Homework 4 - BIOMATH 205

SIMON LEE

November 2023

1 Chapter 9

Q1 In a majorization-minimization algorithm with objective function $f(x)$ and surrogate function $g(x|x_n)$, show that the sequence $g(x_{n+1}|x_n)$ decreases.

Answer:
In order to show that the sequence $g(x_{n+1}|x_n)$ decreases, we need to demonstrate that

$$g(x_{n+1}|x_n) \leq g(x_n|x_{n-1})$$

Therefore we begin with the majorization condition which gives us at each iteration:

$$f(x_n) \leq g(x_n|x_{n-1})$$

Now lets consider the surrogate function at the next iterate

$$f(x_{n+1}) \leq g(x_{n+1}|x_n)$$

However since x_{n+1} is obtained by minimizing $g(x|x_n)$, we also have

$$g(x_{n+1}|x_n) \leq g(x_n|x_{n-1})$$

we now can combine these two inequalities to get

$$f(x_n) \leq g(x_{n+1}|x_n) \leq g(x_n|x_{n-1})$$

Therefore we have shown that the sequence $g(x_{n+1}|x_n)$ decreases or stays the same at each iteration. Since this condition holds, it implies a monotonic decrease in the surrogate function along the sequence of iterates.

Q7: Find a quadratic upper bound majorizing the function e^{-x^2} around the point x_n.
Answer: To find the a quadratic upper bound majorizing the function e^{-x^2} around the point x_n, we can use a Taylor series expansion. The Taylor series expansion for e^{-x^2} around the point x_n is given by:

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + O2 + ...$$

We now plug in our equations derived from below to get the following expansion

$$f(x_n) = e^{-(x_n^2)}$$

$$f'(x_n) = -2x_n e^{-(x_n^2)}$$

$$e^{-x^2} \approx e^{-(x_n^2)} - 2x_n e^{-(x_n^2)}(x - x_n) + O((x - x_n)^2) + ...$$

Now we find the quaratic upper bound by neglecting the higher order terms.

$$f(x) \leq g(x) = e^{-(x_n^2)} - 2x_n e^{-(x_n^2)}(x - x_n)$$

Lastly we define the quadratic upper bound function $Q(x)$ as follows:

$$Q(x) = e^{-(x_n^2)} - 2x_n e^{-(x_n^2)}(x - x_n)$$

This quadratic function majorizes e^{-x^2} around the point x_n.

Note that this is a local approximation and may not be a good global approximation for the entire domain. The accuracy of the approximation depends on how close x is to x_n.

Q8: For the function $f(x) = ln(1 + e^x)$, derive the majorization

$$f(x) \leq f(x_n) + f'(x_n)(x - x_n) + \frac{1}{8}(x - x_n)^2$$

by the quadratic upper bound principle

Answer:
we begin by finding the first and second derivative of $f(x)$

$$f'(x) = \frac{e^x}{1 + e^x}$$

$$f''(x) = \frac{e^x}{(1 + e^x)^2}$$

Next we identify the stationary point of $f''(x)$ and analyze their behavior to find a valid upper bound for $f''(x)$ by setting $f''(x) = 0$ to solve for x.

$$0 = \frac{e^x}{(1 + e^x)^2}$$

This equation is satisified when $e^x = 0$ which has no real solutions. Therefore there are no stationary points for f"(x).

Since there are no stationary points for $f''(x)$, we need to look at the behavior of $f''(x)$ as x approaches infinity and negative infinity.

$$\lim_{x\to\infty} f''(x) = \lim_{x\to\infty} \frac{e^x}{(1+e^x)^2} = 0$$

$$\lim_{x\to-\infty} f''(x) = \lim_{x\to-\infty} \frac{e^x}{(1+e^x)^2} = 0$$

This indicates that $f''(x)$ is bounded and a reasonable upper bound for $f''(x)$ is a constant value M. If we want to find a specific value M, we can choose a certain criteria from our problem statement. We can do this since there is no unique value for M and the majorization will be valid for any value $M \geq 0$. We therefore choose $M = \frac{1}{4}$ to derive and match the criteria of the original problem We then apply the quadratic upper bound principle to get

$$f(x) \leq f(x_n) + f'(x_n)(x - x_n) + \frac{1}{2} \cdot M(x - x_n)^2$$

If we plug in $M = \frac{1}{4}$ we will obtain

$$f(x) \leq f(x_n) + f'(x_n)(x - x_n) + \frac{1}{2} \cdot \frac{1}{4}(x - x_n)^2$$

$$f(x) \leq f(x_n) + f'(x_n)(x - x_n) + \frac{1}{8}(x - x_n)^2$$

which matches the original question hence completing the derivation

2 Chapter 10

Q1: Rewrite and test either k-means or k-nearest neighbors algorithm with l_1 distances subsituted for l_2.

Answer:
In this code I took your original kNN code from the book and subsituted the l2 distance (Euclidean) with the l1 distance (Manhattan or City Block). The code is as follows. In this version I just imported the cityblock built in distance from the Distances package.

```
using Distances, Statistics, StatsBase, Random
export knn

"""Performs k nearest neighbor classification with training data
Y. The classes should be numbered 1, 2,..."""
function knn(X::Matrix{T}, Y::Matrix{T}, class::Vector{Int},
  k::Int) where T <: Real
  testing = size(X, 2)
  predicted_class = zeros(Int, testing)
  distance = pairwise(cityblock, Y, X)  # L1 distance (Manhattan distance)
  for i = 1:testing # find k nearest neighbors
```

```
    perm = partialsortperm(distance[:, i], 1:k)
    predicted_class[i] = mode(class[perm]) # most common class
  end
  return predicted_class
end

# Set seed for reproducibility
seed_value = 123
Random.seed!(seed_value)

(training, testing, features) = (100, 10, 30)
X = randn(features, testing)
Y = randn(features, training)
(k, classes) = (3, 2)
class = rand(1:classes, training)

predicted_class = knn(X, Y, class, k)

>>>10-element Vector{Int64}:
 1
 2
 1
 2
 2
 1
 2
 1
 1
 1
```

However this felt a little too simple so I also manually implemented the Manhattan distance using the formula:

$$d(a, b) = \sum_{i=1}^{m} |a_i - b_i|$$

```
import Pkg; Pkg.add("StatsBase")
using Random, Statistics, StatsBase

export knn

function knn(X::Matrix{T}, Y::Matrix{T}, class::Vector{Int}, k::Int) where T <: Real
    testing = size(X, 2)
    predicted_class = zeros(Int, testing)

    # Manual calculation of L1 distance (Manhattan distance)
```

```
    distance = zeros(T, size(Y, 2), size(X, 2))
    for i = 1:size(Y, 2)
        for j = 1:size(X, 2)
            distance[i, j] = sum(abs.(Y[:, i] .- X[:, j]))
        end
    end

    for i = 1:testing # find k nearest neighbors
        perm = partialsortperm(distance[:, i], 1:k)
        predicted_class[i] = mode(class[perm]) # most common class
    end

    return predicted_class
end

# Set seed for reproducibility
seed_value = 123
Random.seed!(seed_value)

(training, testing, features) = (100, 10, 30)
X = randn(features, testing)
Y = randn(features, training)
(k, classes) = (3, 2)
class = rand(1:classes, training)

predicted_class = knn(X, Y, class, k)

>>> 10-element Vector{Int64}:
 1
 2
 1
 2
 2
 1
 2
 1
 1
 1
```

The outputs match based off a set seed indicating they both work in the same manner.

Q5: Describe and program a Naive Bayes classification algorithm for Gaussian Distributed features. Assume that the features are independently distributed.

Answer:

The Naive Bayes classification algorithm is a probabilistic machine learning method based on Bayes' theorem. When applied to data with Gaussian-distributed features, and assuming that these features are independently distributed, we have a variant known as Gaussian Naive Bayes.

Gaussian Distribution Assumption: The algorithm assumes that each class's feature follows a Gaussian (normal) distribution. This means that the probability density function of each feature in each class is given by the Gaussian distribution formula:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2})$$

- where x_i is a feature,
- y is the class,
- μ_y is the mean of the feature in class y
- σ_y^2 is the variance of the feature in class y

Independence Assumption: The "Naive" in Naive Bayes comes from the assumption that features are conditionally independent given the class label. Mathematically, this can be expressed as:

$$P(x_1, x_2, ..., x_n|y) = P(x_1|y) \cdot P(x_2|y) \cdot .. \cdot P(x_n|y)$$

Algotihm:

1. For each class y, compute the prior probability $P(y)$ which represents the probability of encountering class y in the absence of any information about the feature values.

$$P(y) = \frac{\text{Number of instances of class } y}{\text{Total number of instances}}$$

2. Calculate the class-conditional probability for each feature given the class using the Gaussian distribution formula.
3. Using Bayes' theorem, calculate the posterior probability of each class given the feature values.
4. Assign the class label with the highest posterior probability as the predicted class for the given set of feature values.

Below we see the code for the Gaussian Naive Bayes.

```
using Statistics, Distributions

function fit_naive_bayes(X_train, y_train)
    classes = unique(y_train)
    class_priors = Dict()
    class_means = Dict()
    class_variances = Dict()

    for c in classes
        # Filter training data for the current class
        X_c = X_train[y_train .== c, :]

        # Calculate class prior probability
        class_priors[c] = size(X_c, 1) / size(X_train, 1)

        # Calculate mean and variance for each feature in the current class
        class_means[c] = mean(X_c, dims=1)
        class_variances[c] = var(X_c, dims=1)
    end

    return class_priors, class_means, class_variances
end

function predict_naive_bayes(X, class_priors, class_means, class_variances)
    num_samples, num_features = size(X)
    num_classes = length(keys(class_priors))
    predictions = zeros(Int, num_samples)

    for i in 1:num_samples
        posterior_probs = zeros(Float64, num_classes)

        for c in 1:num_classes
            class_prior = class_priors[c]
            class_mean = class_means[c]
            class_variance = class_variances[c]

            likelihood = prod(
                (1 / sqrt(2 *  * class_variance[j])) *
                exp(-(X[i, j] - class_mean[j])^2 / (2 * class_variance[j]))
                for j in 1:num_features
            )

            posterior_probs[c] = class_prior * likelihood
        end

        predictions[i] = argmax(posterior_probs)
```

```
    end

    return predictions
end

# Function to generate synthetic binary classification data
function generate_data(num_samples, num_features)
    X = randn(num_samples, num_features)

    # Generate random means and variances for two classes
    class_means = randn(2, num_features)
    class_variances = abs.(randn(2, num_features)) .+ 1.0

    # Assign each sample to a class (1 or 2)
    y = rand(1:2, num_samples)

    # Generate synthetic data based on class means and variances
    for i in 1:num_samples
        class_idx = y[i]
        for j in 1:num_features
            X[i, j] += class_means[class_idx, j]
            X[i, j] *= class_variances[class_idx, j]
        end
    end

    return X, y
end

# Example usage:
# Set random seed for reproducibility
Random.seed!(123)

# Generate synthetic binary classification data with 100 samples and 2 features
num_samples_binary = 100
num_features_binary = 2

X_binary, y_binary = generate_data(num_samples_binary, num_features_binary)

# Split the data into training and testing sets
split_ratio_binary = 0.8
split_idx_binary = Int(round(split_ratio_binary * num_samples_binary))

X_train_binary = X_binary[1:split_idx_binary, :]
y_train_binary = y_binary[1:split_idx_binary]

X_test_binary = X_binary[split_idx_binary+1:end, :]
```

```
y_test_binary = y_binary[split_idx_binary+1:end]

# Fit the model using binary classification data
priors_, means_, variances_ = fit_naive_bayes(X_train_binary, y_train_binary)

# Make predictions on binary classification test data
predictions_binary = predict_naive_bayes(X_test_binary, priors_, means_, variances_)

# Evaluate the accuracy
accuracy_binary = sum(predictions_binary .== y_test_binary) / length(y_test_binary)
println("Classification Accuracy: $accuracy_binary")

>>> Binary Classification Data Accuracy: 0.75
```