In [1]:
```
from abc import ABC, abstractmethod
import numpy as np
import time
import pandas as pd
import matplotlib.pyplot as plt
```

Poniżej klasa abstrakcyjna dla liniowych klasyfikatorów. Zawiera w sobie metodę abstrakcyjną fit, implementowaną na swój sposób przez dziedziczące po klasie bazowej inne klasy. Argumenty przyjmowane przez konstruktor klasy:

- coef_ - wektor wag `w` z nauczania klasyfikatora
- interceptr_ - wyraz wolny `b` nauczania klasyfikatora
- class_labels - unikalne klasy

In [2]:
```
class LinearClassifier(ABC):
    def __init__(self, coef=None, intercept=None, class_labels=None):
        self.coef_ = coef
        self.intercept_ = intercept
        self.class_labels_ = class_labels

    @abstractmethod
    def fit(self, x, d):
        ...

    def margin(self, x, y):
        margin = 1 / np.sqrt(np.sum(self.coef_ ** 2))
        down = y - np.sqrt(1 + x ** 2) * margin
        up = y + np.sqrt(1 + x ** 2) * margin

        return down, up

    def decision_function(self, x: np.array):
        return self.margin(x, np.ones(x.shape[0]))

    def predict_proba(self, x: np.array):
        a, b = self.margin(x, np.ones((x.shape[0], )))
        i = 1 - 1 / (1 + np.exp(-b))
        j = 1 / (1 + np.exp(-b))

        return np.array([i, j]).T

    def predict(self, x: np.array):
        results = np.sign(x.dot(self.coef_) + self.intercept_)
        results_mapped = self.class_labels_[1 * (results > 0)]

        return results_mapped

    def get_params(self, deep=True):
        pass

    def set_params(self, **parameters):
        pass

    def __str__(self):
        return f'{self.__class__.__name__}[w={self.coef_}, b={self.intercept_}]'
```

Klasa perceptronu prostego dziedzicząca po abstrakcyjnej klasie bazowej `LinearClassifier`. Implementuje ona metodę `fit`, która odpowiada za uczenie

klasyfikatora. Po przekroczeniu danego czasu, działanie metody jest przerywane. Liczba iteracji jest inkrementowana w momencie zmiany wag.

In [15]:
```
class Perceptron(LinearClassifier):
    def __init__(self, gamma: float = 0.0, max_seconds: int = 3600, **kwargs):
        super().__init__(**kwargs)

        self.gamma = gamma
        self.max_seconds = max_seconds

        self.iteration_count = 0

    def fit(self, x: np.array, d: np.array):
        self.class_labels_ = np.unique(d)

        w, b = np.ones(x.shape[1]), 0
        n = 0
        t1 = time.time()

        while n < x.shape[0]:
            t2 = time.time() - t1
            if t2 > self.max_seconds:
                print(f'Max time reached out: {t2}s! Break algorithm')
                break

            for i in range(x.shape[0]):
                if d[i] * (x[i, :].dot(w) + b) > 0:
                    n += 1
                    continue

                w += d[i] * x[i]
                b += d[i]
                n = 0

                self.iteration_count += 1

        self.coef_ = w
        self.intercept_ = b

        return w, b
```

Metody generujące zbiory danych z decyzjami `[-1 1]`. Pierwszy z nich generuje zbiór danych separowalny liniowo. Drugi generuje nieseparowalny liniowo zbiór danych.

In [4]:
```
def linear_separable_dataset():
    w, b = [1, 1], -1
    x = np.random.randn(100, 2)
    d = np.sign(x.dot(w) + b)

    return x, d


def non_linear_separable_dataset():
    w, b = [1, 1], -1
    x = np.random.randn(100, 2)

    d = np.random.rand(x.shape[0])
    d[d < 0.5] = -1
    d[d >= 0.5] = 1

    return x, d
```

Funkcja zamieniająca klasy decyzyjne na ich postacie zunifikowane, czyli `[-1 1]`.

In [5]:
```
def normalize_decisions(d):
    d_normalized = np.ones(d.shape[0]).astype("int8")
    d_normalized[d == np.unique(d)[0]] = -1

    return d_normalized
```

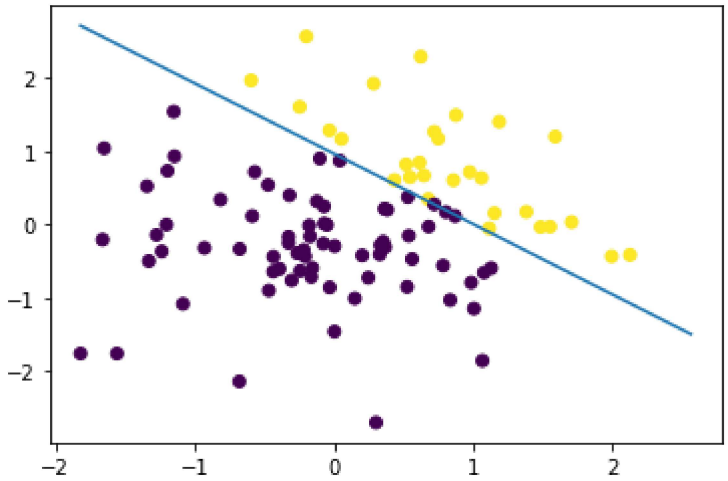
Funkcja przeprowadzająca eksperyment. Tworzy obiekt klasy perceptronu prostego i dokonuje jego nauczania na bazie dostarczonego zbioru danych wraz z decyzjami. Na ekran jest wypisywany czas nauczania oraz liczba iteracji. Następnie wyrysowywany jest zbiór danych wraz z wyliczoną linią separującą próbki.

In [6]:
```
def experiment(x, d):
    perceptron = Perceptron()

    t1 = time.time()
    w, b = perceptron.fit(x, d)
    t2 = time.time()
    print(f'Time of fitting: {t2 - t1}s.\nNumber of iterations: {perceptron.iterat

    plt.figure()
    plt.scatter(x[:, 0], x[:, 1], c=d)

    x1 = np.array([np.min(x[:, 0]), np.max(x[:, 1])])
    x2 = -(b + w[0] * x1) / w[1]
    plt.plot(x1, x2)

    plt.show()
```

Eksperyment dla danych separowalnych liniowo:

In [7]:
```
x_data, decisions = linear_separable_dataset()
experiment(x_data, decisions)
```

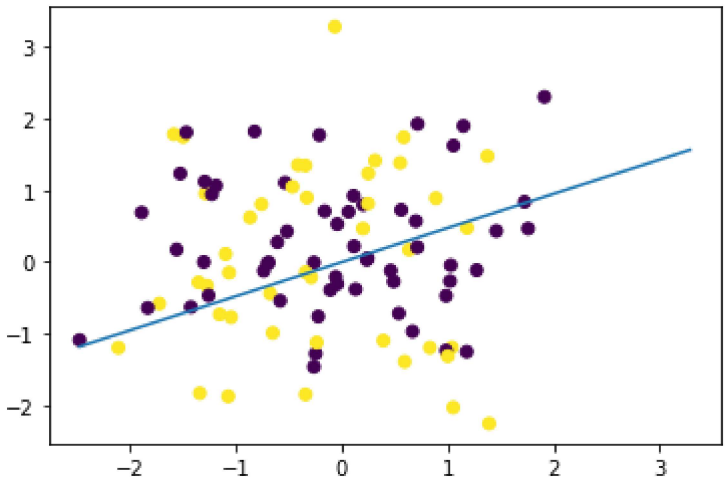
```
Time of fitting: 0.0s.
Number of iterations: 6
```



Eksperyment dla danych nieseparowalnych liniowo:

In [8]:
```
x_data, decisions = non_linear_separable_dataset()
experiment(x_data, decisions)
```

```
Max time reached out: 3600.0002562999725s! Break algorithm
Time of fitting: 3600.0002562999725s.
Number of iterations: 903014978
```



Eksperyment dla zbioru sonar:

In [14]:
```
sonar_data = pd.read_csv('D:\Programming\Python\computational-intelligence\machine
decisions = sonar_data[sonar_data.columns[-1]]
decisions = normalize_decisions(decisions)
x_data = sonar_data.drop(sonar_data.columns[-1], axis=1).to_numpy()

experiment(x_data, decisions)
```

```
Time of fitting: 165.46740746498108s.
Number of iterations: 2735649
```