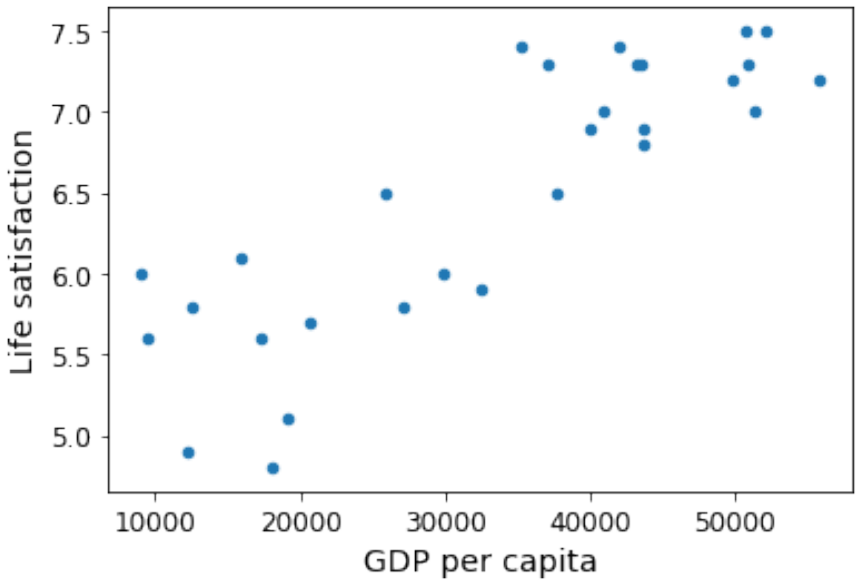
Chapter 2:
Supervised Learning I: Regresi

October 16, 2020

Pada bagian ini akan dibahas algoritma-algoritma yang tergolong pada kategori *supervised learning*. Kita akan mulai dengan contoh permasalahan yang dapat dikategorikan sebagai *supervised learning*.

Jika dimisalkan kita diberikan dataset yang menunjukan hubungan antara GDP per kapita dan kepuasan hidup (*life satisfaction*) untuk beberapa negara seperti ditunjukan pada Gambar 2.1. Bagaimana kita menentukan cara untuk memprediksi kepuasan hidup dari sebuah negara dengan GDP tertentu?



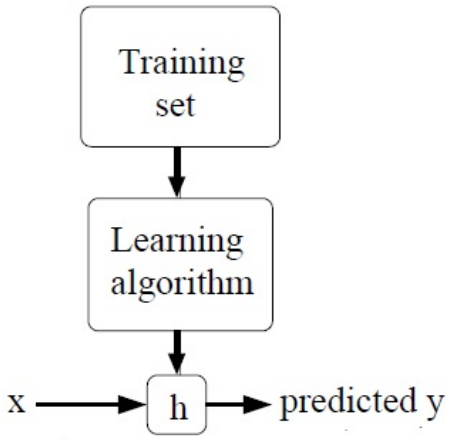
Gambar 2.1: Plot data Life Satisfaction terhadap GDP

Permasalahan di atas merupakan permasalahan *supervised learning* (tepatnya regresi). Karena dataset yang telah ada bisa dijadikan data training dalam menentukan parameter model untuk prediksi.

Notasi yang akan digunakan selanjutnya. $x^{(i)}$ adalah variabel input (GDP pada Gambar 2.1), disebut juga fitur (***features***), dan $y^{(i)}$ menyatakan variabel ***target*** yang kita coba prediksi (*life satisfaction* pada Gambar 2.1). Kemudian pasangan $(x^{(i)}, y^{(i)})$ disebut dengan ***training example***, dan dataset yang akan digunakan untuk pembelajaran (*learn*), yang terdiri dari m *training example* $\{(x^{(i)}, y^{(i)}); i = 1, 2, \cdots, m\}$ disebut dengan *training set*. Kita akan gunakan juga $\mathcal{X}$ sebagai ru-

ang harga input (*space of input values*), dan $\mathcal{Y}$ adalah ruang harga output. Pada contoh di atas $\mathcal{X} = \mathcal{Y} = \mathbb{R}$, dimana $\mathbb{R}$ menyatakan bilangan *real*.

Untuk menjelaskan *supervised learning* secara lebih formal, tujuan kita adalah mempelajari sebuah fungsi $h : \mathcal{X} \mapsto \mathcal{Y}$ setelah diberikan *training set*, sehingga $h(x)$ adalah '*good predictor*' untuk harga y yang bersesuaian. Fungsi h disebut juga ***hypothesis***. Penjelasan gambar dapat dilihat pada Gambar 2.2.



Gambar 2.2: Fungsi Hypothesis yang menjelaskan supervised learning

Ketika variabel target yang ingin kita prediksi bersifat kontinyu, seperti pada contoh kasus di Gambar 2.1., maka masalah ini disebut dengan **regresi (*regression*)**. Ketika y hanya dapat bernilai diskrit dengan jumlah harga sedikit, maka masalah ini kita sebut dengan **klasifikasi (*classification*)**.

1 Regresi Liniear (*Linear Regression*)

Pada bagian ini kita akan mempelajari **Model Regresi Linier**, yang merupakan model yang paling sederhana, dengan dua pendekatan berbeda: * Menggunakan persamaan bentuk tertutup (*closed-form equation*) yang secara langsung menghitung parameter model yang paling baik memenuhi *fitting* model pada training set. Atau dengan kata lain menemukan parameter dari model yang meminimalkan *cost function* pada training set. Persamaan tertutup ini disebut dengan **Persamaan Normal (*Normal Equation*)**. * Menggunakan pendekatan optimasi secara iteratif yang disebut dengan ***Gradient Descent (GD)*** yang secara gradual mengatur parameter model supaya meminimalkan *cost function* pada training set. Yang pada akhirnya parameter yang didapatkan dengan pendekatan ini akan konvergen ke parameter-parameter yang diperoleh dengan menggunakan *closed-form* (poin 1).

Pada bagian sebelumnya (Gambar 2.1.), merupakan contoh yang paling sederhana yang dapat didekati dengan perkiraan y (*life satisfaction*) menggunakan fungsi liniear (disebut juga **hipotesis**):

Persamaan (2.1). *Hyhpothesis*

$$h_\theta(x) = \theta_0 + \theta_1 x_1$$

dimana $\theta_j, j = 0, 1$ adalah dua parameter atau pembobot (*weight*) yang merupakan parameter ruang dari fungsi linier memetakan $\mathcal{X}$ ke $\mathcal{Y}$. θ_0 disebut juga dengan *intercept*

(*bias*), dalam hal ini dapat dianggap $x_0 = 1$, dan x_1 adalah fitur (GDP perkapita)

Secara umum, persamaan model prediksi jika terdapat n fitur dapat dituliskan dengan persamaan (2.2).

Persamaan (2.2). Persamaan model prediksi

$h(x) = \sum_{j=0}^{n} \theta_j x_j = \theta^T x$

dimana θ dan x secara berurutan adalah vektor parameter dan vektor input (*feature*), dan $(\cdot)^T$ adalah *transpose*.

Permasalahan kemudian adalah jika sudah diberikan *training set*, bagaimanakan cara memilih (atau *learn*) parameter-parameter ($\theta_j, j = 0, 1, \cdots, n$) ? Salah satu cara yang logis adalah mengatur $h(x)$ dekat dengan y, setidaknya untuk *training example-training example* yang kita punyai. Untuk memformulasikan hal tersebut, kita dapat mendefinisikan fungsi yang mengukur seberapa dekat $h(x^{(i)})$ ke masing-masing $y^{(i)}$, yang disebut dengan ***cost function***:

Persamaan (2.3). *Cost function*

$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

Persamaan 2.3 di atas juga disebut dengan ***least-squares cost function***. Sebagai catatan $x^{(i)}$ adalah **skalar atau vektor** sampel ke-i dari training dataset dengan ukuran dimensi sama dengan jumlah *feature*.

1.1 Persamaan Normal

Pada metode ini, kita akan meminimalkan J dengan langsung menurunkan persamaan *cost function* terhadap θ_j, kemudian menyamakannya dengan nol sehingga akan diperoleh nilai θ_j. Sebelum menurunkan kita akan melihat terlebih dahulu beberapa notasi pada kalkulus untuk matriks.

- **Turunan Matriks**

Untuk sebuah fungsi $f : \mathbb{R}^{m \times n} \mapsto \mathbb{R}$ yang memetakan m kali n matriks ke bilangan real, definisi turunan f terhadap A adalah sebagai berikut:

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{m1}} & \cdots & \frac{\partial f}{\partial A_{mn}} \end{bmatrix}$$

Oleh karena itu, gradien $\nabla_A f(A)$ sendiri adalah matriks $m \times n$ dimana elemen (i, j) adalah $\frac{\partial f}{\partial A_{ij}}$.

Sebagai contoh, jika $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ adalah matriks 2×2 dan fungsi $f : \mathbb{R}^{2 \times 2} \mapsto \mathbb{R}$ dapat dituliskan sebagai $f(A) = \frac{3}{2} A_{11} + 5A_{12}^2 + A_{21} A_{22}$, dimana A_{ij} menyatakan elemen ke-(i, j) dari matriks A. Maka kita akan memperoleh

$$\nabla_A f(A) = \begin{bmatrix} \frac{3}{2} & 10A_{12} \\ A_{22} & A_{21} \end{bmatrix}.$$

Operator `trace`, yang dituliskan sebagai `tr`. Untuk matriks A berukuran $n \times n$ (matriks *square*), maka `trace` dari A didefinisikan sebagai penjumlahan entri-entri diagonal dari matriks tersebut:

$\text{tr}\, A = \sum_{i=1}^{n} A_{ii}$

Jika a adalah bilangan real, maka $\text{tr}\, a = a$. Operator `trace` mempunyai sifat bahwa jika terdapat matriks A dan B sehingga $A \times B$ adalah matriks *square*, maka kita akan memperoleh $\text{tr}\, AB = \text{tr}\, BA$. Maka kita juga akan mendapatkan persamaan-persamaan berikut:

$\text{tr}\, ABC = \text{tr}\, CAB = \text{tr}\, BCA$
$\text{tr}\, ABCD = \text{tr}\, DABC = \text{tr}\, CDAB = \text{tr}\, BCDA$

Beberapa sifat berikut juga berlaku untuk operator `trace`, dimana A dan B adalah matriks-matriks *square* dan a adalah bilangan real:

$$\begin{aligned} \text{tr}\, A &= \text{tr}\, A^T \\ \text{tr}(A + B) &= \text{tr}\, A + \text{tr}\, B \\ \text{tr}\, aA &= a\, \text{tr}\, A \end{aligned}$$

Persamaan (2.4) merupakan persamaan-persamaan turunan matriks yang diberikan tanpa pembuktian.

Persamaan (2.4). Turunan matriks

$$\begin{aligned} \nabla_A \text{tr}\, AB &= B^T \\ \nabla_{A^T} f(A) &= (\nabla_A f(A))^T \\ \nabla_A \text{tr}\, ABA^TC &= CAB + C^TAB^T \\ \nabla_A |A| &= |A|(A^{-1})^T \end{aligned}$$

Sebagai catatan persamaan terakhir hanya berlaku untuk kasus dimana A adalah matriks *squre non-singular*, dan $|A|$ menyatakan determinan dari matriks A.

Untuk membuat lebih jelas, kita akan mendeskripsikan arti dari persamaan pada baris pertama di Persamaan (2.4). Jika dimisalkan kita mempunyai matriks $B \in \mathbb{R}^{n \times m}$. Maka kita dapat mendefinisikan sebuah fungsi $f : \mathbb{R}^{m \times n} \mapsto \mathbb{R}$ berdasarkan pesamaan $f(A) = \text{tr}\, AB$, dalam hal ini AB adalah matriks *square*. Dalam hal ini kita dapat mengimplementasikan turunan matriks untuk menemukan $\nabla_A f(A)$ yang merupakan matriks $m \times n$. Baris pertama pada Persamaan (2.4) menyatakan bahwa elemen (i, j) dari matriks $\nabla_A f(A)$ sama dengan elemen elemen (i, j) dari matriks B^T, atau ekivalen dengan B_{ji}.

- **Least Square**

Dengan menggunakan persamaan turunan matriks, kita akan tentukan harga θ yang meminimalkan J. Jika diberikan sebuah *training set*, kita definisikan *design matrix* X adalah matriks dengan ukuran $m \times n$ (atau $m \times n + 1$ jikalau menyertakan *intercept*) yang terdiri dari *training-training example* pada masing-masing baris:

$$X = \begin{bmatrix} - - (x^{(1)})^T - - \\ - - (x^{(2)})^T - - \\ \vdots \\ - - (x^{m})^T - - \end{bmatrix}.$$

Dan juga kita definisikan y merupakan vektor dengan m dimensi yang terdiri dari harga-harga target dari *training set*:

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}.$$

Karena $h_\theta(x^{(i)}) = (x^{(i)})^T\theta$, kita bisa buktikan bahwa

$$X\theta - y = \begin{bmatrix} (x^{(1)})^T\theta \\ (x^{(2)})^T\theta \\ \vdots \\ (x^{m})^T\theta \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{m}) - y^{(m)} \end{bmatrix}.$$

Oleh karena itu, dengan menggunakan formulasi bahwa $z^Tz = \sum_i z_i^2$ dimana z adalah sebuah vektor:

$\frac{1}{2}(X\theta - y)^T(X\theta - y) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 = J(\theta)$.

Untuk meminimalkan J, maka akan kita tentukan turunannya terhadap θ.

sehingga

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{2}\left(X\theta - y\right)^T\left(X\theta - y\right) && (1)\\
&= \frac{1}{2}\nabla_\theta\left(\theta^T X^T X\theta - \theta^T X^T y - yX\theta + y^T y\right) && (2)\\
&= \frac{1}{2}\nabla_\theta \operatorname{tr}\left(\theta^T X^T X\theta - \theta^T X^T y - yX\theta + y^T y\right) && (3)\\
&= \frac{1}{2}\nabla_\theta\left(\operatorname{tr}\theta^T X^T X\theta - 2\operatorname{tr} y^T X\theta\right) && (4)\\
&= \frac{1}{2}\left(X^T X\theta + X^T X\theta - 2X^T y\right) && (5)\\
&= X^T X\theta - X^T y && (6)
\end{aligned}$$

Pada penurunan di atas, langkah ke-3 menggunakan kenyataan bahwa `trace` dari sebuah bilangan real adalah bilangan real pula. Langkah ke-4 berasal dari persamaan $\operatorname{tr} A = \mathrm{A}^T$. Sedangkan langkah ke-5 diperoleh dari persamaan berikut, yang mengkombinasikan baris ke-2 dan ke-3 dari Persamaan (2.4):

$\nabla_{A^T} \operatorname{tr} ABA^TC = B^TA^TC^T + BA^TC$.

Untuk meminimalkan J, kita atur turunannya sama dengan nol, dan akan diperoleh **Persamaan Normal (*Normal Equation*)**.

Persamaan (2.5). Persamaan normal

$X^T X\theta = X^T y$

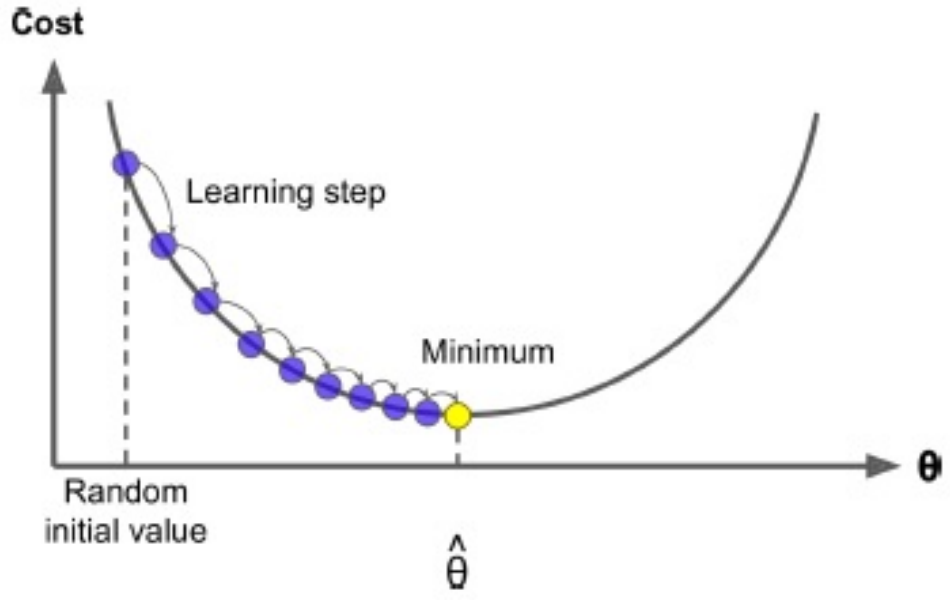
Maka nilai θ yang meminimalkan $J(\theta)$ dalam bentuk tertutup adalah $\hat{\theta} = (X^T X)^{-1} X^T y$, dimana $\hat{\theta}$ adalah nilai estimasi θ yang meminimalkan J

1.2 Gradient Descent

Gradient Descent merupakan algoritma optimasi generik yang mempunyai kemampuan untuk menemukan solusi optimal pada permasalahan-permasalahan yang cukup luas. Ide dari *gradient descent* adalah mengatur parameter secara iteratif untuk meminimalkan *cost function*.

Sebagai ilustrasi cara kerja dari *gradient descent* adalah dimisalkan ketika kita tersesat di gunung yang terselimuti kabut tebal, dan yang bisa kita rasakan adalah kemiringan tanah dengan menggunakan kaki. Strategi terbaik untuk menuruni gunung secara cepat adalah dengan mengambil arah menurun yang mempunyai kemiringan (*slope*) bukit yang paling curam (*steepest*). Cara kerja *gradient descent* sama seperti ilustrasi di atas. Algoritma tersebut mengukur kemiringan lokal (*local gradient*) dari *cost function* terhadap vektor parameter (Dalam kasus di atas *cost function* adalah $J(\theta)$ dan vektor parameter adalah $\vec{\theta}$), kemudian mengambil arah menurun. Dan ketika gradien sama dengan nol, maka disimpulkan bahwa nilai minimum telah dicapai.

Secara konkret, kita inisialisasi nilai θ dengan harga random (disebut dengan *random initialization*). Kemudian nilai θ diperbaiki secara gradual dengan langkah-langkah yang kecil dalam satu waktu, dimana setiap langkah yang diambil adalah untuk mengurangi *cost function* sehingga algoritma konvergen pada harga minimum (lihat gambar 2.3).

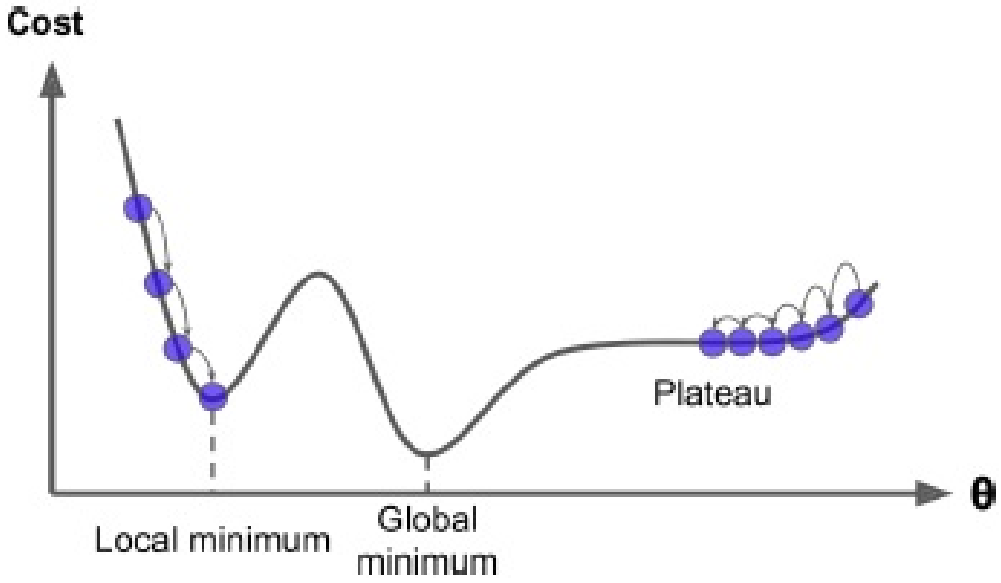


Gambar 2.3: Ilustrasi Gradient Descent

Paremeter yang paling penting dalam *gradient descent* adalah ukuran langkah (*step size*), yang ditentukan oleh *hyperparameter* yang disebut dengan ***learning rate***. Jika *learning rate* terlalu kecil maka algoritma akan memakan waktu lama untuk konvergen mencapai nilai minimum, tetapi kalau terlalu besar maka nilai θ akan loncat-loncat dari satu sisi ke sisi lain dari nilai minimum,

dengan kemungkinan harga *cost function* yang membesar daripada sebelumnya. Hal ini akan menyebabkan algoritma menjadi *divergen* dengan harga yang semakin besar dan gagal mencapai nilai minimum.

Dalam implementasi, tidak selamanya kurva *cost function* sebagai fungsi parameter θ terlihat sangat baik seperti mangkuk terbalik (*convex*). Bisa jadi terdapat nilai minimum lokal dan global, berbentuk *plateu*, seperti yang ditunjukan pada gambar 2.4. Jika inisialisasi random terletak sebelah kiri, maka bisa jadi algoritma akan konvergen ke minimum lokal (*local minimum*), yang tidak sebaik minimum global (*global minimum*). Jika dimulai sebelah kanan, maka akan memakan waktu lebih lama untuk konvergen karena melewati *plateu*. Dan jika berhenti terlalu cepat, maka minimum global tidak akan pernah tercapai.



Gambar 2.4: Kesulitan Gradient Descent

1.2.1 Batch Gradient Descent

Dalam kasus regeresi, kita ingin memilih θ sehingga akan meminimalkan *cost function* $J(\theta)$ seperti pada persamaan 2.3. Dimulai dengan tebakan awal (*initial guess*) untuk nilai θ secara random, dan mengubah θ secara berulang untuk membuat $J(\theta)$ lebih kecil sampai berharap θ konvergen ke harga yang dapat meminimalkan $J(\theta)$. Secara formal *update rule* bisa dituliskan sebagai berikut:

$$\theta_j^{(next)} := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

Update tersebut dilakukan secara simultan untuk semua *features* $j = 0, 1, \cdots, n$, dan α disebut dengan ***learning rate***.

Untuk mengimplementasikan algoritma, kita pertimbangkan terlebih dahulu untuk satu *training example* (x, y), sehingga penjumlahan pada *cost function* persamaan (2.3) bisa diabaikan terlebih dahulu:

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j}\left(h_\theta(x) - y\right)^2 \\ &= 2.(h_\theta(x) - y).\frac{\partial}{\partial \theta_j}(h_\theta(x) - y) \\ &= 2.(h_\theta(x) - y).\frac{\partial}{\partial \theta_j}\left(\sum_{i=0}^{n} \theta_i x_i - y\right) \\ &= 2.(h_\theta(x) - y)x_j\end{aligned}$$

Sehingga, *update rule* untuk satu *training example* menjadi:

$\theta_j^{(next)} := \theta_j - 2\alpha\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}$

Sedangkan untuk m training examples, update rule untuk parameter ke-j yaitu θ_j menjadi:

$\theta_j^{(next)} := \theta_j - \frac{2\alpha}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)},\ \ \forall j = 0, 1, \cdots, n.$

Sehingga, dalam bentuk parameter vektor, *update rule* ditunjukan pada Persamaan (2.6).

Persamaan (2.6). *Update rule* untuk *Batch Gradient Descent*

$\theta^{(next)} = \theta - \frac{2\alpha}{m}X^T\left(X\theta - y\right)$

Karena algoritma ini menggunakan semua training set (*batch*) untuk menghitung update pada setiap langkahnya, maka disebut juga sebagai ***Batch Gradient Descent*** (BGD). Contoh implementasi dari BGD bisa dilihat pada program berikut.

1.2.2 Stochastic Gradient Descent

Masalah utama pada *batch gradient descent* adalah kenyataan menggunakan semua training set untuk menghitung gradient setiap langkahnya, sehingga akan sangat lambat jika training set sangat besar. Metode lain untuk mengatasi ini adalah dengan menggunakan *Stochastic Gradient Descent* (SGD), dimana metode ini mengambil satu *training example (instance)* secara random dari training set pada setiap langkah perhitungan gradien. Sehingga, dengan mengambil satu *training example* saja algoritma bisa dieksekusi lebih cepat karena hanya perlu manipulasi data yang sedikit. Hal ini juga memungkinkan untuk melakukan training dengan data yang sangat besar, karena hanya satu *training example* yang harus disimpan pada memori. Gambaran algoritma SGD adalah sebagai berikut:

Loop epoch {

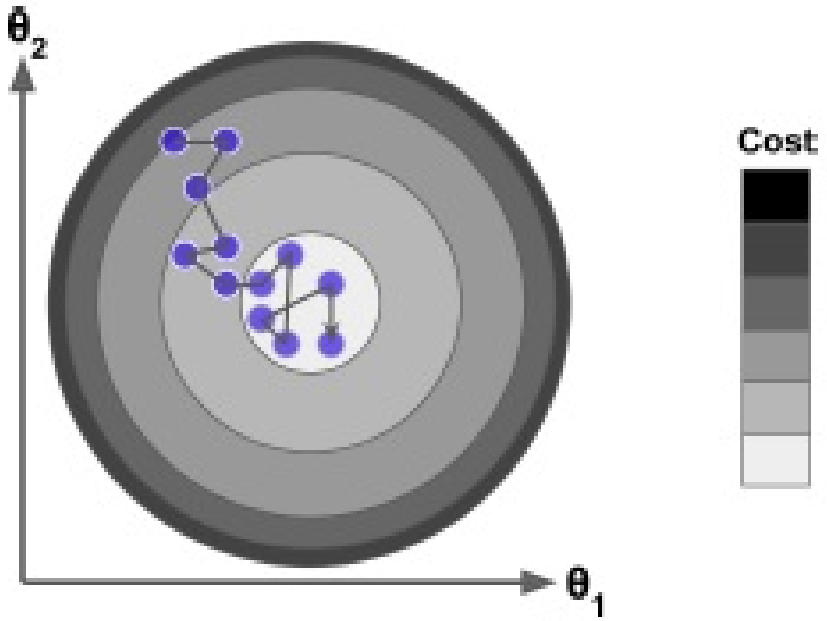
```
for i = 1 to m, {
```

$\theta_j^{(next)} := \theta_j - 2\alpha\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)},\ \ \forall j = 0, 1, \cdots, n.$

```
}
```

}

Algoritma SGD lebih tidak beraturan dibandingkan dengan menggunkan BGD. *Cost function* dari SGD akan bolak-balik naik turun dan mengecil secara rata-rata saja. Dengan waktu, *cost function* akan mendekati nilai minimum, tetapi akan terus berfluktuasi setelahnya, seperti diperlihatkan pada Gambar 2.5. Ketika *cost function* tidak beraturan, keadaan ini akan menolong algoritma loncat keluar dari local minima dan mendapatkan global minima. SGD mempunyai kemungkinan memperoleh global minimum dibandingkan dengan BGD.



Gambar 2.5: Dengan SGD, setiap training step lebih cepat tetapi juga lebih random dibandingkan dengan BGD

Oleh sebab itu, ketidakberaturan (*randomness*) merupakan hal baik untuk keluar dari optimum lokal, tetapi kerugiannya adalah algoritma tidak pernah *settle* pada nilai minimum. Solusi terhadap dilema ini adalah dengan secara gradual mengurangi *learning rate*. Step diinisialisasi cukup besar untuk memperoleh kecepatan dan keluar dari local minima, kemudian dibuat terus mengecil untuk mencapai global minimum. Fungsi untuk mengatur *learning rate* pada setiap iterasi disebut dengan *learning schedule*. Jika *learning rate* dikurangi terlalu cepat, kemungkinan algoritma akan tertahan pada local minima, atau bahkan tidak bergerak lagi sebelum mencapai local minima. Jika *learinig rate* teralu lambat dikurangi, algoritma akan loncat-loncat (*jump around*) disekitar minimum lokal dengan cukup lama sebelum akhirnya mencapai solusi suboptimal jikalau proses training distop teralu dini.

1.2.3 Mini-Batch Gradient Descent

Algoritma *gradient descent* yang terakhir aalah *Mini-Batch Gradient Descent* (MBGD). Akan mudah untuk memahami algoritma ini setelah kita mengerti algoritma BGD dan SGD. Daripada menghitung *gradient* berdasarkan training set secara full seperti pada BGD atau satu-satu seperti pada SGD, algoritma MBGD akan menghitung *gradient* pada setiap *step* berdasarkan subset kecil dari *training instances* yang dipilih secara random yang disebut dengan *mini-batches*. Keuntungan dari *mini-batch* dibandingkan *stochastic* adalah kita akan memperoleh peningkatan kinerja (*performance boost*) dari optimasi hardware dari operasi matriks terutama ketika menggunakan *Graphical Processing Unit* (GPU).

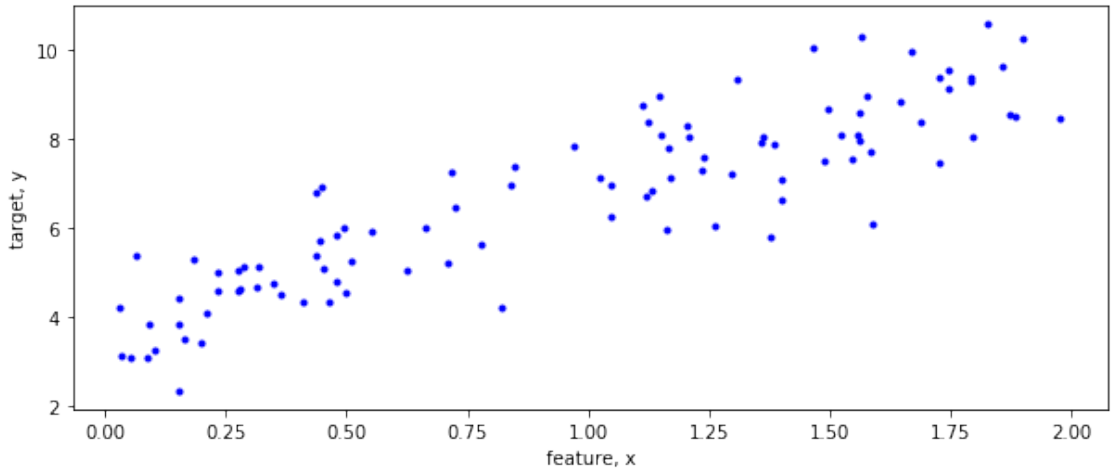
Progress dari algoritma MBGD pada ruang parameter (*parameter space*) lebih beraturan dibandingkand dengan SGD, terutama dengan *mini-batch* yang cukup besar. Sehingga, MBGD akan bergerak di sekitar posisi dengan parameter θ yang menghasilkan nilai minimum, dibandingkan SGD yang bergerak tidak beraturan (*eratic*), tetapi lebih sulit untuk keluar dari minimum lokal untuk mencapai minimum global. Gambar perbandingan pada contoh program `Python` di bawah akan mengilustrasikan perbedaan proses konvergen dari ketiga metode GD yang telah dibicarakan.

1.2.4 Contoh Program `Python` untuk Regresi Linier

Membangkitkan data yang terlihat linier dimana x sebagai fitur dan y adalah target

[49]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

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

plt.figure(figsize=(10,4))
plt.plot(x,y,"b.")
plt.xlabel('feature, x')
plt.ylabel('target, y')
```

[49]:
```
Text(0, 0.5, 'target, y')
```



Solusi dengan persamaan normal Menambahkan $x_0 = 1$ pada setiap baris input x

[50]:
```
X_p = np.c_[np.ones((100,1)), x]
```

[51]:
```
X_p[0:10,]
```

[51]:
```
array([[1.        , 1.20759398],
       [1.        , 1.72627108],
       [1.        , 1.6449893 ],
       [1.        , 0.34668611],
       [1.        , 0.97163486],
       [1.        , 0.23384707],
       [1.        , 1.36430632],
       [1.        , 1.89979934],
       [1.        , 0.27417892],
       [1.        , 1.46776626]])
```

Gunakan persamaan normal di atas untuk mendapatkan estimasi $\hat{\theta} = [\hat{\theta}_0 \ \hat{\theta}_1]^T$:

[52]:
```
theta_best = np.linalg.inv(X_p.T.dot(X_p)).dot(X_p.T).dot(y)
```

Maka akan diperoleh $\hat{\theta} = [\hat{\theta}_0 \ \hat{\theta}_1]^T$

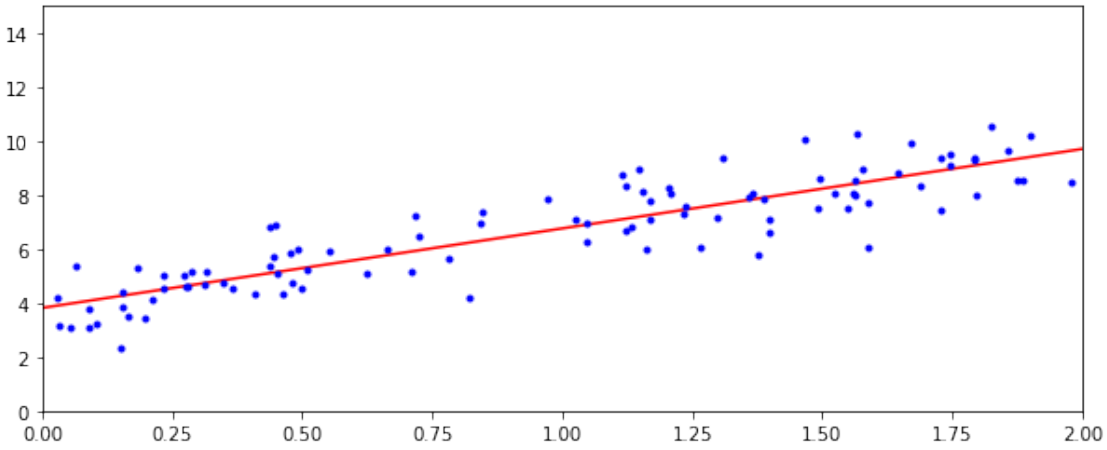
[53]:
```
theta_best
```

[53]:
```
array([[3.81554623],
       [2.94310113]])
```

[54]:
```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]  # add x0 = 1 to each instance
y_predict = X_new_b.dot(theta_best)
y_predict
```

[54]:
```
array([[3.81554623],
       [9.7017485 ]])
```

[55]:
```
plt.figure(figsize=(10,4))
plt.plot(X_new, y_predict, "r-")
plt.plot(x, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```



Solusi *Batch Gradient Descent* untuk menghitung $\hat{\theta} = [\hat{\theta}_0 \;\; \hat{\theta}_1]^T$

[56]:
```
alpha = 0.1
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)

for iteration in range(n_iterations):
    gradients = 2/m* X_p.T.dot(X_p.dot(theta)-y)
    theta = theta - alpha*gradients
```

[57]:
```
theta
```

[57]:
```
array([[3.81554623],
       [2.94310113]])
```

Terlihat perhitungan $\hat{\theta}$ menggunakan *Gradient Descent* sama dengan hasil dari persamaan normal.

Fungsi yang kita gunakan untuk mengenerate data adalah $y = 4 + 3x_1$ + Gaussian Noise, yang kita temukan dari $\hat{\theta}_0$ dekat ke 4 dan $\hat{\theta}_1$ dekat ke 3.

Gunakan hasil $\hat{\theta}$ untuk melakukan prediksi. Contoh di titik $x = 0$ dan $x = 2$ sekaligus disimpan pada array X_new:

[58]:
```
X_new = np.array([[0],[2]])
X_new_b = np.c_[np.ones((2,1)),X_new] # tambahkan x0 = 1 pada setiap training␣
↪example
y_predict = X_new_b.dot(theta_best)
```

[59]:
```
X_new_b
```

[59]:
```
array([[1., 0.],
       [1., 2.]])
```

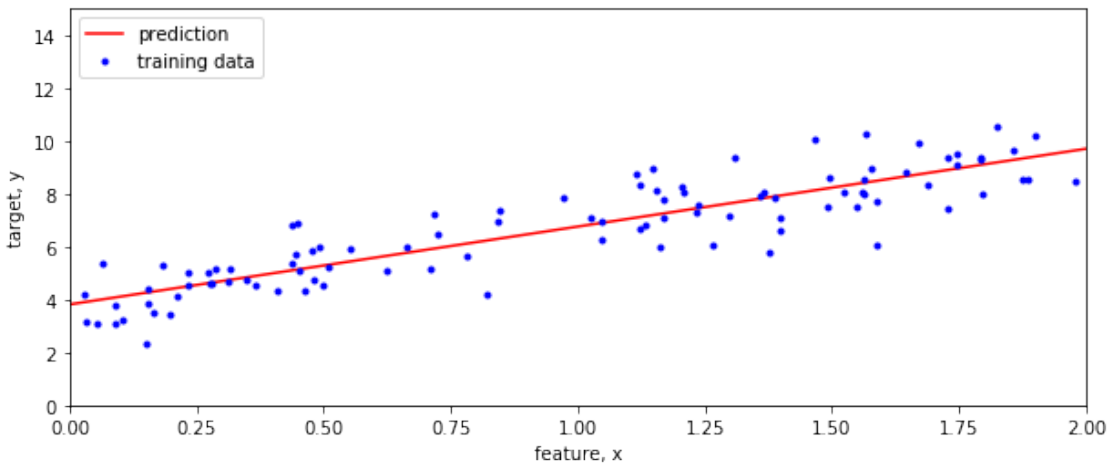
[60]:
```
y_predict
```

[60]:
```
array([[3.81554623],
       [9.7017485 ]])
```

Hasil plotting:

[61]:
```
plt.figure(figsize=(10,4))
plt.plot(X_new,y_predict,'r-',label='prediction')
plt.plot(x,y,'b.',label='training data')
plt.axis([0,2,0,15])
plt.xlabel('feature, x')
plt.ylabel('target, y')
plt.legend(loc='upper left')
plt.show
```

[61]:
```
<function matplotlib.pyplot.show(*args, **kw)>
```



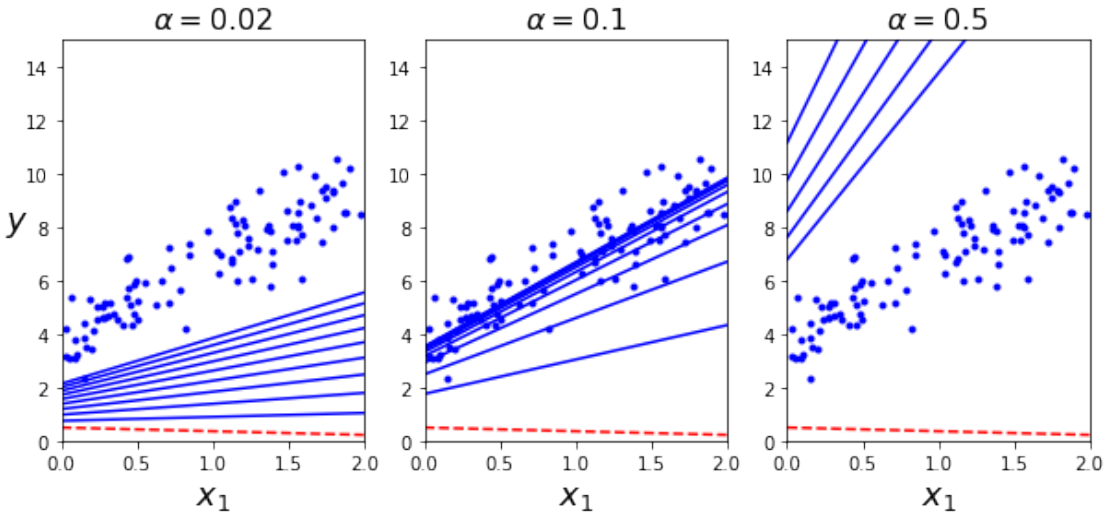
Catatan: Ketika mencari $\hat{\theta}$ dengan menggunakan *Gradient Descent* tidak akan selalu sama dengan hasil dari persamaan normal. Hasil akan berbeda untuk *learning rate* α yang berbeda juga. Untuk menemukan α yang tepat maka bisa digunakan ***grid search***.

Kita akan melihat pengaruh dari pemilihan *learning rate* α pada konvergensi solusi dari BGD.

[62]:
```
theta_path_bgd = []

def plot_gradient_descent(theta, alpha, theta_path=None):
    m = len(X_p)
    plt.plot(x, y, "b.")
```

```
    n_iterations = 1000
    for iteration in range(n_iterations):
        if iteration < 10:
            y_predict = X_new_b.dot(theta)
            style = "b-" if iteration > 0 else "r--"
            plt.plot(X_new, y_predict, style)
        gradients = 2/m * X_p.T.dot(X_p.dot(theta) - y)
        theta = theta - alpha * gradients
        if theta_path is not None:
            theta_path.append(theta)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 2, 0, 15])
    plt.title(r"$\alpha = {}$".format(alpha), fontsize=16)
```

[63]:
```
np.random.seed(42)
theta = np.random.randn(2,1)  # random initialization

plt.figure(figsize=(10,4))
plt.subplot(131); plot_gradient_descent(theta, alpha=0.02)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(132); plot_gradient_descent(theta, alpha=0.1,␣
 ↪theta_path=theta_path_bgd)
plt.subplot(133); plot_gradient_descent(theta, alpha=0.5)

plt.show()
```



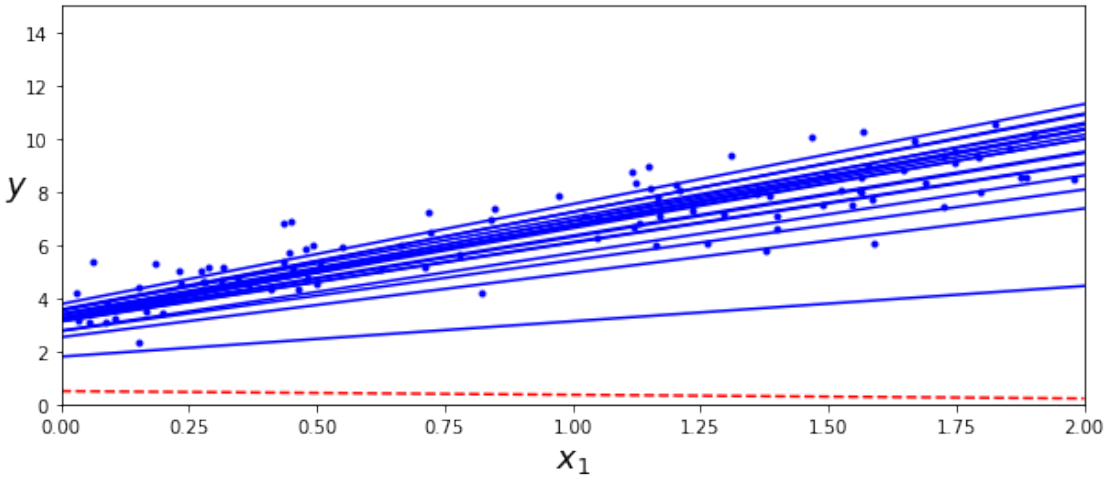
Gambar pertama, *learning rate* $\alpha = 0.02$ terlalu kecil sehingga ketika algoritma berhenti setelah melakukan 1000 iterasi belum konvergen ke nilai minimum. Gambar ketiga paling kanan, pemilihan *learning rate* $\alpha = 0.5$ terlalu besar sehingga hasilnya tidak konvergen. Sedangkan pemilihan

learning rate $\alpha = 0.1$ cukup tepat seperti gambar yang di tengah, dan dihasilkan konvergensi terhadap nilai parameter yang diinginkan. Hasil regresi linier akan serupa dengan hasil menggunakan persamaan normal. Garis strip merah adalah nilai permulaan ketika parameter θ diinisialisasi secara random.

Solusi *Stochastic Gradient Descent* untuk menghitung $\hat{\theta} = [\hat{\theta}_0 \ \hat{\theta}_1]^T$

[64]:
```
theta_path_sgd = []
m = len(X_p)
np.random.seed(42)
```

[65]:
```
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
plt.figure(figsize=(10,4))

for epoch in range(n_epochs):
    for i in range(m):
        if epoch == 0 and i < 20:                    # not shown in the book
            y_predict = X_new_b.dot(theta)           # not shown
            style = "b-" if i > 0 else "r--"         # not shown
            plt.plot(X_new, y_predict, style)        # not shown
        random_index = np.random.randint(m)
        xi = X_p[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        alpha = learning_schedule(epoch * m + i)
        theta = theta - alpha * gradients
        theta_path_sgd.append(theta)


plt.plot(x, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
plt.show()
```



[66]:
```
theta
```

[66]:
```
array([[3.84484929],
       [2.92601368]])
```

Terlihat perhitungan $\hat{\vec{\theta}}$ menggunakan *Stochastic Gradient Descent* mendekati hasil dari persamaan normal.

Solusi *Mini-Batch Gradient Descent* untuk menghitung $\hat{\theta} = [\hat{\theta}_0 \ \ \hat{\theta}_1]^T$

[67]:
```
theta_path_mgd = []

n_iterations = 50
minibatch_size = 20

np.random.seed(42)
theta = np.random.randn(2,1)  # random initialization

t0, t1 = 200, 1000
def learning_schedule(t):
    return t0 / (t + t1)

t = 0
for epoch in range(n_iterations):
    shuffled_indices = np.random.permutation(m)
    X_p_shuffled = X_p[shuffled_indices]
    y_shuffled = y[shuffled_indices]
    for i in range(0, m, minibatch_size):
        t += 1
        xi = X_p_shuffled[i:i+minibatch_size]
        yi = y_shuffled[i:i+minibatch_size]
```

```
        gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta) - yi)
        alpha = learning_schedule(t)
        theta = theta - alpha * gradients
        theta_path_mgd.append(theta)
```

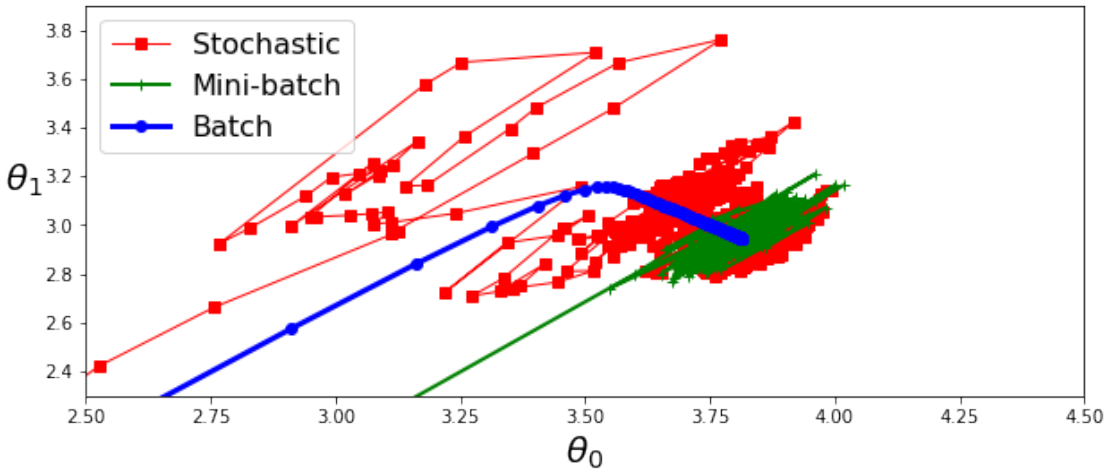
[68]:
```
theta
```

[68]:
```
array([[3.7574483],
       [2.869106 ]])
```

Terlihat perhitungan $\hat{\theta}$ menggunakan *Mini-Batch Gradient Descent* mendekati hasil dari persamaan normal. Program berikut menunjukan perbandingan proses konvergensi dari ketiga algoritma *gradient descent* yaitu: BGD, SGD dan MBGD.

[69]:
```
theta_path_bgd = np.array(theta_path_bgd)
theta_path_sgd = np.array(theta_path_sgd)
theta_path_mgd = np.array(theta_path_mgd)
```

[70]:
```
plt.figure(figsize=(10,4))
plt.plot(theta_path_sgd[:, 0], theta_path_sgd[:, 1], "r-s", linewidth=1,␣
 ↪label="Stochastic")
plt.plot(theta_path_mgd[:, 0], theta_path_mgd[:, 1], "g-+", linewidth=2,␣
 ↪label="Mini-batch")
plt.plot(theta_path_bgd[:, 0], theta_path_bgd[:, 1], "b-o", linewidth=3,␣
 ↪label="Batch")
plt.legend(loc="upper left", fontsize=16)
plt.xlabel(r"$\theta_0$", fontsize=20)
plt.ylabel(r"$\theta_1$    ", fontsize=20, rotation=0)
plt.axis([2.5, 4.5, 2.3, 3.9])
plt.show()
```



1.2.5 Regresi Polinomial (*Polynomial Regression*)

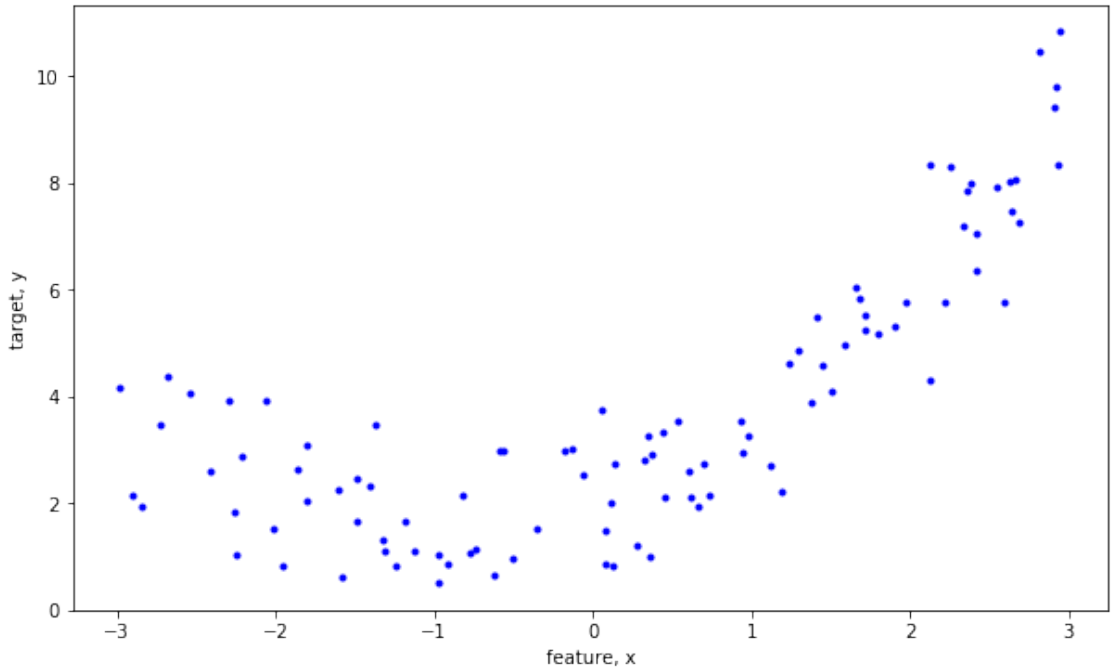
Bagaimana jika data ternyata lebih kompleks dibandingkan dengan sebuah garis lurus (non-linier)? Untuk kasus ini sebetulnya bisa digunakan model linier untuk *fitting* data non-linier. Paling sederhana dengan menambahkan pangkat (*power*) pada setiap *feature* sebagai *feature* baru, kemudian latih model linier dengan *extended feature* tersebut. Teknik ini disebut dengan Regresi Polinomial (*Polynomial Regression*).

Contoh berikut menunjukan teknik regresi polinomial berorde dua (*quadratic equation*).

- Membangkitkan data non-linier:

[71]:
```
import numpy as np
import matplotlib.pyplot as plt

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

#plotting data
plt.figure(figsize=(10,6))
plt.plot(X,y,'b.')
plt.xlabel('feature, x')
plt.ylabel('target, y')
plt.show
```

[71]:
```
<function matplotlib.pyplot.show(*args, **kw)>
```



- Dengan melihat data di atas, maka terlihat garis lurus tidak akan cocok *fitting* terhadap data tersebut. Akan digunakan *Scikit-Learn's Polynomial Features class* untuk mentransformasikan data training, menambahkan polinomial derajat dua pada masing-masing *feature* sebagai *feature* baru.

[72]:
```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree = 2, include_bias = False)
X_poly = poly_features.fit_transform(X)
```

[73]:
```
X[0]
```

[73]:
```
array([2.38942838])
```

[74]:
```
X_poly[0]
```

[74]:
```
array([2.38942838, 5.709368  ])
```

`X_poly` sekarang berisi *feature asal* dari `X` dan harga pangkat 2-nya. Akan di cari model regresi linier untuk training data yang di-*extend* tersebut.

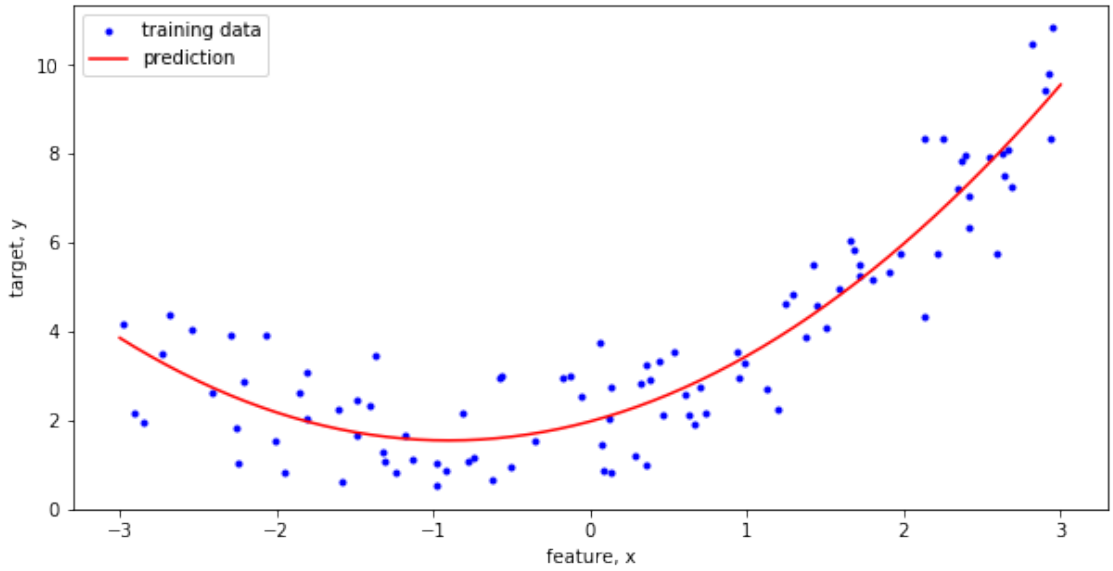
[75]:
```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X_poly,y)
lin_reg.intercept_, lin_reg.coef_
```

[75]:
```
(array([1.9735233]), array([[0.95038538, 0.52577032]]))
```

Estimasi model menghasilkan persamaan $\hat{y} = 0.52x^2 + 0.95x + 1.97$ ketika data sebetulnya berasal dari $y = 0.5x^2 + 1.0x + 2.0$

[77]:
```
x_lin = np.arange(-3,3.1,0.1)
y_est = lin_reg.coef_[0,1]*(x_lin**2)+lin_reg.coef_[0,0]*x_lin+lin_reg.intercept_

#plotting data
plt.figure(figsize=(10,5))
plt.plot(X,y,'b.',label='training data')
plt.plot(x_lin,y_est,'r-',label='prediction')
plt.xlabel('feature, x')
plt.ylabel('target, y')
plt.legend(loc='upper left')
plt.show
```

[77]:
```
<function matplotlib.pyplot.show(*args, **kw)>
```



Sebagai catatan, ketika *feature* banyak, Regersi Polinomial dapat mencari hubungan antar *feature* (dimana model regresi linier biasa tidak dapat melakukannya). Hal ini dimungkinkan karena `PolynomialFeatures` dari `Scikit-Learn` juga menambahkan semua kombinasi dari semua *feature* sampai pada orde yang diberikan. Sebagai contoh, jika kita punya dua *feature* a dan b, dengan `PolynomialFeatures` dengan derajat (`degree = 3`) tidak hanya menambahkan *feature* a^2, a^3, b^2, b^3, tetapi juga kombinasi ab, a^2b dan ab^2.

2 Regresi Logistik (*Logistic Regression*)

Pada bagian ini kita akan membicarakan klasifikasi dengan menggunakan regresi logistik (***logistic regression***). Mirip dengan masalah regresi biasa, tetapi harga y sebagai target hanya bernilai diskrit dan berjumlah kecil. Beberapa algoritma regresi bisa digunakan untuk klasifikasi (dan sebaliknya juga). Regresi logistik (disebut juga ***logit regression***) biasanya digunakan untuk mengestimasi probabilitas menyangkut sebuah data termasuk pada *class* tertentu. Jika estimasi probabilitas > 0.5 maka data dikategorikan sebagai ***positive class*** dengan label `1`, dan jika sebaliknya maka data tidak termasuk *positive class* tetapi ***negative class*** dengan label `0`.

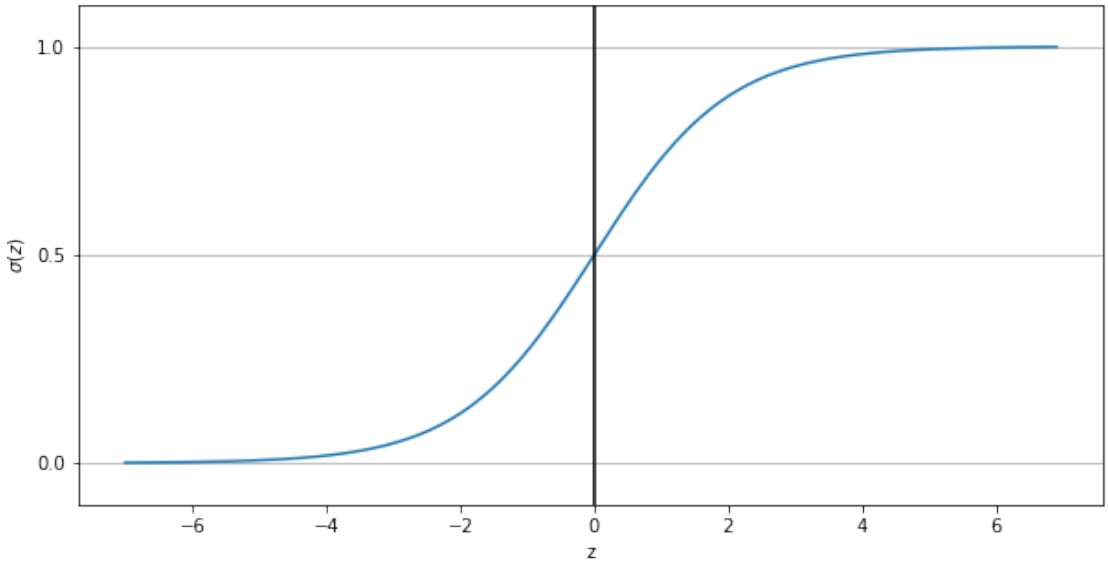
2.1 Fungsi Logistik dan Estimasi Probabilitas)

Kita dapat melakukan pendekatan masalah klasifikasi dengan mengabaikan sementara bahwa target y adalah bernilai diskrit, kemudian menggunakan algoritma regresi linier seperti sebelumnya untuk memprediksi y jika diberikan sebuah data *instance* x. Tetapi secara intuisi, karena berbicara probabilitas ($0 \leq p \leq 1$), maka kita dapat merubah hipotesis $h_\theta(x)$ kita menjadi:

> ***Persamaan (2.7).*** Model regresi logistik untuk estimasi probabilitas (disebut juga hipotesis)
>
> $$\hat{p} = h_\theta(x) = \sigma(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

dimana, $\sigma(z) = \frac{1}{1+e^{-z}}$ adalah **fungsi logistik (*logistic function*)** atau **fungsi sigmoid**. Persamaan (2.7) juga menyatakan estimasi probabilitas $\hat{p}$. (Catatan: bandingkan hipotesis $h_\theta(x)$ Persamaan (2.7) untuk regresi logistik dan (2.2) untuk regresi linier). Berikut adalah program yang menampilkan fungsi logistik atau fungsi sigmoid.

[78]:
```
import matplotlib.pyplot as plt
import numpy as np

def sigmoid(z):
    return 1.0/(1+np.exp(-z))

z = np.arange(-7,7,0.1)
gama_z = sigmoid(z)

plt.figure(figsize=(10,5))
plt.plot(z,gama_z)
plt.axvline(0.0,color='k')
plt.ylim(-0.1,1.1)
plt.xlabel('z')
plt.ylabel('$\sigma (z)$')
# y axis ticks and gridline
plt.yticks([0.0, 0.5,1.0])
ax = plt.gca()
ax.yaxis.grid(True)
plt.show()
```



Berdasarkan persamaan (2.7) model prediksi untuk regresi logistik ditunjukan pada Persamaan (2.8).

Persamaan (2.8). Model prediksi untuk regresi logistik

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

Dapat dilihat bahwa jika $\sigma(z) < 0.5$ ketika $z < 0$ dan $\sigma(z) \geq 0.5$ ketika $z \geq 0$, maka model regresi logistik akan memprediksi *positive class* (label `1`) jika $\theta^T x$ adalah positif, dan memprediksi *negative class* (label `0`) jika $\theta^T x$ adalah negatif.

2.2 Training dan *Cost Function*

Setelah mengetahui bagaimana regresi logistik digunakan untuk estimasi probabilitas dan membuat prediksi, pertanyaan selanjutnya adalah bagaimana melakukan training? Objektif dari training adalah untuk mencari parameter sehingga diperolah vektor θ yang menghasilkan probabilitas besar untuk *positive instance* ($y = 1$) dan probabilitas kecil untuk *negative instance* ($y = 0$). Ide ini direalisasikan dalam bentuk ***Cost Function*** untuk satu sampel data training.

Persamaan (2.9). *Cost function untuk satu sampel data training (single training instance)*

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

Terlihat bahwa jika keputusan salah untuk *positive instance* maka $-\log(\hat{p})$ membesar karena $\hat{p} \to 0$, sebaliknya jika keputusan salah untuk *negative instance* maka $-log(1 - \hat{p})$ membesar karena $\hat{p} \to 1$. Jika keputusan betul untuk *positive instance* maka $-\log(\hat{p})$ mengecil karena $\hat{p} \to 1$, dan jika keputusan betul untuk *negative instance* maka $-log(1 - \hat{p})$ mengecil karena $\hat{p} \to 0$. Dengan kondisi tersebut akan selalu diperoleh *cost function* yang besar untuk keputusan salah, dan *cost function* kecil untuk keputusan benar.

Cost function untuk keseluruhan dataset training adalah rata-rata *cost function* dari setiap *training instance*, yang dapat dituliskan sebagai berikut dan disebut dengan ***log loss***.

Persamaan (2.10). *Log loss merupakan lost function untuk regresi logistik*

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

Pada kasus ini tidak ada solusi bentuk tertutup (*closed form*) untuk menghitung harga $\vec{\theta}$ yang meminimalkan $J(\hat{\theta})$, seperti Persamaan Normal untuk regresi linier. Tetapi karena *cost function* adalah *convex* maka bisa digunakan *Gradient Descent* yang dijamin akan konvergen ke nilai *global minimum* jika menggunakan *learning rate* yang cukup kecil dan agak lama untuk konvergen.

Turunan parsial dari *log loss* terhadap parameter model ke-j (θ_j) dapat dilihat pada Persamaan (2.11).

Persamaan (2.11). *Turunan parsial dari cost function (log loss)*

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

Persamaan (2.11) mempunyai kesamaan dengan persamaan turunan parsial pada kasus regresi linier, yaitu untuk setiap *instance* dihitung error prediksi dan mengalikannya dengan harga *feature* ke j^{th} dan dihitung rata-rata untuk keseluruhan data training. Setelah diperoleh vektor gradien yang merupakan turunan parsial terhadap setiap parameter $\theta_j, \forall j = 1, 2, \cdots, n$, algoritma *Batch*

Gradient Descent bisa digunakan kembali. Implementasi *Stochastic Gradient Descent* sama dengan implementasi sebelumnya pada kasus regresi linier. Perbedaan terletak pada perhitungan menggunakan fungsi sigmoid.

2.3 Contoh Python untuk Regresi Logistik

Pada contoh ini kita akan menggunakan regresi logistik untuk mengklasifikasikan dataset *iris* yang sangat banyak digunakan sebagai contoh pada buku-buku *Machine Learning*, yang dapat diperoleh juga dari UCI, Machine Learning Repository. Data set ini terdiri dari panjang dan lebar bagian *sepal* dan *petal* dari 150 bunga iris dengan tiga spesies yang berbeda, yaitu: *Iris sentosa, Iris versicolor* dan *Iris virginica*.

Gambar 2.6. Ilustrasi parameter sepal dan petal pada bunga Iris.

Pada contoh ini akan dibuat *classifier* untuk mendeteksi jenis *Iris virginica* (dua spesies lain akan dikategorikan sebagai bukan), berdasarkan lebar *petal* saja.

- Load Dataset

[79]:
```
from sklearn import datasets
iris = datasets.load_iris()

list(iris.keys())
```

[79]:
```
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
```

[80]:
```
X = iris['data'][:,3:]  #lebar petal
y = (iris['target']==2).astype(np.int) # 1 jika iris virginica dan yang lain 0
```

- Training dengan regresi logistik

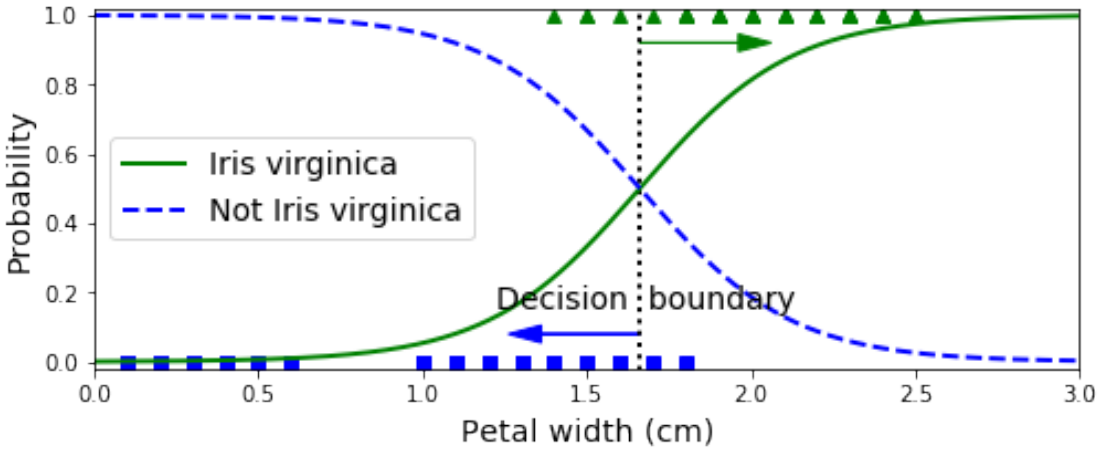
[81]:
```
from sklearn.linear_model import LogisticRegression
```

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

[81]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

- Hasil estimasi probabilitas

[82]:
```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0]

plt.figure(figsize=(8, 3))
plt.plot(X[y==0], y[y==0], "bs")
plt.plot(X[y==1], y[y==1], "g^")
plt.plot([decision_boundary, decision_boundary], [-1, 2], "k:", linewidth=2)
plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2, label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2, label="Not Iris virginica")
plt.text(decision_boundary+0.02, 0.15, "Decision  boundary", fontsize=14,␣
 ↪color="k", ha="center")
plt.arrow(decision_boundary, 0.08, -0.3, 0, head_width=0.05, head_length=0.1,␣
 ↪fc='b', ec='b')
plt.arrow(decision_boundary, 0.92, 0.3, 0, head_width=0.05, head_length=0.1,␣
 ↪fc='g', ec='g')
plt.xlabel("Petal width (cm)", fontsize=14)
plt.ylabel("Probability", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.axis([0, 3, -0.02, 1.02])
plt.show()
```



Lebar petal dari *Iris virginica* (dinyatakan dengan segitiga pada gambar) mempunyai range antara 1.4 cm sampai dengan 2.5 cm, sedangkan bunga *Iris* jenis lain (direpresentasikan dengan kotak pada gambar) mempunyai range antara 0.1 cm sampai dengan 1.8 cm. Terlihat pada gambar terdapat sedikit *overlap*. Jika terdapat data bunga baru dengan lebar petal di atas 2 cm, *classifier* sangat *confident* mengkategorikan bunga tersebut sebagai *Iris virginica* (*classifier* akan menghasilkan probabilitas terbesar untuk *class* tersebut), sementara untuk ukuran lebih kecil dari 1 cm maka dengan sangat *confident*, *classifier* akan mengkategorikan bunga tersebut bukan sebagai *Iris virginica* dengan probabilitas tertinggi. Diantara kedua ekstrim ini, *classifier* kurang tidak *confident* untuk memutuskan. Tetapi kita bisa menggunakan metode `predict()` sebagai pengganti `predict_proba()`, yang akan menghitung hasil prediksi *class* yang paling mungkin. Oleh karena itu, ada *decision boundary* di sekitar 1, 6 cm dimana kedua probabilitias *class* sama yaitu 50%.

Contoh, ketika lebar petal 1,7 dan 1,5 cm, maka hasil prediksi *class* adalah:

[83]:
```
log_reg.predict([[1.7], [1.5]])
```

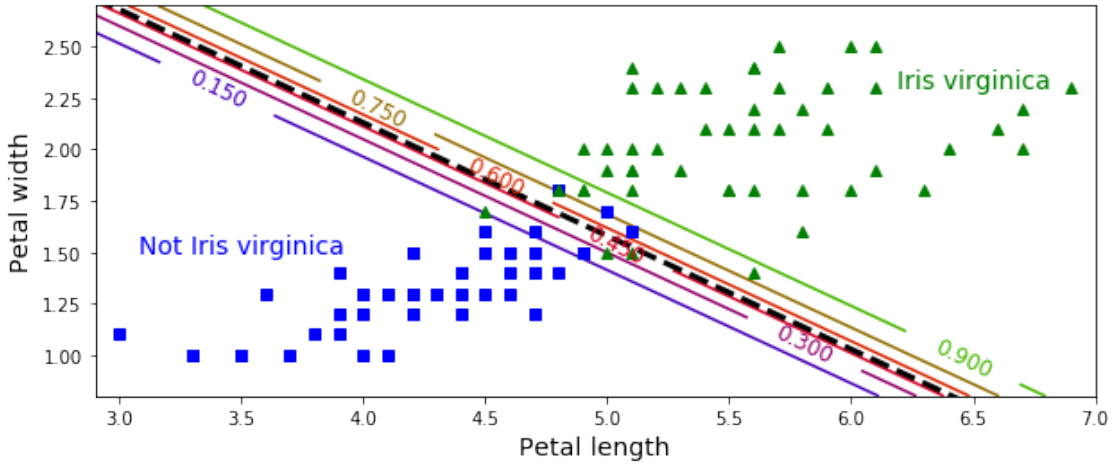
[83]:
```
array([1, 0])
```

Untuk lebar petal 1,7 maka diprediksi sebagai *class* 1 atau *Iris virginica*, sedangkan untuk 1,5 diprediksi sebagai *class* 0 atau bukan *Iris virginica*.

Sedangkan pada program berikut akan ditunjukan *classifier* dengan menggunakan dua *feature* yaitu panjang petal dan lebar petal. Sekali ditraining, *classifier* dengan regresi logistik dapat melakukan estimasi probabilitas bahwa bunga tertentu termasuk *Iris virginica* berdasarkan dua *feature* tersebut.

[84]:
```
from sklearn.linear_model import LogisticRegression

X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.int)

log_reg = LogisticRegression(solver="lbfgs", C=10**10, random_state=42)
log_reg.fit(X, y)

x0, x1 = np.meshgrid(
        np.linspace(2.9, 7, 500).reshape(-1, 1),
        np.linspace(0.8, 2.7, 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]

y_proba = log_reg.predict_proba(X_new)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs")
plt.plot(X[y==1, 0], X[y==1, 1], "g^")

zz = y_proba[:, 1].reshape(x0.shape)
```

```
contour = plt.contour(x0, x1, zz, cmap=plt.cm.brg)


left_right = np.array([2.9, 7])
boundary = -(log_reg.coef_[0][0] * left_right + log_reg.intercept_[0]) / log_reg.
 ↪coef_[0][1]

plt.clabel(contour, inline=1, fontsize=12)
plt.plot(left_right, boundary, "k--", linewidth=3)
plt.text(3.5, 1.5, "Not Iris virginica", fontsize=14, color="b", ha="center")
plt.text(6.5, 2.3, "Iris virginica", fontsize=14, color="g", ha="center")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.axis([2.9, 7, 0.8, 2.7])
plt.show()
```



Berdasarkan gambar di atas, garis strip hitam menyatakan titik-titik dimana model menghasilkan estimasi probabilitas sebesar 50%, atau disebut juga *model decision boundary*. Setiap garis paralel merepresentasikan hasil estimasi model untuk probabilitias dari 15% (bawah kiri) sampai dengan 90% (atas kanan). Semua bunga yang mempunyai posisi berada di atas garis 90% (0.900) maka mempunyai probabilitias 90% berasal dari *Iris virginica* berdasarkan model tersebut.

> Apakah anda tahu mengapa *decision boundary* pada gambar di atas berbentuk garis lurus?

2.4 Regresi Softmax (*Softmax Regeression*)

Model regresi logistik dapat diperluas atau digeneralisasi untuk kasus *multiclasses* (lebih dari 2) secara langsung tanpa harus mentraining dan mengkombinasikan beberapa *binary classifier* (lihat Chapter 3). Model ini disebut dengan ***Softmax Regression*** atau ***Multinomial Logistic Regression***

Ide dari metode ini cukup sederhana, yaitu: 1. Ketika diberikan sebuah *instance* x (skalar atau

vektor), model regresi softmax akan memulai dengan menghitung sebuah *score* $s_k(x)$ untuk setiap *class* k. 2. Kemudian mengestimasi probabilitas untuk setiap *class* k dengan menerapkan fungsi softmax (atau disebut juga *normalized exponential*) pada *score* $s_k(x)$.

Persamaan berikut untuk menghitung *score* $s_k(x)$ mirip dengan persamaan untuk menghitung regresi linier.

Persamaan (2.12). *Softmax score* untuk *class* k

$$s_k(x) = x^T \theta^{(k)}$$

Sebagai catatan, setiap *class* k mempunyai vektor parameter masing-masing $\theta^{(k)}$. Semua vektor parameter untuk keseluruhan *class* biasanya disimpan sebagai baris-baris dari matriks parameter θ.

Setelah ditentukan *score score* $s_k(x)$, maka perhitungan probabilitas $\hat{p}_k$ bahwa sebuah *instance* termasuk *class* k adalah dengan Persamaan (2.13), yang disebut dengan fungsi *softmax*.

Persamaan (2.13). Fungsi *Softmax*

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

pada persamaan ini, K adalah jumlah *class*.

Seperti regresi logistik, *classifier* regresi softmax meprediksi *class* dengan estimasi probabilitas tertinggi yang tentunya merupakan *class* dengan *score* tertinggi, seperti ditunjukan dengan Persamaan (2.14).

Persamaan (2.14). Prediksi dengan *classifier* regresi softmax

$$\hat{y} = \underset{k}{\text{argmax}} \sigma\left(s_k(x)\right) = \underset{k}{\text{argmax}} s_k(x)$$

Operator `argmax` mengembalikan harga dari sebuah variabel yang memaksimalkan fungsi objektif. Pada persamaan di atas, akan mengembalikan harga k yang memaksimalkan estimasi probabilitas $\sigma\left(s_k(x)\right)$. Setelah kita mengetahui model untuk mengestimasi probabilitas dan membuat prediksi, maka pertanyaan selanjutnya adalah bagaimana training dilakukan.

Objektif dari training adalah menghasilkan model yang dapat mengestimasi probabilitas dengan hasil tertinggi untuk *class* target (dan tentunya probabilitas rendah untuk *class* lain. Objektif tersebut dapat dicapai dengan cara meminimalkan *cost function* pada Persamaan (2.15), yang disebut dengan *cross entropy*, karena *cross entropy* akan memberikan *penalty* terhadap model yang memberikan estimasi dengan probabilitas rendah untuk *class* target. *Cross entropy* biasanya digunakan untuk mengukur seberapa baik satu set probabilitas-probabilitas *class* cocok dengan *class-class* target.

Persamaan (2.15). *Cross entropy cost function* regresi softmax

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

Dimana $y_k^{(i)}$ adalah probabilitas target bahwa *instance* ke-i termasuk pada *class* k. Secara umum, sama dengan 1 atau 0, tergantung apakah *instance* tersebut termasuk pada *class* yang dimaksud atau tidak.

Perhatikanlah bahwa jika hanya terdapat dua *class* (K=2), *cross entropy* akan ekivalen dengan *log loss* Persamaan (2.10) yang merupakan *cost function* untuk regresi logistik.

Gradient vector untuk *cost function* ini terhadap $\theta^{(k)}$ diberikan oleh Persamaan (2.16).

Persamaan (2.16). *Cross entropy gradient vector* untuk *class* k

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

Dengan persamaan ini kita menghitung *gradient vector* untuk setiap *class*, kemudian gunakan *gradient descent* (atau algoritma optimasi lain) untuk menentukan parameter matrix Θ yang meminimalkan *cost function*.

2.5 Contoh Python untuk Regresi Softmax (*Softmax Regeression*)

Kita bisa gunakan regresi softmax untuk mengklasifikasikan bunga Iris kedalam 3 *class*.`LogisticRegression` pada `Scikit-Learn` otomatis menggunakan metode *one-versus-the rest* (lihat chapter selanjutnya) ketika ditraining menggunkanan *class* lebih dari tiga. Tetapi kita masih bisa menggunakan *hyperparameter multi_class* dengan setting *multinomial* untuk berpindah ke regresi *softmax*. Kita juga diharuskan untuk menspesifikasikan sebuah *solver* yang mendukung regresi *softmax*, seperti `lbfgs` (lihat dokumentasi Scikit-Learn). Program ini juga menerapkan regularisasi l_2 secara default, yang bisa dikontrol menggunakan *hyperparameter* C.

[85]:
```
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10,␣
 ↪random_state=42)
softmax_reg.fit(X, y)
```

[85]:
```
LogisticRegression(C=10, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='multinomial', n_jobs=None, penalty='l2',
                   random_state=42, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

[86]:
```
softmax_reg.predict_proba([[5, 2]])
```

[86]:
```
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

Ketika kita diminta untuk mengklasifikasikan jenis bunga Iris dengan panjang 5 cm dan lebar 2 cm, maka bisa ditulis sebagai berikut:

[87]:
```
softmax_reg.predict([[5, 2]])
```

[87]:
```
array([2])
```

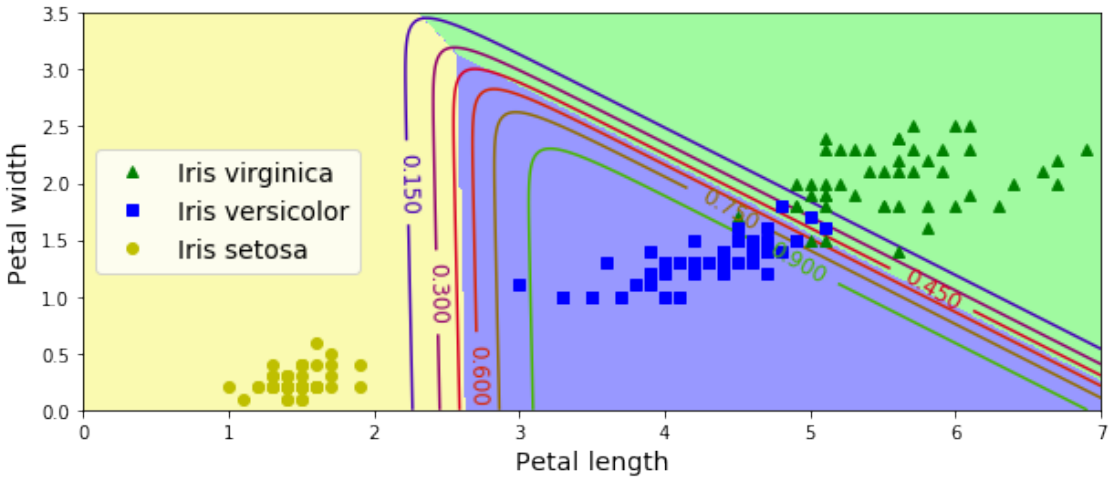
[88]:
```
softmax_reg.predict_proba([[5, 2]])
```

[88]:
```
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

dan bunga tersebut akan diklasifikasikan sebagai *Iris versicolor* (*class* 2) dengan probabilitas tertinggi yaitu 94,2%.

Decision boundary dari tiga *class* yang diklasifikasikan dapat digambar dengan menggunakan program berikut.

[89]:
```
x0, x1 = np.meshgrid(
        np.linspace(0, 8, 500).reshape(-1, 1),
        np.linspace(0, 3.5, 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]


y_proba = softmax_reg.predict_proba(X_new)
y_predict = softmax_reg.predict(X_new)

zz1 = y_proba[:, 1].reshape(x0.shape)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==2, 0], X[y==2, 1], "g^", label="Iris virginica")
plt.plot(X[y==1, 0], X[y==1, 1], "bs", label="Iris versicolor")
plt.plot(X[y==0, 0], X[y==0, 1], "yo", label="Iris setosa")

from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap)
contour = plt.contour(x0, x1, zz1, cmap=plt.cm.brg)
plt.clabel(contour, inline=1, fontsize=12)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.axis([0, 7, 0, 3.5])
plt.show()
```



Dari gambar terlihat bahwa *decision boundary* antara dua *class* adalah linier. Gambar tersebut juga menunjukan probabilitas untuk *class Iris versicolor* yang direpresentasikan dengan kurva-kurva bergaris. Sebagai contoh, garis dengan label 0.450 menunjukan boundary probabilitas 45% bahwa bunga akan diklasifikasikan sebagai *Iris versicolor*. Dapat kita lihat pula bahwa model dapat memprediksi sebuah *class* dengan harga estimasi probabilitas lebih kecil dari 50%. Contohnya adalah daerah dimana ketiga *decision boundary* bertemu, setiap *class* mempunyai estimasi probabilitas sekitar 33%.