```
"""
Name: Regression.py
Course: ADSP31014: Statistical Models for Data Science
Author: Ming-Long Lam, Ph.D.
Organization: University of Chicago
Last Modified: October 2, 2024
(C) All Rights Reserved.
"""

import numpy
import pandas

from scipy.special import digamma, gammaln
from scipy.stats import norm, t

def SWEEPOperator (pDim, inputM, origDiag, sweepCol = None, tol = 1e-7):
    ''' Implement the SWEEP operator

    Parameter
    ---------
    pDim: dimension of matrix inputM, integer greater than one
    inputM: a square and symmetric matrix, numpy array
    origDiag: the original diagonal elements before any SWEEPing
    sweepCol: a list of columns numbers to SWEEP
    tol: singularity tolerance, positive real

    Return
    ------
    A: negative of a generalized inverse of input matrix
    aliasParam: a list of aliased rows/columns in input matrix
    nonAliasParam: a list of non-aliased rows/columns in input matrix
    '''

    if (sweepCol is None):
        sweepCol = range(pDim)

    aliasParam = []
    nonAliasParam = []

    A = numpy.copy(inputM)
    ANext = numpy.zeros((pDim,pDim))

    for k in sweepCol:
        Akk = A[k,k]
        pivot = tol * abs(origDiag[k])
        if (not numpy.isinf(Akk) and abs(Akk) >= pivot and pivot > 0.0):
            nonAliasParam.append(k)
            ANext = A - numpy.outer(A[:, k], A[k, :]) / Akk
            ANext[:, k] = A[:, k] / abs(Akk)
            ANext[k, :] = ANext[:, k]
            ANext[k, k] = -1.0 / Akk
        else:
            aliasParam.append(k)
        A = ANext
    return (A, aliasParam, nonAliasParam)

def PearsonCorrelation (x, y):
   '''Compute the Pearson correlation between two arrays x and y with the
   same number of values
```

```
   Argument:
   ---------
   x : a Pandas Series
   y : a Pandas Series

   Output:
   -------
   rho : Pearson correlation
   '''

   dev_x = x - numpy.mean(x)
   dev_y = y - numpy.mean(y)

   ss_xx = numpy.mean(dev_x * dev_x)
   ss_yy = numpy.mean(dev_y * dev_y)

   if (ss_xx > 0.0 and ss_yy > 0.0):
      ss_xy = numpy.mean(dev_x * dev_y)
      rho = (ss_xy / ss_xx) * (ss_xy / ss_yy)
      rho = numpy.sign(ss_xy) * numpy.sqrt(rho)
   else:
      rho = numpy.nan

   return (rho)

def RankOfValue (v):
   '''Compute the ranks of the values in an array v. For tied values, the
   average rank is computed.

   Argument:
   ---------
   v : a Pandas Series

   Output:
   -------
   rankv : Ranks of the values of v, minimum has a rank of zero
   '''

   uvalue, uinv, ucount = numpy.unique(v, return_inverse = True, return_counts =
True)
   urank = []
   ur0 = 0
   for c in ucount:
      ur1 = ur0 + c - 1
      urank.append((ur0 + ur1)/2.0)
      ur0 = ur1 + 1

   rankv = []
   for j in uinv:
      rankv.append(urank[j])

   return (rankv)

def SpearmanCorrelation (x, y):
   '''Compute the Spearman rank-order correlation between two arrays x and y
   with the same number of values

   Argument:
```

```
    ---------
    x : a Pandas Series
    y : a Pandas Series

    Output:
    -------
    srho : Spearman rank-order correlation
    '''

    rank_x = RankOfValue(x)
    rank_y = RankOfValue(y)

    srho = PearsonCorrelation(rank_x, rank_y)
    return (srho)

def KendallTaub (x, y):
    '''Compute the Kendall's Tau-b correlation between two arrays x and y
    with the same number of values

    Argument:
    ---------
    x : a Pandas Series
    y : a Pandas Series

    Output:
    -------
    taub : Kendall's tau-b correlation
    '''

    nconcord = 0
    ndiscord = 0
    tie_x = 0
    tie_y = 0
    tie_xy = 0

    x_past = []
    y_past = []
    for xi, yi in zip(x, y):
        for xj, yj in zip(x_past, y_past):
            if (xi > xj):
                if (yi > yj):
                    nconcord = nconcord + 1
                elif (yi < yj):
                    ndiscord = ndiscord + 1
                else:
                    tie_y = tie_y + 1
            elif (xi < xj):
                if (yi < yj):
                    nconcord = nconcord + 1
                elif (yi > yj):
                    ndiscord = ndiscord + 1
                else:
                    tie_y = tie_y + 1
            else:
                if (yi == yj):
                    tie_xy = tie_xy + 1
                else:
                    tie_x = tie_x + 1
```

```
        x_past.append(xi)
        y_past.append(yi)

    denom = (nconcord + ndiscord + tie_x) * (nconcord + ndiscord + tie_y)
    if (denom > 0.0):
        taub = (nconcord - ndiscord) / numpy.sqrt(denom)
    else:
        taub = numpy.nan

    return (taub)

def AdjustedDistance (x):
    '''Compute the adjusted distances for an array x

    Argument:
    ---------
    x : a Pandas Series

    Output:
    -------
    adj_distance : Adjusted distances
    '''

    a_matrix = []
    row_mean = []

    for xi in x:
        a_row = numpy.abs(x - xi)
        row_mean.append(numpy.mean(a_row))
        a_matrix.append(a_row)
    total_mean = numpy.mean(row_mean)

    adj_m = []
    for row, rm in zip(a_matrix, row_mean):
        row = (row - row_mean) - (rm - total_mean)
        adj_m.append(row)

    return (numpy.array(adj_m))

def DistanceCorrelation (x, y):
    '''Compute the Distance correlation between two arrays x and y
    with the same number of values

    Argument:
    ---------
    x : a Pandas Series
    y : a Pandas Series

    Output:
    -------
    dcorr : Distance correlation
    '''

    adjD_x = AdjustedDistance (x)
    adjD_y = AdjustedDistance (y)

    v2sq_x = numpy.mean(numpy.square(adjD_x))
    v2sq_y = numpy.mean(numpy.square(adjD_y))
    v2sq_xy = numpy.mean(adjD_x * adjD_y)
```

```
    if (v2sq_x > 0.0 and v2sq_y > 0.0):
       dcorr = (v2sq_xy / v2sq_x) * (v2sq_xy / v2sq_y)
       dcorr = numpy.power(dcorr, 0.25)
    else:
       dcorr = numpy.nan

    return (dcorr)

def CramerV (xCat, yCat):
    ''' Calculate Cramer V statistic

    Argument:
    ---------
    xCat : a Pandas Series
    yCat : a Pandas Series

    Output:
    -------
    cramerV : Cramer V statistic
    '''

    obsCount = pandas.crosstab(index = xCat, columns = yCat, margins = False, dropna
= True)
    xNCat = obsCount.shape[0]
    yNCat = obsCount.shape[1]

    if (xNCat > 1 and yNCat > 1):
       cTotal = obsCount.sum(axis = 1)
       rTotal = obsCount.sum(axis = 0)
       nTotal = numpy.sum(rTotal)
       expCount = numpy.outer(cTotal, (rTotal / nTotal))

       # Calculate the Chi-Square statistics
       chiSqStat = ((obsCount - expCount)**2 / expCount).to_numpy().sum()
       cramerV = chiSqStat / nTotal / (min(xNCat, yNCat) - 1.0)
       cramerV = numpy.sqrt(cramerV)
    else:
       cramerV = numpy.NaN

    return (cramerV)

def create_interaction (df1, df2):
     ''' Return the columnwise product of two dataframes (must have same number of
rows)

     Parameter
     ---------
     df1: first input data frame
     df2: second input data frame

     Return
     ------
     outDF: the columnwise product of two dataframes
     '''

     name1 = df1.columns
     name2 = df2.columns
     outDF = pandas.DataFrame()
```

```
    for col1 in name1:
        outName = col1 + ' * ' + name2
        outDF[outName] = df2.multiply(df1[col1], axis = 'index')
    return(outDF)

def paste_interaction (interactName):
    ipos = interactName.find('*')
    name1 = interactName[:ipos].strip()
    name2 = interactName[(ipos+1):].strip()
    return (name1, name2)

def LinearRegression (X, y, tolSweep = 1e-7):
    ''' Train a linear regression model

    Argument
    --------
    X: A Pandas DataFrame, rows are observations, columns are regressors
    y: A Pandas Series, rows are observations of the response variable
    tolSweep: Tolerance for SWEEP Operator

    Return
    ------
    A list of model output:
    (0) parameter_table: a Pandas DataFrame of regression coefficients and
statistics
    (1) cov_matrix: a Pandas DataFrame of covariance matrix for regression
coefficient
    (2) residual_variance: residual variance
    (3) residual_df: residual degree of freedom
    (4) aliasParam: a list of aliased rows/columns in input matrix
    (5) nonAliasParam: a list of non-aliased rows/columns in input matrix
    '''

    # X: A Pandas DataFrame, rows are observations, columns are regressors
    # y: A Pandas Series, rows are observations of the response variable

    Z = X.join(y)
    n_sample = X.shape[0]
    n_param = X.shape[1]

    ZtZ = Z.transpose().dot(Z)
    diag_ZtZ = numpy.diagonal(ZtZ)
    eps_double = numpy.finfo(numpy.float64).eps
    tol = numpy.sqrt(eps_double)

    ZtZ_transf, aliasParam, nonAliasParam = SWEEPOperator ((n_param+1), ZtZ,
diag_ZtZ, sweepCol = range(n_param), tol = tol)

    residual_df = n_sample - len(nonAliasParam)
    residual_variance = ZtZ_transf[n_param, n_param] / residual_df

    b = ZtZ_transf[0:n_param, n_param]
    b[aliasParam] = 0.0

    parameter_name = X.columns

    XtX_Ginv = - residual_variance * ZtZ_transf[0:n_param, 0:n_param]
    XtX_Ginv[:, aliasParam] = 0.0
    XtX_Ginv[aliasParam, :] = 0.0
```

```
    cov_matrix = pandas.DataFrame(XtX_Ginv, index = parameter_name, columns =
parameter_name)

    parameter_table = pandas.DataFrame(index = parameter_name,
                                       columns = ['Estimate','Standard Error', 't',
'Significance', 'Lower 95 CI', 'Upper 95 CI'])
    parameter_table['Estimate'] = b
    var_b = numpy.diag(cov_matrix)
    parameter_table['Standard Error'] = numpy.sqrt(var_b, where = (var_b > 0.0))
    parameter_table['t'] = numpy.divide(parameter_table['Estimate'],
parameter_table['Standard Error'])
    parameter_table['Significance'] = 2.0 * t.sf(abs(parameter_table['t']),
residual_df)

    t_critical = t.ppf(0.975, residual_df)
    parameter_table['Lower 95 CI'] =  parameter_table['Estimate'] - t_critical *
parameter_table['Standard Error']
    parameter_table['Upper 95 CI'] =  parameter_table['Estimate'] + t_critical *
parameter_table['Standard Error']

    return ([parameter_table, cov_matrix, residual_variance, residual_df,
aliasParam, nonAliasParam])

def PoissonRegression (X, y, offset = None, maxIter = 20, maxStep = 5, tolLLK = 1e-
3, tolBeta = 1e-10, tolSweep = 1e-7):
    ''' Train a Generalized Linear Model with Poisson distribution and Logarithm
link function

    Parameter
    ---------
    X: A Pandas DataFrame, rows are observations, columns are regressors
    y: A Pandas Series, rows are observations of the response variable
    offset: A Pandas Series of offset values
    maxIter: Maximum number of iterations
    maxStep: Maximum number of step-halving
    tolLLK: Minimum absolute difference to get a successful step-halving
    tolBeta: Maximum absolute difference between successive sets of parameter
estimates to call convergence
    tolSweep: Tolerance for SWEEP Operator

    Return
    ------
    outCoefficient: a 2D array of regression coefficients, standard errors, and
confidence interval
    outCovb: a 2D array of covariance matrix of regression coefficients
    outCorb: a 2D array of correlation matrix of regression coefficients
    llk: log-likelihood value
    nonAliasParam: a list of non-aliased rows/columns in input matrix
    outIterationTable: a 2D array of iteration history table
    y_pred: a 1D array of predicted target values
    '''

    modelX = X.copy()
    n_sample = modelX.shape[0]
    n_param = modelX.shape[1]
    param_name = modelX.columns

    modelXT = modelX.transpose()
```

```
    # Precompute the ln(y!)
    constLLK = gammaln(y+1.0)

    # Initialize arrays
    beta = numpy.zeros(n_param)
    y_mean = numpy.mean(y)
    beta[0] = numpy.log(y_mean)
    if (offset is not None):
       beta[0] = beta[0] - numpy.mean(offset)

    nu = modelX.dot(beta)
    if (offset is not None):
        nu = offset + nu

    y_pred = numpy.exp(nu)
    llk = numpy.sum(y * nu - y_pred - constLLK)

    # Prepare the iteration history table (Iteration #, Log-Likelihood, N Step-
Halving, Beta)
    itList = [0, llk, 0]
    itList.extend(beta)
    iterTable = [itList]

    for it in range(maxIter):
        gradient = modelXT.dot((y - y_pred))
        hessian = - modelXT.dot((y_pred.values.reshape((n_sample,1)) * modelX))
        orig_diag = numpy.diag(hessian)
        invhessian, aliasParam, nonAliasParam = SWEEPOperator (n_param, hessian,
orig_diag, sweepCol = range(n_param), tol = tolSweep)
        invhessian[:, aliasParam] = 0.0
        invhessian[aliasParam, :] = 0.0
        delta = numpy.matmul(-invhessian, gradient)
        step = 1.0
        for iStep in range(maxStep):
            beta_next = beta - step * delta
            nu_next = modelX.dot(beta_next)
            if (offset is not None):
                nu_next = offset + nu_next
            y_pred_next = numpy.exp(nu_next)
            llk_next = numpy.sum(y * nu_next - y_pred_next - constLLK)
            if ((llk_next - llk) > - tolLLK):
                break
            else:
                step = 0.5 * step
        diffBeta = beta_next - beta
        llk = llk_next
        beta = beta_next
        y_pred = y_pred_next
        itList = [it+1, llk, iStep]
        itList.extend(beta)
        iterTable.append(itList)
        if (numpy.linalg.norm(diffBeta) < tolBeta):
            break

    it_name = ['Iteration', 'Log-Likelihood', 'N Step-Halving']
    it_name.extend(param_name)
    outIterationTable = pandas.DataFrame(iterTable, columns = it_name)

    # Final covariance matrix
```

```
    stderr = numpy.sqrt(numpy.diag(invhessian))
    z95 = norm.ppf(0.975)

    # Final parameter estimates
    outCoefficient = pandas.DataFrame(beta, index = param_name, columns =
['Estimate'])
    outCoefficient['Standard Error'] = stderr
    outCoefficient['Lower 95% CI'] = beta - z95 * stderr
    outCoefficient['Upper 95% CI'] = beta + z95 * stderr
    outCoefficient['Exponentiated'] = numpy.exp(beta)

    outCovb = pandas.DataFrame(invhessian, index = param_name, columns =
param_name)

    temp_m1_ = numpy.outer(stderr, stderr)
    outCorb = pandas.DataFrame(numpy.divide(invhessian, temp_m1_, out =
numpy.zeros_like(invhessian), where = (temp_m1_ != 0.0)),
                               index = param_name, columns = param_name)

    return ([outCoefficient, outCovb, outCorb, llk, nonAliasParam,
outIterationTable, y_pred])

def NegativeBinomialRegression (X, y, offset = None, nSuccess = None, maxIter = 20,
maxStep = 5, tolLLK = 1e-3, tolBeta = 1e-10, tolSweep = 1e-7):
    ''' Train a Generalized Linear Model with Negative Binomial distribution and
Logarithm link function

    Parameter
    ---------
    X: A Pandas DataFrame, rows are observations, columns are regressors
    y: A Pandas Series, rows are observations of the response variable
    offset: A Pandas Series of offset values
    nSuccess: The number of successes (a positive value)
    maxIter: Maximum number of iterations
    maxStep: Maximum number of step-halving
    tolLLK: Minimum absolute difference to get a successful step-halving
    tolBeta: Maximum absolute difference between successive sets of parameter
estimates to call convergence
    tolSweep: Tolerance for SWEEP Operator

    Return
    ------
    outCoefficient: a 2D array of regression coefficients, standard errors, and
confidence interval
    outCovb: a 2D array of covariance matrix of regression coefficients
    outCorb: a 2D array of correlation matrix of regression coefficients
    llk: log-likelihood value
    nonAliasParam: a list of non-aliased rows/columns in input matrix
    outIterationTable: a 2D array of iteration history table
    y_pred: a 1D array of predicted target values
    nSuccess: The estimated number of successes by method of moments if input
nSuccess is None.
              Otherwise, the input value is echoed back.
    '''

    modelX = X.copy()
    n_sample = modelX.shape[0]
    n_param = modelX.shape[1]
    param_name = modelX.columns
```

```
    modelXT = modelX.transpose()

    # Estimate number of success if nSuccess is None
    y_mean = numpy.mean(y)
    if (nSuccess is None):
        nSuccess = y_mean * y_mean / (numpy.var(y, ddof = 1) -  y_mean)

    # Precompute the ln((k+y-1)!) - ln((k-1)!) - ln(y!)
    constLLK = gammaln(nSuccess+y) - gammaln(nSuccess) - gammaln(y+1.0)

    # Initialize arrays
    beta = numpy.zeros(n_param)
    beta[0] = numpy.log(y_mean)
    if (offset is not None):
        beta[0] = beta[0] - numpy.mean(offset)

    nu = modelX.dot(beta)
    if (offset is not None):
        nu = offset + nu

    y_pred = numpy.exp(nu)
    n_trial = nSuccess + y_pred
    prob_success = nSuccess / n_trial
    prob_failure = y_pred / n_trial
    llk = numpy.sum(y * numpy.log(prob_failure) + nSuccess *
numpy.log(prob_success) + constLLK)

    # Prepare the iteration history table (Iteration #, Log-Likelihood, N Step-
Halving, Beta)
    itList = [0, llk, 0]
    itList.extend(beta)
    iterTable = [itList]

    for it in range(maxIter):
        gradient = modelXT.dot((prob_success * (y - y_pred)))
        v_element = ((y + nSuccess) / (y_pred + nSuccess)) * prob_failure
        hessian = - nSuccess * modelXT.dot((v_element.values.reshape((n_sample,1))
* modelX))
        orig_diag = numpy.diag(hessian)
        invhessian, aliasParam, nonAliasParam = SWEEPOperator (n_param, hessian,
orig_diag, sweepCol = range(n_param), tol = tolSweep)
        invhessian[:, aliasParam] = 0.0
        invhessian[aliasParam, :] = 0.0
        delta = numpy.matmul(-invhessian, gradient)
        step = 1.0
        for iStep in range(maxStep):
            beta_next = beta - step * delta
            nu_next = modelX.dot(beta_next)
            if (offset is not None):
                nu_next = offset + nu_next
            y_pred_next = numpy.exp(nu_next)
            n_trial = nSuccess + y_pred_next
            prob_success_next = nSuccess / n_trial
            prob_failure_next = y_pred_next / n_trial
            llk_next = numpy.sum(y * numpy.log(prob_failure_next) + nSuccess *
numpy.log(prob_success_next) + constLLK)
            if ((llk_next - llk) > - tolLLK):
                break
```

```
            else:
                step = 0.5 * step
        diffBeta = beta_next - beta
        llk = llk_next
        beta = beta_next
        y_pred = y_pred_next
        prob_success = prob_success_next
        prob_failure = prob_failure_next
        itList = [it+1, llk, iStep]
        itList.extend(beta)
        iterTable.append(itList)
        if (numpy.linalg.norm(diffBeta) < tolBeta):
            break

    it_name = ['Iteration', 'Log-Likelihood', 'N Step-Halving']
    it_name.extend(param_name)
    outIterationTable = pandas.DataFrame(iterTable, columns = it_name)

    # Final covariance matrix
    stderr = numpy.sqrt(numpy.diag(invhessian))
    z95 = norm.ppf(0.975)

    # Final parameter estimates
    outCoefficient = pandas.DataFrame(beta, index = param_name, columns =
['Estimate'])
    outCoefficient['Standard Error'] = stderr
    outCoefficient['Lower 95% CI'] = beta - z95 * stderr
    outCoefficient['Upper 95% CI'] = beta + z95 * stderr
    outCoefficient['Exponentiated'] = numpy.exp(beta)

    outCovb = pandas.DataFrame(invhessian, index = param_name, columns =
param_name)

    temp_m1_ = numpy.outer(stderr, stderr)
    outCorb = pandas.DataFrame(numpy.divide(invhessian, temp_m1_, out =
numpy.zeros_like(invhessian), where = (temp_m1_ != 0.0)),
                               index = param_name, columns = param_name)

    return ([outCoefficient, outCovb, outCorb, llk, nonAliasParam,
outIterationTable, y_pred, nSuccess])

def solve4Alpha (c, maxIter = 100, epsilon = 1e-10):
    ''' Use bisection search to solve this equation for alpha:
        log(alpha) - digamma(alpha) = c

    Parameter
    ---------
    c: A positive value

    Return
    ------
    alpha: Solution of the equation, a positive value
    '''

    # Find a0 such that f0 is greater than or equal to c
    a0 = 0.5
    while True:
        f0 = numpy.log(a0) - digamma(a0)
        if (f0 < c):
```

```
            a0 = a0 / 2.0
        else:
            break

    # Find a1 such that f1 is less than or equal to c
    a1 = 2.0
    while True:
        f1 = numpy.log(a1) - digamma(a1)
        if (f1 > c):
            a1 = a1 * 2.0
        else:
            break

    # Update the end-points
    for iIter in range(maxIter):
        alpha = (a0 + a1) / 2.0
        func = numpy.log(alpha) - digamma(alpha)
        if (abs(func-c) > epsilon):
            if (func > c):
                a0 = alpha
            else:
                a1 = alpha
        else:
            break

    return (alpha)

def GammaRegression (X, y, offset = None, maxIter = 20, maxStep = 5, tolLLK = 1e-3,
tolBeta = 1e-10, tolSweep = 1e-7):
    ''' Train a Generalized Linear Model with Gamma distribution and Logarithm link
function

    Parameter
    ---------
    X: A Pandas DataFrame, rows are observations, columns are regressors
    y: A Pandas Series, rows are observations of the response variable
    offset: A Pandas Series of offset values
    maxIter: Maximum number of iterations
    maxStep: Maximum number of step-halving
    tolLLK: Minimum absolute difference to get a successful step-halving
    tolBeta: Maximum absolute difference between successive sets of parameter
estimates to call convergence
    tolSweep: Tolerance for SWEEP Operator

    Return
    ------
    outCoefficient: a 2D array of regression coefficients, standard errors, and
confidence interval
    outCovb: a 2D array of covariance matrix of regression coefficients
    outCorb: a 2D array of correlation matrix of regression coefficients
    llk: log-likelihood value
    nonAliasParam: a list of non-aliased rows/columns in input matrix
    outIterationTable: a 2D array of iteration history table
    y_pred: a 1D array of predicted target values
    alpha: the shape parameter
    '''

    modelX = X.copy()
    n_sample = modelX.shape[0]
```

```
    n_param = modelX.shape[1]
    param_name = modelX.columns

    modelXT = modelX.transpose()

    # Precompute the ln(y)
    y_log = numpy.log(y)

    # Initialize beta array and scale parameter alpha
    beta = numpy.zeros(n_param)
    beta[0] = numpy.log(numpy.mean(y))
    nu = modelX.dot(beta)
    if (offset is not None):
        nu = offset + nu
    y_pred = numpy.exp(nu)
    rvec = numpy.divide(y, y_pred)
    c = numpy.mean(rvec - numpy.log(rvec)) - 1.0
    alpha = solve4Alpha(c)
    uvec = - alpha * (rvec + numpy.log(y_pred)) + (alpha - 1.0) * y_log
    llk = numpy.sum(uvec) + n_sample * (alpha * numpy.log(alpha) - gammaln(alpha))

    # Prepare the iteration history table (Iteration #, Log-Likelihood, N
Iteration, N Step-Halving, Convergence, alpha, Beta)
    itList = [0, llk, 0, 0, numpy.nan, alpha, beta]
    iterTable = [itList]

    for it in range(maxIter):
        rvec = numpy.divide(y, y_pred)
        gradient = modelXT.dot((rvec - 1.0))
        hessian = - modelXT.dot((rvec.values.reshape((n_sample,1)) * modelX))
        orig_diag = numpy.diag(hessian)
        invhessian, aliasParam, nonAliasParam = SWEEPOperator (n_param, hessian,
orig_diag, sweepCol = range(n_param), tol = tolSweep)
        invhessian[:, aliasParam] = 0.0
        invhessian[aliasParam, :] = 0.0
        delta = numpy.matmul(-invhessian, gradient)
        step = 1.0
        for iStep in range(maxStep):
            beta_next = beta - step * delta
            nu_next = modelX.dot(beta_next)
            y_pred_next = numpy.exp(nu_next)
            rvec = numpy.divide(y, y_pred_next)
            c = numpy.mean(rvec - numpy.log(rvec)) - 1.0
            alpha_next = solve4Alpha(c)
            uvec = - alpha_next * (rvec + numpy.log(y_pred_next)) + (alpha_next -
1.0) * y_log
            llk_next = numpy.sum(uvec) + n_sample * (alpha_next *
numpy.log(alpha_next) - gammaln(alpha_next))
            if ((llk_next - llk) > - tolLLK):
                break
            else:
                step = 0.5 * step
        diffBeta = beta_next - beta
        diffBetaNorm = numpy.linalg.norm(diffBeta)
        llk = llk_next
        alpha = alpha_next
        beta = beta_next
        y_pred = y_pred_next
        itList = [it+1, llk, it, iStep, diffBetaNorm, alpha, beta]
```

```
        iterTable.append(itList)
        if (diffBetaNorm < tolBeta):
            break

    it_name = ['Iteration', 'Log-Likelihood', 'N Iteration', 'N Step-Halving',
'Criterion', 'Alpha', 'Parameters']
    outIterationTable = pandas.DataFrame(iterTable, columns = it_name)

    # Final covariance matrix
    stderr = numpy.sqrt(numpy.diag(invhessian))
    z95 = norm.ppf(0.975)

    # Final parameter estimates
    outCoefficient = pandas.DataFrame(beta, index = param_name, columns =
['Estimate'])
    outCoefficient['Standard Error'] = stderr
    outCoefficient['Lower 95% CI'] = beta - z95 * stderr
    outCoefficient['Upper 95% CI'] = beta + z95 * stderr
    outCoefficient['Exponentiated'] = numpy.exp(beta)

    outCovb = pandas.DataFrame(invhessian, index = param_name, columns =
param_name)

    temp_m1_ = numpy.outer(stderr, stderr)
    outCorb = pandas.DataFrame(numpy.divide(invhessian, temp_m1_, out =
numpy.zeros_like(invhessian), where = (temp_m1_ != 0.0)),
                               index = param_name, columns = param_name)

    return ([outCoefficient, outCovb, outCorb, llk, nonAliasParam,
outIterationTable, y_pred, alpha])

def TweedieRegression (X, y, offset = None, tweedieP = 1.5, maxIter = 50, maxStep =
5, tolLLK = 1e-3, tolBeta = 1e-10, tolSweep = 1e-7):
    ''' Train a Generalized Linear Model with Tweedie distribution and Logarithm
link function

    Parameter
    ---------
    X: A Pandas DataFrame, rows are observations, columns are regressors
    y: A Pandas Series, rows are observations of the response variable
    offset: A Pandas Series of offset values
    tweedieP: The power parameter of the distribution
    maxIter: Maximum number of iterations
    maxStep: Maximum number of step-halving
    tolLLK: Minimum absolute difference to get a successful step-halving
    tolBeta: Maximum absolute difference between successive sets of parameter
estimates to call convergence
    tolSweep: Tolerance for SWEEP Operator

    Return
    ------
    outCoefficient: a 2D array of regression coefficients, standard errors, and
confidence interval
    outCovb: a 2D array of covariance matrix of regression coefficients
    outCorb: a 2D array of correlation matrix of regression coefficients
    qllk: quasi log-likelihood value
    nonAliasParam: a list of non-aliased rows/columns in input matrix
    outIterationTable: a 2D array of iteration history table
    y_pred: a 1D array of predicted target values
```

```
    phi: the phi parameter
    '''

    modelX = X.copy()
    n_sample = modelX.shape[0]
    n_param = modelX.shape[1]
    param_name = modelX.columns

    modelXT = modelX.transpose()

    two_p = 2.0 - tweedieP
    one_p = 1.0 - tweedieP
    ypow21 = numpy.power(y, two_p) / two_p / one_p

    # Initialize beta array
    beta = numpy.zeros(n_param)
    beta[0] = numpy.log(numpy.mean(y))
    nu = modelX.dot(beta)
    if (offset is not None):
        nu = nu + offset
    y_pred = numpy.exp(nu)
    powvec = numpy.power(y_pred, one_p)
    devvec = 2.0 * (ypow21 - y * powvec / one_p + y_pred * powvec / two_p)
    qllk = - numpy.sum(devvec) / 2.0

    # Prepare the iteration history table
    itList = [0, qllk, 0, 0, numpy.nan]
    itList.append(beta)
    iterTable = [itList]

    for it in range(maxIter):
        rvec = (y_pred - y) * powvec
        gradient = 2.0 * modelXT.dot(rvec)
        svec = (two_p * y_pred - one_p * y) * powvec
        hessian = 2.0 * modelXT.dot((svec.values.reshape((n_sample,1)) * modelX))
        orig_diag = numpy.diag(hessian)
        invhessian, aliasParam, nonAliasParam = SWEEPOperator (n_param, hessian,
orig_diag, sweepCol = range(n_param), tol = tolSweep)
        invhessian[:, aliasParam] = 0.0
        invhessian[aliasParam, :] = 0.0
        delta = numpy.matmul(-invhessian, gradient)
        step = 1.0
        for iStep in range(maxStep):
            beta_next = beta - step * delta
            nu_next = modelX.dot(beta_next)
            if (offset is not None):
                nu_next = nu_next + offset
            y_pred_next = numpy.exp(nu_next)
            powvec_next = numpy.power(y_pred_next, one_p)
            devvec = 2.0 * (ypow21 - y * powvec_next / one_p + y_pred_next *
powvec_next / two_p)
            qllk_next = - numpy.sum(devvec) / 2.0
            if ((qllk_next - qllk) > - tolLLK):
                break
            else:
                step = 0.5 * step
        diffBeta = beta_next - beta
        diffBetaNorm = numpy.linalg.norm(diffBeta)
        qllk = qllk_next
```

```
        beta = beta_next
        powvec = powvec_next
        y_pred = y_pred_next
        itList = [it+1, qllk, it, iStep, diffBetaNorm]
        itList.append(beta)
        iterTable.append(itList)
        if (diffBetaNorm < tolBeta):
            break

    it_name = ['Iteration', 'Quasi Log-Likelihood', 'N Iteration', 'N Step-
Halving', 'Criterion', 'Parameters']
    outIterationTable = pandas.DataFrame(iterTable, columns = it_name)

    invhessian = - invhessian
    # Final covariance matrix
    stderr = numpy.sqrt(numpy.diag(invhessian))
    z95 = norm.ppf(0.975)

    # Final parameter estimates
    outCoefficient = pandas.DataFrame(beta, index = param_name, columns =
['Estimate'])
    outCoefficient['Standard Error'] = stderr
    outCoefficient['Lower 95% CI'] = beta - z95 * stderr
    outCoefficient['Upper 95% CI'] = beta + z95 * stderr
    outCoefficient['Exponentiated'] = numpy.exp(beta)

    outCovb = pandas.DataFrame(invhessian, index = param_name, columns =
param_name)

    temp_m1_ = numpy.outer(stderr, stderr)
    outCorb = pandas.DataFrame(numpy.divide(invhessian, temp_m1_, out =
numpy.zeros_like(invhessian), where = (temp_m1_ != 0.0)),
                               index = param_name, columns = param_name)

    devvec = (ypow21 - y * powvec / one_p + y_pred * powvec / two_p)
    phi = numpy.sum(devvec) / (n_sample - len(nonAliasParam))
    outCovb = outCovb / phi

    return ([outCoefficient, outCovb, outCorb, qllk, nonAliasParam,
outIterationTable, y_pred, phi])
```