0.1 One-Hot Encoding (OHE)

Used for nominal (unordered) categorical features. It creates a new binary column for each category. 'handle_unknown="ignore"': If an unseen category appears during testing/validation, it's ignored, and the resulting one-hot vector will have zeros for all category columns. 'sparse_output=False': Returns a dense NumPy array instead of a sparse matrix. Binary Features ('drop="if_binary"'): For a feature with only two categories (e.g., "yes"/"no"), this argument creates only one resulting column (e.g., 1 for "yes" and 0 for "no"), avoiding redundancy.

0.2 Ordinal Encoding

Used for ordinal (ordered) categorical features (e.g., "Good", "Average", "Poor"). It maps each category to an integer value. Difference from OHE: Ordinal encoding uses a single column of integers, assuming an order. OHE uses multiple binary columns and assumes no order. rdering: Manually ordering categories is important to correctly reflect the underlying relationship. Unknown Categories: Without careful configuration, an unknown category during testing/validation could lead to an error or an arbitrary value assignment, impacting model performance. Multiple Ordinal Columns: Pass a **list of lists** to the 'categories' parameter of 'OrdinalEncoder', where each inner list contains the ordered categories for one column.

0.3 Cases where it's OK to break the golden rule

If We know the categories in advance and pass the list of known/-possible categories. It's OK to incorporate human knowledge in the model.

0.4 Text Data Representation (NLP)

Raw text data (e.g., SMS, emails) has no fixed length, so it needs to be transformed into a fixed-length numerical representation. Popular Representations: Bag-of-Words (BOW), TF-IDF, and Embedding representations.

0.5 CountVectorizer

Converts a collection of text documents (the **corpus**) into a matrix of token counts (Bag-of-Words representation).

- **Output Structure:** Each row is a document, and each column is a unique word (token) in the vocabulary. The cell value is the word count in that document.
- **Preprocessing:** By default, performs **lowercasing** and gets rid of **punctuation**.
- **Data Type:** 'fit_transform' expects a **Series** of text, unlike other transformers that take a DataFrame.
- **Sparse Matrix:** The output is typically a Compressed Sparse Row (CSR) matrix because most documents only contain a small subset of the total vocabulary, leading to huge computational savings by only storing non-zero elements.

0.6 CountVectorizer Key Hyperparameters

- **'binary=True':** Uses presence/absence (1 or 0) of words instead of their counts.
- **'max_features':** Only considers the top k most frequent words in the corpus, controlling the number of features.
- **'max_df' / 'min_df':** Ignores words that occur in too many (e.g., 90% of documents - potentially a stopword) or too few documents, respectively.
- **'ngram_range':** Considers sequences of words (n-grams) instead of single words.

0.7 Discretizing (Binning)

The process of transforming numeric features into categorical features.

- **Purpose:** Easier interpretation, maintaining privacy (e.g., children-5, teenager-15, young adults-25), or capturing non-linear relationships in linear models for easier interpretation or to maintain privacy, or to capture non-linear relationships.
- **'sklearn' Tool:** Use the 'KBinsDiscretizer' transformer.

0.8 Automated Hyperparameter Search

Treats hyperparameter selection as a large search problem.

- **Why it's needed:** Poor hyperparameters can lead to an underfit or overfit model.
- **Methods:** Manual (expert knowledge) or Data-driven (Automated) optimization.

0.9 Exhaustive Grid Search: 'GridSearchCV'

- **Cost:** The required number of models grows exponentially with the number of hyperparameters. E.g., 5 hyperparameters with 10 values each means 10^5 CV folds.
- **Pipeline Syntax:** Hyperparameters in a pipeline are accessed using a double underscore ('__'), e.g., 'svc__C' or 'columntransformer__countvectorizer__max_features'.
- **Final Model:** After finding the best hyperparameters ('best_params_' based on 'best_score_'), it automatically retrains a model on the entire training set with these best settings.

0.10 Randomized Search: 'RandomizedSearchCV'

- **How it works:** Samples configurations randomly from the defined hyperparameter space until a budget ('n_iter') is exhausted. It doesn't check every combination.
- **Advantages:** Faster than Grid Search, especially when the number of hyperparameters is large. It's more likely to find important parameters because it's not wasting time exploring useless parameter values.
- **Distributions:** Can draw values from continuous probability distributions (like 'loguniform' for C or 'gamma') instead of discrete lists.
- **Recommendation:** Generally recommended over 'GridSearchCV', as it can explore the search space more effectively for a given computational budget.

```
from sklearn.model_selection import GridSearchCV
# pipe_svm includes a preprocessor and an SVC
param_grid = {
    "svc__gamma": loguniform(1e-5, 1e3), # Exponential ranges are
        common
    "svc__C": uniform(0.1, 1e4),
}
gs = GridSearchCV(pipe_svm, param_grid=param_grid, n_jobs=-1)
gs = RandomizedSearchCV(pipe_svm, param_distributions=param_dist,
     n_iter=100, # Number of combinations to try, n_jobs=-1,
     return_train_score=True, random_state=123)
gs.fit(X_train, y_train)
gs.best_score_
results = pd.DataFrame(gs.cv_results_)
gs.score(X_test, y_test)
```

0.11 Optimization Bias (Overfitting the Validation Set)

- **Concept:** When you search over a huge number of hyperparameter combinations, you risk getting a model with a seemingly low cross-validation error purely by chance (similar to taking many random tests and picking the best one).
- **The effect:** The 'best_score_' (CV score) becomes an overly optimistic estimate of the model's true performance.
- **Solution:** This is why we must use a completely separate Test Set for the final evaluation ('gs.score(X_test, y_test)'). The test score is the true, unbiased estimate of generalization performance.

Naive Bayes Classifier

Bernoulli Naive Bayes classifier is a variant of the Naive Bayes algorithm primarily used for **discrete binary data**. It is particularly effective for text classification tasks where features represent whether a certain word *occurs* or *does not occur* in a document.

Binary Feature Vectors: This classifier assumes that all feature vectors are binary (0s and 1s). For a document, a feature x_i is 1 if the word is present and 0 if it is absent.

Naive Assumption: it assumes that the features are **conditionally independent** given the class C_k.

Smoothing (Additive/Laplace Smoothing): To prevent zero probabilities when a feature does not occur with a certain class, Laplace smoothing is applied:,The default smoothing parameter α is typically 1. High α means overfitting (means we are adding large counts to everything and so we are diluting the data), low means underfitting.

Gaussian Naive Bayes This variant is used when features are continuous.

Feature Vectors: Assumed to be continuous and normally distributed. If not make it, using powerTranformer() (e.g., height, temperature)

Conditional Probability (Likelihood): The likelihood $P(x_i|C_k)$ is calculated using the Probability Density Function (PDF) of the Normal distribution

Training: The model simply needs to compute the μ_{ik} and σ^2_{ik} for every feature i and every class C_k from the training data. There is no concept of "smoothing" as used in discrete models.

```
def gaussian_pdf(x, mean, variance):
    return (1 / np.sqrt(2 * np.pi * variance)) * np.exp(-np.power(x
        - mean, 2) / (2 * variance))
observed_weight = 106
observed_sugar_content = 11
likelihoods = {}
for fruit in ['Apple', 'Orange']:
    likelihoods[fruit] = {}
    for feature, observed_value in [('Weight (in grams)',
        observed_weight), ('Sugar Content (in %)',
        observed_sugar_content)]:
        mean = df[df['Fruit'] == fruit][feature].mean()
        variance = df[df['Fruit'] == fruit][feature].var()
        likelihoods[fruit][feature + "=" + str(observed_value)] =
            gaussian_pdf(observed_value, mean, variance)
```

Linear Models: Core Concepts

Linear models constitute a fundamental class of algorithms that make a prediction $\hat{y}$ using a **linear function** of the input features $\mathbf{x}$: where $\mathbf{w}$ is the vector of coefficients (weights) and b is the bias (intercept). Linear Regression (Regression): Predicts a **continuous** output $\hat{y} \in \mathbb{R}$. It optimizes the loss function (e.g., Mean Squared Error) to minimize the distance between $\hat{y}$ and the true output y. Logistic Regression (Classification): Predicts a **probability score** for binary classification. The linear output is passed through the sigmoid function to map it to $[0, 1]$. Linear Support Vector Machine (Linear SVM): A classification model that finds the hyperplane with the largest margin separating the classes. It is generally robust and efficient.

Model Parameters and Optimization

Hyperparameter: A parameter whose value is set **prior** to the training process (e.g., α in Lasso/Ridge regularization, C in SVMs). It controls the model's complexity and learning rate. Interpretation of Coefficients (w_i): The magnitude and sign of a coefficient w_i indicate the strength and direction of the relationship between the feature x_i and the prediction $\hat{y}$, assuming all other features remain constant.

Probability Scores and Functions

Predicting Probability Scores: For classification, linear models often output a score that can be interpreted as the probability of belonging to a certain class (e.g., in Logistic Regression). The Sigmoid Function (σ): Used in binary classification (like Logistic Regression) to map the linear output (logit) to a probability between 0 and 1.

Sigmoid vs. Softmax:

Sigmoid: Used for **binary classification** or independent multi-label classification. Softmax: Used for **multi-class classification** (one class mutually exclusive from others). It converts a vector of scores into a probability distribution that sums to 1.

Case Analysis

For a classification task predicting $P(\text{Class A})$: Most Confident Cases: Predictions closest to 0 or 1 (e.g., 0.01 or 0.99). These are cases where the model is highly certain about its classification. Least Confident Cases: Predictions closest to the decision boundary (e.g., 0.49 or 0.51). These are cases where the model is uncertain and the data point is near the separating hyperplane. Over Confident Cases: High probability predictions (near 0 or 1) that turn out to be **wrong**. This often indicates poor calibration or overfitting.

Multi-Class Classification

Handled either by extending the linear function to output a score for each class (e.g., using the **Softmax** function) or by decomposing the problem into multiple binary problems (e.g., One-vs-Rest).

0.12 Strengths of Linear Models:

Interpretability: Coefficients provide clear insight into feature importance. Efficiency: Fast to train and predict, and scale well to large datasets. Simplicity: Less prone to overfitting on low-dimensional data.

0.13 Limitations of Linear Models:

Limited Expressiveness: Cannot capture complex non-linear relationships without manual feature engineering (e.g., adding polynomial features). Sensitivity to Outliers: Especially for Linear Regression, they can be easily skewed by extreme values. Independence Assumption: Performance suffers if features are highly dependent on each other.