**Client experimental GoDriveCarBox**

**Arhitectura modelelor de predictie a defectarii componetelor autovehiculului**

|  |  |
| --- | --- |
| Proiect | GoDrive |
| Beneficiar | GODRIVE SRL |
| Contract | Nr. 2/25.11.2016 |
| Data modificare | 2017.03.08 |
| Data creere | 2017.02.02 |
| Versiune | 1.0.0.2 |
| Descriere | Arhitectura modele predictie defecte autovehicule |

Problema predictiei defectarii componentelor autovehicolului comporta multiple aspecte ce trebuie luate in consideratie la implementare atat din punct de vedere tehnic cat si din punct de vedere al modelarii de date/informatii. In acest sens s-au identificat urmatoarele aspecte:

**Tabelul analizei solutiilor arhitecturale**

| **Nr** | **Problema** | **Natura** | **Descriere** | **Solutie** |
| --- | --- | --- | --- | --- |
| 1 | Modelarea predictie | Modelare | In vederea determinarii unei predictii a unei potentiale defectari de componenta este necasara analiza factorilor si a modului de inferenta a corelatiei optime. Chiar daca statistic se poate determina o inferenta este necesara gasirea unui algoritm auto-adaptabil care sa “invete” sa gaseasca corelatii si sa determine predictii | Se vor folosi modele de predictie avansate bazate pe tehnologiile actuale ale invatarii automatizate (stadiul curent al tehnologiei, cercetarii si dezvoltarii). S-a optat pentru utilizarea unui model de invatarea automatizata bazat pe retele neurale cu multi-nivele ascunse. In “***Tabelul de descriere a solutiei de modelare***” sunt prezentate detaliile propuse ale arhitecturii matematice solutiei. |
| 2 | Metode de determinare a predictorilor | Modelare | Indentificarea fluxurilor de date ce pot genera potentiale inferente privind functionarea autovehicolului: avand un model de baza matematic este necesara determinarea variabilelor principale si a ponderilor acestora in determinarea functiei ipoteza de invatare automata si predictie | S-au analizat numeroase lucrari stiintifice publicate in ultimii 5 ani in domeniul tehnologiilor bazate pe Inteligenta Artificiala. Din aceste lucrari amintim doua dintre ele axate in special pe utilizarea sistemelor cu invatare automatizata in modele de predictibilitate a intretinerii flotelor de autovehicule si auto-utilitare:   1. R. Prytz, S. Nowaczyk, T. Rögnvaldsson, S. Byttner, "Analysis of Truck Compressor Failures Based on Logged Vehicle Data", in In Proceedings of the 9th International Conference on Data Mining (DMIN’13), Las Vegas, NV, USA. July 2013 2. T. Rögnvaldsson, S. Byttner, R. Prytz, S Nowaczyk, "Wisdom of Crowds for Self-organized Intelligent Monitoring of Vehicle Fleets", submitted to IEEE Transactions on Knowledge and Data Engineering (TKDE), 2014 |
| 3 | Rularea modelului atat in mediu computational intensiv cat si in mediu local | Tehnica | Modelul predictiv ales bazat pe tehnologiile state-of-the-art de deep leaning are inconvenientul necesitatii unei puteri mari de calcul precum si a | Solutia aleasa consta in spargerea modelarii in doua etape:   1. Etapa de preantrenare a modelului cu invatare automatizata la nivelul serverului GoDrive. In cadrul acestei etape modelele bazate pe relete neurale adanci vor fi antrenate prin invatare automatizata si se va genera un model initial al ponderilor predictorilor din retele. Se vor utiliza date off-line. 2. Etapa de auto-invatare continua la nivelul dispozitivului incorporat GoDrive va asigura ajustarea ponderilor predictorilor si a nivelelor ascunse adanci din retelele neurale in functie de dinamica datelor colectate si analizate din mediul real. |
| 4 | Disponibilitatea datelor la faza de experimentare | Modelare / Tehnica | In vederea realizarii procesului de invatare automatizata este necesara si obligatorie existenta unor seturi pe baza carora sa se faca experimentarea modelarii intiale inferentiale | Se vor achizitiona seturi de date din surse externe:   1. UCI Machine Learning Repository collection of databases, domain theories, and data generators, Center for Machine Learning and Intelligent Systems   Bren School of Information and Computer Science, University of California, Irvine   1. Vehicle safety defect investigations, recalls and collision investigations data for Great Britain, Driver and Vehicle Standards Agency, UK |

**Tabelul de descriere a solutiei de modelare**

|  |  |
| --- | --- |
| Algoritm | Model auto-modelare de predictor si predictie a potentialelor defecte in autovehicul |
| Versiune | 1.0.2 |
| Data creatiei | 2017.02.01 |
| Data ultimei modificari | 2017.04.09 |
| Tip algoritm | Model cu invatare automatizata |
| Model de optimizare | Optimizare cu invatare automatizata bazata pe calculul gradientilor erorii functiei de cost |
| Descriere generala | Determinarea atat a factorilor care determina defectarea componentelor autovehicolului cat si a momentului in care o anumita componenta se defecteaza poate fi vazuta ca o functie cu complexitate ridicata H(X)=Y multi-variata (considerand X ca fiind un vector de date stohastice generate de catre autovehicol) si multi-nomiala de date (consideram Y ca fiind un vector multi-nomial in care fiecare element reprezinta un rezultat de predictie sau inferenta). Datorita caracterului atat euristic cat si stohastic al procesului de analiza al variabilelor si de determinare a rezultatelor este putin probabila aplicarea unui algoritm complex prin care sa se defineasca functia H(X)=Y, motiv pentru care functia ipoteza H(X) va fi determinata de un model avansat cu invatare automatizata bazat pe retele neurale cu nivele multiple ascunse precum si convolutiile necesare procesului de determinare a variabilelor calculate. |
| Date de intrare | Vectorul X de parametrii va contine toate datele obtenabile prin intermediul interfetei On Board Diagnostics |
| Date de iesire | In urma analizei literaturii si lucrarilor de specialitate precum si in urma achizitionarii seturilor de date propuse se vor identifica variabilele “tinta” urmarite care sunt in directa corelatie (aproximativ 1-la-1) cu predictia defectarii unuia sau mai multor componente |
| Metoda de calcul | Plecand de la premiza ca intreg modelul reprezinta o functie ipoteza de determinare in baza unui vector multi-variat a unui rezultat multi-nomial se construieste o retea neurala pe structura intuitiva a neuronilor biologici in care avem:   * Fiecare neuron primeste informatii de la mai multi neuroni conectati de la axoni acestora la dendritele sale * Neuronul individual realizeaza o peratie liniara dupa care trece rezultatul operatiei liniare printr-o functie non-liniara de activare (de exemplu sigmoid ) * Prin introducerea unui numar mare de neuroni pe nivele multiple se poate “inmagazina” o cantitate mare de informatie in vederea obtinerii unei functii cu complexitate foarte ridicata non-liniara. Un exemplu este dat de imaginea de mai jos:   Image result for fully connected deep network   * In vederea calcului functiei ipoteza H(X)=Y se aplica un algoritm de “mers inainte” de forma:   Unde W(i) este un vector de ponderi al dendritelor neuronului respectiv ce se aplica intrarilor din neuronii nivelului neural anterior   * In final se calculeaza eroarea intre valoarea multi-nomiala asteptata si cea rezultata din functie ipoteza cu: * Invatarea efectiva consta in ajustarea ponderilor (vectorii/matricile W(i,j) unde i,j reprezinta nivelele retelei neurala) automatizata se realizeaza prin metoda propagarii inapoi a gradientului erorii determinate si prezentate anterior. Concret pentru cazul particular al determinarii ponderilor dendritelor pentru nivelul 3 intr-o retea neurala cu 5 nivele (nivelul 5 fiind nivelul de calcul final al vectorului multi-nomial de iesire):   In cazul de mai sus vectorul/variabile ϴ este echivalenta cu W. Ajustarea efectiva a ponderilor din matricile respective se face prin aplicarea unui algoritm de tip gradient-descent - ajustarea periodica cu fragmente ale gradientului pana la atingerea unui minim al functiei de cost E. |
| Metoda de implementare | Implementarea se va realiza prin utilizarea unui limbaj de tip multi-platforma independent de dispozitivul hardware – in cazul de fata limbajul Python. |
| Mediul de rulare la nivelul dispozitivului incorporat | La nivelul dispozitivului incorporat se va utiliza un micro-interpretor Python capabil sa ruleze pe dispozitive cu capacitati extrem de reduse (Embedded Python) |
| Variante de modele propuse | Varianta A: in primul stadiu al experimentarii se propune un model cu invatare superficiala specializat pe functionarea in medii on-line  Varianta B: in stadiul avansat al proiectului se va face trecerea de la modelul superficial la un model cu parametrizare adanca de tip retea convolutionala adanca |

**Anexa 1 - Cod sursa modul predictiv bazat pe arhitectura superficiala**

|  |
| --- |
| # -\*- coding: utf-8 -\*-  """  @application: Online Classifier Engine  @created: 2017-01-04  @author: 4E SOFTWARE SRL  =======  TODO:  Momentum/Velocity for Online learning setting  NEO LineSearch (adapted for online)  VotingClassification  """  import pandas as pd  import numpy as np  from scipy.special import expit  from scipy import stats  import matplotlib.pyplot as plt # for debug error plotting  from time import time  import sys  class oce\_utils:    def FeatureNormalize(self,X\_data, method = 'z-score'):  if method == 'z-score':  min\_val = X\_data.mean(axis=0)  div\_val = X\_data.std(axis=0)  elif method =="minmax":  ## min-max  min\_val = X\_data.min(axis=0)  div\_val = X\_data.max(axis=0)  else:  raise Exception("Unknown scale/norm method: "+str(method))    div\_val[div\_val == 0] = 1.    X\_norm = X\_data - min\_val  X\_norm = np.array(X\_norm,dtype = float) / div\_val    return X\_norm, min\_val, div\_val    def TestDataNormalize(self, X\_test, min\_val,div\_val):  X\_norm = X\_test - min\_val  X\_norm = np.array(X\_norm,dtype = float) / div\_val    return X\_norm  def loaddata(self, file):  return pd.read\_csv(file)  #  # Kappa: duplicated and generalized from OnlineClassifier version  #  def Kappa(self,y\_pred,y\_truth, classes):  nr\_classes = len(classes)  classes = list(classes)  TP = np.zeros(shape=(nr\_classes))  FP = np.zeros(shape=(nr\_classes))  TN = np.zeros(shape=(nr\_classes))  FN = np.zeros(shape=(nr\_classes))  class\_pred = np.zeros(shape=(nr\_classes))  class\_real = np.zeros(shape=(nr\_classes))  for (i,c\_class) in zip(range(nr\_classes),classes):  TP[i] = np.logical\_and( y\_pred == c\_class, y\_truth == c\_class ).sum()  TN[i] = np.logical\_and( y\_pred != c\_class, y\_truth != c\_class ).sum()  FP[i] = np.logical\_and( y\_pred == c\_class, y\_truth != c\_class ).sum()  FN[i] = np.logical\_and( y\_pred != c\_class, y\_truth == c\_class ).sum()  class\_pred[i] = TP[i] + FP[i]  class\_real[i] = TP[i] + FN[i]    all\_ex = TP[0]+TN[0]+FP[0]+FN[0]  observed\_accuracy = np.sum(TP) / all\_ex  expected\_accuracy = (np.sum(class\_pred\*class\_real) / all\_ex) / all\_ex  kappa = (observed\_accuracy - expected\_accuracy) / \  (1 - expected\_accuracy)  # conf\_matrix !!!  return kappa  #  # ROC: duplicated and generalized from OnlineClassifier version  #  def ROC(self,y\_prc,y\_label, labels):  nr\_labels = len(labels)  if y\_label.ndim>1:  y\_label\_list=y\_label[:,0]  thresholds = np.linspace(1, 0, 101)  if nr\_labels == 2:  nr\_ROCs = 1  else:  nr\_ROCs = nr\_labels    TPR = np.zeros(shape=(101,nr\_ROCs))  FPR = np.zeros(shape=(101,nr\_ROCs))  AUC = np.zeros(shape=(nr\_ROCs))  for cROC in range(nr\_ROCs):  if nr\_ROCs==1:  c\_label=1  else:  c\_label = labels[cROC]  for i in range(101):  c\_thr = thresholds[i]  # Classifier / label agree and disagreements for current threshold.  if i==50:  k=1  TP = np.logical\_and( y\_prc[:,cROC] > c\_thr, y\_label\_list==c\_label ).sum()  TN = np.logical\_and( y\_prc[:,cROC] <=c\_thr, y\_label\_list!=c\_label ).sum()  FP = np.logical\_and( y\_prc[:,cROC] > c\_thr, y\_label\_list!=c\_label ).sum()  FN = np.logical\_and( y\_prc[:,cROC] <=c\_thr, y\_label\_list==c\_label ).sum()    # Compute false positive rate for current threshold.  FPR[i,cROC] = FP / float(FP + TN)    # Compute true positive rate for current threshold.  TPR[i,cROC] = TP / float(TP + FN)    # compute the AUC score for the ROC curve using the trapezoidal method  AUC[cROC] = 0.  for i in range(100):  AUC[cROC] += (FPR[i+1,cROC]-FPR[i,cROC]) \* (TPR[i+1,cROC]+TPR[i,cROC])    AUC[cROC] \*= 0.5      return TPR,FPR, AUC      ##  ## train\_online\_classifier() simulates a real life  ## feed of data to our OnlineClassifier  ## cross-validation is used to obtain best J(Theta)  ##  def train\_online\_classifier(self, clf, X\_train,y\_train, X\_cross = None,y\_cross = None,batch\_size=1):  nr\_examples = X\_train.shape[0]  nr\_batches = nr\_examples / batch\_size  for i in range(nr\_batches):  xi = X\_train[(i\*batch\_size):((i+1)\*batch\_size),:]  yi = y\_train[(i\*batch\_size):((i+1)\*batch\_size)]  clf.OnlineTrain(xi,yi,X\_cross=X\_cross,y\_cross=y\_cross)  return clf  ##  ## ck\_train\_online\_classifier() simulates a real life  ## feed of data to our OnlineClassifier  ##  ##  def ck\_train\_online\_classifier(self, clf, X\_train,y\_train, X\_cross = None,y\_cross = None,batch\_size=1):  nr\_examples = X\_train.shape[0]  nr\_batches = nr\_examples / batch\_size  for i in range(nr\_batches):  xi = X\_train[(i\*batch\_size):((i+1)\*batch\_size),:]  yi = y\_train[(i\*batch\_size):((i+1)\*batch\_size)]  clf.OnlineTrain(xi,yi,X\_cross=X\_cross,y\_cross=y\_cross)  return clf    ##  ## implement OnlineClassifier  ## both multi-class (one-vs-all) and single-class logistic regression  ## y is either multi-class or True/False  ##  ##  class OnlineClassifier:  def \_\_init\_\_(self,nr\_features,classes=[0,1],  alpha=1.0, DecreasingAlpha=False,  alpha\_coef=-1, softmax\_alpha\_search = False,  polyfeats=1,method='sigmoid',  lmbd=0,  random\_init=False,  Verbose = 5, NoVerbose = False,  back\_train = 1):    self.back\_train = back\_train  self.softmax\_alpha\_search = softmax\_alpha\_search  self.alpha\_search\_epochs = 100  self.alpha\_search\_iter = 0;  self.Classes = list(classes) # class labels binary default  self.eps = 1e-15 # constant used for clipping  self.Verbose = Verbose # this is the verbose level: the higher the most-import-only info is displayed  self.NoVerbose = NoVerbose # force to ignore Verbose property  self.\_standard\_binary\_classes = [0,1]  self.lmbd = lmbd # lambda for reguralization DEFAULT 0 (no reg)  self.methods = ['softmax','sigmoid','perceptron'] #"methodation" function  if not (method in self.methods):  raise Exception("Unknown method: "+method)  self.method = method # can be default='sigmoid' or 'softmax'  self.DecreasingAlpha = DecreasingAlpha # alpha gradient step decreases ?  self.alpha\_coef = alpha\_coef # coef for alpha decrease = actually not used  self.base\_alpha = alpha # alpha  self.nr\_Classes = len(self.Classes) # number of classes (2 default)  self.original\_n = nr\_features +1 # original number of features MUST be precoded includes intercept  self.n = nr\_features\*polyfeats + 1 # number of features +1 (poly features + 1 used only if poly = true)  self.m = 0  self.alpha = alpha  self.alpha\_0 = alpha  self.alpha\_array = np.empty((0,1),float)  self.MultiClass = False  if method == 'sigmoid':  self.Costs = np.empty((0,self.nr\_Classes), float)  self.J\_array = np.empty((0,self.nr\_Classes), float)  else:  self.Costs = np.empty((1,0), float)  self.J\_array = np.empty((1,0), float)  self.xi = np.array([])  self.y = None  self.BestAccuracy = 0  self.BestFeed = 0  self.polyfeats = polyfeats # >1 if using polynomial feats remapping  self.all\_X = np.empty((0,self.n), float) # all xi in one matrix  self.all\_y = np.empty((0), dtype = object)  nr\_thetas=1  if (self.nr\_Classes>2) or (self.method == 'softmax'):  nr\_thetas=self.nr\_Classes  self.MultiClass = True    self.BestTheta = None  self.random\_init = random\_init  if random\_init:  ###  ### random Theta initilization  ### wih "noise" values (-0.05 to +0.05)  ###  self.Theta = np.random.uniform(low=-0.05, high=0.05, size=(self.n,nr\_thetas))  else:  self.Theta = np.zeros(shape=(self.n,nr\_thetas))    ## now we need a mechanism to preserve all gradients for each class  ## we will use a 3d matrix (iteration,class,actual\_theta)  ## this way we can analyse exploding gradients  self.gradients = np.empty((0,nr\_thetas,self.n),float)  self.\_nr\_thetas = nr\_thetas    self.LastGrad = None  self.LastYOHM = None  self.LastYHat = None  self.LastYERR = None  self.LastGThe = None  self.LastThet = None  self.LastJ = None  self.LastAlph = None    self.BestAlphas = list()    def SearchBestAlpha(self,x, ohmy,Verbose = True):  bestAlpha = 0  bestDiff = -1e100  if self.LastJ == None:  return bestAlpha    alphas = np.array([1e-5,5e-5,1e-4,5e-4,1e-3,5e-3,1e-2,5e-2,0.1,0.5,1,5])  diff\_list = list()  for i in range(alphas.size):  test\_alpha = alphas[i]  ##  ## now compute test weights based on previous weights  ## updated with previous gradient and tested alpha  ## then compute current J(theta) and determine  ## best previous update step (best previous alpha)  TestTheta = self.LastGThe - test\_alpha\*self.LastGrad  m = np.float64(x.shape[0]) # batch update size not all obs !!!  Theta = np.array(TestTheta)  xT = x.dot(Theta)  yhat = self.softmax(xT)  yhat = np.clip(yhat,self.eps,1-self.eps)  # now final calc incl reguralization  J = self.\_log\_loss\_reg(ohmy,yhat, self.lmbd, Theta, m)    J\_diff = self.LastJ - J  diff\_list.append(J\_diff)    if J\_diff > bestDiff:  bestAlpha = test\_alpha  bestDiff = J\_diff    if Verbose:  self.DebugInfo("[DEBUG] BestAlpha = {:.5f}".format(bestAlpha), 10)  self.alpha\_search\_iter = self.alpha\_search\_iter + 1    return bestAlpha              def DebugInfo(self, Value, lvl=0):  if self.NoVerbose:  return  if lvl<=self.Verbose:  return  text = ""  #text = str(type(Value))  #text += ':\n'  text += str(Value)  if self.Verbose:  print text  sys.stdout.flush()    def GetShortHyperParams(self):  return "Method={} Poly={} BatchSize={} Alpha0={}".format(self.method,  self.polyfeats,  self.batchsize,  self.alpha\_0)  def GetHyperParams(self):  str\_params = "\nHyper Parameters:"  str\_params += "\nHyFunction: "+str(self.method)  str\_params += "\nAlpha-init: "+str(self.alpha\_0)  str\_params += "\nDecrAlpha : "+str(self.DecreasingAlpha)  str\_params += "\nAlphaCoef : "+str(self.alpha\_coef)  str\_params += "\nSM-alpsrch: "+str(self.softmax\_alpha\_search)  str\_params += "\nAlpha-last: "+str(self.alpha\_array[-3:])  str\_params += "\nPolynomial: "+str(self.polyfeats)  str\_params += "\nClasses : "+str(self.Classes)  str\_params += "\nRegLambda : "+str(self.lmbd)  str\_params += "\nRandTheta : "+str(self.random\_init)  str\_params += "\n"  return str\_params    def Kappa(self,y\_pred,y\_truth, classes):  nr\_classes = len(classes)  classes = list(classes)  TP = np.zeros(shape=(nr\_classes))  FP = np.zeros(shape=(nr\_classes))  TN = np.zeros(shape=(nr\_classes))  FN = np.zeros(shape=(nr\_classes))  class\_pred = np.zeros(shape=(nr\_classes))  class\_real = np.zeros(shape=(nr\_classes))  for (i,c\_class) in zip(range(nr\_classes),classes):  TP[i] = np.logical\_and( y\_pred == c\_class, y\_truth == c\_class ).sum()  TN[i] = np.logical\_and( y\_pred != c\_class, y\_truth != c\_class ).sum()  FP[i] = np.logical\_and( y\_pred == c\_class, y\_truth != c\_class ).sum()  FN[i] = np.logical\_and( y\_pred != c\_class, y\_truth == c\_class ).sum()  class\_pred[i] = TP[i] + FP[i]  class\_real[i] = TP[i] + FN[i]    all\_ex = TP[0]+TN[0]+FP[0]+FN[0]  observed\_accuracy = np.sum(TP) / all\_ex  expected\_accuracy = (np.sum(class\_pred\*class\_real) / all\_ex) / all\_ex  kappa = (observed\_accuracy - expected\_accuracy) / \  (1 - expected\_accuracy)  # conf\_matrix !!!  return kappa  def ROC(self,y\_prc,y\_label, labels):  nr\_labels = len(labels)  if y\_label.ndim>1:  y\_label\_list=y\_label[:,0]  thresholds = np.linspace(1, 0, 101)  if nr\_labels == 2:  nr\_ROCs = 1  else:  nr\_ROCs = nr\_labels    TPR = np.zeros(shape=(101,nr\_ROCs))  FPR = np.zeros(shape=(101,nr\_ROCs))  AUC = np.zeros(shape=(nr\_ROCs))  for cROC in range(nr\_ROCs):  if nr\_ROCs==1:  c\_label=1  else:  c\_label = labels[cROC]  for i in range(101):  c\_thr = thresholds[i]  # Classifier / label agree and disagreements for current threshold.  if i==50:  k=1  TP = np.logical\_and( y\_prc[:,cROC] > c\_thr, y\_label\_list==c\_label ).sum()  TN = np.logical\_and( y\_prc[:,cROC] <=c\_thr, y\_label\_list!=c\_label ).sum()  FP = np.logical\_and( y\_prc[:,cROC] > c\_thr, y\_label\_list!=c\_label ).sum()  FN = np.logical\_and( y\_prc[:,cROC] <=c\_thr, y\_label\_list==c\_label ).sum()    # Compute false positive rate for current threshold.  FPR[i,cROC] = FP / float(FP + TN)    # Compute true positive rate for current threshold.  TPR[i,cROC] = TP / float(TP + FN)    # compute the AUC score for the ROC curve using the trapezoidal method  AUC[cROC] = 0.  for i in range(100):  AUC[cROC] += (FPR[i+1,cROC]-FPR[i,cROC]) \* (TPR[i+1,cROC]+TPR[i,cROC])    AUC[cROC] \*= 0.5      return TPR,FPR, AUC  def GetConfusionMatrix(self, y\_pred,y\_label):  nr\_preds = y\_pred.size  pred\_classes = np.unique(y\_pred)  labl\_classes = np.unique(y\_label)  all\_classes = np.unique(np.r\_[pred\_classes,labl\_classes])    conf\_df = pd.DataFrame(index= all\_classes ,columns = all\_classes)  conf\_df.index.name = "Truth"  for row in range(all\_classes.size):  for col in range(all\_classes.size):  c\_preds = y\_pred==all\_classes[col]  c\_label = y\_label == all\_classes[row]  val = np.logical\_and( c\_preds , c\_label ).sum()  conf\_df.at[all\_classes[row],all\_classes[col]]= val    return conf\_df    def add\_observation(self, x,y):    self.all\_X = np.r\_[self.all\_X, x]  self.all\_y = np.append(self.all\_y,y)    return    def get\_train\_obs(self):  nr\_all\_x = self.all\_X.shape[0]  last\_obs = np.arange(nr\_all\_x-self.batchsize,nr\_all\_x)  nr\_obs = int(round(self.batchsize \* self.back\_train))    extra\_obs = nr\_obs - self.batchsize  if (nr\_all\_x >= nr\_obs) and (extra\_obs>0):  old\_idx = np.arange(0,nr\_all\_x -self.batchsize)  np.random.shuffle(old\_idx)  all\_obs = np.append(old\_idx[:extra\_obs],last\_obs)  else:  all\_obs = last\_obs    xi = self.all\_X[all\_obs,:]  yi = self.all\_y[all\_obs]  return xi,yi  def prepare\_x(self, x): # add intercept and poly feats  # convert to a 1xN matrix if single observation  x\_temp = np.array(x,ndmin=2)  x\_prepared = np.array(x\_temp)  mini\_batch\_size = x\_prepared.shape[0]  ones\_column = np.ones(shape=(mini\_batch\_size))  x\_prepared = np.c\_[ones\_column,x\_prepared] # add intercept  if self.polyfeats>1:  for rank in range (2,self.polyfeats+1):  x\_prepared = np.c\_[x\_prepared, np.power(x\_temp,rank)]    return x\_prepared    def Calc\_pValues(self):  yHat,ydf = self.Predict(self.all\_X)  y = self.all\_y  X = self.all\_X    ## THIS IS NOT YET OK !  sse = np.sum((yHat - y) \*\* 2, axis=0) / float(X.shape[0] - X.shape[1])  #se = np.array([ np.sqrt(np.diagonal(sse[i] \* np.linalg.inv(np.dot(X.T, X)))) for i in range(sse.shape[0]) ])  se = np.array([np.sqrt(np.diagonal(sse \* np.linalg.inv(np.dot(X.T, X))))])  self.t = self.Theta / se  self.p = 2 \* (1 - stats.t.cdf(np.abs(self.t), y.shape[0] - X.shape[1]))  return self      def softmax(self,z):  # z is MxK where M=observation K=classes  # first shift the values of f so that the  # highest number is 0:  z -= np.max(z)    ez = np.exp(z)    p = (ez.T / np.sum(ez, axis=1)).T  return p    def sigmoid(self,z):  return expit(z)  #return 1 / (1 + math.exp(-z))      def \_simple\_cross\_entropy\_loss(self, y, ht):  # two class cross-entropy  crs\_entr\_loss = (-1.0 )\* np.sum(y\*np.log(ht)+(1-y)\*np.log(1-ht))  return crs\_entr\_loss      def CostFunctionLogistic(self,i\_theta,x,y):  # implements sigmoid logistic cost function and grad,  # i\_theta = current theta  # works both for single and mini-batch updates !    if self.method != 'sigmoid':  raise Exception('Sigmoid cost function called from non-sigmoid classifier')    if x.ndim != 2:  raise Exception('Sigmoid function received x with ndim!=2')  m = np.float64(x.shape[0]) # batch update size !!! NOT ALL OBS    Theta = np.array(self.Theta[:,i\_theta])  Theta0 = np.array(Theta)  Theta0[0] = 0  xT = x.dot(Theta)  HT = self.sigmoid(xT)  HT = np.clip(HT,self.eps,1-self.eps)  H = HT - y    # cost for linear regression  # cCostRegression = (1/float(2))\*np.power(H,2 )    # cross-entropy cost function  cCost1 = self.\_simple\_cross\_entropy\_loss(y,HT)  cCost2 = cCost1 / (float(m))  # end cross-entropy cost function    # ADD REGURALIZATION  # default lmbd is 0 so no reg by default  cCost = cCost2 + (self.lmbd) / (2 \* m) \* np.sum(Theta0.T.dot(Theta0))  Grad = (1.0 / m) \* x.T.dot(H)  #ADD REGURALIZATION  Grad += (self.lmbd / m) \* Theta0    if np.isnan(cCost):  self.DebugInfo("[ERROR] NaN Cost",100)  prev\_grads = self.gradients[:,i\_theta,:]  sum\_vector = np.sum(prev\_grads, axis = 1)  plt.plot(range(self.m-1),sum\_vector[:-1])  plt.show()  cls = i\_theta  itr = self.m  self.DebugInfo("[ERROR] Grad norm vect= {}".format(sum\_vector),100)    str\_E = "[ERROR]NaN cost Theta={} at batch no. {} ".format(cls,itr)  str\_E += "\nMaxX={:.2f} MinX={:.2f}".format(np.max(x),np.min(x))  str\_E += "\nMaxT={:.2f} MinT={:.2f}".format(np.max(Theta),np.min(Theta))  str\_E += "\nX.dot.Theta={}".format(xT)  str\_E += self.GetHyperParams()  raise Exception(str\_E)    return Grad, cCost    def LogisticTrain(self,xi, yi):  ### 1 step stohastic logistic classifier training based on  ### multi-class logistic    # prepare gradient storage  self.gradients = np.append(self.gradients,  np.zeros(shape=(1,  self.\_nr\_thetas,  self.n)), axis = 0)  if yi.ndim>1:  yi = np.ravel(yi)  if self.MultiClass:  #find right theta for each class !!!  y\_coded = np.empty(shape=(yi.size))  cCosts = np.zeros(shape=(1,self.nr\_Classes))  for i in range(self.nr\_Classes):  y\_coded.fill(0)  if y\_coded.size == 1:  y\_coded[0] = (yi == self.Classes[i])  else:  y\_coded[yi==self.Classes[i]] = 1  Grad, cCost = self.CostFunctionLogistic(i,xi,y\_coded)  ### store gradient  self.gradients[-1,i,:] = Grad  ### done store gradient  self.Theta[:,i] = self.Theta[:,i] - self.alpha\*Grad  cCosts[0,i] = cCost    self.Costs = np.append(self.Costs,cCosts,axis=0)  J = np.empty((1,self.nr\_Classes), float)  J[0,:] = np.nanmean(self.Costs,axis=0)  self.J\_array = np.append(self.J\_array, J, axis=0)  ## done multi class  else:  ##  ## now for single class  ##  i\_c = 0  Grad, cCost = self.CostFunctionLogistic(i\_c,xi,yi)  self.Theta[:,i\_c] = self.Theta[:,i\_c] - self.alpha\*Grad  self.Costs = np.append(self.Costs,cCost)  ### store gradient  self.gradients[-1,0,:] = Grad  ### done store gradient  J = np.sum(self.Costs)/self.Costs.shape[0]  self.J\_array = np.append(self.J\_array, J)  ##  ## done single class  ##  return J  def \_log\_loss(self,y,y\_pred):  ##  ## Generalized cross-entropy. y input is a OneHot matrix  ##  J\_matrix = y\*np.log(y\_pred)  J =-np.sum(J\_matrix)  return J    def \_log\_loss\_reg(self,y,y\_pred,lmbd, theta, m):  J\_temp = self.\_log\_loss(y,y\_pred)  # now apply ridge (L2) regularization  J\_temp = J\_temp / m + 0.5 \* lmbd \* np.sum(theta\*theta)  return J\_temp  def CostFunctionSoftmax(self,x, ohm\_y):  ### 1 step stohastic softmax training based on gradiend descent  ### works BOTH for single observation and multiple observations  ### y is not  if self.method != 'softmax':  raise Exception('Softmax function called from non-softmax classifier')  if x.ndim != 2:  raise Exception('Softmax function received x with ndim!=2')  m = np.float64(x.shape[0]) # batch update size not all obs !!!  Theta = np.array(self.Theta)  xT = x.dot(Theta)  yhat = self.softmax(xT)  yhat = np.clip(yhat,self.eps,1-self.eps)  # now final calc incl reguralization  J = self.\_log\_loss\_reg(ohm\_y,yhat, self.lmbd, Theta, m)  self.LastGThe = Theta  self.LastM = m    TempG = ohm\_y - yhat  self.LastYERR = TempG  self.LastYHAT = yhat  self.LastYOHM = ohm\_y  self.LastObs = x  Grad = (-1.0/m) \* x.T.dot(TempG)  Grad += self.lmbd\*Theta      return Grad, J    def SoftmaxTrain(self,xi,yi):  cur\_m = xi.shape[0]  SparseBoolLabels = np.zeros(shape=(cur\_m,self.nr\_Classes))  softmax\_y = np.zeros(shape=(cur\_m,1))    for i,k in zip(range(self.nr\_Classes),self.Classes):  where\_y = np.where(yi==k)  if where\_y[0].size>0:  if len(where\_y)>1:  row, column = where\_y  else:  row = where\_y  softmax\_y[row] = i+1  # now calculate 1(Yi==col) sparse boolean matrix  SparseBoolLabels[row,i] = 1    """  else: ???  self.SparseBoolLabels = np.zeros(shape=(1,self.nr\_Classes))  self.SparseBoolLabels[self.Classes==yi] = 1  """  # prepare gradient storage  self.gradients = np.append(self.gradients,  np.zeros(shape=(1,  self.nr\_Classes,  self.n)), axis = 0)    if self.alpha\_search\_iter == 200:  test\_stop = True  ### now the gradient descent step  if self.softmax\_alpha\_search:  if self.alpha\_search\_iter < self.alpha\_search\_epochs:  bestAlpha = self.SearchBestAlpha(xi,SparseBoolLabels, Verbose = False)  self.alpha = bestAlpha  self.BestAlphas.append(bestAlpha)    Grad, J = self.CostFunctionSoftmax(xi,SparseBoolLabels)  self.LastGrad = Grad  self.LastJ = J  ### store gradient  self.gradients[-1,:,:] = Grad.T  ### done store gradient  self.Theta = self.Theta - self.alpha\*Grad  self.LastAlph = self.alpha  self.LastThet = self.Theta  self.Costs = np.append(self.Costs,J)  Jmean = np.sum(self.Costs)/self.Costs.shape[0]  self.J\_array = np.append(self.J\_array, Jmean)    return J    ##  ## Stohastic Gradient Check with option of selecting best weights  ## based on cross dataset  ##  def OnlineTrain(self,x\_input, y\_input, X\_cross = None, y\_cross = None):  """ obsolete since mini batch  ## data comes in only 1 dim  if x\_input.shape[0]!=(self.original\_n-1):  raise Exception("Check your data ! x has wrong size (expected="+str(self.n-1)+' received='+x\_input.shape[1]+')')  """    xi = self.prepare\_x(x\_input)  self.batchsize = xi.shape[0]  yi = np.array(y\_input)    ## now add the observation to observation matrices  self.add\_observation(xi,yi)    ## now prepare actual Xi  xi,yi = self.get\_train\_obs()      if (self.m == 0):  self.DebugInfo("[DEBUG] Beginning training: "+self.GetShortHyperParams(), 10)  #now increment nr of received examples  self.m+=1    if (self.m % 100) ==0:  self.DebugInfo("[DEBUG] Training the observation/batch nr {}".format(self.m), 10)      if self.method == 'sigmoid': # sigmoid/logistic single or multi class  J = self.LogisticTrain(xi,yi)  elif self.method == 'softmax': # softmax  J = self.SoftmaxTrain(xi,yi)  self.alpha\_array = np.append(self.alpha\_array, self.alpha)    if self.DecreasingAlpha:  N = 1 #batch size  if self.alpha\_coef>0:  self.alpha = self.alpha \* self.alpha\_coef  else:  self.alpha = float(self.alpha\_0) / (1.0 + (self.m / N ))    ##  ## now cross check  ##  if (not (X\_cross is None)) and (not (y\_cross is None)):  y\_cross\_pred,ydf = self.Predict(X\_cross)  y1 = np.ravel(y\_cross\_pred)  y2 = np.ravel(y\_cross)  preds = y1 == y2  my\_pred = (np.sum(preds)/float(X\_cross.shape[0]))\*100  if my\_pred>=self.BestAccuracy:  self.BestAccuracy = my\_pred  self.BestTheta = np.array(self.Theta)  self.BestFeed = self.m  else:  self.BestTheta = np.array(self.Theta)  self.BestFeed = self.m    return J  def \_PredictSoftmax(self,X,Theta,real\_y):  y = None  y\_floats = None  m=X.shape[0]  X = self.prepare\_x(X) # add poly, intercept  y\_floats = self.softmax(X.dot(Theta))  y\_indices = np.argmax(y\_floats,axis=1)  y = np.empty((0,1))  for i,label in zip(range(m),y\_indices):  y = np.append(y,np.array([[self.Classes[label]]]),axis=0)    return y,y\_floats    def \_PredictSigmoid(self,X,Theta,real\_y):  m=X.shape[0]  X = self.prepare\_x(X) # add poly, intercept    if self.MultiClass:  # predict multi class  y\_floats = self.sigmoid(X.dot(Theta))  y\_indices = np.argmax(y\_floats,axis=1)  y = np.empty((0,1))  for i,label in zip(range(m),y\_indices):  y = np.append(y,np.array([[self.Classes[label]]]),axis=0)  else:  # predict single class 0/1  y\_floats = self.sigmoid (X.dot(Theta))  y = np.round(y\_floats)    return y, y\_floats  def Predict(self, X,real\_y = None, Best = False, return\_floats = False, return\_df = True):  if Best:  if self.BestTheta is None:  raise Exception("BestTheta is "+self.BestTheta.tostring())  Theta = self.BestTheta  else:  Theta = self.Theta    if self.method == 'sigmoid':  y,y\_floats = self.\_PredictSigmoid(X,Theta,real\_y)  elif self.method == 'softmax':  y,y\_floats = self.\_PredictSoftmax(X,Theta,real\_y)  else:  raise Exception("Unknown method: "+self.method)    if (self.Classes == self.\_standard\_binary\_classes) and \  (self.nr\_Classes==2):    if self.method == 'softmax':  columns=["0","1"]  else:  columns=["0/1"]    else:  columns = self.Classes  y\_df = pd.DataFrame(y\_floats,columns=columns)    if not (real\_y is None):  y\_df['Real Y'] = real\_y    y = np.ravel(y)    if return\_floats:  return y,y\_df,y\_floats  else:  if return\_df:  return y,y\_df  else:  return y    ###  ### End OnlineClassifier class  ###        if \_\_name\_\_ == '\_\_main\_\_':  raise Exception("Class only file") |