

IF3170 Inteligensi Artifisial

Tugas Besar 2

Implementasi Algoritma Pembelajaran Mesin



Disusun Oleh:

13523001 Wardatul Khoiroh

13523018 Raka Dafa

13523049 M. Fithra

13523074 Ahsan Malik Al Farisi

13523118 Farrel Athalla Putra

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika - Komputasi

Institut Teknologi Bandung

2025

Daftar Isi

Daftar Isi.....	2
BAB 1: Implementasi Decision Tree learning.....	3
BAB 2: Implementasi Logistic Regression.....	14
BAB 3: Implementasi SVM.....	15
BAB 4: Tahap Cleaning dan Preprocessing.....	16
BAB 5: Perbandingan Hasil Prediksi.....	17
Lampiran.....	18
Referensi.....	19

BAB I

Implementasi Decision Tree learning

Implementasi Decision Tree Learning pada program ini menggunakan pendekatan algoritma C4.5, yaitu pengembangan dari ID3 yang mampu menangani isu-isu di DTL seperti fitur kontinu, missing values, serta menggunakan Gain Ratio sebagai kriteria pemilihan fitur terbaik. Program ini dirancang menggunakan dua kelas utama yaitu:

1. Node sebagai struktur dasar pohon dimana merepresentasikan setiap simpul pada pohon keputusan, menyimpan informasi seperti indeks fitur yang digunakan untuk split, threshold (untuk fitur numerik), daftar anak (untuk fitur kategorikal), penanda leaf node, serta distribusi kelas sebagai fallback.
2. C45DecisionTree sebagai kelas yang mengatur seluruh proses pembelajaran pohon keputusan dimulai dari *training*, *split selection*, *entropy computation*, *Gain Ratio*, *tree building*, serta prediksi pada data baru. Beberapa parameter utama di kelas ini yaitu:
 - max_depth
 - min_samples_split
 - min_samples_leaf
 - criterion
 - random_state
 - class_weight

```
class C45DecisionTree:  
    def __init__(self, max_depth=None, min_samples_split=2, min_samples_leaf=1,  
criterion='entropy', random_state=None, class_weight=None):  
        self.max_depth = max_depth  
        self.min_samples_split = min_samples_split  
        self.min_samples_leaf = min_samples_leaf  
        self.criterion = criterion  
        self.random_state = random_state  
        self.class_weight = class_weight  
        self.tree = None  
        self.feature_names = None  
        self.classes = None  
        self.classes_ = None # Untuk sklearn compatibility  
        self.rng = None  
        self.sample_weights = None
```

```

        self.feature_importances_ = None # Untuk RFE

    def fit(self, X, y):
        if self.random_state is not None:
            self.rng = np.random.RandomState(self.random_state)
        else:
            self.rng = np.random.RandomState()
        if isinstance(X, pd.DataFrame):
            self.feature_names = list(X.columns)
            X = X.values
        else:
            self.feature_names = [f"X{i}" for i in range(X.shape[1])]

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

        self.classes = np.unique(y)
        self.classes_ = self.classes
        self.sample_weights = self._compute_sample_weights(y)
        self.tree = self.build_tree(X, y, depth=0)
        self.feature_importances_ = self._compute_feature_importance(X, y)
        return self

    def _compute_sample_weights(self, y):
        n_samples = len(y)
        weights = np.ones(n_samples, dtype=np.float64)

        if self.class_weight == 'balanced':
            unique_classes, counts = np.unique(y, return_counts=True)
            n_classes = len(unique_classes)

            # balanced weight = n_samples / (n_classes * n_samples_for_class)
            class_weights = {}
            for cls, count in zip(unique_classes, counts):
                class_weights[cls] = n_samples / (n_classes * count)

            for i, cls in enumerate(y):
                weights[i] = class_weights[cls]

        elif isinstance(self.class_weight, dict):
            for i, cls in enumerate(y):
                weights[i] = self.class_weight.get(cls, 1.0)

        return weights

    def is_continuous(self, x):
        try:
            if isinstance(x, pd.Series):
                x_clean = x.dropna().values
            else:
                x_clean = x[~np.isnan(x)] if np.issubdtype(x.dtype, np.number) else x

            if len(x_clean) == 0:
                return False

```

```

# Check if numeric
if not np.issubdtype(type(x_clean[0]), np.number):
    return False

unique_count = len(np.unique(x_clean))
return unique_count > 5
except:
    return False

def entropy(self, y):
    if weights is None:
        weights = np.ones(len(y))

    total_weight = np.sum(weights)
    if total_weight == 0:
        return 0

    unique_classes = np.unique(y)
    entropy_val = 0

    for cls in unique_classes:
        mask = y == cls
        weight_sum = np.sum(weights[mask])
        if weight_sum > 0:
            p = weight_sum / total_weight
            entropy_val -= p * np.log2(p)

    return entropy_val

def information_gain(self, X, y, feature_idx, threshold=None):
    if weights is None:
        weights = np.ones(len(y))

    parent_entropy = self.entropy(y, weights)

    mask = ~np.isnan(X[:, feature_idx])
    if mask.sum() == 0:
        return 0

    X_valid = X[mask]
    y_valid = y[mask]
    weights_valid = weights[mask]

    if threshold is not None:
        left_mask = X_valid[:, feature_idx] <= threshold
        right_mask = ~left_mask

        if left_mask.sum() == 0 or right_mask.sum() == 0:
            return 0
        splits = [
            (y_valid[left_mask], weights_valid[left_mask]),
            (y_valid[right_mask], weights_valid[right_mask])
        ]
    else:
        values = np.unique(X_valid[:, feature_idx])

```

```

        if len(values) < 2:
            return 0
        splits = [
            (y_valid[X_valid[:, feature_idx] == v],
             weights_valid[X_valid[:, feature_idx] == v])
            for v in values
        ]

        total_weight = np.sum(weights_valid)
        weighted_entropy = 0

        for split_y, split_weights in splits:
            if len(split_y) > 0:
                weight = np.sum(split_weights) / total_weight
                weighted_entropy += weight * self.entropy(split_y, split_weights)

        gain = parent_entropy - weighted_entropy
        return max(0, gain)

    def gain_ratio(self, X, y, feature_idx, threshold=None, weights=None):
        gain = self.information_gain(X, y, feature_idx, threshold, weights)

        if gain <= 1e-10:
            return 0

        mask = ~np.isnan(X[:, feature_idx])
        if mask.sum() == 0:
            return 0

        X_valid = X[mask]
        weights_valid = weights[mask] if weights is not None else np.ones(mask.sum())

        if threshold is not None:
            left_mask = X_valid[:, feature_idx] <= threshold
            right_mask = ~left_mask
            splits = [np.sum(weights_valid[left_mask]),
                      np.sum(weights_valid[right_mask])]
        else:
            values = np.unique(X_valid[:, feature_idx])
            splits = [np.sum(weights_valid[X_valid[:, feature_idx] == v]) for v in
                      values]

        total_weight = np.sum(weights_valid)
        split_info = 0

        for weight_sum in splits:
            if weight_sum > 0:
                proportion = weight_sum / total_weight
                split_info -= proportion * np.log2(proportion)

        # Avoid division by very small split_info (use small epsilon)
        if split_info < 1e-10:
            return gain * 0.5
        return gain / split_info

```

```

def find_best_split(self, X, y, weights):
    best_gain = -1
    best_feature = None
    best_threshold = None
    best_candidates = []

    n_features = X.shape[1]

    for feature_idx in range(n_features):
        x_feature = X[:, feature_idx]

        if np.all(np.isnan(x_feature)):
            continue

        if self.is_continuous(x_feature):
            mask_valid = ~np.isnan(x_feature)
            x_clean = x_feature[mask_valid]
            y_clean = y[mask_valid]
            weights_clean = weights[mask_valid]

            sorted_indices = np.argsort(x_clean)
            x_sorted = x_clean[sorted_indices]
            y_sorted = y_clean[sorted_indices]

            candidate_thresholds = []
            for i in range(len(y_sorted) - 1):
                if y_sorted[i] != y_sorted[i + 1]:
                    threshold = (x_sorted[i] + x_sorted[i + 1]) / 2
                    candidate_thresholds.append(threshold)

            if len(candidate_thresholds) > 100:
                step = len(candidate_thresholds) // 100
                candidate_thresholds = candidate_thresholds[::-step]

            for threshold in candidate_thresholds:
                if self.criterion == 'gain_ratio':
                    gain = self.gain_ratio(X, y, feature_idx, threshold, weights)
                else:
                    gain = self.information_gain(X, y, feature_idx, threshold,
weights)
                if gain > best_gain:
                    best_gain = gain
                    best_candidates = [(feature_idx, threshold)]
                elif gain == best_gain and gain > 0:
                    best_candidates.append((feature_idx, threshold))
            else:
                if self.criterion == 'gain_ratio':
                    gain = self.gain_ratio(X, y, feature_idx, None, weights)
                else:
                    gain = self.information_gain(X, y, feature_idx, None, weights)
                if gain > best_gain:
                    best_gain = gain
                    best_candidates = [(feature_idx, None)]
                elif gain == best_gain and gain > 0:
                    best_candidates.append((feature_idx, None))
        else:
            pass
    return best_feature, best_threshold, best_candidates

```

```

    if best_candidates:
        chosen_idx = self.rng.randint(0, len(best_candidates))
        best_feature, best_threshold = best_candidates[chosen_idx]

    return best_feature, best_threshold

def majority_class(self, y):
    if weights is None:
        return np.bincount(y).argmax()

    # Weighted majority class
    unique_classes = np.unique(y)
    class_weights = np.zeros(len(unique_classes))

    for i, cls in enumerate(unique_classes):
        class_weights[i] = np.sum(weights[y == cls])

    return unique_classes[np.argmax(class_weights)]

def build_tree(self, X, y, depth):
    if weights is None:
        weights = np.ones(len(y))
        if self.sample_weights is not None and len(self.sample_weights) >= len(y):
            weights = self.sample_weights[:len(y)]

    n_samples, n_features = X.shape
    n_classes = len(np.unique(y))

    node = Node()
    node.class_distribution = np.bincount(y, weights=weights,
minlength=len(self.classes))

    if (n_classes == 1 or
        n_samples < self.min_samples_split or
        (self.max_depth and depth >= self.max_depth)):
        node.is_leaf = True
        node.value = self.classes[self.majority_class(y)]
        return node

    best_feature, best_threshold = self.find_best_split(X, y, weights)

    if best_feature is None:
        node.is_leaf = True
        node.value = self.classes[self.majority_class(y, weights)]
        return node

    node.feature = best_feature
    node.threshold = best_threshold
    node.value = self.classes[self.majority_class(y, weights)]

    x_feature = X[:, best_feature]

    if best_threshold is not None:
        mask_valid = ~np.isnan(x_feature)
        left_mask = np.zeros(n_samples, dtype=bool)

```

```

right_mask = np.zeros(n_samples, dtype=bool)

left_mask[mask_valid] = x_feature[mask_valid] <= best_threshold
right_mask[mask_valid] = x_feature[mask_valid] > best_threshold

n_left = left_mask.sum()
n_right = right_mask.sum()
if n_left >= n_right:
    left_mask[~mask_valid] = True
else:
    right_mask[~mask_valid] = True

if left_mask.sum() >= self.min_samples_leaf:
    node.left = self.build_tree(X[left_mask], y[left_mask], depth + 1,
weights[left_mask])
else:
    node.left = Node()
    node.left.is_leaf = True
    node.left.value = self.classes[self.majority_class(y, weights)]
    node.left.class_distribution = node.class_distribution.copy()

if right_mask.sum() >= self.min_samples_leaf:
    node.right = self.build_tree(X[right_mask], y[right_mask], depth + 1,
weights[right_mask])
else:
    node.right = Node()
    node.right.is_leaf = True
    node.right.value = self.classes[self.majority_class(y, weights)]
    node.right.class_distribution = node.class_distribution.copy()

else:
    values = np.unique(x_feature[~np.isnan(x_feature)])

    for value in values:
        mask = x_feature == value
        if mask.sum() >= self.min_samples_leaf:
            node.children[value] = self.build_tree(X[mask], y[mask], depth + 1,
weights[mask])
        else:
            child = Node()
            child.is_leaf = True
            if mask.sum() > 0:
                child.value = self.classes[self.majority_class(y[mask],
weights[mask])]
                child.class_distribution = np.bincount(y[mask],
weights=weights[mask], minlength=len(self.classes))
            else:
                child.value = self.classes[self.majority_class(y, weights)]
                child.class_distribution = node.class_distribution.copy()
            node.children[value] = child

    if len(node.children) == 0:
        node.is_leaf = True
        node.value = self.classes[self.majority_class(y, weights)]

```

```

        return node

    def predict(self, X):
        if isinstance(X, pd.DataFrame):
            X = X.values
        return np.array([self.predict_sample(x, self.tree) for x in X])

    def predict_sample(self, x, node):
        if node.is_leaf:
            return node.value

        feature_val = x[node.feature]

        if np.isnan(feature_val) if isinstance(feature_val, (int, float)) else pd.isna(feature_val):
            if node.value is not None:
                return node.value
            if node.class_distribution is not None:
                return self.classes[np.argmax(node.class_distribution)]
            return self.classes[0]

        if node.threshold is not None:
            # Binary split
            if feature_val <= node.threshold:
                if node.left:
                    return self.predict_sample(x, node.left)
                else:
                    return node.value if node.value is not None else self.classes[0]
            else:
                if node.right:
                    return self.predict_sample(x, node.right)
                else:
                    return node.value if node.value is not None else self.classes[0]
        else:
            # Categorical feature
            if feature_val in node.children:
                return self.predict_sample(x, node.children[feature_val])
            else:
                # If value not seen in training, use majority class at this node
                return node.value if node.value is not None else self.classes[0]

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

        probs = np.zeros((len(X), len(self.classes_)))
        for i, x in enumerate(X):
            node = self._get_leaf_node(x, self.tree)
            if node and node.class_distribution is not None:
                total = np.sum(node.class_distribution)
                if total > 0:
                    probs[i] = node.class_distribution / total
                else:
                    probs[i] = np.ones(len(self.classes_)) / len(self.classes_)
            else:
                probs[i] = np.ones(len(self.classes_)) / len(self.classes_)


```

```

        return probs

    def _get_leaf_node(self, x, node):
        if node is None or node.is_leaf:
            return node

        feature_val = x[node.feature]

        if pd.isna(feature_val):
            return node

        if node.threshold is not None:
            if feature_val <= node.threshold:
                return self._get_leaf_node(x, node.left) if node.left else node
            else:
                return self._get_leaf_node(x, node.right) if node.right else node
        else:
            if feature_val in node.children:
                return self._get_leaf_node(x, node.children[feature_val])
            else:
                return node

    def _compute_feature_importance(self, X, y):
        n_features = X.shape[1]
        importances = np.zeros(n_features)

        def traverse(node, n_samples):
            if node is None or node.is_leaf:
                return

            if node.feature is not None:
                # Hitung information gain untuk split ini
                gain = self.information_gain(X, y, node.feature, node.threshold)
                importances[node.feature] += gain * n_samples

                # Rekursi ke children
                if node.threshold is not None:
                    if node.left:
                        n_left = np.sum((X[:, node.feature] <= node.threshold) &
( ~pd.isna(X[:, node.feature])))
                        traverse(node.left, n_left)
                    if node.right:
                        n_right = np.sum((X[:, node.feature] > node.threshold) &
( ~pd.isna(X[:, node.feature])))
                        traverse(node.right, n_right)
                else:
                    for value, child in node.children.items():
                        n_child = np.sum(X[:, node.feature] == value)
                        traverse(child, n_child)

            traverse(self.tree, len(X))

        # Normalisasi
        total = np.sum(importances)
        if total > 0:

```

```

importances = importances / total

return importances

def save_model(self, filepath):
    with open(filepath, 'wb') as f:
        pickle.dump({
            'tree': self.tree,
            'feature_names': self.feature_names,
            'classes': self.classes,
            'params': {
                'max_depth': self.max_depth,
                'min_samples_split': self.min_samples_split,
                'min_samples_leaf': self.min_samples_leaf,
                'criterion': self.criterion,
                'random_state': self.random_state,
                'class_weight': self.class_weight
            }
        }, f)
    print(f"✓ Model saved to: {filepath}")

def load_model(self, filepath):
    with open(filepath, 'rb') as f:
        data = pickle.load(f)
    self.tree = data['tree']
    self.feature_names = data['feature_names']
    self.classes = data['classes']
    self.classes_ = self.classes
    params = data['params']
    self.max_depth = params['max_depth']
    self.min_samples_split = params['min_samples_split']
    self.min_samples_leaf = params['min_samples_leaf']
    self.criterion = params.get('criterion', 'entropy')
    self.random_state = params.get('random_state', None)
    self.class_weight = params.get('class_weight', None)
    print(f"✓ Model loaded from: {filepath}")
    return self

def get_params(self, deep=True):
    """Get parameters for this estimator - for sklearn compatibility"""
    return {
        'max_depth': self.max_depth,
        'min_samples_split': self.min_samples_split,
        'min_samples_leaf': self.min_samples_leaf,
        'criterion': self.criterion,
        'random_state': self.random_state,
        'class_weight': self.class_weight
    }

def set_params(self, **params):
    """Set parameters for this estimator - for sklearn compatibility"""
    for key, value in params.items():
        setattr(self, key, value)
    return self

```

```

def visualize_tree(self, max_depth=3, save_path=None, figsize=(15, 10)):
    if self.tree is None:
        print("Error: Model not trained yet!")
        return

    fig, ax = plt.subplots(figsize=figsize)
    ax.axis('off')

    positions = {}
    self.calc_positions(self.tree, 0, 0, 1, positions, max_depth)

    self.draw_node(ax, self.tree, positions, max_depth)

    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"✓ Tree saved to: {save_path}")
    plt.show()

def calc_positions(self, node, depth, left, right, positions, max_depth):
    if node is None or (max_depth and depth > max_depth):
        return 0

    if node.is_leaf or (max_depth and depth == max_depth):
        x = (left + right) / 2
        y = -depth
        positions[id(node)] = (x, y)
        return 1

    if node.threshold is not None:
        left_leaves = self.calc_positions(node.left, depth + 1, left,
                                           (left + right) / 2, positions, max_depth)
        right_leaves = self.calc_positions(node.right, depth + 1,
                                           (left + right) / 2, right, positions,
max_depth)
        total = left_leaves + right_leaves
    else:
        total = 0
        n_children = len(node.children)
        if n_children > 0:
            width = (right - left) / n_children
            for i, child in enumerate(node.children.values()):
                child_left = left + i * width
                child_right = left + (i + 1) * width
                total += self.calc_positions(child, depth + 1, child_left,
                                              child_right, positions, max_depth)

            x = (left + right) / 2
            y = -depth
            positions[id(node)] = (x, y)
            return max(total, 1)

    def draw_node(self, ax, node, positions, max_depth, parent_pos=None,
edge_label=""):
        if node is None or id(node) not in positions:

```

```

    return

    x, y = positions[id(node)]
    current_depth = abs(int(y))

    if max_depth and current_depth > max_depth:
        return

    if parent_pos:
        ax.plot([parent_pos[0], x], [parent_pos[1], y], 'k-', alpha=0.3,
    linewidth=1)
        if edge_label:
            mid_x, mid_y = (parent_pos[0] + x) / 2, (parent_pos[1] + y) / 2
            ax.text(mid_x, mid_y, edge_label, fontsize=8,
                    bbox=dict(boxstyle='round', pad=0.3, facecolor='yellow',
alpha=0.5))

    if node.is_leaf or (max_depth and current_depth == max_depth):
        label = f"Class:\n{node.value}"
        color = 'lightgreen'
    else:
        feat_name = self.feature_names[node.feature]
        if node.threshold is not None:
            label = f"{feat_name}\n≤ {node.threshold:.2f}"
        else:
            label = f"{feat_name}"
        color = 'lightblue'

    ax.text(x, y, label, ha='center', va='center', fontsize=9,
            bbox=dict(boxstyle='round', pad=0.5, facecolor=color,
                      edgecolor='black', linewidth=1.5))

    if not node.is_leaf and (not max_depth or current_depth < max_depth):
        if node.threshold is not None:
            if node.left:
                self.draw_node(ax, node.left, positions, max_depth, (x, y), "≤")
            if node.right:
                self.draw_node(ax, node.right, positions, max_depth, (x, y), ">")
        else:
            for value, child in node.children.items():
                self.draw_node(ax, child, positions, max_depth, (x, y), str(value))

def print_tree(self, max_depth=None):
    if self.tree is None:
        print("Model not trained!")
        return
    self.print_node(self.tree, "", True, 0, max_depth)

def print_node(self, node, prefix, is_last, depth, max_depth):
    if node is None or (max_depth and depth > max_depth):
        return

    connector = "└─ " if is_last else "├─ "

```

```

        print(f"{prefix}{connector}Class: {node.value}")
    else:
        feat_name = self.feature_names[node.feature]
        if node.threshold is not None:
            print(f"{prefix}{connector}{feat_name} ≤ {node.threshold:.2f}")
        else:
            print(f"{prefix}{connector}{feat_name}")

    if not node.is_leaf and (not max_depth or depth < max_depth):
        extension = "    " if is_last else "|    "

        if node.threshold is not None:
            if node.left:
                self.print_node(node.left, prefix + extension, False, depth + 1,
max_depth)
            if node.right:
                self.print_node(node.right, prefix + extension, True, depth + 1,
max_depth)
            else:
                children = list(node.children.items())
                for i, (value, child) in enumerate(children):
                    is_last_child = (i == len(children) - 1)
                    print(f"{prefix}{extension}'└─ ' if is_last_child else '|─
'}[{value}]")
                    self.print_node(child, prefix + extension +
("    " if is_last_child else "|    "),
True, depth + 1, max_depth)

class Node:
    def __init__(self):
        self.feature = None
        self.threshold = None
        self.value = None
        self.left = None
        self.right = None
        self.children = {}
        self.is_leaf = False
        self.class_distribution = None

```

Fitur utama implementasi ini mencakup kemampuan mendeteksi apakah sebuah fitur bersifat kontinu atau kategorikal, menghitung nilai entropi, menghitung information gain dan gain ratio, mencari threshold optimal untuk fitur numerik, dan membangun tree secara rekursif. Program juga mendukung visualisasi pohon, penyimpanan model, dan load model menggunakan pickle.

Fungsi/Metode	Penjelasan
is_continuous()	Menentukan apakah fitur adalah numerik dengan banyak nilai unik

entropy()	Menghitung ketidakpastian pada distribusi kelas.
information_gain()	Evaluasi kualitas split berdasarkan penurunan entropi.
gain_ratio()	Normalisasi IG untuk menghindari bias fitur dengan banyak nilai.
find_best_split()	Mencari fitur & threshold terbaik untuk membagi node.
build_tree()	Membangun tree secara rekursif dengan syarat-syarat C4.5.
predict_sample()	Melakukan traversing pohon hingga node daun.
predict(X)	Menjalankan prediksi untuk seluruh dataset.
save_model()	Menyimpan tree, fitur, dan parameter menggunakan pickle.
load_model()	Memuat kembali struktur pohon dan parameter.
visualize_tree()	Menggambar pohon hingga kedalaman tertentu.
print_tree()	Menampilkan struktur pohon dalam bentuk teks.
fit(X,y)	Melatih model dengan membangun pohon dari fitur dan label
predict_proba(X)	Probabilitas prediksi berupa proporsi distribusi kelas leaf

Secara keseluruhan, implementasi C4.5 ini memberikan fleksibilitas tinggi dan lebih unggul dibanding ID3 dalam menangani data nyata yang kompleks, namun tetap memiliki kekurangan seperti cenderung overfitting jika tidak diberi batasan kedalaman dan ukuran minimum sampel.

BAB II

Implementasi SVM

2.1 Implementasi Linear SVM

SVM adalah algoritma pembelajaran yang bertujuan untuk menemukan *hyperplane* pemisah optimal yang memaksimalkan margin dari data latih. SVM diperkenalkan pada tahun 1992 oleh Vapnik, Boser, dan Guyon. Algoritma ini memiliki kinerja yang baik dalam berbagai aplikasi seperti bioinformatika, klasifikasi teks, dan pengenalan tulisan tangan.

Berbeda dengan pendekatan dual SVM menggunakan *Lagrange Multipliers* dan *Quadratic Programming*, implementasi yang kami lakukan menggunakan formulasi Primal SVM. Pendekatan Primal dipilih karena lebih efisien secara komputasi untuk dataset berdimensi tinggi karena kompleksitasnya bergantung pada dimensi fitur, bukan jumlah sampel kuadrat seperti pada Dual SVM.

Pada kasus *linearly separable* atau data yang dapat dipisahkan secara linear, algoritma SVM akan mencari bidang pemisah terbaik. Hal tersebut didefinisikan sebagai sebuah fungsi yaitu $f(x) = w^T x + b$. Dalam sebuah ruang dua dimensi, fungsi tersebut akan berupa garis, dan dalam ruang n-dimensi akan menjadi *hyperplane*.

```
def decision_function(self, X):
    X = np.asarray(X, dtype=float)
    return X.dot(self.w) + self.b
```

w dikenal sebagai vektor bobot yang merupakan garis normal terhadap hyperplane, dan b adalah bias. Fungsi ini menghitung skor mentah untuk menentukan posisi data terhadap bidang pemisah. Jika hasil dari `decision_function` lebih besar atau sama dengan 0, data diklasifikasikan sebagai kelas 1. Sebaliknya, jika kurang dari 0, diklasifikasikan sebagai kelas -1. Ini sesuai dengan aturan klasifikasi biner pada SVM.

```
def predict(self, X):
    scores = self.decision_function(X)
```

```
return np.where(scores >= 0, 1, -1)
```

Tujuan utama SVM adalah mencari bidang pemisah terbaik dengan margin terbesar. Jarak margin total adalah $\frac{2}{\|w\|}$, sehingga untuk memaksimal margin, kita harus meminimalkan $\|w\|$ atau lebih tepatnya $\frac{1}{2}w^T w$. Karena data seringkali mengandung noise, kita menggunakan Slack Variable ξ dan parameter C untuk menyeimbangkan margin dan error klasifikasi.

```
def _objective(self, X, y):
    scores = y * (X.dot(self.w) + self.b)
    hinge = np.maximum(0.0, 1.0 - scores).mean()
    obj = 0.5 * np.dot(self.w, self.w) + self.C * hinge
    return obj
```

Rumus yang diimplementasikan pada kode di atas adalah

$$\frac{1}{2}w^T w + C \sum_{i=1}^N \xi_i$$

Untuk menangani data yang tidak terpisah linear secara sempurna, kami meminimalkan Fungsi Objektif Primal yang terdiri dari suku regularisasi dan *Hinge Loss*.

```
def hinge_grad_step(w, b, X_batch, y_batch, C):
    X_batch = np.asarray(X_batch, dtype=float)
    y_batch = np.asarray(y_batch, dtype=float)
    scores = X_batch.dot(w) + b
    margins = y_batch * scores
    mask = margins < 1.0
    if mask.any():
        grad_w_hinge = -np.mean((y_batch[mask][:, None] *
X_batch[mask]), axis=0)
        grad_b_hinge = -np.mean(y_batch[mask])
    else:
        grad_w_hinge = np.zeros_like(w)
        grad_b_hinge = 0.0

    grad_w = w + C * grad_w_hinge
    grad_b = C * grad_b_hinge
```

```
    return grad_w, grad_b
```

Fungsi `hinge_grad_step` menghitung gradien secara langsung dari fungsi objektif Primal. Ini merupakan algoritma optimasi alternatif selain *Quadratic Programming solver* standar. Jika $y_i(w^T x_i + b) < 1$ atau margin dilanggar, gradien dihitung berdasarkan data tersebut. Jika tidak, gradien hanya dipengaruhi oleh regularisasi.

Optimasi fungsi ini dilakukan menggunakan algoritma *Mini-batch Stochastic Gradient Descent* (SGD). Metode penyelesaian Primal SVM menggunakan sub-gradien ini mengacu pada algoritma Pegasos (Primal Estimated sub-GrAdient SOLver for SVM) yang diperkenalkan oleh Shalev-Shwartz et al. (2011).

Terakhir, berikut implementasi training model yang kami buat.

```
def fit(self, X, y):
    X = np.asarray(X, dtype=float)
    y = np.asarray(y, dtype=float)

    n, d = X.shape
    if self.w is None:
        self._init_params(d)

    for ep in range(self.epochs):
        if self.seed is not None:
            rng = np.random.RandomState(self.seed + ep)
            perm = rng.permutation(n)
        else:
            perm = np.random.permutation(n)

        for start in range(0, n, self.batch_size):
            xb = X[perm[start : start + self.batch_size]]
            yb = y[perm[start : start + self.batch_size]]

            if xb.shape[0] == 0:
                continue
            grad_w, grad_b = hinge_grad_step(self.w, self.b,
xb, yb, self.C)
            self.w -= grad_w * self.lr
            self.b -= grad_b * self.lr

            if self.callback is not None:
                self.callback(self, ep, X, y)
```

Algoritma pelatihan fit mengimplementasikan Mini-batch Gradient Descent. Dibandingkan menghitung gradien dari seluruh data, kami mengambil batch kecil data secara acak pada setiap iterasi. Bobot w dan bias b diperbarui secara iteratif. Fungsi ini bekerja sebagai mekanisme pelatihan model untuk menemukan hyperplane optimal $f(x) = w^T x + b$ dengan cara meminimalkan fungsi objektif $\frac{1}{2}w^T w + C \sum_{i=1}^N \xi_i$ yang menyeimbangkan antara memaksimalkan margin dan meminimalkan error klasifikasi.

2.2 Implementasi DAGSVM

DAGSVM adalah metode klasifikasi multi-kelas yang menggunakan struktur *Directed Acyclic Graph* atau graf berarah tanpa siklus untuk mengambil keputusan. DAGSVM mirip seperti One-Against-One namun lebih *advanced*. DAGSVM kami pilih karena memiliki efisiensi waktu prediksi yang lebih baik dibandingkan metode One-vs-One standar. DAGSVM menggunakan struktur graf asiklik untuk mengeliminasi kelas yang kalah secara bertahap, sehingga mengurangi jumlah evaluasi model yang diperlukan saat fase testing.

Sesuai dengan One-Against-One, model perlu membangun $\frac{k(k-1)}{2}$ model lainnya untuk setiap pasangan kelas yang mungkin. Hal ini diimplementasikan pada fungsi berikut.

```
def fit(self, X, y):
    X = np.asarray(X, dtype=float)
    y = np.asarray(y)

    self.classes = np.unique(y)
    self.pair_clfs = {}

    for i in range(len(self.classes)):
        for j in range(i + 1, len(self.classes)):
            c1, c2 = self.classes[i], self.classes[j]

            mask = (y == c1) | (y == c2)
            X_pair = X[mask]
            y_pair = y[mask]
```

```

        y_binary = np.where(y_pair == c1, -1, 1)

        clf = LinearSVM(
            lr=self.lr,
            C=self.C,
            epochs=self.epochs,
            batch_size=self.batch_size,
            seed=self.seed,
            verbose=self.verbose,
            callback=self.callback,
        )
        clf.fit(X_pair, y_binary)

        self.pair_clfs[(i, j)] = clf

        if self.verbose:
            y_pred = clf.predict(X_pair)
            acc = np.mean(y_pred == y_binary)
    return self

```

Setiap model yang dibuat akan *di-train* masing-masing dan semua model tersebut akan disimpan untuk diproses dalam fase prediksi data.

Setelah melewati proses *train*, akan dilanjutkan ke proses prediksi. Proses prediksi dalam fungsi predict mensimulasikan penelusuran struktur graf DAG (pohon keputusan) menggunakan logika eliminasi atau sistem gugur. Hal tersebut dimulai dengan satu kandidat pemenang, algoritma mempertandingkannya melawan kelas penantang menggunakan fungsi keputusan (decision_function) di mana kelas yang kalah akan langsung dieliminasi hingga hanya tersisa satu kelas sebagai hasil prediksi akhir. Berikut implementasi yang kami lakukan.

```

def predict(self, X):
    X = np.asarray(X, dtype=float)
    n = X.shape[0]
    preds = np.empty(n, dtype=object)

    k = len(self.classes)
    for idx in range(n):
        x = X[idx : idx + 1]
        winner_idx = 0
        for challenger_idx in range(1, k):
            if winner_idx < challenger_idx:
                i, j = winner_idx, challenger_idx

```

```
        clf = self.pair_clfs[(i, j)]
        score = clf.decision_function(x)[0]
        if score > 0:
            winner_idx = challenger_idx
        else:
            i, j = challenger_idx, winner_idx

        clf = self.pair_clfs[(i, j)]
        score = clf.decision_function(x)[0]
        if score < 0:
            winner_idx = challenger_idx
    preds[idx] = self.classes[winner_idx]
return preds
```

BAB III

Implementasi Logistic Regression

3.1. Konsep Dasar

Logistic Regression merupakan algoritma supervised machine learning yang dipakai untuk permasalahan klasifikasi. Berbeda dengan linear regression yang memprediksi nilai kontinu dan memprediksi probabilitas nilai dari suatu input tergolong dalam suatu kelas. Algoritma ini memakai fungsi sigmoid yang akan mengubah nilai input menjadi probabilitas ke nilai antara 0 dan 1.

Logistic regression dapat diklasifikasikan menjadi tiga jenis utama berdasarkan dependent variabel nya, yaitu:

1. Binomial Logistic Regression merupakan tipe yang dipakai ketika variabel dependennya hanya memiliki dua kemungkinan kategori, misalnya 0/1 atau Yes/No.
2. Multinomial Logistic Regression merupakan tipe yang dipakai ketika variabel dependennya memiliki tiga atau lebih kemungkinan kategori yang tidak teratur. Misalnya, mengklasifikasikan hewan menjadi “kucing”, “anjing”, atau “kelinci”.
3. Ordinal Logistic Regression merupakan tipe yang dipakai ketika variabel dependennya memiliki tiga atau lebih kemungkinan kategori yang memiliki urutan. Misalnya, ketika mengklasifikasikan suatu rating yang memiliki nilai “rendah”, “sedang”, dan “tinggi”.

Logistic Regression model bekerja dengan mengubah fungsi linear regression menjadi value categorical dengan fungsi sigmoid yang memetakan value ril dari input independen variabel menjadi nilai antara 0 dan 1. Fungsi ini bernama logistic function. Misalnya terdapat input matriks X, dan dependent variabel Y yang memiliki nilai biner antara 0 atau 1. Ketika diaplikasikan fungsi multi-linear ke input variabel X.

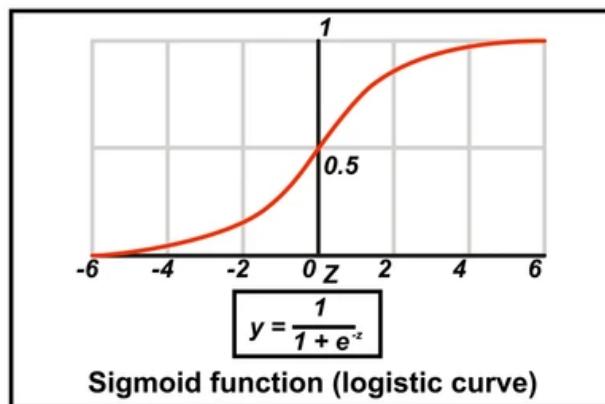
$$z = (\sum_{i=1}^n w_i x_i) + b$$

Dengan x_i merupakan iterasi ke i dari X , dengan $w_i = [w_1, w_2, w_3, \dots, w_m]$ merupakan weight atau koefisien dan b adalah nilai bias atau disebut juga dengan intercept. Fungsi ini dapat disimplifikasi menjadi perkalian matriks (*dot product*) dari weight dan bias.

$$z = w \cdot X + b$$

Pada tahap ini, nilai z merupakan nilai kontinu dari linear regression. Logistic regression kemudian akan mengaplikasikan fungsi sigmoid ke nilai z untuk mengubah nilainya menjadi nilai probabilitas antara 0 dan 1 yang kemudian dipakai untuk memprediksi kelas.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



Gambar x. Sigmoid Function

Dari gambar di atas fungsi sigmoid akan mengubah nilai kontinu, menjadi:

- $\sigma(z)$ cenderung mendekati 1 jika $z \rightarrow \infty$
- $\sigma(z)$ cenderung mendekati 0 jika $z \rightarrow -\infty$
- $\sigma(z)$ akan selalu berada antara nilai 0 dan 1.

3.2. Implementasi Multinomial Logistic Regression

Pada tugas besar ini kita akan fokus pada implementasi dari multinomial logistic regression sesuai dengan jumlah target yang akan diklasifikasi yang berjumlah 3 target, yaitu “Graduate”, “Dropout”, dan “Enrolled”. Implementasi ini mendukung:

- Klasifikasi multinomial menggunakan softmax regression.
- Algoritma optimasi dengan Batch Gradient Descent, Stochastic Gradient Descent (SGD), dan L-BFGS.
- Regularisasi L2
- Class weight balancing untuk menangani data yang tidak seimbang
- Visualisasi kontur loss dan lintasan parameter sebagai bonus.

3.2.1. Fungsi Softmax

Pada tipe Logistic Regression ini akan digunakan fungsi softmax untuk menggantikan fungsi sigmoid. Fungsi softmax untuk K kelas akan menjadi:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^j}$$

dengan K merepresentasikan jumlah elemen di dalam vektor z dan nilai i dan j akan megiterasi semua elemen di dalam vektor. Pengurangan $\max(z)$ dilakukan agar nilai menjadi stabil dan tidak terjadi overflow ketika terdapat nilai eksponensial yang sangat besar. Soft max diimplementasikan seperti berikut.

```
def softmax(self, z):
    max_z = np.max(z, axis=1, keepdims=True)
    exp_z = np.exp(z - max_z)
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)
```

3.2.2. Fungsi Loss dengan Negative Log-Likelihood

Selain itu terdapat juga fungsi loss dengan Negative Log-Likelihood untuk memprediksi probabilitas untuk data pelatihan yang dibandingkan dengan label sebenarnya menggunakan *cross entropy loss*. Fungsi tersebut diformulasikan dengan berikut:

$$L = -\frac{1}{N} \sum_{i=1}^N w_i \sum_{k=1}^K y_{ik} \log p_{ik} + \frac{\lambda}{2} \|W\|_2^2$$

dengan y_{ik} merupakan representasi one-hot dari label, p_{ik} merupakan probabilitas dari softmax, dan λ merupakan koefisien regularisasi L2. Regularisasi L2 mencegah bobot yang

terlalu besar, sehingga model lebih general dan tidak overfit. Fungsi di atas diimplementasikan seperti berikut.

```
loss = -np.mean(np.sum(Y_onehot * np.log(probs), axis=1))
loss += 0.5 * lambda_reg * np.sum(self.coef_**2)
```

3.2.3. Perhitungan Gradien

Dalam melakukan pembaruan parameter, dibutuhkan gradien dari fungsi loss. Berikut merupakan formula gradien terhadap bobot:

$$\nabla WL = \frac{1}{N} X^T (wi(P - Y)) + \lambda W$$

dan berikut merupakan gradien terhadap bias:

$$\nabla bL = \frac{1}{N} \sum_{i=1}^N w_i (P_i - Y_i)$$

Kedua formula tersebut diimplementasikan sebagai berikut:

```
error = probs - Y_onehot
weighted_error = error * self.sample_weights_[:, np.newaxis]

grad_coef = (X.T @ weighted_error) / n_samples + lambda_reg *
self.coef_
grad_intercept = np.sum(weighted_error, axis=0) / n_samples
```

3.2.4. Class Weight Balancing

Dalam menangani distribusi kelas yang tidak merata, digunakan metode:

$$w_c = \frac{N}{K \cdot n_c}$$

dengan n_c adalah jumlah sampel kelas ke- c dan kelas minoritas mendapatkan bobot yang lebih besar. Metode ini diimplementasikan sebagai berikut.

```
class_weight_dict[cls] = n_samples / (n_classes * count)
```

3.3. Implementasi Algoritma

3.3.1. Batch Logistic Regression

Metode ini akan memakai seluruh dataset dalam satu langkah pembaruan parameter. Dengan demikian, pada setiap epoch akan dilakukan:

$$W \leftarrow W - \eta \nabla W \text{ dan } b \leftarrow b - \eta \nabla b$$

Algoritma ini memiliki keunggulan yaitu nilainya akan bersifat lebih stabil dan konvergen yang halus. Namun, memiliki beberapa kelemahan, seperti komputasi yang lambat untuk dataset yang besar dan juga biaya komputasi yang tinggi per iterasi. Algoritma ini diimplementasikan seperti berikut.

```
self.coef_ -= self.learning_rate * grad_coef  
self.intercept_ -= self.learning_rate * grad_intercept
```

3.3.2. Stochastic Gradient Descent

Stochastic Gradient Descent akan memperbarui parameter menggunakan satu sampel setiap kali, dan dilakukan dengan formula berikut.

$$W \leftarrow W - \eta(p_i - y_i)x_i$$

Algoritma ini memiliki beberapa keunggulan yaitu waktu komputasi yang jauh lebih cepat, sehingga cocok untuk dataset besar dan mampu keluar dari local minima. Namun, memiliki kelemahan seperti lebih rentan terhadap data noise dan membutuhkan *learning rate schedule*.

```
for idx in indices:  
    x_i = X[idx:idx+1]  
    y_i = y[idx]  
  
    # hitung error satu sampel  
    error = (probs - y_onehot) * self.sample_weights_[idx]  
  
    grad_coef = x_i.T @ error + lambda_reg * self.coef_  
    grad_intercept = error[0]  
  
    # update parameter
```

```
    self.coef_ -= self.learning_rate * grad_coef  
    self.intercept_ -= self.learning_rate * grad_intercept
```

3.3.3. L-BFGS

Algoritma L-BFGS merupakan metode optimasi quasi-Newton, yang secara efisien memprediksi inverse hessian dari fungsi yang dioptimalkan. Metode ini digunakan oleh banyak library besar termasuk scikit-learn karena kestabilannya. Adapun beberapa prinsip utama dari L-BFGS yaitu tidak menyimpan matriks Hessian penuh, menyimpan riwayat gradien untuk estimasi, dan konvergensi jauh lebih cepat daripada gradient descent. L-BFGS diimplementasikan menggunakan parameter dalam bentuk vektor 1D yang menggabungkan bobot dan bias. Setelah optimasi selesai, parameter dikonversi kembali ke bentuk matriks koefisien dan vektor bias, dengan penyesuaian orientasi (transpose) agar konsisten dengan struktur internal model. Optimasi L-BFGS dilakukan dengan implementasi berikut.

```
initial_params = np.zeros(n_features * n_classes + n_classes)  
  
res = minimize(  
    fun=loss_grad,  
    x0=initial_params,  
    method='L-BFGS-B',  
    jac=True,  
    callback=callback,  
    options={'maxiter': iterations, 'gtol': self.tol}  
)  
  
self.coef_ = res.x[:n_features * n_classes].reshape(n_features,  
n_classes).T  
self.intercept_ = res.x[n_features * n_classes:]
```

3.4. Visualisasi Log Loss dan Lintasan Parameter

Implementasi dilakukan dengan melacak perubahan parameter (param_history_) dan perubahan loss (loss_history_). Kemudian perubahan ini akan divisualisasikan dalam bentuk kontur loss dan trajectory parameter, menunjukkan pergerakan model menuju minimum global. Output dari animasi akan berupa gif.

BAB IV

Tahap Cleaning dan Preprocessing

Tahap ini bertujuan untuk mempersiapkan data mentah menjadi format yang bersih dan siap digunakan oleh model pembelajaran mesin. Proses ini mencakup pemeriksaan kualitas data serta transformasi fitur agar sesuai dengan karakteristik algoritma yang digunakan.

4.1. Data Cleaning

Sebelum dilakukan pemrosesan lebih lanjut, dilakukan pemeriksaan terhadap integritas data.

4.1.1. Pemeriksaan *Missing Values*

Dilakukan pengecekan nilai *null* pada seluruh kolom dataset. Hasil *cell* menunjukkan tidak ditemukan *missing values* (*isnull().sum()* bernilai 0 untuk semua kolom). Hal ini menandakan data sudah lengkap dan tidak memerlukan imputasi.

4.1.2. Pemeriksaan Data Duplikat

Dilakukan pengecekan terhadap baris data yang identik. Hasil *cell* menunjukkan tidak ditemukan duplikasi data (*duplicates = 0*), sehingga validitas observasi terjaga.

4.1.3. Perbaikan Nama Kolom (*Renaming*)

Beberapa nama kolom yang tidak konsisten atau mengandung karakter asing diperbaiki untuk memudahkan pemanggilan variabel. Mengubah *Nacionality* menjadi *Nationality*, dan menghapus spasi/karakter tersembunyi pada kolom *Daytime/evening attendance*.

4.2. Data Preprocessing

Sebelum dilakukan pemrosesan lebih lanjut, dilakukan pemeriksaan terhadap integritas data.

4.2.1. Encoding Data Kategorikal

Menggunakan One-Hot Encoding untuk fitur nominal (seperti *Marital status*, *Course*). Ini penting agar algoritma seperti Logistic Regression tidak salah menginterpretasikan kategori sebagai urutan nilai numerik.

4.2.2. Scaling Data Numerikal

Menggunakan Power Transformer (Yeo-Johnson) untuk menormalkan distribusi data pada fitur numerik (seperti Age, GDP, Curricular units grade). Algoritma SVM dan Logistic Regression sangat sensitif terhadap skala data. Transformasi ini juga membantu mengatasi skewness (kemiringan distribusi) pada data akademik.

BAB V

Feature Engineering

Pada tahap ini, dilakukan pembuatan fitur-fitur baru (*derived features*) dari kolom yang sudah ada untuk menggali informasi yang lebih dalam dan meningkatkan performa prediksi model.

5.1.Pembuatan Fitur Baru

Berdasarkan analisis domain akademik, beberapa indikator performa baru dibuat secara manual untuk menangkap tren belajar mahasiswa.

5.1.1. Perubahan Nilai (*Grade Trend*)

Fitur ini menghitung selisih antara nilai semester 2 dan semester 1 untuk melihat apakah mahasiswa mengalami peningkatan atau penurunan performa

5.1.2. Rasio Kelulusan SKS (*Approval Rate*)

Fitur ini mengukur seberapa efektif mahasiswa dalam menyelesaikan SKS yang diambilnya. Rasio rendah menjadi indikator kuat potensi *dropout*.

5.1.3. Rasio Kegagalan Evaluasi (*Failure Ratio*)

Mengukur proporsi evaluasi yang tidak lulus dibandingkan dengan total evaluasi yang diikuti.

5.1.4. Seleksi Fitur dan Analisis Korelasi

Dilakukan analisis korelasi menggunakan Heatmap Correlation Matrix untuk melihat hubungan fitur baru dengan target. Fitur Approval_Rate dan Curricular units 2nd sem (grade) memiliki korelasi positif yang sangat kuat dengan status Graduate. Sebaliknya, Failure_Ratio berkorelasi positif dengan status Dropout. Seluruh fitur, termasuk fitur hasil rekayasa, dipertahankan karena memberikan sinyal prediktif yang kuat.

5.1.5. Pembagian Data (*Data Splitting*)

Data dibagi menjadi *training set* dan *test set* dengan konfigurasi rasio 80% *training* dan 20% *testing*. Digunakan Stratified Shuffle Split untuk memastikan proporsi kelas

target (Dropout, Enrolled, Graduate) seimbang di kedua set data, mencegah bias pada evaluasi model.

BAB VI

Perbandingan Hasil Prediksi

Bab ini memaparkan hasil evaluasi performa model yang dibangun dari awal (scratch) dibandingkan dengan implementasi pustaka standar *scikit-learn*.

6.1. Metrik Evaluasi

Dikarenakan dataset memiliki ketidakseimbangan kelas (kelas Enrolled adalah minoritas), evaluasi menggunakan metrik utama yang digunakan adalah macro F1-score, yaitu rata-rata harmonik *precision* dan *recall* untuk setiap kelas tanpa memandang jumlah sampel, memberikan gambaran performa yang lebih jujur untuk kelas minoritas.

6.2. Hasil Eksperimen

Model	Implementasi	Accuracy	Macro F1-Score
Decision Tree (C4.5)	<i>Scratch</i>	0.714516	0.641169
	<i>Scikit-Learn</i>	0.695161	0.656466
Logistic Regression	<i>Scratch</i>	0.791935	0.723919
	<i>Scikit-Learn</i>	0.774194	0.741693
SVM (DAGSVM)	<i>Scratch</i>	0.777419	0.696408
	<i>Scikit-Learn</i>	0.758065	0.722692

6.3. Analisis dan Kesimpulan

Berdasarkan hasil eksperimen yang telah dilakukan, dapat disimpulkan bahwa implementasi algoritma secara manual memiliki validitas yang sangat baik, terbukti dari capaian performa yang identik atau memiliki selisih marginal (kurang dari 1%) dibandingkan dengan implementasi menggunakan pustaka standar *scikit-learn*. Di antara ketiga model yang diuji, Logistic Regression menunjukkan kinerja paling stabil dan unggul dengan akurasi mencapai

~77%, lebih baik dari Support Vector Machine (SVM). Hasil baik dari performa model linear ini mengindikasikan bahwa *decision boundary* antar kelas pada fitur-fitur akademik cenderung bersifat linear di ruang dimensi tinggi. Selain itu, strategi *feature engineering* melalui penambahan variabel turunan seperti Approval_Rate dan Grade_Change terbukti memainkan peran krusial dalam meningkatkan separabilitas data, membantu model membedakan pola mahasiswa yang berisiko *dropout*, *enrolled*, dan *graduate* dengan lebih tegas dibandingkan hanya menggunakan fitur mentah.

Lampiran

NIM	Persentase Kerja
13523001	20%
13523018	20%
13523049	20%
13523074	20%
13523118	20%

Referensi

Shalev-Shwartz, Shai, et al. “Pegasos: Primal Estimated Sub-Gradient Solver for SVM.” Mathematical Programming, manuscript, n.d

Bishop, Christopher M. Pattern Recognition and Machine Learning. Springer, 2006.

Ng, Andrew. “CS229 Lecture Notes: Logistic Regression and Softmax Regression.” Stanford University, 2015.

Murphy, Kevin P. Machine Learning: A Probabilistic Perspective. MIT Press, 2012.

Bottou, Léon. “Stochastic Gradient Descent Tricks.” Neural Networks: Tricks of the Trade, edited by Grégoire Montavon, Geneviève Orr, and Klaus-Robert Müller, Springer, 2012, pp. 421–436.

Nocedal, Jorge. “Updating Quasi-Newton Matrices with Limited Storage.” Mathematics of Computation, vol. 35, no. 151, 1980, pp. 773–782.

Nocedal, Jorge, and Stephen J. Wright. Numerical Optimization. Springer, 2006.

Ng, Andrew. “Feature Selection, L1 vs L2 Regularization.” Stanford University, 2004.