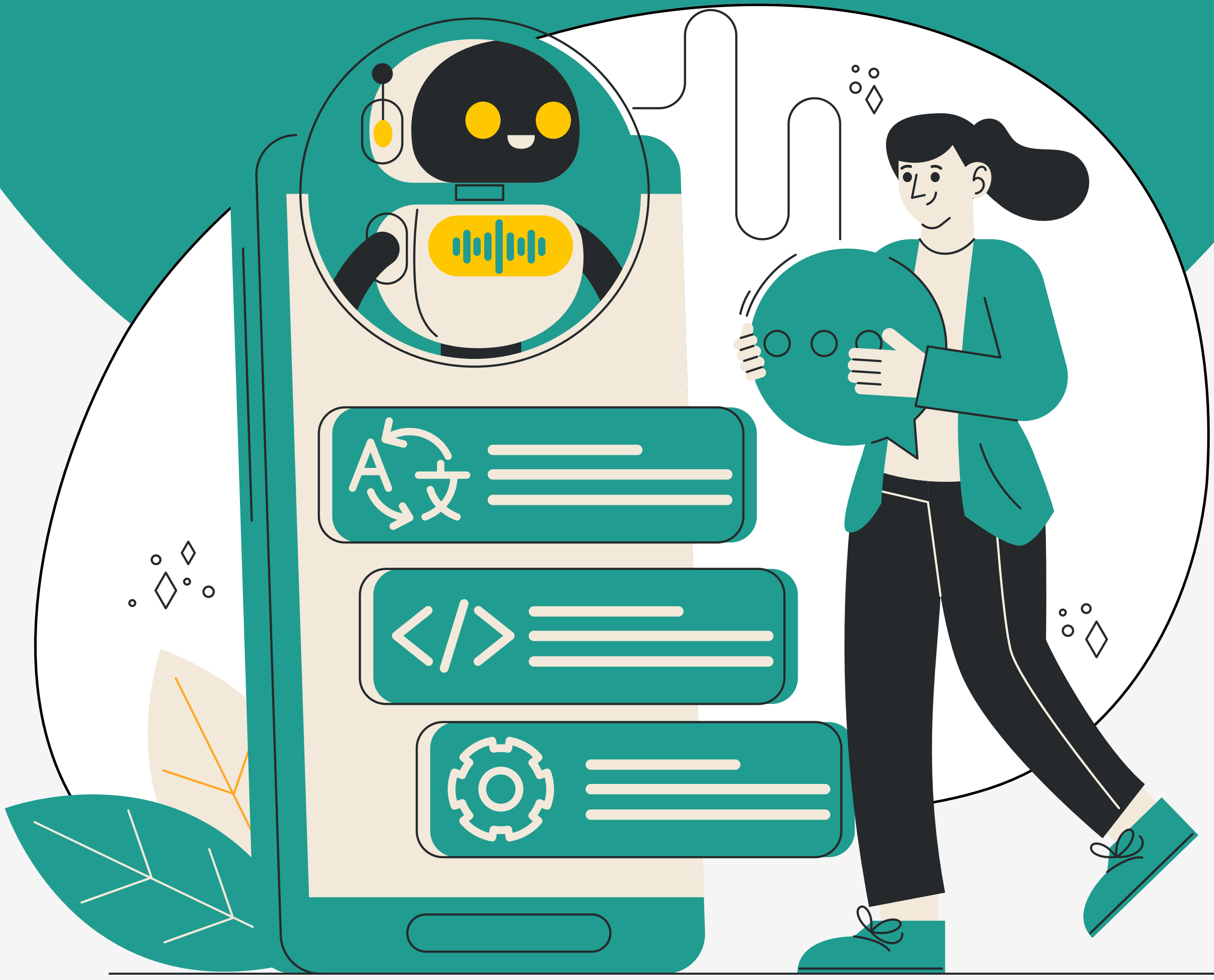
Interview Questions-2

Naive Bayes

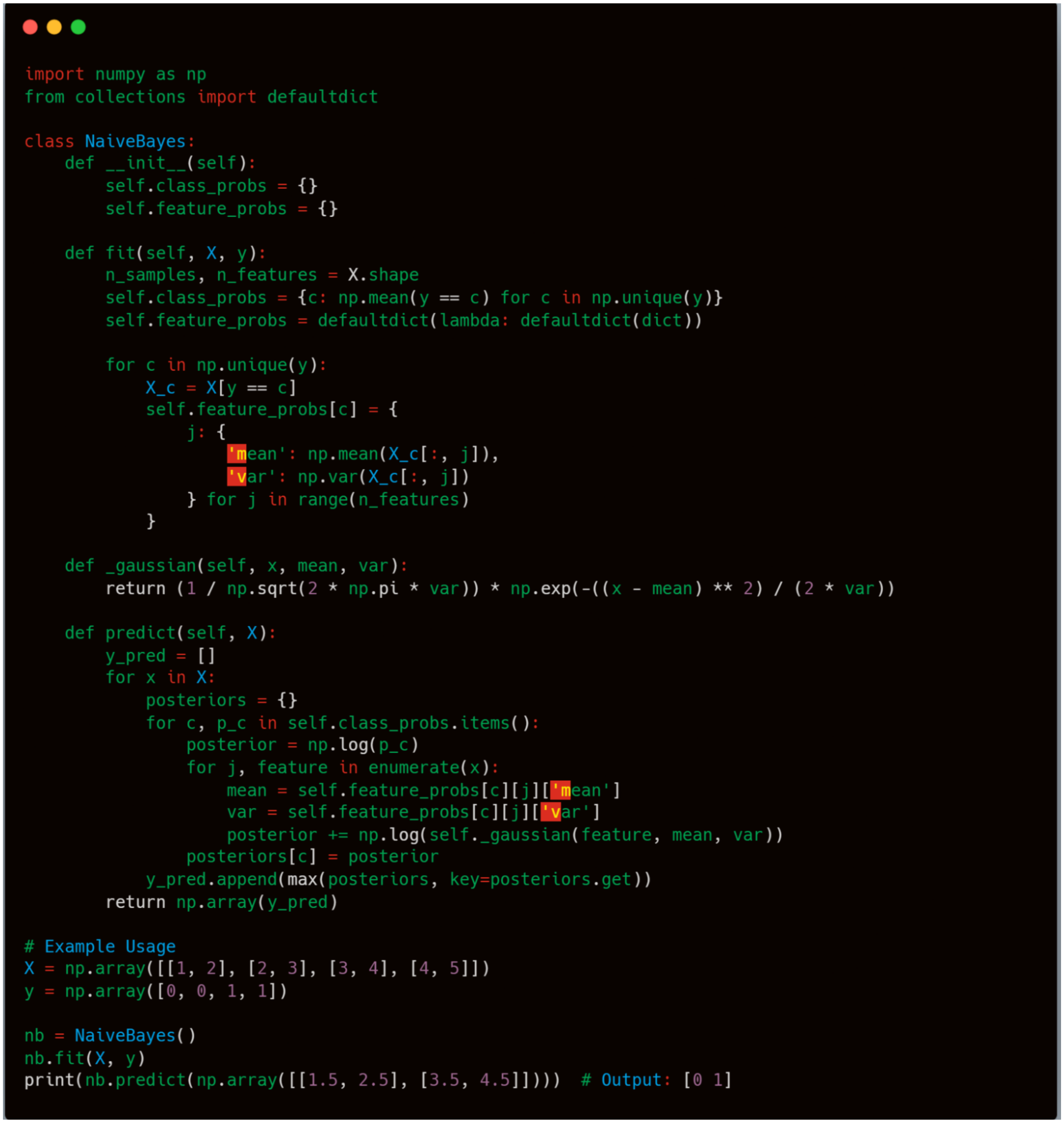
(Practice Projects)



PW SKILLS

1. Implement a Naive Bayes classifier from scratch in Python for a binary classification problem.

Answer:



```python
import numpy as np
from collections import defaultdict

class NaiveBayes:
    def __init__(self):
        self.class_probs = {}
        self.feature_probs = {}

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.class_probs = {c: np.mean(y == c) for c in np.unique(y)}
        self.feature_probs = defaultdict(lambda: defaultdict(dict))

        for c in np.unique(y):
            X_c = X[y == c]
            self.feature_probs[c] = {
                j: {
                    'mean': np.mean(X_c[:, j]),
                    'var': np.var(X_c[:, j])
                } for j in range(n_features)
            }

    def _gaussian(self, x, mean, var):
        return (1 / np.sqrt(2 * np.pi * var)) * np.exp(-((x - mean) ** 2) / (2 * var))

    def predict(self, X):
        y_pred = []
        for x in X:
            posteriors = {}
            for c, p_c in self.class_probs.items():
                posterior = np.log(p_c)
                for j, feature in enumerate(x):
                    mean = self.feature_probs[c][j]['mean']
                    var = self.feature_probs[c][j]['var']
                    posterior += np.log(self._gaussian(feature, mean, var))
                posteriors[c] = posterior
            y_pred.append(max(posteriors, key=posteriors.get))
        return np.array(y_pred)

# Example Usage
X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])
y = np.array([0, 0, 1, 1])

nb = NaiveBayes()
nb.fit(X, y)
print(nb.predict(np.array([[1.5, 2.5], [3.5, 4.5]])))  # Output: [0 1]
```

2.Write a function to calculate the class conditional probabilities for a Gaussian Naive Bayes classifier.

Answer:

```python
import numpy as np

def calculate_class_conditional_probs(X, y):
    class_conditional_probs = {}
    for c in np.unique(y):
        X_c = X[y == c]
        class_conditional_probs[c] = {
            'mean': np.mean(X_c, axis=0),
            'variance': np.var(X_c, axis=0)
        }
    return class_conditional_probs

# Example Usage
X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])
y = np.array([0, 0, 1, 1])

class_conditional_probs = calculate_class_conditional_probs(X, y)
print(class_conditional_probs)
# Output:{0: {'mean': array([1.5, 2.5]), 'variance': array([0.25, 0.25])}, 1: {'mean': array([3.5,
4.5]), 'variance': array([0.25, 0.25])}}
```

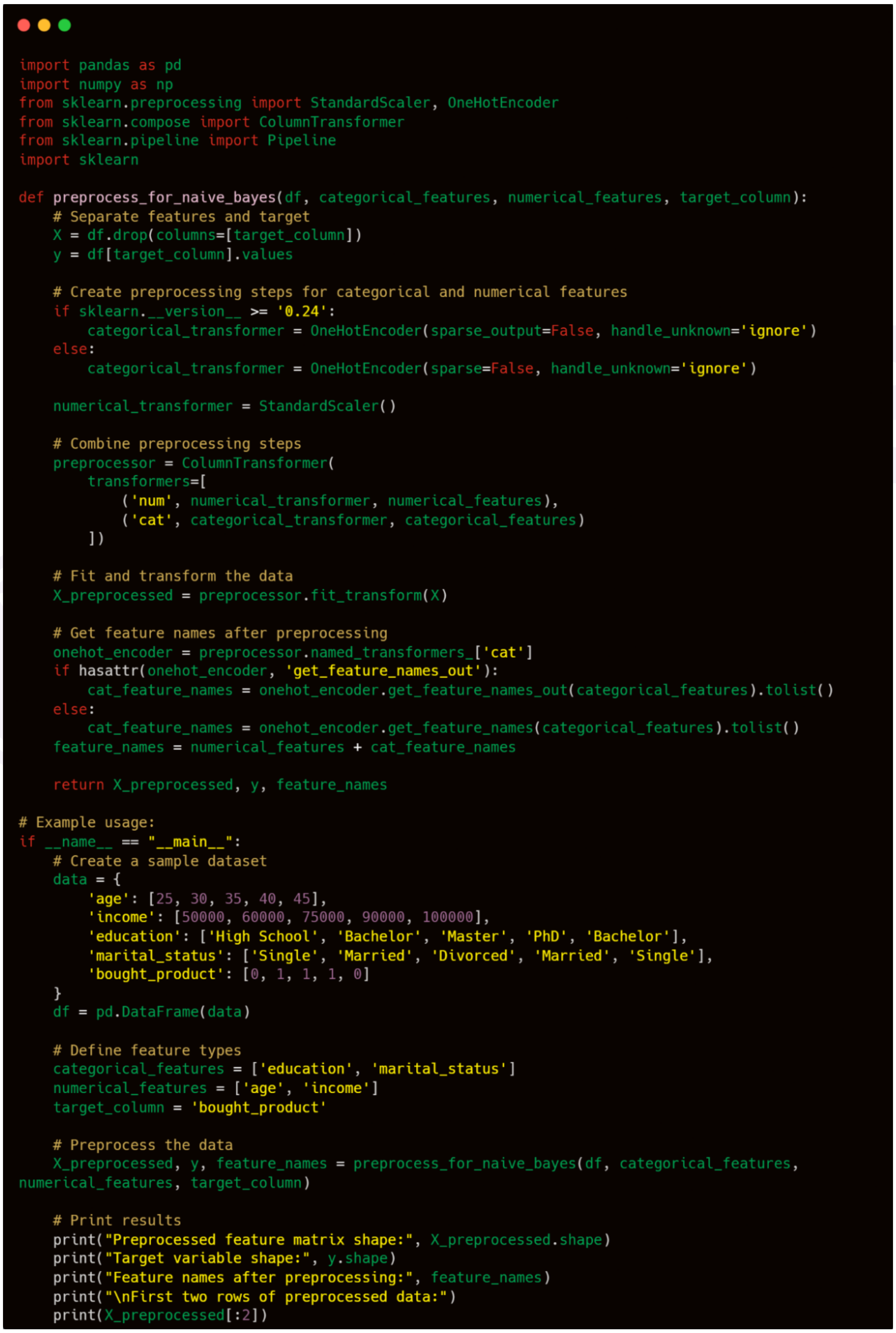
Q3: How would you modify the above Gaussian Naive Bayes classifier to handle missing data? Implement this modification.

Answer:

```python
class NaiveBayesWithMissingData(NaiveBayes):
    def _gaussian(self, x, mean, var):
        if np.isnan(x):
            return 1  # Missing data doesn't affect the probability
        return (1 / np.sqrt(2 * np.pi * var)) * np.exp(-((x - mean) ** 2) / (2 * var))

# Example Usage
X = np.array([[1, np.nan], [2, 3], [3, 4], [np.nan, 5]])
y = np.array([0, 0, 1, 1])

nb = NaiveBayesWithMissingData()
nb.fit(X, y)
print(nb.predict(np.array([[1.5, np.nan], [np.nan, 4.5]])))  # Output: Predictions with missing data
# [0 0]
```

Q4: Given a dataset with both categorical and numerical features, write a function to preprocess the data for use in a Naive Bayes classifier.

Answer:



```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
import sklearn

def preprocess_for_naive_bayes(df, categorical_features, numerical_features, target_column):
    # Separate features and target
    X = df.drop(columns=[target_column])
    y = df[target_column].values

    # Create preprocessing steps for categorical and numerical features
    if sklearn.__version__ >= '0.24':
        categorical_transformer = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
    else:
        categorical_transformer = OneHotEncoder(sparse=False, handle_unknown='ignore')

    numerical_transformer = StandardScaler()

    # Combine preprocessing steps
    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numerical_transformer, numerical_features),
            ('cat', categorical_transformer, categorical_features)
        ])

    # Fit and transform the data
    X_preprocessed = preprocessor.fit_transform(X)

    # Get feature names after preprocessing
    onehot_encoder = preprocessor.named_transformers_['cat']
    if hasattr(onehot_encoder, 'get_feature_names_out'):
        cat_feature_names = onehot_encoder.get_feature_names_out(categorical_features).tolist()
    else:
        cat_feature_names = onehot_encoder.get_feature_names(categorical_features).tolist()
    feature_names = numerical_features + cat_feature_names

    return X_preprocessed, y, feature_names

# Example usage:
if __name__ == "__main__":
    # Create a sample dataset
    data = {
        'age': [25, 30, 35, 40, 45],
        'income': [50000, 60000, 75000, 90000, 100000],
        'education': ['High School', 'Bachelor', 'Master', 'PhD', 'Bachelor'],
        'marital_status': ['Single', 'Married', 'Divorced', 'Married', 'Single'],
        'bought_product': [0, 1, 1, 1, 0]
    }
    df = pd.DataFrame(data)

    # Define feature types
    categorical_features = ['education', 'marital_status']
    numerical_features = ['age', 'income']
    target_column = 'bought_product'

    # Preprocess the data
    X_preprocessed, y, feature_names = preprocess_for_naive_bayes(df, categorical_features,
numerical_features, target_column)

    # Print results
    print("Preprocessed feature matrix shape:", X_preprocessed.shape)
    print("Target variable shape:", y.shape)
    print("Feature names after preprocessing:", feature_names)
    print("\nFirst two rows of preprocessed data:")
    print(X_preprocessed[:2])
```

Q5. Implement a Naive Bayes classifier to handle multi-label classification using the Binary Relevance approach.

Answer:

```
from sklearn.naive_bayes import GaussianNB
from sklearn.multioutput import MultiOutputClassifier

# Sample Data
X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])
y = np.array([[0, 1], [1, 0], [0, 1], [1, 0]])

# Binary Relevance with MultiOutputClassifier
model = MultiOutputClassifier(GaussianNB())
model.fit(X, y)

# Predict
y_pred = model.predict(X)
print(y_pred)  # Output: Predictions for multi-label classification
'''
[[0 1]
 [0 1]
 [1 0]
 [1 0]]
'''
```

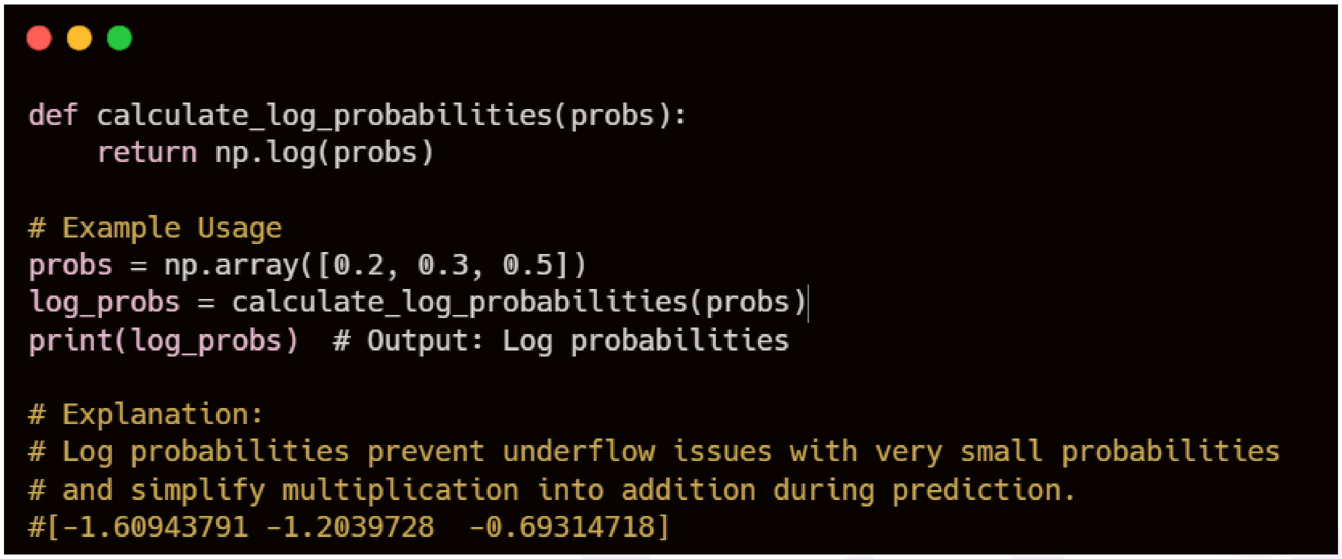
Q6. Write code to perform Laplace smoothing in a Multinomial Naive Bayes classifier.
Solution:

Answer:

```
from sklearn.naive_bayes import MultinomialNB

# Sample Data
X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])
y = np.array([0, 0, 1, 1])

# Apply Laplace Smoothing
alpha_value = 1.0
model = MultinomialNB(alpha=alpha_value)
model.fit(X, y)

# Predict
y_pred = model.predict(X)
print(y_pred)  # Output: Predictions with Laplace smoothing applied
#[0 0 1 1]
```

Q7. Write a function to calculate the log probabilities in a Naive Bayes classifier and explain why this is preferred over normal probabilities.

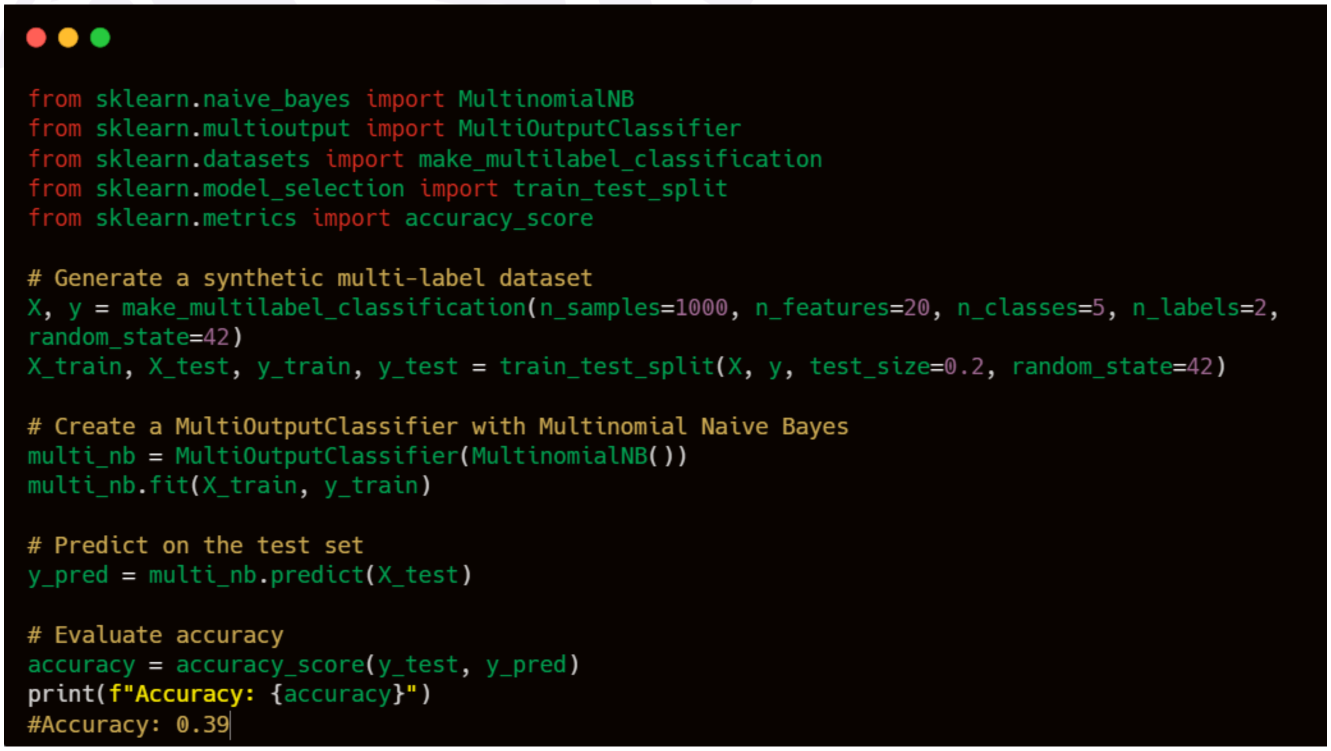
Answer:



```
def calculate_log_probabilities(probs):
    return np.log(probs)

# Example Usage
probs = np.array([0.2, 0.3, 0.5])
log_probs = calculate_log_probabilities(probs)
print(log_probs)  # Output: Log probabilities

# Explanation:
# Log probabilities prevent underflow issues with very small probabilities
# and simplify multiplication into addition during prediction.
#[-1.60943791 -1.2039728  -0.69314718]
```

Q8. Implement a Naive Bayes classifier for a multi-label classification problem using the Binary Relevance method.

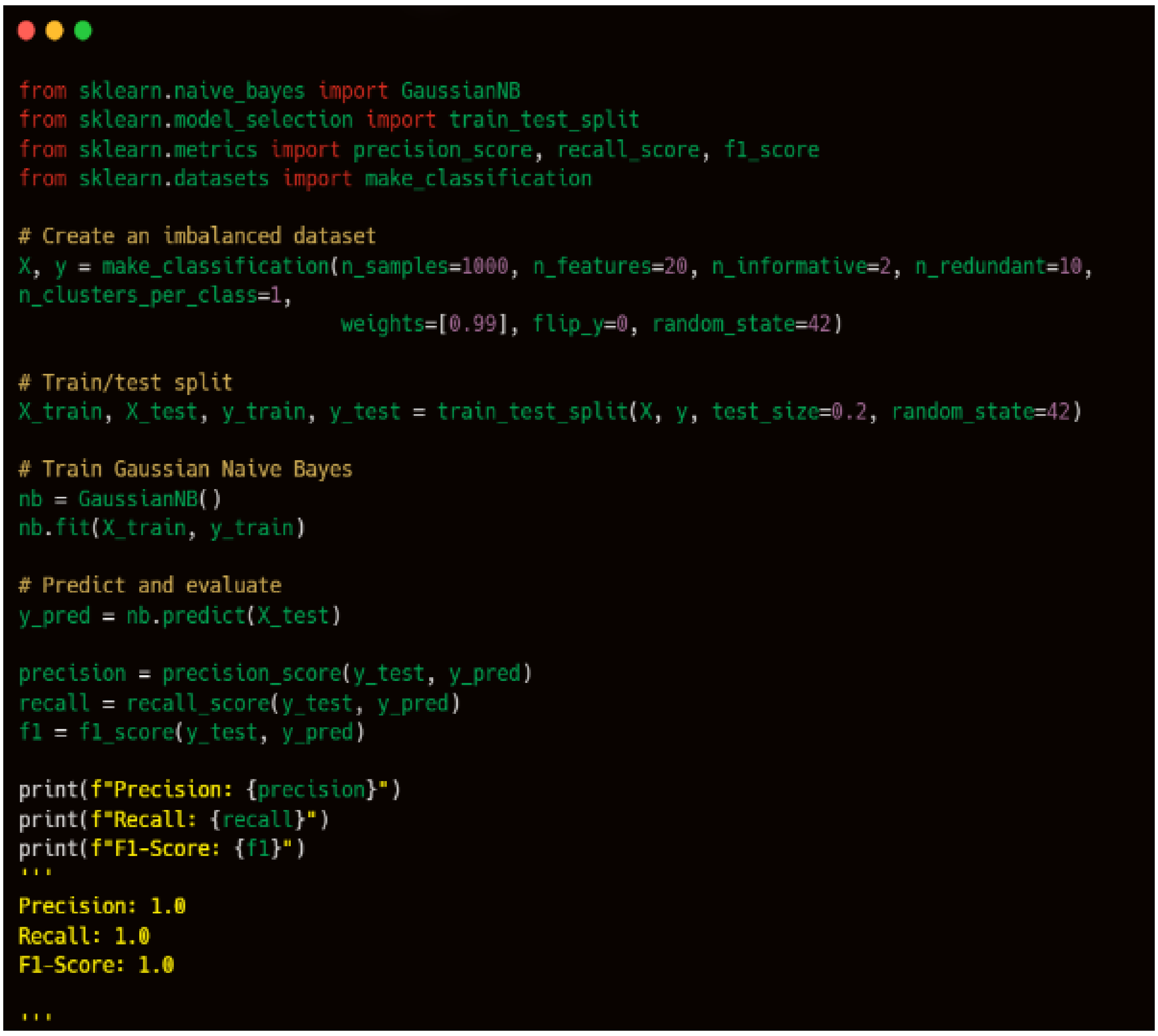
Answer:



```
from sklearn.naive_bayes import MultinomialNB
from sklearn.multioutput import MultiOutputClassifier
from sklearn.datasets import make_multilabel_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generate a synthetic multi-label dataset
X, y = make_multilabel_classification(n_samples=1000, n_features=20, n_classes=5, n_labels=2,
random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a MultiOutputClassifier with Multinomial Naive Bayes
multi_nb = MultiOutputClassifier(MultinomialNB())
multi_nb.fit(X_train, y_train)

# Predict on the test set
y_pred = multi_nb.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")
#Accuracy: 0.39
```

Q9. Write code to extend a Naive Bayes classifier for anomaly detection in a dataset with imbalanced classes, and evaluate its performance using precision, recall, and F1-score.

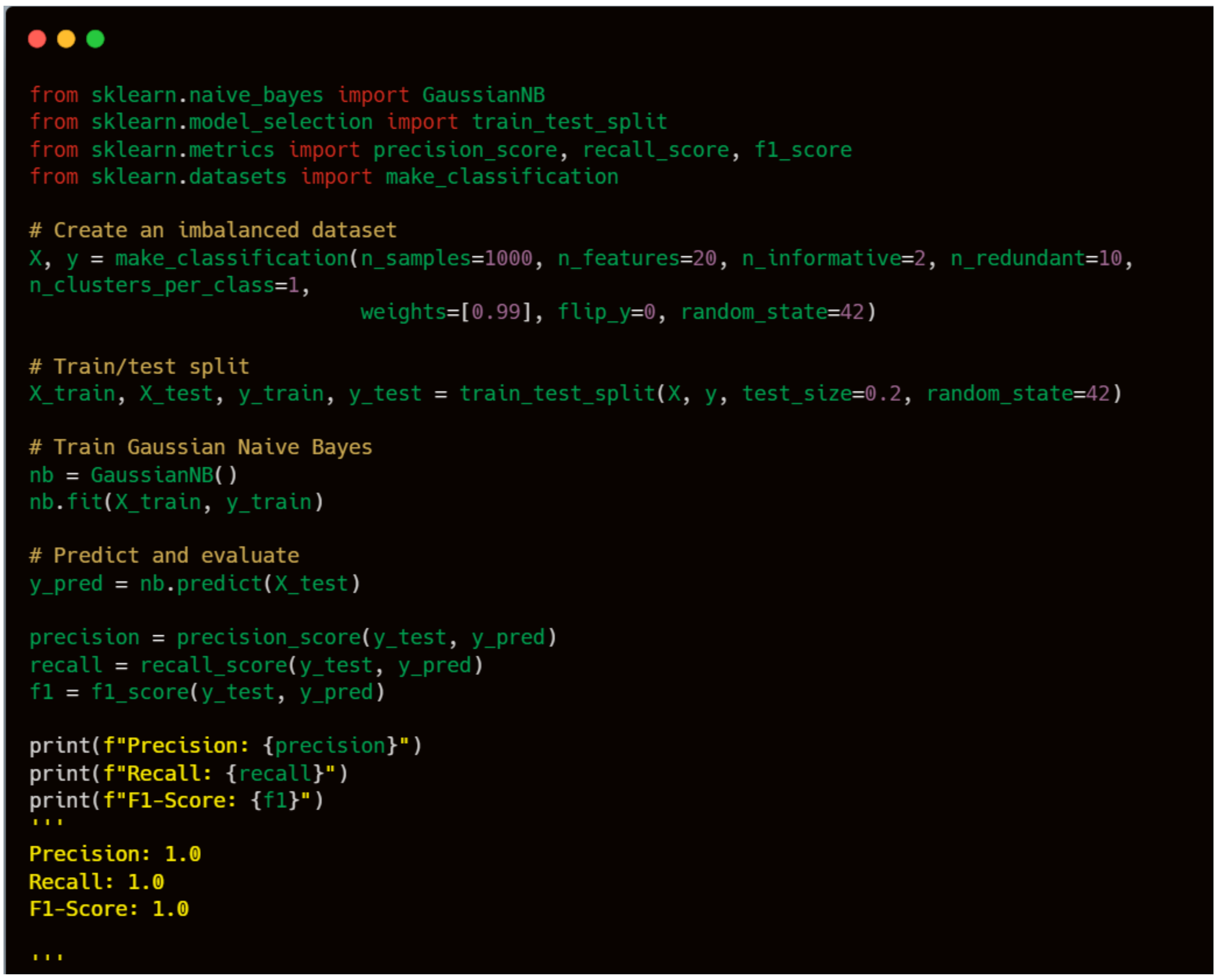
Answer:



```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.datasets import make_classification

# Create an imbalanced dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=2, n_redundant=10,
n_clusters_per_class=1,
                           weights=[0.99], flip_y=0, random_state=42)

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train Gaussian Naive Bayes
nb = GaussianNB()
nb.fit(X_train, y_train)

# Predict and evaluate
y_pred = nb.predict(X_test)

precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1-Score: {f1}")
'''
Precision: 1.0
Recall: 1.0
F1-Score: 1.0

'''
```

Q10. Implement a Gaussian Naive Bayes classifier and apply it to a continuous dataset. Compare the performance using both Gaussian and kernel density estimation (KDE) approaches for handling continuous features.

Answer:



```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.datasets import make_classification

# Create an imbalanced dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=2, n_redundant=10,
n_clusters_per_class=1,
                           weights=[0.99], flip_y=0, random_state=42)

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train Gaussian Naive Bayes
nb = GaussianNB()
nb.fit(X_train, y_train)

# Predict and evaluate
y_pred = nb.predict(X_test)

precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1-Score: {f1}")
'''
Precision: 1.0
Recall: 1.0
F1-Score: 1.0

'''
```