LAPORAN TUGAS BESAR 2

IF3070 Dasar Inteligensi Artifisial

“Implementasi Algoritma Pembelajaran Mesin”

Dosen:

Dr. Fariska Zakhralativa Ruskanda, S.T., M.T.

Oleh:

Owen Tobias Sinurat (13522131)

Ahmad Thoriq Saputra (13522141)

Muhammad Fatihul Irhab (13522143)

Muhammad Fauzan Azhim (13522153)

PROGRAM STUDI TEKNIK INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

SEMESTER II TAHUN 2023/2024

Daftar Isi

DATA CLEANING & PRE-PROCESSING

Handling Missing Data

Data yang hilang dapat menyebabkan bias atau kesalahan dalam analisis dan model prediktif. Penanganan data yang hilang dibagi menjadi dua berdasarkan jenis datanya, yaitu kategorik dan numerik.

Dealing with Outliers

Data outlier dapat mengganggu distribusi data, mengurangi kinerja model, dan memberikan hasil yang bias. Imputation untuk outlier adalah metode mengganti nilai outlier dengan nilai yang lebih wajar atau representatif, seperti mean, median, atau nilai batas (threshold) yang telah ditentukan, agar data tetap konsisten dan analisis tidak terpengaruh ekstrem.

Remove Duplicates

Data duplikat dapat menyebabkan bias pada model dan analisis karena data yang sama dihitung lebih dari sekali. Data duplikat pada training set dihapus untuk menghindari bias ini, tetapi data duplikat pada validation set tidak dihapus.

Feature Engineering

Menambah atau memodifikasi fitur dapat meningkatkan kemampuan model dalam memahami pola. Menambah 4 kolom baru yang dihasilkan dari operasi antar kolom lain.

Feature Scaling

Model berbasis jarak (seperti SVM atau KNN) dan optimisasi berbasis gradien sangat sensitif terhadap skala fitur.

Feature Encoding

Model pembelajaran mesin membutuhkan data numerik. Data kategorikal harus diubah menjadi format numerik. Kolom yang diubah adalah ‘proto’, ‘state’, dan ‘service’.

Handling Imbalanced Dataset

Ketidakseimbangan pada label target (misalnya, kelas minoritas sangat sedikit) dapat membuat model bias terhadap kelas mayoritas.

IMPLEMENTASI KNN

Algoritma k-Nearest Neighbors (KNN) adalah teknik *supervised learning* yang sederhana namun efektif, digunakan untuk tugas klasifikasi dan regresi dalam *machine learning*. Algoritma ini bekerja berdasarkan prinsip kedekatan, dengan asumsi bahwa data dengan karakteristik serupa cenderung berada dalam jarak yang dekat. Dalam klasifikasi, KNN menetapkan label kelas pada titik data dengan mengidentifikasi *k* tetangga terdekatnya dan menerapkan mekanisme *majority voting*. Artinya, label yang paling sering muncul di antara tetangga tersebut akan menentukan klasifikasi. Kesederhanaan, sifat non-parametrik, dan kemampuannya menangani berbagai jenis data menjadikan KNN pilihan populer dalam banyak aplikasi.

Untuk menentukan kedekatan, KNN menggunakan metrik jarak seperti Euclidean, Manhattan, atau Minkowski, tergantung pada jenis data yang digunakan. Metrik-metrik ini membantu membentuk batas keputusan yang membagi data ke dalam wilayah-wilayah sesuai dengan kelasnya. Parameter k sangat penting dalam algoritma ini karena menentukan jumlah tetangga yang mempengaruhi keputusan. Nilai k yang kecil cenderung lebih sensitif terhadap noise, sementara nilai yang besar dapat membuat batas keputusan menjadi lebih halus. Meskipun sederhana, algoritma KNN memerlukan pemilihan metrik jarak dan nilai k yang tepat untuk kinerja yang optimal.

```python
class KNNClassifier:
    def __init__(self, k=3, metric="euclidean", weights="uniform", batch_size=1000):
        self.k = k
        self.metric = metric
        self.weights = weights
        self.batch_size = batch_size
        self.train_data = None
        self.train_labels = None

    def fit(self, train_data, train_labels):
        self.train_data = np.asarray(train_data, dtype=np.float32)
        self.train_labels = np.asarray(train_labels)
        return self

    def _compute_batch_distances(self, test_batch):
        if self.metric == 'euclidean':
            distances = cdist(test_batch, self.train_data, metric='euclidean')
        elif self.metric == 'manhattan':
            distances = cdist(test_batch, self.train_data, metric='cityblock')
        elif self.metric == 'minkowski':
            distances = cdist(test_batch, self.train_data, metric='minkowski')
        else:
            raise ValueError(f"Unsupported metric: {self.metric}")
        return distances

    def predict(self, test_points):
        if self.train_data is None:
            raise ValueError("Model has not been trained. Call 'fit' first.")

        test_points = np.asarray(test_points, dtype=np.float32)

        predictions = []

        for i in range(0, len(test_points), self.batch_size):

            test_batch = test_points[i:i+self.batch_size]

            distances = self._compute_batch_distances(test_batch)

            batch_predictions = []
            for point_distances in distances:
                k_indices = np.argpartition(point_distances, self.k)[:self.k]

                k_labels = self.train_labels[k_indices]
                k_dist = point_distances[k_indices]

                if self.weights == 'distance':
                    weights = 1.0 / (k_dist + 1e-8)
                    unique_labels, _ = np.unique(k_labels, return_counts=True)
                    weighted_votes = []
                    for label in unique_labels:
                        label_mask = (k_labels == label)
                        weighted_vote = np.sum(weights[label_mask])
                        weighted_votes.append((label, weighted_vote))
                    prediction = max(weighted_votes, key=lambda x: x[1])[0]
                else:
                    prediction = Counter(k_labels).most_common(1)[0][0]

                batch_predictions.append(prediction)

            predictions.extend(batch_predictions)

        return np.array(predictions)
```

```
    def get_params(self, deep=True):
        return {
            'k': self.k,
            'metric': self.metric,
            'weights': self.weights,
            'batch_size': self.batch_size
        }

    def set_params(self, **params):
        if 'k' in params:
            self.k = params['k']
        if 'metric' in params:
            self.metric = params['metric']
        if 'weights' in params:
            self.weights = params['weights']
        if 'batch_size' in params:
            self.batch_size = params['batch_size']
        return self

    def score(self, X_test, y_test):
        predictions = self.predict(X_test)
        return np.mean(predictions == y_test)

    def submit(self, X_test, filename='submission.csv'):
        predictions = self.predict(X_test)
        submission_df = pd.DataFrame({'id': test_df['id'], 'attack_cat': predictions})
        submission_df.to_csv(filename, index=False)
        return submission_df

    def __repr__(self):
        return (f"KNNClassifier(k={self.k}, "
                f"metric='{self.metric}', "
                f"weights='{self.weights}', "
                f"batch_size={self.batch_size})")
```

Class KNNClassifier adalah implementasi algoritma k-Nearest Neighbors (KNN) yang dirancang untuk menangani tugas klasifikasi dengan fleksibilitas dan efisiensi. Algoritma ini bekerja dengan cara menghitung jarak antara titik data uji dan titik data pelatihan, kemudian menentukan label kelas berdasarkan k tetangga terdekat. Parameter utama yang dapat disesuaikan meliputi jumlah tetangga (k), jenis metrik jarak (euclidean, manhattan, atau minkowski), metode pembobotan (uniform untuk bobot sama atau distance untuk bobot berbasis jarak), serta ukuran batch untuk mengatur jumlah data yang diproses dalam satu waktu.

Metode fit pada class ini digunakan untuk melatih model dengan menyimpan data pelatihan dan label yang terkait. Data yang diberikan diubah menjadi array numpy untuk memastikan konsistensi format dan efisiensi komputasi. Setelah model dilatih, prediksi dapat dilakukan melalui metode predict. Pada tahap prediksi, jarak dihitung menggunakan fungsi _compute_batch_distances, yang memanfaatkan pustaka scipy untuk menghitung jarak berdasarkan metrik yang dipilih. Untuk setiap titik data uji, tetangga terdekat diidentifikasi

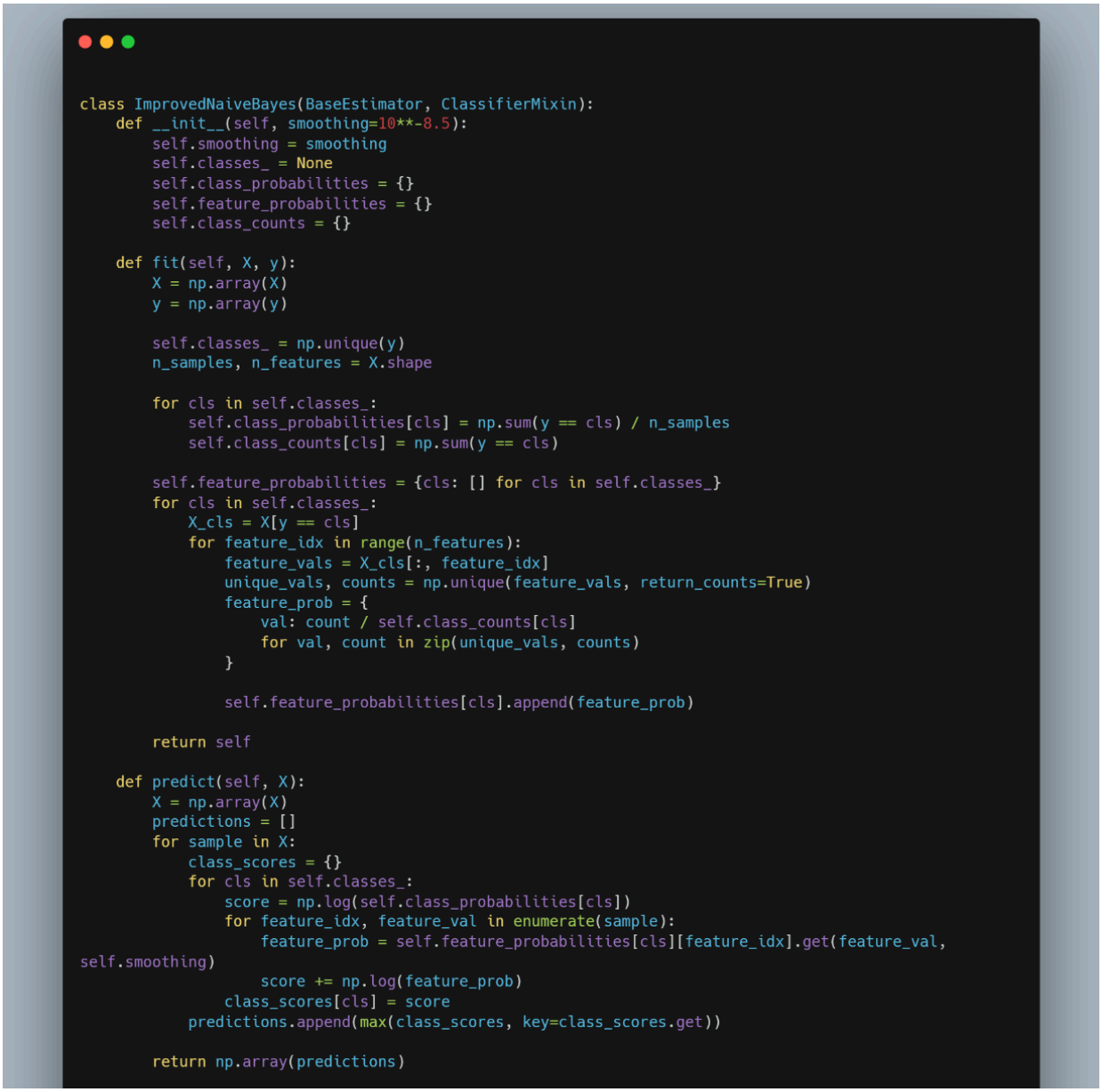
menggunakan fungsi np.argpartition, dan label kelas ditentukan dengan mayoritas suara atau berdasarkan bobot jarak, sesuai parameter yang ditentukan.

Class KNNClassifier juga menyediakan metode get_params dan set_params, yang memungkinkan integrasi dengan pipeline dan grid search dalam pustaka seperti scikit-learn. Metode get_params digunakan untuk mengambil parameter model saat ini dalam bentuk dictionary, termasuk nilai k, metrik jarak, metode pembobotan, dan ukuran batch. Sementara itu, metode set_params memungkinkan pengaturan ulang parameter model dengan fleksibilitas menggunakan keyword arguments. Dengan fitur ini, pengguna dapat dengan mudah menyesuaikan parameter model untuk eksplorasi hyperparameter secara otomatis, menjadikan class ini lebih kompatibel untuk berbagai kebutuhan pembelajaran mesin.

Selain itu, class ini memiliki metode score untuk menghitung akurasi prediksi terhadap data uji dengan membandingkan hasil prediksi dengan label sebenarnya. Metode submit memungkinkan pembuatan file CSV untuk mengunggah hasil prediksi, menjadikannya berguna untuk kompetisi pembelajaran mesin. Dukungan untuk pemrosesan batch meningkatkan efisiensi dalam menangani dataset besar, sementara kemampuan memilih metrik jarak dan metode pembobotan memberikan fleksibilitas dalam menangani berbagai jenis data.

IMPLEMENTASI NAIVE-BAYES

Model Naive Bayes merupakan metode klasifikasi yang berbasis pada teorema bayes dengan asumsi bahwa semua fitur yang digunakan untuk klasifikasi bersifat independen satu sama lain. Proses klasifikasi dilakukan dengan menghitung probabilitas setiap kelas berdasarkan data yang ada dan memilih kelas dengan probabilitas tertinggi sebagai prediksi. Implementasi ini menggunakan konsep dasar Naive Bayes yang menggabungkan probabilitas kelas dan probabilitas fitur dalam setiap kelas untuk menghitung kemungkinan kelas untuk sebuah data baru.



```python
class ImprovedNaiveBayes(BaseEstimator, ClassifierMixin):
    def __init__(self, smoothing=10**-8.5):
        self.smoothing = smoothing
        self.classes_ = None
        self.class_probabilities = {}
        self.feature_probabilities = {}
        self.class_counts = {}

    def fit(self, X, y):
        X = np.array(X)
        y = np.array(y)

        self.classes_ = np.unique(y)
        n_samples, n_features = X.shape

        for cls in self.classes_:
            self.class_probabilities[cls] = np.sum(y == cls) / n_samples
            self.class_counts[cls] = np.sum(y == cls)

        self.feature_probabilities = {cls: [] for cls in self.classes_}
        for cls in self.classes_:
            X_cls = X[y == cls]
            for feature_idx in range(n_features):
                feature_vals = X_cls[:, feature_idx]
                unique_vals, counts = np.unique(feature_vals, return_counts=True)
                feature_prob = {
                    val: count / self.class_counts[cls]
                    for val, count in zip(unique_vals, counts)
                }

                self.feature_probabilities[cls].append(feature_prob)

        return self

    def predict(self, X):
        X = np.array(X)
        predictions = []
        for sample in X:
            class_scores = {}
            for cls in self.classes_:
                score = np.log(self.class_probabilities[cls])
                for feature_idx, feature_val in enumerate(sample):
                    feature_prob = self.feature_probabilities[cls][feature_idx].get(feature_val,
self.smoothing)
                    score += np.log(feature_prob)
                class_scores[cls] = score
            predictions.append(max(class_scores, key=class_scores.get))

        return np.array(predictions)
```

```python
    def get_params(self, deep=True):
        return {"smoothing": self.smoothing}

    def set_params(self, **params):
        for key, value in params.items():
            setattr(self, key, value)
        return self

    def save_model(self, filename):
        with open(filename, 'wb') as file:
            pickle.dump(self, file)
        print(f"Model saved in {filename}.")

    @staticmethod
    def load_model(filename):
        with open(filename, 'rb') as file:
            model = pickle.load(file)
        print(f"Model loaded from {filename}.")
        return model

    def submit(self, X, output_filename="predictions.csv"):
        predictions = self.predict(X)

        prediction_df = pd.DataFrame({
            'id': range(len(predictions)),
            'attack_cat': predictions
        })

        prediction_df.to_csv(output_filename, index=False)
        print(f"Predictions saved to {output_filename}.")

        return prediction_df
```

Fungsi	Penjelasan
__init__(self, smoothing=10**-8.5)	Konstruktor untuk membuat model. Smoothing: Untuk menghindari nilai probabilitas nol, digunakan teknik smoothing pada setiap probabilitas fitur. classes_: Menyimpan kelas dalam dataset. class_probabilities: Menyimpan probabilitas dari setiap kelas dalam dataset. feature_probabilities: Menyimpan probabilitas fitur dalam setiap kelas. class_counts: Menyimpan jumlah sampel untuk setiap kelas.
fit(self, X, y)	Method untuk melatih model berdasarkan dengan data train. Model dilatih dengan langkah-langkah berikut: Menentukan kelas unik pada data target y. Menghitung probabilitas kelas berdasarkan frekuensi relatif kelas di dataset (P(class)). Menghitung probabilitas fitur per kelas. Setiap fitur dihitung distribusinya dalam setiap kelas berdasarkan nilai fitur tersebut (P(feature | class)). Menyimpan hasil perhitungan probabilitas.
predict(self, X)	Method untuk memprediksi hasil berdasarkan data test dan model yang sudah dilatih. Model menghitung probabilitas untuk setiap kelas pada data input dan memilih kelas dengan probabilitas tertinggi. Model memprediksi dengan langkah-langkah berikut: inisialisasi nilai skor untuk setiap

	kelas. Menghitung skor untuk setiap kelas dengan menggunakan rumus probabilitas kelas dikali dengan jumlah dari probabilitas setiap fitur yang diberikan kelas tersebut. Memilih kelas dengan probabilitas tertinggi sebagai hasil prediksi.
get_params(self, deep=True)	Method yang mengembalikan nilai dari smoothing. Method ini akan digunakan pada k-fold cross-validation.
set_params(self, **params)	Method untuk mengubah nilai dari param. Method ini akan digunakan pada k-fold cross-validation.
save_model(self, filename)	Method yang digunakan untuk menyimpan model ke dalam bentuk file dengan format pkl (pickle).
load_model(filename)	Method yang digunakan untuk me-load model yang akan digunakan yang berasal dari file dengan format pkl (pickle).
submit(self, X, output_filename="predictions.csv")	Method yang digunakan untuk memprediksi data test sekaligus menyimpannya ke dalam file bertipe csv untuk di submit di kaggle.

IMPLEMENTASI ID3

Model implementasi *iterative Dichotomiser 3* (ID3) menggunakan entropy sebagai impurity. Model ini membuat decision tree berdasarkan information gain yang didapatkan untuk memilih atribut terbaik yang akan menjadi akar atau node pada pohon keputusan.

```python
class ID3(BaseEstimator, ClassifierMixin):
    def __init__(self, max_depth=None, min_samples_split=2, min_samples_leaf=1, ccp_alpha=0.0):
        """Initialize the ID3 decision tree classifier."""
        self.tree = None
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf
        self.ccp_alpha = ccp_alpha

    def entropy(self, labels, base=np.e):
        """
        Compute the entropy of a list of labels.
        :param labels:
        :param base:
        :return:
        """
        labels = np.asarray(labels)
        _, counts = np.unique(labels, return_counts=True)
        probabilities = counts / len(labels)
        return -np.sum(probabilities * np.log(probabilities) / np.log(base))

    def information_gain(self, X, y, feature):
        """
        Compute the information gain of a feature.
        :param X:
        :param y:
        :param feature:
        :return:
        """
        total_impurity = self.entropy(y)
        if isspmatrix(X):
            feature_series = pd.Series(X[:, feature].toarray().ravel())
        else:
            feature_series = X.iloc[:, feature]
        grouped = y.groupby(feature_series)
        weighted_impurity = sum((len(group) / len(y)) * self.entropy(group) for _, group in grouped)
        return total_impurity - weighted_impurity

    def best_feature_(self, X, y):
        """
        Find the best feature to split on.
        :param X:
        :param y:
        :return:
        """
        features = range(X.shape[1])
        gains = [self.information_gain(X, y, feature) for feature in features]
        if all(gain == 0 for gain in gains):
            return None  # No information gain
        return np.argmax(gains)
```

codesnap.dev


def fit(self, X, y, features=None, depth=0):
"""
Fit the decision tree classifier.
:param X:
:param y:
:param features:
:param depth:
:return:
"""
X, y = self._check_input(X, y)
if features is None:
features = list(range(X.shape[1]))
# Stopping criteria
if len(np.unique(y)) == 1:
return {"value": y.iloc[0], "impurity": 0.0, "samples": len(y)}
if len(features) == 0 or (self.max_depth is not None and depth >= self.max_depth):
return {"value": y.mode()[0], "impurity": self.entropy(y), "samples": len(y)}
if len(y) < self.min_samples_split or len(y) <= self.min_samples_leaf:
return {"value": y.mode()[0], "impurity": self.entropy(y), "samples": len(y)}
# Find the best feature and split data
best_feature = self.best_feature_(X, y)
if best_feature is None:
return {"value": y.mode()[0], "impurity": self.entropy(y), "samples": len(y)}
unique_vals = np.unique(X[:, best_feature].toarray() if isspmatrix(X) else X.iloc[:, best_feature])
tree = {"feature": best_feature, "nodes": {}, "impurity": self.entropy(y), "samples": len(y)}
def process_val(val):
if isspmatrix(X):
mask = (X[:, best_feature].toarray().ravel() == val)
else:
mask = (X.iloc[:, best_feature] == val).values
subset_X = X[mask]
subset_y = y[mask].reset_index(drop=True)
if subset_y.empty:
return val, {"value": None, "impurity": 0.0, "samples": 0}
new_features = [f for f in features if f != best_feature]
return val, self.fit(subset_X, subset_y, new_features, depth + 1)
results = Parallel(n_jobs=-1)(delayed(process_val)(val) for val in unique_vals)
for val, subtree in results:
tree["nodes"][val] = subtree
self.tree = tree
return tree
def prune(self, tree=None):
"""
Prune the decision tree.
:param tree:
:return:
"""
if tree is None:
tree = self.tree
if "nodes" not in tree or not tree["nodes"]:
return tree["impurity"], None
subtree_impurities = []
subtree_trees = []
for key, subtree in tree["nodes"].items():
imp, subpruned = self.prune(subtree)
subtree_impurities.append(imp)
subtree_trees.append((key, subpruned))
subtree_cost = sum(subtree_impurities) + self.ccp_alpha * len(tree["nodes"])
if subtree_cost < tree["impurity"] + self.ccp_alpha:
tree = {"value": max(Counter(
[subtree["value"] for subtree in tree["nodes"].values() if "value" in subtree]).items(),
key=lambda x: x[1])[0]}
else:
for key, subpruned in subtree_trees:
if subpruned:
tree["nodes"][key] = subpruned
return tree["impurity"], tree
codesnap.dev

```python
def predict_one(self, x, tree):
    """Predict a single data point."""
    if "value" in tree:
        return tree["value"]
    feature = tree["feature"]
    if x[feature] in tree["nodes"]:
        return self.predict_one(x, tree["nodes"][x[feature]])
    else:
        leaf_values = [subtree["value"] for subtree in tree["nodes"].values() if "value" in subtree]
        if leaf_values:
            return max(Counter(leaf_values).items(), key=lambda x: x[1])[0]
        return tree.get("value", None)

def predict(self, X):
    """Predict multiple data points."""
    X = self._check_input(X)
    return np.array([self.predict_one(row, self.tree) for row in X.values])

def _check_input(self, X, y=None):
    """Helper function to check and preprocess inputs."""
    if isspmatrix(X):
        X = csc_matrix(X)
    elif isinstance(X, np.ndarray):
        X = pd.DataFrame(X)
    if isinstance(y, np.ndarray):
        y = pd.Series(y)
    return X, y

def save_tree(self, filepath):
    """Save the trained decision tree to a file."""
    with open(filepath, 'wb') as f:
        pickle.dump(self.tree, f)

def load_tree(self, filepath):
    """Load a decision tree from a file."""
    with open(filepath, 'rb') as f:
        self.tree = pickle.load(f)

def evaluate(self, X, y):
    """Evaluate the model."""
    y_pred = self.predict(X)
    accuracy = accuracy_score(y, y_pred)
    classification_rep = classification_report(y, y_pred)
    f1_macro = f1_score(y, y_pred, average='macro')
    print(f'F1-Score Macro Average: {f1_macro}')
    print(f'Accuracy: {accuracy}')
    print(f'Classification Report:\n{classification_rep}')

def submission(self, X, test_id=None):
    """Create a submission file."""
    y_pred = self.predict(X)
    submission = pd.DataFrame(y_pred, columns=['attack_cat'])
    if test_id is not None:
        submission.insert(0, "id", test_id)
    submission.to_csv('ID3-Save/submission_id3.csv', index=False)
    print("Submission file saved as 'submission_id3.csv'")
```

Kelas ID3

Fungsi	Penjelasan
Entropy	Mengkalkulasi entropy dari larik label.
information_gain	Menghitung information gain dari suatu fitur.
best_feature_	Mencari fitur terbaik yang akan dijadikan node atau child tree berdasarkan information gain tertinggi.
fit	Membuat tree secara rekursif dan membagi-bagi data sampai suatu kriteria berhenti terpenuhi.
prune	Mengurangi kompleksitas tree yang memiliki efek yang minim dibandingkan yang lain.
predict_one	Memprediksi class untuk satu label.
predict	Memprediksi banyak data dengan memanggil predict_one secara berurutan.
_check_input	Mengecek input yang dimasukkan, dan mengubah tipe datanya jika tidak sesuai.
Save_tree, load_tree	Menyimpan dan memasukkan tree.
evaluate	Mengevaluasi F1-Score Macro Average, Akurasi dan classification report.
submission	Membuat file submisi untuk .

PERBANDINGAN HASIL & ANALISIS

1. KNN

Perbandingan antara implementasi KNN buatan sendiri dan yang menggunakan library menunjukkan bahwa keduanya memiliki akurasi yang cukup kompetitif, dengan hasil akurasi masing-masing sebesar 71,7% dan 70,8%. Implementasi dari library, seperti scikit-learn, biasanya dirancang dengan optimasi yang tinggi untuk efisiensi komputasi dan kompatibilitas dengan pipeline machine learning lainnya. Namun, library ini terkadang memiliki keterbatasan dalam hal fleksibilitas jika pengguna membutuhkan penyesuaian tertentu, seperti penggunaan metrik jarak khusus atau algoritma voting yang lebih kompleks. Di sisi lain, implementasi buatan sendiri memungkinkan penyesuaian penuh sesuai kebutuhan, seperti yang terlihat pada akurasi yang sedikit lebih tinggi karena tuning khusus, seperti metode pembobotan jarak atau strategi batch processing.

Namun, penting untuk mempertimbangkan aspek lain seperti efisiensi, waktu pengembangan, dan ketahanan terhadap dataset yang lebih besar. Library sering kali lebih unggul dalam pengelolaan memori dan kecepatan karena memanfaatkan implementasi berbasis C atau Python di balik layar. Implementasi buatan sendiri, meskipun menawarkan fleksibilitas dan kontrol, bisa jadi lebih lambat dan memerlukan lebih banyak waktu untuk debugging serta optimasi. Oleh karena itu, pilihan antara keduanya bergantung pada konteks: jika tujuan utama adalah eksplorasi konsep atau penyesuaian mendalam, maka implementasi manual adalah pilihan yang tepat. Namun, untuk aplikasi praktis yang memerlukan efisiensi tinggi dan pengintegrasian cepat, library lebih disarankan.

2. Naive Bayes

Model Naive Bayes scratch memiliki akurasi yang lebih tinggi (76.42%) dibandingkan dengan model dari scikit-learn yang hanya mencapai (47.02%). Ini menunjukkan bahwa model Naive Bayes yang dibuat sendiri lebih baik dalam menangani distribusi kelas atau menangani data yang tidak seimbang. Model Naive Bayes scratch juga menunjukkan hasil cross-validation yang lebih baik dengan mean accuracy lebih tinggi (58.85%) dibandingkan dengan scikit-learn (47.43%). Namun, standard deviation

kedua model cukup rendah (0.20%) dan (0.11%), menunjukkan bahwa model cukup stabil dalam cross-validation.

Berdasarkan *classification report* pada model Naive Bayes scratch menunjukkan performa yang sangat baik pada beberapa kelas seperti Generic dan Normal, dengan nilai F1-score tinggi (0.97 dan 0.89), hal ini menunjukan model ini efektif dalam memprediksi kelas-kelas tersebut. Kelas-kelas yang lebih sulit (Backdoor, DoS) menunjukkan nilai recall yang lebih rendah, tetapi precision tetap stabil di banyak kelas. Sementara Naive Bayes scikit-learn menunjukan performa yang sangat rendah seperti Analysis, DoS, Reconnaissance, dan Shellcode, bahkan pada kelas DoS hanya memiliki recall 0.00, yang menunjukkan bahwa model scikit-learn kesulitan untuk mendeteksi kelas ini sama sekali. Recall untuk kelas Worms sangat tinggi (0.77), tetapi precision sangat rendah (0.03), menunjukkan bahwa model sering kali salah memprediksi kelas ini.

Hal ini dapat terjadi karena pada model Naive Bayes scratch memakan waktu lebih lama daripada Naive Bayes scikit-learn, sehingga memberikan lebih banyak kesempatan bagi model untuk mempelajari pola dalam data, yang dapat meningkatkan akurasi. Dengan kata lain pada model Naive Bayes scikit-learn sedikit underfitting, hal ini lah yang membuat akurasi dari Naive Bayes scikit-learn lebih rendah.

3. ID3

Model ID3 scratch memiliki akurasi 65% dan scikit memiliki 79%. Hal ini mungkin disebabkan adanya sedikit perbedaan implementasi dan optimisasi yang dilakukan pada scratch, baik saat pruning ataupun default parameter (Max depth , dll). Dalam hal F1-score scratch juga memiliki nilai yang lebih rendah di 35% dibandingkan scikit di angka 60%. Optimisasi yang rendah juga menjadi masalah dalam hal ID3 scratch dimana operasi pembuatan tree sangat lama dan membutuhkan memory yang tinggi karena algoritma diimplementasikan dengan python tidak seperti scikit yang menggunakan cython. Overall lama training untuk ID3 scratch tanpa menggunakan PCA dan Feature Selection adalah 30 menit dibandingkan scikit yang hanya 10 detik.

KONTRIBUSI

NIM	Tugas
13522131	Data Cleaning and Preprocessing
13522141	KNN Algorithm
13522143	Naive Bayes Algorithm
13522153	ID3 Algorithm

REFERENSI

- Fariska Zakhralativa Ruskanda (2024). Materi Kuliah IF3170 Inteligensi Artifisial, Institut Teknologi Bandung.