Modul 3 Klasifikasi 1: *Decision Tree, Random Forest* dan *Naïve Bayes*

3.1 Tujuan

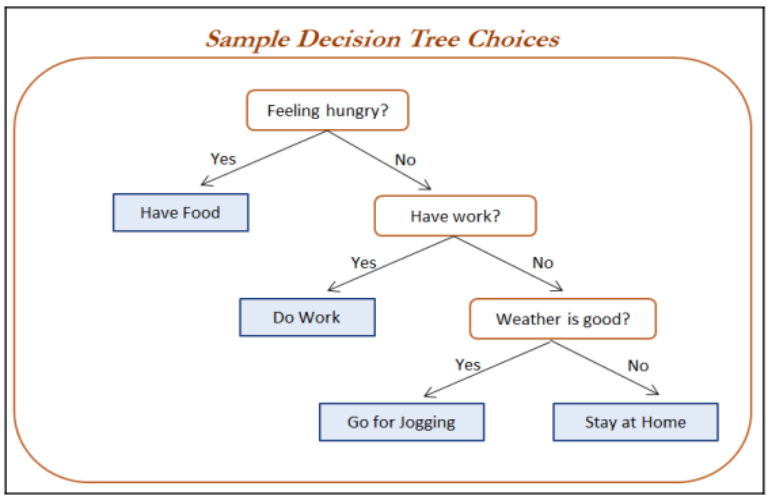
Mahasiswa dapat mempraktikkan metode pembelajaran mesin *Decision Tree, Random Forest,* dan *Naïve Bayes* menggunakan Python dengan benar.

3.2 Dasar Teori

3.2.1 *Decision Tree*

Decision Trees (DTs) adalah sebuah algoritma yang membantu dalam pengambilan keputusan berdasarkan pada pengalaman sebelumnya. *Decision Tree* termasuk dalam metode pembelajaran *supervised* non-parametrik yang digunakan untuk klasifikasi dan regresi. Tujuannya adalah untuk membuat model yang dapat memprediksi nilai variabel target dengan mempelajari aturan keputusan sederhana yang disimpulkan dari fitur data. Pendekatan *decision tree* berupa pendekatan konstan sebagaimana sebuah pohon.

Algoritma ini banyak dipilih karena *decision tree* meniru cara berpikir manusia saat membuat keputusan, sehingga mudah dimengerti. Selain itu, logika dibalik *decision tree* dapat dengan mudah dipahami karena menunjukkan struktur seperti pohon (Gambar 3.1). Tetapi algoritma ini juga memiliki beberapa kekurangan diantaranya sifat tidak stabil, ini menjadi salah satu keterbatasan dari algoritma *decision tree* ketika terdapat perubahan kecil pada data dapat menghasilkan perubahan besar dalam struktur pohon keputusan, serta kurang efektif dalam memprediksi hasil dari variabel kontinu.



Gambar 3.1 Ilustrasi *decision tree* (Dangeti, 2017)

Algoritma *decision tree* menggunakan *information gain* untuk membagi sebuah node. Indeks Gini dan *entropy* adalah kriteria untuk menghitung *information gain*. Entropi berasal dari teori informasi dan merupakan ukuran ketidakmurnian dalam data. Jika sampel benar-benar homogen maka entropinya nol, dan jika sampel terbagi rata maka entropinya satu. Dalam pohon keputusan, prediktor dengan heterogenitas paling besar akan dianggap paling dekat dengan simpul akar untuk mengklasifikasikan data yang diberikan ke dalam kelas-kelas. Entropi direpresentasikan dalam persamaan berikut.

$$Entropi = -p_1 p_1 - \cdots - p_n p_n \quad (3.1)$$

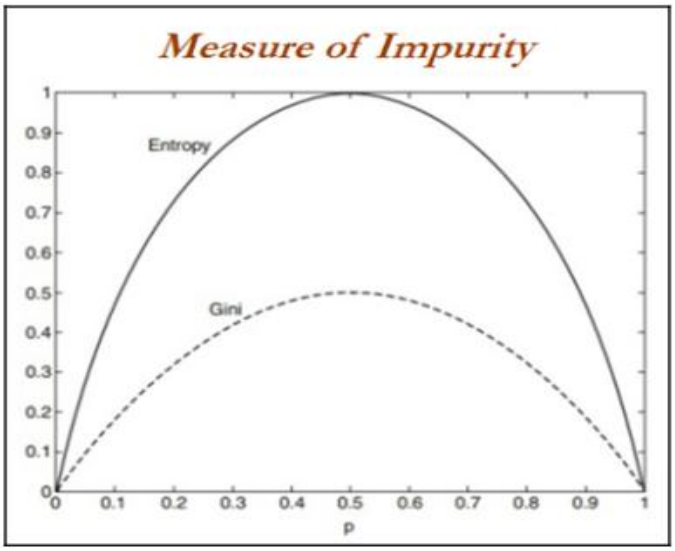
$$Entropi = -\sum_{i=0}^{n} p_i p_i \quad (3.2)$$

Dimana n = jumlah kelas dan p = probabilitas ter klasifikasinya nilai dalam sebuah kelas. Entropi maksimum berada di tengah dengan nilai 1 ($p = 0.5$) dan minimum di ekstrem dengan nilai 0 ($p = 0$ dan $p = 1$). Nilai yang diinginkan dari entropi adalah yang rendah, karena akan memisahkan kelas dengan lebih baik.

Gini index adalah ukuran kesalahan klasifikasi yang berlaku dalam konteks pengklasifikasi multikelas. Cara kerja indeks Gini mirip dengan entropi, hanya saja indeks Gini lebih cepat dihitung. Metode ini digunakan untuk mengukur derajat ketidakmerataan distribusi sampel yang menggunakan persamaan sebagai berikut:

$$Gini\ Index = 1 - \sum_{i} p_i^2 \quad (3.3)$$

Dimana i = jumlah kelas dan p = probabilitas ter klasifikasinya nilai dalam sebuah kelas (Dangeti, 2017). Perbandingan nilai entropi dan indeks Gini untuk mengukur *impurity* tersaji pada Gambar 3.2.



Gambar 3.2 Pengukuran *impurity* menggunakan entropi dan indeks Gini (Dangeti, 2017)

Information gain (IG) adalah metrik yang dapat menunjukkan peningkatan ketika pembuatan pemisah yang berbeda dan biasanya digunakan bersama dengan entropi (dapat juga digunakan dengan indeks Gini, meskipun dalam hal ini tidak disebut sebagai *information gain*). Dengan cara ini dapat dibandingkan apakah pemisahan dengan lebih sedikit *impurity* dapat dihasilkan. Perhitungan *Information gain* akan bergantung pada apakah *decision tree* yang dirancang digunakan untuk menyelesaikan kasus klasifikasi atau regresi dengan persamaan sebagai berikut.

$$IG_{classification} = E(d) - \sum \frac{|s|}{|d|} E(s) \quad (3.4)$$

$$IG_{regresion} = Var(d) - \sum \frac{|s|}{|d|} Var(s) \quad (3.5)$$

Dimana E adalah entropi, s adalah cabang, d adalah batang, dan Var adalah variansi.

3.2.1.1 Contoh Penerapan Algoritma

Secara garis besar, langkah yang dibutuhkan untuk menyusun algoritma *decision tree* adalah sebagai berikut.

1. Hitung *information gain* untuk semua variabel.
2. Pilih pemisahan yang menghasilkan *information gain* tertinggi sebagai pemisahan.
3. Ulangi proses ini hingga setidaknya salah satu kondisi yang ditetapkan oleh *hyperparameter* algoritma terpenuhi.

Untuk membuat sebuah *decision tree* beberapa hal yang perlu disiapkan antara lain sebagai berikut.

1. Seluruh data harus numerik, apabila terdapat data yang tidak numerik maka perlu dilakukan *mapping* ke dalam format numerik.
2. Parameter input dan target telah ditentukan di awal.

Pada kasus yang diangkat kali ini, akan diprediksi apakah seseorang mengalami obesitas atau tidak berdasarkan data tinggi dan berat badannya. Pada deskripsi dataset (https://www.kaggle.com/datasets/yersever/500-person-gender-height-weight-bodymassindex), seseorang dengan indeks 4 atau 5 mengalami obesitas, sehingga dapat dibuat variabel yang mencerminkan hal ini:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

data = pd.read_csv('BMIdata.csv')
data.head()
```

Maka akan diperoleh hasil output sebagai berikut.

	Gender	Height	Weight	Index
0	Male	174	96	4
1	Male	189	87	2
2	Female	185	110	4
3	Female	195	104	3
4	Male	149	61	3

Selanjutnya berdasarkan kriteria, diidentifikasi kategori obesitas dengan kode sebagai berikut.

```
data['obese'] = (data.Index >= 4).astype('int')
data.drop('Index', axis = 1, inplace = True)
```

Hanya saja tidak seluruhnya yang memiliki berat >100 kg adalah obesitas. Ada juga seseorang yang beratnya lebih dari 100 kg tetapi bukan tergolong obesitas. Sehingga perlu dibuat aturan (*rule*) dari *decision tree* agar mampu menempatkan dalam kelas-kelas yang tepat, sehingga prediksi menjadi semakin baik. Berikut contoh kodenya.

```
print(
  " Misclassified when cutting at 100kg:",
  data.loc[(data['Weight']>=100) & (data['obese']==0),:].shape[0], "\n",
  "Misclassified when cutting at 80kg:",
  data.loc[(data['Weight']>=80) & (data['obese']==0),:].shape[0]
)
```

Berikut hasil keluarannya.

```
Misclassified when cutting at 100kg: 18
Misclassified when cutting at 80kg: 63
```

Cara Menghitung Entropi, Indeks Gini, dan *Information Gain* (IG)

1. Entropi

Berikut adalah cara menentukan nilai entropi dari kasus di atas.

```
def entropy(y):
  '''
  Given a Pandas Series, it calculates the entropy.
  y: variable with which calculate entropy.
  '''
  if isinstance(y, pd.Series):
    a = y.value_counts()/y.shape[0]
    entropy = np.sum(-a*np.log2(a+1e-9))
    return(entropy)

  else:
    raise('Object must be a Pandas Series.')
```

```
entropy(data.Gender)
```

2. Indeks Gini

```
def gini_impurity(y):
  '''
  Given a Pandas Series, it calculates the Gini Impurity.
  y: variable with which calculate Gini Impurity.
  '''
  if isinstance(y, pd.Series):
    p = y.value_counts()/y.shape[0]
    gini = 1-np.sum(p**2)
    return(gini)

  else:
    raise('Object must be a Pandas Series.')

gini_impurity(data.Gender)
```

3. *Information Gain* (IG)

```
def variance(y):
  '''
  Function to help calculate the variance avoiding nan.
  y: variable to calculate variance to. It should be a Pandas Series.
  '''
  if(len(y) == 1):
    return 0
  else:
    return y.var()

def information_gain(y, mask, func=entropy):
  '''
  It returns the Information Gain of a variable given a loss function.
  y: target variable.
  mask: split choice.
   func: function to be used to calculate Information Gain in case os
classification.
  '''

  a = sum(mask)
  b = mask.shape[0] - a

  if(a == 0 or b ==0):
    ig = 0

  else:
    if y.dtypes != 'O':
       ig = variance(y) - (a/(a+b)* variance(y[mask])) - (b/(a+b)*variance(y[-
mask]))
    else:
      ig = func(y)-a/(a+b)*func(y[mask])-b/(a+b)*func(y[-mask])

  return ig
```

Cara Menentukan Pemisahan Terbaik dari sebuah Variabel

Untuk menghitung pemisahan terbaik suatu variabel numerik, pertama-tama, semua kemungkinan nilai yang diambil dari variabel tersebut harus diperoleh (p). Setelah memiliki opsi, hitung IG untuk setiap opsi menggunakan filter jika nilainya kurang dari nilai tersebut. Yang jelas, p untuk data pertama akan dihilangkan, karena pemisahan akan mencakup semua nilai.

Jika terdapat variabel kategori, idenya sama, hanya saja dalam kasus ini IG harus dihitung untuk semua kemungkinan kombinasi variabel yang ada, tidak termasuk opsi yang mencakup semua opsi (karena tidak akan menghasilkan pemisahan apa pun). Hal ini cukup memakan biaya komputasi jika jumlah kategori yang dimiliki banyak, sehingga algoritma *decision tree* biasanya hanya menerima variabel kategori sampai dengan <20 kategori.

Jadi, setelah didapatkan semua pemisahan, hasil pemisahan yang digunakan adalah pemisahan yang menghasilkan IG tertinggi. Berikut kode yang dapat digunakan untuk mendapatkan pemisahan terbaik untuk *decision tree*.

```
import itertools

def categorical_options(a):
  '''
  Creates all possible combinations from a Pandas Series.
  a: Pandas Series from where to get all possible combinations.
  '''
  a = a.unique()

  opt = []
  for L in range(0, len(a)+1):
      for subset in itertools.combinations(a, L):
          subset = list(subset)
          opt.append(subset)

  return opt[1:-1]

def max_information_gain_split(x, y, func=entropy):
  '''
  Given a predictor & target variable, returns the best split, the error and the
type of variable based on a selected cost function.
  x: predictor variable as Pandas Series.
  y: target variable as Pandas Series.
  func: function to be used to calculate the best split.
  '''

  split_value = []
  ig = []

  numeric_variable = True if x.dtypes != 'O' else False
```

```
  # Create options according to variable type
  if numeric_variable:
    options = x.sort_values().unique()[1:]
  else:
    options = categorical_options(x)

  # Calculate ig for all values
  for val in options:
    mask =   x < val if numeric_variable else x.isin(val)
    val_ig = information_gain(y, mask, func)
    # Append results
    ig.append(val_ig)
    split_value.append(val)

  # Check if there are more than 1 results if not, return False
  if len(ig) == 0:
    return(None,None,None, False)

  else:
  # Get results with highest IG
    best_ig = max(ig)
    best_ig_index = ig.index(best_ig)
    best_split = split_value[best_ig_index]
    return(best_ig,best_split,numeric_variable, True)

weight_ig,  weight_split,  _,  _  =  max_information_gain_split(data['Weight'],
data['obese'],)

print(
  "The best split for Weight is when the variable is less than ",
  weight_split,"\nInformation Gain for that split is:", weight_ig
)
```

Berikut hasil outputnya.

```
The best split for Weight is when the variable is less than  103
Information Gain for that split is: 0.10625190497954967
```

Cara Memilih Pemisahan Terbaik

Seperti yang telah dijelaskan sebelumnya, pemisahan terbaik adalah pemisahan yang menghasilkan IG tertinggi. Untuk mengetahui yang mana, cukup menghitung IG untuk setiap variabel prediktor model seperti contoh berikut.

```
data.drop('obese', axis= 1).apply(max_information_gain_split, y = data['obese'])
```

Hasilnya adalah sebagai berikut.

	Gender	Height	Weight
0	-0.000281	0.019684	0.106252
1	[Male]	174	103
2	False	True	True
3	True	True	True

Seperti yang bisa dilihat, variabel dengan IG tertinggi adalah *Weight*. Oleh karena itu, variabel tersebut akan menjadi **variabel yang pertama kali digunakan untuk melakukan pemisahan** dengan nilai 103.

Dengan ini, pemisahan pertama sudah dimiliki yang akan menghasilkan dua kerangka data. Jika hal ini diterapkan secara rekursif, maka seluruh pohon keputusan dapat dibuat (dikodekan dengan Python dari awal).

Cara Melatih *Decision Tree* dengan Python dari Awal

1. Menentukan Kedalaman Pohon

Pertama yang dilakukan adalah menentukan *hyperparameter* untuk *Decision Tree* yang relevan. Jumlah *hyperparameter* dapat dibuat lebih banyak lagi, hanya saja lebih baik memilih *hyperparameter* yang sesuai agar mencegah pohon tumbuh terlalu banyak, sehingga menghindari *overfitting*. *Hyperparameter* tersebut adalah sebagai berikut:

1. max_kedalaman: kedalaman maksimum pohon. Jika di set ke None, pohon akan tumbuh hingga semua daunnya murni atau *hyperparameter* min_samples_split telah tercapai.
2. min_samples_split: menunjukkan jumlah minimum observasi yang harus dimiliki suatu *sheet* untuk terus membuat *node* baru.
3. min_information_gain: jumlah minimum yang harus ditingkatkan oleh *Information Gain* agar pohon dapat terus tumbuh.

Selanjutnya menerapkan algoritma *decision tree* sebagai berikut:

1. Pastikan ketentuan yang ditetapkan oleh min_samples_split dan max_ depth terpenuhi.
2. Buat pemisahan.
3. Pastikan min_information_gain terpenuhi.
4. Simpan data pemisahan dan ulangi prosesnya.

Untuk melakukan hal ini, akan dibuat tiga fungsi:

1. fungsi yang, dengan mempertimbangkan beberapa data, mengembalikan pemisahan terbaik dengan informasi terkaitnya;
2. fungsi lain yang, dengan mempertimbangkan beberapa data dan pemisahan, membuat pemisahan dan mengembalikan prediksi;
3. fungsi yang memberikan beberapa data, membuat prediksi.

Catatan: prediksi hanya akan diberikan di cabang dan pada dasarnya terdiri dari pengembalian rata-rata data dalam kasus regresi atau mode dalam kasus klasifikasi. Berikut contoh pengkodean fungsi yang dimaksud sebelumnya.

```
def get_best_split(y, data):
  '''
  Given a data, select the best split and return the variable, the value, the
variable type and the information gain.
  y: name of the target variable
  data: dataframe where to find the best split.
  '''
  masks = data.drop(y, axis= 1).apply(max_information_gain_split, y = data[y])
  if sum(masks.loc[3,:]) == 0:
    return(None, None, None, None)

  else:
    # Get only masks that can be splitted
    masks = masks.loc[:,masks.loc[3,:]]

    # Get the results for split with highest IG
    split_variable = masks.iloc[0].astype(np.float32).idxmax()
    #split_valid = masks[split_variable][]
    split_value = masks[split_variable][1]
    split_ig = masks[split_variable][0]
    split_numeric = masks[split_variable][2]

    return(split_variable, split_value, split_ig, split_numeric)


def make_split(variable, value, data, is_numeric):
  '''
  Given a data and a split conditions, do the split.
  variable: variable with which make the split.
  value: value of the variable to make the split.
  data: data to be splitted.
  is_numeric: boolean considering if the variable to be splitted is numeric or
not.
  '''
  if is_numeric:
    data_1 = data[data[variable] < value]
    data_2 = data[(data[variable] < value) == False]

  else:
    data_1 = data[data[variable].isin(value)]
    data_2 = data[(data[variable].isin(value)) == False]
```

```
  return(data_1,data_2)

def make_prediction(data, target_factor):
  '''
  Given the target variable, make a prediction.
  data: pandas series for target variable
  target_factor: boolean considering if the variable is a factor or not
  '''

  # Make predictions
  if target_factor:
    pred = data.value_counts().idxmax()
  else:
    pred = data.mean()

  return pred
```

2. Melatih Decision Tree dengan Python

Selanjutnya, untuk melatih *Decision Tree*, berikut algoritma yang dapat diterapkan:

1. Pastikan bahwa min_samples_split dan max_ depth terpenuhi.
2. Jika terpenuhi maka dapatkan pemisahan terbaik dan dapatkan nilai IG. Jika salah satu kondisi tidak terpenuhi maka buatlah prediksi.
3. Cek apakah Komprobamos IG melewati jumlah minimum yang ditetapkan oleh min_information_gain.
4. Jika syarat di atas terpenuhi maka lakukan pemisahan dan penyimpaan keputusan. Jika tidak terpenuhi barulah buat prediksi.

Lakukan proses ini secara rekursif, yaitu fungsi akan memanggil dirinya sendiri. Hasil dari fungsi ini adalah aturan untuk mengambil keputusan. Berikut contoh kodenya.

```
def train_tree(data,y, target_factor, max_depth = None,min_samples_split = None,
min_information_gain = 1e-20, counter=0, max_categories = 20):
  '''
  Trains a Decission Tree
  data: Data to be used to train the Decission Tree
  y: target variable column name
  target_factor: boolean to consider if target variable is factor or numeric.
  max_depth: maximum depth to stop splitting.
  min_samples_split: minimum number of observations to make a split.
  min_information_gain: minimum ig gain to consider a split to be valid.
   max_categories: maximum number of different values accepted for categorical
values. High number of values will slow down learning process. R
  '''

  # Check that max_categories is fulfilled
  if counter==0:
    types = data.dtypes
    check_columns = types[types == "object"].index
```

```
    for column in check_columns:
      var_length = len(data[column].value_counts())
      if var_length > max_categories:
        raise ValueError('The variable ' + column + ' has '+ str(var_length) + ' 
unique values, which is more than the accepted ones: ' +  str(max_categories))

  # Check for depth conditions
  if max_depth == None:
    depth_cond = True

  else:
    if counter < max_depth:
      depth_cond = True

    else:
      depth_cond = False

  # Check for sample conditions
  if min_samples_split == None:
    sample_cond = True

  else:
    if data.shape[0] > min_samples_split:
      sample_cond = True

    else:
      sample_cond = False

  # Check for ig condition
  if depth_cond & sample_cond:

    var,val,ig,var_type = get_best_split(y, data)

    # If ig condition is fulfilled, make split
    if ig is not None and ig >= min_information_gain:

      counter += 1

      left,right = make_split(var, val, data,var_type)

      # Instantiate sub-tree
      split_type = "<=" if var_type else "in"
      question =   "{} {}  {}".format(var,split_type,val)
      # question = "\n" + counter*" " + "|->" + var + " " + split_type + " " + 
str(val)
      subtree = {question: []}

      # Find answers (recursion)
                    yes_answer      =      train_tree(left,y,      target_factor, 
max_depth,min_samples_split,min_information_gain, counter)

                    no_answer      =      train_tree(right,y,      target_factor, 
max_depth,min_samples_split,min_information_gain, counter)

      if yes_answer == no_answer:
        subtree = yes_answer
```

```
      else:
        subtree[question].append(yes_answer)
        subtree[question].append(no_answer)

    # If it doesn't match IG condition, make prediction
    else:
      pred = make_prediction(data[y],target_factor)
      return pred

   # Drop dataset if doesn't match depth or sample conditions
  else:
    pred = make_prediction(data[y],target_factor)
    return pred

  return subtree

max_depth = 5
min_samples_split = 20
min_information_gain  = 1e-5

decision = train_tree(data,'obese',True,
max_depth,min_samples_split,min_information_gain)
decision
```

Hasil keluaran dari penerapan kode di atas untuk pembuatan model DT adalah sebagai berikut.

```
{'Weight <=  103': [{'Height <=  175': [{'Weight <=  74': [{'Height <=  148': [1,
       0]},
     {'Height <=  162': [1, {'Weight <=  82': [0, 1]}]}]},
    0]},
  {'Height <=  189': [{'Weight <=  116': [{'Height <=  168': [1,
       {'Height <=  169': [0, 1]}]},
     1]},
    {'Weight <=  115': [0, 1]}]}]}
```

Prediksi Menggunakan *Decision Tree* dengan Python

Untuk membuat prediksi menggunakan model DT yang telah dibangun dapat dilakukan dengan cara:

1. Bagi keputusan menjadi beberapa bagian.
2. Periksa jenis keputusannya (numerik atau kategoris).
3. Mengingat jenis variabelnya, periksa batas keputusannya. Jika keputusan terpenuhi maka kembalikan hasilnya, jika tidak maka lanjutkan pengambilan keputusan.

Berikut contoh kode implementasi prediksi menggunakan model yang telah dibuat sebelumnya.

```
def classifier_data(observation, arbol):
  question = list(arbol.keys())[0]

  if question.split()[1] == '<=':

    if observation[question.split()[0]] <= float(question.split()[2]):
      answer = arbol[question][0]
    else:
      answer = arbol[question][1]

  else:

    if observation[question.split()[0]] in (question.split()[2]):
      answer = arbol[question][0]
    else:
      answer = arbol[question][1]

  # If the answer is not a dictionary
  if not isinstance(answer, dict):
    return answer
  else:
    residual_tree = answer
    return classifier_data(observation, answer)

#Prediction
obese_prediction = []
num_obs = 50

for i in range(num_obs):
  obs_pred = classifier_data(data.iloc[i,:], decision)
  obese_prediction.append(obs_pred)

print("Predictions: ",obese_prediction,
"\n\nReal values:", data.obese[:num_obs].to_numpy())
```

Berikut hasil keluarannya.

```
Predictions:  [1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1]

Real values: [1 0 1 0 0 0 1 1 0 1 0 1 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 1 1 0 0 1 1 1 0
 1 1 0 1 1 0 1 1 0 1 1 0 1]
```

3.2.1.2 Evaluasi: Confusion Matrix

Confusion matrix merupakan tabel yang digunakan untuk menilai tingkat kesalahan dalam model klasifikasi yang telah dibuat. Baris mewakili kelas aktual yang seharusnya menjadi hasil. Sedangkan kolom mewakili prediksi yang telah dibuat. Dengan menggunakan tabel ini, mudah untuk melihat prediksi mana yang salah. Confusion matrix memiliki 4 kuadran yang berbeda, antara lain:

1. *True Negative*/TN (Kuadran kiri-atas)
2. *False Positive*/FP (Kuadran kanan-atas)
3. *False Negative*/FN (Kuadran kiri-bawah)

4. *True Positive*/TP (Kuadran kanan-bawah)

True berarti nilai yang diprediksi secara akurat dan *False* merepresentasikan adanya prediksi yang salah (*error*).

Tabel 3.1 Parameter pengukuran model Pembelajaran Mesin

Pengukuran	Tujuan	Perhitungan
Akurasi	Mengukur seberapa sering model benar.	$Akurasi = \frac{TP + TN}{Total}$
Presisi	Mengukur seberapa besar yang benar-benar positif dari prediksi positif yang dilakukan.	$Presisi = \frac{TP}{TP + FP}$
Sensitivitas	Mengukur seberapa baik model dalam memprediksi hal positif.	$Sensitivitas = \frac{TP}{TP + FN}$
Spesifisitas	Mengukur seberapa baik model dalam memprediksi hal negatif.	$Spesifisitas = \frac{TN}{TN + FP}$
F-*Score*	F-score adalah "rata-rata harmonik" dari presisi dan sensitivitas. Parameter ini digunakan untuk mempertimbangkan kasus positif palsu dan negatif palsu.	$F - score = 2 \times \frac{Presisi \times Sensitivitas}{Presisi + Sensitivitas}$

Berikut ini merupakan contoh kode *Confusion Matrix* dan pengukuran evaluasi model.

```
# Calculate the confusion matrix
from sklearn import metrics
observation = data.obese[:num_obs].to_numpy()
answer = obese_prediction
confusion_matrix = metrics.confusion_matrix(observation, answer)
# Menghitung Akurasi
Accuracy = metrics.accuracy_score(observation, answer)
# Menghitung Presisi
Precision = metrics.precision_score(observation, answer)
# Menghitung Sensitivitas
Sensitivity_recall = metrics.recall_score(observation, answer)
# Menghitung Spesifisitas
Specificity = metrics.recall_score(observation, answer, pos_label=0)
# Menghitung F-Score
F1_score = metrics.f1_score(observation, answer)

print({"Accuracy":Accuracy,"Precision":Precision,"Sensitivity_recall":Sensitivit
y_recall,"Specificity":Specificity,"F1_score":F1_score})
```

3.2.2 *Random Forest*

Random Forest merupakan algoritma pembelajaran mesin pengembangan dari decision tree. Algoritma ini merupakan kumpulkan dari decision tree yang digabungkan menjadi satu model. *Random Forest* dapat digunakan untuk mengatasi permasalahan

klasifikasi dan regresi. Algoritma ini memungkinkan digunakan untuk pengklasifikasian *dataset* dalam jumlah besar. Karena fungsinya bisa digunakan untuk banyak dimensi dengan berbagai skala dan performa yang tinggi. Penggunaan *tree* yang semakin banyak akan memengaruhi akurasi yang didapat menjadi lebih optimal. Penentuan klasifikasi dengan *Random Forest* dilakukan berdasarkan hasil *voting* dan *tree* yang terbentuk.

Secara garis besar, langkah yang dibutuhkan untuk menyusun algoritma *Random Forest* untuk klasifikasi dan regresi adalah sebagai berikut.

1. Lakukan pembagian data (*split*) untuk data *training* dan *testing* dari data (D).
2. Tentukan ukuran ansambel yang diinginkan (jumlah pohon yang tercakup dalam *Random Forest*) menjadi variabel B. Kemudian pilih metrik *impurity* yang dapat digunakan pada setiap *Decision Tree*.
3. Hasilkan B sampel *Bootstrap* pada data *training*.
4. Ulangi untuk setiap $b = 1 \ldots B$ sampel *Bootstrap*:
   a) Bangun sebuah modifikasi *decision tree* T_b pada sampel Bootstrap dengan mengulang secara bolak-balik langkah berikut. Proses ini berhenti ketika data tidak dapat lagi dipisah, baik karena seluruh sampel pada node saat ini memiliki label yang sama ataupun karena beberapa aturan batas yang telah dicapai:
      i. Pilih sebuah subset secara acak fitur M_{sub} dari total himpunan fitur M pada data input, dimana $M_{sub} = \sqrt{M}$.
      ii. Pilih sebuah fitur f* dari fitur Msub yang terpilih dan sebuah nilai batas x_{f^*} untuk f* dimana data training akan dipisahkan. Pilihan {f*, x_{f^*}} dibuat untuk meminimalkan *metric impurity* terpilih.
      iii. Pisahkan data yang tersedia pada node saat ini ($\mathbf{D}_{node}$) berdasarkan nilai {f*, x_{f^*}}: $\mathbf{D}_{left} = \mathbf{D}_{node} \mid \mathbf{x}_{f^*} \leq x_{f^*}$ dan $\mathbf{D}_{right} = \mathbf{D}_{node} \mid \mathbf{x}_{f^*} > x_{f^*}$. Teruskan nilai $\mathbf{D}_{left}$ dan $\mathbf{D}_{right}$ ke node cabang.
   b) Simpan *decision tree* terlatih pada T_b.
5. Kembalikan ansambel terlatih dari pohon $T_{1\ldots B}$. Jika analisis pada sampel diterapkan, hasilnya akan digunakan untuk menghitung standar kesalahan (SE) pada ansambel .
6. Untuk menghasilkan prediksi pada data uji, teruskan data ke setiap pohon $T_{1..B}$ dan hasilkan prediksi sejumlah himpunan B. Kombinasikan prediksi himpunan B untuk menghitung klasifikasi ataupun regresi.

Berdasarkan langkah-langkah algoritma di atas, maka terlebih dahulu perlu dibuat kelas dasar untuk pemodelan *random forest* secara umum sebagaimana pada Contoh 1 berikut.

Contoh 1: Membangun Kelas Dasar Pemodelan *Random Forest*

```
## imports ##
```

```
import numpy as np
import pandas as pd
import seaborn as sn
import matplotlib.pyplot as plt
from sklearn.base import clone
from sklearn.datasets import load_wine,load_boston
from sklearn.model_selection import StratifiedKFold,cross_validate
from sklearn.metrics import
accuracy_score,precision_score,recall_score,mean_absolute_error,mean_squared_err
or,r2_score,make_scorer
from abc import ABC,abstractmethod
from decisiontrees import DecisionTreeClassifier, DecisionTreeRegressor

#base class for the random forest algorithm
class RandomForest(ABC):
    #initializer
    def __init__(self,n_trees=100):
        self.n_trees = n_trees
        self.trees   = []
    #private function to make bootstrap samples
    def __make_bootstraps(self,data):
        #initialize output dictionary & unique value count
        dc   = {}
        unip = 0
        #get sample size
        b_size = data.shape[0]
        #get list of row indexes
        idx = [i for i in range(b_size)]
        #loop through the required number of bootstraps
        for b in range(self.n_trees):
            #obtain boostrap samples with replacement
            sidx   = np.random.choice(idx,replace=True,size=b_size)
            b_samp = data[sidx,:]
            #compute number of unique values contained in the bootstrap sample
            unip  += len(set(sidx))
            #obtain out-of-bag samples for the current b
            oidx   = list(set(idx) - set(sidx))
            o_samp = np.array([])
            if oidx:
                o_samp = data[oidx,:]
            #store results
            dc['boot_'+str(b)] = {'boot':b_samp,'test':o_samp}
        #return the bootstrap results
        return(dc)

    #public function to return model parameters
    def get_params(self, deep = False):
        return {'n_trees':self.n_trees}

    #protected function to obtain the right decision tree
    @abstractmethod
    def _make_tree_model(self):
        pass

    #protected function to train the ensemble
    def _train(self,X_train,y_train):
```

```
        #package the input data
        training_data = np.concatenate((X_train,y_train.reshape(-1,1)),axis=1)
        #make bootstrap samples
        dcBoot = self.__make_bootstraps(training_data)
        #iterate through each bootstrap sample & fit a model ##
        tree_m = self._make_tree_model()
        dcOob    = {}
        for b in dcBoot:
            #make a clone of the model
            model = clone(tree_m)
            #fit a decision tree model to the current sample
            model.fit(dcBoot[b]['boot'][:,:-1],dcBoot[b]['boot'][:,-1].reshape(-
1, 1))
            #append the fitted model
            self.trees.append(model)
            #store the out-of-bag test set for the current bootstrap
            if dcBoot[b]['test'].size:
                dcOob[b] = dcBoot[b]['test']
            else:
                dcOob[b] = np.array([])
        #return the oob data set
        return(dcOob)

    #protected function to predict from the ensemble
    def _predict(self,X):
        #check we've fit the ensemble
        if not self.trees:
            print('You must train the ensemble before making predictions!')
            return(None)
        #loop through each fitted model
        predictions = []
        for m in self.trees:
            #make predictions on the input X
            yp = m.predict(X)
            #append predictions to storage list
            predictions.append(yp.reshape(-1,1))
        #compute the ensemble prediction
        ypred = np.mean(np.concatenate(predictions,axis=1),axis=1)
        #return the prediction
        return(ypred)
```

Selanjutnya karena fokus pada praktikum kali ini adalah pada algoritma klasifikasi berbasis random forest, maka model klasifikasinya secara lebih spesifik dibuat terlebih dahulu sebagaimana pada Contoh 2.

Contoh 2: Membangun Model *Classifier* Berbasis Algoritma *Random Forest*

```
#class for random forest classifier
class RandomForestClassifierCustom(RandomForest):
   #initializer
    def     __init__(self,n_trees=100,max_depth:     int=None,min_samples_split:
int=2,criterion: str='gini',class_weights='balanced'):
        super().__init__(n_trees)
        self.max_depth             = max_depth
```

```
        self.min_samples_split     = min_samples_split
        self.criterion             = criterion
        self.class_weights         = class_weights

    #protected function to obtain the right decision tree
    def _make_tree_model(self):
        return(DecisionTreeClassifier(max_depth             = self.max_depth,
                                      min_samples_split                         =
self.min_samples_split,
                                      criterion             = self.criterion,
                                      class_weight        = self.class_weights))

    #public function to return model parameters
    def get_params(self, deep = False):
        return {'n_trees':self.n_trees,
                'max_depth':self.max_depth,
                'min_samples_split':self.min_samples_split,
                'criterion':self.criterion,
                'class_weights':self.class_weights}

    #train the ensemble
    def fit(self,X_train,y_train,print_metrics=False):
        #call the protected training method
        dcOob = self._train(X_train,y_train)
        #if selected, compute the standard errors and print them
        if print_metrics:
            #initialise metric arrays
            accs = np.array([])
            pres = np.array([])
            recs = np.array([])
            #loop through each bootstrap sample
            for b,m in zip(dcOob,self.trees):
                #compute the predictions on the out-of-bag test set & compute
metrics
                if dcOob[b].size:
                    yp  = m.predict(dcOob[b][:,:-1])
                    acc = accuracy_score(dcOob[b][:,-1],yp)
                    pre = precision_score(dcOob[b][:,-1],yp,average='weighted')
                    rec = recall_score(dcOob[b][:,-1],yp,average='weighted')
                    #store the error metrics
                    accs = np.concatenate((accs,acc.flatten()))
                    pres = np.concatenate((pres,pre.flatten()))
                    recs = np.concatenate((recs,rec.flatten()))
            #print standard errors
            print("Standard error in accuracy: %.2f" % np.std(accs))
            print("Standard error in precision: %.2f" % np.std(pres))
            print("Standard error in recall: %.2f" % np.std(recs))

    #predict from the ensemble
    def predict(self,X):
        #call the protected prediction method
        ypred = self._predict(X)
        #convert the results into integer values & return
        return(np.round(ypred).astype(int))
```

Selanjutnya, sebuah dataset bawaan (Wine Dataset) dipanggil untuk diimplementasikan pada algoritma yang telah dibuat. Untuk memanggil dataset ini cukup mengimpor *library* scikit-learn pada kode yang disusun. Setelah dipanggil, dataset akan dilatihkan pada model dan dievaluasi hasilnya menggunakan *confusion matrix* sebagaimana pada contoh 3.

Contoh 3: Menerapkan Model *Classifier* Random Forest Pada Kasus

```
#load the wine dataset
dfX,sY = load_wine(return_X_y=True, as_frame=True)

#check the dimensions of these data
print('Dimensions of X: ',dfX.shape)
print('Dimensions of y: ',sY.shape)

#what unique classes exist in the label variable?
print('Classes in the label: ',sY.unique())

#what is the frequency of the classes in the dataset?
sY.hist()
plt.show()

#view the first 5 rows of input features
dfX.head(5)

#make a boxplot to view the distribution in these data
dfX.boxplot(figsize=(20,10),rot=45)
plt.show()

## plot the pearson correlation for our input features ##
fig, ax = plt.subplots(figsize = (10, 10))
dfCorr  = dfX.corr()
sn.heatmap(dfCorr)
plt.show()

#convert all correlations to positive values
dfCorr = dfCorr.abs()

#loop through rows
for index, sRow in dfCorr.iterrows():
    #get the valid entries
    sCorrs = sRow[sRow.index != index]
    sCorrs = sCorrs[sCorrs > 0.8]
    #print out results
    if not sCorrs.empty:
        print('highly     correlated     input     features:     ',index,'     &
',sCorrs.index.values)

#create a random forest with balance class weights enabled
rfcC = RandomForestClassifierCustom(class_weights='balanced')

## train the ensemble & view estimates for prediction error ##
rfcC.fit(dfX.values,sY.values,print_metrics=False)
```

```
## use k fold cross validation to measure performance ##
scoring_metrics = {'accuracy': make_scorer(accuracy_score),
                   'precision':                    make_scorer(precision_score,
average='weighted'),
                   'recall': make_scorer(recall_score, average='weighted')}
dcScores                                                                    =
cross_validate(rfcC,dfX.values,sY.values,cv=StratifiedKFold(10),scoring=scoring_
metrics)
print('Mean Accuracy: %.2f' % np.mean(dcScores['test_accuracy']))
print('Mean Precision: %.2f' % np.mean(dcScores['test_precision']))
print('Mean Recall: %.2f' % np.mean(dcScores['test_recall']))
```

Sebagai pembanding, data yang sama juga dilatihkan pada algoritma yang sama tetapi bawaan dari *library* Scikit-Learn sebagaimana pada Contoh 4 berikut.

Contoh 4: Membandingkan Hasil Evaluasi Algoritma Random Forest dengan Scikit-Learn

```
## import the scikit-learn models ##
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

#create a random forest with balanced class weights
rfc = RandomForestClassifier(class_weight='balanced')

## use k fold cross validation to measure performance ##
scoring_metrics = {'accuracy': make_scorer(accuracy_score),
                   'precision':                    make_scorer(precision_score,
average='weighted'),
                   'recall': make_scorer(recall_score, average='weighted')}
dcScores                                                                    =
cross_validate(rfc,dfX.values,sY.values,cv=StratifiedKFold(10),scoring=scoring_m
etrics)
print('Mean Accuracy: %.2f' % np.mean(dcScores['test_accuracy']))
print('Mean Precision: %.2f' % np.mean(dcScores['test_precision']))
print('Mean Recall: %.2f' % np.mean(dcScores['test_recall']))
```

3.2.3 Naïve Bayes

Naïve Bayes merupakan algoritma klasifikasi berbasis probabilitas. Dasar dari model ini adalah teorema Bayes. Sederhananya, Naïve Bayes mengklasifikasikan sebuah *instance* ($P(C|x)$) dengan menghitung posterior tiap kelas sebagaimana pada persamaan 3.6 berikut.

$$P(x) = \frac{P(C)P(C)}{P(x)} \tag{3.6}$$

Dimana $P(x)$ adalah percaya pada C berdasarkan observasi x; $P(C)$ adalah probabilitas pengamatan x jika C benar; $P(C)$ adalah probabilitas pengamatan C; dan $P(x)$adalah probabilitas pengamatan x.

Untuk melatih Naïve Bayes, perhitungan dasar probabilitas menggunakan Teorema Bayes terlebih dahulu harus dilakukan. Kemungkinan pada persamaan di atas dapat dengan cepat dihitung apabila distribusi x diketahui. Hanya saja distribusi x adalah sesuatu yang perlu ditentukan terlebih dahulu. Pada praktikum kali ini jenis distribusi data yang akan diterapkan diasumsikan memiliki distribusi normal (Gaussian) untuk menghitung posterior.

Berdasarkan langkah di atas, dapat disusun secara manual algoritma Naïve Bayes untuk klasifikasi dataset "iris.csv". Secara lebih detail dapat dicermati pada Contoh 5 berikut ini.

Contoh 5: Membangun Model Algoritma Naïve Bayes dari Awal

```
#Importing Libraries
import pandas as pd
import numpy as np

#Load Data
df = pd.read_csv("iris.csv")
df = df.drop("Id", axis = 1)
print(df)

#Train Test Split
train = df.sample(frac = 0.7, random_state = 1)
test = df.drop(train.index)

y_train = train["Species"]
x_train = train.drop("Species", axis = 1)

y_test = test["Species"]
x_test = test.drop("Species", axis = 1)

#Training – Count Posterior
means = train.groupby(["Species"]).mean() # Find mean of each class
var = train.groupby(["Species"]).var() # Find variance of each class
prior = (train.groupby("Species").count() / len(train)).iloc[:,1] # Find prior
probability of each class
classes = np.unique(train["Species"].tolist()) # Storing all possible classes

#Classification
def Normal(n, mu, var):
    # Function to return pdf of Normal(mu, var) evaluated at x
    sd = np.sqrt(var)
    pdf = (np.e ** (-0.5 * ((n - mu)/sd) ** 2)) / (sd * np.sqrt(2 * np.pi))
    return pdf

def Predict(X):
    Predictions = []

    for i in X.index: # Loop through each instances
        ClassLikelihood = []
        instance = X.loc[i]
        for cls in classes: # Loop through each class
            FeatureLikelihoods = []
```

```
            FeatureLikelihoods.append(np.log(prior[cls])) # Append log prior of
class 'cls'
            for col in x_train.columns: # Loop through each feature
                data = instance[col]
                mean = means[col].loc[cls] # Find the mean of column 'col' that
are in class 'cls'
                variance = var[col].loc[cls] # Find the variance of column 'col'
that are in class 'cls'
                Likelihood = Normal(data, mean, variance)
                if Likelihood != 0:
                    Likelihood = np.log(Likelihood) # Find the log-likelihood
evaluated at x
                else:
                    Likelihood = 1/len(train)

                FeatureLikelihoods.append(Likelihood)

            TotalLikelihood = sum(FeatureLikelihoods) # Calculate posterior
            ClassLikelihood.append(TotalLikelihood)

        MaxIndex = ClassLikelihood.index(max(ClassLikelihood)) # Find largest
posterior position
        Prediction = classes[MaxIndex]
        Predictions.append(Prediction)
    return Predictions

def Accuracy(y, prediction):
    # Function to calculate accuracy
    y = list(y)
    prediction = list(prediction)
    score = 0
    for i, j in zip(y, prediction):
        if i == j:
            score += 1
    return score / len(y)

PredictTrain = Predict(x_train)
PredictTest = Predict(x_test)

print('Training Accuracy: %.4f' % round(Accuracy(y_train, PredictTrain), 5))
print('Testing Accuracy: %.4f' % round(Accuracy(y_test, PredictTest), 5))
```

Selanjutnya, data yang telah dilatih dan diujikan menggunakan algoritma yang dibuat dari awal di atas akan dibandingkan dengan hasil training dan uji menggunakan *library* dari Scikit-Learn sebagai berikut.

Contoh 6: Membandingkan Hasil Evaluasi Algoritma Naïve Bayes dengan Scikit-Learn

```
from sklearn.naive_bayes import GaussianNB

clf = GaussianNB()
clf.fit(x_train, y_train)
SkTrain = clf.predict(x_train) # Predicting on the train set
SkTest = clf.predict(x_test) # Predicting on the test set
```

```
print('Training Accuracy: %.4f' % round(Accuracy(y_train, SkTrain), 5))
print('Testing Accuracy: %.4f' % round(Accuracy(y_test, SkTest), 5))
```

3.3 Tugas Praktikum

1. Cobalah kode pembuatan model Decision Tree hingga evaluasinya sesuai contoh pada modul. Buat *flowchart* berdasarkan pemahaman Anda dan analisis hasilnya!
2. Ubahlah fungsi pemisahan menjadi Gini index dari entropi! Analisislah hasilnya.
3. Bukalah dataset 'Iris' pada link berikut: https://archive.ics.uci.edu/dataset/53/iris
   Bagilah dataset di atas dengan pembagian sebagai berikut (petunjuk: gunakan modul data splitting '`train_test_split`' pada sklearn untuk membagi dataset).

Data	% Sample
Training	70
Testing	30

   Berdasarkan data tersebut:
   a. Buat model *decision tree* menggunakan data training yang telah disiapkan, buat model dengan kriteria Gini dan Entropi.
   b. Prediksikan masing-masing model menggunakan data *testing*.
   c. Buat *confusion matrix* dari masing-masing model.
   d. Hitung akurasi, presisi, sensitivitas, dan spesifitas dari masing-masing model.
   e. Bandingkan dan analisis hasil yang diperoleh.
4. Cobalah kode pembuatan model *Random Forest* hingga evaluasinya sesuai **Contoh 1-3** pada modul. Buat *flowchart* berdasarkan pemahaman Anda dan analisis hasilnya!
5. Bandingkan hasil evaluasi klasifikasi wine dataset sesuai tabel berikut ini dan analisis hasilnya!

Algoritma Klasifikasi	Rerata Akurasi (%)	Rerata Presisi (%)	Rerata *Recall* (%)
Random Forest (Manual)			
Random Forest (Scikit-Learn)			
Decision Tree (Manual)			
Decision Tree (Scikit-Learn)			

6. Ubahlah fungsi pemisahan menjadi Gini index dari entropi! Analisislah hasilnya.
7. Cobalah kode pembuatan model Naïve Bayes hingga evaluasinya sesuai **Contoh 5** pada modul. Buat *flowchart* berdasarkan pemahaman Anda dan analisis hasilnya!
8. Buat klasifikasi menggunakan Wine Dataset pada contoh sebelumnya menggunakan algoritma Naïve Bayes! Analisislah hasilnya.

3.4 Referensi

Ben Auffarth, 2021, "*Machine Learning for Time-Series with Python*", PACKT Publishing.

Dangeti, P., 2017, "*Statistics for Machine Learning*", PACKT Publishing

Gopal Sakarkar, Gaurav Patil, dan Prateek Dutta, 2021, "*Machine Learning Algorithms Using Python Programming*", Nova Science Publishers, Inc.