Data Mining
Project Choice 2

Federico Ungarelli A17694736

03/19/2023

1 Linear Regression

Normalizing Features:

Normalizing the input features (X) can help the algorithm converge faster and improve model performance. By normalizing input features, we prevent certain features from dominating the others during the learning process. Normalization is also beneficial when the range of values for the input feature is particularly large since it helps to avoid numerical stability issues during the optimization process. As a matter of fact, we observe that when data are normalized the gradient descent algorithm converges always before 2000 iterations (the exact number varies according to the random initialization of the coefficients); whereas when normalization is not applied the gradient descent algorithm never hits the early stop (considering $self.early\_stop : 1e - 50$).

When it comes to making predictions with a model that was trained on normalized data (both X and y), we need to "denormalize" the results by applying the inverse formula of normalization. This involves reversing the scaling that was applied during the training process, in such a way that the output array is in the same scale as the original data. In our case, this is the formula that is used in the $.predict()$ method:

$$pred = pred \ * (self.max\_target - self.min\_target) + self.min\_target$$

Where $self.max\_target$ and $self.min\_target$ are, respectively, the maximum and minimum value of the target array (before normalization). By doing this, our $Linear\_Regression()$ class predicts the actual values of y.

Looking at the coefficients of the normalized Linear Regression, we can see that θ_0 (coefficient of the intercept term) tends to 0 while θ_1 tends to 1. This is because the intercept term represents the predicted value of the dependent variable when the input feature is equal to 0. However, as a result of the min-max normalization, the input feature values are now scaled within the range [0,1]. Hence, setting the input feature to 0 may not correspond to a meaningful value, and the intercept term may not have a clear interpretation. In addition to that, the intercept term coefficient may tend to zero due to feature scaling that reduces the overall variability of the input features. This reduction in variability can result in a smaller range of values for the input features, which can lead to a smaller range of predicted values for the dependent variable. Thus, the intercept term will have a smaller impact on the dependent variable's prediction.

Different coefficients but similar MSE:

When we run our Linear Regression algorithm on the Wine dataset we observe significantly different coefficients as compared to those output by the sklearn's Linear Regression, even though the loss values are very close to each other.

It is possible to have similar loss values with different coefficients when there is collinearity between independent variables. In this scenario, it is possible to obtain similar loss values with different sets of coefficients, because the redundant information (coming from highly correlated features) can be captured by different combinations of the coefficients. This might be the reason why we get different coefficients when running the two different Linear Regressions.

2 Logistic Regression

Interpretation of Coefficients:

The logistic regression allows to model the characteristics of a binary dependent variable, but, due to its non-linearity, the interpretation of the coefficients is less intuitive than in linear regression. Starting from the original definition of the model, it is possible to re-express it through a different functional form, allowing an easier interpretation for the coefficients:

$$logit(\pi) = log(\frac{\pi}{1-\pi}) = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p$$

Then, the generic coefficient β_k of the k-th independent variable is interpreted as the change in the log-odds for one unit increase in x_k, keeping all other variables constant.

Learning Rate and number of iterations:

For the Gradient Descent algorithm to work properly, we must choose the learning rate wisely. The learning rate α defines how quickly we update the parameters. If the learning rate is too large, we may "overshoot" the optimal value. On the other hand, if it is too small, we will need too many iterations to converge to the best values. That's why it is crucial to use a well-tuned learning rate. For our Logistic Regression class to reach an acceptable level of accuracy (around 0.7), we set α as 0.001 and the number of iterations as 20000. With these values, the algorithm run in a reasonable amount of time as well.

Normalization of y:

A fundamental step in the implementation of the Logistic Regression algorithm is the normalization of the target variable y. As a matter of fact, we might train our model on a dataset where the dependent variable takes values different from 0 and 1 (i.e. in Wine's dataset $y \in \{5, 6\}$). This is why in our $.fit()$ method we implemented a for loop to normalize y such that $y_{norm} \in \{0, 1\}$

3 Naive Bayes

Do we need $P(x)$?

Given Naive Bayes assumption, we get the formula:

$$P(y|x_1, ..., x_d) = \frac{P(y)\prod_{i=1}^{d} P(x_i|y)}{P(x_1, ..., x_n)}$$

Given that our goal is to find the value of y that maximizes the posterior for a given input x, the denominator can be removed, and a proportionality can be introduced such that:

$$P(y|x_1, ..., x_d) \propto P(y)\prod_{i=1}^{d} P(x_i|y)$$

This is because the probability of observing x is the same across all classes. This will not affect the calculation because we do not need to find the actual posterior probability of the classes. Instead, we only need to see which class has the largest posterior probability. Dividing all the posterior probabilities by the same constant would return the same results on whose value is the largest. In any case, these numbers can be converted into a probability by applying a normalization (making the sum equal to 1).

Logarithmic transformation:

When implementing the computation of posteriors, a problem may arise when the feature vector is of high dimension. Since each $P(x_i|y)$ term is between 0 and 1, when we start multiplying them, the overall product starts approaching zero. A possible solution to this issue is to use log-probabilities. That is, instead of using $P(x_1|y)$ and $P(y)$, we use $logP(x_1|y$ and $logP(y)$. This transformation works well both because the logarithm is an increasing function and because the resulting sums ($\sum_{i=1}^{n} log(x_1) = log(\prod_{i=1}^{d} x_i)$) are usually more manageable numbers, than the very tiny probabilities that we get without this transformation. We should keep in mind that since we are taking the logarithm of numbers between 0 and 1, the log-probabilities will always be negative. Hence, a "more negative" number corresponds to a smaller probability than a "less negative" number.

Unseen Values:

To deal with possible unseen feature values we decided to apply "Laplace smoothing". The idea behind Laplace Smoothing is to ensure that our posterior probabilities are never zero and in order to do so we substitute the likelihood ($P(x_i|Y = y)$) of the unseen feature values with the following value:

$$\frac{1}{N + k}$$

where N = number of observations with Y = y; and K = number of features.

4 K-Means and Gaussian Mixture Model

E-M algorithm in K-Means:

We might think of the K-Means algorithm as an EM algorithm, even though this is not the most natural way of visualizing it. From the $train()$ method of our K-Means implementation, we can see that the algorithm is divided into two steps: first updating the Relation Matrix given the current centers and then updating the centers given the Relation Matrix. Mathematically:

- E-step: For each observation j, set $RM_{j,k^*} = 1$ and $RM_{j,k} = 0$ for $k \neq k^*$, where K^* is the index of the closest cluster center:

$$k^* = \arg\min_k \|(x_j - \mu_k)\|^2$$

- M-step: For each cluster k, update the cluster center as the mean of the points within the cluster:

$$\mu_k = \frac{\sum_{j=1}^{J}(r_{j,k}x_j)}{\sum_{j=1}^{J}(r_{j,k})}$$

Through this, we can get a new perspective on what lies behind the K-Means algorithm and, at the same time, we gain a better understanding of the similarities and differences with the Gaussian Mixture Model.

Initialization:

Since the initialization of the centroids is essentially randomized, they can start far away from the "true" cluster centers in the data. Both K-Means and GMM always converge (according to our convergence criterion), but not necessarily to the optimal one as they may get stuck into a local minimum. For this reason, we observe from the plots different clusters every time we run the K-Means algorithm with random initial centroids. To partially tackle this issue, when implementing the GMM we initialize the cluster means (μ_k) as the centroids obtained after running K-Means on the same dataset. Despite this, the cluster means obtained by running the Gaussian Mixture Model (both sklearn's and our implementation) on the same dataset change each time, suggesting that we are not converging to a global optimum.