# SKENARIO PEMODELAN

## 1.1 Pembagian Data

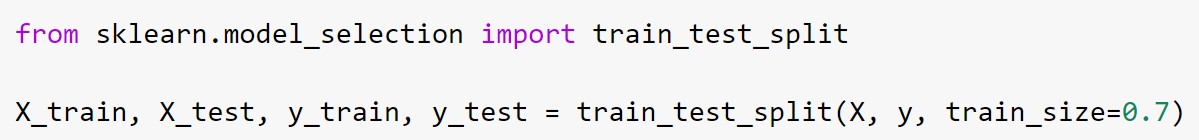
### 1.1.1 Train -Validation Split

Pembagian data sangat sering ditemukan dalam pembelajaran mesin. Dengan melakukan pembagian data, kita dapat menemukan hyperparameter terbaik untuk model yang digunakan. Dengan kata lain, pembagian data dapat membantu mengurangi risiko *underfitting* maupun *overfitting* yang seringkali ditemui oleh nilai hyperparameter yang kurang baik. Selain itu, pembagian data juga dapat digunakan sebagai indikator baik atau buruknya kemampuan model prediktif untuk menyelesaikan suatu masalah tertentu. Secara umum data dapat dibagi menjadi 3 bagian, yaitu data latih (*training data*), data validasi (*validation data*), dan data uji (*testing data*):

1. Data latih atau biasa disebut dengan *training data* merupakan bagian data yang digunakan saat melatih model.
2. Data validasi (*validation data*) adalah data yang digunakan untuk mengevaluasi model yang telah dilatih dengan *training data.* Dengan menguji model yang telah dilatih menggunakan *validation data*, performa model-model dengan *hyperparameter* berbeda akan dibandingkan dan diketahui *hyperparameter* mana yang menyebabkan *underfitting, overfitting,* dan yang menghasilkan *fit* model terbaik.
3. Data uji atau yang biasa disebut dengan *testing data* merupakan data yang digunakan untuk menguji performa suatu algoritme pembelajaran mesin satu dengan yang lainnya (misalkan membandingkan performa algoritme *Logistic Regression* dengan algoritme *Support Vector Machine* dalam klasifikasi digit MNIST).Namun dalam beberapa literatur, data validasi sering dianggap sama dengan data uji.

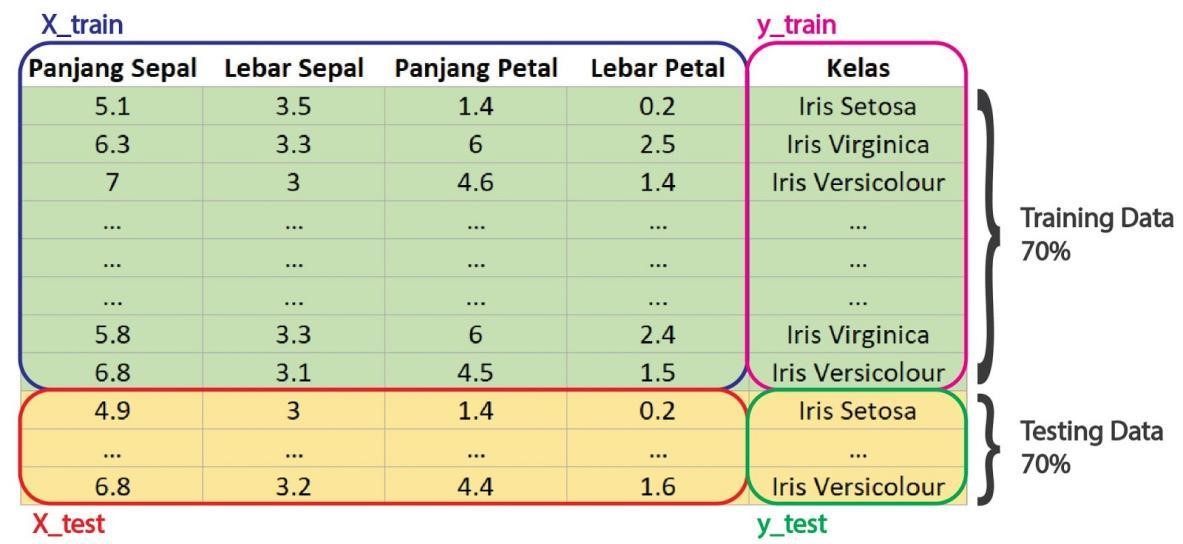
Penggunaan pembagian data menjadi kewajiban dalam menangani data yang jumlahnya besar. Hal ini dikarenakan jika data yang digunakan sedikit, model akan dilatih pada *training data* yang lebih sedikit juga dan menyebabkan tidak terlatihnya model dengan baik. Selain itu *validation data* yang sedikit juga menyebabkan generalisasi model menjadi terlalu “optimis” atau dikatakan memiliki bias yang tinggi. Secara umum, persentase pembagian data untuk *training data*:*validation data* yang sering digunakan adalah 66.6%:33.3%, 75%:25%, dan 80%:20%.

Pada bahasa pemrograman Python, pembagian data dapat dilakukan dengan menggunakan pustaka *Scikit-Learn* dengan menjalankan potongan kode pada Gambar 1.



**Gambar 1. Kode untuk *Train-Validation Split***

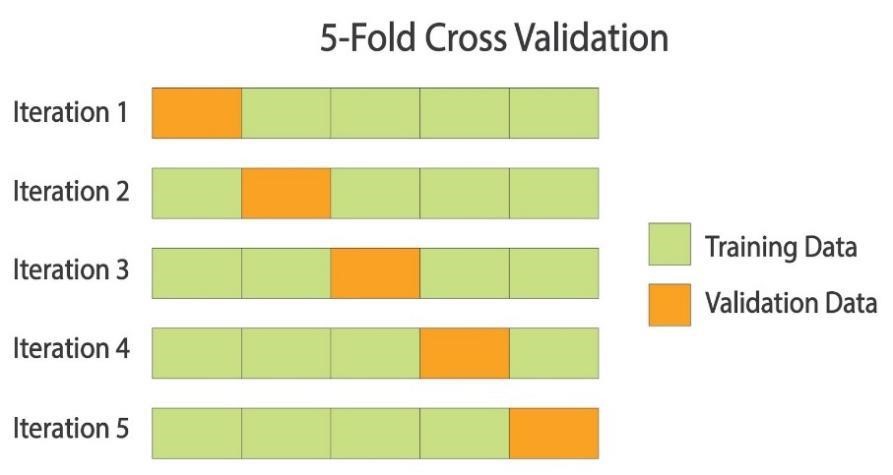
Dimana *X* merupakan fitur input, *y* merupakan label output, dan *train\_size* merupakan berapa persen dari dataset yang akan digunakan menjadi *training data. X\_train* dan *y\_train* merupakan pasangan fitur input dan label output dari *training data*, sementara *X\_test* dan *y\_test* merupakan pasangan fitur input dan label output dari *validation data / testing data.* Misalkan dataset berjumlah 100.000, ketika *train\_size* ditentukan sebesar 0.7, maka, akan terdapat 70.000 fitur input dan label output *training data* dan 30.000 sisanya merupakan fitur input dan label output *validation data / testing data.* Pembagian data diilustrasikan pada Gambar 2.



**Gambar 2. Ilustrasi pembagian data**

### 1.1.2 *k*-Fold Cross Validation

Selain *Train-Validation Split*, terdapat metode pembagian data lain yang sering ditemui dalam pembelajaran mesin, yaitu *K-Fold Cross Validation*. Metode ini seringkali digunakan ketika ukuran dataset tidak terlalu besar. *K-Fold Cross Validation* bekerja dengan melatih dan mengevaluasi model pada sejumlah *K* kombinasi *training data* dan *testing data* yang berbeda. Dengan demikian, model dilatih dengan data yang tidak terlalu besar akan memiliki kemampuan generalisasi yang lebih baik dibandingkan dengan menggunakan *Train-Validation Split* (Ingat bahwa *Train-Validation Split* tidak baik jika digunakan pada data yang tidak terlalu besar)*.* Ilustrasi sederhana dari *K-Fold Cross Validation* disajikan pada Gambar 3.



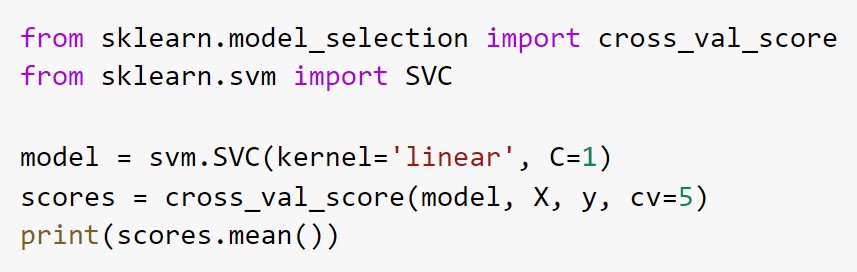
**Gambar 3. Ilustrasi 5-*Fold Cross Validation***

Sebagai contoh, misalkan terdapat data sebanyak 100.000 dan ditentukan nilai *K* = 5, maka index pembagian data untuk setiap iterasi adalah sebagai berikut:

1. Iterasi 1:
   1. Index Training Data: 20.001 sampai 100.000
   2. Index Testing Data: 1 sampai 20.000
2. Iterasi 2:
   1. Index Training Data: 1 sampai 20.001 + 40.001 sampai 100.000
   2. Index Testing Data: 20.001 sampai 40.000 3. Iterasi 3:
   3. Index Training Data: 1 sampai 40.000 + 60.001 sampai 100.000
   4. Index Testing Data: 40.001 sampai 60.000 4. Iterasi 4:
   5. Index Training Data: 1 sampai 60.000 + 80.001 sampai 100.000
   6. Index Testing Data: 60.001 sampai 80.000 5. Iterasi 5:
   7. Index Training Data: 1 sampai 80.000
   8. Index Testing Data: 80.001 sampai 100.000

Performa model (misalkan akurasi) dari setiap iterasi kemudian dirata-rata dan dibandingkan dengan performa model lain yang dibentuk dengan *hyperparameter* berbeda.

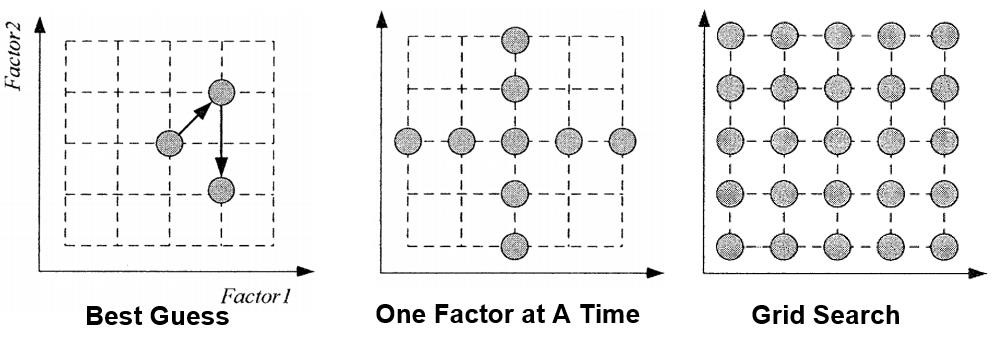
Pada bahasa pemrograman Python, *K-Fold Cross Validation* dapat dilakukan dengan menggunakan pustaka *Scikit-Learn* dengan menjalankan potongan kode yang ditampilkan pada Gambar 4.



**Gambar 4. Kode untuk *K-Fold Cross Validation***

Pada potongan kode di atas, *X* adalah fitur input, *y* adalah output label, *cv* adalah nilai *K* untuk *K-Fold Cross Validation,* dan model yang digunakan adalah *Support Vector Machine* (SVM)*.* Dengan menggunakan potongan kode di atas, proses pembagian data untuk setiap iterasi *K* dilakukan otomatis, sehingga tidak perlu repot untuk membagi data dengan index tertentu. Hasil akurasi untuk setiap iterasi (dalam potongan kode sebanyak 5 iterasi) disimpan di dalam list *scores.* Fungsi *print(scores.mean())* dipanggil untuk mengetahui rerata akurasi dari setiap iterasi.

## 1.2 Menentukan Langkah Eksperimen



**Gambar 5. Ilustrasi strategi pencarianparameter terbaik**

Setiap algoritma dalam pembangunan model memiliki parameter tertentu. Untuk mendapatkan model terbaik, diperlukan eksperimen dengan beberapa variasi parameter. Parameter yang menghasilkan model dengan performa terbaik akan dipilih. Terdapat beberapa strategi pencarian parameter untuk menghasilkan model terbaik, yaitu : (1) Best Guess, (2) One Factor at Time, (3)Grid Search. Ilustrasi strategi pencarian parameter terbaik ditunjukkan pada Gambar 5.

## 1.3 Parameter Evaluasi

Untuk mengetahui performa model yang dibangun, diperlukan parameter evaluasi.

Terdapat beberapa parameter evaluasi berdasarkan jenis task-nya, yaitu

1. Task klasifikasi

Beberapa parameter evaluasi : Akurasi, Presisi, Recall/Sensitivity, Specificity, F1-measure

1. Task regresi

Beberapa parameter evaluasi : MSE (*Mean Squared Error*), MAPE (*Mean Absolute Percentage Error*)

1. Task klastering

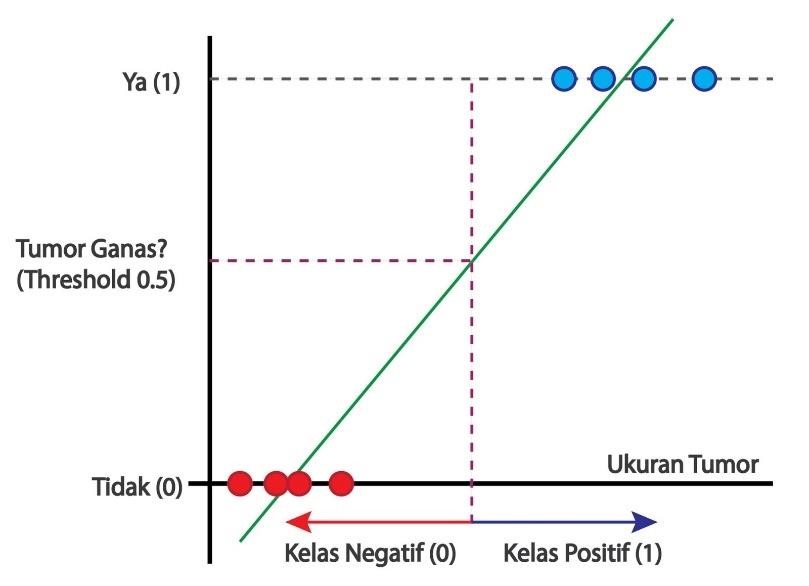
Beberapa parameter evaluasi : Silhouette Score, Davies-Bouldin Index

Parameter Evaluasi akan dijelaskan secara detail pada modul – modul berikutnya.

# II. METODE KLASIFIKASI

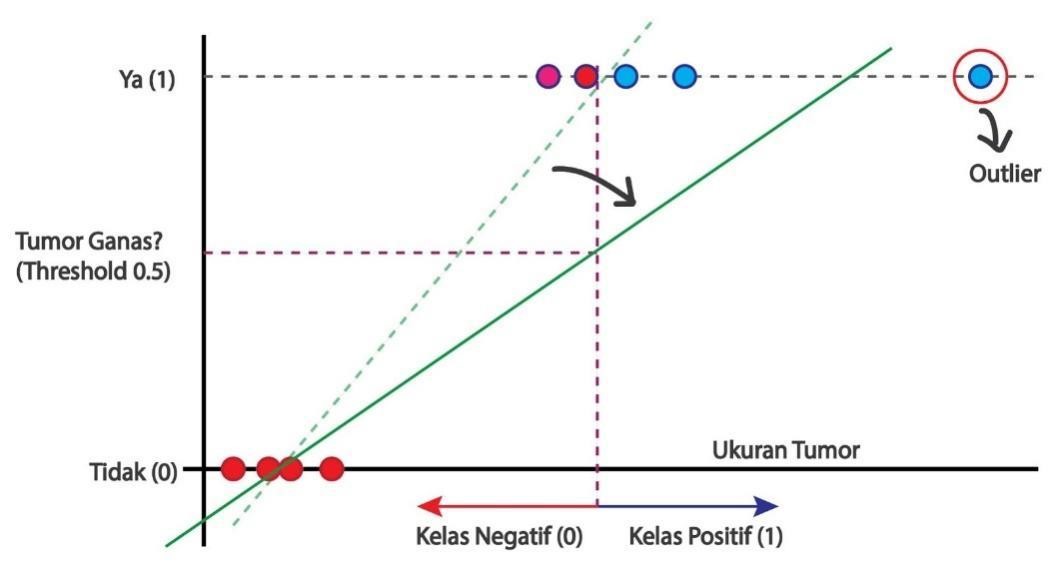
## 2.1 Logistic Regression

*Logistic Regression* adalah algoritma regresi yang digunakan untuk menyelesaikan masalah klasifikasi. Algoritma merupakan pengembangan dari algoritma *Linear Regression* dan bekerja dengan mencari *Logistic Function* (fungsi logistik) yang dapat membagi data menjadi 2 kelas dengan baik. Mengapa dikatakan pengembangan dari *Linear Regression*?



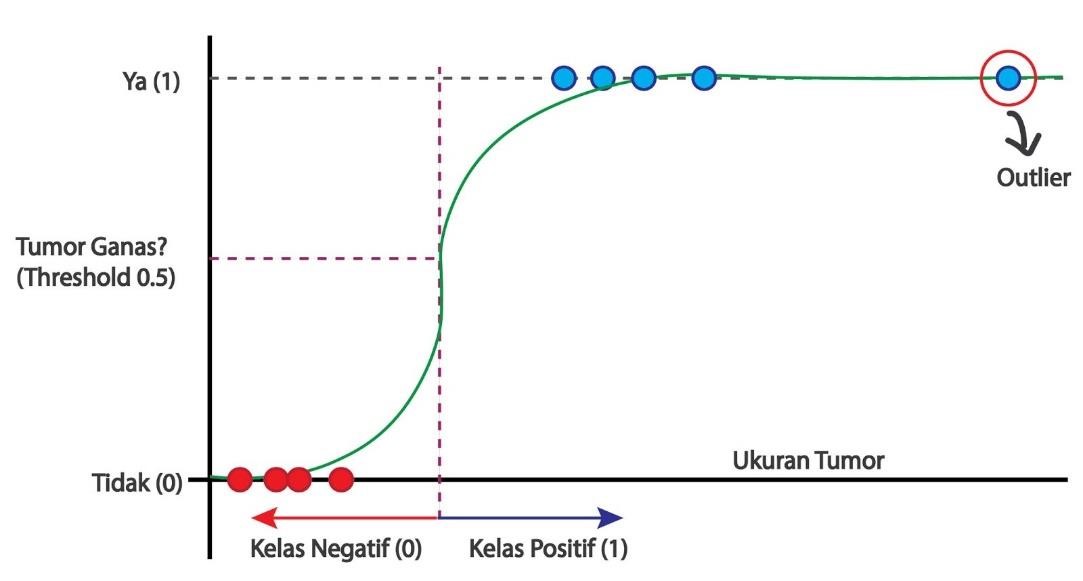
**Gambar 6. *Linear Regression***

Misalkan terdapat data ukuran tumor dan tingkat keganasannya. Ukuran tumor yang semakin besar akan menyebabkan tumor semakin ganas juga. Pada gambar di atas garis hijau merupakan garis *Linear Regression* yang memiliki *Fit* terbaik untuk 8 titik data. Garis linear di atas terlihat dapat mengklasifikasi kelas tumor ganas dan tidak ganas dengan benar (4 tumor tidak ganas berada di kiri garis *threshold* dan 4 tumor ganas berada di kanan garis *threshold*). Namun, ketika terdapat outlier pada data, *Fit* terbaik dari *Linear Regression* akan menghasilkan prediksi yang buruk (Gambar 6).



**Gambar 7. *Linear Regression* gagal membentuk model yang baik**

Ketika data memiliki *outlier* seperti pada Gambar 6, garis *Fit* terbaik akan berubah mengikuti data juga. Dalam kasus diatas garis *Linear Regression* melandai dan menyebabkan pergeseran titik *threshold* pada sumbu x. Pergeseran *threshold* karena terdapat outlier menyebabkan 2 titik data yang seharusnya merupakan kanker ganas, jatuh ke bagian kiri garis *threshold* yang merupakan daerah untuk kanker tidak ganas.



**Gambar 8. *Logistic Regression* berhasil membentuk model yang baik**

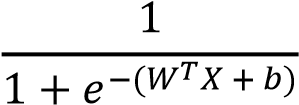
Hal ini tidak akan terjadi jika menggunakan *Logistic Regression*, seperti tersaji pada Gambar 7. Garis yang dibentuk oleh *Logistic Regression* tidak mengubah posisi *threshold* pada sumbu x sehingga tidak ada titik data yang mengalami salah klasifikasi.

*Logistic Function* yang membentuk garis hijau pada ilustrasi di atas dapat dibentuk dengan mensubstitusi *Linear Function* ke dalam *Sigmoid Function.* Dengan menggunakan *Logistic Function*, data yang dimiliki akan direpresentasikan ke dalam bentuk fungsi sigmoid. Karena menggunakan basis *Sigmoid Function*, output dari *Logistic Function* juga mendapat sifat dari *Sigmoid Function* yaitu untuk setiap input *X* maka output *y* akan selalu berada dalam rentang 0 sampai dengan 1. Sehingga label atau kelas pertama akan dianggap sebagai kelas 0 dan label atau kelas kedua akan dianggap sebagai kelas 1. Ketiga fungsi tersebut dituliskan dalam persamaan di bawah:

𝐿𝑖𝑛𝑒𝑎𝑟 𝐹𝑢𝑛𝑐𝑡𝑖𝑜𝑛 = 𝑠 = 𝑊𝑇𝑋 + 𝑏

1

𝑆𝑖𝑔𝑚𝑜𝑖𝑑 𝐹𝑢𝑛𝑐𝑡𝑖𝑜𝑛 = 1 + 𝑒−𝑎

𝐿𝑜𝑔𝑖𝑠𝑡𝑖𝑐 𝐹𝑢𝑛𝑐𝑡𝑖𝑜𝑛 = 𝑦𝑃𝑟𝑒𝑑𝑖𝑘𝑠𝑖 = 

Untuk menghasilkan *Logistic Function* yang baik, diperlukan nilai *W* dan *b* yang menyebabkan nilai *Loss* seminimum mungkin. *Loss* pada *Logistic Regression* dituliskan pada persamaan berikut:

𝐿𝑜𝑠𝑠 = −(𝑦𝐴𝑘𝑡𝑢𝑎𝑙 log(𝑦𝑃𝑟𝑒𝑑𝑖𝑘𝑠𝑖) + (1 − 𝑦𝐴𝑘𝑡𝑢𝑎𝑙)log (1 − 𝑦𝑃𝑟𝑒𝑑𝑖𝑘𝑠𝑖))

Algoritma *Gradient Descent* dapat digunakan untuk mencari nilai 𝑊 dan 𝑏 yang menyebabkan *Loss* minimum. Pertama, dilakukan pencarian nilai perubahan bobot, dengan menggunakan aturan turunan rantai sebagai berikut:

𝑑𝐿𝑜𝑠𝑠 𝑑𝐿𝑜𝑠𝑠 𝑑𝑦𝑃𝑟𝑒𝑑𝑖𝑘𝑠𝑖 𝑑𝑠

= ∗ ∗

𝑑𝑊 𝑑𝑦𝑃𝑟𝑒𝑑𝑖𝑘𝑠𝑖 𝑑𝑠 𝑑𝑊

Dimana untuk persamaan di atas:

𝑑𝐿𝑜𝑠𝑠 𝑦𝐴𝑘𝑡𝑢𝑎𝑙 1 − 𝑦𝐴𝑘𝑡𝑢𝑎𝑙

= (− + )

𝑑𝑦𝑃𝑟𝑒𝑑𝑖𝑘𝑠𝑖 𝑦𝑃𝑟𝑒𝑑𝑖𝑘𝑠𝑖 1 − 𝑦𝑃𝑟𝑒𝑑𝑖𝑘𝑠𝑖

𝑑𝑦𝑃𝑟𝑒𝑑𝑖𝑘𝑠𝑖

𝑑𝑠 = 𝑦𝑃𝑟𝑒𝑑𝑖𝑘𝑠𝑖 ∗ (1 − 𝑦𝑃𝑟𝑒𝑑𝑖𝑘𝑠𝑖)

𝑑𝑠

= 𝑋

𝑑𝑊

𝑑𝐿𝑜𝑠𝑠

Dari penjabaran di atas, aturan turunan rantai dari dapat disederhanakan menjadi:

𝑑𝑊

𝑑𝐿𝑜𝑠𝑠

𝑑𝑊 = 𝑋 ∗ (𝑦𝑃𝑟𝑒𝑑𝑖𝑘𝑠𝑖 − 𝑦𝐴𝑘𝑡𝑢𝑎𝑙)

𝑑𝐿𝑜𝑠𝑠

Setelah didapatkan nilai perubahan bobot , maka selanjutnya hitung nilai *W* baru

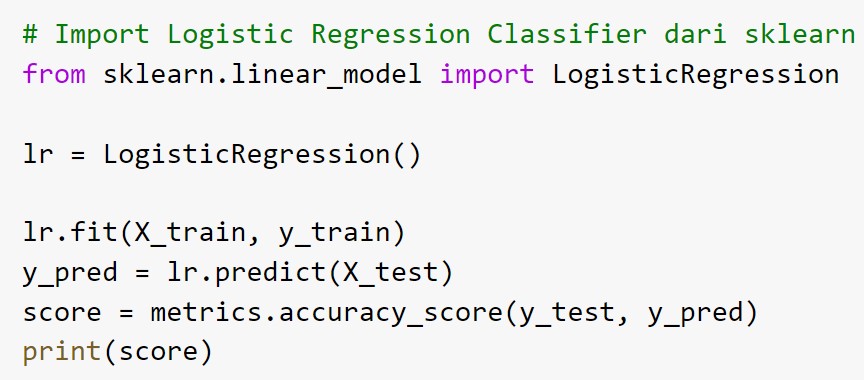
𝑑𝑊

dengan persamaan:

𝑑𝐿𝑜𝑠𝑠

𝑊𝑏𝑎𝑟𝑢 = 𝑊𝑙𝑎𝑚𝑎 − (𝐿𝑒𝑎𝑟𝑛𝑖𝑛𝑔𝑅𝑎𝑡𝑒 ∗ ) 𝑑𝑊

Pada bahasa pemrograman Python, *Logistic Regression* dapat dilakukan dengan menggunakan pustaka *Scikit-Learn* dengan menjalankan potongan kode berikut:



**Gambar 9. Kode untuk *Logistic Regression***

Perlu diingat bahwa algoritma *Logistic Regression* merupakan algoritma klasifikasi 2 kelas, sehingga ketika menggunakan data yang memiliki kelas lebih dari 2, dapat menggunakan skema pelatihan *One-versus-Rest* seperti yang secara otomatis diterapkan oleh pustaka *Scikit-Learn.*

## 2.2 Support Vector Machine

*Support Vector Machine* (SVM) adalah salah satu algoritma klasifikasi dalam bidang pembelajaran mesin. Ide utama dari SVM adalah untuk menemukan hyperplane terbaik yang memisahkan 2 daerah keputusan dengan baik. Secara umum, SVM bekerja dengan mengikuti langkah-langkah berikut:

1. fitur-fitur dari data dipetakan ke dalam ruang dimensi tinggi menggunakan fungsi kernel (*linear, polynomial, sigmoid,* atau *radial basis function*). Artinya, dengan menggunakan fungsi kernel, akan dihasilkan fitur-fitur baru yang akan digunakan untuk mengklasifikasi (tidak lagi menggunakan fitur-fitur lama). Masing-masing fungsi kernel dirumuskan sebagai berikut:
   1. Kernel *Linear*

𝐾(𝑥𝑖, 𝑥𝑗) = 𝑥𝑖𝑇𝑥𝑗

* 1. Kernel *Polynomial*

𝐾(𝑥𝑖, 𝑥𝑗) = (𝛾𝑥𝑖𝑇𝑥𝑗 + 𝑟)𝑑, 𝛾 > 0

* 1. Kernel *Sigmoid*

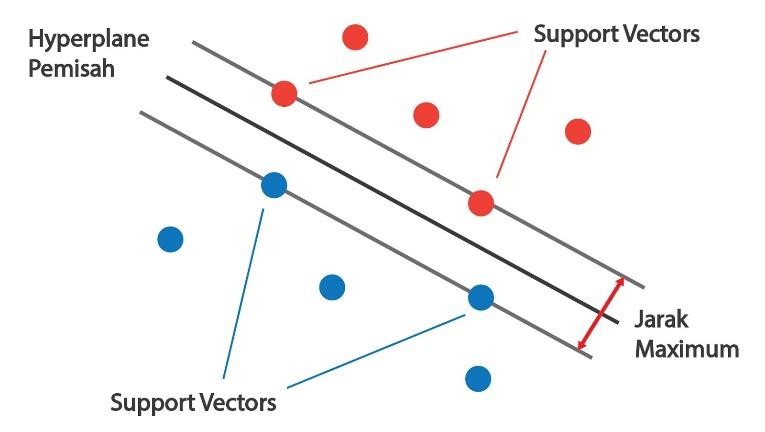
𝐾(𝑥𝑖, 𝑥𝑗) = tanh (𝑥𝑖𝑇𝑥𝑗 + 𝑟)

* 1. Kernel *Radial Basis Function* 2

𝐾(𝑥𝑖, 𝑥𝑗) = exp (−𝛾 ||𝑥𝑖𝑇 − 𝑥𝑗|| ) , 𝛾 > 0

1. Mencari *hyperplane* terbaik yang memisahkan data yang telah dipetakan dalam ruang dimensi tinggi. *Hyperplane* terbaik dapat diperoleh dengan memaksimalkan jarak hyperplane dengan titik data terdekat (*Support Vectors*).

6.



**Gambar 10. *Support Vector Machine***

Contoh penggunaan kernel, adalah sebagai berikut. Misalkan digunakan fungsi kernel *Radial Basis Function* dengan data 𝑥 = 𝑥1,𝑥2, . . . , 𝑥𝑛. Fitur-fitur hasil kernel RBF dimisalkan sebagai 𝑓 = 𝑓1 ,𝑓2,. . . 𝑓𝑚, dimana fitur-fitur tersebut digunakan untuk menentukan hasil klasifikasi sebagai berikut (𝑤adalah bobot, 𝑓 adalah fitur):

𝑖𝑓: 𝑤1𝑓1 + 𝑤2𝑓2 + ⋯ + 𝑤𝑚𝑓𝑚 > 0, 𝑡ℎ𝑒𝑛: 𝑐𝑙𝑎𝑠𝑠 = 1, 𝑒𝑙𝑠𝑒: 𝑐𝑙𝑎𝑠𝑠 = 0

2

Nilai 𝑓dapat dicari dengan 𝑓 = 𝐾(𝑥𝑖, 𝑥𝑗) = exp(−𝛾 ||𝑥𝑖𝑇 − 𝑥𝑗|| ). Untuk sementara, substitusikan 𝑥𝑗 dengan suatu titik random 𝑙, sehingga persamaan akan diubah sementara

2

menjadi 𝑓1 = 𝐾(𝑥𝑖, 𝑙) = exp(−𝛾 ||𝑥𝑖𝑇 − 𝑙|| ). Dengan menggunakan fungsi kernel RBF

2

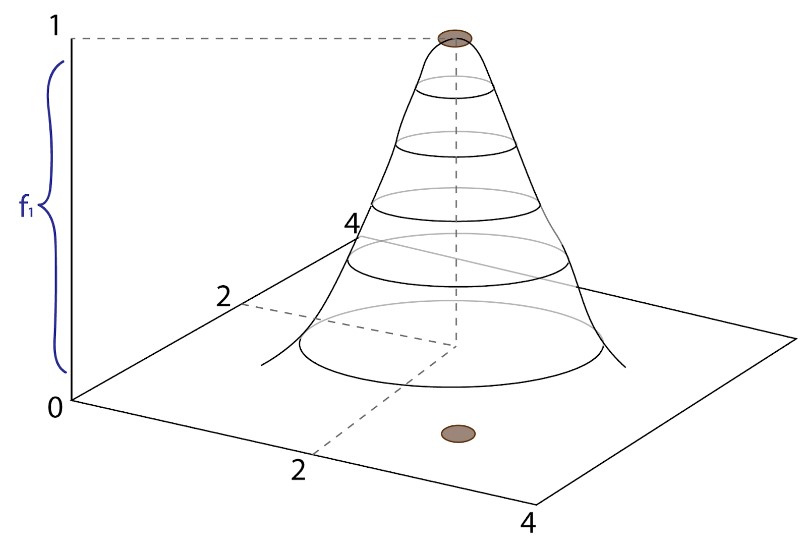
dan memahami bahwa ||𝑥𝑖𝑇 − 𝑙|| adalah jarak antara 𝑥𝑖 dengan titik 𝑙, maka dapat disimpulkan bahwa ketika suatu titik data 𝑥𝑖 berada sangat dekat dengan titik 𝑙, maka nilai 𝑓 akan mendekati 1, sebaliknya jika 𝑥𝑖 semakin jauh dengan titik 𝑙, maka nilai 𝑓 akan mendekati 0.

Sebagai contoh misalkan dimiliki data sebagai berikut: 𝑥1 = [2,2], 𝑙 = [2,2], dan 𝛾 = 20. Maka nilai dari 𝑓1 adalah 2

𝑓1 = 𝐾(𝑥1,𝑙) = exp(−𝛾 ||𝑥𝑖𝑇 − 𝑙|| )

𝑓1 = exp(0) = 1

Contoh diatas dapat divisualisasikan pada Gambar 10, dimana ketika titik berada semakin dekat dengan titik 𝑙 = [2,2] maka nilai 𝑓1 akan mendekati 1 (posisi pada bidang dimensi tingginya semakin memuncak), sebaliknya ketika ada titik 𝑥\_2 = [3,1] yang berada jauh dari titik titik 𝑙 = [2, 2], maka nilai 𝑓1 akan mendekati 0.



**Gambar 11. Nilai *f1* tiap titik ditentukan oleh jarak terhadap puncak**

Posisi titik *l* pada kenyataannya merupakan semua titik data 𝑥 = 𝑥1,𝑥2,… , 𝑥𝑛, oleh

2 karena itu persamaan di awal dituliskan 𝑓 = 𝐾(𝑥𝑖, 𝑥𝑗) = exp(−𝛾 ||𝑥𝑖𝑇 − 𝑥𝑗|| ). Dengan kata lain tiap-tiap 𝑥 akan difungsi kernelkan dengan 𝑥 lainnya dan menghasilkan fitur 𝑓 sebanyak 𝑚. 2 𝑓1 = 𝐾(𝑥1,𝑥1) = exp(−𝛾 ||𝑥1𝑇 − 𝑥1|| )

2

𝑓2 = 𝐾(𝑥1,𝑥2) = exp(−𝛾 ||𝑥1𝑇 − 𝑥2|| )

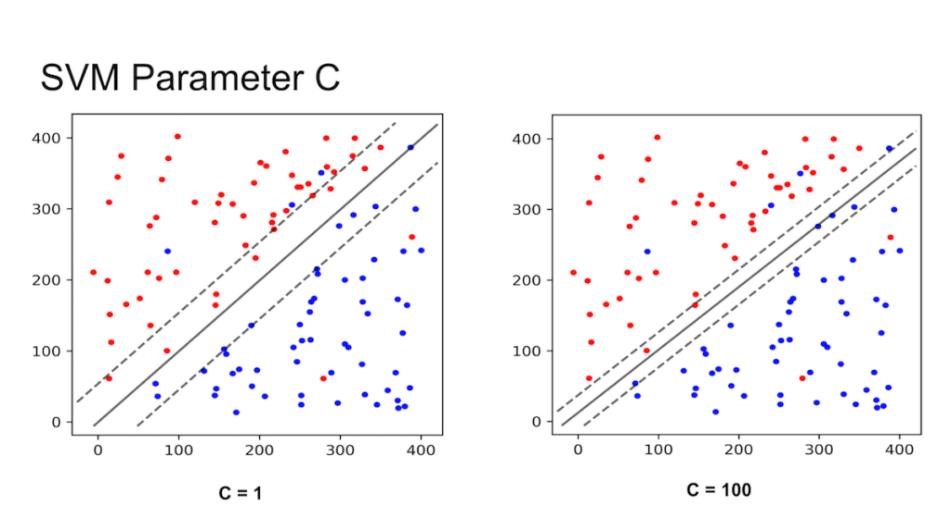
…

𝑓𝑚 = 𝐾(𝑥𝑛,𝑥𝑛) = exp(−𝛾||𝑥𝑛𝑇 − 𝑥𝑛||2)

Dari persamaan sebelumnya yaitu:

𝑖𝑓: 𝑤𝑚𝑓𝑚 > 0, 𝑡ℎ𝑒𝑛: 𝑐𝑙𝑎𝑠𝑠 = 1, 𝑒𝑙𝑠𝑒: 𝑐𝑙𝑎𝑠𝑠 = 0, selama berjalannya proses learning, akan banyak fitur 𝑓yang akan tereliminasi karena memperoleh nilai 𝑤 sebesar 0. Dari fitur 𝑓yang tersisa, dapat dilihat 𝑥 mana saja yang menjadi titik 𝑙 (sering muncul berarti memiliki jarak yang dekat dengan banyak titik).

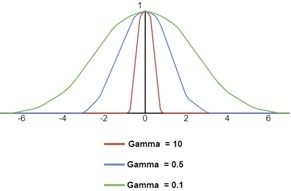
Terdapat parameter-parameter yang dapat diubah untuk menghasilkan hyperplane pemisah terbaik, diantaranya adalah parameter C dan parameter 𝛾. Parameter regularisasi C digunakan dalam SVM untuk menemukan jarak terbaik dari hyperplane. Nilai C yang terbaik harus dicari agar SVM terjadi keseimbangan antara jarak semaksimal dan kesalahan klasifikasi seminimal mungkin. Nilai C yang tinggi menyebabkan bahwa model akan menerima lebih banyak penalti ketika model gagal mengklasifikasikan data dengan tepat. Sebaliknya nilai C yang rendah menyebabkan bahwa model akan mentolerir data yang salah diklasifikasikan. Nilai C yang rendah biasanya baik digunakan pada data yang memiliki banyak *noise*, sebaliknya nilai C yang tinggi biasanya baik digunakan pada data yang memiliki sedikit *noise.*



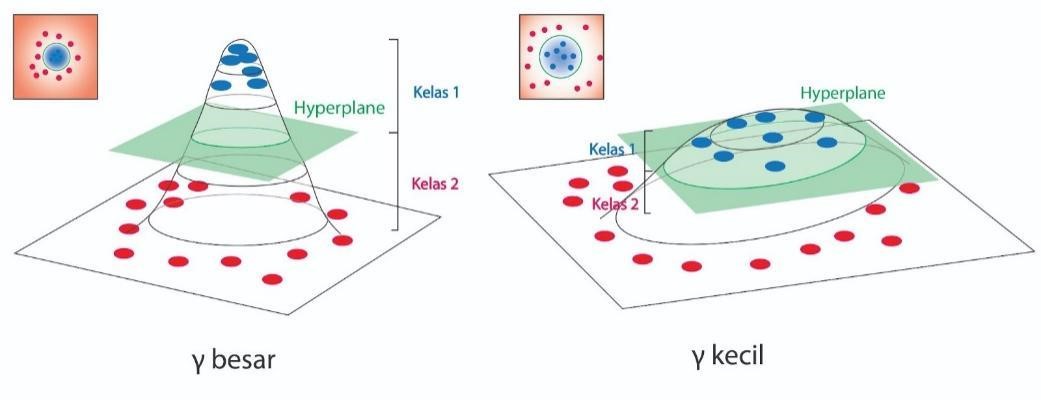
**Gambar 12. Pengaruh parameter C pada SVM**

*Sumber:* [*https://learnopencv.com/svm-using-scikit-learn-in-python/*](https://learnopencv.com/svm-using-scikit-learn-in-python/)

Selain parameter C, pada kernel *Radial Basis Function* terdapat parameter 𝛾. Nilai 𝛾 yang besar akan menghasilkan kelengkungan yang tajam pada ruang dimensi tinggi. Sebaliknya, nilai 𝛾 yang kecil akan menghasilkan ruang dimensi tinggi yang lebih landau (Gambar 12). Pasangan nilai parameter *C* dan 𝛾 harus ditemukan agar SVM dapat menggeneralisasi data dengan baik.

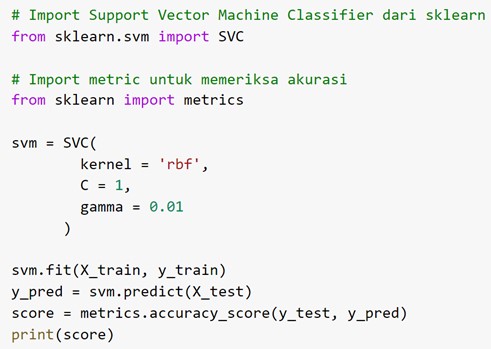


**Gambar 13. Visualisasi kelengkungan akibat parameter γ**



**Gambar 14. Pengaruh parameter γ pada SVM**

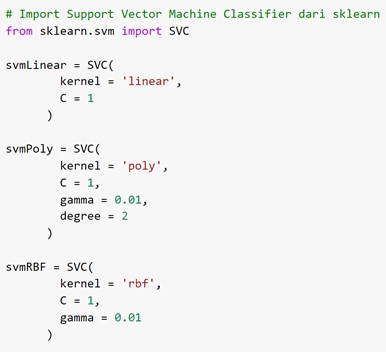
Pada bahasa pemrograman Python, *Support Vector Machine* dapat dilakukan dengan menggunakan pustaka *Scikit-Learn* dengan menjalankan potongan kode pada Gambar 14.



**Gambar 15. Potongan kode untuk SVM**

Beberapa parameter dapat diubah ketika menggunakan SVM dari pustaka *Scikit-Learn*, antara lain pada tabel berikut dan Gambar 15.

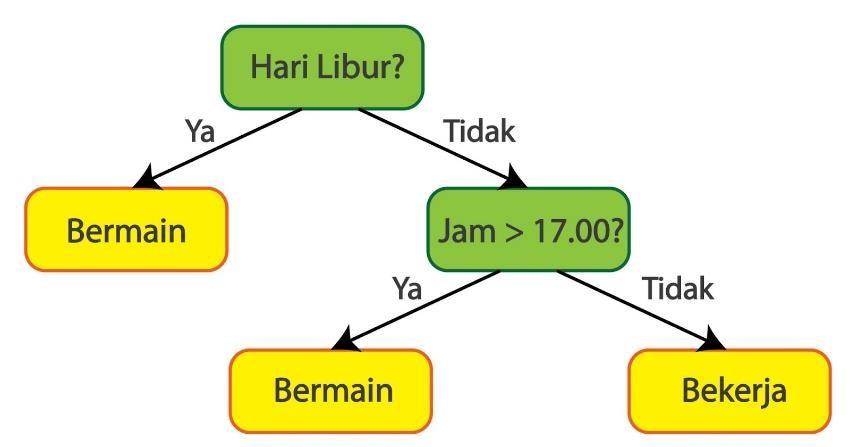
|  |  |  |
| --- | --- | --- |
| **Parameter** | **Keterangan** | **Contoh Nilai** |
| kernel | Memilih kernel untuk  SVM | -‘linear’ untuk linear kernel  -‘poly’ untuk polynomial kernel  -‘rbf’ untuk radial basis function kernel - Nilai default : rbf |
| C | Parameter regularisasi SVM untuk semua kernel | -Bilangan Float positif (…, 0.1, 0.5, 1.0, 1.5 , …)  -Nilai default = 1.0 |
| gamma | Parameter koefisien kernel untuk ‘poly’ dan ‘rbf’ | -Bilangan Float positif (…, 0.001, 0.01, 0.1, 1, …) |
| degree | Derajat polynomial untuk kernel ‘poly’) | * Bilangan Integer (2,3,4,….) * Nilai default : 3 |



**Gambar 16. Potongan kode untuk penggunaan parameter SVM**

## 2.3 Decision Tree

*Decision Tree* (Pohon Keputusan) merupakan salah satu algoritma pembelajaran mesin yang mengklasifikasi dengan mengambil suatu keputusan antara benar atau tidaknya suatu aturan. Kelebihan dari algoritma *Decision Tree* adalah input yang digunakan boleh berupa tipe data apapun (*String, Integer, Float, Boolean*, dll), diilustrasikan secara sederhana pada gambar di bawah:



**Gambar 17. *Decision Tree* sederhana**

Bagian paling puncak dari pohon keputusan (Hari Libur?) disebut dengan *root node.* Bagian selanjutnya dari pohon keputusan adalah *branch*, pada gambar di atas digambarkan oleh node (Jam > 17.00?). Terakhir *leaves* merupakan node yang berisi kelas dari permasalahan yang ingin diklasifikasikan.

Salah satu cara membangun *Decision Tree* adalah dengan menggunakan perhitungan *Gini Impurity*. *Gini Impurity* untuk *leaf* dari *Decision Tree* dapat dihitung menggunakan persamaan berikut:

2 2

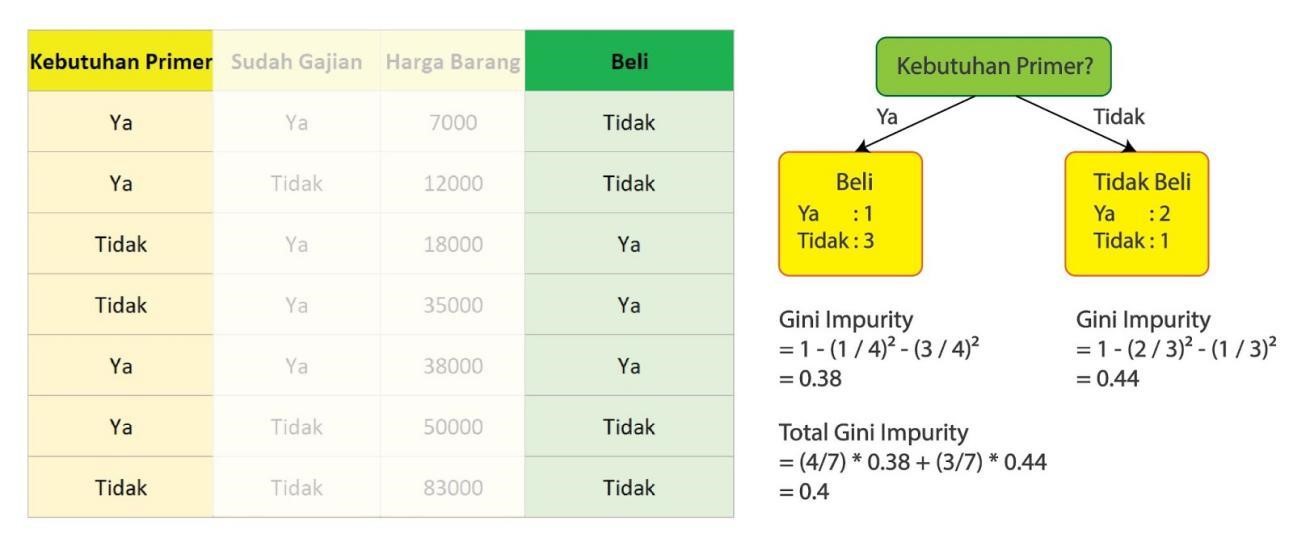
𝐺𝑖𝑛𝑖 𝐼𝑚𝑝𝑢𝑟𝑖𝑡𝑦 = 1 − (𝑃𝑟𝑜𝑏𝑎𝑏𝑖𝑙𝑖𝑡𝑎𝑠(𝑌𝑎)) − (𝑃𝑟𝑜𝑏𝑎𝑏𝑖𝑙𝑖𝑡𝑎𝑠(𝑇𝑖𝑑𝑎𝑘))

Misalkan dimiliki data seperti pada Gambar 17 (data modifikasi dari Youtube *StatQuest with Josh Starmer*).



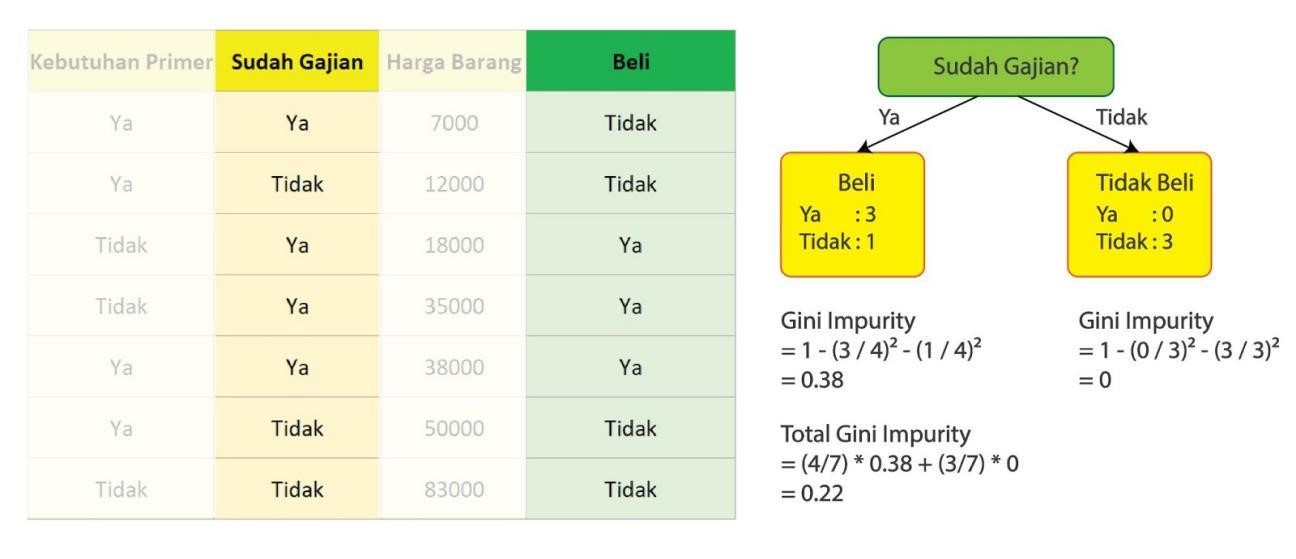
**Gambar 18. Data untuk *Decision Tree***

Untuk menentukan *root*, perlu dicari *Gini Impurity* terkecil dari “Kebutuhan Primer”, “Sudah Gajian”, dan “Harga Barang” terhadap “Beli”. *Gini Impurity* “Kebutuhan Primer” terhadap “Beli” dapat diilustrasikan pada Gambar 18.



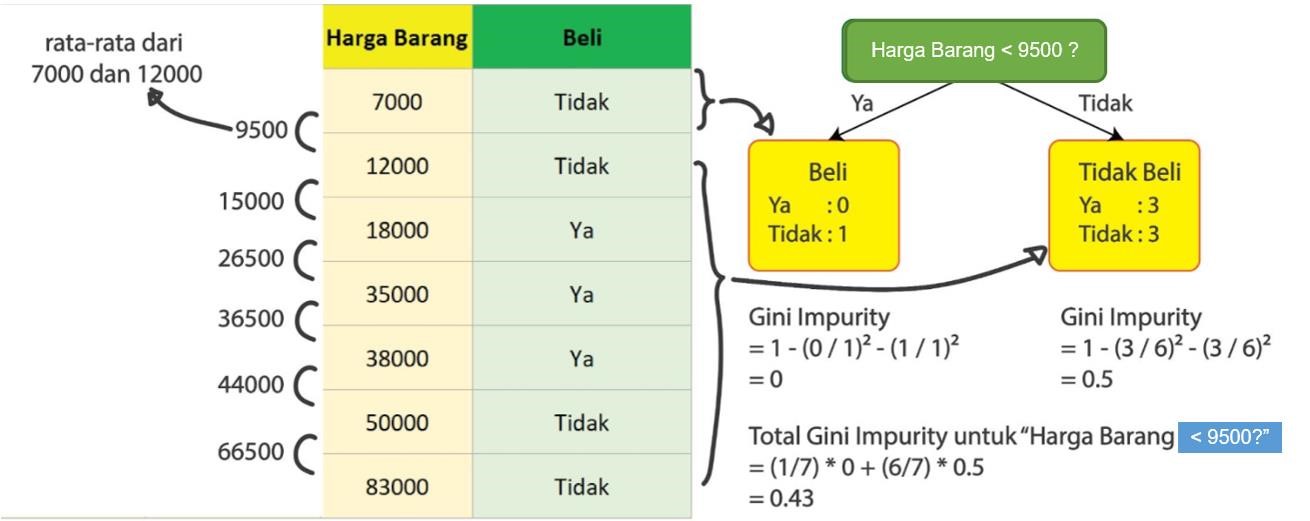
**Gambar 19. Pembentukan *Decision Tree* Langkah 1**

Total Gini Impurity dapat dihitung dengan *weighted average* dari Gini Impurity kedua *leaves.* Jumlah kemunculan “Ya” untuk kebutuhan primer sebanyak 4 dan jumlah kemunculan “Tidak” untuk kebutuhan primer sebanyak 3, maka *weighted average* Gini Impurity dapat dihitung dengan (terdapat di gambar): *Gini Impurity leaf* kiri \* 4 / (4+3) + *Gini Impurity leaf* kanan \* 3 / (4+3). Dengan cara yang sama, *Gini Impurity* “Sudah Gajian” terhadap “Beli” dihitung dan diilustrasikan pada Gambar 19.



**Gambar 20. Pembentukan *Decision Tree* Langkah 2**

Karena “Harga Barang” memiliki variabel kontinu, *Gini Impurity* “Harga Barang” terhadap “Beli” dihitung dengan cara yang sedikit berbeda. Pertama, data harus diurutkan berdasarkan “Harga Barang” terkecil ke “Harga Barang” terbesar. Kemudian untuk setiap data yang berdekatan, hitung *Gini Impurity* untuk rata-rata “Harga Barang” kedua data yang berdekatan. Pada Gambar 20, dihitung *Gini Impurity* untuk “Harga Barang” > 9500.



**Gambar 21. Pembentukan *Decision Tree* Langkah 3**

Dengan cara yang sama, hitung *Gini Impurity* untuk 15000, 26500, 36500, 44000, dan 66500. Setelah itu ambil *Gini Impurity* terkecil sebagai kandidat root. Jika ada lebih dari 1 yang sama, maka bebas menentukan yang mana yang ingin digunakan sebagai *root.*



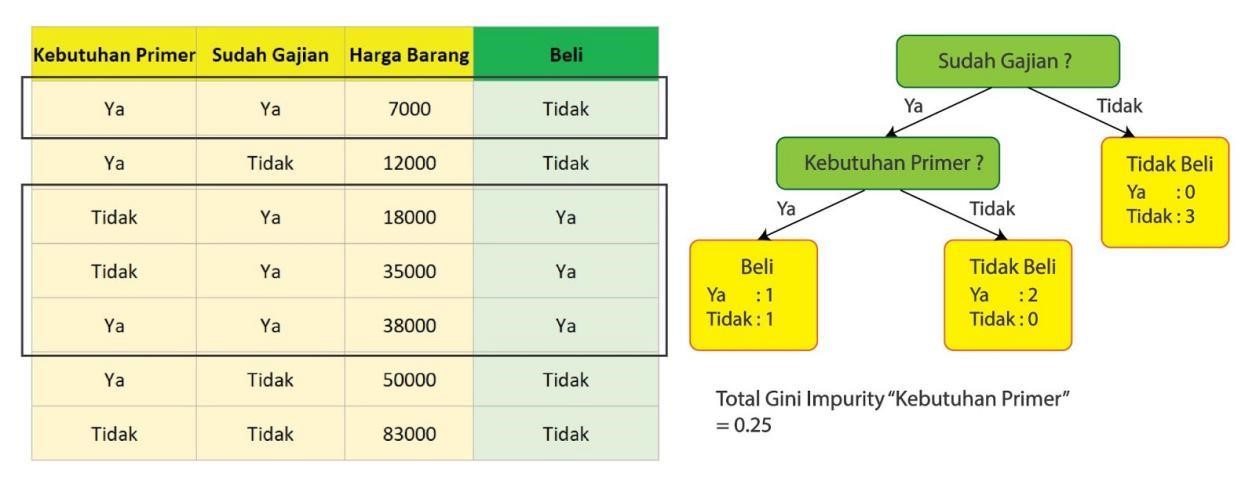
**Gambar 22. Pembentukan *Decision Tree* Langkah 4**

Setelah proses diatas, didapatkan *Gini Impurity* dari “Kebutuhan Primer”, “Sudah Gajian”, dan “Harga Barang” terhadap “Beli” secara berurutan 0.4, 0.22, dan 0.34. Karena *Gini Impurity* “Sudah Gajian” terhadap ”Beli” merupakan yang terkecil (0.22), maka “Sudah Gajian” dijadikan *root* dari *Decision Tree.* Maka *Decision Tree* sementara menjadi seperti pada Gambar 22.

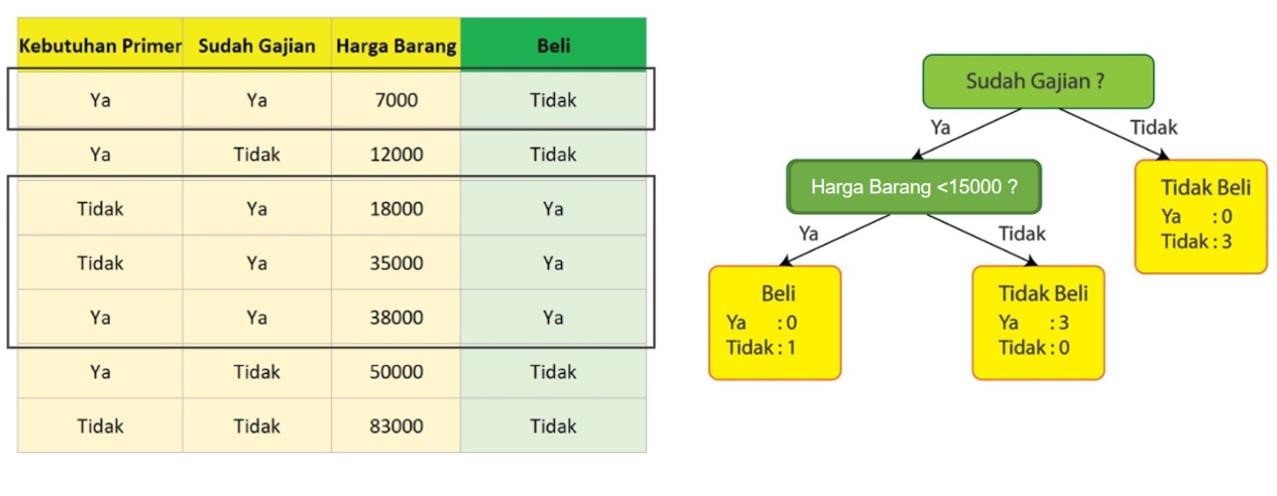


**Gambar 23. *Decision Tree* sementara**

Karena *Gini Impurity leaf* kanan adalah 0, maka tidak perlu di-*split* lagi. Oleh karena itu hanya *leaf* kiri yang perlu di-*split.* *Leaf* kiri dapat di-*split* dengan mencari *Gini Impurity* terkecil dari “Kebutuhan Primer”atau “Harga Barang”. Cara yang dilakukan sama, namun data yang digunakan lebih sedikit karena telah tersaring oleh *Decision root.* Contoh untuk “Kebutuhan Primer” dan “Harga Barang” tersaji pada Gambar 23 dan 24.

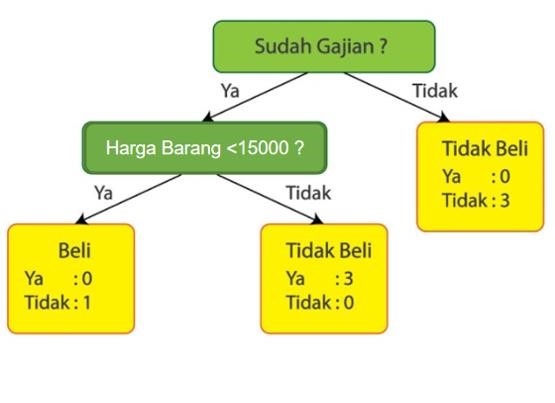


**Gambar 24. Pembentukan *Decision Tree* Langkah 5**



**Gambar 25. Pembentukan *Decision Tree* Langkah 6**

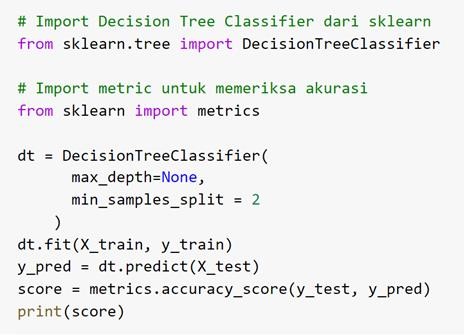
Ingat untuk data kontinu, perlu dihitung *Gini Impurity* dari data berdekatan. Karena *Gini Impurity* dari “Harga Barang > 15000” lebih kecil dibandingkan dengan *Gini Impurity* “Kebutuhan Primer”, maka “Harga Barang > 15000” dipilih sebagai *branch.* Tidak diperlukan proses *split* lagi karena *Gini Impurity* dari seluruh *leaves* sudah mencapai 0. Sehingga *Decision Tree* yang dihasilkan adalah seperti pada Gambar 25.



**Gambar 26. Hasil akhir *Decision Tree***

Untuk mencegah agar *Decision Tree* tidak mengalami *overfitting*, nilai maximum kedalaman tree dapat ditentukan. Selain itu nilai minimum sampel data yang diperlukan agar *node* mengalami *split* juga dapat ditentukan agar *Decision Tree* tidak mengalami *overfitting.*

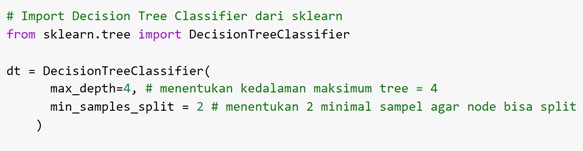
Pada bahasa pemrograman Python, *Decision Tree* dapat dilakukan dengan menggunakan pustaka *Scikit-Learn* dengan menjalankan potongan kode pada Gambar 26.



**Gambar 27. Potongan kode untuk *Decision Tree***

Beberapa parameter dapat diubah nilainya, antara lain adalah parameter *max\_depth* dan *min\_samples\_split*, dengan penjelasan pada tabel berikut dan Gambar 27.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Keterangan** | **Contoh Nilai** |
| max\_depth | Maksimum kedalaman tree yang dibentuk | * Bilangan Integer (1,2,3,4,….) * Nilai default : None (jika None maka tree akan terus mendalam sampai seluruh Gini Impurity = 0) |
| min\_samples\_split | Minimum sampel yang  diperlukan agar node  dapat ‘split’ | * Bilangan Integer (1,2,3,4,….) * Nilai default : 2 |



**Gambar 28. Potongan kode untuk penggunaan parameter *Decision Tree***

## 2.4 Naïve Bayes Classifier

Naïve Bayes Classifier adalah model probabilistik untuk klasifikasi. Terdapat dua model probabilistik, yaitu :

1. Discriminative Model



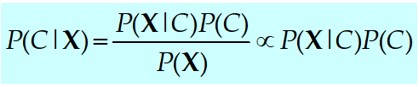
1. Generative Model



Untuk menentukan kelas, digunakan MAP (Maximum A Posterior) Classification

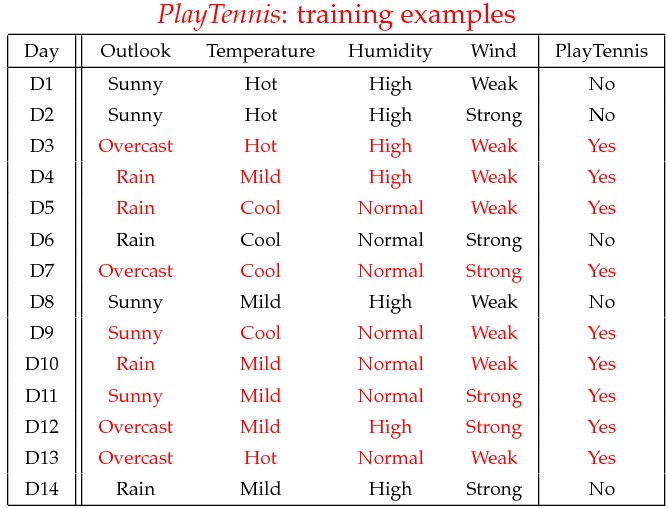
Rule. Assign 

Naïve Bayes Classifier menerapkan Bayesian Rule

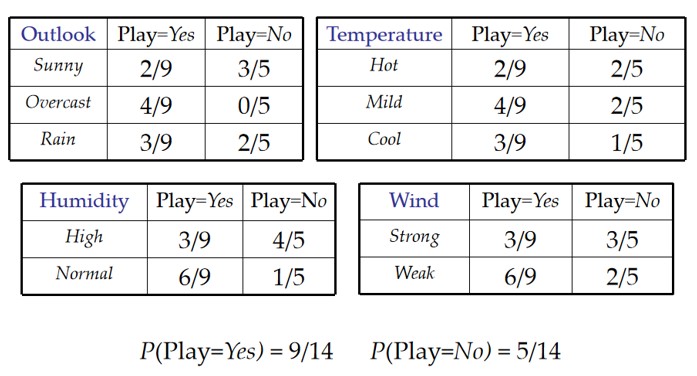


Contoh Kasus :

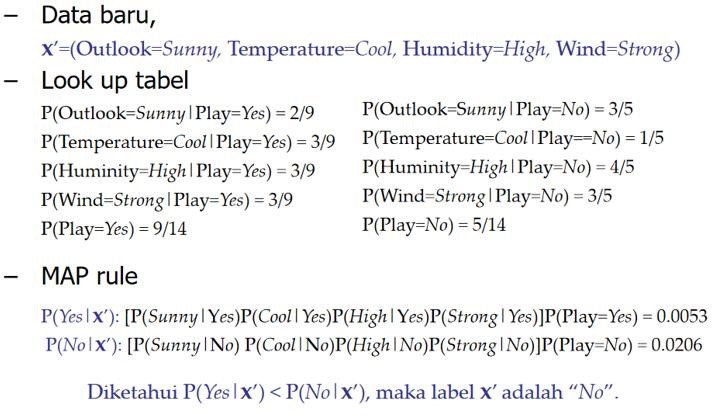
Terdapat 14 data latih. Data memiliki 4 atribut (Outlook, Temperature, Humidity dan Wind). Data latih terdiri dari dua kelas (kelas YES dan kelas NO untuk bermain tenis). Diperlukan klasifikasi untuk menentukan kelas dari data uji, apakah bermain tenis (YES) atau tidak (NO).



Berdasarkan data latih tersebut, dilakukan perhitungan sebagai berikut :



Untuk menentukan kelas pada data uji, menggunakan MAP Classification Rule, sebagai berikut :

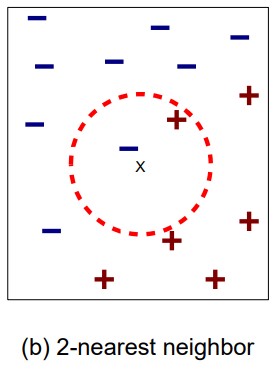
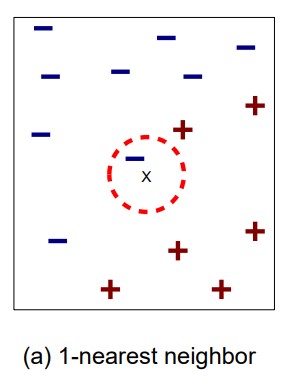


Naïve Bayes Classifier memiliki karakter sebagai berikut :

* Naïve Bayes Classifier bekerja lebih baik pada training data yang kecil
* Proses pembangunan Naïve Bayes Classifier pada data numerik menjadi lebih rumit dan memungkinkan terdapat informasi yang hilang
* Pada Naïve Bayes Classifier, diasumsikan bahwa satu fitur dengan fitur yang lain saling independen, hal ini mungkin tidak selalu terjadi pada kasus nyata

## 2.5 k-Nearest Neighbor

Metode k-Nearest Neighbor Classifier menentukan kelas/label dari suatu data uji berdasarkan label dari data – data latih sekitarnya. Ilustrasi ditunjukkan pada Gambar 28.



**Gambar 29. Ilustrasi k-nearest neighbor**

Algoritma untuk k-nearest neighbor sebagai beirkut :

1. Menentukan nilai k
2. Menghitung jarak antara data baru terhadap semua training data
3. Mengidentifikasi k nearest neighbor
4. Menentukan label/kelas data baru berdasarkan kelas k-nearest neighbor (dapat menggunakan voting)

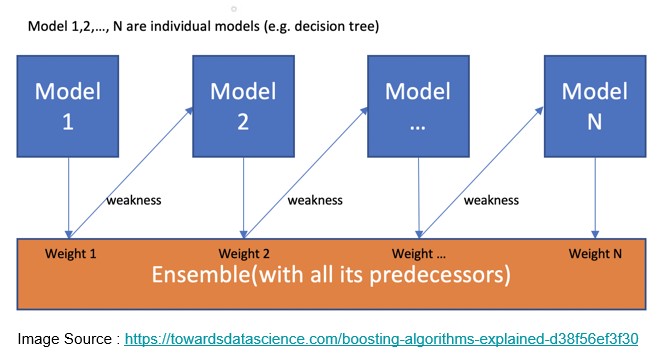
Karakter dari algoritma k-nearest neighbor dapat dituliskan sebagai beirkut :

1. Cocok untuk data numerik
2. Mudah dipahami dan diimplementasikan
3. k-NN merupakan *lazy learner* (tidak membangun model secara eksplisit)
4. Penentuan label/kelas data baru membutuhkan computational cost yang cukup tinggi
5. Perlu menentukan nilai k yang sesuai
   1. Jika k terlalu kecil, sensitif terhadap noise
   2. Jika k terlalu besar, nearest neigbor mungkin mencakup data dari kelas lain

## 2.6 Boosting Algorithm

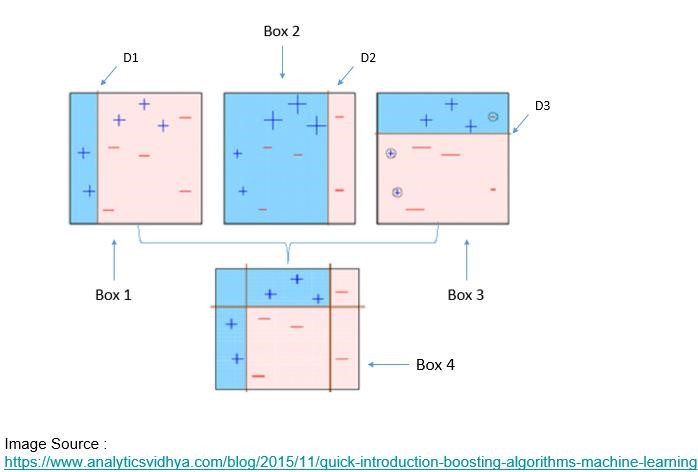
Boosting Algorithm merupakan salah satu strategi dari ensemble untuk mengubah suatu *weak model* menjadi *strong model*. Konsep dari Boosting Algorithm adalah mengkombinasikan hasil prediksi dari setiap weak model untuk menjadi strong model dengan metode Average/Weighted Average dan Higher voting.

Terdapat beberapa tipe Boosting Algorithm, yaitu : (1) AdaBoost (Adaptive Boosting) ; (2) Gradient Tree Boosting dan (3) XGBoost. Ilustrasi cara kerja Boosting Algorithm ditunjukkan pada Gambar 39.



**Gambar 309. Ilustrasi Boosting Algorithm**

Adaboost (Adaptive Boosting) Algorithm memiliki prinsip kerja dengan mengidentifikasi miss-classified data. Setelah teridentifikasi, kemudian missclassified data tersebut, diberikan bobot lebih, sehingga classifier berikutnya memberikan perhatian lebih terhadap hal tersebut. Ilustrasi cara kerja AdaBoost Algorithm ditunjukkan pada Gambar 30.



**Gambar 30. Ilustrasi Boosting Algorithm**