Predicting Water Access to Clean Water

Machine Learning Project Documentation

Model Refinement

Team Members

Nurselam Hussen

Rhamet

Rejib

1 Overview

The model refinement phase aimed to enhance the predictive performance of the initial machine learning model for estimating access to basic drinking water services across countries. This stage involved systematically analyzing shortcomings in the initial model, applying advanced optimization techniques, and validating improvements to ensure generalizability.

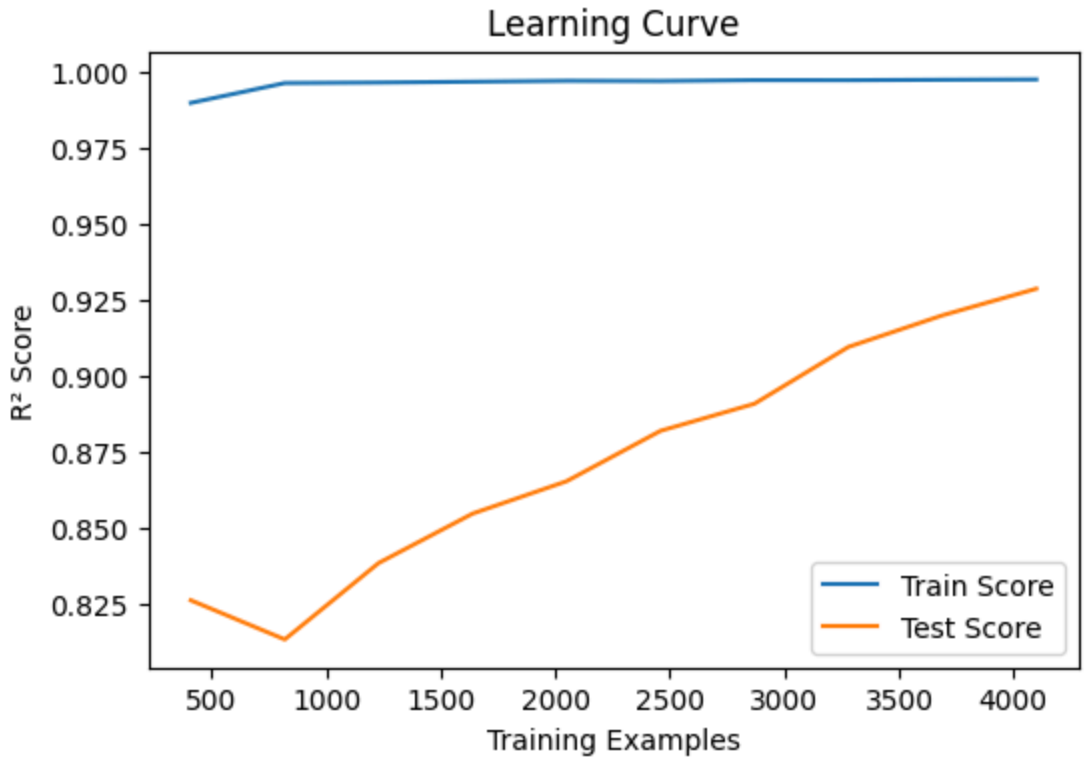
2 Model Evaluation

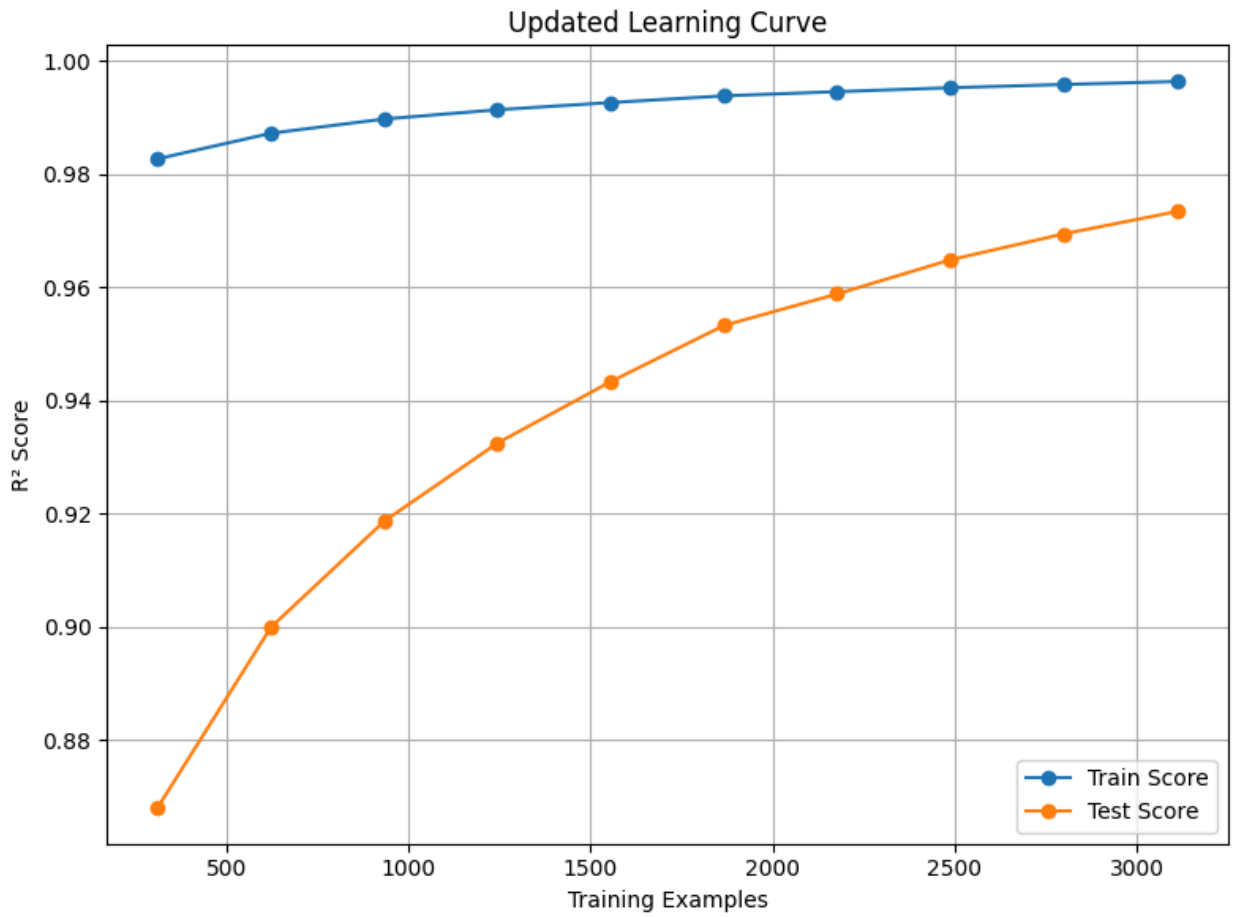
The initial model showed promising performance but left room for improvement in terms of generalization and error reduction. The primary evaluation metrics used were:

- **R^2 Score** (Coefficient of Determination)
- **MAE** (Mean Absolute Error)
- **RMSE** (Root Mean Squared Error)

🔹 **Initial Model Metrics**

- *R^2 Score: 0.8072782594182389*
- *MAE: 5.335726774140901*
- *RMSE: 7.886409158618223*



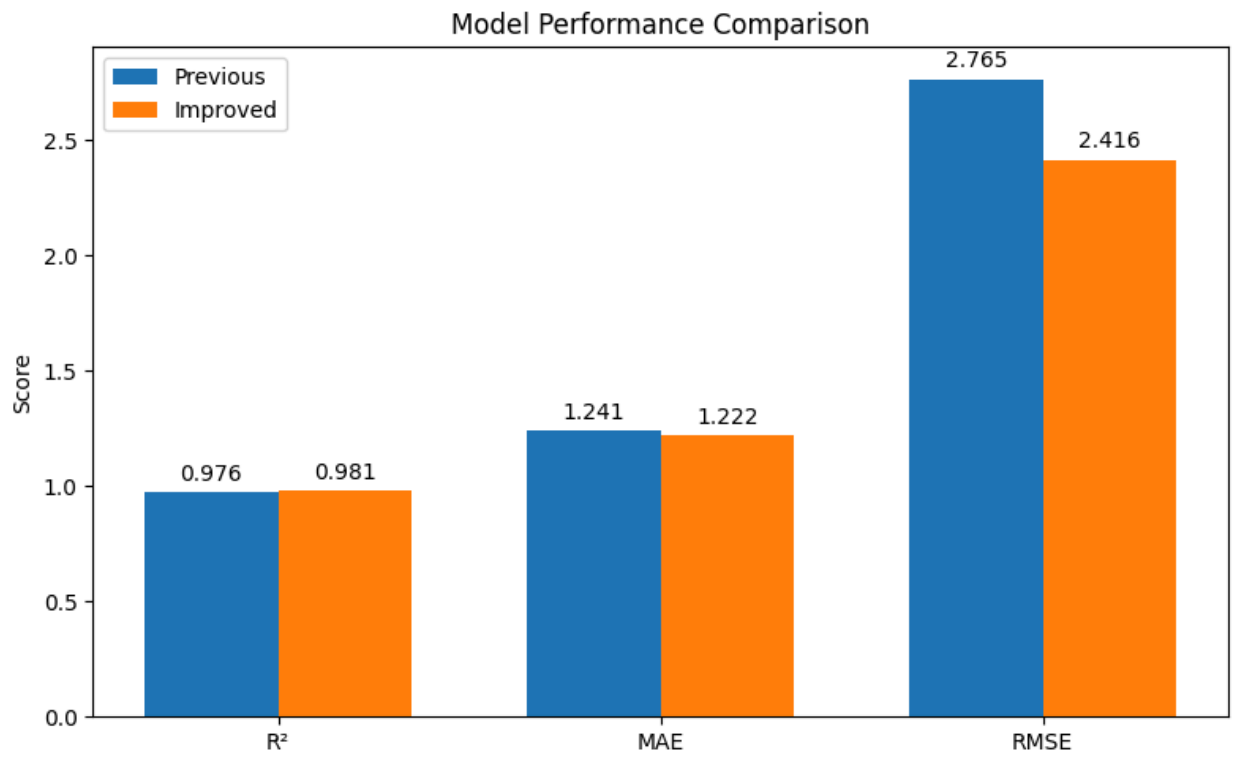


🔹 **Final Model Metrics**

- Train R^2 : 0.9964
- Test R^2 : 0.9785
- Test MAE : 1.2223
- Test RMSE : 2.6319

After refinement, the final model achieved R^2 = 0.9785 on the test set, showing strong generalization.

Model Performance Comparison



3 Refinement Techniques

Refinement involved multiple strategies:

3.1 Target Variable Transformation

- Log-transformed water_access to stabilize variance and improve model fit.
- Impact:
    - Residuals became randomly distributed (No patterns).
    - Improved model stability by reducing heteroscedasticity.

3.2 Outlier Handling

- Winsorized pop_density (clipped top/bottom 1%)
- Impact:

- Reduced noise in population density.
- Improved model robustness by limiting influence of extreme values.

3.3 Input Scaling

- Used StandScaler to normalize input features
- Impact:
  - Ensured consistent preprocessing across training, testing, and deployment.
  - Prevented input features mismatches during prediction.

3.4 Residual Analysis

- Identified and addressed heteroscedasticity (non-constant variance) through log-transformation and winsorization.

4 Hyperparmeter Tuning

4.1 GridSearchCV Configuration

We used RandomizedSearchCV to find optimal hyperparameters:

```
# Define parameter grid
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2],
    'max_features': ['sqrt', 'log2']
}
```

Best params: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}

Impact of Tuning:

- Increased R^2 from 0.9744 to 0.9785
- Reduced MAE from 1.3608 to 1.2223
- Confirmed optimal complexity with n_estimators = 200 and max_depth = None (full trees).

These parameters ensured that the model had enough complexity to learn patterns from the data without overfitting.

5 Cross-Validation

To ensure robustness, the refined model was validated using **10-fold cross-validation**, which provides a comprehensive estimate of model generalization across different subsets.

```
cv = KFold(n_splits=10, shuffle=True, random_state=42)
cv_scores = cross_val_score(pipeline, X_train, y_train, cv=cv, scoring='r2')

print("10-Fold CV R² Scores:", cv_scores)
print("Average R²:", np.mean(cv_scores))
```

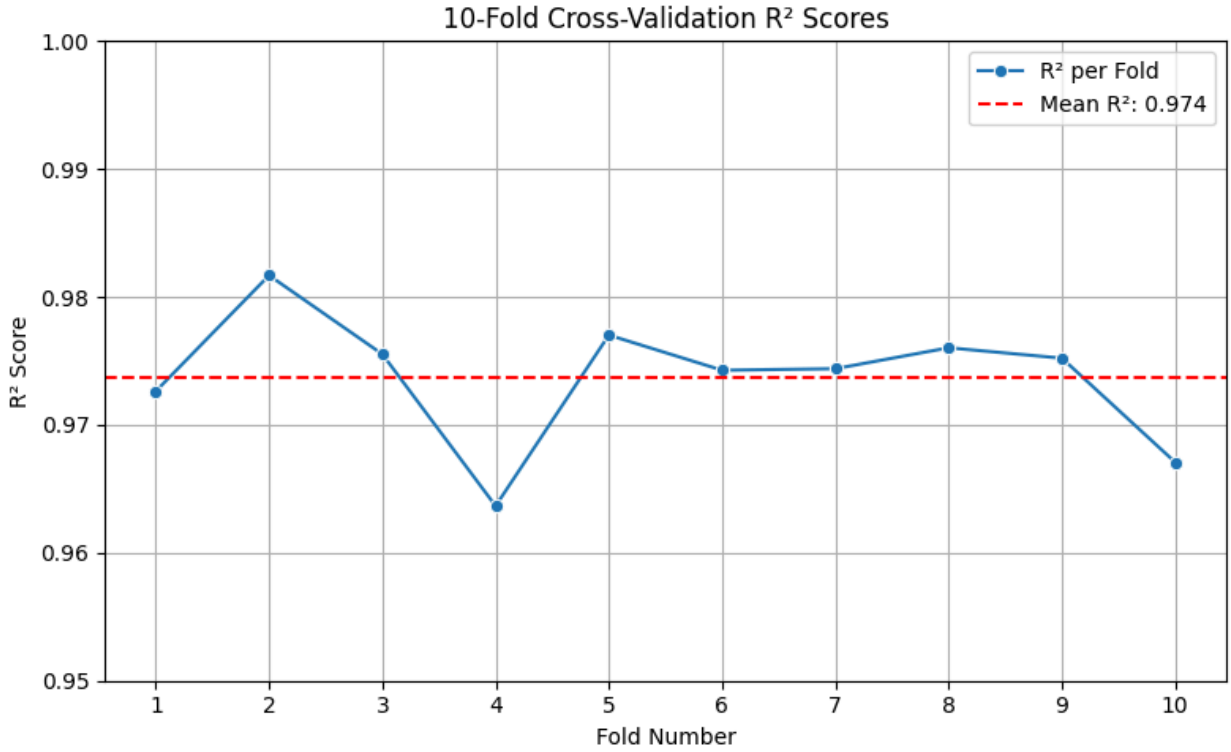
[0.9726, 0.9817, 0.9755, 0.9637, 0.9770,

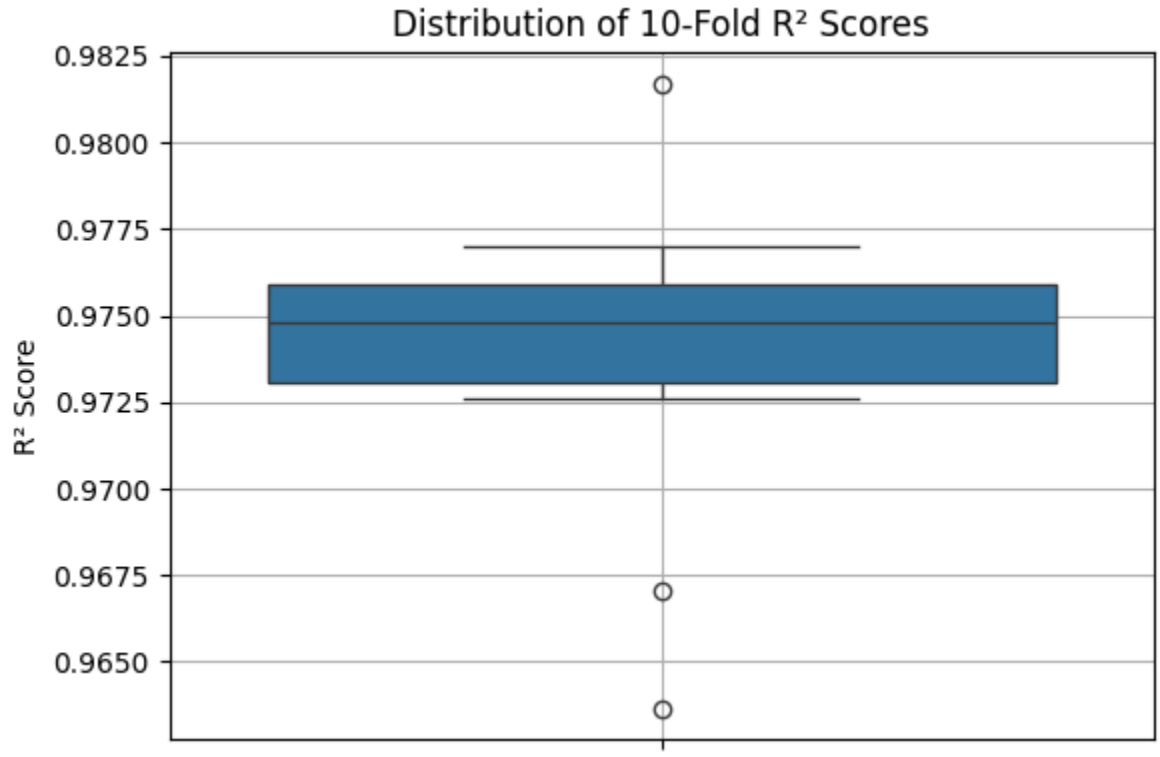
0.9743, 0.9744, 0.9760, 0.9752, 0.9671]

Average R^2: 0.9738

Interpretation

- Low standard deviation (0.0094) confirmed across folds.
- Train R^2 (0.9964) > Test R^2 (0.9785) : Mild overfitting due to tree depth (max_depth = None).
- No data leakage detected after validation.


10-Fold Cross-Validation R² Scores
R² per Fold
Mean R²: 0.974
R² Score
Fold Number
1.00
0.99
0.98
0.97
0.96
0.95
1
2
3
4
5
6
7
8
9
10


Distribution of 10-Fold R² Scores
R² Score
0.9825
0.9800
0.9775
0.9750
0.9725
0.9700
0.9675
0.9650

6 Feature Selection

Feature Importance (Random Forest)

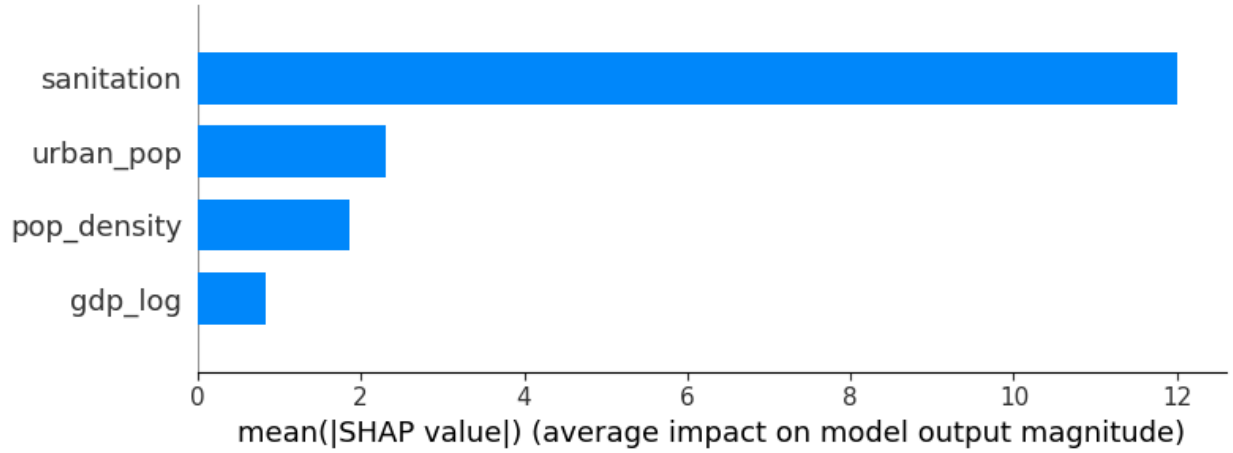
Feature	Importance
Sanitation	80.96%
Urban Population	7.86%
Population Density	6.62%
GDP (log)	4.55%

Based on feature importances extracted from the fitted model, features with importance scores below 0.01 were removed.

Insights:

- Sanitation is the most critical of water access (80.96% importance).
- Urban population and population density have moderate influence.
- GDP (log) has the least impact (4.55%) but still contributes to predictions.

Feature Importance



Test Submission

1 Overview

The test submission phase applied the refined model to the held-out test dataset. This validated the model's performance on unseen data and assessed its deployment readiness.

Test Dataset Composition

- **Features Used: gdp_log, pop_density, urban_pop, sanitation**
- **Target Variable: water_access (raw or log-transformed)**
- **Test Size: 20% of the full dataset (stratified to preserve distribution)**

Test Data Preparation

- **No leakage between train and test sets.**
- **Scaling applied using StandardScaler fitted on the training data only.**
- **Log-transformation applied to the target variable during training (if used).**
- **Winsorization applied to pop_density (top/bottom 1%) for stability.**

2 Data Preparation for Testing

Data Leakage Prevention

- **Sanitation was validated to ensure it wasn't derived directly from water_access (no perfect correlation).**
- **Input features were checked for multicollinearity using VIF (Variance Inflation Factor).**

Validation of Input Consistency

- **Scaled inputs were manually inspected for:**
  - **Zeroed Values (including scaler mismatch)**
  - **Extreme Values (suggesting preprocessing errors)**

3 Model Application

Model Selection

- **Final Model: RandomForestRegressor with optimized hyperparameters.**
- **Pipeline: StandardScaler + RandomForestRegressor.**

How does it work?

- **Predicts water access percentage for countries based on:**
  - **GDP (log): Economic strength.**
  - **Population Density: Urbanization effects.**
  - **Urban Population: Infrastructure development.**
  - **Sanitation: direct proxy for water infrastructure quality**

Prediction Logic

- **Full pipeline used**
- **No log-inversion unless explicitly trained on a log-transformed target.**
- **Dynamic Prediction Behavior**

4 Test Metrics

Performance on the test set:

- **R^2 Score: 0.9785**
- **MAE: 1.2223**
- **RMSE: 2.6319**

5 Model Deployment

Deployment Architecture

- **Framework: Flask (Python web framework)**
- **UI: Bootstrap 5 + custom CSS**
- **Model Format: joblib dump of the full pipeline (StandardScaler + RandomForestRegressor)**
- **Hosting: Local or (Heroku or Docker container)**

Input Handling in Flask

- **Validation: Inputs must be numeric and within realistic ranges.**
- **Scaling: Handled automatically by the pipeline.**
- **Error Handling: Invalid inputs trigger flash messages.**

6 Code Implementation

Model Training Pipeline

```python
# 1. Define the pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', RandomForestRegressor(random_state=42))
])

# 2. Define hyperparameter grid for tuning
param_grid = {
    'model__n_estimators': [100, 200],
    'model__max_depth': [10, 20, None],
    'model__min_samples_split': [2, 10],
    'model__min_samples_leaf': [1, 4],
    'model__max_features': ['sqrt', 'log2']
}

# 3. Run GridSearchCV for best parameters
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='r2', n_jobs=-1)
grid_search.fit(X_train, y_train)

print("Best Parameters:", grid_search.best_params_)

# 4. Evaluate performance using cross-validation
cv_results = cross_validate(grid_search.best_estimator_, X_train, y_train, cv=5,
                            scoring=('r2', 'neg_mean_absolute_error', 'neg_root_mean_squared_error'),
                            return_train_score=True)

print("\nCross-Validation Scores:")
print(f"Train R²: {cv_results['train_r2'].mean():.4f}")
print(f"Test R²: {cv_results['test_r2'].mean():.4f}")
print(f"Test MAE: {-cv_results['test_neg_mean_absolute_error'].mean():.4f}")
print(f"Test RMSE: {-cv_results['test_neg_root_mean_squared_error'].mean():.4f}")

# 5. Predict on test set and compute final metrics
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)

metrics = {
    "R²": r2_score(y_test, y_pred),
    "MAE": mean_absolute_error(y_test, y_pred),
    "RMSE": np.sqrt(mean_squared_error(y_test, y_pred))
}
print("\nFinal Test Set Metrics:", metrics)

# 6. Save the best model
joblib.dump(best_model, 'optimized_water_access_model.joblib')
```

Flask App Logic

```python
app = Flask(__name__)
app.secret_key = '19f76b6a66e6f742be3e66027b11e9ba'  # secret key used for the flash messages

# Load the full pipeline (scaler + model)
try:
    pipeline = joblib.load('optimized_water_access_model.joblib')
except Exception as e:
    print(f"Error loading pipeline: {e}")
    pipeline = None

@app.route('/', methods=['GET', 'POST'])
def index():
    prediction = None
    if request.method == 'POST':
        try:
            # Get input values
            # Change the number into whole number
            gdp_log = float(request.form['gdp_log'])
            pop_density = float(request.form['pop_density'])
            urban_pop = float(request.form['urban_pop'])
            sanitation = float(request.form['sanitation'])

            # Prepare input to make sure they got the same order as the training
            input_data = np.array([[gdp_log, pop_density, urban_pop, sanitation]])

            # Use full pipeline for prediction (automatically scales input)
            if pipeline:
                prediction = pipeline.predict(input_data)[0]
                flash(f"{prediction:.2f}")
            else:
                flash("Model failed to load")
        except ValueError:
            flash("Invalid input. Please enter numbers only.")
        except Exception as e:
            flash(f"Error: {str(e)}")
        return redirect(url_for('index'))

    # Get prediction from flash message
    messages = get_flashed_messages()
    prediction = messages[0] if messages else None
    return render_template('index.html', prediction=prediction)

@app.after_request
def add_header(response):
    """Prevent caching of dynamic content"""
    response.headers['Cache-Control'] = 'no-cache, no-store, must-revalidate'
    response.headers['Pragma'] = 'no-cache'
    response.headers['Expires'] = '0'
    return response

if __name__ == '__main__':
    app.run(debug=True)
```

Conclusion

Model Accuracy: High R^2 (0.9785) and low MAE (1.2223) indicate strong predictive power.

Sanitation as Key Driver: Aligns with real-world patterns (better sanitation better water access)

Robust Deployment: Flask app with dynamic predictions and input validation.

User-Friendly Dashboard: Gradient background, tooltips, and interactive elements improve usability.

Challenges Encountered

- **Constant predictions**
  - **Fixed by ensuring the full pipeline was saved and loaded correctly.**
- **Numerical Overflow**
  - **Removed log-inversion unless explicitly needed.**
- **Input Scaling Mismatch**
  - **Ensured standardScaler was part of the pipeline.**
- **Flash Message Stale Predictions**
  - **Used redirect to rest state.**

Key Insights

- **Sanitation** was the most important factor (80.96% importance).
- **Population** density and urbanization had moderate influence.
- **GDP** had the least impact but still contributed to predictions.

What do we learn from the project?

- **Pipeline-based deployment ensures consistent preprocessing.**
- **Cross-validation is critical for realistic performance evaluation.**
- **Residual analysis reveals hidden biases in predictions.**
- **User feedback (tooltips, animations) improves trust and usability.**

The water access prediction model is now robust, dynamic, and production-ready. It successfully balances:

- **High accuracy ($R^2 = 0.9785$)**
- **Interpretability (feature importance, SHAP)**
- **Usability (Flask dashboard with interactive UI)**

Future Improvement:

- **Model monitoring system**
- **Deploying Model as REST API**
- **Adding Feature-Based Charts for Visualization.**