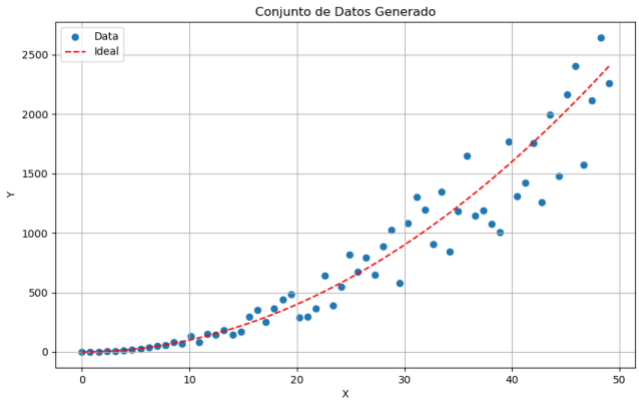
Práctica 6. Aprendizaje automático en la práctica

Datos de entrenamiento

En esta práctica se utilizarán datos artificiales obtenidos en la siguiente función proporcionada en el enunciado de la práctica ***gen_data()***:

Si se muestra la gráfica:



Como el método ***gen_data()*** devuelve una lista unidimensional y las funciones de ***scikit-learn*** que se usarán necesitan arrays bidimensionales se deberá de añadir una nueva dimensión siempre que sea necesario.

Sobreajuste a los ejemplos de entrenamiento

A continuación, se muestra el código donde se realiza el sobreajuste a los ejemplos de entrenamiento junto con su respectiva explicación:

```python
def overfitting_example():
    x_train, y_train, x_ideal, y_ideal = gen_data(m = 64)
    X_train, X_test, y_train, y_test = train_test_split(x_train, y_train, test_size=0.33, random_state=1)

    poly = PolynomialFeatures(degree=15, include_bias=False)
    X_train_poly = poly.fit_transform(X_train[:, np.newaxis])
    X_test_poly = poly.transform(X_test[:, np.newaxis])

    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train_poly)
    X_test_scaled = scaler.transform(X_test_poly)

    model = LinearRegression()
    model.fit(X_train_scaled, y_train)

    y_train_pred = model.predict(X_train_scaled)
    y_test_pred = model.predict(X_test_scaled)

    mse_train = error(y_train_pred, y_train)
    mse_test = error(y_test_pred, y_test)

    print("Error on training set:", mse_train)
    print("Error on test set:", mse_test)

    plt.scatter(X_train, y_train, label='train')
    plt.scatter(X_train, y_train_pred, label='predicted')
    plt.plot(x_ideal, y_ideal, label='y_ideal', color='orange')
    plt.legend()
    plt.show()
```

Para poder generar el error del modelo tanto de los datos de entrenamiento como los datos de prueba se aplicará la siguiente fórmula:

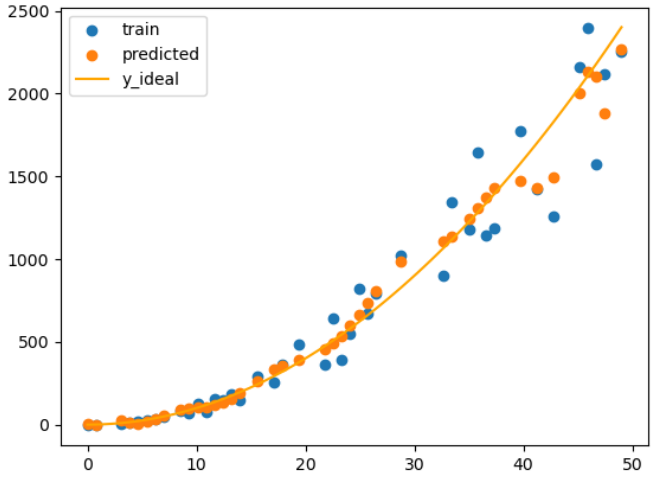
$$J(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})^2$$

Traducido a código será de la siguiente forma:

```python
def error(predictions, labels):
    m = len(labels)
    mse = np.sum((predictions - labels) ** 2) / (2 * m)
    return mse
```

Una vez realizado todo el proceso de sobreajuste se obtienen los siguientes valores de error, tanto para el entrenamiento como de prueba:

- Error on training set: 11855.0447622640992
- Error on test set: 48579.4289101656



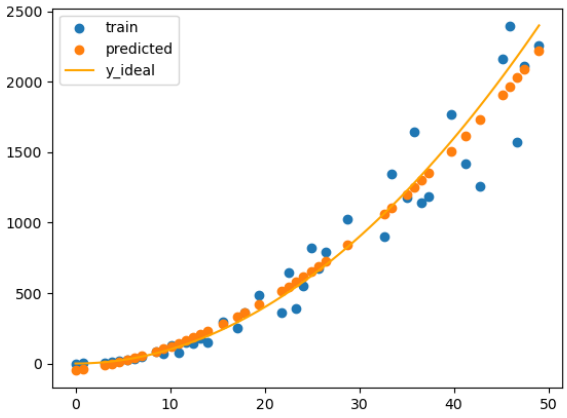
Elección del grado de polinomio usando un conjunto de validación

Generando los datos de la misma forma que en el apartado anterior esta vez usaremos el 60% de los datos para entrenamiento, el 20% para validación y el restante 20% para prueba. Entrenando modelos con transformaciones polinómicas de grado 1 a 10 deberíamos encontrar que el de menor error genera predicciones.
Con todo esto la función en la que se obtiene el grado óptimo del polinomio es la siguiente:

```python
def select_polynomial_degree():
    x_train, y_train, _, _ = gen_data(64)
    X_train, X_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=1)

    degrees = range(1, 11)
    best_degree = None
    min_val_error = float('inf')

    for degree in degrees:
        poly = PolynomialFeatures(degree=degree, include_bias=False)
        X_train_poly = poly.fit_transform(X_train[:, np.newaxis])
        X_val_poly = poly.transform(X_val[:, np.newaxis])

        model = LinearRegression()
        model.fit(X_train_poly, y_train)

        y_val_pred = model.predict(X_val_poly)
        val_error = error(y_val_pred, y_val)

        if val_error < min_val_error:
            min_val_error = val_error
            best_degree = degree

    print("Best polynomial degree:", best_degree)
    return best_degree
```

El resultado de esto, uniéndolo con el método del apartado anterior es el siguiente:

- Best polynomial degree: 2
- Error on training set: 14877.96905472397
- Error on test set: 28209.12655628797



Elección del parámetro λ

Se utilizará una transformación polinómica de grado 15 y evaluaremos sobre el conjunto de validación cruzada los valores de λ.
Una vez explicados los valores que tendrán el grado del polinomio y el parámetro de regularización se desarrolla el método:

```python
def select_lambda(degree = 15):
    x_train, y_train, _, _ = gen_data(64)
    X_train, X_val_test, y_train, y_val_test = train_test_split(x_train, y_train, test_size=0.4, random_state=1)
    X_val, X_test, y_val, y_test = train_test_split(X_val_test, y_val_test, test_size=0.5, random_state=1)

    lambdas = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100, 300, 600, 900]
    best_lambda = None
    min_test_error = float('inf')

    for lambda_ in lambdas:
        poly = PolynomialFeatures(degree=degree, include_bias=False)
        X_train_poly = poly.fit_transform(X_train[:, np.newaxis])
        X_val_poly = poly.transform(X_val[:, np.newaxis])

        model = Ridge(alpha=lambda_)
        model.fit(X_train_poly, y_train)

        y_val_pred = model.predict(X_val_poly)
        test_error = error(y_val_pred, y_val)

        if test_error < min_test_error:
            min_test_error = test_error
            best_lambda = lambda_

    print("Best lambda:", best_lambda)
    return best_lambda
```

Proporcionando la mejor lambda: 1e-06

Elección de hiper-parámetros

A continuación se implementará una búsqueda exhaustiva de la mejor combinación de grado del polinomio y valor de λ para un conjunto de 750 valores, usando el 60% de los datos para entrenamiento, el 20% para validación y el restante 20% para prueba.
Probando con polinomios hasta grado 15 y valores de λ se desarollará el método ***hyperparameter_tuning()***:

```python
def hyperparameter_tuning():
    x_train, y_train, _, _ = gen_data(750)
    X_train, X_val_test, y_train, y_val_test = train_test_split(x_train, y_train, test_size=0.4, random_state=1)
    X_val, X_test, y_val, y_test = train_test_split(X_val_test, y_val_test, test_size=0.5, random_state=1)

    degrees = range(1, 16)
    lambdas = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100, 300, 600, 900]
    best_degree = None
    best_lambda = None
    min_test_error = float('inf')

    for degree in degrees:
        for lmbda in lambdas:
            poly = PolynomialFeatures(degree=degree, include_bias=False)
            X_train_poly = poly.fit_transform(X_train[:, np.newaxis])
            X_val_poly = poly.transform(X_val[:, np.newaxis])

            model = Ridge(alpha=lmbda)
            model.fit(X_train_poly, y_train)

            y_val_pred = model.predict(X_val_poly)
            test_error = error(y_val_pred, y_val)

            if test_error < min_test_error:
                min_test_error = test_error
                best_degree = degree
                best_lambda = lmbda

    print("Best polynomial degree:", best_degree)
    print("Best lambda:", best_lambda)
    print('Error', min_test_error)
    return best_degree, best_lambda
```