Introduction to Machine Learning using Python

Project 1

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 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. **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 in to 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 were implemented for you.

- **test.py** is a script which runs tests on a few of the methods you will implement.

  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

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

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.

```
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.
    """
```

 # TASK: - FIND THIS FUCTION IN project1.py AND COMPLETE THE CODE

CHECK IT BY RUNNING test.py

The Complete Hinge Loss

Now it's time to implement the complete hinge loss for a full set of data. Your input will be a full feature matrix this time, and you will have a vector of corresponding labels. The kth row of the feature matrix corresponds to the kth element of the labels vector. This function should return the appropriate loss of the classifier on the given dataset.

```
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.
    """
```

 # TASK: - FIND THIS FUCTION IN project1.py AND COMPLETE THE CODE

CHECK IT BY RUNNING test.py

Perceptron Single Step Update

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.

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.
  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.
  """
    # TASK: FIND THIS FUCTION IN project1.py AND COMPLETE THE CODE
    # CHECK IT BY RUNNING test.py
```

Full Perceptron Algorithm

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 `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.
        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.

"""

 # TASK: FIND THIS FUCTION IN project1.py AND COMPLETE THE CODE

 # CHECK IT BY RUNNING test.py

Average Perceptron Algorithm

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 `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
```

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.

"""

 # TASK: FIND THIS FUCTION IN project1.py AND COMPLETE THE CODE

 # CHECK IT BY RUNNING test.py

4. Pegasos Algorithm

Now you will implement the Pegasos algorithm. For more information, refer to the original paper at original paper.

The following pseudo-code describes the Pegasos update rule.

Pegasos update rule$(x^{(i)}, y^{(i)}, \lambda, \eta, \theta)$:

if $y^{(i)}(\theta \cdot x^{(i)}) \leq 1$ then
    update $\theta = (1 - \eta\lambda)\theta + \eta y^{(i)} x^{(i)}$

else:
    update $\theta = (1 - \eta\lambda)\theta$

The η parameter is a **decaying factor** that will decrease over time. The λ parameter is a regularizing parameter.

In this problem, you will need to adapt this update rule to add a bias term (θ_0) to the hypothesis, but take care not to penalize the magnitude of θ_0.

Pegasos Single Step Update

Next you will implement the single step update for the Pegasos algorithm. This function is very similar to the function that you implemented in **Perceptron Single Step Update**, except that it should utilize the Pegasos parameter update rules instead of those for perceptron. The function will also be passed a λ and η value to use for updates.

```
def pegasos_single_step_update(
    feature_vector,
    label,
    L,
    eta,
    current_theta,
    current_theta_0):
  """
  Properly updates the classification parameter, theta and theta_0, on a
  single step of the Pegasos algorithm

  Args:
```

feature_vector - A numpy array describing a single data point.

label - The correct classification of the feature vector.

L - The lamba value being used to update the parameters.

eta - Learning rate to update parameters.

current_theta - The current theta being used by the Pegasos
algorithm before this update.

current_theta_0 - The current theta_0 being used by the
Pegasos 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.

"""

 # TASK: FIND THIS FUCTION IN project1.py AND COMPLETE THE CODE

 # CHECK IT BY RUNNING test.py

Full Pegasos Algorithm

Finally you will implement the full Pegasos algorithm. You will be given the same feature matrix and labels array as you were given in **Full Perceptron Algorithm**. 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. For each update, set $\eta = \frac{1}{\sqrt{t}}$ where t is a counter for the number of updates performed so far (between 1 and nT inclusive). 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.

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 `pegasos_single_step_update` which you have already implemented.

```
import math

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

    For each update, set learning rate = 1/sqrt(t),
    where t is a counter for the number of updates performed so far (between 1
    and nT inclusive).

    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 algorithm
            should iterate through the feature matrix.
        L - The lamba value being used to update the Pegasos
            algorithm parameters.

    Returns: A tuple where the first element is a numpy array with the value of
    the 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 theta_0, the offset classification
    parameter, found after T iterations through the feature matrix.
    """
```

 # TASK: FIND THIS FUCTION IN project1.py AND COMPLETE THE CODE

 # CHECK IT BY RUNNING test.py

5. Algorithm Discussion

Once you have completed the implementation of the 3 learning algorithms, you should qualitatively verify your implementations. In **main.py** we have included a block of code that you should uncomment. This code loads a 2D dataset from **toy_data.txt**, and trains your models using $T=10, \lambda=0.2$. **main.py** will compute θ and θ_0 for each of the learning algorithms that you have written. Then, it will call **plot_toy_data** to plot the resulting model and boundary.

Plots

In order to verify your plots, please enter the values of θ and θ_0 for all three algorithms. (For example, if $\theta=(1, 0.5)$, then type **1, 0.5** without the brackets. Make sure your answers are correct up to 4 decimal places.)

For the **perceptron** algorithm:

$\theta=$

$\theta_0=$

For the **average perceptron** algorithm:

$\theta=$

$\theta_0=$

For the **Pegasos** algorithm:

$\theta=$

$\theta_0=$

Convergence

Since you have implemented three different learning algorithm for linear classifier, it is interesting to investigate which algorithm would actually converge. Please run it with a larger number of iterations T to see whether the algorithm would visually converge. You may also check whether the parameter in your theta converge in the first decimal place. Achieving convergence in longer decimal requires longer iterations, but the conclusion should be the same.

Which of the following algorithm will converge on this dataset? (Choose all that apply.)

- [ ] perceptron algorithm
- [ ] average perceptron algorithm
- [ ] pegasos algorithm

6. Automative review analyzer

Now that you have verified the correctness of your implementations, you are ready to tackle the main task of this project: building a classifier that labels reviews as positive or negative using text-based features and the linear classifiers that you implemented in the previous section!

The Data

The data consists of several reviews, each of which has been labeled with -1 or $+1$, corresponding to a negative or positive review, respectively. The original data has been split into four files:

- `reviews_train.tsv` (4000 examples)
- `reviews_validation.tsv` (500 examples)
- `reviews_test.tsv` (500 examples)

To get a feel for how the data looks, we suggest first opening the files with a text editor, spreadsheet program, or other scientific software package (like pandas).

Translating reviews to feature vectors

We will convert review texts into feature vectors using a **bag of words** approach. We start by compiling all the words that appear in a training set of reviews into a **dictionary** , thereby producing a list of d unique words.

We can then transform each of the reviews into a feature vector of length d by setting the ith coordinate of the feature vector to 1 if the ith word in the dictionary appears in the review, or 0 otherwise. For instance, consider two simple documents “Mary loves apples" and “Red apples". In this case, the dictionary is the set $\{Mary;loves;apples;red\}$, and the documents are represented as $(1;1;1;0)$ and $(0;0;1;1)$.

A bag of words model can be easily expanded to include phrases of length m.
A **unigram** model is the case for which $m=1$. In the example, the unigram dictionary would be $(Mary;loves;apples;red)$. In the **bigram** case, $m=2$, the dictionary is $(Mary\ loves;loves\ apples;Red\ apples)$, and representations for each sample are $(1;1;0),(0;0;1)$. In this section, you will only use the unigram word features. These functions are already implemented for you in the `bag of words` function.
In `utils.py`, we have supplied you with the `load data` function, which can be used to read the `.tsv` files and returns the labels and texts. We have also supplied you with the `bag_of_words` function in `project1.py`, which takes the raw data and returns dictionary of unigram words. The resulting dictionary is an input

to `extract_bow_feature_vectors` which computes a feature matrix of ones and zeros that can be used as the input for the classification algorithms. Using the feature matrix and your implementation of learning algorithms from before, you will be able to compute θ and θ_0.

7. Classification and Accuracy

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

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.

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.
    """
```

 # TASK: FIND THIS FUCTION IN project1.py AND COMPLETE THE CODE

 # CHECK IT BY RUNNING test.py

Accuracy

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
    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.
"""
virtual_train_theta, virtual_train_theta_0 = classifier(train_feature_matrix, train_labels, **kwargs)
training_data_prediction = classify(train_feature_matrix, virtual_train_theta, virtual_train_theta_0)
accuracy_train = accuracy(training_data_prediction, train_labels)

validation_data_prediction = classify(val_feature_matrix, virtual_train_theta, virtual_train_theta_0)
accuracy_validation = accuracy(validation_data_prediction, val_labels)

```
    return accuracy_train, accuracy_validation
```

 # TASK: FIND THIS FUCTION IN project1.py AND COMPLETE THE CODE

 # CHECK IT BY RUNNING test.py

Baseline Accuracy

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

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

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

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

8. Parameter Tuning

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 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 λ=0.01 to tune T, and then use the best T to tune λ

Performance After Tuning

7/7 points (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=

With validation accuracy =

For the **average perceptron** algorithm:

T=

With validation accuracy =

For the **pegasos** algorithm:

T=

λ=

With validation accuracy =

Accuracy on the test set

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 :

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 :

Top 2:

Top 3:

Top 4:

Top 5:

Top 6:

Top 7:

Top 8:

Top 9:

Top 10:

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

9. Feature Engineering

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

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:

Change Binary Features to Counts Features

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:

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.