Started on	Saturday, 3 October 2020, 12:53 PM
State	Finished
Completed on	Friday, 9 October 2020, 5:06 PM
Time taken	6 days 4 hours
Marks	5.91/6.00
Grade	**98.48** out of 100.00

Question **1**

Correct

Mark 1.00 out of 1.00

The kNN learning algorithm depends on two functions that must be passed to the algorithm:

- *distance*: this is a function that takes two objects and returns a non-negative number that is the distance between the objects according to some metric. This function is used to identify the neighbours of an object.
- *combine*: this is a function that takes a set of outputs and combines them in order to derive a new prediction.

In this question you have to write two concrete examples of these functions. Write the following functions:

- `euclidean_distance(v1, v2)` where `v1` and `v2` are two numeric vectors (sequences) with the same number of elements. The function must return the euclidean distance between the points represented by v1 and *v2*.
- `majority_element(labels)` where `labels` is a collection of class labels. The function must return a label that has the highest frequency (most common). [if there is a tie it doesn't matter which majority is returned.] This is an example of a combine function.

For example:

Test	Result

`from student_answer import euclidean_distance`
 	
`print(euclidean_distance([0, 3, 1, -3, 4.5],[-2.1, 1, 8, 1, 1]))`
	`9.25526876973327`

`from student_answer import majority_element`
 	
`print(majority_element([0, 0, 0, 0, 0, 1, 1, 1]))`
`print(majority_element("ababc") in "ab")`
	`0`
	`True`

Answer: (penalty regime: 0, 15, ... %)

```
import numpy as np
from collections import Counter

def euclidean_distance(v1, v2):
    vector_1 = np.array(v1)
    vector_2 = np.array(v2)
    distance = np.linalg.norm(vector_2 - vector_1)
    return distance

def majority_element(labels):
    c = Counter(labels)
    c.most_common()
    value, count = c.most_common()[0]
    return value
```

	Test	Expected	Got	

✔	`from student_answer import euclidean_distance`
	
	`print(euclidean_distance([0, 3, 1, -3, 4.5],[-2.1, 1, 8, 1, 1]))`
		`9.25526876973327`	`9.25526876973327`	✔

✔	`from student_answer import majority_element`
	
	`print(majority_element([0, 0, 0, 0, 0, 1, 1, 1]))`
	`print(majority_element("ababc") in "ab")`
		`0`
		`True`
			`0`
			`True`
				✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **2**

Correct

Mark 1.00 out of 1.00

Write a function `knn_predict(input, examples, distance, combine, k)` that takes an input and predicts the output by combining the output of the *k* nearest neighbours. If after selecting *k* nearest neighbours, the distance to the farthest selected neighbour and the distance to the nearest unselected neighbour are the same, more neighbours must be selected until these two distances become different or all the examples are selected. The description of the parameters of the function are as the following:

- `input`: an input object whose output must be predicted. Do not make any assumption about the type of input other than that it can be consumed by the distance function.
- `examples`: a collection of pairs. In each pair the first element is of type input and the second element is of type output.
- `distance`: a function that takes two objects and returns a non-negative number that is the distance between the two objects according to some metric.
- `combine`: a function that takes a set of outputs and combines them in order to derive a new prediction (output).
- `k`: a positive integer which is the number of nearest neighbours to be selected. If there is a tie more neighbours will be selected (see the description above).

Note: the `majority_element` function used in some test cases returns the smallest element when there is a tie. For example `majority_element('--++')` returns `'+'` because it is the most common label (like `-`) and in the character encoding system `'+'` comes before `'-'`.

For example:

Test	Result

`from student_answer import knn_predict`
 	
`examples = [`
`  ([2], '-'),`
`  ([3], '-'),`
`  ([5], '+'),`
`  ([8], '+'),`
`  ([9], '+'),`
`]`
 	
`distance = euclidean_distance`
`combine = majority_element`
 	
`for k in range(1, 6, 2):`
`  print("k =", k)`
`  print("x", "prediction")`
`  for x in range(0,10):`
`  print(x, knn_predict([x], examples, distance, combine, k))`
`  print()`
	`k = 1`
	`x prediction`
	`0 -`
	`1 -`
	`2 -`
	`3 -`
	`4 +`
	`5 +`
	`6 +`
	`7 +`
	`8 +`
	`9 +`
	
	`k = 3`
	`x prediction`
	`0 -`
	`1 -`
	`2 -`
	`3 -`
	`4 -`
	`5 +`
	`6 +`
	`7 +`
	`8 +`
	`9 +`
	
	`k = 5`
	`x prediction`
	`0 +`
	`1 +`
	`2 +`
	`3 +`
	`4 +`
	`5 +`
	`6 +`
	`7 +`
	`8 +`
	`9 +`

Test	Result

`from student_answer import knn_predict`
 	
`# using knn for predicting numeric values`
 	
`examples = [`
`    ([1], 5),`
`    ([2], -1),`
`    ([5], 1),`
`    ([7], 4),`
`    ([9], 8),`
`]`
 	
`def average(values):`
`    return sum(values) / len(values)`
 	
`distance = euclidean_distance`
`combine = average`
 	
`for k in range(1, 6, 2):`
`    print("k =", k)`
`    print("x", "prediction")`
`    for x in range(0,10):`
`        print("{} {:4.2f}".format(x, knn_predict([x], examples, distance, combine,`
`k)))`
`    print()`
	`k = 1`
	`x prediction`
	`0 5.00`
	`1 5.00`
	`2 -1.00`
	`3 -1.00`
	`4 1.00`
	`5 1.00`
	`6 2.50`
	`7 4.00`
	`8 6.00`
	`9 8.00`
	
	`k = 3`
	`x prediction`
	`0 1.67`
	`1 1.67`
	`2 1.67`
	`3 1.67`
	`4 2.25`
	`5 1.33`
	`6 4.33`
	`7 4.33`
	`8 4.33`
	`9 4.33`
	
	`k = 5`
	`x prediction`
	`0 3.40`
	`1 3.40`
	`2 3.40`
	`3 3.40`
	`4 3.40`
	`5 3.40`
	`6 3.40`
	`7 3.40`
	`8 3.40`
	`9 3.40`

Answer: (penalty regime: 0, 15, ... %)

```
import numpy as np
from collections import Counter
#https://towardsdatascience.com/k-nearest-neighbours-introduction-to-machine-learning-algorithms-18e
def knn_predict(input, examples, distance, combine, k):
    neighbours = []
    for i in range(len(examples)):
        #append distance and result to neighbour list
        neighbours.append((distance(input, examples[i][0]), examples[i][1]))
    #sort the list
    neighbours.sort()
    nearest_neighbour = []
    for range_of_neighbours in range(k):
        nearest_neighbour.append(neighbours[range_of_neighbours])
    #checking the value of k i need to use based on conditions provided in the questoin
    while k < len(neighbours) and nearest_neighbour[-1][0] == neighbours[k][0]:
        nearest_neighbour.append(neighbours[k])
        k += 1
    selected_neighbours = []
    for j in range(len(nearest_neighbour)):
        #append all the results of the nearest neighbours
        selected_neighbours.append(nearest_neighbour[j][1])


    #get the prediction based on the majority or selected nearest neighbours results
    majority_result = combine(selected_neighbours)
    return majority_result

```

	Test	Expected	Got	

✔	`from student_answer import knn_predict`
	
	`examples = [`
	`    ([2], '-'),`
	`    ([3], '-'),`
	`    ([5], '+'),`
	`    ([8], '+'),`
	`    ([9], '+'),`
	`]`
	
	`distance = euclidean_distance`
	`combine = majority_element`
	
	`for k in range(1, 6, 2):`
	`    print("k =", k)`
	`    print("x", "prediction")`
	`    for x in range(0,10):`
	`        print(x, knn_predict([x], examples, distance, combine, k))`
	`    print()`
		`k = 1`
		`x prediction`
		`0 -`
		`1 -`
		`2 -`
		`3 -`
		`4 +`
		`5 +`
		`6 +`
		`7 +`
		`8 +`
		`9 +`
		
		`k = 3`
		`x prediction`
		`0 -`
		`1 -`
		`2 -`
		`3 -`
		`4 -`
		`5 +`
		`6 +`
		`7 +`
		`8 +`
		`9 +`
		
		`k = 5`
		`x prediction`
		`0 +`
		`1 +`
		`2 +`
		`3 +`
		`4 +`
		`5 +`
		`6 +`
		`7 +`
		`8 +`
		`9 +`
			`k = 1`
			`x prediction`
			`0 -`
			`1 -`
			`2 -`
			`3 -`
			`4 +`
			`5 +`
			`6 +`
			`7 +`
			`8 +`
			`9 +`
			
			`k = 3`
			`x prediction`
			`0 -`
			`1 -`
			`2 -`
			`3 -`
			`4 -`
			`5 +`
			`6 +`
			`7 +`
			`8 +`
			`9 +`
			
			`k = 5`
			`x prediction`
			`0 +`
			`1 +`
			`2 +`
			`3 +`
			`4 +`
			`5 +`
			`6 +`
			`7 +`
			`8 +`
			`9 +`
				✔

	Test	Expected	Got	

✔	`from student_answer import knn_predict`
	
	`# using knn for predicting numeric values`
	
	`examples = [`
	`  ([1], 5),`
	`  ([2], -1),`
	`  ([5], 1),`
	`  ([7], 4),`
	`  ([9], 8),`
	`]`
	
	`def average(values):`
	`  return sum(values) / len(values)`
	
	`distance = euclidean_distance`
	`combine = average`
	
	`for k in range(1, 6, 2):`
	`  print("k =", k)`
	`  print("x", "prediction")`
	`  for x in range(0,10):`
	`  print("{} {:4.2f}".format(x, knn_predict([x], examples, distance, combine, k)))`
	`  print()`
		k = 1
		x prediction
		0 5.00
		1 5.00
		2 -1.00
		3 -1.00
		4 1.00
		5 1.00
		6 2.50
		7 4.00
		8 6.00
		9 8.00
		
		k = 3
		x prediction
		0 1.67
		1 1.67
		2 1.67
		3 1.67
		4 2.25
		5 1.33
		6 4.33
		7 4.33
		8 4.33
		9 4.33
		
		k = 5
		x prediction
		0 3.40
		1 3.40
		2 3.40
		3 3.40
		4 3.40
		5 3.40
		6 3.40
		7 3.40
		8 3.40
		9 3.40
			k = 1
			x prediction
			0 5.00
			1 5.00
			2 -1.00
			3 -1.00
			4 1.00
			5 1.00
			6 2.50
			7 4.00
			8 6.00
			9 8.00
			
			k = 3
			x prediction
			0 1.67
			1 1.67
			2 1.67
			3 1.67
			4 2.25
			5 1.33
			6 4.33
			7 4.33
			8 4.33
			9 4.33
			
			k = 5
			x prediction
			0 3.40
			1 3.40
			2 3.40
			3 3.40
			4 3.40
			5 3.40
			6 3.40
			7 3.40
			8 3.40
			9 3.40
				✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **3**

Correct

Mark 1.00 out of 1.00

A perceptron is a function that takes a vector (list of numbers) of size *n* and returns 0 or 1 according to the definition of perceptron.

Write a function `construct_perceptron(weights, bias)` where `weights` is a vector (list of numbers) of of length *n* and `bias` is a scalar number and returns the corresponding perceptron function.

For example:

Test	Result

`from student_answer import construct_perceptron`
 	
`weights = [2, -4]`
`bias = 0`
`perceptron = construct_perceptron(weights, bias)`
 	
`print(perceptron([1, 1]))`
`print(perceptron([2, 1]))`
`print(perceptron([3, 1]))`
`print(perceptron([-1, -1]))`
	`0`
	`1`
	`1`
	`1`

Answer: (penalty regime: 0, 15, ... %)

Reset answer

```
def construct_perceptron(weights, bias):
    """Returns a perceptron function using the given parameters."""
    def perceptron(input):
        # Complete (a line or two)
        sums = 0
        for i in range(len(weights)):
            sums += weights[i] * input[i]
        # Note: we are masking the built-in input function but that is
        # fine since this only happens in the scope of this function and the
        # built-in input is not needed here.
        decision = sums + bias
        output = 0
        if decision < 0:
            output = 0
        else:
            output = 1
        return output

    return perceptron # this line is fine
```

	Test	Expected	Got	

✔	`from student_answer import construct_perceptron`
	
	`weights = [2, -4]`
	`bias = 0`
	`perceptron = construct_perceptron(weights, bias)`
	
	`print(perceptron([1, 1]))`
	`print(perceptron([2, 1]))`
	`print(perceptron([3, 1]))`
	`print(perceptron([-1, -1]))`
		`0`
		`1`
		`1`
		`1`
			`0`
			`1`
			`1`
			`1`
				✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **4**

Correct

Mark 1.00 out of 1.00

Write a function `accuracy(classifier, inputs, expected_outputs)` that passes each input in the sequence of inputs to the given classifier function (e.g. a perceptron) and compares the predictions with the expected outputs. The function must return the accuracy of the classifier on the given data. Accuracy must be a number between 0 and 1 (inclusive).

Note: an important application of a metric such as accuracy is to see how a classifier (e.g. a spam filter) performs on unseen data. In this case, the inputs must be some data that it has not seen during training but has been labeled by humans.

For example:

Test	Result

`from student_answer import accuracy`
 	
`perceptron = construct_perceptron([-1, 3], 2)`
`inputs = [[1, -1], [2, 1], [3, 1], [-1, -1]]`
`targets = [0, 1, 1, 0]`
 	
`print(accuracy(perceptron, inputs, targets))`
	`0.75`

Answer: (penalty regime: 0, 15, ... %)

```
def accuracy(classifier, inputs, expected_outputs):
    total_results = len(inputs)
    correct = 0
    wrong = 0
    for i in inputs:
        result = classifier(i)
        if result == 0:
            wrong += 1
        else:
            correct += 1
    return correct/total_results

```

	Test	Expected	Got	

✔	`from student_answer import accuracy`
	
	`perceptron = construct_perceptron([-1, 3], 2)`
	`inputs = [[1, -1], [2, 1], [3, 1], [-1, -1]]`
	`targets = [0, 1, 1, 0]`
	
	`print(accuracy(perceptron, inputs, targets))`
		`0.75`	`0.75`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **5**

Correct

Mark 0.91 out of 1.00

Consider a binary classification problem (i.e. there are two classes in the domain) where each object is represented by 2 numeric values (2 features). We are using a single perceptron as a classifier for this domain and want to learn its parameters. The weight update rule is $w \leftarrow w + \eta x(t - y)$. We use the following configuration.

```
weights = [-0.5, 0.5]
bias = -0.5
learning_rate = 0.5

examples = [
    ([1, 1],   0),    # index 0 (first example)
    ([2, 0],   1),
    ([1, -1],  0),
    ([-1, -1], 1),
    ([-2, 0],  0),
    ([-1, 1],  1),
]
```

Answer the following with numeric values. Do not use fractions.

- After seeing the example at index 0, the value of the weight vector is [-0.5 ✔ , 0.5 ✔] and the value of bias is -0.5 ✔ .
- After seeing the example at index 1, the value of the weight vector is [0.5 ✔ , 0.5 ✔] and the value of bias is 0 ✔ .
- After seeing the example at index 2, the value of the weight vector is [0 ✔ , 1 ✔] and the value of bias is -0.5 ✔ .
- The smallest network of perceptrons required to perfectly learn this problem, has 3 ✔ layers (including the input and output layers) and has a total of 4 ✔ perceptrons.

Correct

Marks for this submission: 1.00/1.00. Accounting for previous tries, this gives **0.91/1.00**.

Question **6**

Correct

Mark 1.00 out of 1.00

Write a function `learn_perceptron_parameters(weights, bias, training_examples, learning_rate, max_epochs)` that adjusts the weights and bias by iterating through the training data and applying the perceptron learning rule. The function must return a pair (2-tuple) where the first element is the vector (list) of adjusted weights and second argument is the adjusted bias. The parameters of the function are:

- `weights`: an array (list) of initial weights of length *n*
- `bias`: a scalar number which is the initial bias
- `training_examples`: a list of training examples where each example is a pair. The first element of the pair is a vector (tuple) of length *n*. The second element of the pair is an integer which is either 0 or 1 representing the negative or positive class correspondingly.
- `learning_rate`: a positive number representing eta in the learning equations of perceptron.
- `max_epochs`: the maximum number of times the learner is allowed to iterate through all the training examples.

For example:

Test	Result

`from student_answer import learn_perceptron_parameters`
 	
`weights = [2, -4]`
`bias = 0`
`learning_rate = 0.5`
`examples = [`
`  ((0, 0), 0),`
`  ((0, 1), 0),`
`  ((1, 0), 0),`
`  ((1, 1), 1),`
`  ]`
`max_epochs = 50`
 	
`weights, bias = learn_perceptron_parameters(weights, bias, examples, learning_rate, max_epochs)`
`print(f"Weights: {weights}")`
`print(f"Bias: {bias}\n")`
 	
`perceptron = construct_perceptron(weights, bias)`
 	
`print(perceptron((0,0)))`
`print(perceptron((0,1)))`
`print(perceptron((1,0)))`
`print(perceptron((1,1)))`
`print(perceptron((2,2)))`
`print(perceptron((-3,-3)))`
`print(perceptron((3,-1)))`
	`Weights: [1.0, 0.5]`
	`Bias: -1.5`
	
	`0`
	`0`
	`0`
	`1`
	`1`
	`0`
	`1`

`from student_answer import learn_perceptron_parameters`
 	
`weights = [2, -4]`
`bias = 0`
`learning_rate = 0.5`
`examples = [`
`  ((0, 0), 0),`
`  ((0, 1), 1),`
`  ((1, 0), 1),`
`  ((1, 1), 0),`
`  ]`
`max_epochs = 50`
 	
`weights, bias = learn_perceptron_parameters(weights, bias, examples, learning_rate, max_epochs)`
`print(f"Weights: {weights}")`
`print(f"Bias: {bias}\n")`
	`Weights: [-0.5, -0.5]`
	`Bias: 0.0`

Answer: (penalty regime: 0, 15, ... %)

```
def learn_perceptron_parameters(weights, bias, training_examples, learning_rate, max_epochs):
    for epoch in range(0, max_epochs):
        #print("weights: ", weights)
        #print("bias: ", bias)
        error = False
        for input, target in training_examples:
            a = bias + sum(weights[i] * input[i] for i in range(len(input)))
            output = 1 if a >= 0 else 0
            #print("input: {} output: {} target: {}".format(input, output, target))
            if output != target:
                error = True
                #update the weights and bias
                weights = [weights[i] + learning_rate * input[i] * (target - output) for i in range(
                bias = bias + learning_rate * (target - output)
```

```
                bias = bias + learning_rate * (target - output)
                #print("updating the weights and bias to: ", weights, bias)
                if not error:
                    def perceptron(input_vector):
                        a = bias + sum(weights[i] * input_vector[i] for i in range(len(input)))
                        output = 1 if a >= 0 else 0
                        return output
    return weights, bias
```

	Test	Expected	Got	

✔	`from student_answer import learn_perceptron_parameters`
	
	`weights = [2, -4]`
	`bias = 0`
	`learning_rate = 0.5`
	`examples = [`
	`  ((0, 0), 0),`
	`  ((0, 1), 0),`
	`  ((1, 0), 0),`
	`  ((1, 1), 1),`
	`  ]`
	`max_epochs = 50`
	
	`weights, bias = learn_perceptron_parameters(weights, bias,`
	`examples, learning_rate, max_epochs)`
	`print(f"Weights: {weights}")`
	`print(f"Bias: {bias}\n")`
	
	`perceptron = construct_perceptron(weights, bias)`
	
	`print(perceptron((0,0)))`
	`print(perceptron((0,1)))`
	`print(perceptron((1,0)))`
	`print(perceptron((1,1)))`
	`print(perceptron((2,2)))`
	`print(perceptron((-3,-3)))`
	`print(perceptron((3,-1)))`
		Weights:
		[1.0, 0.5]
		Bias: -1.5
		
		0
		0
		0
		1
		1
		0
		1
			Weights:
			[1.0, 0.5]
			Bias: -1.5
			
			0
			0
			0
			1
			1
			0
			1
				✔

✔	`from student_answer import learn_perceptron_parameters`
	
	`weights = [2, -4]`
	`bias = 0`
	`learning_rate = 0.5`
	`examples = [`
	`  ((0, 0), 0),`
	`  ((0, 1), 1),`
	`  ((1, 0), 1),`
	`  ((1, 1), 0),`
	`  ]`
	`max_epochs = 50`
	
	`weights, bias = learn_perceptron_parameters(weights, bias,`
	`examples, learning_rate, max_epochs)`
	`print(f"Weights: {weights}")`
	`print(f"Bias: {bias}\n")`
		Weights:
		[-0.5, -0.5]
		Bias: 0.0
			Weights:
			[-0.5, -0.5]
			Bias: 0.0
				✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

◄ 9. Machine learning with naive Bayes nets

Jump to...

11. Games ►