COMP527 Data Mining & Visualization: Text Classification Using Binary Perceptron Algorithm

Student 201442927. University of Liverpool.

Task 1. Explain...

> Explain the Perceptron algorithm for the binary classification case, providing its pseudo code. (20 marks)

The *Perceptron Algorithm*, published by Frank Rosenblatt in 1958, is inspired by the idea of a biological neuron which is sensitive to a number of stimuli and is deterministically activated when the effect of those combined stimuli exceeds some activation threshold.

Mathematically, we model this as the dot product of a vector of quantified stimuli $\mathbf{x}$ and a vector of weighted sensitivities $\mathbf{w}$, plus a bias term b.

We iterate through our training dataset of vectors with labels ($\mathbf{x}_i \in \mathbb{R}^N$, $y_i \in$ {-1, 1}), and whenever we get the wrong result, we adjust our weight vector accordingly: if our result was negative when it should have been positive, we *add* the incorrectly classified vector to the weight vector; if it was positive when it should have been negative then we *subtract* the incorrectly classified vector.

We also adjust our bias term, by adding the label $y_i \in$ {-1, 1}.

We repeat until the Perceptron is able to correctly classify every element in the training set, or when some specified maximum number of iterations is reached.

In the pseudo-code given by Daume III (2017:43) (http://ciml.info/dl/v0_99/ciml-v0_99-ch04.pdf), we have:

PerceptronTrain($\mathbf{D}$, *MaxIter*)

$w_d \leftarrow 0$, for all d = 1 ... D $b \leftarrow 0$

for *iter* = 1 . . . *MaxIter* **do**
. . . **for all** $(\mathbf{x}, y \in \mathbf{D})$ **do**
. . . . . . $a \leftarrow \sum_{d=1}^{D} w_d x_d + b$
. . . . . . **if** $ya \leq 0$ **then**
. . . . $w_d \leftarrow w_d + yx_d$, for all $d = 1 \ldots D$
. . . . $b \leftarrow b + y$
. . . . . . **end if**
. . . **end for**
end for
return $w_0, w_1, \ldots, w_D, b$

Task 2. Prove...

> Prove that for a linearly separable dataset, the perceptron algorithm will converge. (10 marks)

We first define a hyperplane, and show that any hyperplane $\in \mathbb{R}^N$ not intersecting the origin can be mapped to a hyperplane $\in \mathbb{R}^{N+1}$ which does intersect the origin; we then give define *linearly separable* in terms of a hyperplane; and we define the *perceptron algorithm* for an origin-intersecting hyperplane. We then prove the convergence of the perceptron algorithm.

Definition 1: A Hyperplane. A *hyperplane* H^{N-1} is an (N - 1) dimensional subspace of an N dimensional space, defined by a normal $\mathbf{n} \in \mathbb{R}^N$, and some constant $c \in \mathbb{R}$, such that $H^{N-1} = \{\mathbf{x} \in \mathbb{R}^N : \mathbf{x} \cdot \mathbf{n} = c \in \mathbb{R}\}$.

Lemma 1. We note that any hyperplane of dimension (N-1) can be projected to an origin-intersecting hyperplane H_0^N of dimension N (within an N+1 dimensional space $\mathbb{R}^{N+1}$), by mapping $f(x_1, \ldots, x_N) \rightarrow (1, x_1, \ldots, x_N)$, and $g(n_1, \ldots, n_N) \rightarrow (-c, n_1, \ldots, n_N)$ so that if we say $f(\mathbf{x}) = \mathbf{x}'$ and $g(\mathbf{n}) = \mathbf{n}'$ we have
$\mathbf{x}' \cdot \mathbf{n}' = -c + \mathbf{x} \cdot \mathbf{n} = 0$

Therefore we can describe $H^N = \{\mathbf{x}' \in \mathbb{R}^{N+1} : \mathbf{x}' \cdot \mathbf{n}' = 0 \in \mathbb{R}\}$.

Definition 2. A vectorised dataset in an N-dimensional attribute-space is *linearly separable* if there exists a hyperplane H such that all the points to one side of the hyperplane are of one category, and all the points to the other side are of the other.

Thus, given labels $y_i \in \{1, -1\}$ for each datapoint $\mathbf{x}_i \in \mathbb{R}^N$, $\exists \mathbf{n}$ such that $\forall \mathbf{x}_i$, $y_i(\mathbf{x}_i \cdot \mathbf{n}) - c > 0$.

If our hyperplane intersects the origin, then c = 0.

Definition 3: The Perceptron Algorithm.

Since Lemma 1 allows us to assume without loss of generality that our dataset is divided by an origin-intersecting hyperplane, we express the algorithm given that assumption.

$k \leftarrow 1; \mathbf{w}^0 \leftarrow \mathbf{0}$.
While $\exists i \in \{1, 2, \ldots, n\}$ such that $y_i(\mathbf{w}^k \cdot \mathbf{x}_i) \leq 0$:
...Find $j \in \{1, 2, \ldots, n\}$ such that $y_j(\mathbf{w}^k \cdot \mathbf{x}_j) \leq 0$.
... $\mathbf{w}^{k+1} \leftarrow \mathbf{w}^k + y_j \mathbf{x}_j$.
... $k \leftarrow k + 1$.
Return $\mathbf{w}^k$

Theorem. *For a linearly separable dataset, the Perceptron Algorithm will converge.*

Proof.

Lemma 1 means that we can assume without loss of generality that our dataset is divided by an origin-intersecting hyperplane.

In such a case it is simple to see that we can choose a normal $\mathbf{n}$ for our hyperplane such that

$$||\mathbf{n}|| = 1 \tag{0}$$

From the definition of linear separability we know that $\forall \mathbf{x}_i$, .

$$y_i(\mathbf{x}_i \cdot \mathbf{n}) - c > 0$$

We have chosen to remap our dataset so it is divided by an origin-intersecting hyperplane, so c=0 and $\forall \mathbf{x}_i$, .

$$y_i(\mathbf{x}_i \cdot \mathbf{n}) > 0$$

But given a finite dataset, we must have some point(s) that minimizes $y_i(\mathbf{x}_i \cdot \mathbf{n})$ and therefore it must be true $\forall \mathbf{x}_i, y_i$

$$y_i(\mathbf{x}_i \cdot \mathbf{n}) = 2\epsilon > \epsilon \in \mathbb{R} \tag{1}$$

We write $\mathbf{w}^k$ for the k-th iteration of $\mathbf{w}$.

Then the definition of the Perceptron Algorithm, plus (1) gives us

$$\begin{aligned} \mathbf{w}^{k+1} \cdot \mathbf{n} &= (\mathbf{w}^k + y_j \mathbf{x}_j) \cdot \mathbf{n} \\ &= \mathbf{w}^k \cdot \mathbf{n} + y_j(\mathbf{x}_j \cdot \mathbf{n}) \\ &> \mathbf{w}^k \cdot \mathbf{n} + \epsilon \end{aligned} \tag{3}$$

If

$$\mathbf{w}^k \cdot \mathbf{n} > k\epsilon \tag{4}$$

for k = 1, then it does $\forall k \in \mathbb{N}$

since (3) + (4)

$$\begin{aligned} \implies \mathbf{w}^{k+1} \cdot \mathbf{n} &> \mathbf{w}^k \cdot \mathbf{n} + \epsilon \\ &> k\epsilon + \epsilon = (k+1)\epsilon \end{aligned}$$

and from Definition 3 we know that

$$\mathbf{w}^0 = \mathbf{0}.$$

so substituting k=0 into (3), with (4) gives us

$$\mathbf{w}^1 > \mathbf{0} + \epsilon$$

and thus proving (4).

Since we have chosen $||\mathbf{n}|| = 1$ (0)

and we know that

$$||\mathbf{x}||||\mathbf{y}|| \geq \mathbf{x} \cdot \mathbf{y} \qquad \text{(by Cauchy-Schwarz)}$$

we therefore have

$$\begin{aligned} ||\mathbf{w}^k|| = ||\mathbf{w}^k||||\mathbf{n}|| &\geq \mathbf{w}^k \cdot \mathbf{n} \\ &> k\epsilon \end{aligned} \tag{5}$$

which gives us a lower bound on $||\mathbf{w}||$.

For an upper bound, we know from Definition 3 that $\mathbf{w}^{k+1} = \mathbf{w}^k + y_j \mathbf{x}_j$ and so

$$\begin{aligned} \|\mathbf{w}^{k+1}\|^2 &= \|\mathbf{w}^k + y_j \mathbf{x}_j\|^2 \\ &= \|\mathbf{w}^k\|^2 + \|\mathbf{x}_j\|^2 + 2y_j(\mathbf{w}^k \cdot \mathbf{x}_j) \\ &\leq \|\mathbf{w}^k\|^2 + \|\mathbf{x}_j\|^2 \qquad (6) \end{aligned}$$

since we only update $\mathbf{w}^k$ when it misclassifies $\mathbf{x}$ and so

$$y_j(\mathbf{w}^k \cdot \mathbf{x}_j) \leq 0 \qquad (\text{Definition 3})$$

Putting together (6) and (2) gives us

$$\|\mathbf{w}^{k+1}\|^2 \leq \|\mathbf{w}^k\|^2 + R^2 \qquad (7)$$

and so by induction

$$\|\mathbf{w}^{k+1}\|^2 \leq kR^2 \qquad (8)$$

$\forall k \in \mathbb{N}$ since if true for any $k \in \mathbb{N}$
then (8) + (7) $\implies$ $\|\mathbf{w}^{k+1}\|^2 \leq (k-1)R^2 + R^2 = kR^2$ and Def.3 $\implies$ $\mathbf{w}^0 = \mathbf{0} \leq R^2$

Putting together (8) and (5) we then have

$$\begin{aligned} kR^2 &\geq \|\mathbf{w}^{k+1}\|^2 \\ &> ((k+1)\epsilon)^2 \\ &\geq (k\epsilon)^2 \qquad (9) \\ &\geq k^2\epsilon^2 \qquad (10) \end{aligned}$$

(9) is true, since $\epsilon \geq 0$ and $k \in \mathbb{N}$.

From (10), we then have a limit on k:

$$k \leq \frac{R^2}{\epsilon^2}$$

and we are done.

Task 3. Implement...

> Implement a binary perceptron. (20 marks)

We first implement a Dataset object that will load the given data from the `train.data` and `test.data` text-files. We then implement a Perceptron object with Perceptron.train() and Perceptron.test() functions. To avoid having to also implement a Vector object we simply import NumPy.

All our final code is viewable in our `perceptron.py` file. Here we simply show the initial iteration of the Perceptron object class for the binary case:

In [1]:
```
import numpy as np
from perceptron import Dataset
```

In [2]:
```
class Perceptron:

    def __init__(self, positive_label='1', negative_label=False,):
        """Initialize Perceptron with given labels.

        Args:
            positive_label (str): Class names to give label `+1`
                Default to Class '1'.
            negative_label (str): Class names to give label `-1`
                Default to Class '2'.
        """

        self.positive_label = positive_label
        self.negative_label = negative_label

    def _label(self, D):
        """Label elements of dataset D, given Perceptron's positive and negative labe
ls.

        Args:
            D (Dataset): Dataset in the form of our declared Dataset class.
        """

        # store labels in dict
        y = {}
        for i in range(D.size):
            class_name = D.data[i]['class_name']

            if class_name == self.positive_label:
                y[i] = 1

            elif class_name == self.negative_label:
                y[i] = -1
            else:
                y[i] = 0

        return y

    def train(
        self,
        max_iterations = 10000,
        max_updates = 100000,
        D = Dataset('train.data'),
        ):
        """Train Perceptron for given number of iterations or updates.

        Args:
            max_iterations (int):
                Maximum number of times to iterate through dataset.
                Defaults to ten thousand.
            max_updates (int): Max times to update Perceptron weights.
                Defaults to one hundred thousand.
            D (Dataset): training dataset.
                Defaults to loading new Dataset from 'train.data' file.
        """

        # store each iteration of weights in dict
        w = {}

        # initialize weight_vector
        w[0] = np.zeros(D.features + 1)

        # get labels for D dataset
        y = self._label(D)

        k = 0
        updates_after_iteration = {}
```

```
        for n in range(max_iterations):

            for i in range(D.size):

                x = D.data[i]['x']

                if y[i]==0:
                    pass

                elif y[i]*(w[k].dot(x)) <= 0:
                    # ie. if incorrectly classified
                    w[k+1] = w[k] + y[i]*x
                    k = k+1
                    if k == max_updates:
                        break

            updates_after_iteration[n] = k
            if n>0 and updates_after_iteration[n] == updates_after_iteration[n-1]:
                break
            if k >= max_updates:
                break

        self.weights = w[k]
        self.updates = k
        self.history = w

    def test(self, D = Dataset('test.data'), silence=False):
        """Test given Perceptron on given Test Dataset and return score.

        Args:
            D (Dataset)
            silence (Boolean): If 'True' then will not print score.
        """

        w = self.weights

        # get labels for test dataset
        y = self._label(D)

        right = 0
        wrong = 0
        succeeds = []
        fails = []

        for i in range(D.size):

            x = D.data[i]['x']
            if y[i] == 0:
                pass
            elif y[i] * w.dot(x) > 0:
                right += 1
                succeeds.append((i, x, y[i]))
            else:
                wrong += 1
                fails.append((i, x, D.data[i]['class_name']))

        self.succeeds = succeeds
        self.fails = fails

        self.score = right/(right+wrong) * 100
        if silence == False:
            print(f'Success rate: {self.score}%')
        return
```

Task 4. Train...

> Use the binary perceptron to train classifiers to discriminate between (a) class 1 and class 2, (b) class 2 and class 3 and (c) class 1 and class 3. Report the train and test classification accuracies for each of the three classifiers after 20 iterations. Which pair of classes is most difficult to separate? (20 marks)

In [3]:
```
a = Perceptron('1','2')
a.train(max_iterations=20)
a.test()
```

```
Success rate: 100.0%
```

In [4]:
```
b = Perceptron('2','3')
b.train(max_iterations=20)
b.test()
```

```
Success rate: 50.0%
```

In [5]:
```
c = Perceptron('1','3')
c.train(max_iterations=20)
c.test()
```

```
Success rate: 100.0%
```

Classes '2' and '3' seem the most difficult to separate, as they have the worst test score after training with 20 iterations.

In [6]:
```
print(f'(a) converged after {a.updates} updates.')
print(f'(c) converged after {c.updates} updates.')
```

```
(a) converged after 5 updates.
(c) converged after 5 updates.
```

Our Perceptron Algorithm managed to converge on a solution to dividing classes '1' and '2' and classes '1' and '3' after just five updates each.

In [7]:
```
b.train(max_iterations=100000)
b.test(D=Dataset('train.data'))
```

```
Success rate: 93.75%
```

However, even if we allow our Perception Algorithm 100,000 iterations, it will still not converge on a weight-solution that gives 100% accuracy on our training dataset -- this suggests it is not linearly separable.

In [8]:
```
def minimize_failure(updates=10000):
    score = {}
    for i in range(updates):
        d = Perceptron('2','3')
        d.train(max_updates=i)
        d.test(D=Dataset('train.data'), silence=True)
        score[i] = len(d.fails)
    return score
```

In [9]:
```
scores = minimize_failure(1000)
```

In [10]:
```
import operator
min(scores.items(), key=operator.itemgetter(1))
```

Out[10]:
```
(374, 2)
```

In [11]:
```
minimize = Perceptron('2','3')
minimize.train(max_updates=374)
minimize.test(D=Dataset('train.data'),silence=True)
minimize.fails
```

Out[11]:
```
[(60, array([1. , 5.9, 3.2, 4.8, 1.8]), '2'),
 (73, array([1. , 6. , 2.7, 5.1, 1.6]), '2')]
```

Further analysis suggests that is the 60th and 73rd entries (starting from zero) of our dataset that are responsible for the classes not being linearly separable. This suggests they may be incorrectly classified, and would explain why the Perceptron Algorithm is unable to converge.

Task 5. Which...

> For the classifier (a) implemented in part (3) above, which feature is the most discriminative? (5 marks)

In [12]:
```
a.weights
```

Out[12]:
```
array([ 1. ,  1.3,  4.1, -5.2, -2.2])
```

Since we have included the bias term as the first (or perhaps *zero-th*) entry in our weight vector, the fourth and largest (in terms of absolute magnitude) value, that is `-5.2`, corresponds to **the third feature** of the dataset, and therefore seems to be the most discriminative.

Task 6. Extend...

> Extend the binary perceptron that you implemented in part (2) above to perform multi-class classification using the 1-vs-rest approach. Report the train and test classification accuracies for each of the three classes after training for 20 iterations. (15 marks),

To extend the binary perceptron we simply set the default value of `negative_value` to `False`, and adjusted the labelling function of our Perceptron class so that if no negative_value is explicitly given, then to label negatively all datapoints whose class is not the same as `positive_value`.

In [13]:
```
class Perceptron:

    def __init__(self, positive_label='1', negative_label=False):
        self.positive_label = positive_label
        self.negative_label = negative_label

    def _label(self, D):
        """Label elements of dataset D, given Perceptron's positive and negative labe
ls.

        Args:
            D (Dataset): Dataset in the form of our declared Dataset class.
        """

        # store labels in dict
        y = {}
        for i in range(D.size):
            class_name = D.data[i]['class_name']

            if class_name == self.positive_label:
                y[i] = 1
            elif self.negative_label:
                if class_name == self.negative_label:
                    y[i] = -1
                else:
                    y[i] = 0
            else:
                y[i] = -1

        return y
    ...
```

To demonstrate we import our adapted Perceptron from `perceptron.py` and report the accuracy for each class after training for up to 20 iterations.

In [14]:
```
from perceptron import Perceptron
```

In [15]:
```
d = Perceptron('1')
d.train(max_iterations=20)
d.test()
```

```
Success rate: 100.0%
```

In [16]:
```
e = Perceptron('2')
e.train(max_iterations=20)
e.test()
```

```
Success rate: 66.66666666666666%
```

In [17]:
```
f = Perceptron('3')
f.train(max_iterations=20)
f.test()
```

```
Success rate: 70.0%
```

Task 7. Regularise...

> Add an ℓ_2 regularisation term to your multi-class classifier implemented in question (5).
> Set the regularisation coefficient to 0.01, 0.1, 1.0, 10.0, 100.0 and compare the train and test classification accuracy for each of the three classes. (10 marks)

To add an ℓ_2 regularisation term we simply adjusted the update rule of our Perceptron.train() function, and added a `regularisation_coefficient` parameter so that we can set it as we call it. We set the default to be zero, so that if we don't specify it our Perceptron algorithm will function as before.

```
# the new update rule
w[k+1] = w[k] + y[i]*x - 2 * regularisation_coefficient * w[k]
```

In [18]:
```
# import the updated Perceptron
from perceptron import Perceptron
```

In [19]:
```
coefs = [0.01, 0.1, 1.0, 10.0, 100.0]
classes = ['1','2','3']
max_iterations = [20, 100, 1000]
```

In [20]:
```
regularize = {}
machine = {}

for c in classes:
    machine[c] = Perceptron(c)
```

In [22]:
```
for coef in coefs:
    regularize[coef] = {}

    for c in classes:
        regularize[coef][c] = {}
        for i in max_iterations:
            machine[c].train(regularisation_coefficient = coef, max_iterations = i)
            machine[c].test(silence=True)
            regularize[coef][c][i] = machine[c].score
```

We give the accuracy scores for each class, with the regularisation coefficient to 0.01, 0.1, 1.0, 10.0, 100.0, and with max_iterations of 20, 100 and 1000.

In [23]:
```
for coef in coefs:
    print(f'Regularisation Coefficient: {coef}\n')
    for c in classes:
        print(f'Class {c}:')
        for i in max_iterations:
            print(f'Iterations: {i}. Score: {regularize[coef][c][i]}')
        print('\n')
```

```
Regularisation Coefficient: 0.01

Class 1:
Iterations: 20. Score: 100.0
Iterations: 100. Score: 100.0
Iterations: 1000. Score: 100.0


Class 2:
Iterations: 20. Score: 66.66666666666666
Iterations: 100. Score: 60.0
Iterations: 1000. Score: 66.66666666666666


Class 3:
Iterations: 20. Score: 66.66666666666666
Iterations: 100. Score: 70.0
Iterations: 1000. Score: 66.66666666666666


Regularisation Coefficient: 0.1

Class 1:
Iterations: 20. Score: 100.0
Iterations: 100. Score: 100.0
Iterations: 1000. Score: 100.0


Class 2:
Iterations: 20. Score: 66.66666666666666
Iterations: 100. Score: 66.66666666666666
Iterations: 1000. Score: 66.66666666666666


Class 3:
Iterations: 20. Score: 33.33333333333333
Iterations: 100. Score: 53.333333333333336
Iterations: 1000. Score: 53.333333333333336


Regularisation Coefficient: 1.0

Class 1:
Iterations: 20. Score: 66.66666666666666
Iterations: 100. Score: 66.66666666666666
Iterations: 1000. Score: 66.66666666666666


Class 2:
Iterations: 20. Score: 66.66666666666666
Iterations: 100. Score: 66.66666666666666
Iterations: 1000. Score: 66.66666666666666


Class 3:
Iterations: 20. Score: 33.33333333333333
Iterations: 100. Score: 33.33333333333333
Iterations: 1000. Score: 33.33333333333333


Regularisation Coefficient: 10.0

Class 1:
Iterations: 20. Score: 66.66666666666666
Iterations: 100. Score: 66.66666666666666
Iterations: 1000. Score: 0.0
```

```
Class 2:
Iterations: 20. Score: 66.66666666666666
Iterations: 100. Score: 66.66666666666666
Iterations: 1000. Score: 0.0


Class 3:
Iterations: 20. Score: 33.33333333333333
Iterations: 100. Score: 33.33333333333333
Iterations: 1000. Score: 0.0


Regularisation Coefficient: 100.0

Class 1:
Iterations: 20. Score: 66.66666666666666
Iterations: 100. Score: 0.0
Iterations: 1000. Score: 0.0


Class 2:
Iterations: 20. Score: 66.66666666666666
Iterations: 100. Score: 0.0
Iterations: 1000. Score: 0.0


Class 3:
Iterations: 20. Score: 33.33333333333333
Iterations: 100. Score: 0.0
Iterations: 1000. Score: 0.0
```