

# Applied Machine Learning for Business and Economics

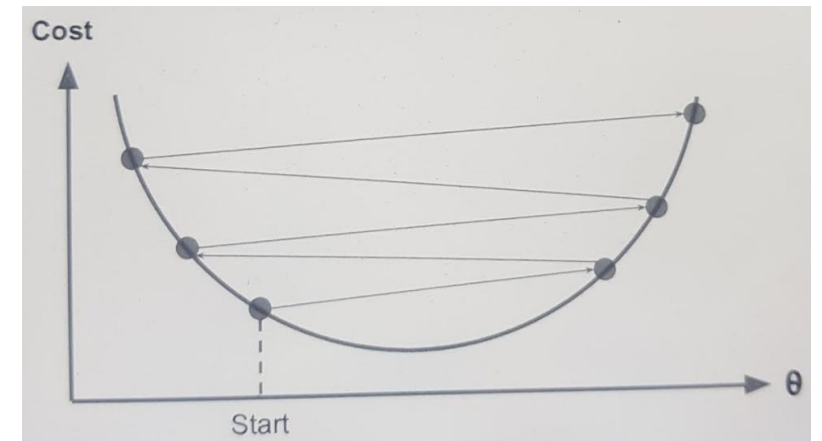
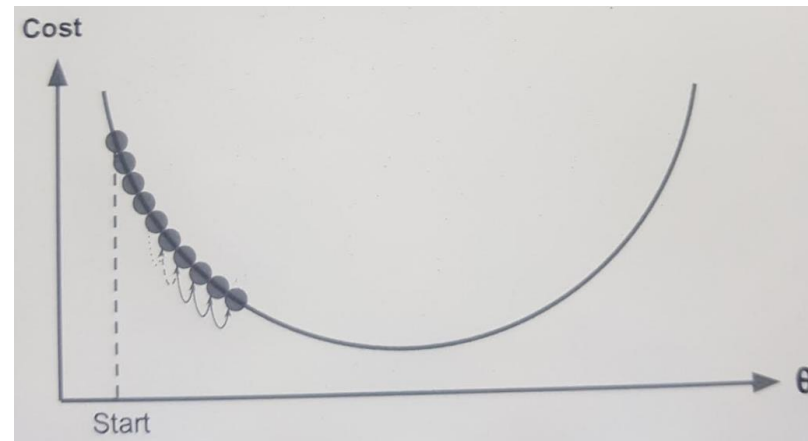
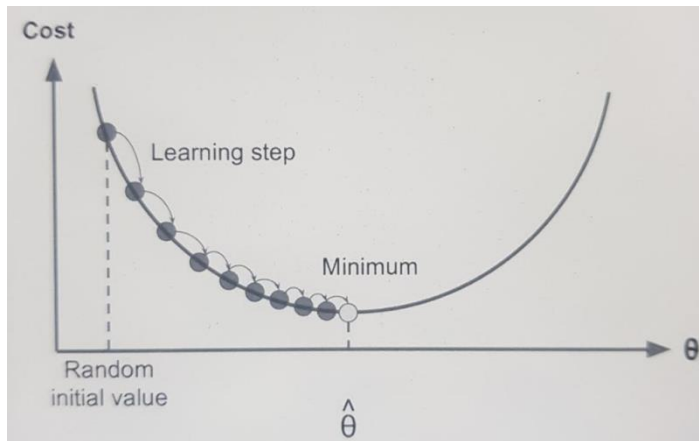
## AMLBE

## Entrenamiento de Modelos: Batch Gradient Descendent

- En adelante veremos diferentes formas de entrenar regresiones lineales para aquellos casos donde existe un gran numero de atributos o muchas instancias de entrenamientos que calibrar en la memoria
- El algoritmo mide el gradiente local del de la función de error respecto del el vector de parámetros  $\Theta$  y va en la dirección del gradiente descendente, una vez que el gradiente es cero se ha alcanzado el mínimo
- Concretamente se comienza llenando  $\Theta$  con valores aleatorios (*random initialization*), y luego se mejora gradualmente por medio del paso de actualización hasta que el algoritmo converge a un mínimo
- Un importante parámetro dentro de éste algoritmo es el tamaño del step determinado por el hyperparámetro learning rate
- La forma de la función de costos y la inicialización aleatoria puede ser clave, idealmente la función de costos debe ser convexa (como es el caso del MSE) hace más fácil encontrar un mínimo global
- Es necesario que las variables estén estandarizadas, de lo contrario la convergencia puede ser más compleja

$$\frac{\partial}{\partial \theta_j} MSE(\Theta) = \frac{2}{m} \sum_{i=1}^m (\Theta^T x^{(i)} - y^{(i)}) x_j \quad \nabla_{\theta} MSE(\Theta) = \begin{bmatrix} \frac{\partial}{\partial \theta_0} MSE(\Theta) \\ \frac{\partial}{\partial \theta_1} MSE(\Theta) \\ \dots \\ \dots \\ \frac{\partial}{\partial \theta_n} MSE(\Theta) \end{bmatrix} = \frac{2}{m} X^T (X\Theta - y)$$

# Entrenamiento de Modelos: Batch Gradient Descendent



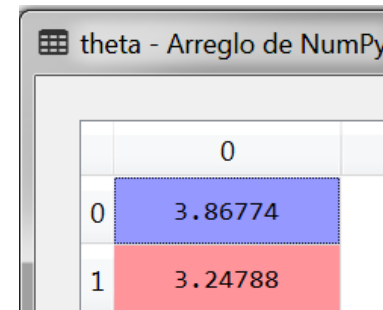
# Entrenamiento de Modelos: Batch Gradient Descendent

- Notar que este procedimiento implica calcular sobre todo el set de entrenamiento,  $X$ , en cada paso del descenso del gradiente por lo cual es de nominado «Batch Gradient Descendent». Dado lo anterior es relativamente lento en grandes sets de entrenamiento
- En contraste, este algoritmo escala bien con el numero de atributos entrenando un modelo de regresión lineal con cientos de miles de atributos es mucho más eficiente en comparación con la ecuación normal:

$$\Theta^{(t+1)} = \Theta^t - \eta \nabla_{\Theta} MSE(\Theta)$$

```
import numpy as np

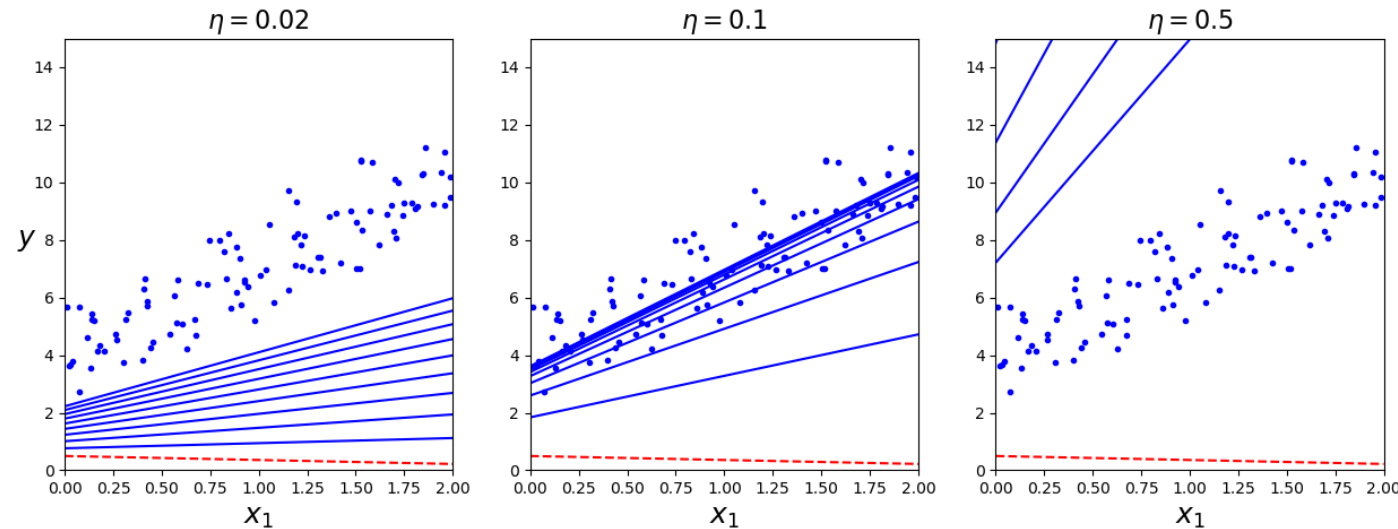
eta=0.1 # Learning rate
n_iterations=1000
m=len(y)
theta=np.random.randn(2,1) #Iniciación aleatoria
for iteration in range(n_iterations):
    gradients=(2.0/m)*np.dot(X_b.T,np.dot(X_b,theta)-y)
    theta=theta-eta*gradients
```



	0
0	3.86774
1	3.24788

# Entrenamiento de Modelos: Batch Gradient Descendent

- Sin embargo, éste algoritmo es sensible al *learning rate*:



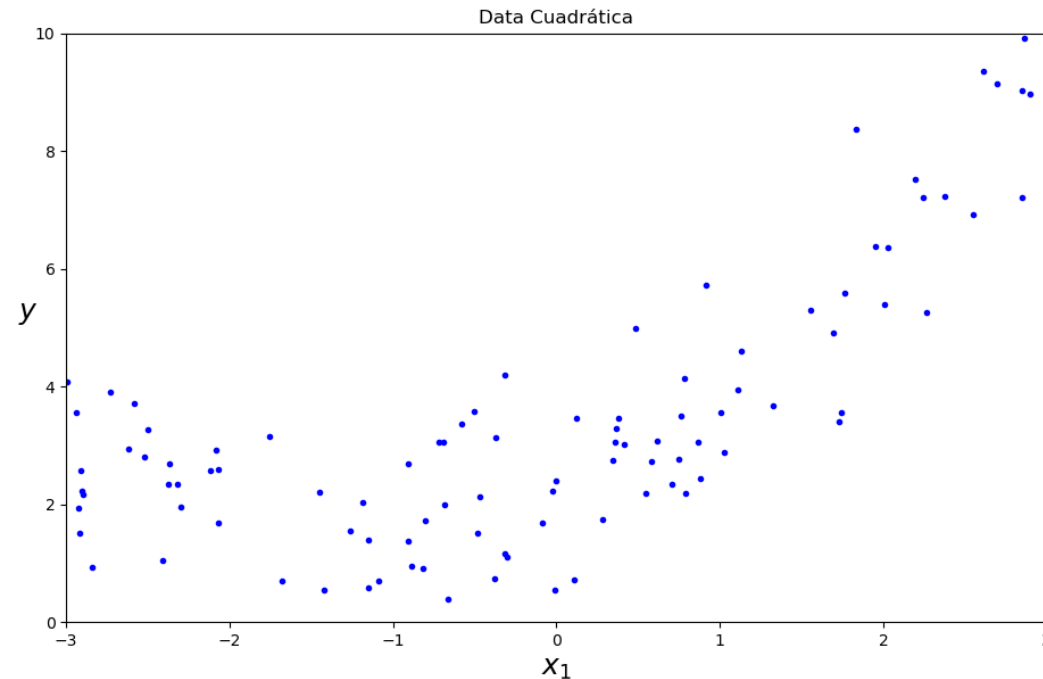
- En el primero el learning rate es demasiado bajo, el algoritmo eventualmente encontrará una solución, pero podría tomar tiempo
- En el segundo caso el learning rate luce bastante bien, de hecho se alcanza el óptimo en muy pocas iteraciones
- En el tercero, el learning rate es muy bajo por lo que el algoritmo diverge yendo cada vez más lejos después de cada iteración
- Para encontrar una buen learning rate un poco más adelante realizaremos el proceso denominado de grid search
- Otra posible solución es aplicar una regla de stop cuando el vector gradiente cae por debajo de cierto threshold denominado tolerancia. Esto, matemáticamente vendría a ser representado por la norma del vector gradiente y debiera ocurrir cuando el algoritmo se encuentra muy cerca del mínimo

# Regresiones Polinómicas

- Que ocurre si la data fuera más compleja que una simple línea recta ?
- Es posible extender nuestro análisis utilizando modelos lineales para hacer fit de data no lineal
- Una forma sencilla de lograr esto es agregar las potencias de cada atributo como un nuevo atributo y luego entrenar un modelo lineal en este set extendido de atributos
- Esta técnica es denominada Regresiones Polinómicas, analicemos el siguiente ejemplo:

```
import numpy as np
import pandas as pd

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

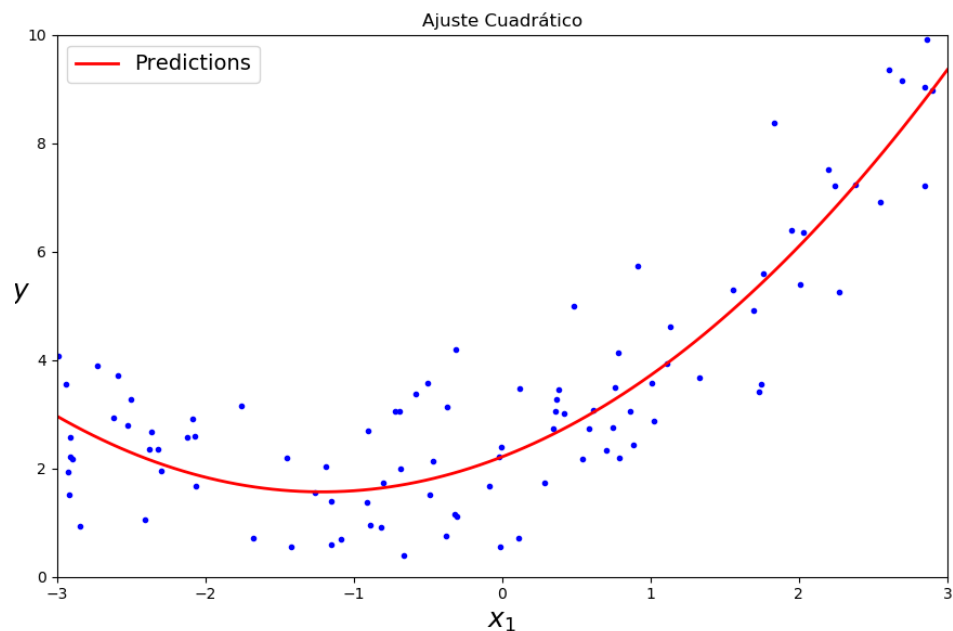


# Regresiones Polinómicas

- Claramente un modelo puramente lineal no será capaz de ajustarse a la data apropiadamente, sin embargo vamos a utilizar una herramienta de Scikit-Learn *PolynomialFeatures* para transformar nuestra data de entrenamiento agregando el cuadrado de cada atributo en el training set como nuevo atributo:

```
from sklearn.linear_model import LinearRegression  
from sklearn.preprocessing import PolynomialFeatures  
poly_features = PolynomialFeatures(degree=2, include_bias=False)  
X_poly = poly_features.fit_transform(X)  
lin_reg = LinearRegression()  
lin_reg.fit(X_poly, y)  
lin_reg.intercept_, lin_reg.coef_
```

(array([ 2.21286333]), array([[ 1.06538512, 0.43871603]]))



$$\hat{y} = 0.5x^2 + 1.0x + 2 + \textit{Gaussian Noise}$$

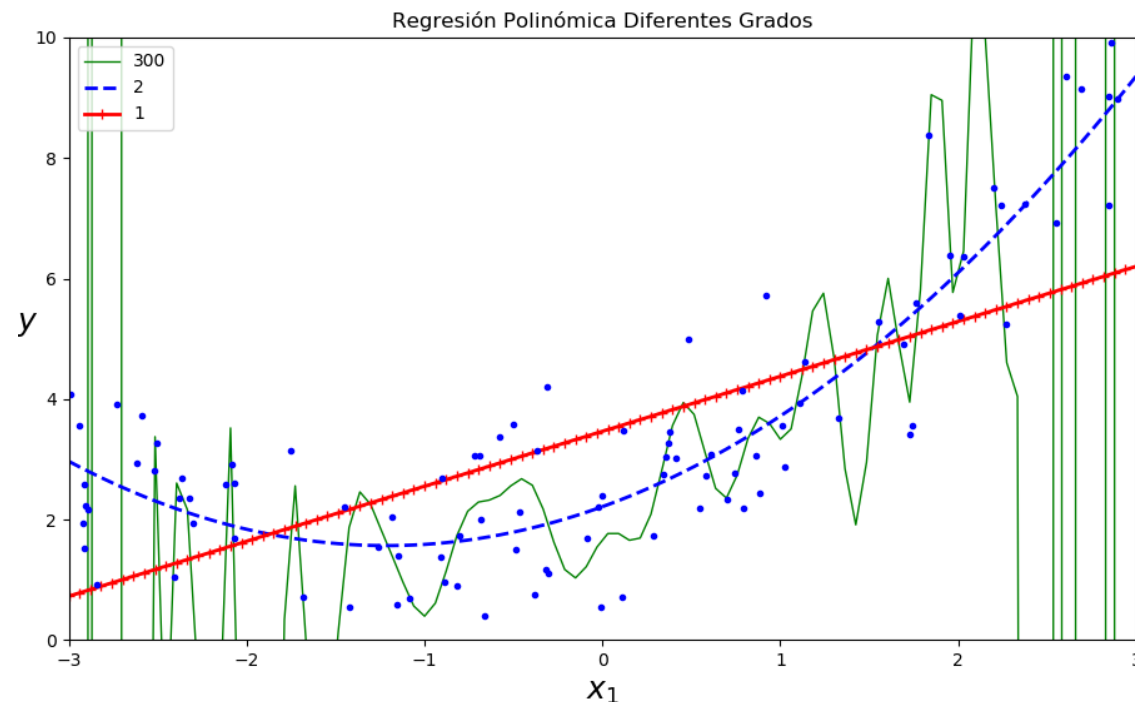
$$\hat{y} = 0.438x^2 + 1.065x + 2.212 + \textit{Gaussian Noise}$$

# Regresiones Polinómicas

- Notar que cuando hay múltiples atributos (en el ejemplo sólo 1), la regresión polinómica es capaz de encontrar las relaciones entre las características (lo que no es posible realizar por una simple regresión lineal)
- Lo anterior pues *PolynomialFeatures* también agrega todas las combinaciones de características hasta un determinado grado dado
- Por ejemplo si tenemos los atributos  $a$  y  $b$ , *PolynomialFeatures*, con grado 3, agregará  $a, b, a^2, b^2, a^3, b^3, ab, a^2b$  y  $ab^2$
- Dado lo anterior hay que cuidar la dimensionalidad pues tiene como trade off inestabilidad numérica pues si tenemos « $n$ » atributos y grado « $d$ », pasamos de tener  $n$  atributos a:

$$\frac{(n + d)!}{d! n!}$$

- Donde podríamos tener gran inestabilidad numérica

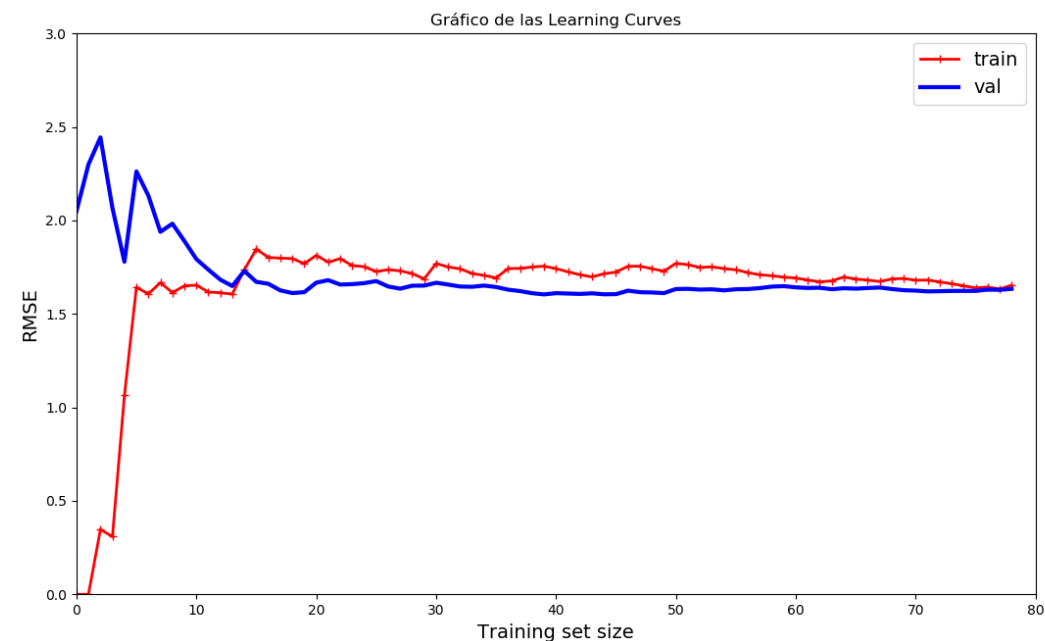




# Regresiones Polinómicas

- El problema es que ex ante no sabemos si tenemos overfitting o underfitting, una forma de comprender la capacidad de generalización del modelo fuera de la muestra por medio de las *learning curves*
- Esto es simplemente graficar el desempeño del modelo en un set de entrenamiento y el set de validación como función del tamaño del training set
- Para generar los gráficos, simplemente se debe entrenar el modelo muchas veces en sub sets de entrenamiento de diferentes tamaños:

```
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_predict, y_train[:m]))  
        val_errors.append(mean_squared_error(y_val_predict, y_val))  
  
    lin_reg = LinearRegression()  
    plot_learning_curves(lin_reg, X, y)
```



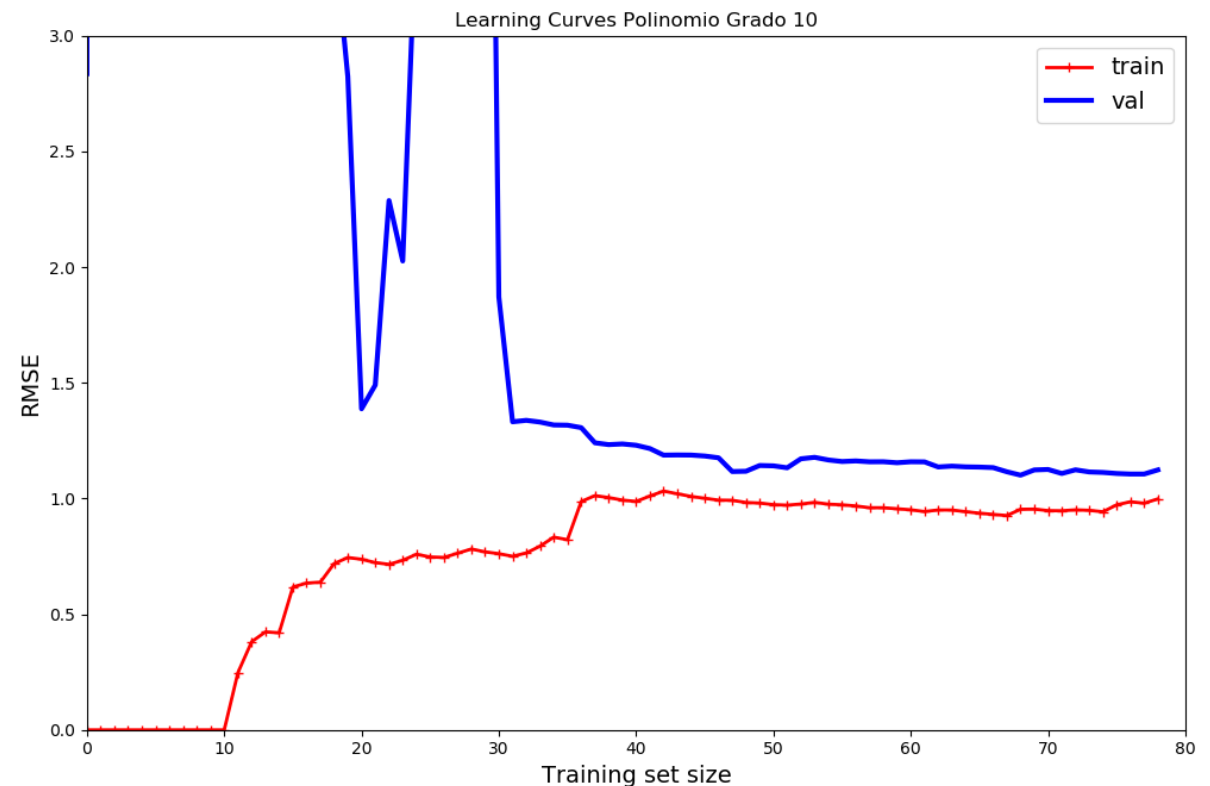
- En este caso tratamos de hacer fit de una regresión lineal
- Demos primero un vistazo a la data de entrenamiento. Cuando sólo hay una o dos instancias en el set de entrenamiento, el modelo puede ajustarlas perfectamente (por esto la curva parte en cero)
- Sin embargo cuando comenzamos a aumentar el tamaño del set de entrenamiento, comienza a ser imposible para el modelo ajustarse a los datos producto que la data es «ruidosa» y no lineal
- Dado esto el error comienza a incrementarse hasta llegar a un nivel donde permanece más o menos estable
- Por otra parte si miramos el modelo en la data de validación, cuando el modelo es entrenado con pocas instancias el modelo es incapaz de generalizar adecuadamente por esto el error de generalización es muy alto
- Luego cuando el modelo es sometido a más ejemplos de entrenamiento, éste aprende y su error en el set de validación comienza a ser menor convergiendo a un nivel bastante cercano al de la otra curva
- Esto es típico de un problema de underfitting donde ambas curvas convergen y se acercan en un nivel alto
- Si el modelo esta underfitting, el hecho de agregar más data no reducirá el error y sugiere que se requiere de un modelo más complejo

# Regresiones Polinómicas

- A continuación realizaremos el mismo ejercicio con un polinomio de grado 10:
- Las principales diferencias son que el error en la data de entrenamiento es mucho menor que el de la regresión lineal simple
- Existe un gap entre las curvas sugiriendo que el modelo tiene un desempeño significativamente mejor en la data de entrenamiento que en la data de validación lo cual es típico de un modelo con overfitting
- Sin embargo si se utiliza en un set de entrenamiento muy grande las curvas continuarían juntándose
- Una forma de mejorar un modelo con overfitting es alimentarlo con mas data de entrenamiento de modo que el error de validación alcance el error de entrenamiento

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

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



## El Bias/Variance Tradeoff

- Un importante resultado teórico de Estadísticas y Machine Learning es el hecho de que el error de generalización puede ser expresado como la suma de 3 muy diferentes errores:
- **1. Bias:** Corresponde a la parte del error de generalización se debe a supuestos errados, tales como asumir que la data es lineal cuando en la práctica es cuadrática. Un modelo con alto Bias es más probable de tener underfit en la data de entrenamiento
- **2. Variance:** Esta parte se debe a la excesiva sensibilidad del modelo a pequeñas variaciones en la data de entrenamiento. Un modelo con muchos grados de libertad (tales como un polinomio de alto grado) es probable que tenga una alta varianza y de éste modelo haga overfit de la data de entrenamiento
- **3. Error Irreducible:** Esta parte se debe al ruido de la data misma. La única forma de reducir esta parte del error es limpiar la data (e.g. arreglar las fuentes de datos, como por ejemplo sensores rotos o bien remover outliers)
- Normalmente incrementar la complejidad del modelo va a incrementar la varianza y reducir el bias. A La inversa reducir la complejidad del modelo aumenta el bias y reduce su varianza, de ahí el trade-off

- **Raíz del Error Cuadrático Medio (RMSE):** Esta métrica asigna mayor peso a los errores altos o outliers

$$RMSE(X, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(x^{(i)}) - y^{(i)})^2}$$

- **Error Absoluto Medio (MAE):** En ocasiones, sobretodo en presencia de varios outliers, puede ser preferible trabajar con otra métrica como el MAE

$$MAE(X, h) = \frac{1}{m} \sum_{i=1}^m |h(x^{(i)}) - y^{(i)}|$$

- Tanto el RMSE como el MAE son formas de medir distancia entre dos vectores, el vector de predicción y el vector de variables objetivo
- Varias métricas de distancia o «normas» son posibles, el RMSE corresponde a la Norma Euclideana y es la noción de distancia con la que estamos probablemente más familiarizados

## Revisitando Métricas de Performance

- La norma euclideana, o RMSE, también es conocida como la norma  $\ell_2$  o  $\| \cdot \|_2$
- Por su parte el MAE corresponde a la norma  $\ell_1$  o  $\| \cdot \|_1$ , o también denominada «Norma de Manhattan», pues mide la distancia entre dos puntos de la ciudad si sólo podemos movernos a través de los bloques o cuadras ortogonales de la ciudad
- Más generalmente la norma de orden «k» o  $\ell_k$  o  $\| \cdot \|_k$

$$\ell_k = \|v\|_k = \left( \sum_{i=0}^n (|v_i|^k) \right)^{\frac{1}{k}}$$

- De modo tal que  $\ell_0$  sólo entregaría el número de elementos distintos de cero en el vector, mientras que  $\ell_\infty$  entrega el máximo valor absoluto en el vector
- Mientras mayor es el índice de la norma, mayor es el foco que la métrica de distancia le da a los valores altos y tiende a no considerar los pequeños
- En la misma línea, ésta es la razón por la cual RMSE es más sensible a outliers que MAE, sin embargo cuando los outliers son muy pocos, RMSE tiene un muy buen desempeño y es preferido normalmente
- Una forma de reducir el overfitting es regularizar el modelo (es decir restringirlo)
- Para un modelo lineal una forma simple de regularizarlo es restringir los pesos del modelo

- **Ridge Regression (Regularización de Tikhonov):**

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

- El término que se suma a la función de costo hace que durante la fase de entrenamiento no sólo haga el fit de la data sino que mantiene los pesos tan pequeños como sea posible
- El término de regularización sólo se utiliza durante el entrenamiento del modelo, una vez que el modelo ya fue entrenado y se requiere evaluar su performance, se utiliza la medida de performance no regularizada
- Otro atributo deseable de la función de costo para entrenamiento es que tenga «derivadas» amigables, mientras que la medida de performance utilizada para testing debe enfocarse en llegar lo más cerca posible del objetivo
- El hiperparámetro  $\alpha$  controla el grado en que se quiere regularizar, si el parámetro es 0, estamos en el caso de la regresión lineal simple, si  $\alpha$  es muy alto luego todos los pesos terminan muy cerca de cero siendo el resultado final una línea plana pasando a travez de la media de los datos
- Notar que el término de bias  $\theta_0$  no está regularizado (la sumatoria parte en  $i=1$ ) para no bajar su peso en la regresión

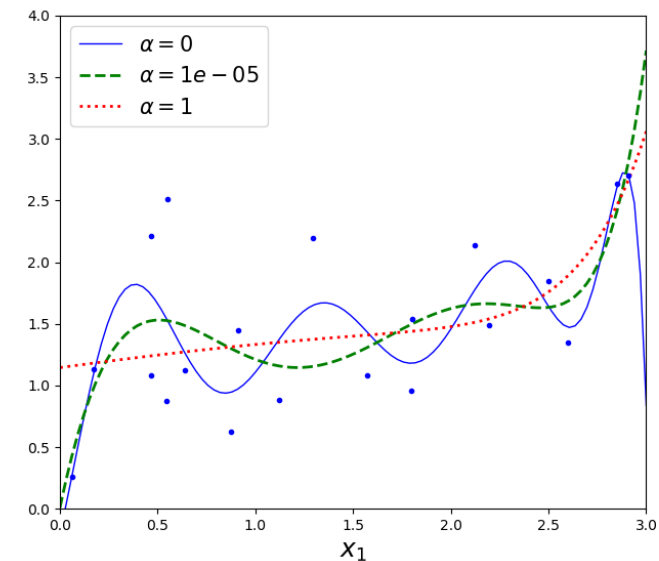
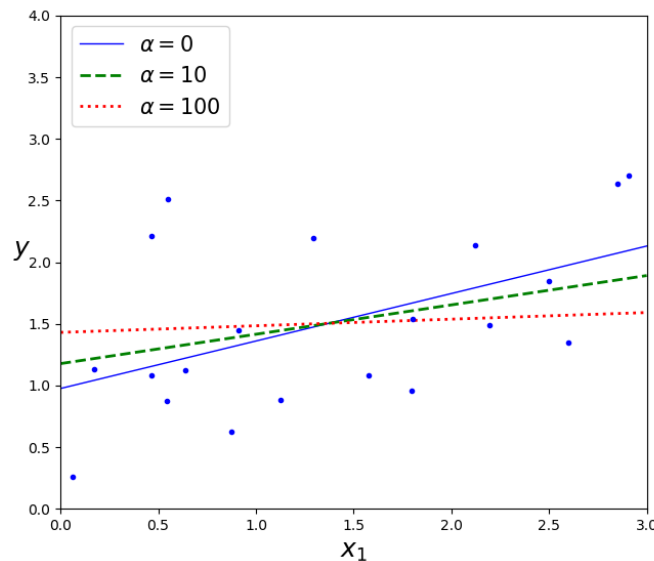
# Modelos Lineales Regularizados: Ridge Regression

- Es importante escalar o normalizar la data antes de aplicar Ridge Regression, lo cual también aplica a todos los modelos regularizados puesto que son sensible a la escala del input
- Otra buena noticia de la función de costo regularizada del Ridge Regression es que tiene solución cerrada con A la matriz identidad de nxn aunque con un cero en el elemento A(1,1) de modo tal que no se remueva o baje el peso del intercepto

$$\hat{\theta} = (X^T X + \alpha A)^{-1} X^T y$$

- Para resolver por gradiente descendente sólo hay que sumar  $\alpha W$  al vector gradiente de la medida MSE ya analizada en el modelo normal
- Los pros y contras de utilizar uno u otro método son los mismos de la ecuación normal

- El gráfico de la izquierda hace un fit de un modelo lineal y el de la derecha un polinomio de grado 10, para diferentes valores de  $\alpha$





# Modelos Lineales Regularizados: Ridge Regression con Scikit-Learn

- El término de regularización en el Ridge Regression corresponde a la mitad del cuadrado de la norma  $\ell_2$  del vector de parámetros

## Ejemplo 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]])
```

## Ejemplo SGD

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(penalty="l2")
sgd_reg.fit(X, y.ravel())
sgd_reg.predict([[1.5]])
```

- **Lasso Regression (*Least Absolute Shrinkage and Selection Operator Regression*):**
- En este caso, al igual que en Ridge Regression se agrega un penalty a la función de costo durante el proceso de entrenamiento
- Sin embargo en este caso se usa la regularización  $\ell_1$  en vez de la mitad del cuadrado de la norma  $\ell_2$

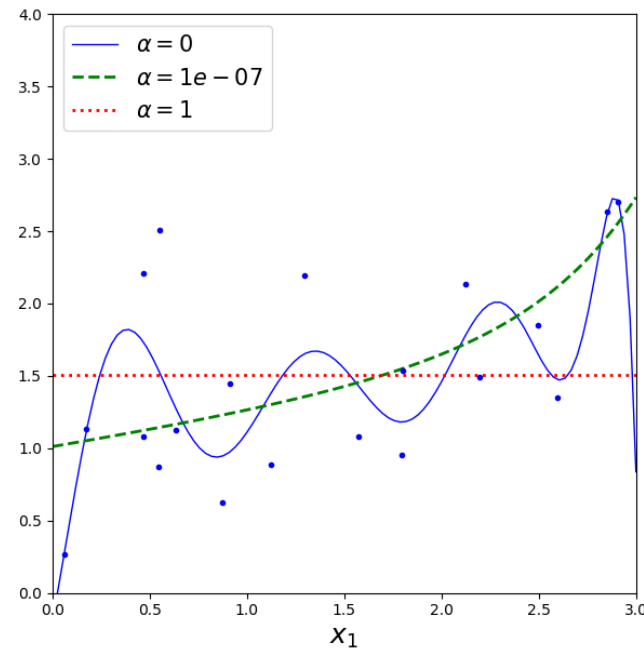
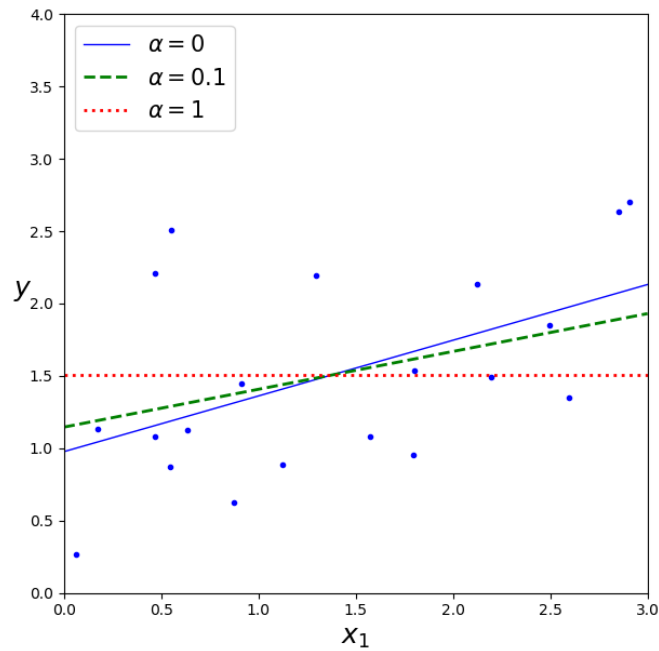
$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^n |\boldsymbol{\theta}_i|$$

- La característica más importante de Lasso Regression es que tiende a eliminar completamente el peso de los atributos menos importantes (les asigna peso cero), por ejemplo en el gráfico, cuando  $\alpha=10^{-7}$ , luce como una función cuadrática, casi lineal, lo que significa que todos los pesos para los atributos de alto grado son 0
- De otro modo, Lasso automáticamente ejecuta una selección de atributos

# Modelos Lineales Regularizados: Lasso Regression

- **Lasso Regression (*Least Absolute Shrinkage and Selection Operator Regression*):**

Ejemplo Scikit-Learn



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

- **Elastic Net Regression:**

- Elastic Net, es el «punto medio» entre el Ridge Regression y el Lasso Regression, pues el término de regularización es la combinación de ambos términos de regularización y se puede controlar la razón de mezcla «r»

$$J(\boldsymbol{\theta}) = MSE(\boldsymbol{\theta}) + r\alpha \sum_{i=1}^n |\boldsymbol{\theta}_i| + \frac{1-r}{2} \alpha \sum_{i=1}^n \boldsymbol{\theta}_i^2$$

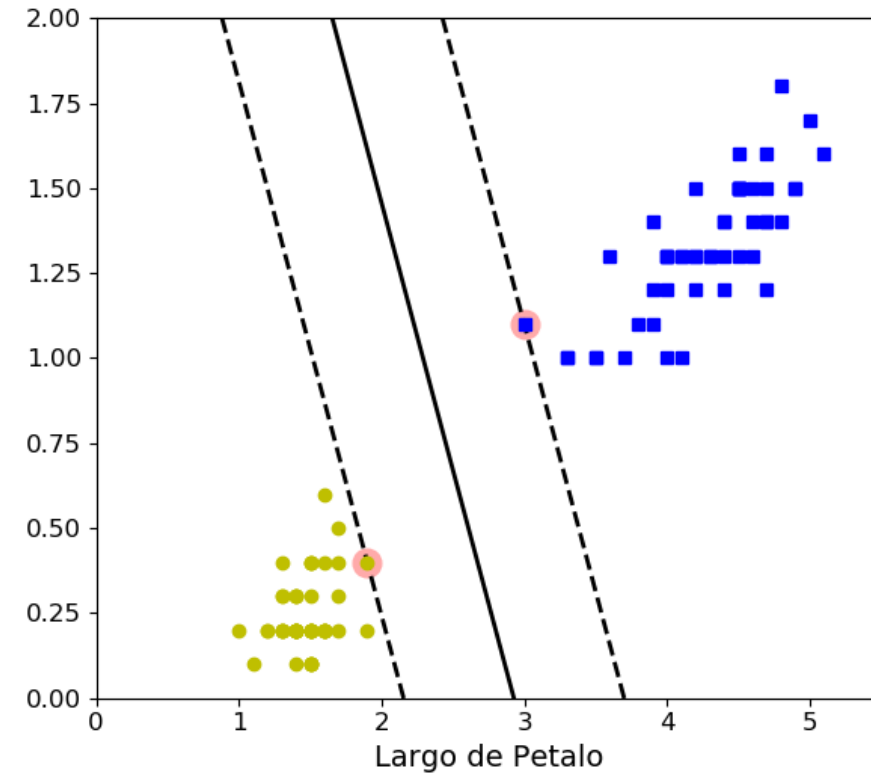
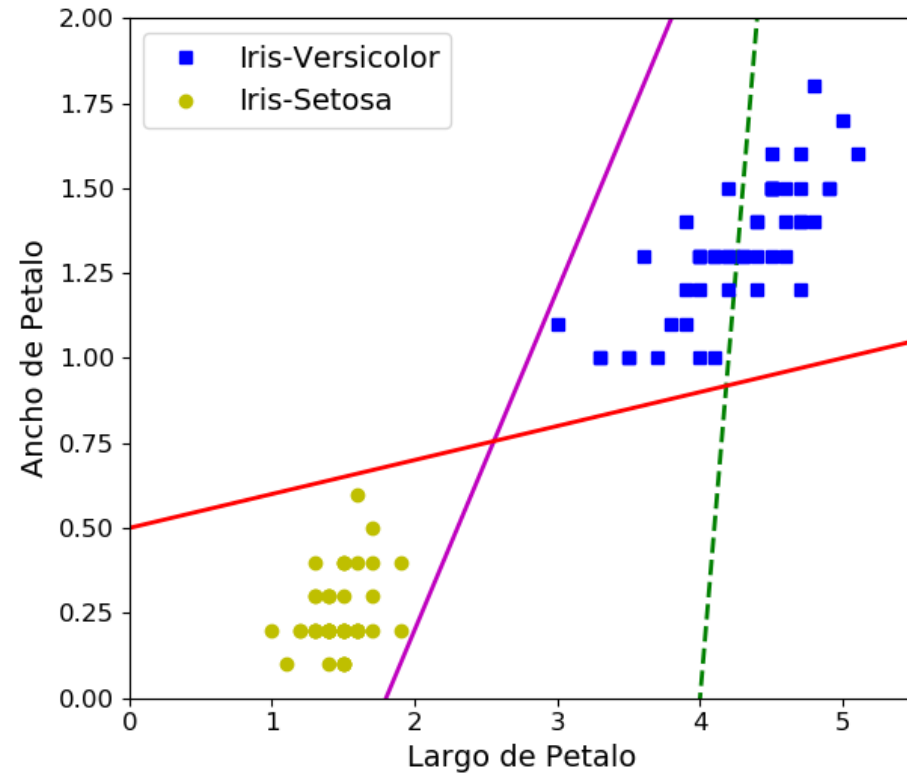
- Cuando  $r=0$ , Elastic Net es equivalente a Ridge Regression y cuando  $r=1$  es equivalente a Lasso Regression
- ¿Cuándo usar una regresión lineal simple, Ridge, Lasso o Elastic Net ?
- Es casi siempre preferible agregar algún grado de regularización
- Ridge Regression es un buen default, pero, si del análisis de los datos, uno tiene sospecha de que sólo algunos de los atributos son útiles, es preferible usar Lasso o Elastic Net dado que tienden a reducir el peso de los atributos inútiles
- En general Elastic Net es preferible por sobre Lasso ya que ésta última podría comportarse erráticamente cuando el número de atributos es mayor que las instancias de entrenamiento o cuando los atributos están fuertemente correlacionados

Ejemplo Scikit-Learn

(l1\_ratio es «r»)

```
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]])
```

- **SVM:** Interpretación Gráfica



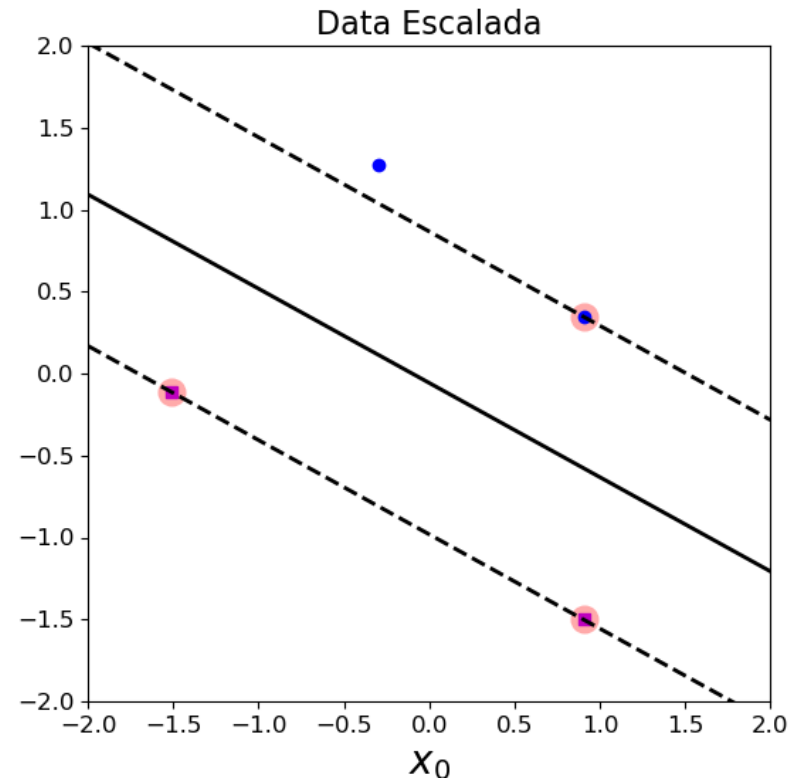
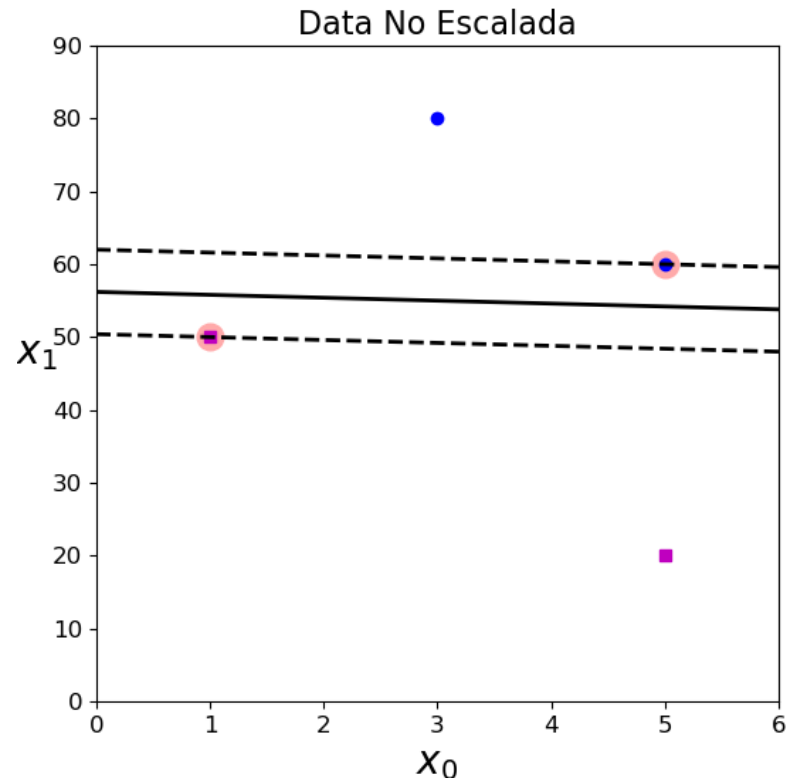
- **SVM:** Interpretación Gráfica
- En el ejemplo del gráfico las clases son separables claramente con una línea recta («linealmente separables»)
- El gráfico de la izquierda muestra las decisiones de borde de 3 posibles clasificaciones:
- El modelo representado por la línea discontinua es tan malo que incluso no separa las clases adecuadamente
- Los otros dos modelos trabajan perfectamente en este set de entrenamiento pero sus decisiones de borde están tan cerca de las instancias que esos modelos con una alta probabilidad no tendrán un buen desempeño clasificando nuevas instancias
- En contraste, la línea sólida del gráfico de la derecha representa la decisión de borde de un Support Vector Machine Classifier, SVC
- Esta línea no sólo separa las dos clases si no que se posiciona lo mas lejos posible de las instancias de entrenamiento más cercanas
- O de otro modo, un SVC, busca la «calle más ancha» (representadas por las líneas discontinuas paralelas) entre las clases
- Por lo anterior éste algoritmo es denominado como clasificación de amplio margen
- Notar que la agregación de nuevas instancias no afectará las condiciones de borde pues ésta está completamente determinada o «soportada» por las instancias ubicadas en el borde de la calle, las cuales son denominadas «vectores de soporte»

- **SVM:** Interpretación Gráfica
- En el ejemplo del gráfico las clases son separables claramente con una línea recta («linealmente separables»)
- Notar que los SVM's son sensibles a la escala de los atributos (o variables explicativas),
- En el gráfico puede observarse que si la escala vertical es mucho más grande que la escala horizontal, la calle más ancha posible es cercana a la horizontal
- Por lo anterior deben re escalarse los atributos (en Scikit-Learn puede utilizarse StandardScaler)

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(Xs)
svm_clf.fit(X_scaled, ys)
```

# Support Vector Machines

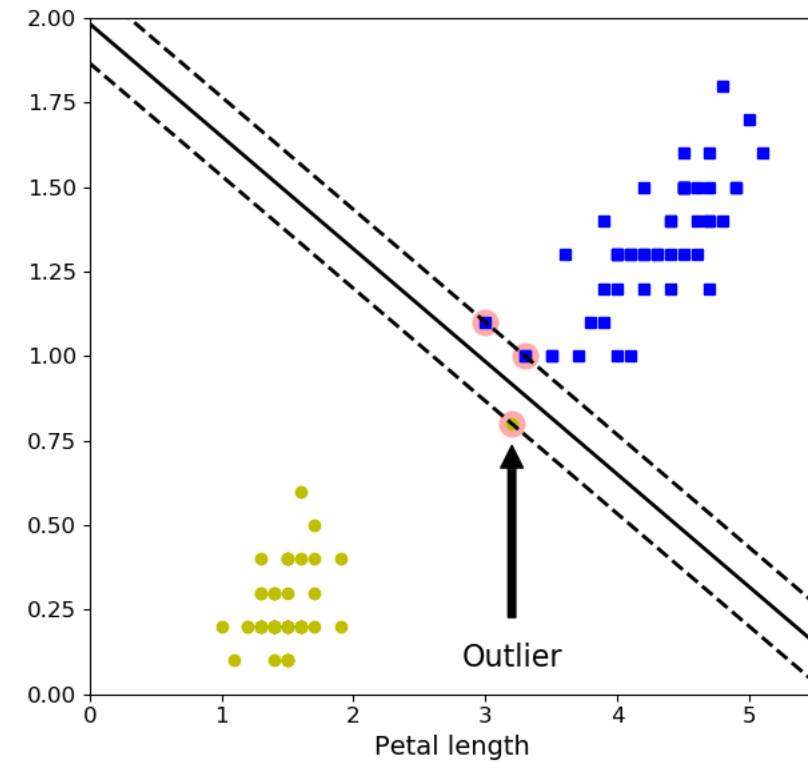
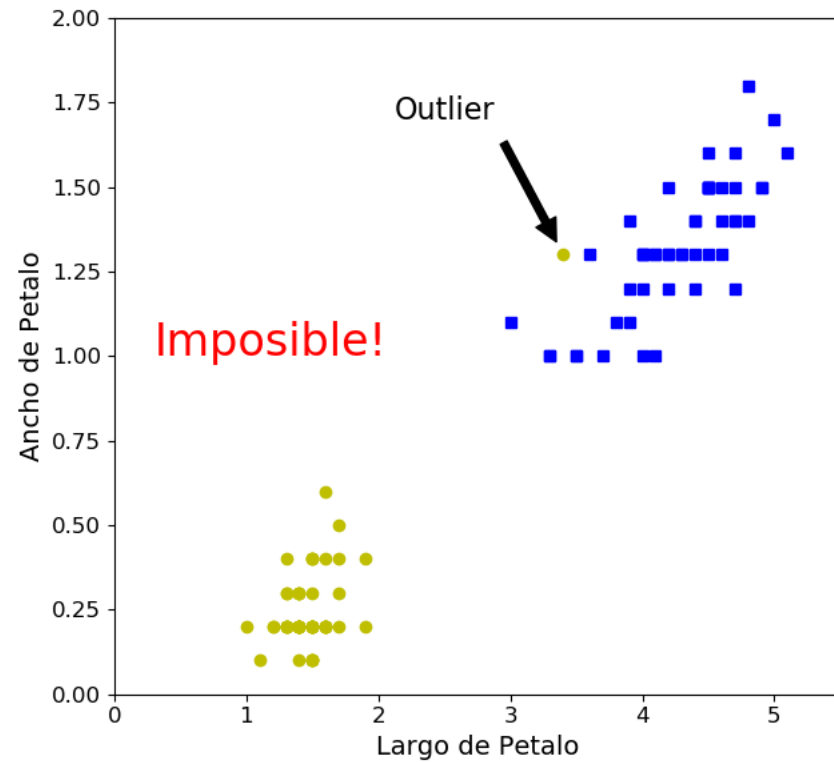
- **SVM:** Interpretación Gráfica





# Support Vector Machines

- **Soft Margin Classification:**
- Si imponemos estrictamente que todas las instancias estén fuera de la calle, esto es denominado «hard margin classification»



- **Soft Margin Classification:**

- Existen 2 problemas principales con la hard margin classification. Primero sólo funciona si la data es linealmente separable
- Segundo, es muy sensible a los outliers
- En el gráfico de la izquierda es imposible encontrar un hard margin y en la derecha la decisión de borde termina siendo muy diferente de la que vimos en el primer gráfico, que es esencialmente el mismo pero sin el outlier y probablemente no será capaz de generalizar cuando enfrente la clasificación de nuevas instancias
- Para evitar estos problemas es preferible utilizar un modelo más flexible
- El objetivo es encontrar un buen balance entre mantener la calle lo más ancha posible pero limitando las trasgresiones del margen (instancias que terminan al medio de la calle e incluso en la «vereda» equivocada)
- Esto es denominado **«soft margin classification»**
- En Scikit-Learn, es posible controlar este balance con el hiperparámetro «C»
- Un menor valor de C lleva a una calle más ancha pero más trasgresiones de margen
- Si un modelo está haciendo SVM esta haciendo overfitting es posible regularizarlo reduciendo el parámetro C
- En el caso de los SVM, a diferencia de la regresión logística no genera como output las probabilidades para cada clase dado un vector de atributos

- **Soft Margin Classification:**
- Ejemplo con C=1 (gráfico de la izquierda)

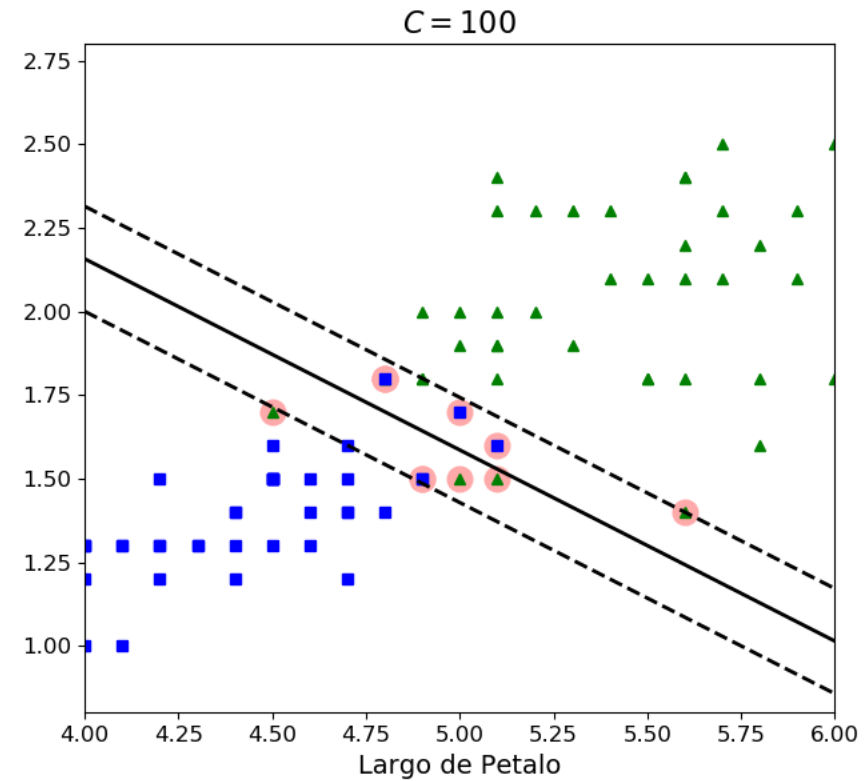
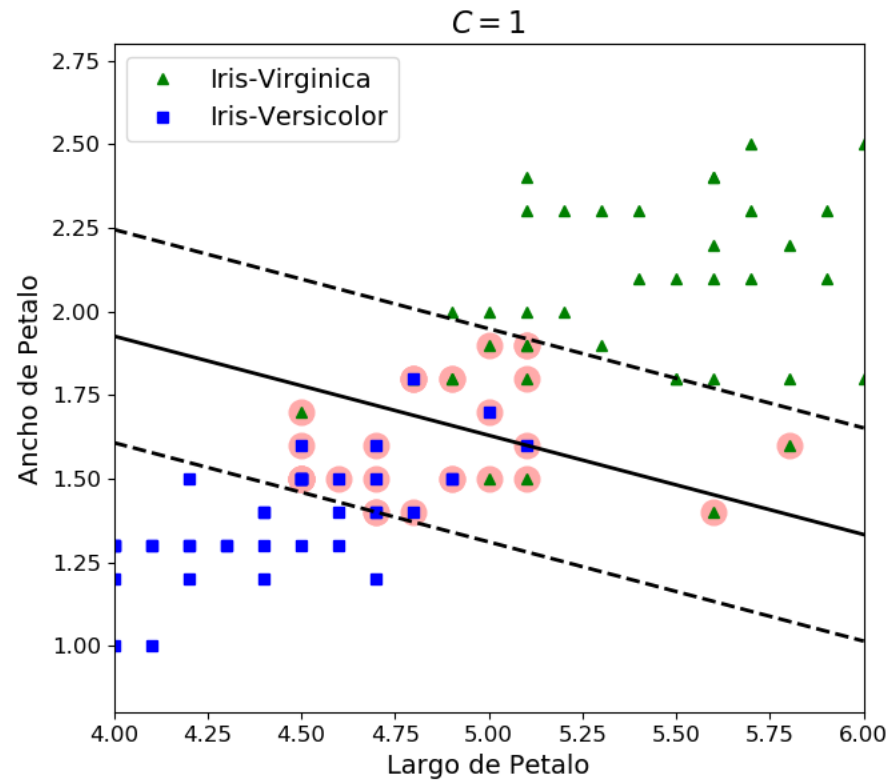
```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # Largo de Petalo y Ancho de Petalo
y = (iris["target"] == 2).astype(np.float64) # Iris-Virginica

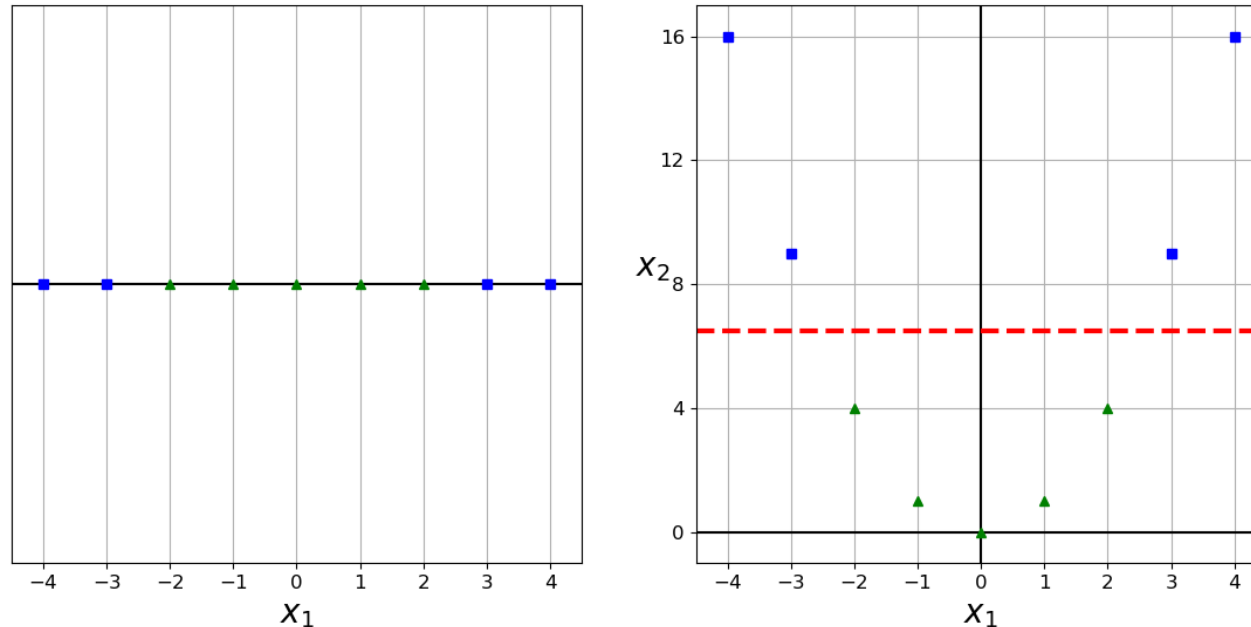
svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("linear_svc", LinearSVC(C=1, random_state=42)),])

svm_clf.fit(X, y)
svm_clf.predict([[5.5, 1.7]])
array([ 1.]
```

- Soft Margin Classification:



- **SVM y Clasificación no Lineal :**
- Aunque los SVM son eficientes y trabajan muy bien, muchos datasets están muy lejos de ser linealmente separables
- Una forma de lidiar con esto es agregar mas atributos como Polinomial Features, en algunos casos esto podría devolvernos al caso linealmente separable
- En la figura se muestra un data set con un solo atributo  $x_1$ , no linealmente separable sin embargo si agregamos como segundo atributo  $x_2 = x_1^2$ , el dataset es linealmente separable

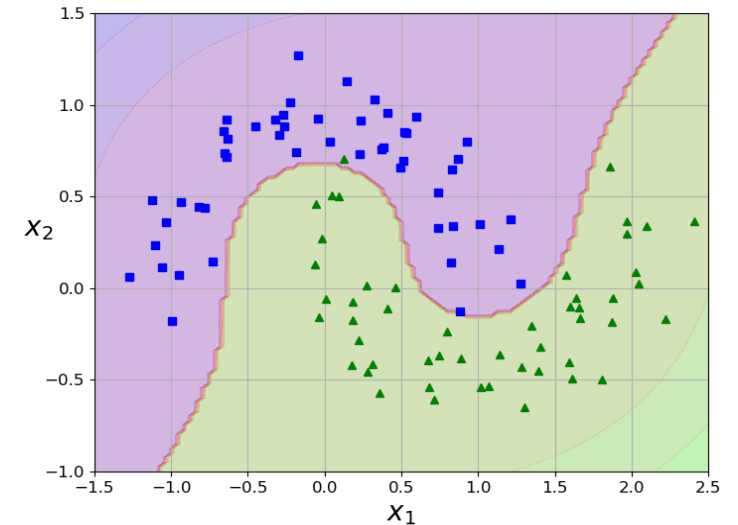
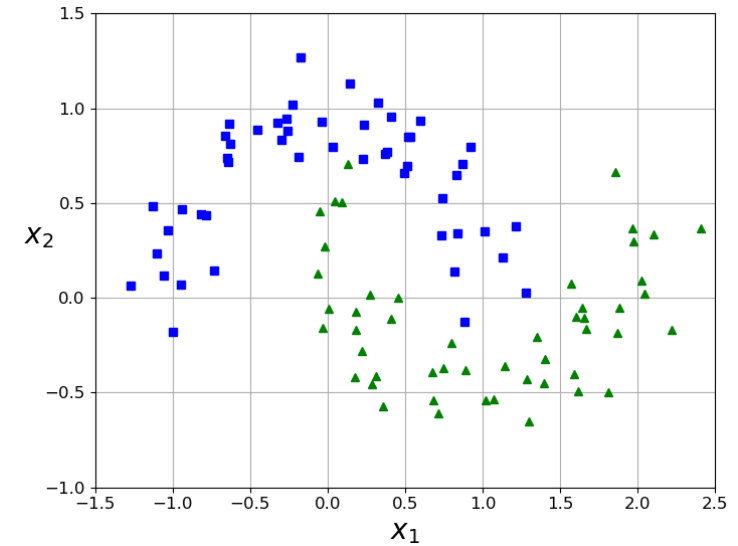


- **SVM y Clasificación no Lineal** : Ejemplo Scikit-Learn agregando como nuevos atributos hasta un polinomio cúbico

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, random_state=42))]

polynomial_svm_clf.fit(X, y)
```

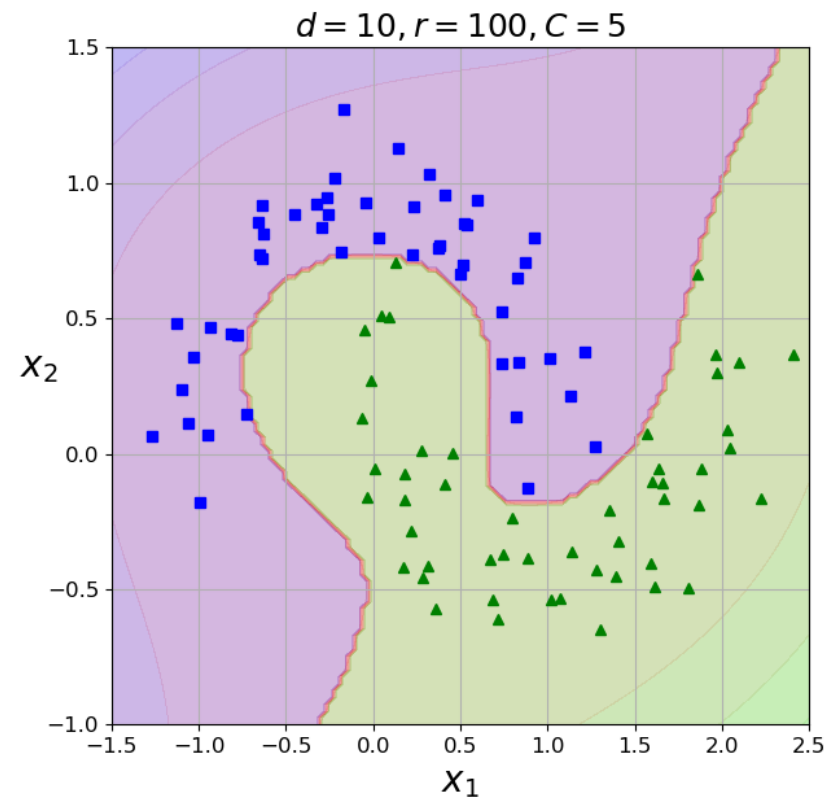
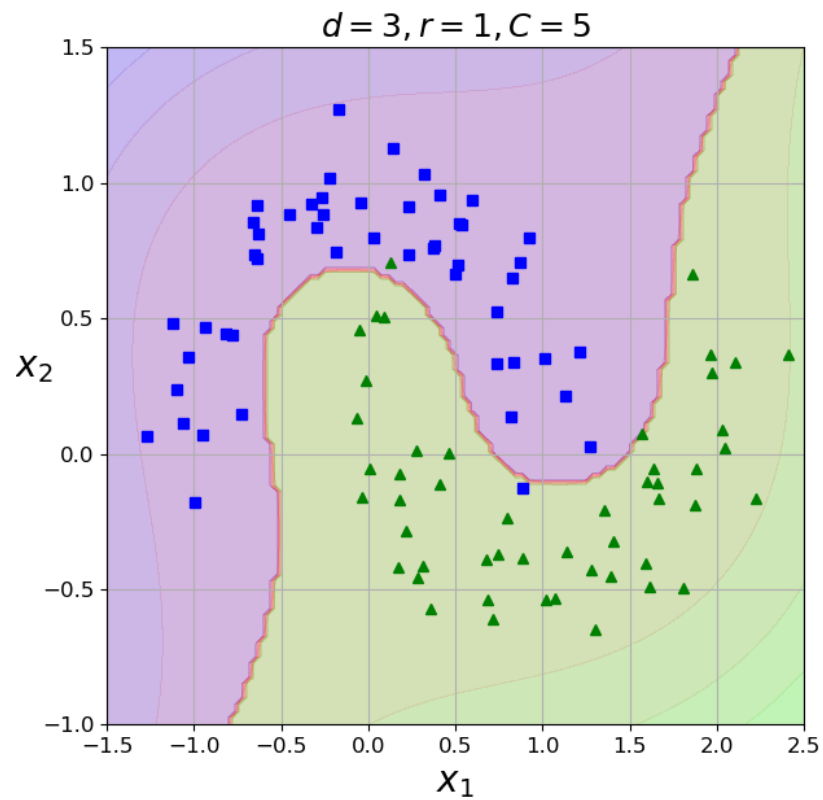


- **Kernel Polinómico:**
- Agregar atributos polinómicos es imple de implementar y funciona bien con todos los tipos de SVM's en un grado bajo de polinomio no siempre puede lidiar con datasets complejos y un alto grado grado de polinomio puede crear una enorme cantidad de atributos haciendo el modelo ineficiente
- Sin embargo cuando trabajamos con SVM's podemos utilizar una técnica matemática denominada el «truco del kernel», la cual posibilita obtener los mismos resultados como si hubieran sido adheridos muchos atributos polinómicos incluso de alto grado sin realmente haber sido adherirlos al set de atributos evitando, de este modo, la explosión combinatoria
- Ejemplo en scikit-learn: Entrenamiento de un clasificador SVM utilizando un kernel polinómico de 3° grado

```
from sklearn.svm import SVC

poly_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))]
poly_kernel_svm_clf.fit(X, y)
```

- Kernel Polinómico:





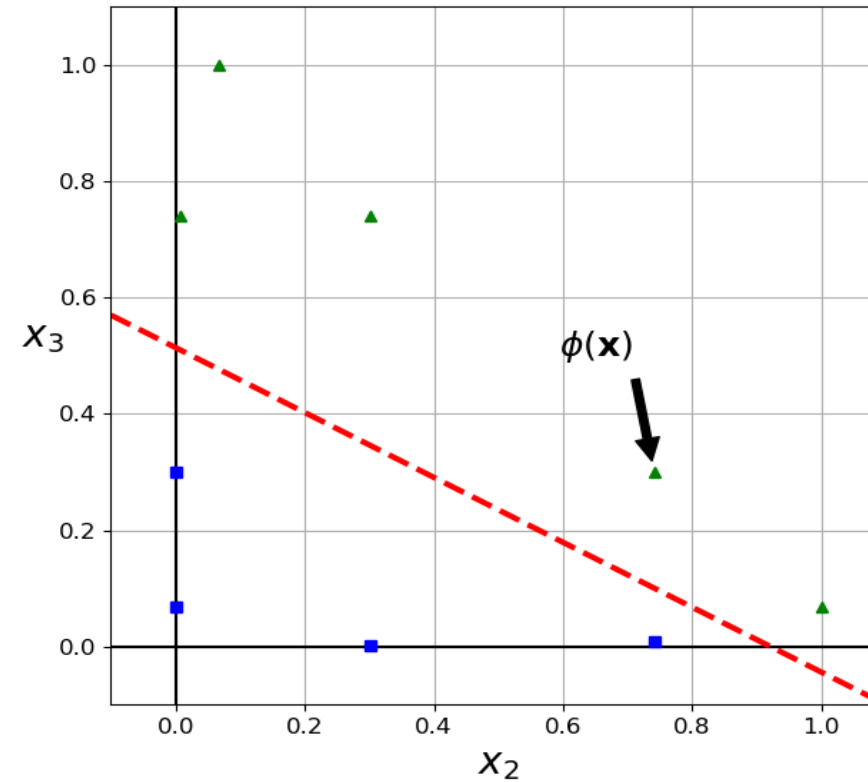
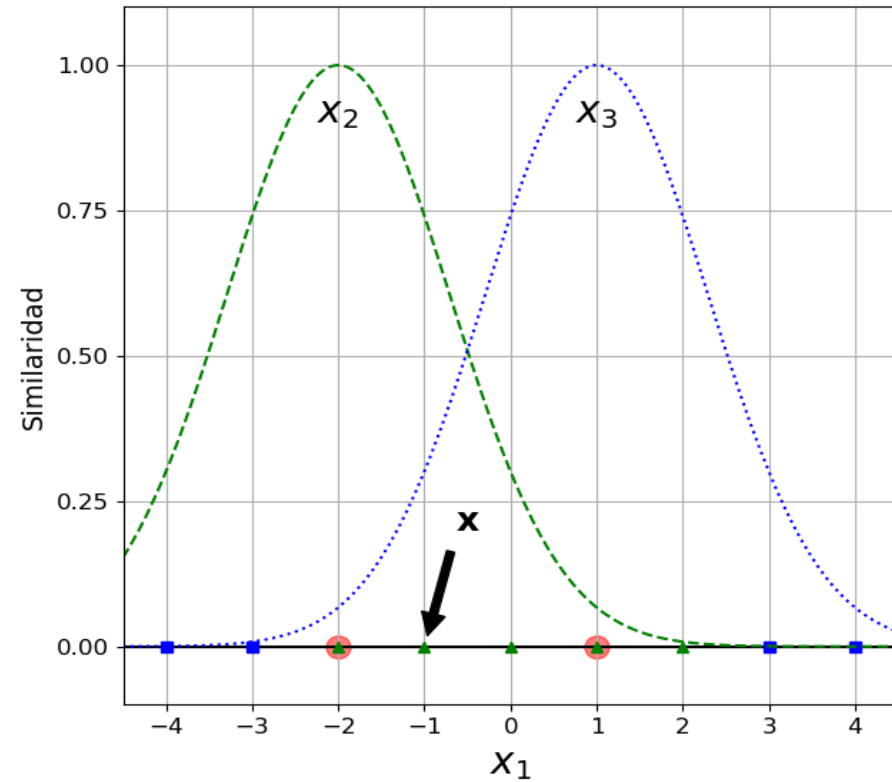
- **Kernel Polinómico:**
- Obviamente si hay evidencia de overfitting o underfitting podría ser deseable reducir o aumentar respectivamente el grado del polinomio, para esto, más adelante utilizaremos una técnica denominada grid search, que permite encontrar el set óptimo de parámetros para un determinado algoritmo
- **Similaridad:**
- Otra técnica para lidiar con problemas no lineales es agregar atributos usando una «función de similaridad» que mide el grado en que cada instancia se parece a cada medida de referencia o «landmark»
- Por ejemplo volvamos sobre el ejemplo del dataset de una dimensión y agreguemos 2 landmarks  $x_1=-2$  y  $x_1=1$
- Posteriormente definimos una función de similaridad por ejemplo la Función Gaussiana de Base Radial (RBF) con  $\gamma=0.3$

$$\phi(X, \ell) = \exp(-\gamma \|X - \ell\|^2)$$

- La cual es una función con forma de campana que varia desde 0 (muy lejos del landmark) a 1 (en el landmark)

# Support Vector Machines

- **Similaridad:**



- **Similaridad:**

- Por ejemplo miremos la instancia  $x_1 = -1$ , está localizada a una distancia de 1 del primer landmark y 2 del segundo landmark, por lo tanto sus nuevas features son  $x_2 = \exp(-0.3x_1^2) \approx 0.74$  y  $x_3 = \exp(-0.3x_2^2) \approx 0.30$
- La figura de la derecha muestra la dataset transformada (eliminando los valores originales) como se puede observar, ahora es linealmente separable
- ¿Cómo elegir los landmarks ?, el procedimiento más simple es crear un landmark en la ubicación de todas y cada una de las instancias en el dataset
- Esto genera muchas dimensiones así incrementando las probabilidades de que los datos transformados sean linealmente separables
- El downside de esto es que un training set con  $m$  instancias y  $n$  atributos se torna en un training set de  $m$  instancias y  $m$  atributos (asumiendo que se eliminan los atributos originales)
- Si el training set es muy grande se terminará con un igualmente grande número de atributos

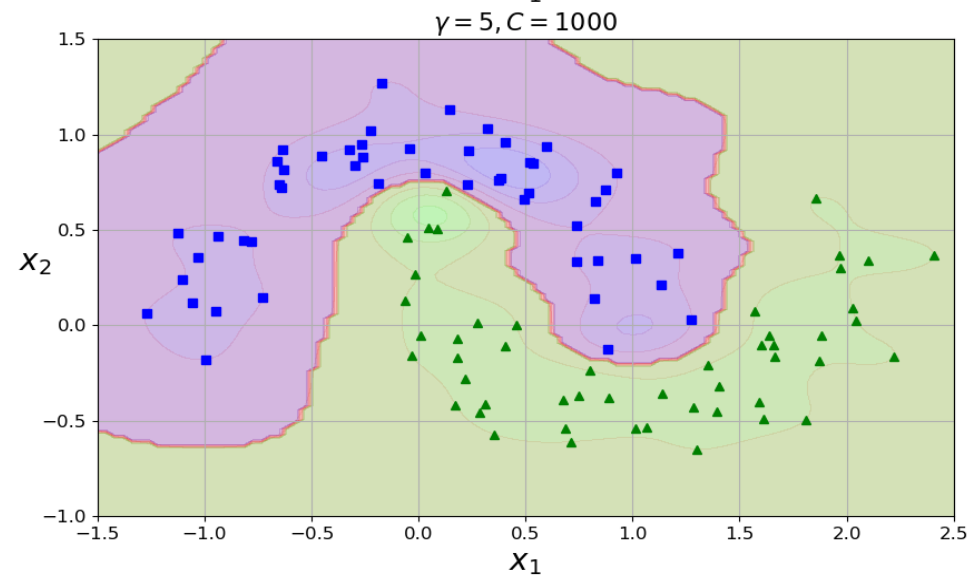
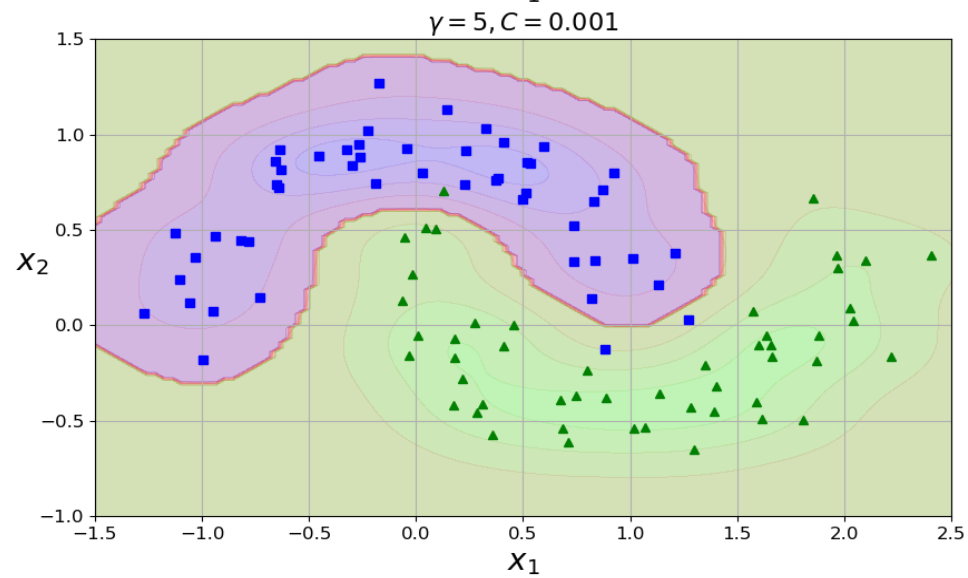
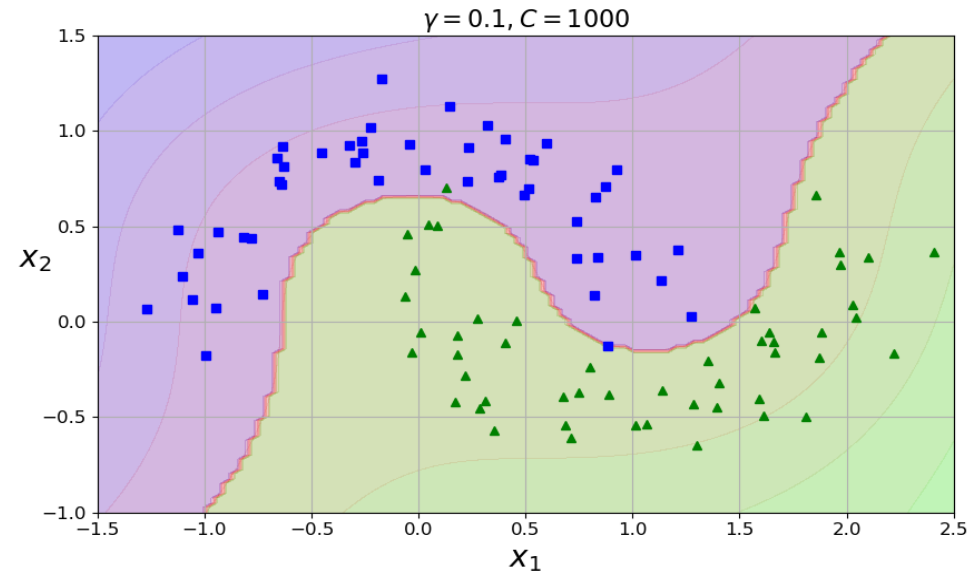
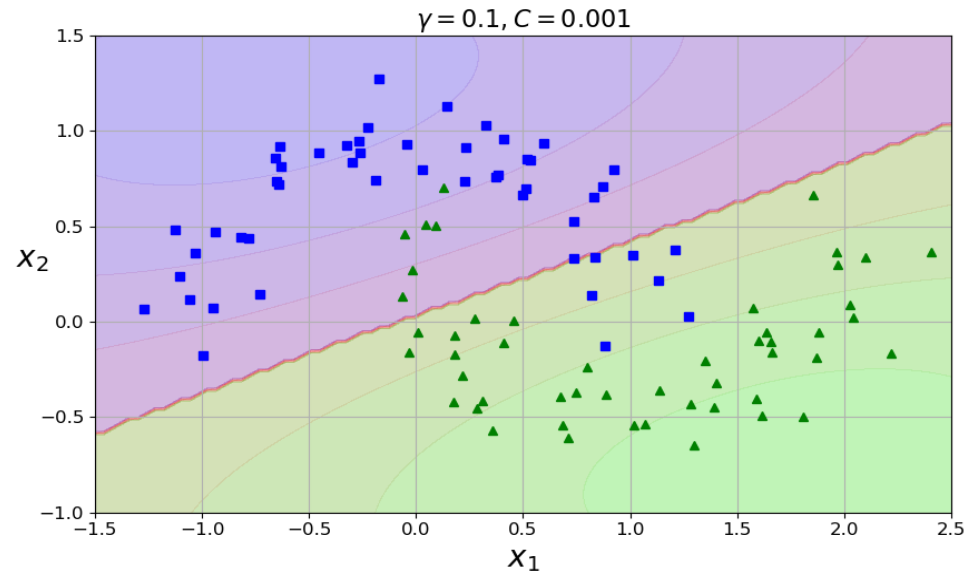
- **RBF Gaussian Kernel:**

- De igual modo como los atributos polinómicos, los atributos de similaridad pueden ser útiles con cualquier algoritmo de Machine Learning pero puede ser computacionalmente caro calcular todos los atributos adicionales especialmente en grandes training sets
- Sin embargo nuevamente podemos aplicar el truco del kernel permite obtener tal como si se hubiesen agregado muchas medidas de similaridad sin realmente agregarlas

```
rbf_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))]
rbf_kernel_svm_clf.fit(X, y)
```

- Éste modelo está representado en la figura de abajo a la izquierda, los otros gráficos muestran modelos entrenados con diferentes valores para los hiperparámetros gamma ( $\gamma$ ) y C
- Mayores valores de gamma hace la campana mas angosta y como resultado cada instancia del rango de influencia es menor: la función de borde de decisión termina siendo más irregular moviéndose en torno de las instancias individuales
- Por otra parte un gamma pequeño hace la curva mas ancha, de modo tal que las instancias tienen un mayor rango de influencia y la función de decisión termina siendo más suave
- De modo tal que  $\gamma$  actúa como un hiperparámetro de regularización: Si el modelo da señales de overfitting  $\gamma$  debe ser reducido mientras que si da señales de underfitting el parámetro debería ser incrementado (similar al hiperparámetro C)

# Support Vector Machines



- **El Truco del Kernel:**

- Supongamos que queremos aplicar una transformación polinómica de 2º grado a un training set de 2 dimensiones para luego aplicar un SVC
- La expresión de abajo muestra la función de mapping a aplicar:

$$\Phi(X) = \Phi \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

- Notar que el vector transformado es de 3 dimensiones en vez de las 2 originales, ahora en términos de los 2 vectores bidimensionales  $a=(a_1 \ a_2)$  y  $b=(b_1 \ b_2)$ , verifiquemos que ocurre con  $\Phi(a)^T \Phi(b)$  (término siempre presente en las funciones de costo):

$$\Phi(a)^T \Phi(b) = \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2$$

$$\Phi(a)^T \Phi(b) = \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = (a_1b_1 + a_2b_2)^2 = \left( \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (a^T b)^2$$

- **El Truco del Kernel:**

- ¿Que ganamos con esto ?
- El producto punto de los vectores transformados es igual al producto punto de los vectores originales:

$$\Phi(a)^T \Phi(b) = (a^T b)^2$$

- Esta es la clave del truco del kernel, si aplicamos la transformación  $\Phi$  a todas las instancias, el problema de optimización contendría el termino:

$$\Phi(x^{(i)})^T \Phi(x^{(j)})$$

- Si asumimos el caso particular de un polinomio de 2º grado, podemos reemplazar este producto punto de la siguiente manera:

$$\Phi(x^{(i)})^T \Phi(x^{(j)}) = (x^{(i)T} x^{(j)})^2$$

- Con esto se evita toda transformación de la data teniendo una ganancia relevante en la eficiencia computacional

- **El Truco del Kernel:**
- La función de abajo se denomina «kernel polinómico de 2° grado»

$$K(a, b) = (a^T b)^2$$

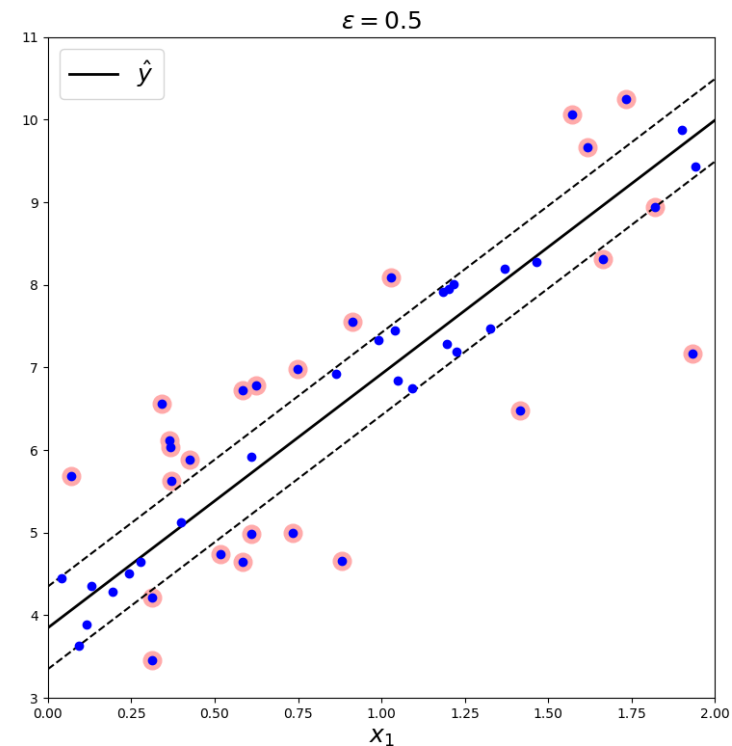
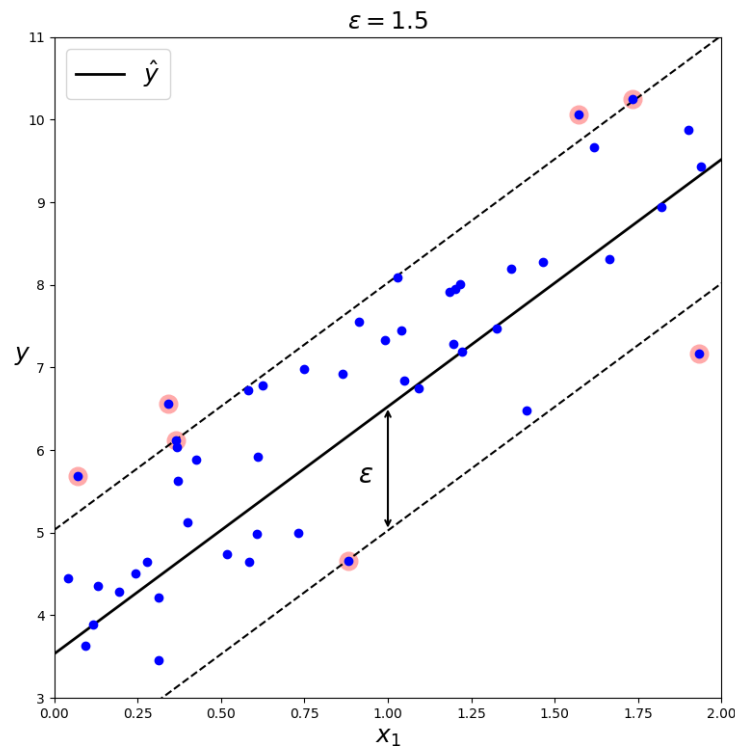
- En Machine Learning , un kernel, es una función capaz de calcular el producto punto  $\Phi(a)^T \Phi(b)$  basada sólo en los vectores originales  $a$  y  $b$  sin la necesidad de calcular la transformación  $\Phi$
- Kernels más comunes:

<i>Lineal</i>	:	$K(a, b) = a^T b$
<i>Polinómico</i>	:	$K(a, b) = (\gamma a^T b + r)^d$
<i>Gaussian RBF</i>	:	$K(a, b) = \exp(\gamma \ a - b\ ^2)$
<i>Sigmoideo</i>	:	$K(a, b) = \tanh(\gamma a^T b + r)$



# Support Vector Machines

- **Support Vector Machines y Regresión SVR:**
- SVM no sólo soporta clasificación lineal y no lineal, si no que también soporta regresiones lineales y no lineales
- El truco es invertir el objetivo: en vez de encontrar la mayor posible calle entre dos clases limitando las transgresiones de margen, SVR trata de ajustar tantas instancias posibles como sea posible limitando las transgresiones de margen
- El largo de la calle es controlado por un hiperparámetro  $\epsilon$ .



- **Support Vector Machines y Regresión SVR:**

- La agregación de más instancias de entrenamiento dentro del margen no afectará las predicciones del modelo, así el modelo se denomina «e-insensitive»

```
from sklearn.svm import LinearSVR

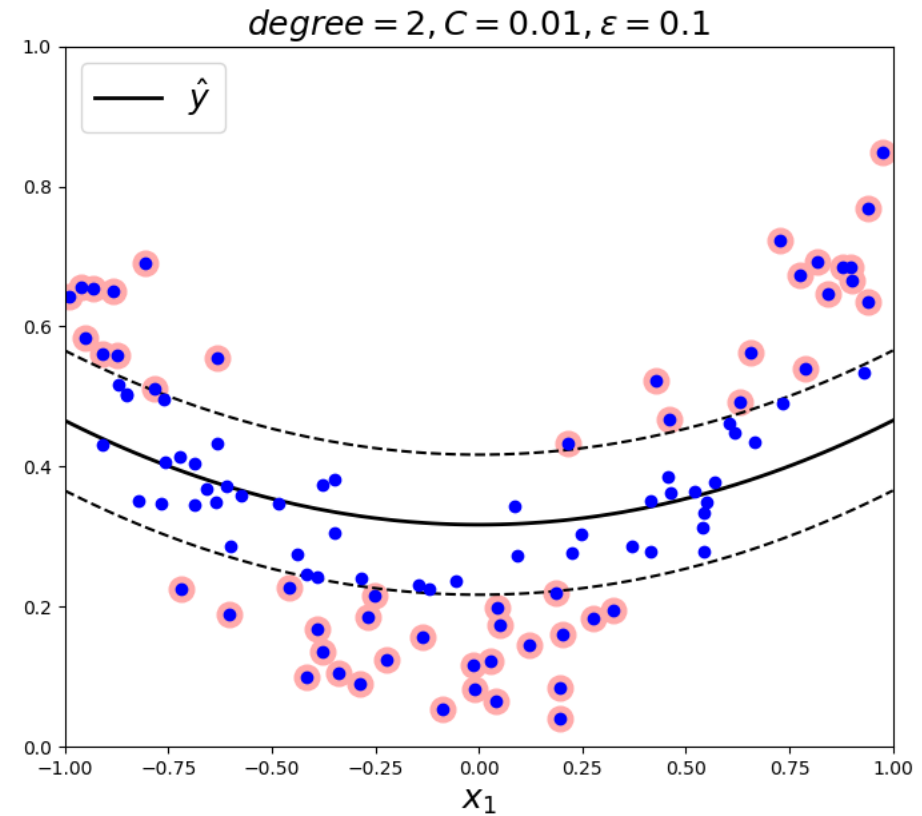
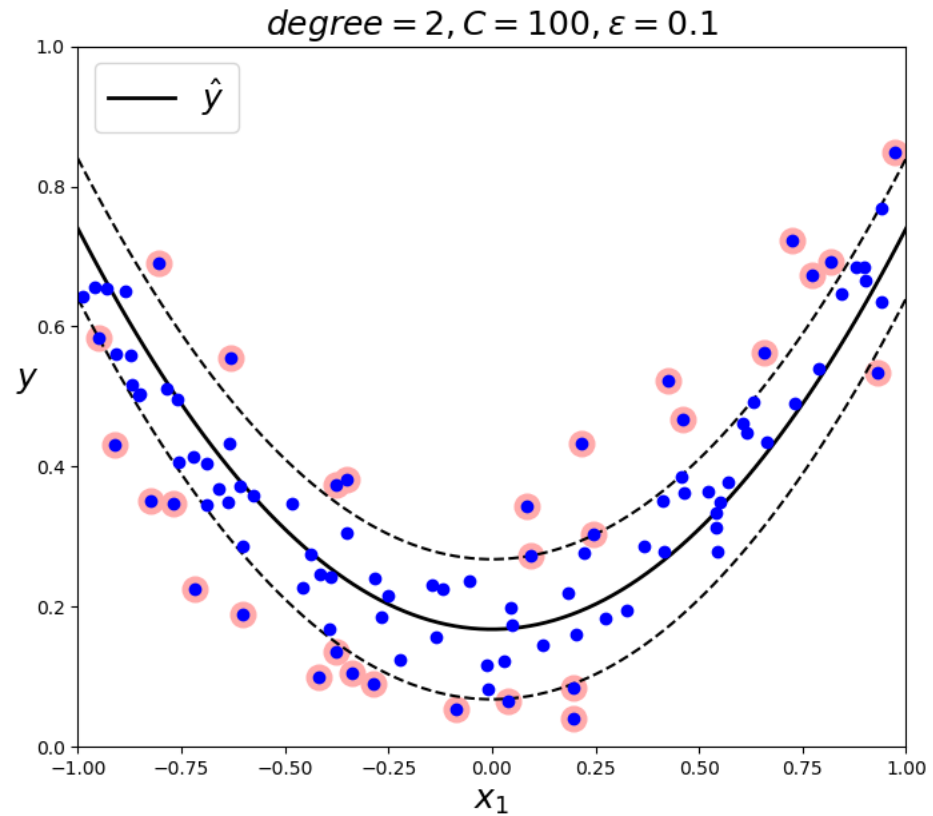
svm_reg = LinearSVR(epsilon=1.5, random_state=42)
svm_reg.fit(X, y)
```

- El código de arriba corresponde a la figura de la izquierda (los datos deben ser escalados y centrados antes de hacer el ajuste del modelo)
- Para abordar tareas no lineales es posible utilizar los modelos con kernel
- La figura muestra una SVR sobre un training set cuadrático usando un kernel polinómico de 2° grado con poca regularización en el gráfico de la izquierda (un valor de C alto) y mucha mas regularización en el gráfico de la derecha (C bajo)

```
from sklearn.svm import SVR

svm_poly_reg1 = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg2 = SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1)
svm_poly_reg1.fit(X, y)
svm_poly_reg2.fit(X, y)
```

- Support Vector Machines y Regresión SVR:

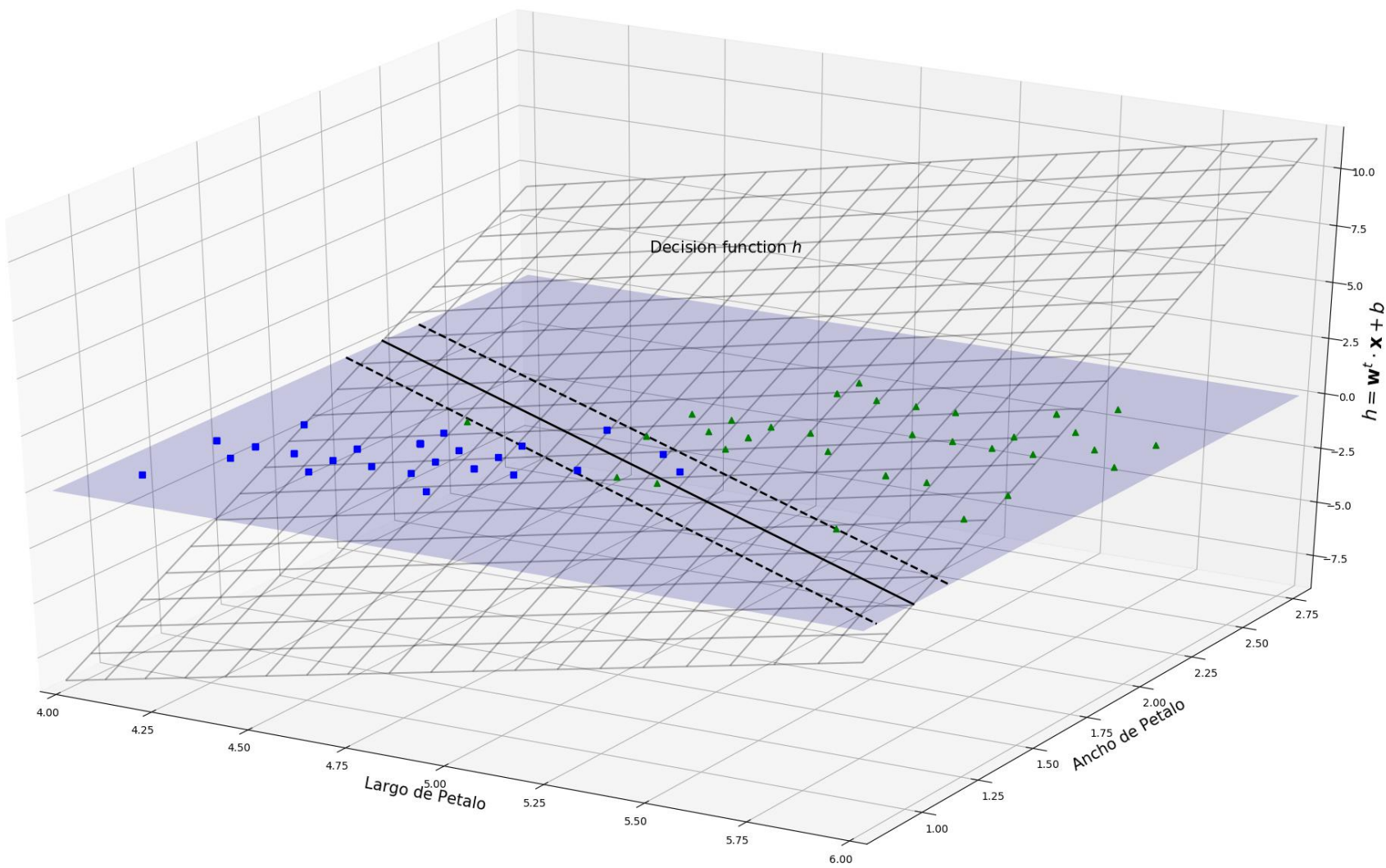
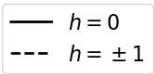


- **SVM Funciones de Decisión y Predicciones:**

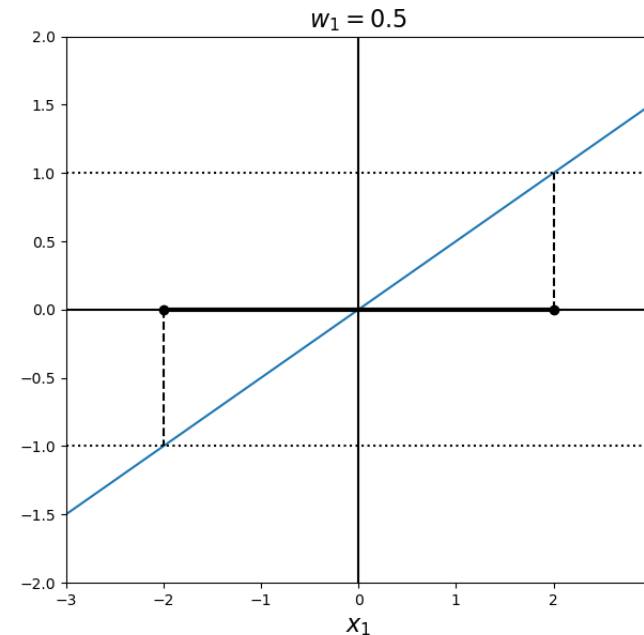
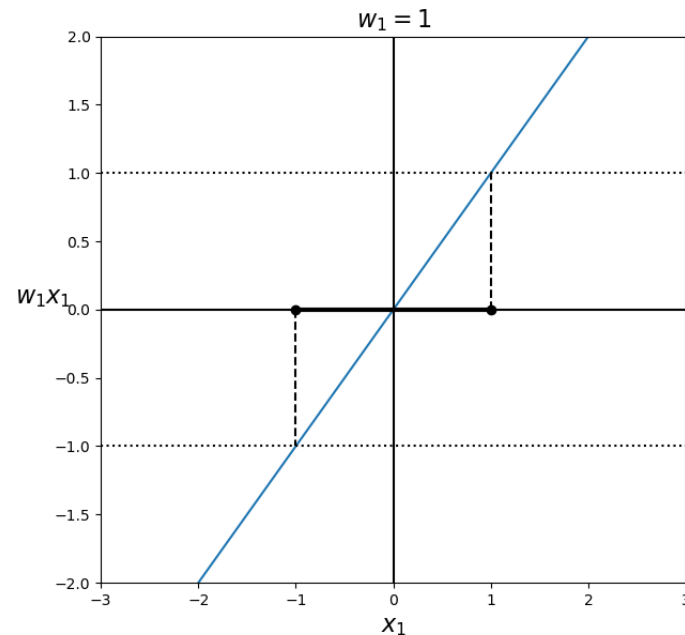
$$\hat{y} = \begin{cases} 0 & \text{si } W^T X + b < 0 \\ 1 & \text{si } W^T X + b \geq 0 \end{cases}$$

- Donde  $W^T$  es el vector de parámetros y  $b$  el término de bias
- La decisión de borde es el set de puntos donde la función de decisión es igual a 0, es decir la intersección de los 2 planos (representado en la línea recta sólida)
- Las líneas discontinuas representan los puntos donde la función de decisión es 1 o -1: ellos son paralelos y a igual distancia de los bordes de decisión formando un margen alrededor de él
- El entrenamiento de un SVM lineal implica encontrar los valores de  $W$  y  $b$  que hacen este margen lo más amplio posible evitando trasgresiones del margen (hard margin) o bien limitandolas (soft margin)

# Support Vector Machines



- **SVM y Objetivos de Entrenamiento:**
- Considere la pendiente de la función de decisión, esta corresponde a la norma del vector de pesos,  $\|W\|$
- Si dividimos esta pendiente por 2, los puntos donde la función de decisión es igual a  $\pm 1$  estarán dos veces mas lejos de la función de decisión
- O en otras palabras dividir la pendiente por 2 multiplicará el margen por 2, visualicemos en 2D:



- **SVM y Objetivos de Entrenamiento:**

- De este modo queremos minimizar  $\|W\|$ , de modo de obtener un margen amplio, sin embargo si queremos evitar cualquier transgresión de margen (hard margin), necesitamos que la función de decisión sea mayor que 1 para todas las instancias de entrenamiento positivas y menor que -1 para todas las instancias de entrenamiento negativas

- Si definimos:

$$t = \begin{cases} t^{(i)} = -1 & \text{para instancias negativas } (y^{(i)} = 0) \\ t^{(i)} = +1 & \text{para instancias positivas } (y^{(i)} = 1) \end{cases}$$

- Luego, podemos expresar esta restricción como:

$$t^{(i)}(W^T X^{(i)} + b) \geq 1 \text{ para todas las instancias}$$

- **SVM y Objetivos de Entrenamiento:**
- Podemos por lo tanto expresar el problema (hard margin) SVM lineal como el siguiente problema de optimización restringido:

$$\underset{W,b}{\text{minimize}} \quad \frac{1}{2} W^T W$$

$$\text{subject to:} \quad t^{(i)} (W^T X^{(i)} + b) \geq 1, \quad \text{for } i=1,2,\dots,m$$

- Utilizamos, como es habitual,  $(1/2)$  del cuadrado de la norma  $l_2$  en vez de la norma  $l_2$  directamente pues llegan al mismo resultado presentando derivadas más simples
- En efecto, por ejemplo, la norma  $l_1$  no es diferenciable en  $w=0$ ,
- Lo anterior hace más simple el problema de optimización
- Asimismo, el problema equivalente de de soft margin corresponde simplemente a una forma regularizada de lo anterior



- **SVM y Objetivos de Entrenamiento:**
- El parámetro  $\xi^{(i)}$ , corresponde a una variable de holgura para cada instancia (i), el cual mide cuanto la i-ésima instancia tiene permitido transgredir el margen
- Con esto, en la versión de soft margin, surgen 2 objetivos «en conflicto», pues queremos hacer la variable de holgura tan pequeña como sea posible para reducir las transgresiones del margen y hacer  $\frac{1}{2}W^TW$  tan pequeño posible para incrementar el margen
- Acá es cuando aparece el hyperparámetro C, pues permite definir el tradeoff entre ambos objetivos, con lo que llegamos al siguiente problema de optimización:

$$\begin{aligned} & \underset{W,b}{\text{minimize}} \quad \frac{1}{2}W^TW + C \sum_{i=1}^m \xi^{(i)} \\ & \text{subject to:} \quad t^{(i)} (W^TX^{(i)} + b) \geq 1 - \xi^{(i)} \text{ and } \xi^{(i)} \geq 0, \text{ for } i=1,2,\dots,m \end{aligned}$$