Tugas Besar 2 IF3170 Intelegensi Artifisial

“Implementasi Algoritma Pembelajaran Mesin”

Disusun oleh:

Kayla Namira Mariadi	K01	13522050
Andhita Naura Hariyanto	K01	13522060
Salsabiila	K02	13522062
Shulha	K02	13522087

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

PROBLEM DESCRIPTION

Machine learning is a branch of artificial intelligence that enables systems to learn from data and make predictions or decisions without being explicitly programmed.

The UNSW-NB15 dataset is a collection of network traffic data that includes various types of cyberattacks and normal activities. By implementing the machine learning algorithms studied in class—KNN, Gaussian Naive-Bayes, and ID3—we aim to predict the target values in the test dataset of UNSW-NB15.

The UNSW-NB15 dataset contains raw network packets generated using IXIA PerfectStorm by the Cyber Range Lab at UNSW Canberra. This dataset comprises 10 types of activities (9 types of attacks and 1 normal activity). The nine types of attacks included in the dataset are Fuzzers, Analysis, Backdoors, DoS, Exploits, Generic, Reconnaissance, Shellcode, and Worms.

To produce high-quality predictions, data cleaning and preprocessing are performed, followed by modeling and validation using the three algorithms: KNN, Gaussian Naive-Bayes, and ID3.

DATA CLEANING AND PREPROCESSING

1. Data Cleaning: Handling Missing Data

This part of the cleaning process will identify and address missing values in the dataset. Missing data can adversely affect the performance and accuracy of machine learning models. There are several strategies to handle missing data in machine learning such as data imputation with mean, median, or mode, with predefined constant value, or even with more advanced techniques involving using predictive models or using domain knowledge to estimate missing values. Another strategy is deletion of missing data listwise or column wise. We can use libraries like SimpleImputer from Scikit-Learn or FancyImpute to help with imputation.

On this project, the EDA on knowing missing values resulted in missing data are about 8000 to 9000 rows from more than 170000 data for each attribute, this shows less than 5% data of each attribute is missing.

Therefore, we will not use the technique of deletion of missing data. This is because the amount of missing data is not relatively small to delete listwise and not very big to delete the column. However, we will delete a row or a data if that data has many missing values in more than 3 attributes.

The strategy to handle the other missing values will be imputation. The numerical value will be imputed with median value (since the data is heavily skewed mostly, even if not, median is safe). The categorical value (include binary attribute) imputed with most-frequent data. We use SimpleImputer from Scikit-Learn for this imputation.

Aside from handling missing data, in this data cleaning we don't perform adjusting typo values because what we see in EDA, the categories are already unique and not typo.

For even better cleaning, we analyze some techniques that may help improve our project

1. Use IterativeImputer for Numerical Data, but computationally expensive
2. Imputation using domain knowledge from EDA
3. Use Fancyimpute, but computationally expensive

2. Data Cleaning: Dealing with Outliers

Outliers are data points that significantly differ from the majority of the data. They can be unusually high or low values that do not fit the pattern of the rest of the dataset. Outliers can significantly impact model performance, so it is important to handle them properly. Some methods are imputation, clipping, transformation, and use algorithms that are robust for outliers.

As we can see in EDA, we have very many outliers. If the outliers are identified using the interquartile method, there would be very many outliers detected. However, in the context of this network attack data, outliers may sometimes signal genuine anomalies, like network attacks in cybersecurity data, rather than mistakes. In this network dataset, we assumed that outliers could be expected due to network attack patterns and normal vs. abnormal behaviors. This makes it essential to identify these values, as they could be meaningful rather than erroneous. Given the statistics previously (min, max, median) and the Q1, Q3 information, almost no outliers lie below lower-bound (even Q1 and Q2 most times are still 0 or low-value), and very many outliers lie above upper-bound. This may indicate that data with high-value in some features is not normal data (maybe delay, overbuffer, or other network-related issue).

Therefore, we do not remove the data with outliers but we clipped/imputed to high percentile. This ensures 'real' outliers is excluded for the data we will use later, and 'meaningful' outliers preserved. Unfortunately, this still result in very many outliers if we use the interquartile method, because most times the outlier upper bound (Q3+1.5*IQR) still lower than the clipping bound. This is okay for now, since we want to preserve the nature of the outlier which we assume is meaningful. Later, maybe we can use algorithms that are robust for outliers.

3. Data Cleaning: Remove Duplicates

Handling duplicate values is crucial because they can compromise data integrity, leading to inaccurate analysis and insights. Duplicate entries can bias machine learning models, causing overfitting and reducing their ability to generalize to new data. They also inflate the dataset size unnecessarily, increasing computational costs and processing times. Additionally, duplicates can distort statistical measures and lead to inconsistencies, ultimately affecting the reliability of data-driven decisions and reporting. Ensuring data quality by removing duplicates is essential for accurate, efficient, and consistent analysis.

I use hashing in my dataset to efficiently detect duplicates in a large dataset with 120,000 rows and 45 columns of network attack data. Since each row has a unique ‘id’, it’s nearly impossible to have exact duplicates, but there could still be partial duplicates or very similar rows. Hashing allows me to summarize the data into a unique value for each row, making it easier to identify and remove near-duplicates without the need for expensive row-by-row comparisons. This approach helps optimize performance and memory usage, which is essential given the dataset's size, while ensuring that I only remove truly redundant data.

4. Data Cleaning: Feature Engineering

Feature engineering involves creating new features (input variables) or transforming existing ones to improve the performance of machine learning models. Feature engineering aims to enhance the model's ability to learn patterns and make accurate predictions from the data. It's often said that "good features make good models". Feature engineering process resulting in giving features that are relevant to the goal of the data analysis process.

Feature engineering steps vary adjusting to the complexity needed for the process goals. One of the feature engineering steps that is implemented in this project data cleaning process is feature selection. In general, there are three most common methods used for feature engineering, which are Filter Method, Wrapper Method, and Embedded Method for Feature Selection. Filter method independently analyze each feature based on a pre-defined metric, such as correlation within the variable. Filter method gives advantages of computationally inexpensive implementation and easiness to implement. Although, filter method has limited interaction with the model. Examples of filter method are information gain, chi-square test, correlation within the independent variable and correlation between independent-dependent variable, and variance threshold. Wrapper method assesses feature subsets by evaluating their performance with the selected machine learning model. This method is more flexible but computationally expensive and greater risk of overfitting. Embedded methods integrate feature selection into the model training process, enabling the model to learn both the relationship between the features and the target variable, as well as identify the most relevant features.

Feature selection process done in this project used a filter method because it enhances the implementation simplicity. Filter method used in the feature selection process are : variance threshold checking for ensuring no column in the table has the same value for all rows. Variance threshold checking will drop column with zero variance which indicates all rows have same value within the column. The process will be continued with quasi-constant features which will dropping column with 99% similarity value within the column. Then, the process will be continued with checking correlation between independent variables because highly correlated independent features can be redundant for a model.

5. Data Preprocessing: Feature Scaling

Data preprocessing is a broader step that encompasses both data cleaning and additional transformations to make the data suitable for machine learning algorithms. Feature scaling ensures that numerical features have similar scales. Common techniques include Min-Max scaling (scaling to a specific range) or standardization (mean-centered, unit variance). Feature scaling is a preprocessing technique used in machine learning to standardize the range of independent variables or features of data. The primary goal of feature scaling is to ensure that all features contribute equally to the training process and that machine learning algorithms can work effectively with the data. Feature scaling is important because machine learning algorithms sensitivity, distance-based algorithms like KNN, regularization techniques like L1 and L2. The common methods include normalization, standardization, robust scaling, and log transformation.

KNN relies on calculating distances between data points. If features have different scales (e.g., dur in float vs proto in categorical), the features with larger ranges will dominate the distance calculation. For Gaussian Naive Bayes, it's also important to scale numerical features to ensure they are on a similar scale since the model assumes normal distribution, hence we will use StandardScaler to scale continuous features.

6. Data Preprocessing: Feature Encoding

Feature encoding, also known as categorical encoding, is the process of converting categorical data (non-numeric data) into a numerical format so that it can be used as input for machine learning algorithms. Most machine learning models require numerical data for training and prediction, so feature encoding is a critical step in data preprocessing. Common methods for encoding are label encoding, one-hot encoding and target encoding.

To decide, first we check the correlation between features and the target variable to decide the encoding method. Using the scipy.stats chi2_contingency we calculate the Cramer’s V for each categorical data to target category. We got this result.

```
Cramér's V for service and attack_cat: 0.28979486965966333
Cramér's V for proto and attack_cat: 0.25621027039091815
Cramér's V for state and attack_cat: 0.26300548651381256
```

We then checked the rare values in the categorical features and got this result. The rare values are only below one percent of the data. There are so many rare values in the proto feature.

```
Rare categories in 'state':
['RST', 'ECO', 'URN']

Rare categories in 'service':
['ssh', 'pop3', 'snmp', 'dhcp', 'ssl', 'irc', 'radius']

Rare categories in 'proto':
['arp', 'sctp', 'any', 'pim', 'gre', 'ipv6', 'sep', 'sun-nd', 'swipe',
'mobile', 'rvd', 'rsvp', 'cbt', 'st2', 'sps', 'leaf-1', 'sdrp', 'dcn',
'vrrp', 'aes-sp3-d', 'trunk-2', 'tcf', 'ipcv', 'egp', 'pup', 'br-sat-mon',
'ddp', 'wsn', 'mux', 'secure-vmtp', 'iplt', 'chaos', 'sccopmce',
'kryptolan', 'encap', 'pri-enc', 'dgp', 'ipv6-opts', 'a/n', 'scps', 'nvp',
'argus', 'qnx', 'igp', 'ax.25', 'snp', 'netblt', 'trunk-1', 'cphb',
'bbn-rcc', 'ib', 'ipv6-no', 'prm', 'compaq-peer', 'cftp', 'sat-expak',
'irtp', 'ipv6-route', 'iatp', 'il', 'uti', 'nsfnet-igp', 'pnni', 'ddx',
'etherip', 'visa', 'ippc', 'ipnip', 'fire', 'cpnx', '3pc', 'eigrp',
'ipv6-frag', 'mfe-nsp', 'tlsp', 'leaf-2', 'aris', 'ip', 'tp++', 'zero',
'sat-mon', 'idpr-cmtp', 'pgm', 'pipe', 'ttp', 'sm', 'ifmp', 'xnet', 'srp',
'vmtp', 'sprite-rpc', 'narp', 'ptp', 'ggp', 'ipcomp', 'l2tp', 'idrp',
'larp', 'skip', 'gmtp', 'rdp', 'mhrp', 'iso-tp4', 'crtp', 'xtp', 'emcon',
'micp', 'stp', 'vines', 'isis', 'fc', 'wb-mon', 'smp', 'wb-expak', 'mtp',
'merit-inp', 'xns-idp', 'hmp', 'idpr', 'ipx-n-ip', 'i-nlsp', 'pvp', 'ipip',
'bna', 'crudp', 'iso-ip', 'igmp', 'icmp', 'rtp']
```

Based on that, we conclude the following, for each categorical feature.

1. State

   The state feature has relatively low cardinality (9 categories). One-hot encoding would result in high dimensionality. There's no natural order to the state categories, so label encoding wouldn't be suitable also.Hence, we use `target encoding` to convert the categorical values to their proportional frequency in the dataset.

2. Service

   The service feature also has a moderate number of cardinality and there's no natural order to the categories, so label encoding wouldn't be suitable. Hence, we use `target encoding` to convert the categorical values to their proportional frequency in the dataset.

3. Proto

   With high cardinality(133), One-hot encoding would result in very high dimensionality, which is impractical and expensive. There's also no meaningful order in the values so label encoding is not considered here. Hence, we use `target encoding` since the cramer's v shows moderate correlation between proto and target. The encoding converts it into numeric values based on the relationship with the target. The encoding is done on the training set only to avoid overfitting.

The implementation is using a class TargetEncoderWrapper inheriting BaseEstimator, TransformerMixin.

7. Data Preprocessing: Handling Imbalanced Dataset

Handling imbalanced datasets is important because imbalanced data can lead to several issues that negatively impact the performance and reliability of machine learning models like biased model performance, misleading accuracy, until poor generalization. The methods to handle imbalanced datasets include resampling methods, evaluation metrics, , algorithmic approaches.

After seeing the distribution of the categorical data, it can be observed that both the training and validation sets have noticeable target imbalance, with the majority class (Normal) dominating and the minority classes being extremely underrepresented. The imbalance in the dataset, especially for classes like Worms, Shellcode, Backdoor, and Analysis, could lead to poor model performance on those underrepresented classes.

To handle the imbalance data, we use SMOTE which will be pipelined also.

```
smote = SMOTE(sampling_strategy='auto', random_state=42)
```

8. Data Preprocessing: Data Normalization

Data normalization is a vital preprocessing step in machine learning used to standardize the scale of numerical features, ensuring that they contribute equally to the model's performance. It minimizes the impact of varying feature magnitudes, particularly in algorithms sensitive to scale, such as gradient-based models and distance-based methods like SVMs or KNN. By transforming the data to have a mean of 0 and a standard deviation of 1, normalization ensures that features follow a standard normal distribution, which helps models converge faster and improves overall stability during training.

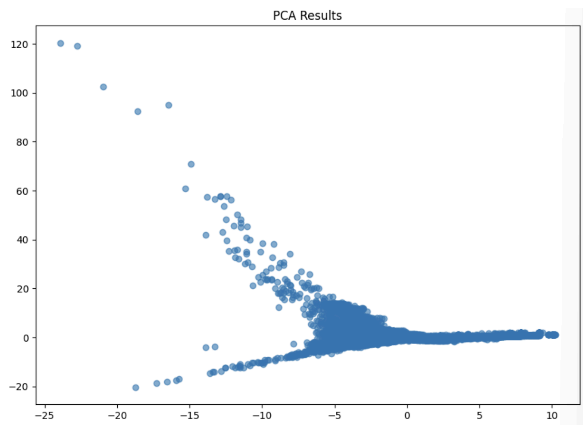
The primary goal of normalization is to enhance the model's ability to learn effectively by reducing biases caused by large-scale features and ensuring numerical stability in computations. Without normalization, features with higher magnitudes can dominate the learning process, leading to suboptimal model performance. By standardizing data, the learning process becomes more balanced, leading to better predictive accuracy and robustness across various datasets.

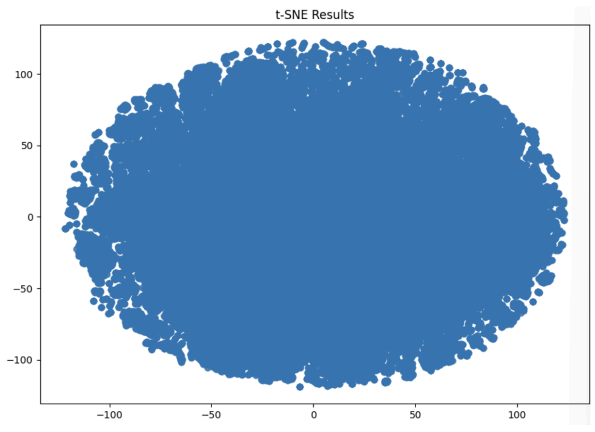
9. Data Preprocessing: Dimensionality Reduction

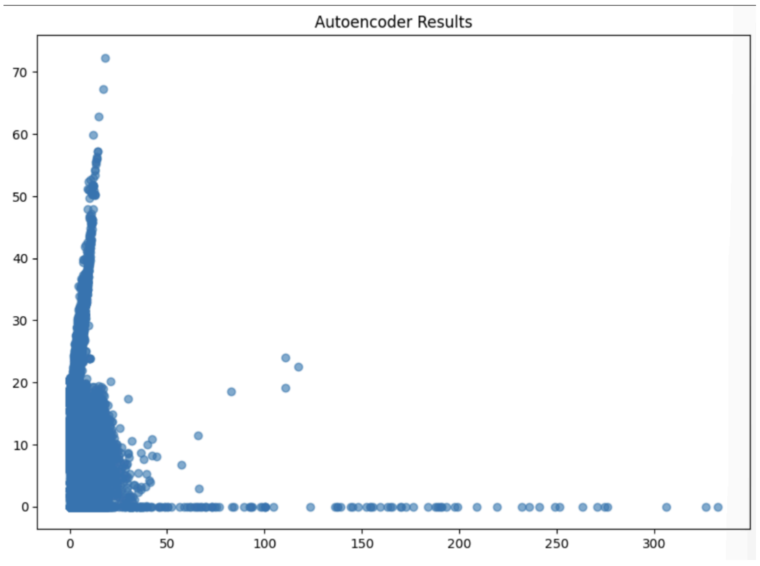
Dimensionality reduction is a technique used in data preprocessing to reduce the number of input features (dimensions) in a dataset while retaining as much important information as possible. It is essential when dealing with high-dimensional data, where too many features can cause problems like increased computational costs, overfitting, and difficulty in visualization. Reducing dimensions simplifies the data, making it easier to analyze and improving the performance of machine learning models.

One of the main approaches to dimensionality reduction is feature extraction. Feature extraction creates new, smaller sets of features that capture the essence of the original data. Common techniques include PCA, t-SNE, and Autoencoders.

Trying some dimensionality reduction techniques and visualizing the reduction results, we got this.







Based on that, we analyze the following, for each categorical feature.

1. PCA

   Based on the explained variance ratio, around 72% of the total variance in the original data is retained after reducing it to 10 components. While the first few components capture most of the variance, the later components contribute less and less. Additionally, with reconstruction error valued at 27.8%, it shows that a notable amount of information is lost and there is a need for some adjustment on n_components value to capture more variance.

2. t-SNE

   Scatter plot for t-SNE displays a circular and even spread since it does not focus on global variance and more often used to show local neighborhoods or patterns. However, as shown in the scatter plot, there are no clear separable clusters in the reduced data.

3. AutoEncoder

   The AutoEncoder scatter plot shows most points concentrated along the vertical axis with some widely spread horizontal outliers. This suggests that the model may require tuning, as the compressed output appears uneven and less organized compared to PCA or t-SNE and its 74.5% error reconstruction value.

After comparing the three methods above, we have decided to use PCA with some changes on the n_components variable to achieve higher variance ratio and lower error reconstruction value. We got that Threshold met at 24 components with 0.952670319097962 variance.

IMPLEMENTASI ALGORITMA MACHINE LEARNING

1. Implementasi Algoritma K-Nearest Neighbors

The k-Nearest Neighbors (kNN) algorithm is a simple, non-parametric, and instance-based machine learning method used for classification and regression tasks. It works by identifying the ‘ k’ nearest data points (neighbors) to a given input point in the feature space, based on a distance metric like Euclidean, Minkowski, or Manhattan distance. Since kNN does not rely on a predefined model, it is highly flexible but computationally expensive for large datasets, as it requires searching through the entire dataset for each prediction. Proper choice of k and feature scaling are crucial to the algorithm’s performance.

In our code, we have two parameters for our kNN class, that is the ‘k’ neighbors and the distance algorithm to calculate the distance between data points. Below is the explanation of the code.

Initialize the KNN classifier.
Args:
- k: Number of nearest neighbors.
- distance_metric: Distance metric to use ('euclidean', 'manhattan', 'minkowski').
- p: Parameter for Minkowski distance (default: 2).
	`class sKNNClassifier:`
	`def __init__(self, k=3, distance_metric='euclidean', p=3):`
	`        self.k = k`
	`        self.distance_metric = distance_metric`
	`        self.p = p`

This function store the training data as the object attribute
Args:
- x_train: Training data features (2D array).
- y_train: Multi-target labels (2D array, each row contains [label, attack_cat]).
	`def fit(self, x_train, y_train):`
	`        self.x_train = x_train`
	`        self.y_train = y_train`
	`        self.m, self.n = x_train.shape`

This function predicts multi-target labels for test data.
Args:
- x_test: Test data features (2D array).
Returns:
- Predicted multi-target labels (2D array).
First, it makes an array at the size of the test data. For each data points in the test dataset, it calculates the distance to the train dataset and get the k nearest
	`def predict(self, x_test):`
	`  m_test = x_test.shape[0]`
	`  y_pred = np.zeros((m_test, self.y_train.shape[1]), dtype=object)`
	
	`  for i in range(m_test):`
	`distances = np.array([self._calculate_distance(x_test[i], x) for x in self.x_train])`
	`k_indices = np.argsort(distances)[:self.k]`

neighbors and then choose the most common target class among that nearest neighbors.	`k_nearest_labels = self.y_train[k_indices]`
	
	`for j in range(self.y_train.shape[1]):`
	`    y_pred[i, j] =`
	`Counter(k_nearest_labels[:,`
	`j]).most_common(1)[0][0]`
	
	`    return y_pred`

This function calculates the distance between two points based on the chosen metric.
Args:
- x1, x2: Two data points (1D arrays).
Returns:
- Distance (float).
	`def _calculate_distance(self, x1, x2):`
	`if self.distance_metric == 'euclidean':`
	`   return np.sqrt(np.sum((x1 - x2) ** 2))`
	`elif self.distance_metric == 'manhattan':`
	`   return np.sum(np.abs(x1 - x2))`
	`elif self.distance_metric == 'minkowski':`
	`  return np.sum(np.abs(x1 - x2) ** self.p)`
	`** (1 / self.p)`

2. Implementasi Algoritma Naive Bayes

Gaussian Naive Bayes is a probabilistic machine learning algorithm based on Bayes' Theorem, used for classification tasks. It assumes that features are independent (Naive assumption) and follow a Gaussian (normal) distribution. For a given input, the algorithm calculates the posterior probability of each class by combining the likelihood of the input features under the class-specific Gaussian distributions and the prior probability of each class. The class with the highest posterior probability is assigned to the input. Despite its simplicity and strong independence assumption, Gaussian Naive Bayes performs well in many real-world scenarios, especially when the independence assumption holds approximately, and it is computationally efficient even for large datasets.

Below is the explanation of the code.

This initiates the Naive Bayes Classifier.
Args:
- smoothing (float): A small value added to probabilities to avoid zero probabilities for unseen feature values.
 	
It sets up the placeholders for storing prior probabilities (priors), likelihoods (likelihoods), class labels (classes), and the smoothing parameter (smoothing). Smoothing helps handle cases where certain feature values are not present in the training data for a specific class.
	`class NaiveBayesClassifier:`
	`    def __init__(self, smoothing=1e-6):`
	`        self.priors = {}`
	`        self.likelihoods = {}`
	`        self.classes = None`
	`        self.smoothing = smoothing`

This function trains the Naive Bayes model by calculating the prior probabilities of each class and the conditional probabilities (likelihoods) of feature values given a class.
Args:
- X (DataFrame): Training data features.
- y (Series): Target labels corresponding to the training data.
 	
For each class, it isolates the subset of training data corresponding to that class and calculates:
- The prior probability (P(H)) as the fraction of samples in that class.
- The likelihood P(E∣H)) for each feature value using the _calculate_likelihood helper function, which applies Laplace smoothing.
	`def fit(self, X, y):`
	`    self.classes = np.unique(y)`
	`    self.priors = {}`
	`    self.likelihoods = {cls: {} for cls in self.classes}`
	
	`    for cls in self.classes:`
	`       X_cls = X[y == cls]`
	`       self.priors[cls] = len(X_cls) / len(X)`
	
	`        for col in X.column:`
	`self.likelihoods[cls][col] = self._calculate_likelihood(X_cls[col])`

This function predicts the class for a single data point by calculating the posterior probabilities of each class using ‘predict_proba_single’. The class with the highest posterior probability is selected as the prediction.
Args:
- instance (Series): A single row of feature values representing a test data point.
Return:
- predicted_class (str or int): The predicted class label for the instance.
	`def predict_single(self, instance):`
	`        posteriors = self.predict_proba_single(instance)`
	
	`        return max(posteriors, key=posteriors.get)`

This function calculates the posterior probabilities for each class for a single instance. Starting with the prior probability of the class (P(H)), it iteratively multiplies it by the likelihood (P(E∣H)) for each feature value. Logarithms are used to avoid numerical underflow during multiplication. Finally, the posterior probabilities are normalized to ensure they sum to 1.
 	
Args:
- instance (Series): A single row of feature values representing a test data point.
Return:
- probabilities (dict): A dictionary where each key is a class label, and the value is the normalized probability of that class for the given instance.
	`def predict_proba_single(self, instance):`
	`        posteriors = {}`
	`        for cls in self.classes:`
	`   posterior = np.log(self.priors[cls])`
	
	`   for feature, value in instance.items():`
	`   likelihood = self.likelihoods[cls].get(feature, {})`
	`   posterior += np.log(likelihood.get(value, likelihood["__unseen__"]))`
	
	`    posteriors[cls] = np.exp(posterior)`
	
	`        total = sum(posteriors.values())`
	`        probabilities = {cls: prob / total for cls, prob in posteriors.items()}`
	
	`        return probabilities`

This function predicts class labels for the entire test dataset. It iterates through each row of the test data, calls predict_single to get the predicted class for that row, and stores the results in a Series.
Args:
- X (DataFrame): Test data features.
Return:
- predictions (Series): Predicted class labels for each instance in the test dataset.
	```
	def predict(self, X):
	        predictions = []
	        for _, instance in X.iterrows():
	            prediction =
	self.predict_single(instance)
	            predictions.append(prediction)

	        return pd.Series(predictions)
	```

This function predicts the class probabilities for the entire test dataset. For each row of test data, it calls predict_proba_single to compute the posterior probabilities and stores them in a list. Each dictionary in the list corresponds to the class probabilities for a single test instance.
Args:
- X (DataFrame): Test data features.
Return:
- probabilities (list): A list of dictionaries, where each dictionary contains the class probabilities for a corresponding test instance.
	```
	def predict_proba(self, X):
	        probabilities = []
	        for _, instance in X.iterrows():
	            proba =
	self.predict_proba_single(instance)
	            probabilities.append(proba)

	        return probabilities
	```

3. Implementasi Algoritma ID3

The ID3 (Iterative Dichotomiser 3) algorithm is a decision tree learning method that uses a top-down, greedy approach to recursively split the dataset into subsets based on feature values. It selects the feature that provides the maximum information gain (a measure of how much uncertainty is reduced) with respect to the target variable at each step. Starting from the root, ID3 evaluates all possible splits for features and chooses the one that best separates the data into pure subsets (those with a single class label). The process repeats until the dataset is perfectly classified or no features are left to split on. ID3 assumes all features are categorical and may overfit when applied to noisy or small datasets, but it is a foundational algorithm for more advanced decision tree methods like C4.5 and CART.

Below is the explanation of the code.

This initiates the ID3 Classifier.
Args:
- min_samples_leaf (int): Minimum number of samples required at a leaf node (default: 1).
- max_depth (int or None): Maximum depth of the decision tree (default: None, meaning no limit).
 	
It sets the initial parameters:
tree to hold the trained decision tree.
min_samples_leaf and max_depth to control the complexity of the tree.
majority_class to store the most frequent class in the dataset (used for fallback predictions).
	`class ID3Classifier:`
	`    def __init__(self, min_samples_leaf=1,`
	`max_depth=None):`
	`        self.tree = None`
	`        self.min_samples_leaf =`
	`min_samples_leaf`
	`        self.max_depth = max_depth`
	`        self.majority_class = None`

Calculates the entropy of the target labels using the formula
$E(S) = \sum_{i=1}^{c} - p_i \log_2 p_i$
Args:
- y: A pandas Series or numpy array representing class labels.
Return:
A float representing the entropy of the class distribution.
	`def entropy(self, y):`
	`   counts = Counter(y)`
	`   total = len(y)`
	`        entropy = -sum((count / total) *`
	`np.log2(count / total) for count in`
	`counts.values())`
	
	`    return entropy`

This function calculates how much a feature reduces uncertainty (entropy) in the class labels when used for splitting. It computes the weighted entropy of subsets split by the feature and subtracts it from the total entropy of the dataset.
Args:
- X: Feature matrix (numpy array)
- y: Class labels (numpy array or pandas Series).
- attribute: Index of the feature to evaluate.
Return:
- A float representing the information gain of splitting on the specified feature.
	`def information_gain(self, X, y,`
	`attribute):`
	`        if isinstance(y, np.ndarray):`
	`            y = pd.Series(y)`
	`        total_entropy = self.entropy(y)`
	`        values, counts = np.unique(X[:,`
	`attribute], return_counts=True)`
	`        weighted_entropy = sum(`
	`            (counts[i] / len(y)) *`
	`self.entropy(y[X[:, attribute] ==`
	`values[i]])`
	`            for i in range(len(values))`
	`        )`
	`        return total_entropy -`
	`weighted_entropy`

This function iterates over all features and selects the one with the highest information gain. It identifies the feature that maximally reduces class uncertainty, ensuring an optimal split at each node.
Args:
- X: Feature matrix (numpy array)
- y: Class labels (numpy array or pandas Series).
Return:
	`def best_attribute(self, X, y):`
	`        return max(range(X.shape[1]),`
	`key=lambda attr: self.information_gain(X,`
	`y, attr))`

- An integer representing the index of the best feature to split on.	

This recursive function builds the decision tree by splitting on the best attribute until stopping conditions are met (e.g., all labels are identical, max depth reached, or insufficient samples). Leaves store class distributions.
Args:
- X: Feature matrix (numpy array)
- y: Class labels (numpy array or pandas Series).
- attributes: List of attribute names or indices.
- depth: Current depth of the tree.
Return:
- An integer representing the index of the best feature to split on.
	```
	def build_tree(self, X, y, attributes, depth=0):
	        if isinstance(X, np.ndarray):
	            X = pd.DataFrame(X, columns=attributes)
	        if isinstance(y, np.ndarray):
	            y = pd.Series(y)

	        # If all labels are the same, return that label
	        if len(np.unique(y)) == 1:
	            return {y.iloc[0]: 1.0}

	        # If no more attributes to split or max depth reached, return the majority label
	        if len(attributes) == 0 or (self.max_depth is not None and depth >= self.max_depth):
	            label_counts = y.value_counts(normalize=True).to_dict()
	            return label_counts

	        # If the node has fewer samples than min_samples_leaf, return the majority label
	        if len(y) < self.min_samples_leaf:
	            label_counts = y.value_counts(normalize=True).to_dict()
	            return label_counts

	        # Select the best attribute
	        best_attr = self.best_attribute(X.to_numpy(), y.to_numpy())
	        best_attr_name = attributes[best_attr]
	        tree = {best_attr_name: {}}

	        # Partition data by the values of the best attribute
	        for value in np.unique(X[best_attr_name]):
	            subset_X = X[X[best_attr_name] == value]
	            subset_y = y[X[best_attr_name] == value]

	            if len(subset_X) < self.min_samples_leaf:
	                label_counts = y.value_counts(normalize=True).to_dict()
	                tree[best_attr_name][value] = label_counts
	            else:
	```

	```
	                remaining_attrs = [attr
	for attr in attributes if attr !=
	best_attr_name]

	tree[best_attr_name][value] =
	self.build_tree(subset_X, subset_y,
	remaining_attrs, depth + 1)

	        return tree
	```

This procedure trains the classifier by first determining the majority class (to handle edge cases in prediction) and then constructing the decision tree using build_tree.
 	
Args:
- X (DataFrame): Feature matrix (numpy array or pandas DataFrame).
- y: Class labels (numpy array or pandas Series).
	```
	def fit(self, X, y):
	        if isinstance(y, np.ndarray):
	            y = pd.Series(y)
	        mode_result = mode(y, axis=None)
	        self.majority_class =
	mode_result.mode[0] if
	isinstance(mode_result.mode, np.ndarray)
	else mode_result.mode
	        self.tree = self.build_tree(X, y,
	list(X.columns) if isinstance(X,
	pd.DataFrame) else range(X.shape[1]))
	```

This function navigates the tree recursively based on the values of the instance features until a leaf is reached, where it predicts the class with the highest frequency.
 	
Args:
- instance: A single data point (pandas Series or dictionary).
- tree: The decision tree (nested dictionary).
Return:
- Predicted class label.
	```
	 def predict_single(self, instance, tree):
	        print("Predicting single instance
	", instance)
	        if not isinstance(tree, dict):
	            return max(tree, key=tree.get)

	        root_attribute = next(iter(tree))
	        subtree =
	tree[root_attribute].get(instance[root_att
	ribute], None)

	        if subtree is None:
	            return self.majority_class

	        return
	self.predict_single(instance, subtree)
	```

This function applies predict_single to each instance in the dataset, generating predictions for all samples.
 	
Args:
- X: Feature matrix (numpy array or pandas DataFrame).
Return:
A pandas Series containing predicted class labels.
	```
	def predict(self, X):
	        print("Predicting labels")
	        if isinstance(X, np.ndarray):
	            X = pd.DataFrame(X)
	        predictions = []
	        for _, instance in X.iterrows():
	            prediction =
	self.predict_single(instance, self.tree)
	            predictions.append(prediction)
	        return pd.Series(predictions)
	```

This function traverses the tree to return the probability distribution over classes at a leaf. If the instance doesn't match any path, the majority class is assigned a probability of 1.0.
 	
Args:
	```
	def predict_proba_single(self, instance,
	tree):
	        print("Predicting probabilities
	for single instance ", instance)
	        if not isinstance(tree, dict):
	            return tree
	```

- instance: A single data point (pandas Series or dictionary).
- tree: The decision tree (nested dictionary).

Return:
- A dictionary of class probabilities.

```
        root_attribute = next(iter(tree))
        subtree =
tree[root_attribute].get(instance[root_att
ribute], None)

        if subtree is None:
            return {self.majority_class:
1.0}

        return
self.predict_proba_single(instance,
subtree)
```

This function computes class probabilities for all instances in the dataset by calling predict_proba_single for each one.

Args:
- X: Feature matrix (numpy array or pandas DataFrame).

Return:
A list of dictionaries containing class probabilities for each instance.

```
 def predict_proba(self, X):
        print("Predicting probabilities")
        if isinstance(X, np.ndarray):
            X = pd.DataFrame(X)
        probabilities = []
        for _, instance in X.iterrows():
            print("instance: ", instance)
            proba =
self.predict_proba_single(instance,
self.tree)
            probabilities.append(proba)
        return probabilities
```

EKSPERIMEN

1. Eksperimen Algoritma KNN

Metriks	Scratch Model	Sklearn Model

2. Eksperimen Algoritma Naive-Bayes

Metriks	Scratch Model	Sklearn Model
Weighted F1	0.6592	0.5920
Macro F1	0.3545	0.3033
Cohen's Kappa	0.5368	0.4540
ROC-AUC	0.9170	0.8804
Validation Log Loss	1.8194	1.5178

When comparing our scratch-built Gaussian Naive Bayes model with the one from the \`sklearn\` library, we observe that our model performs better across most evaluation metrics. Specifically, Weighted F1, Macro F1, Cohen's Kappa, and ROC-AUC indicate that our model achieves superior classification performance and inter-rater agreement. However, the Validation Log Loss for our model is higher compared to the \`sklearn\` model, suggesting that the \`sklearn\` implementation produces better-calibrated probabilistic predictions.

Overall, while our scratch model demonstrates stronger classification performance, the \`sklearn\` model's lower log loss indicates it handles probability estimation more effectively, likely due to numerical optimizations or smoothing techniques. This highlights a trade-off between classification accuracy and probabilistic calibration.

3. Eksperimen Algoritma ID3

Metriks	Scratch Model	Sklearn Model

PENUTUP

1. Kesimpulan

Berbagai metode processing dan cleaning dapat membantu pemodelan data dengan lebih baik.

2. Saran

Melakukan banyak pemodelan dan tidak di akhir waktu agar bisa lebih banyak waktu melakukan eksplorasi

3. Pembagian Tugas

Penanggung Jawab		Tugas
Kayla Namira Mariadi	13522050	Preprocessing, Model (KNN), Pipelining
Andhita Naura Hariyano	13522060	Preprocessing
Salsabiila	13522062	Preprocessing, Model (ID3, GNB), Pipelining
Shulha	13522087	Cleaning, Model (KNN), Report

REFERENSI

https://www.geeksforgeeks.org/k-nearest-neighbours/

https://www.geeksforgeeks.org/gaussian-naive-bayes/