Mental Health Risk Prediction System

Documentation generated on 2024-12-23 13:18:26

System Overview

This system predicts mental health risks using machine learning. It processes patient data through various stages including data preprocessing, model training, and prediction generation.

The system uses XGBoost classifier with optimized hyperparameters to achieve the best possible prediction accuracy.

Code: Main Application

Source code for main.py:

```
from src.data_processing import load_data, preprocess_data
from src.visualization import create_visualizations
from src.model import prepare_data, train_model, evaluate_model, save_model

def main():
    # Load and process data
    data_path = "depression_data.csv"  # Specify the path to your data file
    processed_data = load_data(data_path)

     # Create visualizations
    print("\nCreating visualizations...")
    create_visualizations(processed_data)

    # Print data info for debugging
    print("\nDataset Info:")
    print(processed_data.info())
    print("\nColumn names:")
    print(processed_data.columns.tolist())

    # Prepare and train model
    X_train, X_test, y_train, y_test = prepare_data(processed_data)

    # Train and evaluate model
    model = train_model(X_train, y_train)
    accuracy = evaluate_model(model, X_test, y_test)

    # Save the trained model
    model_path = save_model(model, accuracy)

if __name__ == "__main__":
    main()
```

Code: Model Training and Evaluation

Source code for src/model.py:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from xgboost import XGBClassifier
import joblib
import os
from datetime import datetime
import pandas as pd
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report
import xgboost as xgb

def prepare_data(data):
    """Prepare data for model training"""
    # Create a copy to avoid modifying the original data
    data = data.copy()

    # Drop the Name column if it exists
    if 'Name' in data.columns:
        data = data.drop('Name', axis=1)

    # Define categorical columns (all non-numeric columns except the target)
    categorical_columns = [
        'Marital Status',
        'Education Level',
        'Smoking Status',
        'Physical Activity Level',
        'Employment Status',
        'Alcohol Consumption',
        'Dietary Habits',
        'Sleep Patterns',
        'History of Substance Abuse',
        'Family History of Depression',
        'Chronic Medical Conditions'
    ]

    # Define numeric columns
    numeric_columns = ['Age', 'Number of Children', 'Income']

    # Convert categorical columns to numeric using label encoding
    for column in categorical_columns:
        if column in data.columns:
```

```
            data[column] = pd.Categorical(data[column]).codes

    # Ensure numeric columns are float type
    for column in numeric_columns:
        if column in data.columns:
            data[column] = pd.to_numeric(data[column], errors='coerce')
            # Fill any missing values with mean
            data[column] = data[column].fillna(data[column].mean())

    # Check if the target column exists with exact name
    target_column = 'History of Mental Illness'

    # Try different possible variations of the column name
    if target_column not in data.columns:
        possible_names = [
            'History_of_Mental_Illness',
            'Mental_Illness_History',
            'Mental Illness History',
            'mental_illness'
        ]
        for name in possible_names:
            if name in data.columns:
                target_column = name
                break
        else:
            print("Available columns:", data.columns.tolist())
                raise KeyError(f"Target column '{target_column}' not found in dataset.
Please check column names.")

    # Convert target to numeric if it's not already
    data[target_column] = pd.Categorical(data[target_column]).codes

    # Split features and target
    X = data.drop([target_column], axis=1)
    y = data[target_column]

    # Scale the features
    scaler = StandardScaler()
    X = scaler.fit_transform(X)

    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    return X_train, X_test, y_train, y_test

def train_model(X_train, y_train):
    """Train the model with hyperparameter tuning"""
```

```
    # Define the parameter grid for GridSearchCV
    param_grid = {
        'max_depth': [3, 4, 5, 6],
        'learning_rate': [0.01, 0.05, 0.1],
        'n_estimators': [100] ,#[100, 200, 300],
        'min_child_weight': [1, 3, 5],
        'gamma': [0, 0.1, 0.2],
        'subsample': [0.8, 0.9, 1.0],
        'colsample_bytree': [0.8, 0.9, 1.0],
        'scale_pos_weight': [1, 3, 5]  # Help with imbalanced classes
    }

    # Initialize the model
    xgb_model = xgb.XGBClassifier(
        objective='binary:logistic',
        random_state=42,
        use_label_encoder=False,
        eval_metric='logloss'
    )

    # Perform grid search with cross-validation
    grid_search = GridSearchCV(
        estimator=xgb_model,
        param_grid=param_grid,
        cv=1,
        scoring='accuracy',
        n_jobs=-1,
        verbose=2
    )

    # Fit the model
    print("Training model with hyperparameter tuning...")
    grid_search.fit(X_train, y_train)

    # Print best parameters and score
    print("\nBest parameters found:")
    print(grid_search.best_params_)
    print(f"Best cross-validation accuracy: {grid_search.best_score_:.4f}")

    # Return the best model
    return grid_search.best_estimator_

def evaluate_model(model, X_test, y_test):
    """Evaluate the model and print detailed metrics"""
    # Make predictions
    y_pred = model.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
```

```
    # Print detailed classification report
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    # Get feature importance
    feature_importance = model.feature_importances_
    print("\nFeature Importance:")
    for i, importance in enumerate(feature_importance):
        print(f"Feature {i}: {importance:.4f}")

    return accuracy

def save_model(model, accuracy, model_dir='models'):
    """Save the trained model with timestamp and accuracy"""
    # Create models directory if it doesn't exist
    if not os.path.exists(model_dir):
        os.makedirs(model_dir)

    # Create filename with timestamp and accuracy
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    accuracy_str = f"{accuracy:.4f}".replace(".", "")
    filename = f"model_{timestamp}_acc_{accuracy_str}.joblib"
    filepath = os.path.join(model_dir, filename)

    # Save the model
    joblib.dump(model, filepath)
    print(f"Model saved to: {filepath}")
    return filepath

def load_model(filepath):
    """Load a saved model"""
    if not os.path.exists(filepath):
        raise FileNotFoundError(f"Model file not found: {filepath}")

    model = joblib.load(filepath)
    print(f"Model loaded from: {filepath}")
    return model
```

Code: Data Processing

Source code for src/data_processing.py:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, StandardScaler

def load_data(file_path):
    """Load and perform initial data analysis"""
    data = pd.read_csv(file_path)
    print(data.head())
    print(data.describe())
    print(data.info())
    return data

def preprocess_data(data, for_training=True):
    """
    Preprocess the data for model training or prediction
    """
    # Create a copy of the data to avoid modifying the original
    processed_data = data.copy()

    # Define the feature columns in the correct order
    feature_columns = ['Age', 'Marital Status', 'Education Level', 'Number of Children',
                       'Smoking Status', 'Physical Activity Level', 'Employment Status',
                              'Income', 'Alcohol Consumption', 'Dietary Habits', 'Sleep
Patterns',
                       'History of Substance Abuse', 'Family History of Depression',
                       'Chronic Medical Conditions']

    # Basic preprocessing steps
    # Convert categorical variables to numeric
    categorical_columns = ['Marital Status', 'Education Level', 'Employment Status',
                                   'Smoking Status', 'Physical Activity Level', 'Alcohol
Consumption',
                                'Dietary Habits', 'Sleep Patterns', 'History of Substance
Abuse',
                           'Family History of Depression', 'Chronic Medical Conditions']

    for column in categorical_columns:
        if column in processed_data.columns:
            # Convert to category and then to numeric codes
            processed_data[column] = processed_data[column].astype('category').cat.codes

    # Ensure all numeric columns are float
    numeric_columns = ['Age', 'Number of Children', 'Income']
```

```
for column in numeric_columns:
    if column in processed_data.columns:
        processed_data[column] = processed_data[column].astype(float)

# Select only the feature columns in the correct order
processed_data = processed_data[feature_columns]

# Rename columns to match model expectations
processed_data.columns = [f'f{i}' for i in range(len(feature_columns))]

return processed_data
```

Code: Data Visualization

Source code for src/visualization.py:

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import os

def plot_histograms(data):
    """
    Plot histograms for all numeric columns in the dataset
    """
    # Create a copy and drop the Name column if it exists
    df = data.copy()
    if 'Name' in df.columns:
        df = df.drop('Name', axis=1)

    # Convert categorical columns to numeric
    categorical_columns = [
        'Marital Status',
        'Education Level',
        'Smoking Status',
        'Physical Activity Level',
        'Employment Status',
        'Alcohol Consumption',
        'Dietary Habits',
        'Sleep Patterns',
        'History of Substance Abuse',
        'Family History of Depression',
        'Chronic Medical Conditions',
        'History of Mental Illness'
    ]

    for col in categorical_columns:
        if col in df.columns:
            df[col] = pd.Categorical(df[col]).codes

    # Create subplots for each column
    n_cols = 3
    n_rows = (len(df.columns) + n_cols - 1) // n_cols
    fig, axes = plt.subplots(n_rows, n_cols, figsize=(15, 4*n_rows))
    axes = axes.flatten()

    # Plot histogram for each column
    for idx, col in enumerate(df.columns):
```

```
        sns.histplot(data=df, x=col, ax=axes[idx])
        axes[idx].set_title(f'Distribution of {col}')
        axes[idx].tick_params(axis='x', rotation=45)

    # Remove empty subplots
    for idx in range(len(df.columns), len(axes)):
        fig.delaxes(axes[idx])

    plt.tight_layout()
    plt.savefig('visualizations/feature_distributions.png')
    plt.close()

def plot_correlation_matrix(data):
    """
    Plot correlation matrix for all numeric columns
    """
    # Create a copy and drop the Name column if it exists
    df = data.copy()
    if 'Name' in df.columns:
        df = df.drop('Name', axis=1)

    # Convert categorical columns to numeric
    categorical_columns = [
        'Marital Status',
        'Education Level',
        'Smoking Status',
        'Physical Activity Level',
        'Employment Status',
        'Alcohol Consumption',
        'Dietary Habits',
        'Sleep Patterns',
        'History of Substance Abuse',
        'Family History of Depression',
        'Chronic Medical Conditions',
        'History of Mental Illness'
    ]

    for col in categorical_columns:
        if col in df.columns:
            df[col] = pd.Categorical(df[col]).codes

    # Calculate correlation matrix
    corr_matrix = df.corr()

    # Create correlation matrix plot
    plt.figure(figsize=(12, 10))
    sns.heatmap(corr_matrix,
                annot=True,
                cmap='coolwarm',
```

```
                center=0,
                fmt='.2f',
                square=True)
    plt.title('Feature Correlation Matrix')
    plt.xticks(rotation=45, ha='right')
    plt.yticks(rotation=0)
    plt.tight_layout()
    plt.savefig('visualizations/correlation_matrix.png')
    plt.close()

def create_visualizations(data):
    """
    Create all visualizations
    """
    # Create visualizations directory if it doesn't exist
    if not os.path.exists('visualizations'):
        os.makedirs('visualizations')

    # Create visualizations
    plot_histograms(data)
    plot_correlation_matrix(data)
    print("Visualizations have been saved to the 'visualizations' directory.")
```

Code: Individual Prediction

Source code for predict_individual.py:

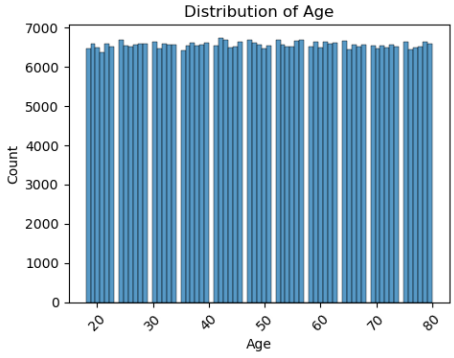
```
import pandas as pd
from src.model import load_model
from src.data_processing import preprocess_data
import os

def get_latest_model(model_dir='models'):
    """Get the most recently saved model"""
    if not os.path.exists(model_dir):
        raise FileNotFoundError(f"Model directory not found: {model_dir}")

    model_files = [f for f in os.listdir(model_dir) if f.endswith('.joblib')]
    if not model_files:
        raise FileNotFoundError("No model files found in models directory")

                        latest_model      =      max(model_files,      key=lambda      x:
os.path.getctime(os.path.join(model_dir, x)))
    return os.path.join(model_dir, latest_model)

def predict_for_individual(name, data_path='depression_data.csv'):
    try:
        # Load the model using get_latest_model()
        model_path = get_latest_model()
        model = load_model(model_path)
        print(f"Model loaded from: {model_path}")

        # Load the full dataset
        df = pd.read_csv(data_path)

        # Find all people with matching name
        people_data = df[df['Name'].str.lower() == name.lower()]

        if len(people_data) == 0:
            return {'error': f"No data found for {name}"}

        results = []
        # Process each person's data
        for idx, person_data in people_data.iterrows():
            # Convert single row to DataFrame
            person_df = pd.DataFrame([person_data])

            # Preprocess the data
            processed_data = preprocess_data(person_df, for_training=False)
```

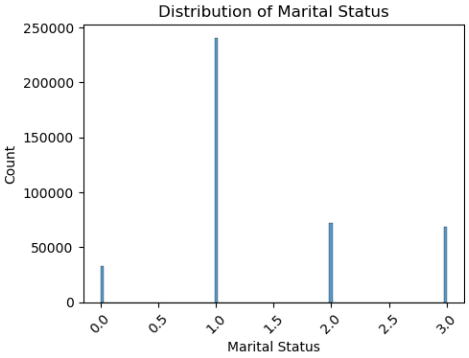
```
            # Make prediction
            prediction = model.predict(processed_data)

            # Store result
            results.append({
                'name': person_data['Name'],
                'prediction': "At risk" if prediction[0] == 1 else "Not at risk",
                'confidence': None,
                'details': person_data.to_dict()
            })

        return {
            'multiple': len(results) > 1,
            'results': results
        }

    except Exception as e:
        print(f"\nError: Error during prediction: {str(e)}")
        return {'error': str(e)}

def main():
    """Interactive command-line interface for making predictions"""
    while True:
        print("\n=== Mental Health Risk Prediction ===")
        name = input("\nEnter name to search (or 'quit' to exit): ")

        if name.lower() == 'quit':
            break

        result = predict_for_individual(name)

        if 'error' in result:
            print(f"\nError: {result['error']}")
        else:
            if result['multiple']:
                print(f"\nFound {len(result['results'])} entries for {name}:")

            for idx, person_result in enumerate(result['results'], 1):
                if result['multiple']:
                    print(f"\n--- Entry {idx} ---")
                print("\nPrediction Results:")
                print(f"Name: {person_result['name']}")
                print(f"Prediction: {person_result['prediction']}")
                if person_result['confidence']:
                    print(f"Confidence: {person_result['confidence']}")
                print("\nIndividual Details:")
                for key, value in person_result['details'].items():
                    if key != 'Name':  # Skip printing name again
```

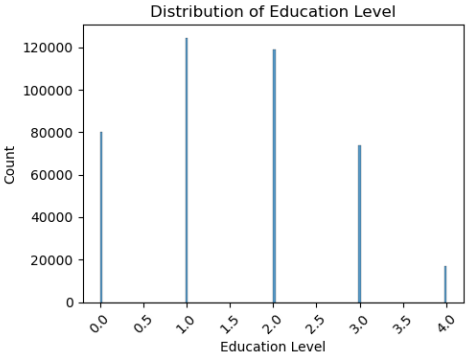
```
                        print(f"{key}: {value}")

if __name__ == "__main__":
    main()
```

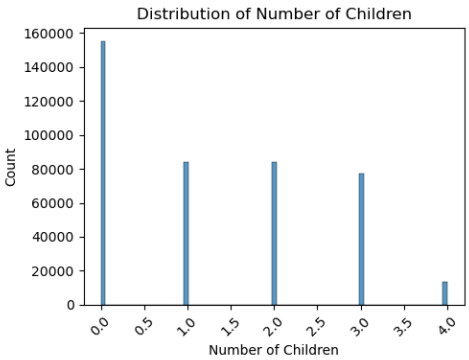
Data Visualizations

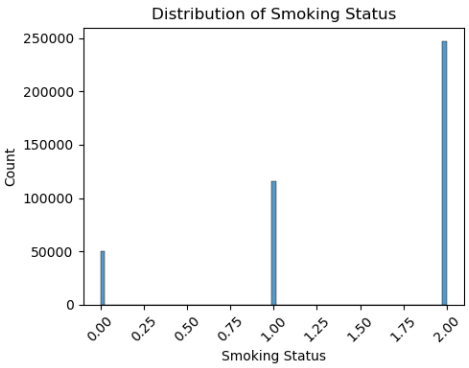
Generated visualizations from the analysis:

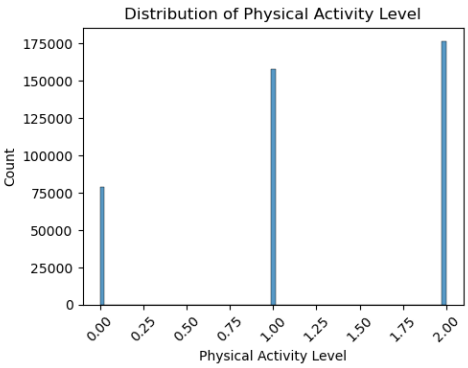

Distribution of Age
Count
Age

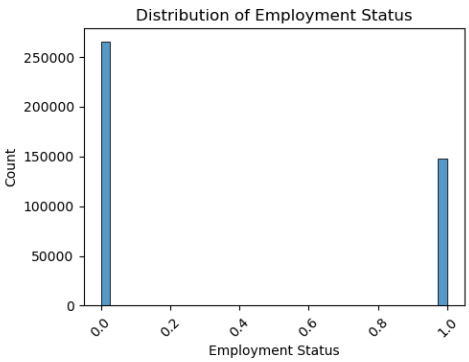

Distribution of Marital Status
Count
Marital Status

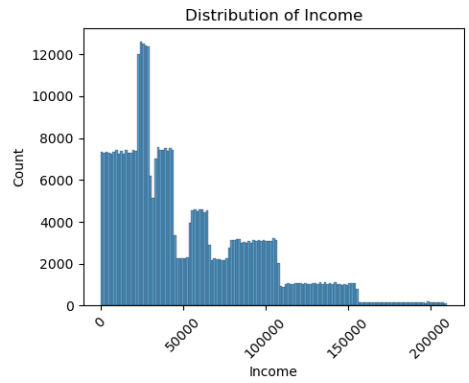

Distribution of Education Level
Count
Education Level

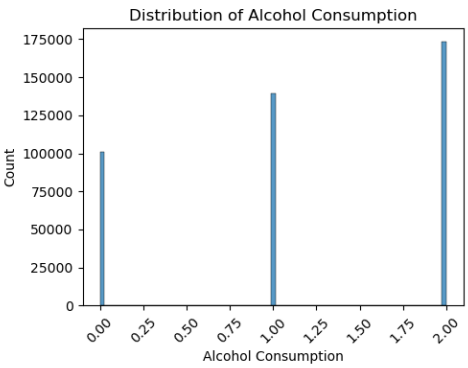

Distribution of Number of Children
Count
Number of Children


Distribution of Smoking Status
Count
Smoking Status

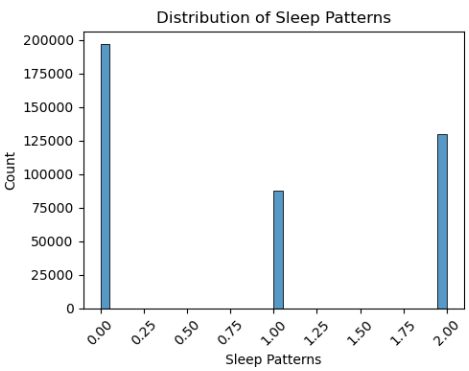

Distribution of Physical Activity Level
Count
Physical Activity Level

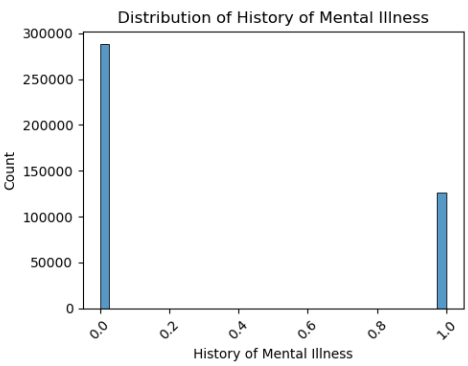

Distribution of Employment Status
Count
Employment Status

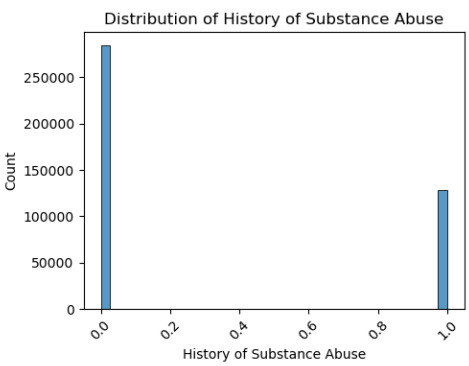

Distribution of Income
Count
Income

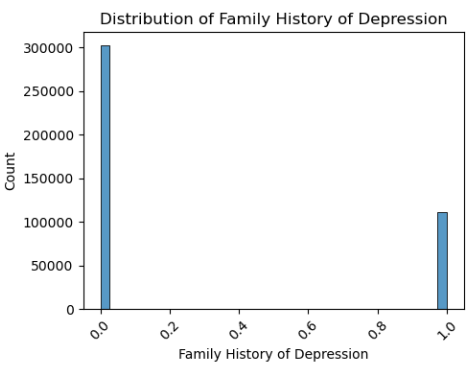

Distribution of Alcohol Consumption
Count
Alcohol Consumption

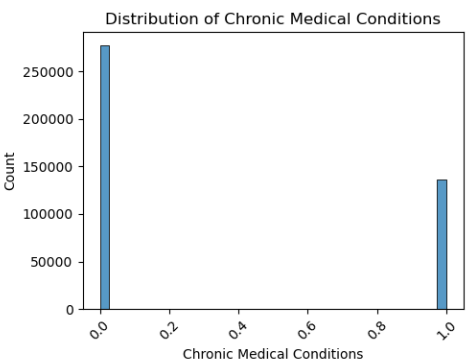

Distribution of Dietary Habits
Count
Dietary Habits


Distribution of Sleep Patterns
Count
Sleep Patterns

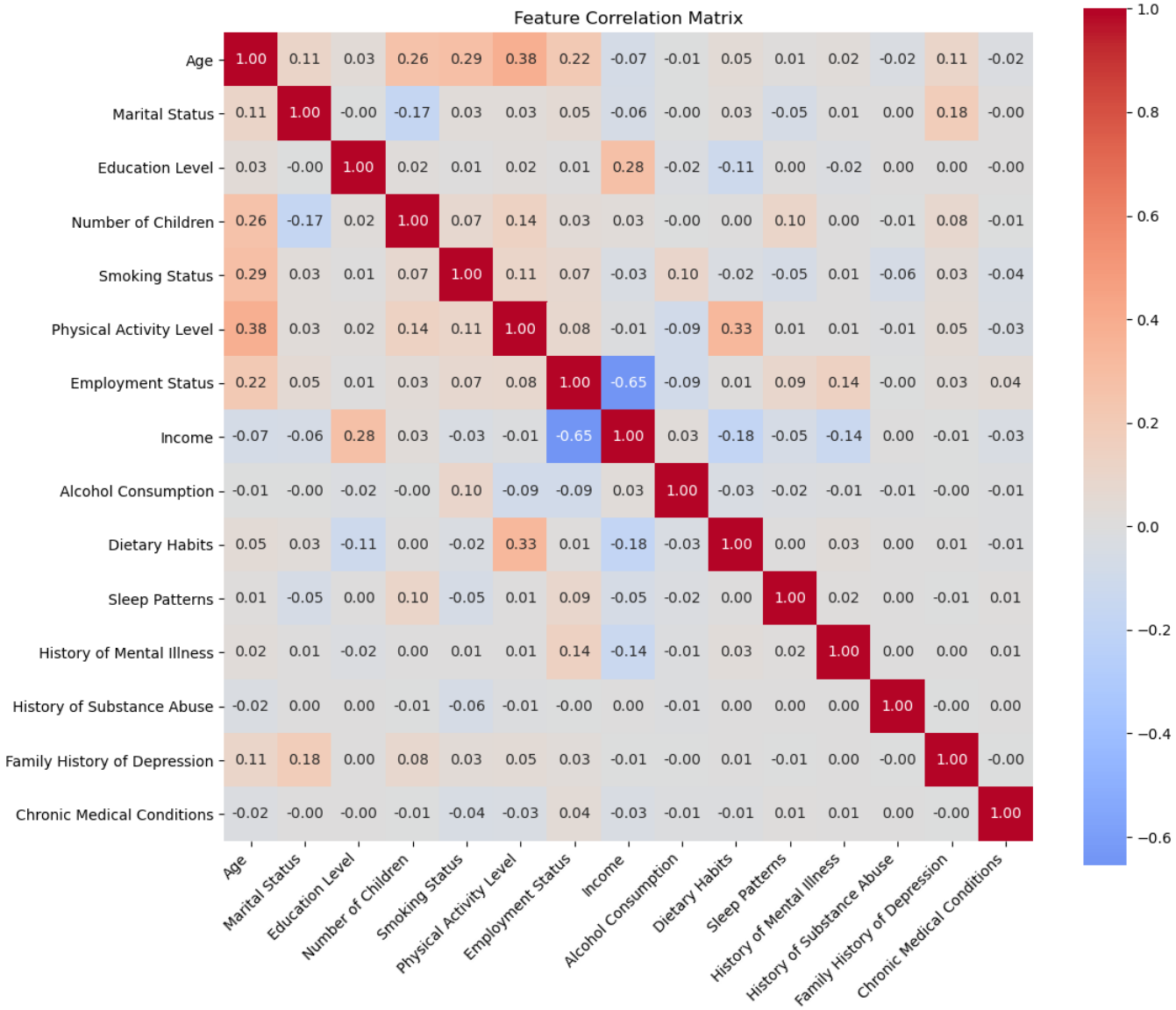

Distribution of History of Mental Illness
Count
History of Mental Illness


Distribution of History of Substance Abuse
Count
History of Substance Abuse


Distribution of Family History of Depression
Count
Family History of Depression


Distribution of Chronic Medical Conditions
Count
Chronic Medical Conditions

Feature Distributions



Correlation Matrix

Model Information

Latest Model File: models\model_20241223_125736_acc_06944.joblib

Created on: 2024-12-23 12:57:36.785205

Model Parameters:

{'base_score': 0.5, 'booster': 'gbtree', 'colsample_bylevel': 1, 'colsample_bynode': 1, 'colsample_bytree': 1, 'gamma': 0, 'learning_rate': 0.1, 'max_delta_step': 0, 'max_depth': 3, 'min_child_weight': 1, 'missing': nan, 'n_estimators': 100, 'n_jobs': 1, 'nthread': None, 'objective': 'binary:logistic', 'random_state': 48, 'reg_alpha': 0, 'reg_lambda': 1, 'scale_pos_weight': 1, 'seed': None, 'silent': None, 'subsample': 1, 'verbosity': 1}