TP1 Reconocimiento de Patrones

Corvalan César, Junqueras Juan, Suárez Juan Carlos

Mayo 2021

Introducción

En el presente trabajo práctico se comparan diferentes modelos de regresión lineal, que buscan predecir el valor medio de una casa en el estado de California a partir de otros atributos incluídos en un mismo dataset.

1. Marco teórico

La regresión lineal múltiple es un método estadístico que trata de modelar la relación entre una variable continua y dos o más variables independientes mediante el ajuste de una ecuación lineal. Tres de las limitaciones que que aparecen en la práctica al tratar de emplear este tipo de modelos (ajustados por mínimos cuadrados ordinarios) son:

- Se ven perjudicados por la incorporación de predictores correlacionados.
- No realizan selección de predictores, todos los predictores se incorporan en el modelo aunque no aporten información relevante. Esto suele complicar la interpretación del modelo y reducir su capacidad predictiva. Existen otros modelos como random forest o gradient boosting que sí son capaces de seleccionar predictores.
- No pueden ajustarse cuando el número de predictores es superior al número de observaciones.

Algunas de las estrategias que se pueden aplicar para atenuar el impacto de estos problemas son:

- Ridge Regression. Penaliza la suma de los coeficientes elevados al cuadrado ($\|\beta\|_2^2 \sum_{j=1}^{p} \beta_j^2$). Esta penalización tiene como objetivo reducir de forma proporcional el valor de todos los coeficientes del modelo pero sin que estos lleguen a cero. El grado de penalización está controlado por el hiperparámetro λ. Cuando $\lambda = 0$, la penalización es nula y el resultado es equivalente al de un modelo lineal por mínimos cuadrados ordinarios (OLS). A medida que λ aumenta, mayor es la penalización y menor el valor de los predictores.

$$\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

- LASSO. Penaliza la suma de los valores absolutos de los coeficientes de regresión ($\|\beta\|_1 \sum_{j=1}^{p} |\beta_j|$).Esta penalización tiene como objetivo forzar que los coeficientes de los predictores tiendan a cero. Dado que un predictor con coeficiente de regresión cero no influye en el modelo, Lasso consigue excluir los predictores menos relevantes. Al igual que en Ridge, el grado de penalización está controlado por el hiperparámetro λ. Cuando $\lambda = 0$, el resultado es equivalente al de un modelo lineal por mínimos cuadrados ordinarios. A medida que λ aumenta, mayor es la penalización y más predictores quedan excluidos.

$$\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

- Elastic Net. Penaliza mediante la combinación de las dos anteriores ($\alpha\lambda|\beta\|_1 + (1-\alpha)\|\beta\|_2^2$)

El grado en que influye cada una de las penalizaciones está controlado por el hiperparámetro α. Su valor está comprendido en el intervalo [0,1]. Cuando α=0, se aplica Ridge y cuando =1 se aplica Lasso. La combinación de ambas penalizaciones suele dar lugar a buenos resultados. Una estrategia frecuentemente utilizada es asignarle casi todo el peso a la penalización utilizada por Ridge (α muy próximo a 1) para conseguir seleccionar predictores y un poco a la utilizada por Lasso para dar cierta estabilidad en el caso de que algunos predictores estén correlacionados.

$$\frac{\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij})^2}{2n} + \lambda(\alpha\sum_{j=1}^{p}|\beta_j| + \frac{(1-\alpha)}{2}\sum_{j=1}^{p}\beta_j^2)$$

1.1. Descripción del problema

El problema a resolver consiste en utilizar los métodos de Ridge Regression, LASSO y Elastic Net, para predecir el valor promedio por distrito de los precios de las propiedades en California, a partir del dataset de California Housing. Adicionalmente, decidir mediante cross validation cuál método y con cuáles valores de hiperparámetros se obtienen los mejores resultados.

Dado que es un problema de regresion, se usará una de las metricas mas comunes. El root mean squared error. Si el target es $\vec{t} = (t_1, t_2, ..., t_N)^T$ y las predicciones son $\vec{y} = (y_1, y_2, ..., y_N)^T$, entonces:

$$RMSE(\vec{t}, \vec{y}) = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(t_n - y_n)^2} \tag{1}$$

Con el RMSE se espera dar un error esperado de la predicción.

2. Desarrollo

Uno de los primeros pasos a la hora de realizar un proyecto que involucre el análisis de datos es explorar y visualizar los datos. El objetivo principal es obtener información sobre el contenido de los datos, ayudar a enmarcar las preguntas que haremos y detectar posibles vías para avanzar en las respuestas a estas preguntas.

2.1. Análisis del dataset

Se trabajó con un conjunto de datos clásico para Machine Learning,California Houisng [2], que consiste en datos inmobiliarios para distritos en el estado de California, EE.UU. Se usó una versión ligeramente modificada, preparada por Aurélien Géron[1].

Con el método $info()$ se permite obtener información útil (una descripción rápida) de los datos, en particular el número total de filas, el tipo de cada atributo y el número de valores no nulos, cómo se puede apreciar en la Figura 1.

```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object 
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Figura 1: Housing info

Se puede ver que hay 20640 entradas, con 10 columnas cada una. A excepción de *ocean_proximity*, todas las columnas son números ($float64$). Además, faltan algunas entradas en la columna *total_bedrooms*. Cada una de estas filas corresponde a un distrito de California.

Todos los atributos son numéricos, excepto el campo *ocean_proximity*. Para averiguar qué categorías existen en ese campo y cuántos distritos pertenecen a cada categoría se utilizó el método *value_counts* (ver Figura 2).

```
housing["ocean_proximity"].value_counts()
```

```
<1H OCEAN     7276
INLAND        5263
NEAR OCEAN    2124
NEAR BAY      1847
ISLAND           2
Name: ocean_proximity, dtype: int64
```

Figura 2: Método *value_counts* aplicado a *housing*

De esa manera, se permite averiguar si se trata de un atributo de tipo categórico.
Por otro lado, los histogramas de la Figura 3 muestran:

1. El atributo *median_income* no está expresado en dólares. Los datos fueron preprocesados, escalado y limitado a 15 (en realidad, 15.0001) para los ingresos medios más altos, y a 0,5 (en realidad, 0,4999) para los ingresos medios más bajos. Los números representan aproximadamente decenas de miles de dólares (por ejemplo, 3 en realidad significa alrededor de30 mil dólares).
2. Los valores de *median_age* y de *median_house* median también fueron escalados, con diferentes escalas.
3. Los histogramas se extienden mucho más a la derecha de la mediana que a la izquierda. Esto puede hacer que sea un poco más difícil para algunos algoritmos de aprendizaje automático

detectar patrones. Según [1] en el capítulo 2, recomienda transformar estos atributos para tener más distribuciones en forma de campana.

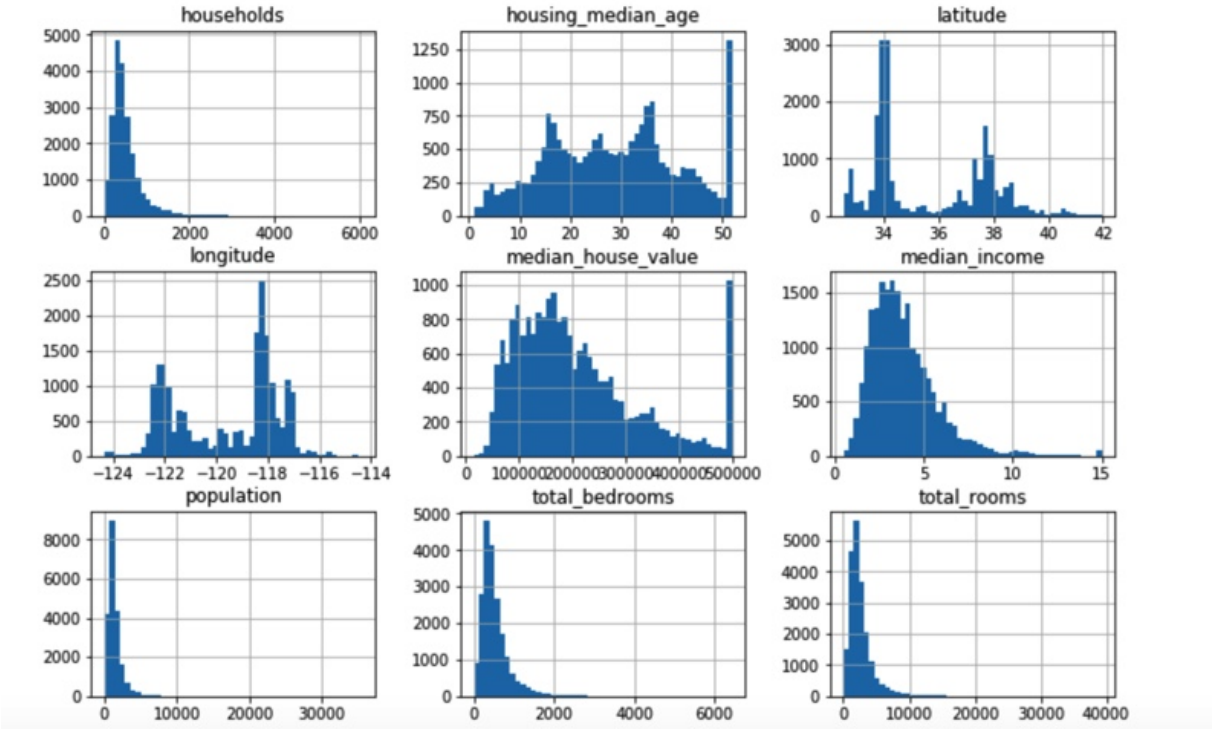

Figura 3: Histograma de cada atributo numérico de *housing*.

Visualización de datos geográficos

Dado que existe información geográfica (latitud y longitud), es una buena idea crear un diagrama de dispersión y densidad poblacional de todos los distritos para visualizar los datos.

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4a22a8e50>
```

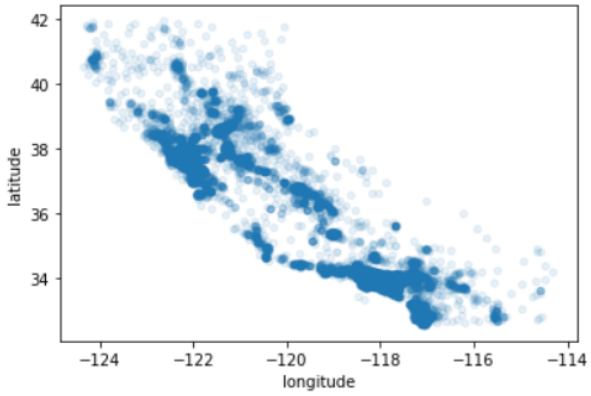

Figura 4: Dispersión geográfica de los datos

La dispersión de los datos nos genera una silueta similar a la geografía de California en donde los puntos más oscuros nos indican aquellas regiones más densamente pobladas.

Ahora observemos los precios de las viviendas (Figura 5). El radio de cada círculo representa la población del distrito (opción s) y el color representa el precio (opción c). Nosotros utilizaremos un mapa de color predefinido (opción `cmap`) llamado `jet`, que va desde el azul (valores bajos) a rojo (precios altos):

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
s=housing["population"]/100, label="population", figsize=(10,7),
c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7ff4a252a650>
```

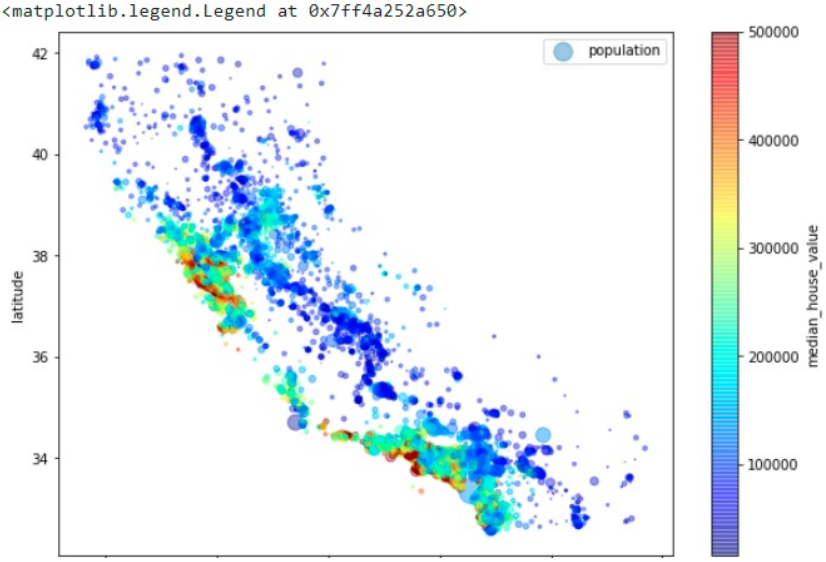

Figura 5: Precios de las casas

Correlación de datos

Ahora veamos cuánto se correlaciona cada atributo con el valor medio de las viviendas:

```
corr_matrix = housing.corr()
```

```
corr_matrix["median_house_value"].sort_values(ascending=False)
```

```
median_house_value    1.000000
median_income         0.687160
total_rooms           0.135097
housing_median_age    0.114110
households            0.064506
total_bedrooms        0.047689
population           -0.026920
longitude            -0.047432
latitude             -0.142724
Name: median_house_value, dtype: float64
```

Figura 6: correlación de cada atributo con el valor medio de la vivienda

Otra forma de comprobar la correlación entre atributos es utilizar la función `scatter_matrix` de la librería `Pandas`, que traza cada atributo numérico contra todos los demás atributos numéricos. Como ahora hay 11 atributos numéricos, obtendría $11^2 = 121$ parcelas, que no caben en una página, así que

centrémonos en algunos prometedores atributos que parecen estar más correlacionados con el valor medio de las viviendas:

```
from pandas.plotting import scatter_matrix
attributes = ["median_house_value", "median_income", "total_rooms",
"housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7ff4a23eac50>]],
      dtype=object)
```

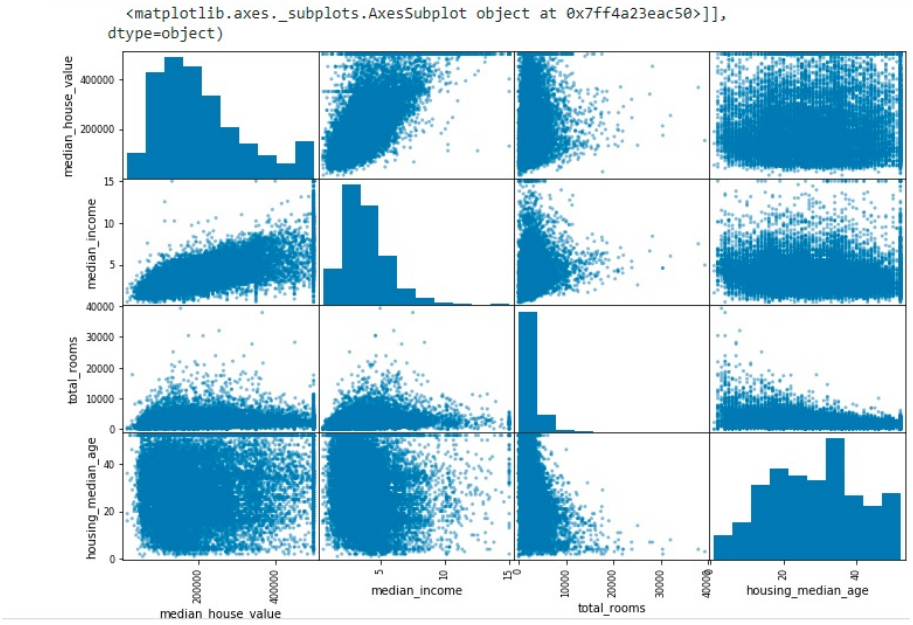

Figura 7: `scatter_matrix` del valor medio de la vivienda

Analizando los gráficos de la figura 7 El atributo más prometedor para predecir el valor medio de la vivienda parece ser el ingreso medio, así que ampliemos su diagrama de dispersión de correlación.

```
housing.plot(kind="scatter", x="median_income", y="median_house_value", alpha=0.1)
```

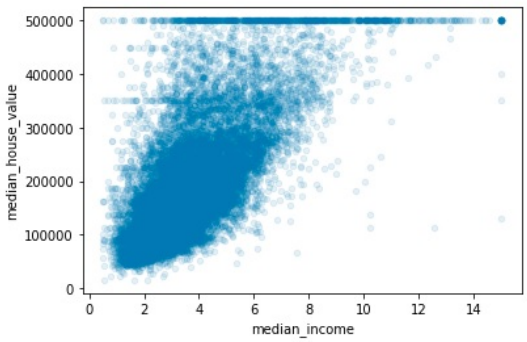

Figura 8: Correlación Ingresos medios/Valor medio de la vivienda

En la figura 8 podemos apreciar que la correlación entre ambos atributos es muy fuerte. Por un lado, se puede ver claramente la tendencia alcista y que los puntos no están demasiado dispersos. Y por otro lado, se puede apreciar una línea horizontal en el valor $ 500.000 y otras líneas rectas menos obvias: una línea alrededor del valor $ 350.000, quizás una alrededor del valor $ 280.000 y algunas más por debajo de eso. Es posible que desee intentar eliminar los distritos correspondientes a esos valores para evitar que estos datos afecten a los algoritmos de machine learning.

Experimentar combinaciones de atributos

Una última cosa que podemos hacer antes de preparar los datos para los algoritmos de machine learning consiste en probar varias combinaciones de atributos. Por ejemplo, el número total de habitaciones en un distrito no es muy útil si no se sabe cuántas hogares hay, sobre todo si se pretende saber la cantidad de habitaciones por hogar. Del mismo modo, el número total de dormitorios por sí solo no es muy útil: probablemente quiere compararlo con el número de habitaciones. Y la población por hogar también parece una combinación de atributos interesante. Generemos estos nuevos atributos.

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]
```

Y ahora veamos la matriz de correlación:

```
corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

```
median_house_value          1.000000
median_income               0.687160
rooms_per_household         0.146285
total_rooms                 0.135097
housing_median_age          0.114110
households                  0.064506
total_bedrooms              0.047689
population_per_household   -0.021985
population                 -0.026920
longitude                  -0.047432
latitude                   -0.142724
bedrooms_per_room          -0.259984
Name: median_house_value, dtype: float64
```

Preparar los datos para los algoritmos de Machine Learning

Antes de invocar a los algoritmos de machine learning es necesario preparar los datos para:

- Poder reproducir estas transformaciones fácilmente en cualquier conjunto de datos (p. Ej., la próxima vez que obtenga un conjunto de datos nuevo).
- Construir gradualmente una biblioteca de funciones de transformación que se podrá reutilizar en proyectos futuros.
- Probar fácilmente varias transformaciones y ver qué combinación de transformaciones funciona mejor

Pero primero volvamos a un conjunto de entrenamiento limpio (copiando `strat_train_set`), y separemos los predictores y las etiquetas, ya que no necesariamente queremos aplicar las mismas transformaciones a los predictores y los valores objetivo (tenga en cuenta que `drop ()` crea una copia de los datos y no afecta a `strat_train_set`):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Limpieza de datos

La mayoría de los algoritmos de machine learning no funcionan si faltan valores en algunos atributos o si están en un formato no numérico, así que vamos a crear algunas funciones para evitar estos problemas. Por ejemplo al atributo `total_bedrooms` le faltan algunos valores, para solucionar esto tenemos tres opciones:

- Eliminar los distritos correspondientes.
- Eliminar todo el atributo.
- asignarles algún valor (cero, la media, la mediana, etc.).

Para esto usamos los métodos `dropna ()`, `drop ()` y `fillna ()`:

```
housing.dropna(subset=["total_bedrooms"])    # option 1
housing.drop("total_bedrooms", axis=1)       # option 2
median = housing["total_bedrooms"].median()  # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

La librería `Scikit-Learn` proporciona una clase útil para ocuparse de los valores faltantes: `SimpleImputer`. Primero, necesita crear una instancia de `SimpleImputer`, especificando que desea reemplazar los valores nulos de cada atributo con la mediana de ese atributo:

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
```

Dado que la mediana solo se puede calcular en atributos numéricos, necesitamos crear una copia de los datos sin el atributo de texto `ocean_proximity`:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

Aplicamos `fit ()`:

```
imputer.fit(housing_num)
```

```
SimpleImputer(add_indicator=False, copy=True, fill_value=None,
              missing_values=nan, strategy='median', verbose=0)
```

`imputer` simplemente calculó la mediana de cada atributo y almacenó el resultado en su variable de instancia `statistics_`. Solo el atributo `total_bedrooms` poseía registros con valores nulos.

```
imputer.statistics_
```

```
array([-118.51  ,   34.26  ,   29.    , 2119.5   ,  433.    , 1164.    ,
        408.    ,    3.5409])
```

```
[ ] housing_num.median().values
```

```
array([-118.51  ,   34.26  ,   29.    , 2119.5   ,  433.    , 1164.    ,
        408.    ,    3.5409])
```

Ahora puede utilizar este `imputer` "entrenado"para transformar el conjunto de entrenamiento reemplazando valores nulos por las medianas aprendidas:

```
X = imputer.transform(housing_num)
```

Manejo de texto y atributos categóricos

Anteriormente omitimos el atributo `ocean_proximity` porque sus valores están expresados en texto y por ende no podemos calcular su mediana:

```
[ ]  housing_cat = housing[["ocean_proximity"]]
     housing_cat.head(10)
```

	ocean_proximity
17606	<1H OCEAN
18632	<1H OCEAN
14650	NEAR OCEAN
3230	INLAND
3555	<1H OCEAN
19480	INLAND
8879	<1H OCEAN
13685	INLAND
4937	<1H OCEAN
4861	<1H OCEAN

La mayoría de los algoritmos de machine learning prefieren trabajar con números, por eso convierte estas categorías de texto a números. Para ello, podemos utilizar la clase `OrdinalEncoder` de la librería `Scikit-Learn`

```
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
housing_cat_encoded[:10]
```

```
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

Puede obtener la lista de categorías utilizando la variable de instancia `categorías_`. Es una lista que contiene una matriz 1D de categorías para cada atributo categórico (en este caso, una lista que contiene una sola matriz ya que solo hay un atributo categórico):

```
ordinal_encoder.categories_
```

```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

Un problema con esta representación es que los algoritmos de machine learning pueden asumir que

dos valores son más similares que otros dos valores distantes. Esto puede estar bien en algunos casos (p. Ej., para categorías ordenadas como “malo”, “promedio”, “bueno”, “excelente”), pero no para el caso de la columna `ocean_proximity` (por ejemplo, las categorías 0 y 4 son claramente más similar que las categorías 0 y 1). Para solucionar este problema, una solución común es crear un atributo binario por categoría: un atributo igual a 1 cuando la categoría es “<1H OCEAN”(y 0 en caso contrario), otro atributo igual a 1 cuando la categoría es “INLAND” (y 0 en caso contrario), y así sucesivamente. Esto se llama one-hot encoding, porque solo un atributo será igual a 1 (caliente), mientras que los otros serán 0 (frío). Los nuevos atributos a veces se denominan atributos dummy. `Scikit-Learn` proporciona un método llamado `OneHotEncoder` para convertir valores categóricos en vectores one-hot:

```
[ ]  from sklearn.preprocessing import OneHotEncoder
     cat_encoder = OneHotEncoder()
     housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
     housing_cat_1hot

     <16512x5 sparse matrix of type '<class 'numpy.float64'>'
             with 16512 stored elements in Compressed Sparse Row format>
```

Pipelines de transformación

Agregamos este apartado simplemente a modo informativo. Comentado que afortunadamente, `Scikit-Learn` proporciona la clase `Pipeline` para ayudar a generar una secuencias de transformaciones. Aquí vemos un pequeño pipeline para atributos numéricos:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

El constructor `Pipeline` toma una lista de pares de “nombre/estimador”que definen una secuencia de pasos. Todos menos el último estimador deben ser transformadores (es decir, deben tener un método `fit_transform ()`).

Para manejar las columnas categóricas y las columnas numéricas por separado se utilizó el método transformador `ColumnTransformer` aplicando las transformaciones apropiadas a cada columna:

```
[ ]  from sklearn.compose import ColumnTransformer

     num_attribs = list(housing_num)
     cat_attribs = ["ocean_proximity"]

     full_pipeline = ColumnTransformer([
             ("num", num_pipeline, num_attribs),
             ("cat", OneHotEncoder(), cat_attribs),
         ])

     housing_prepared = full_pipeline.fit_transform(housing)
```

En el caso anteriror, se especificó que las columnas numéricas se transformaron usando el `num_pipeline` definido anteriormente, y las columnas categóricas deben transformarse usando un `OneHotEncoder`. Finalmente, aplicando este `ColumnTransformer` a los datos de `housing_num`: se aplica cada transformador a las columnas apropiadas y concatena las salidas a lo largo del segundo eje (los transformadores deben devolver el mismo número de filas).

2.2. Entrenando los modelos

En el apartado anterior nos encargamos de realizar el análisis y la transformación necesaria de los datos para poder aplicar correctamente los algoritmos de regresión que mencionamos en el marco teórico. Recordemos que el objetivo principal es predecir con el menor error posible el valor medio de las propiedades que figuran en el data set. Para ello es necesario realizar todos los pasos que se enumeran a continuación:

- Cargar el dataset.
- Dividir en Train y Test (en 80/20).
- Entrenar el modelo de Regresión con los datos de Train (Cross-Validation) con distintos hiperparámetros
- Fijar parámetrios, realizar `predict` sobre el Conjunto de Test.

En el apartado anterior fijamos el data set y lo dividimos en dos, un set de training con un 80 % de los datos y un set de testing con el 20 % restante.

Antes de empezar con el entrenamiento de los distintos métodos vamos a describir el funcionamiento de una herramienta que nos provee Scikit-Learn llamado `GridSearchCV`, la cual nos facilita el fine-tune y la elección de los mejores hiperparámetros para cada método. Se le debe indicar con qué hiperparámetros se desea experimentar y con qué valores probar, y evaluará todas las posibles combinaciones de valores de hiperparámetros utilizando cross-validation. Por ejemplo, el siguiente código busca la mejor combinación de valores de hiperparámetros para `RandomForestRegressor`:

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
param_grid = [
        {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
        {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
    ]
forest_reg = RandomForestRegressor()
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
scoring='neg_mean_squared_error',
return_train_score=True)
grid_search.fit(housing_prepared, housing_labels)
```

`GridSearchCV` recibe el método de regresión y el vector de hiperparámetros entre otros. `Param_grid` le dice a `Scikit-Learn` que primero evalúe todas las combinaciones $3 \times 4 = 12$ valores de hiperparámetros especificados en `n_estimators` y `max_features` de la primer linea, y luego intente todas las combinaciones $2 \times 3 = 6$ de valores de hiperparámetros en la segunda linea, pero esta vez con el hiperparámetro `bootstrap` establecido en False en lugar de True (que es el valor predeterminado para este hiperparámetro). Con todo, la búsqueda de la cuadrícula explorará $12 + 6 = 18$ combinaciones de valores de hiperparámetros para `RandomForestRe gressor` , y entrenará cada modelo cinco veces (ya que estamos utilizando validación cruzada de cinco veces). En otras palabras, en total, habrá $18 \times 5 = 90$ rondas de entrenamiento. Y nos retornará aquella combinación de hiperparámetros que minimicen una condición particular, en este caso `scoring=neg_mean_squared_error`.

3. Resultados

Ridge.

```
[ ] from sklearn.model_selection import GridSearchCV
    param_grid_ridge = [
            {'alpha': [0.1, 0.5, 1.0, 5.0, 10.0, 20.0], 'fit_intercept':[True, False], 'normalize':[True, False],
             'copy_X':[True, False], 'max_iter': [10000],
             'solver':["auto", "svd", "cholesky", "lsqr", "sparse_cg", "sag", "saga"]}
        ]
    ridge_reg = Ridge()
    grid_search_ridge = GridSearchCV(ridge_reg, param_grid_ridge, cv=5,
    scoring='neg_mean_squared_error',
    return_train_score=True)
    grid_search_ridge.fit(housing_prepared, housing_labels)
```

```
GridSearchCV(cv=5, error_score=nan,
             estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True,
                             max_iter=None, normalize=False, random_state=None,
                             solver='auto', tol=0.001),
             iid='deprecated', n_jobs=None,
             param_grid=[{'alpha': [0.1, 0.5, 1.0, 5.0, 10.0, 20.0],
                          'copy_X': [True, False],
                          'fit_intercept': [True, False], 'max_iter': [10000],
                          'normalize': [True, False],
                          'solver': ['auto', 'svd', 'cholesky', 'lsqr',
                                     'sparse_cg', 'sag', 'saga']}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='neg_mean_squared_error', verbose=0)
```

Hiperparámetros que minimizan el error en Ridge.

```
[ ] grid_search_ridge.best_params_
```

```
{'alpha': 10.0,
 'copy_X': True,
 'fit_intercept': True,
 'max_iter': 10000,
 'normalize': False,
 'solver': 'saga'}
```

Error cuadrático medio:

```
67083.49092405377
```

Lasso.

```
from sklearn.model_selection import GridSearchCV
param_grid_lasso = [
        {'alpha': [0.5, 1], 'fit_intercept':[True, False], 'normalize':[True, False],
         'copy_X':[True, False], 'max_iter': [10000], 'tol':[1e-3,1e-4],
         'warm_start':[True, False], 'positive':[True, False], 'random_state':[3],
         'selection':['cyclic', 'random']}
    ]
lasso_reg = Lasso()
grid_search_lasso = GridSearchCV(lasso_reg, param_grid_lasso, cv=5,
scoring='neg_mean_squared_error',
return_train_score=True)
grid_search_lasso.fit(housing_prepared, housing_labels)
```

```
GridSearchCV(cv=5, error_score=nan,
             estimator=Lasso(alpha=1.0, copy_X=True, fit_intercept=True,
                             max_iter=1000, normalize=False, positive=False,
                             precompute=False, random_state=None,
                             selection='cyclic', tol=0.0001, warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'alpha': [0.5, 1], 'copy_X': [True, False],
                          'fit_intercept': [True, False], 'max_iter': [10000],
                          'normalize': [True, False], 'positive': [True, False],
                          'random_state': [3],
                          'selection': ['cyclic', 'random'],
                          'tol': [0.001, 0.0001],
                          'warm_start': [True, False]}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='neg_mean_squared_error', verbose=0)
```

Hiperparámetros que minimizan el error en Lasso.

```
[ ]  grid_search_lasso.best_params_
```

```
{'alpha': 1,
 'copy_X': True,
 'fit_intercept': True,
 'max_iter': 10000,
 'normalize': True,
 'positive': False,
 'random_state': 3,
 'selection': 'cyclic',
 'tol': 0.0001,
 'warm_start': True}
```

Error cuadrático medio:

```
66922.00371026425
```

Elastic_net.

```
[ ] from sklearn.model_selection import GridSearchCV
    param_grid_elastic_net = [
           {'alpha': [0.1, 0.5, 1], 'l1_ratio': [0.1, 0.25, 0.5, 0.75, 0.99], 'fit_intercept':[True, False],
            'normalize':[True, False], 'copy_X':[True, False], 'max_iter': [10000], 'tol':[1e-3,1e-4],
            'warm_start':[True, False], 'positive':[True, False], 'random_state':[12344],
            'selection':['cyclic', 'random']}
       ]
    elastic_net = ElasticNet()
    grid_search_elastic_net = GridSearchCV(elastic_net, param_grid_elastic_net, cv=5,
    scoring='neg_mean_squared_error',
    return_train_score=True)
    grid_search_elastic_net.fit(housing_prepared, housing_labels)
```

```
GridSearchCV(cv=5, error_score=nan,
             estimator=ElasticNet(alpha=1.0, copy_X=True, fit_intercept=True,
                                  l1_ratio=0.5, max_iter=1000, normalize=False,
                                  positive=False, precompute=False,
                                  random_state=None, selection='cyclic',
                                  tol=0.0001, warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'alpha': [0.1, 0.5, 1], 'copy_X': [True, False],
                          'fit_intercept': [True, False],
                          'l1_ratio': [0.1, 0.25, 0.5, 0.75, 0.99],
                          'max_iter': [10000], 'normalize': [True, False],
                          'positive': [True, False], 'random_state': [12344],
                          'selection': ['cyclic', 'random'],
```

Hiperparámetros que minimizan el error en Elastic_net.

```
[ ] grid_search_elastic_net.best_params_
```

```
{'alpha': 0.5,
 'copy_X': True,
 'fit_intercept': True,
 'l1_ratio': 0.99,
 'max_iter': 10000,
 'normalize': False,
 'positive': False,
 'random_state': 12344,
 'selection': 'random',
 'tol': 0.0001,
 'warm_start': True}
```

Error cuadrático medio:

```
67096.0579433548
```

4. Conclusiones

Basándonos en los resultados podemos afirmar que la combinación de parámetros que involucran al método `Lasso` fueron los que mejor predijeron el valor medio de las casas californianas. Sin embargo, la diferencia sobre los otros métodos no es significativa, entendemos que esto se debe a los parámetros elegidos en cada caso, es decir, para este dataset, es posible utilizar cualquiera de los tres métodos siempre y cuando se lo invoque con los parámetros adecuados.

Referencias

[1] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019. ISBN: 9781492032618. URL: `https://github.com/ageron/handson-ml2/tree/master/datasets/housing`.

[2] M Meyer y P Vlachos. *Statlib: Data, software and news from the statistics community, 2009*. 2009. URL: `http://lib.stat.cmu.edu/`.