Jose Daniel Rave Robayo
Ángel López Benítez

Practica 6

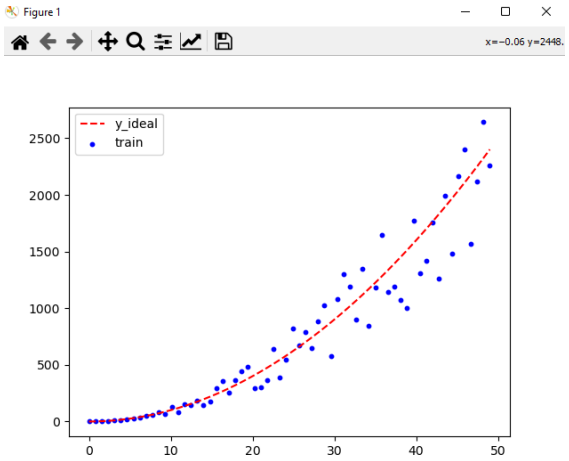
Aprendizaje Automático
en la práctica

En esta práctica vamos a utilizar un modelo de datos generados por nosotros mismos. Se trata de una función exponencial a la que le haremos varias perturbaciones/ruido sobre los puntos para generar así una nube de puntos.

¿Cómo los generamos? Utilizamos la siguiente implementación para ello:

```python
def gen_data(m, seed=1, scale=0.7):
    """ generate a data set based on a x^2 with added noise """
    c = 0
    x_train = np.linspace(0, 49, m)
    np.random.seed(seed)
    y_ideal = x_train**2 + c
    y_train = y_ideal + scale * y_ideal*(np.random.sample((m,))-0.5)
    x_ideal = x_train
    return x_train, y_train, x_ideal, y_ideal
```

Lo cual nos proporciona lo siguiente:



Sobre estos datos realizaremos un entrenamiento para poder asi realizar una predicción lo mas acertada y parecida a la función original. En este caso haremos uso de las implementaciones proporcionadas por la clase ***sklearn.***

Para cada dato/punto añadiremos unos polinomios para ajustar la predicción a dichos datos. En principio veremos como al usar polinomios de grado 15 sobre ajustamos la predicción a los datos de entrada.

Jose Daniel Rave Robayo
Ángel López Benítez

Dividiremos los datos en 2 secciones; la primera constará del 67% de los datos, que serán usados para entrenar el modelo, y la segunda del 33% será para probar nuevos datos sobre el modelo para comprobar qué tan bien predice sobre datos nunca vistos.

```
X_trainData, X_testData, Y_trainData, Y_testData = train_test_split(X, Y,
                                                    test_size=0.33, random_state= 1)
```

Fase entrenamiento:

En esta fase entrenaremos el modelo con los datos seleccionados para ello. Empezaremos por añadirles polinomios de grado 15. Luego vamos a transformarlos en datos manejables, es decir, relativamente pequeños con respecto a la media y desviación típica. Ya que de lo contrario obtendríamos valores enormes con los cuales es difícil trabajar.

Una vez normalizados, los introducimos para que sean entrenados.

```
def trainPolinomic(degree, XTrain, y_train):
    poly = PolynomialFeatures(degree, include_bias=False)
    xTrainMapped = poly.fit_transform(XTrain) # entrena X

    scaler = StandardScaler() # normalizacion
    XtrainMappedScaled = scaler.fit_transform(xTrainMapped)

    linear_model = LinearRegression()
    linear_model.fit(XtrainMappedScaled, y_train)

    return poly, scaler, linear_model
```

Fase de prueba:

Con el modelo entrenados, ahora debemos realizar una predicción con los otros datos seleccionados y calcular un error con respecto a los datos ideales.

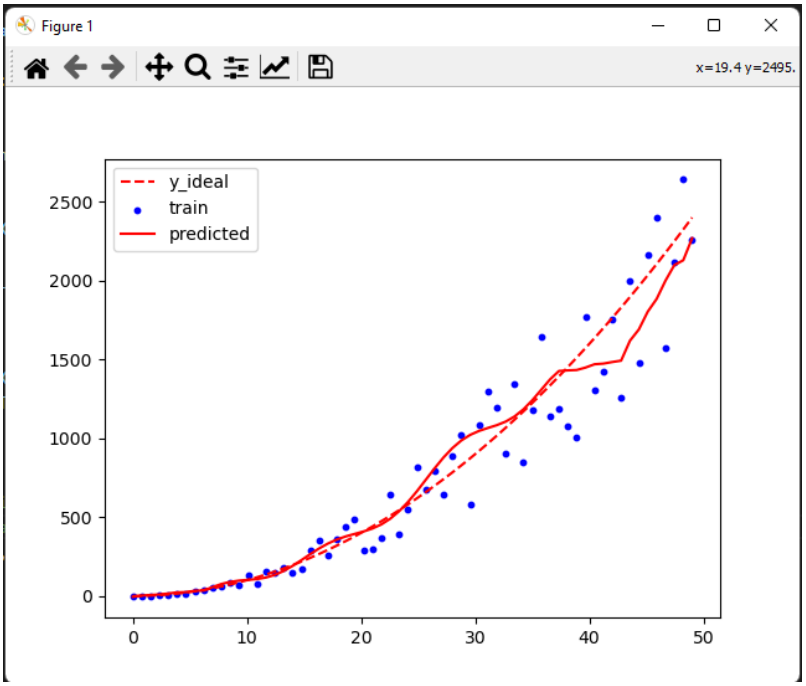
```
test , yTestPredict = calcError(X_testData_matrix, Y_testData, poly, scaler, linear_model)
print("Test:" + str(test))
```

Como es de esperar, al utilizar un grado de polinomio muy alto, tendremos un sobreajuste del modelo sobre los datos de entrenamiento, lo que implica que sobre los datos de prueba habrá un error muy alto a diferencia de los de entrenamiento.

```
PS C:\Users\joseda\Desktop\Uni\Cuarto\APA\AprendizajeAutomaticoP0\P6> python .\main.py
Train:11855.047472036993
Test:48579.55375377189
```

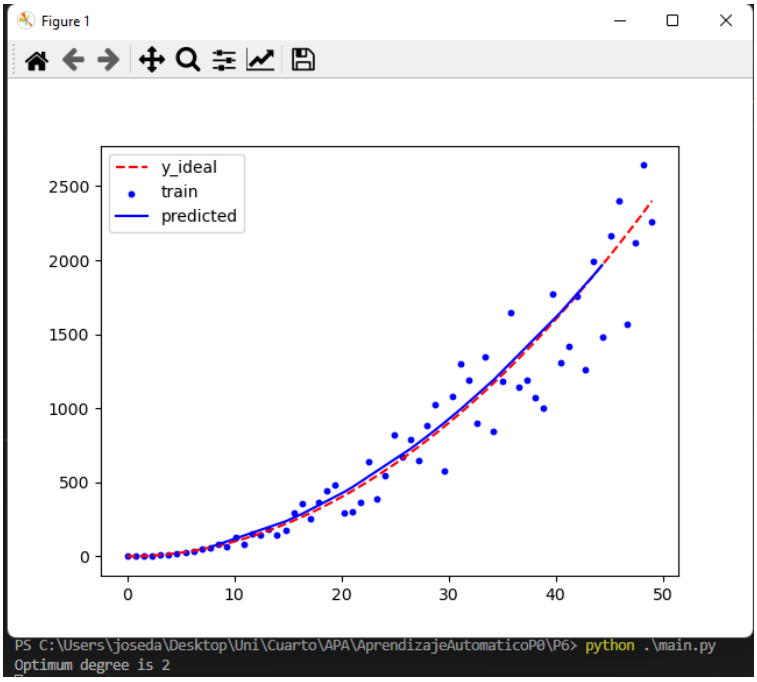
Tanto es así que cuadriplica el error con respecto a los de entrenamiento.

Si representamos el resultado gráficamente, veremos que dicho sobreajuste:



Ahora realizaremos el mismo proceso, pero en vez de utilizar un valor fijo de grado, vamos a buscar cuál nos da un valor de error mínimo para los datos nuevos. Sin embargo, en este caso vamos a dividir los datos, en 3 secciones. Dejaremos los datos de entrenamiento sobre el 60%, y los otras dos secciones en 20%. Ambas secciones serán de prueba, solo que una de ellas será para **validar** dicho error mínimo que buscamos. Es decir, será la que decida qué grado es el correcto.

Mientras tanto, los otros datos serán de **prueba**, y indicarán si realmente es el adecuado, volviendo a realizar una predicción sobre datos nuevos que no ha visto el modelo.



Vemos que ahora la predicción es mucho mas acertada con respecto a lo ideal.

Jose Daniel Rave Robayo
Ángel López Benítez

Sigamos avanzando. Ahora, el modelo de regresión lineal actual no penaliza los valores que están sobreajustados. Por ellos vamos a cambiarlo por la estrategia de regularización Ridle, que acepta un valor de regularización λ. En este caso vamos a probar con valores valores tales que:

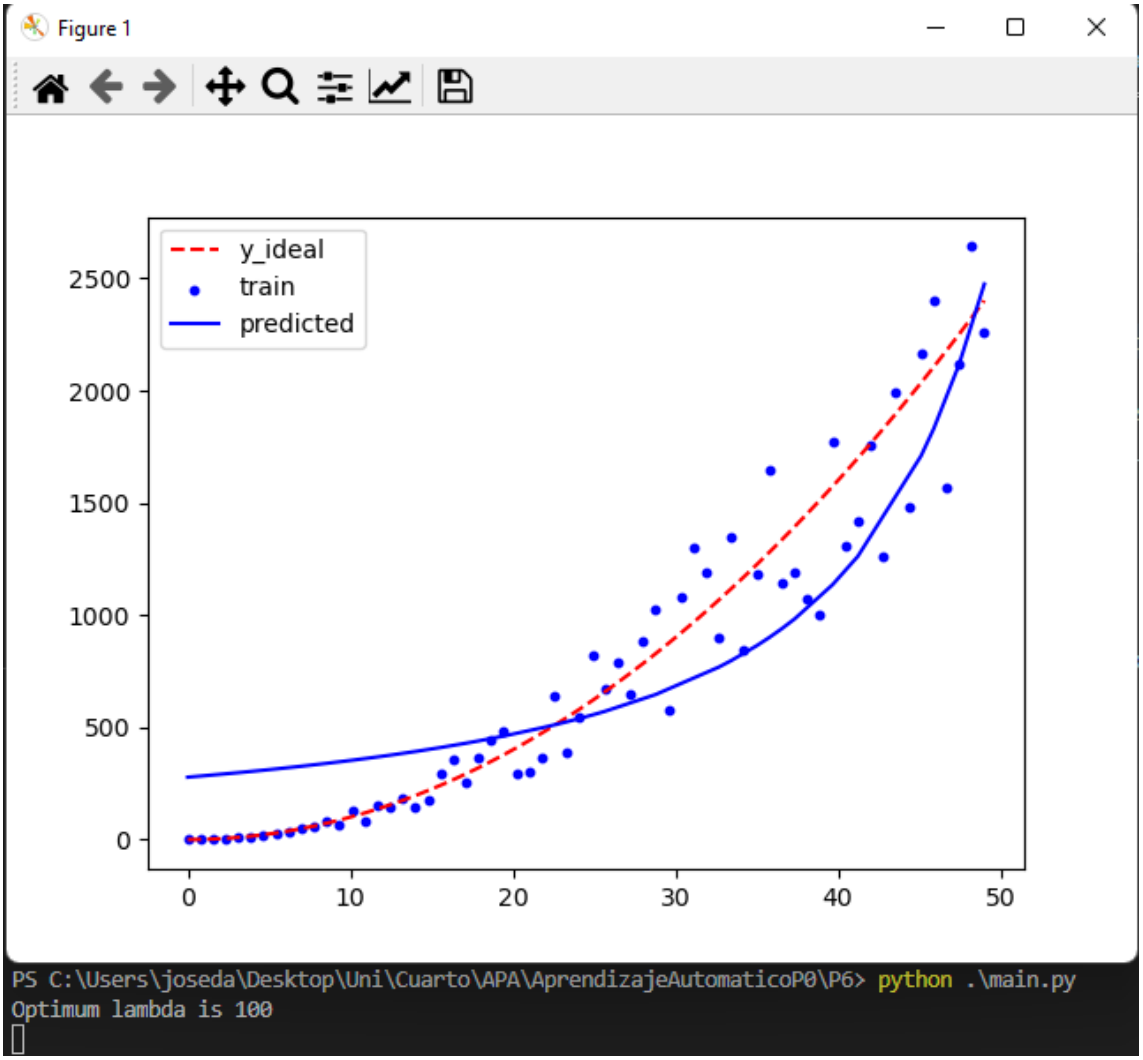
$$[\lambda = 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-2, 1, 10, 100, 300, 600, 900]$$

Por cada valor de lambda vamos a entrenar el modelo con un polinomio de grado 15, buscando así aquel cuyo error de predicción sea el mínimo.

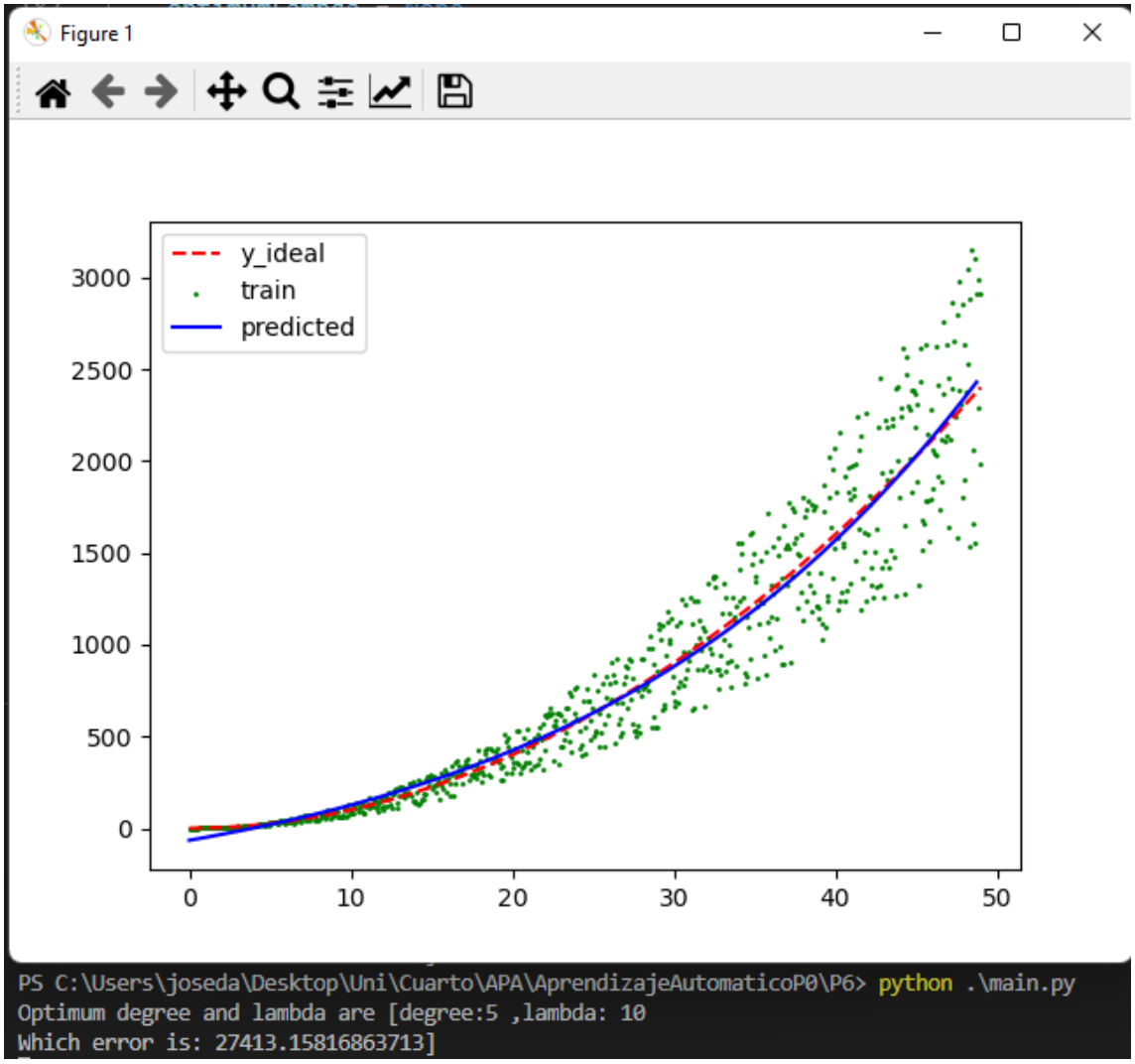
```python
RegularizerLambdas =  [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-2, 1, 10, 100, 300, 600, 900]
optimumLambda = None
minErrorValidate = float('inf')
for i in range(len(RegularizerLambdas)):
    poly, scaler, linear_model = trainPolinomicRegularized(15, X_trainData_matrix, Y_trainData, RegularizerLambdas[i])

    error = calcError(X_validateData_matrix, Y_validateData, poly, scaler, linear_model)[0]
    # print(f"Validate {i} :" + str(error))
    if(error < minErrorValidate):
        minErrorValidate = error
        optimumLambda = i
```

Ejecutando el código de arriba, nos sale que el mejor valor de lambda es 100, y la gráfica es la siguiente:



Se aprecia que con el mismo grado de polinomio, pero con un parámetro de regularización adecuado, la grafica se asemeja mucho mas a la ideal, pero aun con cierto error. Por ende, vamos ahora a buscar el valor optimo tanto de lambda como del grado del polinomio, y vamos a incrementar el número de datos para resultados mas precisos.

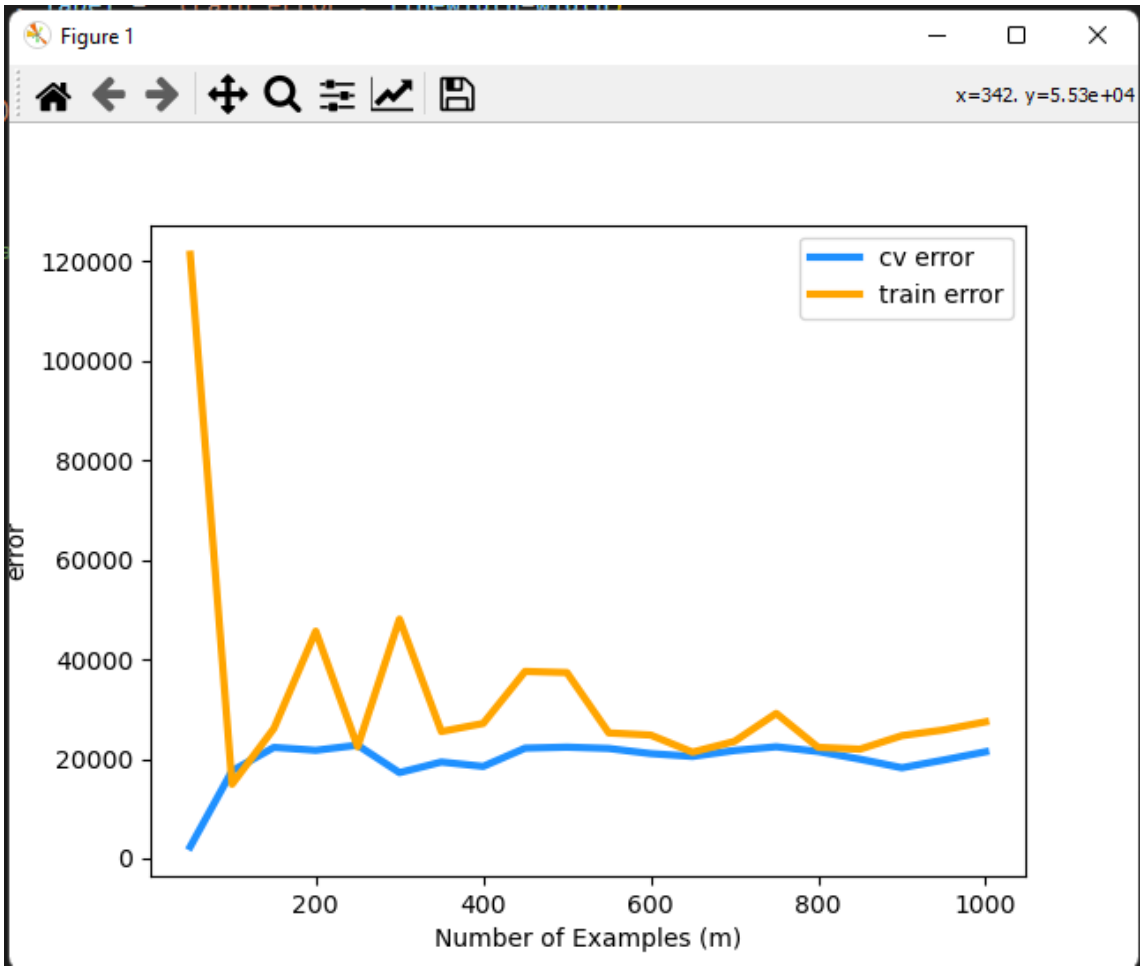


Podemos ver que la gráfica ahora es mucho mas parecida a la ideal con lambda 10 y un grado polinómico de 5. Además el error es correcto, ya que debería estar entre 20.000 y 30.000.

Para finalizar, vamos a observar como afecta el numero de datos a un modelo que está sobre ajustado. Empezaremos con datos de tamaño 50 e incrementaremos hasta 1000, saltando de 50 en 50. Usaremos un grado polinomico de 16, y contrastaremos el error en cada caso con el error de entrenamiento y el error de validación.

Observaremos que en un principio el error de validación será muy alto (tal y como vimos al principio) y el de entrenamiento bajo (ya que es con lo que se ha entrenado el

modelo). Pero conforme incrementamos el tamaño de los datos, el error de validación va menguando y se aproxima al de entrenamiento.



Implementación de gráfica:

```python
def learningCurves():
    min  = 50
    max = 1001
    X = np.arange(min, max, 50)
    Y_errorV = []
    Y_errorT = []
    for i in range(X.shape[0]):
        x_train, y_train, x_ideal, y_ideal = gen_data(X[i])
        errorTrain, errorValide = Train(x_train, y_train)
        Y_errorT.append(errorTrain)
        Y_errorV.append(errorValide)
        print(f"Error Train: {errorTrain}")
        print(f"Error errorValidate : {errorValide}")
        print("====================")

    width = 3
    plt.plot(X, Y_errorV, c = 'dodgerblue', label = 'cv error', linewidth=width)
    plt.plot(X, Y_errorT, c = 'orange', label = 'train error', linewidth=width)
    # print(Y_errorT)

    plt.xlabel("Number of Examples (m)")
    plt.ylabel("error")
```

Jose Daniel Rave Robayo
Ángel López Benítez

```python
def Train(X, Y):
    X_trainData, X_testData, Y_trainData, Y_testData = train_test_split(X, Y,
                                                        test_size=0.4, random_state= 1)

    X_validateData, X_testData, Y_validateData, Y_testData = train_test_split(X_testData, Y_testData,
                                                        test_size=0.5, random_state= 1)

    X_trainData_matrix = X_trainData[:, None]
    X_testData_matrix = X_testData[:, None]
    X_validateData_matrix = X_validateData[:, None]

    poly, scaler, linear_model = trainPolinomic(16, X_trainData_matrix, Y_trainData)

    errorValidate = calcError(X_validateData_matrix, Y_validateData, poly, scaler, linear_model)[0]
    errorTrain = calcError(X_trainData_matrix, Y_trainData, poly, scaler, linear_model)[0]

    return errorValidate, errorTrain
```