Programming Assignment 1

K-nearest neighbor (KNN) for binary classification (100 points)

Instructions

- Deadline for the assignment is July 10 2020 at 23:59:59 PDT.
- Do not import other libraries. You are only allowed to use Math, Numpy packages which are already imported in the file. DO NOT use scipy functions.
- Please use Python 3.5 or 3.6 (for full support of typing annotations). You have to make the functions' return values match the required type.
- In this programming assignment you will implement **k-Nearest Neighbours**. We have provided the bootstrap code and you are expected to complete the **classes** and **functions**.
- Download all files of PA1 from Vocareum and save in the same folder.
- Only modifications in files { `knn.py` , `utils.py` } will be accepted and graded. `test.py` can be used testing purposes on your local system for your convenience. It will not be graded on vocareum. Submit { `knn.py` , `utils.py` } on Vocareum once you are finished. Please **delete** unnecessary files before you submit your work on Vocareum.
- DO NOT CHANGE THE OUTPUT FORMAT. DO NOT MODIFY THE CODE UNLESS WE INSTRUCT YOU TO DO SO. A homework solution that mismatches the provided setup, such as format, name initializations, etc., will not be graded. It is your responsibility to make sure that your code runs well on Vocareum.

Office Hours (Zoom):

2nd July - Thursday 10:00 AM to 12:00 PM [Amulya] | Meeting Link: **https://usc.zoom.us/j/98558703974**
6th July - Monday 5:00 PM to 7:00 PM [Sowmya] | Meeting Link: **https://usc.zoom.us/j/5288651362**
7th July - Tuesday 10:30 AM 12:30 PM [Anamay] | Meeting Link: https://usc.zoom.us/j/6625239116
9th July - Thursday 10:00 AM to 12:00 PM [Amulya] | Meeting Link: **https://usc.zoom.us/j/98558703974**

Notes on distances and F-1 score

In this task, we will use four distance functions: (we removed the vector symbol for simplicity)

- Canberra Distance:

$$d(x, y) = \sum_{i=1}^{n} \frac{|x_i - y_i|}{|x_i| + |y_i|}$$

- Minkowski Distance:

$$d(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^3 \right)^{1/3}$$

- Euclidean distance:

$$d(x, y) = \sqrt{\langle x - y, x - y \rangle}$$

- Inner product distance:

$$d(x, y) = \langle x, y \rangle$$

- Gaussian kernel distance:

$$d(x, y) = -\exp(-\frac{1}{2}\langle x - y, x - y \rangle)$$

- Cosine Similarity:

$$d(x, y) = \cos(\theta) = \frac{\langle x, y \rangle}{\|x\|\|y\|}$$

An **inner product** is a generalization of the dot product. In a vector space, it is a way to multiply vectors together, with the result of this multiplication being a scalar.

Cosine Distance = 1 - Cosine Similarity

F1-score is a important metric for binary classification, as sometimes the accuracy metric has the false positive (a good example is in MLAPP book 2.2.3.1 “Example: medical diagnosis”, Page 29). We have provided a basic definition. For more you can read 5.7.2.3 from MLAPP book.

F-scores *

For a fixed threshold, one can compute a single precision and recall value. These are often combined into a single statistic called the **F score**, or **F1 score**, which is the harmonic mean of precision and recall:

$$F_1 \triangleq \frac{2}{1/P + 1/R} = \frac{2PR}{R + P} \tag{5.116}$$

Using Equation 5.115, we can write this as

$$F_1 = \frac{2\sum_{i=1}^{N} y_i \hat{y}_i}{\sum_{i=1}^{N} y_i + \sum_{i=1}^{N} \hat{y}_i} \tag{5.117}$$

Part 1.1 F-1 score and Distances

Implement the following items in utils.py

- function f1_score
- class Distances
    - function canberra_distance
    - function minkowski_distance
    - function euclidean_distance
    - function inner_product_distance
    - function gaussian_kernel_distance
    - function cosine distance

Simply follow the notes above and to finish all these functions. You are not allowed to call any packages which are already not imported. Please note that all these methods are graded individually so you can take advantage of the grading script to get partial marks for these methods instead of submitting the complete code in one shot.

In [13]:
```
def f1_score(real_labels, predicted_labels):
    """
    Information on F1 score - https://en.wikipedia.org/wiki/F1_score
    :param real_labels: List[int]
    :param predicted_labels: List[int]
    :return: float
    """

class Distances:
    @staticmethod
    def canberra_distance(point1, point2):
        """
        :param point1: List[float]
        :param point2: List[float]
        :return: float
        """

    @staticmethod
    def minkowski_distance(point1, point2):
        """
        Minkowski distance is the generalized version of Euclidean Distance
        It is also know as L-p norm (where p>=1) that you have studied in class
        For our assignment we need to take p=3
        Information on Minkowski distance - https://en.wikipedia.org/wiki/Minkowsk
i_distance
        :param point1: List[float]
        :param point2: List[float]
        :param p: int
        :return: float
        """

    @staticmethod
    def euclidean_distance(point1, point2):
        """
        :param point1: List[float]
        :param point2: List[float]
        :return: float
        """

    @staticmethod
    def inner_product_distance(point1, point2):
        """
        :param point1: List[float]
        :param point2: List[float]
        :return: float
        """

    @staticmethod
    def cosine_similarity_distance(point1, point2):
        """
       :param point1: List[float]
       :param point2: List[float]
       :return: float
       """

    @staticmethod
    def gaussian_kernel_distance(point1, point2):
        """
       :param point1: List[float]
       :param point2: List[float]
       :return: float
       """
```

Part 1.2 KNN Class

The following functions are to be implemented in knn.py:

In [14]:
```
class KNN:

    def train(self, features, labels):
        """
        In this function, features is simply training data which is a 2D list with
float values.
        For example, if the data looks like the following: Student 1 with features
age 25, grade 3.8 and labeled as 0,
        Student 2 with features age 22, grade 3.0 and labeled as 1, then the featu
re data would be
        [ [25.0, 3.8], [22.0,3.0] ] and the corresponding label would be [0,1]

        For KNN, the training process is just loading of training data. Thus, all
 you need to do in this function
        is create some local variable in KNN class to store this data so you can u
se the data in later process.
        :param features: List[List[float]]
        :param labels: List[int]
        """

    def get_k_neighbors(self, point):
        """
        This function takes one single data point and finds k-nearest neighbours i
n the training set.
        You already have your k value, distance function and you just stored all t
raining data in KNN class with the
        train function. This function needs to return a list of labels of all k ne
ighours.
        :param point: List[float]
        :return:  List[int]
        """

    def predict(self, features):
        """
        This function takes 2D list of test data points, similar to those from tra
in function. Here, you need process
        every test data point, reuse the get_k_neighbours function to find the nea
rest k neighbours for each test
        data point, find the majority of labels for these neighbours as the predic
t label for that testing data point.
        Thus, you will get N predicted label for N test data point.
        This function need to return a list of predicted labels for all test data
 points.
        :param features: List[List[float]]
        :return: List[int]
        """
```

Part 1.3 Hyperparameter Tuning

In this section, you need to implement tuning_without_scaling function of HyperparameterTuner class in utils.py. You should try different distance functions you implemented in part 1.1, and find the best k. Use k range from 1 to 30 and increment by 2. Use f1-score to compare different models.

In [15]:
```
class HyperparameterTuner:
    def __init__(self):
        self.best_k = None
        self.best_distance_function = None
        self.best_scaler = None

    def tuning_without_scaling(self, distance_funcs, x_train, y_train, x_val, y_val):
        """
        :param distance_funcs: dictionary of distance functions you must use to calculate the distance.
            Make sure you loop over all distance functions for each data point and each k value.
            You can refer to test.py file to see the format in which these functions will be
            passed by the grading script
        :param x_train: List[List[int]] training data set to train your KNN model
        :param y_train: List[int] train labels to train your KNN model
        :param x_val:  List[List[int]] Validation data set will be used on your KNN predict function to produce
            predicted labels and tune k and distance function.
        :param y_val: List[int] validation labels

        Find(tune) best k, distance_function and model (an instance of KNN) and assign to self.best_k,
        self.best_distance_function and self.best_model respectively.
        NOTE: self.best_scaler will be None

        NOTE: When there is a tie, choose model based on the following priorities:
        Then check distance function  [canberra > minkowski > euclidean > gaussian > inner_prod > cosine_dist]
        If they have same distance fuction, choose model which has a less k.
        """
```

Part 2 Data transformation

We are going to add one more step (data transformation) in the data processing part and see how it works. Sometimes, normalization plays an important role to make a machine learning model work. This link might be helpful https://en.wikipedia.org/wiki/Feature_scaling (https://en.wikipedia.org/wiki/Feature_scaling)

Here, we take two different data transformation approaches.

Normalizing the feature vector

This one is simple but some times may work well. Given a feature vector x, the normalized feature vector is given by

$$x' = \frac{x}{\sqrt{\langle x, x \rangle}}$$

If a vector is a all-zero vector, we let the normalized vector also be a all-zero vector.

Min-max scaling the feature matrix

The above normalization is data independent, that is to say, the output of the normalization function doesn't depend on rest of the training data. However, sometimes it is helpful to do data dependent normalization. One thing to note is that, when doing data dependent normalization, we can only use training data, as the test data is assumed to be unknown during training (at least for most classification tasks).

The min-max scaling works as follows: after min-max scaling, all values of training data's feature vectors are in the given range. Note that this doesn't mean the values of the validation/test data's features are all in that range, because the validation/test data may have different distribution as the training data.

Implement the functions in the classes NormalizationScaler and MinMaxScaler in utils.py

1.normalize

normalize the feature vector for each sample . For example, if the input features = [[3, 4], [1, -1], [0, 0]], the output should be [[0.6, 0.8], [0.707107, -0.707107], [0, 0]]

2.min_max_scale

normalize the feature vector for each sample . For example, if the input features = [[2, -1], [-1, 5], [0, 0]], the output should be [[1, 0], [0, 1], [0.333333, 0.16667]]

In [16]:
```
class NormalizationScaler:
    def __init__(self):
        pass

    def __call__(self, features):
        """
        Normalize features for every sample

        Example
        features = [[3, 4], [1, -1], [0, 0]]
        return [[0.6, 0.8], [0.707107, -0.707107], [0, 0]]

        :param features: List[List[float]]
        :return: List[List[float]]
        """

class MinMaxScaler:
    """
    Please follow this link to know more about min max scaling
    https://en.wikipedia.org/wiki/Feature_scaling
    You should keep some states inside the object.
    You can assume that the parameter of the first __call__
    will be the training set.

    Hint: Use a variable to check for first __call__ and only compute
            and store min/max in that case.

    Note: You may assume the parameters are valid when __call__
            is being called the first time (you can find min and max).

    Example:
        train_features = [[0, 10], [2, 0]]
        test_features = [[20, 1]]

        scaler1 = MinMaxScale()
        train_features_scaled = scaler1(train_features)
        # train_features_scaled should be equal to [[0, 1], [1, 0]]

        test_features_scaled = scaler1(test_features)
        # test_features_scaled should be equal to [[10, 0.1]]

        new_scaler = MinMaxScale() # creating a new scaler
        _ = new_scaler([[1, 1], [0, 0]]) # new trainfeatures
        test_features_scaled = new_scaler(test_features)
        # now test_features_scaled should be [[20, 1]]

    """

    def __init__(self):
        pass

    def __call__(self, features):
        """
        normalize the feature vector for each sample . For example,
        if the input features = [[2, -1], [-1, 5], [0, 0]],
        the output should be [[1, 0], [0, 1], [0.333333, 0.16667]]

        :param features: List[List[float]]
        :return: List[List[float]]
        """
```

Hyperparameter tuning with scaling

This part is similar to Part 1.3 except that before passing your trainig and validation data to KNN model to tune k and distance function, you need to create the normalized data using these two scalers to transform your data, both training and validation. Again, we will use f1-score to compare different models. Here we have 3 hyperparameters i.e. k, distance_function and scaler.

In [18]:
```
class HyperparameterTuner:
    def __init__(self):
        self.best_k = None
        self.best_distance_function = None
        self.best_scaler = None
    def tuning_with_scaling(self, distance_funcs, scaling_classes, x_train, y_trai
n, x_val, y_val):
        """
         :param distance_funcs: dictionary of distance funtions you use to calcula
te the distance. Make sure you
            loop over all distance function for each data point and each k value.
            You can refer to test.py file to see the format in which these functio
ns will be
            passed by the grading script
        :param scaling_classes: dictionary of scalers you will use to normalized y
our data.
        Refer to test.py file to check the format.
        :param x_train: List[List[int]] training data set to train your KNN model
        :param y_train: List[int] train labels to train your KNN model
        :param x_val: List[List[int]] validation data set you will use on your KNN
predict function to produce predicted
            labels and tune your k, distance function and scaler.
        :param y_val: List[int] validation labels

        Find(tune) best k, distance_funtion, scaler and model (an instance of KNN)
and assign to self.best_k,
        self.best_distance_function, self.best_scaler and self.best_model respecti
vely

        NOTE: When there is a tie, choose model based on the following priorities:
        For normalization, [min_max_scale > normalize];
        Then check distance function  [canberra > minkowski > euclidean > gaussian
> inner_prod > cosine_dist]
        If they have same distance function, choose model which has a less k.
        """
```

Use of test.py file

Please make use of test.py file to debug your code and make sure your code is running properly. After you have completed all the classes and functions mentioned above, `test.py` file will run smoothly and will show a similar output as follows (your actual output values might vary):

```
x_train shape =  (242, 14)
y_train shape =  (242,)
**Without Scaling**
k = 1
distance function = canberra

**With Scaling**
k = 23
distance function = cosine_dist
scaler = min_max_scale
```

Grading Guideline for KNN (100 points)

1. F-1 score and Distance functions: 30 points
2. MinMaxScaler and NormalizationScaler (20 points- 10 each)
3. Finding best parameters before scaling - 20 points
4. Finding best parameters after scaling - 20 points
5. Doing classification of the data - 10 points

In [ ]: