CS8803: STR, Spring 2017, Lab-2

Authors: Marcus Pereira, Shiyu Feng and Jintasit Pravitra

Date: March 13^{th}, 2017

1. Gradient Descent on Squared Loss

1.1 Algorithm

We define the loss function to be $l_i(t) = ||W_t f_i - y_i||^2$, where $f_i \in \mathbb{R}^{10}$ is a feature vector given by the problem. $W \in \mathbb{R}^{5\times 10}$ is the weight matrix that we seek to learn. $y_i \in \mathbb{R}^5$ is an indicator vector corresponds to each label. We remap the labels to be:

Veg : $y = [\ 1\ 0\ 0\ 0\ 0\ ]^T$ Wire :$y = [\ 0\ 1\ 0\ 0\ 0\ ]^T$ Pole : $y = [\ 0\ 0\ 1\ 0\ 0\ ]^T$ Ground : $y = [\ 0\ 0\ 0\ 1\ 0\ ]^T$ Facade : $y = [\ 0\ 0\ 0\ 0\ 1\ ]^T$

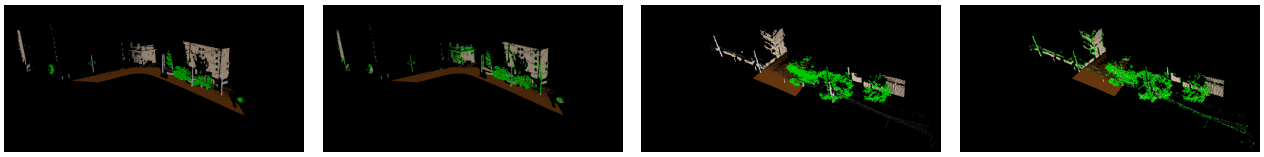
We use the update law:

$$W_{t+1} = W_t - \alpha \frac{dl}{dW}, \quad \text{where } \frac{dl}{dW} = 2(W_t f_i - y_i) f_i$$

We choose the learning rate $\alpha = \frac{1}{\sqrt{T}}$. Our algorithm reshuffles the given data and pass them through the learning algorithm several times.

1.2 Results

We implement the result using C++ and Qt framework. We use OpenGL for visualization. The result of classification after learning both datasets are shown in Fig.1 (a)-(d). Here we show classification after 10 passes of both datasets. However, we discovered that passing datasets twice is already sufficient to obtain a consistent W. There is very little difference in outcome between 2 and 10 passes. Fig. 2 shows another test case. In this case, we learn with Dataset 1 and use W to classify Dataset 2.



(a) Actual Plot for dataset1 (b) Classified Plot for dataset1 (c) Actual Plot for dataset2 (d) Classified Plot for dataset2

Figure 1: Learning Results for Gradient Descent on Squared Loss Fuction

Wires and poles do not get classified very well for many potential reasons. One reason is they are consideraly "thinner features". Poles and wires have very little feature content in transverse direction. Other features are easier to classify because they tend to be clumped together. Size, density, and color information of clumped features are easier to extract. We also note that we have much less samples of poles and wires in the training set. This is especially true in our second test case where we learn using Dataset 1 (which do not have many wires) and attempt classify Dataset 2. Fig. 2 shows that wires are classified very poorly.

As often the case, choice of implemention is a direct tradeoff between implementation difficulty and CPU time. Our choice of C++ makes the learning and classification very fast. However, some implementation such as matrix multiplication is not trivial. Overall the algorithm is not difficult to implement. To evaluate

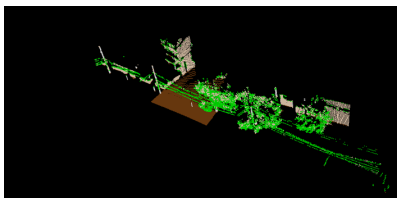


Figure 2: Learning Dataset1, Classified Dataset2

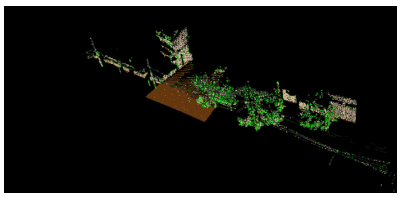


Figure 3: Add noise to features vector

robustness of the learned data, we apply 1σ noise into Dataset 2. The classification result is shown in Fig. 3. Despite many misclassifications, facade and ground are still mostly correctly classified .

2. Linear Support Vector Machine

2.1 Algorithm

An online version of the Linear SVM classifier was implemeted based on the notes from lecture-12. The two classes considered were - Facade (1400) and Veg (1004). The performance for online learning was not as good as compared to Gradient Descent on Squared Loss and BLR. For training and testing, the two original data sets were combined and mixed at random. Also, another set of training and test data was generated from the original combined data sets and corrupted with noise to test the accuracy of the Linear SVM classifier. The linear SVM learner was fairly easy to implement in Python. Based on the dot product $w^T f_i$ and the node_id of the current observed point, the weights vector was updated using one of the following two equations:

$w = w - 2\alpha_t \lambda w$ if $\left(w^T f_i > 0\right)$ and node_id = 1004 (true positive case)
$w = w - 2\alpha_t \lambda w$ if $\left(w^T f_i < 0\right)$ and and node_id = 1400 (true negative case)
$w = w - 2\alpha_t \lambda w + \alpha_t\, y_t\, f_i$ if $\left(w^T f_i < 0\right)$ and node_id = 1004 (false negative case)
$w = w - 2\alpha_t \lambda w + \alpha_t\, y_t\, f_i$ if $\left(w^T f_i > 0\right)$ and node_id = 1400 (false positive case)

2.2 Results

The hyperparameters λ and α_t were chosen as follows:
1) $\lambda = 0.495$ as anything lower than this was causing the classifier to have poor accuracy as weights begin to diverge far away from the initial set of weights. And, anything higher caused the weights to overfit and perform poorly on test data and corrupted noisy data set.
2) As far as $\alpha_{t,}$, as per the notes in lecture-12, we pick it proportional to $1/\sqrt{T}$

It is important to monitor error convergence of the classifier during training. This graph for Linear SVM is shown in Fig. 4. Each 1000 learning steps, we calculate a sum of squared error and plot them as the convergence curve. From the graph, it is obvious that the error is decreasing rapidly.

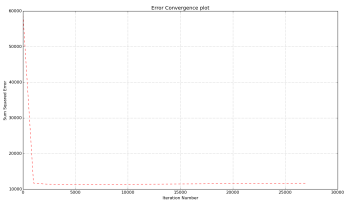


Figure 4: Error Convergence for Linear SVM

After implementing the algorithm above, Fig. 5 [Left] is the complete original 3D point cloud data set and Fig. 5 [Right] is the performance of Linear SVM on combined randomized test data set for Facade and Veg. The confusion matrix elements (TP, TN, FP and FN) are shown in the graph, too. Then the algorithm trains

on the noise corrupted version of features. The result is shown in Fig. 6. From the plot, it hard to see the difference. But based on the calculated accuracy, noise corrupted version will decrease the accuracy from 87% to 82%.

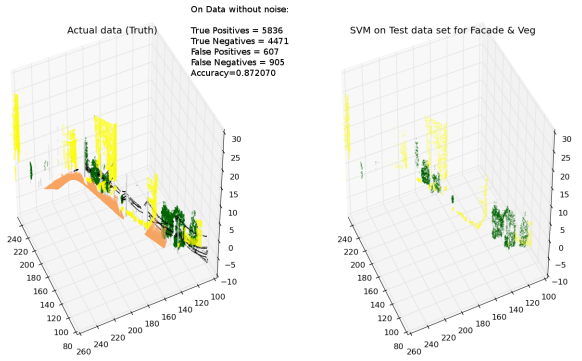


Figure 5: Actual and Classified Point-clouds without Noise

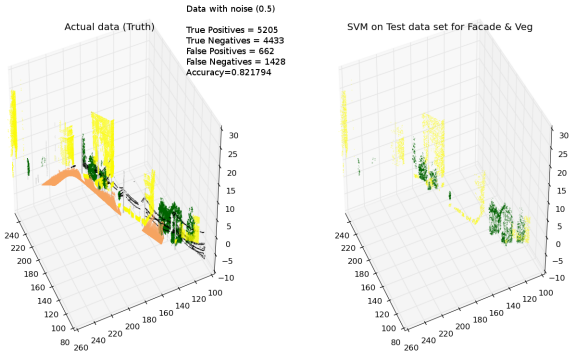


Figure 6: Actual and Classified Point-clouds with Corrupted Noise

3. Bayesian Linear Regression

3.1 Algorithm

Our output model is $y_t = \theta^T \mathbf{x}_t + \epsilon_t$, where θ is the weight vector and $\epsilon_t \sim N(0, \sigma^2)$. In Bayesian Linear Regression (BLK), we maitain a distribution to represent our beliefs about what θ is likely to be when given previous data points D. Using Bayes rule and natural parameterization, we can derive the update rule for BLK. Detailed derivations are in Lecture 13. $P(\theta|D) \propto \exp\{(J + \frac{\sum_i y_i x_i}{\sigma^2})^T\theta - \frac{1}{2}\theta^T(P + \frac{\sum_i x_i x_i^T}{\sigma^2})\theta\}$. Update rules for information vector J and precision matrix P are as follows

$$J \leftarrow J + \frac{y_i x_i}{\sigma^2},\ P \leftarrow P + \frac{x_i x_i^T}{\sigma^2}$$

Then we can use J_{final} and P_{final} to compute mean vector and covariance matrix. $\Sigma_{final} = P_{final}^{-1}$ and $\mu_{final} = \Sigma_{final} J_{final}$. The distribution $P(\theta|D)$ can be represented by two parameters μ_{final} and Σ_{final} of Gaussian. Finally, we use them to do prediction. The mean of y_{t+1} is the output of BLK model.

$$\mu_{y_{t+1}} = \mu_{\theta_{final}}^T \mathbf{x}_{t+1},\ \Sigma_{y_{t+1}} = \mathbf{x}_{t+1}^T \Sigma_{\theta_{final}} \mathbf{x}_{t+1} + \sigma^2$$

There are three parameters we need to define or tune. Mean vector μ_θ and covariance matrix Σ_θ of prior $P(\theta)$, and variance σ^2 of noise. We firstly choose $\sigma^2 = 1$ and μ_θ with a zero vector. Then we assume each element of the weight vector is independent, which causes a diagonal matrix for Σ_θ.

3.2 Result

Our BLK algorithm is implemented in Python. It is easy to define a class BayesianLinearRegression to apply the learner model. We choose two classes Veg (1004) and Facade (1400) to apply BLK. Veg is labeled 1 and Facade is labeled -1. Points of other classes will be ignored. Similarly, we randomly split the mixed dataset 70%/30% into training and test set. In order to moniter the error rate when learning, each 1000 steps, we test current model on test set and obtain sum of squared error. The error convergence curve is shown in Fig. 7. The error also convergences quickly.

Even though monitoring error will slow down the training lagorithm, it is very fast if we only consider the update steps, which is 444ms totally. And test time is also pretty fast, 256ms. After training, we test our model on both training set and test set. The accuracy are 92.82% and 92.69%. Confusion matrixs are as follows.

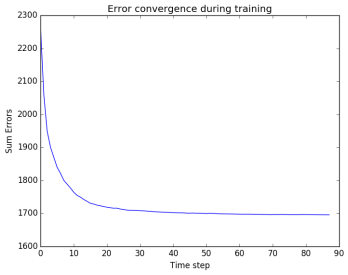
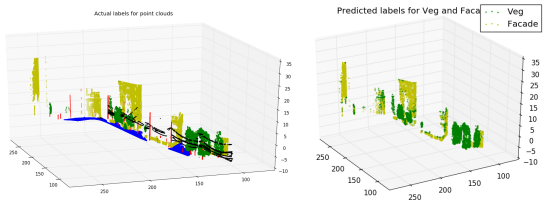


Figure 7: Error Convergence for BLK



(a) Actual Points for Mixed Dataset (b) Predicted Points with BLK

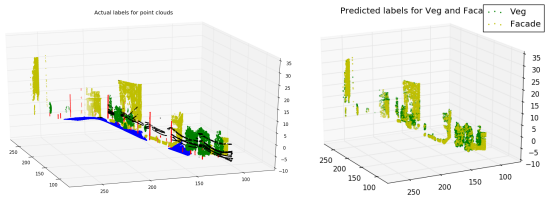
Figure 8: Point-clouds Graph for BLK

Veg	$Facade$		Veg	$Facade$	
14880	836	Veg	6397	344	Veg
1134	10582	$Facade$	520	4588	$Facade$

The left is for training set, right is for test set. In addition, Fig. 8 (a) (b) show the actual and predicted point-clouds graph. Aimed at Veg and Facade, online BLK learner has a good perfomance. Meanwhile, we test our algorithm on other pairs of classes. When we choose groud as one of them, the performance is very good with a 99% accuracy. But when considering wires, the performance becomes worse. The reason could be that features of wires are not significantly different from others. The similarity between wire and other objects is big.

At the end, we introduce noise into features. ϵ_t is only noise for output, not features. Apply BLK model on noise corrupted dataset and repeat same procedure. Results are shown in Fig. 9 and Fig. 10. From error convergence curve, we can observe some fluctuations. And the accuracy is decreasing from 92% to 87%, but most are classified correctly. The confusion matrix for test set is $\begin{matrix} 5804 & 829 \\ 814 & 4281 \end{matrix}$.

Figure 9: Error Convergence for BLK



(a) Actual Points for Mixed Dataset (b) Predicted Points with BLK

Figure 10: Point-clouds Graph for BLK

4. Extra Credits

After finishing the required 3 algorithms, we touch a little about online learning logistic regression. The model is $y_t = sign(\sigma(\theta^T x_t))$, where $\sigma(x) = 1/(1+\exp(-x))$ is a sigmoid function. We can also implement gradient descent on the loss function. The difference between this and above 3 algorithms is the loss function. $l(\theta, x, y) = log(1+\exp\{-y\theta^T x\})$. Similarly, we can implement a gradient descent on the loss function.

$$\theta \leftarrow \theta + yx\frac{1}{1+\exp(y\theta^T x)}$$

We implement this algorithm, the accuracy is 87% and confusion matrix for test set is $\begin{matrix} 5890 & 851 \\ 653 & 4425 \end{matrix}$. The performance is worse than BLK. There is no space to show the graph. But **all useful graphs are included in the submitted folder**.