Clase14 IMA539

Mg. Alejandro Ferreira Vergara

May 15, 2023

1 Análisis de Regresión

Los modelos de regresión se utilizan para predecir variables objetivo del tipo continuas. Algunas aplicaciones son:

- Comprensión de las relaciones entre variables
- Comprensión de tendencias
- Predicción de probabilidades

1.1 Regresión Lineal Simple

El objetivo de la regresión lineal simple (univariante) es modelar la relación entre una única característica (variable explicativa x) y una respuesta de valor continuo (variable objetivo y).

La ecuación de un modelo lineal con una variable explicativa se define como:

$$y = w_0 + w_1 x$$

w_0 : representa el intercepto con el eje Y.

w_1 : coeficiente de peso de la variable explicativa x.

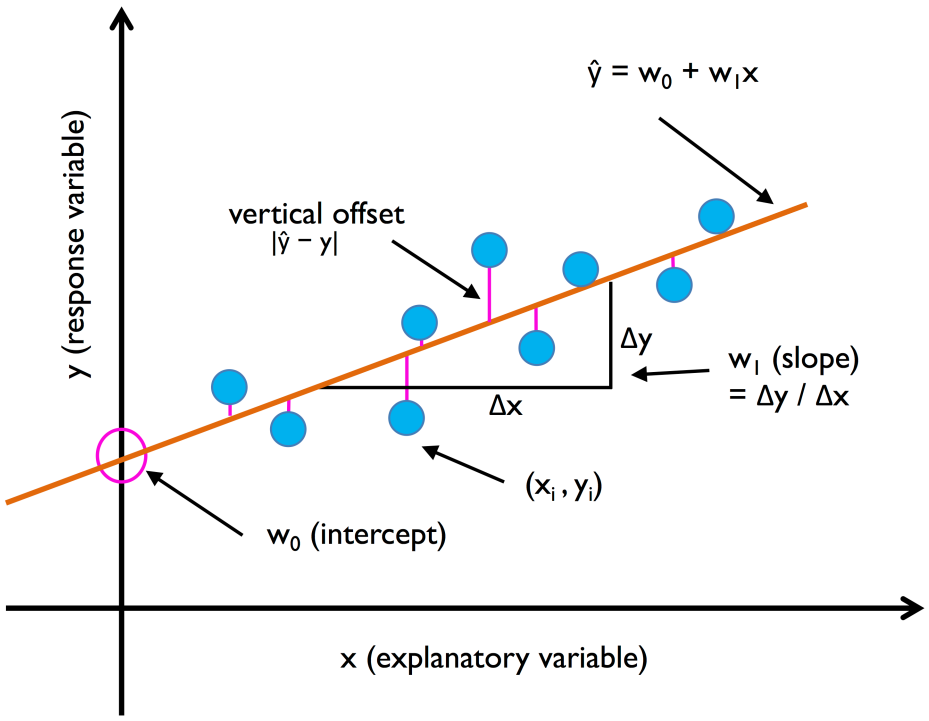
Nuestro objetivo es aprender los pesos de la ecuación lineal para describir la relación entre la variable explicativa y la variable objetivo.

Luego, esto puede utilizarse para predecir las respuestas de nuevas variables explicativas que no formaban parte del conjunto de datos de entrenamiento.

[1]:
```
from IPython.display import Image

Image(filename=r'clase14/14_1.png', width=500)
```

[1]:

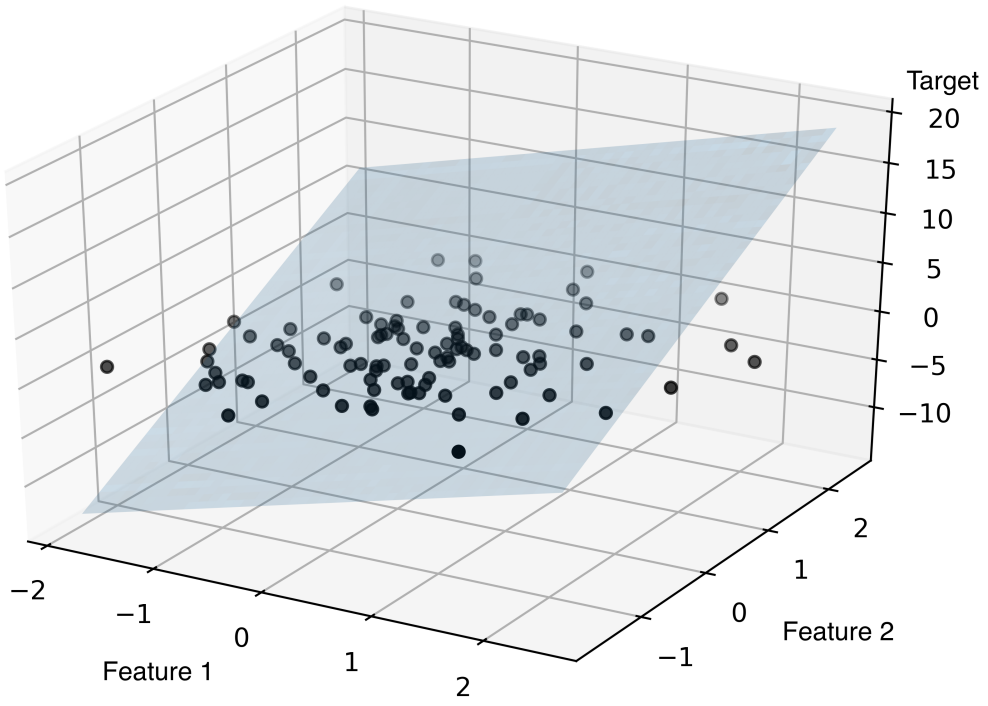


1.2 Regresión Lineal Múltiple

Generaliza la idea de la regresión lineal simple, mendiate la siguiente ecuación:

$$y = w_0x_0 + w_1x_1 + w_2x_2 + \cdots + w_mx_m = \sum_{i=0}^{m} w_ix_i = w^Tx; \quad x_0 = 1$$

[2]:
```
Image(filename=r'clase14/14_2.png', width=500)
```
[2]:



1.3 Base de datos Housing

[ ]:
```
import pandas as pd

df = pd.read_csv('https://raw.githubusercontent.com/rasbt/
 ↪python-machine-learning-book-2nd-edition'
                 '/master/code/ch10/housing.data.txt', header=None, sep='\s+')

df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',␣
 ↪'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
df.head(10)
```

- CRIM: Índice de criminalidad per cápita por barrio
- ZN: Proporción de suelo residencial con lotes de más de 25.000 pies cuadrados
- INDUS: Proporción de acres comerciales no minoristas por barrio
- CHAS: Variable ficticia del río Charles (= 1 si la zona colinda con el río; 0 en caso contrario)
- NOX: Concentración de óxido nítrico (partes por 10 millones)
- RM: Número promedio de habitaciones por vivienda
- AGE: Proporción de unidades ocupadas por sus propietarios construidas antes de 1940
- DIS: Distancias ponderadas a cinco centros de empleo de Boston
- RAD: Índice de accesibilidad a las autopistas radiales (categórica)
- TAX: Tasa del impuesto sobre la propiedad por cada 10.000 dólares
- PTRATIO: Tasa de alumnos por profesor por barrio

- B=1000(Bk - 0,63)^2: donde Bk es la proporción de personas de origen afroamericano por barrio
- LSTAT: Porcentaje de la población de menor estatus
- MEDV: Valor medio de las viviendas ocupadas por sus propietarios en miles de dólares

[ ]:
```
df.info()
```

[ ]:
```
df.CHAS = df.CHAS.astype('category')
df.RAD = df.RAD.astype('category')

df.describe(include="all")
```

[ ]:
```
df['RAD'].value_counts()
```

1.4 Análisis Exploratorio sobre las variables contínuas

- **Observar cómo se distribuyen los datos.**
- **Observar relación (directo, inversa, etc) de las variables.**
- **Observar datos atípicos.**

[ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns

cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']

sns.pairplot(df[cols], size=1.5)
plt.tight_layout()
# plt.savefig('images/10_03.png', dpi=300)
plt.show()
```

1.5 Matriz de correlación

Intuitivamente, podemos interpretar la matriz de correlación como una versión reescalada de la matriz de covarianza (la matriz de correlación es idéntica a una matriz de covarianza calculada a partir de características estandarizadas).

Coeficiente de Correlación de Pearson:

$$r = \frac{\sum_{i=1}^{n} \left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)}{\sqrt{\sum_{i=1}^{n} \left(x^{(i)} - \mu_x\right)^2} \sqrt{\sum_{i=1}^{n} \left(y^{(i)} - \mu_y\right)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

μ_x : media de la característica x

μ_y : media de la característica y

σ_{xy} : covarianza entre las características x e y

σ_x : desviación estándar de la característica x

σ_y : desviación estándar de la característica y

[ ]:
```
import numpy as np

cm = np.corrcoef(df[cols].values.T)

#sns.set(font_scale=1.5)
hm = sns.heatmap(cm,cbar=True,annot=True,square=True,fmt='.
 ↪2f',annot_kws={'size': 15},yticklabels=cols,xticklabels=cols)

plt.tight_layout()
# plt.savefig('images/10_04.png', dpi=300)
plt.show()
```

- Los coeficientes de correlación están en el rango de -1 a 1.
- Dos características tienen una correlación positiva perfecta si $r = 1$.
- Dos características tienen ninguna correlación si $r = 0$.
- Dos características tienen correlación negativa perfecta si $r = -1$.

Nos interesan las variables que tienen alta correlación con la variable objetivo (MEDV).

1.6 Ajuste de una Regresión Lineal por Mínimos Cuadrados

Se utiliza una función de coste idéntica a la utilizada en el modelo Adaline (SSE):

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Esencialmente, la regresión por mínimos cuadrados puede entenderse como Adaline sin la función de paso unitario, de modo que obtenemos valores objetivo continuos en lugar de las etiquetas de clase -1 y 1.

Implementaremos este modelo utilizando Descenso de Gradiente.

[ ]:
```
class LinearRegressionGD(object):

    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
```

```
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return self.net_input(X)
```

[ ]:
```
X = df[['RM']].values
y = df['MEDV'].values
```

[ ]:
```
from sklearn.preprocessing import StandardScaler

sc_x = StandardScaler()
sc_y = StandardScaler()

X_std = sc_x.fit_transform(X)
y_std = sc_y.fit_transform(y[:, np.newaxis]).flatten()

lr = LinearRegressionGD()
lr.fit(X_std, y_std)
```

[ ]:
```
plt.plot(range(1, lr.n_iter+1), lr.cost_)
plt.ylabel('SSE')
plt.xlabel('Epoch')
#plt.tight_layout()
#plt.savefig('images/10_05.png', dpi=300)
plt.show()
```

NOTA: Como alternativa, también existe una solución cerrada para resolver OLS (Mínimos cuadrados ordinarios) que implica un sistema de ecuaciones lineales:

$$w = (X^T X)^{-1} X^T y$$

[ ]:
```
def lin_regplot(X, y, model):
    plt.scatter(X, y, c='steelblue', edgecolor='white', s=70)
    plt.plot(X, model.predict(X), color='black', lw=2)
    return
```

[ ]:
```
lin_regplot(X_std, y_std, lr)
plt.xlabel('Average number of rooms [RM] (standardized)')
plt.ylabel('Price in $1000s [MEDV] (standardized)')
```

```
#plt.savefig('images/10_06.png', dpi=300)
plt.show()
```

[ ]:
```
X = df[['LSTAT']].values
sc_x = StandardScaler()
sc_y = StandardScaler()

X_std = sc_x.fit_transform(X)
y_std = sc_y.fit_transform(y[:, np.newaxis]).flatten()

lr = LinearRegressionGD()
lr.fit(X_std, y_std)

lin_regplot(X_std, y_std, lr)
plt.xlabel('[LSTAT] (standardized)')
plt.ylabel('Price in $1000s [MEDV] (standardized)')

#plt.savefig('images/10_06.png', dpi=300)
plt.show()
```

[ ]:
```
print('Pendiente (w1): %.3f' % lr.w_[1])
print('Intercepto (w0): %.3f' % lr.w_[0])
```

1.6.1 Implementación con scikit-learn

[ ]:
```
from sklearn.linear_model import LinearRegression

slr = LinearRegression()
slr.fit(X, y)
y_pred = slr.predict(X)
print('Pendiente (w1): %.3f' % slr.coef_[0])
print('Intercepto (w0): %.3f' % slr.intercept_)
```

[ ]:
```
lin_regplot(X, y, slr)
plt.xlabel('Average number of rooms [RM]')
plt.ylabel('Price in $1000s [MEDV]')

#plt.savefig('images/10_07.png', dpi=300)
plt.show()
```

1.7 Evaluación de una Regresión Lineal

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

X = df.iloc[:, :-1].values
y = df['MEDV'].values
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=0)

slr = LinearRegression()
slr.fit(X_train, y_train)

y_train_pred = slr.predict(X_train)
y_test_pred = slr.predict(X_test)
```

[ ]:
```
X.shape
```

[ ]:
```
plt.scatter(y_train_pred,  y_train_pred - y_train,
            c='steelblue', marker='o', edgecolor='white',
            label='Training data')
plt.scatter(y_test_pred,  y_test_pred - y_test,
            c='limegreen', marker='s', edgecolor='white',
            label='Test data')
plt.xlabel('Predicted values')
plt.ylabel('Residuals')
plt.legend(loc='upper left')
plt.hlines(y=0, xmin=-10, xmax=50, color='black', lw=2)
plt.xlim([-10, 50])
plt.tight_layout()

# plt.savefig('images/10_09.png', dpi=300)
plt.show()
```

Otra medida cuantitativa útil para evaluar el rendimiento de un modelo de regresión es el **Error Cuadrático Medio (MSE)**.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

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

print('MSE train: %.3f, test: %.3f' % (mean_squared_error(y_train,␣
 ↪y_train_pred),mean_squared_error(y_test, y_test_pred)))
```

Coeficiente de Determinación (R^2):

$$R^2 = 1 - \frac{SSE}{SST}$$

SSE : Suma de errores al cuadrado

$SST = \sum_{i=1}^{n}\left(\hat{y^{(i)}} - \mu_y\right)^2$ es la suma de errores totales al cuadrado

SST es simplemente la varianza de la respuesta.

Para el conjunto de datos de entrenamiento, el R^2 está acotado entre 0 y 1, pero puede llegar a ser negativo para el conjunto de pruebas.

Si $R^2 = 1$, el modelo se ajusta perfectamente a los datos con un MSE igual a cero.

[ ]:
```
from sklearn.metrics import r2_score

print('R^2 train: %.3f, test: %.3f' % (r2_score(y_train,␣
 ↪y_train_pred),r2_score(y_test, y_test_pred)))
```

1.8 Algunas variantes de la Regresión Lineal mediante Regularización

Ridge Regression:

$$J(w)_{Ridge} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \|w\|_2^2$$

Least Absolute Shrinkage and Selection Operator (LASSO):

$$J(w)_{LASSO} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \|w\|_1$$

Elastic Net:

$$J(w)_{ElasticNet} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda_1 \sum_{j=1}^{m} w_j^2 + \lambda_2 \sum_{j=1}^{m} |w_j|$$

[ ]:
```
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet

ridge = Ridge(alpha=1.0)
lasso = Lasso(alpha=1.0)
elanet = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

1.9 Regresión de Árbol de Decisión para modelar relaciones No Lineales

Recordando, definimos la Entropía como una medida de impureza para determinar qué división (en un árbol) de características maximiza la ganancia de información (IG). Para una división binaria:

$$IG(D_p, x_i) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

N_p es el número de muestras en el nodo padre, I es la función de impureza, D_p es el subconjunto de muestras de entrenamiento en el nodo padre, y D_{left} y D_{right} son los subconjuntos de muestras de entrenamiento en el nodo hijo izquierdo y derecho después de la división.

Sin embargo, para utilizar un árbol de decisión para la regresión, **necesitamos una medida de impureza que sea adecuada para las variables continuas**, por lo que definimos la medida de impureza de un nodo t como el MSE:

$$I(t) = MSE(t) = \frac{1}{N_t} \sum_{i \in D_t} \left(y^{(i)} - \hat{y}_t \right)^2$$

Aquí, N_t es el número de muestras de entrenamiento en el nodo t, D_t es el subconjunto de entrenamiento en el nodo t, $y^{(i)}$ es el valor objetivo verdadero, $\hat{y}_t$ es el valor objetivo predicho (media de la muestra):

$$\hat{y}_t = \frac{1}{N_t} \sum_{i \in D_t} y^{(i)}$$

En el contexto de la regresión del árbol de decisión, el **MSE también suele denominarse varianza dentro del nodo**, por lo que el criterio de división también se conoce mejor como **reducción de la varianza.**

[ ]:
```
from sklearn.tree import DecisionTreeRegressor

X = df[['LSTAT']].values
y = df['MEDV'].values

tree = DecisionTreeRegressor(max_depth=3)
tree.fit(X, y)

sort_idx = X.flatten().argsort()

lin_regplot(X[sort_idx], y[sort_idx], tree)
plt.xlabel('% lower status of the population [LSTAT]')
plt.ylabel('Price in $1000s [MEDV]')
#plt.savefig('images/10_13.png', dpi=300)
plt.show()
```

1.10 Regresión Random Forest

[ ]:
```
X = df.iloc[:, :-1].values
y = df['MEDV'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,␣
↪random_state=1)
```

[ ]:
```
from sklearn.ensemble import RandomForestRegressor

forest =␣
↪RandomForestRegressor(n_estimators=1000,criterion='mse',random_state=1,n_jobs=-1)
forest.fit(X_train, y_train)
```

```
y_train_pred = forest.predict(X_train)
y_test_pred = forest.predict(X_test)

print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train, y_train_pred),
        mean_squared_error(y_test, y_test_pred)))
print('R^2 train: %.3f, test: %.3f' % (
        r2_score(y_train, y_train_pred),
        r2_score(y_test, y_test_pred)))
```

[ ]:
```
plt.scatter(y_train_pred,
            y_train_pred - y_train,
            c='steelblue',
            edgecolor='white',
            marker='o',
            s=35,
            alpha=0.9,
            label='training data')
plt.scatter(y_test_pred,
            y_test_pred - y_test,
            c='limegreen',
            edgecolor='white',
            marker='s',
            s=35,
            alpha=0.9,
            label='test data')

plt.xlabel('Predicted values')
plt.ylabel('Residuals')
plt.legend(loc='upper left')
plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='black')
plt.xlim([-10, 50])
plt.tight_layout()

# plt.savefig('images/10_14.png', dpi=300)
plt.show()
```

1.11 XGBoost como modelo de Regresión

Documentación: https://xgboost.readthedocs.io/en/stable/parameter.html

[ ]:
```
import xgboost as xgb

data_dmatrix = xgb.DMatrix(data=X,label=y)
```

```
xg_reg = xgb.XGBRegressor(objective='reg:linear',colsample_bytree=0.
 ↪3,learning_rate=0.1,
                max_depth=5,alpha=10,n_estimators=1000)

xg_reg.fit(X_train,y_train)
```

[ ]:
```
y_train_pred = xg_reg.predict(X_train)
y_test_pred = xg_reg.predict(X_test)

print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train, y_train_pred),
        mean_squared_error(y_test, y_test_pred)))
print('R^2 train: %.3f, test: %.3f' % (
        r2_score(y_train, y_train_pred),
        r2_score(y_test, y_test_pred)))
```

[ ]:
```
preds = xg_reg.predict(X_test)

rmse = np.sqrt(mean_squared_error(y_test, preds))
print("RMSE: %f" % (rmse))
```