0.1 Preprocessing Categorical Features

Preprocessing is crucial for preparing data for machine learning models.

0.2 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.

0.3 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.
- **Ordering:** 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.

```
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder,
      StandardScaler
ohe = OneHotEncoder(drop="if_binary", dtype=int,
      sparse_output=False)
ohe_encoded = ohe.fit_transform(grades_df[['col']]).ravel()
oe = OrdinalEncoder()
oe_encoded = oe.fit_transform(grades_df[['col']]).ravel()
data = { "oe_encoded": oe_encoded,
         "ohe_encoded": ohe_encoded}
pd.DataFrame(data)
```

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

If it's some fix number of categories. For example, if it's something like provinces in Canada or majors taught at UBC. We know the categories in advance and this is one of the cases where it might be OK to violate the golden rule and pass the list of known/possible categories. It's OK to incorporate human knowledge in the model.

0.5 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.

0.6 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, 'CountVectorizer' 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.

```
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer() # X_counts is a sparse matrix of word counts
X_counts = vec.fit_transform(toy_df["sms"])
```

0.7 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.

0.8 Discretizing (Binning)

The process of transforming numeric features into categorical features.

- **Purpose:** Easier interpretation, maintaining privacy (e.g., grouping ages), or capturing non-linear relationships in linear models.
- **'sklearn' Tool:** Use the 'KBinsDiscretizer' transformer.
- **Example**: You might want to group ages into categories like children, teenager, young adults, middle-aged, and seniors for easier interpretation or to maintain privacy, or to capture non-linear relationships in linear models.

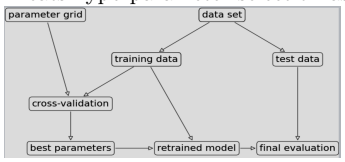
0.9 Hyperparameter Optimization Motivation

The fundamental goal of supervised machine learning is to generalize beyond the training data. We use cross-validation (CV) on the training set to tune hyperparameters (settings not learned from data, like C or 'max_depth') and approximate the generalization error.

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

0.10 Automated Hyperparameter Search

Treats hyperparameter selection as a large search problem.



0.11 Exhaustive Grid Search: 'GridSearchCV'

- **How it works:** Evaluates every possible combination (the Cartesian product) using cross-validation.
- **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.

```
from sklearn.model_selection import GridSearchCV
# pipe_svm includes a preprocessor and an SVC
param_grid = {
    "svc__gamma": [0.001, 0.1, 10], # Exponential ranges are common
    "svc__C": [0.1, 1.0, 100],
}
gs = GridSearchCV(pipe_svm, param_grid=param_grid, n_jobs=-1)
gs.fit(X_train, y_train)
gs.best_score_
best_C = gs.best_params_['svc__C']
results = pd.DataFrame(gs.cv_results_)
gs.score(X_test, y_test) #use the best parameters for scoring test
```

0.12 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 RandomizedSearchCV
from scipy.stats import loguniform
param_dist = {
  "svc__C": loguniform(1e-3, 1e3), # Sample log-uniform distribution
  "svc__gamma": loguniform(1e-5, 1e3),
}
rs = 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
  )
rs.fit(X_train, y_train)
```

0.13 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.