Machine Learning Spring 2020 Project

Viktor Moros

Extension 1: LDA with Ledoit-Wolf shrinkage

Summary

The wine dataset has 178 wines belonging to 3 classes. Each wine has 13 features.

I used scikit-learn's LDA (Linear Discriminant Analysis) to classify the wines. I then used scikit-learn's extension to LDA called Ledoit-Wolf shrinkage to classify the wines and the accuracy increased. I then made my own implementations of both plain LDA and Ledoit-Wolf LDA and achieved the same accuracy as scikit-learn.

Accuracy

On the wine dataset, both scikit-learn's plain LDA and my implementation of plain LDA yielded 97.2% accuracy. With Ledoit-Wolf LDA, both scikit-learn's implementation and my implementation yielded 100% accuracy.

I also tested on the iris dataset from HW3 but only with a subset of the features because otherwise, both classifiers yielded perfect accuracy. On this truncated iris dataset, both my plain LDA and scikit-learn's plain LDA yielded 96.6% accuracy. With Ledoit-Wolf LDA, scikit-learn's implementation yielded 96.6% accuracy and my implementation yielded 93.3% accuracy.

	My LDA	My Ledoit-Wolf LDA	sklearn's LDA	sklearn's Ledoit-Wolf LDA
Wine	97.2%	100%	97.2%	100%
Iris	96.6%	93.3%	96.6%	96.6%

Explanation of Ledoit-Wolf shrinkage

In typical datasets, we have many more examples than features. Consider the salmon dataset from HW1 or the housing dataset from HW4, for example. We can have many thousands of examples with only a handful of features.

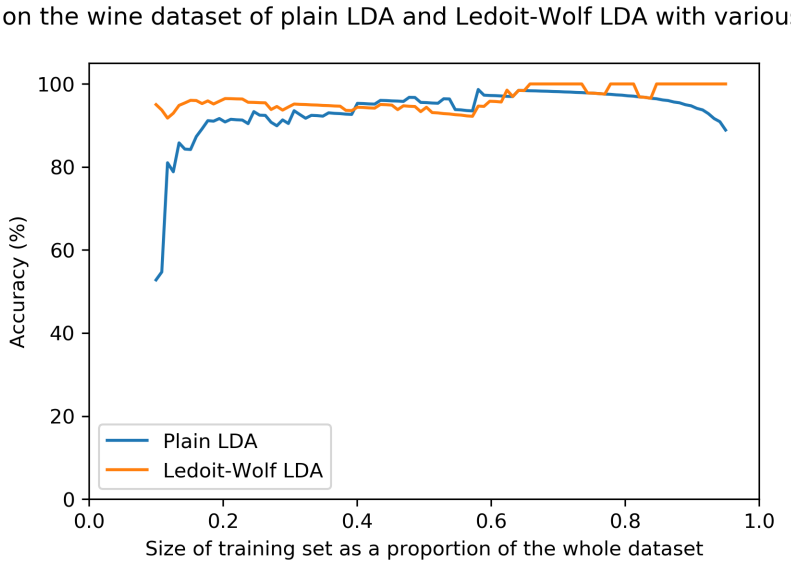
When that's not the case, i.e. when the number of examples is not much bigger than the number of features, the sample covariance is a poor estimator of the true covariance. That's because when the number of examples is not much bigger than the number of features, the sample covariance matrix is invertible but ill-conditioned, so inverting it dramatically amplifies estimation errors.[1]

[1] Olivier Ledoit, Michael Wolf, et al. "A well-conditioned estimator for large-dimensional covariance matrices". In: *Journal of Multivariate Analysis* 88.2 (2004), pp. 365–411.

The idea of covariance shrinkage in general is to fix this ill-conditioning problem by replacing our sample covariance S with the "shrunk covariance" $S^* = \alpha S + (1-\alpha)I_d$, where α is a "shrinkage factor". Ledoit-Wolf shrinkage gives a particular value for α (specifically $\alpha = \frac{1}{n^2}\frac{||X^T X - S||^2}{||S - \frac{Tr(S)}{d} I_d||^2}$) which is optimal under certain assumptions.

To put it informally, the smaller our sample size is relative to our number of features, the more the sample covariance tends to "overfit", so we use a linear combination of our sample covariance and the identity matrix. Conceptually, this is similar to using a low degree polynomial when we have a small sample size so that we don't overfit.

Therefore, the smaller our sample size is, the bigger is the advantage of Ledoit-Wolf LDA over plain LDA, as illustrated in the following graph:



We see that Ledoit-Wolf LDA performs better than plain LDA for almost all sample sizes, and it especially shines when we have small sample sizes, for the reasons explained above. When we train on only one tenth of our data and test on the rest of the data, the accuracy of Ledoit-Wolf LDA is dozens of percentage points higher than that of plain LDA.

Given the reasoning above, Ledoit-Wolf LDA is a good choice for the wine dataset because our number of examples (178) is not much bigger than our number of features (13). And indeed, when we train on 80% of the data and test on the rest, we see that the accuracy of Ledoit-Wolf LDA is higher than that of plain LDA (100% vs 97.2%).

Code

Here is the part of my code that computes the Ledoit-Wolf shrunk covariance. I made more changes than are shown here, but this is the most important part. The comments indicate which parts of the original paper[2] each line of code came from.

```
def ledoit_wolf(X):
    n, d = X.shape
    cov = np.cov(X, rowvar=False, bias=1) # the paper calls this Sn, defined at
                                          # the beginning of section 3.2 on page 377
    mn = np.trace(cov) / d # lemma 3.2 on page 379, the brackets are defined on page 368
    dn = np.sum(np.square(cov - mn)) # lemma 3.3 on page 379
    b_bar = (1 / (n ** 2)) * np.sum(np.square(X.T @ X / n - cov)) # top of page 380
    bn = min(b_bar, dn) # top of page 380
    return (bn / dn) * mn + ((dn - bn) / dn) * cov # lemma 3.5 on page 380
```

[2] Ledoit, Wolf, et al., "A well-conditioned estimator for large-dimensional covariance matrices".

Extension 2: logistic regression with L1 regularization

Summary

The dogs dataset has images of 20,580 dogs belonging to 120 breeds. To make the classification task more manageable, I decided to create two classes. One class contained the breeds Chihuahua and Japanese Spaniel. The other class contained the breeds Maltese and Pekinese.

I used scikit-learn's Stochastic Gradient Descent classifier with logistic loss to classify the images. I then used the same classifier but with L1 regularization and the F1 score improved. I then made my own implementations of both logistic regression with gradient descent and logistic regression with gradient descent with L1 regularization.

F1 scores

On the dogs dataset, sklearn's Stochastic Gradient Descent classifier with logistic loss yielded an F1 score of 58.7%. The same classifier but with L1 regularization yielded an F1 score of 70.9%.

My implementation of plain logistic regression with gradient descent yielded an F1 score of 67.1%. My implementation of logistic regression with gradient descent with L1 regularization yielded an F1 score of 64.9%.

	My LR	My LR w/ L1	sklearn's LR	sklearn's LR w/ L1
Dogs	67.1%	64.9%	58.7%	70.9%
Breast cancer	96.2%	97.1%	95.5%	97.0%

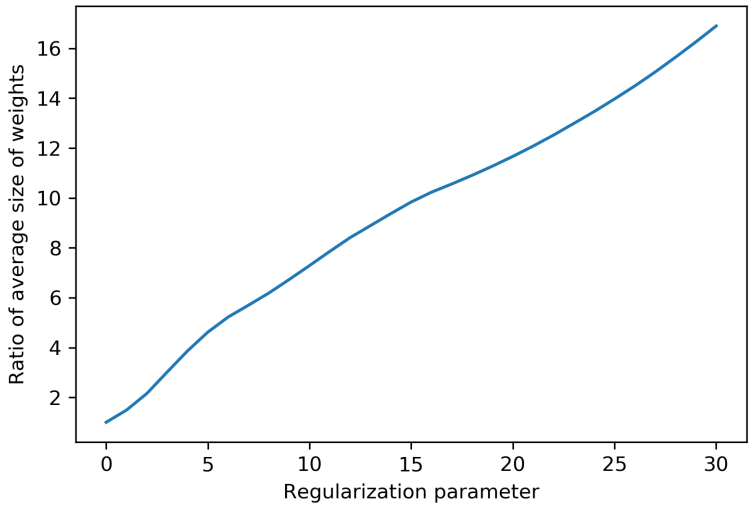
Explanation of logistic regression with L1 regularization

In plain logistic regression, we minimize a log-linear cost function. This helps us achieve good accuracy on our training set but there's nothing preventing our weights from growing. If our weights become enormous, we would suffer from overfitting and from too much variance in the bias-variance tradeoff.

One attempt at solving this problem is called L1 regularizaiton. The idea is that we incorporate into our cost function a penalty for the size of our weights. So our cost function becomes $J(w) = \sum_{i=1}^{N} [y^{(i)} \ln(h(x)) + (1 - y^{(i)}) \ln(1 - h(x))] + \lambda \sum_{i=1}^{d} |w_i|$ where λ is our regularization parameter. So by penalizing the size of our weights, we hope to prevent overfitting. Note that our second sum starts at 1 instead of 0 because we don't want to penalize our intercept term.

One way to see the effect of this regularization is to compare the size of the weights we get from plain logistic regression to the size of those we get from logistic regression with L1 regularization. We expect to see that the bigger λ is, the bigger is the disparity between the sizes of the weights, and that is what we see. Interestingly, the relationship is almost perfectly linear.

The effect of the regularization parameter on the size of our weights



Notes about my implementation

sklearn has a classifier named LogisticRegression() but it uses a technique called LBFGS instead of gradient descent. So I decided to use their SGDClassifier() instead because it's more similar to my approach.

I noticed that my results were quite different from those of sklearn, so I read about their implementation to understand the differences. One difference is that I used batch gradient descent while they used stochastic gradient descent. Another difference is that they use the truncated/cumulative penalty approach from this paper:[3] I used the simpler "clipping" approach described on page 479 of that paper.

[3] Yoshimasa Tsuruoka, Jun'ichi Tsujii, and Sophia Ananiadou. "Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty". In: *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*. Association for Computational Linguistics. 2009, pp. 477–485.

Code

Here is the part of my code that computes the L1 regularization. I made more changes than are shown here, but this is the most important part. The page numbers in the comments refer to this paper:[4]

```
def L1_grad(w):
    grad = np.sign(w)
    grad[0][0] = 0     # don't penalize the intercept term
    return grad

def Logistic_Regression_Gradient_Ascent(X, y, learning_rate, reg_rate, num_iters, reg):
    log_likelihood_values = []
    w = np.zeros((X.shape[1], 1))
    N = X.shape[0]
    y = y.reshape(-1,1)

    for i in range(num_iters):
        h = hypothesis(X, w)
        w += (learning_rate / N) * (X.T @ (y - h))
        if reg:
            new_w = w - (learning_rate * reg_rate / N) * L1_grad(w)
            new_w[np.sign(new_w) != np.sign(w)] = 0 # clipped L1 from page 479 of the paper
            w = new_w

    return w
```

[4] Tsuruoka, Tsujii, and Ananiadou, "Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty".

Extension 3: Neural networks with Nesterov momentum

Summary

In HW8, we trained a neural network to classify digits from the MNIST digits dataset. In this extension, I added a technique called Nesterov momentum to our neural network and the accuracy improved.

Accuracy

	My plain NN	My NN with mom.	sklearn's plain NN	sklearn's NN with mom.
Digits	90.0%	97.5%	91.4%	94.4%
Spam	93.6%	98.3%	87.3%	97.9%

For more information about how these neural networks were trained, including their hyperparameters, see the included code.

Explanation of Nesterov momentum

As we've discussed in class, the loss function of a neural network is non-convex and therefore has many local minima. The gradient at a local minimum is 0 so gradient descent can get stuck there, which is a problem because we'd prefer to reach the global minimum. With a plain neural network, one technique for avoiding getting stuck is tuning the learning rate.

Nesterov momentum is a more sophisticated and effective technique for avoiding getting stuck in local minima. The idea is inspired by the metaphor of gradient descent as the process of rolling a ball down a mountain. Plain gradient descent is not truly analogous to rolling a ball down a mountain because at each step, we only consider the local gradient, so the ball does not build up speed, so to speak.

With Nesterov momentum, we allow the ball to build up speed by considering not only the gradient of the current location of the ball, but also the history of gradients that the ball has experienced. With this method, even if the ball goes into a local minimum, it will have built up some speed and can roll out of that local minimum and move on.

To put it more mathematically, we use an exponential moving average of our weights. So our weight update becomes:

$$v_t = \mu \cdot v_{t-1} - \alpha \cdot \nabla_w$$
$$w = w - \mu \cdot v_{t-1} + (1 + \mu) \cdot v_t$$

where v_t is the velocity at iteration t, μ is a momentum parameter, α is our training rate, and ∇_w is the gradient of our loss function with respect to our weights.[5]

In practice, Nesterov momentum very often increases the rate of convergence of the weights of a neural network, which is consistent with our accuracy results.

[5] *Stanford CS231n.* URL: https://cs231n.github.io/neural-networks-3/.

Code

Here is the part of my code that computes the Nesterov momentum. I made more changes than are shown here, but this is the most important part. Some parts of my approach came from this page:[6]

```
    for l in range(len(nn_structure) - 1, 0, -1):
        grad_w = 1.0/N * tri_W[l] + lamb * W[l]
        grad_b = 1.0/N * tri_b[l]
        if momentum:
            prev_vel_w[l] = vel_w[l]
            vel_w[l] = 0.9 * vel_w[l] - alpha * grad_w
            W[l] += -0.9 * prev_vel_w[l] + 1.9 * vel_w[l]

            prev_vel_b[l] = vel_b[l]
            vel_b[l] = 0.9 * vel_b[l] - alpha * grad_b
            b[l] += -0.9 * prev_vel_b[l] + 1.9 * vel_b[l]
```

[6] *Stanford CS231n*.

References

Ledoit, Olivier, Michael Wolf, et al. "A well-conditioned estimator for large-dimensional covariance matrices". In: *Journal of Multivariate Analysis* 88.2 (2004), pp. 365–411.

Stanford CS231n. URL: https://cs231n.github.io/neural-networks-3/.

Tsuruoka, Yoshimasa, Jun'ichi Tsujii, and Sophia Ananiadou. "Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty". In: *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*. Association for Computational Linguistics. 2009, pp. 477–485.