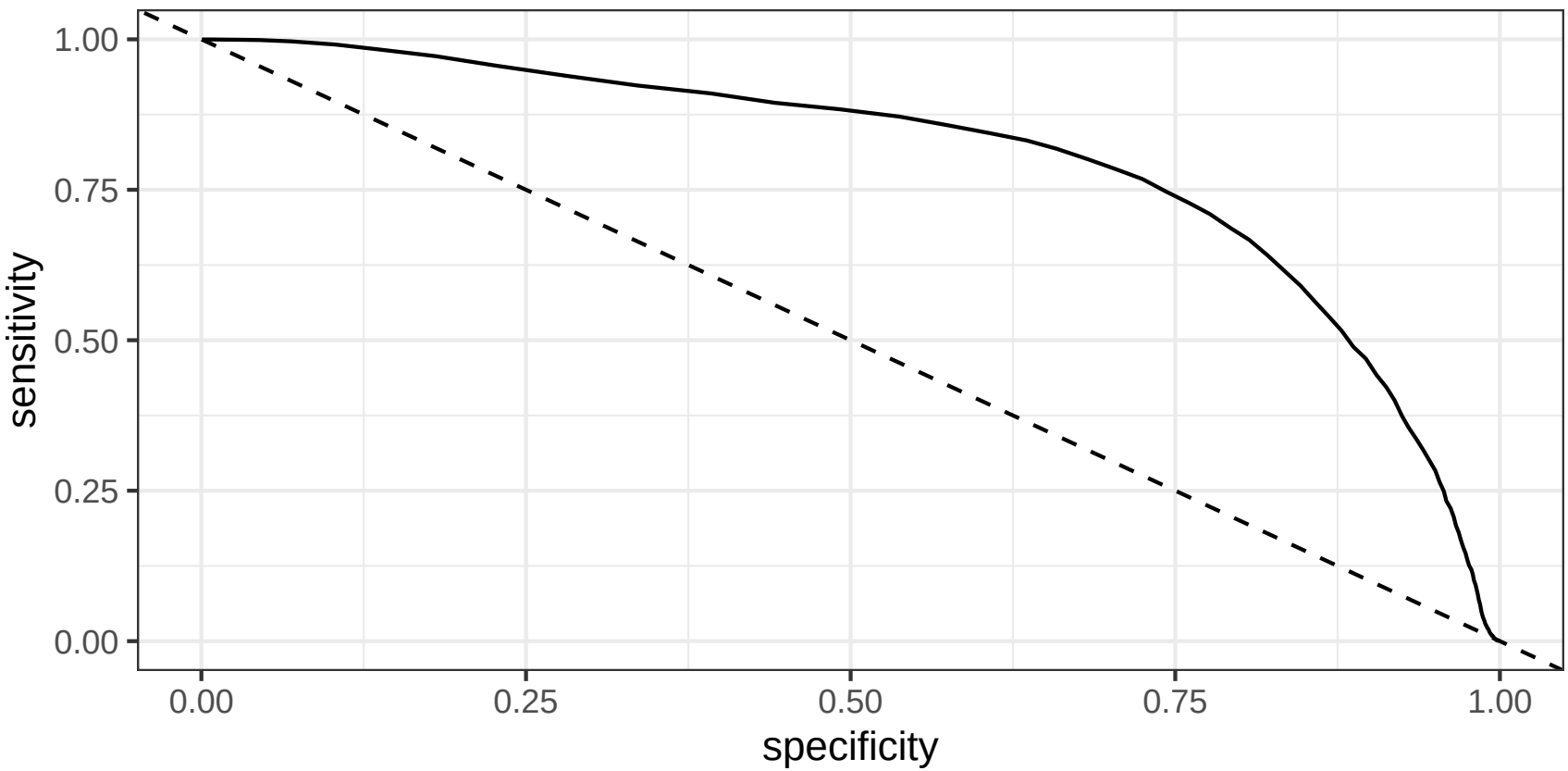
```
## # A tibble: 4 x 3
##       id premium premium_pred
##    <int>   <dbl>        <dbl>
## 1     2        1        0.539
## 2     5        0        0.117
## 3     8        0        0.131
## 4    17        0        0.141
```

Calculo ROC

```
curva_roc<-map_dfr(seq(0,1,0.01), function(x) {
      a<-diamonds_test
      a$pred_cat<-NA
      a$pred_cat[a$premium_pred>= x]<-1
      a$pred_cat[a$premium_pred<x]<-0
      a<-
        a %>%
        count(premium, pred_cat)

 num_sensitivity <- a$n[a$premium==1 & a$pred_cat==1]
 den_sensitivity<- num_sensitivity + a$n[a$premium==1 & a$pred_c
 sensitivity<-num_sensitivity/den_sensitivity
 num_specificity <-a$n[a$premium==0 & a$pred_cat==0]
 den_specificity<- num_specificity + a$n[a$premium==0 & a$pred_c
 specificity<-num_specificity/den_specificity

 roc_table<-tibble("p" = x,
                   "sensitivity" = sensitivity,
                   "specificity" = specificity)
```

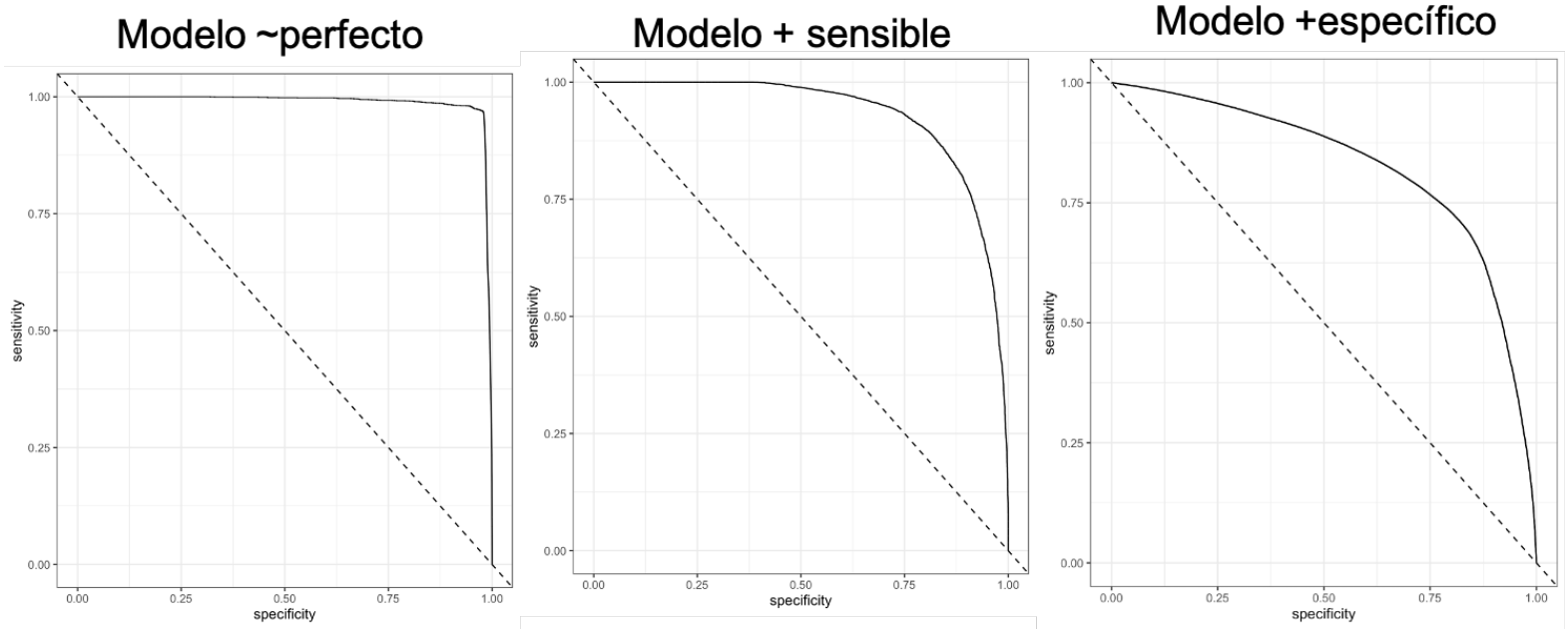
Veamos el resultado

```
ggplot(curva_roc, aes(specificity, sensitivity))+
  geom_line()+
  geom_abline(intercept = 1, slope = -1,
              linetype = 'dashed')+theme_bw()
```

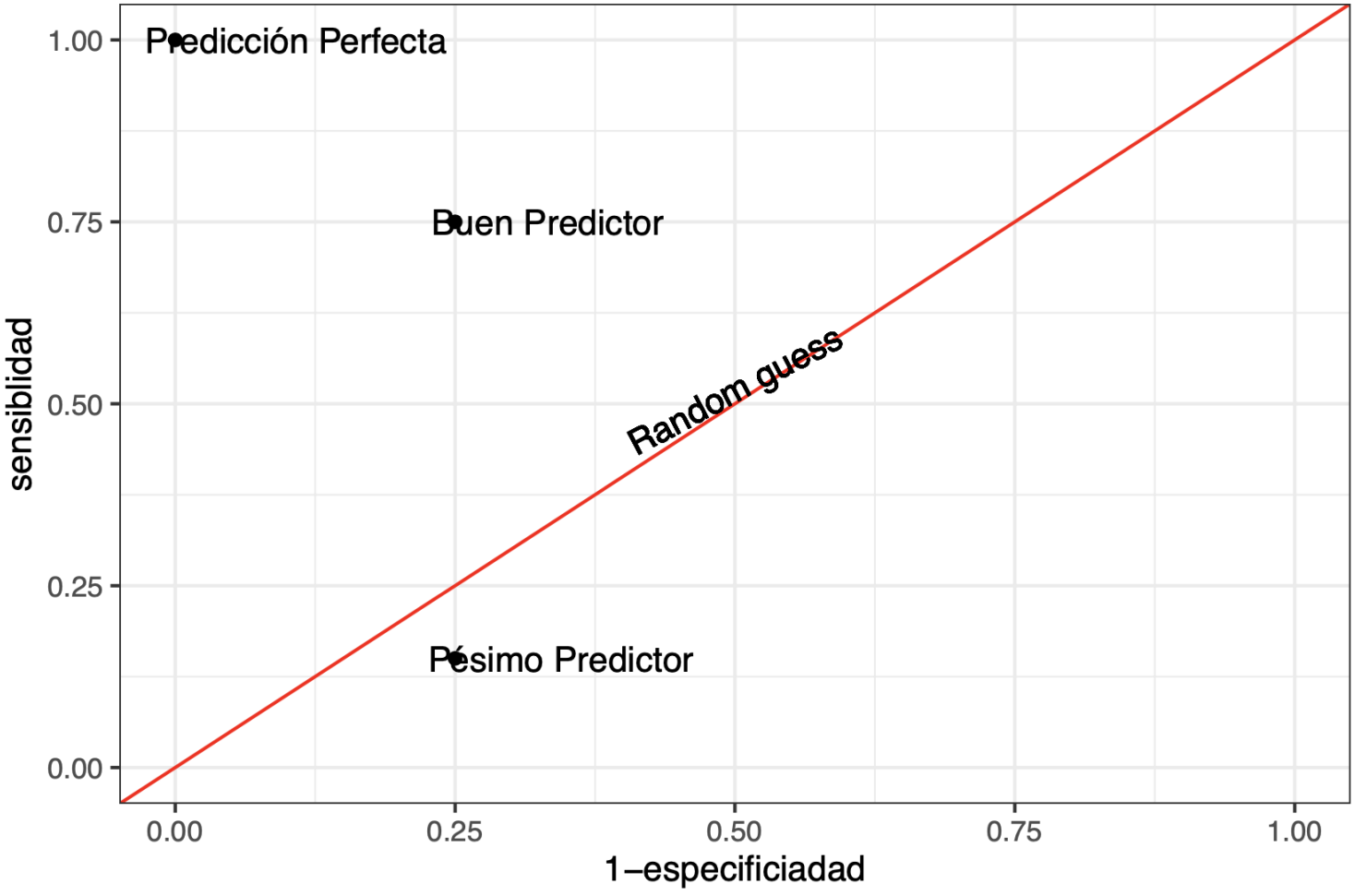


Como se ve una buena curva ROC

Al final de este procedimiento tendras una curva que se verá (ojalá) así:



Curva ROC


Predicción Perfecta
Buen Predictor
Random guess
Pésimo Predictor
1.00
0.75
0.50
0.25
0.00
sensiblidad
0.00
0.25
0.50
0.75
1.00
1−especificiadad

AUC ROC

Una vez con la curva dibujada, que sigue? Calcular el Area under the Curve (AUC) ROC (Noten que el área de todo el cuadrado es 1):

- Si $AUC = 0.5$, nuestro modelo no supera a la aleatoriedad
- Si $AUC = 1$, nuestro modelo es perfecto

Típicamente se espera un modelo con AUC (obvio OOS) $AUC \geq 0.75$ para ser bueno. Un modelo excelente es $AUC \geq 0.85$ y un gran modelo es $AUC \geq 0.90$.

Finalmente, debemos elegir un punto de corte p^* para hacer la clasificación cuando pongamos el modelo a producción.

Balanceo de errores

Noten como la Curva ROC hace explícito el trade off entre falsos positivos y falsos negativos.

Si elegimos un punto de corte muy bajo, vamos a clasificar muchas observaciones como $1's$. Con esto es muy probable que todos los $1's$ sean clasificados correctamente (modelo muy sensible).

Sin embargo, va a haber muchas observaciones que sean $0's$ que clasifiquemos como $1's$. Nuestro modelo sería entonces muy poco específico.

La Curva ROC nos ayuda a balancear ambos errores según creamos conveniente. Esto es, dependiendo del problema, que tipo de error es más grave?

Finalmente, noten como AUC ROC es mucho mejor indicador que el accuracy. Si tenemos una base donde alguna de las clases es muy común (i.e. 95% $0's$ y 5% $1's$), el accuracy puede dar una señal falsa de un buen modelo.

Calculemos la Matriz de Confusión

```
library(pracma)
trapz(curva_roc$specificity, curva_roc$sensitivity)
```

```
## [1] 0.7989509
```

```
# Matriz de confusion
curva_roc %>% filter(specificity>=0.74, sensitivity>=0.74)
```

```
## # A tibble: 1 x 3
##       p sensitivity specificity
##   <dbl>       <dbl>       <dbl>
## 1  0.26       0.747       0.743
```

```
#Escogemos 0.26
diamonds_test<-
  diamonds_test %>%
  mutate(pred_cat = if_else(premium_pred>=0.26, 1,0))

table(diamonds_test$pred_cat, diamonds_test$premium)
```

```
##
```

Calculemos la AUC ROC en nuestro ejemplo II

```
table(diamonds_test$pred_cat, diamonds_test$premium)
```

```
##
##         0    1
##    0 8976 1040
##    1 3101 3065
```

- Sensitivity/Recall $= 3074/(3074+1048) = 0.746$
- Specificity $= 9001/(9001+2059) = 0.7463$
- Precission $= 3074/(3074+3059) = 0.50$
- Accuracy $= (3074+9001)/16182 = 0.746$

Balanceo de errores

Por ejemplo, ante (i.e. 95% $0's$ y 5% $1's$) un modelo podría decir 'Todos son 0'. Esto haría que 95% de las veces el modelo sea correcto!!!!!

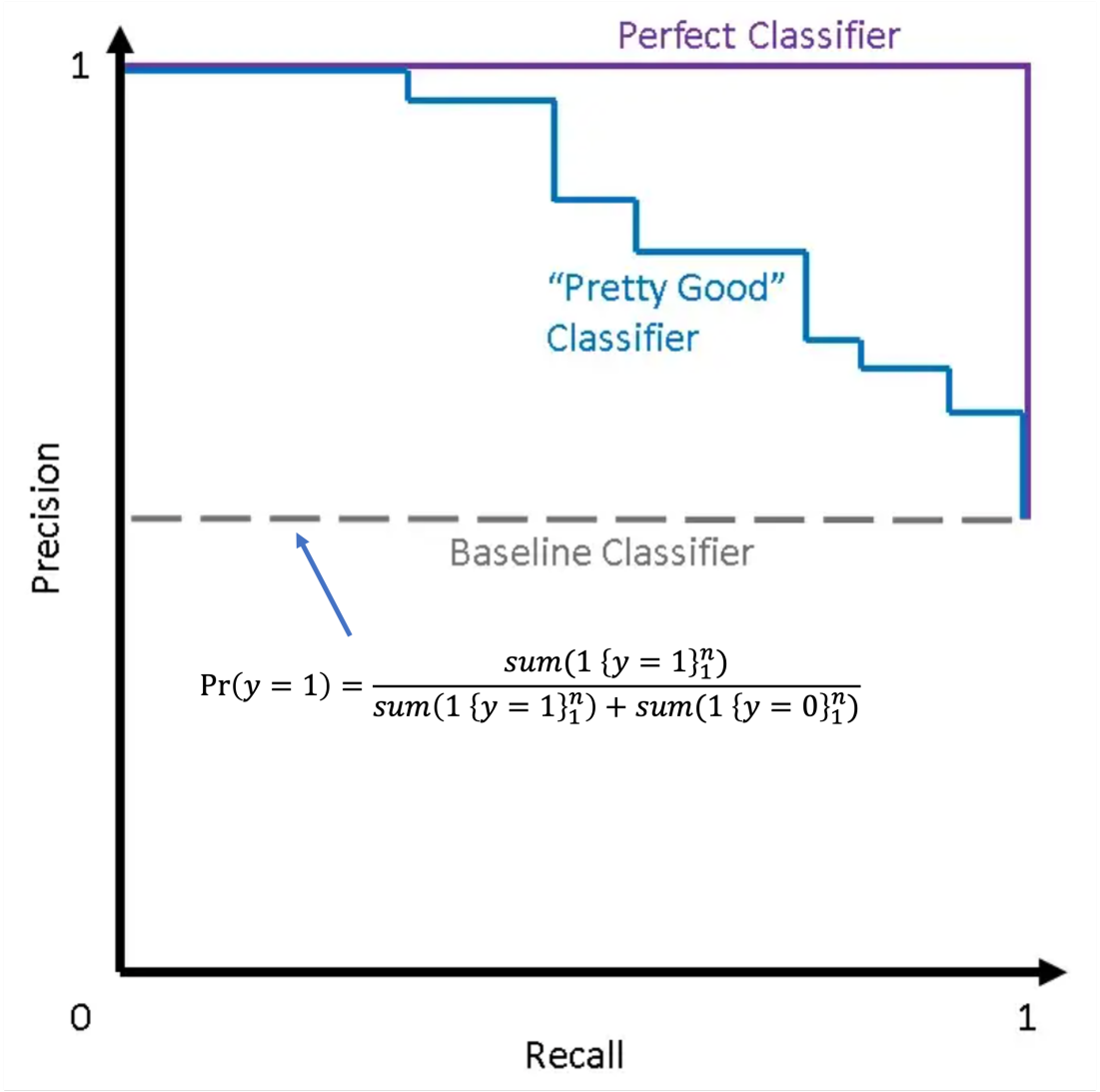
- Este modelo sería muy especifico (hasta un mono sería en este caso...), pero muy poco sensible.
- La curva ROC ayudaría a entender el desempeño del modelo de una manea mucho más balanceada.
- No obstante, aún tenemos que ver cómo entrenar modelo que no 'tomen' la decisión descrita en el primer párrafo de esta slide.

Precission-Recall curve

La curva PR es otra manera de evaluar el poder predictivo. Vive en el espacio $\{Recall, Precission\}$

- Es muy útil cuando el modelo de predicción tiene mucho imbalance de clases
- Porqué? Imagen que tenemos un problema donde en la base hay muchos 0's. Es muy probable que el modelo tenga buena specificity. Sin embargo, tenemos que ver si es sensible y preciso.
- La PR es muy buena para evaluar esta clase de casos
- Existen 2 métricas adicionales que resumen la PR curve:
- $F1 = 2\frac{precission * recall}{precission + recall}$. El dominio es 0 a 1. 0 indica un modelo aleatorio; 1 indica un modelo perfecto. Le damos igual valor a precission and recall.
- $F2 = 5\frac{precission * recall}{4precission + recall}$. Esta le da un poco de más peso a recall que a la precission.

Precission-Recall curve

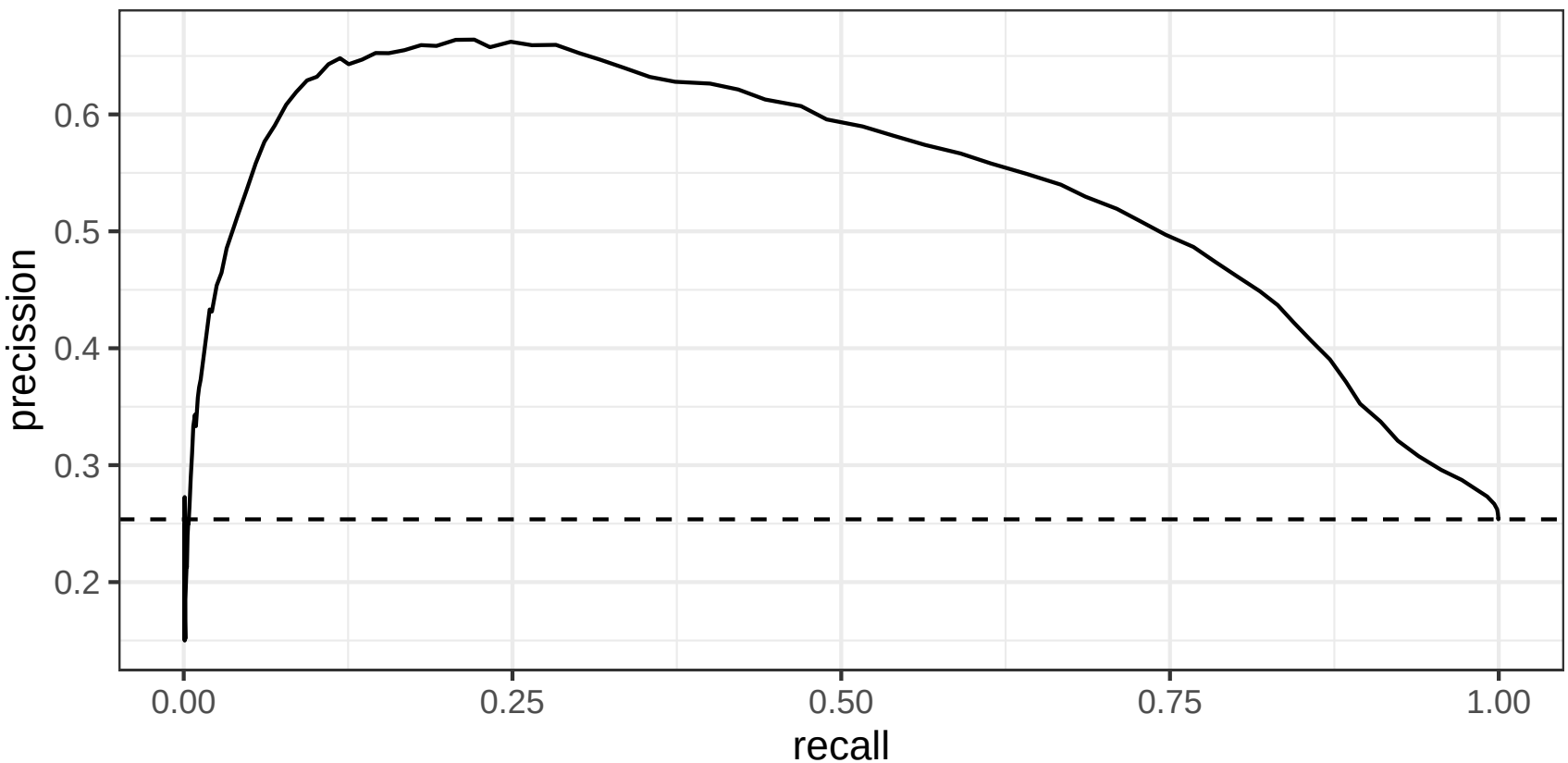


Veamos el PR curve en nuestro ejemplo

```
curva_pr<-map_dfr(seq(0,1,0.01), function(x) {
      a<-diamonds_test
      a$pred_cat<-NA
      a$pred_cat[a$premium_pred>= x]<-1
      a$pred_cat[a$premium_pred<x]<-0
      a<-
        a %>%
        count(premium, pred_cat)

 num_sensitivity <- a$n[a$premium==1 & a$pred_cat==1]
 den_sensitivity<- num_sensitivity + a$n[a$premium==1 & a$pred_c
 sensitivity<-num_sensitivity/den_sensitivity
 den_precission<- num_sensitivity + a$n[a$premium==0 & a$pred_ca
 precission<-num_sensitivity/den_precission

 roc_table<-tibble("p" = x,
                   "recall" = sensitivity,
                   "precission" = precission)
}
```

Veamos el PR curve en nuestro ejemplo

```
ggplot(curvas, aes(recall, precission))+
  geom_line()+
  geom_hline(yintercept = p,
             linetype = 'dashed')+theme_bw()
```



Escojamos de vuelta el punto de corte

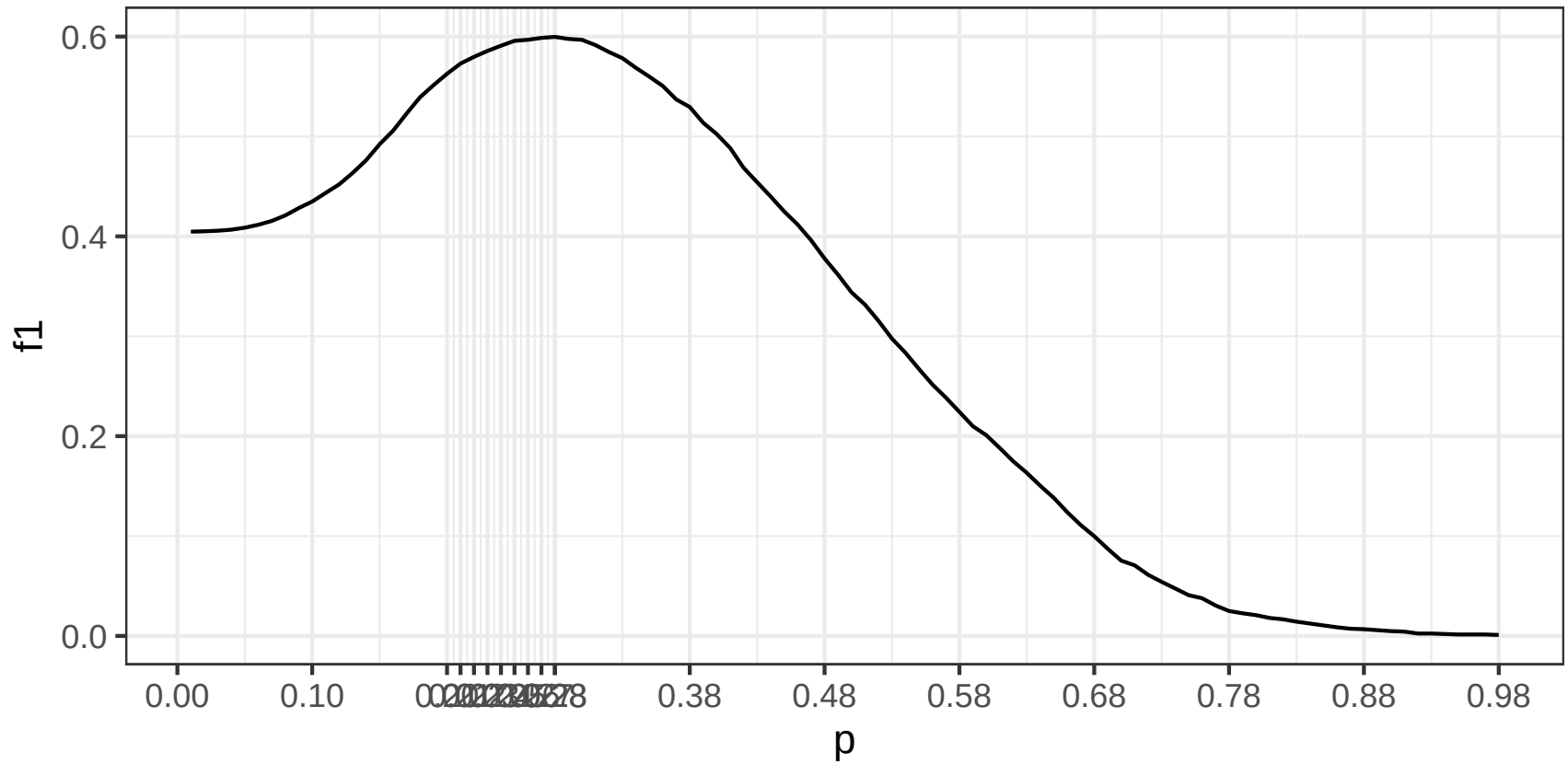
p=0.28 parece un mejor corte ahora!

```
curvas<-
  curvas %>%
  mutate(f1 = 2*(precission*recall)/(precission+recall),
         f2 = 5*(precission*recall)/(4*precission+recall))

curvas %>% filter(p>=0.25, p<=0.3)
```

```
## # A tibble: 6 x 7
##       p sensitivity specificity recall precission    f1    f2
##   <dbl>       <dbl>       <dbl>  <dbl>      <dbl> <dbl> <dbl>
## 1  0.25       0.768       0.725  0.768      0.487 0.596 0.688
## 2  0.26       0.747       0.743  0.747      0.497 0.597 0.679
## 3  0.27       0.727       0.761  0.727      0.509 0.599 0.670
## 4  0.28       0.710       0.777  0.710      0.519 0.600 0.661
## 5  0.29       0.686       0.793  0.686      0.530 0.598 0.648
## 6  0.3        0.667       0.807  0.667      0.540 0.597 0.637
```

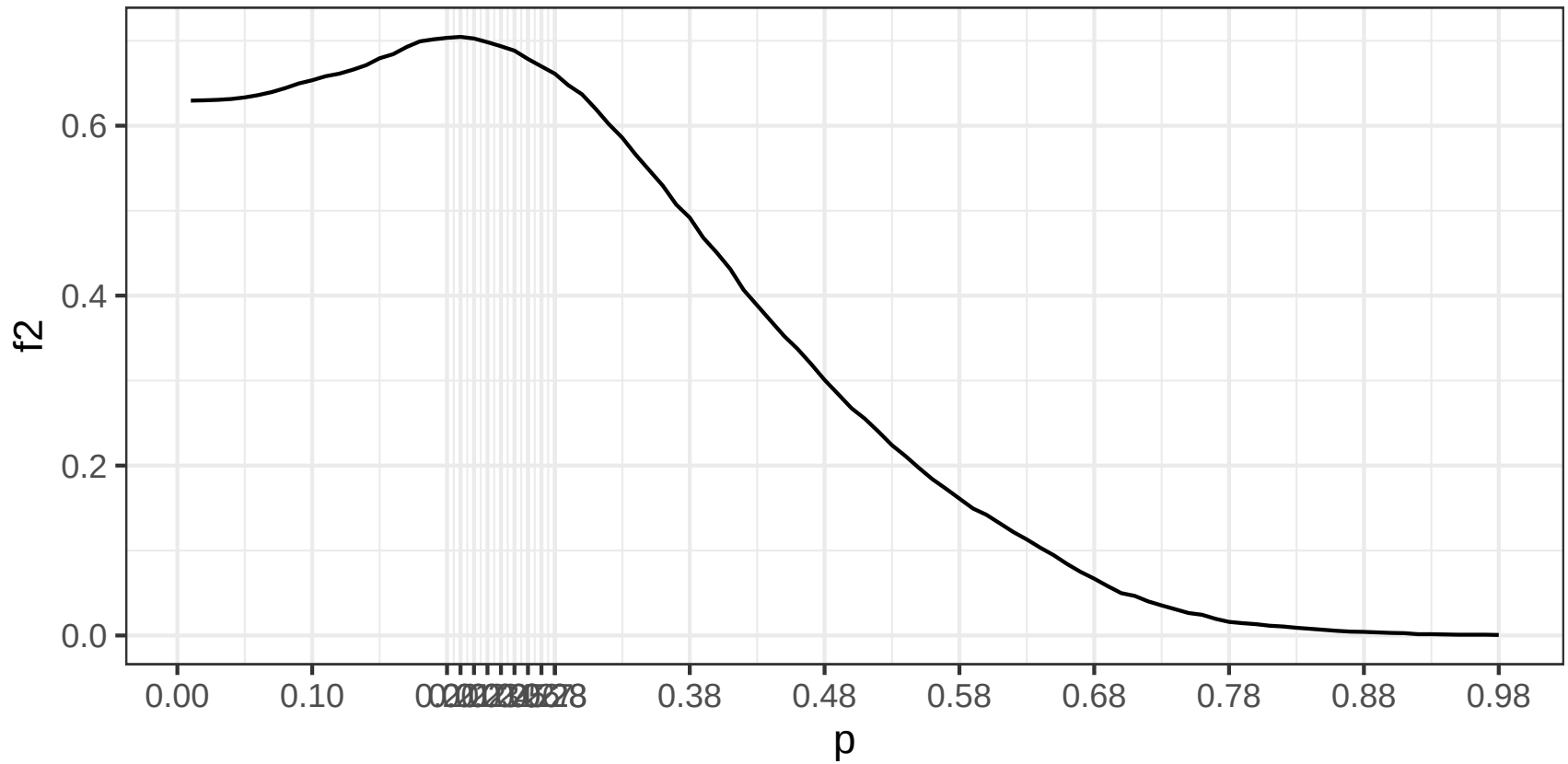
F1 graficamente

```
ggplot(curvas, aes(p, f1))+
  geom_line()+theme_bw()+
  scale_x_continuous(breaks = c(seq(0,0.2,0.1), seq(0.21, 0.28,
```



F2 graficamente

```
ggplot(curvas, aes(p, f2))+
  geom_line()+theme_bw()+
  scale_x_continuous(breaks = c(seq(0,0.2,0.1), seq(0.21, 0.28,
```



AUC PR curve

```
trapz(-curvas$sensitivity, curvas$precission)
```

```
## [1] 0.5402433
```

Desbalance en las clases

Los modelos de Machine Learning buscan patrones en los datos para minimizar el error (deviance) de predicción.

- Esto es, predecir correctamente la mayor cantidad de veces fuera de la muestra.

Con esto, **cualquier** algoritmo va a priorizar aprender los patrones de la clase más común y nada de la clase menos común. Así, es importante asegurarnos de crear las condiciones en la base de entrenamiento que **obliguen** al algoritmo a aprender patrones para ambas clases.

Existen dos formas de corregir el desbalance de clases:

1. Rebalanceo de clases mediante muestreo
2. Introducción de funciones de costos de error tipo I y tipo II: No lo veremos en esta clase. Pocos algoritmos tienen forma de incluir esto.

Rebalance de clases mediante muestreo

El rebalanceo de clase es consiste en conseguir que las dos clases **sean igual de comunes**. Con esto el algoritmo se verá obligado a aprender qué patrones en los datos generan cada clase.

Esto se puede lograr de dos maneras:

- Reducir las observaciones de la clase más común
- Incrementar las observaciones de la clase menos común

Reducción de la clase más común

Imaginemos que tenemos dos clases $\{0, 1\}$. La clase 0 sucede 95% de las veces. Cómo balancearían las clases disminuyendo la clase más común?

- Podemos tomar una muestra de ese 95% de la base para que quede sólo el 5% de la base menos común.
- Como muestrear? Lo mejor es intentar estratificar por las variables más importantes de la base para no perder información valiosa.
  - Ejemplo: Construyes un PCA; te quedas con los k componenetes que expliquen mucho de la varianza; estratificas en forma de cascada por estos compenentes.
  - Así, aseguras que la muestra resultante incluye todas las señales de la base completa
- Pros: El modelo va a aprender de manera balanceada; la base más pequeña puede hacer más estimable el algoritmo.
- Cons: Se pierden muchas observaciones. Si tenemos una base total pequeña, podemos llegar a una base inservible.

Incrementar las obs de la clase menos común

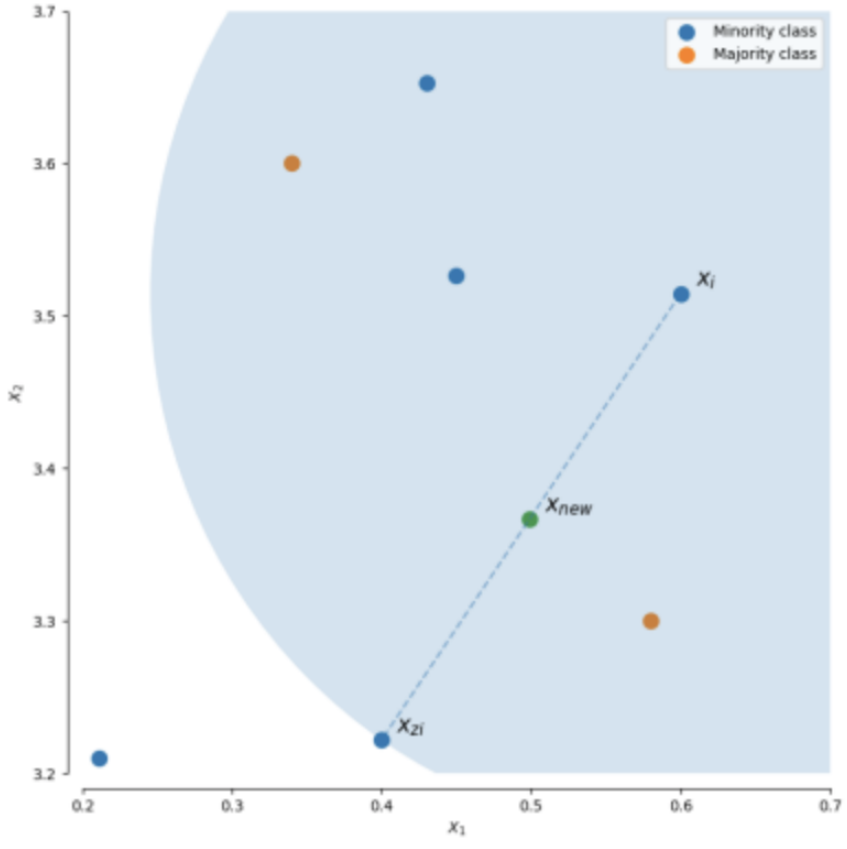
La otra opción es incrementar las observaciones de la clase menos común. Usemos el mismo ejemplo:

- Como generamos observaciones nuevas? Oversampling!
- Hacemos un oversampling equivalente al que hicimos en undersampling? No! Esto haría muchas observaciones repetidas y reduciría la varianza de la base.

El método más común es Synthetic Minority Over-sampling Technique (SMOTE).

1. Tomamos una la base de la clase menos común
2. Construímos un algoritmo K-Nearest Neighbors
3. Tomamos una muestra i. Encontramos la otra 'fila' que es muy similar (el vecino más cercano)
4. Creamos una observación sintética entre ambas con una $\lambda \in [0, 1]$

SMOTE


Minority class
Majority class
x_i
x_{new}
x_{zi}
3.7
3.6
3.5
3.4
3.3
3.2
x_2
0.2
0.3
0.4
0.5
0.6
0.7
x_1

Oversampling vs Undersampling

La decisión de cual usar (Over vs Under sampling) es básicamente el tamaño de la base.

En general, si se logra tener suficiente información con Under sampling, es preferible.

- No generas información sintética

Si la base queda muy pequeña, entonces hay que optar por Over-sampling.

Logit LASSO

Combinemos un poco de lo que hemos visto hasta ahora y apliquemoslo a clasificación. Construyamos un Logit LASSO:

- Es paramétrico (Interpretable!)
- Muy estable
- Podemos hacer selección variables
- En `R` esto sería con: `gamlr::gamlr(X=X, Y=Y, family = 'binomial')`

Logit LASSO: Compra de cripto

Veamos un ejemplo de probabilidad de compra de cripto-monedas.

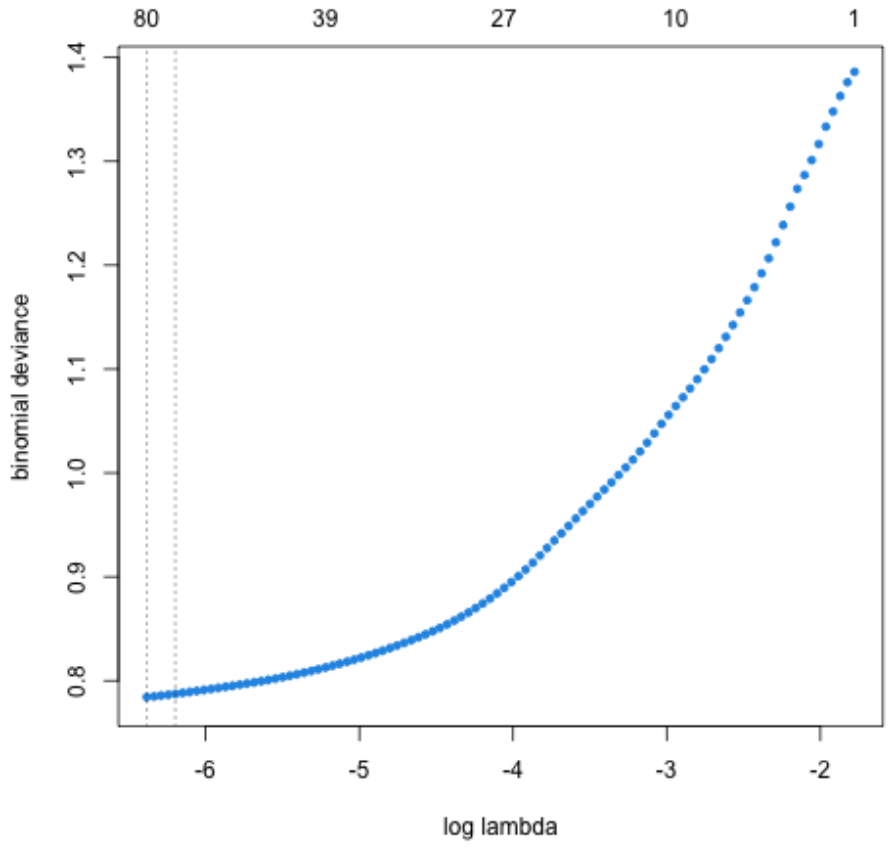
```
##############################################
# CV.LASSO
# k=5
##############################################
library(tidyverse)
library(tidymodels)
library(parallel)
library(gamlr)

# La Matriz de los covariates
Xs<-data %>% select(-id, -y)

# Quitamos el intercepto, lo incluye gamlr
Xs<-sparse.model.matrix(~. + 0, data = Xs)
```

Logit LASSO: Compra de cripto

```
# Y
y<-data$y

# Clusters for parallelization
detectCores()
cl<-makeCluster(12)
cl

lasso<-cv.gamlr(x = x, y = y, verb = T, cl = cl, family = 'binom
stopCluster(cl)

save(lasso, file = 'Modelos/cv_lasso.Rdata')

plot(lasso)
```

Logit LASSO

El modelo seleccionó 80 variables para minimizar el binomial deviance.

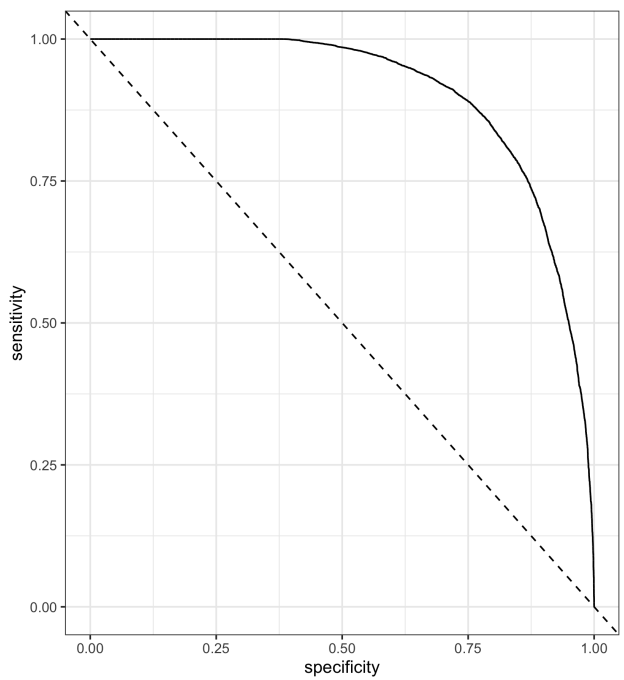


Logit LASSO

```
##############
# Predictions
# base con la verdad vs predecida
##############
# prediction vectors
pred<-drop(predict(lasso$gamlr,
                   X_validation ,
                  type = 'response',
                  select = x$seg.min))

eval<-bind_cols(data_validation$y, pred)


# ROC Curve
roc_curve(data = eval, truth = truth, "1")
```

Logit LASSO: ROC

```
ggplot(roc_curve, aes(x = specificity, y = sensitivity))+
        geom_abline(slope = -1, intercept = 1, linetype = 'dashe
        geom_path()+theme_bw()

ggsave(height = 6, width = 5.5, filename = 'Graficas/roc.png')
roc_auc(eval, truth, "1")
```



Clasificación Multiclase

- Multiclass models: Modelos con multiples categorías que son mutuamente excluyentes.
  - Para estos modelos, existen casi todos los algoritmos requeridos. En el caso de los modelos lineales, existe el multinominal logit y el distributed multinomial regression .
- Multi label classifications:
  - Las categorías no son mutuamente excluyentes

Clasificación Multiclase

Hay tres maneras de solucionar esto:

- Transformation approach

1. Binary relevance: Un modelo para cada categoria en la forma 1 vs all.
2. Pairwise: Una base para cada combinación de las categorías. Un modelo para cada escenario.
3. Powerset: Transformas las categorías en una sola categoría con todas las combinaciones.

- Adaption approach: Adaptar los modelos a un multi-label output (multivariados). Sólo las redes neuronales pueden hacer esto a día de hoy

K-Nearest Neighbors

La idea es estimar $P(y = j|x^{validation})$ por vecindades, dado las etiquetas que asignamos en la base de entrenamiento.

KNN: Cuál es la clase más común alrededor de X?

1. Escoges K
2. Dado un punto x_f, encuentra los K vecinos cercanos de la base:

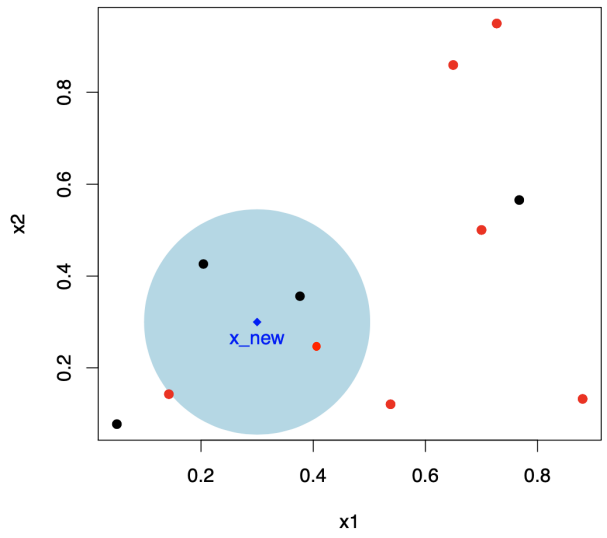
$$d(x_i, x_f) = [\Sigma_{j=1}^{p}(x_{ij} - x_f)^2]^{0.5}$$

3. Estima la probabilidad para cada clase en estos K vecinos
4. Escoge la clase con la mayor probabilidad

Ejemplo KNN

K-NN es un algoritmo de votación. Cada vecino vota.

En la gráfica, la vecindad está representada por el círculo.



Noten como para 3-nn los puntos negros son 2/3. Para 4-nn, serían 2/4. La el algoritmo es sensible a K.

KNN

Pros:

- Es un algoritmo muy sencillo
- Maneja de manera muy natural las clasificaciones multiclase.
- Puede ser mejor que los modelos lineales cuando $f(X)$ es compleja

Cons:

- Again, no paramétrico puede ser costoso computacionalmente.
- No podemos elegir variables.
- K es díficil de escoger. Nuestro mejor shot es Cross-Validation

Multinomial Logit LASSO

El Logit Multinomial puede ajustar las probabilidades para y multiclase. Para cada categoria:

$$P(y = 1|X)$$
$$P(y = 2|X)$$
$$\dots$$
$$P(y = M|X)$$

Tenemos que estimar el vector β_k para cada categoria...

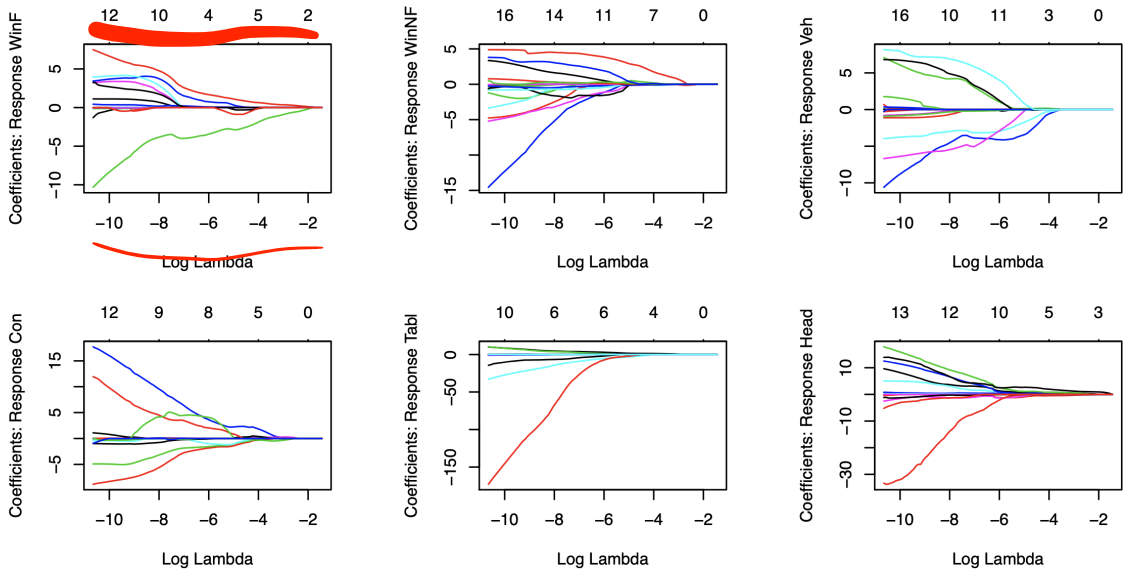
Multinomial regression

$$P(y_i = M|x_i) = \frac{e^{x_i' \beta_M}}{\sum_{j=1}^{M} e^{x_i' \beta_j}}$$

$$Dev(B) = -2\Sigma_i \Sigma_k y_{ik} log(p_{ik})$$

Multinomial regression

Usando `glmnet` con `family=multinomial`, podemos correr un multinomial LASSO



Cons: El multinomial logit puede ser muy lento, pues estas estimando parámetros para k categorias...

Distribute Multinomial Regression

Finalmente, existe una alternativa más veloz a Multinomial LASSO que se llama Distributed Multinomial regression.

La idea es genial. Resulta que la regresión multnomial es muy parecida a una transformación independiente donde cada categoria $y_k \sim Poisson(e^{x_i' \beta_k})$.

Así, cada clase se puede correr de manera independiente... Esto hace que no tengamos que estimar tantos parámetros y podamos agilizar el proceso.

La syntaxis en `R` es `dmr(covars,counts,cl)`

Lo más interesante es que también hace selección de variables tipo LASSO y puedes usar `parallel` para tener resultados más rápidos.

Resumen

Clasificación:

- Beware de los desbalances de clases!
- Usa la curva ROC para entender la sensibilidad y especificidad del modelo para distintos cortes
- Decide consciensudamente el punto de corte

Clasificación Binaria:

- K-NN o LASSO Logit (depende de $f(x)$). Aunque el LASSO Logit es más fácil de estimar
- Existen otros como XGB o Random Forest o Support Vector Machine

Clasificicación Multi clase:

- Si son mutuamente excluyentes: K-NN o DMR.
- Si no son mutuamente excluyentes: Decide entre hacer one vs all o one vs one.