HOMEWORK 1: Sampling in Python & R

In this assignment, I created a uniform pseudorandom number generator on the interval [0,1] using the linear congruential method. I set a seed as the last 4 digits of the current time in milliseconds and used the recursive update rule $X_{i+1} = (aX_i + b) \bmod M$, where M is a large integer, in this case $2^{31} - 1$. I checked the function visually by plotting a histogram of the uniformly distributed pseudo-random samples X_i, using 100,000 samples and 100 bins. I also plotted a scatterplot of (X_i, X_i+1) for all i in the sample set. This two-dimensional plot is a visual check that adjacent points in the distribution are uncorrelated.

I next created a function to sample from an exponential distribution, using the inversion method. In the inversion method, we first generate a uniform distribution U on [0,1] and then let $X = F^{-1}(U)$, where F is the probability distribution we wish to sample from. I checked the function visually by plotting a histogram with 200 bins of 100,000 sampled points, which can be seen to follow an exponential distribution.

Thirdly, I built a function to sample from a 2-dimensional Gaussian distribution, with mean (0,0) and standard deviation of 1, using the polar method. In the polar method, we sample from the normal distribution by first generating independent uniform distributions U_1, U_2 on [0,1], then computing $\theta = 2\pi U_1$ and $R = \sqrt{-2\log(1 - U_2)}$, and then letting $X = R\cos(\theta)$ and $Y = R\sin(\theta)$. I checked the function visually by plotting a scatterplot of the distribution of 100,000 samples in two dimensions, and plotting a histogram of 100,000 samples of $T = R^2/2$, where R is the distance of a point from the origin.

Fourthly, I wrote code for Monte Carlo computation of π, by generating (X_t, Y_t) from the unit square $[0, 1]^2$, and computing the frequency that the points fall below $x^2 + y^2 = 1$. I then used a Monte Carlo method to compute the volume of a d-dimensional unit ball, for $d = 2, 3, 5, 10$, and compared the results to the theoretical values.

<u>Functions:</u>

sample_uniform(low=0, high=1)
: Samples pseudorandom number from a uniform distribution from low to high, using the linear congruential method. Plots scatter plot.
Output: None

sample_exponential(k=1)
: Samples from an exponential distribution, using the inversion method. Plots histogram.
Output: None

sample_normal(mean=0,var=1)
: Samples from a normal distribution, using the polar method. Plots scatter plot, histogram.
Output: None

monte_carlo(d=2)
: Samples from two independent uniform distribution and computes a Monte Carlo computation of π. Computes the volume of a ball of dimension d. Print results.
Output: None

HOMEWORK 2: Metropolis & Gibbs Sampling in Python and R

I implemented a Monte Carlo Markov Chain (MCMC) sampling method to sample a target distribution $\pi(x)$. At time step t and state X_t, we generate X_t+1 by modifying X_t, so that $P(X_t+1 = y \mid X_t = x, X_t-1, ..., X0) = K(x, y)$, where $K(x, y)$ is the transition probability, and where we assume the Markov Property: $P(X_t+1 = y \mid X_t = x, X_t-1, ..., X0) = P(X_t+1 = y \mid X_t = x)$. The Metropolis Algorithm is an MCMC algorithms which is implemented as follows: at time t, let the current state be $X_t = x$. We generate $U \sim$ uniform[0,1]. If $U \leq \pi(y)/\pi(x)$, then we let $X_t+1 = y$, otherwise we let $X_t+1 = x$. I used the Metropolis algorithm to sample from $N(0, 1)$. The proposal distribution at x is $y \sim$ Uniform[x-c, x+c]. I ran 1,000 parallel chains with $x_0 \sim$ uniform[a, b] and made a movie for the change of the histogram of x_t. I also experimented with different values of [a, b] and c.

I next implemented a Gibbs sampler (another MCMC algorithm). The Gibbs sampler samples from a multivariate distribution $\pi(x)$, where $x = (x_1,...,x_k,...,x_d)$. A step k, the algorithm updates the state $x_k \sim \pi(x_k|x_{-k})$, where $x-k$ denotes the current values of all the other components. In the case of a bivariate normal distribution, letting (X_t, Y_t) be the values of (X, Y) at iteration t, at iteration $t + 1$ we sample $X_{t+1} \sim N(\rho Y_t, 1 - \rho^2)$, and then sample $Y_{t+1} \sim N(\rho X_{t+1}, 1 - \rho^2)$. I ran a visual check by running 1000 parallel chains from the same starting point and making a movie of the scatterplot of (x_t, y_t). I also ran a single chain for T steps, discarding the first B steps, and plotted the scatterplot of the footsteps for the rest of the steps. Finally, I experimented with different values of ρ, and demonstrated that the sampled density collapses to a point when $|\rho| = 1$.

Functions:

sample_uniform(size=1000, low=0, high=1)
- Samples from a uniform distribution.
- Ouput: An array of uniform random values

sample_normal_chain(x0, c, chain_length=100, mean=0, var=1)
- Returns multiple chains by Metropolis sampling from N(mean, var).
- For every train, the proposal distribution at x is $y \sim$ Uniform[x-c, x+c].
- Output: An array of sampled values

metropolis_simulation (num_chain=1000, chain_length=100, mean=0, var=1)
- Simulates the metropolis algorithm with different settings. Shows a movie.
- Output: None

gibbs_sample(x0, y0, rho, num_chain=1000, chain_length=100, mean=0, var=1)
- Returns multiple chains by Gibbs sampling.
- Output: Array of Gibbs samples

gibbs_simulation()
- Parameters are inside function. Simulates Gibbs sampling with different ρ and plots.
- Output: None

HOMEWORK 3: The Sweep Operator

I implemented a sweep operator function in R, which takes an input matrix A and value k and outputs a swept matrix B, with the pivot element A_{kk}. The sweep operator is a fundamental operator in regression which performs elementary row operators on matrix equations. The algorithm can be executed efficiently in Python or R by vectorizing the operations in the following manner: We define the pivot element $m = A_{kk}$, and then compute the negative outer product of A's kth column $A_{:k}$ and row $A_{k:}$ divided by m: $B = A - (A_{:k} \otimes A_{k:})/m$. We then overwrite the kth row of B as $B_{k:} = A_{k:}/m$, we overwrite the kth row of B as $B_{:k} = -B_{:k}/m$, and we overwrite the kkth element of B as $B_{kk} = 1/m$. This ordering of operations allows the matrix B to overwritten in place without the use of placeholder variables.

The sweep operator allows us to efficiently compute least squares estimates and build regression models. Given a data matrix X and label vector Y, we seek to build a model of the form $Y = X\beta + \varepsilon$. By "sweeping in" or "sweeping out" certain rows of X^TX, the sweep operator allows us to simultaneously update our estimates of the regression coefficients, the error sum of squares, and the Moore-Penrose pseudoinverse of the data matrix. We first construct the uncorrected sum of squares and cross-products matrix M (the USSCP matrix), which in block form contains the submatrices X^TX, X^TY, Y^TX, and Y^TY. We then sweep M along the rows of X^TX. Our regression coefficient estimates are the upper right block of the swept matrix.

Functions:

myLinearRegression(X, Y)

Computes linear regression coefficient estimates, given data X and labels Y.
Output: β estimates

HOMEWORK 4: QR Decomposition & Linear Regression

I built a function, *myQR*, which given an input matrix A, computes an upper triangular matrix T and a unitary matrix U such that $A = UTU^*$ is the Schur decomposition of A. The function $A_0 = A$ and $U_0 = I$, computes the QR decomposition of A_{k-1} as $A_{k-1} = Q_kR_k$ (Recall that the QR decomposition produces an orthogonal matrix Q and an upper triangular matrix R), and then applies the recursion relations $A_k = R_kQ_k$ and $U_k = U_{k-1}Q_k$. In the limit that k goes to infinity, A_k and U_k approach T and U, respectively.

To improve the speed of the algorithm, we first reduce the matrix A to Hessenberg form, that is a matrix whose elements below the lower off-diagonal are zero. The Hessenberg form is preserved by the QR algorithm described above, and introducing the lower off-diagonal zeros results in significant computational savings. A matrix can be reduced to Hessenberg form using Givens Rotations or Householder Reflections. If A is $n \times n$, $n-1$ Givens Rotations are required to transform the matrix to upper diagonal form. After transforming A to an upper Hessenberg matrix H, the algorithm performs the recursion in the previous paragraph by employing Givens Rotations to overwrite H with $H' = RQ$, where $H = QR$ is the QR factorization of H.

Alternatively, we can use Householder Reflections. A Householder Reflector is a matrix of the form $P = I - 2uu^*$, $||u|| = 1$. Householder Reflectors are Hermitian and unitary. We can reduce a

matrix A to Hessenberg form by repeated application of Householder Reflections. We first perform the recursion $P_k = I_k \oplus (I_{n-k} - 2u_k u_k^*)$ and then update $A = AP_k$ and $U = P_k U$.

Finally, we can use our QR decomposition algorithm to efficiently perform linear regression. Given input data matrix X and labels Y, we first compute the QR decomposition of X. We then solve the equation $R\beta = Q^T Y$. Since R is upper triangular, this equation can be efficiently solving from the bottom row of the matrix to the top. In particular, we solve for the last (the pth) element of β as $\beta_p = (Q^T Y)_p / R_{pp}$ and then solve the recursion relation: $\beta_i = [(Q^T Y)_i - R_{i,i+1:p}\, \beta_{i+1:p}]/R_{ii}$.

Functions:

givens(a,b)

Computes Givens rotation matrix.
Output: Given rotation matrix.

myQR(A)

Performs QR decomposition on the matrix A.
Ouput: A list containing the matrices Q and R.

myLinearRegression(X, Y)

Perform the linear regression on data matrix X and label vector Y.
Output: Estimated β and mean squared error.

HOMEWORK 5: Eigen-Decomposition and PCA

QR decomposition can be used to efficiently compute eigenvectors and eigenvalues. The algorithm is as follows: Initialize $A_i = A$ and $U_i = I$. Compute the QR decomposition of $A_i = QR$ and recursively update A_i and U_i using the relations $A_i = RQ$ and $U_i = U_i Q$. As the number of recursions tends toward infinity, A_i and U_i converge, respectively, to a diagonal matrix of the eigenvalues D and a matrix Q of eigenvectors.

Using our eigen-decomposition, we can efficiently compute a principal component analysis (PCA). Given a data matrix $X \in \mathrm{R}^{n \times p}$, we first compute μ, an array of the means along the columns of X, and then compute $B = X - X_\mu$, where $X_\mu = 1^T \mu$. We then compute the QR decomposition of $C = B^T B/(n-1)$ and compute $Z = XQ$.

Functions:

myEigen_QR(A, numIter = 1000)

Computes eigenvectors & eigenvalues for A using myQR.
Output: A list of eigenvectors and eigenvalues

myPCA(X)

Performs PCA on matrix X using myEigen_QR().
Output: A basis matrix Q and data matrix Z, such that $X = ZQ^T$.

HOMEWORK 6: Logistic Regression, Adaboost, & XGBoost

Using the *myQR* function for QR decomposition from Homework 5, I built a function *myLogisticSolution* to perform logistic regression. Given input data X and labels Y, the function runs the Newton-Raphson algorithm to compute estimates of β in a logistic model, with β

initialized as a zero vector. At each iteration, the algorithm first computes the variance matrix $V_m = \text{diag}(p(1-p))$, where $p = \exp(X\beta)/(1+\exp(X\beta))$ and then computes β with the recursion relation $\beta_{i+1} = \beta_i + (X^T V_m X)^{-1}(X^T Y - p)$. We can compute the inverse of $M = X^T V_m X$ efficiently by using the *myQR* function to compute the QR decomposition of M. Then, $M^{-1} = R^{-1}Q^T$. We can compute R^{-1} recursively, using the relation $R^{-T}_{ik} = -(R^T_{i,k:(i-1)} R^{-T}_{k:(i-1),k})/R^T_{ii}$. The function *myLogisticSolution* inputs X and Y and outputs the estimates of β.

I next implemented Adaptive Boosting (AdaBoost) in the *myAdaboost* function. The AdaBoost algorithm constructs a boost classifier of the form $F_T(x) = \sum_{t=1}^{T} f_t(x)$, where the f_t are weak classifiers that take inputs x and return predicted classes. The algorithm functions as follows: at each iteration, the function *ensemble* constructs an ensemble of linear classifiers $\{k_i\}$ with slope and y-intercept values sampled from a uniform distribution. The algorithm chooses the classifier k_m that minimizes the total weighted error $\sum_{y_i \neq k_m(x_i)} w_i^{(m)}$, uses this to compute the error rate $\varepsilon_m = \sum_{y_i \neq k_m(x_i)} w_i^{(m)} / \sum_{i=1}^{N} w_i^{(m)}$, and then calculates the classifier weight $\alpha_m = \frac{1}{2} \ln\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right)$. Finally, the classifier C_{m-1} is boosted to $C_m = C_{m-1} + \alpha_m k_m$.

I then implemented Extreme Gradient Boosting (XGBoost) in the *myXGBoost* function. This function divides the plane into four randomly drawn quadrants. It then constructs a one-layer decision tree on the quadrants. As in AdaBoost, we add a decision tree to the main classifier by determining the weight that would minimize a loss function.

Functions:

myLogisticSolution(X, Y)
- Performs logistic regression on input data matrix X and label vector Y.
- Output: Estimated β coefficients.

myAdaboost(x1, x2, y)
- Performs Adaptive Boosting. Plots the classification results and prints accuracy.
- Output: None
- grad(X,Y,W,b)
  - Calculates the gradient.
  - Output: Returns db, the gradient with respect to b.
- gradient_descent(X,Y,W,b,l_rate,num_iter)
  - Performs gradient descent on b.
  - Output: Updated b.
- ensemble(X,Y,range,n_ensmb)
  - Creates a random ensemble of weak linear classifiers.
  - Output: Ct, a list of W, b, and α values for each classifier in the ensemble.
- predict(X,Y,C)
  - Given data matrix X and classifier C, this function predicts the labels.
  - Output: Yp, an array of predicted labels.
- L_exp(X,Y,C,At,s=1)
  - Computes the exponential loss function.
  - Output: Exponential loss.
- Choose(X,Y,C,Ct)
  - Chooses the weak classifier that minimizes the total weighted error.

Output: At, a list of the *W* and *b* values for the chosen weak classifier.

New_Weight(X,Y,C,At)

Calculates the new weight for the newly chosen weak classifier.

Output: Weight for the newly chosen weak classifier.

Update_Classifier(X,Y,C,At)

Updates the main classifier with the newly chosen weak classifier.

Output: C, a list of list of *W*, *b*, and α values for the updated classifier.

myXGBoost(x1, x2, y)

Performs the Extreme Gradient Boosting algorithm. Plots results and prints accuracy.

Output: None

Loss(y,yp)

Computes the loss function.

Output: The loss, L = sum(1–y*yp)

ChooseQuad(x1,x2,x1t,x2t,y)

Choose point quadrant.

Output: Classifier parameters.

DecisionTree(x1,x2,y,n_ensmb)

Generates a decision tree using *ChooseQuad*.

Output: Classifier parameters

predict(x1,x2,C)

Given x1, x2, and classifier C, this function predicts the labels.

Output: Yp, an array of predicted labels.

LineSearch(x1,x2,y,yp,rp)

Performs a line search on gamma.

Output: Updated gamma.

HOMEWORK 7: *k*-Fold Cross Validation

I used the *KFold* function in *sklearn.model_selection* to batch cancer mortality data into training and testing sets for 5-fold cross validation. I then used the inbuilt *xgb.XGBClassifier* to train an Extreme Gradient Boosting algorithm on the training data. Using a max depth of 3, the model achieved a mean accuracy of 96.7%, with a standard deviation of 2.2%. I then performed a grid search on the max depth and minimum child weight hyperparameters. That is, I trained the model for all combinations of max_depth $\in$ {3,5,7} and min_child_weight $\in$ {0.1, 1, 5}. Of the nine combinations, I the grid search returned an optimal maximum depth of 3 and minimum child weight of 1. Using these optimal parameters, I used the inbuilt *clf.fit.importances* function to compute the F score of the data features.

Functions:

XGB(X,y,max_depth,min_child_weight)

Performs 5-fold validation for cancer mortality data. Print the mean and standard deviation of the 5-fold validation accuracy

Ouput: None

GridXGB(X,y,max_depth,min_child_weight)

Performs grid search for parameters max_depth and min_child_weight.

Prints the grid search mean test score for each parameter combination.

Output: None

XGB_importances(X,y,max_depth,min_child_weight)

Plots the feature importance of the best model.

Output: None

HOMEWORK 8: Support Vector Machine

A set of points $\{v_1, ..., v_N\}$ with binary labels $s_i = \pm 1$, is said to be linearly separable if it is possible to find a hyperplane that strictly separates the two classes. In this case, a linear programming approach can be taken to find a separating hyperplane. To allow for nonlinearly separable data, we introduce the hinge loss, and, since the margin size is proportional to $1/\|w\|^2$, introduce a term proportional to $\|w\|^2$ to maximize the margin size. The general support-vector machine thus minimizes the cost function:

$$\frac{1}{n}\sum_{i=1}^{n} \max\{0,1 - y_i(wx_i - b)\} + \lambda\|w\|^2$$

This cost function is convex (and in fact, quadratic), and can be efficiently solved with techniques from quadratic programming. However, more recent approaches instead rely on sub-gradient descent or coordinate descent. Sub-gradient methods are effective in case of nondifferentiable functions (including the max() function, which is not differentiable at zero). I implemented a sub-gradient method in which we take a step in the direction of the partial derivative of a single component of the cost function at a time. In coordinate descent, in contrast, we perform a line search on a single component, continuing until we have reached an optimum for that component, and then continue to the next component, iterating until the cost levels. For convex cost functions, both methods are guaranteed to converge to the global optimum, given a sufficiently small step size.

Introducing the variable $\zeta_i = \max\{0,1 - y_i(wx_i - b)\}$ (the ith component of the hinge loss), the minimization problem can be written in primal form:

$$\frac{1}{n}\sum_{i=1}^{n} \zeta_i + \lambda\|w\|^2$$

$$s.t.\ y_i(w \cdot x_i - b) \geq 1 - \zeta_i,$$

$$\zeta_i \geq 0, \qquad \forall i \in [1,n]$$

The dual problem is:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j K(x_i, x_j)\alpha_i\alpha_j$$

$$s.t. \sum_{i=1}^{n} y_i\alpha_i = 0\,, \quad 0 \leq \alpha_i \leq C, \quad \forall i \in [1,n]$$

Here, K is a kernel function, which in the linear case is simply $K(x_i, x_j) = x_i \cdot x_j$. The above problems can be generalized to nonlinear classification by introducing a nonlinear kernel, e.g. a polynomial kernel $K(x_i, x_j) = (x_i \cdot x_j + \gamma)^d$, a Gaussian radial basis function kernel $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2, \gamma > 0$, or a hyperbolic tangent kernel $K(x_i, x_j) = \tanh(\kappa x_i \cdot x_j + c), \kappa > 0, c < 0$.

I implemented two methods for solving the dual form of the optimization problem. Firstly, I augmented the cost function with the constraints using Lagrange multipliers, resulting in a new cost function, to which one can apply a sub-gradient descent / coordinate descent algorithm.

$$\min_{\alpha} \left(-\sum_{i=1}^{n} \alpha_i + \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j K(x_i, x_j) \alpha_i \alpha_j + \lambda_1 \sum_{i=1}^{n} y_i \alpha_i - \lambda_2 \sum_{i=1}^{n} y_i \alpha_i + \nu_1 (\alpha - C) - \nu_2 \alpha \right)$$

An alternative approach is to use the Sequential Minimal Optimization (SMO) algorithm on the dual problem. The SMO algorithm selects the values for two components of α and optimizes the objective value jointly for both values. It then computes b using the calculated values of α, iterating until α converges. I implemented the following version of SMO: iterating over all α_i, if α_i does not satisfy the KKT conditions, select an α_j at random and jointly optimize the constrained maximization problem on α_i and α_j, using the algorithm in: http://cs229.stanford.edu/materials/smo.pdf.

Functions:

Kernel(u,v,gamma=0.1,coef0=0,type="linear")
: Computes the kernel, given input vectors and parameters.
: Output: Kernel

dK(w,xi,gamma,coef0,type="linear")
: Computes the derivative of the kernel, given input vectors and parameters.
: Ouput: Kernel derivative

HingeLoss(y,w,x,lambda)
: Computes hinge loss, given input *y*, *w*, *x*, and λ.
: Ouput: Hinge loss

SubGrad (yi,w,xi,lambda)
: Computes the sub-gradient: $dL = -y_i \, dK(w, x_i) + \lambda w$
: Ouput: Subgradient

SGD(y,w,x,lambda,scale,iter)
: Performs sub-gradient descent on random, individual components using *SubGrad*.
: Output: Updated *w*

CoordDesc(y,x,lambda,scale,iter)
: Performs coordinate descent on the dual problem.
: Output: Updated *w*

SMO(y,x,lambda,scale,iter)
: Performs sequential minimal optimization on the dual problem.
: Output: Updated *w*