1. Introduction

Bookmark this page

The goal of this project is to design a classifier to use for sentiment analysis of product reviews. Our training set consists of reviews written by Amazon customers for various food products. The reviews, originally given on a 5 point scale, have been adjusted to a +1 or -1 scale, representing a positive or negative review, respectively.

Below are two example entries from our dataset. Each entry consists of the review and its label. The two reviews were written by different customers describing their experience with a sugar-free candy.

Review	label
Nasty No flavor. The candy is just red, No flavor. Just plan and chewy. I would never buy them again	-1
YUMMY! You would never guess that they're sugar-free and it'□s so great that you can eat them pretty much guilt free! i was so impressed that i've ordered some for myself (w dark chocolate) to take to the office. These are just EXCELLENT!	1

In order to automatically analyze reviews, you will need to complete the following tasks:

1. Implement and compare three types of linear classifiers: the **perceptron** algorithm, the **average perceptron** algorithm, and the **Pegasos** algorithm.

2. Use your classifiers on the food review dataset, using some simple text features.

3. Experiment with additional features and explore their impact on classifier performance.

Setup Details:

For this project and throughout the course we will be using Python 3.6 with some additional libraries. We strongly recommend that you take note of how the NumPy numerical library is used in the code provided, and read through the on-line NumPy tutorial. **NumPy arrays are much more efficient than Python's native arrays when doing numerical computation. In addition, using NumPy will substantially reduce the lines of code you will need to write.**

1. *Note on software: For this project, you will need the **NumPy** numerical toolbox, and the **matplotlib** plotting toolbox.*

2. Download sentiment_analysis.tar.gz and untar it into a working directory. The `sentiment_analysis` folder contains the various data files in **.tsv** format, along with the following python files:

   - **project1.py** contains various useful functions and function templates that you will use to implement your learning algorithms.
   - **main.py** is a script skeleton where these functions are called and you can run your experiments.
   - **utils.py** contains utility functions that the staff has implemented for you.
   - **test.py** is a script which runs tests on a few of the methods you will implement. Note that these tests are provided to help you debug your implementation and are not necessarily representative of the tests used for online grading. Feel free to add more test cases locally to further validate the correctness of your code before submitting to the online graders in the codeboxes.

Tip: Throughout the whole online grading system, you can assume the NumPy python library is already imported as `np`. In some problems you will also have access to python's `random` library, and other functions you've already implemented.

This project will unfold both on MITx and on your local machine. You are welcome to implement functions locally and run **test.py** to validate basic functionality, and then copy+paste your code into the MITx code boxes to fully check correctness and receive your grade for individual function implementations. Alternatively, you can also implement the functions online first and after finishing, copy+paste the solution to your local **project1.py** file. Be wary of the number of attempts you have for each problem, especially if you choose the second development flow.

How to Test Locally: In your terminal, navigate to the directory where your project files reside. Execute the command `python test.py` to run all the available tests.

How to Run your Project 1 Functions Locally: In your terminal, enter `python main.py`. You will need to uncomment/comment the relevant code as you progress through the project.

2. Hinge Loss

Bookmark this page

Project due Oct 6, 2020 19:59 EDT **Past Due**

In this project you will be implementing linear classifiers beginning with the Perceptron algorithm. You will begin by writing your loss function, a hinge-loss function. For this function you are given the parameters of your model θ and θ_0. Additionally, you are given a feature matrix in which the rows are feature vectors and the columns are individual features, and a vector of labels representing the actual sentiment of the corresponding feature vector.

Hinge Loss on One Data Sample

1 point possible (graded)

First, implement the basic hinge loss calculation on a single data-point. Instead of the entire feature matrix, you are given one row, representing the feature vector of a single data sample, and its label of +1 or -1 representing the ground truth sentiment of the data sample.

Reminder: You can implement this function locally first, and run `python test.py` in your `sentiment_analysis` directory to validate basic functionality before checking against the online grader here.

Available Functions: You have access to the NumPy python library as np; No need to import anything.

```
def hinge_loss_single(feature_vector, label, theta, theta_0):
    """
    Finds the hinge loss on a single data point given specific classification
    parameters.

    Args:
        feature_vector - A numpy array describing the given data point.
        label - A real valued number, the correct classification of the data
            point.
        theta - A numpy array describing the linear classifier.
        theta_0 - A real valued number representing the offset parameter.


    Returns: A real number representing the hinge loss associated with the
    given data point and parameters.
    """
    y = np.dot(theta, feature_vector) + theta_0
    loss = max(0.0, 1 - y * label)
    return loss
```

Solution:

See above for expected answer.

Another possible solution is:

```
def hinge_loss_single(feature_vector, label, theta, theta_0):
    y = theta @ feature_vector + theta_0
    return max(0, 1 - y * label)
```

```
def hinge_loss_full(feature_matrix, labels, theta, theta_0):
    """
    Finds the total hinge loss on a set of data given specific classification
    parameters.

    Args:
        feature_matrix - A numpy matrix describing the given data. Each row
            represents a single data point.
        labels - A numpy array where the kth element of the array is the
            correct classification of the kth row of the feature matrix.
        theta - A numpy array describing the linear classifier.
        theta_0 - A real valued number representing the offset parameter.


    Returns: A real number representing the hinge loss associated with the
    given dataset and parameters. This number should be the average hinge
    loss across all of the points in the feature matrix.
    """
    loss = 0
    for i in range(len(feature_matrix)):
        loss += hinge_loss_single(feature_matrix[i], labels[i], theta, theta_0)
    return loss / len(labels)
```

Solution:

See above for expected answer: we simply sum and take the average.

Another possible solution is:

```
def hinge_loss_full(feature_matrix, labels, theta, theta_0):
    ys = feature_matrix @ theta + theta_0
    loss = np.maximum(1 - ys * labels, np.zeros(len(labels)))
    return np.mean(loss)
```

Here, we use the fact that matrix multiplication is equivalent to stacking the result of dot products to from `ys`, an array of outputs. We then use `np.maximum` to take the element-wise maximum of two arrays.

This solution is more efficient because it makes use of NumPy's fast matrix multiplication capability.

Show answer

Submit You have used 0 of 25 attempts

Perceptron Single Step Update

1 point possible (graded)

Now you will implement the single step update for the perceptron algorithm (implemented with $0-1$ loss). You will be given the feature vector as an array of numbers, the current θ and θ_0 parameters, and the correct label of the feature vector. The function should return a tuple in which the first element is the correctly updated value of θ and the second element is the correctly updated value of θ_0.

Available Functions: You have access to the NumPy python library as `np`.

Tip:: Because of numerical instabilities, it is preferable to identify 0 with a small range $[-\varepsilon, \varepsilon]$. That is, when x is a float, "$x = 0$" should be checked with $|x| < \varepsilon$.

```
def perceptron_single_step_update(
        feature_vector,
        label,
        current_theta,
        current_theta_0):
    """
    Properly updates the classification parameter, theta and theta_0, on a
    single step of the perceptron algorithm.

    Args:
        feature_vector - A numpy array describing a single data point.
        label - The correct classification of the feature vector.
        current_theta - The current theta being used by the perceptron
            algorithm before this update.
        current_theta_0 - The current theta_0 being used by the perceptron
            algorithm before this update.
```

Press ESC then TAB or click outside of the code editor to exit

```
def perceptron_single_step_update(
        feature_vector,
        label,
        current_theta,
        current_theta_0):
    """
    Properly updates the classification parameter, theta and theta_0, on a
    single step of the perceptron algorithm.

    Args:
        feature_vector - A numpy array describing a single data point.
        label - The correct classification of the feature vector.
        current_theta - The current theta being used by the perceptron
            algorithm before this update.
        current_theta_0 - The current theta_0 being used by the perceptron
            algorithm before this update.

    Returns: A tuple where the first element is a numpy array with the value of
    theta after the current update has completed and the second element is a
    real valued number with the value of theta_0 after the current updated has
    completed.
    """
    if label * (np.dot(current_theta, feature_vector) + current_theta_0) <= 0:
        current_theta += label * feature_vector
        current_theta_0 += label
    return (current_theta, current_theta_0)
```

Solution:

See above for expected answer.

We need to make sure that if the output of the perceptron is 0, the weights are still updated, hence the check for large inequality. In fact, because of numerical instabilities, it is preferable to identify 0 with a small range $[-\varepsilon, \varepsilon]$.

```
def perceptron_single_step_update(feature_vector, label, current_theta, current_theta_0):
    if label * (np.dot(current_theta, feature_vector) + current_theta_0) <= 1e-7:
        return (current_theta + label * feature_vector, current_theta_0 + label)
    return (current_theta, current_theta_0)
```

Show answer

Submit You have used 0 of 25 attempts

Answers are displayed within the problem

Full Perceptron Algorithm

1 point possible (graded)

In this step you will implement the full perceptron algorithm. You will be given the same feature matrix and labels array as you were given in **The Complete Hinge Loss**. You will also be given T, the maximum number of times that you should iterate through the feature matrix before terminating the algorithm. Initialize θ and θ_0 to zero. This function should return a tuple in which the first element is the final value of θ and the second element is the value of θ_0.

Tip: Call the function `perceptron_single_step_update` directly without coding it again.

Hint: Make sure you initialize `theta` to a 1D array of shape `(n,)` and **not** a 2D array of shape `(1, n)`.

Note: Please call `get_order(feature_matrix.shape[0])`, and use the ordering to iterate the feature matrix in each iteration. The ordering is specified due to grading purpose. In practice, people typically just randomly shuffle indices to do stochastic optimization.

Available Functions: You have access to the NumPy python library as `np` and `perceptron_single_step_update` which you have already implemented.

```
def perceptron(feature_matrix, labels, T):
    """
    Runs the full perceptron algorithm on a given set of data. Runs T
    iterations through the data set, there is no need to worry about
    stopping early.

    NOTE: Please use the previously implemented functions when applicable.
    Do not copy paste code from previous parts.

    NOTE: Iterate the data matrix by the orders returned by get_order(feature_matrix.shape[0])

    Args:
        feature_matrix -  A numpy matrix describing the given data. Each row
            represents a single data point.
        labels - A numpy array where the kth element of the array is the
            correct classification of the kth row of the feature matrix
```

Press ESC then TAB or click outside of the code editor to exit

Unanswered

```
def perceptron(feature_matrix, labels, T):
    """
    Runs the full perceptron algorithm on a given set of data. Runs T
    iterations through the data set, there is no need to worry about
    stopping early.

    NOTE: Please use the previously implemented functions when applicable.
    Do not copy paste code from previous parts.

    Args:
        feature_matrix -  A numpy matrix describing the given data. Each row
            represents a single data point.
        labels - A numpy array where the kth element of the array is the
            correct classification of the kth row of the feature matrix.
        T - An integer indicating how many times the perceptron algorithm
            should iterate through the feature matrix.

    Returns: A tuple where the first element is a numpy array with the value of
    theta, the linear classification parameter, after T iterations through the
    feature matrix and the second element is a real number with the value of
    theta_0, the offset classification parameter, after T iterations through
    the feature matrix.
    """
    (nsamples, nfeatures) = feature_matrix.shape
    theta = np.zeros(nfeatures)
    theta_0 = 0.0
    for t in range(T):
        for i in get_order(nsamples):
            theta, theta_0 = perceptron_single_step_update(
                feature_matrix[i], labels[i], theta, theta_0)
    return (theta, theta_0)
```

Average Perceptron Algorithm

1 point possible (graded)

The average perceptron will add a modification to the original perceptron algorithm: since the basic algorithm continues updating as the algorithm runs, nudging parameters in possibly conflicting directions, it is better to take an average of those parameters as the final answer. Every update of the algorithm is the same as before. The returned parameters θ, however, are an average of the θs across the nT steps:

$$\theta_{final} = \frac{1}{nT}(\theta^{(1)} + \theta^{(2)} + \ldots + \theta^{(nT)})$$

You will now implement the average perceptron algorithm. This function should be constructed similarly to the Full Perceptron Algorithm above, except that it should return the average values of θ and θ_0

Tip: Tracking a moving average through loops is difficult, but tracking a sum through loops is simple.

Note: Please call `get_order(feature_matrix.shape[0])`, and use the ordering to iterate the feature matrix in each iteration. The ordering is specified due to grading purpose. In practice, people typically just randomly shuffle indices to do stochastic optimization.

Available Functions: You have access to the NumPy python library as `np` and `perceptron_single_step_update` which you have already implemented.

```
def average_perceptron(feature_matrix, labels, T):
    """
    Runs the average perceptron algorithm on a given set of data. Runs T
    iterations through the data set, there is no need to worry about
    stopping early.

    NOTE: Please use the previously implemented functions when applicable.
    Do not copy paste code from previous parts.

    NOTE: Iterate the data matrix by the orders returned by get_order(feature_matrix.shape[0])


    Args:
        feature_matrix -  A numpy matrix describing the given data. Each row
            represents a single data point.
        labels - A numpy array where the kth element of the array is the
```

Press ESC then TAB or click outside of the code editor to exit

Unanswered


```
def average_perceptron(feature_matrix, labels, T):
    """
    Runs the average perceptron algorithm on a given set of data. Runs T
    iterations through the data set, there is no need to worry about
    stopping early.

    NOTE: Please use the previously implemented functions when applicable.
    Do not copy paste code from previous parts.

    Args:
        feature_matrix -  A numpy matrix describing the given data. Each row
            represents a single data point.
```

```
    labels - A numpy array where the kth element of the array is the
        correct classification of the kth row of the feature matrix.
    T - An integer indicating how many times the perceptron algorithm
        should iterate through the feature matrix.

Returns: A tuple where the first element is a numpy array with the value of
the average theta, the linear classification parameter, found after T
iterations through the feature matrix and the second element is a real
number with the value of the average theta_0, the offset classification
parameter, found after T iterations through the feature matrix.

Hint: It is difficult to keep a running average; however, it is simple to
find a sum and divide.
"""
(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 get_order(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))
```

Show answer

Submit You have used 0 of 25 attempts

7. Classification and Accuracy

Bookmark this page

Project due Oct 6, 2020 19:59 EDT **Past Due**

Now we need a way to actually use our model to classify the data points. In this section, you will implement a way to classify the data points using your model parameters, and then measure the accuracy of your model.

Classification

1 point possible (graded)

Implement a classification function that uses θ and θ_0 to classify a set of data points. You are given the feature matrix, θ, and θ_0 as defined in previous sections. This function should return a numpy array of -1s and 1s. If a prediction is **greater than** zero, it should be considered a positive classification.

Available Functions: You have access to the NumPy python library as `np`.

Tip:: As in previous exercises, when x is a float, "$x = 0$" should be checked with $|x| < \epsilon$.

```
def classify(feature_matrix, theta, theta_0):
    """
    A classification function that uses theta and theta_0 to classify a set of
    data points.

    Args:
        feature_matrix - A numpy matrix describing the given data. Each row
            represents a single data point.
```

```
            theta - A numpy array describing the linear classifier.
        theta - A numpy array describing the linear classifier.
        theta_0 - A real valued number representing the offset parameter.

    Returns: A numpy array of 1s and -1s where the kth element of the array is
    the predicted classification of the kth row of the feature matrix using the
    given theta and theta_0. If a prediction is GREATER THAN zero, it should
    be considered a positive classification.
    """
    # Your code here
    raise NotImplementedError

```

Press ESC then TAB or click outside of the code editor to exit

Unanswered

```
def classify(feature_matrix, theta, theta_0):
    """
    A classification function that uses theta and theta_0 to classify a set of
    data points.

    Args:
        feature_matrix - A numpy matrix describing the given data. Each row
            represents a single data point.
                theta - A numpy array describing the linear classifier.
        theta - A numpy array describing the linear classifier.
        theta_0 - A real valued number representing the offset parameter.

    Returns: A numpy array of 1s and -1s where the kth element of the array is
    the predicted classification of the kth row of the feature matrix using the
    given theta and theta_0. If a prediction is GREATER THAN zero, it should
```

```
    be considered a positive classification.
    """
    (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
```

Solution:

See above for expected answer.

Another possible solution is:

```
def classify(feature_matrix, theta, theta_0):
    return (feature_matrix @ theta + theta_0 > 1e-7) * 2.0 - 1
```

Here, we use the fact that a boolean will be implicitly casted by NumPy into 0 or 1 when mutiplied by a float.

Again, note that we identified 0 to the range $[-\varepsilon, +\varepsilon]$ for numerical reasons.

Show answer

Accuracy

1 point possible (graded)

We have supplied you with an `accuracy` function:

```
def accuracy(preds, targets):
        """
        Given length-N vectors containing predicted and target labels,
        returns the percentage and number of correct predictions.
        """
        return (preds == targets).mean()
```

The `accuracy` function takes a numpy array of predicted labels and a numpy array of actual labels and returns the prediction accuracy. You should use this function along with the functions that you have implemented thus far in order to implement `classifier_accuracy`.

The `classifier_accuracy` function should take 6 arguments:

- a classifier function that, itself, takes arguments `(feature_matrix, labels, **kwargs)`
- the training feature matrix
- the validation feature matrix
- the training labels
- the valiation labels

- a `**kwargs` argument to be passed to the classifier function

This function should train the given classifier using the training data and then compute compute the classification accuracy on both the train and validation data. The return values should be a tuple where the first value is the training accuracy and the second value is the validation accuracy.

Implement classifier accuracy in the coding box below:

Available Functions: You have access to the NumPy python library as `np`, to `classify` which you have already implemented and to `accuracy` which we defined above.

```
def classifier_accuracy(
        classifier,
        train_feature_matrix,
        val_feature_matrix,
        train_labels,
        val_labels,
        **kwargs):
    """
    Trains a linear classifier and computes accuracy.
    The classifier is trained on the train data. The classifier's
    accuracy on the train and validation data is then returned.

    Args:
        classifier - A classifier function that takes arguments
            (feature matrix, labels, **kwargs) and returns (theta, theta_0)
        train_feature_matrix - A numpy matrix describing the training
```

Press ESC then TAB or click outside of the code editor to exit

Unanswered

```
def classifier_accuracy(
        classifier,
        train_feature_matrix,
        val_feature_matrix,
        train_labels,
        val_labels,
        **kwargs):
    """
    Trains a linear classifier and computes accuracy.
    The classifier is trained on the train data. The classifier's
    accuracy on the train and validation data is then returned.

    Args:
        classifier - A classifier function that takes arguments
            (feature matrix, labels, **kwargs) and returns (theta, theta_0)
        train_feature_matrix - A numpy matrix describing the training
            data. Each row represents a single data point.
        val_feature_matrix - A numpy matrix describing the training
            data. Each row represents a single data point.
        train_labels - A numpy array where the kth element of the array
            is the correct classification of the kth row of the training
            feature matrix.
        val_labels - A numpy array where the kth element of the array
            is the correct classification of the kth row of the validation
            feature matrix.
        **kwargs - Additional named arguments to pass to the classifier
            (e.g. T or L)

    Returns: A tuple in which the first element is the (scalar) accuracy of the
    trained classifier on the training data and the second element is the
    accuracy of the trained classifier on the validation data.
    """
```

```
    theta, theta_0 = classifier(train_feature_matrix, train_labels, **kwargs)
    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)
```

Solution:

See above for expected answer.

In this code, `**kwargs` stands for keyword-arguments. If you are not familiary with the `**` syntax, you can take a look at this tutorial.

Show answer

Submit You have used 0 of 25 attempts

Answers are displayed within the problem

Baseline Accuracy

3 points possible (graded)

Now, uncomment the relevant lines in **main.py** and report the training and validation accuracies of each algorithm with T = 10 and λ = 0.01 (the λ value only applies to Pegasos).

Please enter the **validation accuracy** of your Perceptron algorithm.

Answer: 0.7160

Please enter the **validation accuracy** of your Average Perceptron algorithm.

Answer: 0.7980

Please enter the **validation accuracy** of your Pegasos algorithm.

Answer: 0.7900

Solution:

- The Perceptron validation accuracy should be 0.7160
- The Average Perceptron validation accuracy should be 0.7980
- The Pegasos validation accuracy should be 0.7900

Show answer

8. Parameter Tuning

Bookmark this page

Project due Oct 6, 2020 19:59 EDT **Past Due**

You finally have your algorithms up and running, and a way to measure performance! But, it's still unclear what values the hyperparameters like T and λ should have. In this section, you'll tune these hyperparameters to maximize the performance of each model.

One way to tune your hyperparameters for any given Machine Learning algorithm is to perform a grid search over all the possible combinations of values. If your hyperparameters can be any real number, you will need to limit the search to some finite set of possible values for each hyperparameter. For efficiency reasons, often you might want to tune one individual parameter, keeping all others constant, and then move onto the next one; Compared to a full grid search there are many fewer possible combinations to check, and this is what you'll be doing for the questions below.

In **main.py** uncomment Problem 8 to run the staff-provided tuning algorithm from **utils.py**. For the purposes of this assignment, please try the following values for T: [1, 5, 10, 15, 25, 50] and the following values for λ [0.001, 0.01, 0.1, 1, 10]. For pegasos algorithm, first fix $\lambda = 0.01$ to tune T, and then use the best T to tune λ

Performance After Tuning

7 points possible (graded)

After tuning, please enter the best T value for each of the perceptron and average percepton algorithms, and both the best T

and λ for the Pegasos algorithm.

Note: Just enter the values printed in your main.py. Note that for the Pegasos algorithm, the result does not reflect the best combination of T and λ.

For the **perceptron** algorithm:

$T =$ **Answer:** 25

With validation accuracy =

Answer: 0.7940

For the **average perceptron** algorithm:

$T =$ **Answer:** 25

With validation accuracy =

Answer: 0.8000

For the **pegasos** algorithm:

$T =$ **Answer:** 25

$\lambda =$ **Answer:** 0.01

With validation accuracy =

Answer: 0.8060

Solution:

- The best Perceptron T should be 25 with validation accuracy = 0.7940
- The best Average Perceptron T should be 25 with validation accuracy = 0.8000
- The best Pegasos T should be 25 with validation accuracy = 0.8060
- The best Pegasos λ should be 0.01 with validation accuracy = 0.8060
- You may notice that all the algorithms result in similar performance. It is due to the fact that they are all linear models.
- In project 2, you will experience how to properly choose model family can lead to betterperformance.

Show answer

Submit You have used 0 of 20 attempts

Accuracy on the test set

1 point possible (graded)

After you have chosen your best method (perceptron, average perceptron or Pegasos) and parameters, use this classifier to compute testing accuracy on the test set.

We have supplied the feature matrix and labels in `main.py` as `test_bow_features` and `test_labels`.

Note: In practice the validation set is used for tuning hyperparameters while a heldout test set is the final benchmark used to compare disparate models that have already been tuned. You may notice that your results using a validation set don't always align with those of the test set, and this is to be expected.

Accuracy on the test set :

Answer: 0.8020

Show answer

Submit You have used 0 of 20 attempts

Answers are displayed within the problem

The most explanatory unigrams

According to the largest weights (i.e. individual i values in your vector), you can find out which unigrams were the most impactful ones in predicting **positive** labels. Uncomment the relevant part in `main.py` to call `utils.most_explanatory_word`.

Report the top ten most explanatory word features for positive classification below:

Top 1 : **Answer:** delicious

Top 2: **Answer:** great

Top 3: **Answer:** !

Top 4: **Answer:** best

Top 5: **Answer:** perfect

Top 6: **Answer:** loves

Top 7: **Answer:** wonderful

Top 8: **Answer:** glad

Top 9: **Answer:** love

Top 10: **Answer:** quickly

Also experiment with finding unigrams that were the most impactful in predicting negative labels.

9. Feature Engineering

Bookmark this page

Project due Oct 6, 2020 19:59 EDT **Past Due**

Frequently, the way the data is represented can have a significant impact on the performance of a machine learning method. Try to improve the performance of your best classifier by using different features. In this problem, we will practice two simple variants of the bag of words (BoW) representation.

Remove Stop Words

1 point possible (graded)

Try to implement stop words removal in your feature engineering code. Specifically, load the file **stopwords.txt**, remove the words in the file from your dictionary, and use features constructed from the new dictionary to train your model and make predictions.

Compare your result in the **testing** data on Pegasos algorithm using $T = 25$ and $L = 0.01$ when you remove the words in **stopwords.txt** from your dictionary.

Hint: Instead of replacing the feature matrix with zero columns on stop words, you can modify the `bag_of_words` function to prevent adding stopwords to the dictionary

Accuracy on the test set using the original dictionary: 0.8020

Accuracy on the test set using the dictionary with stop words removed:

Answer: 0.8080

Solution:

- The original dictionary size is 13401, while the dictionary size after stop word removal is 13276.

Show answer

Submit You have used 0 of 20 attempts

Answers are displayed within the problem

Change Binary Features to Counts Features

1 point possible (graded)

Again, use the same learning algorithm and the same feature as the last problem. However, when you compute the feature vector of a word, use its count in each document rather than a binary indicator.

Hint: You are free to modify the `extract_bow_feature_vectors` function to compute counts features.

Accuracy on the test set using the dictionary with stop words removed and counts features:

Answer: 0.7700

Try to compare your result to the last problem, and see the discussion in solution after answering the question.

Solution:

- The performance is 0.7700, which is worse than the previous problem.
- Even if you use the original feature sets (without stop word removal), it will still decrease performance.
- It is possible that giving models more information leads to worse performance, when model overfits irrelevant information.
- From the learning perspective, it is helpful to somehow “regularize" feature representations (e.g., binarizing them).

Show answer

Submit You have used 0 of 20 attempts

Answers are displayed within the problem

Some additional features that you might want to explore are:

- Length of the text
- Occurrence of all-cap words (e.g. “AMAZING", “DON'T BUY THIS")

- Word embeddings

Besides adding new features, you can also change the original unigram feature set. For example,

- Threshold the number of times a word should appear in the dataset before adding them to the dictionary. For example, words that occur less than three times across the train dataset could be considered irrelevant and thus can be removed. This lets you reduce the number of columns that are prone to overfitting.

There are also many other things you could change when training your model. Try anything that can help you understand the sentiment of a review. It's worth looking through the dataset and coming up with some features that may help your model. Remember that not all features will actually help so you should experiment with some simpler ones before trying anything too complicated.
