Machine Learning Assignment 2

Supervised Learning Classification using K-Nearest Neighbor with KDTree Algorithm

by:

Muhammad Furqon Fahlevi 1301194214

Muhammad Ilham Mubarak 1301194276

A. Problem Formulation

Classification refers to a predictive modeling problem where a class label is predicted for a given example of input data. In this assignment, we are going to predict whether customers are interested in buying a new vehicle or not based on customer data at the dealer**.**

B. Data Exploration and Preparation

1. Import Library

In this step, we import all of the necessary library to complete the assignment

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from scipy import stats
from math import sqrt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from collections import Counter
import heapq
```

2. Import Datasets

i. Data Train

```
[ ] # Import dataset (kendaraan_train.csv) into new dataframe named data_train
    data_train = pd.read_csv("https://raw.githubusercontent.com/furqonfahlevi/classification-ml/main/kendaraan_train.csv")
    data_train.head()
```

	id	Jenis_Kelamin	Umur	SIM	Kode_Daerah	Sudah_Asuransi	Umur_Kendaraan	Kendaraan_Rusak	Premi	Kanal_Penjualan	Lama_Berlangganan	Tertarik
0	1	Wanita	30.0	1.0	33.0	1.0	< 1 Tahun	Tidak	28029.0	152.0	97.0	0
1	2	Pria	48.0	1.0	39.0	0.0	> 2 Tahun	Pernah	25800.0	29.0	158.0	0
2	3	NaN	21.0	1.0	46.0	1.0	< 1 Tahun	Tidak	32733.0	160.0	119.0	0
3	4	Wanita	58.0	1.0	48.0	0.0	1-2 Tahun	Tidak	2630.0	124.0	63.0	0
4	5	Pria	50.0	1.0	35.0	0.0	> 2 Tahun	NaN	34857.0	88.0	194.0	0

ii. Data Test

```
[ ] # Import dataset (kendaraan_test.csv) into new dataframe named data_test
    data_test = pd.read_csv("https://raw.githubusercontent.com/furqonfahlevi/classification-ml/main/kendaraan_test.csv")
    data_test.head()
```

	Jenis_Kelamin	Umur	SIM	Kode_Daerah	Sudah_Asuransi	Umur_Kendaraan	Kendaraan_Rusak	Premi	Kanal_Penjualan	Lama_Berlangganan	Tertarik
0	Wanita	49	1	8	0	1-2 Tahun	Pernah	46963	26	145	0
1	Pria	22	1	47	1	< 1 Tahun	Tidak	39624	152	241	0
2	Pria	24	1	28	1	< 1 Tahun	Tidak	110479	152	62	0
3	Pria	46	1	8	1	1-2 Tahun	Tidak	36266	124	34	0
4	Pria	35	1	23	0	1-2 Tahun	Pernah	26963	152	229	0

iii. Data Shape

Shape function return the shape of an array. Shape of an array is a tuple with the number of elements per axis (dimension). In our case, data train has 285831 lines and 12 columns, data test has 47639 lines and 11 columns.

```
[ ]  print("Data Train Shape", data_train.shape)
     print("Data Test Shape", data_test.shape)

     Data Train Shape (285831, 12)
     Data Test Shape (47639, 11)
```

3. Pre-processing for Datasets

i. Drop Unnecessary Column

Drop unnecessary column

```
[ ]  del data_train['id']
     data_train.head()
```

	Jenis_Kelamin	Umur	SIM	Kode_Daerah	Sudah_Asuransi	Umur_Kendaraan	Kendaraan_Rusak	Premi	Kanal_Penjualan	Lama_Berlangganan	Tertarik
0	Wanita	30.0	1.0	33.0	1.0	< 1 Tahun	Tidak	28029.0	152.0	97.0	0
1	Pria	48.0	1.0	39.0	0.0	> 2 Tahun	Pernah	25800.0	29.0	158.0	0
2	NaN	21.0	1.0	46.0	1.0	< 1 Tahun	Tidak	32733.0	160.0	119.0	0
3	Wanita	58.0	1.0	48.0	0.0	1-2 Tahun	Tidak	2630.0	124.0	63.0	0
4	Pria	50.0	1.0	35.0	0.0	> 2 Tahun	NaN	34857.0	88.0	194.0	0

ii. Duplicated Data

Duplicated data leads to inaccurate results, we tried to figure it out with dropping them.

Duplicated data

```
[ ]  # Check duplicate data
     print("Duplicated data:",data_train.duplicated().sum())
     print("Duplicated data:",data_test.duplicated().sum())
     data_train.drop_duplicates(inplace=True)
     data_test.drop_duplicates(inplace=True)

     Duplicated data: 169
     Duplicated data: 3

[ ]  print("Duplicated data:",data_train.duplicated().sum())
     print("Duplicated data:",data_test.duplicated().sum())

     Duplicated data: 0
     Duplicated data: 0
```

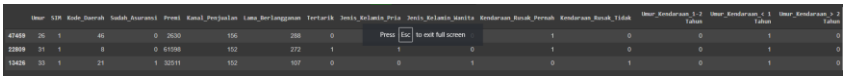
iii. Encoder Data

Categorical data are variables that contain label values rather than numeric values. In this case, we tried to convert a numerical variable to an ordinal variable. Each categorical column assigned with an integer value.

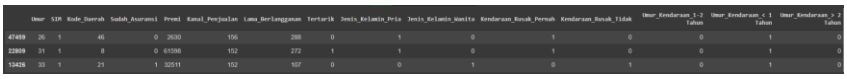
Encoder Data

```
[8]  categorical = ["Jenis_Kelamin", "Kendaraan_Rusak", "Umur_Kendaraan"]
```

```
[9]  data_train = pd.get_dummies(data_train, columns=categorical)
     data_train.sample(3)
```



```
[10] data_test = pd.get_dummies(data_test, columns=categorical)
     data_test.sample(3)
```



iv. Check NaN

We check whether there is a NaN data or not in every column. After checked, we tried to fill every NaN value with 0, mean and median. The data that are filled with 0 because the data only consist of 0 and 1.

Check NaN

```
[11] data_train.isna().sum()
```

```
Umur                          14199
SIM                           14404
Kode_Daerah                   14291
Sudah_Asuransi                14229
Premi                         14510
Kanal_Penjualan               14297
Lama_Berlangganan             13926
Tertarik                          0
Jenis_Kelamin_Pria                0
Jenis_Kelamin_Wanita              0
Kendaraan_Rusak_Pernah            0
Kendaraan_Rusak_Tidak             0
Umur_Kendaraan_1-2 Tahun          0
Umur_Kendaraan_< 1 Tahun          0
Umur_Kendaraan_> 2 Tahun          0
dtype: int64
```

```
[12] data_train["SIM"].fillna(0, inplace=True)
     data_train["Sudah_Asuransi"].fillna(0, inplace=True)
     data_train["Umur"].fillna(data_train["Umur"].mean(), inplace=True)
     data_train["Premi"].fillna(data_train["Premi"].median(), inplace=True)
     data_train["Kanal_Penjualan"].fillna(data_train["Kanal_Penjualan"].mean(), inplace=True)
     data_train["Lama_Berlangganan"].fillna(data_train["Lama_Berlangganan"].mean(), inplace=True)
     data_train["Kode_Daerah"].fillna(data_train["Kode_Daerah"].mean(), inplace=True)
```

```
[13] data_train.isna().sum()
```

```
Umur                          0
SIM                           0
Kode_Daerah                   0
Sudah_Asuransi                0
Premi                         0
Kanal_Penjualan               0
Lama_Berlangganan             0
Tertarik                      0
Jenis_Kelamin_Pria            0
Jenis_Kelamin_Wanita          0
Kendaraan_Rusak_Pernah        0
Kendaraan_Rusak_Tidak         0
Umur_Kendaraan_1-2 Tahun      0
Umur_Kendaraan_< 1 Tahun      0
Umur_Kendaraan_> 2 Tahun      0
dtype: int64
```

```
[14] data_test.isna().sum()
```

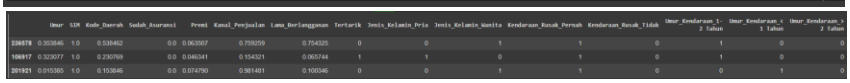
```
Umur                          0
SIM                           0
Kode_Daerah                   0
Sudah_Asuransi                0
Premi                         0
Kanal_Penjualan               0
Lama_Berlangganan             0
Tertarik                      0
Jenis_Kelamin_Pria            0
Jenis_Kelamin_Wanita          0
Kendaraan_Rusak_Pernah        0
Kendaraan_Rusak_Tidak         0
Umur_Kendaraan_1-2 Tahun      0
Umur_Kendaraan_< 1 Tahun      0
Umur_Kendaraan_> 2 Tahun      0
dtype: int64
```

v. Data Standardization

Standardization is scaling technique where the values are centered around the mean with a unit standard deviation. This means that the mean of the attribute becomes zero and the resultant distribution has a unit standard deviation.

Data Standarization

```
[40] scaler = MinMaxScaler()
     columns = ["Umur", "Kode_Daerah", "Premi", "Kanal_Penjualan", "Lama_Berlangganan"]
     data_train[columns] = scaler.fit_transform(data_train[columns])
     data_train.sample(3)
```



```
[41] data_test[columns] = scaler.fit_transform(data_test[columns])
     data_test.sample(3)
```

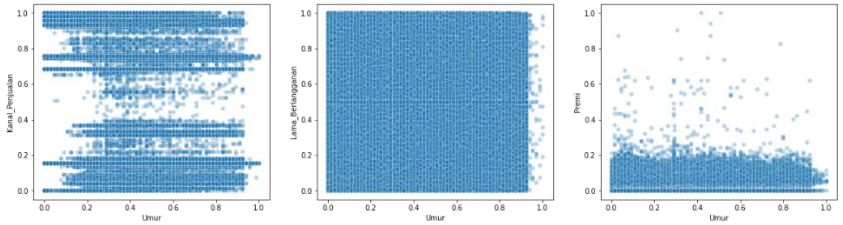
	Umur	SIM	Kode_Daerah	Sudah_Asuransi	Premi	Kanal_Penjualan	Lama_Berlangganan	Tertarik	Jenis_Kelamin_Pria	Jenis_Kelamin_Wanita	Kendaraan_Rusak_Pernah	Kendaraan_Rusak_Tidak	Umur_Kendaraan_1-2 Tahun	Umur_Kendaraan_< 1 Tahun	Umur_Kendaraan_> 2 Tahun
34902	0.461538	1	0.557692	0	0.060915	0.154321	0.410685	0	0	1	1	0	1	0	0
37929	0.800000	1	0.269462	0	0.053844	0.080247	0.114187	0	0	1	1	0	1	0	0
13272	0.430769	1	0.153846	0	0.074927	0.759259	0.217993	0	0	1	1	0	1	0	0

vi. Check Outliers

Outlier is an observation point that is distant from other observations. We tried to check continuous data whether there is an outlier or not.

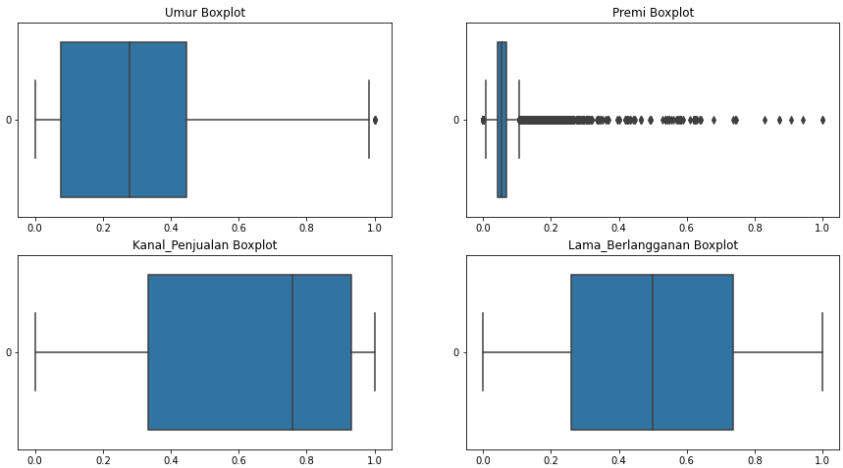
Check outliers

```
[17] fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(20, 5))
     sns.scatterplot(data=data_train, x="Umur", y="Kanal_Penjualan", ax=axes[0], alpha=0.3)
     sns.scatterplot(data=data_train, x="Umur", y="Lama_Berlangganan", ax=axes[1], alpha=0.3)
     sns.scatterplot(data=data_train, x="Umur", y="Premi", ax=axes[2], alpha=0.3)
     plt.show()
```




```
[18] fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(15, 8))

     axes[0, 0].title.set_text("Umur Boxplot")
     axes[0, 1].title.set_text("Premi Boxplot")
     axes[1, 0].title.set_text("Kanal_Penjualan Boxplot")
     axes[1, 1].title.set_text("Lama_Berlangganan Boxplot")

     sns.boxplot(data=data_train["Umur"], ax=axes[0, 0], orient="horizontal")
     sns.boxplot(data=data_train["Premi"], ax=axes[0, 1], orient="horizontal")
     sns.boxplot(data=data_train["Kanal_Penjualan"], ax=axes[1, 0], orient="horizontal")
     sns.boxplot(data=data_train["Lama_Berlangganan"], ax=axes[1, 1], orient="horizontal")
     plt.show()
```

As we can see from the boxplot below, "Umur" and "Premi" column have outliers.



vii. Z-Score Method

Z-score is measured in terms of standard deviations from the mean. Any z-score greater than 3 or less than -3 is considered to be an

outlier (threshold). In this case, we tried to find z-score from the chosen columns.

Z-Score method

```
[42] columns = ['Kanal_Penjualan', 'Premi', 'Umur', 'Lama_Berlangganan']
     z = np.abs(stats.zscore(data_train[columns]))
     threshold = 3

     data_train_new = data_train[(z < threshold).all(axis=1)]
     data_train_new.sample(3)
```

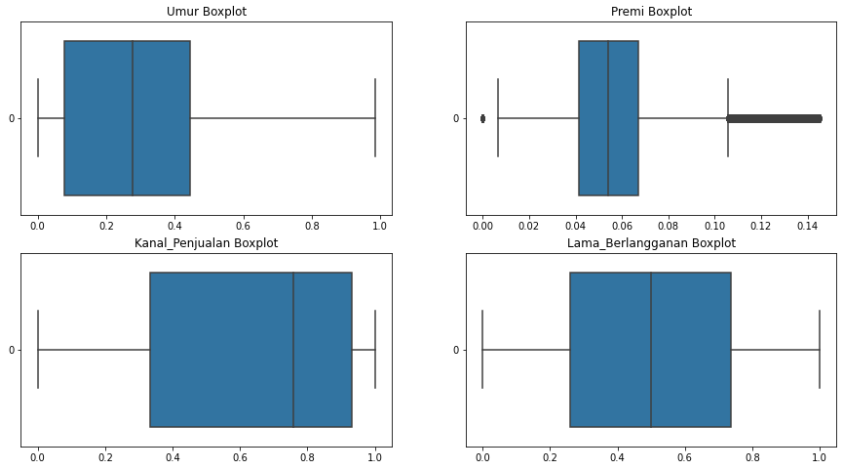
	Umur	SIM	Kode_Daerah	Sudah_Asuransi	Premi	Kanal_Penjualan	Lama_Berlangganan	Tertarik	Jenis_Kelamin_Pria	Jenis_Kelamin_Wanita	Kendaraan_Rusak_Pernah	Kendaraan_Rusak_Tidak	Umur_Kendaraan_1-2 Tahun	Umur_Kendaraan_< 1 Tahun	Umur_Kendaraan_> 2 Tahun
106872	0.046154	1.0	0.557692	1.0	0.057985	0.932099	0.499247	0	1	0	0	1	0	1	0
128116	0.569231	1.0	0.538462	0.0	0.082276	0.154321	0.096886	1	1	0	1	0	0	0	1
91279	0.289956	1.0	0.346154	0.0	0.066494	0.944444	0.543253	1	0	1	1	0	1	0	0

```
[43] columns = ['Kanal_Penjualan', 'Premi', 'Umur', 'Lama_Berlangganan']
     z = np.abs(stats.zscore(data_test[columns]))
     threshold = 3

     data_test_new = data_test[(z < threshold).all(axis=1)]
     data_test_new.sample(3)
```

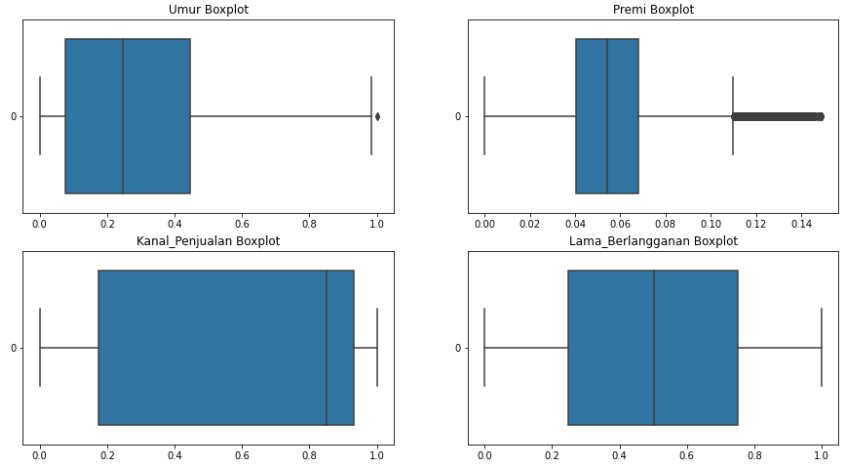
	Umur	SIM	Kode_Daerah	Sudah_Asuransi	Premi	Kanal_Penjualan	Lama_Berlangganan	Tertarik	Jenis_Kelamin_Pria	Jenis_Kelamin_Wanita	Kendaraan_Rusak_Pernah	Kendaraan_Rusak_Tidak	Umur_Kendaraan_1-2 Tahun	Umur_Kendaraan_< 1 Tahun	Umur_Kendaraan_> 2 Tahun
6473	0.430000	1	0.384615	1	0.00000	0.154321	0.982699	0	1	0	0	1	1	0	0
11679	0.523077	1	0.153846	0	0.00000	0.154321	0.301038	0	0	1	1	0	1	0	0
16672	0.600000	1	0.538462	0	0.04958	0.154321	0.435986	0	1	0	1	0	1	0	0

```
[21] fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(15, 8))

     axes[0, 0].title.set_text("Umur Boxplot")
     axes[0, 1].title.set_text("Premi Boxplot")
     axes[1, 0].title.set_text("Kanal_Penjualan Boxplot")
     axes[1, 1].title.set_text("Lama_Berlangganan Boxplot")

     sns.boxplot(data=data_train_new["Umur"], ax=axes[0, 0], orient="horizontal")
     sns.boxplot(data=data_train_new["Premi"], ax=axes[0, 1], orient="horizontal")
     sns.boxplot(data=data_train_new["Kanal_Penjualan"], ax=axes[1, 0], orient="horizontal")
     sns.boxplot(data=data_train_new["Lama_Berlangganan"], ax=axes[1, 1], orient="horizontal")
     plt.show()
```

After doing the z-score method, we can see from the boxplot below that most of the outliers almost disappear especially the “Premi”.



```
[22] fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(15, 8))

     axes[0, 0].title.set_text("Umur Boxplot")
     axes[0, 1].title.set_text("Premi Boxplot")
     axes[1, 0].title.set_text("Kanal_Penjualan Boxplot")
     axes[1, 1].title.set_text("Lama_Berlangganan Boxplot")

     sns.boxplot(data=data_test_new["Umur"], ax=axes[0, 0], orient="horizontal")
     sns.boxplot(data=data_test_new["Premi"], ax=axes[0, 1], orient="horizontal")
     sns.boxplot(data=data_test_new["Kanal_Penjualan"], ax=axes[1, 0], orient="horizontal")
     sns.boxplot(data=data_test_new["Lama_Berlangganan"], ax=axes[1, 1], orient="horizontal")
     plt.show()
```



viii. Dimensional Reduction (PCA)

Dimension reduction eliminates noisy data dimensions and thus and improves accuracy in classification, in addition to reduced computational cost.

Dimensial Reduction (PCA)

```
[ ]  X = data_train_new.drop(columns=["Tertarik"])
     y = data_train_new[["Tertarik"]]

     X_test_data = data_test_new.drop(columns=["Tertarik"])
     y_test_data = data_test_new[["Tertarik"]]
```

```
[ ]  pca = PCA(n_components=2)
     principal_df = pd.DataFrame(data = pca.fit_transform(X))
     principal_test = pd.DataFrame(data = pca.fit_transform(X_test_data))
```

```
[ ]  print(f"Data Train New Shape: {principal_df.shape}")
     print(f"Data Test New Shape: {principal_test.shape}")

     Data Train New Shape: (284032, 2)
     Data Test New Shape: (47383, 2)
```

We can see from the image above; those datasets have been reduced from 15 columns into 2 columns. This are very important because PCA creates a smaller data in volume and has the same analytical results as the original representation.

C. Modeling

1. Data Splitting

In our case, we tried to split the data with Pareto Principle, or it also called 80/20 rule, 80% of effects come from 20% of causes.

Data Splitting

```
[ ]  X = data_train_new.drop(columns=["Tertarik"])
     y = data_train_new[["Tertarik"]]
```

```
[ ]  X_train, X_test, y_train, y_test = train_test_split(principal_df, y, test_size = 0.2, random_state = 42)
```

```
[ ]  print(f"X_train shape: {X_train.shape} | X_test shape: {X_test.shape}")
     print(f"y_train shape: {y_train.shape} | y_test shape: {y_test.shape}")
```

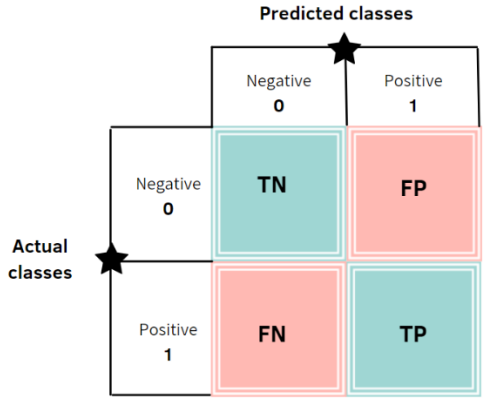
```
X_train shape: (227225, 2) | X_test shape: (56807, 2)
y_train shape: (227225, 1) | y_test shape: (56807, 1)
```

2. Evaluation Metrics

```python
class Metrics:
  def __init__(self, prediction, test):

    if isinstance(test, pd.DataFrame):
      test = test.values

    if isinstance(prediction, pd.DataFrame):
      prediction = prediction.values

    self.prediction =  np.array(prediction)
    self.test = np.array(test).reshape(-1)

    unique = set(self.prediction)
    matrix = [list() for x in range(len(unique))]
    for i in range(len(unique)):
      matrix[i] = [0 for x in range(len(unique))]
    lookup = dict()
    for i, value in enumerate(unique):
      lookup[value] = i
    for i in range(len(self.prediction)):
      x = lookup[self.prediction[i]]
      y = lookup[self.test[i]]
      matrix[y][x] += 1
    matrix[0][1], matrix[1][0] = matrix[1][0], matrix[0][1]

    self.tn = matrix[0][0]
    self.tp = matrix[1][1]
    self.fn = matrix[1][0]
    self.fp = matrix[0][1]
    self.matrix = matrix

  def accuracy_metric(self):
    correct = 0
    for i in range(len(self.prediction)):
      if self.prediction[i] == self.test[i]:
        correct += 1
    return correct / float(len(self.prediction)) * 100.0

  def confusion_matrix(self):
    return self.matrix

  def f1_score(self):
    p = self.precision_score()
    r = self.recall_score()
    return (2 * ((p * r) / (p + r))) / 100

  def precision_score(self):
    return (self.tp / ((self.tp) + (self.fp))) * 100

  def recall_score(self):
    return (self.tp / ((self.tp) + (self.fn))) * 100

  def visualize_confustion_matrix(self):
    cf = self.confusion_matrix()
    sns.heatmap(cf,
                xticklabels=["Negative (0)", "Positive (1)"],
                yticklabels=["Negative (0)", "Positive (1)"],
                annot=True, fmt='d',
                cmap="YlGnBu", linewidths=10)
    plt.show()

  def show(self):
    self.visualize_confustion_matrix()
    print("Accuracy with K-NN: {0:.2f}%".format(self.accuracy_metric()))
    print("Precision with K-NN: {0:.2f}%".format(self.precision_score()))
    print("Recall with K-NN: {0:.2f}%".format(self.recall_score()))
    print("F1-Score with K-NN: {0:.2f}".format(self.f1_score()))
```

i. Confusion Matrix

Confusion matrix is a summary of prediction results on a classification problem. It shows the ways in which your classification model is confused when it makes predictions. We create ours based on image below:



Therefore, we applied into our code:

```python
def __init__(self, prediction, test):

    if isinstance(test, pd.DataFrame):
      test = test.values

    if isinstance(prediction, pd.DataFrame):
      prediction = prediction.values

    self.prediction =  np.array(prediction)
    self.test = np.array(test).reshape(-1)

    unique = set(self.prediction)
    matrix = [list() for x in range(len(unique))]
    for i in range(len(unique)):
      matrix[i] = [0 for x in range(len(unique))]
    lookup = dict()
    for i, value in enumerate(unique):
      lookup[value] = i
    for i in range(len(self.prediction)):
      x = lookup[self.prediction[i]]
      y = lookup[self.test[i]]
      matrix[y][x] += 1
    matrix[0][1], matrix[1][0] = matrix[1][0], matrix[0][1]

    self.tn = matrix[0][0]
    self.tp = matrix[1][1]
    self.fn = matrix[1][0]
    self.fp = matrix[0][1]
    self.matrix = matrix
```

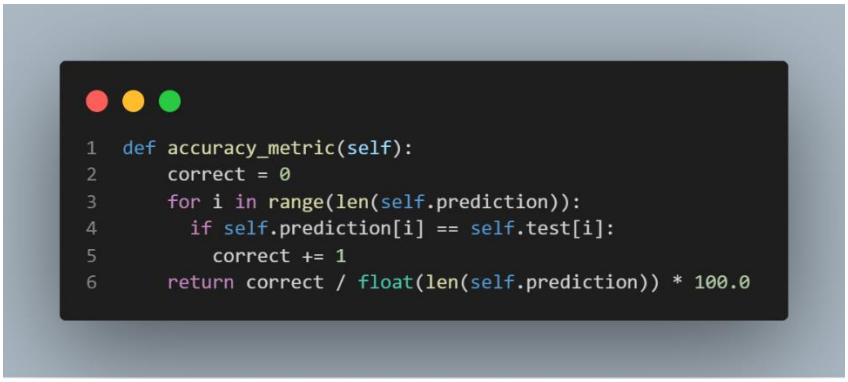
ii. Accuracy Score

Classification accuracy is the ratio of number of correct predictions to the total number of input samples. We need to find is predict

data same as test data. If they are equal, they will be assigned into "correct" variable and then we calculate based on formula below.

$$Accuracy = \frac{number\ of\ correct\ predictions}{total\ number\ of\ predictions\ made}$$

We applied this formula into our case to find the classification accuracy.



```
def accuracy_metric(self):
    correct = 0
    for i in range(len(self.prediction)):
      if self.prediction[i] == self.test[i]:
        correct += 1
    return correct / float(len(self.prediction)) * 100.0
```

iii. Precision Score

Precision is the ratio of correctly predicted positive observations to the total predicted positive observations, with formula:

$$Precision = \frac{True\ Positive}{(True\ Positive\ + False\ Positive)}$$

```
def precision_score(self):
    return (self.tp / ((self.tp) + (self.fp))) * 100
```

iv. Recall Score

Recall is the ratio of correctly predicted positive observations to all observations in actual class, with formula:

$$Recall = \frac{True\ Positive}{(True\ Positive + False\ Negative)}$$

```
def recall_score(self):
    return (self.tp / ((self.tp) + (self.fn))) * 100
```

v. F1 Score

F1 score is defined as the harmonic mean of precision and recall. Combining the precision and recall metrics into a single metric, F1 score work well on imbalanced data. With formula:

$$F1\ Score = 2 * \frac{(Precision * Recall)}{(Precision + Recall)}$$

```python
def f1_score(self):
    p = self.precision_score()
    r = self.recall_score()
    return (2 * ((p * r) / (p + r))) / 100
```

3. KDTree

KDTree is a generalization of binary search tree that stores points in k-dimensional space. This means that KDTree can be used to store an array of points in the Cartesian plane. Early stages of starting KDTree are define the nodes first. In our case, every node contains points/value, label (Tertarik) with binary value and distance. Every node contains left and right node to connect between other nodes.

kNN

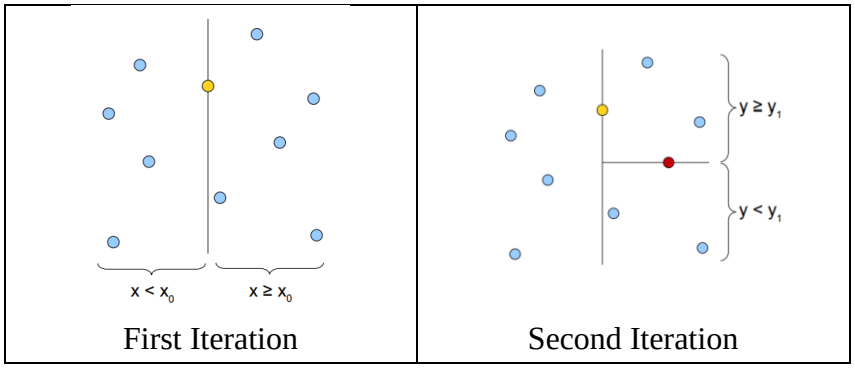
```python
class KDNode:
    def __init__(self, points, y, left = None, right = None, distance = np.inf):
        self.points = points
        self.y = y
        self.left = left
        self.right = right
        self.distance = distance
    def __lt__(self, other):
        return self.distance < other.distance
```

```python
class KDTree:
    def __init__(self, k = 2, p = 2):
        self.tree = None
        self.k = k
        self.depth = 0
        self.heap = []
        self.p = p

    def fit(self, X, y):
        self.X = X
        self.y = y

        if isinstance(X, pd.DataFrame):
            X = X.values

        if isinstance(y, pd.DataFrame):
            y = y.values

        self.tree = self._construct_tree(np.array(X), np.array(y).reshape(-1), 0)

    def _construct_tree(self, points, y, depth):
        if len(points) == 0:
            return None

        k = len(points[0])
        axis = depth % k

        sort_by_axis = np.argsort(points[:, axis])
        sorted_points = points[sort_by_axis]
        sorted_y =  y[sort_by_axis]
        mid = len(sorted_points) // 2

        return KDNode(
            sorted_points[mid],
            sorted_y[mid],
            self._construct_tree(sorted_points[:mid], sorted_y[:mid], depth + 1),
            self._construct_tree(sorted_points[mid + 1:],sorted_y[mid + 1:], depth + 1)
        )

    def nearest_neighbour_search(self, query_point):
        k = len(query_point)
        heap = []

        def search(node, depth):
            if node == None:
                return

            nonlocal heap

            d = np.linalg.norm(query_point - node.points, ord = self.p)
            node.distance = -d

            if len(heap) < self.k:
                heapq.heappush(heap, node)
            else:
                heapq.heappushpop(heap, node)

            axis = depth % k

            if query_point[axis] < node.points[axis]:
                close, other = node.left, node.right
            else:
                close, other = node.right, node.left

            search(close, depth + 1)

            delta = abs(query_point[axis] - node.points[axis])
            nearest = abs(heap[0].distance)
            isFull = len(heap) > self.k

            if len(heap) < self.k or delta < nearest:
                search(other, depth + 1)

        search(self.tree, 0)
        return heap

    def predict(self, X_test):
        results = []
        if isinstance(X_test, pd.DataFrame):
            X_test = X_test.values

        for test in X_test:
            result = self.nearest_neighbour_search(test)
            predict_values = [item.y for item in result]
            counter = Counter(predict_values)
            results.append(counter.most_common(1)[0][0])
        return results
```

i. Tree Construction

The tree construction follows the basic tree construction of a binary search tree. On every iteration, the algorithm will find the median of our sorted list of points at a certain axis. Then it will recursively

split the data by doing the same steps and changing the axis. The simplified process can be seen in the image below.



First Iteration | Second Iteration

As for the implementation, it can be seen in the image below

```python
def _construct_tree(self, points, y, depth):
        if len(points) == 0:
            return None

        k = len(points[0])
        axis = depth % k

        sort_by_axis = np.argsort(points[:, axis])
        sorted_points = points[sort_by_axis]
        sorted_y =  y[sort_by_axis]
        mid = len(sorted_points) // 2

        return KDNode(
            sorted_points[mid],
            sorted_y[mid],
            self._construct_tree(sorted_points[:mid], sorted_y[:mid], depth + 1),
            self._construct_tree(sorted_points[mid + 1:],sorted_y[mid + 1:], depth + 1)
        )
```

ii. k-Nearest Neighbor Search

The next concept of our KDTree is the searching method. As a new test point appears, the algorithm will traverse the tree recursively in a binary manner. For each visit, a Priority Queue will record the distance, thus storing all the best distance up until the lead node. After finding the leaf node, it will check whether there is a smaller distance. If it holds true, then the algorithm will backtrack to the correct points. The implementation for traversing the tree can be seen below.

```python
def nearest_neighbour_search(self, query_point):
        k = len(query_point)
        heap = []

        def search(node, depth):
            if node == None:
                return

            nonlocal heap

            d = np.linalg.norm(query_point - node.points, ord = self.p)
            node.distance = -d

            if len(heap) < self.k:
                heapq.heappush(heap, node)
            else:
                heapq.heappushpop(heap, node)

            axis = depth % k

            if query_point[axis] < node.points[axis]:
                close, other = node.left, node.right
            else:
                close, other = node.right, node.left

            search(close, depth + 1)

            delta = abs(query_point[axis] - node.points[axis])
            nearest = abs(heap[0].distance)
            isFull = len(heap) > self.k

            if len(heap) < self.k or delta < nearest:
                search(other, depth + 1)

        search(self.tree, 0)
        return heap
```

D. Evaluation

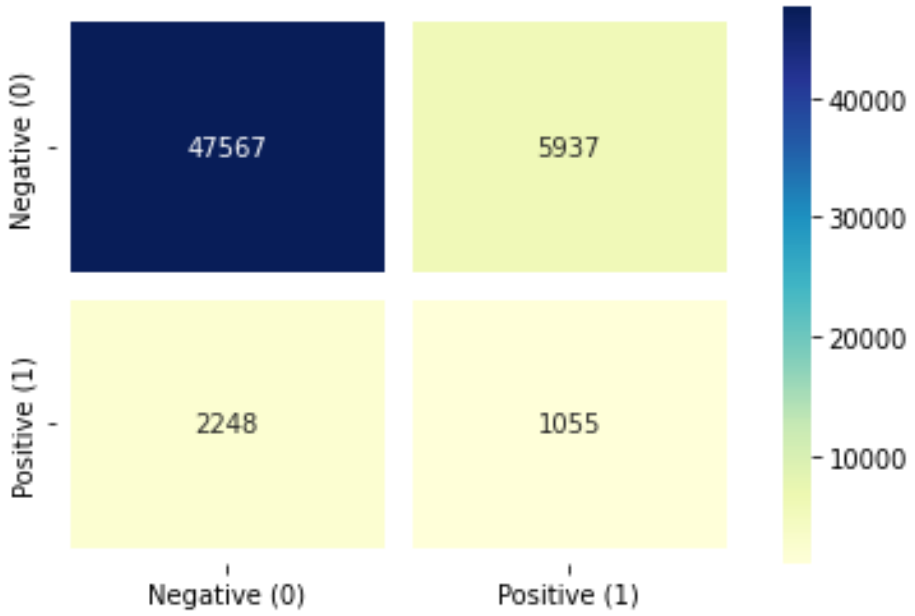
1. Creating k-NN Model

We create k-NN Model with K = 5 and p = 2 (Euclidean Distance). Here we pass in our training and test data which are 'X_train', and 'y_train'. In our case, 'y_pred' is the result of our predictions.

```
[ ]  knn = KDTree(k = 5, p = 2)
     knn.fit(X_train, y_train)
     y_pred = knn.predict(X_test)
```

2. Default Result

We can see from the image below that, the metrics has a high accuracy. But somehow, the F1 Score is quite small. Since F1 score is designed work well on imbalanced data, we trying to reach the best F1 Score as much as possible.




```
Accuracy with K-NN: 85.59%
Precision with K-NN: 15.09%
Recall with K-NN: 31.94%
F1-Score with K-NN: 0.20
```

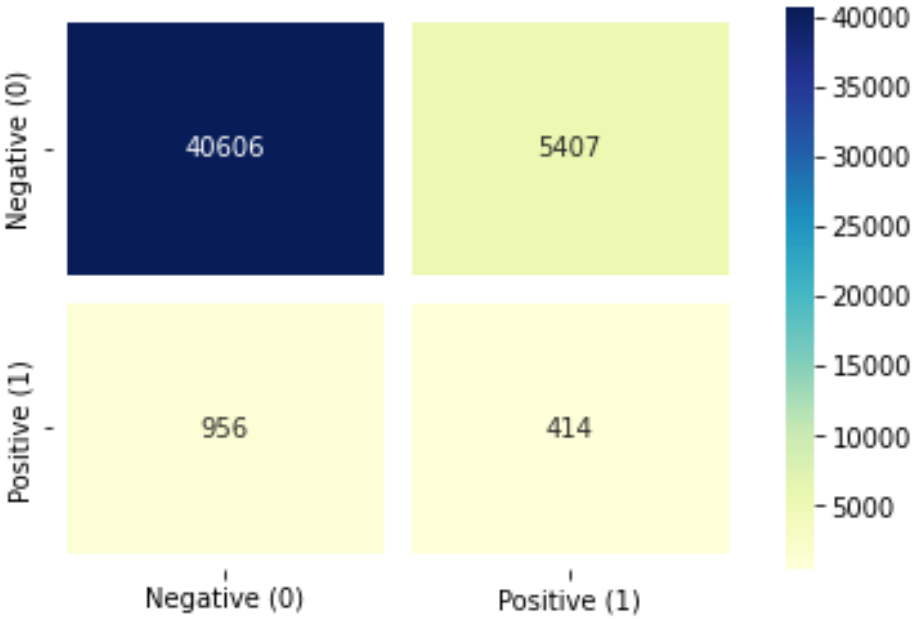
3. **Validation Test**

   With validation test, we change the data frame of our test data with principal_test (data test that has gone through the PCA process).

```
[ ]  y_pred_test = knn.predict(principal_test)
```

4. **Validation Result**

   We can see after validation test is done; F1 Score is still not high enough. This can be affected by imbalanced data.



```
Accuracy with K-NN: 86.57%
Precision with K-NN: 7.11%
Recall with K-NN: 30.22%
F1-Score with K-NN: 0.12
```

E. Experiment

In this experiment, we will try the pre-processing step. We know that our original data is imbalance, which we will try to oversample and undersample the dataset to balance it. We will also try to check the k values and check the accuracy.

1. Oversampling and Undersampling

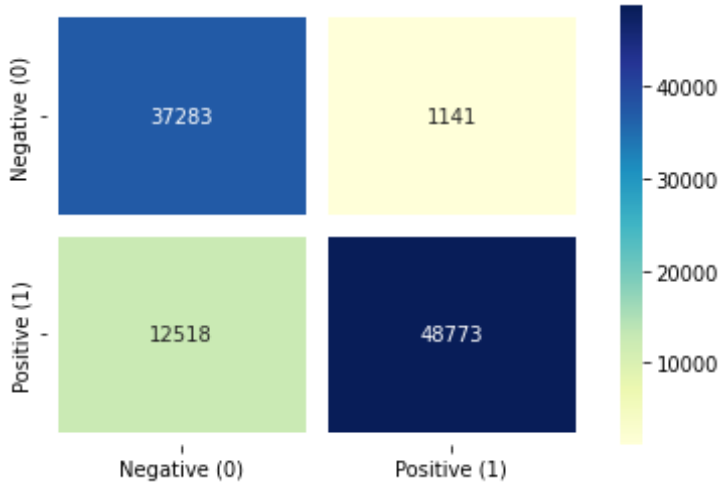
Imbalanced data refers to those types of datasets where the target class has an uneven distribution of observations. Example of a classification problem where the distribution of examples across the known classes is biased or skewed. This can cause a poor predictive performance. We trying to figure it out with oversampling and undersampling.

i. Oversampling

Oversampling is duplicating samples from the minority class. We call RandomOverSampler library into our case.

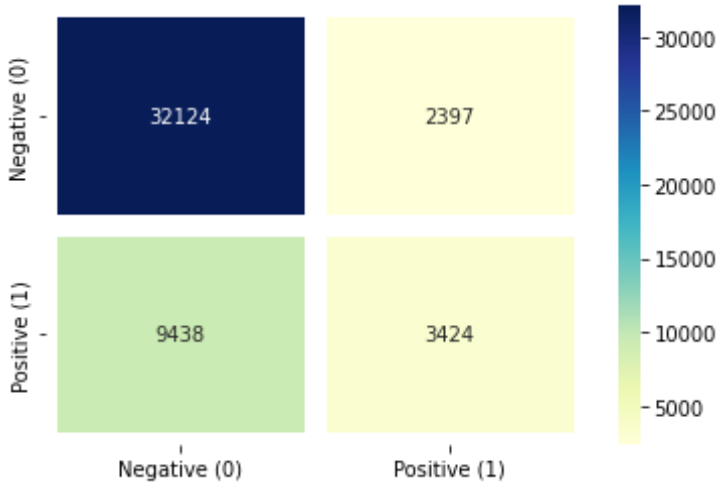
- **Original Oversampling**

  With similar K and P or distance, we got:




```
Accuracy with K-NN: 86.30%
Precision with K-NN: 97.71%
Recall with K-NN: 79.58%
F1-Score with K-NN: 0.88
```

- **Validation Test Oversampling**



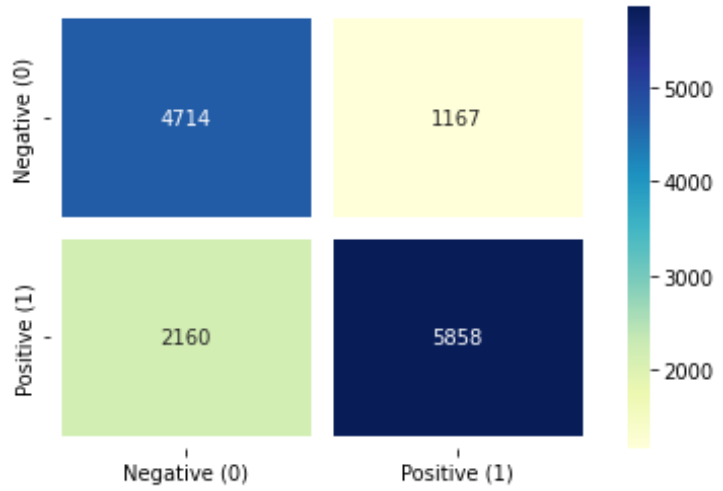

```
Accuracy with K-NN: 75.02%
Precision with K-NN: 58.82%
Recall with K-NN: 26.62%
F1-Score with K-NN: 0.37
```

ii. Undersampling

Undersampling is duplicating samples from the majority class. We call RandomUnderSampler library into our case.

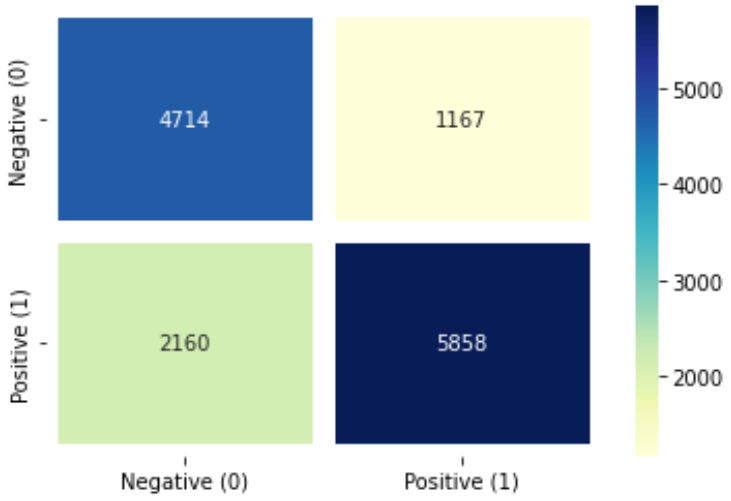
- **Original Undersampling**

  With similar K and P or distance, we got:




```
Accuracy with K-NN: 76.06%
Precision with K-NN: 83.39%
Recall with K-NN: 73.06%
F1-Score with K-NN: 0.78
```

- **Validation Test Undersampling**



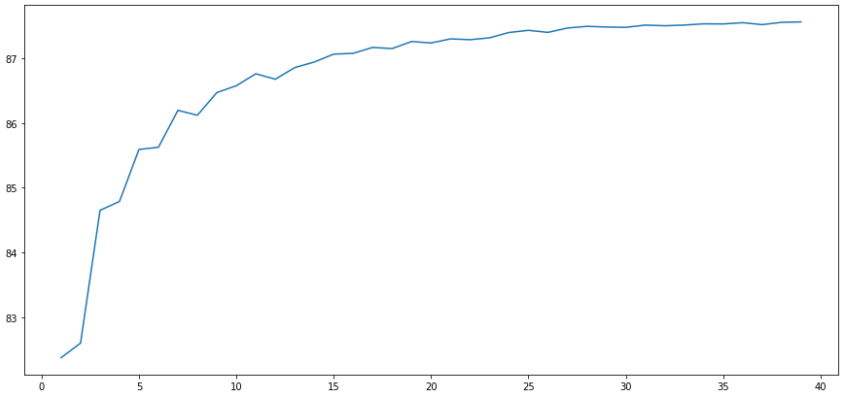

```
Accuracy with K-NN: 72.94%
Precision with K-NN: 75.38%
Recall with K-NN: 27.81%
F1-Score with K-NN: 0.41
```

2. **Accuracy Score vs K-Value**

   In this stage, we tried to figure out the trend between accuracy score and K-Value.

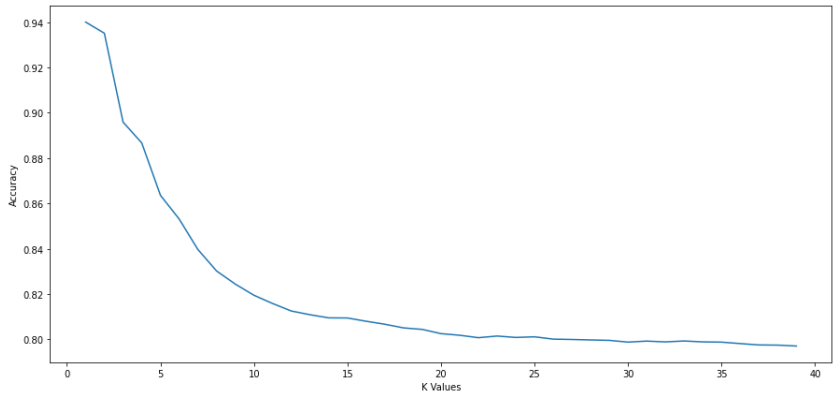
   i. **Original**

   We can see from original /without over and under sampling, the larger K makes larger accuracy. This is expected since our original dataset is populated more with the '0' label. When the k is increased, it would discover more '1' which gradually improve the accuracy.



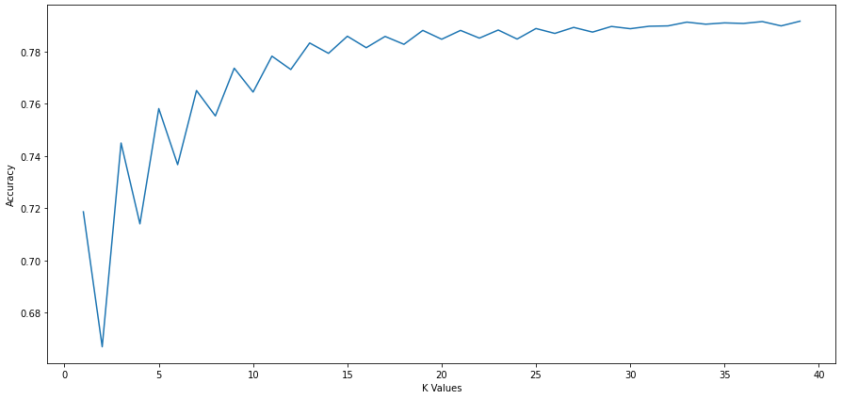
   ii. **Oversampling**

   This is trend after oversampling. We can see that the larger K makes accuracy is decreasing. This happens because our dataset is balance. When increasing the k values, it will found more the wrong neighbor value, thus decreasing the accuracy.



iii. Undersampling

This is trend after undersampling. We can see that the larger K, the larger also the accuracy, but not as much as the original data. We can also clearly see that the trend forms spikes.



F. Conclusion

In this assignment, we are given the task to do classification using machine learning. Therefore, we use K-Nearest Neighbor for our model with KDTree implementation. This reduces the time taken when doing the search compared to brute force. We also found out that our original data is not balance which means that there are more ‘Non-Tertarik’ class than the ‘Tertarik’ which result our model to perform poorly with the high-accuracy but low F1-score. To improve our model, we pre-process our dataset to balance the label class. Our method to balance the dataset is by using oversampling and undersampling. We found that model that train using the oversampled dataset performs better than the original and the undersampling one. In conclusion, our best model is the one with the oversampled dataset and K of 5. The result of our evaluation is that our model has a F1-score of 0.78 and accuracy of 72.06% when predicting the test data.