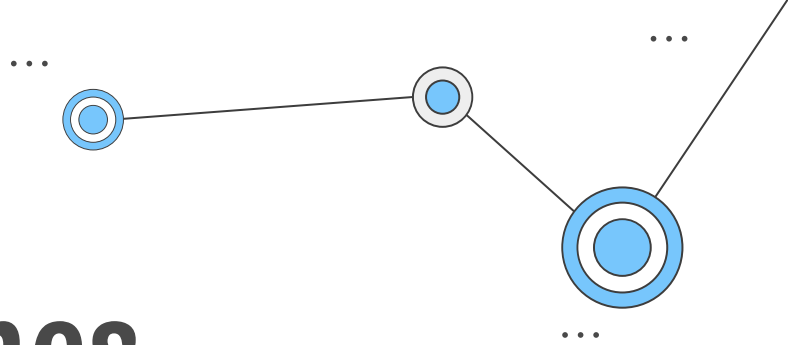


Regresiones

UNIVERSIDAD CATÓLICA DE CÓRDOBA
JESUITAS

Dr. Francisco Arduh
2023

Introducción

- Vimos procedimientos sin detenernos en los modelos
- Entender los algoritmos nos va a ayudar elegir el correcto para el problema qué queremos atacar, elegir el rango de hiper parámetros correctos o entender por qué falla en ciertos contextos.

En esta presentación vamos a

- Analizar la regresión lineal en dos formas diferentes:
  - Usando la “forma cerrada” de la regresión. Esto significa su solución exacta.
  - Usando un enfoque iterativo qué nos va a acercar a la solución gradualmente llamado “Descenso de gradiente” (GD).
- Regresión Polinomial.
- Regresión Logística y Softmax.

Regresión Lineal

Se construye como la suma pesada de las características:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

O en forma vectorial:

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

¿Cómo ajustamos el modelo a los datos?

Necesitamos medir qué tan bien nuestro modelo se ajusta o no a los datos.

Regresión Lineal

Se construye como la suma pesada de las características:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

O en forma vectorial:

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

¿Cómo ajustamos el modelo a los datos?
Necesitamos medir qué tan bien nuestro modelo se ajusta o no a los datos.
Como vimos en la clase anterior podríamos utilizar MSE (RMSE)

$$\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^{\mathsf{T}} \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

Ecuación normal (solución cerrada)

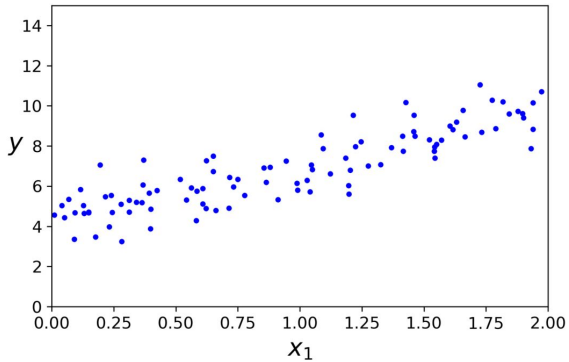
Para encontrar el valor de los pesos **θ** qué minimiza la función de costo, existe un solución cerrada.

$$\widehat{\boldsymbol{\theta}} = (\mathbf{X}^\intercal \mathbf{X})^{-1} \quad \mathbf{X}^\intercal \quad \mathbf{y}$$

Donde $\widehat{\boldsymbol{\theta}}$ es el valor de **θ** qué minimiza MSE.

Lo veamos en un ejemplo, generamos un muestra lineal con ruido:

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```



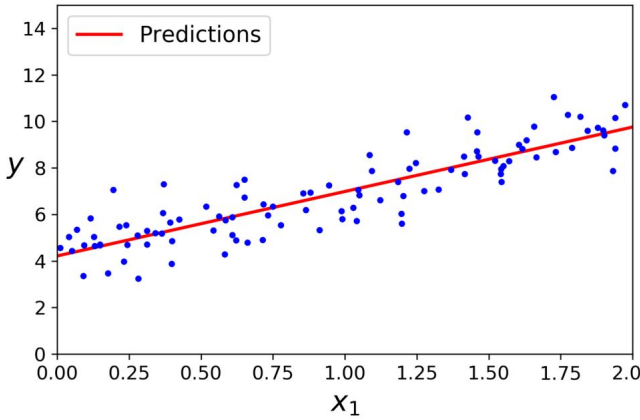
Ecuación normal: Implementación

Utilizando NumPy (Solución exacta):

```
>>> X_b = np.c_[np.ones((100, 1)), X] # add x0 = 1 to each instance
>>> theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
>>> theta_best
array([[4.21509616],
[2.77011339]])
```

Utilizando Scikit-learn (cálculo de pseudo inversa):

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]), array([[2.77011339]]))
```

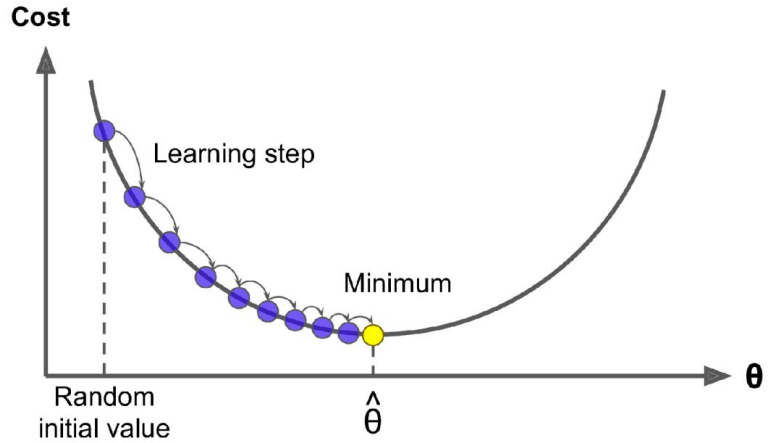


Ecuación normal: complejidad computacional

- Solución exacta (invertir la matriz) su complejidad temporal es de **$O(n^{2.4})$** a **$O(n^3)$**; n es el número de características.
- Solución con Scikit-learn (cálculo de pseudo inversa) su complejidad temporal es de **$O(n^2)$**; n es el número de características.
- La complejidad espacial es lineal con respecto al número de instancias.
- La complejidad para hacer predicciones es lineal al número de instancias y características.

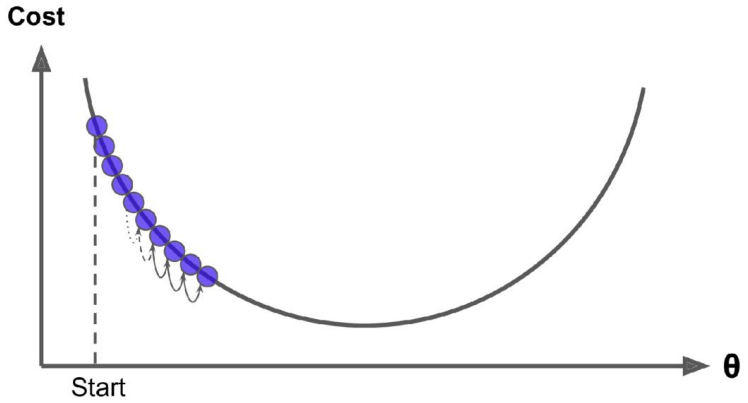
Descenso de gradiente

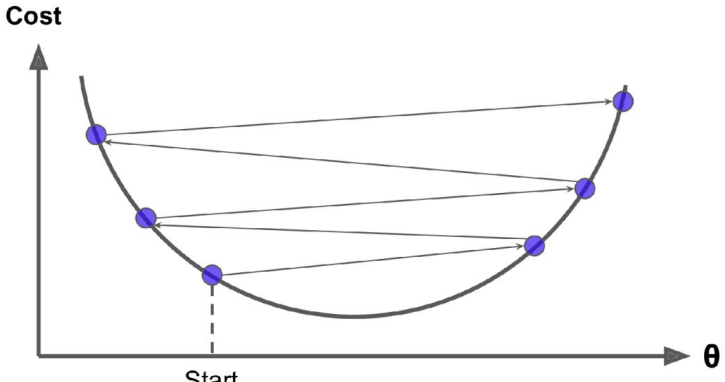
- Es un algoritmo de optimización genérico capaz de encontrar soluciones óptimas a un gran número de problemas.
- Se adapta mejor a casos con gran número de parámetros e instancias.
- La idea general es inicializar de forma aleatoria **θ** e ir modificando iterativamente los parámetros para minimizar la función de costo.



Descenso de gradiente: Learning rate

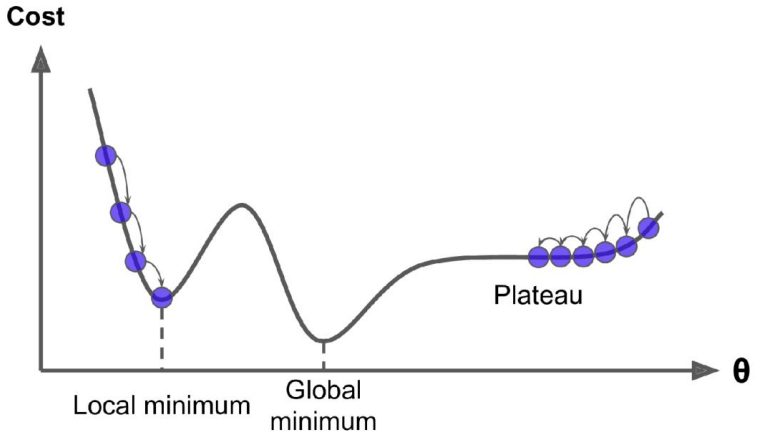
- Un hiperparámetro importante del algoritmo Descenso de gradiente es el *learning rate*.
- Un learning rate bajo va a hacer qué al algoritmo le tome mucho tiempo alcanzar el mínimo.
- En el otro extremo, un learning rate alto resulta en qué el algoritmo empiece a ir de un extremo a otro, llegando incluso a diverger en ciertos casos.





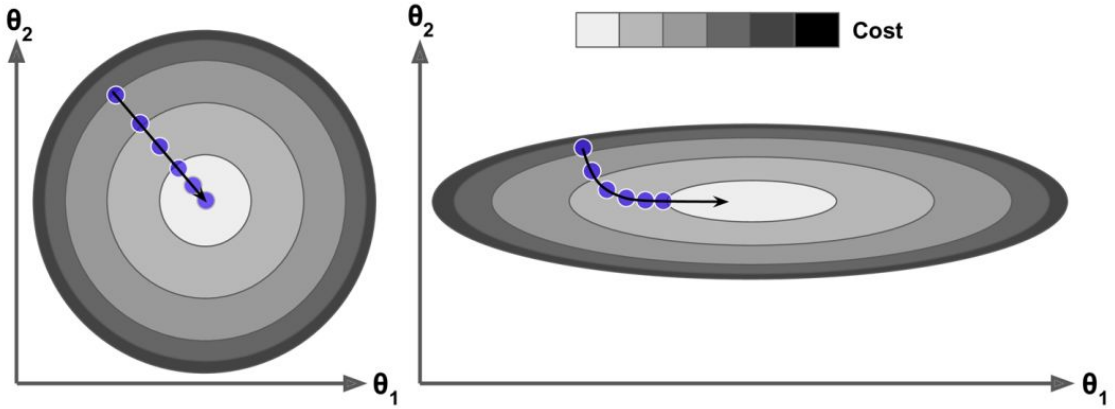
Descenso de gradiente: Forma de las funciones de costo

- La convergencia del algoritmo a un mínimo global no está garantizada. Va a depender de la inicialización.
- Podríamos encontrarnos con un mínimo local o con una meseta en la función de costo qué haría qué al algoritmo le tome mucho tiempo llegar al mínimo.
- En el caso de la función de costo MSE es convexa.



Descenso de gradiente: Funciones de costo MSE

- En el ejemplo el algoritmo de Descenso de Gradiente se dirige hacia el mínimo global.
- En caso de normalizarse la características le va a tomar más tiempo converger.



Descenso de gradiente por lote (Batch gradient descent)

- Para aplicar el Descenso de Gradiente es necesario calcular el gradiente de la función de costo con respecto a cada parámetro $\boldsymbol{\theta}_j$.
- Cada paso del algoritmo incluye los cálculos sobre el training set completo **X**.

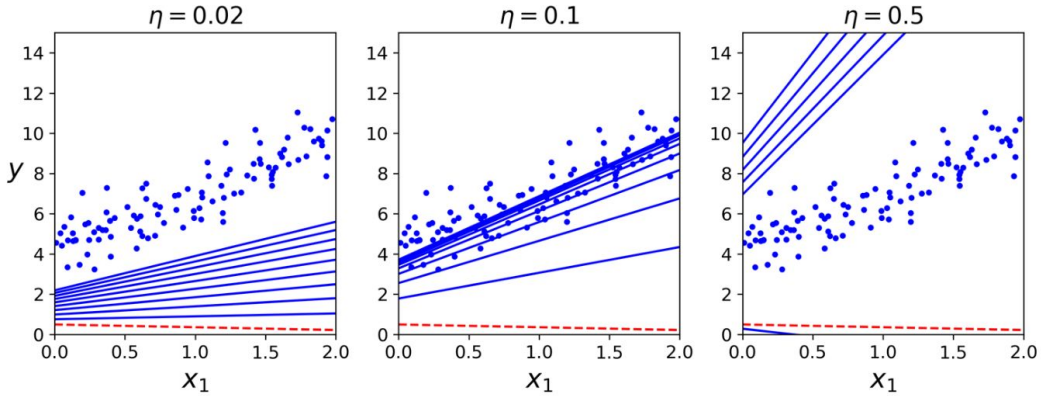
$$\nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^{\mathsf{T}} (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

- Una vez computado el gradiente se lo multiplica por el learning rate η y se lo sustrae de $\boldsymbol{\theta}$.

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta})$$

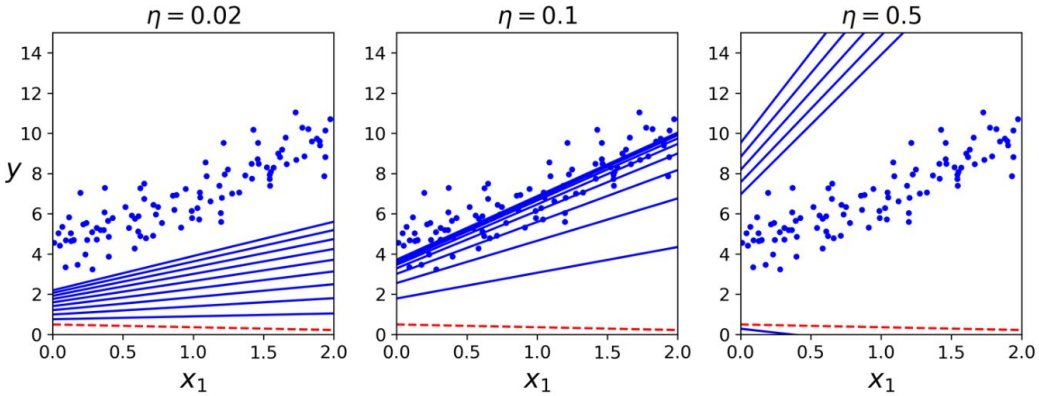
Descenso de gradiente por lote: Implementación

```
>>> eta = 0.1 # learning rate
>>> n_iterations = 1000
>>> m = 100
>>> theta = np.random.randn(2,1)
>>> # random initialization
>>> for iteration in range(n_iterations):
>>>     gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
>>>     theta = theta - eta * gradients
>>> theta
array([[4.21509616],
[2.77011339]])
```



Descenso de gradiente por lote: Implementación

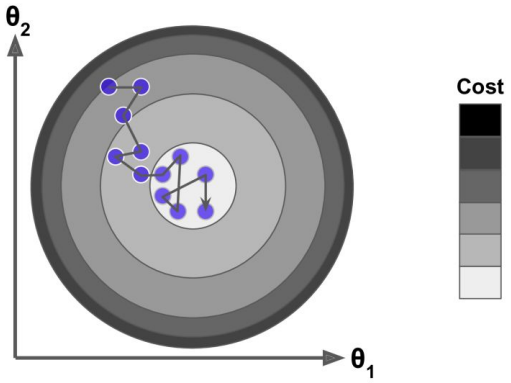
```
>>> eta = 0.1 # learning rate
>>> n_iterations = 1000
>>> m = 100
>>> theta = np.random.randn(2,1)
>>> # random initialization
>>> for iteration in range(n_iterations):
>>> gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
>>> theta = theta - eta * gradients
>>> theta
array([[4.21509616],
[2.77011339]])
```



Se puede setear una tolerancia ϵ, tal que si la norma se vuelve menor a este valor el algoritmo termine.

Descenso de Gradiente Estocástico (Stochastic Gradient Descent)

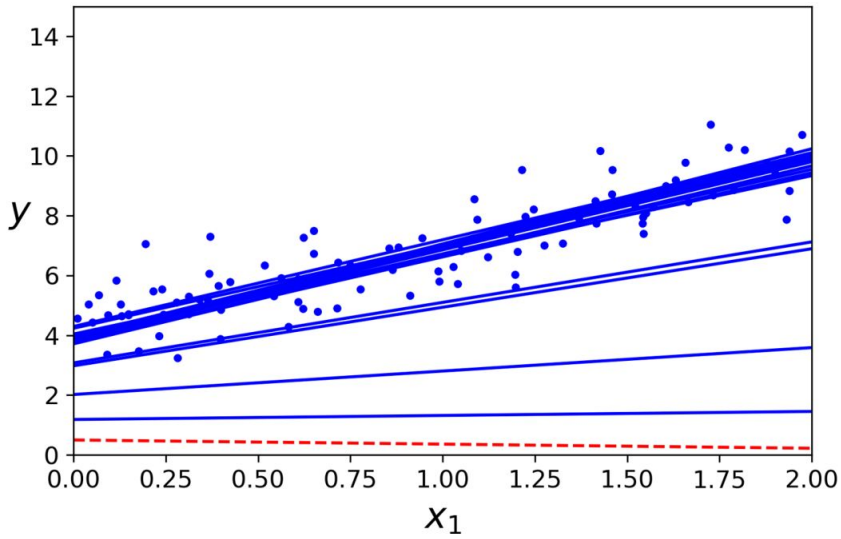
- A diferencia del descenso de gradiente por lote que incluye en los cálculos todo el training set **X**, el descenso de gradiente estocástico sólo incluye una instancia de forma aleatoria.
- A diferencia del DG por lote, este algoritmo no va a descender suavemente al mínimo.
- Cuando el algoritmo para el valor final de los parámetros es bueno, pero no óptimo.
- Si la función de costo es muy irregular, el algoritmo tiene posibilidad de escapar de mínimos locales.



- Debido a qué el algoritmo no alcanza el mínimo se puede ir reduciendo el learning rate en cada iteración con un *cronograma de apredizaje* (*learning schedule*).

Descenso de Gradiente Estocástico Implementación

```
>>> n_epochs = 50
>>> t0, t1 = 5, 50
>>> # learning schedule hyperparameters
>>> def learning_schedule(t):
>>>     return t0 / (t + t1)
>>> theta = np.random.randn(2,1)
>>> # random initialization
>>> for epoch in range(n_epochs):
>>>     for i in range(m):
>>>         random_index = np.random.randint(m)
>>>         xi = X_b[random_index:random_index+1]
>>>         yi = y[random_index:random_index+1]
>>>         gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
>>>         eta = learning_schedule(epoch * m + i)
>>>         theta = theta - eta * gradients
>>> theta
array([[4.21076011],
[2.74856079]])
```



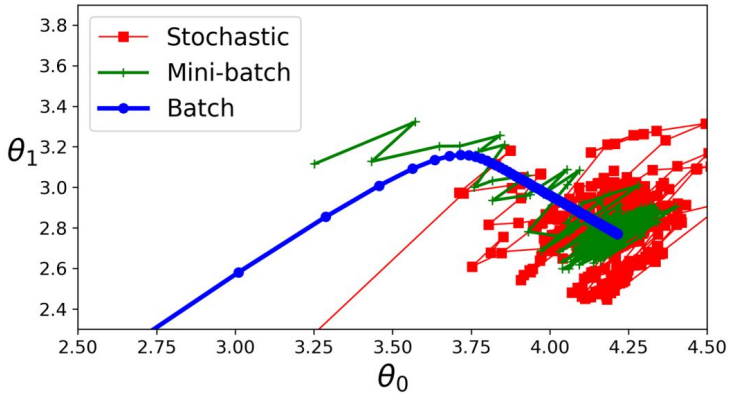
Descenso de Gradiente Estocástico en Scikit-learn

- La clase implementada es **SGDRegressor**
- Variables:
  - max_iter: cuantas iteraciones va a realizar el algoritmo.
  - tol: la tolerancia con respecto al cambio en la función de pérdida.
  - eta0: valor de learning rate a utilizar
  - learning_rate: learning rate schedule, por defecto 'inv_scaling' $\frac{\eta}{step^{\text{power\_t}}}$

```
>>> from sklearn.linear_model import SGDRegressor
>>> sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

Descenso de Gradiente en mini lote (Mini-Batch Gradiente Descent)

- Computa el gradiente en un set de instancias random llamada mini lote (o mini-batch).
- Es menos errático qué el SGD y termina más cerca del mínimo.
- Puede tener más problemas para escapar de un mínimo local.
- Al igual qué SGD necesita un parámetros de tolerancia.



Comparación entre algoritmos

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`

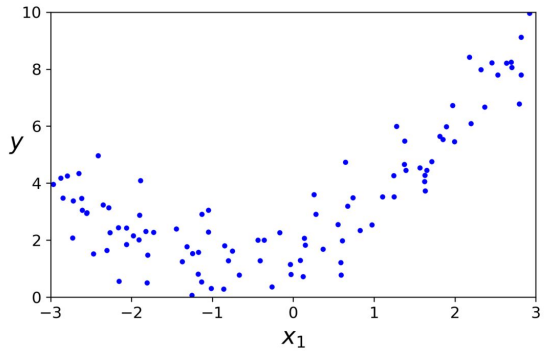
Regresión polinomial

¿Qué podemos hacer en el caso de tener datos más complejos qué una línea recta?

- Se puede añadir características qué sean potencias como nuevas características.
- Hacer lo anterior y ajustarlo por un modelo lineal se denomina Regresión Polinomial.

Generemos un conjunto de puntos qué siga una forma cuadrática:

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```



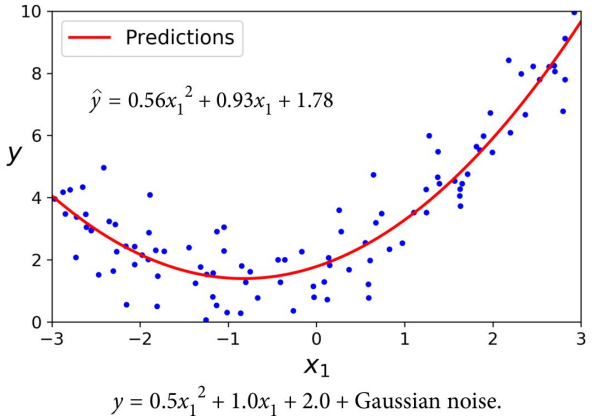
Regresión polinomial: implementación

Utilizo **PolynomialFeatures:**

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

Ajusto una regresión lineal

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```



$y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise.}$

La regresión polinomial es capaz de relacionar las variables a diferencia de la regresión lineal.

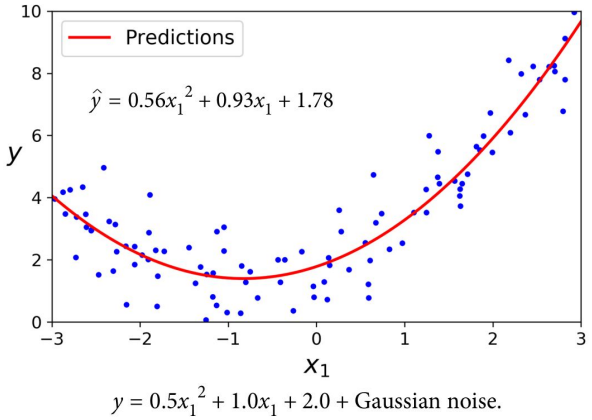
Regresión polinomial: implementación

Utilizo **PolynomialFeatures:**

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

Ajusto una regresión lineal

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```



$y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise.}$

La regresión polinomial es capaz de relacionar las variables a diferencia de la regresión lineal.

Cuidado!

PolynomialFeatures(degree=d) escala como (n+d)!/d!n!

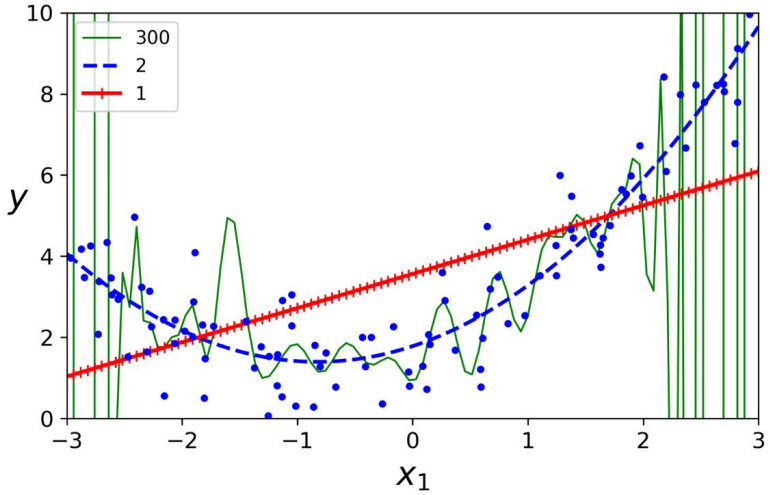
¿Cómo puedo saber qué grado utilizar?

- Polinomio de alto grado
  ->Overfitting en training set.
- Polinomio de bajo grado
  ->Underfitting en training set.

¿Cómo decidimos qué modelo utilizar?

En clases anteriores vimos que:

- Qué se podía hacer esto utilizando validación cruzada.
- Otra forma es utilizar curvas de aprendizaje (*learning curves*)



Curvas de aprendizaje

Son gráficas que presentan el desempeño del modelo en el training set y en el validation set en función del tamaño del training set.

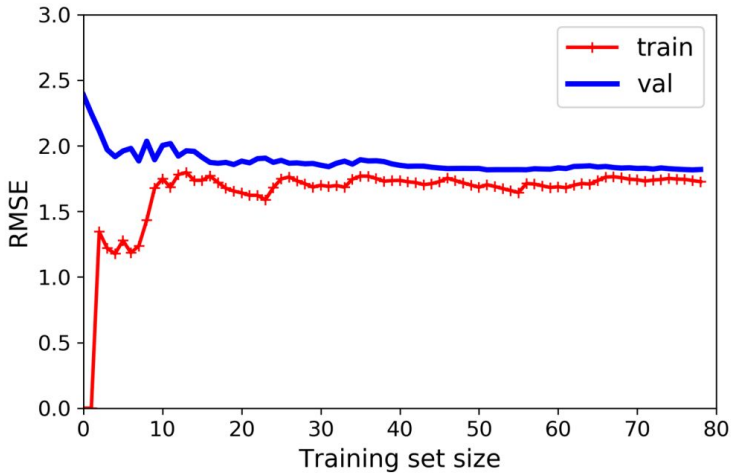
En código:

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

Curvas de aprendizaje: underfitting

- Curva train empieza en cero
- Curva val empieza muy arriba porque el modelo no puede generalizar bien.
- Las curvas llegan a una meseta y llegan a estar muy juntas.

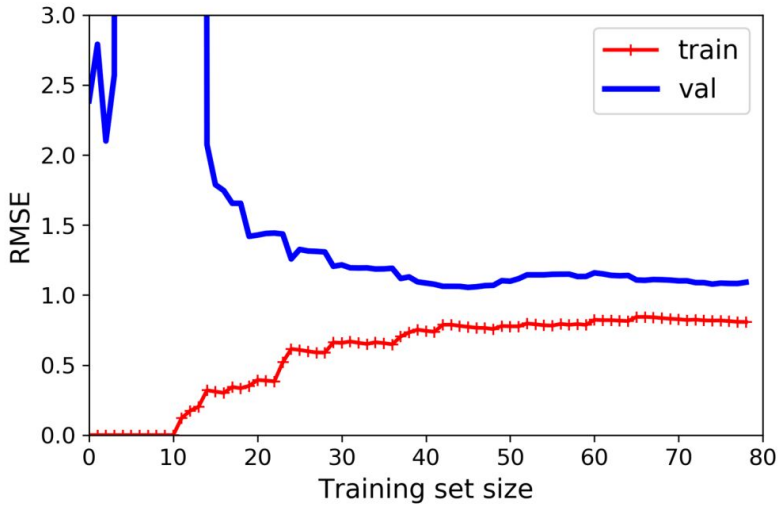
```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```



Curvas de aprendizaje: overfitting

- El error del training set es mucho mejor que en el caso anterior.
- Existe un espacio mucho mayor entre las curvas.

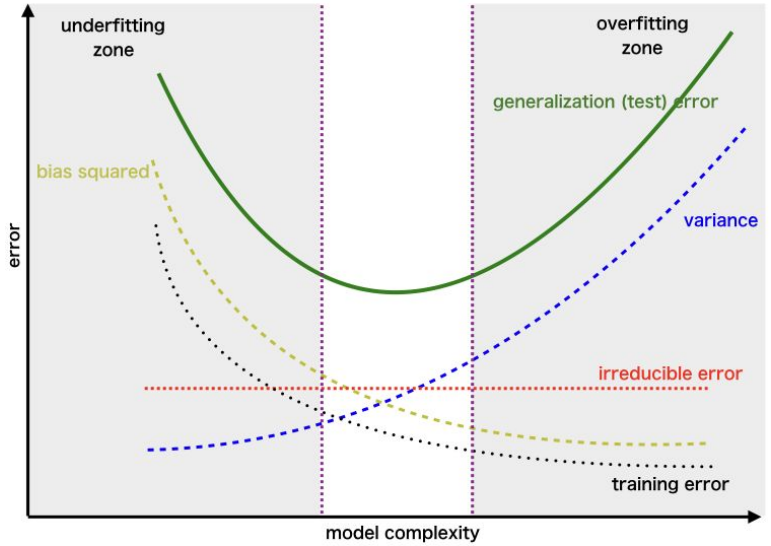
```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
```



Bias/Variance trade-off

El error de la generalización del modelo tiene tres fuentes de error diferentes:

- Bias (Sesgo): Se debe a suposiciones erróneas (Modelo lineal cuando en realidad es cuadrático).
- Varianza: Debido al exceso de sensibilidad a pequeñas variaciones en los datos de entrenamiento.
- Error irreducible: Es la parte del error que proviene del ruido.



Más info: https://towardsdatascience.com/the-bias-variance-tradeoff-8818f41e39e9

Regularización de modelos lineales

Una forma de reducir el sobreajuste es la regularización del modelo. Esto se logra reduciendo los grados de libertad del modelo.
En modelos lineales esto se logra restringiendo los pesos del modelo y se puede implementar de distintas formas:

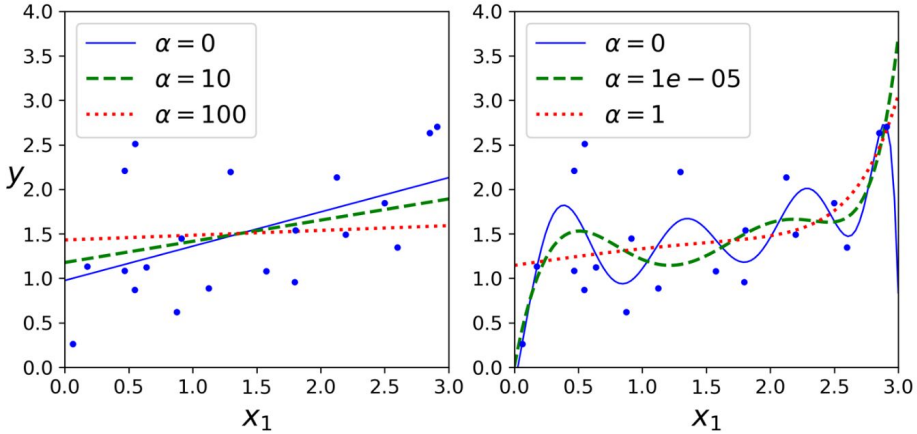
- Ridge Regression,
- Lasso Regression,
- Elastic Net

Ridge Regression

Se modifica la función de costo de la regresión lineal de la siguiente manera:

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \Sigma_{i=1}^{n} \theta_i^2$$

- La normalización sólo se aplica en el training set
- Se debe escalar los datos, antes de utilizar RidgeRegression



Ridge Regression en Scikit-learn

Forma cerrada:

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[1.55071465]])
```

Utilizando descenso de gradiente estocástico:

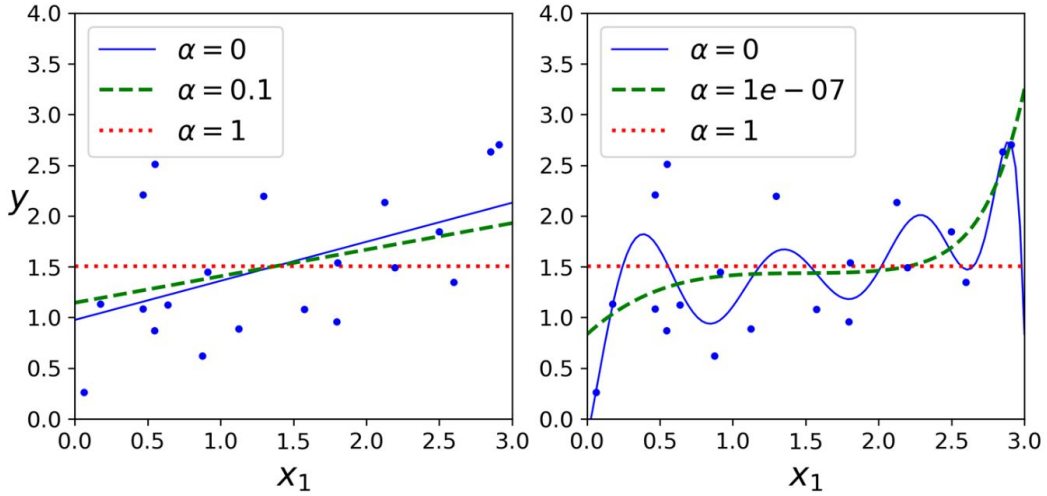
```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

Lasso Regression

Se modifica la función de costo de la regresión lineal de la siguiente manera:

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \Sigma_{i=1}^{n} \left|\theta_i\right|$$

- Utiliza la norma ℓ_1 en vez ℓ_2.
- Tiende a eliminar pesos de menor importancia. Automáticamente realiza una selección de características.



Lasso Regression en Scikit-learn

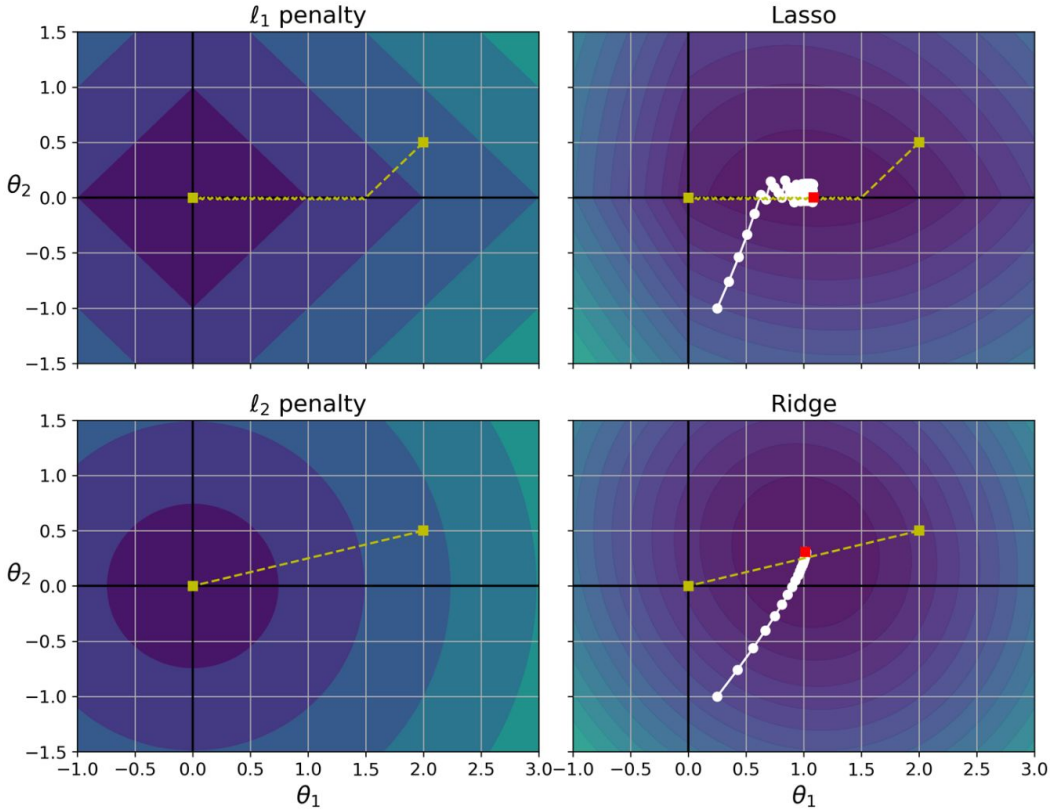
```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([1.53788174])
```

Para utilizar descenso de gradiente en este caso se utiliza la clase **SGDRegressor**, pero con una penalidad ℓ_1, donde se utiliza el vector subgradiente.

Vector subgradiente:

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \begin{pmatrix} \operatorname{sign}(\theta_1) \\ \operatorname{sign}(\theta_2) \\ \vdots \\ \operatorname{sign}(\theta_n) \end{pmatrix} \quad \text{where} \quad \operatorname{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

Comparación de funciones de costo



Elastic Net

Se modifica la función de costo de tal forma qué sea una mezcla entre las regresiones Lasso y Ridge:

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + r\alpha\Sigma_{i=1}^{n}\left|\theta_i\right| + \frac{1-r}{2}\alpha\Sigma_{i=1}^{n}\theta_i^2$$

r es un hiperparámetro qué controla la mezcla.

- Se trata de evitar usar una regresión lineal sin regularización.
- Se utiliza por defecto Ridge.
- Si se quiere eliminar características se utiliza ElasticNet.
- Se evita el uso de Lasso por sus problemas de convergencia.

Implementación en Scikit-Learn:

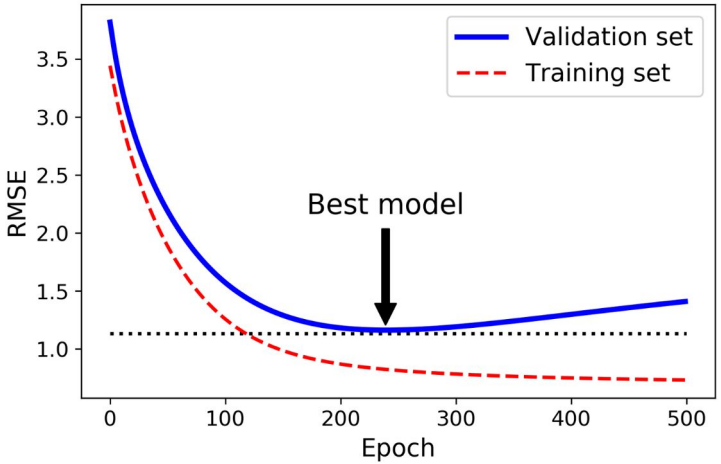
```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([1.54333232])
```

Para descenso de gradiente se utiliza la clase **SDGRegressor** con penalidad 'elasticnet' y l1_ratio es el hiperparámetro de mezcla.

Early Stopping

Método de regularización qué consiste en parar el entrenamiento tan pronto la curva de performance de la región de validación alcance el mínimo.

- Se evita overfitting
- Con descenso de gradiente en mini-batch o estocástico las curvas pueden ser muy ruidosas. Se toma el Mínimo después de un número de eventos.



Early Stopping: Implementación

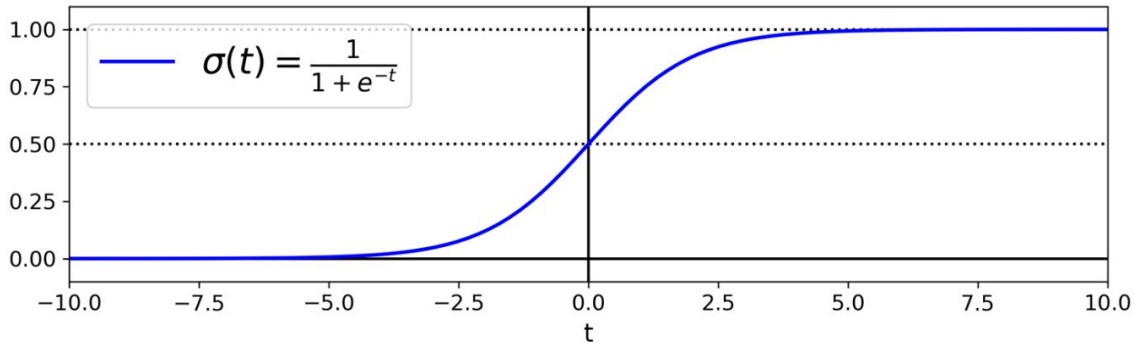
```
from sklearn.base import clone

# prepare the data
poly_scaler = Pipeline([
        ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
        ("std_scaler", StandardScaler())
    ])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)  # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

Regresión Logística

- Calcula la probabilidad de que una instancia pertenezca a una clase (clase positiva).
- Si la probabilidad es mayor al 50% se dice qué pertenece a esta clase.
- Es un clasificador binario.

Para calcular la probabilidad se computa una suma pesada de las característica más el término bias y se lo utiliza como input a la función logística.

$$\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma(\mathbf{x}^\intercal \boldsymbol{\theta})$$



$$\hat{y} = \begin{cases} 0 & \text{si } \hat{p} < 0.5 \\ 1 & \text{si } \hat{p} \geq 0.5 \end{cases}$$

$$\hat{y} = \begin{cases} 0 & \text{si } x^\intercal \theta < 0 \\ 1 & \text{si } x^\intercal \theta \geq 0 \end{cases}$$

Regresión Logística: función de costo

¿Cómo se entrena la regresión logística?

- La idea es entrenar el vector **θ** parámetros.
- No existe una solución cerrada al problema.
- Para cada instancia la función de costo es:

$$c(\theta) = \begin{cases} -log(\hat{p}) & \text{si } y = 1 \\ -log(1 - \hat{p}) & \text{si } y = 0 \end{cases}$$

- La función de costo sobre promediando sobre todo el training set (*logloss*):

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\Sigma_{i=1}^{m}\left[y^{(i)}log\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right)log\left(1 - \hat{p}^{(i)}\right)\right]$$

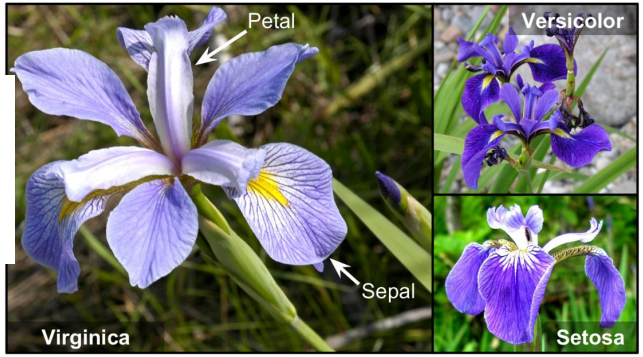
$$\frac{\partial}{\partial\theta_j}\mathrm{J}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma\left(\boldsymbol{\theta}^{\mathrm{T}}\mathbf{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

Nota: No se utiliza MSE como función de costo debido a qué no es convexa.

Regresión Logística: ejemplo

- Vamos a utilizar Iris dataset:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
>>> X = iris["data"][:, 3:]  # petal width
>>> y = (iris["target"] == 2).astype(np.int)  # 1 if Iris virginica, else 0
```

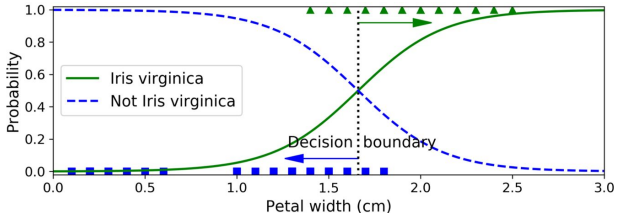


Entrenamos el modelo de regresión logística:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Graficamos

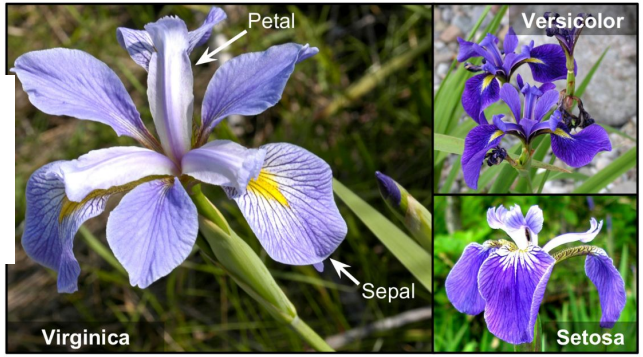
```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris virginica")
```



Regresión Logística: ejemplo

- Vamos a utilizar Iris dataset:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
>>> X = iris["data"][:, 3:]  # petal width
>>> y = (iris["target"] == 2).astype(np.int)  # 1 if Iris virginica, else 0
```

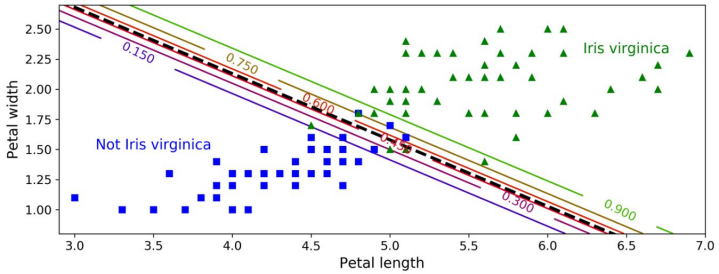


Entrenamos el modelo de regresión logística:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Graficamos

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris virginica")
```



Nota: por defecto Scikit-learn utiliza penalidad ℓ_2

Regresión Softmax

- También llamada regresión logística multinomial.
- Primero se computa unos puntajes, qué corresponde a cada una de las clases: $s_k(\mathbf{x}) = \mathbf{x}^\top \boldsymbol{\theta}^{(k)}$
- Luego se computa la función softmax para calcular las probabilidades:

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

- La predicción de la clase se corresponde con la qué tiene probabilidad más alta.
- Como función de costo se utiliza la entropía cruzada (cross-entropy), definida como:

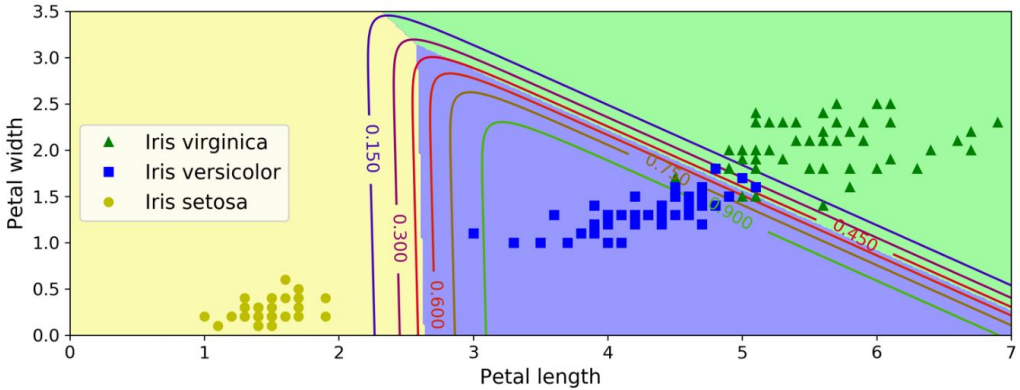
$$J(\boldsymbol{\Theta}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$

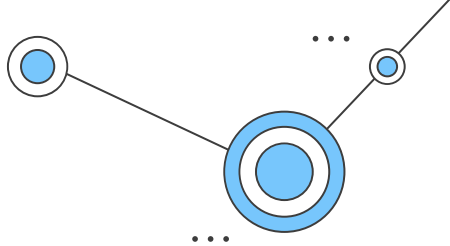
- Computando el gradiente obtenemos:

$$\nabla_{\boldsymbol{\theta}^{(k)}} J(\boldsymbol{\Theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)}\right) \mathbf{x}^{(i)}$$

Regresión Softmax: implementación

```
>>> softmax_reg = LogisticRegression(multi_class="multinomial",
                                     solver="lbfgs",
                                     C=10)
>>> softmax_reg.fit(X, y)
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```





¿Dudas?