MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.036—Introduction to Machine Learning
Spring 2016

Project 1 (Solutions). Issued: Fri., 2/2 Due: Fri., 2/26 at 9am

1. Implementing classifiers

2. The source code for `hinge_loss`:

```
def hinge_loss(feature_matrix, labels, theta, theta_0):
    loss = 0.0
    for i in range(len(feature_matrix)):
        x = feature_matrix[i]
        y = labels[i]
        predicted = np.dot(theta, x) + theta_0
        if y*predicted <=1:
            loss += 1-y*predicted
    return loss/len(feature_matrix)
```

3. The source code for `perceptron_single_step_update`:

```
def perceptron_single_step_update(feature_vector, label, current_theta,
    current_theta_0):
    if label*(np.dot(current_theta, feature_vector)+current_theta_0) <= 0:
        return (current_theta + label*feature_vector, current_theta_0 + label)
    return (current_theta, current_theta_0)
```

4. The source code for `perceptron`:

```
def perceptron(feature_matrix, labels, T):
    (nsamples, nfeatures) = feature_matrix.shape
    theta = np.zeros(nfeatures)
    theta_0 = 0.0
    for t in range(T):
        for i in range(nsamples):
            theta, theta_0 = perceptron_single_step_update(feature_matrix[i],
                labels[i], theta, theta_0)
    return (theta, theta_0)
```

5. The source code for `passive_aggressive_single_step_update`:

```
def passive_aggressive_single_step_update(feature_vector, label, L, current_theta
    , current_theta_0):
    loss = hinge_loss([feature_vector],[label],current_theta, current_theta_0)
    eta = min(loss/(np.linalg.norm(feature_vector)**2+1), 1.0/L)
    return (current_theta + eta*label*feature_vector, current_theta_0+eta*label)
```

6. The source code for `average_perceptron` and `average_passive_aggressive`:

```
def average_perceptron(feature_matrix, labels, T):
    (nsamples, nfeatures) = feature_matrix.shape
    theta = np.zeros(nfeatures)
    theta_sum = np.zeros(nfeatures)
    theta_0 = 0.0
```

```
    theta_0_sum = 0.0
    for t in range(T):
        for i in range(nsamples):
            theta, theta_0 = perceptron_single_step_update(feature_matrix[i],
                labels[i], theta, theta_0)
            theta_sum += theta
            theta_0_sum += theta_0
    return (theta_sum/(nsamples*T), theta_0_sum/(nsamples*T))
```

```
def average_passive_aggressive(feature_matrix, labels, T, L):
  (nsamples, nfeatures) = feature_matrix.shape
    theta = np.zeros(nfeatures)
    theta_sum = np.zeros(nfeatures)
    theta_0 = 0.0
    theta_0_sum = 0.0
    for t in range(T):
        for i in range(nsamples):
            theta, theta_0 = passive_aggressive_single_step_update(feature_matrix
    [i],
                labels[i], L, theta, theta_0)
            theta_sum += theta
            theta_0_sum += theta_0
    return (theta_sum/(nsamples*T), theta_0_sum/(nsamples*T))
```

7. The plots for each of the 3 algorithms are shown below in Figure 1. The standard perceptron algorithm performs the worst on our toy dataset while averaged passive-aggressive performs the best. Averaging is particularly helpful for non-separable datasets since it prevents the online algorithms from being sensitive to orderings and values of T chosen. Averaged passive-aggressive performs the best since the optimal decision boundary can be found while stepping towards minimizing hinge loss while minimizing zero-one loss is more erratic on the non-separable toy dataset.

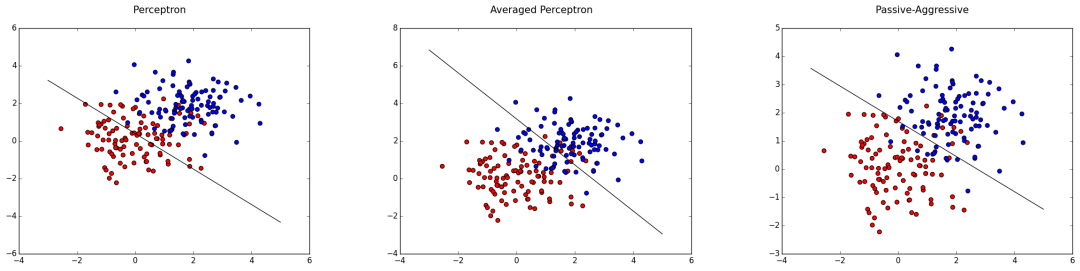

Figure 1: The plots of the decision boundary on the toy dataset for $T = 5$ and $\lambda = 1$.

2. The automatic review analyzer

8. The source code for `classify`:

```
def classify(feature_matrix, theta, theta_0):
    (nsamples, nfeatures) = feature_matrix.shape
    predictions = np.zeros(nsamples)

    for i in range(nsamples):
        feature_vector = feature_matrix[i]
        prediction = np.dot(theta, feature_vector) + theta_0
        if (prediction > 0):
            predictions[i] = 1
        else:
            predictions[i] = -1
    return predictions
```

9-a The source code for `perceptron_accuracy`, `average_perceptron_accuracy` and `average_passive_aggressive_accuracy`:

```
def perceptron_accuracy(train_feature_matrix, val_feature_matrix, train_labels,
    val_labels, T):
    theta, theta_0 = perceptron(train_feature_matrix, train_labels, T)

    train_predictions = classify(train_feature_matrix, theta, theta_0)
    val_predictions = classify(val_feature_matrix, theta, theta_0)

    train_accuracy = accuracy(train_predictions, train_labels)
    validation_accuracy = accuracy(val_predictions, val_labels)

    return (train_accuracy, validation_accuracy)
```

```
def average_perceptron_accuracy(train_feature_matrix, val_feature_matrix,
    train_labels, val_labels, T):
    theta, theta_0 = average_perceptron(train_feature_matrix, train_labels, T)

    train_predictions = classify(train_feature_matrix, theta, theta_0)
    val_predictions = classify(val_feature_matrix, theta, theta_0)

    train_accuracy = accuracy(train_predictions, train_labels)
    val_accuracy = accuracy(val_predictions, val_labels)

    return (train_accuracy, val_accuracy)
```

```
def average_passive_aggressive_accuracy(train_feature_matrix, val_feature_matrix,
     train_labels, val_labels, T, L):
    theta, theta_0 = average_passive_aggressive(train_feature_matrix,
    train_labels, T, L)

    train_predictions = classify(train_feature_matrix, theta, theta_0)
    val_predictions = classify(val_feature_matrix, theta, theta_0)

    train_accuracy = accuracy(train_predictions, train_labels)
    val_accuracy = accuracy(val_predictions, val_labels)

    return (train_accuracy, val_accuracy)
```

9-b. The training and validation accuracies are reported in below.

Method	Training Score	Validation Score
Perceptron	96.35%	80.57%
Avg Perceptron	97.35%	83.43%
Avg PA	98.05%	83.14%

Table 1: Training and validation accuracies for $\lambda = 1$ and $T = 5$

10-a. The training and validation accuracies both improve across the training algorithms as a function of T. However, the training accuracy strictly decreases as a function of lambda; whereas, the validation accuracy improves before decreasing. This happens because lambda stops the algorithm from over fitting. While over fitting is bad in general and hurts the validation accuracy, it does improve the training accuracy. When lambda gets too large however, the algorithm under fits the training data which leads to decreased validation accuracy.

Below are the following accuracy graphs on the training and validation sets while varying T for both perceptron and averaged perceptron:

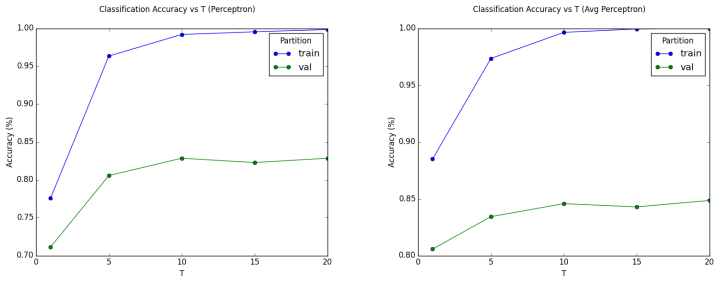

Figure 2: The perceptron(left) and average perceptron(right) accuracy graphs while varying T.

Below are the following accuracy graphs on the training and validation sets while varying T and L for averaged passive aggressive:

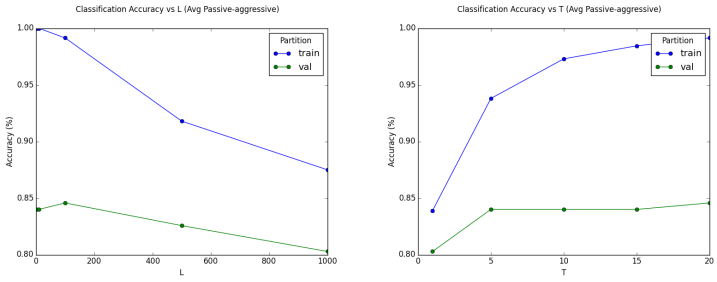

Figure 3: The accuracy graphs for the average passive aggressive algorithm while varying L (left) and T (right).

10-b. The algorithm that performs the best using the given T and L values is averaged perceptron. Note that this could change depending on the T and L values attempted.

10-c. The optimal value is $T = 20$ giving a validation accuracy of 84.85%

11-a. The accuracy we obtained using averaged perceptron with $T = 20$ is 86.29%

11-b. Below are the top 10 unigrams that were most impactful:

- delicious
- great
- smooth
- best
- amazing
- wonderful
- brands
- yummy
- here
- hooked

3. New Features and the challenge

12. Open ended question. Adding any of the suggested additional features without binarizing tends to hurt performance. Removing stopwords and limiting the word feature set to length $k > 3$ improves the accuracy approximately by 1%. Adding bigram bag-of-words features increases the test accuracy for 1 to 1.5% when run on average passive-aggressive algorithm, but only by a couple of decimal points on others.