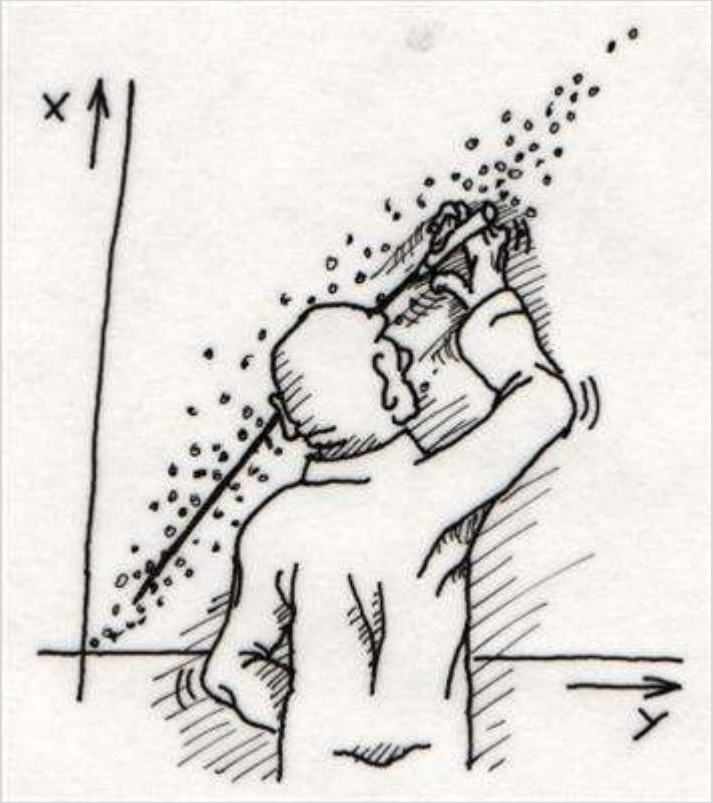


DigitalHouse > Coding School

DATA SCIENCE

MÓDULO 3

Hiperparámetros
y
Model Validation

Pensando en Model Validation

Receta básica para aplicar un modelo de machine learning supervisado:

1. Elegir una clase de modelo
2. Elegir los hiperparámetros del modelo
3. Ajustar el modelo a los datos de entrenamiento
4. Usar el modelo para predecir etiquetas para nuevos datos

¿Qué necesitamos para realizar una elección informada?

Necesitamos una manera de *validar* que nuestro modelo y nuestros hiperparámetros representan un buen **fit** (ajuste) para los datos y que va a tener un **buen poder de generalización** con datos nuevos.

Consideremos el siguiente procedimiento::

In [1]:

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
```

Luego **elegimos un modelo y sus hiperparámetros.** Aquí usaremos un clasificador k-neighbors con n_neighbors=1. Este es un modelo muy simple e intuitivo que dice "la etiqueta de un punto desconocido es la misma que la etiqueta de sus puntos de entrenamiento más cercanos".

In [2]:

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=1)
```

Luego **entrenamos el modelo**, y lo usamos para **predecir etiquetas para datos que ya conocemos:**

In [3]:

```
model.fit(X, y)
y_model = model.predict(X)
```

Finalmente, **computamos la fracción de puntos etiquetados correctamente:**

In [4]:

```
from sklearn.metrics import accuracy_score
accuracy_score(y, y_model)
```

Out[4]:

1.0

¿100% de accuracy? Empecemos a sospechar que algo está mal...

Este procedimiento contiene una **falla fundamental:**

Entrenar y evaluar el modelo

con los mismos datos

¿Qué podemos hacer?

Podemos hacernos una mejor idea de la performance del modelo usando lo que se conoce como **holdout sets:** esto es, nos reservamos un subconjunto de los datos de entrenamiento, y luego usamos este subconjunto para verificar la performance del modelo.

Esta separación puede hacerse usando la función **train_test_split** en Scikit-Learn:

```
In [5]:
from sklearn.model_selection import train_test_split

# dividimos los datos con un 50% en cada subconjunto
X1, X2, y1, y2 = train_test_split(X, y, random_state=0,
                                  train_size=0.5)
```

```
# ajustamos el modelo en un subconjunto de los datos
model.fit(X1, y1)

# evaluamos el modelo con el otro conjunto de datos
y2_model = model.predict(X2)
accuracy_score(y2, y2_model)
```

Out[5]:

0.90666666666666662

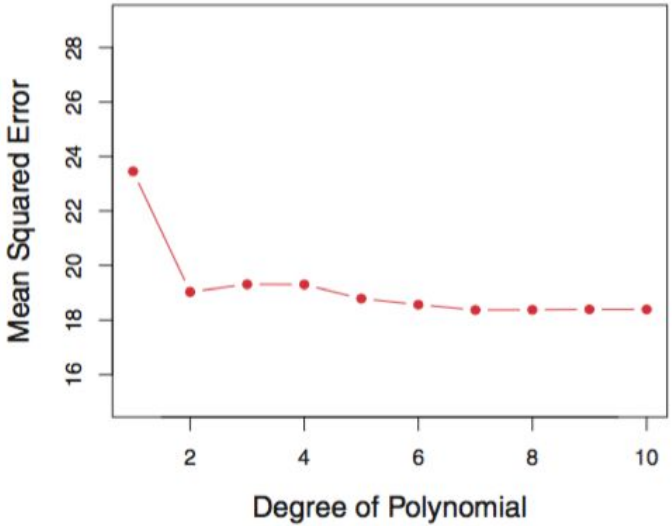
Aquí vemos un resultado más razonable: el clasificador nearest-neighbor tiene un 90% de precisión con este subconjunto de datos de testeo (no usados para entrenar el modelo).

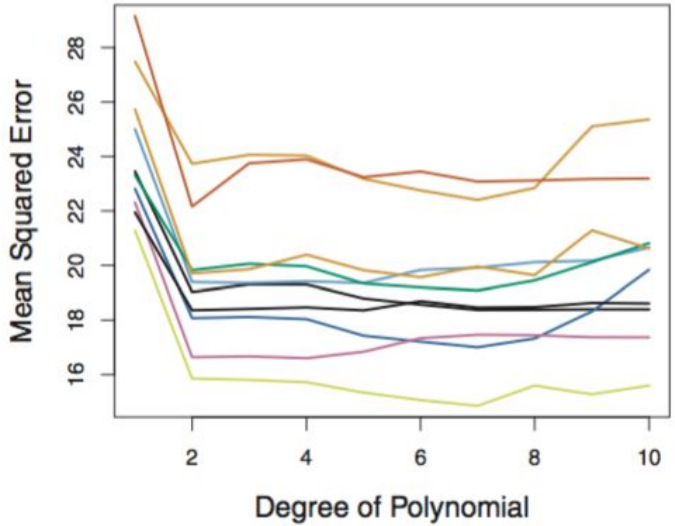
El **set de testeo** es similar a un nuevo set de datos no conocidos, porque **el modelo no los ha visto antes**.

Una desventaja de usar un holdout set para model validation es que perdemos una porción de nuestros datos para el entrenamiento del modelo.

En el caso anterior, la mitad del dataset no contribuye al training del modelo.

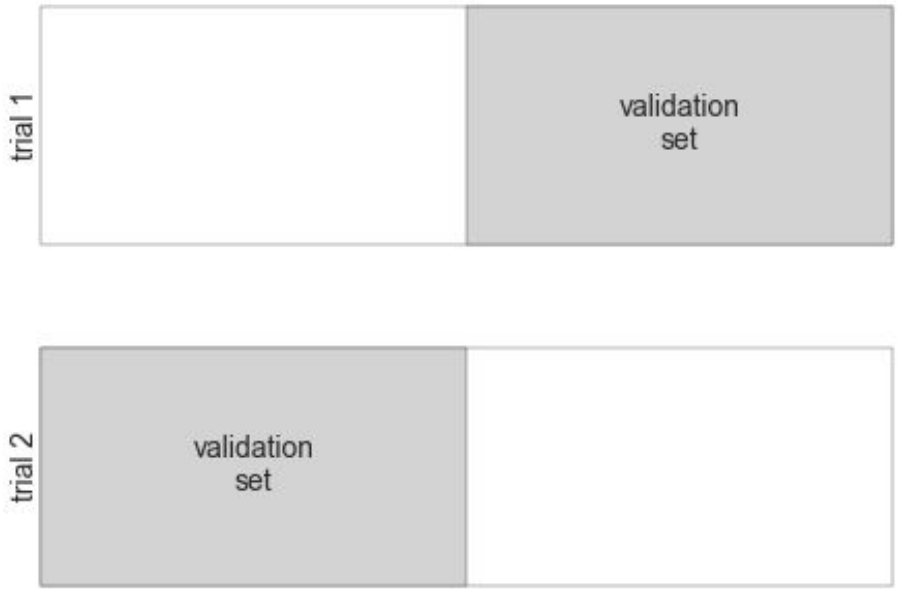
Esta no es una solución óptima, y puede causar problemas, especialmente si el set inicial de entrenamiento es pequeño.





Gareth James • Daniela Witten • Trevor Hastie Robert Tibshirani (2017)

Una forma de resolver esto es usar **cross-validation**; es decir, realizar una secuencia de ajustes donde cada subconjunto de los datos de entrenamiento es usado primero como set de entrenamiento (training set) y después como set de validación (validation set).



Aquí hacemos **dos iteraciones** de ajuste y predicción (notar el **chaining** en la invocación de los métodos **fit()** y **predict()**)

Alternativamente, **cada iteración usa cada mitad de los datos como un set de validación**.

Usando los datos divididos como antes, podríamos implementar **cross-validation** de esta forma:

In [6]:

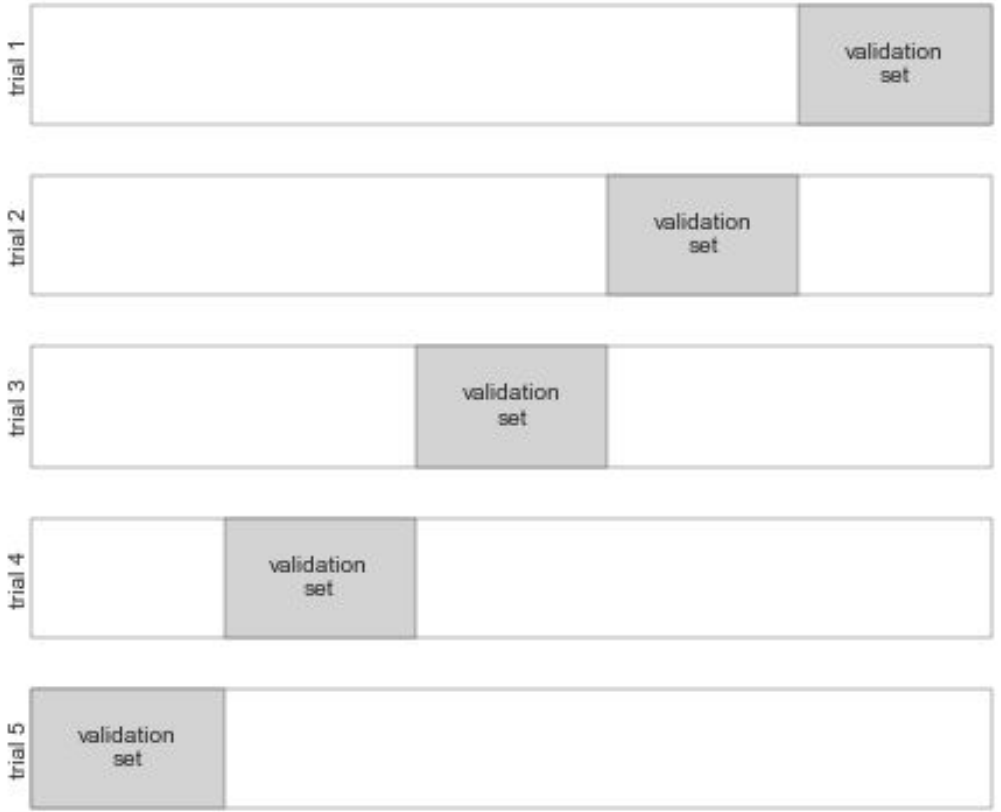
```
y2_model = model.fit(X1, y1).predict(X2)
y1_model = model.fit(X2, y2).predict(X1)
accuracy_score(y1, y1_model), accuracy_score(y2, y2_model)
```

Out[6]:

(0.95999999999999996, 0.90666666666666662)

Se muestran dos scores de accuracy, que podríamos combinar (por ejemplo, tomando la media) para obtener una mejor medida de la performance global del modelo. Esta forma particular de cross-validation se conoce como ***two-fold cross-validation,*** es decir dividimos los datos en dos conjuntos que usamos, por turnos, como como set de validación.

Podríamos extender esta idea y usar incluso más iteraciones, y más divisiones (folds) de los datos. Por ejemplo, aquí vemos una descripción de una **five-fold cross-validation:**



Aquí dividimos los datos en cinco grupos, y usamos a cada uno de ellos en turnos para evaluar el ajuste del modelo sobre las ⅘'s partes restante del dataset.

Esto sería tedioso de programar a mano, por lo que podemos usar la función **cross_val_score()** :

In [7]:

```
from sklearn.cross_validation import cross_val_score
cross_val_score(model, X, y, cv=5)
```

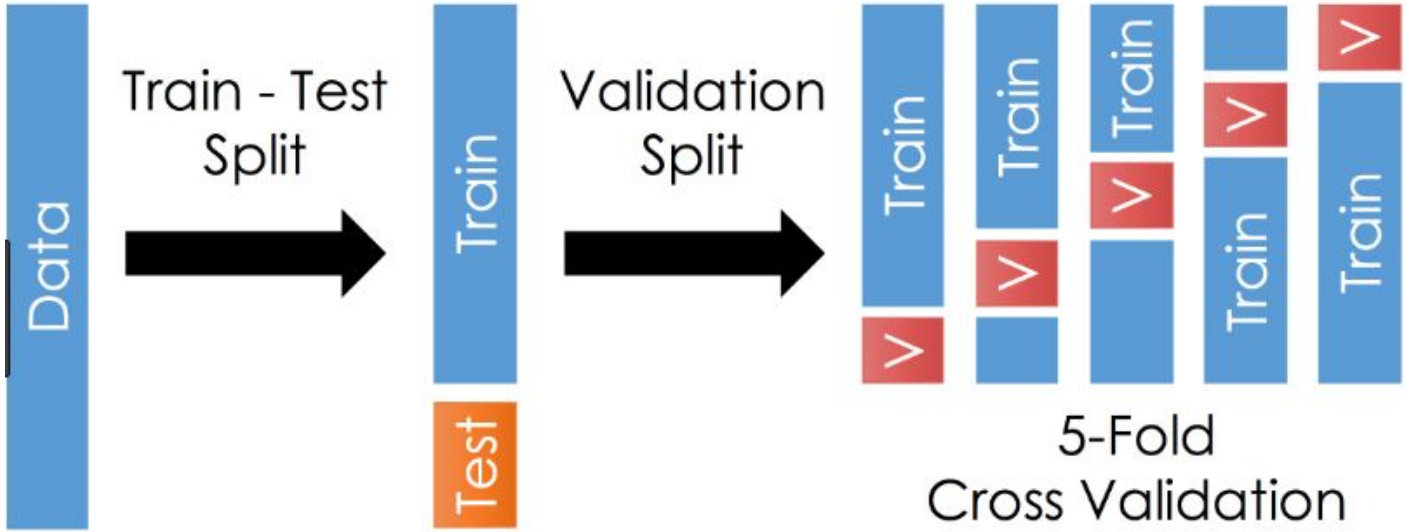
Out[7]:

```
array([ 0.96666667,  0.96666667,  0.93333333,  0.93333333,  1.  ])
```

Repetir la validación con diferentes subconjuntos de los datos nos da una mejor estimación de la performance del algoritmo.

Recuerde que también debemos tener un **conjunto de test** separado con el que no se optimiza ningún aspecto del modelo.

El esquema completo sería el siguiente. La cantidad de **folds** es otro aspecto a probar (en general, **entre 5 y 10** particiones)



Seleccionar el mejor modelo

La siguiente pregunta es **una de las más importantes** que vamos a hacernos:

si nuestro estimador tiene un rendimiento bajo, ¿cómo deberíamos avanzar?

Estas son varias **respuestas posibles**:

- Usar un modelo más complicado / flexible
- Usar un modelo menos complicado / flexible
- Obtener más muestras de entrenamiento
- Obtener más datos para agregar features a cada muestra

La selección del modelo y la selección de los hiperparámetros son de los aspectos más importantes en la práctica de machine learning, y frecuentemente esta información es subestimada en los tutoriales introductorios de la disciplina.

Si nuestro estimador tiene un rendimiento bajo, ¿cómo deberíamos avanzar?

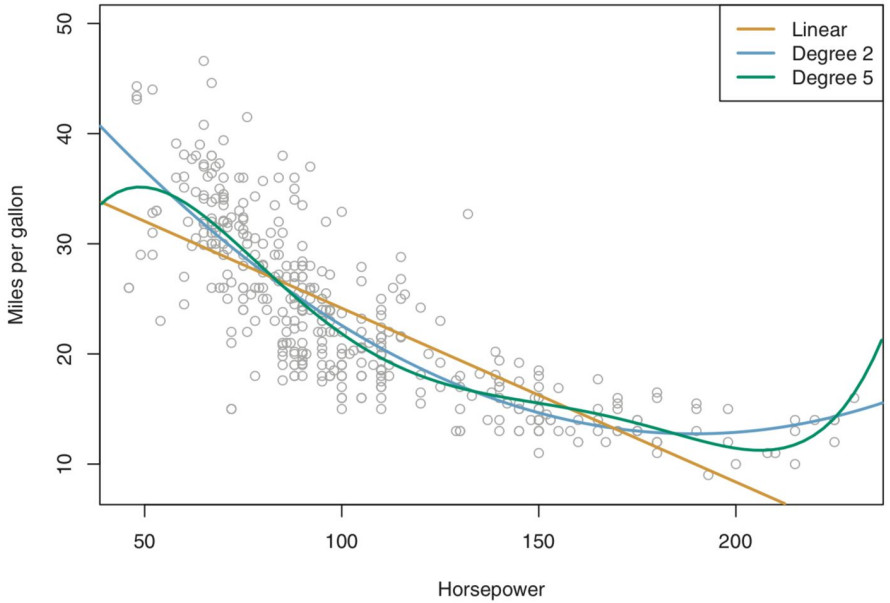
La respuesta a esta pregunta es frecuentemente contraintuitiva.

Por ejemplo:

- A veces, usar un modelo más complicado nos dará peores resultados
- Agregar más muestras de training puede no mejorar tus resultados

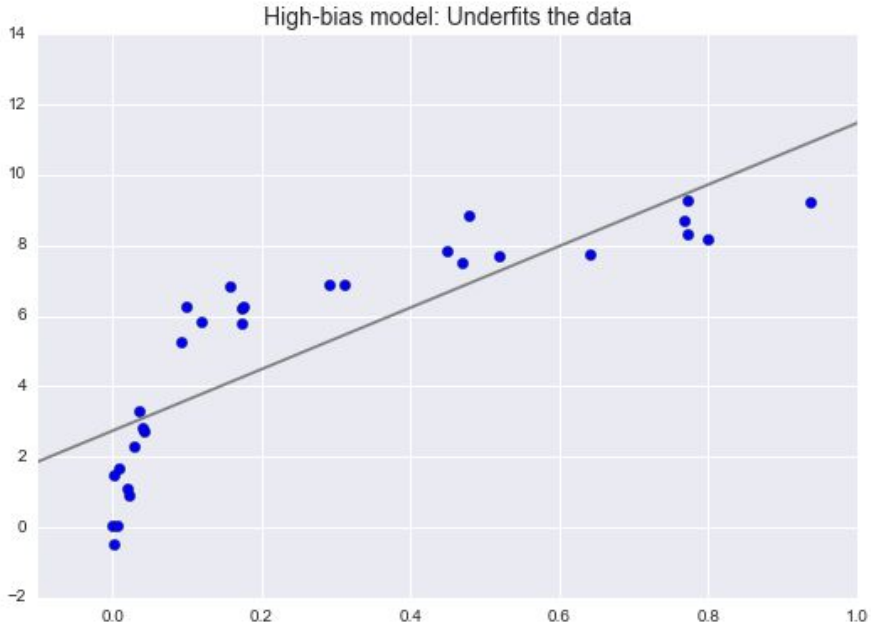
Fundamentalmente, la búsqueda del "mejor modelo" se trata en encontrar un punto óptimo en lo que se conoce como el dilema entre el sesgo y la varianza.

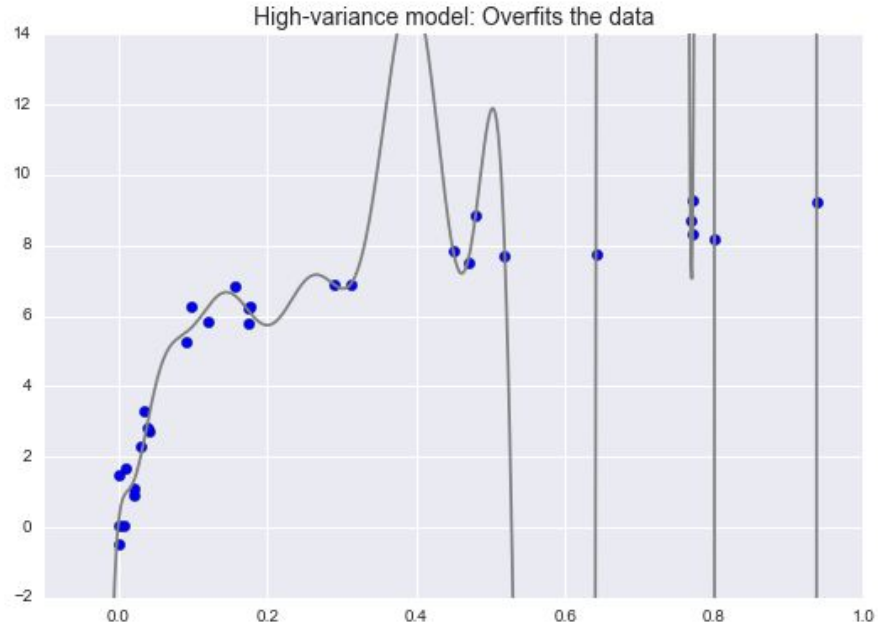
Consideremos **el dataset de autos** presentado en Gareth James, Daniela Witten, Trevor Hastie & Robert Tibshirani (2017). Podemos darle **más poder de ajuste** a una regresión agregando términos polinómicos:



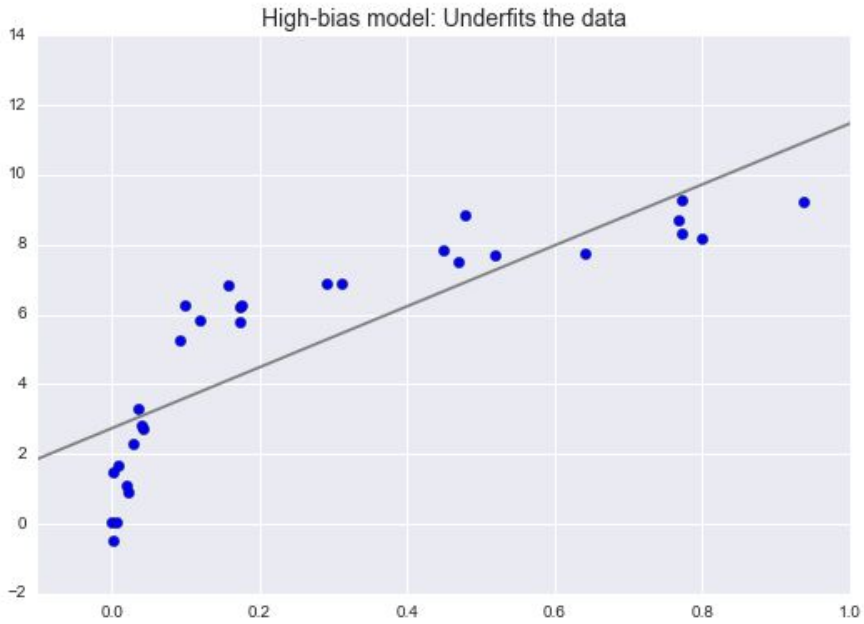
Por ejemplo, polinomio de grado 2 : $\texttt{mpg} = \beta_0 + \beta_1 \times \texttt{horsepower} + \beta_2 \times \texttt{horsepower}^2 + \epsilon$

Consideremos la siguiente figura, que representa **dos regresiones ajustadas al mismo dataset**





Claramente, ninguno de estos modelos es particularmente un buen ajuste a los datos, pero cada uno falla de diferentes formas.



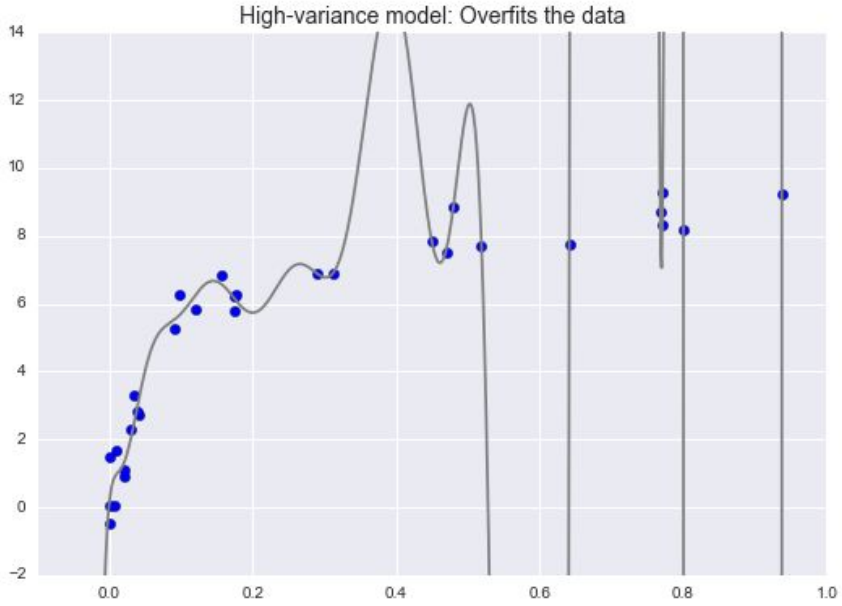
Este modelo intenta encontrar una línea recta para ajustar los datos.

Como **los datos son intrínsecamente más complicados**, el modelo de línea recta nunca será capaz de describir bien el dataset.

En estos casos se dice que el modelo **sub-ajusta (underfit)** los datos.

Es decir, el modelo no tiene la suficiente flexibilidad para representar apropiadamente todas las características en los datos

Otra forma de decirlo es que el modelo tiene un **sesgo alto** (**high bias**).



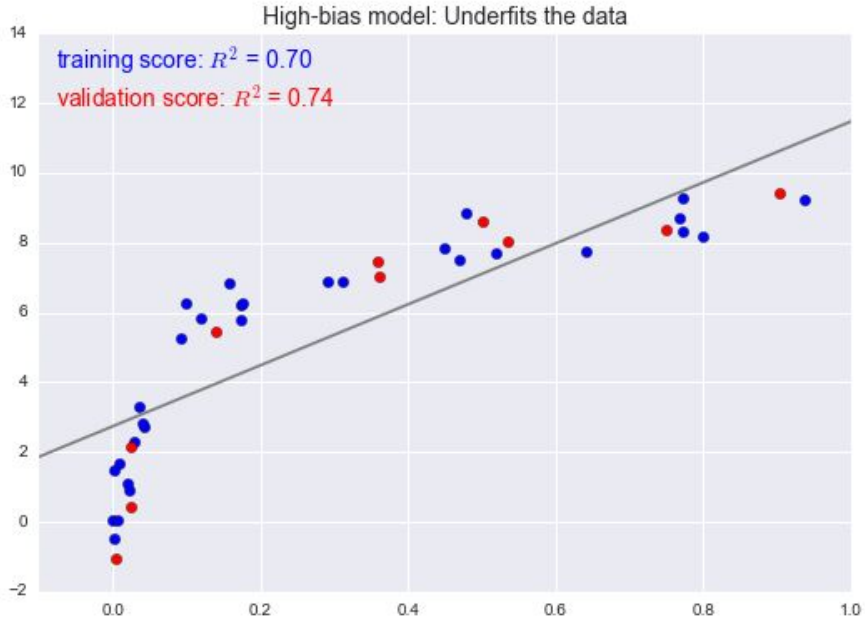
Este modelo intenta ajustar un **polinomio de grado alto** a los datos.

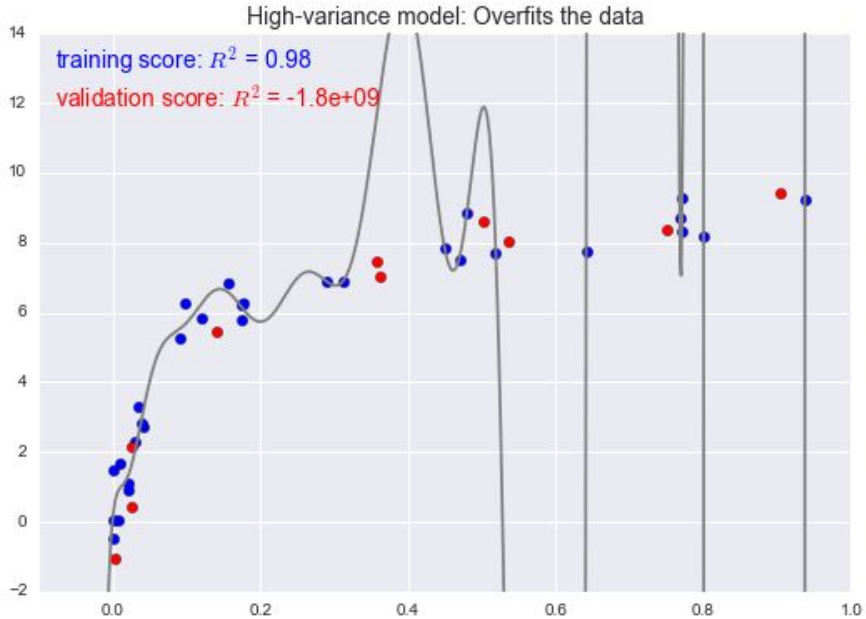
Aquí el ajuste del modelo tiene suficiente flexibilidad para tener en cuenta casi perfectamente los detalles finos en las características de los datos

Pero aunque describe muy precisamente los datos de entrenamiento, su forma pareciera estar **reflejando más las propiedades del ruido** que las propiedades intrínsecas del proceso generador de los datos.

Se dice que tal modelo **sobre-ajusta (overfit)** a los datos: es decir, el modelo tiene tanta flexibilidad que termina incluyendo tanto los errores aleatorios como la distribución de los datos; otra forma de decir esto es que el modelo tiene **alta varianza (high variance)**.

Para arrojar más luz a este problema, consideremos qué pasa si **usamos estos dos modelos para predecir el valor de "y" para datos nuevos.** En el diagrama, los puntos rojos son los que fueron omitidos del entrenamiento:





El score usado aquí es R^2, o coeficiente de determinación, el cual mide **cuán bien se comporta un modelo con respecto a una media simple de valores target.**

$R^2 = 1$ indica una precisión perfecta,

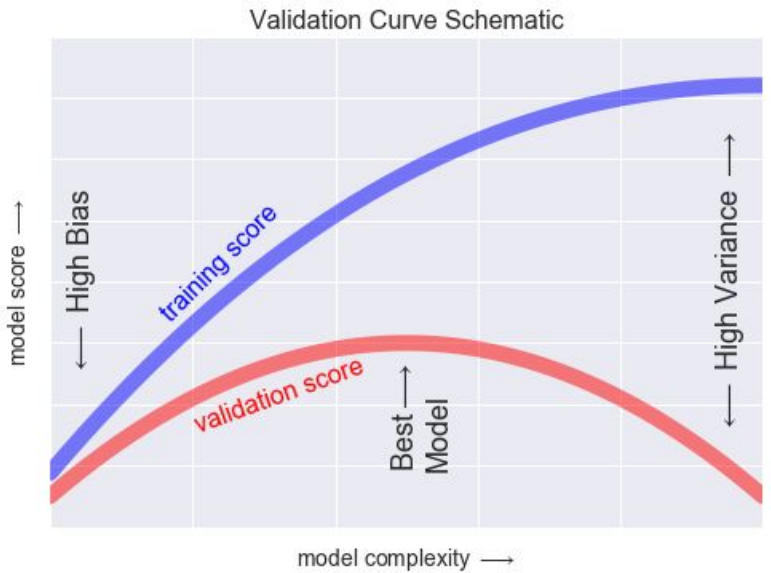
$R^2 = 0$ indica que el modelo no funciona mejor que simplemente tomar la media de los datos,

$R^2 < 0$ nos dice que el modelo rinde incluso peor que tomar la media de los valores target.

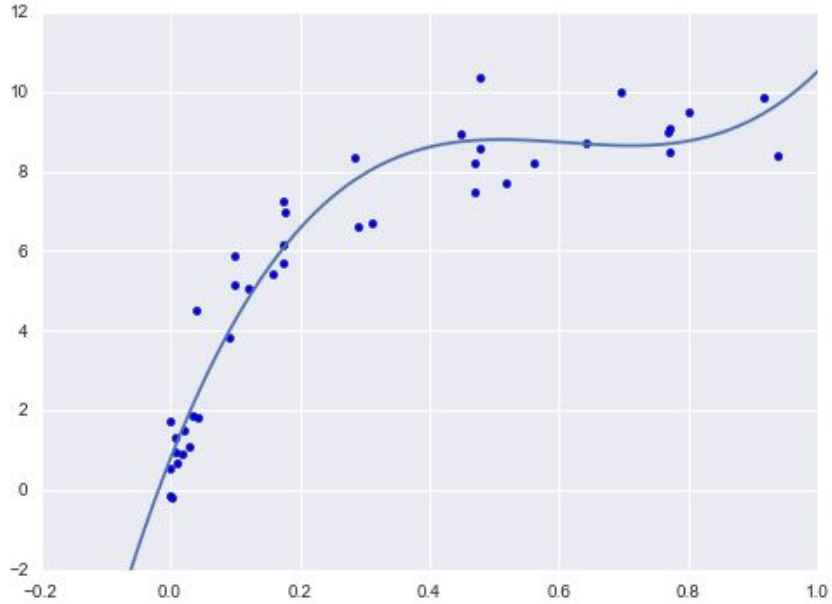
De los scores asociados con estos dos modelos, podemos hacer **observaciones que pueden generalizarse:**

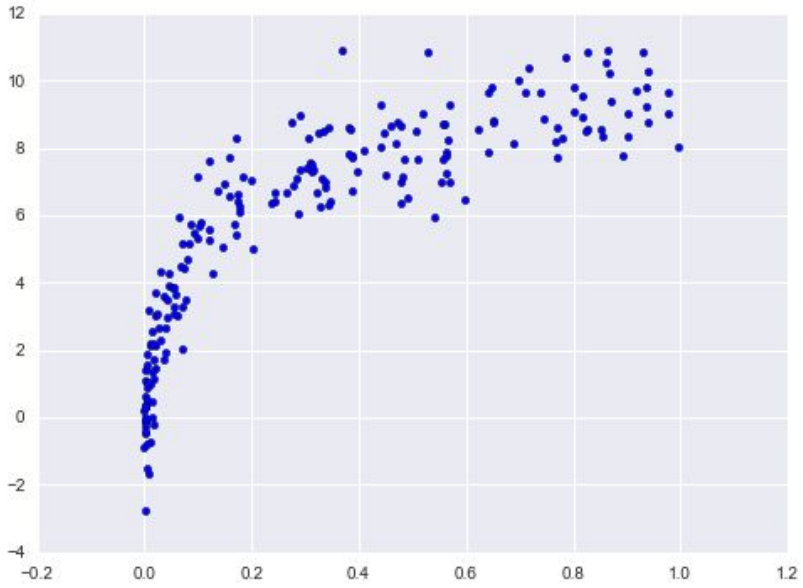
- Para modelos con **alto sesgo**, la performance del modelo con el set de validación es similar a la performance con el set de entrenamiento, y **ambas son bajas.**
- Para modelos con **alta varianza**, la performance del modelo con el set de validación es mucho peor que la performance con el set de entrenamiento.

- El score de entrenamiento es mayor que el score de validación.
- Para un modelo de muy baja complejidad (sesgo alto), el set de entrenamiento se **underfitea**
- Para modelos con muy alta complejidad (modelos con alta varianza), los datos de entrenamiento son **overfiteados**
- Para algún modelo intermedio, la validation curve tiene un máximo. Este nivel de complejidad indica un **compromiso apropiado entre sesgo y varianza**.



El modelo óptimo generalmente dependerá del tamaño del set de entrenamiento...

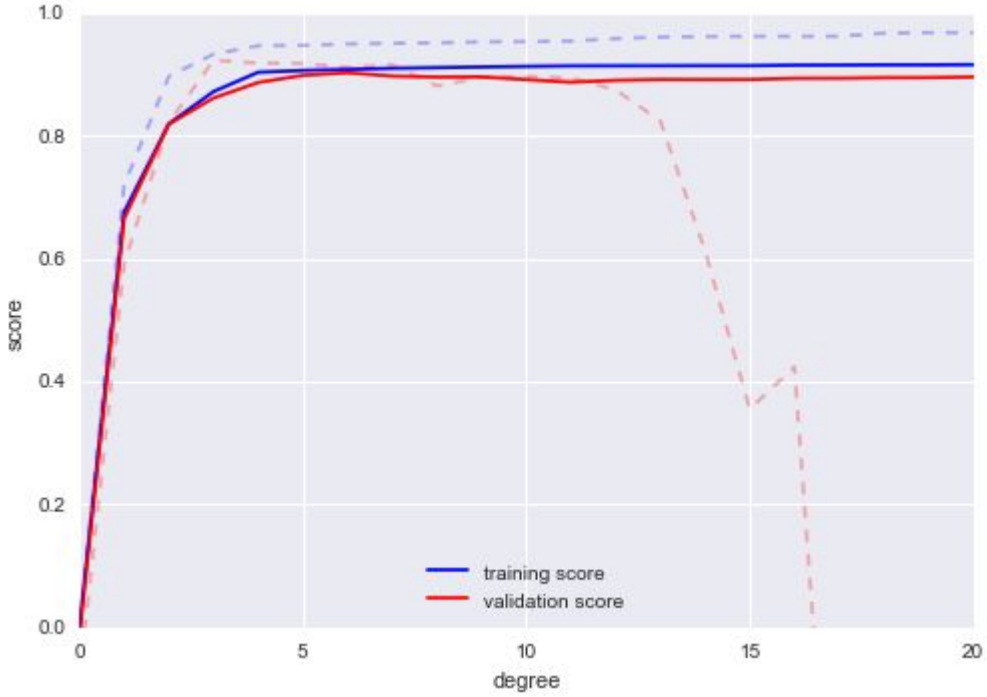




Las líneas llenas muestran los del dataset "grande", mientras que las líneas cortadas muestran los resultados del dataset más pequeño.

Se observa que la validation curve que el dataset más grande puede soportar un modelo mucho más complicado.

El pico en este ejemplo está alrededor del grado 6, pero incluso un modelo de grado 20 no está overfiteando gravemente - los scores de validación y entrenamiento se mantienen muy cerca.



De esta forma, vemos que **el comportamiento de la validation curve tiene dos inputs importantes:**

- **la complejidad del modelo**
- **el número de puntos de entrenamiento**

Usualmente es útil explorar el comportamiento del modelo como una función del número de muestras de entrenamiento. Esto puede hacerse usando subconjuntos de tamaño creciente de nuestros datos para ajustar nuestro modelo.

Un plot del score de entrenamiento/validación con respecto al tamaño del set de entrenamiento se conoce como **learning curve.**

El comportamiento general que podríamos esperar de una learning curve es este:

- **Un modelo con una complejidad dada sobre-ajustará a un dataset pequeño:** esto significa que el score de entrenamiento será relativamente alto, mientras que el score de validación será relativamente bajo.
- **Un modelo con una dada complejidad sub-ajustará a un dataset grande:** lo que significa que el score de entrenamiento decaerá pero, pero el score de validación se incrementará.
- **Un modelo nunca, excepto por azar, dará un mejor score de validación que de entrenamiento**: esto significa que las curvas deberían acercarse pero nunca cruzarse.

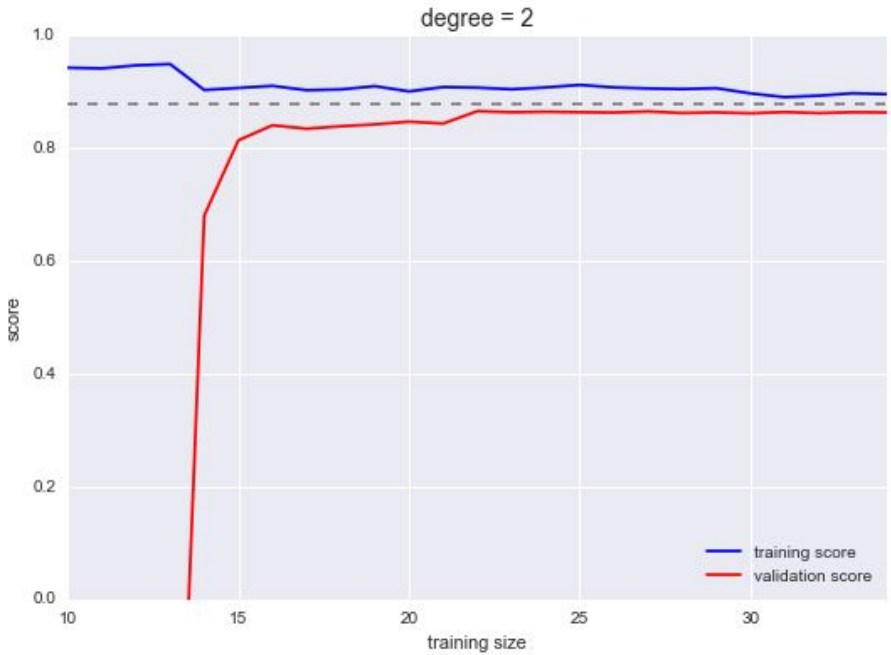
Con estas características en mente, deberíamos esperar que una learning curve se vea, cualitativamente, como en la siguiente figura:

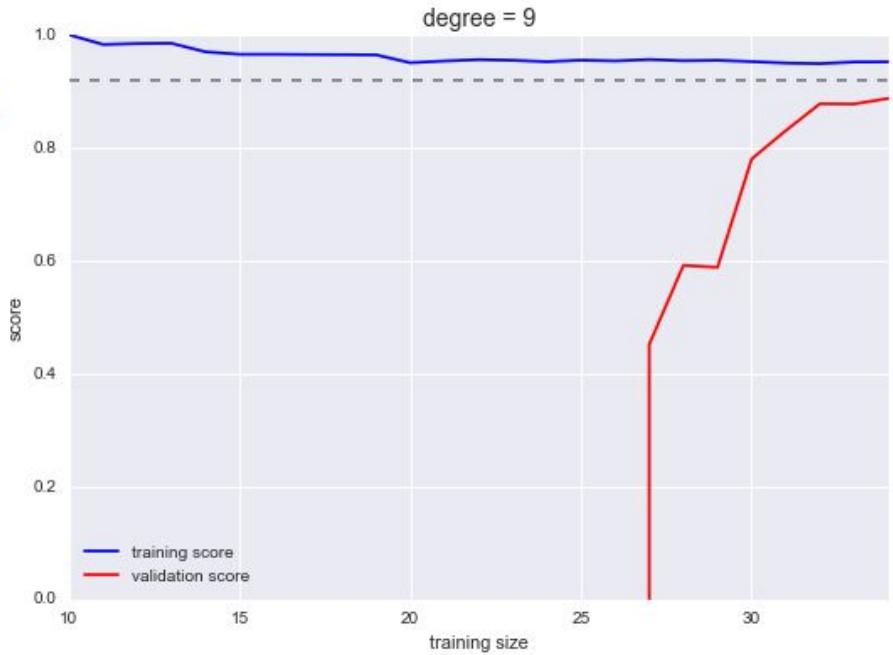
La característica notable de la learning curve es **la convergencia a un score particular a medida que el número de muestras de entrenamiento crece.**

En particular, una vez que tenemos puntos suficientes para que el modelo haya convergido, **agregar más puntos de entrenamiento no ayudará a mejorar el score.**

La única forma de incrementar la performance del modelo en este caso es usar otro modelo, generalmente más complejo.

Ejemplos en Scikit-Learn





En la práctica, los modelos generalmente tienen **más de un hiperparámetro para configurar** (más de una "perilla para mover"), y por lo tanto los plots de las validation curves y learning curves cambian de líneas a **superficies multidimensionales.**

En estos casos, **tales visualizaciones son difíciles** y deberíamos simplemente **encontrar el modelo particular que maximice el score de validación.**

Scikit-Learn provee herramientas automatizadas para realizar esta búsqueda, tales como **Grid Search y Random Search.**

- **Model validation**
  - Cross validation
  - Validation curve
  - Learning curve

- **Dilema entre sesgo y varianza**

- **Optimización de hiperparámetros**