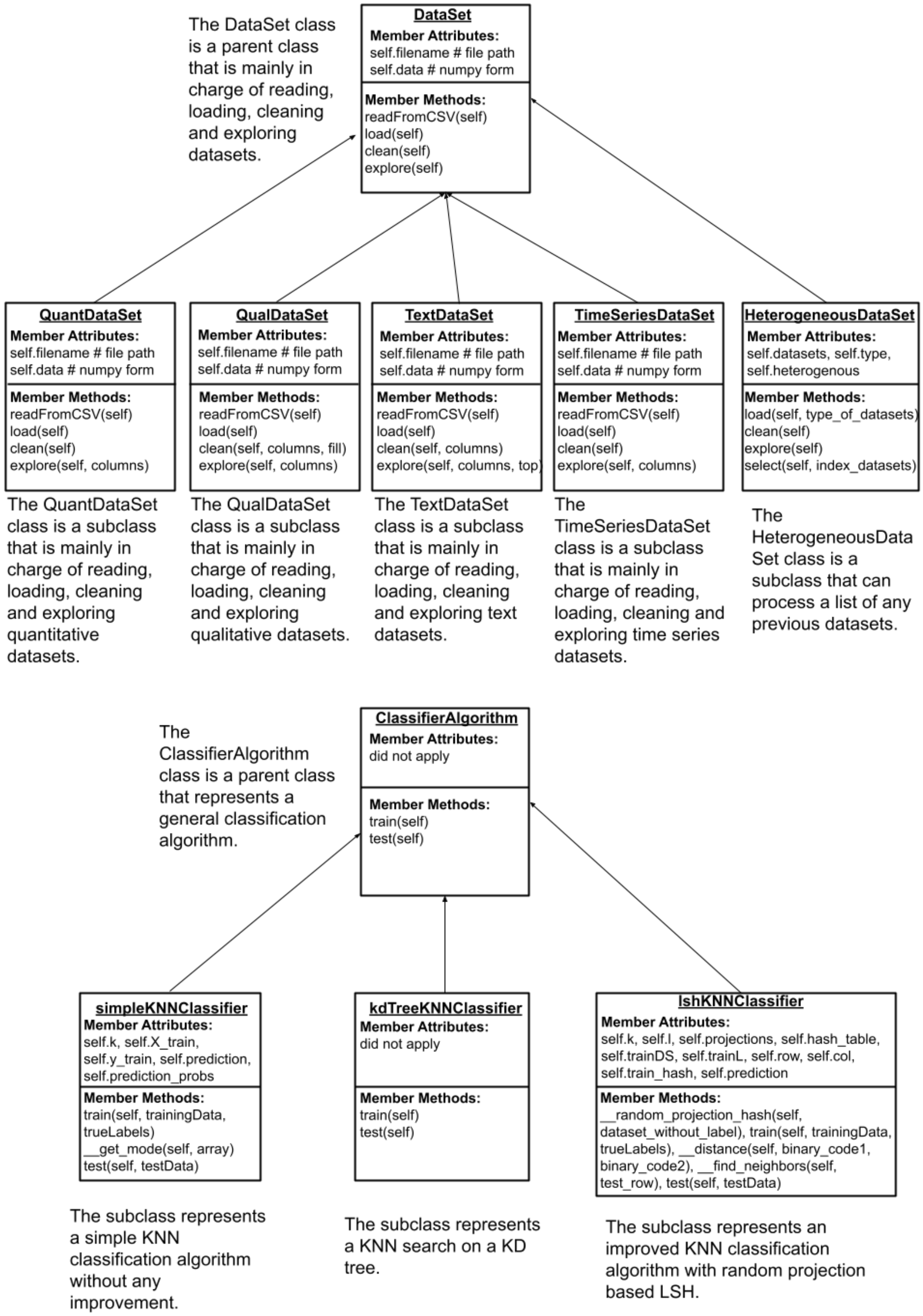
Summary of Object-oriented Toolbox Design



The Experiment class performs k-fold cross-validation and can evaluate the model performance with accuracy, confusion matrix, and ROC curves.

Experiment

Member Attributes:
self.data, self.labels,
self.classifiers, self.predLabels,
self.true, self.scores

Member Methods:
runCrossVal(self, k=5)
score(self)
__printScore(self)
confusionMatrix(self)
sortReverse(self, alist)
ROC(self, prediction_probs, trueLabels)

ROC Algorithm

1. Summary

The ROC algorithm is implemented according to the paper given. We can simply sort the probabilities reversely and update TP and FP for each predicted label. For each positive instance, we increase TP and for each negative instance, we increment FP. We also save each new ROC point to stack R. Finally, we plot the curve with the stack.

2. pseudo code or actual code

```
    def ROC(self, prediction_probs, trueLabels):
        """Produce a ROC plot.
        Implemented according to the paper (An introduction to ROC analysis) by Tom Fawcett.

        Keyword arguments:
        prediction_probs -- a list of dictionaries containing prediction probabilities
        trueLabels -- a list of true labels for the test set
        """                                                                           # step count      # space count
        legends = []                                                                  # 1 op           # y
        labels_graph = np.unique(trueLabels)                                          # n ops          # y

        for label_val in labels_graph:                                                # y ops
            probs = [d[label_val] if label_val in d.keys() else 0 for d in prediction_probs]  # n ops          # n

            # sort the prediction probabilities in decreasing order
            probs_sorted, idx_sorted = self.sortReverse(probs)                        # n*log(n) ops   # 2n
            # sort the test labels according to sorted indices
            trueLabels = list(trueLabels)                                             # 2 ops          # n
            labels_sorted = [trueLabels[i] for i in idx_sorted]                       # n ops          # n

            # initialize parameters
            TP = 0                                                                    # 1 op           # 1
            FP = 0                                                                    # 1 op           # 1
            R = []  # a list of ROC points                                            # 1 op           # n
            P = len([i for i in trueLabels if i == label_val])                        # n ops          # 1
            N = len(trueLabels) - P                                                   # 3 ops          # 1
            f_prev = float('-inf')                                                    # 1 op           # 1
            i = 0                                                                     # 1 op           # 1

            # implement the algorithm
            while i < len(labels_sorted):                                             # n ops
                f_i = probs_sorted[i]                                                 # 2 ops          # 1
                label = labels_sorted[i]                                              # 2 ops          # 1
                if f_i != f_prev:                                                     # 1 op
                    R.append((FP/N, TP/P)) # push point onto R                        # 1 op
                    f_prev = f_i                                                      # 1 op
                if label == label_val:                                                # 1 op
                    TP += 1  # true positive                                          # 1 op
                else:                                                                 # 1 op
                    FP += 1  # false positive                                         # 1 op
                i += 1                                                                # 1 op

            val_1 = (FP/N, TP/P)                                                      # 1 op           # 1
            R.append(val_1) # push (1, 1) onto R                                      # 1 op
            legends.append(f'Class {label_val}')                                      # 1 op
            plt.plot(*zip(*R))                                                        # 1 op

        plt.plot([0, 1], [0, 1], 'k--')                                               # 1 op
        plt.xlabel('False Positive Rate')                                             # 1 op
        plt.ylabel('True Positive Rate')                                              # 1 op
        plt.title('ROC')                                                              # 1 op
        plt.legend(legends)                                                           # 1 op
        plt.show()                                                                    # 1 op

# n: length of test data
# y: number of unique labels
# y is much less than n. Thus, y can be regarded as a constant when computing time complexity.

# T(n) = 1 + n + y * (nlogn + 3n + 12n + 14) + 6
# Tight-fit upperbound for T(n): O(n * log(n))

# S(n) = 6n + 2y + 9
# Tight-fit upperbound for S(n): O(n)
```

3. time complexity analysis: shown above.

kd-trees Algorithm

1. Summary

To build a KNN-KD tree, first, select its root node and select an axis. Then sort the first dimensional data and found the median. Then divide the data into root nodes, the left subtrees, and the right subtrees. The above process goes recursively until all data is divided. The search process includes a forward search and a retrospective search. If the data to be searched is smaller than the axis data, it will go to the left subtree. Otherwise, it will take the right subtree until the last point is the leaf node. Then update the points repeatedly in the search path until we find the closest point.

2. pseudo code or actual code

Algorithm 2: KNN-KD-tree building (KNNKDTB)

Input: *P, a set of training points; depth, the current depth*
Output: *treeroot, the root of the KD-tree storing P*

```
Treeroot ← root(P)
While (P != null)
   select axis = depth mod k
   sort P and select median by axis from P
   root.location ←median
   root.leftChild ← KNNKDTB(points in P before median, depth+1)
   root.rightChild ← KNNKDTB(points in P after median, depth+1)
End while
Output treeroot
```

Algorithm 3: K Nearest-neighbor Classification based on KD-tree (KNN-KD-tree)

Input: *treeroot, the root of a KD-tree; test_point, a point of test data*
Output: *predict_label, the label of the point*

```
While (root is not a leaf)
    searchPath.add(root)
    root[axis] > test_point[axis] ? Searchtree(rightChild) :
Searchtree(leftChild)
End while
For point in searchPath:
    nearest_dis ← Compute_distance (point, test_point)
    If (|point[axis] - test_point[axis]| > |test_point[axis] - root[axis]|)
       Travel (root.nextchild)
       dis ← Compute_distance (childpoint, test_point)
       If (dis < nearest_dis)
          nearest_dis ← dis
       End if
    End if
End for
predict_label←nearestPoint.label
Output predict_label
```

3. time complexity analysis

The time complexity of a KNN search on a KD-tree is O(n^(1-(1/k)) + m), where m is the number of nodes and k is the dimension of the KD-tree. In high-dimensional data, it is not as efficient as the simple KNN search.

lsh-knn Algorithm

1. Summary

First, we create an m*l random matrix A from the standard normal distribution. Then we project the training data onto l dimensional space with this random matrix A. Then convert the n*l dimensional matrix to n l-bits binary codes. For each point in the test set, we can compute the Hamming distance between it and each training point. Finally, we sort the distances and get the k neighbors.

2. pseudo code or actual code (helper functions not include)

```
    def train(self, trainingData, trueLabels):
        """Return the hash indexes of the training data.

        Keyword arguments:
        trainingData -- data to train the classifier
        trueLabels -- training labels
        """                                                        # step count          # space count
        self.trainDS = trainingData                                # 1 op                # n
        self.trainL = trueLabels                                   # 1 op                # n
        self.row, self.col = self.trainDS.shape                    # 2 ops               # n + m
        self.train_hash = self.__random_projection_hash(self.trainDS)  # n*m*l+3n+ml+2 ops  # n
        return self.train_hash                                     # 1 op

    # Dimension of training data: n * m
    # l: length of binary code

    # T(n) = n*m*l + 3n + ml + 7
    # Tight-fit upperbound: O(n * m * l)

    # S(n) = 4n + m
    # Tight-fit upperbound: O(n + m)
```

```
    def test(self, testData):
        """Return the prediction result and scores.

        Keyword arguments:
        testData -- data to test the classifier
        """                                                        # step count             # space count
        # store the prediction result
        self.prediction = []                                       # 1 op                   # n
        # transform the test data
        testData_hash = self.__random_projection_hash(testData)    # n*m*l+3n+ml+2 ops      # n
        # for each test row
        for row in testData_hash:                                  # n ops
            # get the nearest k neighbors
            neighbor = self.__find_neighbors(row)                  # nlogn+4nl+4n+k+4 ops   # k
            # count the labels
            lab, count = np.unique(neighbor, return_counts=True)   # klogk ops              # 2k
            # get the mode
            predicted_class = lab[np.argmax(count)]                # k ops                  # k
            # append the predicted label to the result list
            self.prediction.append(predicted_class)                # 1 op
        # return predicted labels
        return self.prediction                                     # 1 op

    # Dimension of test data: n * m
    # l: length of binary code
    # k: number of nearest neighbors

    # T(n) = n^2 * log(n) + 4n^2 * l + 4n^2 + nml + ml + n*k*log(k) + 2nk + 8n + 4
    # Tight-fit upperbound: O(n^2 * log(n))

    # S(n) = 2n + 4k
    # Tight-fit upperbound: O(n + k)

    # For my implementation, the time complexity does not improve compared to the simple KNN algorithm.
    # Mainly due to the inefficient sorting method. In theory, the LSH KNN algorithm should have an
    # improved time complexity compared to the simple KNN algorithm.

    # The accuracy of LSH KNN is less than the accuracy of simple KNN. This is acceptable since LSH
    # algorithms often require lots of tuning. Perhaps the current parameters still need to be adjusted
    # to get the optimal result.
```

3. time complexity analysis: shown above.