In this homework we will be implementing the K-nearest neighbor algorithm and a cross validation error estimate. We will implement a K-nearest neighbor classification algorithm using the l-p distances and l-infinity distance. You will then use cross validation to check which combination of distance function and K (the number of neighbors) will give you the best performance.

1. Download the hw4.zip folder.
2. Inside of a project (e.g. Homework4) create a package called hw4
3. Move MainHW4.java and Knn.java into this package
4. In Knn.java fill in code for and implement the Knn algorithm. You should implement the following methods (see further explanation about each method at the end of the exercise): **classify**, **findNearestNeighbors**, **getClassVoteResult, getWeightedClassVoteResult, distance, lPDistance, lInfinityDistance, calcAvgError,** and **crossValidationError.**
   1. You will also need to add properties to the class.
   2. If you think you need more methods in order to improve readability of the code - add them.
   3. Don't forget to add comments explaining your code - the more, the merrier.
5. You should, in MainHW4’s main method, find the combination of k, p, and vote function that gives you the lowest cross validation error. To do this try many different values for K (number of neighbors) and p (the l-p distance) and the two different vote functions. That is you should go over all combinations of k, p, and vote function where k is in the range {1,2,3,…,30} (unless your cross validation size makes it so that k cannot be this large), p is in the range { infinity , 1,2,3 }, and the two vote functions you can use are **getClassVoteResult** and **getWeightedClassVoteResult.**
6. You should print the cross validation error (the output of crossValidationError) of your algorithm with the best parameters for the glass data set and the cancer data set like so:

Cross validation error with K = <my\_k>, p = <my\_p>, vote function = <either weighted or uniform> for glass data is: <my\_error>

Cross validation error with K = <my\_k>, p = <my\_p>, vote function = <either weighted or uniform> for cancer data is: <my\_error>

1. For this step of the home work we will use the optimal parameters you found for the Glass data set. Implement the edited knn algorithm in methods **editedForward** and **editedBackward** as shown in class. **editedForward** should implement the training for the forwards edited knn we learned and **editedBackward** should implement the backwards edited knn training algorithm that we learned.
2. Calculate the error and the average elapsed time of calculating the error of one fold for each method. The methods you will compare are: non-edited, forwards edited, and backwards edited. In order to measure time use Java’s System.nanoTime(). More specifically: For each fold of the 10 different folds, calculate the error and also measure the time it takes to do this. Make sure to include training time in your time measurement, you should only calculate the time it takes to calculate the test error. At the end - sum the measured times and divide by 10 to get the average elapsed time for one fold.

Output to the console the both the cross validation error and the average elapsed time for the glass data set like so:

Cross validation error of non-edited knn on glass dataset is <my\_error> and the average elapsed time is <elapsed\_time \_in\_nano\_seconds>

Cross validation error of forwards-edited knn on glass dataset is <my\_error> and the average elapsed time is <elapsed\_time \_in\_nano\_seconds>

Cross validation error of backwards-edited knn on glass dataset is <my\_error> and the average elapsed time is <elapsed\_time \_in\_nano\_seconds>

The process: break glass data set into 10 parts. For each part calculate the error when the other parts are used as training. Measure the time it takes to do this. Make sure to only measure the time it takes to calculate the error and not how long it takes to train.

Sum the 10 different time measurements you took. Divide this number 10. This is the average elapsed time of calculating the error of one fold.

Explanations about the methods:

Clarification: output refers to the outcome of the method. You do not necessarily need to return what is in the output description

**buildClassifier**

input: Instances

output: should train your Knn algorithm using the configuration settings chosen by the user

Note: this method is already implemented for you. The method is implemented using a switch statement where the input to the switch statement is the private class variable *M\_MODE*. *M\_MODE* is the variable that you should set in order to tell your algorithm whether to use non-edited, forwards, or backwards training. If *M\_MODE* is set to “forward” then the method **editedForward** is used for training. If *M\_MODE* is set to “backward” then the method **editedBackward** is used for training. If *M\_MODE* is set to anything else, then **noEdit** is used for training.

**classify:**

input: an instance

output: the predicted target/class value of your algorithm for the input instance

**findNearestNeighbors:**

input: an instance, other variables you might need

output: finds the K nearest neighbors (and perhaps their distances)

**getClassVoteResult:**

input: a set of K nearest neighbors

output: should take a vote on what the class of the neighbors are and return the class value with the most votes

**getWeightedClassVoteResult**

input: a set of K nearest neighbors

output: should take a vote, weighted by each neighbor’s distance from the instance being classified, on what the class of the neighbors are and return the class value with the most votes

description: this method should be the same as getClassVoteResult except instead of giving one vote to every class, you give a vote of 1/(distance from instance)^2.

**distance**

input: two instances

output: the input instances’ distance according the distance function that your algorithm is configured to use.

**lPDistance**

input: two instances

output: the l-p distance between the two instances

note: p can be a variable of your class or you can set p some other way

**lInfinityDistance**

input: two instances

output: the l-infinity distance between two instances

**calcAvgError**

input: Instances

output: the average error on the input instances.

Description: the error you should use is the number of mistakes divided by the number of input instances.

**crossValidationError**

input: Instances

output: the cross validation error of your algorithm on the input instances

Note: Use 10 folds for the cross validation

**editedForward**

input: Instances

output: should train your Knn algorithm using the edited Knn forwards algorithm shown in class

**editedBackward**

input: instances

output: should train your Knn algorithm using the edited Knn backwards algorithm shown in class

**noEdit**

input: Instances

output: should train your Knn algorithm using the standard Knn training algorithm (that is just storing all of the training instances in memory)

Note: this method is already implemented for you