Journal of Machine Learning Research 20 (2019) 1-10 Submitted 9/19; Published 10/19

Machine Learning Project 2: Design Document

Brandon Sladek BRANDONSLADEK@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Jared Thompson J.A.THOMPSON22@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Kyle Hagerman HAGERMANKYLE96@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Ryan Hansen RYANHANSEN2222@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

1. Design Overview

In this section we present an overview of our proposed architecture for a Python application that does classification and regression using five variants of the k-Nearest-Neighbor algorithm. We discuss the implementation details of each of the core classes used in the proposed application, how we anticipate tuning the various parameters used by the algorithms, and the major design decisions made while developing the UML diagram (see Figure 1 below).

1.1 Class Descriptions

The core of our design is based upon a main driver class, ExperimentRunner. This class will instantiate a class for each algorithm implementation that extends a parent class, Algorithm. We chose to create the Algorithm class to provide common tools to each algorithm. This gives our code good reusability. Should we have to add another version of k-Nearest-Neighbor then we will only have to create a new class and add the algorithm specific methods required for its implementation.

The ExperimentRunner will also use the Datalayer, Preprocessor, and CrossValidator classes to call their respective methods. Specifically, the Datalayer class handles creating the Pandas DataFrames from the csv files, and will store each frame as a local variable for easy and efficient reference from anywhere that the Datalayer class is instantiated. The Preprocessor class will call a general method that analyzes each raw DataFrame and handles replacing or imputing missing values. The CrossValidator class will implement a 10-fold

cross validation and reserve one tenth of the dataset as a test set for each iteration. The ParameterTuner class will simply be used as a single source of parameters for each of the algorithms, fetched by calling the get_parameters() method and specifying the name of the algorithm for which we want the corresponding list of parameters to use in our experiment. The Results class will analyze each model's predictions (classifications or regressions depending on the data set) using two loss functions, 0/1 Loss and Mean Squared Error, and will print/save the results for later analysis to be presented in the final report.

1.2 Parameter Tuning

The first parameter to be tuned is the value for k for the k-Nearest-Neighbor algorithm, which will be tuned against a performance measure (the loss functions) to determine the optimal value needed for classification. It is often the case that k-NN classification performance improves with increasing k (Kulkarni and Harman (2011)). However, since this is not always the case, we want to try a wide range of k-values for each of the algorithms. For example, with regular k-NN, we want to try a small value for k, a medium value for k, and a large value for k with respect to the total number of points in the training set. In this way, we'll be able to see how the classification performance of k-NN changes with increasing k.

The second two parameters to tune are the number of centroids and medoids to be used in k-means and Partitioning Around Medoids, respectively, both of which will be iteratively tuned until we find the optimal number that no longer increases performance.

1.3 Major Decisions

We implemented a k-NN parent class from which all four remaining algorithm classes inherit in order to reuse methods that can be used across all the algorithm classes in this project. These reusable methods are k_nn() and get_distance(). We chose to include k_nn() in the parent k-NN class because some version of k-NN classification is used in all five algorithms. The primary difference is not in the logic, but in the data passed in. Each of the algorithm classes will use these methods, but the four classes that inherit from k-NN (Edited k-NN, Condensed k-NN, k-Means, and k-Medoids) will pass in alternative training sets built with child-class-specific logic. For example, the Edited k-NN class will still use the regular (parent-class) implementation of the k_nn() method, but it will run the k_nn() method with an "edited" training set built with the edit() method in the child class. In a similar manner, the k-Means class will still use the k-NN impementation of the k_nn() method, but it will run the k_nn() method with a training set that only consists of the centroids returned from the get_centroids() method in the child class. In this way, each of the child classes handles any of the additional data preprocessing that needs to be done prior to running k_nn().

We will use Euclidean distance to calculate the distance between two data points, since Euclidean distance scales better than Manhattan distance in higher dimensions (Davies and Bouldin (1979)), and is typically one of the most common distance metrics used in machine learning. However, if time permits, we might consider using other distance metrics if we find reason to believe that we can improve our results by doing so (Weinberger et al. (2006)).

In the Preprocessor class, we chose to have a general function preprocess(), which will generically handle replacing/imputing missing values and discretizing continuous data according to the actual data present in each data set. More specifically, the preprocess()

method will pass the raw (unchanged) data frame to the replace() method, which will either simply remove rows with missing values if the percentage of such rows is sufficiently small with respect to the total number of rows in the data set, or supplement (impute) missing values with randomly generated values in the domain of possible values for that attribute, if the percentage of missing values is too large to allow for removal.

Finally, as k-means and k-medoids do not necessarily terminate, we chose to implement a hard cap of 100 iterations to guarantee the algorithms terminate.

2. Experiment

This section discusses the experimental motivation and general approach for accepting or rejecting our hypothesis for the various data sets and potential k-values.

2.1 Hypothesis

We have five different algorithms we want to compare using different datasets/parameters over the course of this experiment. We want to compare performance of the various dataset-algorithm-parameter combinations through comparable loss function values. The following table depicts our hypothesis.

	Performance Rank (1 best, 5 worst)
k-NN	**2** - aggregate of the whole data set (best with mid range k)
Edited k-NN	**1** - constructed for accuracy for the most points (best with higher k)
Condensed k-NN	**5** - constructed for identifying bad data (best with higher k)
k-means	**4** - prone to outliers, and no guarantee clusters form classes (best with low k)
k-medoids	**3** - step above k-means, outlier resistant (best with low k)

2.2 Approach

First, we plan to use Python with the Pandas library to code the algorithms we wish to test (McKinney (2010)). Second, for the sake of consistency, we plan to use 10-fold cross validation to ensure our results are not particular to a specific training/test set. In probabilistic experiments, it is possible to get good or bad results due to random chance. Cross validation helps mitigate this problem by running the same process (in this case the algorithms) on various samples of the data. In particular, 10-fold cross validation is one of the most common forms of cross validation used in modern machine learning research.

3. Proposed Analysis

Our analysis will compare the performance of each algorithm on each dataset using two loss functions. We have chosen to use 0/1 Loss for the label classification datasets and Mean Squared Error for the regression datasets.

In our implementation, 0/1 Loss will sum the number of incorrect classifications for each algorithm and average over each of the 10 folds in cross validation. This measurement will be a direct measure of the accuracy of each model, and will be used to rank each accordingly.

We will also implement Mean Squared Error (MSE) for another look at the accuracy of our regression models. MSE will compare the predicted class values to the actual class values from the dataset. The closer to 0, the more accurate the model. This measure of accuracy averages over the number of points we classified from the test set and averages again over each of the 10 folds in cross validation. Between both measures of accuracy we will have a clear picture of which model predicted data points better for different k-values.

References

David L. Davies and Donald W. Bouldin. A cluster separation measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):224–227, Apr 1979. doi: 10.1109/tpami.1979.4766909.

Sanjeev Kulkarni and Gilbert Harman. *An elementary introduction to statistical learning theory*, volume 853. John Wiley & Sons, 2011.

Wes McKinney. Data structures for statistical computing in python, 2010. URL `https://pandas.pydata.org/`.

Kilian Q Weinberger, John Blitzer, and Lawrence K Saul. Distance metric learning for large margin nearest neighbor classification. *Advances in neural information processing systems*, page 1473, 2006.

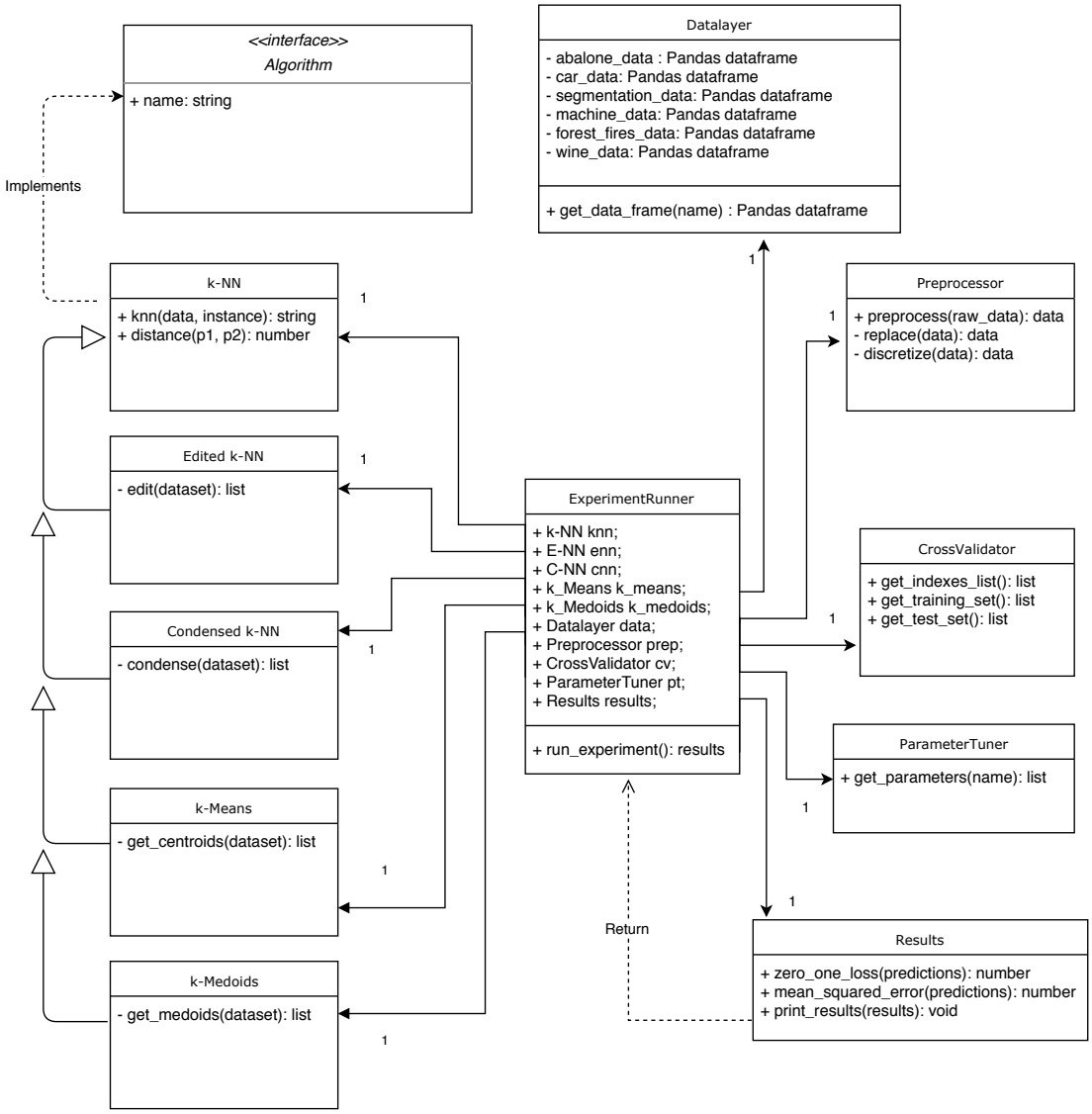

Figure 1: UML Implementation of k-NN Algorithm Suite