APRENDIZAJE AUTOMÁTICO

UNIVERSIDAD
DE GRANADA

Práctica 3: Programación

Alba Casillas Rodríguez
76738108B
albacaro@correo.ugr.es

Índice

1 – Problema de clasificación

OPTICAL RECOGNITION OF HANDWRITTEN DIGITS

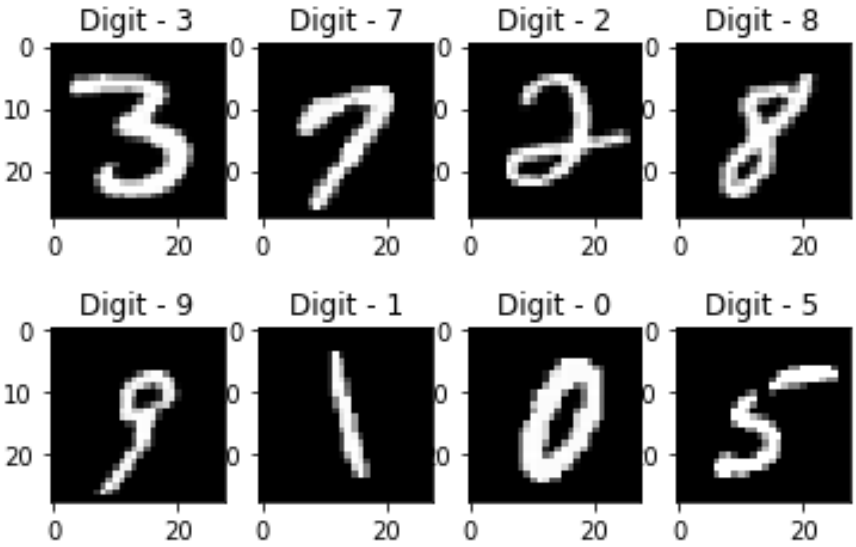
- Comprensión del problema

Trabajaremos con el conjunto de datos *Optical Recognition of Handwritten Digits*; se trata de un mapa de bits normalizados que contienen información sobre distintos dígitos manuscritos.

El conjunto de datos ya viene dividido en un conjunto training y en un conjunto test. En ambos conjuntos, tenemos 65 columnas, donde las 64 primeras son valores enteros en un rango de [0,16], siendo estos los atributos de cada una de las instancias (elementos X); y un valor entero entre [0,9] (elementos Y), que se corresponde con el código de la clase.

El problema trata de clasificar los ejemplos, donde f: X → Y asigna la clase correspondiente a cada muestra x de X.

Se trata de un problema de aprendizaje supervisado puesto que conocemos a priori las clases a las que pertenecen las instancias antes de ser clasificadas. Añadir que utiliza variables categóricas, es decir, conocemos el atributo que representan a pesar de ser valores numéricos.



- *Análisis de los datos*

El primer paso será realizar la lectura de los datos y la separación de sus características de las etiquetas.

Para ello, dispondremos de una función que lee los datos de un archivo. Esta función llamará a *"read_csv"* de pandas, con la que leeremos los datos guardándolos en un DataFrame, le añadiremos el parámetro *"header=None"* ya que, de lo contrario, al visualizar los datos nos daremos cuenta de que se han leído con una estructura incorrecta al considerar la primera fila como la cabecera.

```
def leer_datos(archivo, separador=None):
    datos ← pd.read_csv(archivo, separador, header=None)
    return datos
```

y otra que separa las características de las etiquetas:

```
def separar_datos(datos):
    valores ← datos.values  #Devuelve una representación numpy del DataFrame
    X ← valores[: , :-1] #Todas las filas y columnas (excepto la última)
    Y ← valores[:, -1] #La última columna de todas las filas

    return X, Y
```

De esta forma, nuestros datos serán visualizados de la siguiente manera:

```
Leemos los datos:
Mostramos los datos de entrenamiento:
      0   1   2   3   4   5   6   7   8  ...  56  57  58  59  60  61  62  63  64
0      0   1   6  15  12   1   0   0   0  ...   0   0   6  14   7   1   0   0   0
1      0   0  10  16   6   0   0   0   0  ...   0   0  10  16  15   3   0   0   0
2      0   0   8  15  16  13   0   0   0  ...   0   0   9  14   0   0   0   0   7
3      0   0   0   3  11  16   0   0   0  ...   0   0   0   1  15   2   0   0   4
4      0   0   5  14   4   0   0   0   0  ...   0   0   4  12  14   7   0   0   6
...   ..  ..  ..  ..  ..  ..  ..  ..  ..  ...  ..  ..  ..  ..  ..  ..  ..  ..  ..
3818   0   0   5  13  11   2   0   0   0  ...   0   0   8  13  15  10   1   0   9
3819   0   0   0   1  12   1   0   0   0  ...   0   0   0   4   9   0   0   0   4
3820   0   0   3  15   0   0   0   0   0  ...   0   0   4  14  16   9   0   0   6
3821   0   0   6  16   2   0   0   0   0  ...   0   0   5  16  16  16   5   0   6
3822   0   0   2  15  16  13   1   0   0  ...   0   0   4  14   1   0   0   0   7

[3823 rows x 65 columns]
```

Para empezar, resulta interesante tener una idea de cómo están distribuidas las clases del conjunto de entrenamiento. Por ello, paso a mostrar un gráfico (*realizado con Google Docs*) para ver de manera visual su distribución:

Como podemos observar, el número de muestras para cada clase está muy equilibrado, es decir, no encontramos ninguna clase que tenga un mayor numero de muestras que el resto.

Una vez dicho esto, el preprocesado de los datos es una de las fases más importantes de los problemas de machine learning, ya que nos permite obtener un conjunto de datos de mayor calidad con el que trabajar. Esto conlleva tener en cuenta varias situaciones como por ejemplo: asegurarnos de que no faltan datos, prescindir de variables irrelevantes o erróneas para evitar un aprendizaje erróneo u overfitting, reducir la complejidad del conjunto...

En mi caso, tras probar distintas técnicas de preprocesado me he decantado por realizar:

- ***Asegurar que no faltan datos***: aunque en la información proporcionada en la web sobre el conjunto de datos indica que no faltan datos, nunca está de más comprobarlo. Para ello, antes de separar las características de las clases, le he aplicado a los DataFrame de entrenamiento y test la función: **pandas.DataFrame.dropna()** . La cual se encarga de eliminar aquellas filas a las que les faltan datos.

- ***Eliminar datos sin/con poca variabilidad***: para ello, hago uso de a función **VarianceThreshold()** , la cual elimina las características cuya varianza no supera un límite (*threshold*) establecido (donde threshold=0.1). Tras su uso, llamamos a la función **fit_transform()**, para que los datos estén mejor ajustados y distribuidos.

- ***Eliminar outliers y datos irrelevantes***: es decir, datos que o bien tienen valores no validos (NaN) o que no son representativos porque se alejen demasiado de la media.

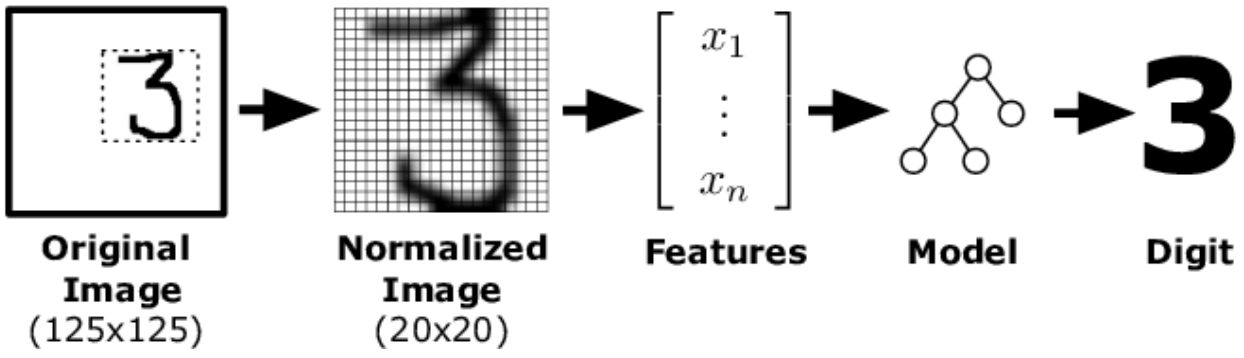
Para ello, usamos variables **LOF (Local Outlier Factor)** para una cantidad de contaminación del conjunto de datos del 0.1 y el atributo **negative_outlier_factor** , donde ordenamos los valores de los datos a sabiendas que que los datos atípicos tenderán a tener una puntuación mayor a -1, eliminándolos si los hay.

- ***Reducción de dimensionalidad***: reduciremos el número de dimensiones, quedándonos con aquellas que expliquen en mayor medida la varianza de los datos.

Utilizaremos la técnica **PCA (Principal Component Anlaysis)** que, atendiendo al libro *Learning From Data*, vemos que trabaja sobre la matriz de datos basándose en *SVD (Singular Value Descomposition).* Manteniendo una representatividad del 95% conseguimos mejorar un poco más los valores de los datos, aplicando una reducción de dimensionalidad de:

```
Dimensiones antes de aplicar PCA:  53
Dimensiones depsués de aplicar PCA:  29
```

Además, durante el desarrollo de la práctica, cabe destacar que probé a normalizar los datos con la función **Normalizer()** de *sklearn.preprocession* , sin embargo, los resultados empeoraron; por lo cual me decanté por no usar normalización. Tras investigar el por qué de este suceso, podría decir que el hecho de transformar los datos para que sigan la norma de la unidad (es decir, valores entre 0 y 1), causa una destrucción de la diversidad, y por ende, pérdida de información.



- *Selección de funciones*

Vamos a utilizar la clases de **funciones lineales**. No tiene sentido aplicar combinaciones no lineales a los datos puesto que hacer transformaciones como añadir más dimensiones pueden llevarnos a problemas como el overfitting.

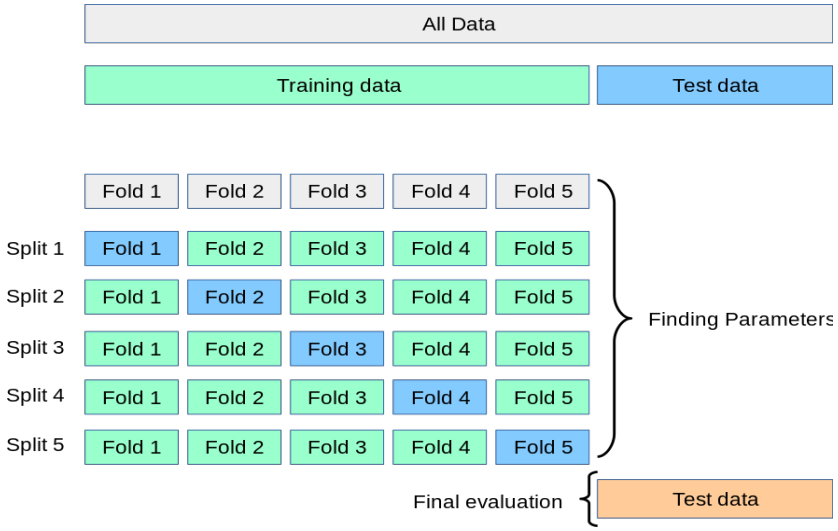
- *Fijar los conjuntos de training, validación y test.*

En nuestro problema, los conjuntos training y test ya se encontraban separados en los conjuntos training (*optdigits.tra*) y test (*optdigits.tes*).

Con el conjunto de entrenamiento (data_train) realizaremos **Cross-Validation (10-fold)**. Para ello, utilizaremos dos funciones:

- ***StratifiesKFold(n_splits=10, shuffle=True, random_state=1)*** , la cual nos proporciona los indices train y test para dividir los conjuntos.

- ***cross_val_score(model, x , y, scoring = 'accuracy', cv = cv)*** , función que se encarga de realizar la validación cruzada para cada modelo que indicamos en el parámetro "model" con los conjuntos de datos proporcionados.



La validación cruzada funciona de la siguiente manera: dividimos el conjunto de datos en 10 (ya que *n_splits = 10*) particiones. Se realizarán n_splits iteraciones, y en cada una se elegirá una partición como conjunto de prueba, siendo la unión de las demás particiones el conjunto de entrenamiento. De esta manera, se entrenará el modelo y obtendremos una estimación más precisa del mismo.

Este proceso lo he realizado para todos los modelos evaluados. Un pseudocódigo general será:

```
def cross_validation(X, Y, cv):
    means ← []
    deviations ← []

    model ← Modelo_A_Usar(Parámetros)
    scores ← cross_val_score(model, x , y, cv = cv)

    means ← scores.mean()
    deviations ← std(scores)

    return means, deviations
```

– *Necesidad de regularización*

La **regularización** son técnicas utilizadas para reducir el error al ajustar una función de manera apropiada en el conjunto de entrenamiento dado y evitar el sobreajuste.

Aparte de que el módulo que estamos utilizando, *scikit-learn*, alude en todos los modelos que utilizamos el uso de la regularización; es recomendable usarlo puesto que esta técnica reduce la varianza de los estimadores de los coeficientes, y el hecho de restringir el espacio nos proporcionará unas soluciones de mayor calidad.

Utilizaremos las dos técnicas de regularización más utilizadas: **Ridge** y **Lasso**, las cuales se incorporan al usar los modelos mediante el argumento *"penalty"* , el cual toma los valores:

- l1 → Regularización Lasso
- l2 → Regularización Ridge

<u>Regresión Ridge</u>: añade "*magnitud al cuadrado*" del coeficiente como término de penalización en la función de pérdida:

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

Cost function

- As $\lambda \to 0$, $\hat{\beta}_{ridge} \to \hat{\beta}_{OLS}$;
- As $\lambda \to \infty$, $\hat{\beta}_{ridge} \to 0$.

El parámetro λ es el que regula la penalización. Cuando $\lambda \to 0$ (tiende a 0), la regresión Ridge se vuelve muy similar a los mínimos cuadrados; sin embargo, cuando λ crece, los coeficientes tienden a cero sin llegar a ser cero nunca ; por lo que cuanto más grande es su valor, más fuerte será el tamaño de los coeficientes penalizados.

Esta técnica suele funcionar bien si hay muchos parámetros grandes de aproximadamente el mismo valor, es decir, cuando la mayoría de los predictores tiene impacto en la respuesta.

Regresión Lasso: añade "magnitud de valor absoluto" del coeficiente como término de penalización en la función de pérdida.

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Cost function

En este caso, cuanto mayor sea λ, más coeficientes tendrán valor cero.

Esta técnica tiende a funcionar bien is hay un pequeño número de parámetros significativos y los otros tienen valores cercanos a cero (es decir, cuando solo unos pocos predictores influyen en la respuesta).

Para los modelos de Regresión Logística y SVM, donde utilizaremos la regresión Ridge (usada por defecto), además, le añadiremos el parámetro "C", el cual sirve para regular la relevancia de λ.

La relación entre C y λ es: $C = 1/\lambda$. Por tanto, cuanto menor sea el valor de C, más fuerte será la regularización.

En el resto de modelos no usaremos el parámetro C ya que, consultando la documentación de los correspondientes algoritmos, estos no tienen implementados este parámetro.

– Identificar los modelos LINEALES a usar

Para este apartado, no sólo hablaré de los modelos lineales, sino que, para cada uno de ellos , hablaré de su ***tipo de regularización, métrica de ajuste, y técnica de ajuste,*** ya que son apartados necesarios en la

descripción de la práctica y de esta manera podré explicarlos con más profundidad.

1 – Regresión Logística Multinomial:

logistic = LogisticRegression(multi_class = 'multinomial', C = c, random_state = 1)

Elegimos esta variación de la Regresión Logística ya que es la generalización para problemas multiclase. La regresión Logística Multinomial entrena un único modelo, al contrario de la Regresión Logística cuando utiliza el criterio **ORV** (*One vs Rest*), donde se entrenarían múltiples modelos.

Para indicar que realizaremos Regresión Logística Multinomial, modificaremos el argumento *"multi_class"* , el cual usa *"auto"* por defecto, nosotros lo cambiaremos por *"multinomial"*. Al marcar esta opción, además, utilizaremos una métrica de ajuste de **Cross-Entropy**, con la que minimizamos el error que se comete al intentar predecir que un elemento pertenece a una clase determinada.

En cuanto a la técnica de ajuste, es decir, cómo se actualizan los pesos en nuestro modelo, existe un parámetro llamado *"class_weight"*, el cual dejaremos con su valor por defecto(*None*), que da por defecto peso = 1 a cada clase. Estos pesos serán multiplicados por un valor *"sample_weight"*, si es especificado al usar la función **fit()**, aunque lo omitiremos en nuestro caso.

Cómo ya se ha comentado antes, el tipo de regularización será una **regularización Ridge** por defecto la cual es ajustada por el **parámetro C.** Al usar este tipo de regularización, a su vez, nos permite utilizar un solver, es decir, el algoritmo utilizado para la optimización, *'lbfgs'* ; método de optimización quasi-Newton que haciendo un uso limitado de la memoria permite obtener el mínimo de una función, solo conociendo esta y su gradiente.

Otros parámetros de interés que usa este modelo y sus valores por defecto son:

- **dual = False.** Es preferible mantener falso este parámetro cuando el número de muestras es mayor que el de características.
- **tol = 0.0001** , el cual indica el criterio de parada.
- **max_iter = 100**, indica el número de iteraciones para converger.

2 – Perceptron (PLA):

logistic = Perceptron(penalty='l2', n_jobs = -1, random_state = 1)

El Perceptron es un algoritmo de clasificación. Por defecto, no requiere tasa de aprendizaje y actualiza el modelo sólo en caso de errores, lo que hace que sea más rápido de entrenar en comparación con otros algoritmos como el SGD. Este algoritmo utiliza un conjunto de funciones para aprender las entradas y determinar si pertenecen a alguna clase o no.

Utilizará una métrica de ajuste de **"loss 0-1"**, es decir, por cada predicción errónea incurrirá en una penalización de 1, y las predicciones correctas no influirán en la penalización.

Aunque no es necesario utilizar regularización, le aplicaré ambos tipos de regresiones (Regresión Ridge y Lasso), a través del parámetro *"penalty"*, en busca de una mejor solución.

Por defecto, todas las clases tienen peso 1. A parte de esto, el Perceptron tiene otros parámetros cuyos valores por defecto son:

- **tol = 0.01** , el cual indica el criterio de parada.
- **max_iter = 1000**, indica el número de iteraciones para converger.
- **early-stopping = False**, es decir, el entrenamiento no finalizará en cuando la validación no mejore.

Es importante destacar el hecho de que este algoritmo, al usarse en un problema de clasificación multiclase, utiliza el criterio **OVA** (*One vs All*). Por ende, existe un parámetro llamado *"n_jobs"* , que indica cuantas CPUs utilizan para el OVA. Por defecto utiliza 1 CPU, pero cambiar este parámetro a -1 hará que utilicemos todas las CPUs en paralelo.

3 – Gradiente Descendente Estocástico (SGDClassifier)

logistic = SGDClassifier(penalty='l2', loss='squared_hinge', n_jobs = -1, random_state = 1)

Clasificador lineal con entrenamiento SGD. Este modelo implementa modelos lineales regularizados con un aprendizaje mediante gradiente descendente estocástico, donde el gradiente de la pérdida se estima a la vez en cada muestra y el modelo se actualiza mediante la tasa de aprendizaje.

La regularización es una penalización agregada a la función de pérdida, la cual reduce los parámetros del modelo mediante Regresión Ridge o Lasso. Utilizaré los dos tipos modificando el parámetro *"penalty".*

La métrica de ajuste utilizada es **'squared_hinge'**, una variación de Hinge (la cual proporciona un SVM linear) donde se penaliza cuadráticamente , puesto que mejora un poco los errores obtenidos. Para este algoritmo también sigue un criterio **OVA** (*One vs All*), con el mismo funcionamiento en el Perceptron.

Otros parámetros y sus valores por defecto de interés son:

- **max_iter = 1000**, máximo de iteraciones.
- **early-stopping = False**, es decir, el entrenamiento no finalizará en cuando la validación no mejore.
- **class_weight = None**, es decir, el peso de casa clase es de 1.
- **learning_rate** = la tasa de aprendizaje, la cual por defecto tiene la opción "optimal" de valor: eta = 1.0 / (alpha * (t + t0)), donde t0 es elegido mediante una heurística (heurística de Leon Bottou).

4 – Support Vector Machine con Kernel lineal (SVM)

logistic = LinearSVC(C=c, random_state = 1, loss = 'hinge')

Resulta de interés observar cómo funciona un modelo SVM, ya que son bastante eficientes a la hora de encontrar la mejor separación entre clases; minimizando la suma de las violaciones del margen establecido entre clases (margen que se intenta maximizar).

Por defecto, se utilizará *Regularización Ridge*, la cual, de la misma manera que hicimos con la Regresión Logística, regularemos mediante el **parámetro C**. A mayor sea el valor de C, mayor será el ajuste de los datos, limitando el número de puntos dentro del margen, aunque haciendo que la amplitud sea más pequeña (causando una peor generalización).

La métrica de ajuste ideal para este modelo sera **Hinge**, ya que mide tanto si ha habido una buena predicción del resultado como si se ha dejado suficiente margen.

Por defecto, se aplicarán los valores:

- **dual = True.**
- **tol = 0.0001** , el cual inidca el criterio de parada.
- **max_iter = 1000**, indica el número de iteraciones para converger.
- **multi_class = OVR** , One vs Rest.

– Selección del mejor modelo

Para elegir el mejor modelo, aplicaremos todo lo anteriormente explicado para cada uno de los modelos indicados en el apartado anterior.

Por ello, para cada modelo haremos *cross-validation* con el conjunto de entrenamiento, e iremos probando un conjunto de hiperparámetros (como los valores de C: 0'01; 0'1 y 1'0 en Regresión Logística y SVM; o el tipo de regularización).

Como nos interesa saber si estamos acertando en la clasificación, evaluaremos los modelos basándonos en la **precisión** (*accuracy*) es decir, el número total de predicciones correctas dividido entre el número total de predicciones (nuestro valor *media*). Será mejor el modelo que mayor precisión tenga.

Los resultados son los siguientes:

Modelo	***Media***	***Desviación***
MLR c=0.01	***0.970396***	***0.009131***
MLR c=0.1	***0.967772***	***0.007111***
MLR c=1.0	***0.960171***	***0.005931***
PLA l1	***0.917451***	***0.015306***
PLA l2	***0.907507***	***0.012966***
SGD l1	***0.957813***	***0.008627***
SGD l2	***0.959645***	***0.009740***
SVMC c=0.01	***0.947074***	***0.008934***
SVMC c=0.1	***0.958337***	***0.006651***
SVMC c=1.0	***0.959913***	***0.010325***

Observando los resultados, vemos que la **Regresión Logística Multinomial con C=0.01** es la que obtiene **una mayor precisión**.

Sin embargo, es el *SVM con Kernel lineal con C = 0.1* es quien tiene menor desviación típica en los resultados; a pesar de que su valor de la media no mejore con respecto a ningún caso de la Regresión Logística.

Para la *Regresión Logística Multinomial*, el valor de la media empeora al aumentar el valor de C, por lo que podemos decir que el usar una mayor regularización proporciona un mejor resultado; lo que nos lleva a pensar que un valor de C= 10.0 con total probabilidad nos hubiese proporcionado un resultado aún peor.

Si miramos el *Perceptron*, vemos que sus resultados tanto en la media como en la desviación son los peores en comparación con el resto de los valores, de manera que concluimos que no es un modelo que consiga adaptarse del todo bien a los datos de nuestro problema. En cuanto al *SGDClassifier*, del cual nos decantamos por el que usa la regularización Lasso, puesto que nos proporciona valores ligeramente mejores, vemos que puede adaptarse bastante bien al problema y, aunque no haya conseguido alcanzar la calidad de los valores de la *Regresión Logística Multinomial,* se asemejan a los del *modelo SVM*; siendo ambos modelos bastante buenos también.

Tras ver estos resultados, entrenaremos el **modelo elegido**: Regresión Logística Multinomial con el hiperparámetro C = 0.01 con todo el conjunto de entrenamiento.

Como ya se ha dicho anteriormente, la **precisión** es el elemento clave para evaluar nuestros modelos. Sin embargo, es inapropiada para problemas de clasificación desequilibrados. Este no es nuestro caso, ya que durante la lectura de datos comprobamos que las clases estuviesen bien distribuidas; aún así, podemos aprovechar para observar cómo nuestros modelos se comportan ante otro tipo de métricas de evaluación que también nos pueden guiar sobre qué tan bueno es un modelo:

- **Precision**: cuantifica el número de predicciones de clase positivas que realmente pertenecen a una clase positiva.

- **Recall**: cuantifica el número de predicciones de clase positivas hechas de todos los ejemplos positivos del conjunto de datos.

- **F-Measure**: es la media armónica entre precision y recall.

Estos valores lo obtenemos a partir de la función **classification_report** de *sklearn.metrics*. Para conocer esta información, se debe entrenar el *modelo* con el conjunto de entrenamiento y llamar a la función con el *conjunto de prueba Y_test*.

Cabe añadir que el parámetro **support** es el número de ocurrencias de cada clase en el conjunto de datos; por lo que además veremos de forma numérica que nuestras clases están balanceadas.

Nuestros resultados son:

- <u>Regresión Logística Multinomial</u>:

- Matriz de resultados:

	precision	recall	f1-score	support
0	1.00	0.99	0.99	178
1	0.92	0.95	0.94	182
2	0.97	0.98	0.97	177
3	1.00	0.91	0.95	183
4	0.96	0.97	0.96	181
5	0.91	0.97	0.94	182
6	0.99	0.98	0.98	181
7	0.98	0.91	0.94	179
8	0.92	0.90	0.91	174
9	0.88	0.94	0.91	180
accuracy			0.95	1797
macro avg	0.95	0.95	0.95	1797
weighted avg	0.95	0.95	0.95	1797

<u>LRM C = 0.01</u>

- Matriz de resultados:

	precision	recall	f1-score	support
0	0.99	0.98	0.98	178
1	0.93	0.97	0.95	182
2	0.96	0.97	0.96	177
3	0.98	0.91	0.94	183
4	0.99	0.97	0.98	181
5	0.91	0.96	0.93	182
6	0.96	0.98	0.97	181
7	0.97	0.92	0.94	179
8	0.91	0.91	0.91	174
9	0.89	0.93	0.91	180
accuracy			0.95	1797
macro avg	0.95	0.95	0.95	1797
weighted avg	0.95	0.95	0.95	1797

<u>LRM C = 0.1</u>

- Matriz de resultados:

	precision	recall	f1-score	support
0	0.99	0.96	0.97	178
1	0.94	0.98	0.96	182
2	0.96	0.97	0.96	177
3	0.96	0.90	0.93	183
4	0.99	0.97	0.98	181
5	0.89	0.93	0.91	182
6	0.96	0.97	0.97	181
7	0.93	0.90	0.91	179
8	0.88	0.84	0.86	174
9	0.87	0.93	0.90	180
accuracy			0.94	1797
macro avg	0.94	0.94	0.94	1797
weighted avg	0.94	0.94	0.94	1797

<u>LRM C = 1.0</u>

Contemplamos que no difieren mucho los valores entre las tres imágenes aunque, corroborando los resultados de la gráfica anterior, es la Regresión Logística Multinomial con C = 0.01 quien obtiene resultados algo mejores (podemos observar esto en la columna “precision” donde para la clase de dígitos 0 y 3 tiene una tasa del 100% de aciertos)

- Perceptron :

- Matriz de resultados:

	precision	recall	f1-score	support
0	0.88	0.99	0.93	178
1	0.75	0.84	0.79	182
2	0.96	0.90	0.93	177
3	0.76	0.81	0.78	183
4	0.96	0.82	0.88	181
5	0.84	0.90	0.87	182
6	0.90	0.98	0.94	181
7	0.86	0.93	0.89	179
8	0.85	0.68	0.75	174
9	0.93	0.78	0.85	180
accuracy			0.86	1797
macro avg	0.87	0.86	0.86	1797
weighted avg	0.87	0.86	0.86	1797

PLA + LASSO

- Matriz de resultados:

	precision	recall	f1-score	support
0	0.93	0.98	0.95	178
1	0.77	0.87	0.82	182
2	0.97	0.97	0.97	177
3	0.95	0.90	0.92	183
4	0.96	0.76	0.85	181
5	0.89	0.98	0.93	182
6	0.97	0.92	0.94	181
7	0.92	0.89	0.91	179
8	0.83	0.81	0.82	174
9	0.83	0.90	0.86	180
accuracy			0.90	1797
macro avg	0.90	0.90	0.90	1797
weighted avg	0.90	0.90	0.90	1797

PLA+ RIDGE

No es sorpresa ver que los valores son peores en comparación a los del modelo anterior; donde nos damos cuenta que el Perceptron sobre todo tiene problemas al acertar con números como el 1, 3, y 8; donde en términos de precisión tiene un porcentaje en torno al 70% de predicciones positivas. Entre ambos modelos, el PLA que utiliza regularización Ridge se adapta algo mejor.

- Gradiente Descendente Estocástico.

- Matriz de resultados:

	precision	recall	f1-score	support
0	0.96	0.98	0.97	178
1	0.90	0.92	0.91	182
2	0.96	0.97	0.96	177
3	0.96	0.89	0.93	183
4	0.94	0.97	0.95	181
5	0.90	0.97	0.93	182
6	0.98	0.97	0.97	181
7	0.94	0.92	0.93	179
8	0.92	0.86	0.89	174
9	0.88	0.89	0.88	180
accuracy			0.93	1797
macro avg	0.93	0.93	0.93	1797
weighted avg	0.93	0.93	0.93	1797

SGD + LASSO

- Matriz de resultados:

	precision	recall	f1-score	support
0	0.98	0.99	0.98	178
1	0.92	0.92	0.92	182
2	0.96	0.98	0.97	177
3	0.98	0.90	0.94	183
4	0.95	0.97	0.96	181
5	0.91	0.97	0.94	182
6	0.96	0.97	0.96	181
7	0.95	0.94	0.94	179
8	0.92	0.86	0.89	174
9	0.89	0.91	0.90	180
accuracy			0.94	1797
macro avg	0.94	0.94	0.94	1797
weighted avg	0.94	0.94	0.94	1797

SGD + RIDGE

Para este modelo, se obtienen valores prácticamente iguales con ambos tipos de regularización, con lo que vemos que ambos modelos se adaptan bastante bien a los datos, a pesar de no ser los mejores.

- Support Vector Machine con Kernel Lineal

- Matriz de resultados:

	precision	recall	f1-score	support
0	0.99	0.99	0.99	178
1	0.87	0.86	0.87	182
2	0.91	0.97	0.94	177
3	0.94	0.93	0.94	183
4	0.97	0.97	0.97	181
5	0.90	0.98	0.94	182
6	0.96	0.96	0.96	181
7	0.98	0.92	0.95	179
8	0.92	0.80	0.86	174
9	0.87	0.92	0.89	180
accuracy			0.93	1797
macro avg	0.93	0.93	0.93	1797
weighted avg	0.93	0.93	0.93	1797

SVM C = 0.01

- Matriz de resultados:

	precision	recall	f1-score	support
0	0.99	0.98	0.99	178
1	0.90	0.93	0.91	182
2	0.97	0.97	0.97	177
3	0.95	0.92	0.93	183
4	0.96	0.98	0.97	181
5	0.91	0.99	0.95	182
6	0.97	0.97	0.97	181
7	0.98	0.93	0.96	179
8	0.92	0.83	0.87	174
9	0.87	0.92	0.90	180
accuracy			0.94	1797
macro avg	0.94	0.94	0.94	1797
weighted avg	0.94	0.94	0.94	1797

SVM C = 0.1

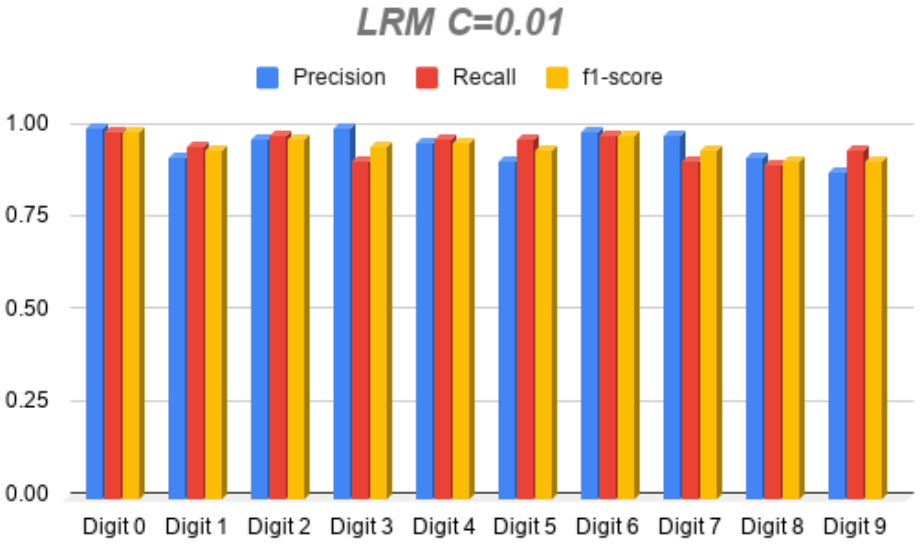
- Matriz de resultados:

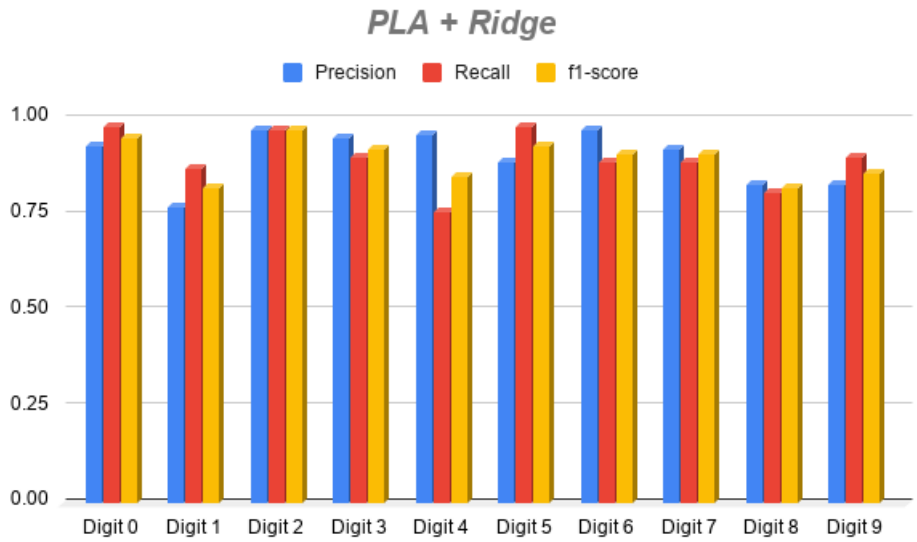
	precision	recall	f1-score	support
0	0.99	0.98	0.99	178
1	0.89	0.93	0.91	182
2	0.98	0.98	0.98	177
3	0.98	0.91	0.94	183
4	0.94	0.97	0.96	181
5	0.90	0.99	0.94	182
6	0.97	0.98	0.98	181
7	0.96	0.92	0.94	179
8	0.93	0.87	0.90	174
9	0.87	0.89	0.88	180
accuracy			0.94	1797
macro avg	0.94	0.94	0.94	1797
weighted avg	0.94	0.94	0.94	1797

SVM C = 1.0

En este caso, aunque ya se sabía mediante los valores de la tabla, los SVM con C 0’1 y 1’0 son los dos que mejor se adaptan (aunque el otro también lo haga bastante bien). Atendiendo a la tabla, al no ser capaz de elegir al mejor entre esos dos, elegiré el de hiperparámetro C = 0.1 por ser, como ya vimos, aquel con desviación más baja.

Dicho esto, insertaré para el mejor de los modelos y uno de los peores dos gráficos de barras (generados mediante *google docs*) para tener una imagen visual de la diferencia entre los resultados de ambos modelos:





Viendo las gráficas, de dónde ya no se puede sacar tanta información a parte de la ya dicha; llama la atención que, a pesar de ser modelos diferentes, podemos observar una cierta similitud: los modelos entrenados suelen fallar en su mayoría en los mismos dígitos (como podemos ver en el caso de los dígitos 1 y 8).

Cabe desatacar positivamente el hecho de que para un buen modelo como es el caso del LRM, las métricas (precision, recall y f1-score) tienen, en general, valores muy similares en una misma clase; al contrario del PLA, donde suele haber algo más de variabilidad entre los 3 valores para cada clase.

Como conclusión, es obvio que ya tenemos elegido y ajustado nuestro mejor modelo: Regresión Logística con C = 0.01.

– Estimación de Eout

En el mundo real, no podemos calcular un Eout exacto, puesto que no conocemos la totalidad de las instancias posibles; es por ello que estimaremos Eout mediante Etest.

Sabiendo que Etest es una buena cota para Eout, podremos asegurar con certeza que, ante datos futuros, obtendríamos un error parecido al cometido con los datos de test.

Dicho esto, nuestro Etest será el error cometido por nuestro modelo con los datos de prueba. Para estimar Eout, usaremos la siguiente desigualdad:

$$E_{out}(g) <= E_{test}(g) + \sqrt{(\frac{1}{2N} ln \frac{2}{\alpha})}$$

donde α = 0.05 , es decir, una confianza del 95 %

```
def cota_Etest(Etest):
    Eout ← sqrt(1/2*X_test.size)*log(2/□))
    return Etest + Eout
```

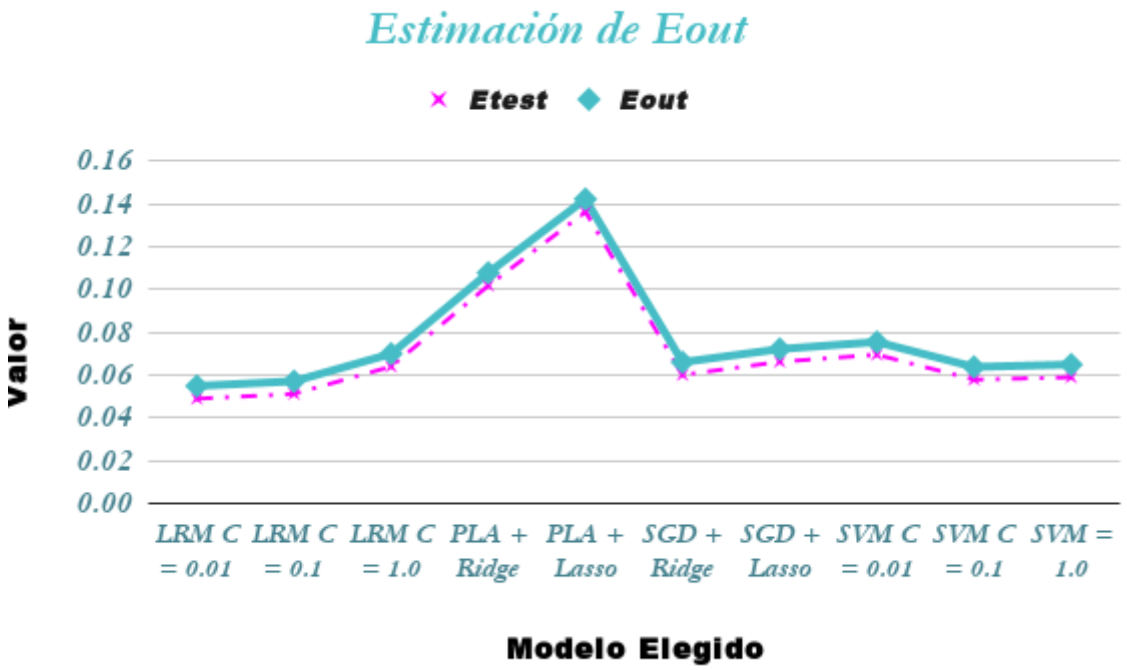
Pseudocódigo del proceso de ajuste y entrenamiento del modelo elegido:

```
logistic = LogisticRegression(multi_class = 'multinomial', C = 0'01 ,
random_state=1).fit(X_train, Y_train)
predicted = logistic.predict(X_test)
Etest = 1 – accuracy_score(Y_test, predicted)
Eout = cota_Etest( 1 – accuracy_score(Y_test, predicted))
```

- Etest: 0.0489705063995548
- Estimacion del out: 0.05491971499057739

Como podemos observar, nuestro mejor modelo ha acertado al **predecir el 95% de las etiquetas**. Este cálculo es algo peor que el que obtuvimos al entrenar el modelo con validación cruzada; aunque esto no quiere decir que hayamos fallado entrenando al modelo; puesto que un **5% de error** en un problema de clasificación de muchas dimensiones y con 10 clases es muy buen resultado.

Aunque estos sean los valores para el mejor modelo, insertaré un gráfico con el Eout estimado mediante el Etest de todos los modelos trabajados; y una tabla con los valores exactos para las configuraciones de los mejores modelos discutidos en el apartado anterior:



Modelo	Etest	Eout
LRM C = 0.01	**0.048970**	**0.054919**
PLA + Ridge	**0.101836**	**0.107785**
SGD + Ridge	**0.060100**	**0.066049**
SVM C = 0.1	**0.057874**	**0.063823**

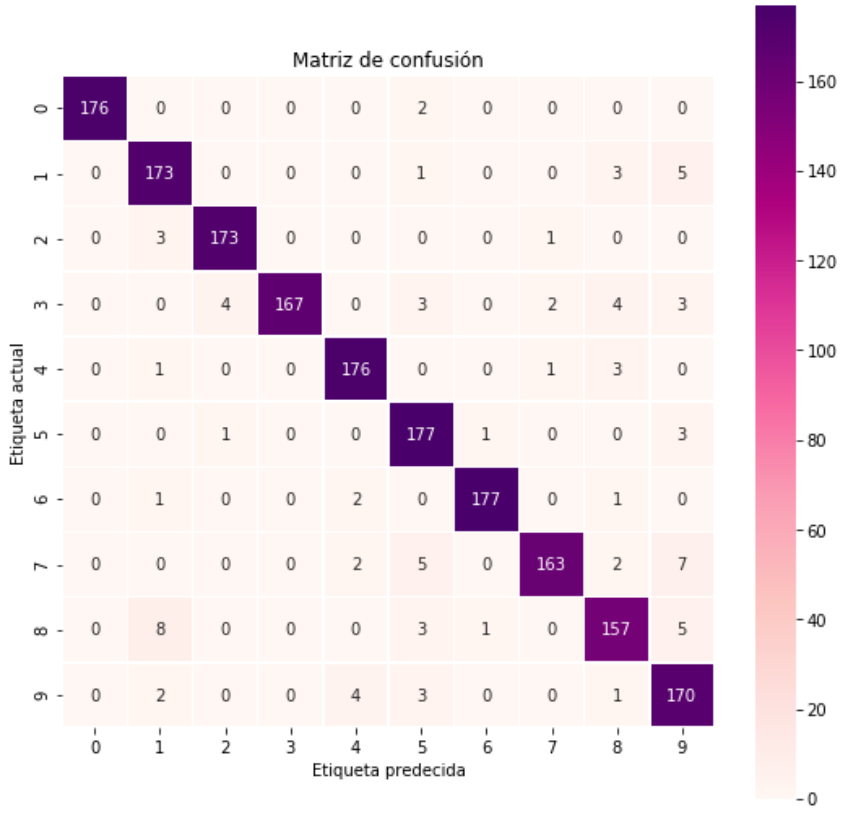
Como se puede observar, el calculo del Estet con el conjunto de prueba y la estimación del Eout sigue, en términos generales, el mismo comportamiento que en los resultados explicados de la validación cruzada para cada modelo.

A su vez, y como era de esperar, la estimación del Eout tiene valor un *0'005* más elevado que el Etest obtenido; puesto que al estimarlo le sumamos al valor de Etest la expresión definida anteriormente, la cual comprobamos que no es demasiado significativa en el resultado final.

Esto significa que la muestra de entrenamiento representa muy bien a la mayoría de los modelos y que estos han sido entrenados. Por eso, a excepción del PLA que no ha podido tener una buena adaptación, los modelos han aprendido el patrón que hay entre los datos de entrada y los resultados; de manera que nos asegura un buen funcionamiento ante nuevos datos.

Para hacer algo más completo el análisis, vamos a ver la matriz de confusión para nuestro mejor modelo.

La **matriz de confusión** es una herramienta que permite la visualización del desempeño de un modelo de aprendizaje automático. El ejer horizontal (las columnas) representa los valores que el modelo ha predicho, mientras que el eje vertical se corresponde a los valores reales de las etiquetas.



Como no hemos obtenido una *matriz diagonal* (matriz cuadrada en la que todos los valores son nulos menos los de la diagonal principal) sabemos que nuestro modelo no ha obtenido una precisión del 100%. La mayoría de los errores se han producido al predecir el dígito ‘8’ , donde ha acertado 157 veces pero lo ha confundido 8 veces con el dígito ‘1’ y 5 veces con el dígito ‘9’; al igual que al intentar predecir el dígito ‘7’ , ha acertado 163 veces frente a 7 veces que lo ha confundido con un ‘9’. Es-

tos fallos podrían deberse a la pérdida de información producida ante alguna fase del preprocesado como reducir la dimensionalidad; con lo que aprendemos a que hay que utilizar estas técnicas con mucho cuidado, puesto que podemos perder características a priori, irrelevantes; pero que al final resulten ser decisivas en algunos casos. A pesar de ello, es evidente (y a vista de los errores obtenidos) que la gran mayoría de las veces se ha producido un acierto en la predicción.

Pseudocódigo de la matriz de confusión:

```
matrix = confussion_matrix(Y_test, predicted)
```

– *Si fuésemos una empresa...*

En un caso real, tras haber realizado todo el proceso de selección de modelos (mediante el uso de técnicas como cross-validation) y las evaluaciones de los mismos, con el objetivo de estimar un error de generalización que explique el rendimiento de los datos no vistos, no sólo bastaría con observar las diferentes estimaciones de los errores fuera de la muestra para estimar el mejor modelo.

Para proporcionar un juicio adecuado, además de comparar qué tan buenos son los errores obtenidos, habría que realizar un análisis comparativo más exhaustivo con técnicas que he mostrado anteriormente como *"classification_report"* o *"confussion_matrix"*; ya que a la hora de tratarse en de un problema real, debemos asegurarnos que nuestro modelo es realmente bueno.

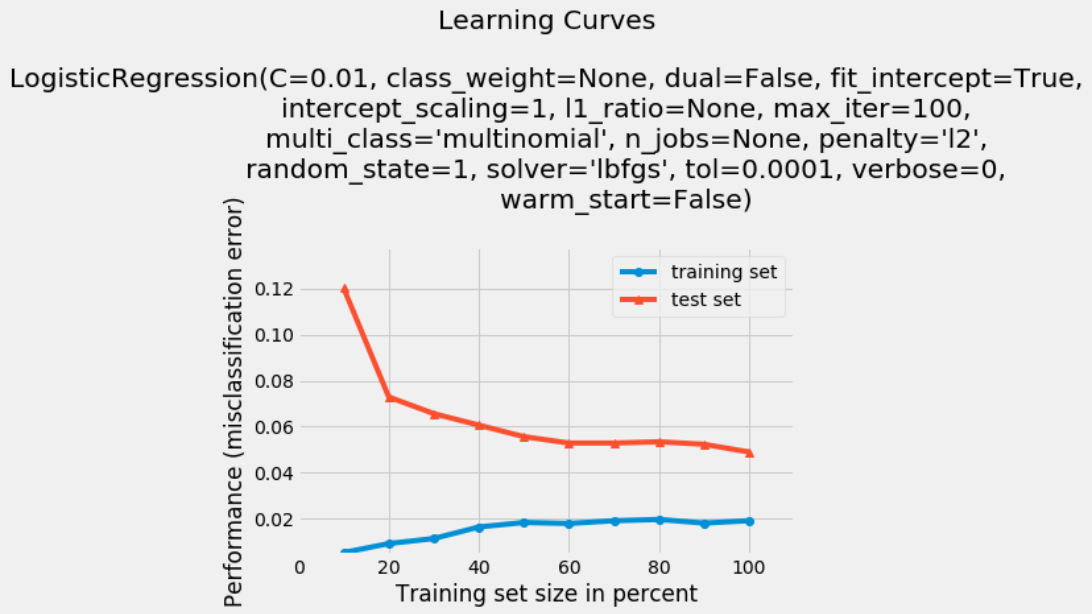
Habiendo utilizado dichas técnicas en nuestro problema, puedo asegurar que el mejor modelo es el anteriormente dicho: **Regresión Logística con una regularización Ridge y C = 0.01**. Y diría que presenta un **error del 5%** , lo cual es bastante bueno, ya que al haber sido entrenado usando dígitos escritos por personas con distinta caligrafía, es normal que existe un cierto margen de error.

Además, acudiendo a la documentación proporcionada por del Data Set, vemos que este problema ha sido estudiado con un modelo no lineal **K-NN** (con la distancia Euclídea como métrica), con el cual se han obtenido resultados de mucha precisión; sin embargo, nuestro mejor modelo no difiere mucho de estos resultados; por lo que podemos probar que el modelo elegido ganador , de entre los estudiados en esta práctica, es el más adecuado al problema.

Aún así, considero que no deberíamos quedarnos únicamente con este estudio, ya que en un caso real podemos probar a ampliar el número de instancias del conjunto de prueba para ver si el error aumenta o disminuye ; o aplicar otras técnicas como mostrar la curva de aprendizaje.

A continuación y como conclusión a este problema, mostraré la **curva de aprendizaje** obtenida para el modelo elegido.

NOTA: la gráfica obtenida está comentada en el código ya que para implementarla se necesita tener instalado el módulo *mlxtend*; y el ejecutarlo sin el módulo nos llevaría a un fallo de ejecución de la práctica.



Las curvas de aprendizaje permiten ilustrar fácilmente el concepto de *sesgo* (suposiciones erróneas) y *varianza* (cuánto varía el modelo a medida que cambiamos los datos de entrenamiento). Como se puede observar, si nos centramos en el lado derecho de la gráfica, donde hay suficientes datos para una evaluación, observamos que el espacio entre las dos curvas ha disminuido considerablemente, lo que nos indica que nuestro modelo no sufre *overfitting*; ni están excesivamente juntas (lo que podría suponer un problema de *underfitting*).

Con todo este estudio, podemos asegurar que nuestra elección y el error presentado son de alta calidad.

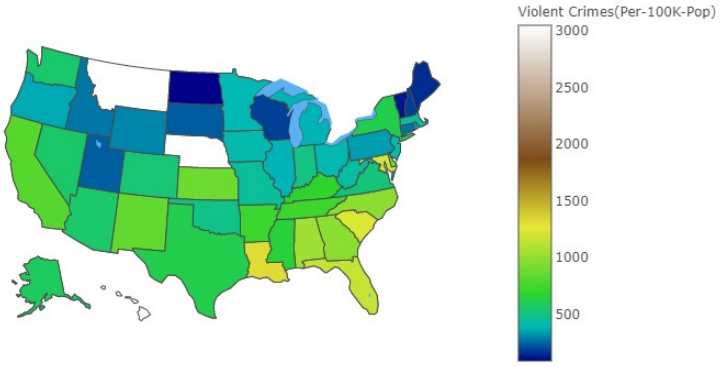
2 – Problema de regresión

Communities and Crime

- *Comprensión del problema*

Trabajaremos con el conjunto de datos *Communities and Crime* , que contiene un conjunto de datos recogidos de comunidades de Estados Unidos (variable X) con el propósito de predecir el número total de crímenes violentos (variable Y) por cada 100000 habitantes (función f:X→Y).

Trabajaremos con variables numéricas representadas por atributos reales. De donde obtenemos 1994 instancias, las cuales serán las filas de nuestro conjunto de datos; y 128 columnas, donde cada una representa un atributo de entre los cuales solo 122 son predictivos.



- *Análisis de los datos*

El primer paso dentro del preprocesamiento será realizar la lectura de los datos y la separación de sus características de las etiquetas.

Para ello, dispondremos de una función que lee los datos de un archivo. Esta función llamará a *"read_csv"* de pandas, con la que leeremos los datos guardándolos en un DataFrame, le añadiremos el parámetro *"header=None"* ya que, de lo contrario, al visualizar los datos nos daremos cuenta de que se han leído con una estructura incorrecta al considerar la primera fila como la cabecera.

```
def leer_datos(archivo, separador=None):
    datos ← pd.read_csv(archivo, separador, header=None)
    return datos
```

De esta forma, nuestros datos serán visualizados de la siguiente manera:

```
Leemos los datos:
Mostramos los datos de entrenamiento:
   0   1    ...   126   127
0    8   ?  ...  0.14  0.20
1   53   ?  ...     ?  0.67
2   24   ?  ...     ?  0.43
3   34   5  ...     ?  0.12
4   42  95  ...     ?  0.03
```

Leyendo la descripción del conjunto de datos, vemos que se especifica el nombre de cada una de las columnas del conjunto de datos. Estos nombres, los cuales representarían la cabecera de nuestro conjunto de datos, tienen la estructura:

@attribute nombre tipo_variable

Al tener una gran cantidad de valores, veo adecuado añadir esta cabecera a nuestro conjunto de datos para saber con más seguridad el significado de cada una de las características de nuestro conjunto.

```
names_url ← archive.ics.uci
r ← requests.get(names_url)
columns ← re.findall(@attribute (\w*) , r.text)
del r
data.columns ← columns
```

Aunque en el propio código comento con detalle el funcionamiento de estas lineas, la idea básica sería hacer una petición a la *url* donde se encuentra el conjunto de datos, que analiza el documento descriptivo del conjunto de datos y busca todas aquellas lineas que cumplan la estructura *"@attribute + cadena de caracteres"*

El resultado es el siguiente:

```
   state county  ... PolicBudgPerPop ViolentCrimesPerPop
0      8      ?  ...            0.14                0.20
1     53      ?  ...               ?                0.67
2     24      ?  ...               ?                0.43
3     34      5  ...               ?                0.12
4     42     95  ...               ?                0.03
```

También se nos indica que de las 128 columnas, 5 de ellas no son predictivas. Se refiere a las 5 primeras columnas: state, country, com-

munity, communityname y fold. Por tanto, se eliminarán del conjunto de datos para evitar problemas en los cálculos:

- ***Eliminar variables no predictivas:*** Puesto que conocemos el nombre de las columnas, solo se necesitará llamar a la función **drop(columns=['nombre_1', 'nombre_2' , ... , 'nombre_n'], axis = 1)**. El parámetro *"axis = 1"* indica que se desea eliminar por columnas y no por filas.

Al visualizar los datos, lo primero que podemos observar es que faltan valores ('?').

- ***Eliminar valores desconocidos:*** Para ello, primero sustituiremos todas las '?' del conjunto de datos por valores *'NaN'*: **data.replace[to_replace='?', value=NaN]**.

Una forma de visualizar las columnas a las que le faltan valores y cuántos valores faltan, es la siguiente:

```
miss_values ← data.columns[data.isnull().any()]
data[miss_values[rango_filas].describe()
```

```
Columnas a las que le faltan valores:
Index(['OtherPerCap', 'LemasSwornFT', 'LemasSwFTPerPop', 'LemasSwFTFieldOps',
       'LemasSwFTFieldPerPop', 'LemasTotalReq', 'LemasTotReqPerPop',
       'PolicReqPerOffic', 'PolicPerPop', 'RacialMatchCommPol',
       'PctPolicWhite', 'PctPolicBlack', 'PctPolicHisp', 'PctPolicAsian',
       'PctPolicMinor', 'OfficAssgnDrugUnits', 'NumKindsDrugsSeiz',
       'PolicAveOTWorked', 'PolicCars', 'PolicOperBudg', 'LemasPctPolicOnPatr',
       'LemasGangUnitDeploy', 'PolicBudgPerPop'],
      dtype='object')
(23,)

       OtherPerCap LemasSwornFT  ... PctPolicBlack PctPolicHisp
count         1993          319  ...           319          319
unique          97           38  ...            73           54
top              0         0.02  ...             0            0
freq           129           80  ...            23           72

[4 rows x 13 columns]

       PctPolicAsian PctPolicMinor  ... LemasGangUnitDeploy PolicBudgPerPop
count            319           319  ...                319            319
unique            50            72  ...                  3             51
top                0          0.07  ...                  0           0.12
freq             189            14  ...                126             22

[4 rows x 10 columns]
```

De esta manera, podemos saber el nombre de todas las columnas a la que le faltan valores; y mediante la función **describe()** , vemos información básica de estas columnas. El parámetro *"count"* indica cuántos valores son distintos de *NaN*.

Eliminaremos estas columnas mediante la función **data.dropna(axis=1)**

Por tanto, nuestro conjunto de datos se verá de la siguiente manera:

```
Eliminamos las filas que no contengan todos los datos:
   population  householdsize  ...  LemasPctOfficDrugUn  ViolentCrimesPerPop
0        0.19           0.33  ...                 0.32                 0.20
1        0.00           0.16  ...                 0.00                 0.67
2        0.00           0.42  ...                 0.00                 0.43
3        0.04           0.77  ...                 0.00                 0.12
4        0.01           0.55  ...                 0.00                 0.03
```

Aplicamos la función que separa las características (a las que llamaremos X) de las etiquetas (a las que llamaremos Y):

```
def separar_datos(datos):
    valores ← datos.values  #Devuelve una representación numpy del DataFrame
    X ← valores[: , :-1] #Todas las filas y columnas (excepto la última)
    Y ← valores[:, -1] #La última columna de todas las filas

    return X, Y
```

- ***Reducción de dimensionalidad***: reduciremos el número de dimensiones, quedándonos con aquellas que expliquen en mayor medida la varianza de los datos.

Utilizaremos la técnica **PCA (Principal Component Anlaysis)** que, atendiendo al libro Learning From Data, vemos que trabaja sobre la matriz de datos basándose en *SVD (Singular Value Descomposition).* Manteniendo una representatividad del 98% conseguimos mejorar un poco más los valores de los datos, aplicando una reducción de dimensionalidad de:

```
Dimensiones antes de aplicar PCA:  99
Dimensiones depsués de aplicar PCA:  45
```

Además, cabe destacar que leyendo la descripción del conjunto de datos, se nombra lo siguiente: *"all numeric data was normalize into de decimal range 0.00-1.00"* , por lo que se da por hecho que ya se ha producido una normalización de los datos y no será necesaria usar la función **Normalize()**.

- *Selección de funciones*

Vamos a utilizar la clases de **funciones lineales**. No tiene sentido aplicar combinaciones no lineales a los datos puesto que hacer transformaciones como añadir más dimensiones pueden llevarnos a problemas como el overfitting.

- *Fijar conjuntos training, validación y test.size*

En este problema, los conjuntos training y test no se encuentran por separado. Para ello, vamos a hacer que el 70% de los datos formen parte del conjunto training y que el 30% restante formen parte del conjunto test con el que se estimará posteriormente un error Etest con el que acotar Eout.

Para llevar a cabo esta división, hacemos uso de la función:

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state = 1, shuffle = True) , donde *test_size* será el tamaño del conjunto test (30%).

Para cada modelo, con el conjunto de entrenamiento (data_train) realizaremos **Cross-Validation (10-fold)**. Para ello, utilizaremos dos funciones:

- ***ShuffleSplit(n_splits=10, shuffle=True, random_state=1)*** , la cual muestreará aleatoriamente el conjunto de datos durante cada iteración para generar un conjunto de entrenamiento y un conjunto de pruebas.

- ***cross_val_score(model, x , y, scoring = 'neg_mean_squared_error' cv = cv)*** , función que se encarga de realizar la validación cruzada para cada modelo que indicamos en el parámetro *"model"* con los conjuntos de datos proporcionados.

A diferencia del problema de clasificación, ya no podemos utilizar como parámetro de *scoring* el valor *"accuracy"*; sino que al tratarse de un problema de regresión, empleamos **"neg_mean_squared_error'** , que especifica el error cuadrático medio (el promedio de los errores al cuadrado), es decir, la diferencia entre el estimador y lo que se estima.

Se realizarán n_splits iteraciones, y en cada una el conjunto de datos es barajado y se utilizará el conjunto de entrenamiento y prueba de cada iteración, de manera que los conjuntos de prueba pueden superponerse entre las divisiones.

Aunque utilizar validación cruzada aumenta el tiempo de ejecución, se entrenará el modelo y obteniendo una estimación más precisa y reduciendo el posible overfitting.

Este proceso lo he realizado para todos los modelos evaluados:

```
models ← []
models ← modelo_1
models ← modelo_2
...
models ← modelo_n

cv ← ShuffleSplit(n_splits=10, shuffle=True, random_state=1)
means ← []
deviations ← []

for model in models:
    scores ← cross_val_score(model, x , y, scoring=male_scorer(r2_score), cv=cv)

    means ← scores.mean()
    deviations ← std(scores)
```

- Necesidad de regularización

Al igual que en el problema de clasificación, Utilizaremos las dos técnicas de regularización más utilizadas: **Ridge** y **Lasso**, las cuales se incorporan al usar los modelos mediante el argumento *"penalty"* , el cual toma los valores:

- l1 → Regularización Lasso
- l2 → Regularización Ridge

Regresión Ridge: añade "*magnitud al cuadrado*" del coeficiente como término de penalización en la función de pérdida:

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

Cost function

- As $\lambda \to 0$, $\hat{\beta}_{ridge} \to \hat{\beta}_{OLS}$;
- As $\lambda \to \infty$, $\hat{\beta}_{ridge} \to 0$.

El parámetro λ es el que regula la penalización. Cuando $\lambda \to 0$ (tiende a 0), la regresión Ridge se vuelve muy similar a los mínimos cuadrados; sin embargo, cuando λ crece, los coeficientes tienden a cero sin llegar a ser cero nunca ; por lo que cuanto más grande es su valor, más fuerte será el tamaño de los coeficientes penalizados.

Esta técnica suele funcionar bien si hay muchos parámetros grandes de aproximadamente el mismo valor, es decir, cuando la mayoría de los predictores tiene impacto en la respuesta.

Regresión Lasso: añade "magnitud de valor absoluto" del coeficiente como término de penalización en la función de pérdida.

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Cost function

En este caso, cuanto mayor sea λ, más coeficientes tendrán valor cero.

Esta técnica tiende a funcionar bien is hay un pequeño número de parámetros significativos y los otros tienen valores cercanos a cero (es decir, cuando solo unos pocos predictores influyen en la respuesta).

- Identificar los modelos LINEALES a usar

La métrica de ajuste empleada para estos modelos **MSE (Mean Squared Error, error cuadrático**), para penalizar en mayor medida los errores que se alejen en mayor medida de los reales.

1 – Regresión Lineal:

```
logistic = LinearRegression()
```

Probaremos a evaluar la Regresión Lineal sin regularización, sobre todo, para comparar su rendimiento frente a las regularizaciones de otros modelos lineales.

Por defecto:

- **fit_intercept = True**. Calcula la intercepción para el modelo
- **normalize = False.** Normalizamos los datos antes de pasarlos al modelo
- **copy_x = True.** Genera una copia de X para no sobreescribir la nuestra.

A parte de la Regresión lineal sin regularización, probaremos los modelos:

Ridge: **logistic = Ridge()**

Evaluaremos la Regresión Lineal de tipo norma l2. Como ya se ha explicado antes, se añade la suma de los cuadrados de cada peso, ponderado con un hiperparámetro α , del que usaremos su valor por defecto 1.0.

Lasso: **logistic = Lasso()**

Evaluaremos la Regresión Lineal de tipo *norma l1*, que añade la suma del valor absoluto de los coeficientes con un α de valor 1.0.

2 – Support Vector Regression:

logistic = SVR()

Usaremos un modelo de tipo **Support Vector Machine** enfocado a problemas de regresión: **Support Vector Regression (SVR)**. Su implementación se basa en **libsvm** (biblioteca de aprendizaje automático de código abierto), que nos proporciona una complejidad mayor que la cuadrática; por lo que está interesante saber si es capaz de obtener mejores resultados que los modelos de Regresión Lineal.

Con un modelo SVM se intenta encontrar un plano (de entre los infinitos planos que existen en los problemas de regresión) junto con un margen, donde se minimice la suma de los valores de violación del margen (ε_n) para cada punto de la muerta. Nosotros utilizaremos el parámetro de C con su valor por defecto, el cual es 1.0, ya que cuanto más peso se le de a este parámetro, mayor será el ajuste del modelo, lo que nos podría llevar a una situación de *overfitting*.

Por defecto:

- **Kernel = 'rbf'.**
- **gamma = 3** ; es el coeficiente que usará el algoritmo 'rbf'
- **tol = 0'0004** . Criterio de parada.

53– SGD Regressor:

logistic = SGDRegressor(pentalty='l')

Probaremos, al igual que hicimos en el problema de clasificación, con un modelo basado en Gradiente Descendente Estocástico, para problemas de regresión. El gradiente de la pérdida se estima a la vez en cada muestra y el modelo se actualiza mediante la tasa de aprendizaje.

Para este modelo, probaremos los dos tipos de regularización explicados: *Ridge y Lasso*; modificando el parámetro *"penalty"*.

Por defecto:

- **tol = 0'003** , criterio de parada.
- **max_iter = 1000** , número de iteraciones máximo.

4 – Random Forest Regressor:

logistic = RandomForestRegressor(n_estimators=100, random_state=1)

Por último, he querido probar por qué resultados puede dar un modelo **Random Forest**, el cual es un meta-estimador que se ajusta a varios árboles de decisión de clasificación en varias submuestras del conjunto de datos y utiliza un valor promedio para mejorar la precisión predictiva y mejorar el sobreajuste.

En nuestro caso, se generará 100 árboles y por defecto:

- **max_features = 'auto'** . Número de características a considerar cuando se busca la mejor partición. Existen otros parámetros como 'log2' o 'sqrt', pero el valor por defecto es el que mejor resultados proporcionó al probar.
- **max_depth = None** . Profundidad máxima del árbol.
- **min_impurity_split = None** . Tope establecido cuando se utiliza la técnica de *"early-stopping".*

- Selección del mejor modelo

Para elegir el mejor modelo, aplicaremos todo lo anteriormente explicado para cada uno de los modelos indicados en el apartado anterior.

Por ello, para cada modelo haremos *cross-validation* con el conjunto de entrenamiento, e iremos probando un conjunto de hiperparámetros (cómo el tipo de regularización el la Regresión Lineal o en el SGD).

En el problema anterior, al ser de clasificación, nos basábamos en el resultado de la precisón (accuracy) para saber qué tan bueno era nuestro modelo. En un problema de regresión, no sólo se trata de decir si se ha predicho bien o mal; sino que queremos saber a partir de la métrica qué tan cerca se encuentra de la predicción.

Esto quiere decir que vamos a evaluar nuestros modelos con una métrica MSE (Mean Squared Error); donde se penalizará más los errores grandes que los pequeños; siendo una mejor métrica que, por ejemplo, MAE (Mean Absolute Error), el cual sería de más utilidad en problemas con datos con muchos outliers.

Los resultados son los siguientes:

Modelos	***Media MSE***	***Desviación***
Regresión Lineal	***0.018949***	***0.001052***
Ridge	***0.018806***	***0.000979***
Lasso	***0.052194***	***0.002618***
SVR	***0.020090***	***0.000949***
SGDR + Ridge	***0.018834***	***0.000815***
SGDR + Lasso	***0.018834***	***0.000789***
Random Forest R.	***0.021506***	***0.001774***

Observando los datos obtenidos, podemos obtener una serie de conclusiones. Primero, **la Regresión Lineal con regularización Ridge es la que ha obtenido el menor error medio de entre todos los modelos**, a diferencia de la Regresión Lineal con regularización Lasso, que ha obtenido el peor de los resultados.

Los modelos de SGDRegression han tenido el mismo comportamiento que los SGD del problema de clasificación: se adaptan bien al

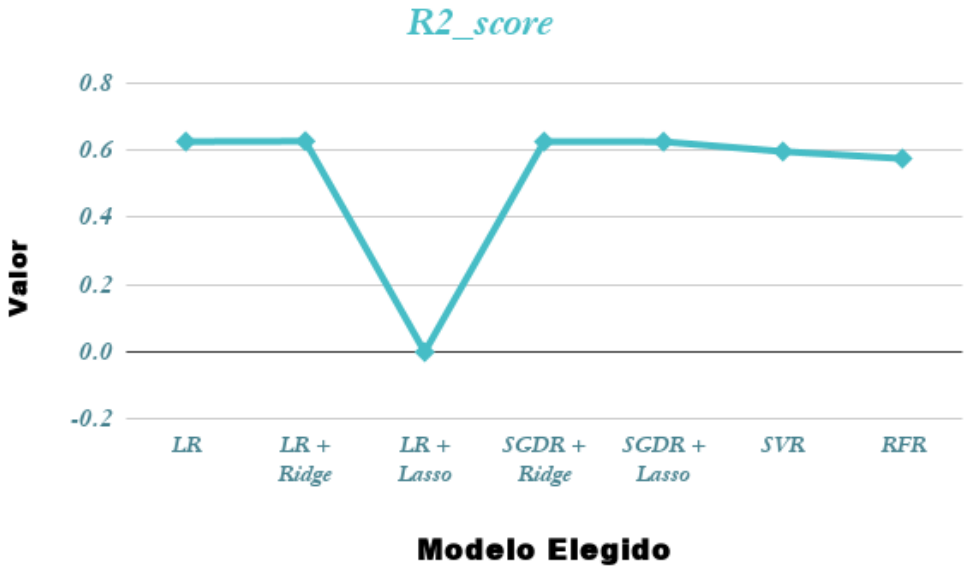
problema, dan resultado generalmente buenos, aunque ninguno llega a ser el mejor.

La diferencia entre los resultados obtenidos en general con todos los modelos y el resultado de Lasso, hace que concluyamos que este tipo de regularización no es útil para este problema; esto podría deberse a que quizás hay parámetros con un coeficiente más bajo que tienda a ser 0 rápidamente, a pesar de que tengan una influencia mayor en el resultado del modelo; por lo que los resultados se vuelven peores.

También llama la atención los valores de la Regresión Lineal sin regularización, pues es está entre los mejores valores. Al contrario de lo que ocurre en los modelos Random Forest Regression y SVR ya que, para mi sorpresa, son los que peores errores han obtenido. Con esto podemos deducir dos conclusiones: primero, es posible que, para este problema y conjunto de datos, no fuese muy necesaria la regularización, es decir, no había tanta probabilidad de obtener overffiting; y segundo, la tabla de resultados demuestra que elegir un modelo más complicado no conlleva obtener mejores resultados. Podemos pensar que quizás, en el caso del SVR al necesitar un ajuste más específico de sus hiperparámetros, hace que sa más difícil ajustarlos a mano para obtener unos mejores resultados.

Para los problemas de regresión, no existe una función como *classification_reports* ; sin embargo, una buena métrica sería calcular para los modelos el coeficiente de determinación , mediante la función **r2_score(Y_test, Y_predict)**

A continuación mostraré de manera gráfica (*usando Google Docs*) los valores obtenidos para este parámetro de cada uno de los modelos:



El mejor valor posible a obtener es 1.0 , y puede tomar valores negativos si el modelo se adapta muy mal. A simple vista, lo que mas llama la atención es la Regresión Lineal con regularización Lasso que, como ya hemos comentado con anterioridad, ha sido el que peores resultados ha obtenido, reflejándose en el valor del *r2_score*, siendo prácticamente negativo.

De entre los restantes, todos toman valores entre 0’55 y 0’65, no habiendo demasiadas variaciones entre ellos.

Como conclusión, el mejor modelo obtenido será Regresión Lineal con regularización Ridge.

- Estimación del Eout

En el mundo real, no podemos calcular un Eout exacto, puesto que no conocemos la totalidad de las instancias posibles; es por ello que estimaremos Eout mediante Etest.

Sabiendo que Etest es una buena cota para Eout, podremos asegurar con certeza que, ante datos futuros, obtendríamos un error parecido al cometido con los datos de test.

Dicho esto, nuestro Etest será el error cometido por nuestro modelo con los datos de prueba. Para estimar Eout, usaremos la siguiente desigualdad:

$$E_{out}(g) <= E_{test}(g) + \sqrt{(\frac{1}{2N} ln \frac{2}{\alpha})}$$

donde $\boldsymbol{\alpha}$ = 0.05 , es decir, una confianza del 95 %

```
def cota_Etest(Etest):
    Eout ← sqrt(1/2*X_test.size)*log(2/□))
    return Etest + Eout
```

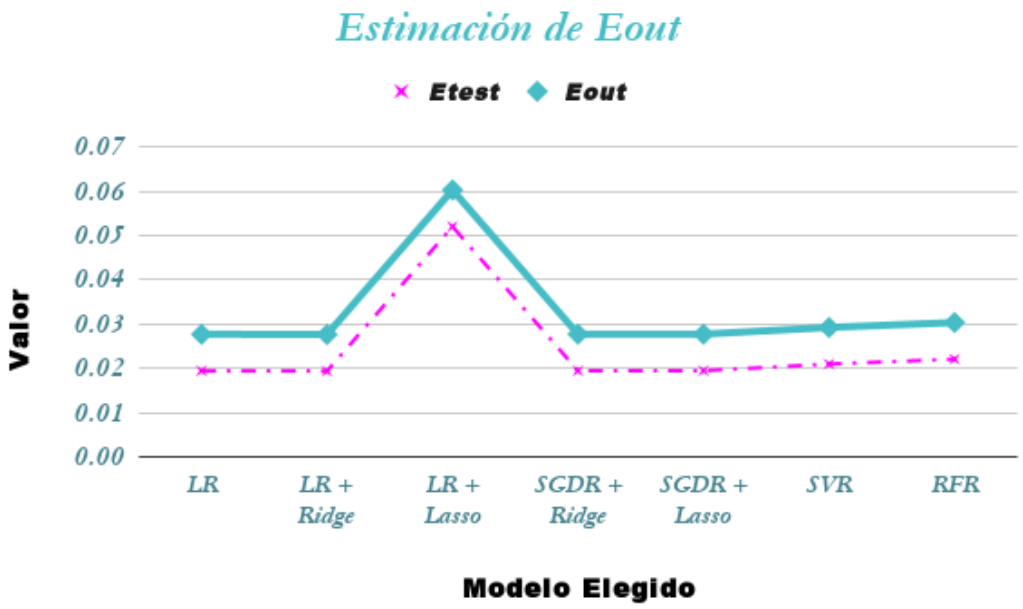
Pseudocódigo del proceso de ajuste y entrenamiento del modelo elegido:

```
logistic = Ridge().fit(X_train, Y_train)
predicted = logistic.predict(X_test)
Etest = mean_squared_error(Y_test, predicted)
Eout = cota_Etest( mean_squared_error(Y_test, predicted))
r2_score = r2_score(Y_test, predicted)
```

- Etest: 0.0194035683068686436
- Estimacion del out: 0.027675607164471455

Como podemos observar, nuestro mejor modelo ha acertado al **predecir el 99% de las etiquetas**. El valor de Etest es muy similar a los resultados concluidos del uso de validación cruzada. Uno de los motivos por el que se ha reducido un poco el error es que en este caso hemos ajustado el modelo con todos los datos de entrenamiento, no solo de una parte como en la validación cruzada. Además de que aunque la estimación del Eout sea algo mayor, sigue siendo muy bueno porque estaría indicando un **2% de error** ante datos fuera de la muestra.

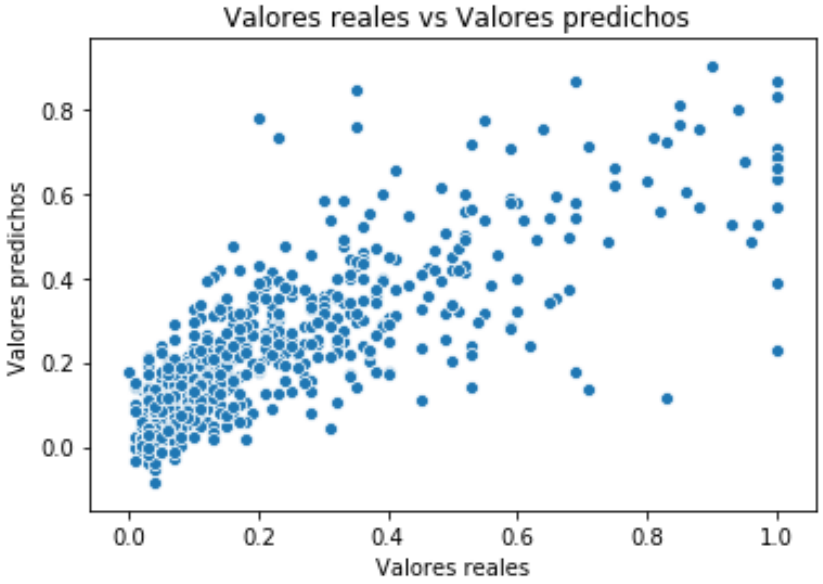
Aunque estos sean los valores para el mejor modelo, insertaré un gráfico con el Eout estimado mediante el Etest de todos los modelos trabajados; y una tabla con los valores exactos para las configuraciones de los mejores modelos discutidos en el apartado anterior:



Modelo	*Etest*	*Eout*
LR + Ridge	**0.019403**	**0.027675**
SGD + Ridge	**0.019462**	**0.027734**
SVR	**0.020998**	**0.02927**

En definitiva, viendo la tabla concluimos que la muestra de entrenamiento representa muy bien a la mayoría de los modelos y que estos han sido entrenados. Los modelos, en general, han aprendido el patrón que hay entre los datos de entrada y los resultados; de manera que nos asegura un buen funcionamiento ante nuevos datos.

Por último, para ver cómo de bien predice nuestro modelo ganador los datos, vamos a generar una gráfica de puntos:



En el eje X se representan los valores reales y en el eje Y los valores predichos. Fácilmente se ve que los valores predichos tienden a generar una recta lineal, la cual se ve algo distorsionada debido a los fallos presenten en en las predicciones. Los errores son más comunes cuando los valores crecen, ya que han sido representados más dispersados en comparación a cuando son más pequeños; aunque también influye el hecho de que la mayoría de los valores reales se encuentran en un rango entre [0.0, 0.5].

- Si fuese una empresa...

Como persona que ha realizado este ajuste en la empresa, debería hacerme las siguientes preguntas: ¿Representa el modelo de manera adecuada los datos? ¿Consideras que la calidad del modelo es buena? ¿Es tu modelo el que proporciona el mejor error?... Basándome en los resultados obtenidos, respondería afirmativamente las preguntas barajadas.

El Eout obtenido es una aproximación con respecto al valor de Etest; y cabe la posibilidad de que este Eout pueda llegar a ser mucho mayor si la muestra escogida para nuestro modelo y con la que hemos realizado las pruebas no representen bien la población. Sin embargo, para veitar este tipo de situaciones hemos realizado técnicas como la validación cruzada; ente otras con las que además intentamos evitar el overfitting mientras que conseguimos un buen ajuste del modelo final.

El hecho de realizar las técnicas y de hacer un estudio comparativo con respecto a otros modelos, nos asegura en gran medida la calidad del mismo. Concluimos también con que calidad no implica obligatoriamente la elección de un modelo de mayor complejidad, pues en nuestro caso, el mejor modelo ha sido uno de los más simples.

En mi opinión, elegir la Regresión Lineal con regularización Ridge ha sido una buena elección, ya que ha dado muy buenos resultados en general, obteniendo un error fuera de la muestra razonable y ha ofrecido una buena capacidad de generalización y desempeño a pesar de ser un modelo bastante simple.

3 – Bibliografía

https://scikit-learn.org/stable/modules/preprocessing.html

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html#sklearn.model_selection.cross_val_score

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html

https://scikit-learn.org/stable/modules/impute.html

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html#sklearn.linear_model.Ridge

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html#sklearn-metrics-classification-report

https://scikit-learn.org/0.15/modules/model_evaluation.html

https://scikit-learn.org/stable/modules/sgd.html

https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html

https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html#sklearn.metrics.r2_score

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

https://towardsdatascience.com/preprocessing-with-sklearn-a-complete-and-comprehensive-guide-670cb98fcfb9

https://datascience.stackexchange.com/questions/12321/difference-between-fit-and-fit-transform-in-scikit-learn-models

https://carlosjuliopardoblog.wordpress.com/2017/12/31/regularizacion-de-regresion-logistica/

https://es.wikipedia.org/wiki/Validaci%C3%B3n_cruzada#Objetivo_de_la_validaci%C3%B3n_cruzada

https://machinelearningmastery.com/k-fold-cross-validation/

https://machinelearningmastery.com/k-fold-cross-validation/

https://www.youtube.com/watch?v=6dbrR-WymjI

https://www.youtube.com/watch?v=0yI0-r3Ly40

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html

https://www.quora.com/What-is-the-C-parameter-in-logistic-regression

https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c

https://towardsdatascience.com/regularization-an-important-concept-in-machine-learning-5891628907ea

https://towardsdatascience.com/ridge-and-lasso-regression-a-complete-guide-with-python-scikit-learn-e20e34bcbf0b

https://www.datacamp.com/community/tutorials/tutorial-ridge-lasso-elastic-net

https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

https://towardsdatascience.com/a-short-introduction-to-model-selection-bb1bb9c73376

https://datascience.stackexchange.com/questions/29520/how-to-plot-learning-curve-and-validation-curve-while-using-pipeline

https://stats.stackexchange.com/questions/220827/how-to-know-if-a-learning-curve-from-svm-model-suffers-from-bias-or-variance/220852#220852

https://www.youtube.com/watch?v=EQWr3GGCdzw

https://www.it-swarm.dev/es/python/diferencia-entre-stratifiedkfold-y-stratifiedshufflesplit-en-sklearn/834810622/

- Tema 2: Linear Models
- Tema 4: Validation Model Selection
-Tema 5: Overfitting Regularizatin
- Learning From Data – Yaser S. Abu-Mostafa