1. First, we will use the string kernel function for our kernel. Recall from class that given two strings s and t, the string kernel $K_p(s, t)$ is the number of substrings of length p that are common to both s and t, where a string that occurs a times in s and b times in t is counted ab times. For this problem, use p = 3, p = 4 and p = 5. Write down the training and test errors of kernel perceptron for p = 3, 4, 5 on this dataset.

	Training Error	Test Error
p = 3	0.0124	0.0409
p = 4	0.0069	0.0264
p = 5	0.0069	0.0343

2. Next, repeat Part (1) with a slight modification of the string kernel, $M_p(s, t)$. Given two strings s and t, the modified string kernel $M_p(s, t)$ is the number of substrings of length p that are common to both s and t, where a string that occurs a times in s and b times in t is counted only once. What are the training and test errors for this kernel for p = 3, 4, 5?

	Training Error	Test Error
p = 3	0.0127	0.0541
p = 4	0.0074	0.0290
p = 5	0.0069	0.0343

3. Find the two coordinates in w with the highest positive values. You should be able to do this without explicitly computing all the coordinates of w. What are the substrings corresponding to these coordinates? These coordinates correspond to those substrings whose presence most strongly indicates that the protein belongs in the family.

The substrings corresponding to the coordinates with the highest positive values are “WDTAG”, “DTAGQ”, “LFLNK”, “GKSSL”, and “KVGPD”. All of them have coordinate value of 3.

pa4

May 30, 2018

In [ ]:
```
import numpy
import operator

class Solver():
    def __init__(self):
        # list that contains all the training data
        self.training_data = []
        # list that contains all the test data
        self.test_data = []
    def load_data(self):
        # open the training data file
        training_file = open("pa4train.txt")
        # load the training data
        for line in training_file:
            data = line.split()
            data[-1] = int(data[-1])
            self.training_data.append(data)
        # close the training data file
        training_file.close()
        # open the test data file
        test_file = open("pa4test.txt")
        # load the test data
        for line in test_file:
            data = line.split()
            data[-1] = int(data[-1])
            self.test_data.append(data)
        # close the test data file
        test_file.close()
    def kernel1(self, s, t, p):
        # dictionary that maps the substring to its appearances in s
        map1 = {}
        # get all the substrings of length p in string s
        for i in range(0, (len(s) - p + 1)):
            # get the substring
            substring = s[i : (i + p)]
            # update the dictionary
            if substring not in map1:
```

```
                map1[substring] = 1
            else:
                map1[substring] = map1[substring] + 1
        # dictionary that maps the substring to its appearances in t
        map2 = {}
        # get all the substrings of length p in string t
        for i in range(0, (len(t) - p + 1)):
            # get the substring
            substring = t[i : (i + p)]
            # update the dictionary
            if substring not in map2:
                map2[substring] = 1
            else:
                map2[substring] = map2[substring] + 1
        # initialize the count
        count = 0
        for i in map1:
            if i in map2:
                # update the count
                count = count + map1[i] * map2[i]
        return count
    def kernel2(self, s, t, p):
        # dictionary that maps the substring to its appearances in s
        map1 = {}
        # get all the substrings of length p in string s
        for i in range(0, (len(s) - p + 1)):
            # get the substring
            substring = s[i : (i + p)]
            # update the dictionary
            if substring not in map1:
                map1[substring] = 1
        # dictionary that maps the substring to its appearances in t
        map2 = {}
        # get all the substrings of length p in string t
        for i in range(0, (len(t) - p + 1)):
            # get the substring
            substring = t[i : (i + p)]
            # update the dictionary
            if substring not in map2:
                map2[substring] = 1
        # initialize the count
        count = 0
        for i in map1:
            if i in map2:
                # update the count
                count = count + map1[i] * map2[i]
        return count
    def kernelization(self, option, p, current_data, data_set):
```

```
        # return 0 if the data set is empty
        if len(data_set) == 0:
            return 0
        # initialize the value
        value = 0
        # calculate the value
        for i in data_set:
            if option == 1:
                value = value + i[-1] * self.kernel1(current_data[0], i[0], p)
            else:
                value = value + i[-1] * self.kernel2(current_data[0], i[0], p)
        return value
    def perceptron(self, option, p):
        # list that contains all the training data which requires update of the classi
        data_set = []
        # iterate through the training data
        for i in self.training_data:
            if (i[-1] * self.kernelization(option, p, i, data_set)) <= 0:
                data_set.append(i)
        # return the data set
        return data_set
    def calculate_errors(self, option, p, classifier):
        # initialize the training error
        training_error = 0.0
        errors = 0
        # iterate through the training data
        for i in self.training_data:
            # make the classification
            classification = numpy.sign(self.kernelization(option, p, i, classifier))
            if classification == 0:
                classification = -1
            # check whether the classification is correct
            if classification != i[-1]:
                errors = errors + 1
        # calculate the training error
        training_error = float(errors) / float(len(self.training_data))
        # initialize the test error
        test_error = 0.0
        errors = 0
        # iterate through the training data
        for i in self.test_data:
            # make the classification
            classification = numpy.sign(self.kernelization(option, p, i, classifier))
            if classification == 0:
                classification = -1
            # check whether the classification is correct
            if classification != i[-1]:
                errors = errors + 1
```

```
        # calculate the test error
        test_error = float(errors) / float(len(self.test_data))
        return training_error, test_error
    def find_coordinates(self, classifier):
        feature_map = {}
        # iterate through the training data
        for i in classifier:
            # get all the substrings of length 5
            for j in range(0, (len(i[0]) - 5 + 1)):
                substring = i[0][j : (j + 5)]
                # update the feature map
                if substring not in feature_map:
                    feature_map[substring] = i[-1]
                else:
                    feature_map[substring] = feature_map[substring] + i[-1]
        # sort the feature
        sorted_map = sorted(feature_map.items(), key=operator.itemgetter(1))
        print(sorted_map)
        # find the substrings
        print(sorted_map[-1])
        print(sorted_map[-2])


if __name__ == '__main__':
    # create the solver
    solver = Solver()
    # load the data
    solver.load_data()

    print("String Kernel K")
    print("p = 2")
    data_set = solver.perceptron(1, 2)
    # calculate the errors
    training_error, test_error = solver.calculate_errors(1, 2, data_set)
    print("training error:", training_error, "test error:", test_error)
    print("p = 3")
    data_set = solver.perceptron(1, 3)
    # calculate the errors
    training_error, test_error = solver.calculate_errors(1, 3, data_set)
    print("training error:", training_error, "test error:", test_error)
    print("p = 4")
    data_set = solver.perceptron(1, 4)
    # calculate the errors
    training_error, test_error = solver.calculate_errors(1, 4, data_set)
    print("training error:", training_error, "test error:", test_error)
    print("p = 5")
    data_set = solver.perceptron(1, 5)
    # find the two coordinates
```

```python
solver.find_coordinates(data_set)
# calculate the errors
training_error, test_error = solver.calculate_errors(1, 5, data_set)
print("training error:", training_error, "test error:", test_error)

print("String Kernel M")
print("p = 3")
data_set = solver.perceptron(2, 3)
# calculate the errors
training_error, test_error = solver.calculate_errors(2, 3, data_set)
print("training error:", training_error, "test error:", test_error)
print("p = 4")
data_set = solver.perceptron(2, 4)
# calculate the errors
training_error, test_error = solver.calculate_errors(2, 4, data_set)
print("training error:", training_error, "test error:", test_error)
print("p = 5")
data_set = solver.perceptron(2, 5)
# calculate the errors
training_error, test_error = solver.calculate_errors(2, 5, data_set)
print("training error:", training_error, "test error:", test_error)
```