Analysis of the Florida iBudget Algorithm: Current Limitations and Proposed Quantitative Alternatives

agency for persons with disabilities

State of Florida

Information Systems of Florida

September 5, 2025

iBudget Algorithm Study
Agency for Persons with Disabilities (ADP)
Procurement Office: Tamara Harrington
4030 EWsplanade Way
Tallahassee, Florida 32399

Study prepared by Information Systems of Florida.
Abbie David, PMP, PhD, Project Manager.
Jason Dillaberry, Client Partner
Matthew Fisher, JD, LLM, MBA, Principal Consultant
Allison Kunerth, PhD, Senior Management Consultant
Juan B. Gutiérrez, PhD, Mathematician
Jessica Kemper, MPA, Senior Management Consultant
Chris Klass, MPP Finance, Senior Management Consultant
Annika Baeten, Management Consultant
Daniel Margolis, Associate Management Consultant
adavid@isf.com

Information Systems of Florida
September 5, 2025

Contents

Chapter 1

Introduction

1.1 Introduction

The Florida iBudget algorithm represents a critical component of the state's developmental disability services infrastructure, determining individual budget allocations for Home and Community-Based Services (HCBS) under the Developmental Disabilities Individual Budgeting waiver program. This system currently serves over 36,000 enrollees, making algorithmic decisions that directly impact the quality of life and service access for individuals with developmental disabilities across Florida. The algorithm's role extends beyond mere budget calculation; it fundamentally shapes how resources are distributed, what services individuals can access, and how person-centered planning principles are implemented in practice.

The enactment of House Bill 1103 in the 2025 legislative session has fundamentally altered the regulatory landscape for iBudget allocation methodologies. This legislation mandates a comprehensive study to review, evaluate, and identify recommendations regarding the current algorithm, with particular emphasis on ensuring compliance with person-centered planning requirements under section 393.0662, Florida Statutes. The bill's requirements extend beyond simple algorithmic refinement, demanding a fundamental reassessment of how statistical methods align with person-centered planning principles and contemporary disability services philosophy.

This analysis addresses three interconnected questions that form the foundation for algorithm evaluation and redesign. First, we examine what the current algorithm accomplishes, including its mathematical formulation, variable selection, and operational mechanics. This examination reveals both the system's statistical foundations and its practical implications for budget determination across diverse disability populations. Second, we identify critical weaknesses in the current approach, ranging from temporal validity issues stemming from outdated data to fundamental limitations in capturing person-centered planning elements. These weaknesses extend beyond technical statistical concerns to encompass broader questions about algorithmic fairness, transparency, and compliance with evolving disability rights frameworks.

Third, we analyze specific areas where the current algorithm fails to meet the requirements established in House Bill 1103, particularly regarding person-centered planning integration, data currency, and algorithmic robustness. This compliance analysis reveals systematic gaps between the algorithm's actuarial focus and the legislation's emphasis on individualized, preference-driven service planning. The analysis demonstrates that addressing these compliance issues requires more than technical adjustments; it demands a fundamental reconceptualization of how algorithmic systems can support rather than constrain person-centered planning processes.

The analysis presented in this document extends beyond identifying weaknesses to propose systematic approaches for algorithmic improvement that address both technical limitations and compliance requirements. These approaches range from enhanced linear regression methods that maintain interpretability while improving robustness, to sophisticated machine learning techniques that can capture complex relationships between individual characteristics and support needs, to hybrid approaches that combine statistical prediction with clinical judgment and person-centered planning elements.

The implementation strategy outlined in this analysis emphasizes phased deployment with comprehensive validation and monitoring to ensure that algorithmic improvements translate into meaningful improvements in service delivery and individual outcomes. This approach recognizes that algorithmic change in disability services carries profound implications for individual wellbeing and requires careful attention to unintended consequences and implementation challenges.

This comprehensive analysis serves multiple audiences and purposes within Florida's disability services ecosystem. For policymakers and legislative oversight bodies, it provides the technical foundation required by House Bill 1103 while translating complex statistical concepts into policy-relevant insights about algorithmic performance and compliance. For APD administrators and program managers, it offers practical guidance for algorithm selection and implementation while highlighting operational considerations that affect day-to-day service delivery.

For disability advocacy organizations and individuals receiving services, this analysis provides transparency about algorithmic decision-making processes and identifies specific areas where current methods may not adequately serve person-centered planning principles. For researchers and technical practitioners, it offers detailed methodological analysis and implementation guidance that can inform algorithm development and validation processes.

The analysis ultimately argues that effective algorithmic systems in disability services require more than statistical sophistication; they demand explicit integration of person-centered planning principles, transparent decision-making processes, and ongoing adaptation to changing service delivery contexts. The current algorithm's limitations stem not merely from technical deficiencies but from a fundamental misalignment between actuarial prediction methods and the individualized, preference-driven approaches that define quality disability services.

Moving forward, Florida's iBudget system requires algorithmic approaches that can simultaneously achieve statistical rigor, regulatory compliance, person-centered planning integration, and operational practicality. The alternative approaches presented in this analysis offer pathways toward these multiple objectives while acknowledging the inherent tensions and tradeoffs involved in algorithmic design for disability services. The ultimate success of these approaches will depend not only on their technical implementation but on their ability to support rather than constrain the person-centered planning processes that remain central to effective disability services.

1.2 Analysis of the Questionnaire for Situational Information (QSI): Data Types and Model Deficiencies

The Florida Questionnaire for Situational Information (QSI) Version 4.0 represents a comprehensive assessment instrument designed to evaluate support needs for individuals with developmental disabilities. This analysis examines the data structure, identifies critical deficiencies in the proposed statistical models, and recommends advanced modeling approaches to address these limitations.

1.2.1 QSI Data Structure and Question Categories

The QSI contains comprehensive assessment data organized into three primary domains, each utilizing ordinal scales ranging from 0 (no support needed) to 4 (intensive support required).

1.2.1.1 Functional Status Questions (Q14-Q24)

The functional status domain comprises 11 elements assessing daily living support needs:

- **Q14 - Vision**: Visual impairment assessment (0=no impairment, 4=constant assistance required)
- **Q15 - Hearing**: Hearing impairment assessment (0=no impairment, 4=constant assistance required)
- **Q16 - Eating**: Eating support needs (0=independent, 4=total assistance required)
- **Q17 - Ambulation**: Mobility support needs (0=independent, 4=constant assistance required)
- **Q18 - Transfers**: Transfer support needs (0=independent, 4=total assistance required)
- **Q19 - Toileting**: Toileting support needs (0=independent, 4=total assistance required)

- **Q20 - Hygiene**: Personal hygiene support needs (0=independent, 4=total assistance required)
- **Q21 - Dressing**: Dressing support needs (0=independent, 4=total assistance required)
- **Q22 - Communications**: Communication support needs (0=no impairment, 4=constant assistance required)
- **Q23 - Self-Protection**: Safety awareness and self-protection (0=independent, 4=constant supervision required)
- **Q24 - Evacuation Ability**: Emergency evacuation capability (0=independent, 4=total assistance required)

1.2.1.2 Behavioral Status Questions (Q25-Q30)

The behavioral domain encompasses 6 elements evaluating intervention needs for challenging behaviors:

- **Q25 - Self-Injurious Behavior**: Interventions for self-harm behaviors (0=none required, 4=physical/mechanical restraint used)
- **Q26 - Aggressive/Hurtful to Others**: Interventions for aggressive behaviors (0=none required, 4=secure facility placement)
- **Q27 - Destructive to Property**: Interventions for property damage (0=none required, 4=secure facility placement)
- **Q28 - Inappropriate Sexual Behavior**: Interventions for sexual behavior issues (0=none required, 4=secure facility placement)
- **Q29 - Running Away**: Interventions for elopement behaviors (0=none required, 4=secure facility placement)
- **Q30 - Other Behaviors**: Other behaviors leading to separation (0=none required, 4=secure facility placement)

1.2.1.3 Physical Status Questions (Q32-Q50)

The physical domain contains 19 elements addressing health and medical concerns:

- **Q32 - Self-Injury Related Injuries**: Injury severity from self-injurious behavior
- **Q33 - Aggression Related Injuries**: Injury severity from aggressive behavior
- **Q34 - Mechanical Restraints**: Use of protective equipment for behavioral issues
- **Q35 - Emergency Chemical Restraint**: Use of emergency chemical interventions
- **Q36 - Psychotropic Medications**: Psychotropic medication usage patterns
- **Q37 - Gastrointestinal Conditions**: GI-related health issues including reflux, vomiting
- **Q38 - Seizures**: Seizure-related conditions and management
- **Q39 - Anti-Epileptic Medications**: Anti-seizure medication usage
- **Q40 - Skin Breakdown**: Skin integrity issues

- **Q41 - Bowel Function**: Bowel management needs
- **Q42 - Nutrition**: Nutritional support requirements
- **Q43 - Treatment (Physician Prescribed)**: Physician-prescribed treatments
- **Q44 - Chronic Healthcare Needs**: Assistance with ongoing healthcare management
- **Q45 - Individual's Injuries**: Personal injury patterns
- **Q46 - Falls**: Fall-related concerns
- **Q47 - Physician Visits/Nursing Services**: Healthcare service utilization
- **Q48 - Emergency Room Visits**: Emergency healthcare utilization
- **Q49 - Hospital Admissions**: Inpatient healthcare utilization
- **Q50 - Days Missed**: Activity missed due to illness

1.2.1.4 Composite and Additional Variables

The QSI generates several composite scores and includes demographic variables:

- **FSum**: Functional status raw score (sum of Q14-Q24, range 0-44)
- **BSum**: Behavioral status raw score (sum of Q25-Q30, range 0-24)
- **PSum**: Physical status raw score (sum of Q32-Q50, range 0-76)
- **Living Setting**: Six categorical levels ranging from family home to intensive residential care
- **Age Groups**: Multiple categorical classifications (3-20, 21-30, 31+ years)

1.2.2 Structural Inconsistencies in the QSI Assessment Instrument

The QSI exhibits several fundamental design inconsistencies that compromise its reliability as a standardized assessment tool. These include non-uniform scaling systems, unvalidated question exclusions, inconsistent temporal frameworks, and ad-hoc scoring rules that violate the instrument's stated ordinal structure.

Binary vs. Ordinal Scale Inconsistency (Q43) Question 43 (Treatment/physician prescribed) employs a binary scale (0 or 4 only) while all other QSI questions utilize a consistent 5-point ordinal scale (0-4). The standard QSI scaling pattern follows: 0 = none, 1 = minimal, 2 = moderate, 3 = frequent/planned, 4 = intensive. However, Q43 deviates from this structure with only two possible values: 0 = no physician-prescribed procedures required, 4 = requires physician-prescribed procedures carried out by a licensed nurse. This anomaly eliminates intermediate levels 1, 2, and 3, breaking the uniform scaling structure and potentially creating statistical modeling complications due to the bimodal distribution.

Inconsistent Temporal Assessment Frameworks The questionnaire employs multiple, incompatible time frames across different assessment domains without clear justification for the temporal variations. Behavioral interventions are assessed over the "past 12 months," emergency room visits use a "last year" timeframe, hospital admissions reference the "last six months," medication changes examine the "past year," while functional abilities assess "current status." Some items fail to specify any temporal framework entirely. This temporal inconsistency complicates data interpretation and may introduce systematic bias when comparing support needs across different assessment domains.

Special Scoring Rules Violating Ordinal Structure Several questions employ automatic scoring rules that bypass the standard 0-4 ordinal scale, creating methodological inconsistencies. Q43 mandates an "automatic score of '4' if physician-prescribed procedures are required," while Q36 includes a special provision that "anyone on Reglan/Metoclopramide, regardless of the reason, has this rating" of 4. These categorical override rules violate the ordinal measurement principles underlying the assessment instrument and may introduce artificial ceiling effects that distort the distribution of scores and compromise statistical modeling assumptions.

Version Control and Documentation Issues The questionnaire exhibits evidence of poor version control with conflicting information about revision dates, effective dates, and rule references. The document simultaneously references Version 4.0 as effective 2-15-08 and revised 5-21-15, while mentioning earlier versions with different scaling systems where "Level 5 that is now identical to Level 4." Rule numbers and revision protocols appear inconsistent across different sections of the documentation. This suggests inadequate document management and quality assurance procedures that could lead to implementation inconsistencies across different assessment sites or time periods.

Chapter 2

Previous Algorithm

2.1 Previous Algorithm

The current algorithm, designated as Model 5b, operates as a multiple linear regression model that calculates individual budget allocations based on a square-root transformation of fiscal year 2013-14 claims data. This approach incorporates 22 independent variables spanning living settings, age categories, and Questionnaire for Situational Information (QSI) assessment scores that evaluate behavioral, functional, and physical support needs. While the algorithm achieves an R-squared value of 0.7998, explaining approximately 80% of expenditure variation, this statistical performance comes with significant methodological concerns, including the removal of 9.40% of cases as outliers and reliance on data that is now over a decade old.

The temporal disconnect between the algorithm's 2013-14 data foundation and current service delivery realities represents perhaps the most immediate concern. Over the intervening decade, disability services have experienced significant evolution in cost structures, service delivery models, demographic patterns, and regulatory requirements. The algorithm's inability to reflect these changes compromises its predictive validity and creates systematic biases that may disadvantage certain populations or service categories.

Beyond data currency issues, the algorithm's statistical architecture raises fundamental questions about its alignment with person-centered planning principles. The current approach prioritizes actuarial prediction based on historical patterns rather than incorporating individual preferences, goals, and strengths that form the cornerstone of person-centered planning. This disconnect between statistical methodology and philosophical foundation creates a system that may achieve statistical significance while failing to serve the individualization requirements that define quality disability services.

The outlier management approach presents additional concerns about the algorithm's ability to serve the full spectrum of disability support needs. The requirement to remove nearly 10% of cases to achieve acceptable statistical fit suggests fundamental limitations in the model's capacity to accommodate complex or atypical support requirements. This exclusion rate is particularly concerning given that individuals with the most intensive or unique needs may be precisely those most dependent on accurate algorithmic predictions for service access.

Variable validation limitations further compromise the algorithm's comprehensiveness and potential compliance with statutory requirements. The exclusion of QSI questions 8 through 13 due to validation concerns creates gaps in needs assessment that may conflict with requirements for thorough evaluation of individual characteristics and support needs. This limitation reflects broader challenges in balancing statistical rigor with comprehensive needs assessment in algorithmic systems.

2.2 Statistical Methods Analysis

2.2.1 Overview of Statistical Framework

The documentation of Model 5b presents the statistical methods employed in developing the Florida APD's iBudget Algorithm. This section examines the technical approaches used for multiple linear regression modeling with transformations, model selection techniques, and outlier detection methods applied to predict APD consumers' FY 2013-2014 expenditures.

2.2.2 Multiple Linear Regression Foundation

The statistical framework builds upon linear regression as the primary analytical method for modeling relationships between dependent and independent variables. The study defines:

- **Dependent variable**: APD consumers' FY 2013-2014 expenditures

- **Independent variables**: consumers' age, living setting status, individual characteristics and support needs specified in QSI assessments

The classical multiple linear regression model is specified as:

$$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \cdots + \beta_p x_{pi} + \varepsilon_i, \quad i = 1, 2, \ldots, n \tag{2.1}$$

where y_i represents the dependent variable, $\{x_{1i}, x_{2i}, \ldots, x_{pi}\}$ are independent variables or predictors, β_0 is the intercept, $\{\beta_0, \beta_1, \ldots, \beta_p\}$ are unknown coefficients, and $\{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\}$ are random error terms.

2.2.3 Statistical Assumptions and Limitations

The regression framework requires three critical assumptions for the random error terms:

1. Each term ε_i follows a normal distribution
2. Error terms $\{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\}$ are mutually independent
3. Each term ε_i has constant variance σ^2 (homoscedasticity)

These assumptions present immediate challenges when applied to disability expenditure data, which typically exhibits high variability and non-normal distributions due to the diverse and individualized nature of support needs.

2.2.4 Box-Cox Power Transformation

To address distributional concerns, the methodology employs Box-Cox power transformation to normalize the response variable. The transformation is defined as:

$$z_i^{(\lambda)} = \begin{cases} \frac{y_i^\lambda - 1}{\lambda \cdot GM(y)^{\lambda-1}} & \text{if } \lambda \neq 0 \\ GM(y) \cdot \ln(y_i) & \text{if } \lambda = 0 \end{cases} \tag{2.2}$$

where $GM(y) = \left[\prod_{i=1}^{n} y_i\right]^{1/n}$ represents the geometric mean of observations. The scale adjustment by $GM(y)$ ensures unit comparability across different transformation values.

The optimal transformation parameter λ is selected to minimize the Residual Sum of Squares:

$$RSS(\lambda) = \sum_{i=1}^{n} \left(\hat{\varepsilon}_i^{(\lambda)}\right)^2 \tag{2.3}$$

In practice, $RSS(\lambda)$ is evaluated for discrete values:
$\lambda \in \{-3, -2.5, -2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, 2.5, 3\}$.

2.2.5 Model Selection via Bayesian Information Criterion

The methodology employs the Bayesian Information Criterion (SBC) for model selection among 125 candidate independent variables. This approach aims to identify variables with significant predictive power while maintaining model parsimony and avoiding overfitting.

The SBC framework compares multiple candidate models to select the configuration with optimal predictive capability for individual budget allocation. However, this statistical optimization approach prioritizes mathematical fit over substantive considerations of individual needs assessment and person-centered planning principles.

2.2.6 Methodological Concerns

The statistical framework reveals several fundamental limitations that impact the algorithm's suitability for person-centered disability services:

2.2.6.1 Outlier Management

The requirement to remove 9.40% of cases to achieve acceptable statistical fit indicates fundamental model limitations in accommodating diverse support needs. This exclusion rate suggests the methodology cannot adequately serve individuals with complex or atypical requirements—precisely those who may most depend on accurate algorithmic predictions.

2.2.6.2 Temporal Validity

The reliance on FY 2013-2014 data for model development creates a significant temporal disconnect with current service delivery realities. The statistical framework lacks mechanisms for updating or recalibrating the model to reflect evolving cost structures, service models, or demographic patterns.

2.2.6.3 Person-Centered Alignment

The emphasis on actuarial prediction based on historical patterns conflicts with person-centered planning principles that prioritize individual preferences, goals, and strengths. The statistical methodology treats individuals as data points to be fitted to historical patterns rather than unique persons with individualized support requirements.

2.2.7 Implementation Implications

The statistical methods underlying Model 5b demonstrate technical competency within traditional regression frameworks while revealing fundamental misalignment with contemporary disability services principles. The methodology's focus on statistical optimization may achieve mathematical significance while failing to serve the individualization and person-centered requirements that define quality disability services.

The documented approach establishes that while the statistical framework follows accepted practices for regression modeling, its application to disability budget allocation raises significant concerns about equity, individualization, and compliance with person-centered planning requirements. These methodological limitations provide important context for evaluating the algorithm's overall suitability for Florida's disability services system.

2.3 Critical Deficiencies in Model 5b

Questions Q8, Q9, Q12, and Q13 were systematically excluded from statistical modeling because "items were not validated and the reliability of these items was not examined." This represents a fundamental design flaw where questions addressing life changes and community inclusion were incorporated into the instrument without proper psychometric validation. The exclusion of these variables reduced the total usable predictors from 125 to a smaller subset, eliminating potentially valuable contextual information about life transitions and community participation that could influence support needs. This suggests inadequate instrument development protocols and quality control procedures.

The statistical analysis revealed multiple fundamental deficiencies that compromise the validity and utility of the proposed linear regression models for resource allocation.

2.3.0.1 Counter-Intuitive Negative Coefficients

The most egregious deficiency involved negative coefficients for the functional status sum (FSum) and physical status sum (PSum) variables in Model 5a1. These negative coefficients mathematically implied that individuals with greater functional or physical impairments would receive *less* funding, directly contradicting the logical expectation that higher support needs should correspond to increased resource allocation. This fundamental violation of face validity forced the removal of these theoretically important variables from subsequent models, eliminating key predictors that should logically drive resource allocation decisions.

2.3.0.2 Widespread Statistical Insignificance

Multiple predictor variables demonstrated non-significant relationships with the outcome variable, including disability type categories, individual QSI items, and interaction terms. For example, Q24 (evacuation ability) became non-significant (p-value = 0.53) after removing FSum and PSum from the model. Many disability type variables showed coefficients that were not statistically different from zero, despite their theoretical relevance to support needs. This pattern of insignificance suggests either inadequate model specification or fundamental measurement issues in the predictor variables.

2.3.0.3 Violation of Distributional Assumptions

Residual diagnostic analysis revealed persistent deviations from normality assumptions even after square-root transformation of the dependent variable. The Q-Q normal plots demonstrated heavy tails inconsistent with the normal distribution required for valid linear regression inference. The diagnostic plots showed that "the distribution is still away from the normal distribution in the two tails," indicating that standard linear regression assumptions were not met, potentially invalidating hypothesis tests and confidence intervals.

2.3.0.4 Excessive Outlier Exclusion

The final recommended Model 5b required exclusion of 9.40% of cases (2,410 consumers) as outliers—an extraordinarily high proportion suggesting either systematic data quality issues or fundamental model misspecification. Removing nearly one in ten cases raises serious concerns about model generalizability and may indicate that the linear modeling approach is fundamentally inappropriate for this data structure. Such extensive outlier removal suggests the presence of unmodeled nonlinear relationships or heteroscedasticity that the current approach cannot accommodate.

2.3.0.5 Limited Construct Validity

Primary, secondary, and other disability type variables were ultimately excluded from the final model because they were "not statistically predictive for the response variable." The analysis noted that "estimated coefficients for some categories of the three variables are negative and/or the estimated coefficients are not statistically different from zero." This represents a critical failure of construct validity, as disability type should logically influence support needs and resource requirements.

2.3.0.6 Validation and Reliability Gaps

The exclusion of questions Q8, Q9, Q12, and Q13 due to lack of validation represents a significant methodological weakness. These items address life changes and community inclusion—factors that could substantially influence support needs. The systematic exclusion of unvalidated items, while methodologically sound, highlights the incomplete development of the assessment instrument.

2.4 Model 5b Implementation and Testing Framework

2.4.1 Implementation Overview

The Model 5b algorithm has been implemented in Python as `model5b.py`, providing a complete computational framework for budget prediction based on the statistical methodology described in Section II. This implementation translates the regression coefficients from Table 4 of the UpdateStatisticalModelsiBudget document into a functional prediction system that can process individual QSI assessments and generate budget allocations according to the square-root transformation methodology.

The implementation maintains full fidelity to the original statistical model, including all 22 independent variables, interaction terms, and the critical square-root transformation that enables the algorithm to achieve its documented R-squared value of 0.7998. The program architecture emphasizes transparency, validation, and reproducibility, ensuring that predictions can be traced through each computational step.

2.4.2 Program Execution

2.4.2.1 System Requirements

The implementation requires Python 3.6 or higher with standard library modules only. No external dependencies are required, ensuring compatibility across diverse computing environments. The program consists of two primary files:

- `model5b.py` - Complete Model 5b implementation
- `QSI-unit-test1.json` - Comprehensive test dataset

2.4.2.2 Execution Instructions

To execute the Model 5b test program, ensure both files are located in the same directory and run the following command:

```
python model5b.py
```

The program automatically loads the test dataset, processes all test cases through the Model 5b algorithm, and generates comprehensive output including individual predictions, summary statistics, and model performance metrics. No command-line arguments or configuration files are required for basic operation.

2.4.2.3 Expected Output Structure

Program execution produces structured output organized into several sections:

1. **Loading Confirmation** - Verification of test data file access and case count
2. **Individual Predictions** - Detailed results for each test case including:
   - Individual identifier and demographic information
   - Living setting and age group classification

- Predicted budget amount in dollars
- Square-root scale intermediate calculation

3. **Summary Statistics** - Aggregate analysis including:
   - Count of successful predictions
   - Average, minimum, and maximum predicted budgets
   - Distribution characteristics across test cases
4. **Model Information** - Technical specifications including R-squared, outlier removal percentage, and statistical performance metrics

2.4.3 Test Dataset Specification

2.4.3.1 Dataset Structure

The `QSI-unit-test1.json` file contains a comprehensive test dataset designed to validate Model 5b implementation across the full spectrum of disability support scenarios. The dataset employs JSON formatting for platform independence and includes both test cases and extensive metadata documentation.

The file structure consists of four primary components:

- **Metadata Section** - Dataset description, version information, and data source documentation
- **Test Cases Array** - Twelve individual assessment records representing diverse support scenarios
- **Variable Definitions** - Complete specification of all input variables and their valid ranges
- **Model Information** - Technical parameters and performance characteristics of Model 5b

2.4.3.2 Test Case Coverage

The dataset includes twelve carefully constructed test cases that systematically cover the parameter space defined by Model 5b variables:

Living Setting Distribution:

- Family Home (FH): 4 cases representing the reference level
- Independent Living & Supported Living (ILSL): 3 cases with varying support intensities
- Residential Habilitation Standard (RH1): 2 cases including standard residential care
- Residential Habilitation Behavior Focus (RH2): 1 case with behavioral specialization
- Residential Habilitation Intensive Behavior (RH3): 1 case with intensive behavioral support
- Residential Habilitation Special Medical (RH4): 1 case with complex medical needs

Age Group Representation:

- Under 21 (reference level): 2 cases representing adolescent populations
- Age 21-30: 4 cases covering young adult transition period
- Age 31+: 6 cases spanning adult and senior populations (ages 31-67)

Support Need Variation: The test cases systematically vary across support intensity levels:

- **Minimal Support** (TEST007): Teenager with limited intervention requirements
- **Moderate Support** (TEST001, TEST008, TEST011): Individuals with balanced functional and behavioral needs
- **High Support** (TEST003, TEST010): Adults requiring intensive assistance across multiple domains
- **Severe Support** (TEST005, TEST006): Complex cases with maximum intervention requirements

2.4.3.3 Variable Validation Framework

Each test case includes validation of all required Model 5b input variables:

Demographic Variables:

- `living_setting` - Categorical variable with six valid levels
- `age` - Continuous variable determining age group classification

QSI Sum Scores:

- `bsum` - Behavioral status sum (0-24 range)
- `fsum` - Functional status sum (0-44 range)
- `psum` - Physical status sum (0-76 range)

Individual QSI Questions: Ten specific questions (Q16, Q18, Q20, Q21, Q23, Q28, Q33, Q34, Q36, Q43) each scored on 0-4 scales representing:

- Functional domains: Eating, transfers, hygiene, dressing, self-protection
- Behavioral domains: Sexual behavior, aggression, restraint use
- Medical domains: Psychotropic medications, physician-prescribed treatments

2.4.3.4 Data Integrity and Realism

The test dataset maintains realistic relationships between variables, ensuring that sum scores align with individual question responses and that support needs correspond appropriately to living settings. For example, individuals in RH3 and RH4 settings demonstrate correspondingly higher QSI scores, while those in family homes show more variable support profiles reflecting diverse family capacity.

The dataset also incorporates edge cases and boundary conditions to test algorithm robustness, including individuals with minimal support needs, maximum scoring scenarios, and atypical combinations that may occur in real-world assessments.

2.4.4 Implementation Validation

2.4.4.1 Computational Accuracy

The Python implementation reproduces the exact coefficient structure documented in the UpdateStatisticalModelsiBudget report, ensuring mathematical fidelity to the research methodology. All regression coefficients, interaction terms, and transformation procedures match the specifications in Table 4, enabling direct comparison with the original statistical analysis.

The program includes comprehensive input validation to prevent computational errors and ensure that all QSI scores fall within their defined ranges. Invalid inputs generate descriptive error messages identifying the specific validation failure, supporting quality assurance in operational deployment.

2.4.4.2 Transparency and Traceability

Each prediction includes detailed intermediate calculations, allowing users to trace the contribution of individual variables to the final budget prediction. The output format displays the square-root scale calculation before transformation to dollars, enabling verification of the mathematical procedures against manual calculations.

The implementation also provides complete documentation of which coefficients were applied for each individual, including living setting classification, age group determination, and specific QSI question contributions. This transparency supports both technical validation and policy analysis of algorithmic decision-making.

2.4.5 Testing Framework Applications

This implementation and testing framework serves multiple analytical purposes beyond basic algorithm validation:

Policy Analysis: The comprehensive test cases enable examination of how different policy scenarios (changes in living setting availability, age group definitions, or QSI scoring protocols) would affect budget predictions across diverse populations.

Equity Assessment: The systematic coverage of demographic and support need combinations facilitates analysis of potential disparities in budget allocation across different population subgroups.

Sensitivity Analysis: The modular implementation structure supports investigation of how changes to individual coefficients or variable definitions would propagate through the prediction system.

Validation Studies: The test framework provides a standardized basis for comparing alternative algorithmic approaches or validating implementation accuracy across different programming environments.

The combination of comprehensive implementation and systematic test data establishes a robust foundation for ongoing analysis and refinement of the Florida APD iBudget algorithm methodology.

2.4.6 Source code & Output

Python Implementation:

```
#!/usr/bin/env python3
"""
Model 5b Implementation for Florida APD iBudget Algorithm

```

```python
This module implements the final Model 5b from the UpdateStatisticalModelsiBudget document.
The model uses square-root transformation and multiple linear regression to predict
individual budget allocations based on QSI assessment data.

Model 5b uses the following coefficients (from Table 4):
- Intercept: 27.5720
- Living Settings: ILSL (35.8220), RH1 (90.6294), RH2 (131.7576), RH3 (209.4558), RH4 (267.0995)
- Age Groups: Age21-30 (47.8473), Age31+ (48.9634)
- Behavioral/Functional Sums: BSum (0.4954), FHFSum (0.6349), SLFSum (2.0529), SLBSum (1.4501)
- QSI Questions: Q16 (2.4984), Q18 (5.8537), Q20 (2.6772), Q21 (2.7878), Q23 (6.3555),
                 Q28 (2.2803), Q33 (1.2233), Q34 (2.1764), Q36 (2.6734), Q43 (1.9304)

Reference levels (coefficients = 0):
- Living Setting: Family Home (FH)
- Age: Under 21
"""

import json
import math
import sys
from typing import Dict, Any, Optional
from dataclasses import dataclass
from datetime import datetime


@dataclass
class Model5bCoefficients:
    """Model 5b regression coefficients from the final algorithm."""

    # Intercept
    intercept: float = 27.5720

    # Living Setting coefficients (FH is reference level with 0)
    live_ilsl: float = 35.8220  # Independent Living & Supported Living
    live_rh1: float = 90.6294   # Residential Habilitation, Standard and Live In
    live_rh2: float = 131.7576  # Residential Habilitation, Behavior Focus
    live_rh3: float = 209.4558  # Residential Habilitation, Intensive Behavior
    live_rh4: float = 267.0995  # Residential Habilitation, CTEP and Special Medical Home Care

    # Age Group coefficients (Under 21 is reference level with 0)
    age_21_30: float = 47.8473  # Age 21-30
    age_31_plus: float = 48.9634  # Age 31+

    # Sum and interaction coefficients
    bsum: float = 0.4954        # Behavioral status sum score
```

```python
    fhfsum: float = 0.6349      # Family Home by Functional status interaction
    slfsum: float = 2.0529      # ILSL by Functional status interaction
    slbsum: float = 1.4501      # ILSL by Behavioral status interaction

    # QSI Question coefficients
    q16: float = 2.4984         # Eating
    q18: float = 5.8537         # Transfers
    q20: float = 2.6772         # Hygiene
    q21: float = 2.7878         # Dressing
    q23: float = 6.3555         # Self-protection
    q28: float = 2.2803         # Inappropriate Sexual Behavior
    q33: float = 1.2233         # Injury to Person Caused by Aggression
    q34: float = 2.1764         # Use of Mechanical Restraints
    q36: float = 2.6734         # Use of Psychotropic Medications
    q43: float = 1.9304         # Treatment (Physician Prescribed)


class TeeOutput:
    """
    Helper class to write output to both console and file simultaneously.
    """
    def __init__(self, filename):
        self.terminal = sys.stdout
        self.log = open(filename, 'w')

    def write(self, message):
        self.terminal.write(message)
        self.log.write(message)

    def flush(self):
        self.terminal.flush()
        self.log.flush()

    def close(self):
        self.log.close()


class Model5b:
    """
    Implementation of Model 5b for Florida APD iBudget Algorithm.

    This class implements the final regression model with square-root transformation
    that achieved R-squared = 0.7998 after removing 9.40% outliers.
    """

    def __init__(self):
        self.coefficients = Model5bCoefficients()
        self.model_info = {
            "name": "Model 5b",
            "r_squared": 0.7998,
            "outliers_removed": 0.094,
            "residual_standard_error": 30.82,
            "degrees_of_freedom": 23193,
```

```python
            "f_statistic": 4412,
            "p_value": "< 2.2e-16"
        }

    def validate_input(self, qsi_data: Dict[str, Any]) -> Dict[str, Any]:
        """
        Validate and normalize QSI input data.

        Args:
            qsi_data: Dictionary containing QSI assessment data

        Returns:
            Validated and normalized data dictionary

        Raises:
            ValueError: If required fields are missing or invalid
        """
        required_fields = ['living_setting', 'age', 'bsum', 'fsum', 'psum']
        qsi_questions = ['Q16', 'Q18', 'Q20', 'Q21', 'Q23', 'Q28', 'Q33', 'Q34', 'Q36', 'Q43']

        # Check required fields
        for field in required_fields:
            if field not in qsi_data:
                raise ValueError(f"Missing required field: {field}")

        # Check QSI questions
        for q in qsi_questions:
            if q not in qsi_data:
                raise ValueError(f"Missing required QSI question: {q}")

        # Validate living setting
        valid_living_settings = ['FH', 'ILSL', 'RH1', 'RH2', 'RH3', 'RH4']
        if qsi_data['living_setting'] not in valid_living_settings:
            raise ValueError(f"Invalid living_setting. Must be one of: {valid_living_settings}")

        # Validate age
        if not isinstance(qsi_data['age'], (int, float)) or qsi_data['age'] < 0:
            raise ValueError("Age must be a non-negative number")

        # Validate QSI scores (0-4 scale)
        for q in qsi_questions:
            score = qsi_data[q]
            if not isinstance(score, (int, float)) or score < 0 or score > 4:
                raise ValueError(f"{q} must be between 0 and 4, got: {score}")

        # Validate sum scores
        if not (0 <= qsi_data['bsum'] <= 24):  # 6 questions by 4 max score
```

```python
            raise ValueError("BSum must be between 0 and 24")
        if not (0 <= qsi_data['fsum'] <= 44):  # 11 questions by 4 max score
            raise ValueError("FSum must be between 0 and 44")
        if not (0 <= qsi_data['psum'] <= 76):  # 19 questions by 4 max score
            raise ValueError("PSum must be between 0 and 76")

        return qsi_data

    def calculate_interaction_terms(self, qsi_data: Dict[str, Any]) -> Dict[str, float]:
        """
        Calculate interaction terms between living setting and sum scores.

        Args:
            qsi_data: Validated QSI data

        Returns:
            Dictionary containing interaction term values
        """
        living_setting = qsi_data['living_setting']
        fsum = qsi_data['fsum']
        bsum = qsi_data['bsum']

        interactions = {
            'fhfsum': 0,  # Family Home by Functional Sum
            'slfsum': 0,  # ILSL by Functional Sum
            'slbsum': 0   # ILSL by Behavioral Sum
        }

        if living_setting == 'FH':
            interactions['fhfsum'] = fsum
        elif living_setting == 'ILSL':
            interactions['slfsum'] = fsum
            interactions['slbsum'] = bsum

        return interactions

    def predict_square_root_scale(self, qsi_data: Dict[str, Any]) -> float:
        """
        Calculate prediction in square-root scale using Model 5b coefficients.

        Args:
            qsi_data: Validated QSI assessment data

        Returns:
            Predicted value in square-root scale
        """
        # Start with intercept
        prediction = self.coefficients.intercept

```

```
        # Add living setting effects (FH is reference level)
        living_setting = qsi_data['living_setting']
        if living_setting == 'ILSL':
            prediction += self.coefficients.live_ilsl
        elif living_setting == 'RH1':
            prediction += self.coefficients.live_rh1
        elif living_setting == 'RH2':
            prediction += self.coefficients.live_rh2
        elif living_setting == 'RH3':
            prediction += self.coefficients.live_rh3
        elif living_setting == 'RH4':
            prediction += self.coefficients.live_rh4
        # FH has coefficient 0 (reference level)

        # Add age effects (Under 21 is reference level)
        age = qsi_data['age']
        if 21 <= age <= 30:
            prediction += self.coefficients.age_21_30
        elif age >= 31:
            prediction += self.coefficients.age_31_plus
        # Under 21 has coefficient 0 (reference level)

        # Add behavioral sum effect
        prediction += self.coefficients.bsum * qsi_data['bsum']

        # Add interaction terms
        interactions = self.calculate_interaction_terms(qsi_data)
        prediction += self.coefficients.fhfsum * interactions['fhfsum']
        prediction += self.coefficients.slfsum * interactions['slfsum']
        prediction += self.coefficients.slbsum * interactions['slbsum']

        # Add QSI question effects
        prediction += self.coefficients.q16 * qsi_data['Q16']
        prediction += self.coefficients.q18 * qsi_data['Q18']
        prediction += self.coefficients.q20 * qsi_data['Q20']
        prediction += self.coefficients.q21 * qsi_data['Q21']
        prediction += self.coefficients.q23 * qsi_data['Q23']
        prediction += self.coefficients.q28 * qsi_data['Q28']
        prediction += self.coefficients.q33 * qsi_data['Q33']
        prediction += self.coefficients.q34 * qsi_data['Q34']
        prediction += self.coefficients.q36 * qsi_data['Q36']
        prediction += self.coefficients.q43 * qsi_data['Q43']

        return prediction

    def predict_budget(self, qsi_data: Dict[str, Any]) -> Dict[str, Any]:
        """
        Predict individual budget allocation using Model 5b.

        Args:
            qsi_data: QSI assessment data

        Returns:
            Dictionary containing prediction results
        """
```

```
        # Validate input
        validated_data = self.validate_input(qsi_data)

        # Calculate prediction in square-root scale
        sqrt_prediction = self.predict_square_root_scale(validated_data)

        # Transform back to dollar scale by squaring
        budget_prediction = sqrt_prediction ** 2

        # Calculate interaction terms for transparency
        interactions = self.calculate_interaction_terms(validated_data)

        return {
            'predicted_budget': round(budget_prediction, 2),
            'sqrt_scale_prediction': round(sqrt_prediction, 4),
            'model_info': self.model_info,
            'input_data': validated_data,
            'interaction_terms': interactions,
            'coefficients_used': {
                'living_setting': validated_data['living_setting'],
                'age_group': self._get_age_group(validated_data['age']),
                'qsi_scores': {q: validated_data[q] for q in ['Q16', 'Q18', 'Q20', 'Q21', 'Q23', 'Q28', 'Q33', 'Q34', 'Q36', 'Q43']}
            }
        }

    def _get_age_group(self, age: float) -> str:
        """Helper function to determine age group."""
        if age < 21:
            return "Under 21 (reference)"
        elif 21 <= age <= 30:
            return "21-30"
        else:
            return "31+"

    def predict_batch(self, qsi_data_list: list) -> list:
        """
        Predict budgets for multiple individuals.

        Args:
            qsi_data_list: List of QSI assessment data dictionaries

        Returns:
            List of prediction results
        """
        results = []
        for i, qsi_data in enumerate(qsi_data_list):
            try:
                result = self.predict_budget(qsi_data)
                result['record_index'] = i
                results.append(result)
            except Exception as e:
                results.append({
                    'record_index': i,
```

```python
                    'error': str(e),
                    'input_data': qsi_data
                })
        return results


def main():
    """
    Main function to test Model 5b implementation using QSI-unit-test1.
        json
    Output is written to both console and model5b_output.txt
    """
    # Set up dual output to console and file
    output_filename = 'model5b_output.txt'
    tee = TeeOutput(output_filename)
    original_stdout = sys.stdout
    sys.stdout = tee

    try:
        # Add timestamp to output
        print("Florida APD iBudget Algorithm - Model 5b Implementation")
        print("=" * 60)
        print(f"Execution Date/Time: {datetime.now().strftime('%Y-%m-%d %
            H:%M:%S')}")
        print(f"Output File: {output_filename}")
        print("=" * 60)

        # Initialize the model
        model = Model5b()

        try:
            # Load test data
            with open('QSI-unit-test1.json', 'r') as f:
                test_data = json.load(f)

            print(f"\nLoaded {len(test_data['test_cases'])} test cases
                from QSI-unit-test1.json")
            print(f"Test data description: {test_data['description']}")

            # Run predictions
            results = model.predict_batch(test_data['test_cases'])

            # Display results
            print(f"\nModel 5b Prediction Results:")
            print("-" * 40)

            for result in results:
                if 'error' in result:
                    print(f"Record {result['record_index']}: ERROR - {
                        result['error']}")
                else:
                    data = result['input_data']
                    print(f"\nRecord {result['record_index']}:")
                    print(f"  Individual: {data.get('individual_id', 'N/A
                        ')}")
```

```python
                    print(f"  Living Setting: {data['living_setting']}")
                    print(f"  Age: {data['age']} ({result['coefficients_used']['age_group']})")
                    print(f"  Predicted Budget: ${result['predicted_budget']:,.2f}")
                    print(f"  Square-root Scale: {result['sqrt_scale_prediction']}")

            # Summary statistics
            successful_predictions = [r for r in results if 'error' not in r]
            if successful_predictions:
                budgets = [r['predicted_budget'] for r in successful_predictions]
                print(f"\nSummary Statistics:")
                print(f"  Successful predictions: {len(successful_predictions)}")
                print(f"  Average predicted budget: ${sum(budgets)/len(budgets):,.2f}")
                print(f"  Minimum predicted budget: ${min(budgets):,.2f}")
                print(f"  Maximum predicted budget: ${max(budgets):,.2f}")

            print(f"\nModel Information:")
            print(f"  R-squared: {model.model_info['r_squared']}")
            print(f"  Outliers removed: {model.model_info['outliers_removed']*100:.1f}%")
            print(f"  Residual standard error: {model.model_info['residual_standard_error']}")

            print(f"\n" + "=" * 60)
            print(f"Execution completed successfully.")
            print(f"Results saved to: {output_filename}")

        except FileNotFoundError:
            print("\nError: QSI-unit-test1.json not found.")
            print("Please ensure the test data file is in the same directory.")
        except json.JSONDecodeError as e:
            print(f"\nError reading JSON file: {e}")
        except Exception as e:
            print(f"\nUnexpected error: {e}")

    finally:
        # Restore original stdout and close file
        sys.stdout = original_stdout
        tee.close()
        print(f"\nOutput has been written to both console and {output_filename}")


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

Output:

```
Florida APD iBudget Algorithm - Model 5b Implementation
============================================================
Execution Date/Time: 2025-09-05 17:40:02
Output File: model5b_output.txt
============================================================

Loaded 12 test cases from QSI-unit-test1.json
Test data description: Unit test data for Model 5b iBudget Algorithm
    based on QSI assessments

Model 5b Prediction Results:
----------------------------------------

Record 0:
  Individual: TEST001
  Living Setting: ILSL
  Age: 25 (21-30)
  Predicted Budget: $42,960.19
  Square-root Scale: 207.2684

Record 1:
  Individual: TEST002
  Living Setting: FH
  Age: 19 (Under 21 (reference))
  Predicted Budget: $4,709.91
  Square-root Scale: 68.6288

Record 2:
  Individual: TEST003
  Living Setting: RH1
  Age: 35 (31+)
  Predicted Budget: $69,109.36
  Square-root Scale: 262.8866

Record 3:
  Individual: TEST004
  Living Setting: RH2
  Age: 28 (21-30)
  Predicted Budget: $96,521.94
  Square-root Scale: 310.6798

Record 4:
  Individual: TEST005
  Living Setting: RH3
  Age: 42 (31+)
  Predicted Budget: $169,866.55
  Square-root Scale: 412.1487

Record 5:
  Individual: TEST006
  Living Setting: RH4
  Age: 55 (31+)
  Predicted Budget: $215,268.90
  Square-root Scale: 463.9708
```

```

Record 6:
  Individual: TEST007
  Living Setting: FH
  Age: 16 (Under 21 (reference))
  Predicted Budget: $3,662.69
  Square-root Scale: 60.5202

Record 7:
  Individual: TEST008
  Living Setting: ILSL
  Age: 31 (31+)
  Predicted Budget: $56,536.28
  Square-root Scale: 237.7736

Record 8:
  Individual: TEST009
  Living Setting: FH
  Age: 24 (21-30)
  Predicted Budget: $19,418.14
  Square-root Scale: 139.349

Record 9:
  Individual: TEST010
  Living Setting: RH1
  Age: 67 (31+)
  Predicted Budget: $68,804.18
  Square-root Scale: 262.3055

Record 10:
  Individual: TEST011
  Living Setting: ILSL
  Age: 29 (21-30)
  Predicted Budget: $40,415.35
  Square-root Scale: 201.0357

Record 11:
  Individual: TEST012
  Living Setting: FH
  Age: 38 (31+)
  Predicted Budget: $27,697.45
  Square-root Scale: 166.4255

Summary Statistics:
  Successful predictions: 12
  Average predicted budget: $67,914.24
  Minimum predicted budget: $3,662.69
  Maximum predicted budget: $215,268.90

Model Information:
  R-squared: 0.7998
  Outliers removed: 9.4%
  Residual standard error: 30.82

============================================================
```

```
Execution completed successfully.
Results saved to: model5b_output.txt
```

Chapter 3

Alternative Algorithms

3.1 Alternative Algorithms

The proposed alternative algorithms represent six distinct categories of quantitative approaches, each designed to address specific limitations in the current system while advancing compliance with person-centered planning requirements. Enhanced linear regression approaches focus on updating data sources, improving outlier management, and expanding variable inclusion while maintaining the interpretability advantages of traditional statistical methods. Machine learning ensemble approaches leverage advanced algorithms to capture non-linear relationships and complex interactions while providing transparency through feature importance analysis and prediction explanation techniques.

Hybrid statistical-clinical approaches represent a fundamental reconceptualization of algorithmic design, combining statistical prediction with explicit mechanisms for incorporating clinical judgment and person-centered planning elements. These approaches acknowledge that purely statistical methods may be insufficient for capturing the full complexity of individual needs and preferences that effective disability services require.

Person-centered optimization approaches directly address compliance requirements by formulating budget allocation as a multi-objective optimization problem that balances statistical accuracy with goal alignment and fairness considerations. These methods represent a paradigm shift from prediction-focused algorithms to optimization-focused systems that explicitly incorporate individual preferences and societal equity concerns into the mathematical formulation.

Modern time-aware approaches address temporal validity concerns through dynamic regression methods that adapt coefficients over time and longitudinal models that track individual trajectories. These approaches recognize that both population-level service patterns and individual needs evolve over time, requiring algorithmic systems that can adapt rather than remain static.

Specialized needs-based approaches acknowledge the heterogeneity within disability populations through mixture models that identify distinct subpopulations and support vector regression methods that can accommodate high-dimensional assessment data and non-linear relationships. These approaches recognize that one-size-fits-all algorithms may be inherently inadequate for serving diverse disability populations with varying support requirements and preferences.

3.1.1 Advanced Mathematical and Statistical Modeling Approaches

Given the identified deficiencies, several advanced modeling approaches could substantially improve the analysis of QSI data while addressing the fundamental limitations of the current linear regression framework.

3.1.1.1 Regularization Methods for High-Dimensional Data

The QSI dataset presents a high-dimensional modeling challenge with 125 potential predictors and complex multicollinearity among related assessment items. Regularization methods provide principled approaches to variable selection and coefficient estimation.

LASSO Regression (L1 Regularization) LASSO regression addresses the variable selection problem through automatic feature selection:

$$\hat{\beta}_{LASSO} = \arg\min_{\beta} \left\{ \|y - X\beta\|_2^2 + \lambda \|\beta\|_1 \right\} \tag{3.1}$$

where λ controls the sparsity penalty, automatically setting irrelevant coefficients to exactly zero. This approach would eliminate the need for ad-hoc variable removal while providing a principled method for identifying the most predictive QSI items.

Ridge Regression (L2 Regularization) Ridge regression addresses multicollinearity among QSI items without variable elimination:

$$\hat{\beta}_{Ridge} = \arg\min_{\beta} \left\{ \|y - X\beta\|_2^2 + \lambda\|\beta\|_2^2 \right\} \tag{3.2}$$

This approach shrinks correlated coefficients toward each other, potentially resolving the negative coefficient problem by stabilizing parameter estimates.

Elastic Net Regularization Elastic Net combines both L1 and L2 penalties to simultaneously address variable selection and multicollinearity:

$$\hat{\beta}_{EN} = \arg\min_{\beta} \left\{ \|y - X\beta\|_2^2 + \lambda_1\|\beta\|_1 + \lambda_2\|\beta\|_2^2 \right\} \tag{3.3}$$

Given the natural groupings in QSI data, this approach could identify relevant subsets of questions while maintaining stable coefficient estimates.

3.1.1.2 Sparse Estimation Techniques

Adaptive LASSO Adaptive LASSO incorporates data-driven weights to reduce bias in coefficient estimation:

$$\hat{\beta}_{AL} = \arg\min_{\beta} \left\{ \|y - X\beta\|_2^2 + \lambda \sum_{j=1}^{p} w_j|\beta_j| \right\} \tag{3.4}$$

where weights $w_j = 1/|\hat{\beta}_j^{(0)}|^\gamma$ are based on initial consistent estimates, potentially addressing the coefficient sign problems observed in the current models.

Group LASSO Given the natural grouping of QSI items into functional, behavioral, and physical domains, Group LASSO enables selection of entire groups:

$$\hat{\beta}_{GL} = \arg\min_{\beta} \left\{ \|y - X\beta\|_2^2 + \lambda \sum_{g=1}^{G} \sqrt{p_g}\|\beta_g\|_2 \right\} \tag{3.5}$$

This approach could determine whether entire assessment domains should be included or excluded from the allocation algorithm.

3.1.1.3 Robust Regression Approaches

To address the outlier problem without arbitrary data exclusion, robust regression methods provide principled alternatives.

M-Estimation M-estimators minimize robust loss functions:

$$\hat{\beta}_M = \arg\min_{\beta} \sum_{i=1}^{n} \rho\left(\frac{y_i - x_i^T\beta}{\sigma}\right) \tag{3.6}$$

Using Huber or Tukey bisquare loss functions $\rho(\cdot)$ that downweight extreme observations rather than excluding them entirely.

Quantile Regression Given the apparent heteroscedasticity and non-normal residuals, quantile regression models conditional quantiles rather than means:

$$\hat{\beta}_\tau = \arg\min_\beta \sum_{i=1}^{n} \rho_\tau(y_i - x_i^T \beta) \tag{3.7}$$

where $\rho_\tau(u) = u(\tau - \mathbf{1}_{u<0})$ is the quantile loss function. This approach could model different resource allocation patterns across the support needs distribution.

3.1.1.4 Machine Learning Approaches for Nonlinear Relationships

Random Forest Regression Random Forest can capture complex nonlinear relationships and interactions among QSI items:

$$\hat{f}(x) = \frac{1}{B} \sum_{b=1}^{B} T_b(x) \tag{3.8}$$

where T_b represents individual decision trees trained on bootstrap samples. This approach provides variable importance measures and handles interactions naturally.

Gradient Boosting Gradient boosting sequentially builds weak learners to minimize prediction error:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x) \tag{3.9}$$

where h_m minimizes the residual from iteration $m-1$. This approach could identify complex patterns in the QSI data that linear models cannot capture.

3.1.1.5 Ordinal Regression Methods

Given the ordinal nature of QSI responses (0-4 scale), proportional odds models may be more appropriate than treating the data as continuous:

$$\text{logit}(P(Y \leq j|x)) = \alpha_j - x^T \beta \tag{3.10}$$

This approach respects the ordinal structure of the assessment data while potentially improving model fit.

3.1.1.6 Hierarchical and Mixed-Effects Models

To account for potential clustering within service areas or provider organizations:

$$y_{ij} = X_{ij}\beta + Z_{ij}b_i + \epsilon_{ij} \tag{3.11}$$

where $b_i \sim N(0, D)$ represents random effects for cluster i. This approach could account for systematic differences in resource allocation patterns across regions or providers.

3.1.1.7 Ensemble Methods

Model Stacking Stacking combines multiple base models using a meta-learner:

$$\hat{y} = \alpha_0 + \sum_{k=1}^{K} \alpha_k \hat{f}_k(x) \tag{3.12}$$

where $\hat{f}_k(x)$ represents predictions from different modeling approaches, potentially combining the strengths of parametric and nonparametric methods.

Bayesian Model Averaging BMA incorporates model uncertainty into predictions:

$$\hat{y} = \sum_{k=1}^{K} P(M_k|\text{data}) \cdot \hat{f}_k(x) \tag{3.13}$$

providing principled uncertainty quantification for resource allocation decisions.

3.1.2 Recommendations for Model Development

Given the complexity of the QSI data and the fundamental deficiencies in the current approach, we recommend a multi-stage modeling strategy:

1. **Baseline Establishment**: Implement cross-validated elastic net regression as a regularized linear baseline, addressing multicollinearity and variable selection issues.
2. **Nonlinear Enhancement**: Apply gradient boosting to detect and model nonlinear relationships and interactions among QSI variables.
3. **Robustness Testing**: Evaluate quantile regression and robust methods to assess sensitivity to distributional assumptions and outliers.
4. **Ensemble Integration**: Combine multiple approaches using stacking or Bayesian model averaging to leverage the strengths of different methodological frameworks.
5. **Validation Framework**: Implement rigorous cross-validation and holdout testing to ensure model generalizability and prevent overfitting.

This comprehensive approach would address the identified deficiencies while fully utilizing the rich multidimensional structure of the QSI assessment data, providing a more reliable foundation for equitable resource allocation decisions.

3.2 Current Algorithm Analysis

3.2.1 Mathematical Formulation

The current iBudget algorithm (Model 5b) employs a multiple linear regression model with square-root transformation:

$$\sqrt{Y_i} = \beta_0 + \sum_{j=1}^{5} \beta_j^{Live} \cdot Live_{ij} + \sum_{k=1}^{2} \beta_k^{Age} \cdot Age_{ik} + \sum_{l} \beta_l^{QSI} \cdot QSI_{il} + \varepsilon_i \tag{3.14}$$

where:

- Y_i represents FY 2013-14 expenditures for individual i
- $Live_{ij}$ are dummy variables for living settings (Family Home, ILSL, RH1-RH4)
- Age_{ik} are age category indicators (21-30, 31+)
- QSI_{il} are Questionnaire for Situational Information scores
- $\varepsilon_i \sim N(0, \sigma^2)$ are error terms

The final budget allocation is computed as:

$$Budget_i = \left(\sum_j \hat{\beta}_j \cdot X_{ij} \right)^2 \cdot ApportionmentFactor \tag{3.15}$$

3.2.2 Model Performance Metrics

The current algorithm achieves:

$$R^2 = 0.7998 \tag{3.16}$$
$$n_{outliers} = 2,410 \text{ (9.40\% of sample)} \tag{3.17}$$
$$n_{total} = 25,615 \text{ (after outlier removal)} \tag{3.18}$$

3.2.3 Critical Mathematical Limitations

3.2.3.1 Outlier Dependency

The model's performance critically depends on outlier removal:

$$R^2_{full} = 0.7549 \ll R^2_{reduced} = 0.7998 \tag{3.19}$$

This indicates the algorithm fails to capture the full distribution of individual needs, particularly for complex cases.

3.2.3.2 Temporal Validity Issues

Using data from fiscal year 2013-14 introduces significant temporal bias:

$$\hat{\beta}_{2025} \neq \hat{\beta}_{2013-14} \tag{3.20}$$

The assumption of parameter stability over 11+ years is statistically untenable given:

- Service cost inflation: $\Delta Cost \approx 30\%$ over period
- Demographic shifts in disability population
- Changes in service delivery models

3.2.3.3 Transformation Bias

The square-root transformation creates systematic bias:

$$E[Y_i|X_i] \neq E[\hat{Y}_i^2|X_i] \tag{3.21}$$

This Jensen's inequality violation leads to consistent underestimation of high-needs individuals.

3.3 Compliance Analysis with House Bill 1103

3.3.1 Person-Centered Planning Deficiencies

The current algorithm violates HB 1103 person-centered requirements through:

$$Utility_i = f(Needs_i, Demographics_i) \not\supset f(Preferences_i, Goals_i, Strengths_i) \tag{3.22}$$

where the algorithm fails to incorporate individual preferences, goals, and strengths as required by statute.

3.3.2 Data Currency Violations

HB 1103 requires "recent expenditure data," but:

$$Age(Data) = 2025 - 2014 = 11 \text{ years} \gg \text{Acceptable threshold} \tag{3.23}$$

3.4 Proposed Alternative Algorithms

3.4.1 Enhanced Linear Regression Approaches

3.4.1.1 Algorithm A1: Robust Linear Regression

Mathematical Formulation:

$$\hat{\beta}_{robust} = \arg\min_{\beta} \sum_{i=1}^{n} \rho\left(\frac{y_i - x_i^T\beta}{\sigma}\right) \tag{3.24}$$

where $\rho(\cdot)$ is a robust loss function (Huber or Tukey bisquare):

$$\rho_{Huber}(u) = \begin{cases} \frac{1}{2}u^2 & \text{if } |u| \leq c \\ c|u| - \frac{1}{2}c^2 & \text{if } |u| > c \end{cases} \tag{3.25}$$

Python Implementation:

```python
from sklearn.linear_model import HuberRegressor
from sklearn.preprocessing import StandardScaler
import numpy as np

# Robust regression implementation
def robust_ibudget_algorithm(X, y):
    """
    Implements robust linear regression for iBudget allocation

    Args:
        X: Feature matrix (n_samples, n_features)
        y: Target expenditures (n_samples,)

    Returns:
        Trained robust regression model
    """
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

```

```
    # Huber regressor handles outliers without removal
    model = HuberRegressor(epsilon=1.35, alpha=0.0001)
    model.fit(X_scaled, y)

    return model, scaler

# Usage example
model, scaler = robust_ibudget_algorithm(qsi_features, expenditures)
predictions = model.predict(scaler.transform(new_features))
```

3.4.1.2 Algorithm A2: Regularized Regression

Mathematical Formulation:

$$\hat{\beta}_{LASSO} = \arg\min_{\beta} \left\{ \frac{1}{2n} \sum_{i=1}^{n} (y_i - x_i^T \beta)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right\} \tag{3.26}$$

$$\hat{\beta}_{Ridge} = \arg\min_{\beta} \left\{ \frac{1}{2n} \sum_{i=1}^{n} (y_i - x_i^T \beta)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\} \tag{3.27}$$

Python Implementation:

```
from sklearn.linear_model import LassoCV, RidgeCV, ElasticNetCV
from sklearn.model_selection import cross_val_score

def regularized_ibudget_algorithm(X, y, method='elastic'):
    """
    Implements regularized regression for iBudget allocation

    Args:
        X: Feature matrix including all QSI variables
        y: Target expenditures
        method: 'lasso', 'ridge', or 'elastic'

    Returns:
        Optimized regularized model
    """
    if method == 'lasso':
        model = LassoCV(cv=5, random_state=42)
    elif method == 'ridge':
        model = RidgeCV(cv=5)
    else:  # elastic net
        model = ElasticNetCV(cv=5, random_state=42)

    model.fit(X, y)

    # Feature importance for transparency
    importance = np.abs(model.coef_)
    feature_importance = dict(zip(range(len(importance)), importance))

    return model, feature_importance

# Implementation with QSI features
```

```python
model, importance = regularized_ibudget_algorithm(qsi_matrix,
    expenditures)
print(f"Selected features: {np.sum(model.coef_ != 0)} out of {len(model.
    coef_)}")
```

3.4.2 Machine Learning Ensemble Approaches

3.4.2.1 Algorithm B1: Random Forest Regression

Mathematical Formulation:

$$\hat{f}_{RF}(x) = \frac{1}{B} \sum_{b=1}^{B} T_b(x) \tag{3.28}$$

where each tree T_b is trained on bootstrap sample $\mathcal{D}_b$ with random feature subset.
Variance Estimation:

$$\text{Var}[\hat{f}_{RF}(x)] = \frac{1}{B^2} \sum_{b=1}^{B} \text{Var}[T_b(x)] \tag{3.29}$$

Python Implementation:

```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
import pandas as pd

def random_forest_ibudget_algorithm(X, y, person_centered_features=None):
    """
    Implements Random Forest for iBudget with person-centered planning

    Args:
        X: QSI and demographic features
        y: Target expenditures
        person_centered_features: Individual goals/preferences

    Returns:
        Optimized Random Forest model with feature importance
    """
    # Combine traditional and person-centered features
    if person_centered_features is not None:
        X_combined = np.hstack([X, person_centered_features])
    else:
        X_combined = X

    # Hyperparameter tuning
    param_grid = {
        'n_estimators': [100, 200, 500],
        'max_depth': [10, 20, None],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    }

    rf = RandomForestRegressor(random_state=42)
    rf_tuned = GridSearchCV(rf, param_grid, cv=5, scoring='r2', n_jobs
        =-1)
```

```python
    rf_tuned.fit(X_combined, y)

    # Feature importance analysis
    importance_df = pd.DataFrame({
        'feature': range(X_combined.shape[1]),
        'importance': rf_tuned.best_estimator_.feature_importances_
    }).sort_values('importance', ascending=False)

    # Prediction intervals using quantile forests
    from sklearn.ensemble import RandomForestRegressor

    class QuantileRandomForest:
        def __init__(self, **kwargs):
            self.rf = RandomForestRegressor(**kwargs)

        def fit(self, X, y):
            self.rf.fit(X, y)

        def predict_quantiles(self, X, quantiles=[0.1, 0.5, 0.9]):
            predictions = []
            for estimator in self.rf.estimators_:
                predictions.append(estimator.predict(X))

            predictions = np.array(predictions).T
            return np.quantile(predictions, quantiles, axis=1).T

    return rf_tuned.best_estimator_, importance_df

# Usage with prediction intervals
rf_model, feature_importance = random_forest_ibudget_algorithm(
    qsi_features, expenditures, person_centered_data
)

# Generate prediction intervals for budget planning
quantile_rf = QuantileRandomForest(n_estimators=500, random_state=42)
quantile_rf.fit(X_train, y_train)
budget_intervals = quantile_rf.predict_quantiles(X_test, [0.1, 0.5, 0.9])
```

3.4.2.2 Algorithm B2: Gradient Boosting with Custom Objective

Mathematical Formulation:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x) \tag{3.30}$$

where $h_m(x)$ minimizes:

$$h_m = \arg\min_h \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + h(x_i)) \tag{3.31}$$

Custom Person-Centered Loss Function:

$$L_{PC}(y_i, \hat{y}_i) = \alpha \cdot L_{MSE}(y_i, \hat{y}_i) + \beta \cdot L_{PersonCentered}(goals_i, \hat{y}_i) \tag{3.32}$$

Python Implementation:

```python
import xgboost as xgb
import lightgbm as lgb
from sklearn.metrics import mean_squared_error

class PersonCenteredGradientBoosting:
    """
    Custom gradient boosting with person-centered objective function
    """
    def __init__(self, alpha=0.7, beta=0.3):
        self.alpha = alpha  # Weight for prediction accuracy
        self.beta = beta    # Weight for person-centered goals

    def person_centered_objective(self, y_pred, y_true):
        """
        Custom objective combining MSE with person-centered goals
        """
        # Standard MSE component
        mse_grad = 2 * (y_pred - y_true.get_label())
        mse_hess = np.ones_like(y_pred) * 2

        # Person-centered component (example: goal alignment)
        goal_alignment = self._calculate_goal_alignment(y_pred, y_true)
        pc_grad = self._person_centered_gradient(y_pred, goal_alignment)
        pc_hess = self._person_centered_hessian(y_pred, goal_alignment)

        # Combined objective
        grad = self.alpha * mse_grad + self.beta * pc_grad
        hess = self.alpha * mse_hess + self.beta * pc_hess

        return grad, hess

    def fit(self, X, y, person_centered_goals=None):
        """
        Fit XGBoost model with custom objective
        """
        dtrain = xgb.DMatrix(X, label=y)

        params = {
            'objective': self.person_centered_objective,
            'eval_metric': 'rmse',
            'max_depth': 6,
            'learning_rate': 0.1,
            'subsample': 0.8,
            'colsample_bytree': 0.8
        }

        self.model = xgb.train(params, dtrain, num_boost_round=1000)
        return self

    def predict(self, X):
        dtest = xgb.DMatrix(X)
        return self.model.predict(dtest)

# Alternative implementation with LightGBM
```

```python
def lightgbm_ibudget_algorithm(X, y, categorical_features=None):
    """
    LightGBM implementation for iBudget allocation
    """
    train_data = lgb.Dataset(X, label=y, categorical_feature=
        categorical_features)

    params = {
        'objective': 'regression',
        'metric': 'rmse',
        'boosting_type': 'gbdt',
        'num_leaves': 31,
        'learning_rate': 0.05,
        'feature_fraction': 0.9,
        'bagging_fraction': 0.8,
        'bagging_freq': 5,
        'verbose': 0
    }

    model = lgb.train(
        params,
        train_data,
        num_boost_round=1000,
        valid_sets=[train_data],
        early_stopping_rounds=100,
        verbose_eval=False
    )

    return model

# SHAP values for explainability
import shap

def explain_predictions(model, X_test):
    """
    Generate SHAP explanations for individual predictions
    """
    explainer = shap.TreeExplainer(model)
    shap_values = explainer.shap_values(X_test)

    # Individual explanation
    shap.plots.waterfall(explainer.expected_value, shap_values[0], X_test
        .iloc[0])

    return shap_values

# Usage
pc_gb = PersonCenteredGradientBoosting()
pc_gb.fit(X_train, y_train, person_centered_goals)
predictions = pc_gb.predict(X_test)

# Explainability
shap_values = explain_predictions(pc_gb.model, X_test)
```

3.4.3 Hybrid Statistical-Clinical Approaches

3.4.3.1 Algorithm C1: Two-Stage Hybrid Model

Mathematical Formulation:

Stage 1 - Base Statistical Model:

$$\hat{Y}_{base,i} = f_{stat}(QSI_i, Demographics_i, Living_i) \tag{3.33}$$

Stage 2 - Person-Centered Adjustment:

$$\hat{Y}_{final,i} = \hat{Y}_{base,i} \cdot (1 + \delta_i) \tag{3.34}$$

where:

$$\delta_i = g_{PC}(Goals_i, Preferences_i, Strengths_i, Context_i) \tag{3.35}$$

Python Implementation:

```python
from sklearn.base import BaseEstimator, RegressorMixin
import numpy as np

class TwoStageHybridModel(BaseEstimator, RegressorMixin):
    """
    Two-stage hybrid model combining statistical prediction
    with person-centered adjustments
    """

    def __init__(self, base_estimator=None, pc_weight=0.2):
        self.base_estimator = base_estimator or RandomForestRegressor()
        self.pc_weight = pc_weight

    def fit(self, X_statistical, y, X_person_centered=None):
        """
        Fit the two-stage model

        Args:
            X_statistical: Traditional predictors (QSI, demographics)
            y: Target expenditures
            X_person_centered: Person-centered planning features
        """
        # Stage 1: Statistical model
        self.base_estimator.fit(X_statistical, y)
        base_predictions = self.base_estimator.predict(X_statistical)

        # Stage 2: Person-centered adjustment model
        if X_person_centered is not None:
            # Calculate residuals for person-centered modeling
            residuals = y - base_predictions
            relative_residuals = residuals / base_predictions

            # Fit adjustment model
            from sklearn.linear_model import Ridge
            self.adjustment_model = Ridge(alpha=1.0)
            self.adjustment_model.fit(X_person_centered,
                relative_residuals)

            self.has_pc_features = True
```

```python
        else:
            self.has_pc_features = False

        return self

    def predict(self, X_statistical, X_person_centered=None):
        """
        Generate predictions using both stages
        """
        # Stage 1 predictions
        base_pred = self.base_estimator.predict(X_statistical)

        if self.has_pc_features and X_person_centered is not None:
            # Stage 2 adjustments
            adjustments = self.adjustment_model.predict(X_person_centered
                )
            final_pred = base_pred * (1 + self.pc_weight * adjustments)
        else:
            final_pred = base_pred

        return np.maximum(final_pred, 0)  # Ensure non-negative budgets

    def get_explanation(self, X_stat, X_pc, individual_idx):
        """
        Provide explanation for individual prediction
        """
        base_pred = self.base_estimator.predict(X_stat[individual_idx:
            individual_idx+1])

        explanation = {
            'base_allocation': base_pred[0],
            'statistical_factors': self._get_statistical_explanation(
                X_stat[individual_idx]),
        }

        if self.has_pc_features and X_pc is not None:
            pc_adjustment = self.adjustment_model.predict(X_pc[
                individual_idx:individual_idx+1])
            explanation['person_centered_adjustment'] = pc_adjustment[0]
            explanation['final_allocation'] = base_pred[0] * (1 + self.
                pc_weight * pc_adjustment[0])
        else:
            explanation['final_allocation'] = base_pred[0]

        return explanation

# Example usage
def implement_two_stage_model(qsi_data, expenditures,
    person_centered_data):
    """
    Complete implementation of two-stage model
    """
    model = TwoStageHybridModel(
        base_estimator=RandomForestRegressor(n_estimators=200),
        pc_weight=0.15
```

```
    )

    model.fit(qsi_data, expenditures, person_centered_data)

    # Generate predictions
    predictions = model.predict(qsi_test, pc_test)

    # Individual explanations
    explanations = []
    for i in range(len(predictions)):
        exp = model.get_explanation(qsi_test, pc_test, i)
        explanations.append(exp)

    return model, predictions, explanations

# Usage
two_stage_model, preds, explanations = implement_two_stage_model(
    qsi_features, expenditures, person_centered_features
)
```

3.4.3.2 Algorithm C2: Bayesian Hierarchical Model

Mathematical Formulation:

Level 1 (Individual):

$$Y_{ij}|\theta_j, \sigma^2 \sim N(X_{ij}^T\theta_j, \sigma^2) \tag{3.36}$$

Level 2 (Group):

$$\theta_j|\mu, \Sigma \sim N(\mu, \Sigma) \tag{3.37}$$

Level 3 (Population):

$$\mu \sim N(\mu_0, \Sigma_0), \quad \Sigma \sim IW(\nu_0, S_0) \tag{3.38}$$

Python Implementation:

```
import pymc3 as pm
import numpy as np
import pandas as pd
import theano.tensor as tt

def bayesian_hierarchical_ibudget_model(data, group_var='region'):
    """
    Bayesian hierarchical model for iBudget allocation

    Args:
        data: DataFrame with individual-level data
        group_var: Grouping variable (e.g., region, age_group)

    Returns:
        PyMC3 model and trace
    """
    # Prepare data
    groups = data[group_var].unique()
    group_idx = data[group_var].map({g: i for i, g in enumerate(groups)})
        .values
```

```python

    n_groups = len(groups)
    n_obs = len(data)
    n_features = data.select_dtypes(include=[np.number]).shape[1] - 1

    with pm.Model() as hierarchical_model:
        # Hyperpriors
        mu_alpha = pm.Normal('mu_alpha', 0, sigma=100)
        sigma_alpha = pm.HalfNormal('sigma_alpha', sigma=100)

        mu_beta = pm.Normal('mu_beta', 0, sigma=100, shape=n_features)
        sigma_beta = pm.HalfNormal('sigma_beta', sigma=100, shape=n_features)

        # Group-level parameters
        alpha = pm.Normal('alpha', mu=mu_alpha, sigma=sigma_alpha, shape=n_groups)
        beta = pm.Normal('beta', mu=mu_beta, sigma=sigma_beta, shape=(n_groups, n_features))

        # Individual-level likelihood
        X = pm.Data('X', data.select_dtypes(include=[np.number]).iloc[:, :-1].values)
        y_obs = pm.Data('y_obs', data.iloc[:, -1].values)

        mu = alpha[group_idx] + pm.math.dot(X, beta[group_idx].T).diagonal()
        sigma = pm.HalfNormal('sigma', sigma=50)

        likelihood = pm.Normal('y', mu=mu, sigma=sigma, observed=y_obs)

        # Sampling
        trace = pm.sample(2000, tune=1000, cores=4, return_inferencedata=True)

    return hierarchical_model, trace

# Alternative implementation with Stan (via PyStan)
def stan_hierarchical_model():
    """
    Stan implementation for more complex hierarchical models
    """
    stan_code = """
    data {
        int<lower=0> N;              // number of observations
        int<lower=0> J;              // number of groups
        int<lower=0> K;              // number of predictors
        int<lower=1,upper=J> group[N]; // group indicator
        matrix[N,K] X;               // predictor matrix
        vector[N] y;                 // outcome
    }

    parameters {
        real mu_alpha;
        real<lower=0> sigma_alpha;
```

```
        vector[K] mu_beta;
        vector<lower=0>[K] sigma_beta;

        vector[J] alpha;
        matrix[J,K] beta;
        real<lower=0> sigma;
    }

    model {
        // Hyperpriors
        mu_alpha ~ normal(0, 100);
        sigma_alpha ~ normal(0, 50);
        mu_beta ~ normal(0, 10);
        sigma_beta ~ normal(0, 10);

        // Group-level priors
        alpha ~ normal(mu_alpha, sigma_alpha);
        for (k in 1:K) {
            beta[,k] ~ normal(mu_beta[k], sigma_beta[k]);
        }

        // Likelihood
        for (n in 1:N) {
            y[n] ~ normal(alpha[group[n]] + X[n] * beta[group[n]]', sigma);
        }
    }
    """

    return stan_code

# Prediction with uncertainty quantification
def bayesian_predictions_with_uncertainty(model, trace, X_new, group_new):
    """
    Generate predictions with full uncertainty quantification
    """
    with model:
        pm.set_data({'X': X_new, 'group_idx': group_new})
        posterior_pred = pm.sample_posterior_predictive(trace, samples=1000)

    # Extract prediction intervals
    predictions = posterior_pred['y']

    pred_summary = {
        'mean': np.mean(predictions, axis=0),
        'std': np.std(predictions, axis=0),
        'ci_lower': np.percentile(predictions, 2.5, axis=0),
        'ci_upper': np.percentile(predictions, 97.5, axis=0)
    }

    return pred_summary

# Usage example
```

```
hierarchical_model, trace = bayesian_hierarchical_ibudget_model(
    budget_data)
predictions = bayesian_predictions_with_uncertainty(hierarchical_model,
    trace, X_test, group_test)
```

3.4.4 Person-Centered Optimization Approaches

3.4.4.1 Algorithm D1: Multi-Objective Optimization

Mathematical Formulation:

$$\min_{\mathbf{b}} \quad \mathbf{F}(\mathbf{b}) = [f_1(\mathbf{b}), f_2(\mathbf{b}), f_3(\mathbf{b})]^T \tag{3.39}$$

$$\text{subject to} \quad \sum_{i=1}^{n} b_i \leq B_{total} \tag{3.40}$$

$$b_i \geq b_{min,i} \quad \forall i \tag{3.41}$$

$$g_j(\mathbf{b}) \leq 0 \quad j = 1, \ldots, m \tag{3.42}$$

where:

$$f_1(\mathbf{b}) = \sum_{i=1}^{n} (b_i - \hat{b}_i)^2 \quad \text{(prediction accuracy)} \tag{3.43}$$

$$f_2(\mathbf{b}) = -\sum_{i=1}^{n} w_i^{goals} \cdot GoalAlignment_i(b_i) \quad \text{(person-centered goals)} \tag{3.44}$$

$$f_3(\mathbf{b}) = \sum_{i=1}^{n} \sum_{j=1}^{n} |b_i - b_j| \cdot Similarity_{ij} \quad \text{(fairness)} \tag{3.45}$$

Python Implementation:

```python
from pymoo.algorithms.moo.nsga2 import NSGA2
from pymoo.core.problem import Problem
from pymoo.optimize import minimize
import numpy as np

class iBudgetMultiObjectiveProblem(Problem):
    """
    Multi-objective optimization problem for iBudget allocation
    """

    def __init__(self, predicted_budgets, person_centered_goals,
                 total_budget, min_budgets=None):
        self.predicted_budgets = predicted_budgets
        self.person_centered_goals = person_centered_goals
        self.total_budget = total_budget
        self.n_individuals = len(predicted_budgets)
        self.min_budgets = min_budgets or np.zeros(self.n_individuals)

        super().__init__(
            n_var=self.n_individuals,
            n_obj=3,
```

```
            n_constr=1,
            xl=self.min_budgets,
            xu=np.full(self.n_individuals, self.total_budget)
        )

    def _evaluate(self, X, out, *args, **kwargs):
        """
        Evaluate the multi-objective functions
        """
        n_solutions = X.shape[0]
        f1 = np.zeros(n_solutions)  # Prediction accuracy
        f2 = np.zeros(n_solutions)  # Person-centered goal alignment
        f3 = np.zeros(n_solutions)  # Fairness measure
        g1 = np.zeros(n_solutions)  # Budget constraint

        for i in range(n_solutions):
            budgets = X[i, :]

            # Objective 1: Minimize prediction error
            f1[i] = np.sum((budgets - self.predicted_budgets) ** 2)

            # Objective 2: Maximize person-centered goal alignment (minimize negative)
            goal_alignment = self._calculate_goal_alignment(budgets)
            f2[i] = -np.sum(goal_alignment)

            # Objective 3: Minimize inequality (Gini coefficient)
            f3[i] = self._gini_coefficient(budgets)

            # Constraint: Total budget limit
            g1[i] = np.sum(budgets) - self.total_budget

        out["F"] = np.column_stack([f1, f2, f3])
        out["G"] = g1.reshape(-1, 1)

    def _calculate_goal_alignment(self, budgets):
        """
        Calculate alignment between budgets and person-centered goals
        """
        alignment = np.zeros(len(budgets))
        for i, budget in enumerate(budgets):
            # Example: alignment based on service categories funded
            goals = self.person_centered_goals[i]
            alignment[i] = self._goal_budget_alignment(budget, goals)
        return alignment

    def _goal_budget_alignment(self, budget, goals):
        """
        Calculate how well budget aligns with individual goals
        """
        # Simplified alignment calculation
        # In practice, this would involve complex service matching
        target_services = goals.get('preferred_services', [])
        budget_adequacy = min(budget / goals.get('estimated_need', budget), 1.0)
```

```python
        service_availability = len(target_services) / 10.0  # Normalize

        return budget_adequacy * service_availability

    def _gini_coefficient(self, budgets):
        """
        Calculate Gini coefficient for fairness assessment
        """
        sorted_budgets = np.sort(budgets)
        n = len(sorted_budgets)
        cumsum = np.cumsum(sorted_budgets)
        return (2 * np.sum((np.arange(1, n + 1) * sorted_budgets))) / (n * cumsum[-1]) - (n + 1) / n

def solve_multi_objective_ibudget(predicted_budgets, person_centered_goals, total_budget):
    """
    Solve the multi-objective iBudget optimization problem
    """
    problem = iBudgetMultiObjectiveProblem(
        predicted_budgets, person_centered_goals, total_budget
    )

    algorithm = NSGA2(pop_size=100)

    res = minimize(
        problem,
        algorithm,
        ('n_gen', 200),
        verbose=True
    )

    # Extract Pareto front solutions
    pareto_solutions = res.X
    pareto_objectives = res.F

    return pareto_solutions, pareto_objectives

# Goal programming alternative
from scipy.optimize import minimize as scipy_minimize

def goal_programming_ibudget(predicted_budgets, goals, weights, total_budget):
    """
    Goal programming approach for person-centered budget allocation
    """
    n = len(predicted_budgets)

    def objective(x):
        budgets = x[:n]
        pos_dev = x[n:2*n]  # Positive deviations
        neg_dev = x[2*n:3*n]  # Negative deviations

        # Weighted sum of deviations from goals
        return np.sum(weights['accuracy'] * (pos_dev + neg_dev) +
```

```python
                               weights['goals'] * neg_dev +
                               weights['fairness'] * pos_dev)

    def constraints(x):
        budgets = x[:n]
        pos_dev = x[n:2*n]
        neg_dev = x[2*n:3*n]

        constraints = []

        # Budget constraint
        constraints.append(total_budget - np.sum(budgets))

        # Deviation constraints
        for i in range(n):
            target = goals[i]['target_budget']
            constraints.append(budgets[i] - target + neg_dev[i] - pos_dev[i])

        return np.array(constraints)

    # Initial guess
    x0 = np.concatenate([
        predicted_budgets,
        np.zeros(n),  # positive deviations
        np.zeros(n)   # negative deviations
    ])

    # Bounds
    bounds = (
        [(0, total_budget) for _ in range(n)] +  # budgets
        [(0, None) for _ in range(2*n)]          # deviations
    )

    result = scipy_minimize(
        objective, x0, method='SLSQP',
        constraints={'type': 'eq', 'fun': constraints},
        bounds=bounds
    )

    return result.x[:n]  # Return optimized budgets

# Usage example
pareto_solutions, objectives = solve_multi_objective_ibudget(
    predicted_budgets, person_centered_goals, total_budget_available
)

# Select preferred solution from Pareto front
optimal_budgets = pareto_solutions[0]  # or use decision-making criteria
```

3.4.4.2 Algorithm D2: Constrained Optimization with Fairness

Mathematical Formulation:

$$\min_{\mathbf{b}} \quad \sum_{i=1}^{n} \left(b_i - \hat{b}_i\right)^2 + \lambda \sum_{i=1}^{n} w_i \left(GoalScore_i - \frac{b_i}{\bar{b}}\right)^2 \tag{3.46}$$

$$\text{subject to} \quad \sum_{i=1}^{n} b_i = B_{total} \tag{3.47}$$

$$b_i \geq b_{min,i} \quad \forall i \tag{3.48}$$

$$\frac{1}{n_g} \sum_{i \in G_g} b_i \geq \alpha \cdot \bar{b} \quad \forall g \in \{demographic\_groups\} \tag{3.49}$$

$$\left| \frac{1}{n_a} \sum_{i \in A_a} b_i - \frac{1}{n_b} \sum_{i \in A_b} b_i \right| \leq \epsilon \quad \forall a, b \in \{groups\} \tag{3.50}$$

Python Implementation:

```
from scipy.optimize import minimize
import cvxpy as cp
import numpy as np

def constrained_fair_ibudget_allocation(predicted_budgets, demographic_groups,
                                        person_centered_scores, total_budget,
                                        fairness_tolerance=0.1):
    """
    Constrained optimization with fairness constraints

    Args:
        predicted_budgets: Initial statistical predictions
        demographic_groups: Group membership for fairness constraints
        person_centered_scores: Individual person-centered alignment scores
        total_budget: Total available budget
        fairness_tolerance: Maximum allowed group budget difference

    Returns:
        Optimized budget allocation
    """
    n = len(predicted_budgets)
    unique_groups = np.unique(demographic_groups)
    n_groups = len(unique_groups)

    # Decision variable
    budgets = cp.Variable(n, pos=True)

    # Objective function
    prediction_error = cp.sum_squares(budgets - predicted_budgets)
    person_centered_alignment = cp.sum(
        cp.multiply(person_centered_scores,
                    cp.square(budgets - np.mean(predicted_budgets)))
    )

    objective = cp.Minimize(
```

```
        prediction_error + 0.1 * person_centered_alignment
    )

    # Constraints
    constraints = []

    # Budget constraint
    constraints.append(cp.sum(budgets) == total_budget)

    # Minimum budget constraints
    min_budgets = 0.1 * predicted_budgets  # 10% minimum
    constraints.append(budgets >= min_budgets)

    # Fairness constraints between demographic groups
    group_means = []
    for group in unique_groups:
        group_mask = (demographic_groups == group)
        group_indices = np.where(group_mask)[0]
        group_mean = cp.sum(budgets[group_indices]) / np.sum(group_mask)
        group_means.append(group_mean)

    # Pairwise fairness constraints
    for i in range(n_groups):
        for j in range(i + 1, n_groups):
            constraints.append(
                cp.abs(group_means[i] - group_means[j]) <=
                fairness_tolerance * np.mean(predicted_budgets)
            )

    # Solve optimization problem
    problem = cp.Problem(objective, constraints)
    problem.solve(solver=cp.OSQP)

    if problem.status == cp.OPTIMAL:
        return budgets.value
    else:
        raise ValueError(f"Optimization failed with status: {problem.
            status}")

# Alternative formulation with robust optimization
def robust_fair_ibudget_allocation(predicted_budgets, uncertainty_sets,
                                   demographic_groups, total_budget):
    """
    Robust optimization approach handling prediction uncertainty
    """
    n = len(predicted_budgets)

    # Decision variables
    budgets = cp.Variable(n, pos=True)
    slack_vars = cp.Variable(n, pos=True)  # For robust constraints

    # Worst-case objective considering uncertainty
    worst_case_error = 0
    for i in range(n):
        # Uncertainty set for individual i (e.g., confidence interval)
```

```
        uncertainty_radius = uncertainty_sets[i]
        worst_case_error += cp.maximum(
            cp.square(budgets[i] - (predicted_budgets[i] +
                uncertainty_radius)),
            cp.square(budgets[i] - (predicted_budgets[i] -
                uncertainty_radius))
        )

    objective = cp.Minimize(worst_case_error + cp.sum(slack_vars))

    # Constraints with robustness
    constraints = [
        cp.sum(budgets) == total_budget,
        budgets >= 0.05 * total_budget / n,  # Minimum allocation
        slack_vars >= 0
    ]

    # Robust fairness constraints
    unique_groups = np.unique(demographic_groups)
    for group in unique_groups:
        group_mask = (demographic_groups == group)
        group_indices = np.where(group_mask)[0]

        # Ensure group gets fair share even under uncertainty
        group_min_share = 0.8 * np.sum(predicted_budgets[group_mask])
        constraints.append(
            cp.sum(budgets[group_indices]) >= group_min_share -
                slack_vars[group_indices[0]]
        )

    problem = cp.Problem(objective, constraints)
    problem.solve()

    return budgets.value, slack_vars.value

# Fairness auditing function
def audit_allocation_fairness(budgets, demographic_groups,
    protected_attributes):
    """
    Comprehensive fairness audit of budget allocation
    """
    fairness_metrics = {}

    # Statistical parity
    for attr in protected_attributes:
        groups = np.unique(demographic_groups[attr])
        group_means = []
        for group in groups:
            mask = (demographic_groups[attr] == group)
            group_mean = np.mean(budgets[mask])
            group_means.append(group_mean)

        fairness_metrics[f'{attr}_statistical_parity'] = {
            'group_means': dict(zip(groups, group_means)),
            'max_difference': max(group_means) - min(group_means),
```

```
            'coefficient_variation': np.std(group_means) / np.mean(
                group_means)
        }

    # Equalized opportunity (for different need levels)
    # This would require additional need-level data

    return fairness_metrics

# Usage example
optimized_budgets = constrained_fair_ibudget_allocation(
    statistical_predictions, demographic_data, pc_scores, total_budget
)

# Audit the results
fairness_audit = audit_allocation_fairness(
    optimized_budgets, demographic_data, ['age_group', 'disability_type',
        'region']
)
```

3.4.5 Modern Time-Aware Approaches

3.4.5.1 Algorithm E1: Dynamic Regression with Time Effects

Mathematical Formulation:

Time-Varying Coefficient Model:

$$Y_{it} = X_{it}^T \beta_t + \varepsilon_{it} \tag{3.51}$$

where coefficients evolve as:

$$\beta_t = \beta_{t-1} + \omega_t, \quad \omega_t \sim N(0, Q) \tag{3.52}$$

State-Space Representation:

$$\beta_t = F\beta_{t-1} + \omega_t \quad \text{(State equation)} \tag{3.53}$$

$$Y_t = H_t\beta_t + \varepsilon_t \quad \text{(Observation equation)} \tag{3.54}$$

Python Implementation:

```
from statsmodels.tsa.statespace import MLEModel
from scipy.linalg import block_diag
import numpy as np
import pandas as pd

class DynamicRegressioniBudget(MLEModel):
    """
    Dynamic regression model for iBudget allocation with time-varying
        coefficients
    """

    def __init__(self, endog, exog, **kwargs):
        self.k_exog = exog.shape[1]

        # Initialize state space model
```

```
        super().__init__(
            endog,
            k_states=self.k_exog,
            k_posdef=self.k_exog,
            **kwargs
        )

        self.exog = exog

        # Initialize state space matrices
        self._initialize_state_space()

    def _initialize_state_space(self):
        """
        Initialize state space representation
        """
        # Transition matrix (random walk for coefficients)
        self['transition'] = np.eye(self.k_exog)

        # Selection matrix
        self['selection'] = np.eye(self.k_exog)

        # Initial state covariance
        self['state_cov'] = np.eye(self.k_exog)

    def update(self, params, **kwargs):
        """
        Update state space matrices with current parameters
        """
        # Parameter mapping
        obs_var = params[0]
        state_vars = params[1:1+self.k_exog]

        # Update observation equation
        self['obs_intercept'] = 0
        self['design'] = self.exog
        self['obs_cov'] = obs_var

        # Update state equation
        self['state_cov'] = np.diag(state_vars)

    @property
    def param_names(self):
        return ['obs_var'] + [f'state_var_{i}' for i in range(self.k_exog
            )]

    @property
    def start_params(self):
        return [1.0] + [0.1] * self.k_exog

def fit_dynamic_ibudget_model(expenditure_data, qsi_features, time_index)
    :
    """
    Fit dynamic regression model to iBudget data

```

```
    Args:
        expenditure_data: Time series of expenditures
        qsi_features: QSI features over time
        time_index: Time index for observations

    Returns:
        Fitted model and time-varying coefficients
    """
    # Prepare data
    endog = expenditure_data.values
    exog = qsi_features.values

    # Fit model
    model = DynamicRegressioniBudget(endog, exog)
    results = model.fit()

    # Extract time-varying coefficients
    states = results.states.filtered
    time_varying_coeffs = pd.DataFrame(
        states.T,
        index=time_index,
        columns=[f'coeff_{i}' for i in range(qsi_features.shape[1])]
    )

    return results, time_varying_coeffs

# Alternative implementation with rolling regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

def rolling_regression_ibudget(data, window_size=12, min_periods=6):
    """
    Rolling regression approach for time-adaptive iBudget algorithm
    """
    results = []

    for i in range(min_periods, len(data)):
        start_idx = max(0, i - window_size)
        end_idx = i + 1

        # Extract window data
        window_data = data.iloc[start_idx:end_idx]
        X = window_data.drop('expenditure', axis=1)
        y = window_data['expenditure']

        # Fit model on window
        model = LinearRegression()
        model.fit(X, y)

        # Store results
        result = {
            'date': data.index[i],
            'coefficients': model.coef_,
            'intercept': model.intercept_,
            'r2': model.score(X, y),
```

```python
            'mse': mean_squared_error(y, model.predict(X))
        }
        results.append(result)

    return pd.DataFrame(results)

# Inflation adjustment mechanism
def adjust_for_inflation(historical_budgets, inflation_rates, base_year
    =2024):
    """
    Adjust historical budget data for inflation
    """
    adjusted_budgets = historical_budgets.copy()

    for year, rate in inflation_rates.items():
        if year != base_year:
            adjustment_factor = (1 + rate) ** (base_year - year)
            year_mask = adjusted_budgets.index.year == year
            adjusted_budgets.loc[year_mask] *= adjustment_factor

    return adjusted_budgets

# Forecasting future budget needs
from statsmodels.tsa.arima.model import ARIMA

def forecast_budget_trends(time_series_data, horizon=12):
    """
    Forecast future budget trends using ARIMA
    """
    forecasts = {}

    for column in time_series_data.columns:
        # Fit ARIMA model
        model = ARIMA(time_series_data[column], order=(1, 1, 1))
        fitted_model = model.fit()

        # Generate forecasts
        forecast = fitted_model.forecast(steps=horizon)
        conf_int = fitted_model.get_forecast(steps=horizon).conf_int()

        forecasts[column] = {
            'forecast': forecast,
            'lower_bound': conf_int.iloc[:, 0],
            'upper_bound': conf_int.iloc[:, 1]
        }

    return forecasts

# Usage example
dynamic_model, time_coeffs = fit_dynamic_ibudget_model(
    expenditure_time_series, qsi_time_series, date_index
)

# Rolling regression for comparison
rolling_results = rolling_regression_ibudget(combined_time_series_data)
```

```python

# Forecast future needs
budget_forecasts = forecast_budget_trends(historical_budget_data)
```

3.4.5.2 Algorithm E2: Longitudinal Mixed-Effects Model

Mathematical Formulation:

Mixed-Effects Model:

$$Y_{ij} = X_{ij}^T \beta + Z_{ij}^T b_i + \varepsilon_{ij} \tag{3.55}$$

where:

$$b_i \sim N(0, G) \quad \text{(Random effects)} \tag{3.56}$$

$$\varepsilon_{ij} \sim N(0, R_{ij}) \quad \text{(Within-individual errors)} \tag{3.57}$$

Individual Growth Curves:

$$Y_{ij} = \beta_0 + \beta_1 t_{ij} + \beta_2 t_{ij}^2 + b_{0i} + b_{1i} t_{ij} + \varepsilon_{ij} \tag{3.58}$$

Python Implementation:

```python
import statsmodels.api as sm
from statsmodels.regression.mixed_linear_model import MixedLM
import numpy as np
import pandas as pd

def longitudinal_ibudget_model(data, individual_id='client_id', time_var=
    'time'):
    """
    Longitudinal mixed-effects model for iBudget needs prediction

    Args:
        data: Panel data with repeated observations per individual
        individual_id: Column name for individual identifier
        time_var: Column name for time variable

    Returns:
        Fitted mixed-effects model
    """
    # Prepare fixed effects design matrix
    fixed_effects = ['age', 'qsi_behavioral_sum', 'qsi_functional_sum',
                    'qsi_physical_sum', time_var, f'{time_var}_squared']

    # Add squared time term
    data[f'{time_var}_squared'] = data[time_var] ** 2

    # Fit mixed-effects model
    model = MixedLM(
        endog=data['expenditure'],
        exog=data[fixed_effects],
        groups=data[individual_id],
        exog_re=data[[time_var]]  # Random slope for time
    )

    results = model.fit()
```

```python
    return results

# Alternative implementation with scikit-learn style
from sklearn.base import BaseEstimator, RegressorMixin
from sklearn.linear_model import LinearRegression

class LongitudinaliBudgetPredictor(BaseEstimator, RegressorMixin):
    """
    Longitudinal predictor for individual budget trajectories
    """

    def __init__(self, max_time_horizon=5):
        self.max_time_horizon = max_time_horizon
        self.individual_models = {}
        self.population_model = None

    def fit(self, X, y, individual_ids, time_points):
        """
        Fit individual trajectory models

        Args:
            X: Feature matrix
            y: Target expenditures
            individual_ids: Individual identifiers
            time_points: Time points for observations
        """
        # Fit population-level model
        X_with_time = np.column_stack([X, time_points, time_points**2])
        self.population_model = LinearRegression()
        self.population_model.fit(X_with_time, y)

        # Fit individual models for those with sufficient data
        unique_individuals = np.unique(individual_ids)

        for ind_id in unique_individuals:
            mask = individual_ids == ind_id
            if np.sum(mask) >= 3:  # Need at least 3 observations
                X_ind = X[mask]
                y_ind = y[mask]
                t_ind = time_points[mask]

                # Individual trajectory model
                X_ind_with_time = np.column_stack([X_ind, t_ind, t_ind
                    **2])
                ind_model = LinearRegression()
                ind_model.fit(X_ind_with_time, y_ind)

                self.individual_models[ind_id] = {
                    'model': ind_model,
                    'last_observation_time': np.max(t_ind),
                    'last_features': X_ind[-1],
                    'trajectory_slope': ind_model.coef_[-2]  # Linear
                        time coefficient
                }

```

```
        return self

    def predict_trajectory(self, individual_id, future_time_points,
                          latest_features=None):
        """
        Predict future trajectory for an individual
        """
        if individual_id in self.individual_models:
            # Use individual model
            ind_info = self.individual_models[individual_id]
            model = ind_info['model']

            if latest_features is None:
                latest_features = ind_info['last_features']

            # Create prediction matrix
            n_points = len(future_time_points)
            X_pred = np.tile(latest_features, (n_points, 1))
            X_pred = np.column_stack([
                X_pred,
                future_time_points,
                future_time_points**2
            ])

            return model.predict(X_pred)
        else:
            # Use population model
            n_points = len(future_time_points)
            if latest_features is None:
                # Use population averages
                latest_features = np.mean(self.population_model.coef_[:-2])

            X_pred = np.tile(latest_features, (n_points, 1))
            X_pred = np.column_stack([
                X_pred,
                future_time_points,
                future_time_points**2
            ])

            return self.population_model.predict(X_pred)

    def identify_high_risk_individuals(self, threshold_slope=100):
        """
        Identify individuals with rapidly increasing needs
        """
        high_risk = []

        for ind_id, info in self.individual_models.items():
            if info['trajectory_slope'] > threshold_slope:
                high_risk.append({
                    'individual_id': ind_id,
                    'slope': info['trajectory_slope'],
                    'last_time': info['last_observation_time']
                })
```

```python

        return sorted(high_risk, key=lambda x: x['slope'], reverse=True)

# Survival analysis for service transitions
from lifelines import CoxPHFitter

def service_transition_analysis(data, duration_col='time_to_transition',
                                event_col='transitioned'):
    """
    Analyze transitions between service levels using survival analysis
    """
    # Prepare data for Cox regression
    cph = CoxPHFitter()

    # Fit Cox proportional hazards model
    cph.fit(
        data,
        duration_col=duration_col,
        event_col=event_col
    )

    return cph

# Longitudinal clustering for trajectory identification
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

def identify_trajectory_patterns(longitudinal_data, n_clusters=5):
    """
    Identify common trajectory patterns in budget needs
    """
    # Reshape data for clustering (individuals x time points)
    pivot_data = longitudinal_data.pivot_table(
        index='client_id',
        columns='time',
        values='expenditure'
    ).fillna(method='ffill').fillna(method='bfill')

    # Standardize trajectories
    scaler = StandardScaler()
    scaled_trajectories = scaler.fit_transform(pivot_data)

    # Cluster trajectories
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    trajectory_clusters = kmeans.fit_predict(scaled_trajectories)

    # Analyze cluster characteristics
    cluster_profiles = {}
    for cluster_id in range(n_clusters):
        mask = trajectory_clusters == cluster_id
        cluster_data = pivot_data.iloc[mask]

        cluster_profiles[cluster_id] = {
            'n_individuals': np.sum(mask),
            'mean_trajectory': cluster_data.mean(axis=0),
```

```python
            'std_trajectory': cluster_data.std(axis=0),
            'trend': 'increasing' if cluster_data.iloc[:, -1].mean() >
                cluster_data.iloc[:, 0].mean() else 'stable'
        }

    return trajectory_clusters, cluster_profiles

# Usage example
longitudinal_model = longitudinal_ibudget_model(panel_data)

# Individual trajectory prediction
trajectory_predictor = LongitudinaliBudgetPredictor()
trajectory_predictor.fit(X_features, expenditures, client_ids,
    time_points)

# Predict future needs
future_times = np.array([1, 2, 3, 4, 5])  # Next 5 time periods
individual_forecast = trajectory_predictor.predict_trajectory('client_123
    ', future_times)

# Identify high-risk individuals
high_risk_clients = trajectory_predictor.identify_high_risk_individuals()
```

3.4.6 Specialized Needs-Based Approaches

3.4.6.1 Algorithm F1: Latent Class Mixture Model

Mathematical Formulation:

Mixture Model:

$$f(y_i|x_i, \Theta) = \sum_{k=1}^{K} \pi_k f_k(y_i|x_i, \theta_k) \tag{3.59}$$

where:

$$\pi_k = P(\text{Individual } i \text{ belongs to class } k) \tag{3.60}$$

$$f_k(y_i|x_i, \theta_k) = \text{Class-specific density function} \tag{3.61}$$

EM Algorithm for Estimation:

E-step:

$$\gamma_{ik} = \frac{\pi_k f_k(y_i|x_i, \theta_k)}{\sum_{j=1}^{K} \pi_j f_j(y_i|x_i, \theta_j)} \tag{3.62}$$

M-step:

$$\pi_k^{(new)} = \frac{1}{n} \sum_{i=1}^{n} \gamma_{ik} \tag{3.63}$$

$$\theta_k^{(new)} = \arg\max_{\theta_k} \sum_{i=1}^{n} \gamma_{ik} \log f_k(y_i|x_i, \theta_k) \tag{3.64}$$

Python Implementation:

```python
from sklearn.mixture import GaussianMixture
from sklearn.linear_model import LinearRegression
import numpy as np
import pandas as pd

class LatentClassiBudgetModel:
    """
    Latent class mixture model for iBudget allocation
    """

    def __init__(self, n_classes=4, max_iter=100, random_state=42):
        self.n_classes = n_classes
        self.max_iter = max_iter
        self.random_state = random_state
        self.class_models = {}
        self.mixture_model = None
        self.class_interpretations = {}

    def fit(self, X, y, feature_names=None):
        """
        Fit latent class mixture model

        Args:
            X: Feature matrix (QSI scores, demographics)
            y: Target expenditures
            feature_names: Names of features for interpretation
        """
        # Step 1: Initial clustering to identify latent classes
        initial_gmm = GaussianMixture(
            n_components=self.n_classes,
            random_state=self.random_state
        )

        # Use both features and outcomes for clustering
        clustering_data = np.column_stack([X, y.reshape(-1, 1)])
        class_assignments = initial_gmm.fit_predict(clustering_data)

        # Step 2: Fit class-specific regression models
        for k in range(self.n_classes):
            class_mask = (class_assignments == k)
            X_class = X[class_mask]
            y_class = y[class_mask]

            if len(y_class) > 10:  # Minimum samples for stable estimation
                model = LinearRegression()
                model.fit(X_class, y_class)

                self.class_models[k] = {
                    'model': model,
                    'n_samples': len(y_class),
                    'mean_expenditure': np.mean(y_class),
                    'mean_features': np.mean(X_class, axis=0)
                }
```

```python

        # Step 3: Final mixture model for class assignment
        self.mixture_model = GaussianMixture(
            n_components=len(self.class_models),
            random_state=self.random_state
        )
        self.mixture_model.fit(X)

        # Step 4: Interpret classes
        self._interpret_classes(X, y, feature_names)

        return self

    def _interpret_classes(self, X, y, feature_names):
        """
        Generate interpretations for each latent class
        """
        if feature_names is None:
            feature_names = [f'feature_{i}' for i in range(X.shape[1])]

        for k, class_info in self.class_models.items():
            mean_features = class_info['mean_features']
            mean_expenditure = class_info['mean_expenditure']

            # Identify distinguishing features
            overall_means = np.mean(X, axis=0)
            feature_deviations = mean_features - overall_means

            # Find most distinctive features
            top_features = np.argsort(np.abs(feature_deviations))[-5:]

            interpretation = {
                'class_size': class_info['n_samples'],
                'avg_expenditure': mean_expenditure,
                'distinguishing_features': [
                    {
                        'feature': feature_names[i],
                        'class_mean': mean_features[i],
                        'overall_mean': overall_means[i],
                        'deviation': feature_deviations[i]
                    }
                    for i in top_features
                ]
            }

            self.class_interpretations[k] = interpretation

    def predict(self, X):
        """
        Predict expenditures using mixture of class-specific models
        """
        # Get class probabilities
        class_probs = self.mixture_model.predict_proba(X)

        predictions = np.zeros(len(X))
```

```python

        for i, x in enumerate(X):
            class_prediction = 0
            for k, class_info in self.class_models.items():
                if k < len(class_probs[i]):
                    class_pred = class_info['model'].predict(x.reshape(1,
                         -1))[0]
                    class_prediction += class_probs[i][k] * class_pred

            predictions[i] = class_prediction

        return predictions

    def assign_class(self, X):
        """
        Assign individuals to most likely class
        """
        return self.mixture_model.predict(X)

    def get_class_interpretation(self, class_id):
        """
        Get human-readable interpretation of a class
        """
        return self.class_interpretations.get(class_id, "Class not found"
            )

# Usage example
latent_class_model = LatentClassiBudgetModel(n_classes=5)
latent_class_model.fit(qsi_features, expenditures, qsi_feature_names)

# Make predictions
predictions = latent_class_model.predict(qsi_test)

# Assign individuals to classes
class_assignments = latent_class_model.assign_class(qsi_test)

# Interpret classes
for class_id in range(5):
    interpretation = latent_class_model.get_class_interpretation(class_id
        )
    print(f"Class {class_id}: {interpretation}")
```

3.4.6.2 Algorithm F2: Support Vector Regression

Mathematical Formulation:

SVR Optimization Problem:

$$\min_{w,b,\xi,\xi^*} \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*) \tag{3.65}$$

$$\text{subject to} \quad y_i - w^T\phi(x_i) - b \leq \epsilon + \xi_i \tag{3.66}$$

$$w^T\phi(x_i) + b - y_i \leq \epsilon + \xi_i^* \tag{3.67}$$

$$\xi_i, \xi_i^* \geq 0 \tag{3.68}$$

Dual Formulation:

$$f(x) = \sum_{i=1}^{n} (\alpha_i - \alpha_i^*) K(x_i, x) + b \tag{3.69}$$

where $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel function.

Python Implementation:

```python
from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
import numpy as np

class SVRiBudgetAllocator:
    """
    Support Vector Regression for iBudget allocation
    """

    def __init__(self, kernel='rbf', multi_output=False):
        self.kernel = kernel
        self.multi_output = multi_output
        self.models = {}
        self.scaler = StandardScaler()
        self.is_fitted = False

    def fit(self, X, y, service_categories=None):
        """
        Fit SVR model(s)

        Args:
            X: Feature matrix
            y: Target expenditures (total or by category)
            service_categories: If provided, fit separate models for each
                category
        """
        X_scaled = self.scaler.fit_transform(X)

        if self.multi_output and service_categories is not None:
            # Fit separate SVR for each service category
            unique_categories = np.unique(service_categories)

            for category in unique_categories:
                # Parameter grid for optimization
                param_grid = {
                    'C': [0.1, 1, 10, 100],
                    'epsilon': [0.01, 0.1, 0.2],
                    'gamma': ['scale', 'auto', 0.001, 0.01, 0.1, 1]
                }

                svr = SVR(kernel=self.kernel)
                grid_search = GridSearchCV(
                    svr, param_grid, cv=5, scoring='r2', n_jobs=-1
                )

                # Extract category-specific targets
```

```python
                category_mask = service_categories == category
                y_category = y[category_mask] if len(y.shape) == 1 else y
                    [:, category]

                grid_search.fit(X_scaled, y_category)
                self.models[category] = grid_search.best_estimator_

        else:
            # Single SVR model
            param_grid = {
                'svr__C': [0.1, 1, 10, 100, 1000],
                'svr__epsilon': [0.01, 0.1, 0.2, 0.5],
                'svr__gamma': ['scale', 'auto', 0.001, 0.01, 0.1, 1]
            }

            pipeline = Pipeline([
                ('scaler', StandardScaler()),
                ('svr', SVR(kernel=self.kernel))
            ])

            grid_search = GridSearchCV(
                pipeline, param_grid, cv=5, scoring='r2', n_jobs=-1
            )

            grid_search.fit(X, y)
            self.models['total'] = grid_search.best_estimator_

        self.is_fitted = True
        return self

    def predict(self, X):
        """
        Generate predictions
        """
        if not self.is_fitted:
            raise ValueError("Model must be fitted before prediction")

        if len(self.models) == 1 and 'total' in self.models:
            return self.models['total'].predict(X)
        else:
            # Multi-output prediction
            predictions = {}
            X_scaled = self.scaler.transform(X)

            for category, model in self.models.items():
                predictions[category] = model.predict(X_scaled)

            return predictions

# Usage example
svr_allocator = SVRiBudgetAllocator(kernel='rbf', multi_output=True)
svr_allocator.fit(qsi_features, expenditures, service_categories)

# Make predictions
svr_predictions = svr_allocator.predict(qsi_test)
```

3.5 Implementation Framework and Validation

3.5.1 Model Selection Criteria

For algorithm selection, we propose a comprehensive evaluation framework:

Performance Metrics:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{3.70}$$

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100\% \tag{3.71}$$

$$R^2_{adj} = 1 - \frac{(1 - R^2)(n - 1)}{n - p - 1} \tag{3.72}$$

Fairness Metrics:

$$\text{Statistical Parity} = \max_{g,h} |E[\hat{Y}|G = g] - E[\hat{Y}|G = h]| \tag{3.73}$$

$$\text{Equalized Opportunity} = \max_{g,h} |P(\hat{Y} > t|Y > t, G = g) - P(\hat{Y} > t|Y > t, G = h)| \tag{3.74}$$

Person-Centered Compliance Score:

$$PCC = \frac{1}{n}\sum_{i=1}^{n}\text{GoalAlignment}_i(\hat{Y}_i, Goals_i, Preferences_i) \tag{3.75}$$

3.5.2 Validation Framework

```python
from sklearn.model_selection import TimeSeriesSplit, cross_val_score
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

def comprehensive_algorithm_validation(models, X, y, temporal_data=None):
    """
    Comprehensive validation framework for iBudget algorithms
    """
    validation_results = {}

    for name, model in models.items():
        print(f"Validating {name}...")

        # Time series validation if temporal data available
        if temporal_data is not None:
            tscv = TimeSeriesSplit(n_splits=5)
            cv_scores = []

            for train_idx, test_idx in tscv.split(X):
                X_train, X_test = X[train_idx], X[test_idx]
                y_train, y_test = y[train_idx], y[test_idx]

                model.fit(X_train, y_train)
                y_pred = model.predict(X_test)

```

```python
                cv_scores.append({
                    'rmse': np.sqrt(mean_squared_error(y_test, y_pred)),
                    'r2': r2_score(y_test, y_pred),
                    'mape': np.mean(np.abs((y_test - y_pred) / y_test)) *
                        100
                })

            validation_results[name] = {
                'temporal_cv': cv_scores,
                'mean_rmse': np.mean([s['rmse'] for s in cv_scores]),
                'mean_r2': np.mean([s['r2'] for s in cv_scores]),
                'mean_mape': np.mean([s['mape'] for s in cv_scores])
            }

        else:
            # Standard cross-validation
            cv_scores = cross_val_score(model, X, y, cv=5, scoring='r2')
            validation_results[name] = {
                'cv_r2_mean': np.mean(cv_scores),
                'cv_r2_std': np.std(cv_scores)
            }

    return validation_results

def fairness_audit_framework(predictions, demographics,
    protected_attributes):
    """
    Comprehensive fairness auditing
    """
    fairness_results = {}

    for attr in protected_attributes:
        groups = np.unique(demographics[attr])
        group_stats = {}

        for group in groups:
            mask = demographics[attr] == group
            group_stats[group] = {
                'mean_prediction': np.mean(predictions[mask]),
                'std_prediction': np.std(predictions[mask]),
                'n_individuals': np.sum(mask)
            }

        # Calculate statistical parity difference
        group_means = [stats['mean_prediction'] for stats in group_stats.
            values()]
        statistical_parity = max(group_means) - min(group_means)

        fairness_results[attr] = {
            'group_statistics': group_stats,
            'statistical_parity_difference': statistical_parity,
            'coefficient_of_variation': np.std(group_means) / np.mean(
                group_means)
        }

```

```python
    return fairness_results
```

3.6 Recommendations and Implementation Roadmap

3.6.1 Phased Implementation Approach

Phase 1: Foundation Models

- Implement Algorithm A1 (Robust Linear Regression)
- Implement Algorithm B1 (Random Forest)
- Establish validation framework
- Compare against current Model 5b

Phase 2: Advanced Approaches

- Deploy Algorithm C1 (Two-Stage Hybrid)
- Implement Algorithm D1 (Multi-Objective Optimization)
- Conduct fairness audits
- Pilot with subset of enrollees

Phase 3: Specialized Models

- Implement Algorithm E1 (Dynamic Regression)
- Deploy Algorithm F1 (Latent Class Mixture)
- Full system integration
- Policy compliance verification

3.7 Conclusion

The current iBudget algorithm exhibits significant limitations in prediction accuracy, temporal validity, and compliance with person-centered planning requirements mandated by House Bill 1103. The proposed collection of alternative algorithms addresses these deficiencies through:

1. **Enhanced statistical robustness** via outlier-resistant methods
2. **Person-centered integration** through multi-objective optimization
3. **Temporal adaptability** using dynamic regression approaches
4. **Specialized population modeling** via mixture models
5. **Fairness assurance** through constrained optimization
6. **Transparency and explainability** via interpretable ML methods

The mathematical formulations and Python implementations provided offer a comprehensive foundation for developing a next-generation iBudget allocation system that meets both statistical rigor and regulatory compliance requirements. The phased implementation approach ensures systematic validation and stakeholder engagement throughout the transition process.

Key success metrics for the new algorithms should include:

$$R^2 \geq 0.85 \text{ (vs. current 0.80)} \tag{3.76}$$

$$\text{Outlier Rate} \leq 2\% \text{ (vs. current 9.4\%)} \tag{3.77}$$

$$\text{Fairness Score} \geq 0.95 \tag{3.78}$$

$$\text{Person-Centered Compliance} \geq 0.90 \tag{3.79}$$

This comprehensive approach ensures Florida's iBudget system evolves to better serve individuals with developmental disabilities while maintaining fiscal responsibility and regulatory compliance.