AveragePerceptron

October 23, 2024

1 The Average Perceptron algorithm

In this part of the assignment on Perceptrons, you will:

1. Implement the Average Perceptron training procedures.
2. Evaluate the Perceptron and Average Perceptron on a Spam filtering text classification task.
3. Bonus points.
4. Analyze the results.

1.1 Write Your Name Here:

2 Submission instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of ll cells).
4. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
5. Once you've rerun everything, select File -> Download as -> PDF via LaTeX and download a PDF version showing the code and the output of all cells, and save it in the same folder that contains the notebook file.
6. Look at the PDF file and make sure all your solutions are there, displayed correctly.
7. Submit **both** your PDF and the notebook file .ipynb on Canvas.
8. Make sure your your Canvas submission contains the correct files by downloading it after posting it on Canvas.

[ ]:
```
import numpy as np
import utils

np.random.seed(1)
```

2.1 The Perceptron algorithm

You can copy here the implementation of the training and test procedures for the Perceptron algorithm in homework 1.

The training algorithm runs for the specified number of epochs or until convergence, whichever happens first. The algorithm converged when it makes no mistake on the training examples. If the algorithm converged, display a message “Converged in <e> epochs!’.

[ ]:
```python
def perceptron_train(X, y, E):
    """Perceptron training function.
    Args:
        X (np.ndarray): A 2D array training instances, one per row.
        y (np.ndarray): A vector of labels.
        E (int): the maximum number of epochs.

    Returns:
        np.ndarray: The learned vector of weights / parameters.
    """
    # Add bias feature to X.
    X = # YOUR CODE HERE

    # Initialize w with zero's.
    w = # YOUR CODE HERE

    for e in range(E):
        # YOUR CODE HERE






    return w
```

Implement the Perceptron prediction function that takes as input the Perceptron parameters $\mathbf{w}$ and returns a vector with the labels (+1 or -1) predicted for the test examples in input data X.

Given a dataset of examples X (one example $\mathbf{x}$ per row of X), use the trained perceptron parameters in $\mathbf{w}$ to predict a label: $+1$ if $\mathbf{w}^T\mathbf{x} > 0$, -1 otherwise.

Return the vector of predicted labels, one label for each example $\mathbf{x} \in X$.

[ ]:
```python
def perceptron_test(X, w):
    """Perceptron prediction function.
    Args:
        X (np.ndarray): A 2D array training instances, one per row.
        w (np.ndarray): A vector of parameters.


    Returns:
        np.ndarray: The vector of predicted labels.
    """
    # Add bias feature to X.
    X = # YOUR CODE HERE

    # Compute the perceptron predictions on the examples in X.
```

```
    pred = # YOUR CODE HERE



    return pred
```

2.2 The Average Perceptron algorithm (30 points)

Implement the training procedure for the Average Perceptron algorithm.

The training algorithm runs for the specified number of epochs or until convergence of the normal Perceptron, whichever happens first. The algorithm has converged when it makes no mistake on the training examples. If the algorithm converged, display a message "Converged in <e> epochs!'. Return a tuple (w, avg) containing the last **w** vector and the average vector avg.

[ ]:
```
def aperceptron_train(X, y, E):
    """Average Perceptron training function.
    Args:
        X (np.ndarray): A 2D array training instances, one per row.
        y (np.ndarray): A vector of labels.
        E (int): the maximum number of epochs.

    Returns:
        (w, avg): The learned vector of weights and the average vector.
    """
    # Add bias feature to X.
    X = # YOUR CODE HERE

    # Initialize w and avg with zero's.
    w = # YOUR CODE HERE
    avg = # YOUR CODE HERE

    for e in range(E):
        # YOUR CODE HERE






    return w, avg
```

2.3 Spam Filtering with Perceptrons (60 points)

In this problem, you will train and evaluate spam classifiers using the perceptron and average perceptron algorithms. The dataset contains two files: *spam_train.txt* with 4,000 training examples and *spam_test.txt* with 1,000 test examples. The dataset is based on a subset of the Spam Assassin Public Corpus. Each line in the training and test files contains the pre-processed version of one email. The line starts with the label, followed by the email tokens separated by spaces.

Figure 1 shows a sample source email, while Figure 2 shows its pre-processed version in which web addresses are replaced with the "httpaddr" token, numbers are replaced with a "number" token, dollar amounts are replaced with "dollarnumb", and emaill addresses are replaced with "emailaddr". Furthermore, all words are lower-cased, HTML tags are removed, and words are reduced to their stems i.e. "expecting", "expected", "expectation" are all replaced with "expect". Non-words and punctuation symbols are removed.

Implement a function *create_vocabulary()* that takes as input the path to the training file and the path to a vocabulary file (we will use *spam_vocab.txt*) that is created by the function. The vocabulary file should contain a list of all the (pre-processed) tokens that appear in the training examples, together with their counts. The file should contain one token per line in the format <id> <token> <count>, where each token is associated a unique integer identifier. The tokens should be listed in increasing order of their identifiers, starting from 1. See for example the vocabulary file *data/newsgroups_vocab.txt* that we generated for the newsgroup topic classification problem.

[ ]:
```
def create_vocabulary(finput, foutput):
    """Function that creates the vocabulary and saves it into a file.
    Args:
        finput (string): The path to the training file.
        foutput (string): The path to the output file where the vocabulary will␣
 ↪be saved
    """

    # Initialize the vocabulary to an empty dictionary, create it from the␣
 ↪input file.
    # The dictionary will map each token to a tuple (id, count).
    vocab = {}
    with open(finput, "r", encoding="utf-8") as fi:
        for line in fi:
            # YOUR CODE HERE


    # Write the vocabulary into the output file.
    with open(foutput, "w", encoding="utf-8") as fo:
        # YOUR CODE HERE
```

Implement a function *load_vocabulary()* the reads the vocabulary from a file into a dictionary that maps each token to a tuple (id, count). Only include tokens that have counts greater than or equal with *mincount*.

[ ]:
```
def load_vocabulary(finput, mincount):
    """Function that reads the vocabulary from an input file.
    Args:
        finput (string): The path to the vocabulary file.
        mincount (int): The minimum count number.
```

```
    Returns:
        vocab: A Python dictionary containing the vocabulary, using only tokens␣
↪appearing at least mincount times.
    """

    vocab = {}
    with open(finput, "r", encoding="utf-8") as fi:
        for line in fi:
            # YOUR CODE HERE




    return vocab
```

Implement a function *create_svm_format* that reads an input file (e.g. training or testing data) and for each example in the file it creates a sparse feature vector representation wherein each example is represented as one line in the file using the format <label> <id1>:<val1> <id2>:<val2> ..., where the id's are listed in increasing order and correspond only to tokens that appear in that example (use 1 for all values, representing that fact that the corresponding token appeared in the example). An example of this sparse representation can be seen in the file *data/newsgroups_train1.txt* that we generated for the newsgroup topic classification problem. Save the new version of the dataset in the files *data/ spam_train_svm.txt* and *data/spam_test_svm.txt*.

[ ]:
```
def create_svm_format(finput, foutput, vocab):
    """Function that creates the sparse feature vector representation of a␣
↪dataset.
    Args:
        finput (string): The path to the file containing the dataset (e.g.,␣
↪training or test).
        foutput (string): The path to the output file where the new format of␣
↪the dataset will be saved.
        vocab (dict): The Python dictionary containing the vocabulary.
    """

    with open(finput, "r", encoding="utf-8") as fi, open(foutput, "w",␣
↪encoding="utf-8") as fo:
        for line in fi:
            # YOUR CODE HERE
```

The function *read_examples()* below reads all examples from a file with sparse feature vectors and returns a tuple *(data, labels)* where the *data* is a two dimensional array containing all feature vectors, one per row, in the same order as in the file, and the *labels* is a vector containing the corresponding labels. You can use this function to verify that your implementation of *create_svm_format()* is correct.

```
[ ]: def read_examples(file_name, nfeatures):
         """Function that creates the sparse feature vector representation of a␣
      ↪dataset.
         Args:
             file_name (string): the path to the file containing a set of examples␣
      ↪in the sparse feature vector format.
             nfeatures (int): the total number of features.

         Returns:
             (data, labels): 'data' is a two dimensional array containing all␣
      ↪feature vectors, one per row, in the same order as in the input file;
                             'labels' is a vector containing the corresponding␣
      ↪labels.
         """

         X = []
         t = []
         with open(file_name, "r", encoding="utf-8") as fi:
             for line in fi:
                 label, *features = line.strip().split(" ")
                 t.append(int(label))
                 f = [0] * nfeatures
                 for x in features:
                     index, value = x.split(":")
                     f[int(index) - 1] = float(value)
                 X.append(f)
         Xarray = np.array(X)
         tarray = np.array(t)

         return Xarray, tarray
```

Train the perceptron algorithm for 100 epochs or until convergence, whichever comes first. Make sure your code reads the training examples from *data/spam_train_svm.txt* and processes the examples in the order they are listed in the files. Use as features only tokens that appear at least 30 times in the training data.

Report the number of epochs needed for convergence, the number of mistakes made during each epoch for the first 25 epochs, and the total number of mistakes made, and save the returned parameter vector in *spam_model_p.txt*.

Test the perceptron algorithm by reading the parameter vector from *spam_model_p.txt* and the test examples from *data/spam_test_svm.txt* and calling `perceptron_test()`. Report the test accuracy.

Run the same experiment for the average perceptron algorithm, training for 100 epochs or until the corresponding perceptron convergences, whichever comes first. Save the returned parameter vector in *spam_model_ap.txt*.

Test the average perceptron algorithm by reading the parameter vector from *spam_model_ap.txt* and the test examples from *data/spam_test_svm.txt* and calling `perceptron_test()`. Report the test accuracy.

Compare the Perceptron vs. Average Perceptron performance and report in the Analysis section at the end of this notebook.

[ ]:
```python
# Create vocabulary from training examples.
create_vocabulary("../data/spam/spam_train.txt", "../data/spam/spam_vocab.txt")

# Read vocabulary, use only tokens appearing at least 30 times in the training␣
↪data..
vocab = load_vocabulary("../data/spam/spam_vocab.txt", 30)
print("Number of features is:", len(vocab))

# Generate sparse format file.
create_svm_format("../data/spam/spam_train.txt", "../data/spam/spam_train_svm.
↪txt", vocab)
create_svm_format("../data/spam/spam_test.txt", "../data/spam/spam_test_svm.
↪txt", vocab)

# Read the training and test examples from the sparse format files.
Xtrain, ttrain = read_examples("../data/spam/spam_train_svm.txt", len(vocab))
Xtest, ttest = read_examples("../data/spam/spam_test_svm.txt", len(vocab))

# Add the bias column to Xtrain and Xtest
Xtrain = # YOUR CODE HERE
Xtest = # YOUR CODE HERE

################# Perceptron experiment #################
# Train the Perceptron algorithm and save its parameters.
print("** Perceptron **")
np.savetxt("spam_model_p.txt", perceptron_train(Xtrain, ttrain, 100), "%.4f")

# Load the Perceptron parameters and evaluate its accuracy on the test examples.
w = np.loadtxt("spam_model_p.txt")
pred = perceptron_test(Xtest, w)
acc = # YOUR CODE HERE

print("Accuracy for Perceptron: %.4f\n" % acc)


################# Average Perceptron experiment #################
# Train the Average Perceptron algorithm and save its parameters.

print("** Average Perceptron **")
np.savetxt("spam_model_ap.txt", aperceptron_train(Xtrain, ttrain, 100)[1], "%.
↪4f")
```

```python
wavg = np.loadtxt("spam_model_ap.txt")
pred = perceptron_test(Xtest, wavg)
acc = # YOUR CODE HERE

print("Accuracy for Average Perceptron: %.4f\n" % acc)
```

[ ]:
```python
def read_examples(file_name, nfeatures):
    """Function that creates the sparse feature vector representation of a␣
 ↪dataset.
    Args:
        file_name (string): the path to the file containing a set of examples␣
 ↪in the sparse feature vector format.
        nfeatures (int): the total number of features.

    Returns:
        (data, labels): 'data' is a two dimensional array containing all␣
 ↪feature vectors, one per row, in the same order as in the input file;
                        'labels' is a vector containing the corresponding␣
 ↪labels.
    """

    # YOUR CODE HERE
```

For each version of the dataset, use the perceptron and average perceptron implementations that you wrote earlier and train the two algorithms for 10,000 epochs or until convergence, whichever comes first. Process the examples in the order they are listed in the files. Save the parameters in the corresponding *newsgroups_model_<x>.txt* files. Then evaluate the two perceptron algorithms on the corresponding test examples for each version. For each version, report and compare the accuracies of the two models.

[ ]:
```python
# YOUR CODE HERE
```

2.4 [Bonus] Spam Filtering with Support Vector Machines (20 points)

Train and test the SVM algorithm on the same *Spam vs. Non-spam* classification problem described above. Use a linear kernel, with the cost parameter $C = 5$. Report and compare the accuracy of the trained SVM models with the perceptron and average perceptron accuracies.

You are free to use packages with interfaces in Python such as *SVMLight*, *LIBSVM*, or *Scikit-Learn*. Their web sites contain plenty of documentation on how to use them. If you use *Scikit-Learn*, the following functionality from the `sklearn.svm` will be useful:

- `SVC()`: This is the main class used for SVM classification models. Its implementation is based on *LIBSVM*. Make sure that you properly map the SVM hyper-parameters to the parameters in the constructor of this class. The formulas for the kernels implemented by SVC are described in this UserGuide.

- `fit()`: This is the function used to train the classifier.
- `predict(x)`: This is used to calculate the (binary) label for sample x.

[ ]:
```
# YOUR CODE HERE
```

2.5 Anything extra goes here

[ ]:
```
```

2.6 Analysis of results (20 points)

Include here a nicely formatted report of the results, comparisons. Include explanations and any insights you can derive from the algorithm behavior and the results. This section is important, so make sure you address it appropriately.

Take advantage of the Jupyter Notebook markdown language, which can also process Latex and HTML, to format your report so that it looks professional.

[ ]:
```
```