TOOLING ON MATLAB FOR ONLINE CONVEX OPTIMIZATION

Philip Naveen
University of Virginia
Charlottesville
philipnaveen@email.virginia.edu

ABSTRACT

This manuscript is merely a formal documentation of the purpose and details surrounding the online convex optimization toolbox (OCOBox) for MATLAB. The purpose of this toolbox is to provide a collection of algorithms that work under stochastic situations where traditional algorithmic theory does not fare so well. The toolbox encompasses a wide range of methods including Bayesian persuasion, bandit optimization, Blackwell approachability, boosting, game theory, projection-free algorithms, and regularization. In the future, we plan to extend OCOBox to interactive machine learning algorithms and develop a more robust GUI.

***K*eywords** MATLAB · Online Convex Optimization · Stochastic Processes

1 Introduction

Optimization is a fundamental area of mathematics and computer science, with applications spanning a wide range of fields including machine learning, operations research, economics, engineering, and more . Optimization algorithms seek to find the best solution from a set of possible solutions, often by maximizing or minimizing a certain objective function. The diversity and complexity of optimization problems have led to the development of numerous algorithms, each with its unique strengths, weaknesses, and areas of applicability [7].

MATLAB, a high-level language and interactive environment for numerical computation, visualization, and programming, is widely used in both academic and industrial settings for developing and implementing optimization algorithms. However, researchers and practitioners often face challenges when it comes to testing and comparing different optimization methods due to the lack of a unified framework [5].

To address this challenge, we introduce the OCOBox, a comprehensive MATLAB toolbox that consolidates a wide array of optimization algorithms into a single, user-friendly interface. The toolbox is designed to facilitate the implementation, comparison, and analysis of various optimization techniques, making it an invaluable resource for researchers, educators, and practitioners alike.

The primary motivation behind the development of the OCOBox is to provide a versatile and accessible platform for exploring optimization algorithms. Specifically, the toolbox aims to achieve the following objectives:

1. **Comprehensive Coverage**: Include a diverse set of optimization algorithms covering various categories such as Bayesian persuasion, bandit optimization, gradient descent methods, boosting, game theory, and more. This breadth ensures that users can find suitable algorithms for a wide range of optimization problems.
2. **Ease of Use**: Offer a user-friendly collection of algorithms to save time from searching the internet and textbook for implementations.
3. **Comparative Analysis**: Facilitate the comparative analysis of different optimization algorithms by providing a standardized framework for evaluation. Users can easily switch between algorithms and compare their performance on various benchmark problems.
4. **Educational Value**: Serve as a valuable educational resource for teaching optimization concepts. The toolbox includes well-documented code and examples, making it an excellent tool for students and educators in optimization courses.

1.1 Overview of the Toolbox

The OCOBox is organized into several subdirectories, each corresponding to a specific category of optimization algorithms. This modular structure allows users to navigate the toolbox easily and focus on the algorithms relevant to their needs. The main categories and their respective algorithms are as follows:

- **Algorithmic Bayesian Persuasion**: Models and solves problems where a sender influences the beliefs and actions of a receiver through strategic information disclosure.
- **Bandit Optimization**: Addresses decision-making problems where an agent must balance exploration and exploitation.
- **Blackwell Approachability**: Applies game-theoretic concepts to ensure outcomes converge to a desired set.
- **Boosting**: Implements ensemble methods to improve the performance of weak learners.
- **Games and Duality**: Includes algorithms related to game theory and linear programming duality.
- **Gradient Descent**: Provides various methods for gradient-based optimization.
- **Learning Theory**: Covers foundational algorithms in statistical learning theory.
- **Momentum-Based Optimization**: Includes algorithms that accelerate gradient descent by considering past gradients.
- **Online Optimization**: Updates models incrementally as new data arrives.
- **Projection-Free Algorithms**: Utilizes methods that avoid computationally expensive projections.
- **Regularization**: Prevents overfitting by adding constraints or penalties.

Each algorithm in the toolbox is implemented as a MATLAB script or function, adhering to best practices in software development to ensure readability, modularity, and reusability. The toolbox includes detailed documentation and examples for each algorithm, guiding users through the process of setting up and running optimization tasks.

The OCOBox is applicable to a wide range of domains. In machine learning, it can be used to fine-tune models and improve performance through stochastic optimization and regularization techniques [8]. In operations research, it provides tools for solving complex decision-making problems. In economics, it offers methods for strategic information disclosure and game theory analysis. Furthermore, its educational value makes it an excellent resource for teaching optimization concepts in academic settings. The algorithms are almost exclusively based on textbooks on the OCO, and research conducted on algorithmic Bayesian persuasion [4, 8, 3].

2 Algorithmic Bayesian Persuasion

Bayesian persuasion involves a sender who aims to influence the receiver's actions by sharing information about an underlying state of the world. The sender's goal is to design an optimal signaling strategy that maximizes their payoff, considering the receiver's response to the information provided. This concept has applications in economics, political science, and beyond.

The Optimization Toolbox UI includes several MATLAB classes that implement different Bayesian persuasion strategies. These classes are:

- **BayesianPersuasion**: A general framework for Bayesian persuasion.
- **BayesianPersuasionArbitrary**: A strategy where the sender picks arbitrary hypotheses.
- **BayesianPersuasionMajority**: A strategy where the sender computes the majority hypothesis.
- **BayesianPersuasionRandomized**: A strategy where the sender picks random hypotheses.

3 BayesianPersuasion Class

Bayesian persuasion involves a sender who aims to influence the receiver's actions by sharing information about an underlying state of the world. The sender's goal is to design an optimal signaling strategy that maximizes their payoff, considering the receiver's response to the information provided. This concept has applications in economics, political science, and beyond.

The Optimization Toolbox UI includes several MATLAB classes that implement different Bayesian persuasion strategies. These classes are:

- **BayesianPersuasion**: A general framework for Bayesian persuasion.
- **BayesianPersuasionArbitrary**: A strategy where the sender picks arbitrary hypotheses.
- **BayesianPersuasionMajority**: A strategy where the sender computes the majority hypothesis.
- **BayesianPersuasionRandomized**: A strategy where the sender picks random hypotheses.

4 BayesianPersuasion Class

4.1 Overview

The `BayesianPersuasion` class provides a general framework for Bayesian persuasion. It defines the properties of the states, actions, sender's payoffs, receiver's payoffs, and the prior distribution over states. It also includes methods for finding the optimal signaling strategy and calculating the expected payoff for the sender.

4.2 Inputs and Outputs

- **Inputs:**
  - `states`: The possible states of the world.
  - `actions`: The actions available to the receiver.
  - `senderPayoff`: The sender's payoff matrix.
  - `receiverPayoff`: The receiver's payoff matrix.
  - `prior`: The prior distribution over the states.
- **Outputs:**
  - `sigma`: The optimal signaling strategy.
  - `rho`: The optimal response strategy.
  - `expectedPayoff`: The expected payoff for the sender given the strategies.

4.2.1 Algorithm

Algorithm 1 BayesianPersuasion: Finding Optimal Strategy

Initialize σ and ρ with random values.
for each state i **do**
    Normalize $\sigma(i, :)$ to sum to 1.
    Normalize $\rho(i, :)$ to sum to 1.
end for
return σ, ρ

Algorithm 2 BayesianPersuasion: Calculating Expected Payoff

Initialize $expectedPayoff$ to 0.
for each state i **do**
    for each action j **do**
        Update $expectedPayoff$ using `prior`(i), $\sigma(i, j)$, `senderPayoff`(i, j), and $\rho(j, i)$.
    end for
end for
return $expectedPayoff$

5 BayesianPersuasionArbitrary Class

5.1 Overview

The `BayesianPersuasionArbitrary` class implements a strategy where the sender selects arbitrary hypotheses until no counter-examples are found. This approach iteratively refines the hypothesis space based on counter-examples.

5.2 Inputs and Outputs

- **Inputs:**
  - `H`: The concept class matrix.
  - `P`: The probability distribution over the feature space.
- **Outputs:**
  - `h`: The selected hypothesis.
  - `x`: The counter-example provided by the teacher.

5.2.1 Algorithm

Algorithm 3 BayesianPersuasionArbitrary: Arbitrary Learning

Initialize i to 1.
while true **do**
  Select an arbitrary hypothesis h from H.
  Obtain a counter-example x from the teacher.
  if x is empty **then**
    break
  end if
  Remove inconsistent hypotheses from H.
  Increment i.
end while
return h, x

6 BayesianPersuasionMajority Class

6.1 Overview

The `BayesianPersuasionMajority` class implements a strategy where the sender computes the majority hypothesis. This method iteratively refines the hypothesis space based on majority votes and counter-examples.

6.2 Inputs and Outputs

- **Inputs:**
  - `H`: The concept class matrix.
  - `P`: The probability distribution over the feature space.
- **Outputs:**
  - `h`: The majority hypothesis.
  - `x`: The counter-example provided by the teacher.

6.2.1 Algorithm

Algorithm 4 BayesianPersuasionMajority: Majority Learning

while true **do**
  Compute the majority hypothesis h.
  Obtain a counter-example x from the teacher.
  if x is empty **then**
    break
  end if
  Remove inconsistent hypotheses from H.
end while
return h, x

7 BayesianPersuasionRandomized Class

7.1 Overview

The `BayesianPersuasionRandomized` class implements a strategy where the sender randomly picks hypotheses. This approach iteratively refines the hypothesis space based on random selections and counter-examples.

7.2 Inputs and Outputs

- **Inputs:**
  - `H`: The concept class matrix.
  - `P`: The probability distribution over the feature space.
- **Outputs:**
  - `h`: The randomly selected hypothesis.
  - `x`: The counter-example provided by the teacher.

7.2.1 Algorithm

Algorithm 5 BayesianPersuasionRandomized: Randomized Learning

while true **do**
  Randomly select a hypothesis h from H.
  Obtain a counter-example x from the teacher.
  if x is empty **then**
    break
  end if
  Remove inconsistent hypotheses from H.
end while
return h, x

8 Bandit Optimization

Bandit optimization algorithms are widely used in scenarios where decision-makers need to balance exploration and exploitation to maximize rewards. The Optimization Toolbox UI provides several MATLAB functions implementing bandit optimization algorithms for different scenarios.

9 banditLinearOptimization Function

9.1 Overview

The `banditLinearOptimization` function performs bandit optimization for linear functions. It optimizes the function f defined by A and b starting from x_0 using stepsize α for $maxIter$ iterations with perturbation parameter δ.

9.1.1 Algorithm

9.1.2 Inputs

- f - Function handle of the objective function.
- A - Matrix defining the linear function.
- b - Vector defining the linear function.
- $x0$ - Initial guess.
- α - Step size.
- $maxIter$ - Maximum number of iterations.
- δ - Perturbation parameter.

Algorithm 6 banditLinearOptimization

procedure BANDITLINEAROPTIMIZATION(f, A, b, $x0$, α, $maxIter$, δ)
  Inputs: f, A, b, $x0$, α, $maxIter$, δ
  Outputs: x, $history$
  $x \leftarrow x0$
  $history \leftarrow$ zeros($maxIter$, 1)
  for $k = 1$ to $maxIter$ **do**
    $u \leftarrow$ randn(n, 1)
    $u \leftarrow u/$norm(u) ▷ Unit vector
    $fk\_plus \leftarrow f(x + \delta \cdot u, A, b)$
    $fk\_minus \leftarrow f(x - \delta \cdot u, A, b)$
    $gradEstimate \leftarrow (fk\_plus - fk\_minus)/(2 \cdot \delta) \cdot u$
    $x \leftarrow x - \alpha \cdot gradEstimate$
    $history(k) \leftarrow f(x, A, b)$
  end for
end procedure

9.1.3 Outputs

- x - Optimized parameters.
- $history$ - History of function values.

10 banditOptimization Function

10.1 Overview

The `banditOptimization` function performs bandit optimization using gradient estimates. It optimizes the function f starting from $x0$ using stepsize α for $maxIter$ iterations with perturbation parameter δ.

10.1.1 Algorithm

Algorithm 7 banditOptimization

procedure BANDITOPTIMIZATION(f, $x0$, α, $maxIter$, δ)
  Inputs: f, $x0$, α, $maxIter$, δ
  Outputs: x, $history$
  $x \leftarrow x0$
  $history \leftarrow$ zeros($maxIter$, 1)
  for $k = 1$ to $maxIter$ **do**
    $u \leftarrow$ randn(n, 1)
    $u \leftarrow u/$norm(u) ▷ Unit vector
    $fk\_plus \leftarrow f(x + \delta \cdot u)$
    $fk\_minus \leftarrow f(x - \delta \cdot u)$
    $gradEstimate \leftarrow (fk\_plus - fk\_minus)/(2 \cdot \delta) \cdot u$
    $x \leftarrow x - \alpha \cdot gradEstimate$
      history(k) $\leftarrow f(x)$
  end for
end procedure

10.1.2 Inputs

- f - Function handle of the objective function.
- $x0$ - Initial guess.
- α - Step size.
- $maxIter$ - Maximum number of iterations.
- δ - Perturbation parameter.

10.1.3 Outputs

- x - Optimized parameters.
- $history$ - History of function values.

11 exp3 Function

11.1 Overview

The `exp3` function performs the EXP3 algorithm for the multi-armed bandit problem. It runs the EXP3 algorithm for T rounds with K arms and exploration parameter γ.

11.1.1 Algorithm

Algorithm 8 exp3

procedure EXP3(T, K, γ)
  Inputs: T, K, γ
  Outputs: $weights$, $history$
  $weights \leftarrow$ ones(K, 1)
  $history \leftarrow$ zeros(T, 2) ▷ Column 1: chosen arm, Column 2: received reward
  for $t = 1$ to T **do**
    $probs \leftarrow (1 - \gamma) \cdot (weights / \sum(weights)) + (\gamma / K)$
    $arm \leftarrow$ randsample($1 : K$, 1, true, $probs$)
    $reward \leftarrow$ banditReward(arm, t)
    $estimatedReward \leftarrow reward / probs(arm)$
    $weights(arm) \leftarrow weights(arm) \cdot \exp(\gamma \cdot estimatedReward / K)$
    $history(t, :) \leftarrow [arm, reward]$
  end for
end procedure

11.1.2 Inputs

- T - Number of rounds.
- K - Number of arms.
- γ - Exploration parameter.

11.1.3 Outputs

- $weights$ - Final weights of the arms.
- $history$ - History of chosen arms and rewards.

12 exp3WithEstimators Function

12.1 Overview

The `exp3WithEstimators` function performs the EXP3 algorithm using unbiased estimators. It runs the EXP3 algorithm for T rounds with K arms and exploration parameter γ using unbiased estimators for the rewards.

12.1.1 Algorithm

This function shares the same algorithmic structure as the `exp3` function, with the difference lying in the reward calculation, which employs unbiased estimators.

12.1.2 Inputs

- T - Number of rounds.
- K - Number of arms.

Algorithm 9 exp3WithEstimators

procedure EXP3WITHESTIMATORS(T, K, γ)
  Inputs: T, K, γ
  Outputs: $weights$, $history$
  $weights \leftarrow$ ones(K, 1)
  $history \leftarrow$ zeros(T, 2) ▷ Column 1: chosen arm, Column 2: received reward
  for $t = 1$ to T **do**
    $probs \leftarrow (1 - \gamma) \cdot (weights / \sum(weights)) + (\gamma / K)$
    $arm \leftarrow$ randsample($1 : K$, 1, true, $probs$)
    $reward \leftarrow$ banditRewardWithEstimators(arm, t)
    $estimatedReward \leftarrow reward / probs(arm)$
    $weights(arm) \leftarrow weights(arm) \cdot \exp(\gamma \cdot estimatedReward / K)$
    $history(t, :) \leftarrow [arm, reward]$
  end for
end procedure

- γ - Exploration parameter.

12.1.3 Outputs

- $weights$ - Final weights of the arms.
- $history$ - History of chosen arms and rewards.

13 Blackwell Approachability

The `approachability` function computes the approachability solution for a given matrix A and vector b.

13.0.1 Algorithm

Algorithm 10 approachability

procedure APPROACHABILITY(A, b)
  Inputs: A, b
  Output: x_{opt}
  $n \leftarrow$ size(A, 2)
  cvx_begin quiet
  variable $x(n)$
  minimize $\|A \cdot x - b\|$
  cvx_end
  $x_{\text{opt}} \leftarrow x$
end procedure

13.0.2 Inputs

- A - Matrix A.
- b - Vector b.

13.0.3 Outputs

- x_{opt} - Approachability solution.

13.1 polarCones Function

13.2 Overview

The `polarCones` function computes the polar cones for a given matrix A and vector b.

13.2.1 Algorithm

Algorithm 11 polarCones

procedure POLARCONES(A, b)
  Inputs: A, b
  Outputs: $cones$, $dualCones$
  $[m, n] \leftarrow$ size(A)
  $cones \leftarrow$ cell(n, 1)
  $dualCones \leftarrow$ cell(m, 1)
  for $i = 1$ to n **do**
    cvx_begin quiet
    variable $x(n)$
    minimize $\|A \cdot x - b\|$
    subject to $\|x(i+1 : \text{end})\|_2 \leq x(i)$
    cvx_end
    $cones\{i\} \leftarrow x$
  end for
  for $j = 1$ to m **do**
    cvx_begin quiet
    variable $y(m)$
    maximize $y' \cdot b$
    subject to $y' \cdot A \leq 0$
    $y(j) = 1$
    cvx_end
    $dualCones\{j\} \leftarrow y$
  end for
end procedure

Inputs

- A - Matrix A.
- b - Vector b.

13.2.2 Outputs

- $cones$ - Polar cones.
- $dualCones$ - Dual polar cones.

14 Boosting

The `adaboost` function implements the AdaBoost algorithm, which trains an ensemble of base learners using the given base learning algorithm.

14.0.1 Algorithm

14.0.2 Inputs

- `baseLearner` - A function handle for the base learning algorithm with signature @(X, y, weights).
- `X_train` - Training data features.
- `y_train` - Training data labels.
- `numRounds` - Number of boosting rounds.

14.0.3 Outputs

- `models` - Cell array of trained base learners.
- `alphas` - Weights for each base learner.

Algorithm 12 AdaBoost

procedure ADABOOST($baseLearner, X\_train, y\_train, numRounds$)
  $n \leftarrow$ size$(X_train, 1)$
  $weights \leftarrow$ ones$(n, 1)/n$ ▷ Initialize weights
  $models \leftarrow$ cell$(numRounds, 1)$
  $alphas \leftarrow$ zeros$(numRounds, 1)$
  for $t \leftarrow 1$ **to** $numRounds$ **do**
    $models\{t\} \leftarrow$ baseLearner$(X_train, y_train, weights)$ ▷ Train base learner
    $y\_pred \leftarrow$ predict$(models\{t\}, X_train)$ ▷ Predict with base learner
    $\epsilon_t \leftarrow \sum_{i=1}^{n}(weights(i) \cdot (y\_pred(i) \neq y\_train(i)))$ ▷ Compute weighted error
    $\alpha_t \leftarrow 0.5 \cdot \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$ ▷ Compute alpha
    $weights \leftarrow$ updateWeights$(weights, \alpha_t, y_train, y_pred)$ ▷ Update weights
    $models\{t\} \leftarrow models\{t\}$ ▷ Store model
    $alphas(t) \leftarrow \alpha_t$ ▷ Store alpha
  end for
  return $models, alphas$
end procedure

14.0.4 boostingByOCO Function

The `boostingByOCO` function implements boosting using Online Convex Optimization (OCO).

14.0.5 Algorithm

Algorithm 13 Boosting by OCO

procedure BOOSTINGBYOCO($baseLearner, X\_train, y\_train, numRounds$)
  $n \leftarrow$ size$(X_train, 1)$
  $weights \leftarrow$ ones$(n, 1)/n$ ▷ Initialize weights
  $models \leftarrow$ cell$(numRounds, 1)$
  for $t \leftarrow 1$ **to** $numRounds$ **do**
    $models\{t\} \leftarrow$ baseLearner$(X_train, y_train, weights)$ ▷ Train base learner
    $y\_pred \leftarrow$ predict$(models\{t\}, X_train)$ ▷ Predict with base learner
    $\epsilon_t \leftarrow \sum_{i=1}^{n}(weights(i) \cdot (y\_pred(i) \neq y\_train(i)))$ ▷ Compute weighted error
    $\alpha_t \leftarrow 0.5 \cdot \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$ ▷ Compute alpha
    $weights \leftarrow$ updateWeights$(weights, \alpha_t, y_train, y_pred)$ ▷ Update weights
    $models\{t\} \leftarrow models\{t\}$ ▷ Store model
  end for
  return $weights, models$
end procedure

14.0.6 Inputs

- `baseLearner` - A function handle for the base learning algorithm with signature @(X, y, weights).
- `X_train` - Training data features.
- `y_train` - Training data labels.
- `numRounds` - Number of boosting rounds.

14.0.7 Outputs

- `models` - Cell array of trained base learners.
- `alphas` - Weights for each base learner.

15 AdaBoost Algorithms

15.1 adaboost Function

Description The `adaboost` function implements the AdaBoost algorithm, which trains an ensemble of base learners using the given base learning algorithm.

Algorithm

Algorithm 14 adaboost

Initialize weights: $w_i = \frac{1}{n}$ for all $i = 1, \ldots, n$
for t = 1 to `numRounds` **do**
  Train base learner with current weights: $h_t = \texttt{baseLearner}(X, y, w)$
  Predict with base learner: $\hat{y} = h_t(X)$
  Compute weighted error: $\epsilon_t = \sum_{i=1}^{n} w_i \mathbf{1}(\hat{y}_i \neq y_i)$
  Compute alpha: $\alpha_t = 0.5 \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
  Update weights: $w_i \leftarrow w_i \exp(-\alpha_t y_i \hat{y}_i)$
  Normalize weights: $w \leftarrow \frac{w}{\sum w}$
end for
Return models and alphas: $\{h_t\}_{t=1}^{\texttt{numRounds}}, \{\alpha_t\}_{t=1}^{\texttt{numRounds}}$

Inputs

- `baseLearner` - A function handle for the base learning algorithm with signature @(X, y, weights).
- `X_train` - Training data features.
- `y_train` - Training data labels.
- `numRounds` - Number of boosting rounds.

Outputs

- `models` - Cell array of trained base learners.
- `alphas` - Weights for each base learner.

15.2 boostingByOCO Function

Description The `boostingByOCO` function implements boosting using Online Convex Optimization (OCO).

Algorithm

Algorithm 15 boostingByOCO

Initialize weights: $w_i = \frac{1}{n}$ for all $i = 1, \ldots, n$
for t = 1 to `numRounds` **do**
  Train base learner with current weights: $h_t = \texttt{baseLearner}(X, y, w)$
  Predict with base learner: $\hat{y} = h_t(X)$
  Compute weighted error: $\epsilon_t = \sum_{i=1}^{n} w_i \mathbf{1}(\hat{y}_i \neq y_i)$
  Compute alpha: $\alpha_t = 0.5 \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
  Update weights: $w_i \leftarrow w_i \exp(-\alpha_t y_i \hat{y}_i)$
  Normalize weights: $w \leftarrow \frac{w}{\sum w}$
end for
Return weights and models: $w, \{h_t\}_{t=1}^{\texttt{numRounds}}$

Inputs

- `baseLearner` - A function handle for the base learning algorithm with signature @(X, y, weights).
- `X_train` - Training data features.
- `y_train` - Training data labels.
- `numRounds` - Number of boosting rounds.

Outputs

- `weights` - Final weights for each base learner.
- `models` - Cell array of trained base learners.

15.3 contextualLearningModel Function

Description The `contextualLearningModel` function implements a contextual learning model using boosting.

Algorithm

Algorithm 16 contextualLearningModel

Initialize weights: $w_i = \frac{1}{n}$ for all $i = 1, \ldots, n$
for t = 1 to `numRounds` **do**
    Train base learner with current weights and context: $h_t = \texttt{baseLearner}(X, y, w, \texttt{contextFeature})$
    Predict with base learner: $\hat{y} = h_t(X)$
    Compute weighted error: $\epsilon_t = \sum_{i=1}^{n} w_i \mathbf{1}(\hat{y}_i \neq y_i)$
    Compute alpha: $\alpha_t = 0.5 \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
    Update weights: $w_i \leftarrow w_i \exp(-\alpha_t y_i \hat{y}_i)$
    Normalize weights: $w \leftarrow \frac{w}{\sum w}$
end for
Return model: $\{\texttt{contextFeature}, \{h_t\}_{t=1}^{\texttt{numRounds}}, \{\alpha_t\}_{t=1}^{\texttt{numRounds}}\}$

Inputs

- `baseLearner` - A function handle for the base learning algorithm with signature @(X, y, context).
- `contextFeature` - Contextual feature vector.
- `X_train` - Training data features.
- `y_train` - Training data labels.
- `numRounds` - Number of boosting rounds.

Outputs

- `model` - Trained contextual learning model.

15.4 onlineBoosting Function

Description The `onlineBoosting` function implements the Online Boosting algorithm.

Algorithm

Inputs

- `baseLearner` - A function handle for the base learning algorithm with signature @(X, y).
- `X_train` - Training data features.
- `y_train` - Training data labels.
- `numRounds` - Number of boosting rounds.

Algorithm 17 onlineBoosting

Initialize weights: $w_i = \frac{1}{n}$ for all $i = 1, \ldots, n$
for t = 1 to `numRounds` **do**
    Train base learner with current weights: $h_t = \texttt{baseLearner}(X, y)$
    Predict with base learner: $\hat{y} = h_t(X)$
    Compute weighted error: $\epsilon_t = \sum_{i=1}^{n} w_i \mathbf{1}(\hat{y}_i \neq y_i)$
    Compute alpha: $\alpha_t = 0.5 \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
    Update weights: $w_i \leftarrow w_i \exp(-\alpha_t y_i \hat{y}_i)$
    Normalize weights: $w \leftarrow \frac{w}{\sum w}$
end for
Return models and alphas: $\{h_t\}_{t=1}^{\texttt{numRounds}}, \{\alpha_t\}_{t=1}^{\texttt{numRounds}}$

Outputs

- `models` - Cell array of trained base learners.
- `alphas` - Weights for each base learner.

15.5 predictBoostedModel Function

Description The `predictBoostedModel` function predicts using a boosted model.

Algorithm

Algorithm 18 predictBoostedModel

Initialize aggregated predictions: $F(x) = 0$ for all x
for t = 1 to `numRounds` **do**
    Predict with base learner: $\hat{y}_t = h_t(X)$
    Aggregate predictions: $F(x) \leftarrow F(x) + \alpha_t \hat{y}_t$
end for
Return final prediction: $\hat{y} = \texttt{sign}(F(x))$

Inputs

- `model` - A struct containing the trained boosted model with fields 'models' and 'alphas'.
- `X` - Input data features.

Outputs

- `y_pred` - Predicted labels.

16 Approximating Linear Programs

16.1 approximatingLinearPrograms Function

Description The `approximatingLinearPrograms` function solves a linear programming problem approximately by optimizing the function $c'x$ subject to $A * x \leq b$ using an iterative method starting from an initial guess $x0$.

Algorithm

Inputs

- `A` - Constraint matrix.
- `b` - Constraint vector.

Algorithm 19 approximatingLinearPrograms

1: Initialize $x = x0$
2: Initialize history array to store objective values
3: **for** k = 1 to `maxIter` **do**
4: Calculate subgradient: $g = A(idx, :)'$ where $idx = \arg\max(A * x - b)$
5: Compute stepsize: $\alpha = \frac{1}{\sqrt{k}}$
6: Update x: $x = x - \alpha * (g + c)$
7: Project x onto feasible set: $x =$ `projectionOntoFeasibleSet`(x, A, b)
8: Calculate objective value: `history(k) = c' * x` norm(g) < `tol`
9: Break and store history up to iteration k
10: **end for**
11: Return solution x and history of objective values

- `c` - Cost vector.
- `x0` - Initial guess.
- `maxIter` - Maximum number of iterations.
- `tol` - Tolerance for convergence.

Outputs

- `x` - Solution to the linear programming problem.
- `history` - History of objective function values.

16.2 projectionOntoFeasibleSet Function

Description The `projectionOntoFeasibleSet` function projects the vector x onto the feasible set defined by $A * x \leq b$.

Algorithm

Algorithm 20 projectionOntoFeasibleSet

1: Solve quadratic program: $x =$ `quadprog`(`eye`$(length(x)), -x, A, b,$ `options`$)$
2: Return projected x

Inputs

- `x` - Vector to be projected.
- `A` - Constraint matrix.
- `b` - Constraint vector.

Outputs

- `x` - Projected vector.

17 Linear Programming Duality

17.1 linearProgrammingDuality Function

Description The `linearProgrammingDuality` function solves the primal and dual linear programming problems.

Algorithm

Inputs

Algorithm 21 linearProgrammingDuality

1: Solve primal problem: `[primal, , , , lambda] = linprog(c, A, b, [], [], [], [], options)`
2: Extract dual solution: `dual = lambda.ineqlin`
3: Return primal and dual solutions

- `c` - Cost vector.
- `A` - Constraint matrix.
- `b` - Constraint vector.

Outputs

- `primal` - Solution to the primal problem.
- `dual` - Solution to the dual problem.

18 Von Neumann Minimax Theorem

18.1 vonNeumannTheorem Function

Description The `vonNeumannTheorem` function verifies the von Neumann Minimax Theorem by returning the value of the game using von Neumann's theorem.

Algorithm

Algorithm 22 vonNeumannTheorem

1: Solve zero-sum game: `[ , , value] = zeroSumGame(A)`
2: Return value of the game

Inputs

- `A` - Payoff matrix.

Outputs

- `value` - Value of the game.

19 Zero-Sum Game

19.1 zeroSumGame Function

Description The `zeroSumGame` function solves a zero-sum game using linear programming.

Algorithm

Inputs

- `A` - Payoff matrix.

Outputs

- `x` - Mixed strategy for player 1.
- `y` - Mixed strategy for player 2.
- `value` - Value of the game.

Algorithm 23 zeroSumGame

1: Define sizes: $[m, n] = \texttt{size}(A)$
2: Solve for player 1:
- Define problem: $f = [-\mathbf{1}_m; 1]$
- Define constraints: $A1 = [A', -\mathbf{1}_n], b1 = \mathbf{0}_n$
- Define equality: $Aeq1 = [\mathbf{1}'_m, 0], beq1 = 1$
- Define bounds: $lb1 = [\mathbf{0}_m; -\infty]$
- Solve: $x1 = \texttt{linprog}(f, A1, b1, Aeq1, beq1, lb1)$

3: Solve for player 2:
- Define problem: $g = [\mathbf{1}_n; -1]$
- Define constraints: $A2 = [-A, \mathbf{1}_m], b2 = \mathbf{0}_m$
- Define equality: $Aeq2 = [\mathbf{1}'_n, 0], beq2 = 1$
- Define bounds: $lb2 = [\mathbf{0}_n; -\infty]$
- Solve: $y2 = \texttt{linprog}(g, A2, b2, Aeq2, beq2, lb2)$

4: Extract strategies and value:
- $x = x1(1 : m)$
- $y = y2(1 : n)$
- $value = x1(end)$

5: Return strategies x, y and game value

20 Constrained Gradient Descent

20.1 constrainedGradientDescent Function

Description The `constrainedGradientDescent` function performs constrained gradient descent optimization. It optimizes the function f with its gradient `gradf` starting from $x0$ using stepsize α for `maxIter` iterations, subject to the constraint specified by `constraintFunc`.

Algorithm

Algorithm 24 constrainedGradientDescent

1: Initialize $x = x0$
2: Initialize history array to store function values
3: **for** k = 1 to `maxIter` **do**
4: Update x: $x = x - \alpha * \texttt{gradf}(x)$
5: Project x onto the constraint set: $x = \texttt{constraintFunc}(x)$
6: Store function value: `history(k) = f(x)`
7: **end for**
8: Return optimized parameters x and history of function values

Inputs

- `f` - Function handle of the objective function.
- `gradf` - Function handle of the gradient of f.
- `x0` - Initial guess.
- `alpha` - Step size.
- `maxIter` - Maximum number of iterations.
- `constraintFunc` - Function handle of the constraint projection function.

Outputs

- `x` - Optimized parameters.
- `history` - History of function values.

21 Gradient Descent

21.1 gradientDescent Function

Description The `gradientDescent` function performs gradient descent optimization. It optimizes the function f with its gradient `gradf` starting from $x0$ using stepsize α for `maxIter` iterations.

Algorithm

Algorithm 25 gradientDescent

Initialize $x = x0$
Initialize history array to store function values
for k = 1 to `maxIter` **do**
    Update x: $x = x - \alpha * \texttt{gradf}(x)$
    Store function value: `history(k) = f(x)`
end for
Return optimized parameters x and history of function values

Inputs

- `f` - Function handle of the objective function.
- `gradf` - Function handle of the gradient of f.
- `x0` - Initial guess.
- `alpha` - Step size.
- `maxIter` - Maximum number of iterations.

Outputs

- `x` - Optimized parameters.
- `history` - History of function values.

22 Distance to Optimality

22.1 measureDistanceToOptimality Function

Description The `measureDistanceToOptimality` function measures the Euclidean distance to the optimal point. It computes the Euclidean distance between the current point x and the optimal point x_{opt}.

Algorithm

Algorithm 26 measureDistanceToOptimality

Compute Euclidean distance: `distance = norm(x - x_opt)`
Return distance

Inputs

- `x` - Current point.
- `x_opt` - Optimal point.

Outputs

- `distance` - Euclidean distance to the optimal point.

23 Polyak Stepsize

23.1 polyakStepsize Function

Description The `polyakStepsize` function computes the Polyak stepsize for gradient descent. It calculates the stepsize for the function f at point x given the optimal value f_{opt}.

Algorithm

Algorithm 27 polyakStepsize

Compute gradient: $\texttt{gradf} = \texttt{gradient}(f, x)$
Compute Polyak stepsize: $\alpha = \frac{f(x) - f_{\text{opt}}}{\|\texttt{gradf}(x)\|^2}$
Return stepsize α

Inputs

- `f` - Function handle of the objective function.
- `x` - Current point.
- `f_opt` - Optimal value of the function.

Outputs

- `alpha` - Polyak stepsize.

24 SVM Training

24.1 svmTraining Function

Description The `svmTraining` function trains a Support Vector Machine using gradient descent. It trains an SVM with regularization parameter λ using gradient descent for `maxIter` iterations and stepsize α.

Algorithm

Algorithm 28 svmTraining

Initialize weights w and bias b to zero
Initialize history array to store loss values
for iter = 1 to `maxIter` **do**
    for i = 1 to m **do** y(i) * (X(i, :) * w + b) < 1
        Update w: $w = w - \alpha * (\lambda * w - y(i) * X(i, :)')$
        Update b: $b = b + \alpha * y(i)$
        Update w: $w = w - \alpha * \lambda * w$
    end for
    Compute hinge loss: `loss = sum(max(0, 1 - y .* (X * w + b))) / m + (lambda / 2) * norm(w)2`
    Store loss: `history(iter) = loss`
end for
Return weights w, bias b, and history of loss values

Inputs

- `X` - Matrix of training examples (each row is a training example).
- `y` - Vector of labels (+1 or -1).
- `lambda` - Regularization parameter.
- `maxIter` - Maximum number of iterations.
- `alpha` - Step size.

Outputs

- `w` - Weight vector.
- `b` - Bias term.
- `history` - History of loss values.

25 Generalization and Learnability

25.1 generalizationAndLearnability Function

Description The `generalizationAndLearnability` function evaluates the generalization error of a model with different complexity levels. It trains the model on the training data and computes the training and validation error for different complexity levels.

Algorithm

Algorithm 29 Generalization and Learnability Evaluation

procedure GENERALIZATIONANDLEARNABILITY(model, $X_{\text{train}}, y_{\text{train}}, X_{\text{val}}, y_{\text{val}}$, complexity)
    trainError $\leftarrow$ zeros(length(complexity), 1)
    valError $\leftarrow$ zeros(length(complexity), 1)
    for $i = 1$ **to** length(complexity) **do**
        Train the model with current complexity level: $y_{\text{train\_pred}} \leftarrow \text{model}(X_{\text{train}}, y_{\text{train}}, X_{\text{train}}, \text{complexity}(i))$
        Compute training error: $\text{trainError}(i) \leftarrow \text{mean}(y_{\text{train\_pred}} \neq y_{\text{train}})$
        Predict on validation data: $y_{\text{val\_pred}} \leftarrow \text{model}(X_{\text{train}}, y_{\text{train}}, X_{\text{val}}, \text{complexity}(i))$
        Compute validation error: $\text{valError}(i) \leftarrow \text{mean}(y_{\text{val\_pred}} \neq y_{\text{val}})$
    end for
    return trainError, valError
end procedure

Inputs

- model - A function handle for the learning model with signature @(X, y, X_val, complexity).
- X_{train} - Training data features.
- y_{train} - Training data labels.
- X_{val} - Validation data features.
- y_{val} - Validation data labels.
- complexity - Vector of complexity levels.

Outputs

- trainError - Training error for each complexity level.
- valError - Validation error for each complexity level.

26 Statistical Learning Theory

26.1 statisticalLearningTheory Function

Description The `statisticalLearningTheory` function evaluates a model using training and testing data. It trains the model on the training data and computes the training and testing error.

Algorithm

Algorithm 30 Statistical Learning Theory Evaluation

procedure STATISTICALLEARNINGTHEORY(model, $X_{\text{train}}, y_{\text{train}}, X_{\text{test}}, y_{\text{test}}$)
    Train the model: $y_{\text{train\_pred}} \leftarrow \text{model}(X_{\text{train}}, y_{\text{train}}, X_{\text{train}})$
    Compute training error: trainError $\leftarrow \text{mean}(y_{\text{train\_pred}} \neq y_{\text{train}})$
    Predict on test data: $y_{\text{test\_pred}} \leftarrow \text{model}(X_{\text{train}}, y_{\text{train}}, X_{\text{test}})$
    Compute testing error: testError $\leftarrow \text{mean}(y_{\text{test\_pred}} \neq y_{\text{test}})$
    return trainError, testError
end procedure

Inputs

- model - A function handle for the learning model with signature @(X, y, X_val).
- X_{train} - Training data features.
- y_{train} - Training data labels.
- X_{test} - Testing data features.
- y_{test} - Testing data labels.

Outputs

- trainError - Training error.
- testError - Testing error.

27 AdaDelta Optimization

27.1 adadelta Function

Description The `adadelta` function implements the AdaDelta optimization algorithm. It performs optimization of the objective function f using the AdaDelta algorithm.

Algorithm

Algorithm 31 AdaDelta Optimization Algorithm

procedure ADADELTA(f, grad_f, $x0$, options)
  if nargin < 4 **then**
    options $\leftarrow$ struct()
  end if
  rho $\leftarrow 0.9$ ▷ decay rate
  epsilon $\leftarrow 1e-8$ ▷ small constant for numerical stability
  maxIter $\leftarrow 1000$ ▷ maximum number of iterations
  if isfield(options,$'rho'$) **then**
    rho $\leftarrow$ options.rho
  end if
  if isfield(options,$'epsilon'$) **then**
    epsilon $\leftarrow$ options.epsilon
  end if
  if isfield(options,$'maxIter'$) **then**
    maxIter $\leftarrow$ options.maxIter
  end if
  $x \leftarrow x0$ ▷ initialize
  $E_{\delta x} \leftarrow$ zeros(size(x)) ▷ accumulated squared deltas
  $E_{\delta x_t} \leftarrow$ zeros(size(x)) ▷ accumulated squared deltas (time varying)
  for $t = 1$ **to** maxIter **do** ▷ optimization loop
    $grad \leftarrow$ grad_f(x) ▷ compute gradient
    $grad\_sq \leftarrow$ grad.^{2} ▷ compute squared gradient
    $E_{\delta x} \leftarrow$ rho $\times E_{\delta x} + (1 -$ rho$) \times$ grad_sq ▷ compute exponentially decaying average of squared deltas
    $RMS_{\delta x} \leftarrow \sqrt{E_{\delta x} +$ epsilon$}$ ▷ compute RMS of squared deltas
    $RMS_{\delta x_t} \leftarrow \sqrt{E_{\delta x_t} +$ epsilon$}$ ▷ compute RMS of squared deltas (time varying)
    $\Delta x \leftarrow -\left(\frac{RMS_{\delta x_t}}{RMS_{\delta x}}\right) \times$ grad ▷ update parameters
    $x \leftarrow x + \Delta x$ ▷ compute exponentially decaying average of squared deltas (time varying)
    $E_{\delta x_t} \leftarrow$ rho $\times E_{\delta x_t} + (1 -$ rho$) \times (\Delta x)^2$
  end for
  $f_{\text{opt}} \leftarrow f(x)$ ▷ compute optimal function value
  return $x_{\text{opt}}, f_{\text{opt}}$
end procedure

Inputs

- f - Objective function.
- `grad_f` - Gradient of the objective function.
- $x0$ - Initial point.
- `options` - Optimization options (optional).

Outputs

- x_{opt} - Optimal point.
- f_{opt} - Optimal function value.

28 Adagrad Optimization

28.1 adagrad Function

Description The `adagrad` function implements the Adagrad optimization algorithm. It performs optimization of the objective function f using the Adagrad algorithm.

Algorithm

Algorithm 32 Adagrad Optimization Algorithm

procedure ADAGRAD(f, grad_f, $x0$, options)
  if nargin < 4 **then**
    options $\leftarrow$ struct()
  end if
  alpha $\leftarrow 0.01$ ▷ learning rate
  epsilon $\leftarrow 1e-8$ ▷ small constant for numerical stability
  maxIter $\leftarrow 1000$ ▷ maximum number of iterations
  if isfield(options,$'alpha'$) **then**
    alpha $\leftarrow$ options.alpha
  end if
  if isfield(options,$'epsilon'$) **then**
    epsilon $\leftarrow$ options.epsilon
  end if
  if isfield(options,$'maxIter'$) **then**
    maxIter $\leftarrow$ options.maxIter
  end if
  $x \leftarrow x0$ ▷ initialize
  $G \leftarrow$ zeros(size(x)) ▷ sum of squares of past gradients
  for $t = 1$ **to** maxIter **do** ▷ optimization loop
    $grad \leftarrow$ grad_f(x) ▷ compute gradient
    $G \leftarrow G +$ grad^2 ▷ accumulate squared gradient
    $x \leftarrow x -$ alpha $\times$ grad$/(\sqrt{G} +$ epsilon) ▷ update parameters
  end for
  $f_{\text{opt}} \leftarrow f(x)$ ▷ compute optimal function value
  return $x_{\text{opt}}, f_{\text{opt}}$
end procedure

Inputs

- f - Objective function.
- `grad_f` - Gradient of the objective function.
- $x0$ - Initial point.
- `options` - Optimization options (optional).

Outputs

- x_{opt} - Optimal point.
- f_{opt} - Optimal function value.

29 Adam Optimization

29.1 adam Function

Description The `adam` function implements the Adam optimization algorithm. It performs optimization of the objective function f using the Adam algorithm.

Algorithm

Algorithm 33 Adam Optimization Algorithm

procedure ADAM(f, grad_f, $x0$, options)
  if nargin < 4 **then**
    options $\leftarrow$ struct()
  end if
  alpha $\leftarrow 0.001$ ▷ learning rate
  beta1 $\leftarrow 0.9$ ▷ decay rate for 1st moment estimate
  beta2 $\leftarrow 0.999$ ▷ decay rate for 2nd moment estimate
  epsilon $\leftarrow 1e-8$ ▷ small constant for numerical stability
  maxIter $\leftarrow 1000$ ▷ maximum number of iterations
  if isfield(options,$'alpha'$) **then**
    alpha $\leftarrow$ options.alpha
  end if
  if isfield(options,$'beta1'$) **then**
    beta1 $\leftarrow$ options.beta1
  end if
  if isfield(options,$'beta2'$) **then**
    beta2 $\leftarrow$ options.beta2
  end if
  if isfield(options,$'epsilon'$) **then**
    epsilon $\leftarrow$ options.epsilon
  end if
  if isfield(options,$'maxIter'$) **then**
    maxIter $\leftarrow$ options.maxIter
  end if
  $x \leftarrow x0$ ▷ initialize
  $m \leftarrow$ zeros(size(x)) ▷ 1st moment estimate
  $v \leftarrow$ zeros(size(x)) ▷ 2nd moment estimate
  $t \leftarrow 0$
  while $t <$ maxIter **do** ▷ optimization loop
    $t \leftarrow t + 1$
    $grad \leftarrow$ grad_f(x) ▷ compute gradient
    $m \leftarrow$ beta1 $\times\, m + (1 -$ beta1$) \times$ grad ▷ update biased 1st moment estimate
    $v \leftarrow$ beta2 $\times\, v + (1 -$ beta2$) \times$ (grad^2) ▷ update biased 2nd moment estimate
    $m_{\text{hat}} \leftarrow m/(1 - \text{beta1}^t)$ ▷ correct bias in 1st moment estimate
    $v_{\text{hat}} \leftarrow v/(1 - \text{beta2}^t)$ ▷ correct bias in 2nd moment estimate
    $x \leftarrow x -$ alpha $\times\, m_{\text{hat}}/(\sqrt{v_{\text{hat}}} +$ epsilon) ▷ update parameters
  end while
  $f_{\text{opt}} \leftarrow f(x)$ ▷ compute optimal function value
  return $x_{\text{opt}}, f_{\text{opt}}$
end procedure

Inputs

- f - Objective function.
- `grad_f` - Gradient of the objective function.
- $x0$ - Initial point.
- `options` - Optimization options (optional).

Outputs

- x_{opt} - Optimal point.
- f_{opt} - Optimal function value.

30 AMSGrad Optimization

30.1 amsgrad Function

Description The `amsgrad` function implements the AMSGrad optimization algorithm. It performs optimization of the objective function f using the AMSGrad algorithm.

Algorithm

Algorithm 34 AMSGrad Optimization Algorithm

procedure AMSGRAD(f, grad_f, $x0$, options)
  if nargin < 4 **then**
    options $\leftarrow$ struct()
  end if
  alpha $\leftarrow 0.01$ ▷ learning rate
  beta1 $\leftarrow 0.9$ ▷ decay rate for 1st moment estimate
  beta2 $\leftarrow 0.999$ ▷ decay rate for 2nd moment estimate
  epsilon $\leftarrow 1e-8$ ▷ small constant for numerical stability
  maxIter $\leftarrow 1000$ ▷ maximum number of iterations
  if isfield(options,$'alpha'$) **then**
    alpha $\leftarrow$ options.alpha
  end if
  if isfield(options,$'beta1'$) **then**
    beta1 $\leftarrow$ options.beta1
  end if
  if isfield(options,$'beta2'$) **then**
    beta2 $\leftarrow$ options.beta2
  end if
  if isfield(options,$'epsilon'$) **then**
    epsilon $\leftarrow$ options.epsilon
  end if
  if isfield(options,$'maxIter'$) **then**
    maxIter $\leftarrow$ options.maxIter
  end if
  $x \leftarrow x0$ ▷ initialize
  $m \leftarrow$ zeros(size(x)) ▷ 1st moment estimate
  $v \leftarrow$ zeros(size(x)) ▷ 2nd moment estimate
  $v_{\text{hat}} \leftarrow$ zeros(size(x)) ▷ AMSGrad correction term
  $t \leftarrow 0$
  while $t <$ maxIter **do** ▷ optimization loop
    $t \leftarrow t + 1$
    $grad \leftarrow$ grad_f(x) ▷ compute gradient
    $m \leftarrow$ beta1 $\times\, m + (1 -$ beta1$) \times$ grad ▷ update biased 1st moment estimate
    $v \leftarrow$ beta2 $\times\, v + (1 -$ beta2$) \times$ (grad^2) ▷ update biased 2nd moment estimate
    $v_{\text{hat}} \leftarrow \max(v_{\text{hat}}, v)$ ▷ update AMSGrad correction term
    $x \leftarrow x -$ alpha $\times\, m/(\sqrt{v_{\text{hat}}} +$ epsilon$)$ ▷ update parameters
  end while
  $f_{\text{opt}} \leftarrow f(x)$ ▷ compute optimal function value
  return $x_{\text{opt}}, f_{\text{opt}}$
end procedure

Inputs

- f - Objective function.
- `grad_f` - Gradient of the objective function.
- $x0$ - Initial point.

- `options` - Optimization options (optional).

Outputs

- x_{opt} - Optimal point.

- f_{opt} - Optimal function value.

31 Eve Optimization

31.1 eve Function

Description The `eve` function implements the Eve optimization algorithm. It performs optimization of the objective function f using the Eve algorithm.

Algorithm

Algorithm 35 Eve Optimization Algorithm

procedure EVE(f, grad_f, $x0$, options)
  if nargin < 4 **then**
    options $\leftarrow$ struct()
  end if
  alpha $\leftarrow 0.001$ ▷ learning rate
  beta1 $\leftarrow 0.9$ ▷ decay rate for 1st moment estimate
  beta2 $\leftarrow 0.999$ ▷ decay rate for 2nd moment estimate
  epsilon $\leftarrow 1e-8$ ▷ small constant for numerical stability
  gamma $\leftarrow 0.01$ ▷ smoothing parameter
  maxIter $\leftarrow 1000$ ▷ maximum number of iterations
  if isfield(options,$'alpha'$) **then**
    alpha $\leftarrow$ options.alpha
  end if
  if isfield(options,$'beta1'$) **then**
    beta1 $\leftarrow$ options.beta1
  end if
  if isfield(options,$'beta2'$) **then**
    beta2 $\leftarrow$ options.beta2
  end if
  if isfield(options,$'epsilon'$) **then**
    epsilon $\leftarrow$ options.epsilon
  end if
  if isfield(options,$'gamma'$) **then**
    gamma $\leftarrow$ options.gamma
  end if
  if isfield(options,$'maxIter'$) **then**
    maxIter $\leftarrow$ options.maxIter
  end if
  $x \leftarrow x0$ ▷ initialize
  $m \leftarrow$ zeros(size(x)) ▷ 1st moment estimate
  $v \leftarrow$ zeros(size(x)) ▷ 2nd moment estimate
  $t \leftarrow 0$
  while $t <$ maxIter **do** ▷ optimization loop
    $t \leftarrow t + 1$
    $grad \leftarrow$ grad_f(x) ▷ compute gradient
    $m \leftarrow$ beta1 $\times\, m + (1 -$ beta1$) \times$ grad ▷ update 1st moment estimate
    $v \leftarrow$ beta2 $\times\, v + (1 -$ beta2$) \times$ (grad^2) ▷ update 2nd moment estimate
    $m_{\text{hat}} \leftarrow m/(1 - \text{beta1}^t)$ ▷ bias correction
    $v_{\text{hat}} \leftarrow v/(1 - \text{beta2}^t)$ ▷ bias correction
    $x \leftarrow x -$ alpha $\times\, (m_{\text{hat}}/(\sqrt{v_{\text{hat}}} +$ epsilon$) +$ gamma $\times$ grad) ▷ update parameters
  end while
  $f_{\text{opt}} \leftarrow f(x)$ ▷ compute optimal function value
  return $x_{\text{opt}}, f_{\text{opt}}$
end procedure

Inputs

- f - Objective function.
- `grad_f` - Gradient of the objective function.
- $x0$ - Initial point.
- `options` - Optimization options (optional).

Outputs

- x_{opt} - Optimal point.

- f_{opt} - Optimal function value.

32 Fast Adaptive Stochastic Function Accelerator (FASFA)

32.1 fasfa Function

Description The `fasfa` function implements the Fast Adaptive Stochastic Function Accelerator (FASFA) algorithm. It performs optimization of the objective function f using the FASFA algorithm.

Algorithm

Algorithm 36 FASFA Optimization Algorithm

procedure FASFA(f, grad_f, $x0$, options)
  if nargin < 4 **then**
    options $\leftarrow$ struct()
  end if
  alpha $\leftarrow 0.001$ ▷ learning rate
  mu $\leftarrow 0.8$ ▷ first order momentum decay estimate
  nu $\leftarrow 0.999$ ▷ second order momentum decay estimate
  epsilon $\leftarrow 1e-8$ ▷ small constant for numerical stability
  maxIter $\leftarrow 1000$ ▷ maximum number of iterations
  if isfield(options,$'alpha'$) **then**
    alpha $\leftarrow$ options.alpha
  end if
  if isfield(options,$'mu'$) **then**
    mu $\leftarrow$ options.mu
  end if
  if isfield(options,$'nu'$) **then**
    nu $\leftarrow$ options.nu
  end if
  if isfield(options,$'epsilon'$) **then**
    epsilon $\leftarrow$ options.epsilon
  end if
  if isfield(options,$'maxIter'$) **then**
    maxIter $\leftarrow$ options.maxIter
  end if
  $x \leftarrow x0$ ▷ initialize
  $m \leftarrow$ zeros(size(x)) ▷ first moment vector
  $n \leftarrow$ zeros(size(x)) ▷ second moment vector
  $t \leftarrow 0$ ▷ timestep/iteration
  while $t <$ maxIter **do** ▷ optimization loop
    $t \leftarrow t + 1$
    $grad \leftarrow$ grad_f(x) ▷ compute gradient
    $m \leftarrow$ mu $\times\, m + (1 -$ mu$) \times$ grad ▷ update biased first moment estimation
    $n \leftarrow$ nu $\times\, n + (1 -$ nu$) \times$ (grad^2) ▷ update biased second moment estimation
    $m_{\text{hat}} \leftarrow m/(1 - \text{mu}^t)$ ▷ compute raw moment estimates
    $n_{\text{hat}} \leftarrow n/(1 - \text{nu}^t)$
    $x \leftarrow x -$ alpha $\times\, m_{\text{hat}}/(\sqrt{n_{\text{hat}}} +$ epsilon) ▷ implement FASFA update rule
  end while
  $f_{\text{opt}} \leftarrow f(x)$ ▷ compute optimal function value
  return $x_{\text{opt}}, f_{\text{opt}}$
end procedure

Inputs

- f - Objective function.

- `grad_f` - Gradient of the objective function.
- $x0$ - Initial point.
- `options` - Optimization options (optional).

Outputs

- x_{opt} - Optimal point.
- f_{opt} - Optimal function value.

33 Gradient Descent with Momentum

33.1 gradientDescentWithMomentum Function

Description The `gradientDescentWithMomentum` function implements the Gradient Descent with Momentum optimization algorithm. It performs optimization of the objective function f using the Gradient Descent with Momentum algorithm.

Algorithm

Algorithm 37 Gradient Descent with Momentum Optimization Algorithm

procedure GRADIENTDESCENTWITHMOMENTUM(f, grad_f, $x0$, options)
  if nargin < 4 **then**
    options $\leftarrow$ struct()
  end if
  alpha $\leftarrow 0.01$ ▷ learning rate
  beta $\leftarrow 0.9$ ▷ momentum coefficient
  maxIter $\leftarrow 1000$ ▷ maximum number of iterations
  if isfield(options,$'alpha'$) **then**
    alpha $\leftarrow$ options.alpha
  end if
  if isfield(options,$'beta'$) **then**
    beta $\leftarrow$ options.beta
  end if
  if isfield(options,$'maxIter'$) **then**
    maxIter $\leftarrow$ options.maxIter
  end if
  $x \leftarrow x0$ ▷ initialize
  $v \leftarrow$ zeros(size(x)) ▷ momentum
  for $t = 1$ **to** maxIter **do** ▷ optimization loop
    $grad \leftarrow$ grad_f(x) ▷ compute gradient
    $v \leftarrow$ beta $\times\, v +$ alpha $\times$ grad ▷ update momentum
    $x \leftarrow x - v$ ▷ update parameters
  end for
  $f_{\text{opt}} \leftarrow f(x)$ ▷ compute optimal function value
  return $x_{\text{opt}}, f_{\text{opt}}$
end procedure

Inputs

- f - Objective function.
- `grad_f` - Gradient of the objective function.
- $x0$ - Initial point.
- `options` - Optimization options (optional).

Outputs

- x_{opt} - Optimal point.
- f_{opt} - Optimal function value.

34 Mini-Batch Gradient Descent with Momentum

34.1 miniBatchGradientDescentWithMomentum Function

Description The `miniBatchGradientDescentWithMomentum` function implements the Mini-Batch Gradient Descent with Momentum optimization algorithm. It performs optimization of the objective function f using the Mini-Batch Gradient Descent with Momentum algorithm.

Algorithm

Algorithm 38 Mini-Batch Gradient Descent with Momentum Optimization Algorithm

procedure MINIBATCHGRADIENTDESCENTWITHMOMENTUM(f, grad_f, $x0$, options)
  if nargin < 4 **then**
    options $\leftarrow$ struct()
  end if
  alpha $\leftarrow 0.01$ ▷ learning rate
  beta $\leftarrow 0.9$ ▷ momentum coefficient
  batchSize $\leftarrow 32$ ▷ batch size
  maxIter $\leftarrow 1000$ ▷ maximum number of iterations
  if isfield(options,$'alpha'$) **then**
    alpha $\leftarrow$ options.alpha
  end if
  if isfield(options,$'beta'$) **then**
    beta $\leftarrow$ options.beta
  end if
  if isfield(options,$'batchSize'$) **then**
    batchSize $\leftarrow$ options.batchSize
  end if
  if isfield(options,$'maxIter'$) **then**
    maxIter $\leftarrow$ options.maxIter
  end if
  $x \leftarrow x0$ ▷ initialize
  $v \leftarrow$ zeros(size(x)) ▷ momentum
  for $t = 1$ **to** maxIter **do** ▷ optimization loop
    rand_indices $\leftarrow$ randperm(length($x0$), batchSize) ▷ sample mini-batch
    grad_sum $\leftarrow$ zeros(size(x))
    for $i = 1$ **to** length(rand_indices) **do** grad $\leftarrow$ grad_f(x(:, rand_indices(i)))
      grad_sum $\leftarrow$ grad_sum + grad

      grad_avg $\leftarrow$ grad_sum/batchSize ▷ update momentum
      $v \leftarrow$ beta $\times$ v + alpha $\times$ grad_avg
      $x \leftarrow x - v$ ▷ update parameters

      $f_{\text{opt}} \leftarrow f(x)$ ▷ compute optimal function value
      return $x_{\text{opt}}, f_{\text{opt}}$
      =0

Inputs

- f - Objective function.
- `grad_f` - Gradient of the objective function.

- $x0$ - Initial point.
- `options` - Optimization options (optional).

Outputs

- x_{opt} - Optimal point.
- f_{opt} - Optimal function value.

35 Nesterov Momentum

35.1 nestrovMomentum Function

Description The `nestrovMomentum` function implements the Nesterov Momentum optimization algorithm. It performs optimization of the objective function f using the Nesterov Momentum algorithm.

Algorithm

Algorithm 39 Nesterov Momentum Optimization Algorithm

procedure NESTROVMOMENTUM(f, grad_f, $x0$, options)
  if nargin < 4 **then**
    options $\leftarrow$ struct()
  end if
  alpha $\leftarrow 0.01$ ▷ learning rate
  beta $\leftarrow 0.9$ ▷ momentum coefficient
  maxIter $\leftarrow 1000$ ▷ maximum number of iterations
  if isfield(options,$'alpha'$) **then**
    alpha $\leftarrow$ options.alpha
  end if
  if isfield(options,$'beta'$) **then**
    beta $\leftarrow$ options.beta
  end if
  if isfield(options,$'maxIter'$) **then**
    maxIter $\leftarrow$ options.maxIter
  end if
  $x \leftarrow x0$ ▷ initialize
  $v \leftarrow$ zeros(size(x)) ▷ velocity
  for $t = 1$ **to** maxIter **do** ▷ optimization loop
    lookahead_x $\leftarrow x -$ beta $\times\, v$ ▷ compute gradient at lookahead point
    grad $\leftarrow$ grad_f(lookahead_x)
    $v \leftarrow$ beta $\times\, v +$ alpha $\times$ grad ▷ update velocity
    $x \leftarrow x - v$ ▷ update parameters
  end for
  $f_{\text{opt}} \leftarrow f(x)$ ▷ compute optimal function value
  return $x_{\text{opt}}, f_{\text{opt}}$
end procedure=0

Inputs

- f - Objective function.
- `grad_f` - Gradient of the objective function.
- $x0$ - Initial point.
- `options` - Optimization options (optional).

Outputs

- x_{opt} - Optimal point.
- f_{opt} - Optimal function value.

36 RMSProp

36.1 rmsprop Function

Description The `rmsprop` function implements the RMSProp optimization algorithm. It performs optimization of the objective function f using the RMSProp algorithm.

Algorithm

Algorithm 40 RMSProp Optimization Algorithm

procedure RMSPROP(f, grad_f, $x0$, options)
  if nargin < 4 **then**
    options $\leftarrow$ struct()
  end if
  alpha $\leftarrow 0.001$ ▷ learning rate
  beta $\leftarrow 0.9$ ▷ decay rate
  epsilon $\leftarrow 1e-8$ ▷ small constant for numerical stability
  maxIter $\leftarrow 1000$ ▷ maximum number of iterations
  if isfield(options,$'alpha'$) **then**
    alpha $\leftarrow$ options.alpha
  end if
  if isfield(options,$'beta'$) **then**
    beta $\leftarrow$ options.beta
  end if
  if isfield(options,$'epsilon'$) **then**
    epsilon $\leftarrow$ options.epsilon
  end if
  if isfield(options,$'maxIter'$) **then**
    maxIter $\leftarrow$ options.maxIter
  end if
  $x \leftarrow x0$ ▷ initialize
  E_g $\leftarrow$ zeros(size(x)) ▷ running average of squared gradients
  for $t = 1$ **to** maxIter **do** ▷ optimization loop
    grad $\leftarrow$ grad_f(x) ▷ compute gradient
    E_g $\leftarrow$ beta $\times$ E_g $+ (1 -$ beta$) \times$ grad^2 ▷ compute squared gradient
    $x \leftarrow x -$ alpha $\times$ grad$/(\sqrt{\text{E_g}} +$ epsilon$)$ ▷ update parameters
  end for
  $f_{\text{opt}} \leftarrow f(x)$ ▷ compute optimal function value
  return $x_{\text{opt}}, f_{\text{opt}}$
end procedure

Inputs

- f - Objective function.
- `grad_f` - Gradient of the objective function.
- $x0$ - Initial point.
- `options` - Optimization options (optional).

Outputs

- x_{opt} - Optimal point.
- f_{opt} - Optimal function value.

37 Stochastic Gradient Descent with Momentum

37.1 stochasticGradientDescentWithMomentum Function

Description The `stochasticGradientDescentWithMomentum` function implements the Stochastic Gradient Descent with Momentum optimization algorithm. It performs optimization of the objective function f using the Stochastic Gradient Descent with Momentum algorithm.

Algorithm

Algorithm 41 Stochastic Gradient Descent with Momentum Optimization Algorithm

procedure STOCHASTICGRADIENTDESCENTWITHMOMENTUM(f, grad_f, $x0$, options)
  if nargin < 4 **then**
    options $\leftarrow$ struct()
  end if
  alpha $\leftarrow 0.01$ ▷ learning rate
  beta $\leftarrow 0.9$ ▷ momentum coefficient
  maxIter $\leftarrow 1000$ ▷ maximum number of iterations
  if isfield(options,$'alpha'$) **then**
    alpha $\leftarrow$ options.alpha
  end if
  if isfield(options,$'beta'$) **then**
    beta $\leftarrow$ options.beta
  end if
  if isfield(options,$'maxIter'$) **then**
    maxIter $\leftarrow$ options.maxIter
  end if
  $x \leftarrow x0$ ▷ initialize
  $v \leftarrow$ zeros(size(x)) ▷ momentum
  for $t = 1$ **to** maxIter **do** ▷ optimization loop
    rand_index $\leftarrow$ randi($[1, \text{length}(x0)]$) ▷ sample a random data point
    grad $\leftarrow$ grad_f($x(:, \text{rand\_index})$) ▷ compute gradient
    $v \leftarrow$ beta $\times\, v +$ alpha $\times$ grad ▷ update momentum
    $x \leftarrow x - v$ ▷ update parameters
  end for
  $f_{\text{opt}} \leftarrow f(x)$ ▷ compute optimal function value
  return $x_{\text{opt}}, f_{\text{opt}}$
end procedure

Inputs

- f - Objective function.
- `grad_f` - Gradient of the objective function.
- $x0$ - Initial point.
- `options` - Optimization options (optional).

Outputs

- x_{opt} - Optimal point.
- f_{opt} - Optimal function value.

38 Exponentially Weighted Online Convex Optimization

38.1 expWeightedOCO Function

Description The `expWeightedOCO` function performs exponentially weighted online convex optimization. It optimizes the objective function f with its gradient gradf starting from the initial guess $x0$ using the stepsize α for a maximum of maxIter iterations.

Algorithm

Algorithm 42 Exponentially Weighted Online Convex Optimization

procedure EXPWEIGHTEDOCO(f, gradf, $x0$, α, maxIter)
  $x \leftarrow x0$ ▷ Initialize parameters
  history $\leftarrow$ zeros(maxIter, 1) ▷ Initialize history of function values
  weights $\leftarrow$ ones(maxIter, 1) ▷ Initialize weights
  for $k = 1$ **to** maxIter **do**
    $f_k \leftarrow f(k, x)$ ▷ Current function value
    gradf$_k \leftarrow$ gradf(k, x) ▷ Current gradient
    weights$(k) \leftarrow \exp(-\alpha \cdot f_k)$ ▷ Update weight
    $x \leftarrow x - \alpha \cdot$ gradf$_k \cdot$ weights(k) ▷ Update parameters
    history$(k) \leftarrow f_k$ ▷ Store current function value in history
  end for
  return x, history
end procedure

Inputs

- f - Function handle of the objective function (sequence of functions).
- `gradf` - Function handle of the gradient of f (sequence of gradients).
- $x0$ - Initial guess.
- α - Step size.
- `maxIter` - Maximum number of iterations.

Outputs

- x - Optimized parameters.
- `history` - History of function values.

38.2 onlineGradientDescent Function

Description The `onlineGradientDescent` function performs online gradient descent optimization. It optimizes the objective function f with its gradient gradf starting from the initial guess $x0$ using the stepsize α for a maximum of maxIter iterations.

Algorithm

Algorithm 43 Online Gradient Descent

procedure ONLINEGRADIENTDESCENT(f, gradf, $x0$, α, maxIter)
    $x \leftarrow x0$ ▷ Initialize parameters
    history $\leftarrow$ zeros(maxIter, 1) ▷ Initialize history of function values
    for $k = 1$ **to** maxIter **do**
        $f_k \leftarrow f(k, x)$ ▷ Current function value
        gradf$_k \leftarrow$ gradf(k, x) ▷ Current gradient
        $x \leftarrow x - \alpha \cdot$ gradf$_k$ ▷ Update parameters
        history$(k) \leftarrow$f$_k$ ▷ Store current function value in history
    end for
    return x, history
end procedure

Inputs

- f - Function handle of the objective function (sequence of functions).
- `gradf` - Function handle of the gradient of f (sequence of gradients).
- $x0$ - Initial guess.
- α - Step size.
- `maxIter` - Maximum number of iterations.

Outputs

- x - Optimized parameters.
- `history` - History of function values.

39 Online Newton Step

39.1 onlineNewtonStep Function

Description The `onlineNewtonStep` function performs online Newton step optimization. It optimizes the objective function f with its gradient gradf and Hessian hessf starting from the initial guess $x0$ using the stepsize α for a maximum of maxIter iterations.

Algorithm

Algorithm 44 Online Newton Step

procedure ONLINENEWTONSTEP(f, gradf, hessf, $x0$, α, maxIter)
    $x \leftarrow x0$ ▷ Initialize parameters
    history $\leftarrow$ zeros(maxIter, 1) ▷ Initialize history of function values
    for $k = 1$ **to** maxIter **do**
        $f_k \leftarrow f(k, x)$ ▷ Current function value
        gradf$_k \leftarrow$ gradf(k, x) ▷ Current gradient
        hessf$_k \leftarrow$ hessf(k, x) ▷ Current Hessian
        $x \leftarrow x - \alpha \cdot$ (hessf$_k \backslash$gradf$_k$) ▷ Update parameters
        history$(k) \leftarrow f_k$ ▷ Store current function value in history
    end for
    return x, history
end procedure

Inputs

- f - Function handle of the objective function (sequence of functions).

- `gradf` - Function handle of the gradient of f (sequence of gradients).
- `hessf` - Function handle of the Hessian of f (sequence of Hessians).
- $x0$ - Initial guess.
- α - Step size.
- `maxIter` - Maximum number of iterations.

Outputs

- x - Optimized parameters.
- `history` - History of function values.

40 Stochastic Gradient Descent

40.1 stochasticGradientDescent Function

Description The `stochasticGradientDescent` function performs stochastic gradient descent optimization. It optimizes the objective function f with its gradient gradf starting from the initial guess $x0$ using the stepsize α for a maximum of maxIter iterations and batch size batchSize.

Algorithm

Algorithm 45 Stochastic Gradient Descent

procedure STOCHASTICGRADIENTDESCENT(f, gradf, $x0$, α, maxIter, batchSize)
  $x \leftarrow x0$ ▷ Initialize parameters
  history $\leftarrow$ zeros(maxIter, 1) ▷ Initialize history of function values
  $n \leftarrow$ numel(f) ▷ Total number of functions
  for $k = 1$ **to** maxIter **do**
    idx $\leftarrow$ randperm(n, batchSize) ▷ Randomly sample batch indices
    gradBatch $\leftarrow$ zeros(size(x)) ▷ Initialize gradient batch
    for $j = 1$ **to** batchSize **do**
      gradBatch $\leftarrow$ gradBatch + gradf(idx(j), x) ▷ Accumulate gradients
    end for
    gradBatch $\leftarrow$ gradBatch/batchSize ▷ Average gradient
    $x \leftarrow x - \alpha \cdot$ gradBatch ▷ Update parameters
    history(k) $\leftarrow$ mean(arrayfun(@(i)$f(i, x)$, idx)) ▷ Store current function value in history
  end for
  return x, history
end procedure

Inputs

- f - Function handle of the objective function (sequence of functions).
- `gradf` - Function handle of the gradient of f (sequence of gradients).
- $x0$ - Initial guess.
- α - Step size.
- `maxIter` - Maximum number of iterations.
- `batchSize` - Size of the batch for stochastic updates.

Outputs

- x - Optimized parameters.
- `history` - History of function values.

41 Conditional Gradient

41.1 conditionalGradient Function

Description The `conditionalGradient` function implements the conditional gradient (Frank-Wolfe) method. It optimizes the objective function f with its gradient gradf starting from the initial guess $x0$ using linear constraints defined by A and b for a maximum of `maxIter` iterations with a tolerance of `tol`.

Algorithm

Algorithm 46 Conditional Gradient

procedure CONDITIONALGRADIENT(f, gradf, $x0$, A, b, maxIter, tol)
  $x \leftarrow x0$ ▷ Initialize parameters
  history $\leftarrow$ zeros(maxIter, 1) ▷ Initialize history of function values
  for $k = 1$ **to** maxIter **do**
    gradient $\leftarrow$ gradf(x) ▷ Compute gradient
    $s \leftarrow$ linearOracle(gradient, A, b) ▷ Solve linear problem
    $\alpha \leftarrow \frac{2}{k+2}$ ▷ Line search stepsize
    $x \leftarrow (1 - \alpha) \cdot x + \alpha \cdot s$ ▷ Update parameters
    history(k) $\leftarrow f(x)$ ▷ Store current function value in history
    if $\|$gradf(x)$\| <$ tol **then** ▷ Check for convergence
      history $\leftarrow$ history($1 : k$) ▷ Trim history
      break
    end if
  end for
  return x, history
end procedure

Inputs

- f - Function handle of the objective function.
- `gradf` - Function handle of the gradient of f.
- $x0$ - Initial guess.
- A - Matrix defining the linear constraints.
- b - Vector defining the linear constraints.
- `maxIter` - Maximum number of iterations.
- `tol` - Tolerance for convergence.

Outputs

- x - Optimized parameters.
- `history` - History of function values.

41.2 linearOracle Function

Description The `linearOracle` function solves the linear problem to find the update direction in the conditional gradient method.

Algorithm The linear problem is solved using the `linprog` function, which is a linear programming solver.

Inputs

- `gradient` - Gradient vector.
- A - Matrix defining the linear constraints.
- b - Vector defining the linear constraints.

41.2.1 Output

- s - Update direction.

42 Conditional Gradient Projection-Free

42.1 conditionalGradientProjectionFree Function

Description The `conditionalGradientProjectionFree` function implements the projection-free conditional gradient method. It optimizes the objective function f with its gradient gradf starting from the initial guess $x0$ using linear constraints defined by A and b for a maximum of `maxIter` iterations with a tolerance of `tol`.

Algorithm

Algorithm 47 Conditional Gradient Projection-Free

procedure CONDITIONALGRADIENTPROJECTIONFREE(f, gradf, $x0$, A, b, maxIter, tol)
  $x \leftarrow x0$ ▷ Initialize parameters
  history $\leftarrow$ zeros(maxIter, 1) ▷ Initialize history of function values
  for $k = 1$ **to** maxIter **do**
    gradient $\leftarrow$ gradf(x) ▷ Compute gradient
    $s \leftarrow$ linearOracle(gradient, A, b) ▷ Solve linear problem
    $\alpha \leftarrow \frac{2}{k+2}$ ▷ Line search stepsize
    $x \leftarrow (1 - \alpha) \cdot x + \alpha \cdot s$ ▷ Update parameters
    history(k) $\leftarrow f(x)$ ▷ Store current function value in history
    if $\|\text{gradf}(x)\| <$ tol **then** ▷ Check for convergence
      history $\leftarrow$ history($1 : k$) ▷ Trim history
      break
    end if
  end for
  return x, history
end procedure

Inputs

- f - Function handle of the objective function.
- `gradf` - Function handle of the gradient of f.
- $x0$ - Initial guess.
- A - Matrix defining the linear constraints.
- b - Vector defining the linear constraints.
- `maxIter` - Maximum number of iterations.
- `tol` - Tolerance for convergence.

Outputs

- x - Optimized parameters.
- `history` - History of function values.

43 Linear Oracle

43.1 linearOracle Function

Description The `linearOracle` function solves the linear problem to find the update direction subject to linear constraints.

43.1.1 Function

Algorithm 48 Linear Oracle

procedure LINEARORACLE(gradient, A, b)
    $K \leftarrow$ length(b) ▷ Number of constraints
    $f \leftarrow$ gradient ▷ Objective function
    options $\leftarrow$ optimoptions$('linprog', 'Display', 'none')$ ▷ Options for solver
    $s \leftarrow$ linprog$(f, A, b, [], [], \text{zeros}(K, 1), [], \text{options})$ ▷ Solve linear problem
    return s ▷ Return update direction
end procedure

Inputs

- `gradient` - Gradient vector.
- A - Matrix defining the linear constraints.
- b - Vector defining the linear constraints.

43.1.2 Output

- s - Update direction.

44 Online Conditional Gradient

44.1 onlineConditionalGradient Function

Description The `onlineConditionalGradient` function implements the online conditional gradient method to optimize a function subject to linear constraints.

44.1.1 Function

Algorithm 49 Online Conditional Gradient

procedure ONLINECONDITIONALGRADIENT(f, gradf, $x0$, A, b, maxIter, tol, η)
    $x \leftarrow x0$ ▷ Initial guess
    history $\leftarrow$ zeros(maxIter, 1) ▷ History of function values
    for $k = 1$ to maxIter **do**
        gradient $\leftarrow$ gradf(x) ▷ Compute gradient
        $s \leftarrow$ linearOracle(gradient, A, b) ▷ Solve linear problem
        $x \leftarrow (1 - \eta) \cdot x + \eta \cdot s$ ▷ Update parameters
        history$(k) \leftarrow f(x)$ ▷ Record function value
        if $\|$gradf$(x)\| <$ tol **then** ▷ Check convergence
            history $\leftarrow$ history$(1 : k)$
            break
        end if
    end for
    return x, history ▷ Return optimized parameters and history
end procedure

Inputs

- f - Function handle of the objective function.
- gradf - Function handle of the gradient of the objective function.
- $x0$ - Initial guess.
- A - Matrix defining the linear constraints.

- b - Vector defining the linear constraints.
- maxIter - Maximum number of iterations.
- tol - Tolerance for convergence.
- η - Step size.

Outputs

- x - Optimized parameters.
- history - History of function values.

45 Adaptive Gradient Descent

45.1 adaptiveGradientDescent Function

Description The `adaptiveGradientDescent` function performs adaptive gradient descent optimization by adjusting the stepsize based on the adaptation parameter.

45.1.1 Function

Algorithm 50 Adaptive Gradient Descent

procedure ADAPTIVEGRADIENTDESCENT(f, gradf, $x0$, α, β, maxIter)
  $x \leftarrow x0$ ▷ Initial guess
  history $\leftarrow$ zeros(maxIter, 1) ▷ History of function values
  gradHist $\leftarrow$ zeros(size($x0$)) ▷ History of gradients
  for $k = 1$ to maxIter **do**
    $f_k \leftarrow f(k, x)$ ▷ Current function
    gradfk $\leftarrow$ gradf(k, x) ▷ Current gradient
    gradHist $\leftarrow$ gradHist + gradfk^2 ▷ Update gradient history
    adjAlpha $\leftarrow \alpha/(\sqrt{\text{gradHist}} + \beta)$ ▷ Adjust stepsize
    $x \leftarrow x -$ adjAlpha $\cdot$ gradfk ▷ Update parameters
    history(k) $\leftarrow f_k$ ▷ Record function value
  end for
  return x, history ▷ Return optimized parameters and history
end procedure

Inputs

- f - Function handle of the objective function.
- gradf - Function handle of the gradient of the objective function.
- $x0$ - Initial guess.
- α - Initial stepsize.
- β - Adaptation parameter.
- maxIter - Maximum number of iterations.

46 Follow The Perturbed Leader

46.1 followPerturbedLeader Function

Description The `followPerturbedLeader` function performs Follow-The-Perturbed-Leader optimization by updating the parameters based on the gradient with added perturbations.

46.1.1 Function

Algorithm 51 Follow The Perturbed Leader

procedure FOLLOWPERTURBEDLEADER(f, gradf, $x0$, α, σ, maxIter)
  $x \leftarrow x0$ ▷ Initial guess
  history $\leftarrow$ zeros(maxIter, 1) ▷ History of function values
  $G \leftarrow$ zeros(length($x0$), maxIter) ▷ Matrix to store gradients
  for $k = 1$ to maxIter **do**
    perturbation $\leftarrow \sigma \cdot$ randn(size($x0$)) ▷ Add perturbation
    $G(:, k) \leftarrow$ gradf(k, x) + perturbation ▷ Update gradients with perturbation
    sumG $\leftarrow$ sum($G(:, 1:k), 2$) ▷ Compute sum of gradients
    $x \leftarrow -(\alpha/k) \cdot$ sumG ▷ Update parameters
    history(k) $\leftarrow f(k, x)$ ▷ Record function value
  end for
  return x, history ▷ Return optimized parameters and history
end procedure

Inputs

- f - Function handle of the objective function.
- gradf - Function handle of the gradient of the objective function.
- $x0$ - Initial guess.
- α - Step size.
- σ - Perturbation parameter.
- maxIter - Maximum number of iterations.

Outputs

- x - Optimized parameters.
- history - History of function values.

47 Online Mirror Descent

47.1 onlineMirrorDescent Function

Description The `onlineMirrorDescent` function performs online mirror descent optimization by updating the parameters using a mirror map and its inverse.

47.1.1 Function

Algorithm 52 Online Mirror Descent

procedure ONLINEMIRRORDESCENT(f, gradf, $x0$, α, maxIter, mirrorMap, invMirrorMap)
  $x \leftarrow x0$ ▷ Initial guess
  history $\leftarrow$ zeros(maxIter, 1) ▷ History of function values
  for $k = 1$ to maxIter **do**
    $fk \leftarrow f(k, x)$ ▷ Current function
    $gradfk \leftarrow$ gradf(k, x) ▷ Current gradient
    $z \leftarrow$ mirrorMap(x) $- \alpha \cdot gradfk$ ▷ Mirror descent step
    $x \leftarrow$ invMirrorMap(z) ▷ Update parameters
    history(k) $\leftarrow fk$ ▷ Record function value
  end for
  return x, history ▷ Return optimized parameters and history
end procedure

Inputs

- f - Function handle of the objective function.
- gradf - Function handle of the gradient of the objective function.
- $x0$ - Initial guess.
- α - Step size.
- maxIter - Maximum number of iterations.
- mirrorMap - Mirror map function handle.
- invMirrorMap - Inverse mirror map function handle.

Outputs

- x - Optimized parameters.
- history - History of function values.

48 Regularized Follow-The-Leader (RFTL) Optimization

48.1 rftlAlgorithm Function

Description The `rftlAlgorithm` function performs Regularized Follow-The-Leader (RFTL) optimization by updating the parameters using a stepsize and regularization parameter.

48.1.1 Function

Algorithm 53 Regularized Follow-The-Leader (RFTL) Optimization

procedure RFTLALGORITHM(f, gradf, $x0, \alpha, \lambda$, maxIter)
  $x \leftarrow x0$ ▷ Initial guess
  history $\leftarrow$ zeros(maxIter, 1) ▷ History of function values
  $G \leftarrow$ zeros(length($x0$), maxIter) ▷ Matrix to store gradients
  for $k = 1$ to maxIter **do**
    $G(:, k) \leftarrow$ gradf(k, x) ▷ Compute gradient
    $sumG \leftarrow$ sum$(G(:, 1 : k), 2)$ ▷ Compute cumulative sum of gradients
    $x \leftarrow -\frac{\alpha}{k+\lambda} \cdot sumG$ ▷ Update parameters
    history$(k) \leftarrow f(k, x)$ ▷ Record function value
  end for
  return x, history ▷ Return optimized parameters and history
end procedure

Inputs

- f - Function handle of the objective function.
- gradf - Function handle of the gradient of the objective function.
- $x0$ - Initial guess.
- α - Step size.
- λ - Regularization parameter.
- maxIter - Maximum number of iterations.

Outputs

- x - Optimized parameters.
- history - History of function values.

49 Discussion

Optimization has become an versatile tool across multiple disciplines, including deep learning to finance and physics. As the complexity of problems in these domains continues to grow, the need for efficient and well-organized optimization toolboxes becomes increasingly needed [4].

The OCO is a range of algorithms made specifically for solving these optimization problems in dynamic, data-driven environments where decisions must be made sequentially without knowledge of future data. This is different from a purely stochastic proccess, which is completely memoryless. Among these OCO algorithms, Stochastic Gradient Descent (SGD) stands out as a workhorse, widely employed in machine learning, particularly with vast datasets or when the objective function consists of numerous individual loss functions. The efficiency stems from updating parameters based on gradients computed from mini-batches of data, making it well-suited for online learning scenarios [8, 4].

ExpWeightedOCO, on the other hand, is adept at handling changing data distributions or non-stationary environments. By assigning exponentially decreasing weights to historical data, it adapts to evolving conditions by prioritizing recent observations. This makes it valuable for scenarios where data patterns fluctuate over time [4].

Meanwhile, Online Gradient Descent (OGD) offers a fundamental approach in OCO, providing regret guarantees in convex online learning settings. OGD updates parameters iteratively by moving them in the direction of the negative gradient of the current loss function. It's a simple yet powerful algorithm applicable across a wide spectrum of problems, including online prediction and dynamic resource allocation [4].

For problems with strongly convex objectives and Lipschitz continuous Hessians, the Online Newton Step tends to to better. Leveraging gradient and curvature information, it achieves faster convergence compared to gradient-based methods, making it suitable for scenarios where such properties hold [4].

This toolbox further includes more modern solvers, such as Adam and AMSGrad. The former combines properties of AdaGrad and RMSProp, which are also in this toolbox, in order to achieve fast convergence. The latter was created for one dimensional and strongly convex settings, where Adam struggles to converge [6].

These algorithms collectively form a versatile toolbox for addressing a broad array of optimization challenges in online learning, ranging from simple convex problems to more intricate and structured optimization tasks. Their selection hinges on factors like the convexity of the objective function, the availability of gradient or Hessian information, and the computational resources at hand, ensuring tailored solutions to diverse optimization scenarios. However, there can be other tools built into the toolbox that can help with optimization and basic machine learning problems. [7]

There are many additions to this toolbox that are in the works. Hyperparameterization and learning rate scheduling are both techniques in machine learning and optimization that aimed at improving model performance and convergence. While they share the overarching goal of optimizing model parameters, they operate at different levels and address distinct aspects of the optimization process. In essence, they try to accomplish the same goal in two different ways [4, 2].

Hyperparameterization involves tuning the hyperparameters of a learning algorithm, such as regularization strength or kernel parameters in a support vector machine. One of the most effective use cases for this approach is with the deep convolutional neural networks, and furthermore on generative adversarial networks. Usually, this is an automated process aimed at finding the optimal configuration of hyperparameters that leads to improved model performance, generalization, and convergence. Hyperparameterization is crucial for fine-tuning models to specific datasets and problem domains, as it directly influences the model's capacity to learn and generalize from data [9].

Learning rate scheduling, on the other hand, focuses on adapting the learning rate during training to facilitate efficient optimization and convergence. It is the simpler approach of the two, and can have a low computational cost. The learning rate determines the stepsize taken in the parameter space during some gradient-based optimization algorithms. Learning rate scheduling techniques adjust the learning rate over time, often based on heuristics, to balance between rapid convergence in the early stages of training and fine-tuning towards the end. Common scheduling strategies include exponential decay, step decay, polynomial decay, cosine annealing, and adaptive methods like Adam and RMSprop. The most effective versions of this technique often involve geometrically annealing a repeating pattern [2].

While hyperparameterization and learning rate scheduling share the goal of improving optimization performance, they operate at different levels of abstraction and address different aspects of the optimization process. Hyperparameterization involves tuning model-specific parameters that influence the learning process [9], while learning rate scheduling focuses on dynamically adjusting the optimization algorithm's behavior during training [2]. Currently, both of these are being added to the toolbox.

In some scenarios, one of these can be better than the other, but that is largely depending on the nature of the problem and available resources. For instance, in settings with limited computational resources or when manual intervention is feasible, hyperparameterization may offer more flexibility and control over model performance. On the other hand, learning rate scheduling can be particularly beneficial in deep learning applications, where finding the optimal learning rate manually can be challenging due to the complexity of the model and the large-scale datasets involved. Adaptive methods like Adam and RMSprop automatically adjust the learning rate based on the gradient's statistics anyways, alleviating the need for extensive hyperparameter tuning [2, 9].

Approximately five years ago, Jagdeep Bhatia proposed new algorithms for machine learning. They use a principle called interactive machine learning using randomized counter examples. Among these algorithms, arbitrary learning stands out for its flexibility, allowing users to provide diverse forms of feedback or labels to guide the learning process. This approach is particularly beneficial in scenarios where traditional labeled datasets are scarce or expensive to obtain, enabling the model to learn from various sources of information, including domain knowledge and user preferences. Though these algorithms are somewhat outside the domain of the OCO, perhaps they could be used in conjunction with some of the optimization algorithms to improve convergence and produce interesting future results [1]. Thus, they would be an interesting addition to OCOBox.

50 Conclusion

We introduced a new MATLAB toolbox for online convex optimization. It covers various aspects of the OCO including Boosting, momentum based optimization, and bandit learning. Furthermore, it includes some algorithms on learning rate scheduling and interactive machine learning. This toolbox could hopefully grow into an open sourced community in the future, as we accrue more contributors starting at the University of Virginia. Currently, the toolbox is far from complete, and one of the first major things we plan to release is a GUI for the many algorithms included for a better, user-friendly experience. In the future, we plan to also develop a better sense of modularity between algorithms and schedulers, and add a better visualization support system. The current toolbox can be found at: `https://github.com/PhilipNaveen/OCOBox`

References

[1] Jagdeep Bhatia. *Simple and Fast Algorithms for Interactive Machine Learning with Random Counter-examples*. 2019. arXiv: 1810.00506 [cs.LG].

[2] Aaron Defazio et al. *When, Why and How Much? Adaptive Learning Rate Scheduling by Refinement*. 2023. arXiv: 2310.07831 [cs.LG].

[3] Shaddin Dughmi and Haifeng Xu. *Algorithmic Bayesian Persuasion*. 2016. arXiv: 1503.05988 [cs.GT].

[4] Elad Hazan. "Introduction to Online Convex Optimization". In: *Found. Trends Optim.* 2 (2016), pp. 157–325. URL: https://api.semanticscholar.org/CorpusID:30482768.

[5] Klaus Höllig and Jörg Hörner. "Matlab®". In: *Aufgaben und Lösungen zur Höheren Mathematik 1* (2020). URL: https://api.semanticscholar.org/CorpusID:239294939.

[6] Amri Omar et al. "The commonly used algorithms to optimize a neural network in supervised learning: Overview, and comparative study". In: *2021 International Conference on Digital Age & Technological Advances for Sustainable Development (ICDATA)* (2021), pp. 31–38. URL: https://api.semanticscholar.org/CorpusID:243946835.

[7] Warrren B Powell. "Reinforcement Learning and Stochastic Optimization". In: *Reinforcement Learning and Stochastic Optimization: A Unified Framework for Sequential Decisions* (2022). URL: https://api.semanticscholar.org/CorpusID:235632698.

[8] Shai Shalev-Shwartz. "Online Learning and Online Convex Optimization". In: *Found. Trends Mach. Learn.* 4 (2012), pp. 107–194. URL: https://api.semanticscholar.org/CorpusID:51730029.

[9] Li Yang and Abdallah Shami. "On Hyperparameter Optimization of Machine Learning Algorithms: Theory and Practice". In: *ArXiv* abs/2007.15745 (2020). URL: https://api.semanticscholar.org/CorpusID:220919678.