TUGAS BESAR 2: IMPLEMENTASI ALGORITMA PEMBELAJARAN MESIN

IF3170 INTELIGENSI ARTIFISIAL

Disusun oleh:

Machine Learningn't

Mayla Yaffa Ludmilla	13523050
Salman Hanif	13523056
Noumisyifa Nabila Nareswari	13523058
Diyah Susan Nugrahani	13523080
Muhammad Edo Raduputu Aprima	13523096

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG 2025

DAFTAR ISI

A. Implementasi *Decision Tree Learning*

Implementasi algoritma Decision Tree ini dibuat menggunakan algoritma C4.5 dan ID3 dengan pemilihan atribut terbaik menggunakan Gain Ratio. Model ini mendukung tipe fitur numerik dan nominal, serta menangani *missing values* dengan mekanisme tertentu pada saat splitting.

Berikut adalah atribut-atribut pada kelas DecisionTreeLearning

```
def __init__(self, min_samples_split=2, max_depth=100, feature_types=None):
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.feature_types = feature_types
        self.root = None
```

Atribut	Peran
min_samples_split	Batas minimal jumlah sampel sebelum bisa dilakukan *split* untuk mencegah *overfitting*.
max_depth	Batas maksimal kedalaman pohon untuk mencegah *overfitting*.
feature_types	Menyimpan tipe fitur per kolom (numerical/nominal)
root	Simpul (*node)* akar dari *decision tree.*

Setiap simpul (*node*) direpresentaskan dengan kelas Node.

```
class Node:
    def __init__(self, feature=None, threshold=None, categories=None,
is_nominal=False, children=None, value=None):
        self.feature = feature
        self.threshold = threshold
        self.categories = categories
        self.is_nominal = is_nominal
        self.children = children if children is not None else {}
        self.value = value

    def is_leaf_node(self):
        return self.value is not None
```

Atribut	Peran
feature	Index fitur yang terpilih sebagai pemisah
threshold	Nilai *threshold* untuk split numerik
categories	Daftar kategori untuk split nominal
children	Anak dari simpul

value	Menyimpan label prediksi jika *node* adalah daun
is_nominal	Boolean penanda apakah fitur adalah kategori
is_leaf_node()	Mengecek apakah *node* adalah daun

```
def _entropy(self, y):
    _, counts = np.unique(y, return_counts=True)
    p = counts / len(y)
    return -np.sum(p * np.log2(p))

def _entropy_from_counts(self, counts, total):
    entropy = 0
    for count in counts.values():
        if count > 0:
            p = count / total
            entropy -= p * np.log2(p)
    return entropy

def _is_missing(self, value):
    if value is None: return True
    if isinstance(value, float) and np.isnan(value):
        return True
    if isinstance(value, str) and value.lower() in ['nan', 'none', '', '?',
'na', 'n/a']:
        return True
    return False

def _determine_feature_types(self, X):
    feature_types = []
    n_features = X.shape[1]
    for i in range(n_features):
        col_values = X[:, i]
        valid_vals = [v for v in col_values if not self._is_missing(v)]
        is_numerical = True
        for val in valid_vals:
            if isinstance(val, str) or isinstance(val, bool):
                is_numerical = False
                break
        feature_types.append('numerical' if is_numerical else 'nominal')
    return feature_types

def _find_best_numerical_split(self, X, y, feature_idx):
    feature_values = X[:, feature_idx]
    valid_mask = ~pd.isna(feature_values) if hasattr(feature_values, 'isna')
else np.array([not self._is_missing(v) for v in feature_values])

    X_col = feature_values[valid_mask]
    y_col = y[valid_mask]

    if len(X_col) < 2:
        return None, -1

    try:
        X_col = X_col.astype(float)
    except:
        return None, -1
```

```
    sorted_indices = np.argsort(X_col)
    X_sorted = X_col[sorted_indices]
    y_sorted = y_col[sorted_indices]

    n_samples = len(y_sorted)
    right_counts = Counter(y_sorted)
    left_counts = Counter()

    parent_entropy = self._entropy_from_counts(right_counts, n_samples)

    best_gain_ratio = -1
    best_threshold = None

    left_n = 0
    right_n = n_samples

    for i in range(1, n_samples):
        c = y_sorted[i-1]
        left_counts[c] += 1
        right_counts[c] -= 1
        if right_counts[c] == 0:
            del right_counts[c]

        left_n += 1
        right_n -= 1

        if X_sorted[i] == X_sorted[i-1]:
            continue

        left_entropy = self._entropy_from_counts(left_counts, left_n)
        right_entropy = self._entropy_from_counts(right_counts, right_n)

        child_entropy = (left_n / n_samples) * left_entropy + (right_n /
n_samples) * right_entropy
        info_gain = parent_entropy - child_entropy

        p_left = left_n / n_samples
        p_right = right_n / n_samples
        split_info = -(p_left * np.log2(p_left) + p_right * np.log2(p_right))

        if split_info == 0:
            gain_ratio = 0
        else:
            gain_ratio = info_gain / split_info

        if gain_ratio > best_gain_ratio:
            best_gain_ratio = gain_ratio
            best_threshold = (X_sorted[i] + X_sorted[i-1]) / 2

    return best_threshold, best_gain_ratio

def _find_best_nominal_split(self, X, y, feature_idx):
    feature_values = X[:, feature_idx]
    non_missing_indices = [i for i, v in enumerate(feature_values) if not
self._is_missing(v)]

    if not non_missing_indices:
        return None, -1

    non_missing_vals = np.array([feature_values[i] for i in
non_missing_indices])
    non_missing_y = np.array([y[i] for i in non_missing_indices])
```

```
    unique_categories = np.unique(non_missing_vals)
    if len(unique_categories) <= 1:
        return None, -1

    children_y_list = []
    for category in unique_categories:
        children_y_list.append(non_missing_y[non_missing_vals == category])

    parent_entropy = self._entropy(non_missing_y)
    children_entropy = 0
    split_info = 0
    total_len = len(non_missing_y)

    for child in children_y_list:
        if len(child) > 0:
            p = len(child) / total_len
            children_entropy += p * self._entropy(child)
            split_info -= p * np.log2(p)

    info_gain = parent_entropy - children_entropy
    gain_ratio = info_gain / split_info if split_info != 0 else 0

    return list(unique_categories), gain_ratio

def _find_best_split(self, X, y):
    max_gain_ratio = -1
    best_feature_idx = None
    best_split_info = None
    n_features = X.shape[1]

    for idx in range(n_features):
        feature_type = self.feature_types[idx] if self.feature_types else
'numerical'
        if feature_type == 'nominal':
            split_info, gain_ratio = self._find_best_nominal_split(X, y, idx)
        else:
            split_info, gain_ratio = self._find_best_numerical_split(X, y,
idx)

        if gain_ratio > max_gain_ratio:
            max_gain_ratio = gain_ratio
            best_feature_idx = idx
            best_split_info = split_info

    return best_feature_idx, best_split_info, max_gain_ratio

def _create_leaf(self, y):
    if len(y) == 0:
        return Node(value=None)
    counter_y = Counter(y)
    majority_class = counter_y.most_common(1)[0][0]
    return Node(value=majority_class)
```

Berikut penjelasan fungsi-fungsi pembantu dalam algoritma *decision tree:*

- Fungsi `_entropy(self,y)` menghitung nilai entropi dari suatu himpunan label y.
- Fungsi `_entropy_from_counts(self, counts, total)` menghitung entropi langsung dari jumlah kelas (*counts*)

- Fungsi `_is_missing(self, value)` menentukan apakah suatu nilai dianggap sebagai *missing value* (nilai None, NaN, string kosong dan string bertuliskan nan, none, n/a)
- Fungsi `_determine_feature_types(self, X)` menentukan tipe kolom fitur (numerical/nominal). String dan boolean dianggap nominal sedangkan sisanya numerical.
- Fungsi `_find_best_numerical_split(self, X, y, feature_idx)` menguji kandidat *threshold* di fitur numerik dengan alur sebagai berikut:
    - Sort data berdasarkan nilai fitur
    - Geser *threshold* dari kiri ke kanan
    - Hitung entropi kiri dan kanan, *information gain, split info,* dan *gain ratio*
    - Pilih threshold dengan *gain ratio* tertinggi dan mengembalikannya
- Fungsi `_find_best_nominal_split(self, X, y, feature_idx)` melakukan *split* berdasarkan kategori dengan alur sebagai berikut:
    - Identifikasi kategori yang ada
    - Membuat satu *child node* untuk setiap kategori
    - Hitung entropi *parent* dan setiap *child*, *information gain, split info,* dan *gain ratio* dan mengembalikan list kategori serta *gain ratio*-nya.
- Fungsi `_find_best_split(self, X, y)` mencoba *split* pada seluruh fitur dan mengambil fitur dengan *gain ratio* tertinggi. Jika semua fitur memberi gain <= 0, *node* menjadi daun.
- Fungsi `_create_leaf(self, y)` membuat *node* daun yang berisi kelas mayoritas sampel dari *node* tersebut.

```
def _build_tree(self, X, y, depth=0):
    n_samples, n_features = X.shape
    unique_y = np.unique(y)
    n_labels = len(unique_y)

    if (depth >= self.max_depth) or (n_samples < self.min_samples_split) or
(n_labels == 1):
        return self._create_leaf(y)

    best_feature, split_info, gain_ratio = self._find_best_split(X, y)

    if best_feature is None or gain_ratio <= 0:
        return self._create_leaf(y)

    node = Node(feature=best_feature)
    f_type = self.feature_types[best_feature] if self.feature_types else
'numerical'

    if f_type == 'numerical':
        node.threshold = split_info
        node.is_nominal = False

        col_vals = X[:, best_feature]

        left_indices = []
        right_indices = []
        missing_indices = []

        for i, v in enumerate(col_vals):
            if self._is_missing(v):
                missing_indices.append(i)
```

```
                continue
            try:
                val_float = float(v)
                if val_float <= node.threshold:
                    left_indices.append(i)
                else:
                    right_indices.append(i)
            except ValueError:
                missing_indices.append(i)

        if len(left_indices) >= len(right_indices):
            left_indices.extend(missing_indices)
        else:
            right_indices.extend(missing_indices)

        if not left_indices or not right_indices:
            return self._create_leaf(y)

        node.children['left'] = self._build_tree(X[left_indices],
y[left_indices], depth+1)
        node.children['right'] = self._build_tree(X[right_indices],
y[right_indices], depth+1)

    else:
        node.categories = split_info
        node.is_nominal = True

        idx_map = {cat: [] for cat in node.categories}
        missing_indices = []

        for i, v in enumerate(X[:, best_feature]):
            if self._is_missing(v):
                missing_indices.append(i)
            elif v in idx_map:
                idx_map[v].append(i)

        majority_class = self._create_leaf(y).value

        counter_y = Counter(y)
        if counter_y:
                majority_global = counter_y.most_common(1)[0][0]
                if majority_global in idx_map:
                    idx_map[majority_global].extend(missing_indices)
                else:
                    first_cat = list(node.categories)[0]
                    idx_map[first_cat].extend(missing_indices)

        for category in node.categories:
            indices = idx_map[category]
            if not indices:
                node.children[category] = self._create_leaf(y)
            else:
                node.children[category] = self._build_tree(X[indices],
y[indices], depth+1)
    return node

def fit(self, X, y):
    X = np.array(X, dtype=object)
    y = np.array(y)
    if self.feature_types is None:
        self.feature_types = self._determine_feature_types(X)
    self.root = self._build_tree(X, y)
```

Data *training* akan *fitted* ke model melalui fungsi `fit()` untuk memulai pelatihan model. Fungsi `fit()` berperan memulai proses pembentukan tree. Data X dikonversi menjadi tipe objek agar dapat menangani campuran tipe nilai. Jika tipe fitur belum ditentukan, ia akan dianalisis menggunakan _determine_feature_types() untuk membedakan apakah fitur numerik atau nominal.

Fungsi `fit()` menggunakan fungsi `_build_tree()` yang merupakan fungsi rekursif untuk membangun *decision tree.* Fungsi ini mengecek *base case* yaitu ketika kedalaman tree telah mencapai batas maksimum, jumlah sampel pada *node* terlalu sedikit untuk dibagi lagi, atau seluruh label pada *node* sudah homogen. Jika salah satu dari kondisi tersebut terpenuhi, *node* tersebut akan langsung dijadikan daun. Jika belum, fungsi akan mencari pemisahan fitur terbaik dengan menghitung *gain ratio* menggunakan `_find_best_split()`. Bila tidak ada fitur yang memberikan peningkatan informasi (gain ratio ≤ 0), *node* kembali dijadikan daun.

Setelah fitur terbaik ditentukan, dibuat *node* baru sesuai tipe fitur tersebut. Jika fitur bersifat numerik, digunakan *threshold* sebagai titik pemisah, dan data akan dibagi menjadi dua cabang, kiri untuk nilai $\leq$ threshold dan kanan untuk nilai > threshold. *Missing value* ditempatkan pada cabang yang memiliki jumlah sampel lebih banyak untuk menjaga kestabilan pembagian. Jika pembagian menghasilkan cabang kosong, maka *node* tetap dibuat sebagai daun. Jika valid, pembagian diteruskan secara rekursif ke cabang kiri dan kanan. Untuk fitur kategorikal, data dipecah berdasarkan kategori uniknya. *Missing values* akan dialokasikan pada kategori yang memiliki kelas mayoritas secara global, atau ke kategori pertama jika tidak ditemukan. Jika suatu kategori tidak memiliki data setelah pembagian, cabang tersebut dijadikan daun dengan kelas mayoritas. Sebaliknya, jika berisi data, fungsi `_build_tree()` dipanggil kembali untuk kategori tersebut.

```
def _traverse_tree(self, sample, node):
    if node.is_leaf_node():
        return node.value

    val = sample[node.feature]

    if self._is_missing(val):
        if not node.children:
                return None
        first_child = list(node.children.values())[0]
        return self._traverse_tree(sample, first_child)

    if node.is_nominal:
        if val in node.children:
            return self._traverse_tree(sample, node.children[val])
        if node.children:
            return self._traverse_tree(sample,
list(node.children.values())[0])
        return None
    else:
        try:
            val_float = float(val)
            if val_float <= node.threshold:
                return self._traverse_tree(sample, node.children['left'])
```

```
            else:
                return self._traverse_tree(sample, node.children['right'])
        except:
            return self._traverse_tree(sample, node.children['left'])

def predict(self, X):
    X = np.array(X, dtype=object)
    return np.array([self._traverse_tree(sample, self.root) for sample in X])
```

Pada proses prediksi, fungsi `predict()` akan memanggil `traverse_tree()` untuk setiap sampel dan menelusuri pohon keputusan mulai dari *root* hingga mencapai daun. Jika *node* yang dikunjungi merupakan daun, nilai kelas pada node tersebut langsung dikembalikan sebagai hasil prediksi. Jika fitur pada node bertipe numerikal, nilai sampel akan dibandingkan dengan *threshold* untuk menentukan apakah penelusuran diarahkan ke cabang kiri atau kanan, dan apabila nilai tidak dapat dikonversi atau hilang maka diarahkan ke cabang kiri sebagai *fallback*. Jika fitur bertipe nominal, penelusuran dilanjutkan ke cabang sesuai kategori nilai fitur, dan jika kategori tidak ditemukan atau nilainya missing maka diarahkan ke anak pertama yang tersedia.

B. Implementasi *Logistic Regression*

Implementasi algoritma ini dibuat berdasarkan algoritma Stochastic Gradient Ascent, seperti yang tertera pada bahan ajar di kelas, dengan sedikit modifikasi khusus untuk kasus label target yang *multiclass*. Penanganan kasus *multiclass* dilakukan dengan strategi *one-against-all* yang akan membangun sebanyak k model untuk kasus label target dengan k kelas, kemudian, data akan dicoba ke semua model lalu kelas akhir akan ditentukan berdasarkan model yang memberikan nilai prediksi terbaik.

Berikut adalah atribut-atribut pada kelas LogisticRegression

```
def __init__(self, learning_rate=0.01, n_iterations=100, threshold=0.5,
mini_batch=True, batch_size=32):
    self.learning_rate = learning_rate
    self.n_iteration = n_iterations
    self.threshold = threshold
    self.weights = None
    self.bias = None
    self.mini_batch = mini_batch
    self.batch_size = batch_size

    self.classes_ = None
    self.is_multiclass = False
    self.binary_classifiers = {}

    self.training_history = []
```

`learning_rate`, `n_iteration`, `threshold`, adalah *hyperparameter* yang wajib ada untuk konsep algoritma *logistic regression*. Lalu, `weights`, dan `bias` berfungsi untuk menyimpan parameter algoritma yang dipelajari selama proses *training*. Untuk atribut `classes`, `is_multiclass`, dan `binary_classifiers`, ada untuk mendukung kasus *multiclass*. Terakhir, `training_history` ada untuk menyimpan jejak bobot tiap *epoch* untuk bonus generasi video/GIF.

Selanjutnya, seperti pada umumnya, data *training* akan *fitted* ke model melalui *method* fit untuk memulai pelatihan model.

```
def fit(self, x, y, track_history=True):

    # Deteksi jumlah kelas
    self.classes_ = np.unique(y)
    n_classes = len(self.classes_)

    # Cek apakah multiclass
    if n_classes > 2:
        self.is_multiclass = True
        self._fit_ovr(x, y, track_history)
    else:
        self.is_multiclass = False
        y_binary = np.where(y == self.classes_[0], 0, 1)
        self._fit_binary(x, y_binary, track_history)


def _fit_ovr(self, x, y, track_history=True):

    for class_label in self.classes_:
        # Buat binary labels: 1 untuk class ini, 0 untuk lainnya
        y_binary = (y == class_label).astype(int)

        # Buat binary classifier baru
        binary_clf = LogisticRegression(
            learning_rate=self.learning_rate,
            n_iterations=self.n_iteration,
            threshold=self.threshold,
            mini_batch=self.mini_batch,
            batch_size=self.batch_size
        )

        # Train binary classifier
        binary_clf._fit_binary(x, y_binary, track_history)

        # Simpan classifier
        self.binary_classifiers[class_label] = binary_clf

def _fit_binary(self, x, y, track_history=True):
    """Train single binary classifier"""
    if self.mini_batch:
        self._fit_mbgd(x, y, track_history)
    else:
        self._fit_sgd(x, y, track_history)

def _fit_sga(self, x, y, track_history=True):
    """melatih model dengan stochastic gradient ascent"""
    n_samples = x.shape[0]

    # bias diinisiasi 1
    x_biased = np.hstack((np.ones((n_samples, 1)), x))
    n_total_features = x_biased.shape[1]

    self.weights = np.zeros(n_total_features)

    # Clear history
    if track_history:
        self.training_history = []
```

```
    for epoch in range(self.n_iteration):
        # urutan data dirandom
        shuffled_indice = np.random.permutation(n_samples)
        x_shuffeled = x_biased[shuffled_indice]
        y_shuffeled = y[shuffled_indice]

        # pake yg jit
        self.weights = sgd_update_numba(
            x_shuffeled, y_shuffeled, self.weights,
            self.learning_rate, n_samples
        )

        # Track weights dan loss
        if track_history:
            loss = self._compute_loss(x, y)
            self.training_history.append({
                'epoch': epoch,
                'weights': self.weights.copy(),
                'loss': loss
            })

def _fit_mbsga(self, x, y, track_history=True):
    """melatih model dengan mini batch stochastic gradient ascent"""
    n_samples, n_features = x.shape

    # Bias diinisiasi 1
    x_biased = np.hstack((np.ones((n_samples, 1)), x))
    n_total_features = x_biased.shape[1]

    # Initialize weights
    self.weights = np.zeros(n_total_features)

    # Clear history
    if track_history:
        self.training_history = []

    # Hitung jumlah batch
    n_batches = n_samples // self.batch_size

    for epoch in range(self.n_iteration):
        # Shuffle sekali di awal epoch
        indices = np.random.permutation(n_samples)
        x_shuffled = x_biased[indices]
        y_shuffled = y[indices]

        # Proses per mini-batch
        for batch_idx in range(n_batches):
            start_idx = batch_idx * self.batch_size
            end_idx = start_idx + self.batch_size

            # Ambil batch
            x_batch = x_shuffled[start_idx:end_idx]
            y_batch = y_shuffled[start_idx:end_idx]

            # Vectorized z calc
            z = x_batch @ self.weights
            p = self.sigmoid(z)  # Batch predictions

            # Vectorized gradient jadi seluruh batch sekaligus
            gradient = x_batch.T @ (y_batch - p) / self.batch_size
```

```
            # Update weights
            self.weights += self.learning_rate * gradient

        # Handle sisa yang gk masuk batch
        if n_samples % self.batch_size != 0:
            x_batch = x_shuffled[n_batches * self.batch_size:]
            y_batch = y_shuffled[n_batches * self.batch_size:]

            z = x_batch @ self.weights
            p = self.sigmoid(z)
            gradient = x_batch.T @ (y_batch - p) / len(x_batch)
            self.weights += self.learning_rate * gradient

        # Track weights dan loss
        if track_history:
            loss = self._compute_loss(x, y)
            self.training_history.append({
                'epoch': epoch,
                'weights': self.weights.copy(),
                'loss': loss
            })
```

Berikut penjelasan alur kode *fitting* secara umum:

- Dalam fungsi `fit()`, jika terdeteksi hanya ada dua target kelas (*binary classification*), maka dijalankan fungsi `fit_binary()`, jika terdiri lebih dari dua target kelas (*multiclass classification*) maka dijalankan fungsi `fit_ovr()`
- Dalam fungsi `fit_ovr()` diimplementasikan strategi *one-against-all* (disebut juga *one-versus-rest*) yang berulang kali menjalankan `fit_binary()` sebanyak jumlah kelas target.
- Dalam fungsi `fit_binary()` akan dicek apakah *flag* `mini_batch` bernilai `True` atau `False`. Jika `True` dijalankan fungsi `_fit_mbgd()`, jika False dijalankan fungsi `_fit_sgd()`
- Fungsi`_fit_sga()` berisikan implementasi algoritma *stochastic gradient ascent* dengan alur sebagai berikut:
    - Tambah bias sebagai fitur ke-0
    - Inisialisasi bobot 0
    - Untuk setiap iterasi/epoch, data diacak lalu memanggil `sgd_update_numba()`
    - Menambahkan bobot tiap *epoch* ke training_history jika flag track_history bernilai True
- Fungsi `sgd_update_numba()` berfungsi untuk *update* weight. Fungsi ini menerima fitur yang sudah diacak, label yang sudah disejajarkan, bobot saat ini, *learning rate*, dan jumlah sampel yang kemudian melakukan:
    - *Loop* per *data point* (*Stochastic*)
    - Hitung *linear combination* $\sum(x\_i \times beban\_i)$
    - Hitung sigmoid dan p_i
    - *Update* bobot berdasarkan *error* (*gradient ascent*)
    - *Return* bobot terbaru

Fungsi ini dioptimasi dengan Numba JIT untuk menghemat waktu sekaligus tetap menjaga esensi dari "s*tochastic*".

- Jika menggunakan fungsi `_mbga_fit()`, algoritma tidak lagi stochastic walaupun tetap secara *gradient ascent*. Yang terjadi kurang lebih sama seperti `_fit_sga()`, hanya saja sebelum ke perhitungan sigmoid, error, gradient, dan bobot, data dibagi jadi beberapa *mini batch* terlebih dahulu. Selain itu, fungsi ini tidak menggunakan Numba JIT karena pakai operasi secara vektor.

Selanjutnya, setelah proses *training model* selesai, pengguna dapat melakukan prediksi menggunakan *method* `fit()`.

```
def predict_probability(self, x):
    """untuk prediksi data test"""
    if not self.is_multiclass:
        n_samples = x.shape[0]
        x_biased = np.hstack((np.ones((n_samples, 1)), x))

        linear_model = x_biased @ self.weights
        probabilities = self.sigmoid(linear_model)
        return probabilities

    else:
        n_samples = x.shape[0]
        n_classes = len(self.classes_)
        probs = np.zeros((n_samples, n_classes))

        for idx, class_label in enumerate(self.classes_):
            clf = self.binary_classifiers[class_label]
            n_samples_clf = x.shape[0]
            x_biased = np.hstack((np.ones((n_samples_clf, 1)), x))
            linear_model = x_biased @ clf.weights
            probs[:, idx] = clf.sigmoid(linear_model)

        return probs

def predict(self, x):
    probabilities = self.predict_probability(x)
    if not self.is_multiclass:
        predictions = (probabilities >= self.threshold).astype(int)
        return predictions
    else:
        # note: argmax adalah fungsi buat cari param yg hasilin nilai max
        # a.k.a argument of the maxima
        class_indices = np.argmax(probabilities, axis=1)
        return self.classes_[class_indices]
```

Berikut penjelasan alur kode *predicting* secara umum:

- Fungsi `predict()` pertama akan *generate* nilai `probabilities` dengan memanggil `predict_probabilty()`. Jika *binary classification,* maka fungsi akan membandingkan `probabilities` dengan `threshold` lalu `return 0` atau `1` berdasarkan komparasi tersebut. Jika *multiclass classification*, maka fungsi akan ambil indeks kelas dengan probabilitas maksimum untuk tiap sampel lalu return label target sesuai indeks.

- Untuk fungsi `predict_probabilty()` juga terbagi dua kasus:
    - Jika *binary classification*, program akan tambahkan bias ke fitur, lalu hitung nilai `z` dan `p`, lalu return nilai `p` yang merupakan probabilitas tiap sampel kelas 1.
    - Jika *multiclass classification*, program akan buat matriks probs dengan ukuran (`n_samples, n_classes`), lalu iterasi per kelas. Dalam iterasi per kelas program akan ambil *classifier binary* untuk kelas itu, lalu melakukan proses yang dilalui kasusnya *binary classification.*

Selain itu terdapat fungsi *helper* lain seperti `_compute_loss(x, y)` untuk *log loss tracking*, `sigmoid(z)` untuk hitung nilai sigmoid, `save_model(filename)` untuk menyimpan model, `load_model(filename)` untuk memuat suatu model, dan `generate_training_gif()` untuk menghasilkan GIF yang menampilkan *loss contour* and *parameter trajectory* selama proses *model training*.

C. Implementasi SVM

Algoritma SVM atau *support vector machine* merupakan algoritma *machine learning* yang digunakan untuk mengklasifikasikan data dengan cara mencari garis optimal atau *hyperplane* yang dapat memaksimalkan margin setiap kelas. Implementasi yang dipakai untuk SVM ini menggunakan optimasi kuadratik dengan tujuan menghindari *local minimum.*

Implementasi kode untuk algoritma ini dilakukan dengan pendekatan *object oriented programming* dengan membuat kelas SVM. Kelas SVM memiliki beberapa atribut yang merupakan parameter yang akan digunakan untuk perhitungan nantinya. Atributnya antara lain adalah sebagai berikut :

```
def __init__(self, C=1.0, kernel='linear', kernel_param=1.0, tol=1e-3,
max_iter=100):
        self.C = C
        self.kernel_type = kernel
        self.kernel_param = kernel_param
        self.tol = tol
        self.max_iter = max_iter

        # parameter optimasi
        self.alphas = None # pengali langrange
        self.support_vectors = None # supprot vectors
        self.sv_labels = None
        self.sv_indices = None

        # hyperplane
        self.w = None
        self.b = None

        # chace error
        self.errors = None
        self.K = None
```

Parameter	Peran
C	Parameter regularisasi untuk mengontrol *trade-off* antara memaksimalkan margin dan meminimalkan kesalahan klasifikasi. C yang tinggi artinya meminimalkan kesalahan.
kernel_type	Menentukan fungsi kernel yang akan digunakan untuk memetakan data ke ruang berdimensi tinggi yang sesuai dengan persebaran data.
kernel_param	Parameter spesifik untuk kernel non-linear seperti gamma untuk kernel RPF dan *degree* untuk kernel polynomial.
tol	Toleransi konvergensi yang masih bisa diterima.
max_iter	Jumlah maksimum iterasi yang akan dijalankan oleh algoritma untuk mencegah pelatihan berjalan terlalu lama jikan konvergensi lambat.
alphas	Vektor yang berisi semua nilai alpha yang lebih besar dari nol yang melambangkan *support vector.*
support_vectors	Sample data yang memiliki alpha lebih dari 0 yang digunakan untuk menentukan *hyperplane.*
sv_labels	Label target y yang sesuai dengan *support vectors*
sv_indices	Indeks asli dari *support vectors* dalam dataset
w (weight)	Vektor bobot yang digunakan untuk menghitung kernel linear.
b (bias)	Nilai skalar yang menggeser *hyperplan* ke atas dan ke bawah.
errors	Vektor yang menyimpan nilai error untuk setiap sampel.
K (Kernel matrix)	Cached yang menyimpan semua hasil K antara setiap pasangan data pelatihan, dihitung sekali di awal untuk menghindari perhitungan berulang.

SVM dapat digunakan untuk mengklasifikasikan data linear maupun non-linear. Data non-linear dapat diklasifikasikan dengan menggunakan fungsi kernel. Fungsi kernel merupakan fungsi yang digunakan untuk memetakan data ke dimensi lebih tinggi. Terdapat empat jenis kernel yang diimplementasikan yaitu linear, polynomial, rbf, dan sigmoid. Kernel linear merupakan kernel default, tidak menggunakan transformasi dan hanya menerapkan *dot product.* Jenis ini cocok untuk data linear. Kernel *polynomial* merupakan kernel yang memetakan data hingga ruang fitur derajat d. Jenis ini cocok digunakan untuk data yang berbentuk lengkungan, parabola, atau elips. Kernel RBF merupakan fungsi kernel yang mentransformasikan data ke ruang berdimensi tak terhingga. Kernel sigmoid merupakan fungsi kernel yang memetakan data dengan tanh. Jenis ini jarang digunakan karena performanya kurang baik.

```
def kernel(self, x1, x2):
        # menghitung kernel matrix

        if self.kernel_type == 'linear':
            return np.dot(x1, x2.T)

        elif self.kernel_type == 'polynomial':
            d = self.kernel_param
            return (np.dot(x1, x2.T) + 1) ** d

        elif self.kernel_type == 'rbf':
            gamma = self.kernel_param
            if x1.ndim == 1:
                x1 = x1.reshape(1, -1)
            if x2.ndim == 1:
                x2 = x2.reshape(1, -1)

            xx = np.sum(x1**2, axis=1).reshape(-1, 1)
            yy = np.sum(x2**2, axis=1).reshape(-1, 1)
            xy = np.dot(x1, x2.T)
            sq_dist = xx + yy - 2*xy
            return np.exp(-gamma * sq_dist)

        elif self.kernel_type == 'sigmoid':
            return np.tanh(np.dot(x1, x2.T) + 1)

        else:
            raise ValueError(f"Kernel '{self.kernel_type}' tidak didukung")
```

Sequential Minimal Optimization atau SMO merupakan algoritma untuk menyelesaikan masalah optimasi kuadratik pada SVM. Fungsi solve_qp_problem bertujuan untuk menemukan semua alpha yang memenuhi kondisi Karush-Kuhn-Tucker (KTT). KTT merupakan teori optimasi yang digunakan untuk memastikan solusi program optimasi nonlinear sudah optimal. Dalam memastikan solusi memenuhi KTT dapat dilakukan dengan dua pencarian yaitu *examine all* dan *non-bound. Examine all* artinya algoritma akan melakukan *loop* melalui semua sampel data, sedangkan *non-bound* artinya algoritma hanya *loop* alpha yang berada di batas margin.

```
def solve_qp_problem(self):
        n_samples = len(self.y)
        self.alphas = np.zeros(n_samples)
        self.b = 0.0

        #  error cache
        self.errors = -self.y.copy()

        # main SMO loop
        num_changed = 0
        examine_all = True
        iteration = 0

        while (num_changed > 0 or examine_all) and iteration < self.max_iter:
            num_changed = 0

            if examine_all:
                # loop semua data points
                for i in range(n_samples):
                    num_changed += self.examine_example(i)
```

```
            else:
                # loop hanya non-bound alphas
                # ini adalah support vectors aktif
                non_bound_idx = np.where((self.alphas > self.tol) &
(self.alphas < self.C - self.tol))[0]
                for i in non_bound_idx:
                    num_changed += self.examine_example(i)

            iteration += 1

            if examine_all:
                examine_all = False
            elif num_changed == 0:
                examine_all = True

        return self.alphas
```

Pada fungsi take_step, dua pengali lagrange alpha1 dan alpha2 akan dipotimalkan. Kita akan menghitung L batas bawah dan H batas atas serta menghitung eta (laju perubahan) yang merupakan turunan kedua dari fungsi tujuan dual. Nilai alpha optimal ini nantinya akan digunakan untuk menentukan *support_vector* dan memprediksi fungsi keputusan f(x) untuk klasifikasi.

```
def take_step(self, i1, i2):
        if i1 == i2:
            return 0

        alpha1_old = self.alphas[i1]
        alpha2_old = self.alphas[i2]
        y1 = self.y[i1]
        y2 = self.y[i2]
        E1 = self.errors[i1]
        E2 = self.errors[i2]

        # Compute L and H
        if y1 != y2:
            L = max(0, alpha2_old - alpha1_old)
            H = min(self.C, self.C + alpha2_old - alpha1_old)
        else:
            L = max(0, alpha1_old + alpha2_old - self.C)
            H = min(self.C, alpha1_old + alpha2_old)

        if abs(L - H) < 1e-10:
            return 0

        # Hitung eta
        k11 = self.K[i1, i1]
        k12 = self.K[i1, i2]
        k22 = self.K[i2, i2]
        eta = 2 * k12 - k11 - k22

        if eta >= 0:
            return 0

        # Update alpha2
        alpha2_new = alpha2_old - (y2 * (E1 - E2)) / eta
        if alpha2_new > H:
            alpha2_new = H
```

```
        elif alpha2_new < L:
            alpha2_new = L

        if abs(alpha2_new - alpha2_old) < 1e-5:
            return 0

        # Update alpha1
        alpha1_new = alpha1_old + y1 * y2 * (alpha2_old - alpha2_new)

        # Update bias (b)
        b1 = self.b - E1 - y1 * (alpha1_new - alpha1_old) * k11 - y2 *
(alpha2_new - alpha2_old) * k12
        b2 = self.b - E2 - y1 * (alpha1_new - alpha1_old) * k12 - y2 *
(alpha2_new - alpha2_old) * k22

        b_old = self.b # Simpan b lama untuk update error cache
        if 0 < alpha1_new < self.C:
            self.b = b1
        elif 0 < alpha2_new < self.C:
            self.b = b2
        else:
            self.b = (b1 + b2) / 2

        # Simpan alpha baru
        self.alphas[i1] = alpha1_new
        self.alphas[i2] = alpha2_new

        delta_alpha1 = alpha1_new - alpha1_old
        delta_alpha2 = alpha2_new - alpha2_old
        delta_b = self.b - b_old

        # Error baru = Error lama + perubahan kontribusi support vector +
perubahan bias
        self.errors += y1 * delta_alpha1 * self.K[i1, :] + y2 * delta_alpha2 *
self.K[i2, :] + delta_b

        if 0 < alpha1_new < self.C:
            self.errors[i1] = 0
        if 0 < alpha2_new < self.C:
            self.errors[i2] = 0

        return 1
```

Algoritma SVM didesain untuk klasifikasi biner. Untuk menangani kasus klasifikasi dengan lebih dari dua kelas (*multiclass*), implementasi ini menggunakan strategi *One-vs-All*. Pendekatan ini diimplementasikan dalam kelas `MulticlassSVM`.

Strategi ini bekerja dengan cara melatih sejumlah k model SVM biner, di mana k adalah jumlah kelas unik pada label target. Dalam kode, proses ini dilakukan pada method fit. Algoritma akan melakukan iterasi pada setiap kelas unik, mengubah label menjadi masalah biner (target vs bukan target), dan melatih model `BinarySVM` terpisah untuk setiap kelas tersebut.

```
# Potongan kode dari method fit() pada kelas MulticlassSVM
for class_label in self.classes:
    # Target label diubah menjadi 1, label lain menjadi -1
    y_binary = np.where(y == class_label, 1, -1)
```

```
# Inisialisasi model biner baru
model = BinarySVM(
    C=self.C,
    kernel=self.kernel,
    kernel_param=self.kernel_param,
    tol=self.tol,
    max_iter=self.max_iter
)

# Latih model dan simpan ke dalam list models
model.fit(X, y_binary)
self.models.append(model)
```

Proses prediksi dilakukan dengan memanfaatkan model-model yang telah dilatih untuk menghasilkan keputusan kelas akhir. Implementasi ini terbagi menjadi dua tahap: perhitungan skor keputusan dan penentuan kelas akhir.

1. Perhitungan Skor Keputusan (Decision Score)
   Sebelum menentukan label kelas, setiap model biner perlu menghitung skor keputusan *f(x)* yang merepresentasikan jarak data ke *hyperplane*. Implementasi perhitungan ini terdapat pada method `predict_score` di kelas `BinarySVM`.

```
# Potongan kode method predict_score() pada kelas BinarySVM
def predict_score(self, X):
    if self.kernel_type == 'linear' and self.w is not None:
        # Perhitungan skor untuk kernel linear: w.x + b
        return np.dot(X, self.w) + self.b
    else:
        # Perhitungan skor untuk kernel non-linear menggunakan support
vectors
        # f(x) = sum(alpha_i * y_i * K(x, x_i)) + b
        K_test = self.kernel(X, self.support_vectors)
        scores = np.sum(self.alphas * self.sv_labels * K_test, axis=1)
+ self.b
        return scores
```

2. Penentuan Kelas Akhir (Voting)
   Untuk kasus multiclass, prediksi dilakukan dengan mekanisme voting berdasarkan skor tertinggi. Data uji dimasukkan ke seluruh model biner yang ada, dan skor keputusannya dikumpulkan. Kelas akhir ditentukan menggunakan fungsi `argmax`, yaitu mengambil indeks kelas yang memiliki skor keputusan paling positif (paling yakin).

```
# Potongan kode method predict() pada kelas MulticlassSVM
def predict(self, X):
    X = np.array(X)
    n_samples = X.shape[0]
    n_classes = len(self.classes)
    scores = np.zeros((n_samples, n_classes))

    # Kumpulkan skor dari setiap model biner
    for i, model in enumerate(self.models):
        scores[:, i] = model.predict_score(X)
```

```
    # Ambil indeks dengan skor tertinggi sebagai prediksi kelas
    predicted_indices = np.argmax(scores, axis=1)
    return self.classes[predicted_indices]
```

D. Proses *Data Cleaning* dan *Data Preprocessing*

Tahapan ini krusial untuk menjamin kualitas data sebelum masuk ke pemodelan. Berdasarkan eksplorasi data (*Exploratory Data Analysis*) dan eksperimen yang dilakukan, berikut adalah langkah-langkah detail yang diterapkan:

1. Data Cleaning dan Pengecekan Integritas Data
   Langkah pertama difokuskan pada validitas data mentah untuk memastikan tidak ada anomali logis.
   a. Pengecekan Missing Value & Duplikasi
      Berdasarkan pemeriksaan awal, dataset latih tidak memiliki *missing value* maupun baris duplikat (*duplicate rows*). Namun, sebagai langkah preventif untuk data uji (*test set*), kami tetap menyiapkan potongan kote imputasi menggunakan nilai Median.

```
# Imputasi Median untuk data latih dan uji
num_cols = X_train_raw.select_dtypes(include=np.number).columns
for col in num_cols:
    med = X_train_raw[col].median()
    X_train_raw[col] = X_train_raw[col].fillna(med)
    X_test_raw[col] = X_test_raw[col].fillna(med)
```

   b. Validasi Logika Akademik
      Dilakukan penyaringan terhadap anomali data yang tidak masuk akal secara akademik. Hasil pengecekan menunjukkan 0 baris error untuk kondisi berikut:
      - Jumlah SKS lulus (Approved) > SKS diambil (Enrolled).
      - Mahasiswa "Hantu" (SKS Enrolled 0, tapi memiliki nilai Evaluasi/Approved > 0).
      - Mahasiswa lulus tanpa mengikuti evaluasi.

```
# Approved tidak boleh lebih besar dari Enrolled (Tidak mungkin
jumlah matkul lulus lebih dari jumlah matkul diambil)

error1_sem1 = df_train[df_train['Curricular units 1st sem
(approved)'] > df_train['Curricular units 1st sem (enrolled)']]
error1_sem2 = df_train[df_train['Curricular units 2nd sem
(approved)'] > df_train['Curricular units 2nd sem (enrolled)']]
print(f"Approved > Enrolled (Sem 1): {len(error1_sem1)} row")
print(f"Approved > Enrolled (Sem 2): {len(error1_sem2)} row")

# Enrolled 0, tapi Evaluations > 0 atau Approved > 0
error2_sem1 = df_train[(df_train['Curricular units 1st sem

(enrolled)'] == 0) & ((df_train['Curricular units 1st sem
```

```
(evaluations)'] > 0) | (df_train['Curricular units 1st sem
(approved)'] > 0))]
error2_sem2 = df_train[(df_train['Curricular units 2nd sem
(enrolled)'] == 0) & ((df_train['Curricular units 2nd sem
(evaluations)'] > 0) | (df_train['Curricular units 2nd sem
(approved)'] > 0))]
print(f"Ghost Student (Sem 1): {len(error2_sem1)} row")
print(f"Ghost Student (Sem 2): {len(error2_sem2)} row")

# Approved > 0, tapi Evaluations = 0

error3_sem1 = df_train[(df_train['Curricular units 1st sem
(approved)'] > 0) & (df_train['Curricular units 1st sem
(evaluations)'] == 0)]
error3_sem2 = df_train[(df_train['Curricular units 2nd sem
(approved)'] > 0) & (df_train['Curricular units 2nd sem
(evaluations)'] == 0)]
print(f"Lulus tanpa ikut evaluasi (Sem 1): {len(error3_sem1)}
row")
print(f"Lulus tanpa ikut evaluasi (Sem 2): {len(error3_sem2)}
row")
```

Karena tidak ditemukan anomali logis, tidak ada penghapusan baris yang dilakukan pada tahap ini.

c. Penanganan Outlier

Berdasarkan visualisasi distribusi data menggunakan Boxplot, teridentifikasi adanya nilai-nilai outlier (titik-titik di luar rentang kuartil) pada beberapa fitur numerik, misalnya pada fitur Usia Saat Mendaftar (Age at enrollment) dan Nilai Kualifikasi Sebelumnya (Previous qualification grade).

Namun setelah melakukan diskusi, diputuskan bahwa data outlier ini tetap dipertahankan dan tidak dihapus. Alasannya adalah data tersebut merepresentasikan variasi demografi mahasiswa yang valid secara kontekstual (misalnya mahasiswa paruh baya atau mahasiswa dengan nilai akademik luar biasa), bukan merupakan kesalahan input (error).

2. Feature Engineering

Untuk meningkatkan akurasi model, dilakukan ekstraksi fitur baru yang mampu menangkap pola perilaku mahasiswa secara lebih mendalam.t:

- Tren & Rata-rata Akademik

  Dibuat fitur `Grade_Mean` (rata-rata nilai semester 1 dan 2) dan `Grade_Trend` untuk melihat kenaikan atau penurunan prestasi mahasiswa. Selain itu, rasio kelulusan dihitung melalui fitur `Approval_Rate_S1` dan `Approval_Rate_S2`.

```
df['Grade_Mean'] = (df['Curricular units 1st sem (grade)'] +
df['Curricular units 2nd sem (grade)']) / 2
df['Grade_Trend'] = df['Curricular units 2nd sem (grade)'] -
df['Curricular units 1st sem (grade)']
```

```
# Approval Rates (Rasio Kelulusan)
df['Approval_Rate_S1'] = df['Curricular units 1st sem
(approved)'] / (df['Curricular units 1st sem (enrolled)'] +
1e-9)
df['Approval_Rate_S2'] = df['Curricular units 2nd sem
(approved)'] / (df['Curricular units 2nd sem (enrolled)'] +
1e-9)
df['Approval_Rate_Mean'] = (df['Approval_Rate_S1'] +
df['Approval_Rate_S2']) / 2
```

- Indikator Risiko & Partisipasi
  Fitur `Failures_Total` dibuat untuk mengakumulasi kegagalan SKS di kedua semester. Tingkat keaktifan mahasiswa diukur melalui `Eval_Participation` (rasio evaluasi yang diikuti berbanding SKS yang diambil).

```
# Total SKS gagal (Enrolled - Approved)df['Failures_Total'] =
(df['Curricular units 1st sem (enrolled)'] - df['Curricular
units 1st sem (approved)']) + (df['Curricular units 2nd sem
(enrolled)'] - df['Curricular units 2nd sem (approved)'])

# Participation (Evaluations / Enrolled)
df['Eval_Participation'] = (df['Curricular units 1st sem
(evaluations)'] + df['Curricular units 2nd sem (evaluations)'])
/ (df['Curricular units 1st sem (enrolled)'] + df['Curricular
units 2nd sem (enrolled)'] + 1e-9)
```

- Scoring Risiko Finansial
  Berbeda dengan pendekatan biner sederhana, variabel `Financial_Risk` dihitung menggunakan pembobotan khusus: status hutang (bobot 3), keterlambatan SPP (bobot 4), dan ketiadaan beasiswa (bobot 2) untuk mengkuantifikasi tekanan ekonomi secara lebih presisi.

```
# Financial Pressure Score (Bobot: Hutang=3, SPP Nunggak=4,
Beasiswa=-2)

df['Financial_Risk'] = ((~df['Scholarship
holder'].astype(bool)).astype(int) * 2 + df['Debtor'] * 3 +
(~df['Tuition fees up to date'].astype(bool)).astype(int) * 4)
```

- Fitur Interaksi
  Fitur `Fin_Performance` dibuat dari interaksi antara skor risiko finansial dan tingkat ketidaklulusan, bertujuan menangkap bagaimana tekanan ekonomi berdampak langsung pada performa akademik.

```
df['Fin_Performance'] = df['Financial_Risk'] * (1 -
df['Approval_Rate_Mean'])
```

3. Feature Selection

Proses seleksi dilakukan secara bertahap untuk mengurangi dimensi dan membuang fitur yang dianggap tidak relevan (Noise).

- Recursive Feature Elimination with Cross-Validation (RFECV)
  Alih-alih hanya menggunakan korelasi sederhana, seleksi fitur dilakukan menggunakan metode RFE dengan estimator *LinearSVC (Support Vector Classifier)*. Algoritma ini secara iteratif memangkas fitur yang paling lemah kontribusinya dan menentukan jumlah fitur terbaik yang dianggap paling berpengaruh terhadap prediksi, kemudian menyimpan fitur-fitur tersebut.

```
# Seleksi fitur otomatis menggunakan RFECV
selector = RFECV(
    estimator=LinearSVC(dual=False, max_iter=2000,
class_weight='balanced'),
    step=1,
    cv=StratifiedKFold(5),
    scoring='f1_macro',
    n_jobs=-1
)
selector.fit(X_train_clean, y_train_final)
```

- Principal Component Analysis (PCA)
  Dengan memanfaatkan sifat dataset sebagai matrix, nilai eigen dari tiap fitur/kolom dapat ditemukan, seleksi fitur didasarkan pada nilai tersebut. Pada akhirnya, PCA tidak diimplementasikan dan digantikan dengan RFE.

4. Data Transformation
   Mengingat banyaknya variasi pada data kategorikal, dilakukan transformasi khusus untuk mencegah dimensi yang terlalu besar.
   - One-Hot Encoding
     Seluruh fitur kategorikal (termasuk yang telah direduksi) ditransformasi menggunakan *One-Hot Encoding* untuk menghasilkan representasi biner yang lebih mudah dipahami oleh model.
   - Scalling
     Seluruh fitur numerik dinormalisasi menggunakan StandardScaler agar memiliki rata-rata 0 dan standar deviasi 1.
   - Power Transformer (yeo-johnson)
     Seluruh fitur numerik bentuknya berubah menjadi Distribusi Normal.
   - Reduksi Kardinalitas
     Untuk kolom dengan banyak sekali data unik yang membuat model menerima terlalu ragam informasi, maka dilakukan reduksi kardinalitas untuk mengklasifikasikan data yang unik tetapi sedikit sebagai 1 kelas “other”.

5. Imbalanced Data Handling
   Karena distribusi kelas target yang tidak seimbang (didominasi oleh kelas *Graduate*), teknik SMOTE (Synthetic Minority Over-sampling Technique), spesifiknya varian

Borderline-SMOTE, diterapkan pada data latih. Hal ini bertujuan menyeimbangkan jumlah sampel antar kelas agar model tidak bias memprediksi kelas mayoritas saja.

6. Hyperparameter Tuning
   Implementasi dilakukan hanya pada SVM, hyperparameter tuning dilakukan untuk mencari kombinasi parameter terbaik untuk model, settingan model yang akan memberikan matrix score paling baik. Pemilihan nilai parameter dilakukan secara acak pada rentang dan opsi nilai yang tersedia.

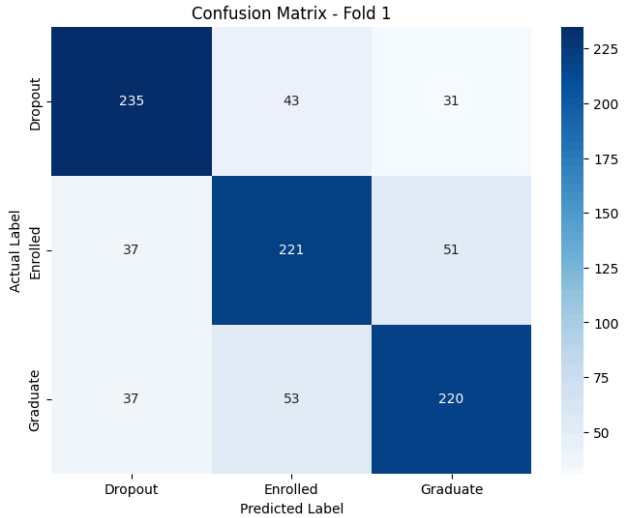
E. Perbandingan Performa Algoritma Hasil Implementasi dengan *Preexisting Library*

Library yang kami gunakan sebagai perbandingan performa adalah implementasi algoritma pada *library* Scikit Learn dan berikut adalah hasil perbandingan performa baik dari segi akurasi maupun dari segi waktu untuk masing-masing algoritma pembelajaran mesin yang kami implementasi.

1. Decision Tree Learning

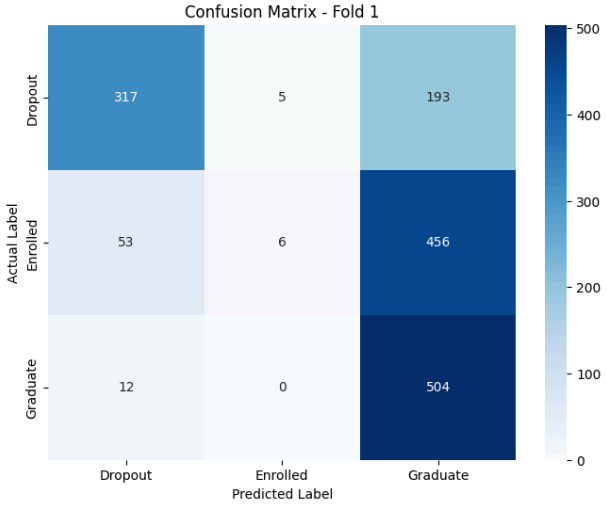
Hasil performa model yang diimplementasi dari *scratch* lebih buruk dari model dari *library* Scikit-Learn. Model dari Scikit-Learn dapat mengenali ketiga kelas target secara relatif seimbang dengan *accuracy* 0.7439 sedangkan model yang diimplementasi dari *scratch* tidak dapat mengenali kelas Enrolled dan memiliki *accuracy* sebesar 0.5427 . keterbatasan waktu pelatihan menyebabkan parameter penting seperti max_depth dan min_samples_split harus dibatasi agar proses berjalan lebih cepat sehingga mengakibatkan penurunan performa yang cukup signifikan dibandingkan Scikit-Learn.

1. Hasil Scikit



```
Detailed Classification Report (Fold 1):
              precision    recall  f1-score   support

     Dropout       0.76      0.76      0.76       309
    Enrolled       0.70      0.72      0.71       309
    Graduate       0.73      0.71      0.72       310

    accuracy                           0.73       928
   macro avg       0.73      0.73      0.73       928
weighted avg       0.73      0.73      0.73       928

------------------------------------------------------------
Mean Accuracy across 5 folds: 0.7439 (+/- 0.0146)
```

2. Hasil implementasi

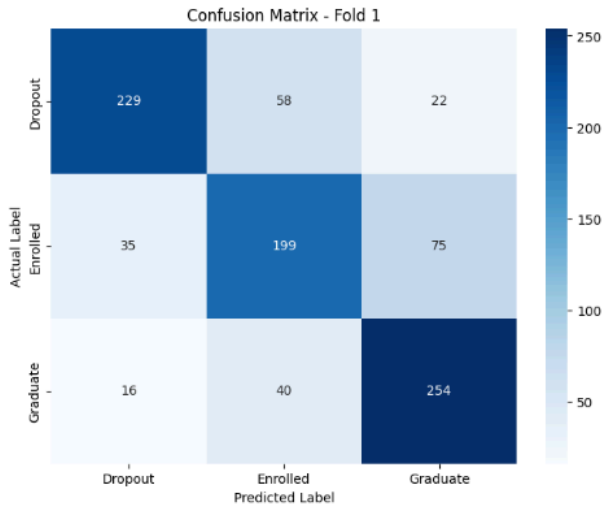



```
Detailed Classification Report (Fold 1):
              precision    recall  f1-score   support

     Dropout       0.83      0.62      0.71       515
    Enrolled       0.55      0.01      0.02       515
    Graduate       0.44      0.98      0.60       516

    accuracy                           0.53      1546
   macro avg       0.60      0.53      0.44      1546
weighted avg       0.60      0.53      0.44      1546

------------------------------------------------------------
Mean Accuracy across 3 folds: 0.5427 (+/- 0.0059)
```

2. Logistic Regression

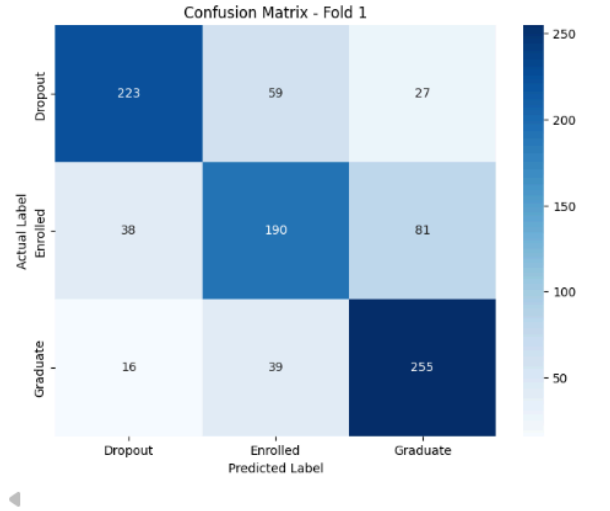
Hasil performa model antara menggunakan *library* Scikit-Learn dan implementasi dari *scratch* terlihat memiliki perbedaan *accuracy.* Namun selisih performa tidak terlalu besar. Hal ini menunjukkan bahwa algoritma yang diimplementasikan mampu mengklasifikasikan kelas dengan cukup baik.

1. Hasil Scikit




```
Detailed Classification Report (Fold 1):
              precision    recall  f1-score   support

     Dropout       0.82      0.74      0.78       309
    Enrolled       0.67      0.64      0.66       309
    Graduate       0.72      0.82      0.77       310

    accuracy                           0.73       928
   macro avg       0.74      0.73      0.73       928
weighted avg       0.74      0.73      0.73       928

----------------------------------------------------------------
Mean Accuracy across 5 folds: 0.7333 (+/- 0.0140)
```

2. Hasil implementasi



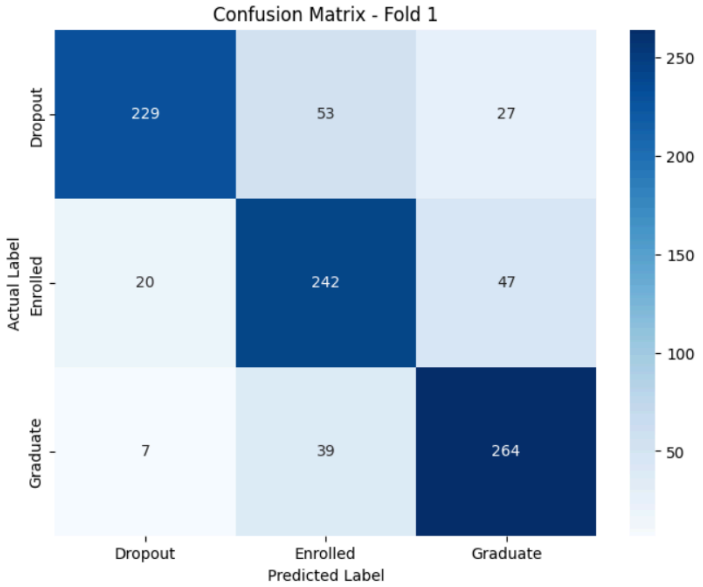

```
Detailed Classification Report (Fold 1):
              precision    recall  f1-score   support

     Dropout       0.81      0.72      0.76       309
    Enrolled       0.66      0.61      0.64       309
    Graduate       0.70      0.82      0.76       310

    accuracy                           0.72       928
   macro avg       0.72      0.72      0.72       928
weighted avg       0.72      0.72      0.72       928

----------------------------------------------------------------
Mean Accuracy across 5 folds: 0.7229 (+/- 0.0130)
```

3. SVM

SVM hasil prediksi Scikit-Learn terlihat memprediksi data dengan lebih terdistribusi dan memiliki f1-score yang cukup seimbang. Pada model hasil implementasi, data masih cukup bias dan cenderung memberikan prediksi pada kelas yang datanya paling banyak di train. Terdapat perbedaan akurasi yang masih dapat ditoleransi secara keseluruhan.

1. Hasil Scikit-learn

REPORT FOR FOLD 1



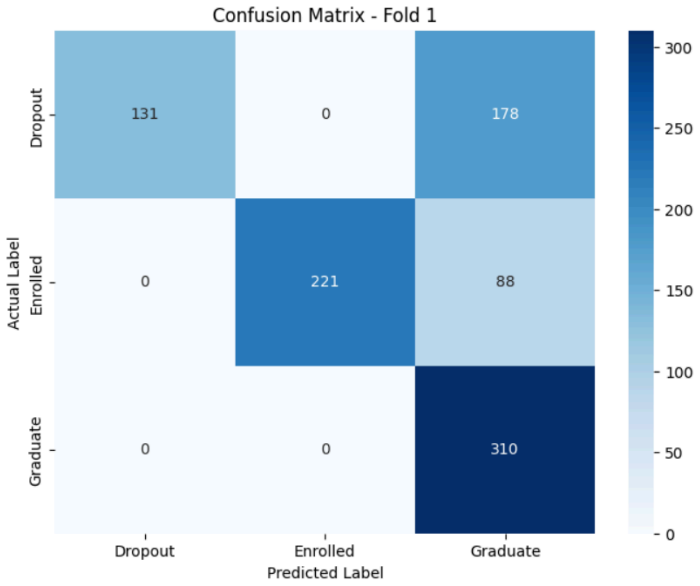
Detailed Classification Report (Fold 1):

	precision	recall	f1-score	support
Dropout	0.89	0.74	0.81	309
Enrolled	0.72	0.78	0.75	309
Graduate	0.78	0.85	0.81	310
accuracy			0.79	928
macro avg	0.80	0.79	0.79	928
weighted avg	0.80	0.79	0.79	928

Mean Accuracy across 5 folds: 0.7993 (+/- 0.0201)

2. Hasil implementasi

REPORT FOR FOLD 1



Detailed Classification Report (Fold 1):

	precision	recall	f1-score	support
Dropout	1.00	0.42	0.60	309
Enrolled	1.00	0.72	0.83	309
Graduate	0.54	1.00	0.70	310
accuracy			0.71	928
macro avg	0.85	0.71	0.71	928
weighted avg	0.85	0.71	0.71	928

Mean Accuracy across 5 folds: 0.7415 (+/- 0.0159)

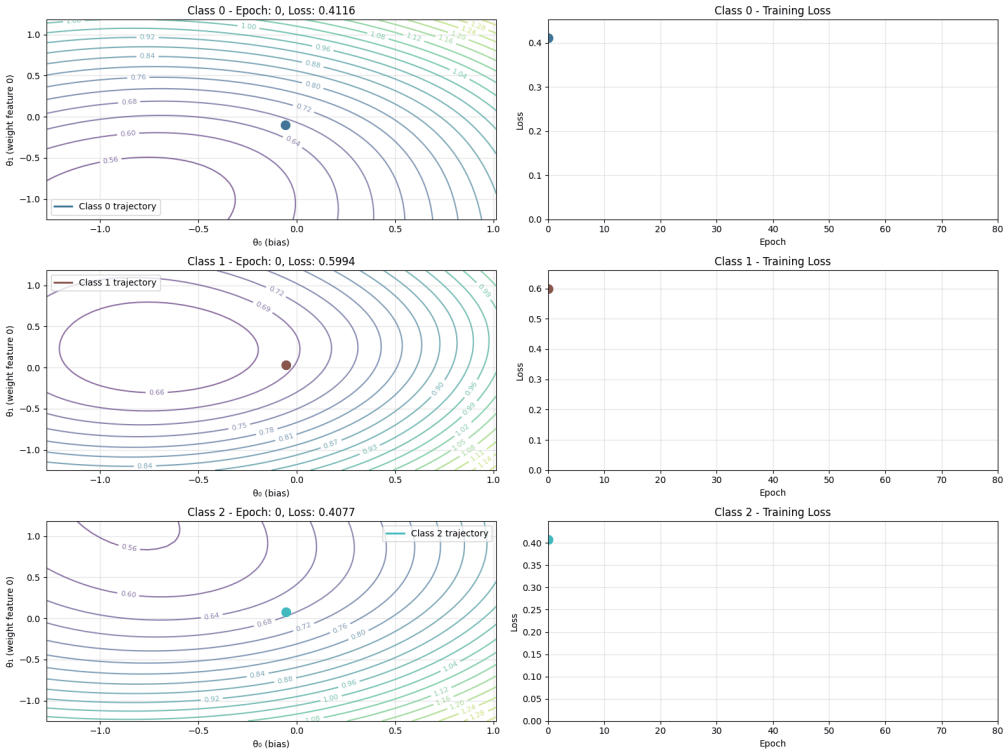
F. Implementasi Bonus

1. Tree DTL

   Visualisasi Tree pada hasil data pelatihan dengan kedalaman tree sekitar 30. Setiap node memiliki nilai threshold dan setiap leaf memiliki kelas klasifikasi dari target.

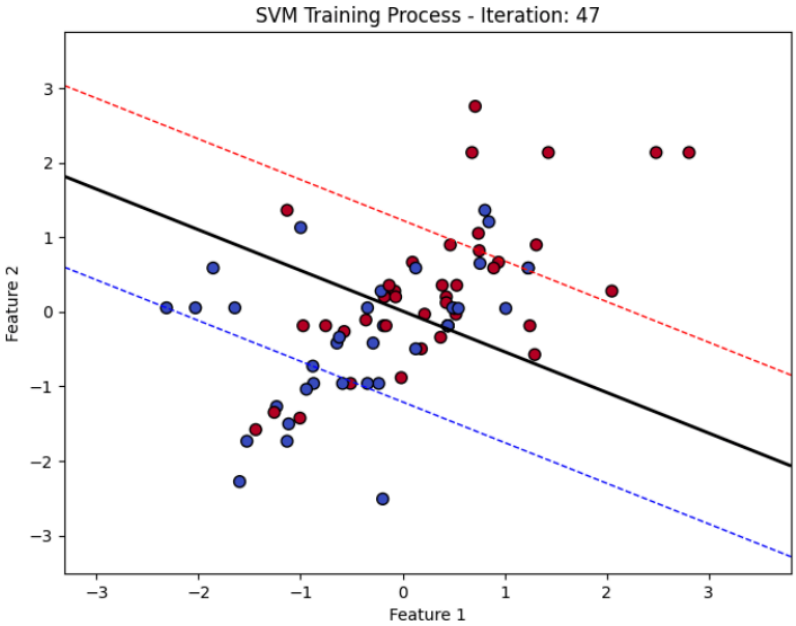
2. Gif LogReg

   Visualisasi GIF yang menampilkan garis kontur fungsi loss dan lintasan parameter selama training.



3. GifSVM

   Visualisasi GIF saat melakukan iterasi pada demo proses training model dengan iterasi sebanyak max_iter kali.



G. Pembagian Tugas

Berikut pembagian tugas kelompok kami:

NIM	Nama	Tugas
13523050	Mayla Yaffa Ludmilla	Implementasi DTL
13523056	Salman Hanif	Implementasi bonus DTL + Data Preprocessing, Modelling
13523058	Noumisyifa Nabila Nareswari	Implementasi Logistic Regression
13523072	Diyah Susan Nugrahani	Implementasi Logistic Regression + Implementasi SVM
13523108	Muhammad Edo Raduputu Aprima	Implementasi SVM + Data Preprocessing

H. Referensi

Bhuva, L. (2025, Februari 25). *Mini-Batch Gradient Descent: A Comprehensive Guide*. Medium. https://medium.com/@lomashbhuva/mini-batch-gradient-descent-a-comprehensive-guide-ba27a6dc4863

Khodra, M. L. (n.d.). *Logistic Regression*. KK IF - Teknik Informatika- STEI ITB, Institut Teknologi Bandung.

Maulidevi, N. U. (n.d.). *Modul: Decision Tree Learning (DTL)*. KK IF - Teknik Informatika- STEI ITB, Institut Teknologi Bandung.

Platt, John C. 1998. "Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines," 21. https://ajdillhoff.github.io/notes/sequential_minimal_optimization/

Ruskanda, F. Z. (n.d.). *Modul: Supervised Learning SVM*. KK IF - Teknik Informatika- STEI ITB, Institut Teknologi Bandung.