Guía para la obtención de modelos de regresión logística con Python.

1 Obtención de modelos

Los pasos a seguir para construir un modelo de regresión logística con Python son los siguientes:

1. Cargar los datos en un DataFrame y preparar las variables predictoras (explicativas) y la variable objetivo. Los datos deben estar limpios y sin valores faltantes. Cargar los datos desde un archivo CSV o EXCEL.

```
data = pd.read_csv('tu_archivo.csv')
data = pd.read_excel('tu_archivo.xlsx')
```

   Una vez cargados los datos se definen las variables predictoras (X) y la variable objetivo (Y)

```
X = data[['Variable1', 'Variable2', ...]]
Y = data['VariableObjetivo']
```

2. Dividir los datos en un conjunto de entrenamiento y un conjunto de prueba para evaluar el modelo.

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size
    =0.2, random_state=1234567)
```

   Al ser la variable objetivo categórica le indico que la variable respuesta es un entero para la construcción del modelo

```
y_train, y_test = y_train.astype(int), y_test.astype(int)
```

   $train\_test\_split(X, Y, test\_size = 0.2, random\_state = 1234567)$ divide los datos en conjuntos de entrenamiento y prueba de acuerdo con los siguientes parámetros:

   - X y Y son dos conjuntos de datos. X contiene las variables predictoras, mientras que Y contiene la variable objetivo que se quiere predecir.
   - $test\_size = 0.2$ - Establece la proporción del conjunto de prueba en 0.2, lo que significa que el 20% de los datos se utilizarán como conjunto de prueba y el 80% restante se utilizarán como conjunto de entrenamiento.

- *random_state* = 1234567 - Establece una semilla para la generación de números aleatorios. Esto asegura que la división de datos sea reproducible. Si se usa la misma semilla en diferentes ejecuciones, se obtendrá la misma división de datos. Esto es útil para la reproducibilidad de los resultados.

Después de ejecutar la función *train_test_split*, se obtienen los siguientes conjuntos de datos:

- *X_train*.- Conjunto de entrenamiento que contiene las variables predictoras para entrenar el modelo.
- *X_test*.- Conjunto de prueba que contiene las variables predictoras, pero se utiliza para evaluar el rendimiento del modelo.
- *Y_train* - Variable objetivo correspondiente al conjunto de entrenamiento.
- *Y_test* - Variable objetivo correspondiente al conjunto de prueba.

3. Ajustar un modelo de regresión logística. La función *glm*, definida en *FuncionesMineria*, permite ajustar modelos de regresión logística en Python, con la capacidad de manejar variables continuas, categóricas e interacciones entre variables.

```
    def glm(varObjBin, datos, var_cont, var_categ, var_interac = []):
    """
    Ajusta un modelo de regresión logística a datos binarios.

    Parameters:
        varObjBin (array-like): Variable objetivo binaria.
        datos (DataFrame): Conjunto de datos que incluye las variables
            predictoras.
        var_cont (list): Lista de nombres de variables continuas.
        var_categ (list): Lista de nombres de variables categóricas.
        var_interac (list, opcional): Lista de interacciones entre
            variables (por defecto es una lista vacía)..

    Returns:
        dict: Un diccionario que contiene el modelo ajustado, las
            variables utilizadas y el conjunto de datos utilizado en la
             predicción.
    """

    # Preprocesar los datos aplicando la función crear_data_modelo
    datos = crear_data_modelo(datos, var_cont, var_categ, var_interac)

    # Crear un modelo de regresión logística y ajustarlo a los datos
    output = {
        'Modelo': LogisticRegression(max_iter=1000, solver='newton-cg')
            .fit(datos, varObjBin),
        'Variables': {
            'cont': var_cont,
            'categ': var_categ,
            'inter': var_interac
        },
```

```
        'X': datos
    }

    return output
```

Para crear los datos de entrada al modelo se utiliza la función propia *crear_data_modelo*. Esta función convierte las variables categóricas en *dummies*. Para evitar la colinealidad perfecta entre las variables *dummies*, se elimina una de las categorías para evitar multicolinealidad, concretamente la primera (ordenadas alfabéticamente o numéricamente) que será considerada como categoría de referencia. Además, genera *interacciones* entre las variables seleccionadas. Finalmente, el conjunto de datos contiene las variables continuas, las variables categóricas que se han convertido en *dummies* y las *interacciones* entre las variables seleccionadas.

la función *LogisticRegression* de la biblioteca *sklearn* ajusta un modelo de regresión logística utilizando las variables predictoras contenidas en *datos* y la variable objetivo contenida en $varObjBin$.

Si se quiere obtener información acerca de los parámetros estimados y de los contrastes de hipótesis sobre los parámetros, debemos utilizar la función propia (definida en $FuncionesMineria$): $summary\_glm(modelo, varObjBin, datos)$.

Por ejemplo, para el conjunto de datos de características del vino se tiene:

```
# Identificamos las variables predictoras continuas.
var_cont = ['pH', 'Acidez', 'Azucar']
# Identificamos las variables predictoras categóricas.
var_categ = ['Etiqueta', 'CalifProductor', 'Clasificacion', '
    prop_missings']
# Creamos el modelo de regresión logística
modelo = glm(y_train, x_train, var_cont, var_categ)
# Resumen del modelo
summary_glm(modelo['Modelo'], y_train, modelo2['X'])
```

```
Out[613]:
{'Contrastes':                                Variable  Estimate  ...   p value  signif
 0                          (Intercept)  2.066288  ...  0.000000     ***
 1                                   pH -0.189176  ...  0.009554      **
 2                               Acidez -0.113098  ...  0.071286       .
 3                               Azucar  0.002960  ...  0.044579       *
 4                            Etiqueta_M  0.936009  ...  0.000000     ***
 5                           Etiqueta_MB -0.314114  ...  0.294390
 6                           Etiqueta_MM  1.370815  ...  0.000000     ***
 7                            Etiqueta_R  0.399648  ...  0.001205      **
 8                     CalifProductor_2 -0.131444  ...  0.440672
 9                     CalifProductor_3 -0.255044  ...  0.135403
 10                    CalifProductor_4 -0.998579  ...  0.000000     ***
 11                CalifProductor_5-12 -1.766683  ...  0.000000     ***
 12                   Clasificacion_**  2.349508  ...  0.000000     ***
 13                  Clasificacion_***  4.144950  ...  0.000000     ***
 14                 Clasificacion_****  3.303037  ...  0.000000     ***
 15          Clasificacion_Desconocido -1.573148  ...  0.000000     ***
 16  prop_missings_0.07692307692307693 -0.253182  ...  0.073523       .
 17  prop_missings_0.15384615384615385 -0.405578  ...  0.017376       *
 18  prop_missings_0.23076923076923078 -0.511834  ...  0.073689       .
 19   prop_missings_0.3076923076923077 -0.006022  ...  0.992909

 [20 rows x 6 columns],
 'BondadAjuste':           LLK         AIC          BIC
 0 -1540.35976  3084.71952  3097.790372}
```

Figure 1: Resumen función glm()

Como podemos observar en la Figura 1, nos ofrece información acerca de los parámetros (y su significatividad), el AIC, el BIC y los logaritmos de las verosimilitudes para que podamos evaluar la bondad del modelo.

Los objetos de tipo *glm* contienen toda la información del modelo. En particular, podemos destacar:

- $modelo['Modelo'].coef\_$ contiene los coeficientes estimados.
- $modelo['Modelo'].predict\_proba(datos\_nuevos)[:, 1]$ contiene los valores de la probabilidad estimada de pertenecer a la clase 1 para la variable objetivo a partir del modelo ajustado con las variables explicativas contenidas en el conjunto de datos $'datos\_nuevos'$. Por tanto, permite obtener las probabilidades de pertenecer a la clase 1 para un nuevo conjunto de datos.

Este nuevo conjunto de datos debe contener exactamente las mismas variables explicativas que han sido usadas en la estimación del modelo, es decir, si hay variables categóricas entre las variables explicativas se deben crear variables *dummy* y si hay *interacciones* entre las variables también deben ser creadas. Esto se realiza con la función propia $'crear\_data\_modelo'$ creada en el fichero $'FuncionesMineria'$. Concretamente, "$datos\_nuevos$" se genera de la siguiente forma:

```
datos_nuevos = crear_data_modelo(x_test, var_cont, var_categ, var_interac)
```

Además, para obtener la bondad del ajuste mediante el cálculo del $Pseudo - R^2$ se ha creado una función en Python:

```
pseudoR2(modelo, x_train, y_train, var_cont, var_categ, var_interac)
```

Para obtener una estimación de la bondad del ajuste del modelo de regresión logística más realista se debe calcular el valor del $Pseudo - R^2$ para los datos test. Para asegurarnos de que el modelo estimado no ha sido sobreajustado, la diferencia en el valor del $Pseudo - R^2$ para los datos de entrenamiento y test deben ser similares y no obtener un valor del $Pseudo - R^2$ en el entrenamiento muy superior al obtenido para los datos test.

Recordemos que, a la hora de evaluar un modelo de regresión, es importante hacerse una idea de la importancia de las variables que lo componen. Para ello, una alternativa es calcular como se ve afectado el mismo al eliminar cada una de las variables individualmente. De esta manera, las variables más importantes harían que el modelo empeore mucho al eliminarlas mientras que las variables menos útiles apenas tendrían efecto sobre la calidad del mismo.

En el caso de la regresión logística, podemos medir este "empeoramiento" en términos del $Pseudo - R2$. Para obtener un gráfico que nos muestre la pérdida a la que dan lugar cada una de las variables, podemos usar la siguiente función:

```
impVariablesLog(modelo, y_train, x_train, var_cont, var_categ, var_interac)
```

Por último, es importante conocer siempre el número de parámetros que componen el modelo para tener una idea de la complejidad del mismo. Una manera rápida de obtenerlo es contando el número de parámetros que contiene: $len(modelo['Modelo'].coef\_[0])$.

2 Selección de variables

En Python podemos llevar a cabo los métodos de selección de variables ya estudiados: hacia delante (forward), hacia atrás (backward) y paso a paso (stepwise).

En cada iteración de los métodos se evalúa el AIC/BIC del modelo resultante. De esa forma, se incluyen o eliminan variables siempre y cuando mejore dicho estadístico. Para poder llevar a cabo dicha selección, es necesario que indiquemos el rango de los modelos a evaluar, es decir, el modelo con el máximo número de variables y el modelo con el mínimo número de variables. Este proceso se realiza a partir de las siguientes funciones:

```
glm_backward(varObjCont, datos, var_cont, var_categ, var_interac = [],
    metodo = 'AIC')
```

Esta función parte del modelo con todas las variables indicadas ($var\_cont, var\_categ, var\_interac$) y elimina variables en cada iteración según el criterio AIC. Pudiendo llegar, como mínimo, al modelo con ninguna variable. Si en lugar de usar el criterio AIC se utiliza el criterio BIC en la función únicamente hay que cambiar el método ($metodo =' BIC'$).

```
glm_forward(varObjCont, datos, var_cont, var_categ, var_interac = [],
    metodo = 'AIC')
```

Esta función parte del modelo sin ninguna variable y va añadiendo variables en cada iteración según el criterio AIC. Pudiendo llegar como máximo, al modelo con todas las variables indicadas ($var\_cont, var\_categ, var\_interac$). Si en lugar de usar el criterio AIC se utiliza el criterio BIC en la función únicamente hay que cambiar el método ($metodo =' BIC'$).

```
glm_stepwise(varObjCont, datos, var_cont, var_categ, var_interac, metodo =
    'AIC')
```

Esta función parte del modelo sin ninguna variable y va añadiendo variables en cada iteración según el criterio AIC. La diferencia con el método forward es que si una variable ha sido añadida en sucesivas iteraciones puede volver a ser eliminada. Pudiendo llegar como máximo, al modelo con todas las variables indicadas ($var\_cont, var\_categ, var\_interac$) y como mínimo, al modelo sin ninguna de ellas. Si en lugar de usar el criterio AIC se utiliza el criterio BIC en la función únicamente hay que cambiar el método ($metodo =' BIC'$).

2.1 Lista de todas las posibles interacciones

De cara a considerar todos los posibles efectos que puedan aportar información sobre la variable objetivo, sería interesante poder generar automáticamente una fórmula que los contenga todos para poderla usar en el proceso de selección de variables o de búsqueda exhaustiva. Para ello, podemos crear en Python diferentes formas de crear interacciones, en función de nuestros datos y objetivos.

No obstante, siempre que añadamos interacciones tenemos que tener en cuenta nuestras limitaciones de computación y tiempo, pues es común en esta parte del proceso de predicción encontrar con dichas limitaciones.

A continuación se muestran diferentes alternativas para crear interacciones.

- Queremos todas las posibles combinaciones n a n de una única lista con variables

```
#======Iteraciones n a n de unalista===================
 interacciones_unicas = list(itertools.combinations(list1, n)) #
#=======================================================
```

- Si tenemos dos listas diferentes de variables, por ejemplo por un lado las variables continuas y por otro las variables categóricas, nos podría interesar tener todas las posibles interacciones resultantes de combinar los elementos de la primera lista con los de la segunda.

```
# == Interacciones entre 2 listas de variables distintas ======
# Genero interacciones:
 interacciones = list(itertools.product(list1, list2))
# itertools.product genera duplicados
# eliminamos duplicados
 interacciones_unicas = []
for x in interacciones:
    if (sorted(x) not in [sorted(t) for t in interacciones_unicas]) and
        (x[0] != x[1]):
        interacciones_unicas.append(x)
#============================================================
```

Recordamos que para unir dos listas diferentes en python, hay que hacer simplemente lo siguiente:

```
    list_total = list1 + list2
```

3 Curva ROC

La curva ROC, así como su área se calculan como medida de bondad de ajuste en los modelos de Regresión Logística y se obtienen de la siguiente forma:

```
# Una vez ajustado el modelo con regresión logística. Se calculan las
    probabilidades de la categoría 1 para el conjunto de datos test.
x_test = crear_data_entrenamiento(x_test, var_cont, var_categ, var_interac)
# Calcula las probabilidades de la clase positiva
y_prob = modelo['Modelo'].predict_proba(x_test_modelo)[:, 1]
# Calcula la tasa de falsos positivos (FPR) y la tasa de verdaderos
    positivos (TPR)
fpr, tpr, thresholds = roc_curve(y_test, y_prob)

# Grafica la curva ROC
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='Curva ROC')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('Tasa de Falsos Positivos (FPR)')
plt.ylabel('Tasa de Verdaderos Positivos (TPR)')
plt.title('Curva ROC')
plt.legend(loc='lower right')
plt.show()
```

La función *roc_curve* calcula la Curva *ROC* y devuelve tres arrays:

- *fpr* (False Positive Rate): Contiene las tasas de falsos positivos en diferentes puntos de corte.
- *tpr* (True Positive Rate o Sensibilidad): Contiene las tasas de verdaderos positivos en diferentes puntos de corte.
- *thresholds*: Contiene los puntos de corte utilizados para calcular las tasas de falsos positivos y verdaderos positivos. Estos puntos de corte son valores en el rango $[0, 1]$ que determinan cómo se clasifican las observaciones.

La función *roc_auc_score* calcula el área bajo la *Curva ROC*.

Podemos realizar tanto el cálculo del area bajo la curva como la representación del gráfico, llamando a la función propia $curva\_roc(x\_test\_modelo, y\_test, modelo)$ introduciendo como variables de entrada los datos *test* y el modelo ajustado.

4 Comparación de modelos a partir de validación cruzada

Este método consiste en dividir el conjunto de datos en submuestras e iterativamente construir el modelo con todas las observaciones menos las de una submuestra y evaluarlo a continuación con las observaciones de dicha submuestra excluida.

La función que nos permitirá realizar este proceso de validación cruzada es la siguiente:

```
def validacion_cruzada_glm(n_cv, datos, varObjBin, var_cont, var_categ,
    var_interac = []):
"""
Realiza una validación cruzada para evaluar un modelo de regresión logí
    stica con datos binarios.

Parameters:
    n_cv (int): Número de particiones en la validación cruzada (k-fold)
        .
    datos (DataFrame): Conjunto de datos que incluye las variables
        predictoras.
    varObjBin (array-like): Variable objetivo binaria.
    var_cont (list): Lista de nombres de variables continuas.
    var_categ (list): Lista de nombres de variables categóricas.
    var_interac (list, opcional): Lista de interacciones entre
        variables.

Returns:
    list: Una lista de puntuaciones ROC AUC obtenidas en cada partición
        de la validación cruzada.
"""
# Prepara los datos según las variables de entrada y las interacciones.
datos = crear_data_modelo(datos, var_cont, var_categ, var_interac)


# Realiza la validación cruzada utilizando el modelo de regresión logí
    stica y calcula el ROC AUC.
return list(cross_val_score(LogisticRegression(max_iter=1000, solver='
    newton-cg'), datos, varObjBin, cv = n_cv, scoring = 'roc_auc'))
```

La diferencia con la validación cruzada en Regresión Lineal es la métrica utilizada. En Regresión Lineal se utiliza el R^2 y en Regresión Logística se utiliza el área bajo la *curva ROC* (AUC).

Hay que indicar el número de grupos de la validación cruzada (*n_cv*), el conjunto de datos, la variable objetivo, las variables continuas, categóricas e interacciones del modelo que se ha estimado.

Con el objetivo de obtener una evaluación más robusta y confiable del rendimiento de un modelo, en lugar de realizar este proceso de validación cruzada solo una vez, se repite el proceso múltiples veces. Cada repetición implica una nueva división del conjunto de datos en submuestras e iterativamente construir el modelo con todas las observaciones menos las de una submuestra y evaluarlo a continuación con las observaciones de dicha submuestra excluida.

A este método se le denomina validación cruzada repetida. Una forma de interpretar fácilmente los últimos resultados es construir un diagrama de cajas con todos los R^2 obtenidos al repetir el proceso de validación cruzada. Si se representan estos diagramas de cajas para distintos modelos sobre la misma escala, podremos concluir que modelo es preferible sobre el resto.

```
# Validacion cruzada repetida para ver que modelo es mejor
# Crea un DataFrame vacío para almacenar resultados
results = pd.DataFrame({
    'AUC': []
    , 'Resample': []
    , 'Modelo': []
})

# Realiza el siguiente proceso 20 veces (representado por el bucle `for rep
    in range(20)`)
for rep in range(20):
    # Realiza validación cruzada en cuatro modelos diferentes y almacena
        sus R-squared en listas separadas
    modelo1VC = validacion_cruzada_glm(5, x_train, y_train, var_cont1,
        var_categ1)
    modelo2VC = validacion_cruzada_glm(5, x_train, y_train, var_cont2,
        var_categ2)


    # Crea un DataFrame con los resultados de validación cruzada para esta
        repetición
    results_rep = pd.DataFrame({
        'AUC': modelo1VC + modelo2VC
        , 'Resample': ['Rep' + str((rep + 1))]*5*2  # Etiqueta de repetició
            n (5 repeticiones 2 modelos)
        , 'Modelo': [1]*5 + [2]*5  # Etiqueta de modelo (2 modelos 5
            repeticiones)
    })
    results = pd.concat([results, results_rep], axis = 0)

# Boxplot de la validacion cruzada
plt.figure(figsize=(10, 6))  # Crea una figura de tama o 10x6
plt.grid(True)  # Activa la cuadrícula en el gráfico
# Agrupa los valores de AUC por modelo
grupo_metrica = results.groupby('Modelo')['AUC']
# Organiza los valores de R-squared por grupo en una lista
boxplot_data = [grupo_metrica.get_group(grupo).tolist() for grupo in
    grupo_metrica.groups]
# Crea un boxplot con los datos organizados
plt.boxplot(boxplot_data, labels=grupo_metrica.groups.keys())  # Etiqueta
    los grupos en el boxplot
# Etiqueta los ejes del gráfico
plt.xlabel('Modelo')  # Etiqueta del eje x
plt.ylabel('AUC')  # Etiqueta del eje y
plt.show()  # Muestra el gráfico
```

Para el ejemplo de las características del vino se ha utilizado la siguiente sentencia y obtenido los resultados que aparecen en la Figura 2.

```
var_cont1 = ['Acidez', 'AcidoCitrico', 'Azucar', 'CloruroSodico', 'Densidad', 'pH', 'Sulfatos',
             'Alcohol', 'PrecioBotella']
var_categ1 = ['Etiqueta', 'CalifProductor', 'Clasificacion', 'Region', 'prop_missings']


var_cont2 = ['pH', 'Acidez', 'Azucar']
var_categ2 = ['Etiqueta', 'CalifProductor', 'Clasificacion', 'prop_missings']

var_cont3 = ['Densidad', 'Acidez', 'CloruroSodico']
var_categ3 = ['CalifProductor', 'Clasificacion', 'prop_missings']

var_cont4 = var_cont3
var_categ4 = var_categ3
var_interac4 = [('Clasificacion', 'prop_missings')]

var_cont5 = []
var_categ5 = ['Clasificacion', 'CalifProductor', 'Etiqueta']

var_cont6 = []
var_categ6 = ['Clasificacion', 'CalifProductor', 'Etiqueta']
var_interac6 = [('Clasificacion', 'Etiqueta')]

# Validacion cruzada repetida para ver que modelo es mejor
# Crea un DataFrame vacío para almacenar resultados
results = pd.DataFrame({
    'AUC': []
    , 'Resample': []
    , 'Modelo': []
})

# Realiza el siguiente proceso 20 veces (representado por el bucle `for rep in range(20)`)
for rep in range(20):
    # Realiza validación cruzada en cuatro modelos diferentes y almacena sus R-squared en listas separadas
    modelo1VC = validacion_cruzada_glm(5, x_train, y_train, var_cont1, var_categ1)
    modelo2VC = validacion_cruzada_glm(5, x_train, y_train, var_cont2, var_categ2)
    modelo3VC = validacion_cruzada_glm(5, x_train, y_train, var_cont3, var_categ3)
    modelo4VC = validacion_cruzada_glm(5, x_train, y_train, var_cont4, var_categ4, var_interac4)
    modelo5VC = validacion_cruzada_glm(5, x_train, y_train, var_cont5, var_categ5)
```

```
    modelo6VC = validacion_cruzada_glm(5, x_train, y_train, var_cont6,
        var_categ6, var_interac6)

    # Crea un DataFrame con los resultados de validación cruzada para esta
        repetición
    results_rep = pd.DataFrame({
        'AUC': modelo1VC + modelo2VC + modelo3VC + modelo4VC + modelo5VC +
            modelo6VC
        , 'Resample': ['Rep' + str((rep + 1))]*5*6  # Etiqueta de repetició
            n (5 repeticiones 6 modelos)
        , 'Modelo': [1]*5 + [2]*5 + [3]*5 + [4]*5 + [5]*5 + [6]*5 #
            Etiqueta de modelo (6 modelos 5 repeticiones)
    })
    results = pd.concat([results, results_rep], axis = 0)

# Boxplot de la validacion cruzada
plt.figure(figsize=(10, 6))  # Crea una figura de dimensiones 10x6
plt.grid(True)  # Activa la cuadrícula en el gráfico
# Agrupa los valores de AUC por modelo
grupo_metrica = results.groupby('Modelo')['AUC']
# Organiza los valores de R-squared por grupo en una lista
boxplot_data = [grupo_metrica.get_group(grupo).tolist() for grupo in
    grupo_metrica.groups]
# Crea un boxplot con los datos organizados
plt.boxplot(boxplot_data, labels=grupo_metrica.groups.keys())  # Etiqueta
    los grupos en el boxplot
# Etiqueta los ejes del gráfico
plt.xlabel('Modelo')  # Etiqueta del eje x
plt.ylabel('AUC')  # Etiqueta del eje y
plt.show()  # Muestra el gráfico
```

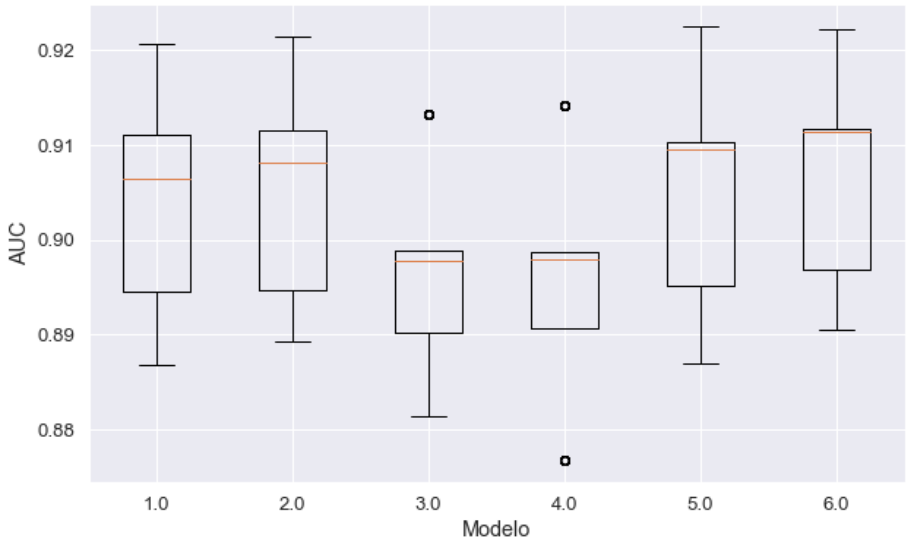

Figure 2: Gráfico de barras y bigotes para validación cruzada repetida

Como se puede comprobar en la figura 2, todos los modelos tienen una 'bondad media' bastante similar, así como una variabilidad también parecida. Por tanto, para determinar el mejor modelo habría que observar el número de parámetros de cada uno de ellos y elegir el que menor número de parámetros tenga. Otra ventaja de estos gráficos es que permiten evaluar fácilmente la mejora conseguida al incluir/eliminar factores. Concretamente, se observa que las variables continuas y las interacciones no aportan información al modelo. En ocasiones, no obstante, no queda claro visualmente que modelo es preferible. Para esos casos, podemos calcular la media y desviación típica del área bajo la curva ROC (AUC) de las repeticiones de la validación cruzada como complemento a los diagramas de cajas anteriores, utilizando el siguiente código:

```
# Calcular la media de las métricas R-squared por modelo
media_r2 = results.groupby('Modelo')['AUC'].mean()
# Calcular la desviación estándar de las métricas R-squared por modelo
std_r2 = results.groupby('Modelo')['AUC'].std()
```

5 Selección del punto de corte

Como ya se ha comentado, la gran diferencia con los modelos de regresión lineal es que para regresión logística es necesario determinar la probabilidad a partir de la cual se consideraría una observación como evento.

Para ello, vamos a obtener, para una rejilla de posibles puntos de corte y el conjunto de datos de prueba, el valor de la tasa de acierto, la sensibilidad, la especificidad y el índice de Youden, y, a partir de ahí, tomar la decisión. Recurriremos a la función $confusion\_Matrix$. Esta función

requiere que le demos, para el conjunto de datos deseado, los valores predichos y los reales.

Para automatizar el proceso de predicción a través del modelo y producir solo las medidas indicadas, podemos utilizar la función propia *sensEspCorte* (creada en *FuncionesMineria*), la cual obtiene los valores de estas medidas solo para un punto de corte concreto, que es el indicado en *ptoCorte*:

```
    def sensEspCorte(modelo, dd, varObjBin, ptoCorte, var_cont, var_categ,
        var_interac = []):
    """
    Calcula medidas de calidad para un punto de corte dado.

    Parameters:
        modelo (dict): Diccionario que contiene el modelo ajustado.
        dd (DataFrame): Conjunto de datos de prueba.
        varObjBin (array-like): Variable objetivo binaria.
        ptoCorte (float): Punto de corte para la clasificación.
        var_cont (list): Lista de nombres de variables continuas.
        var_categ (list): Lista de nombres de variables categóricas.
        var_interac (list, opcional): Lista de interacciones entre
            variables.

    Returns:
        DataFrame: Un DataFrame que contiene las medidas de calidad para el
             punto de corte dado.
    """
    # Prepara los datos de prueba según el modelo
    if len(var_interac) > 0:
        dd = crear_data_modelo(dd, var_cont, var_categ, var_interac)
    else:
        dd = pd.get_dummies(dd[var_cont + var_categ], columns=var_categ,
            drop_first=True)

    # Calcula las probabilidades de la clase positiva
    probs = modelo.predict_proba(dd)[:, 1]

    # Realiza la clasificación en función del punto de corte
    preds = (probs > ptoCorte).astype(int)

    # Calcula la matriz de confusión
    cm = confusion_matrix(varObjBin, preds)
    tn, fp, fn, tp = cm.ravel()

    # Calcula medidas de calidad
    output = pd.DataFrame({
        'PtoCorte': [ptoCorte],
        'Accuracy': [(tp + tn) / (tn + fp + fn + tp)],
        'Sensitivity': [tp / (tp + fn)],
        'Specificity': [tn / (tn + fp)],
        'PosPredValue': [tp / (tp + fp)],
```

```
        'NegPredValue': [tn / (tn + fn)]
    })

    return output
```

Para determinar el punto de corte óptimo se calculan las medidas anteriormente obtenidas con la función *sensEspCorte* para una rejilla de posibles puntos de corte y se elige como punto de corte óptimo el que maximice la *Accuracy* o el índice de *Youden*. Todo esto se realiza con el siguiente código:

```
# Generamos una rejilla de puntos de corte
posiblesCortes = np.arange(0, 1.01, 0.01).tolist()  # Generamos puntos de corte de 0 a 1 con intervalo de 0.01
rejilla = pd.DataFrame({
    'PtoCorte': [],
    'Accuracy': [],
    'Sensitivity': [],
    'Specificity': [],
    'PosPredValue': [],
    'NegPredValue': []
})  # Creamos un DataFrame para almacenar las métricas para cada punto de corte

for pto_corte in posiblesCortes:  # Iteramos sobre los puntos de corte
    rejilla = pd.concat(
        [rejilla, sensEspCorte(modelo['Modelo'], x_test, y_test, pto_corte, var_cont, var_categ)],
        axis=0
    )  # Calculamos las métricas para el punto de corte actual y lo agregamos al DataFrame

rejilla['Youden'] = rejilla['Sensitivity'] + rejilla['Specificity'] - 1  # Calculamos el índice de Youden
rejilla.index = list(range(len(rejilla)))  # Reindexamos el DataFrame para que los índices sean consecutivos



# Graficamos el índice de Youden en función de los posibles puntos de corte
plt.plot(rejilla['PtoCorte'], rejilla['Youden'])
plt.xlabel('Posibles Cortes')
plt.ylabel('Youden')
plt.title('Youden')
plt.show()

# Graficamos la precisión (Accuracy) en función de los posibles puntos de corte
plt.plot(rejilla['PtoCorte'], rejilla['Accuracy'])
plt.xlabel('Posibles Cortes')
plt.ylabel('Accuracy')
plt.title('Accuracy')
```

```
35 plt.show()
36
37 # Encontramos el punto de corte que maximiza el índice de Youden
38 p1 = rejilla['PtoCorte'][rejilla['Youden'].idxmax()]
39
40 # Encontramos el punto de corte que maximiza la precisión (Accuracy)
41 p2 = rejilla['PtoCorte'][rejilla['Accuracy'].idxmax()]
```