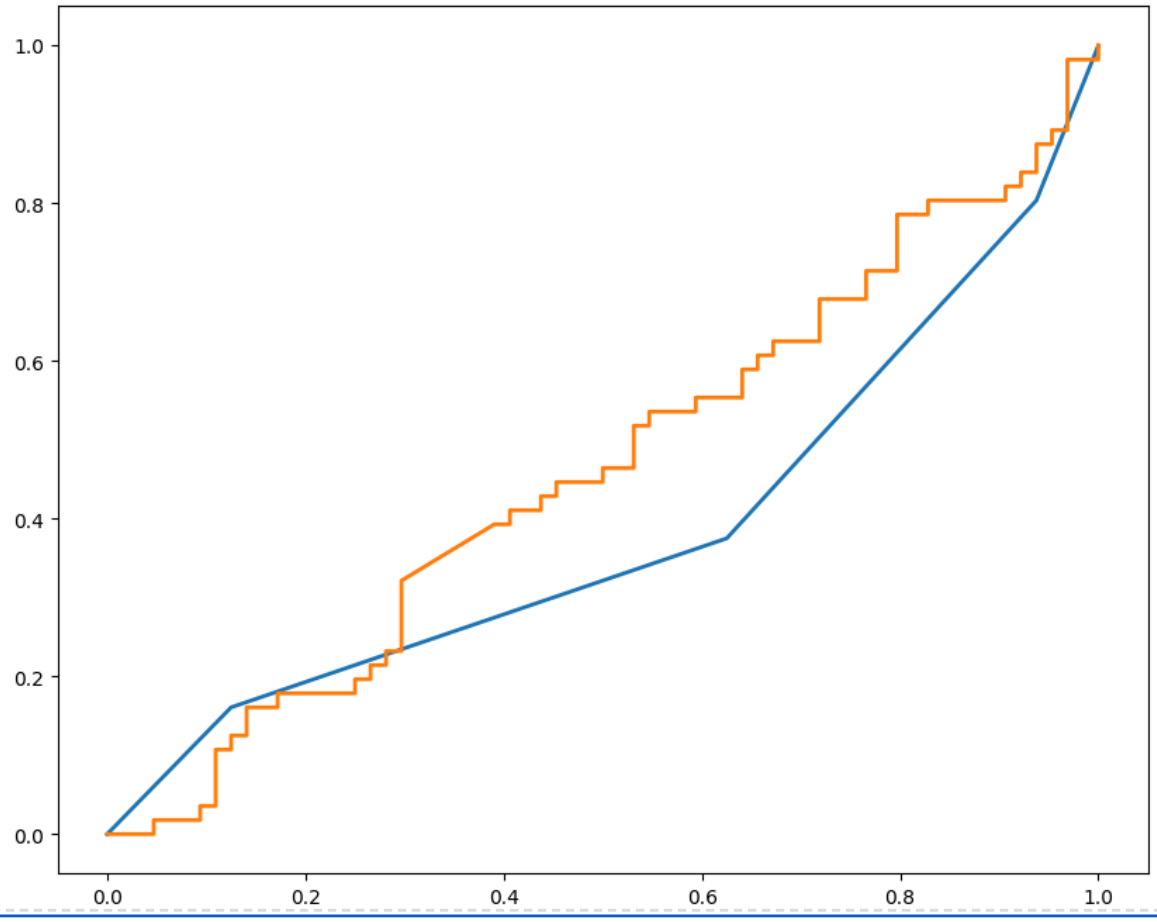
Gemini

```
# Step 1: Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, roc_curve, auc
from sklearn.preprocessing import label_binarize
from sklearn.multiclass import OneVsRestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder

# Step 2: Load your CSV dataset
# Replace 'data.csv' with your file path
df = pd.read_csv('student_dropout_dataset.csv')

# Step 3: Separate features and labels
X = df.drop('Dropout_Status', axis=1)  # Features
y = df['Dropout_Status']               # Target

# Identify categorical and numerical columns
categorical_cols = X.select_dtypes(include=['object', 'category']).columns
numerical_cols = X.select_dtypes(include=['number']).columns

# Create preprocessing pipelines for numerical and categorical features
numerical_transformer = StandardScaler()
categorical_transformer = OneHotEncoder(handle_unknown='ignore')

# Create a column transformer to apply different transformations to different columns
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

# Step 4: Split into training and testing data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Binarize labels for ROC (needed for multiclass)
# Check if the number of classes is greater than 2
if y.nunique() > 2:
    classes = sorted(y.unique())
    y_train_bin = label_binarize(y_train, classes=classes)
    y_test_bin = label_binarize(y_test, classes=classes)
    n_classes = y_test_bin.shape[1]
else:
    # For binary classification, y_test_bin is just y_test
    y_train_bin = y_train
    y_test_bin = y_test
    n_classes = 1


# Step 5: Initialize models with preprocessing pipelines
models = {
    "KNN": Pipeline(steps=[('preprocessor', preprocessor),
                           ('classifier', KNeighborsClassifier(n_neighbors=3))]),
    "SVM": Pipeline(steps=[('preprocessor', preprocessor),
                           ('classifier', SVC(kernel='linear', probability=True, random_state=42))]),
    "Naive Bayes": Pipeline(steps=[('preprocessor', preprocessor),
                                   ('classifier', GaussianNB())]),
    "Gradient Boosting": Pipeline(steps=[('preprocessor', preprocessor),
                                         ('classifier', GradientBoostingClassifier(n_estimators=100, learning_rate=
}

# Step 6: Train, predict, and evaluate accuracy + ROC
accuracies = {}
plt.figure(figsize=(10, 8))

for name, model in models.items():
    # Train
    model.fit(X_train, y_train)
    if name == "Naive Bayes":
```

```
            # Convert sparse matrix to dense array for GaussianNB
            X_train_processed = model.named_steps['preprocessor'].transform(X_train).toarray()
            model.named_steps['classifier'].fit(X_train_processed, y_train)
        else:
            model.fit(X_train, y_train)

        y_pred = model.predict(X_test)

        # Accuracy
        acc = accuracy_score(y_test, y_pred)
        accuracies[name] = acc
        print(f"{name} Accuracy: {acc:.2f}")

        # ROC curve
        if hasattr(model, "predict_proba"):
            y_score = model.predict_proba(X_test)
        else:
            # For SVM with decision_function in multiclass, we need to adjust
            if n_classes > 2:
                 y_score = model.decision_function(X_test)
                 # Binarize the true labels for ROC calculation in multiclass
                 y_test_bin_roc = label_binarize(y_test, classes=model.classes_)
            else:
                 y_score = model.decision_function(X_test)
                 y_test_bin_roc = y_test_bin # Use the original binary labels

        # Define y_test_bin_roc before using it
        if n_classes <= 2:
            y_test_bin_roc = y_test_bin
        else:
            y_test_bin_roc = label_binarize(y_test, classes=model.classes_)


        # Handle binary vs multiclass ROC
        if n_classes <= 2: # Binary or single class (will be treated as binary)
            # Ensure y_score is 1D for binary roc_curve
            if y_score.ndim > 1:
              y_score = y_score[:, 1]

            fpr, tpr, _ = roc_curve(y_test_bin_roc, y_score)
            roc_auc = auc(fpr, tpr)
            plt.plot(fpr, tpr, lw=2, label=f'{name} (AUC = {roc_auc:.2f})')
        else: # Multiclass
            # Compute ROC curve and ROC area for each class
            fpr = dict()
            tpr = dict()
            roc_auc = dict()
            for i in range(n_classes):
                fpr[i], tpr[i], _ = roc_curve(y_test_bin_roc[:, i], y_score[:, i])
                roc_auc[i] = auc(fpr[i], tpr[i])

            # Compute micro-average ROC curve and ROC area
            fpr["micro"], tpr["micro"], _ = roc_curve(y_test_bin_roc.ravel(), y_score.ravel())
            roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

            plt.plot(fpr["micro"], tpr["micro"], lw=2,
                     label=f'{name} (Micro-avg AUC = {roc_auc["micro"]:.2f})')


    # Step 7: Plot Accuracy comparison
    plt.figure(figsize=(8,5))
    plt.bar(accuracies.keys(), accuracies.values(), color=['skyblue', 'orange', 'green', 'red'])
    plt.ylim(0, 1)
    plt.title("Model Accuracy Comparison")
    plt.ylabel("Accuracy")
    plt.show()

    # Step 8: Plot ROC Curve comparison
    plt.plot([0, 1], [0, 1], 'k--', lw=2)  # Random guess line
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve Comparison')
    plt.legend(loc="lower right")
    plt.show()
```

```
KNN Accuracy: 0.38
SVM Accuracy: 0.50
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-3379193309.py in <cell line: 0>()
     73 for name, model in models.items():
     74     # Train
---> 75     model.fit(X_train, y_train)
     76     y_pred = model.predict(X_test)
     77

                                 8 frames
/usr/local/lib/python3.12/dist-packages/sklearn/utils/validation.py in _ensure_sparse_format(sparse_container,
accept_sparse, dtype, copy, ensure_all_finite, accept_large_sparse, estimator_name, input_name)
    611     if accept_sparse is False:
    612         padded_input = " for " + input_name if input_name else ""
--> 613         raise TypeError(
    614             f"Sparse data was passed{padded_input}, but dense data is required. "
    615             "Use '.toarray()' to convert to a dense numpy array."

TypeError: Sparse data was passed for X, but dense data is required. Use '.toarray()' to convert to a dense numpy
array.
```



Next steps: Explain error