Práctica 3 de Aprendizaje Automático

May 15, 2020

1 Communities And Crime

1.1 El data set

- Lo que tenemos entre manos son datos socioeconómicos del año 1990 en Estados Unidos. Un total de 1994 instancias con 128 atributos, aunque veremos cuántos de esos realmente nos sirven.
- Sabemos además que hay valores perdidos/nulos.
- Los atributos reales están ya normalizados a un rango 0.00-1.00.
- La etiqueta es el atributo "ViolentCrimesPerPop".

1.2 El problema

Tenemos que predecir el número de crímenes violentos en proporción a la población. Para ello tenemos muchas "features" como: el porcentaje de hogares que reciben ingresos de la seguridad social, la mediana de ingresos, el presupuesto de la policia en proporción a la población, etc...

1.3 Elementos

- X: Atributos predictores, nos dan 126, pero tendremos que descartar alguno no predictor.
- Y: Número de crímenes violentos en proporción a la población.
- Función objetivo: Dados los atributos de una comunidad, predecir el número de crímenes violentos en proporción a su población.

1.4 Archivo de datos

El CSV que nos proporcionan no tiene el nombre de los atributos, así que lo primero que he hecho es extraer el nombre de los mismos del archivo que nos dan (communities.name) y añadirlo como columnas para que sea más fácil preprocesarlos.

1.5 Semillas

Ya que *sklearn* nos ofrece parámetros para fijar semillas cuando haga falta, no las he puesto de manera global. Sin embargo, siempre que llamo a una función con algún componente aleatorio, la fijo como parámetro.

1.6 Clase de funciones a utilizar

Es lo habitual probar primero la clase de funciones lineal, es decir, combinaciones lineales de los predictores sin ninguna transformación. Si al validar modelos hubiera observado mucho error en el ajuste, es decir, demasiado sesgo, intentaría añadir alguna transformación no lineal. Sin embargo, no ha hecho falta.

1.7 Eliminando atributos no predictores

He considerado adecuado eliminar algunos atributos considerados no predictores:

- County: Es un código de area administrativa y además tiene muchos datos perdidos.
- Community: Un código que identifica la comunidad en cuestión, no nos interesa porque es un identificador de una instancia en concreto.
- Community Name: Lo mismo, pero con el nombre de la comunidad correspondiente al código.
- Fold: Nos proporcionan ya una descomposición para validación cruzada, la desechamos.
- State: Tal como está dado no puede usarse. Ya que confundiría a los algoritmos de entrenamiento (que un estado tenga el código 20 no significa que sea más que uno con código 3). Al principio iba a tratar de codificar este attributo categórico con un "One Hot Encoder", pero al ser tantos estados distintos el número de categorías iba a aumentar demasiado.

Nota: Esto entraría en el punto 4 de preprocesamiento, lo he hecho antes del split entre training y test para poder visualizar los datos sin las columnas que sobran.

[2]:
```
# Dropeamos las columnas con datos no predictores

data_without_some_columns = raw_data.drop(['state','county', 'community',␣
 ↪'communityname', 'fold'], axis=1)
data_without_some_columns
```

1.8 ¿Qué hacemos con los datos perdidos?

Al ser un data set real, es normal que nos encontremos con datos que faltan. Vamos a echar un vistazo, a ver cuantos datos faltan en cada columna.

[3]:
```
# Itero por cada columna y cuento los datos perdidos que tiene

for i,c in (data_without_some_columns == '?').sum().items() :
    if c > 0:
        print(i,c)
```

```
OtherPerCap 1
LemasSwornFT 1675
LemasSwFTPerPop 1675
LemasSwFTFieldOps 1675
```

```
LemasSwFTFieldPerPop 1675
LemasTotalReq 1675
LemasTotReqPerPop 1675
PolicReqPerOffic 1675
PolicPerPop 1675
RacialMatchCommPol 1675
PctPolicWhite 1675
PctPolicBlack 1675
PctPolicHisp 1675
PctPolicAsian 1675
PctPolicMinor 1675
OfficAssgnDrugUnits 1675
NumKindsDrugsSeiz 1675
PolicAveOTWorked 1675
PolicCars 1675
PolicOperBudg 1675
LemasPctPolicOnPatr 1675
LemasGangUnitDeploy 1675
PolicBudgPerPop 1675
```

Podemos ver que hay unas cuantas columnas en esta situación, tenemos tres opciones:

- Eliminar las instancias que tienen datos perdidos en alguna de las categorías (la descartamos, porque son demasiadas).
- Eliminar las columnas con datos perdidos.
- De alguna manera rellenar los datos que faltan. Esta sería quizás la ideal, pero no tenemos la información suficiente para inferir los datos perdidos.

Tendremos que eliminar las columnas con datos perdidos. Salvo "OtherPerCap" que solamente tiene 1 instancia sin información. En ese caso simplemente buscamos dicha instancia y la eliminamos para que no cause problemas.

[4]:
```
# Saco por pantalla la fila con la celda 'OtherPerCap' perdida

print(data_without_some_columns.index[data_without_some_columns['OtherPerCap']␣
 ↪== '?'].tolist())

# Elimino la fila

data_without_some_columns = data_without_some_columns.drop(130)
```

```
[130]
```

Por último, eliminamos las columnas con 1675 datos perdidos:

[5]:
```
# Utilizo "list comprenhension" para obtener el nombre de las columnas
# que aún tienen datos perdidos
```

```
columns_to_drop = [i for i,c in (data_without_some_columns == '?').sum().items()␣
 ↪if c > 0]

# Elimino esas columnas

clean_data = data_without_some_columns.drop(columns_to_drop, axis=1)
clean_data
```

[6]:
```
# Ahora que no tenemos ningún '?' podemos asegurarnos de que todas las columnas␣
 ↪son de tipo float64
clean_data = clean_data.astype('float64')
```

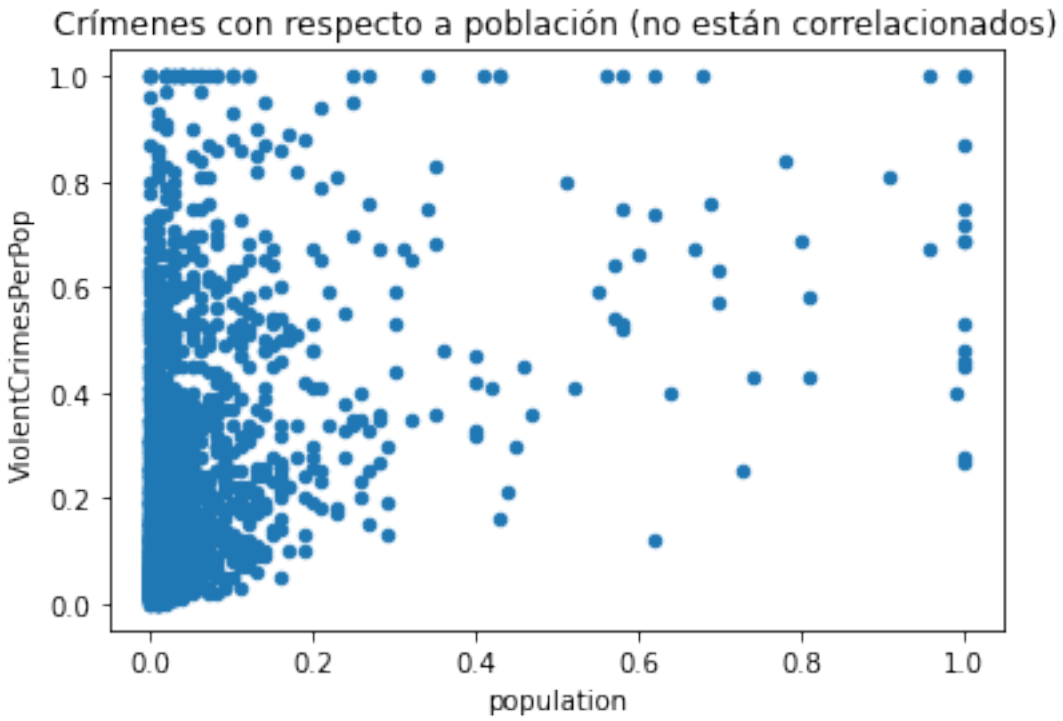
1.9 ¿Proyección?

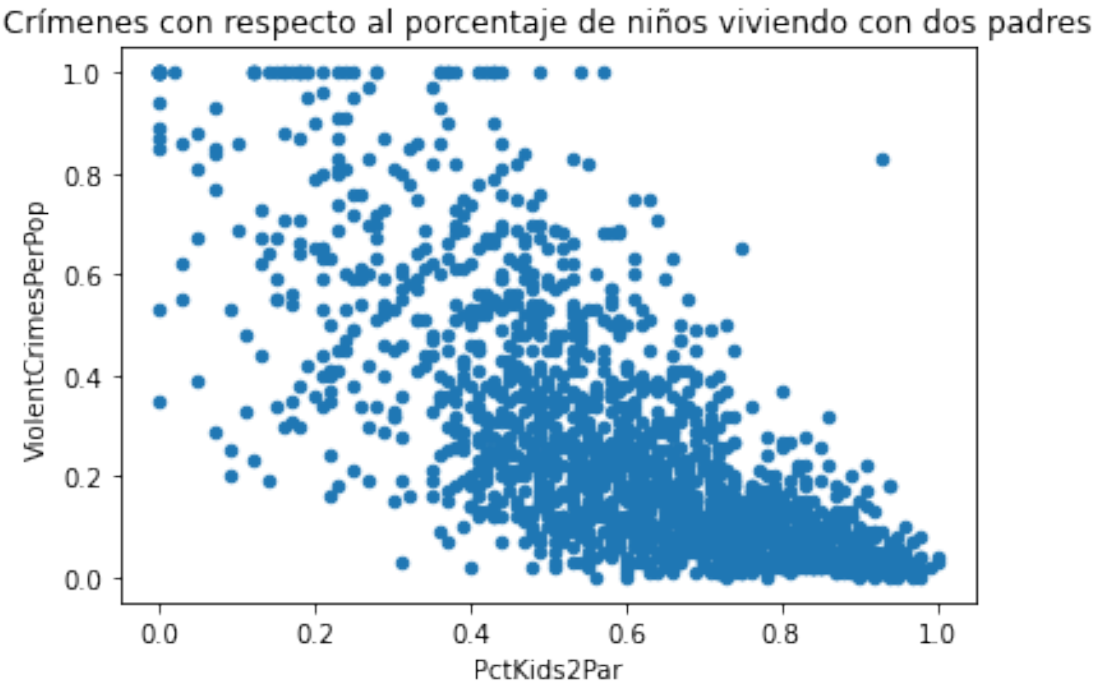
Según la documentación de sklearn: *Random Projections are a simple and computationally efficient way to reduce the dimensionality of the data by trading a controlled amount of accuracy (as additional variance) for faster processing times and smaller model sizes.*

La proyección nos ayuda a reducir la dimensionalidad de X, sin embargo, no he tenido problemas de tiempo a la hora de ajustar los modelos, por lo que no veo necesario hacer ningún *tradeoff* aquí.

1.10 Alguna visualización

Ahora que tenemos los datos "limpios", podemos obtener alguna representación visual.





1.11 Obteniendo X, Y

ViolentCrimesPerPop es la etiqueta que queremos predecir. Por eso, generamos una matriz X sin esa columna y un vector Y con solo esa columna.

[9]:
```
# Para X dropeo la etiqueta

X = clean_data.drop('ViolentCrimesPerPop', axis=1).to_numpy()

# Para Y me quedo solo con la etiqueta

Y = clean_data['ViolentCrimesPerPop'].to_numpy()

# Imprimo la forma de X e Y

print('X shape:', X.shape)
print('Y shape:', Y.shape)
```

```
X shape: (1993, 100)
Y shape: (1993,)
```

1.12 Conjuntos de entrenamiento y test

He aplicado la regla del 80/20 ya que el dataset no es inmenso. Quizás un 70/30 hubiese sido idílico para poder establecer la cota de Eout con más seguridad, pero al no ser tan grande, me he

decantado por tener más datos para entrenar.

[10]:
```
from sklearn.model_selection import train_test_split

# Utilizamos esta función de sklearn que nos permite además
# fijar una semilla y el tamaño del conjunto test cómodamente

X_train, X_test, Y_train, Y_test = train_test_split(X,Y,test_size=0.2,␣
 ↪random_state=9, shuffle=True)

# Imprimo la forma de los conjuntos

print('X_test shape:', X_test.shape)
print('Y_test shape:', Y_test.shape)
print('X_train shape:', X_train.shape)
print('Y_train shape:', Y_train.shape)
```

```
X_test shape: (399, 100)
Y_test shape: (399,)
X_train shape: (1594, 100)
Y_train shape: (1594,)
```

1.13 Métrica de error

Aquí tenemos fundamentalmente dos opciones:

- Error cuadrático medio (MSE)
- Error absoluto medio (MAE)

La diferencia esencial es que MSE tiene una penalización adicional a los errores grandes. ¿Cuál es la ventaja de MAE? Que es mas robusto a *outliers*.

Por lo general, si consideramos los *outliers* anomalías en observaciones, debemos usar MAE. Si, en cambio, tenemos que considerarlos para nuestro estudio deberíamos usar MSE.

En concreto, considero que en este data set es perfectamente válido que una comunidad sea un *outlier* en crimen por lo que me he decantado por MSE.

1.14 Regularización

La regularización es considerada nuestro primer remedio contra el overfitting:

- Restringe el modelo de aprendizaje para reducir el error fuera de la muestra.
- Ayuda especialmente si hay ruido en el conjunto de datos de entrenamiento.

Alguno de los modelos que vamos a validar tienen opciones de regularización, veremos si aplicarla es beneficioso para el aprendizaje.

1.15 Validación cruzada

Para elegir un modelo, vamos a dividir el training test en 10 partes iguales:

- Probamos cada modelo entrenando en 9 partes y validando en 1.
- Lo hacemos 10 veces, dejando para validar 1 parte distinta cada vez. - Hacemos la media de las validaciones.
- En este caso las 10 partes se hacen de manera aleatoria (no hay nada que estratificar para hacer representativas todas las partes), será distinto en el problema de clasificación.

¿Cuál es el objetivo de la validación?

Utilizar Eval como un estimador de Eout no sesgado para cada modelo que probemos y así hacer una buena elección, y no guiarnos con Ein.

¿Cuál es la ventaja de la validación cruzada con respecto a la validación?

Está considerada empíricamente un mejor estimador del error fuera de la muestra. Y nos permite usar la validación para la elección de un modelo con más confianza.

[11]:
```
from sklearn.model_selection import cross_val_score

# Helper para evaluar los modelos con cross validation y además
# tener un buen output para comparar

def validate_models(model_strings, models, X_train, Y_train):
    validation_results = {'mean_squared_error':[], 'std_dev_error': []}
    for model in models:
        # Utilizamos cross_val_score de sklearn que hace la validación cruzada␣
 ↪por nosotros.
        # Devuelve el valor de cada validación (en este caso 10)
        results = cross_val_score(model, X_train, Y_train, scoring =␣
 ↪"neg_mean_squared_error", cv=10)

        # Hacemos la media y desviación típica de las 10 validaciones

        validation_results['mean_squared_error'].append(np.mean(np.abs(results)))
        validation_results['std_dev_error'].append(np.std(np.abs(results)))

        # Devolvemos una tabla con los modelos y sus métricas

    return pd.DataFrame(data=validation_results, index=model_strings)
```

1.16 Ordinary Least Squares

Empezamos por uno de los métodos que implementamos en la práctica 1, no hay hiperparámetros que ajustar.

Tratamos de minimizar: $\min_w ||Xw - y||_2^2$

Computa la pseudoinversa $(X^T X)^{-1} X^T$ y después simplemente multiplica esta por y (las labels) del dataset, para obtener los pesos w con pérdida mínima.

[12]:
```
# Modelo Ordinary Least Squares

from sklearn.linear_model import LinearRegression

models = [
    LinearRegression(),
]

model_strings = [
    'OLS',
]

validate_models(model_strings, models, X_train, Y_train)
```

[12]:
```
     mean_squared_error  std_dev_error
OLS            0.019191       0.002272
```

1.17 Ridge Regression

Este modelo es básicamente *Linear Least Squares* con regularización por la norma l2. Trata de minimizar la función: ||y - Xw||^2_2 + alpha * ||w||^2_2

Con esta regularizacón tratamos de hacer los pesos más pequeños, más adelante, veremos otro tipo de regularización con Lasso.

He probado con alpha {0.1, 0.5, 1, 1.5, 2, 3}.

Alpha marca la fuerza de la regularización. El solver es por defecto SVD, que no tiene en cuenta la semilla, por eso no se la pongo.

[13]:
```
# Modelo Ridge Regression

from sklearn.linear_model import Ridge
models = [
    Ridge(alpha=0.1),
    Ridge(alpha=0.5),
    Ridge(alpha=1),
    Ridge(alpha=1.5),
    Ridge(alpha=2),
    Ridge(alpha=3),
]

model_strings = [
    "Ridge alpha = 0.1",
    "Ridge alpha = 0.5",
    "Ridge alpha = 1.0",
```

```
    "Ridge alpha = 1.5",
    "Ridge alpha = 2.0",
    "Ridge alpha = 3.0"
]

validate_models(model_strings, models, X_train, Y_train)
```

[13]:

	mean_squared_error	std_dev_error
Ridge alpha = 0.1	0.019039	0.002247
Ridge alpha = 0.5	0.018877	0.002230
Ridge alpha = 1.0	0.018828	0.002238
Ridge alpha = 1.5	0.018816	0.002246
Ridge alpha = 2.0	0.018816	0.002252
Ridge alpha = 3.0	0.018830	0.002257

1.18 Stochastic Gradient Descent Regressor

Es el método que ya conocemos:

- Se estima el gradiente de la perdida para cada dato.
- Se actualizan los pesos con el learning rate (por defecto 0.01) que le pasemos.
- Por defecto se barajean los datos después de cada *epoch*.
- Por defecto el learning rate se va actualizando con el siguiente esquema: $eta = \frac{eta_0}{t^power\_t)}$ Siendo power_t otro parámetro (por defecto 0.25)

Con opciones de regularización:

- Norma l2: $R(w) := \frac{1}{2}\sum_{j=1}^{m} w_j^2 = ||w||_2^2$
- Norma l1: $R(w) := \sum_{j=1}^{m} |w_j|$

He probado sin regularización, con regularización l1 y con regularización l2 para alpha = {0.0001, 0.0002, 0.00025}

[14]:
```
# Modelo Stochastic Gradient Descent Regressor

from sklearn.linear_model import SGDRegressor
models = [
    SGDRegressor(alpha=0, random_state=1),
    SGDRegressor(alpha = 0.0001, random_state=1),
    SGDRegressor(alpha=0.0002, random_state=1),
    SGDRegressor(alpha=0.00025, random_state=1),
    SGDRegressor(penalty='l1', alpha=0.0001, random_state=1),
    SGDRegressor(penalty='l1',alpha=0.0002, random_state=1),
    SGDRegressor(penalty='l1', alpha=0.0025, random_state=1),
]

model_strings = [
```

```
    "SGD, sin regularización",
    "SGD, regularización L2 alpha = 0.0001",
    "SGD, regularización L2 alpha = 0.0002",
    "SGD, regularización L2 alpha = 0.00025",
    "SGD, regularización L1 alpha = 0.0001",
    "SGD, regularización L1 alpha = 0.0002",
    "SGD, regularización L1 alpha = 0.00025",
]

validate_models(model_strings, models, X_train, Y_train)
```

[14]:

	mean_squared_error	std_dev_error
SGD, sin regularización	0.019703	0.001945
SGD, regularización L2 alpha = 0.0001	0.019703	0.001945
SGD, regularización L2 alpha = 0.0002	0.019704	0.001944
SGD, regularización L2 alpha = 0.00025	0.019705	0.001944
SGD, regularización L1 alpha = 0.0001	0.019726	0.001927
SGD, regularización L1 alpha = 0.0002	0.019751	0.001915
SGD, regularización L1 alpha = 0.00025	0.020545	0.001738

1.19 Lasso

Pese a que este método no lo conocemos también, he decidido probarlo. Tiene como característica principal intentar que haya muchos pesos a cero. Para que las soluciones dependan de pocos atributos.

- La función a minimizar es: $\min_w \frac{1}{2n_{\text{samples}}}||Xw - y||_2^2 + \alpha||w||_1$
- Utiliza L1 como regularización.
- Sigue el modelo de OLS y Ridge de la pseudoinversa.

[15]:
```
# Modelo Lasso

from sklearn.linear_model import Lasso
models = [
    Lasso(alpha=0.1),
]

model_strings = [
    "Lasso, regularización alpha = 0.1",

]

validate_models(model_strings, models, X_train, Y_train)
```

[15]:

	mean_squared_error	std_dev_error
Lasso, regularización alpha = 0.1	0.053907	0.007081

1.20 Selección del mejor modelo

Me he quedado con los hiperparámetros de cada modelo que mejor resultado han dado

[16]:
```
# Vuelvo a hacer output de los mejores hiperparámetros para
# cada modelo

models = [
    LinearRegression(),
    Ridge(alpha=1.5),
    SGDRegressor(alpha=0.0001, random_state=1),
    Lasso(alpha=0.1)
]

model_strings = [
    "OLS",
    "Ridge alpha = 1.5",
    "SGD, regularización L2 alpha = 0.0001",
    "Lasso, regularización alpha = 0.1"
]

validate_models(model_strings, models, X_train, Y_train)
```

[16]:

	mean_squared_error	std_dev_error
OLS	0.019191	0.002272
Ridge alpha = 1.5	0.018816	0.002246
SGD, regularización L2 alpha = 0.0001	0.019703	0.001945
Lasso, regularización alpha = 0.1	0.053907	0.007081

- Podemos observar que la regularización de Ridge mejora un poco los resultados de OLS.
- SGD se queda un poco más atrás y Lasso está directamente fuera de la competición (parece que quedarse con pocas features en este problema no es buena idea).
- La regularización l2 para disminuir el valor de los pesos mejora efectivamente el error fuera de la muestra (al menos con el estimador que tenemos de validación cruzada)

1.21 Utilizamos el conjunto de test

Ahora tenemos que entrenar al modelo ganador (Ridge con alpha 1.5) con todos los datos de training y probarlo con los datos de test.

[17]:
```
from sklearn.metrics import mean_squared_error

# Ridge con alpha = 1.5 es el modelo final

final_model = Ridge(alpha = 1.5)

# Lo entrenamos con el conjunto de training
```

```
final_model.fit(X_train, Y_train)

# Predecimos las etiquetas del conjunto de test

predictions = final_model.predict(X_test)

# Medimos el error de la predicción

etest = mean_squared_error(predictions, Y_test)

print("Etest (MSE): ", etest)
```

```
Etest (MSE):  0.01720165090244771
```

1.22 Comparando estimador de Eout (validación) con Etest

Consideramos el *overfitting* un problema, el ruído (determinístico y estocástico) la causa, la regularización y la validación, la cura. Con ambas hacemos un esfuerzo por reducir Eout, más que Ein.

El error que hemos obtenido en validación para el modelo ganador, debe ser un estimador no sesgado del error fuera de la muestra (si no, no valdría de nada elegir un modelo basándonos en esta métrica). En teoría, este error que hemos obtenido en validación, debería ser mayor o igual que el que obtenemos fuera de la muestra, ya que después de elegir el modelo, entrenamos con todos los datos antes de valorarlo con los datos de test.

En este caso, ha resultado que: Etest < Eval.

1.23 ¿Qué modelo propones y que error Eout tiene?

Como hemos comprobado, Ridge Regression con alpha = 1.5 ha sido el modelo que por validación cruzada mejor se comporta y es el que hemos utilizado para predecir datos del conjunto de test.

Gracias a la desigualdad de Hoeffding podemos estimar la cota de generalización con el conjunto test:

$E_{out}(g) <= E_{in}(g) + \sqrt{(\frac{1}{2N} ln \frac{2M}{\alpha})}$ Siendo α el nivel de tolerancia.

Como en este caso utilizamos E_{test} y el nivel de hipótesis es solamente una, podemos utilizarla así:

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

Con $M = 1$.

Haciendo las cuentas aquí abajo podemos afirmar con un 95% de confianza que E_{out} será menor o igual que 0.085.

[27]:
```
N = len(X_test)
alpha = 0.05

eout_bound = etest + np.sqrt((1 / (2*N)) * np.log(2 / alpha))
```

```
print('Eout(g) <= ', eout_bound)
```

```
Eout(g) <=  0.08519176751902854
```

2 Optical Recognition of Handwritten Digits

2.1 El problema

Tenemos un problema de clasificación con 10 clases. Los datos vienen de una universidad de Turquía en la que procesaron dígitos escritos a mano como mapas de bits.

2.2 El data set

Los mapas de bits tienen 32 x 32 bits. Por suerte, los datos no son 32 x 32 columnas de booleanos. En cambio, estos investigadores, dividieron todo el mapa de bits en 64 cuadros de 4x4 bits e hicieron de cada cuadro una columna. Por tanto, cada uno de esos atributos, contiene la cuenta de bits "true" que hay en ese cuadro, por lo tanto, están entre 0 y 16. Es así como consiguieron reducir las dimensiones del problema.

2.3 Elementos

- X: 64 columnas, cada una tiene un número entre 0 y 16 que representa la cuenta de bits activos en un cuadro del mapa.
- Y: La etiqueta tiene como rango 0-9 (que son todos los dígitos que contempla el sistema numérico decimal).
- Función objetivo: Dados los bits que contiene cada uno de los cuadros del mapa, predecir de que dígito se trata.

2.4 Archivo de datos

En este caso no tenemos ni si quiera que hacer nosotros la separación entre datos de entrenamiento y test. Tenemos dos archivos CSV sin cabecera:

- optdigits.tra: Datos de training.
- optdigits.tes: Datos de test.

2.5 Funciones a utilizar

De nuevo, empiezo con combinaciones lineales sin transformación ninguna. Creo que no hace falta añadir transformaciones no lineales (y he comprobado más tarde estar en lo cierto), además añadir complejidad innecesaria a nuestro modelo provocaría aumentar el error fuera de la muestra al sacrificar varianza por sesgo.

2.6 Datos perdidos

Según el fichero "optdigits.name", que da información sobre el data set, no hay datos perdidos. Vamos sin embargo a comprobar rápidamente que no hay datos perdidos y que hemos hecho correctamente la lectura.

[1]:
```
import pandas as pd
import numpy as np

import warnings
warnings.filterwarnings("ignore")

# Leemos los CSV con header=None para que no cuente la primera línea como␣
 ↪cabecera
raw_data_training = pd.read_csv('./datos/optdigits.tra', header=None)
raw_data_test = pd.read_csv('./datos/optdigits.tes', header=None)

# Miramos a ver si hay algún dato perdido
print('Hay datos nulos en training:',raw_data_training.isna().any().any())
print('Hay datos nulos en test: ', raw_data_test.isna().any().any())
```

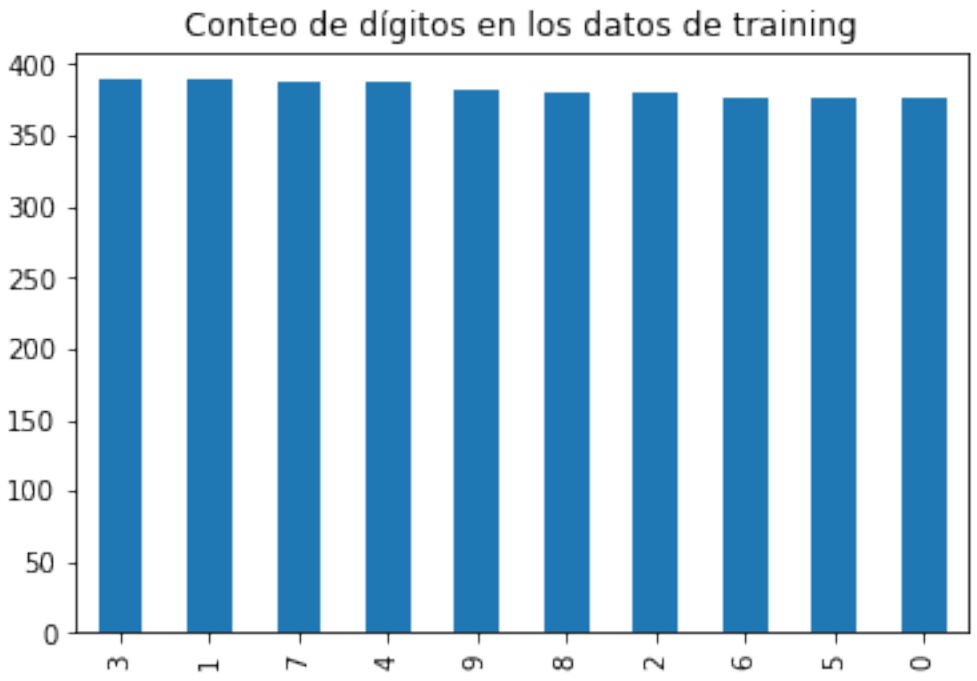
```
Hay datos nulos en training: False
Hay datos nulos en test:  False
```

2.7 Muestreo estratificado

Podemos ver que tanto los datos de training como los de entrenamiento están estratificados por dígitos. Es decir, hay un equilibrio de ocurrencias de cada dígito. De esta forma, no hay falta de información sobre ninguno de ellos.

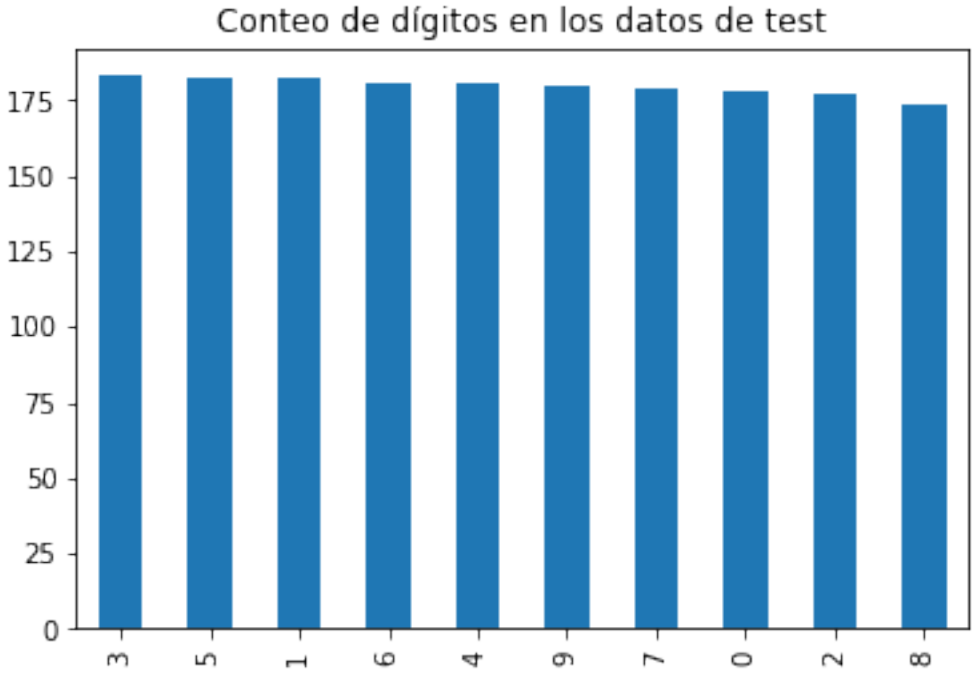
[2]:
```
# Imprimimos una representación gráfica del número de ocurrencias de cada dígito.

pd.value_counts(raw_data_training[64]).plot.bar(title="Conteo de dígitos en los␣
 ↪datos de training")
```



[3]:
```
# Imprimimos una representación gráfica del número de ocurrencias de cada dígito.

pd.value_counts(raw_data_test[64]).plot.bar(title="Conteo de dígitos en los␣
 ↪datos de test")
```



2.8 Métrica de ajuste

Hablaré en detalle de la función de pérdida de cada modelo que utilice. En cuanto a la que he considerado en validación y Etest, considero accuary (el porcentaje de aciertos) pragmática y representativa.

2.9 Validación cruzada

En este caso tenemos una validación cruzada un poco más sofisticada que en el caso de regresión:

- Utilizamos "StratifiedKFold" para que parta los datos de training en 10 partes estratificadas, con aproximadamente la misma cantidad de ocurrencias de cada dígito.
- Utilizamos pipeline para encadenar también un StandardScaler que estandariza el rango de las columnas a una distribución Gaussiana con media 0 y varianza 1. Aunque realmente ya estaban todas las columnas regularizadas a un rango 0-16, se considera que normalizarlo a 0-1 o a media 0 varianza 1 es beneficioso para los modelos (incluso requerido por algunos). z = (x - u) / s (siendo u la media de la columna y s la desviación tipica) para cada dato (x).
- Utilizamos como métrica la precisión (porcentaje de aciertos)

[5]:
```
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

# Utilizamos cross_val_score de sklearn que hace la validación cruzada por␣
 ↪nosotros.
# Devuelve el valor de cada validación (en este caso 10)

def validate_models(model_strings, models, X_train, Y_train):
    validation_results = {'accuracy':[], 'std_dev_error': []}
    cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
    for model in models:
        pipeline = make_pipeline(StandardScaler(), model)
        results = cross_val_score(pipeline, X_train, Y_train, scoring =␣
 ↪"accuracy", cv=cv)

        # Hacemos la media y desviación típica de las 10 validaciones

        validation_results['accuracy'].append(np.mean(np.abs(results)))
        validation_results['std_dev_error'].append(np.std(np.abs(results)))

        # Devolvemos una tabla con los modelos y sus métricas

    return pd.DataFrame(data=validation_results, index=model_strings)
```

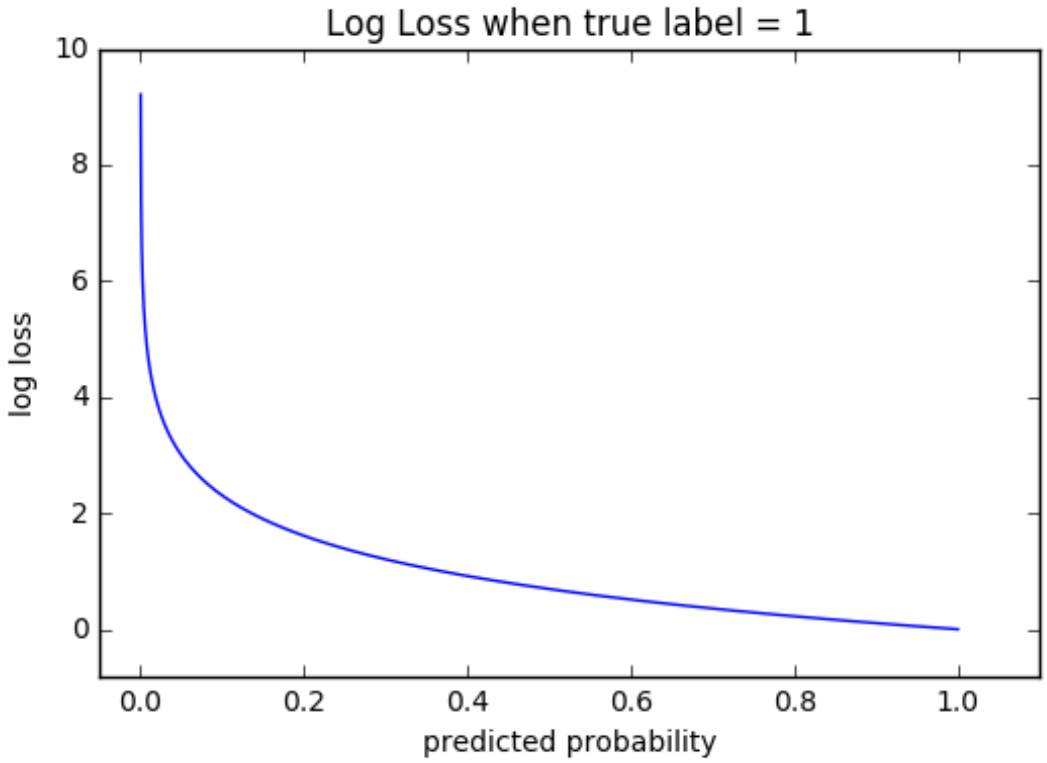
2.10 Regresión logística

- A diferencia de la regresión logística que hemos trabajado en otras prácticas, en esta contamos con regularización por normal l2, minimizando la siguiente función: $\min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$
- Además, controlamos la regularización con el parámetro C (valores más pequeños, mas regularización).
- Como función de pérdida, como el parámetro multi_class a multinomial, se utiliza cross-entropy.

Se calcula como $\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$, donde:

- M es el núero de clases (en este caso 10)
- Y: 0 o 1 dependiendo si c es la clasificación correcta del dato o
- p: Probabilidad predicha de que el dato o pertenezca a la clase c

También se denomina a veces perdida logarítmica por su curva:



[6]:
```
# Modelo de regresión logística

from sklearn.linear_model import LogisticRegression
models = [
    LogisticRegression(multi_class='multinomial', C=0.5),
    LogisticRegression(multi_class='multinomial', C=1.0),
    LogisticRegression(multi_class='multinomial', C=1.5),
]
```

```
model_strings = [
    'Logistic Regression C = 0.5',
    'Logistic Regression C = 1.0',
    'Logistic Regression C = 1.5',
]

validate_models(model_strings, models, X_train, Y_train)
```

[6]:

	accuracy	std_dev_error
Logistic Regression C = 0.5	0.973318	0.010844
Logistic Regression C = 1.0	0.974104	0.010693
Logistic Regression C = 1.5	0.973057	0.010668

2.11 Perceptrón

Un viejo conocido, en este caso también tiene opción de regularización. He probado, como se puede ver más abajo, con y sin ella (norma l2).

- Itera por todos los datos.
- Si está bien situado para el dato dado, no cambia.
- Si no lo está, se corrige.
- Si no cambia en una pasada completa o llega a las iteraciones máximas, para.

[7]:
```
# Modelo Perceptrón

from sklearn.linear_model import Perceptron
models = [
    Perceptron(random_state=1),
    Perceptron(alpha=0.0001, penalty='l2',random_state=1),
    Perceptron(alpha=0.00025, penalty='l2',random_state=1),
    Perceptron(alpha=0.0004, penalty='l2',random_state=1),

]

model_strings = [
    'Perceptron sin regularización',
    'Perceptron alpha = 0.0001',
    'Perceptron alpha = 0.00025',
    'Perceptron alpha = 0.0004',

]

validate_models(model_strings, models, X_train, Y_train)
```

[7]:

	accuracy	std_dev_error
Perceptron sin regularización	0.950039	0.008639

```
Perceptron alpha = 0.0001      0.938268       0.007688
Perceptron alpha = 0.00025     0.925184       0.016293
Perceptron alpha = 0.0004      0.916817       0.008671
```

2.12 Ridge Classifier

- Se optimiza la misma función que hemos visto para el problema de regresión.
- Se menciona en la documentación que puede ser más rápido para problemas multiclases (como este) en comparación regresión logística ya que solo se computa la matriz de proyección una vez. $(X^T X)^{-1} X^T$
- De nuevo, he probado con y sin regularización.
- Utilizo como algoritmo el de la descomposición en valores singulares, al igual que OLS.

[8]:
```
# Modelo Ridge Classifier

from sklearn.linear_model import RidgeClassifier
models = [
    RidgeClassifier(alpha=0, solver='svd'),
    RidgeClassifier(alpha=1.0, solver='svd')

]

model_strings = [
    'Ridge alpha = 0',
    'Ridge alpha = 1.0',
]

validate_models(model_strings, models, X_train, Y_train)
```

[8]:
```
                   accuracy  std_dev_error
Ridge alpha = 0    0.933298       0.014812
Ridge alpha = 1.0  0.933036       0.014051
```

2.13 Elección del mejor modelo

De nuevo, gracias a la validación podemos escoger (con seguridad de que no ha habido overfitting y de que Eval es un indicador no sesgado del error fuera de la muestra) el modelo que mejor resultados nos ha dado.

- Ha sido Regresión Logística con C=1.5 para regularizar.
- A diferencia del anterior problema, ahora si que le aplicamos un StandardScaler (igual que hemos hecho al validar) a los datos antes de entrenar con ellos.

[9]:
```
# Normalizamos los datos de training antes de usarlos

stdScaler = StandardScaler()
stdScaler.fit(X_train)
```

```
X_train_scaled = stdScaler.transform(X_train)

# Entrenamos con ellos

final_model = LogisticRegression(multi_class='multinomial', C=1.5)
final_model.fit(X_train_scaled, Y_train)
```

2.14 Usamos Etest para ver cómo de bueno es nuestro modelo final con datos nuevos.

- De nuevo normalizamos.
- Predecimos.
- Medimos el porcentaje de aciertos.

[10]:
```
from sklearn.metrics import accuracy_score

# Normalizamos los datos de test antes de usarlos.

stdScaler = StandardScaler()
stdScaler.fit(X_test)
X_test_scaled = stdScaler.transform(X_test)

# Predecimos


# Medimos el porcentaje de aciertos

predictions = final_model.predict(X_test_scaled)
accuracy_test = accuracy_score(Y_test, predictions)
print('Accuracy in test set: ', accuracy_test)
```

```
Accuracy in test set:  0.9510294936004452
```

2.15 Analizando la confusion-matrix

- El número más grande de confusiones se ha dado clasificando unos como ochos (ha ocurrido 9 veces)
- También es destacable la confusión de clasificar nueves como sietes y nueves como ochos (6 y 7 veces respectivamente)

[11]:
```
from sklearn.metrics import confusion_matrix

print(confusion_matrix(Y_test, predictions))
```

```
[[174   0   0   0   0   3   1   0   0   0]
 [  0 177   0   0   0   0   0   0   3   2]
 [  0   3 171   1   0   0   1   0   1   0]
 [  0   0   4 173   0   2   0   1   1   2]
 [  0   1   0   0 173   0   0   1   2   4]
```

```
[  0   0   1   1   0 176   1   0   0   3]
[  0   1   0   0   2   0 177   0   1   0]
[  0   0   0   0   2   5   0 165   0   7]
[  0   9   0   3   0   3   0   0 153   6]
[  0   3   0   0   2   3   0   0   2 170]]
```

2.16 Analizando classifcation report

- Acorde a lo que hemos visto en la confusion matrix, el nueve ha sido el dígito que peor se ha clasificado (solo se ha hecho bien el 88% de las veces).
- El cero no se ha clasificado mal ninguna vez.

[12]:
```
from sklearn.metrics import classification_report
print(classification_report(Y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99       178
           1       0.91      0.97      0.94       182
           2       0.97      0.97      0.97       177
           3       0.97      0.95      0.96       183
           4       0.97      0.96      0.96       181
           5       0.92      0.97      0.94       182
           6       0.98      0.98      0.98       181
           7       0.99      0.92      0.95       179
           8       0.94      0.88      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
```

2.17 Estimación de Eout

Ya he hablado del modelo que he elegido y hemos usado el conjunto test para probarlo con datos "nuevos". Vamos ahora a usar la misma cota que en el ejercicio de regresión: $E_{out}(g) <= E_{test}(g) + \sqrt{(\frac{1}{2N} ln \frac{2}{\alpha})}$

Para poder asegurar al 95% de confianza que Eout va a ser menor o igual a

[13]:
```
N = len(X_test)
alpha = 0.05
etest = 1 - accuracy_test

eout_bound = etest + np.sqrt((1 / (2*N)) * np.log(2 / alpha))

print('Eout(g) <= ', eout_bound)
```

```
Eout(g) <=  0.08100797513423191
```