```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, f1_score, roc_auc_score
import xgboost as xgb
import matplotlib.pyplot as plt
import seaborn as sns
from collections import defaultdict

# Enhanced Data Preprocessing for Mixed Data Types
def preprocess_student_data(df):
    """
    Preprocessing specifically designed for student depression dataset
    """
    df_processed = df.copy()

    # Separate numerical and categorical columns
    numerical_cols = ['Age', 'Academic Pressure', 'Work Pressure', 'CGPA',
                     'Study Satisfaction', 'Job Satisfaction', 'Work/Study Hours',
                     'Financial Stress']

    categorical_cols = ['Gender', 'City', 'Profession', 'Sleep Duration',
                       'Dietary Habits', 'Degree', 'Have you ever had suicidal thoughts ?',
                       'Family History of Mental Illness']

    # Handle numerical features
    imputer = SimpleImputer(strategy='median')
    df_processed[numerical_cols] = imputer.fit_transform(df_processed[numerical_cols])

    # Handle categorical features with Label Encoding
    label_encoders = {}
    for col in categorical_cols:
        if col in df_processed.columns:
            le = LabelEncoder()
            df_processed[col] = le.fit_transform(df_processed[col].astype(str))
            label_encoders[col] = le

    # Normalize all features
    scaler = MinMaxScaler()
    feature_columns = [col for col in df_processed.columns if col not in ['id', 'Depression']]
    df_processed[feature_columns] = scaler.fit_transform(df_processed[feature_columns])

    return df_processed, label_encoders

# Enhanced Risk Assessment for Depression
def assess_depression_risk(probability):
    """
    Categorize depression risk based on probability
    """
    if probability < 0.25:
        return "Low Risk"
    elif probability < 0.50:
        return "Moderate Risk"
    elif probability < 0.75:
        return "High Risk"
    else:
        return "Critical Risk"

# Enhanced Coati Optimization for Depression Features
def enhanced_coati_optimization_depression(X, y, num_features=10, num_agents=15, max_iter=25):
    """
    Adapted ECO for depression prediction with enhanced parameters
    """
    n_features = X.shape[1]
    agents = np.random.randint(0, 2, (num_agents, n_features))

    def fitness(agent):
        selected_features = [index for index in range(len(agent)) if agent[index] == 1]
        if len(selected_features)
            return 0

        X_selected = X[:, selected_features]
        X_train, X_test, y_train, y_test = train_test_split(X_selected, y, test_size=0.3, random_state=42)
```

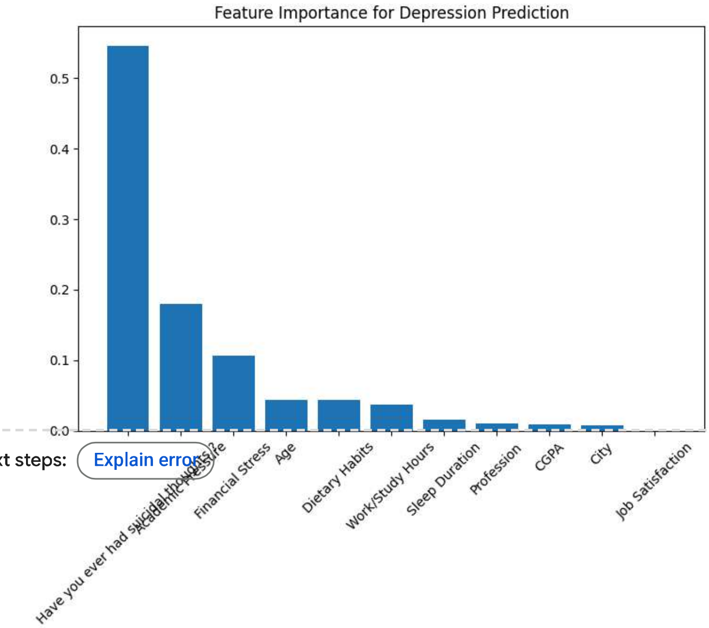
```python
        clf = RandomForestClassifier(n_estimators=20, random_state=42)
        clf.fit(X_train, y_train)
        predictions = clf.predict(X_test)

        # Use F1 score for better performance with potentially imbalanced data
        return f1_score(y_test, predictions)

    best_agent = agents[0]
    best_score = fitness(best_agent)

    for iteration in range(max_iter):
        for i in range(num_agents):
            current_agent = agents[i].copy()

            # Opposition-based learning
            opposite_agent = 1 - current_agent
            if fitness(opposite_agent) > fitness(current_agent):
                current_agent = opposite_agent

            # Enhanced mutation with adaptive rate
            mutation_rate = 0.1 + (0.3 * (max_iter - iteration) / max_iter)
            for j in range(n_features):
                if np.random.random() < mutation_rate:
                    current_agent[j] = 1 - current_agent[j]

            current_score = fitness(current_agent)
            if current_score > best_score:
                best_agent = current_agent.copy()
                best_score = current_score

            agents[i] = current_agent.copy()

    selected_indices = [index for index in range(len(best_agent)) if best_agent[index] == 1]
    if len(selected_indices) > num_features:
        selected_indices = selected_indices[:num_features]

    return selected_indices

# Student Depression Risk Classifier
class StudentDepressionClassifier:
    def __init__(self):
        self.model = None
        self.selected_features = []
        self.feature_names = []
        self.label_encoders = {}

    def train(self, X, y, feature_names):
        """
        Train the depression risk prediction model
        """
        self.feature_names = feature_names

        print("Training Student Depression Risk Classifier...")
        print(f"Dataset shape: {X.shape}")
        print(f"Target distribution: {np.bincount(y)}")

        # Feature selection using Enhanced Coati Optimization
        selected_indices = enhanced_coati_optimization_depression(X.values, y.values, num_features=12)
        X_selected = X.iloc[:, selected_indices]
        self.selected_features = X_selected.columns.tolist()

        print(f"Selected features: {', '.join(self.selected_features)}")

        # Train model with White Shark optimization
        model, metrics = self._train_depression_model(X_selected, y)
        self.model = model

        print("Performance Metrics:")
        for metric, value in metrics.items():
            print(f"{metric}: {value:.4f}")

        return metrics

    def _train_depression_model(self, X, y):
        """
        Train XGBoost model with White Shark optimization
        """
```

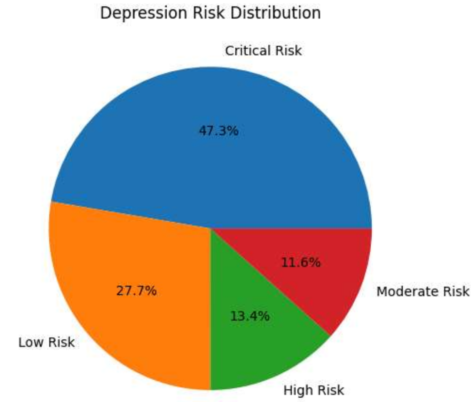
```python
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

        # Enhanced White Shark optimization for depression
        best_params = self._white_shark_optimization_depression(X_train, X_test, y_train, y_test)

        # Train final model
        model = xgb.XGBClassifier(**best_params, random_state=42)
        model.fit(X_train, y_train)

        # Calculate comprehensive metrics
        y_pred = model.predict(X_test)
        y_prob = model.predict_proba(X_test)[:, 1]

        metrics = {
            'Accuracy': accuracy_score(y_test, y_pred),
            'Precision': precision_score(y_test, y_pred),
            'F1 Score': f1_score(y_test, y_pred),
            'AUC-ROC': roc_auc_score(y_test, y_prob)
        }

        return model, metrics

    def _white_shark_optimization_depression(self, X_train, X_test, y_train, y_test):
        """
        Enhanced White Shark optimization for depression prediction
        """
        param_space = {
            'max_depth': [3, 4, 5, 6, 7],
            'learning_rate': [0.01, 0.05, 0.1, 0.15, 0.2],
            'n_estimators': [50, 100, 150, 200],
            'subsample': [0.6, 0.7, 0.8, 0.9, 1.0],
            'colsample_bytree': [0.6, 0.7, 0.8, 0.9, 1.0],
            'reg_alpha': [0, 0.1, 0.5, 1.0],
            'reg_lambda': [0, 0.1, 0.5, 1.0]
        }

        best_params = {}
        best_score = 0

        # Enhanced search with more iterations
        for _ in range(30):
            params = {
                'max_depth': np.random.choice(param_space['max_depth']),
                'learning_rate': np.random.choice(param_space['learning_rate']),
                'n_estimators': np.random.choice(param_space['n_estimators']),
                'subsample': np.random.choice(param_space['subsample']),
                'colsample_bytree': np.random.choice(param_space['colsample_bytree']),
                'reg_alpha': np.random.choice(param_space['reg_alpha']),
                'reg_lambda': np.random.choice(param_space['reg_lambda'])
            }

            model = xgb.XGBClassifier(**params, random_state=42)
            model.fit(X_train, y_train)

            # Use F1 score for optimization
            y_pred = model.predict(X_test)
            score = f1_score(y_test, y_pred)

            if score > best_score:
                best_score = score
                best_params = params

        return best_params

    def predict_risk(self, X):
        """
        Predict depression risk for new students
        """
        if self.model is None:
            raise ValueError("Model not trained yet!")

        X_selected = X[self.selected_features]
        probabilities = self.model.predict_proba(X_selected)[:, 1]
        risk_levels = [assess_depression_risk(p) for p in probabilities]

        return {
            'probabilities': probabilities,
```

```
            'risk_levels': risk_levels,
            'predictions': self.model.predict(X_selected)
        }

    def visualize_results(self, X, y):
        """
        Create comprehensive visualizations
        """
        X_selected = X[self.selected_features]

        fig, axes = plt.subplots(2, 2, figsize=(15, 12))

        # Feature importance
        feature_importance = self.model.feature_importances_
        sorted_idx = np.argsort(feature_importance)[::-1]

        axes[0, 0].bar(range(len(feature_importance)), feature_importance[sorted_idx])
        axes[0, 0].set_xticks(range(len(feature_importance)))
        axes[0, 0].set_xticklabels([self.selected_features[i] for i in sorted_idx], rotation=45)
        axes[0, 0].set_title('Feature Importance for Depression Prediction')

        # Risk distribution
        results = self.predict_risk(X)
        risk_counts = pd.Series(results['risk_levels']).value_counts()
        axes[0, 1].pie(risk_counts.values, labels=risk_counts.index, autopct='%1.1f%%')
        axes[0, 1].set_title('Depression Risk Distribution')

        # Probability distribution
        axes[1, 0].hist(results['probabilities'], bins=20, alpha=0.7, edgecolor='black')
        axes[1, 0].set_xlabel('Depression Probability')
        axes[1, 0].set_ylabel('Number of Students')
        axes[1, 0].set_title('Distribution of Depression Probabilities')

        # Actual vs Predicted
        axes[1, 1].scatter(y, results['predictions'], alpha=0.6)
        axes[1, 1].set_xlabel('Actual Depression Status')
        axes[1, 1].set_ylabel('Predicted Depression Status')
        axes[1, 1].set_title('Actual vs Predicted Depression')

        plt.tight_layout()
        plt.show()

# Main execution function
def run_student_depression_analysis(data_path):
    """
    Main pipeline for student depression analysis
    """
    try:
        # Load dataset
        df = pd.read_csv(data_path)
        print(f"Loaded dataset with {len(df)} records and {len(df.columns)} features")

        # Preprocess data
        df_processed, label_encoders = preprocess_student_data(df)

        # Separate features and target
        X = df_processed.drop(['id', 'Depression'], axis=1)
        y = df_processed['Depression']

        # Initialize and train classifier
        classifier = StudentDepressionClassifier()
        classifier.label_encoders = label_encoders

        metrics = classifier.train(X, y, X.columns.tolist())

        # Visualize results
        classifier.visualize_results(X, y)

        return classifier, metrics

    except Exception as e:
        print(f"Error in analysis: {str(e)}")
        return None, None

# Example usage
if __name__ == "__main__":
    data_path = "/content/Student Depression Dataset.csv"
```

```python
classifier, metrics = run_student_depression_analysis(data_path)

if classifier:
    # Example prediction on new data
    sample_data = pd.DataFrame({
        'Gender': ['Male'],
        'Age': [22],
        'Academic Pressure': [4],
        'CGPA': [7.5],
        'Sleep Duration': ['5-6 hours'],
        'Family History of Mental Illness': ['No']
        # ... add other features
    })

    # Preprocess and predict
    sample_processed, _ = preprocess_student_data(sample_data)
    risks = classifier.predict_risk(sample_processed)

    print(f"Sample prediction: {risks['risk_levels'][0]} (Probability: {risks['probabilities'][0]:.2f})")
```

```
Loaded dataset with 27901 records and 18 features
Training Student Depression Risk Classifier...
Dataset shape: (27901, 16)
Target distribution: [11565 16336]
Selected features: Age, City, Profession, Academic Pressure, CGPA, Job Satisfaction, Sleep Duration, Dietary Habits, Have you ever had s
Performance Metrics:
Accuracy: 0.8377
Precision: 0.8474
F1 Score: 0.8626
AUC-ROC: 0.9125
```





Next steps: Explain error