Ejercicio3_Clasificacion

January 10, 2025

1 Ejercicio 3: Clasificación

Felipe Andres Castillo

[1]:
```
using Pkg
#Pkg.update()
#Pkg.precompile()
#Pkg.activate("my_MLJ_env", shared=true)
#Pkg.add("MLJ")
#Pkg.add("RDatasets")
#Pkg.add("GLM")
#Pkg.add("GLMNet")
#Pkg.add(url = "https://github.com/diegozea/ROC.jl")
#Pkg.add("Plots")
#Pkg.add("DecisionTree")
#Pkg.add("NearestNeighbors")
#Pkg.add("LIBSVM")

using RDatasets
using DataFrames
using MLJ #tuve problemas al ejecutar using MLJ, por lo que eliminé la carpeta .
 ↪julia/compiled y después ejecuté Pkg.precompile()
#using CairoMakie
using GLMNet
using Random
using ROC
using DecisionTree
using Plots
using NearestNeighbors
using LIBSVM

include("./../src/exercise1_code.jl");
```

1.1 Introducción

Un **índice bursátil** es un indicador de la bolsa de valores que actúa como un termómetro: tiene la capacidad de hacernos ver, en un solo vistazo, el movimiento mayoritario de las empresas de dicho mercado. **Miden el crecimiento o el decrecimiento del valor de las acciones** que lo

componen, para dar una imagen del comportamiento del mercado al completo. Existen múltiples índices en todo el mundo y son de gran importancia para poder analizar cómo varía el precio de una serie de activos cotizados con unas características determinadas.

Son muy utilizados por los gestores para llevar a cabo comparativas entre todos ellos y poder operar e invertir en función de los datos objeto de análisis. Además de como guía para dar una perspectiva del mercado, estos indicadores económicos se pueden utilizar para hacer un análisis y estudiar opciones a la hora de medir la rentabilidad y el riesgo del mercado, así como el rendimiento de un gestor de activos, o crear carteras que puedan reproducir el comportamiento de dicho índice, entre otros.

En este reporte, se utilizan datos del índice bursátil Standard & Poor's para aplicar diversos algoritmos de clasificación con el objetivo de predecir si el mercado estará al alza o a la baja, basándose en los rendimientos porcentuales de un día específico.

1.2 Stock Market Data

[2]:
```
smarket = dataset("ISLR", "Smarket")
rows, cols = dataShape(smarket)
println("El DataFrame consta de $(cols) columnas y $(rows) registros")
describe(smarket)
```

```
El DataFrame consta de 9 columnas y 1250 registros
```

[2]:

	variable	mean	min	median	max	nmissing	eltype
	Symbol	Union…	Any	Union…	Any	Int64	DataType
1	Year	2003.02	2001.0	2003.0	2005.0	0	Float64
2	Lag1	0.0038344	-4.922	0.039	5.733	0	Float64
3	Lag2	0.0039192	-4.922	0.039	5.733	0	Float64
4	Lag3	0.001716	-4.922	0.0385	5.733	0	Float64
5	Lag4	0.001636	-4.922	0.0385	5.733	0	Float64
6	Lag5	0.0056096	-4.922	0.0385	5.733	0	Float64
7	Volume	1.4783	0.35607	1.42295	3.15247	0	Float64
8	Today	0.0031384	-4.922	0.0385	5.733	0	Float64
9	Direction		Down		Up	0	CategoricalValue{String, UInt8}

Este *data set* contiene el valor de los rendimientos porcentuales del índice bursátil S&P 500 durante 1250 días, desde principios de 2001 hasta finales de 2005. Para cada día, se tiene registrado los rendimientos porcentuales de cada uno de los cinco días de negociación anteriores, etiquetados como **Lag1**, …, **Lag5**. También se tiene registrado **Volume** (que es la cantidad de acciones negociadas el día anterior, en miles de millones), **Today** (el rendimiento porcentual en la fecha en cuestión) y **Direction** (si el mercado estaba en alza o en baja en esta fecha). Esta ultima es nuestra **variable dependiente**, es decir, será la variable que necesitamos predecir usando los diversos métodos de clasificación.

1.3 Preparación de los datos

1.3.1 Missing values

[3]:
```
for col in names(smarket)
    println("La columna $(col) tiene $(dataMissingPercentage(smarket[!,col])) %␣
 ↪de datos faltantes")
end
```

```
La columna Year tiene 0.0 % de datos faltantes
La columna Lag1 tiene 0.0 % de datos faltantes
La columna Lag2 tiene 0.0 % de datos faltantes
La columna Lag3 tiene 0.0 % de datos faltantes
La columna Lag4 tiene 0.0 % de datos faltantes
La columna Lag5 tiene 0.0 % de datos faltantes
La columna Volume tiene 0.0 % de datos faltantes
La columna Today tiene 0.0 % de datos faltantes
La columna Direction tiene 0.0 % de datos faltantes
```

En el *data set* no hay ningún dato faltante.

1.3.2 One Hot Encoding

La variable **Direction** tiene dos posibles valores categoricos: *Up* y *Down*. Estos valores necesitan ser transformados a valores númericos 0 y 1.

[4]:
```
# y será el array que contenga la codificación de la variable Direction
y = map( x -> if x == "Down" 0 elseif x == "Up" 1 end, smarket.Direction);
```

[5]:
```
# X serán los datos númericos
X = select(smarket, Not(:Direction));
```

1.3.3 Datos de entrenamiento y prueba

Lo siguiente es dividir nuestro conjunto de datos en dos subconjuntos: el de entrenamiento (70%) y el de prueba (30%).

[6]:
```
Random.seed!(18)

# rows es el número total de registros
# Muestra aleatoria de indices para los datos de entrenamiento (70% de los␣
 ↪datos)
idx_train = sample(1:rows, Int(round(0.7*rows)), replace = false)
# Indices restantes para los datos de prueba
idx_test = Not(idx_train)
#Datos entrenamiento
train_X = Matrix(X[idx_train,:])
train_y = y[idx_train]
#Datos prueba
test_X = Matrix(X[idx_test,:])
```

```
test_y = y[idx_test];
```

1.4 Algoritmos

1.4.1 Ridge

La regresión lineal (que es el método más sencillo entre los modelos lineales) estima la variable objetivo mediante una combinación lineal de las características predictivas, minimizando la *suma de los cuadrados de los residuales* (RSS)

$$RSS = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

donde n es el número de muestras, y_i es el valor real observado y $f(x_i)$ el valor predicho.

La **Regresión Rigde** regulariza el modelo resultante imponiendo una penalización al tamaño de los coeficientes de la relación lineal entre las características predictivas y la variable objetivo, con el objetivo de corregir la multicolinealidad en el análisis de regresión. En este caso, los coeficientes calculados minimizan la suma de los cuadrados de los residuos penalizada al añadir el cuadrado de la norma L2 del vector formado por los coeficientes:

$$RSS = \sum_{i=1}^{n} (y_i - f(x_i))^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

donde β_j son los coeficientes de la relación lineal y λ es el parámetro que controla el grado de penalización.

[7]:
```
# ridge_models es el conjunto de modelos generados para diferentes valores de
# pero sólo nos interesa el modelo que tenga el mejor valor de  (menor RSS
 ↪promedio, MSE); para ello se usa la validación cruzada:
ridge_models = glmnet(train_X, train_y, alpha = 0)#, standardize=true)
cv_rm = glmnetcv(train_X, train_y, alpha = 0)
```

[7]:
```
Least Squares GLMNet Cross Validation
100 models for 8 predictors in 10 folds
Best   0.036 (mean loss 0.119, std 0.002)
```

[8]:
```
# lambda y modelo elegido
 = ridge_models.lambda[argmin(cv_rm.meanloss)]
ridge_model = glmnet(train_X, train_y, alpha = 0, lambda = [])
```

[8]:
```
Least Squares GLMNet Solution Path (1 solutions for 8 predictors in 7 passes):

     df   pct_dev

[1]   8  0.533414  0.0363067
```

De un conjunto de 100 modelos, el mejor es aquel que tiene como parámetro $\lambda = 0.0363067$

[9]:
```
# Predicciones usando el set de prueba
predictions_ridge = GLMNet.predict(ridge_model, test_X);
```

[10]:
```
# Como se trata de un algoritmo de regresión, los valores predichos son continuos;
# y dado que nuestro problema es categorico es necesario asignarle a cada valor su respectiva clase
# assign_class es una función que obtiene la diferencia entre el valor predicho y el valor real, de manera que se elige el que tenga menor diferencia
assign_class(predictedvalue) = [0,1][argmin(abs.(predictedvalue .- [0,1]))]
```

[10]:
```
assign_class (generic function with 1 method)
```

[11]:
```
# Predicciones ya clasificadas
yhat = vec(assign_class.(predictions_ridge))
# Precisión
acc_ridge = accuracy(yhat, test_y)
println("Precisión: $acc_ridge")
confusion_matrix(yhat, test_y)
```

```
Precisión: 0.96
```

[11]:

	Ground Truth	
Predicted	0	1
0	162	4
1	11	198

1.4.2 LASSO

Lasso (*Least Absolute Shrinkage and Selection Operator*) es un modelo lineal que penaliza el vector de coeficientes añadiendo su norma L1 (basada en la distancia Manhattan) a la función de coste:

$$RSS = \sum_{i=1}^{n}(y_i - f(x_i))^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

La regularización L1 funciona reduciendo los coeficientes a cero, eliminando esencialmente esas variables independientes del modelo. Tanto la regresión Lasso como la regresión Ridge reducen la complejidad del modelo, aunque por medios diferentes. La regresión Lasso reduce el número de variables independientes que afectan a la salida. La regresión Ridge reduce el peso que cada variable independiente tiene en la salida.

[12]:
```
lasso_models = glmnet(train_X, train_y)
cv_lm = glmnetcv(train_X, train_y)
```

[12]:
```
Least Squares GLMNet Cross Validation
67 models for 8 predictors in 10 folds
Best   0.007 (mean loss 0.119, std 0.003)
```

[13]:
```
# lambda y modelo elegido
  = lasso_models.lambda[argmin(cv_lm.meanloss)]
lasso_model = glmnet(train_X, train_y, lambda = [])
```

[13]:
```
Least Squares GLMNet Solution Path (1 solutions for 8 predictors in 8 passes):

     df    pct_dev

[1]   6  0.534248  0.00729486
```

De un conjunto de 67 modelos, el mejor es aquel que tiene como parámetro $\lambda = 0.0080061$, pero a diferencia de Ridge, este modelo **sólo tiene 6 variables predictoras**.

[14]:
```
# Predicciones usando el set de prueba
predictions_lasso = GLMNet.predict(lasso_model, test_X)
# Predicciones ya clasificadas
yhat = vec(assign_class.(predictions_lasso))

acc_lasso = accuracy(yhat, test_y)
println("Precisión: $acc_lasso")
# Matriz de confusión
confusion_matrix(yhat, test_y)
```

```
Precisión: 0.976
```

[14]:

	Ground Truth	
Predicted	0	1
0	165	1
1	8	201

1.4.3 Elastic Net

Elastic Net es un modelo de regresión lineal que normaliza el vector de coeficientes con las normas L1 y L2. Mientras que la regresión Ridge obtiene su parámetro de regularización a partir de la suma de errores al cuadrado y Lasso obtiene el suyo a partir de la suma del valor absoluto de los

errores, Elastic Net incorpora ambos parámetros de regularización en la función de coste RSS.

$$RSS = \sum_{i=1}^{n}(y_i - f(x_i))^2 + \alpha \left(\lambda \sum_{j=1}^{p} \beta_j^2 + (1-\lambda) \sum_{j=1}^{p} |\beta_j| \right)$$

[15]:
```
EN_models = glmnet(train_X, train_y)
cv_ENm = glmnetcv(train_X, train_y)
```

[15]:
```
Least Squares GLMNet Cross Validation
67 models for 8 predictors in 10 folds
Best   0.008 (mean loss 0.119, std 0.004)
```

[16]:
```
# lambda y modelo elegido
  = EN_models.lambda[argmin(cv_ENm.meanloss)]
EN_model = glmnet(train_X, train_y, lambda = [ ])
```

[16]:
```
Least Squares GLMNet Solution Path (1 solutions for 8 predictors in 8 passes):

     df    pct_dev

[1]   6  0.533975  0.0080061
```

[17]:
```
# Predicciones usando el set de prueba
predictions_EN = GLMNet.predict(EN_model, test_X)
# Predicciones ya clasificadas
yhat = vec(assign_class.(predictions_EN))

acc_EN = accuracy(yhat, test_y)
println("Precisión: $acc_EN")
# Matriz de confusión
confusion_matrix(yhat, test_y)
```

```
Precisión: 0.9733333333333334
```

[17]:

	Ground Truth	
Predicted	0	1
0	164	1
1	9	201

1.4.4 Decision Tree

Un árbol de decisión, tal y como su nombre sugiere, crea, a partir de los datos de entrenamiento, una estructura en forma de árbol en la que, en cada nodo, va dividiendo los datos de forma que los bloques resultantes de cada división sean más "puros" que el bloque original.

[18]:
```
# Definición de modelo y entrenamiento
DT_model = DecisionTreeClassifier(max_depth=3)
DecisionTree.fit!(DT_model, train_X, train_y)
```

[18]:
```
DecisionTreeClassifier
max_depth:                3
min_samples_leaf:         1
min_samples_split:        2
min_purity_increase:      0.0
pruning_purity_threshold: 1.0
n_subfeatures:            0
classes:                  [0, 1]
root:                     Decision Tree
Leaves: 2
Depth:  1
```

El arbol generado es el más sencillo; solo se realiza la división directamente a los dos casos posibles.

[19]:
```
# Visualizar el árbol
print_tree(DT_model)

# Predecir con el modelo entrenado
yhat = DecisionTree.predict(DT_model, test_X)

acc_DT = accuracy(yhat, test_y)
println("Precisión: $acc_DT")
# Matriz de confusión
confusion_matrix(yhat, test_y)
```

```
Feature 8 < -0.0005 ?
  0 : 429/429
  1 : 446/446
Precisión: 1.0
```

[19]:

	Ground Truth	
Predicted	0	1
0	173	0
1	0	202

1.4.5 Random Forest

A partir de un conjunto de entrenamiento, este algoritmo genera un cierto número de árboles de decisión, siendo entrenado cada uno de ellos con un subconjunto aleatorio de muestras extraídas con reemplazo. Además, durante la división de cada nodo, en lugar de considerar todas las características predictivas para encontrar el mejor criterio de división, solo considera un subconjunto de ellas. Como consecuencia de la aleatoriedad introducida, el sesgo o error del modelo aumenta ligeramente pero, gracias a la combinación de los modelos, decrece la varianza compensando el efecto negativo mencionado, dando como resultado un mejor resultado.

[20]:
```
RF_model = RandomForestClassifier(n_trees = 5)
DecisionTree.fit!(RF_model, train_X, train_y)
```

[20]:
```
RandomForestClassifier
n_trees:             5
n_subfeatures:       -1
partial_sampling:    0.7
max_depth:           -1
min_samples_leaf:    1
min_samples_split:   2
min_purity_increase: 0.0
classes:             [0, 1]
ensemble:            Ensemble of Decision Trees
Trees:      5
Avg Leaves: 8.0
Avg Depth:  3.0
```

[21]:
```
# Predecir con el modelo entrenado
yhat = DecisionTree.predict(RF_model, test_X)

acc_RF = accuracy(yhat, test_y)
println("Precisión: $acc_RF")
# Matriz de confusión
confusion_matrix(yhat, test_y)
```

```
Precisión: 1.0
```

[21]:

	Ground Truth	
Predicted	0	1
0	173	0
1	0	202

1.4.6 Nearest Neighbors

El algoritmo **k-Nearest Neighbors** es un algoritmo de Machine Learning extremadamente simple: en clasificación, asigna una etiqueta de clase a un punto de datos basándose en las etiquetas de clase de los k vecinos más cercanos en el conjunto de entrenamiento. Los vecinos se seleccionan según la distancia euclidiana u otra medida de distancia, y el número k se especifica previamente. Este algoritmo se basa en la suposición de que los datos con etiquetas semejantes se encuentran cerca unos de otros.

[22]:
```
# Indice k-d tree
kdtree = KDTree(train_X')
```

[22]:
```
KDTree{StaticArraysCore.SVector{8, Float64}, Euclidean, Float64,
StaticArraysCore.SVector{8, Float64}}
  Number of points: 875
  Dimensions: 8
  Metric: Euclidean(0.0)
  Reordered: true
```

[23]:
```
# Encontrar los k = 5 vecinos más cercanos
# Para cada punto de test_X se obtienen los k vecinos cercanos (sus indices en␣
 ↪el array train_X) y su respectiva distancia
indices, distancias = knn(kdtree, test_X', 10, true)

# Predicción
# Sólo nos interesan los indices de los vecinos;
# para determinar la clase de un punto de test_X, se determina a que clase␣
 ↪mayormente pertenecen los k vecinos
# la función mode devuelve el número más frecuente de un conjunto
yhat = [mode(train_y[idx]) for idx in indices]

acc_kNN = accuracy(yhat, test_y)
println("Precisión: $acc_kNN")
# Matriz de confusión
confusion_matrix(yhat, test_y)
```

```
Precisión: 0.88
```

[23]:

	Ground Truth	
Predicted	0	1
0	150	22
1	23	180

1.4.7 Support Vector Machines (SVM)

Una *máquina de vector de soporte* (SVM) es un algoritmo que clasifica los datos al encontrar una línea o hiperplano óptimo que maximice la distancia entre cada clase en un espacio N-dimensional. Se denomina *margen* a la suma de las dos distancias que separan al hiperplano del punto más próximo de cada clase. Luego, la posición del hiperplano de máximo margen y, por lo tanto, el propio margen dependen de los puntos de cada clase más próximos a la frontera de decisión, puntos que se conocen como *Support Vectors* o *Vectores de Soporte*.

[24]:
```
SVM_model = svmtrain(train_X', train_y)
yhat, = svmpredict(SVM_model, test_X')

acc_SVM = accuracy(yhat, test_y)
println("Precisión: $acc_SVM")
# Matriz de confusión
confusion_matrix(yhat, test_y)
```

```
Precisión: 0.968
```

[24]:

	Ground Truth	
Predicted	0	1
0	168	7
1	5	195

1.5 Resumén de precisión

[26]:
```
method = ["Ridge", "LASSO", "Elastic Net", "Decision Tree", "Random Forest",␣
 ↪"kNN", "SVM"]
accuracies = [acc_ridge, acc_lasso, acc_EN, acc_DT, acc_RF, acc_kNN, acc_SVM]
hcat(method, accuracies)
```

[26]:
```
7×2 Matrix{Any}:
 "Ridge"          0.96
 "LASSO"          0.976
 "Elastic Net"    0.973333
 "Decision Tree"  1.0
 "Random Forest"  1.0
 "kNN"            0.88
 "SVM"            0.968
```

El mejor método de clasificación es **Decision Tree**, al tener una precisión del 100%. Si bien, Random Forest también logra clasificar perfectamente, Decision Tree es un algoritmo más simple. Por otro lado, kNN es el metodo con menor precisión.