return nb

Run the model

model = run_naive_bayes(merged_df)

explain this code

This code implements a **custom Naive Bayes classifier** from scratch (without using external libraries like scikit-learn) to predict categorical outcomes based on discrete features.

Overview

Naive Bayes is a **probabilistic classification algorithm** based on Bayes' theorem. It assumes that all features are **independent given the class**, which simplifies computation of probabilities.

The formula computed is:

$$P(\text{Class}|\text{Features}) \propto P(\text{Class}) \prod_i P(\text{Feature}_i|\text{Class})$$

The predicted class is the one with the **highest posterior probability**.

Class: `NaiveBayes`

This class has three main methods: `__init__`, `fit`, and `predict`.

1. `__init__(self)`

Initializes two dictionaries:

- `self.class_prob`: Stores prior probabilities $ P(Class) $
- `self.feature_prob`: Stores conditional probabilities $ P(Feature|Class) $

2. `fit(self, X, y)`

Trains the model using the training data.

Steps:

- **Calculate prior probabilities $ P(Class) $:**
  For each unique class in `y`,
  $$P(\text{Class}) = \frac{\#\text{ samples in class}}{\text{total samples}}$$
  Stored in `self.class_prob`.
- **Calculate conditional probabilities $ P(Feature| Class) $:**
  For each column (feature) and each class, counts how often each feature value appears

within that class.
Uses **Laplace smoothing**:

$$P(\text{Feature=val}|\text{Class}) = \frac{\text{count} + 1}{\#\text{class samples} + \#\text{unique feature values}}$$

This avoids zero probabilities when a feature value is unseen for a class.

All probabilities are stored in a nested dictionary structure:

```
self.feature_prob = {
   column_name: {
       class_value: { feature_value: probability }
   }
}
```

3. `predict(self, X)`

Makes predictions for the given feature dataframe.

Steps:

- For each row in `X`:
  - Initialize a `class_scores` dictionary.
  - For each possible class:
    - Start with the prior `P(Class)`.
    - Multiply by each feature's conditional probability $ P(Feature value| Class) $.
      If a feature value was unseen during training, multiply by a small fallback probability (0.0001).
  - The final score for a class is proportional to $ P(Class| Features) $.
  - Choose the class with the highest score as the prediction.

Returns a list of predicted classes.

Function: `run_naive_bayes(merged_df)`

This function prepares the data, trains the model, and evaluates it.

Steps:

1. **Feature and target selection:**

   ```
   features = ['TeamName', 'DriverCode', 'Weather_Condition']
   target = 'Status'
   ```

2. **Data cleaning:** Drops any rows with missing values in those columns.
3. **Train-test split:** Uses 70% for training and 30% for testing.
4. **Model training:**

```
nb = NaiveBayes()
nb.fit(X_train, y_train)
```

5. **Prediction:**

```
predictions = nb.predict(X_test)
```

6. **Accuracy calculation:**

$$\text{Accuracy} = \frac{\#\text{ correct predictions}}{\text{total test samples}}$$

7. **Results printed:**
   - Total number of test samples
   - Correct predictions count
   - Overall accuracy (percentage)
8. **Return:** The trained `NaiveBayes` model instance.

Example Output

If 70% of records are used for training and the model performs well, you might see something like:

```
Total Test Samples: 60
Correct Predictions: 48
Accuracy: 80.00%
```

Would you like me to show how to modify this code to handle **continuous numeric features** (e.g., by modeling them with Gaussian distributions)?