LMLVQ using distance Documentation

Release 1.1

Avinash Maheshwari

Oct 21, 2020

CONTENTS:

CHAPTER

ONE

ALGORITHM DESCRIPTION

Learning Vector Quantization(LVQ) is well-known for Supervised Vector Quantization. Large margin LVQ is to maximize the distance of sample margin or to maximize the distance between decision hyperplane and datapoints.

1.1 Pseudo-code

1. Get data with labels e.g. $x \in X, |X| = n, X \subset R^n$, where *X* are datapoints and $c(X) \in C$, where *C* are data labels.
2. Initialize prototypes with labels e.g. $w \in R^n$, where *w* are prototypes and $c(w) \in C$, where *C* are prototype labels.
3. Calculate Euclidean distance between datapoints and prototypes.

$$d_{i,j} = d_E(x_i, w_j) = \sqrt{\sum (x_i - w_k)^2}$$

4. Calculate closest correct matching prototype for every data point and also calculate $|P_k|$ is the number of data points for which w_k is the closest prototype with same label.
5. Calculate 1_{P_k}, A vector which has **1** where data point has closest prototype otherwise zero.
6. Compute A_k,

- The index $A_k[i, K * i + l]$, should be **+1** if data point *i* is in $|P_k|$, i.e. if prototype *k* is the closest prototype to data point *i* with the same label, and if prototype *l* has a different label.
- The index $A_k[i, K * i + k]$ should be **-1** if datapoint *i* has a different label than prototype *k*.
- The index $A_k[i, K * i + l]$ should be zero in all other cases. So most of A_k is zero.

7. Compute the cost function:

$$E = \min_{\vec{\lambda} \in \mathbb{R}^{m \cdot K}} \frac{1}{2} \vec{\lambda}^T \cdot \left(C \cdot I - \sum_{k=1}^{K} \mathbf{A}_k^T \cdot \frac{D}{|P_k|} \cdot \mathbf{A}_k \right) \cdot \vec{\lambda} - \left(\gamma \cdot \vec{1}^T + \sum_{k=1}^{K} \vec{1}_{P_k}^T \cdot \frac{D}{|P_k|} \cdot \mathbf{A}_k \right) \cdot \vec{\lambda}$$

- such that $\vec{\lambda} \geq 0$ and $\vec{1}^T \cdot \mathbf{A}_k^T \cdot \vec{\lambda} = 0, \forall k \in \{1, \ldots, K\}$

8. Finally updates the prototypes:

$$\lambda(t+1) = \lambda(t) - \eta \frac{\partial E}{\partial w(t)}$$

CHAPTER

TWO

INSTALLATION REQUIREMENTS

Following are the basic requirements to run this program:

1. python with minimum version 3.8.
2. numpy with minimum version 1.19.0.
3. matplotlib.
4. Scikit Learn .

2.1 Execution

- Open the **lmlvq_call.py** file and set parameters with margin, prototypes per class and epochs then run it.

CHAPTER

THREE

CLASSES AND FUNCTIONS

Created on Thu Oct 1 09:46:20 2020

@author: avinash

class lmlvq_distance.**LMLVQ**(*prototype_per_class=1*)

Bases: object

Large margin LVQ is to maximize the distance of sample margin or to maximize the distance between decision hyperplane and data point.

prototype_per_class: The number of prototypes per class to be learned.

A_k(*input_data*, *prototype_labels*, *w_plus_index*)

A_K used to translating the lambda numbers to the beta numbers.

The index A_k[i, K*i+l] should be +1 if data point i is in P_k, i.e. if prototype k is the closest prototype to data point i with the same label, _and_ if prototype l has a different label.

The index A_k[i, K*i+k] should be -1 if datapoint i has a different label than prototype k.

The index A_k[i, K*i+l] should be zero in all other cases. So most of A_k is zero.

input_data: A n x m matrix of datapoints.

prototype_labels: A n-dimensional vector containing the labels for each prototype.

w_plus_index: A n-dimensional vector containing the indices for nearest prototypes to datapoints with same label.

a_k: A N x (N*K) dimentional array where N is data points and K are prototypes. This is for every prototype

beta_k(*lam*, *prototype_labels*, *ak*, *one_pk*)

Use to compute Euclidean distance between data points and prototypes.

beta_k = A_k * lambda + 1_{P_k}

lam: A N*K lambda vector where N is the number of data points and K is the number of prototypes

prototype_labels: A n-dimensional vector containing the labels for each

ak: A N x (N*K) dimentional array where N is data points and K are prototypes. This is for every prototype.

one_pk: A m-dimensional vector which has 1 for data point has closest prototype otherwise zero.

result: A K x (N x 1) matrix with K is number of prototypes, N is number of datapoints.

cost_function(*lam*, *C*, *gamma*, *ak*, *D*, *one_pk*, *pk*, *prototype_labels*, *kappa*)
Calculate cost function of LMLVQ.

cost function: 0.5 * lambda^T * H * lambda - q^T * lambda where H = C*I - sum_k (A_k * (D/P_k) * A_k) and q = gamma * 1.T + sum_k (1_{P_k} * (D/P_k) * A_k)

lam: A N*K lambda vector where N is the number of data points and K is the number of prototypes

C: The regularization constant.

gamma: The margin parameter.

ak: A N x (N*K) dimentional array where N is data points and K are prototypes. This is for every prototype.

D: A n x n matrix with Euclidean distance between datapoints.

one_pk: A m-dimensional vector which has 1 for data point has closest prototype otherwise zero.

pk: A list of numbers of datapoints for which w_k is closest prototype.

prototype_labels: A n-dimensional vector containing the labels for each prototype.

kappa: A hyperparameter.

H: A n x n matrix of result C*I - sum_k (A_k * (D/P_k) * A_k)

q: A n-dimensional vector of result: gamma * 1.T + sum_k (1_{P_k} * (D/P_k) * A_k)

cf: A n x n matrix of result of cost function.

d_i_k(*input_data*, *prototype_labels*, *D*, *beta*)
Calculate distance between data points and prototypes.

Formula:

d_{i,k} = sum_j eta_k[j] * D[i, j] / np.sum(beta_k) - 0.5 * np.dot(beta_k, np.dot(D, beta_k)) / (np.sum(beta_k) ** 2)

input_data: A n x m matrix of datapoints.

prototype_labels: A n-dimensional vector containing the labels for each

D: A n x n matrix with Euclidean distance between datapoints.

beta: A K x (N x 1) matrix with K is number of prototypes, N is number of datapoints.

dik: A n x m matrix with distance between datapoints and prototypes.

euclidean_dist(*input_data*, *prototypes*)
Calculate squared Euclidean distance between datapoints and prototypes.

input_data: A n x m matrix of datapoints.

prototpes: A n x m matrix of prototyes of each class.

eu_dist: A n x m matrix with Euclidean distance between datapoints and prototypes.

D: A n x n matrix with Euclidean distance between datapoints.

fit(*input_data*, *data_labels*, *learning_rate*, *epochs*, *margin*, *constant*, *kappa*)
Train the Algorithm.

input_data: A n x m matrix of datapoints.

data_labels: A n-dimensional vector containing the labels for each datapoint.

learning_rate: The step size.

epochs: The maximum number of optimization iterations.

margin: The margin parameter.

Constant: The regularization constant.

kappa: A hyperparameter.

beta: A K x (N x 1) updated beta matrix with K is number of prototypes, N is number of datapoints.

mod_Pk (*input_data*, *data_labels*, *prototype_labels*, *eu_dist*)
Calculate the number of datapoints for which w_k is closest prototype.

input_data: A n x m matrix of datapoints.

data_labels: A n-dimensional vector containing the labels for each datapoint.

prototype_labels: A n-dimensional vector containing the labels for each prototype.

eu_dist: A n x m matrix with Euclidean distance between datapoints and prototypes.

w_plus_index: A n-dimensional vector containing the indices for nearest prototypes to datapoints with same label.

pk: A list of numbers of datapoints for which w_k is closest prototype.

normalization (*input_data*)
Normalize the data between range 0 and 1.

input_value: A n x m matrix of input data.

normalized_data: A n x m matrix with values between 0 and 1.

one_p_k (*input_data*, *prototype_labels*, *w_plus_index*)
A vector which has 1 where data point has closest prototype otherwise zero.

input_data: A n x m matrix of datapoints.

prototype_labels: A n-dimensional vector containing the labels for each prototype.

w_plus_index: A n-dimensional vector containing the indices for nearest prototypes to datapoints with same label.

pk: A m-dimensional vector which has 1 for data point has closest prototype otherwise zero.

predict (*input_value*, *input_data*)
Predicts the labels for the data represented by the given test-to-training distance matrix.

input_value: A n x m matrix of distances from the test to the training datapoints.

input_data: A n x m matrix of datapoints.

ylabel: A n-dimensional vector containing the predicted labels for each datapoint.

prt (*input_data*, *data_labels*, *prototype_per_class*)
Calculate prototypes with labels either at mean or randomly depends on prototypes per class.

input_value: A n x m matrix of datapoints.

data_labels: A n-dimensional vector containing the labels for each datapoint.

prototypes per class: The number of prototypes per class to be learned. If it is equal to 1 then prototypes assigned at mean position else it assigns randolmy.

prototype_labels: A n-dimensional vector containing the labels for each prototype.

prototypes: A n x m matrix of prototyes.for training.

lambda: A n*m array of zeros

prt_labels = array([], dtype=float64)

update(*lam*, *H*, *q*, *learning_rate*)
To update the lambda vector.

gradient = H * lambda - q lambda(t+1) = lambda - learning rate * gradient

lam: A N*K lambda vector where N is the number of data points and K is the number of prototypes

H: A n x n matrix of result C*I - sum_k (A_k * (D/P_k) * A_k)

q: A n-dimensional vector of result: gamma * 1.T + sum_k (1_{P_k} * (D/P_k) * A_k)

learning_rate: The step size.

lam_update: A N*K updated lambda vector where N is the number of data points and K is the number of prototypes.

update_beta = array([], dtype=float64)

PYTHON MODULE INDEX

INDEX