Tugas Kecil Strategi Algoritma

Laporan Implementasi *Convex Hull* untuk Visualisasi Tes *Linear Separability Dataset* dengan Algoritma *Divide and Conquer*

Oleh:

David Karel Halomoan

13520154

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2022

A. Algoritma *Divide and Conquer*

Algoritma *divide and conquer* adalah algoritma yang membagi persoalan menjadi beberapa upa-persoalan yang memiliki kemiripan dengan persoalan semula namun berukuran lebih kecil (idealnya berukuran hampir sama) (*divide*), algoritma lalu menyelesaikan masing-masing upa-persoalan (secara langsung jika sudah berukuran kecil atau secara rekursif jika masih berukuran besar) (*conquer*), algoritma lalu mengabungkan solusi masing-masing upa-persoalan sehingga membentuk solusi persoalan semula (*combine*). Algoritma ini disebut juga dengan algoritma naif (*naïve algorithm*) karena algoritma ini biasanya didasarkan langsung pada pernyataan pada persoalan (*problem statement*) dan definisi/konsep yang dilibatkan.

Pada tugas kecil ini, penulis mendapatkan tugas untuk mengimplementasikan Implementasi *convex hull* untuk Visualisasi Tes Linear *separability dataset* dengan bahasa pemrograman Python. Himpunan titik pada bidang planar disebut convex jika untuk sembarang dua titik pada bidang tersebut (misal p dan q), seluruh segmen garis yang berakhir di p dan q berada pada himpunan tersebut.

Penulis memutuskan untuk melakukan implementasi *convex hull* dengan fitur *class* pada bahasa pemrograman Python. Hal ini dialkukan karena penulis mendapatkan tugas untuk membuat pustaka baru yang menggantikan Pustaka ConvexHull pada *package scipy.spatial*. Pustaka tersebut berupa *class* sehingga penulis juga membuat pustaka yang berupa *class*. Pustaka penulis juga mengembalikan objek (*instance*) dari sebuah kelas buatan pengguna, yaitu kelas *ReturnClass*. Hal ini dilakukan karena pada contoh pada spesifikasi tugas kecil, kode mengakses properti / atribut *simplices* sehingga Pustaka harus mengembalikan objek (*instance*) dari kelas yang memiliki properti/atribut *simplices*.

Selanjutnya akan dijelaskan langkah-langkah pada algoritma yang penulis buat. Pertama-tama, program menerima masukan kumpulan titik (*points*) yang disebut *bucket*. Titik disini adalah sebuah *tuple* yang memiliki 2 nilai, yaitu *x* (absis) dan *y* (ordinat). *Tuple* diimplementasikan menggunakan larik (*array*) yang memiliki 2 elemen. Program lalu membuat *Hash Map* (*dictionary* dalam bahasa pemrograman Python) yang memiliki *key* representasi *string* dari salah satu titik dan *value* indeks dari titik tersebut dalam *bucket*. Hal ini dilakukan untuk mengembalikan indeks dari setiap titik yang termasuk dalam

convex hull yang berhubungan (sesuai Pustaka *ConvexHull* pada *package scipy.spatial*). Program lalu mencari 2 titik, yaitu p1 dan p2, kedua titik merupakan dua titik ekstrim jika *bucket* diurutkan (*sort*) berdasarkan nilai absis menaik, dan jika ada nilai absis yang sama, diurutkan dengan nilai ordinat yang menaik. Hanya saja program tidak melakukan pengurutan *bucket*, tetapi melakukan traversal pada *bucket* untuk menemukan 2 titik tersebut, sehingga bagian ini memliki notasi Big O *O(n)*, bukan *O(n log n*) seperti pada algoritma pengurutan *(Quicksort*). p1 dan p2 adalah dua titik yang akan membentuk *convex hull* untuk kumpulan titik tersebut. Selanjutnya, program akan membagi *bucket* menjadi 2, yaitu titik-titik yang berada di atas garis yang dibentuk p1 dan p2 dan titik-titik yang berada di bawah garis yang dibentuk p1 dan p2. Pembagian ini dilakukan menggunakan rumus:

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3$$

Gambar 1 Rumus Penentuan Posisi Suatu Titik Relatif terhadap Garis yang Dibentuk oleh 2 Titik Lain

Pada rumus tersebut, *x1* dan *y1* merupakan *tuple* dari p1, *x2* dan *y2* merupakan *tuple* dari p2, dan *x3* dan *y3* merupakan *tuple* titik yang akan ditentukan posisinya relatif terhadap garis yang dibentuk p1 dan p2. Jika hasil rumus tersebut positif (> 0), *tuple x3* dan *y3* berada di atas garis, sedangkan jika hasil rumus tersebut negative (<0), *tuple x3* dan *y3* berada di bawah garis. Penulis melakukan pendekatan pada *source code* dengan menggunakan nilai 0.00001 (untuk yang di atas garis) dan -0.0001 (untuk yang di bawah garis), hal ini dilakukan karena saat menggunakan nilai 0, terdapat masalah saat melakukan pembulatan untuk determinan kecil sehingga rekursi berlangsung secara tak terbatas (*infinite loop*). Semua titik yang berada pada garis p1 p2 (selain p1 dan p2) tidak mungkin membentuk *convex hull*, sehingga bisa diabaikan dari pemeriksaan (tidak masuk rekursi berikutnya). Program lalu melakukan rekursi pertama, program mencari *convex hull* yang berada di atas dan di bawah garis yang dibentuk oleh p1 dan p2. Jika di atas atau di bawah garis tidak ada titik lain, p1 dan p2 menjadi pembentuk *convex hull* di atas atau bawah garis tersebut (p1 dan p2 dihubungkan). Jika di atas atau di bawah garis hanya terdapat satu titik lain, missal kita sebut p3, p1 dan p3 dan p3 dan p2 menjadi pembentuk *convex hull* di bagian tersebut (p1 dan p3 dihubungkan dan p3 dan p2 dihubungkan). Jika

terdapat lebih dari satu titik lain pada bagian bawah atau atas garis, pilih sebuah titik yang memiliki jarak terbesar dari garis yang dibentuk p1 dan p2, titik ini disebut pmax. Jika terdapat lebih dari satu titik yang memiliki jarak terbesar, pilih sebuah titik yang memaksimalkan sudut <pmax p1 p2> (p1 sebagai sudut). Program mendapatkan jarak titik dari garis menggunakan *shoelace algorithm* dan rumus luas segitiga. *Shoelace algorithm* adalah algoritma atau rumus untuk mendapatkan luas poligon dengan sisi *n*. *Shoelace algorithm* digunakan untuk mencari luas segitiga (poligon dengan sisi 3), yaitu luas segitiga yang dibentuk oleh sudut p1, p2, dan pmax. Berikut rumus *shoelace algorithm*:

$$1/2\left(\left|\Sigma_{i=1}^{\mathrm{n}-1}(x_i y_{i+1}) + x_n y_1 - \Sigma_{i=1}^{n-1}(x_{i+1} y_i) - x_1 y_n\right|\right)$$

Karena algoritma di atas hanya berupa rumus matematika, *shoelace algorithm* hanya memiliki notasi Big O *O(1)*. Segitiga tersebut dianggap memiliki alas garis yang dibentuk p1 dan p2, sehingga untuk mendapatkan jarak pmax ke garis, hanya diperlukan menghitung tinggi segitiga (tinggi segitiga pasti tegak lurus terhadap garis p1 p2 dan melewati pmax, sehingga tinggi sehitiga adalah jarak pmax ke garis p1 p2).

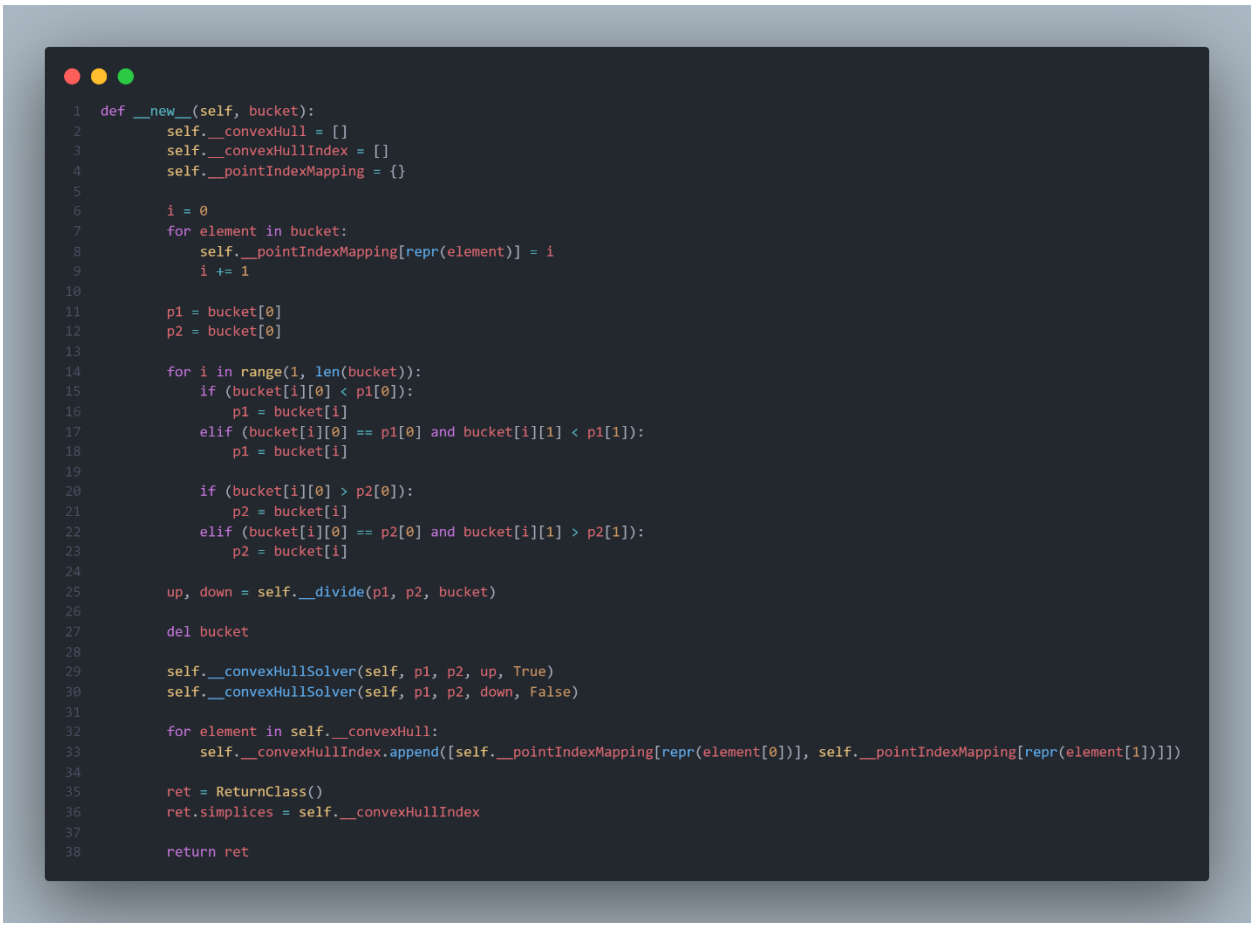
$$Luas\ Segitiga\ =\ 1/2\ \times Alas\ Segitiga \times Tinggi\ Segitiga$$

$$TinggiSegitiga = (2 \times LuasSegitiga)/AlasSegitiga$$

Program lalu melakukan rekursi dengan p1 dan pmax serta pmax dan p2 sebagai p1 dan p2 yang baru. Rekursi dilakukan sampai tidak ada titik yang berada di bawah atau atas garis yang dibentuk oleh p1 dan p2 setiap kali rekursi. Saat melakukan rekursi, program mengabaikan titik yang berada di sisi yang tidak sesuai dari garis (jika saat pembagian (*divide*), program mengambil titik-tit yang berada di atas garis, program akan terus mengambil titik yang berada di atas garis saat melakukan rekursi, dan sebaliknya).

B. *Source Code* Program Dalam Bahasa Python

1. Konstruktor Kelas MyConvexHull

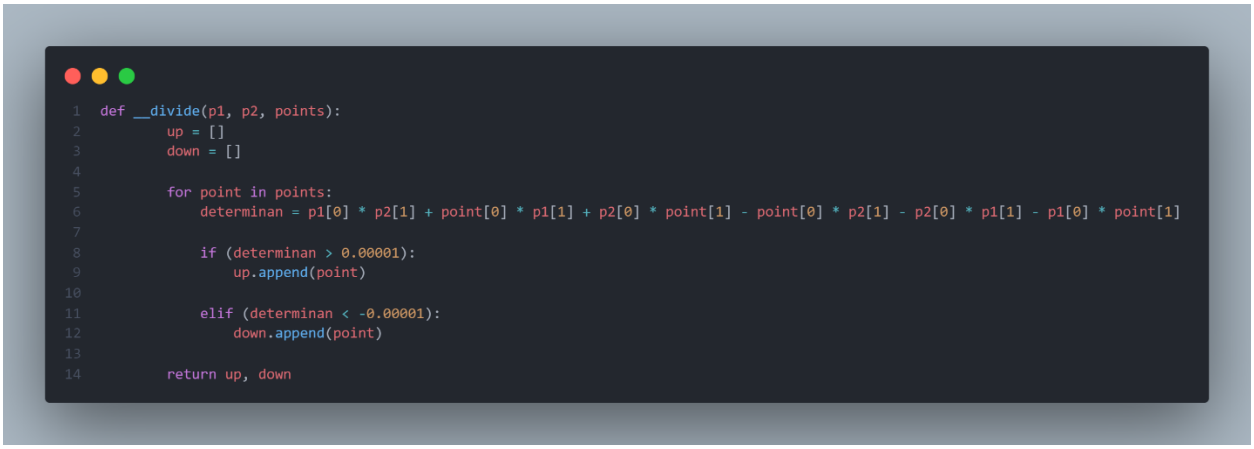


```
def __new__(self, bucket):
        self.__convexHull = []
        self.__convexHullIndex = []
        self.__pointIndexMapping = {}

        i = 0
        for element in bucket:
            self.__pointIndexMapping[repr(element)] = i
            i += 1

        p1 = bucket[0]
        p2 = bucket[0]

        for i in range(1, len(bucket)):
            if (bucket[i][0] < p1[0]):
                p1 = bucket[i]
            elif (bucket[i][0] == p1[0] and bucket[i][1] < p1[1]):
                p1 = bucket[i]

            if (bucket[i][0] > p2[0]):
                p2 = bucket[i]
            elif (bucket[i][0] == p2[0] and bucket[i][1] > p2[1]):
                p2 = bucket[i]

        up, down = self.__divide(p1, p2, bucket)

        del bucket

        self.__convexHullSolver(self, p1, p2, up, True)
        self.__convexHullSolver(self, p1, p2, down, False)

        for element in self.__convexHull:
            self.__convexHullIndex.append([self.__pointIndexMapping[repr(element[0])], self.__pointIndexMapping[repr(element[1])]])

        ret = ReturnClass()
        ret.simplices = self.__convexHullIndex

        return ret
```

2. **Fungsi convexHullSolver (fungsi yang dipanggil secara rekursif)**

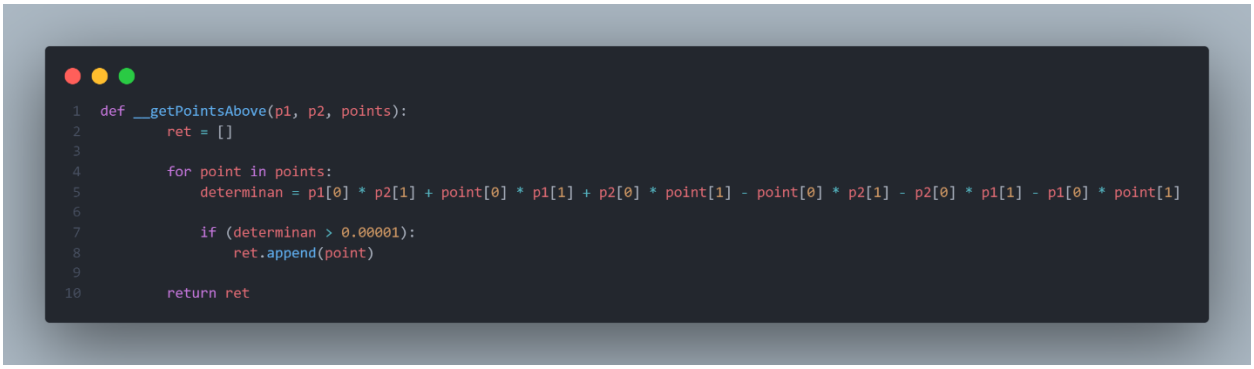
```python
def __convexHullSolver(self, p1, p2, points, upSide):
        if (len(points) == 0):
            self.__convexHull.append([p1, p2])

        elif (len(points) == 1):
            self.__convexHull.append([p1, points[0]])
            self.__convexHull.append([points[0], p2])

        else:
            maxPoints = [points[0]]
            maxDistance = self.__distanceOfPointToLine(self, points[0], p1, p2)

            for i in range(1, len(points)):
                distance = self.__distanceOfPointToLine(self, points[i], p1, p2)

                if (distance > maxDistance):
                    maxPoints = [points[i]]
                    maxDistance = distance

                elif (distance == maxDistance):
                    maxPoints.append(points[i])

            maxPoint = []
            if (len(maxPoints) == 1):
                maxPoint = maxPoints[0]

            else: # Sudah dipastikan len(maxPoints) > 1
                maxAngle = self.__angleOfThreePoints(self, maxPoints[0], p2, p1)
                maxPointIndex = 0

                for i in range(1, len(maxPoints) - 1):
                    currentAngle = self.__angleOfThreePoints(self, maxPoints[i], p1, p2)

                    if (currentAngle > maxAngle):
                        maxAngle = currentAngle
                        maxPointIndex = i

                maxPoint = maxPoints[maxPointIndex]

            if (upSide):
                self.__convexHullSolver(self, p1, maxPoint, self.__getPointsAbove(p1, maxPoint, points), upSide)
                self.__convexHullSolver(self, maxPoint, p2, self.__getPointsAbove(maxPoint, p2, points), upSide)

            else:
                self.__convexHullSolver(self, p1, maxPoint, self.__getPointsBelow(p1, maxPoint, points), upSide)
                self.__convexHullSolver(self, maxPoint, p2, self.__getPointsBelow(maxPoint, p2, points), upSide)

        return
```

3. **Fungsi Divide (fungsi yang pertama kali membagi kumpulan titik menjadi 2 bagian (di atas dan di bawah garis)**



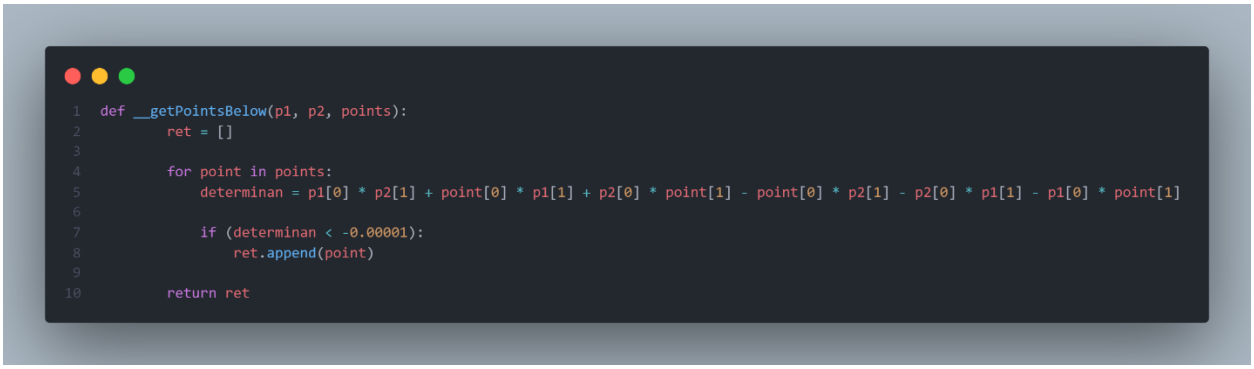
```
def __divide(p1, p2, points):
        up = []
        down = []

        for point in points:
            determinan = p1[0] * p2[1] + point[0] * p1[1] + p2[0] * point[1] - point[0] * p2[1] - p2[0] * p1[1] - p1[0] * point[1]

            if (determinan > 0.00001):
                up.append(point)

            elif (determinan < -0.00001):
                down.append(point)

        return up, down
```

4. **Fungsi getPointsAbove (fungsi yang mengembalikan titik yang berada di atas garis yang dibentuk p1 dan p2)**



```
def __getPointsAbove(p1, p2, points):
        ret = []

        for point in points:
            determinan = p1[0] * p2[1] + point[0] * p1[1] + p2[0] * point[1] - point[0] * p2[1] - p2[0] * p1[1] - p1[0] * point[1]

            if (determinan > 0.00001):
                ret.append(point)

        return ret
```

5. **Fungsi getPointsBelow (fungsi yang mengembalikan titik yang berada di bawah garis yang dibentuk p1 dan p2)**



```
def __getPointsBelow(p1, p2, points):
        ret = []

        for point in points:
            determinan = p1[0] * p2[1] + point[0] * p1[1] + p2[0] * point[1] - point[0] * p2[1] - p2[0] * p1[1] - p1[0] * point[1]

            if (determinan < -0.00001):
                ret.append(point)

        return ret
```

6. **Fungsi lengthBetweenPoints (mengembalikan jarak antara dua titik)**

```
def __lengthBetweenPoints(p1, p2):
        return ((p2[0] - p1[0]) ** 2 + (p2[1] - p1[1]) ** 2) ** 0.5
```

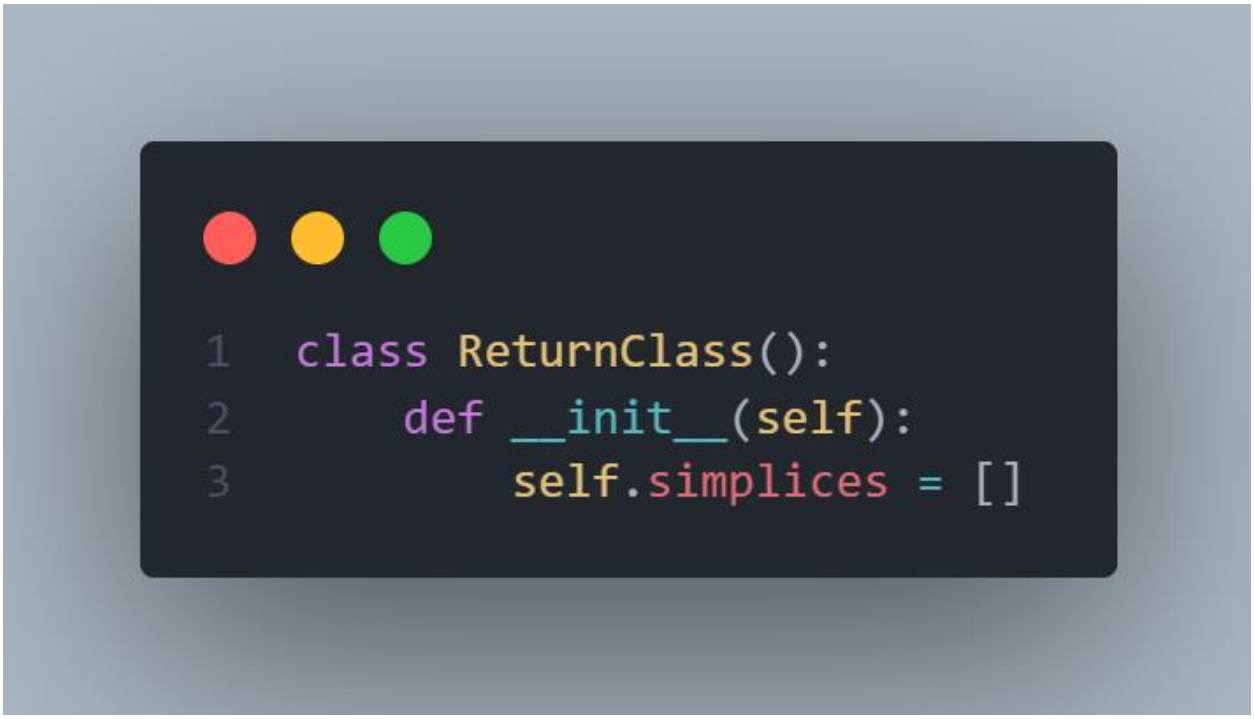
7. **Fungsi distanceOfPointToLine (mengembalikan jarak point ke garis yang dibentuk oleh p1 dan p2, fungsi ini menggunakan *shoelace algorithm* dan rumus luas segitiga)**

```
def __distanceOfPointToLine(self, point, p1, p2):
        # Menguunakan Shoelace Algorithm
        # x1 y2 + x2 y3 + x3 y1 - x2 y1 - x3 y2 - x1 y3
        area = 0.5 * abs(p1[0] * p2[1] + p2[0] * point[1] + point[0] * p1[1] - p2[0] * p1[1] - point[0] * p2[1] - p1[0] * point[1])

        return 2 * area / self.__lengthBetweenPoints(p1, p2)
```

8. **Fungsi angleOfThreePoints (mengembalikan besar susut yang dibentuk oleh p1, p2, dan corner (corner adalah titik sudut) dalam radian, menggunakan rumus cosinus)**

```
def __angleOfThreePoints(self, p1, p2, corner):
        a = self.__lengthBetweenPoints(p1, corner)
        b = self.__lengthBetweenPoints(p2, corner)
        c = self.__lengthBetweenPoints(p1, p2)

        # Rumus Cosinus
        cosOfAngle = ((a * a) + (b * b) - (c * c)) / (2 * a * b)

        return acos(cosOfAngle)
```

9. **Kelas ReturnClass**

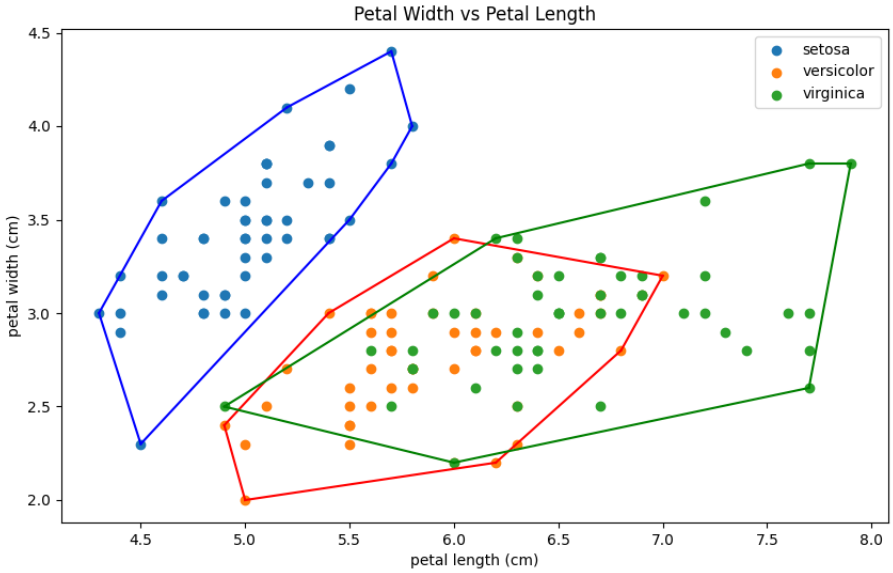


C. *Screenshot* dari *Input* dan *Output* Program

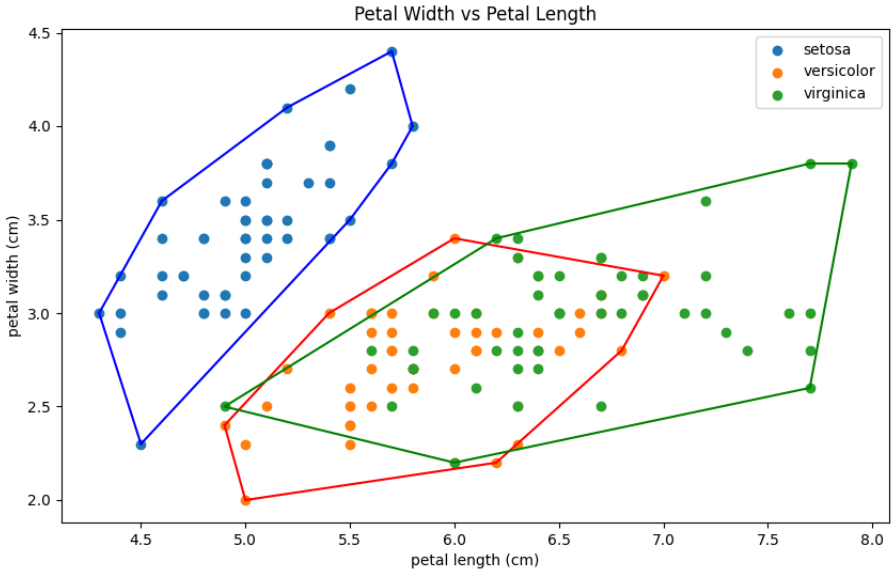
- **Dataset iris**

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	Target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

  - **Atribut: sepal Width vs Sepal Length**



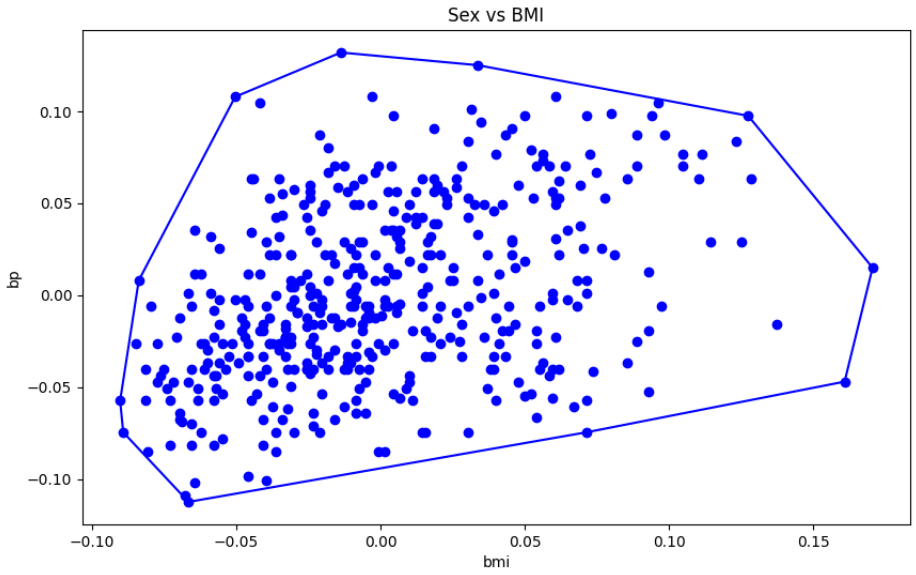
  - **Atribut: Sepal Width vs Sepal Length**



- **Dataset diabetes**

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6	Target
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019908	-0.017646	151.0
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068330	-0.092204	75.0
2	0.085299	0.050680	0.044451	-0.005671	-0.045599	-0.034194	-0.032356	-0.002592	0.002864	-0.025930	141.0
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022692	-0.009362	206.0
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031991	-0.046641	135.0

  - **Atribut: Sex vs BMI**



Alamat Drive

a. **Google Drive**
   https://drive.google.com/drive/folders/12cbIYZEywHLJxagOaJakObRrL8FOjfX2?usp=sharing
b. **Github *Repository***

   https://github.com/davidkarelh/davidkarelh-Tugas-Kecil-2-IF2211-Strategi-Algoritma-Implementasi-Convex-Hull-untuk-Visualisasi-Tes-L

D.Tabel Ceklist

Poin	**Ya**	**Tidak**
Pustaka *myConvexHull* berhasil dibuat dan tidak ada kesalahan	√	
Convex hull yang dihasilkan sudah benar	√	
Pustaka *myConvexHull* dapat digunakan untuk menampilkan *convex hull* setiap label dengan warna yang berbeda.	√	
Bonus: program dapat menerima input dan menuliskan output untuk dataset lainnya.	√	