IMPLEMENTASI ALGORITMA ADABOOST DARI SCRATCH
(Course Advanced Machine Learning)

I. Component of Learning

- Fitting

Dalam *boosting*, *cost function* yang ingin diminimalkan adalah *error*. Untuk permasalahan klasifikasi 2 kelas, algoritma *boosting* 'AdaBoost' setara dengan meminimalkan *cost-function*:

$$C = \sum_{\mu} \exp(-t^{\mu}\Phi^{\mu}/2)$$

dimana μ merupakan indeks dari item-item pelatihan, t ∈ {±1} merupakan target dan Φ adalah jumlah bobot dari klasifikasi individu ke dalam klasifikasi komposit (model dari beberapa klasifikasi individu yang bekerja sama) untuk menghasilkan *output* +1, jika Φ > 0 dan -1 jika sebaliknya. Pada algoritma AdaBoost, komponen klasifikasi dilatih untuk meminimalkan jumlah kesalahan melalui sebuah distribusi *input* [1].

Cost function pada AdaBoost adalah *exponential loss function*. AdaBoost dapat dikatakan sebagai optimasi *greedy* dari *loss function* tertentu. Didefinisikan $f(\mathbf{x}) = \frac{1}{2}\sum_m \alpha_m f_m(\mathbf{x})$ dan klasifikasi sebagai *g(x)* = *sign* $(f(\mathbf{x}))$ (faktor dari ½ tidak memiliki pengaruh pada output klasifikasi) [2]. Sehingga AdaBoost dapat mengoptimalkan *exponential loss* :

$$L_{exp}(\mathbf{x}, y) = e^{-yf(\mathbf{x})}$$

maka dari itu, pembelajaran penuh pada *objective function*, dengan data pelatihan yang diberikan $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ adalah :

$$E = \sum_i e^{-\frac{1}{2}y_i \sum_{m=1}^M \alpha_m f_m(\mathbf{x}_i)}$$

dimana yang harus dioptimalkan berhubungan dengan bobot α dan *weak classifier*. Proses optimasi bersifat *greedy* dan *sequential* : tambahkan 1 *weak classifier* pada satu waktu, kemudian pilih, dan α akan menjadi optimal sehubungan dengan E yang tidak akan akan berubah lagi. *Exponential loss* adalah batas atas dari kerugian 0-1 :

$$L_{exp}(\mathbf{x}, y) \geq L_{0-1}(\mathbf{x}, y)$$

oleh karena itu, jika *exponential loss* nol tercapai, maka kerugian 0-1 juga adalah nol dan semua pelatihan akan diklasifikasikan dengan benar. Pertimbangkan *weak classifier* f_m ditambahkan pada langkah m. Keseluruhan *objective function* dapat ditulis untuk memisahkan kontribusi dari klasifikasi ini :

$$\begin{aligned} E &= \sum_i e^{-\frac{1}{2}y_i \sum_{j=1}^{m-1} \alpha_j f_j(\mathbf{x}_i) - \frac{1}{2}y_i \alpha_m f_m(\mathbf{x}_i)} \\ &= \sum_i e^{-\frac{1}{2}y_i \sum_{j=1}^{m-1} \alpha_j f_j(\mathbf{x}_i)} \; e^{-\frac{1}{2}y_i \alpha_m f_m(\mathbf{x}_i)} \end{aligned}$$

karena terdapat suku pertama $m-1$ yang konstan, maka dapat diganti dengan konstanta tunggal $w_i^{(m)} = e^{-\frac{1}{2}y_i \sum_{j=1}^{m-1} \alpha_j f_j(\mathbf{x}_i)}$. Perhatikan bahwa ini adalah bobot yang sama yang dihitung secara rekursi yang digunakan oleh AdaBoost yaitu $w_i^{(m)} \propto w_i^{(m-1)} e^{-\frac{1}{2}y_i \alpha_j f_{m-1}(\mathbf{x}_i)}$ (terdapat konstansta proporsionalitas yang bisa diabaikan), maka :

$$E = \sum_i w_i^{(m)} e^{-\frac{1}{2}y_i \alpha_m f_m(\mathbf{x}_i)}$$

selanjutnya bagi menjadi dua penjumlahan yaitu untuk yang diklasifikasikan dengan benar oleh f_m dan data yang salah diklasifikasikan :

$$E = \sum_{i:f_m(\mathbf{x}_i)=y_i} w_i^{(m)} e^{-\frac{\alpha_m}{2}} + \sum_{i:f_m(\mathbf{x}_i)\neq y_i} w_i^{(m)} e^{\frac{\alpha_m}{2}}$$

susun kembali suku-sukunya menjadi :

$$E = (e^{\frac{\alpha_m}{2}} - e^{-\frac{\alpha_m}{2}}) \sum_i w_i^{(m)} I(f_m(\mathbf{x}_i) \neq y_i) + e^{-\frac{\alpha_m}{2}} \sum_i w_i^{(m)}$$

mengoptimalkan hal tersebut dengan f_m akan sama dengan melakukan optimasi $\sum_i w_i^{(m)} I(f_m(\mathbf{x}_i) \neq y_i)$, dimana hal tersebut merupakan hal yang dilakukan oleh AdaBoost. Nilai optimal α_m dapat diperoleh dengan menyelesaikan $\frac{dE}{d\alpha_m} = 0$:

$$\frac{dE}{d\alpha_m} = \frac{\alpha_m}{2}\left(e^{\frac{\alpha_m}{2}} + e^{-\frac{\alpha_m}{2}}\right) \sum_i w_i^{(m)} I(f_m(\mathbf{x}_i) \neq y_i) - \frac{\alpha_m}{2} e^{-\frac{\alpha_m}{2}} \sum_i w_i^{(m)} = 0$$

bagi kedua sisi dengan $\frac{\alpha_m}{2\sum_i w_i^{(m)}}$ maka :

$$\begin{aligned} 0 &= e^{\frac{\alpha_m}{2}} \epsilon_m + e^{-\frac{\alpha_m}{2}} \epsilon_m - e^{-\frac{\alpha_m}{2}} \\ e^{\frac{\alpha_m}{2}} \epsilon_m &= e^{-\frac{\alpha_m}{2}} (1 - \epsilon_m) \\ \frac{\alpha_m}{2} + \ln \epsilon_m &= -\frac{\alpha_m}{2} + \ln(1 - \epsilon_m) \\ \alpha_m &= \ln \frac{1 - \epsilon_m}{\epsilon_m} \end{aligned}$$

untuk mendapatkan nilai ϵ_m :

$$\epsilon_m = \frac{\sum_i w_i^{(m)} I(f_m(\mathbf{x}_i) \neq y_i)}{\sum_i w_i^{(m)}}$$

Objektif optimasi pada AdaBoost yaitu meminimalkan *cost function*. AdaBoost adalah algoritma sequential yang meminimalkan batas atas dari kesalahan pengklasifikasian empiris dengan memilih *weak* klasifikasi dan bobotnya. Satu per satu dipilih untuk mengurangi batas atas kesalahan secara maksimal [3]. AdaBoost berfokus meningkatkan kinerja model dengan cara mengurangi kesalahan prediksi. Setiap iterasi AdaBoost, bobot sampel yang salah diklasifikasikan akan ditingkatkan untuk memprioritaskan sampel-sampel yang sulit diklasifikasikan dengan benar oleh model sebelumnya. Dengan meningkatkan bobot pada sampel yang salah diklasifikasikan, model selanjutnya akan lebih fokus pada sampel-sampel yang sulit diklasifikasikan dan berusaha untuk mengurangi kesalahan prediksi tersebut. Maka dari itu, AdaBoost secara iteratif berusaha untuk meminimalkan *cost function* yang berkaitan dengan kesalahan prediksi pada setiap iterasinya.

Pada dasarnya AdaBoost adalah algoritma *greedy*, yang membangun "klasifikasi kuat". Optimasi *function* yang divariasikan adalah bobot sampel dan model parameter [2].

1. Bobot Sampel :

   AdaBoost mendefinisikan distribusi bobot pada sampel data. Bobot-bobot ini diperbarui setiap kali *weak learner* yang baru ditambahkan, sehingga sampel yang salah diklasifikasikan oleh *weak learner* diberikan bobot yang lebih besar. Dengan cara ini, sampel yang saat ini salah diklasifikasikan lebih ditekankan selama pemilihan *weak learner* berikutnya [3]. Maka dari itu, penyesuaian bobot sampel digunakan sebagai strategi optimasi untuk meminimalkan kesalahan prediksi dan meningkatkan akurasi model.

2. Model Parameter :

   AdaBoost menggunakan model-model lemah (*weak learner*) sebagai komponen dasar dalam ensemble. *Weak learner* adalah algoritma pembelajaran yang menghasilkan hipotesis (contoh; klasifikasi) yang memiliki performa sedikit lebih baik daripada peluang acak. Contoh *weak learner* yang efisien secara komputasi yaitu *decision tree*, dan *decision stump* (*decision tree* dengan satu level pemisahan) [4]. Dalam setiap iterasi AdaBoost, *weak learner* diberikan bobot berdasarkan kinerja dalam mengklasifikasikan sampel. Setiap pemanggilan menghasilkan klasifikasi yang lemah dan semuanya harus digabungkan menjadi satu klasifikasi tunggal untuk hasil yang lebih akurat [3]. Dengan demikian, AdaBoost mengoptimalkan model parameter, yaitu parameter yang digunakan oleh *weak learner* untuk mengurangi kesalahan prediksi dan meningkatkan akurasi model keseluruhan.

Dalam algoritma AdaBoost, *cost function* dioptimasi dengan melakukan penyesuaian bobot pada sampel dan model-model lemah (*weak learner*). AdaBoost (*Adaptive Boosting*), bertujuan menggabungkan berbagai hasil klasifikasi dari model lemah (*weak learner*) untuk meningkatkan kinerja klasifikasi. Setiap *weak learner* dilatih menggunakan sekumpulan sampel pelatihan yang sederhana dan setiap sampel memiliki bobot [5]. Berikut adalah langkah-langkah umum untuk mengoptimasi *cost function* pada AdaBoost :

Input: Data $\mathcal{D}_N = \{\mathbf{x}^{(n)}, t^{(n)}\}_{n=1}^N$ where $t^{(n)} \in \{-1, +1\}$
- This is different from previous lectures where we had $t^{(n)} \in \{0, +1\}$
- It is for notational convenience, otw equivalent.

A classifier or hypothesis $h : \mathbf{x} \to \{-1, +1\}$

0-1 loss: $\mathbb{I}[h(x^{(n)}) \neq t^{(n)}] = \frac{1}{2}(1 - h(x^{(n)}) \cdot t^{(n)})$

Input: Data $\mathcal{D}_N$, weak classifier WeakLearn (a classification procedure that returns a classifier h, e.g. best decision stump, from a set of classifiers $\mathcal{H}$, e.g. all possible decision stumps), number of iterations T

Output: Classifier $H(x)$

Initialize sample weights: $w^{(n)} = \frac{1}{N}$ for $n = 1, \ldots, N$

For $t = 1, \ldots, T$
- Fit a classifier to data using weighted samples ($h_t \leftarrow \text{WeakLearn}(\mathcal{D}_N, \mathbf{w})$), e.g.,

$$h_t \leftarrow \operatorname*{argmin}_{h \in \mathcal{H}} \sum_{n=1}^{N} w^{(n)} \mathbb{I}\{h(\mathbf{x}^{(n)}) \neq t^{(n)}\}$$

- Compute weighted error $\text{err}_t = \frac{\sum_{n=1}^N w^{(n)} \mathbb{I}\{h_t(\mathbf{x}^{(n)}) \neq t^{(n)}\}}{\sum_{n=1}^N w^{(n)}}$
- Compute classifier coefficient $\alpha_t = \frac{1}{2} \log \frac{1 - \text{err}_t}{\text{err}_t}$ $(\in (0, \infty))$
- Update data weights

$$w^{(n)} \leftarrow w^{(n)} \exp\left(-\alpha_t t^{(n)} h_t(\mathbf{x}^{(n)})\right) \left[\equiv w^{(n)} \exp\left(2\alpha_t \mathbb{I}\{h_t(\mathbf{x}^{(n)}) \neq t^{(n)}\}\right)\right]$$

Return $H(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^T \alpha_t h_t(\mathbf{x})\right)$

<u>*Input*</u> berupa data D_N yang berisikan *n* sampel ke-1 hingga sampel ke-*N* dimana terdapat *input x* dan *output t* yang merupakan elemen set dari dua nilai, yaitu -1 (kelas negatif) dan 1 (kelas positif), selanjutnya *input weak classifier* (sebuah prosedur klasifikasi yang mengembalikan sebuah pengklasifikasian *h* seperti *decision stump*), dan nomor dari sebuah iterasi *T*.

<u>*Ouput*</u> berupa klasifikasi *H(x)*.

1. Inisialisasi bobot: setiap sampel pada training diberi bobot awal yang sama, dengan cara 1 dibagi dengan *N* (jumlah keseluruhan dari *training* sampel).
2. Lakukan perulangan untuk setiap *weak learner* yang dimiliki.
3. Prediksi *weak learner*: setiap *weak learner* (contoh: *decision stump*) melakukan klasifikasi menggunakan bobot sampel yang telah diberikan.
4. Hitung bobot *error* dari klasifikasi *weak learner* pada iterasi *learner* ke-*t*: setelah *weak learner* dilatih, bobot *error* dihitung berdasarkan perbandingan antara prediksi model dan label aktual. Sampel yang salah diprediksi memiliki bobot *error* yang lebih tinggi, begitupun

sebaliknya. Hal ini memberikan penekanan pada sampel yang sulit diklasifikasi oleh model sebelumnya.

5. Hitung klasifikasi bobot: setiap *weak learner* diberi bobot berdasarkan kinerjanya. Model dengan bobot kesalahan lebih rendah akan memiliki bobot yang lebih tinggi.
6. *Update* bobot pada masing-masing data.

Dengan demikian, optimasi *cost function* adalah langkah yang dilakukan dalam proses pelatihan model AdaBoost untuk meminimalkan kesalahan prediksi.

- **Prediksi**

Setelah proses perulangan optimasi selesai, prediksi dari setiap *weak learner* dikalikan dengan bobot klasifikasi yang sesuai. Kemudian hasil prediksi digabungkan dengan cara menjumlahkannya. Hasil prediksi model ensemble dapat digunakan untuk memprediksi kelas atau nilai target pada data baru.

II. Pseudocode AdaBoost

- *Pseudocode melakukan fitting model*
  - *Input*
    - X : Dataset pelatihan
    - y : Label/target dari dataset pelatihan
    - n_estimator : jumlah estimator atau *model base*
  - *Proses*

    def fit(X, y):
    - Inisialiasi base estimator, base_estimator = DecisionStump()
    - Dapatkan jumlah sampel dan fitur dalam data latihan
    - Inisialisasi bobot untuk setiap sampel
    - Inisialisasi nilai alpha untuk setiap estimator
    - Lakukan pelatihan untuk setiap estimator (perulangan):
      1. Latih estimator dengan data latih dan bobot saat ini, fit estimator(X, y, weights)
      2. Lakukan prediksi menggunakan estimator yang telah dilatih
      3. Hitung tingkat kesalahan berbobot, err_t
      4. Hitung nilai alpha untuk estimator saat ini, α_t
      5. Perbarui bobot sampel
      6. Tambahkan estimator yang telah dilatih ke dalam list
    - Akhir pelatihan
  - *Output*
    - Model yang telah dilatih dengan estimator
    - Nilai alpha α_t
- *Pseudocode melakukan prediction*
  - *Input*
    - X : sampel input untuk melakukan prediksi
  - *Proses*

    def predict(X):
    - Inisialisasi label yang diprediksi
    - Iterasi setiap estimator dalam ensemble
      1. Lakukan prediksi menggunakan estimator saat ini.
      2. Tambahkan prediksi ke dalam prediksi keseluruhan.

- Ubah prediksi menjadi label biner, y_pred = sign(y_pred)
- Kembalikan label yang diprediksi

- *Output*
  - y_pred (prediksi akhir) – prediksi kelas

III. Implementasi Algoritma dan Analisa

Untuk mengaplikasikan *code from scratch* pada AdaBoost, dilakukan langkah-langkah dan analisis implementasi sebagai berikut:

1. Melakukan inisialisasi *base estimator*.
   Implementasi AdaBoost dari *code from scratch* menggunakan *DecisionStump* dan *DecisionTreeMaxDepth1* sebagai *base estimator*.

```
class DecisionStump:
    def __init__(
        self,
        polarity=1,
        feature_idx=None,
        threshold=None
    ):
        self.polarity = polarity
        self.feature_idx = feature_idx
        self.threshold = threshold
```

- `*DecisionStump*` : Kelas `*DecisionStump*` digunakan sebagai *base estimator* dimana model pembelajarannya sangat sederhana, hanya menggunakan satu fitur dan satu *threshold* untuk membuat prediksi.

```
def _best_split(self, X, y, weights):
    best_polarity = None
    best_feature_idx = None
    best_threshold = None
    min_error = float('inf')

    for feature_i in range(self.n_features):
        X_column = X[:, feature_i]
        thresholds = np.unique(X_column)

        for threshold in thresholds:
            polarity = 1
            predictions = np.ones(self.n_samples)
            predictions[X_column <= threshold] = -1

            error = np.sum(weights[y != predictions])

            if error > 0.5:
                error = 1 - error
                polarity = -1

            if error < min_error:
                best_polarity = polarity
                best_feature_idx = feature_i
                best_threshold = threshold
                min_error = error

    return best_polarity, best_feature_idx, best_threshold
```

- Metode `*best_split*` digunakan untuk mencari pemisahan terbaik pada setiap node. Metode ini akan mencoba semua kemungkinan pemisahan dengan memvariasikan fitur dan ambang batas serta mencari pemisahan dengan *error* terendah.

```
def fit(self, X, y, weights):
    X = np.array(X).copy()
    y = np.array(y).copy()
    self.n_samples, self.n_features = X.shape

    best_split_result = self._best_split(X, y, weights)

    self.polarity = best_split_result[0]
    self.feature_idx = best_split_result[1]
    self.threshold = best_split_result[2]
```

- Metode `*fit*` digunakan untuk melatih *DecisionStump* dengan menggunakan data latih yang diberikan.

```
def predict(self, X):
    X = np.array(X).copy()
    n_samples = X.shape[0]
    X_column = X[:, self.feature_idx]

    y_pred = np.ones(n_samples)
    if self.polarity == 1:
        y_pred[X_column <= self.threshold] = -1
    else:
        y_pred[X_column >= self.threshold] = -1

    return y_pred
```

- Metode `*predict*` digunakan untuk membuat prediksi dengan *DecisionStump* yang telah dilatih. Metode ini membandingkan fitur yang diberikan dengan *threshold* yang ada dan memberikan prediksi berdasarkan polaritas dan hubungan antar fitur dan *threshold*.

```
class DecisionTreeMaxDepth1:
    def __init__(
        self
    ):
        self.tree = DecisionTreeClassifier(max_depth=1)

    def fit(self, X, y, weights):
        self.tree.fit(X, y, sample_weight=weights)

    def predict(self, X):
        return self.tree.predict(X)
```

- `*DecisionTreeMaxDepth1*` : Kelas `*DecisionTreeMaxDepth1*` digunakan sebagai alternatif dari *DecisionStump* dalam AdaBoost. Kelas ini merupakan *decision tree* dengan kedalaman maksimum 1, yang artinya hanya satu pemisahan yang diizinkan pada setiap node menggunakan *DecisionTreeClassifier* dari *library* scikit-learn dengan parameter `max_depth = 1`.

`*DecisionStump*` dan `*DecisionTreeMaxDepth1*` pada implementasi tersebut memiliki perbedaan utama pada cara mereka melakukan pemisahan dan tingkat kompleksitas model yang dihasilkan. Pada *DecisionStump* hanya memiliki satu tingkat pemisahan (kedalaman 1) dalam bentuk *stump*, memiliki dua kemungkinan prediksi (polaritas) berdasarkan nilai fitur yang memenuhi kondisi pemisahan, melakukan pemisahan berdasarkan kombinasi fitur dan *threshold* terbaik yang menghasilkan error terendah dan menghasilkan pemisah dengan lebih dari satu tingkat jika diperlukan. Sedangkan `*DecisionTreeMaxDepth1*` hanya memiliki dua node: *root node* dan *leaf node*. *Root node* melakukan pemisahan berdasarkan fitur dan *threshold* yang terbaik. *Leaf node* menyediakan prediksi yang sama untuk semua contoh yang diberikan. *DecisionTreeMaxDepth1* menggunakan *DecisionTreeClassifier* dari scikit-learn sebagai estimator untuk membangun pohon keputusan dengan kedalaman maksimum 1 sehingga menghasilkan pohon keputusan yang sederhana.

2. Implementasi algoritma AdaBoost

```
class AdaboostClassifier:
    def __init__(
        self,
        estimator=None,
        n_estimators=5,
        learning_rate=1.0,
    ):
        self.estimator = estimator
        self.n_estimators = n_estimators
        self.learning_rate = learning_rate
```

- Kelas `*AdaBoostClassifier*` merupakan kelas yang mewakili implementasi algoritma AdaBoost untuk klasifikasi. Pada inisialisasi `*AdaBoostClassifier*` menerima 3 parameter utama yaitu :
  - `*estimator*` : *base estimator* yang digunakan dalam ensemble, jika tidak ada *base estimator* yang didefinisikan maka *DecisionStump* akan digunakan sebagai *base estimator*.
  - `n_*estimator*` : jumlah estimator dalam ensemble, defaultnya adalah 5.
  - `*learning_rate*` : tingkat pembelajaran untuk algoritma AdaBoost, defaultnya adalah 1.0.

```
def fit(self, X, y):
    if self.estimator is None:
        base_estimator = DecisionStump()
    else:
        base_estimator = DecisionTreeMaxDepth1()

    X = np.array(X).copy()
    y = np.array(y).copy()
    self.n_samples, self.n_features = X.shape

    self.weights = np.full(self.n_samples, (1/self.n_samples))
    self.alpha = np.zeros(self.n_estimators)
    self.estimators = []

    for i in range(self.n_estimators):
        estimator = copy.deepcopy(base_estimator)

        estimator.fit(X, y, weights = self.weights)

        y_pred = estimator.predict(X)
        error = np.sum(self.weights[y != y_pred])

        eps = 1e-10
        alpha = self.learning_rate * np.log((1-error) / (error+eps))

        self.weights *= np.exp(-alpha * y * y_pred)
        self.weights /= np.sum(self.weights)

        self.estimators.append(estimator)
        self.alpha[i] = alpha
```

- Metode `*fit(X, y)*`, metode ini digunakan untuk melatih `*AdaBoostClassifier*` menggunakan data pelatihan yang diberikan. Pertama, metode ini memeriksa *base estimator* yang ditentukan. Jika tidak ada yang ditentukan, *DecisionStump* akan digunakan sebagai *base estimator*. Selanjutnya, data pelatihan dan targetnya diubah menjadi format array numpy. Metode ini menginisialisasi berbagai atribut yang diperlukan untuk proses pelatihan, seperti *weights* (bobot sampel), *alpha* (nilai bobot), dan *estimators* (daftar estimator). Kemudian, melalui iterasi sebanyak n_estimators, dilakukan pelatihan *base estimator* dengan menggunakan *sample weights* yang diperbarui berdasarkan hasil prediksi sebelumnya. Selama iterasi, *weights* di*update* berdasarkan *error-weighted loss* dari prediksi sebelumnya, dan alpha dihitung berdasarkan `*learning_rate*` dan *error* yang dihasilkan. Pada akhir pelatihan, estimators dan alpha disimpan sebagai bagian dari model ensemble.

```
def predict(self, X):
    X = np.array(X).copy()

    y_pred = np.zeros(X.shape[0])

    pred = [self.alpha[i] * self.estimators[i].predict(X) for i in range(self.n_estimators)]
    y_pred = np.sum(pred, axis=0)

    y_pred = np.sign(y_pred)
    return y_pred
```

- Metode `*predict(X)*`, metode ini digunakan untuk melakukan prediksi pada data *input* X menggunakan model ensemble yang telah dilatih. Pertama, data *input* diubah menjadi format array numpy. Selanjutnya, metode ini menginisialisasi variabel y_pred dengan array kosong. Setiap estimator dalam ensemble diprediksi pada data *input*, kemudian hasil

prediksi tersebut dikalikan dengan alpha masing-masing estimator. Prediksi dari semua estimator dikombinasikan dan dijumlahkan untuk menghasilkan prediksi akhir. Nilai prediksi akhir diberikan dengan tanda fungsi signum (sign), yang menghasilkan label kelas biner -1 atau 1. Hasil prediksi akhir dikembalikan sebagai array numpy dengan bentuk (n_samples,).

3. *Import library*

```
import numpy as np
import pandas as pd

from ml_from_scratch.ensemble import AdaboostClassifier
from ml_from_scratch.ensemble import DecisionTreeMaxDepth1
from ml_from_scratch.metrics import accuracy_score
```

Lakukan *import* pada beberapa *library* yang diperlukan seperti numpy dan pandas untuk manipulasi data, serta beberapa modul khusus dari library "ml_from_scratch" yang mencakup implementasi `*AdaBoostClassifier*`, `*DecisionTreeMaxDepth1*` dan `*metrics*`.

4. *Load* dan *preprocessing data*

```
from sklearn import datasets
from sklearn.model_selection import train_test_split

data = datasets.load_breast_cancer()
X, y = data.data, data.target

y[y == 0] = -1

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=5
)
```

Dataset merupakan data kanker payudara yang berasal dari sklearn. Fitur-fitur dan label dari dataset di *assign* ke variabel X dan y. Nilai label y yang awalnya 0 dan 1 diubah menjadi -1 dan 1 menggunakan y[y == 0] = -1. Ini dilakukan karena AdaBoostClassifier mengasumsikan label kelas adalah -1 dan 1.

5. *Train-test split*

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=5
)
```

Data dibagi menjadi data pelatihan dan data pengujian menggunakan `*train_test_split()*` dari sklearn. Data pelatihan digunakan untuk melatih model, dan data pengujian digunakan untuk menguji performa model yang dilatih. Selanjutnya ukuran pengujian yang diinginkan adalah 20% dari data dan menetapkan random_state untuk mengontrol keacakan yang terjadi dalam suatu proses.

6. Inisialisasi *AdaBoostClassifier*

```
clfs = AdaboostClassifier()
```

Objek *AdaBoostClassifier* `clfs` dibuat tanpa parameter. Dalam hal ini, *DecisionStump* akan digunakan sebagai *base estimator* karena tidak ada parameter yang diberikan.

7. Latih model

```
clfs.fit(X_train, y_train)
```

Model *AdaBoostClassifier* dilatih menggunakan metode `*fit()*` dengan menggunakan data pelatihan. Saat memanggil `*clfs.fit(X_train, y_train)*`, *AdaBoostClassifier* akan melatih sejumlah *base estimator* (DecisionStump) dengan memperbarui bobot sampel selama pelatihan.

8. Melakukan prediksi

```
y_pred = clfs.predict(X_test)
print("Prediksi Data :", y_pred)
```

Setelah pelatihan, model yang dilatih digunakan untuk melakukan prediksi pada data pengujian menggunakan metode `*predict()*`. Hasil prediksi disimpan dalam variabel y_pred.

9. Evaluasi performa

```
acc = accuracy_score(y_test, y_pred)
print("Accuracy :", acc)
```

Untuk mengukur performa model, akurasi prediksi dihitung menggunakan metode `*accuracy_score()*` dari modul `*metrics*`. Hasil akurasi dicetak menggunakan `*print("Accuracy :", acc)*`.

Untuk meningkatkan performa model dilakukan eksperimen pada beberapa *hyperparameter* seperti berikut:

1. Estimator dasar (*base_estimator*)
   Saat *base_estimator* tidak didefinisikan maka *base_estimator* yang digunakan adalah *DecisionStump*. Pada saat `*learning_rate = 1.0*` dan `*n_estimator = 5*` dimana nilai tersebut merupakan nilai *default*, akurasi yang didapatkan adalah 0.92. Hal tersebut tidak berbeda jauh dengan akurasi yang diperoleh pada *base estimator DecisionTreeMaxDepth1* yaitu 0.93.
2. Jumlah estimator (*n_estimators*)
   n_estimators berguna untuk menentukan jumlah estimator (*model base*) yang akan digunakan dalam ensemble AdaBoost. Estimator adalah model pembelajaran mesin yang digunakan untuk melakukan prediksi pada setiap iterasi. Saat menaikkan jumlah estimator, akurasi yang diperoleh baik pada *base estimator DecisionStump* maupun *DecisionTreeMaxDepth1* mengalami penurunan yang signifikan. Pada saat `*n_estimator = 50*` `*n_estimator = 50*` dan `*n_estimator = 100*` dengan *base_estimator decisionstump* dan `*learning_rate = 1.0*`, akurasinya adalah 0.087 sedangkan pada *base_estimator DecisionTreeMaxDepth1* dengan `*learning_rate = 1.0*`, akurasinya adalah 0.64 dan 0.63. Selanjutnya nilai pada jumlah estimator dilakukan pengurangan menjadi `*n_estimator = 50*`
3. *Learning rate* (*learning_rate*)
   learning_rate berguna mengontrol kontribusi setiap estimator dalam model ensemble. *Learning rate* menyesuaikan bobot prediksi dari setiap estimator pada setiap iterasi. Pada saat `*learning_rate = 1.0*`, `*n_estimator* = 5` dan `*base_estimator = DecisionStump*`, akurasi yang diperoleh adalah 0.92. Saat *learning_rate* diturunkan menjadi 0.5 akurasinya naik menjadi 0.96. Selanjutnya nilai *learning_rate* diturunkan lagi menjadi 0.1 sehingga akurasi yang diperoleh menjadi 0.95.

Detail pada hasil akurasi ketika *hyperparameter* diubah, dapat dilihat pada gambar berikut :

base_estimator	***DecisionStump***				***DecisionTreeMaxDepth1***			
akurasi	**0.91**	**0.92**	**0.09**	**0.09**	**0.91**	**0.93**	**0.64**	**0.63**
n_estimator	1	5	50	100	1	5	50	100
learning_rate	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
akurasi	**0.91**	**0.96**	**0.97**	**0.97**	**0.91**	**0.95**	**0.97**	**0.97**
n_estimator	1	5	50	100	1	5	50	100
learning_rate	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
akurasi	**0.91**	**0.95**	**0.96**	**0.96**	**0.91**	**0.95**	**0.96**	**0.96**
n_estimator	1	5	50	100	1	5	50	100
learning_rate	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

IV. Kesimpulan

Scratch from code pada algoritma AdaBoost dilakukan dengan membangun *base estimator* yang merupakan estimator lemah yaitu berupa *decisionstump* dan *decision tree* dengan kedalaman maksimum 1 (sebagai alternatif base estimator). AdaBoost adalah metode ensemble yang menggabungkan beberapa estimator lemah menjadi satu model yang lebih kuat. Algoritma AdaBoost melakukan iterasi untuk melatih estimator lemah dengan memperbarui bobot sampel berdasarkan prediksi dan label aktual. Bobot estimator lemah dan bobot kesalahan digunakan untuk menghitung bobot akumulatif dan digunakan untuk memprediksi label kelas. Selanjutnya dilakukan pelatihan pada model dan melakukan penghitungan akurasi. Setelah didapatkan akurasi, dilakukan beberapa kali eksperimen agar meningkatkan performa model dengan mengubah ketiga hyperparameter yaitu `*base_estimator*`, `*n_estimator*` dan `*learning_rate*` dengan menaikkan atau menurunkan nilainya. Didapatkan performa model terbaik dengan akurasi tertinggi yaitu 0.97 dengan `*n_estimator = 50*` dan `*learning_rate = 0.5*`.

V. Referensi

[1] Frean M, Downs Tom. A Simple Cost Function for Boosting. University of Queensland. 2012.

[2] Hertzmann A., Fleet D., Brubaker M. Machine Learning and Data Mining Lecture Notes CSC C11/D11. University of Toronto Scarborough. 2015.

[3] Corso J. AdaBoost Lecture 8. Suny at Buffalo. 2010.

[4] Zemel R., Erdogdu M. A. CSC 311: Introduction to Machine Learning Lecture 6 - SVMs and Ensembles: Boosting. University of Toronto.

[5] Tharwat A. AdaBoost Classifier: an Overview. 2018