Predicting Maternal Health Risk Using AI

(This is one of the possible solutions for the final project.)

```
# Install required packages
!pip install joblib
!pip install pandas
!pip install matplotlib
!pip install scikit-learn
!pip install tensorflow
```

1. Import required libraries

```
# Data manipulation and analysis
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Machine Learning - Preprocessing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Machine Learning - Models
from sklearn.ensemble import RandomForestClassifier

# Machine Learning - Metrics
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_curve, auc, classification_report, confusion_matrix
)

# Deep Learning
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Model persistence
import joblib

# Warnings
import warnings
warnings.filterwarnings('ignore')

# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

print("Libraries imported successfully!")
```

2. Data preparation

2.1 Load the dataset

```
# Load the dataset
df = pd.read_csv('https://advanced-machine-learning-for-medical-data-8e1579.gitlab.io/labs/final_project/Maternal_Health_R

print(f"Dataset shape: {df.shape}")
print(f"\nFirst few rows:")
df.head()
```

2.2 Initial data exploration

```
# Display basic information
print("Dataset Info:")
print(df.info())
print("\n" + "="*50 + "\n")
```

```
# Display statistical summary
print("Statistical Summary:")
df.describe()
```

```
# Check for missing values
print("Missing values per column:")
missing_values = df.isnull().sum()
print(missing_values)
print(f"\nTotal missing values: {missing_values.sum()}")
```

```
# Check for duplicates
duplicates = df.duplicated().sum()
print(f"Number of duplicate rows: {duplicates}")
```

```
# Check target variable distribution
print("Target variable (RiskLevel) distribution:")
print(df['RiskLevel'].value_counts())
print("\nPercentage distribution:")
print(df['RiskLevel'].value_counts(normalize=True) * 100)
```

2.3 Data cleaning

Remove duplicates

```
# Remove duplicate rows
df_cleaned = df.drop_duplicates()
print(f"Shape before removing duplicates: {df.shape}")
print(f"Shape after removing duplicates: {df_cleaned.shape}")
print(f"Removed {df.shape[0] - df_cleaned.shape[0]} duplicate rows")
```

Remove Personally Identifiable Information (PII)

```
# Remove CitizenID column (PII)
if 'CitizenID' in df_cleaned.columns:
    df_cleaned = df_cleaned.drop('CitizenID', axis=1)
    print("CitizenID column removed (PII)")

print(f"\nColumns after PII removal: {df_cleaned.columns.tolist()}")
```

Standardize risk level labels

```
# Standardize risk level labels (handle case variations)
df_cleaned['RiskLevel'] = df_cleaned['RiskLevel'].str.lower().str.strip()
print("Unique risk levels after standardization:")
print(df_cleaned['RiskLevel'].unique())
```

Handle missing values

```
# Impute missing values with median for numerical columns
numerical_cols = df_cleaned.select_dtypes(include=[np.number]).columns

print("Imputing missing values with median...")
for col in numerical_cols:
    if df_cleaned[col].isnull().sum() > 0:
        median_value = df_cleaned[col].median()
        df_cleaned[col].fillna(median_value, inplace=True)
        print(f"  - {col}: filled {df_cleaned[col].isnull().sum()} missing values with median {median_value}")

print(f"\nMissing values after imputation: {df_cleaned.isnull().sum().sum()}")
```

Data normalization (for visualization purposes)

```
# Create a normalized copy for visualization
df_normalized = df_cleaned.copy()
feature_cols = [col for col in numerical_cols if col in df_normalized.columns]

# Min-Max normalization for visualization
for col in feature_cols:
    min_val = df_normalized[col].min()
    max_val = df_normalized[col].max()
    df_normalized[col] = (df_normalized[col] - min_val) / (max_val - min_val)
```

```
print("Data normalized for visualization")
print("\nNormalized data summary:")
df_normalized[feature_cols].describe()
```

2.4 Exploratory data analysis (EDA)

```
# Visualize target variable distribution
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
df_cleaned['RiskLevel'].value_counts().plot(kind='bar', color=['green', 'red'])
plt.title('Risk Level Distribution (Count)')
plt.xlabel('Risk Level')
plt.ylabel('Count')
plt.xticks(rotation=45)

plt.subplot(1, 2, 2)
df_cleaned['RiskLevel'].value_counts().plot(kind='pie', autopct='%1.1f%%', colors=['green', 'red'])
plt.title('Risk Level Distribution (Percentage)')
plt.ylabel('')

plt.tight_layout()
plt.show()
```

3. Patient risk scoring - Random Forest model

3.1 Prepare data for modeling

```
# Identify features and target
feature_columns = ['Age', 'SystolicBP', 'DiastolicBP', 'BS', 'BodyTemp', 'HeartRate']
target_column = 'RiskLevel'

# Prepare X (features) and y (target)
X = df_cleaned[feature_columns]
y = (df_cleaned[target_column] == 'high risk').astype(int)  # Binary: 1=high risk, 0=low risk

print(f"Features shape: {X.shape}")
print(f"Target shape: {y.shape}")
print(f"\nTarget distribution:")
print(f"  Low Risk (0): {(y == 0).sum()}")
print(f"  High Risk (1): {(y == 1).sum()}")
```

3.2 Train-Test split (DO NOT scale for Random Forest)

```
# Train-Test split with stratification
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y
)

print(f"Training set size: {X_train.shape[0]}")
print(f"Test set size: {X_test.shape[0]}")
print(f"\nTraining set target distribution:")
print(f"  Low Risk: {(y_train == 0).sum()} ({(y_train == 0).sum() / len(y_train) * 100:.2f}%)")
print(f"  High Risk: {(y_train == 1).sum()} ({(y_train == 1).sum() / len(y_train) * 100:.2f}%)")
print(f"\nTest set target distribution:")
print(f"  Low Risk: {(y_test == 0).sum()} ({(y_test == 0).sum() / len(y_test) * 100:.2f}%)")
print(f"  High Risk: {(y_test == 1).sum()} ({(y_test == 1).sum() / len(y_test) * 100:.2f}%)")
```

3.3 Train Random Forest model

```
# Initialize and train Random Forest Classifier
rf_model = RandomForestClassifier(
    n_estimators=100,
    random_state=42,
    n_jobs=-1
)

print("Training Random Forest model...")
```

```
    rf_model.fit(X_train, y_train)
    print("Training complete!")
```

3.4 Model evaluation

```
# Make predictions
y_pred = rf_model.predict(X_test)
y_pred_proba = rf_model.predict_proba(X_test)[:, 1]

# Calculate metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print("Random Forest Model Performance on Test Set:")
print("=" * 50)
print(f"Accuracy:  {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall:    {recall:.4f}")
print(f"F1-Score:  {f1:.4f}")
```

```
# Detailed classification report
print("\nDetailed Classification Report:")
print("=" * 50)
target_names = ['Low Risk', 'High Risk']
print(classification_report(y_test, y_pred, target_names=target_names))
```

```
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Low Risk', 'High Risk'],
            yticklabels=['Low Risk', 'High Risk'])
plt.title('Confusion Matrix - Random Forest')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.tight_layout()
plt.show()
```

3.5 ROC-AUC curve

```
# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure(figsize=(10, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2,
         label=f'ROC curve (AUC = {roc_auc:.4f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--',
         label='Random Classifier (AUC = 0.5)')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve - Random Forest')
plt.legend(loc="lower right")
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()

print(f"ROC-AUC Score: {roc_auc:.4f}")
```

3.6 Feature importance

```
# Get feature importances
feature_importance = pd.DataFrame({
    'Feature': feature_columns,
    'Importance': rf_model.feature_importances_
}).sort_values('Importance', ascending=False)

print("Feature Importances:")
print("=" * 50)
```

```
for idx, row in feature_importance.iterrows():
    print(f"{row['Feature']:15s}: {row['Importance']:.4f}")
```

```
# Visualize feature importances
plt.figure(figsize=(10, 6))
plt.barh(feature_importance['Feature'], feature_importance['Importance'], color='steelblue')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importance - Random Forest Model')
plt.gca().invert_yaxis()
plt.grid(alpha=0.3, axis='x')
plt.tight_layout()
plt.show()
```

3.7 Save Random Forest model

```
# Save the trained Random Forest model
joblib.dump(rf_model, 'risk_model.joblib')
print("Random Forest model saved as 'risk_model.joblib'")
```

4. Dense neural network model

4.1 Scale data for neural network

```
# Scale the features using StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("Data scaled using StandardScaler")
print(f"\nScaled training data shape: {X_train_scaled.shape}")
print(f"Scaled test data shape: {X_test_scaled.shape}")
print(f"\nMean of scaled training data: {X_train_scaled.mean(axis=0)}")
print(f"Std of scaled training data: {X_train_scaled.std(axis=0)}")
```

4.2 Build dense neural network architecture

```
# Build neural network model
dnn_model = keras.Sequential([
    layers.Input(shape=(X_train_scaled.shape[1],)),
    layers.Dense(64, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(16, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

# Compile the model
dnn_model.compile(
    loss='binary_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
)

print("Neural Network Architecture:")
print("=" * 50)
dnn_model.summary()
```

4.3 Train neural network model

```
# Train the model
print("Training Dense Neural Network...")
history = dnn_model.fit(
    X_train_scaled, y_train,
    epochs=50,
    batch_size=32,
    validation_split=0.2,
    verbose=1
)
print("\nTraining complete!")
```

4.4 Visualize training history

```
# Plot training history
fig, axes = plt.subplots(1, 2, figsize=(15, 5))

# Plot loss
axes[0].plot(history.history['loss'], label='Training Loss')
axes[0].plot(history.history['val_loss'], label='Validation Loss')
axes[0].set_title('Model Loss During Training')
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Loss')
axes[0].legend()
axes[0].grid(alpha=0.3)

# Plot accuracy
axes[1].plot(history.history['accuracy'], label='Training Accuracy')
axes[1].plot(history.history['val_accuracy'], label='Validation Accuracy')
axes[1].set_title('Model Accuracy During Training')
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('Accuracy')
axes[1].legend()
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()
```

4.5 Evaluate neural network model

```
# Make predictions
y_pred_dnn_proba = dnn_model.predict(X_test_scaled).flatten()
y_pred_dnn = (y_pred_dnn_proba > 0.5).astype(int)

# Calculate metrics
accuracy_dnn = accuracy_score(y_test, y_pred_dnn)
precision_dnn = precision_score(y_test, y_pred_dnn)
recall_dnn = recall_score(y_test, y_pred_dnn)
f1_dnn = f1_score(y_test, y_pred_dnn)

print("Dense Neural Network Model Performance on Test Set:")
print("=" * 50)
print(f"Accuracy:  {accuracy_dnn:.4f}")
print(f"Precision: {precision_dnn:.4f}")
print(f"Recall:    {recall_dnn:.4f}")
print(f"F1-Score:  {f1_dnn:.4f}")
```

```
# Detailed classification report
print("\nDetailed Classification Report:")
print("=" * 50)
print(classification_report(y_test, y_pred_dnn, target_names=['Low Risk', 'High Risk']))
```

```
# Confusion Matrix for Neural Network
cm_dnn = confusion_matrix(y_test, y_pred_dnn)

plt.figure(figsize=(8, 6))
sns.heatmap(cm_dnn, annot=True, fmt='d', cmap='Greens',
            xticklabels=['Low Risk', 'High Risk'],
            yticklabels=['Low Risk', 'High Risk'])
plt.title('Confusion Matrix - Dense Neural Network')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.tight_layout()
plt.show()
```

4.6 Save neural network model

```
# Save the trained neural network model
dnn_model.save('risk_model_dnn.keras')
print("Dense Neural Network model saved as 'risk_model_dnn.keras'")
```

5. Model comparison

```python
# Compare model performance
comparison_df = pd.DataFrame({
    'Model': ['Random Forest', 'Dense Neural Network'],
    'Accuracy': [accuracy, accuracy_dnn],
    'Precision': [precision, precision_dnn],
    'Recall': [recall, recall_dnn],
    'F1-Score': [f1, f1_dnn]
})

print("Model Performance Comparison:")
print("=" * 70)
print(comparison_df.to_string(index=False))
```

```python
# Visualize model comparison
metrics = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
x = np.arange(len(metrics))
width = 0.35

fig, ax = plt.subplots(figsize=(12, 6))
rects1 = ax.bar(x - width/2, comparison_df.iloc[0, 1:], width, label='Random Forest', color='steelblue')
rects2 = ax.bar(x + width/2, comparison_df.iloc[1, 1:], width, label='Dense Neural Network', color='forestgreen')

ax.set_ylabel('Score')
ax.set_title('Model Performance Comparison')
ax.set_xticks(x)
ax.set_xticklabels(metrics)
ax.legend()
ax.set_ylim([0, 1.1])
ax.grid(alpha=0.3, axis='y')

# Add value labels on bars
def autolabel(rects):
    for rect in rects:
        height = rect.get_height()
        ax.annotate(f'{height:.3f}',
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3),
                    textcoords="offset points",
                    ha='center', va='bottom')

autolabel(rects1)
autolabel(rects2)

plt.tight_layout()
plt.show()
```