Homework 8

1. *Imputing missing entries in data using a k-means data model.* In `impute_via_kmeans.json`, you will find a 750×2 matrix `X_train` and a 750×2-vector `X_test` consisting of training and test data. (There is no output data.) In this exercise you will fit k-means data models and impute missing entries in data, for $k = 5, 10, 15, 20, 25$. This will allow us to choose a suitable value of k, based on how well each data model does at imputing data on the test set.

   (a) Edit `k_means.jl` to fit a k-means data model to `X_train` for $k = 5, 10, 15, 20, 25$. The provided function `fit_kmeans(X, k)` fits k clusters to data matrix `X` and returns a $2 \times k$ matrix `theta`. Each column of `theta` is the center of one of the k clusters.

   (b) In `k_means.jl`, we additionally provide a 750×2 matrix `X_test_missing`. The matrix `X_test_missing` is identical to `X_test` except that one of the two entries of every row of `X_test_missing` has the value `NaN`, which stands for 'not a number', *i.e.*, missing data.

   Use each of the five data models created in part (a) to impute the missing values in `X_test_missing`, in a matrix called `X_test_imputed`. Evaluate the RMS imputation error for each data model, and plot it versus k.

   Suggest a choice of k based on this plot, with a very brief explanation.

   (c) Scatter plot `X_train` and `X_test`. Approximately how many clusters appear in the data?

   Julia hints. The function `isnan(value)` returns `true` if `value` is `NaN` and returns `false` otherwise. The function `argmin(v)` returns the index of vector `v` that corresponds to the smallest element of `v`. You can use these functions to iterate over the rows of `X_test_missing`, iterate over the two entries of the row to find the non-missing entry, find the center with the smallest distance to the non-missing entry, and fill the missing entry using the identified center. Do not be afraid to use nested `if` and `for` loops.

2. *Using PCA for data visualization.* In `pca.jl` we provide a function `pca(X, r)` that returns the rank-r PCA model of the data given in `X`. The `pca` function returns the first `r` archetypes of of a PCA data model on `X` as the columns of a $10101 \times r$ matrix. You will use this function to compress the data into $\mathbf{R}^2$ so it can be visualized.

   (a) The starter code in `pca.jl` loads a 995×10101 matrix `X` of genetic data from some African and African diaspora populations and a list of 995 strings `demographics` of population identifiers from `pca_data.json`. The row `X[i,:]` is the genetic data for the ith individual, and the string `demographics[i]` indicates the population to which they belong. These populations are listed below.

Population identifier	Population
ACB	African Caribbean in Barbados
GWD	Gambian in Western Division, The Gambia
ESN	Esan in Nigeria
MSL	Mende in Sierra Leone
YRI	Yoruba in Ibadan, Nigeria
LWK	Luhya in Webuye, Kenya
ASW	African Ancestry in Southwest US

   Note that some of these populations are similar to each other, for example ACB and ASW are both in the Americas, and ESN and YRI are both in Nigeria.

   Use the function `pca` to create a PCA data model with $r = 2$. You should use the PCA model to create a 995×2 matrix of compressed genetic data.

   (b) In `pca.jl`, call the provided function `plot_pca(points, labels)`. Pass the function your compressed data from part (a) for `points` and the list of demographics from `pca_data.json` for `labels`. The `plot_pca` function will produce a 2d scatter plot of the compressed data with the populations corresponding to distinct colors. Note that the population was not used in creating the original data matrix or the compressed one found from PCA.

   (c) Make a few brief comments on the results.

   Notes. Here we describe the data in a bit more detail. None of this is needed to solve the problem. The data is originally from the International Genome Sample Resource; we obtained it from CS 168 at Stanford taught by Prof. Greg Valiant. The raw data is available in the starter code in `igsr_africa.txt`. The first column of the raw data is an identifier, the second column is the individual's sex, the third column is the population identifier, and the remaining columns list the nucleobase ('A', 'G', 'C', or 'T') found at specific positions in an individual's DNA. Each column corresponds to a specific nucleotide position in the human genome. Each position can take only one of two values, either 'A' or 'T' or 'G' or 'C'). The matrix `X` you are given in `pca_data.json` was formed by embedding 'A's and 'G's as 1 and all 'T's and 'C's as 0, and then demeaning the columns.

3. *Gradient descent for regularized logistic regression.* On previous homework assignments, you fit linear models with logistic loss for binary classification using the provided `rerm_lin_reg` function. In this exercise you will write a similar function from scratch.

   In `opt_data.json`, you will find a 300×30 matrix `X` and a 300-vector `y`, which represents Boolean data and takes the two values ± 1. The goal of this problem is to find the vector $\theta^\star \in \mathbf{R}^{10}$ that minimizes the function

$$f(\theta) = \frac{1}{300} \sum_{i=1}^{300} \log(1 + e^{-y_i x_i^T \theta}) + \lambda \|\theta\|_2^2,$$

   with $\lambda = 10^{-2}$. (This is the logistic loss plus a sum-squares regularizer.)

   (a) The gradient method is as follows:

   i. Initialize θ^1 to be the zero vector in $\mathbf{R}^{10}$ and $h^1 = 1$.

   ii. For $k = 1, 2, \ldots, k^{\max}$:

   A. Set $\theta^{\text{tent}} = \theta^k - h^k \nabla f(\theta^k)$.

   B. If $f(\theta^{\text{tent}}) < f(\theta^k)$ then set $\theta^{k+1} = \theta^{\text{tent}}$ and $h^{k+1} = 1.2h^k$.

   C. Otherwise, set $\theta^{k+1} = \theta^k$ and $h^{k+1} = 0.5h^k$.

   iii. Return $\theta_1, \ldots, \theta^{k^{\max}+1}$.

   (We would normally only be interested in the last iterate, but here we want to see the progress of the method.)

   In `gradient.jl`, complete the `gradient_method` by implementing the gradient method as described here. While the derivative function $\nabla f(\theta)$ of $f(\theta)$ can be computed by hand, this is often not done in practice. Instead, libraries can be used to automatically compute the gradients of functions. Such tools are known as *autograds* or *autodiffs*. In Julia, the function `gradient(f, theta)[1]` evaluates the derivative of function `f` at point `theta`. Use the `gradient` function in your implementation of `gradient_method`; do not compute the gradient explicitly.

   (b) Run `gradient_method` on the given function f with $k^{\max} = 35$. Since f is convex the gradient method is non-heuristic; we are guaranteed that the iterates converge to a minimizer of f.

   Using the iterates $\theta_1, \ldots, \theta^{k^{\max}+1}$ output by `gradient_method`, plot the loss values $f(\theta^k)$ and the gradient norms $\|\nabla f(\theta^k)\|_2$ versus the iteration number k. You should see that the loss values are decreasing, and the gradient norms are tending toward zero. This indicates that the gradient method is approaching the exact solution.