Analyzing Air Quality and Liver Disease Using Binary Logistic Regression

Max Ardito
maxwell.ardito@mail.mcgill.ca

Edwin Meriaux
edwin.meriaux@mail.mcgill.ca

Ohood Sabr
ohood.sabr@mail.mcgill.ca

Abstract

This work focuses on using a binary logistic regression model to try and solve two different problems related to health and medicine. The first problem attempts to classify polluted air samples provided a dataset containing information about airborn chemicals, and the second problem attempts to classify instances of liver disease in patients provided similar biochemical data. A common gradient descent approach was used to train a binary classifier, and k-fold cross validation was used to choose the final model. After implementing the base regression model, various techniques were used in an attempt to improve accuracy, including brute-force feature reduction, regularization, feature expansion, and principal component analysis (PCA). It was observed that most of these techniques made very some difference in improving the mean accuracy of the training and validation set. However, these differences were very subtle, even when using optimal hyper-parameters. We concluded that the largest qualitative improvement in the model was found in using a combination of feature reduction, feature expansion, and PCA.

1 Introduction

There are a number of approaches for forecasting data of various forms in the field of machine learning. Classification is one of these approaches, which includes many types of algorithms such as Naive Bayes, Decision Trees, and Logistic Regression. The type of data used, the dimensions of the dataset, and the aptitude of the dataset's features have a significant impact on the model or classifier that is chosen [8, 10].

This work focuses on using binary logistic regression to try and predict the likelihood of a patient testing positive for liver disease and the likelihood of an air sample being polluted, given a number of respective chemical features. Logistic regression is a supervised learning algorithm that is used to solve binary classification problems. It calculates the likelihood of an event occurring based on a collection of variables that are assumed to be independent from one another, thus drawing decision boundaries between data points in N-dimensional feature space in an attempt to classify future data [3]. This type of classification is widely applied in studies about the health sciences because it is especially suitable for models involving illness condition (diseased or healthy) and decision making (yes or no) [4].

In order to build a logistic regression model from scratch, we first must define an error function such as the negative log likelihood or "cross-entropy" function (1) that we attempt to minimize by taking its derivative and using gradient descent (2) to slowly and iteratively modify the weights and the bias of the linear model. [7]

$$Err(w) = -\sum_{i=1}^{N} y_i \ln(\mu_i) + (1 - y_i) \ln(1 - \mu_i) \tag{1}$$

$$\Delta = \frac{\partial Err(w_k)}{\partial(w_k)} = -\sum_{i=1}^{N} x_i(y_i - \mu_i) \tag{2}$$

In (1) and (2), μ is equal to the sigmoid function $\sigma(w^T x_i) = P(y_i|x_i, w) = \frac{1}{1+e^{-w^T x}}$ which is used to transform the continuous output into a discrete binary measure (0 or 1) during classification, depending on whether its output is greater than or less than 0.5.[11] The weights in this model are updated following the rule $w_{k+1} = w_k + \alpha_k \Delta$, where α_k is a hyper-parameter representing the step-size [2].

The overall success of logistic regression depends heavily on finding ideal hyper-parameters, variations on the given dataset, and expansions on the model. In Section 2, we describe the Datasets and give an overview of the features analysis. In Section 3, the various results are presented which include a model selection and feature selection and expansion. In Section 4, we the discuss our findings in conclusion and propose some future work. Section 5 gives a statement of contribution, and finally an Appendix is given containing the complete complementary code, plots, and results.

2 Datasets

2.1 Overview

In this study, the aforementioned logistic regression algorithm is employed to carry out classification on two different datasets. The first dataset represents information on air quality to estimate the probability of air pollution. The second datast contains similar information to determine the probability of a patient having liver disease. In order to classify future data with known feature information and unknown class information, we have built and tested a binary logistic regression model from scratch by using the Python programming language and the Google Collab environment.

The original air quality dataset contains 1599 samples and 11 features each, with a class value of 1 representing a sample of polluted air and a class value of 0 representing normal air. The dataset for liver disease contains 330 samples of 8 features each, with class 1 assigned to a positive diagnosis for liver disease and class 0 assigned to a negative diagnosis. Furthermore, all of the features in each dataset are continuous.

2.2 Feature Analysis

In Figure 1, we can see that most of the features in the liver disease dataset form Gaussian distributions. This is the same case for the air quality dataseet (see Appendix A for air quality histograms). In a hypothetical feature selection stage in which features deemed unnecessary or noisy are removed, these distributions can serve as qualitative tools to both improve and justify the accuracy of our model. For instance, the variance between the two binary classes of each respective feature can be calculated, but also judged qualitatively by looking at the histogram plots. Features that hold the least amount of variance—in other words, features with the most overlap in their distributions—might serve as unnecessary in the training process due to their indistinguishable nature between class 0 and class 1. These methods will be discussed further in Section 3.2, as well as methods for improving the model by expanding the feature space.

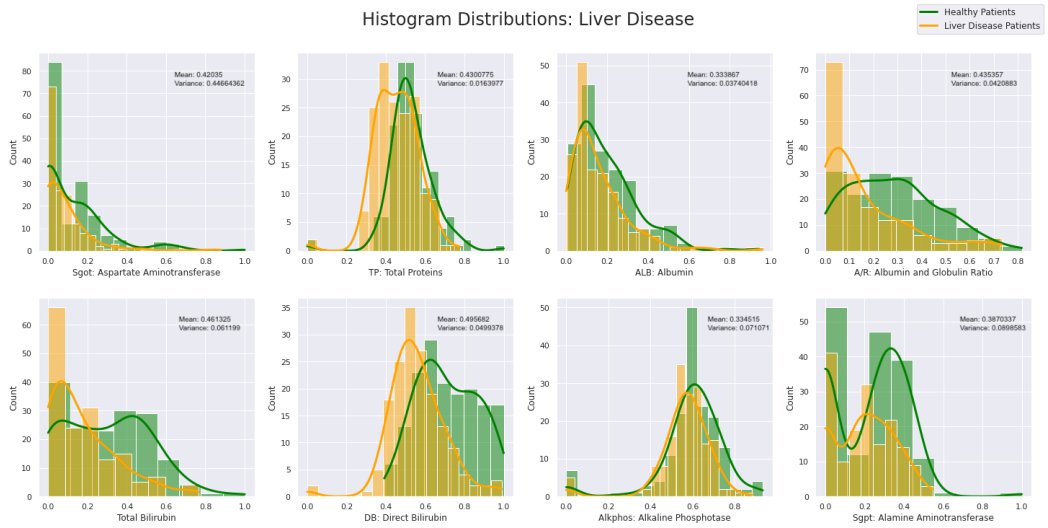

Figure 1: Histogram plot of each liver disease feature comparing data points from people who tested positive (yellow) and tested negative (green). See Appendix A for a similar histogram of the air quality dataset.

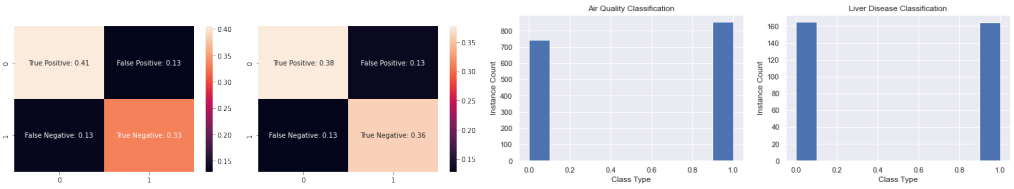

Figure 2: (Left) Confusion matrices for base logistic regression model $\mathcal{M}$ run on air quality dataset and liver disease dataset. (Right) Histogram representing the total count of data instances of each class.

3 Results

3.1 Model Selection

In experimenting with different variations of the logistic regression model, we started with a base model $\mathcal{M} = \{X, y, \alpha, k\}$ where X and y are the training set's feature matrix and class vector respectively, α represents the step size, and k represents the number of steps taken during gradient descent.

The base model $\mathcal{M}$ trains the binary logistic regression model on each dataset given strictly as-is with no additional feature reduction or feature expansion methods. However, one row-reduction step was implemented prior to training that removes identical rows. The logic behind this is to remove biases in the data. The odds that two points are identical over 8-11 features are low. In this case there were only a few duplicates over each individual dataset, which means duplicates don't currently affect the model too much, but very well could affect performance if more data was added.

It was observed that the performance of each model discussed in this section demonstrated similar trends in accuracy between both the liver disease dataset and the air quality dataset. But a few variations of the base model displayed some key differences in the performance of each dataset. These differences will be discussed further while each variation of the base model is discussed. A table containing the differences in performance between the two datasets is provided in Appendix A, and can be consulted throughout this section.

The base model $\mathcal{M}$ was found to yield the highest accuracy when setting $\alpha = 0.001$ and $k = 100000$. When trained on a 10-fold cross validation, the model yields a mean accuracy score of 73.41% on the training data and 71.88% on the validation set data for liver disease, as well as a mean accuracy of 74.42% on the air quality training set and 73.63% on the air quality validation set. In the event that these two datasets were not initially normalized, we added a normalization algorithm prior to the model training. This was found to increase the accuracy of the model by anywhere from 1-3% on the air quality dataset, but made little to no improvement on the liver disease dataset.

For both datasets, the fluctuation in accuracy on each fold of the training set was found to range anywhere from 72% to 76%, while the validation set for each fold was observed to range anywhere between 59.38% and 81.25%, even despite the model's mean accuracy on the training and validation sets being nearly equal. It is possible to conclude from this information that model $\mathcal{M}$ might produce an unpredictably high variance, given a larger version of the dataset. An L1 regularization was thus implemented on the model to penalize any overfitting that might be occurring.

$$\hat{w} = \text{argmin}_{\text{w}}\{\sum_{i=1}^{N} y_i - (w^T x_i)^2 + \Lambda\} \begin{cases} \Lambda \triangleq \lambda \sum_{j=0}^{M} |w_j| & \text{for L1} \\ \Lambda \triangleq \lambda \sum_{j=0}^{M} w_j^2 & \text{for L2} \end{cases} \tag{3}$$

By implementing the L1 cost function in (3), we derived a model $\hat{\mathcal{M}} = \{X, y, \alpha = 0.001, k = 100000, \lambda = 0.001\}$ containing the added λ hyper-parameter. We concluded that L1 regularization tends to improve this fluctuation in the validation set's results of both datasets, but makes little to no difference in the mean accuracy of each model. Additionally an L2 regularization was also implemented but made no quantitative difference to the any of the base model's accuracy metrics.

3.2 Feature Selection and Expansion

Feature selection techniques proved to be more successful in decreasing the base model's variance. Our approach combined implementing a brute force technique to calculate which features to remove and qualitatively analyzing the aforementioned histograms.

$$\mathscr{P}(y) = 2^N, |y| = N \tag{4}$$

In order to figure out how many features to remove from the model, we first derived the power set $\mathscr{P}(y)$ containing all N subsets of our feature vector y and calculated the mean accuracy for each subset $\{s\}^y$.[6] We then compared each of these accuracy measures for each subset using cross-validation and calculated which one had the least error (See Appendix A - Figure 4).

When manually removing features our findings suggested that the accuracy of the model decreases significantly after removing more than one feature, but increases slightly when removing a single feature just like from the air quality dataset. As discussed in Section 2.2, analyzing the histograms in Figure 1 led us to believe that removing a feature whose binary class distributions overlapped the most might be advantageous. These can be quantitatively decided by using the variance values captured in Figure 1, and were decidedly TP: Total Proteins in the liver disease dataset and Temperature in the air quality dataset. Indeed, dropping these two features and retraining resulted in a consistent 1% increase in both the validation set accuracy and the total accuracy of the model trained on each dataset. But removing multiple features can increase the accuracy when all combos are checked. The top non brute force accuracy in the air quality dataset is with PCA and Expansion with a 75.07 percent accuracy but this is increased to 76.7 percent with brute force reduction where features 4, 6, 8, and 9 were removed (from the original dataset). A similar effect is seen in the liver disease dataset where the top accuracy goes from 74.6 percent to 74.74 percent. The only features removed were

1, 4, 5, and 8 (from the original dataset). This method of course is very time intensive so its utility decrease as the dataset grows more and more at an exponential rate.

After focusing on variance, we decided to try and construct a more complex feature space, as this is a common methodology for lowering a model's bias [8]. Starting from the feature-reduced model from above, a feature expansion on the model was implemented, thus adding powers 2, 3, 1/2, 1/3, and the natural log into the data's feature space. Of course, there are also other possible values that the data can be expanded to like exponential but for the sake of this experiment, this proved the point of feature expansion. By itself feature expansion reduced validation accuracy, but in combination with a Principal Component Analysis (PCA) it helped increase accuracy by about 1-1.5%.

Deriving a dataset from performing Principal Component Analysis (PCA) reduces the dimensionality of the feature space into the first N principal components, privileging new columns that describe the most variance in the data over lower-ranking components that contain unnecessary information.

$$\text{argmin}_{\text{P,U}}\{\sum_{i=1}^{N} ||x_i - UP^T x_i||^2\} \tag{5}$$

Using eigenvalue decomposition, a matrix of eigenvectors P is derived and sorted, representing N vectors ordered by the amount of variance captured in the original data. This decomposition method is optimized by (5) to ensure that the minimum amount of information is lost from compression of the original data x_i to decompression of the principal components P^T through multiplying by U [1].

We cross validated the model for training sets that used the first 1 - 10 principal components and compared the accuracy with one another. Our findings showed that by simply using the first 4 principal components, our liver disease model yielded both the highest accuracy and lowest variance yet scoring 74.44% on the training set and 74.06% on the validation set. Similarly, the air quality dataset scored an accuracy of 75.07% trained on the first 9 principal components. These models also used the feature reduction, expansion, and normalization steps mentioned earlier.

4 Discussion and Conclusion

We have seen that using a number of variations on the binary logistic regression model have proven to marginally increase the accuracy of classifying unknown air quality and medical patient data. The extension on the base model that yielded the highest accuracy was a model that used a number of techniques prior to the training process including data normalization, feature reduction, feature expansion, and PCA. These techniques provided an improved dataset for which we trained the logistic regression model.

Possible future work could include investigating augmented gradient descent techniques such as Stochastic Gradient Descent (SGD) and the Stochastic Average Gradient (SAG) [9]. These techniques are used heavily in the SciKit Learn implementation of the binary logistic regression model [5]. An additional step for improving the model could be to collect more data, which might improve the accuracy of our model significantly.

5 Statement of Contribution

1. Max Ardito: Logistic regression, K-Fold, evaluation methods, writing report, principal component analysis, plots, regularization

2. Ohood Sabr: Logistic regression, K-Fold, writing report

3. Edwin Meriaux: Logistic regression, principal component analysis, brute force feature selection, feature expansion, average feature decrease

References

[1] Narges Armanfard. Ecse 551 - machine learning for engineers lecture 13 — dimension reduction and knn, October 2022.

[2] Narges Armanfard. Ecse 551 - machine learning for engineers lecture 5–linear regression, overfitting, logistic regression, September 2022.

[3] Hoss Belyadi and Alireza Haghighat. Chapter 5 - supervised learning. In Hoss Belyadi and Alireza Haghighat, editors, *Machine Learning Guide for Oil and Gas Using Python*, pages 169–295. Gulf Professional Publishing, 2021.

[4] Ernest Yeboah Boateng and Daniel A Abaye. A review of the logistic regression model with emphasis on medical research. *Journal of data analysis and information processing*, 7(4):190–207, 2019.

[5] https://scikit learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html. sklearn.linear_model.logisticregression, Oct 2022.

[6] https://www.geeksforgeeks.org/power set/. Power set, Aug 2022.

[7] Kevin P. Murphy. *Machine learning: A probabilistic perspective*, page 247–249. MIT Press, 2021.

[8] omdena.com/blog/machine-learning-classification algorithms. Best machine learning classification algorithms + real-world projects, 2022.

[9] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, 162(1):83–112, 2017.

[10] Pratap Chandra Sen, Mahimarnab Hajra, and Mitadru Ghosh. Supervised classification algorithms in machine learning: A survey and review. In *Emerging technology in modelling and graphics*, pages 99–111. Springer, 2020.

[11] Xiaonan Zou, Yong Hu, Zhewen Tian, and Kaiyuan Shen. Logistic regression model optimization and case analysis. In *2019 IEEE 7th International Conference on Computer Science and Network Technology (ICCSNT)*, pages 135–139, 2019.

5.1 Appendix A. - Plots

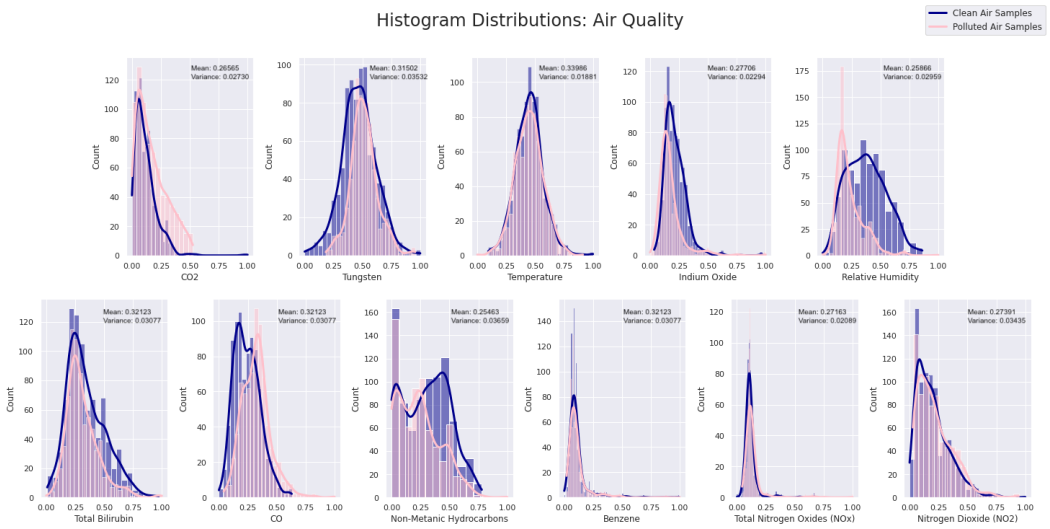

Figure 3: Histogram plot of each air quality feature comparing data points from clean air samples (purple) and polluted air samples (purple).

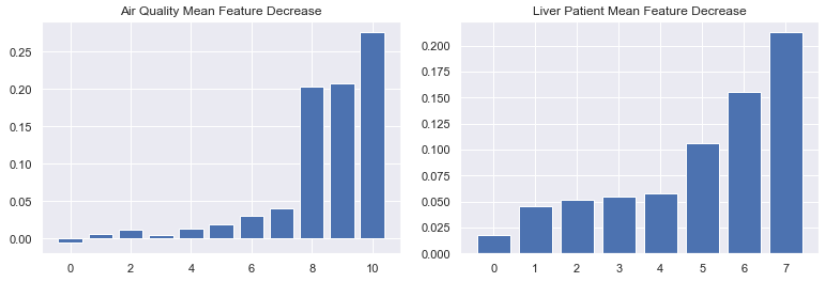

Figure 4: Brute force tests for the mean error of the model when dropping the number of features displayed on the X axis.

5.2 Appendix B. - Accuracy Table

Table 1: Mean accuracy metrics between air quality data and liver disease data for various logistic regression models trained using 10-fold cross-validation

Model	Air Quality	Liver Disease
Base Logistic Regression	72.3%	71.56%
Normalized	73.78%	70.95%
Regularized (L1) + Normalized	74.07%	70.93%
Regularized (L2) + Normalized	73.4%	71.56%
Feature Reduction + Normalized	74.5%	72.5%
Feature Reduction + Feature Expansion + Normalized	74.9%	71.56%
Feature Reduction + Feature Expansion + PCA + Normalized	75.07% [PC = 9]	74.68% [PC = 4]

5.3 Appendix C. - Code

```
# -*- coding: utf-8 -*-
# -*- coding: utf-8 -*-
"""Logistic_Regression_10-10.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1vFsscCXpbgXur3w8On_tpxEa_feUOpYj

# Logistic Regression

# Regression Class

Class containing tools for a binary logistic regression
"""

import numpy as np
from numpy import log as ln
from enum import Enum
import math

"""
  Class for implementing a binary logistic regression. Includes additional
  functionality for penalizing weights using L1 and L2 regularization.

  Code citations: https://github.com/python-engineer/MLfromscratch
"""
class LogisticRegression:

    def __sigmoid(self, a: float):
        return (1 / (1 + np.exp(-a)))

    class Regularization(Enum):
        NONE = 0
        L1 = 1
        L2 = 2

    def __init__(self, step_size: float, iterations: int):
        self.step_size = step_size
        self.iterations = iterations

    def fit(self,
            X: np.ndarray,
            Y: np.ndarray,
            regularization: Regularization = Regularization.NONE,
            lmda: float = 0):
        # Initialize random weights and bias
        self.w = np.random.rand(X.shape[1])
        self.bias = np.random.rand()
        y_hat = np.ndarray(Y.shape)

        # Perform gradient descent
        for _ in range(self.iterations):
            wTx = np.dot(X, self.w) + self.bias
            y_hat = self.__sigmoid(wTx)

            # Implement a Lasso or Ridge regularization if specified
            if (regularization == self.Regularization.L1):
              dW = (1 / X.shape[0]) * np.add(np.dot(X.transpose(),
                                                    (y_hat - Y.flatten())),
                                                    ↪  np.dot(self.w, lmda * 2))
```

```
            elif (regularization == self.Regularization.L2):
              dW = (1 / X.shape[0]) * np.add(np.dot(X.transpose(),
                                                    (y_hat - Y.flatten())),
                                                    np.dot(self.w, np.sign(self.w)
                                                    * lmda))
            else:
              dW = (1 / X.shape[0]) * np.dot(X.transpose(), (y_hat -
               Y.flatten()))

            dB = (1 / X.shape[0]) * np.sum(y_hat - Y.flatten())

            self.w -= self.step_size * dW
            self.bias -= self.step_size * dB

    def predict(self, X_new):
        #prediction of the LR model
        wTx = np.dot(X_new, self.w) + self.bias
        y_hat = self.__sigmoid(wTx)

        for i in range(y_hat.shape[0]):
            if y_hat[i] > 0.5:
                y_hat[i] = 1
            else:
                y_hat[i] = 0

        return y_hat

"""# Evaluation

Class containing various evaluation tools
"""

import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
"""
    Class for evaluating the performance of a logistic regression model.
    Includes tools for calculating confusion matrix, accuracy, precision,
    recall, specificity, and false positive rate.

    Assumes that the input data is the result of a binary logistic regression
    (e.g. y && y_hat = {0, 1})
"""


class LogisticEvaluation:
    """
        Initializs the evaluation from a vector of predicted binary values (y)
        and a vector of actual values (y_hat). Stores these values in a
        confusion matrix variable (cm) as well as individual cell
        values (tp, tn, fp, fn)
    """

    def __init__(self, y_hat: np.ndarray, y: np.ndarray):
        size = y_hat.size
        self.cm = np.array([[0, 0], [0, 0]])
        for i in range(size):
            if (y_hat[i] == 1) and (y[i] == 1):
                self.cm += np.array([[1, 0], [0, 0]])
            elif (y_hat[i] == 0) and (y[i] == 0):
                self.cm += np.array([[0, 0], [0, 1]])
            elif (y_hat[i] == 1) and (y[i] == 0):
                self.cm += np.array([[0, 1], [0, 0]])
```

```
            else:
                self.cm += np.array([[0, 0], [1, 0]])
        # Normalize confusion matrix
        self.cm = np.divide(self.cm, np.array(size))

        # Store individual table values
        self.tp = self.cm[[0], [0]].item()
        self.tn = self.cm[[1], [1]].item()
        self.fp = self.cm[[1], [0]].item()
        self.fn = self.cm[[0], [1]].item()

    """
        Prints a heatmap of the confusion matrix
    """

    def confusion_matrix(self):
        df = pd.DataFrame(self.cm)
        fig = plt.figure()
        cell_labels = np.array(
            [[
                "True Positive: " + str(round(self.tp, 2)),
                "False Positive: " + str(round(self.fp, 2))
            ],
             [
                 "False Negative: " + str(round(self.fn, 2)),
                 "True Negative: " + str(round(self.tn, 2))
             ]])
        print(round(self.tp, 2),round(self.fp, 2),round(self.fn,
        ↪  2),round(self.tn, 2))

        sns.heatmap(df, fmt='', annot=cell_labels)
        plt.savefig("Heatmap.png")
        return self.cm

    """
        Returns the model's accuracy
    """

    def accu_eval(self):
        accuracy = (self.tp + self.tn) / (self.cm.sum())
        return accuracy

    """
        Returns the model's precision
    """

    def precision_eval(self):
        precision = self.tp / (self.tp + self.fp)
        return precision

    """
        Returns the model's recall
    """

    def recall_eval(self):
        recall = self.tp / (self.tp + self.fn)
        return recall

    """
        Returns the model's specificity
    """

    def spec_eval(self):
        specificity = self.tn / (self.fp + self.tn)
```

```
        return specificity

    """
        Returns the model's false positive rate
    """

    def fpr_eval(self):
        fp_rate = self.fp / (self.fp + self.tn)
        return fp_rate

"""# K-Fold Cross Validation Class

Class that uses the logistic regression class in the context of a K-fold cross
↪  validation. Optional flags can be set for normalizing, expanding, and/or
↪  using the first N principal components to train instead of the features
↪  themselves
"""

import pandas as pd, numpy as np
from sklearn.preprocessing import StandardScaler
"""
        Class for performing K-fold cross validation and
        returning its mean error. Ideally used for comparing
        the performance of multiple logistic regression models.
"""


class KFold:
    """
        Initializes class by shuffling the input data and obtaining
        a validation set size based on the input dimensions and specified
        K value.

    """

    def __init__(self,
                 X: np.ndarray,
                 k: int,
                 step_size: float,
                 iterations: int,
                 Normalization_toggle: int,
                 Regularization_toggle: int,
                 PCA_toggle: int,
                 Expansion_toggle: int,
                 pca_size: int,
                 broute_force_toggle: int):
        # Remove dupliacate data points
        self.X = np.unique(X, axis=0)
        self.k = k
        self.step_size = step_size
        self.iterations = iterations
        self.PCA_toggle = PCA_toggle
        self.Normalization_toggle = Normalization_toggle
        self.Regularization_toggle = Regularization_toggle
        self.Expansion_toggle = Expansion_toggle
        self.pca_size = pca_size
        self.brout_force_toggle = broute_force_toggle
        self.best_accuracy = 0
        self.best_combo = []

        # Validation size set = number of rows / k
        self.validation_set_size = int(self.X.shape[0] / k)

        # Shuffle the rows of the input data
```

```
246         np.random.shuffle(self.X)
247
248     """
249        Performs the cross validation on K iterations of the input data.
250        The cross validation is performed by taking the first validation
251        set from the top of the input data and then subsequently shifting
252        (rolling) the input data N elements, where N = validation set size.
253
254        Note that this method of cycling and partitioning will automatically
255        throw any remainder of input data into the validation set if the data
256        set cannot be evenly divided into K sets.
257     """
258     #calculates variance and expectation based on np libraries
259     def varience_expectation(self,Xdata):
260       print(Xdata.shape[1])
261       for i in range(Xdata.shape[1]):
262           print("mean: ",np.mean(Xdata[i]))
263           print("variance: ",np.var(Xdata[i]))
264
265     #power set calculation for the brute for feature reduction
266     #https://www.geeksforgeeks.org/power-set/ source used to help with the bit
       ↪  shift if statement
267     def powerset(self,fullset):
268       listsub = list(fullset)
269       subsets = []
270       for i in range(2**len(listsub)):
271         subset = []
272         for k in range(len(listsub)):
273           if i & 1<<k:
274             subset.append(int(listsub[k]))
275         subsets.append(subset)
276       return subsets
277
278     #normalization calculation based on expectation (mu) and standard
       ↪  deviation (std)
279     def normalization(self, Xtrain):
280       normalized = Xtrain
281       #operation on each feature in dataset
282       #in theory this can be accelerated if a custom np function was used
283       #would remove the need for the for loop
284       #the function would effectively be everything in the current for loop
285       for i in range(Xtrain.shape[1]):
286         #mu calculation
287         mu = np.sum(Xtrain[:,i])/Xtrain.shape[0]
288         #std calculation
289         std_subraction = np.subtract(Xtrain[:,i],mu)
290         std_square = np.square(std_subraction)
291         std_sum = np.sum (std_square)/Xtrain.shape[0]
292         std = std_sum**(1/2)
293         #normalization calculation
294         normalized[:,i] = np.subtract(Xtrain[:,i],mu)/std
295
296       return normalized
297
298     # Principal Component Analysis function for dimensionality reduction
299     # Code citation:
       ↪  https://www.askpython.com/python/examples/principal-component-analysis
300     def PCA(self, X, num_components):
301       # Center the data set by subtracting the mean
302       X_mean = X - np.mean(X , axis = 0)
303
304       # Get the covariance matrix
305       covariance_matrix = np.cov(X_mean , rowvar = False)
306
```

```
307         # Calculate the eigenvalues and eigenvectors of the covariance matrix
308         eigen_values , eigen_vectors = np.linalg.eigh(covariance_matrix)
309 
310         # Sort the eigenvalues and eigenvectors in descending order
311         sorted_index = np.argsort(eigen_values)[::-1]
312         sorted_eigenvalue = eigen_values[sorted_index]
313         sorted_eigenvectors = eigen_vectors[:,sorted_index]
314 
315         # Take a subset of the sorted vectors to represent the first N principal
            ↪  components
316         eigenvector_subset = sorted_eigenvectors[:,0:num_components]
317 
318         # Reduce dimensionality by taking the dot product of the transposed
319         # principal components mean of the original data set
320         X_reduced = np.dot(eigenvector_subset.transpose(),
            ↪  X_mean.transpose()).transpose()
321 
322         return X_reduced
323 
324       #power calculation for feature expansion for which ever power needed ^2
          ↪  ^3...
325       def to_power(self, data, power):
326         return data ** power
327 
328       #power log calculation for the features expansion
329       def natural_log(self, data):
330         math.log1p(data)
331         return math.log1p(data)
332 
333       #mean feature decrease calculation to find the importance of each feature
334       def feature_importance(self,X_train,Y_train):
335 
336         iterations = 10000
337         learning_rate = 0.0015
338 
339         #setting up the baseline for the test results assuming all features
            ↪  tested on
340         W, B, true_cost_list, true_accuracy, true_confusion = model(X_train,
            ↪  Y_train, learning_rate = learning_rate, iterations = iterations)
341         feature_number_list = [i for i in range(X_train.shape[1])]
342 
343         #data storage for feature drop comparision
344         importance_accuracy = []
345         importance_confusion = []
346         importance_cost = []
347         for i in range(X_train.shape[1]):
348             #runs test on each set of features. Set of features being n-1
                ↪  features of n combinations
349             tmp_list = feature_number_list
350             tmp_list.remove(i)
351             tmp_array = X_train[:,tmp_list]
352             #current Logitic Regression test with modified dataset (see line
                ↪  above)
353             W, B, cost_list, accuracy, confusion = model(tmp_array, Y_train,
                ↪  learning_rate = learning_rate, iterations = iterations)
354 
355             #data storage for future plots
356             importance_accuracy.append(true_accuracy - accuracy)
357             importance_confusion.append(confusion)
358             importance_cost.append(cost_list)
359             print(true_accuracy - accuracy)
360         print(importance_accuracy)
361         return importance_accuracy #data return
362 
```

```
    #plotting mean feature decrease data
    def plot_results(self,data,val_list):
      plt.title("Air Quality Data Importance") #line changed depending on the
      ↪  data being run for feature decrease
      plt.bar(val_list,data)
      plt.show()

    #feature expansion for the data set to certain powers
    # powers: 2,3,1/2,1/3 and natural log
    def Feature_Increase(self, xdata):
      new_xdata = xdata
      for i in range(xdata.shape[1]):
          #higher exponential powers
          for j in range(2,4):
              tmp = xdata[:,i]
              tmp_vector = np.vectorize(self.to_power)
              tmp = np.asmatrix(tmp_vector(tmp,j))
              new_xdata = np.hstack((new_xdata, tmp.T))
          #1/n power of exponential powers
          for j in range(2,4):
              tmp = xdata[:,i]
              tmp_vector = np.vectorize(self.to_power)
              tmp = np.asmatrix(tmp_vector(tmp,1/j))
              new_xdata = np.hstack((new_xdata, tmp.T))
          #natural log power
          tmp = xdata[:,i]
          tmp_vector = np.vectorize(self.natural_log)
          tmp = np.asarray(tmp_vector(tmp))
          tmp = np.asmatrix(tmp)

          new_xdata = np.hstack((new_xdata, tmp.T))
      print(new_xdata)
      print(new_xdata.shape)
      return new_xdata.real

    def cross_validation(self):
        # Initialize error count
        self.train_err = 0
        self.test_err = 0

        # Separate the data's classes (Y) from the features (X)
        # and use expansion, normalization, and/or PCA if flags are set
        self.Y = np.array([self.X[:, -1]]).transpose()
        self.X = self.X[:, :-1]
        if self.Expansion_toggle == 1:
          self.X = self.Feature_Increase(self.X)
          self.X = np.asarray(self.X)
        if self.Normalization_toggle == 1:
          self.X = self.normalization(self.X)
        if self.PCA_toggle == 1:
          self.X = self.PCA(self.X,self.pca_size)
        if self.brout_force_toggle == 1:
          print(self.X.shape[1])
          print(list(range(self.X.shape[1])))
          test_set = self.powerset(list(range(self.X.shape[1])))

          for combo in test_set:
            print("____________")
            print("current combo: ",combo)
            print("best combo:", self.best_combo)
            print("best acc:", self.best_accuracy)
            print("____________")
            for i in range(self.k):
              tmpX = self.X[:,combo]
```

```
                        # Training set = all rows where index is larger than
                        ↪ validation set size
            X_train = tmpX[self.validation_set_size:, :]
            Y_train = self.Y[self.validation_set_size:, :]

            # Test set = all rows where index is below validation set size
            X_test = tmpX[:self.validation_set_size, :]
            Y_test = self.Y[:self.validation_set_size, :]

            # Initialize logistic regression class for training data
            logistic = LogisticRegression(self.step_size, self.iterations)

            # Fit the model (get Ws)
            if (self.Regularization_toggle == 2):
              logistic.fit(X_train, Y_train, logistic.Regularization.L2,
              ↪ 0.001)
            elif (self.Regularization_toggle == 1):
              logistic.fit(X_train, Y_train, logistic.Regularization.L2,
              ↪ 0.001)
            else:
              logistic.fit(X_train, Y_train)

            # Predict Y ouput on X training data to measure accuracy
            Y_hat_train = logistic.predict(X_train)

            # Predict Y output on X test data
            Y_hat_test = logistic.predict(X_test)

            # Evaluate the predicted Y with the actual Y from the training
            ↪ data
            self.train_eval = LogisticEvaluation(
                Y_hat_train, Y_train)

            # Evaluate the predicted Y with the actual Y from the test data
            self.test_eval = LogisticEvaluation(Y_hat_test, Y_test)

            # Accumulate the error
            self.train_err += self.train_eval.accu_eval()
            self.test_err += self.test_eval.accu_eval()

            print("Fold - ", i + 1, " / ", self.k)

            acc_train = self.train_eval.accu_eval() * 100
            acc_test = self.test_eval.accu_eval() * 100

            print("Accuracy (Training Data): ", float(f'{acc_train:.2f}'),
            ↪ "%")
            print("Accuracy (Validation Data): ", float(f'{acc_test:.2f}'),
                  "%")
            if acc_train > self.best_accuracy:
              self.best_accuracy = acc_train
              self.best_combo = combo

            # Shift the X data over a validation set size to ensure a new
            # validation set data for the next training iteration
            tmpX = np.roll(tmpX, -self.validation_set_size, axis=0)
            self.Y = np.roll(self.Y, -self.validation_set_size, axis=0)


        if self.brout_force_toggle == 0:

          # Train on training set, test on validation set
          for i in range(self.k):
```

```
484                 # Training set = all rows where index is larger than validation
                    ↪  set size
485                 X_train = self.X[self.validation_set_size:, :]
486                 Y_train = self.Y[self.validation_set_size:, :]
487
488                 # Test set = all rows where index is below validation set size
489                 X_test = self.X[:self.validation_set_size, :]
490                 Y_test = self.Y[:self.validation_set_size, :]
491
492                 # Initialize logistic regression class for training data
493                 logistic = LogisticRegression(self.step_size, self.iterations)
494
495                 # Fit the model (get Ws)
496                 if(self.Regularization_toggle):
497                   logistic.fit(X_train, Y_train, logistic.Regularization.L2,
                      ↪  0.001)
498                 else:
499                   logistic.fit(X_train, Y_train)
500
501                 # Predict Y ouput on X training data to measure accuracy
502                 Y_hat_train = logistic.predict(X_train)
503
504                 # Predict Y output on X test data
505                 Y_hat_test = logistic.predict(X_test)
506
507                 # Evaluate the predicted Y with the actual Y from the training
                    ↪  data
508                 self.train_eval = LogisticEvaluation(
509                     Y_hat_train, Y_train)
510
511                 # Evaluate the predicted Y with the actual Y from the test data
512                 self.test_eval = LogisticEvaluation(Y_hat_test, Y_test)
513
514                 # Accumulate the error
515                 self.train_err += self.train_eval.accu_eval()
516                 self.test_err += self.test_eval.accu_eval()
517
518                 print("Fold - ", i + 1, " / ", self.k)
519
520                 acc_train = self.train_eval.accu_eval() * 100
521                 acc_test = self.test_eval.accu_eval() * 100
522
523                 print("Accuracy (Training Data): ", float(f'{acc_train:.2f}'),
                    ↪  "%")
524                 print("Accuracy (Validation Data): ", float(f'{acc_test:.2f}'),
525                       "%")
526
527                 # Shift the X data over a validation set size to ensure a new
528                 # validation set data for the next training iteration
529                 self.X = np.roll(self.X, -self.validation_set_size, axis=0)
530                 self.Y = np.roll(self.Y, -self.validation_set_size, axis=0)
531
532         # Normalize error of all iterations of the validation set
533         self.train_err /= self.k
534         self.test_err /= self.k
535
536         print("-------------------------------------------")
537
538         total_acc_train = self.train_err * 100
539         total_acc_test = self.test_err * 100
540
541         print("Total Accuracy (Training Data): ",
542               float(f'{total_acc_train:.2f}'), "%")
543         print("Total Accuracy (Validation Data): ",
```

```
                float(f'{total_acc_test:.2f}'), "%")

        print("Confusion matrix for final fold: ")
        self.train_eval.confusion_matrix()
        return(total_acc_test)



"""# File Upload

Here you can upload the CSV you intend to use for the rest of the notebook.


NOTE: Running these cells prompts you to upload a CSV files. Choose the original `air_quality.csv` or `liver_disease.csv` files. Be sure to replace the filename in the `io.BytesIO` function with the name of the file you intend to upload

"""

#importing data from either google collab or locally
import io
from google.colab import files, drive

uploaded = files.upload()
df = pd.read_csv(io.BytesIO(uploaded['air_quality.csv']))
data = df.to_numpy()

# Main Program - Base Model
"""# Main Program - Optimal Normalized Model

Below is the optimal model discussed in the report: a variation on the base model using a normalization step implemented on the dataset before training, afeature reduction, a feature expansion, and 11 different PCA sets cross-validated with each other using 10-fold.
"""

import pandas as pd, numpy as np

def main():
    # Hyperparameters
    step_size = 0.001 #step sizes on the gradient descent
    iterations = 100000 #max epoch count of the test
    folds = 10 #number of folds on the kfolds
    normalization = 1 #normalization enabling bit (1 = active, 0 = inactive)
    regularization = 0 #regularization enabling bit (2 = active L2 regularization, 1 = active L1 regularization, 0 = inactive)
    pca = 1 #pca enabling bit (1 = active, 0 = inactive)
    pca_size = 11 #initiates the size of the pca set
    expansion = 1 #expansion enabling bit (1 = active, 0 = inactive)
    brute_force = 0 #brute_force enabling bit (1 = active, 0 = inactive)


    # Trains the model with no additional techniques using 10-fold cross validation
    k = KFold(data,
              folds,
              step_size,
              iterations,
              normalization,
              regularization,
              pca,
              expansion,
              pca_size,
```

```
                brute_force)

    acc = k.cross_validation()


    print("_________________________________")
    print("Total Accuracy: ", acc)
    print("_________________________________")


if __name__ == "__main__":
    main()

```