Lab1 Report

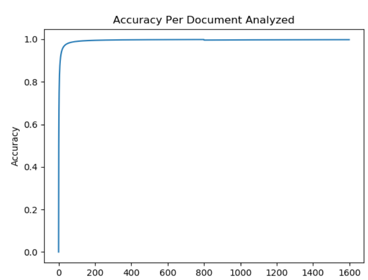
fmonteroperez1 - aca15fm

Our task was to implement a binary perceptron in python and investigate how by applying different enhancements we can enhance our classification algorithm to predict the classes of unseen document.

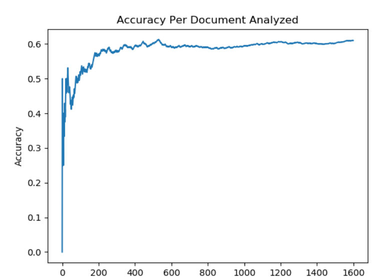
The first implementation was a binary perceptron which would only iterate over the training data once. The perceptron was firstly given 800 positive labelled documents and then 800 negatively labelled documents. From Figure 1a we can see that the more documents that the more test documents the perceptron uses to calculate weights, the better the accuracy. But at the time of processing the test data we can see from Table 1 1a) that it has a poor accuracy and a poor precision. From the Recall we can see see that the classifier is very good at labelling documents positive documents as positives, but does badly at correctly labelling negative documents.

To improve our perceptron the first step was to shuffle the training data to mitigate the effects of over-fitting our model The shuffled training data is more representative of the overall distribution of the data. From figure 1b we can see that as more documents are analysed the weights are correcting themselves, thus the fluctuations in the accuracy. From Table 1 1b) we can see that theres a slight improvement from the results of the non-shuffle test documents.

We stated that the more documents the perceptron sees the better it predicts labels, thus, to improve the classifying accuracy of the perceptron we want to iterate various times over the training data in order for our step function to produce more accurate weights. Figure 1b validates this claim, as we can see that for every iteration over the training data the accuracy of the classifier increases. Table 1.1b reflects this in the higher values for every aspect evaluated

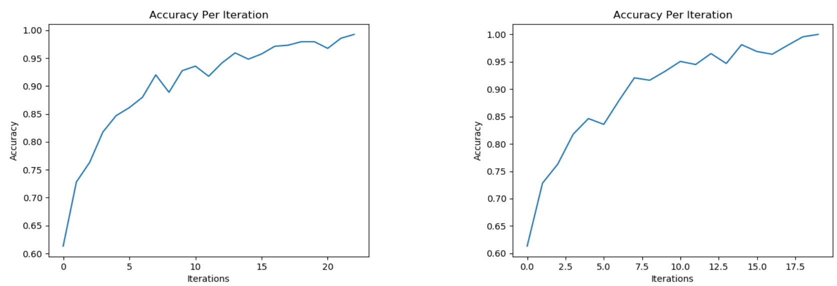


(a) Non-shuffled training docs



(b) Shuffled training docs

Figure 1: Accuracy increase for every training document analysed



(a) Normal Weights (b) Averaged Weights

Figure 2: Accuracy increase for every iteration over training set

	Accuracy	**Precision**	**Recall**	**F1-Score**
1a) Basic perceptron	0.5	0.5	1.0	0.667
1b) Shuffled datased	0.563	0.534	0.98	0.691
2a) Multiple iterations with shuffled training docs	0.837	0.826	0.855	0.840
2b) Multiple iterations with averaged weights 2b	0.87	0.873	0.865	0.869

Table 1: Perceptron analysis after enhancements

in comparison to the previous perceptron which only iterated once over the training documents.

The final improvement to the perceptron allowed it to use the average of the weights for every iteration. From the Figure 2b we can observe that the accuracy of the predicted results increases at a slower rate because the averaged weights reduce the effect of the model over-fitting. This results in the classifier being better overall as seen in the F-Score in 1 due to correctly classifying more documents, thus the higer precision, and also due to producing less false positives, thus the higher recall.

The most positively weighted features for each class are: great, job, seen, world, war, back, quite, Jackie, always for the positive class, and: if, boring, unfortunately, looks, script, nothing, plot, only, worst, bad for the negative class. The top negative features contain various strong sentiment words whilst the positive features seem to be less generalised. This implies that this same perceptron wont perform as well with test data that is from another domain and it will better predict negative documents.

Find attached the code for the assignment. The code was written and tested in Linux.

```python
import os, sys, re, random, argparse
from collections import Counter
from numpy import array, dot, random
import matplotlib.pyplot as plt

directory = os.path.join("c:\\", "path")

class Nlp:
    def __init__(self):
        self.training_documents = list()
        self.test_documents = list()
        self.weights = {}
        self.sum_of_weights = {}
        self.zeroed_weights = {}
        self.dictionary = Counter()
        self.vector_of_weights = []
        self.errors = 0
        self.errors_per_iteration = list()
        self.correct_predictions = 0
        self.positive_label = 1.0
        self.negative_label = -1.0
        self.seed = 20
        self.true_positives = 0
        self.false_positives = 0
        self.true_negatives = 0
        self.false_negatives = 0
        self.debug = False

    """
    This function takes in a path to a directory and the label for
    the documents in that directory. Then using those documentas
    it adds tuples containing (Document word counts, Label)
    to the training_documents list.
    """

    def process_training_data(self, path_to_files, label):
        #        documents = list()
        for file in os.listdir(path_to_files)[:800]:
            with open(path_to_files + file, 'r') as f:
                counted_words = re.sub("[^\w']", " ", f.read()).split()
                dictionary = Counter(counted_words)
                self.dictionary += dictionary
                training_element = (dictionary, label)
```

```
            self.training_documents.append(training_element)

    for key, count in self.dictionary.items():
        self.weights[key] = 0
        self.zeroed_weights[key] = 0

def process_test_data(self, path_to_files, label):
    for file in os.listdir(path_to_files)[800:]:
        with open(path_to_files + file, 'r') as f:
            counted_words = re.sub("[^\w']", " ", f.read()).split()
            dictionary = Counter(counted_words)
            test_data = (dictionary, label)
            self.test_documents.append(test_data)



"""
This function calculates the weights for different words
and plots a point in the graph for every document analysed
"""

def calculate_weights_bag_of_words(self, shuffle):
    errors = 0
    count = 0
    if shuffle:
        random.seed(self.seed)
        random.shuffle(self.training_documents)
    for document, label in self.training_documents:
        count += 1
        predicted_label = self.predict_labels(document, self.weights)
        if predicted_label != label:
            errors += 1
            for word, counts in document.items():
                self.weights[word] = self.weights[word] + (label * counts)
        self.errors_per_iteration.append(1 - (errors / count))

    if self.debug:
        print("-----------------------")
        print("total errors", errors)
        print("accuracy %", 1 - (errors / len(self.training_documents)))
        print("doc count", len(self.training_documents))

"""
This function calculates the weights of the words found
```

```python
    in the training_data list
    """

    def calculate_weights(self, iterations, shuffle):
        random.seed(self.seed)
        for i in range(iterations):
            errors = 0
            self.correct_predictions = 0
            if shuffle:
                random.shuffle(self.training_documents)
            for document, label in self.training_documents:
                predicted_label = self.predict_labels(document, self.weights)
                if predicted_label != label:
                    errors += 1
                    for word, counts in document.items():
                        self.weights[word] = self.weights[word] + label * counts
            if self.debug:
                print("-----------------------")
                print("iteration", i + 1)
                print("total errors", errors)
                print("accuracy %", 1 - (errors / len(self.training_documents)))
                print("doc count", len(self.training_documents))
            self.errors_per_iteration.append((1 - (errors / len(self.training_documents))))

    """
    Takes in a number of iterations for processing the data and a "debug" boolean
    to output to the console data about every iteration
    """

    def calculate_weights_averaged(self, iterations):
        random.seed(self.seed)
        c = 1  # used to keep track of which columns is being edited per iteration
        self.vector_of_weights.append(self.zeroed_weights.copy())
        self.sum_of_weights = self.zeroed_weights.copy()
        for i in range(iterations):
            errors = 0
            random.shuffle(self.training_documents)
            self.vector_of_weights.append(self.vector_of_weights[i].copy())

            for document, label in self.training_documents:
                predicted_label = self.predict_labels(document, self.weights)
                if predicted_label != label:
                    errors += 1
```

```
                for word, counts in document.items():
                    self.sum_of_weights[word] = self.sum_of_weights[word] + label * counts
            else:
                for word, counts in document.items():
                    self.sum_of_weights[word] = self.sum_of_weights[word]
            self.calculate_average_weights(c)
            c += 1

        if self.debug:
            print("-----------------------")
            print("iteration", i + 1)
            print("errors", errors)
            print("doc count", len(self.training_documents))
            print("accuracy", 1- (errors/len(self.training_documents)))
        self.errors_per_iteration.append(1 - (errors/len(self.training_documents)))

"""
Sets self.weights to the average of self.sum_of_weights
"""

def calculate_average_weights(self, iterations):
    for word, weight in self.sum_of_weights.items():
        self.weights[word] = weight / iterations

"""
Used to plot a graph for the errors per iteration
"""

def plot_errors(self, title, xlabel, ylabel):
    plt.plot(self.errors_per_iteration)
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)
    plt.title(title)
    # plt.ylim([0, 1])
    plt.show()


"""
This function returns a label for a given documents
based on the words that occur in that same document
"""
```

```python
def predict_labels(self, document, weights):
    score = 0.0
    for word, counts in document.items():
        if word not in self.weights:
            self.weights[word] = 0
        score += counts * weights[word]
    if score >= 0.0:
        return 1.0
    else:
        return -1.0

def evaluate_test_data(self):
    errors = 0
    for document, label in self.test_documents:
        # print(self.weights)
        predicted_label = self.predict_labels(document, self.weights)
        self.record_results(label, predicted_label)
        if predicted_label != label:
            # print(label, predicted_label)
            errors += 1

    print("-----------------------")
    print("total errors", errors)
    print("accuracy %", 1- (errors / 400))
    print("total documents %", len(self.test_documents))
    print("TPos {}, FPos {}, TNeg {}, FNeg{}".format(self.true_positives, self.false_positives, self.true_negatives,
                                                       self.false_negatives))


def record_results(self, actual_label, predicted_label):
    if actual_label == predicted_label:
        if actual_label == self.positive_label:
            self.true_positives += 1
        else:
            self.true_negatives += 1
    else:
        if predicted_label == self.positive_label:
            self.false_positives += 1
        else:
            self.false_negatives += 1

"""
```

```python
    Perfoms an analysis based on the classified documents
    and prints them in a human-readable way
    """

    def print_evaluation(self):
        print("TPos {}, FPos {}, TNeg {}, FNeg{}".format(self.true_positives, self.false_positives, self.true_negatives,
                                                          self.false_negatives))
        accuracy = (self.true_positives + self.true_negatives) / (
                    self.true_positives + self.true_negatives + self.false_positives + self.false_negatives)
        print("Accuracy: {}".format(accuracy))
        precision = self.true_positives / (self.true_positives + self.false_positives)
        print("Precision: {}".format(precision))
        recall = self.true_positives / (self.true_positives + self.false_negatives)
        print("Recall: {}".format(recall))
        f1_score = 2 * ((precision * recall) / (precision + recall))
        print("f1-score: {}".format(f1_score))


def print_top_weights(weights):
    s = [(k, weights[k]) for k in sorted(weights, key=weights.get, reverse=True)]
    for k, v in s[:10]:
        print(k, v)

    for k, v in s[-10:]:
        print(k, v)


# labels
positive_label = 1.0
negative_label = -1.0

"""
Process data to use in NLP object
"""
def do_data_processing(nlp, folder_name):
    nlp.process_training_data(folder_name+'/txt_sentoken/neg/', negative_label)
    nlp.process_training_data(folder_name+'/txt_sentoken/pos/', positive_label)
    nlp.process_test_data(folder_name+'/txt_sentoken/neg/', negative_label)
    nlp.process_test_data(folder_name+'/txt_sentoken/pos/', positive_label)


if __name__ == '__main__':
```

```python
parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('folder_name')

args = parser.parse_args()
folder_name = args.folder_name
print(args)

print('Number of arguments:', len(sys.argv), 'arguments.')
print('Argument List:', str(sys.argv))


## ----------------------------
## First perceptron, non-shuffled training data, only 1 iteration
## ----------------------------
print("-----------------------")
print("nlp - Perceptron 0, non-shuffled training data, only 1 iteration")
nlp = Nlp()
# nlp.debug = True
do_data_processing(nlp, folder_name)
# 1 repetition without shuffling the training data
shuffle_data = False
nlp.calculate_weights_bag_of_words(shuffle_data)

print("Plot Learning Curve")
title = "Accuracy Per Document"
ylabel = 'Accuracy'
xlabel = 'Documents'
nlp.plot_errors(title,xlabel,ylabel)
print("Evaluate data")
nlp.evaluate_test_data()
# nlp.print_evaluation()
print("-----------------------")



## ----------------------------
## Second perceptron, shuffled training data, only 1 iteration
## ----------------------------
print("-----------------------")
print("nlp1 - Perceptron 1, non-shuffled training data, only 1 iteration")
nlp1 = Nlp()
# nlp.debug = True
```

```python
do_data_processing(nlp1, folder_name)
# 1 repetition without shuffling the training data
shuffle_data = True
nlp1.calculate_weights_bag_of_words(shuffle_data)

print("Plot Learning Curve")
title = "Accuracy Per Document"
ylabel = 'Accuracy'
xlabel = 'Documents'
nlp1.plot_errors(title, xlabel, ylabel)

print("Evaluate data")
nlp1.evaluate_test_data()
# nlp1.print_evaluation()
print("-----------------------")


## -------------------------------
## Third perceptron, shuffled training data, 23 iterations
## -------------------------------
print("-----------------------")
print("nlp2 - Perceptron 2, shuffled training data, 23 iteration")
nlp2 = Nlp()
# nlp2.debug = True

do_data_processing(nlp2, folder_name)

# n repetitions and shuffling the training data
nlp2.calculate_weights(23, True)

print("Plot Learning Curve")
title = "Accuracy Per Iteration"
ylabel = 'Accuracy'
xlabel = 'Iterations'
nlp2.plot_errors(title,xlabel,ylabel)

print("Evaluate data")
nlp2.evaluate_test_data()
nlp2.print_evaluation()
# print_top_weights(nlp2.weights)
print("-----------------------")
```

```python
## ----------------------------
## Fourth Perceptron
## ----------------------------
print("-----------------------")
print("nlp3 - Perceptron 3, shuffled training data, weighted averaged, 18 iteration")
nlp3 = Nlp()
nlp3.debug = True

do_data_processing(nlp3, folder_name)
nlp3.calculate_weights_averaged(18) # accuracy .61

print("Plot Learning Curve")
title = "Accuracy Per Iteration"
ylabel = 'Accuracy'
xlabel = 'Iterations'
nlp3.plot_errors(title,xlabel,ylabel)

print("Evaluate data")
nlp3.evaluate_test_data()
nlp3.print_evaluation()
# print_top_weights(nlp3.weights)
print("-----------------------")
```