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

Machine Learning Project 3: 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

Biologically inspired machine learning techniques have proven to be powerful tools in discerning patterns embedded in data (Schmidhuber (2015)). Our goal for this project is to compare performance of two neural nets. In this section we present an overview of our proposed architecture for a Python application that orchestrates the implementation of both separate neural nets. We implement both a multi-layered feed forward network and a radial basis function neural net. Their performance will be compared through loss function analysis, and their convergence rates (runtime) will be compared directly. The following document outlines the details of how we will implement, tune, train, test, and compare the networks against each other.

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 neural network implementation that extends a parent class, NeuralNetwork. The ExperimentRunner will also use the DataAPI, Preprocessor, and CrossValidator classes to call their respective methods. Specifically, the DataAPI 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 DataAPI class is instantiated. See Figure 2 at the end of the document for the associated UML diagram.

The Preprocessor class will call a general method that analyzes each raw DataFrame and handles replacing or imputing missing values. We also normalize all data at this stage and handle whether a dataset is discrete classification or a continuous regression.

The CrossValidator class will implement a 10-fold cross validation dividing each preprocessed data frame into ten subsets. Each iteration of the program, nine subsets will make up the training set and the tenth will be reserved as the test set. The test set will rotate each iteration so that each subset of the data will be used as a test set. To determine these subsets, the indices of the entire data frame are shuffled to randomly sample our data set.

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. It's a one-stop shop for any global definitions we need to make for the entire experiment.

The Results class will analyze each model's predictions post training using 0/1 Loss. It will print/save the results for later analysis to be presented in the final report.

Finally, the NeuralNetwork class will be customizable based on which network we are instantiating. There are boolean flags that will be set to build the network specific hidden layers and output functions. We chose to write a general class because both networks house the same node and layer building functionality as well as both including activation functions and backpropagation methods.

1.2 Parameter Tuning

Multi-layer perceptron

1. **Number of Layers** We expect more layers to help capture more abstract patterns of the data. However, if there aren't any helpful patterns to find, adding another layer may serve only to confuse the learning process. Also, more layers increases runtime.
2. **Number of neurons per layer** We expect more neurons to identify finer details in each pattern. Again, it is possible to go overboard and force the network to try to find patterns that are not there. More neurons increases runtime, too.

Radial Basis Function Network

1. **Number of neurons** Again, we expect more neurons to help better gauge a data point, but bad data is always a concern, and runtime will increase with more nodes.
2. **Width of RBF** We plan to use gaussians as our radial basis activation functions. However, differing widths of the Gaussian may dramatically impact the training process

1.3 Major Decisions

1. **NeuralNetwork Class**
   By writing a general class to house the neural networks, we can reduce the redundancy of our code and write concise methods that will behave differently based on the active boolean flags of the instantiation.

2. **Loss Functions** - We care about two components of performance.
   Correctness - First is related to how well the machine is trained. We will gauge this by 0-1 loss, translated into an accuracy when divided by number of total attempts. It goes without saying, an accuracy close to 1 indicated the best performance.
   Convergence - Second is related to runtime. How long does it take for the algorithm to finish training. We will be comparing these directly. A higher runtime for training means it took longer for the learning algorithm to converge.

3. **Activation Functions** For simplicity, we chose to use some of the most commonly used activation functions. For the multilayered perceptron, we will use the sigmoid function. For the RBF, we will use gaussians.

4. **RBF Hidden Layer** We are required to use either edited nearest neighbors or condensed nearest neighbors to generate the centers for the gaussian activation functions for the RBF. We opted for ENN, as it runs faster and gives us a smaller data set.

5. **Backprop Objective Function** In order to do gradient descent, we need to try to optimize some objective function. We chose squared error, as it is widely used and simple to implement.

6. **MiniBatching** If we have time, we will try to incorporate minibatching to improve runtime.

2. Experiment

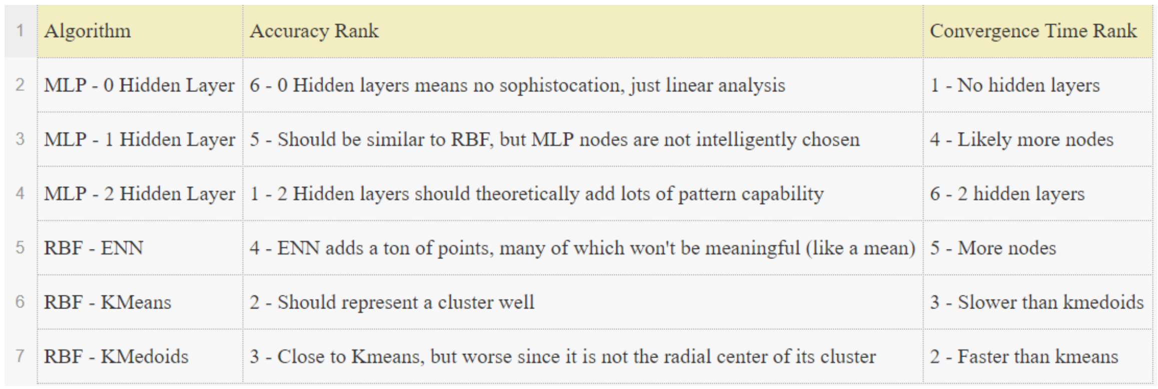
This section discusses the experimental motivation and general approach for accepting or rejecting our hypothesis for the various data sets and parameter values.

2.1 Approach

We want to test various versions of MLP and RBF on correctness and runtime. We are particularly interested in MLP with 0,1, or 2 hidden layers, and RBF where hidden node activation functions are defined by points returned from ENN, Kmeans, and Kmedoids. For implementation, we will rely heavily on the Pandas library (McKinney (2010)). As our data does not necessarily have discrete, comparable, classes, we will discritize the classes with bins and normalize our attributes. These neural networks will then be trained using backpropagation, implemented as described in class and in (Boué (2018)). Last, 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.

2.2 Hypothesis

The following table contains our prediction for performance.



Algorithm	Accuracy Rank	Convergence Time Rank
MLP - 0 Hidden Layer	6 - 0 Hidden layers means no sophistocation, just linear analysis	1 - No hidden layers
MLP - 1 Hidden Layer	5 - Should be similar to RBF, but MLP nodes are not intelligently chosen	4 - Likely more nodes
MLP - 2 Hidden Layer	1 - 2 Hidden layers should theoretically add lots of pattern capability	6 - 2 hidden layers
RBF - ENN	4 - ENN adds a ton of points, many of which won't be meaningful (like a mean)	5 - More nodes
RBF - KMeans	2 - Should represent a cluster well	3 - Slower than kmedoids
RBF - KMedoids	3 - Close to Kmeans, but worse since it is not the radial center of its cluster	2 - Faster than kmeans

Figure 1: Table depicting our performance hypothesis. Justification is included.

3. Proposed Analysis

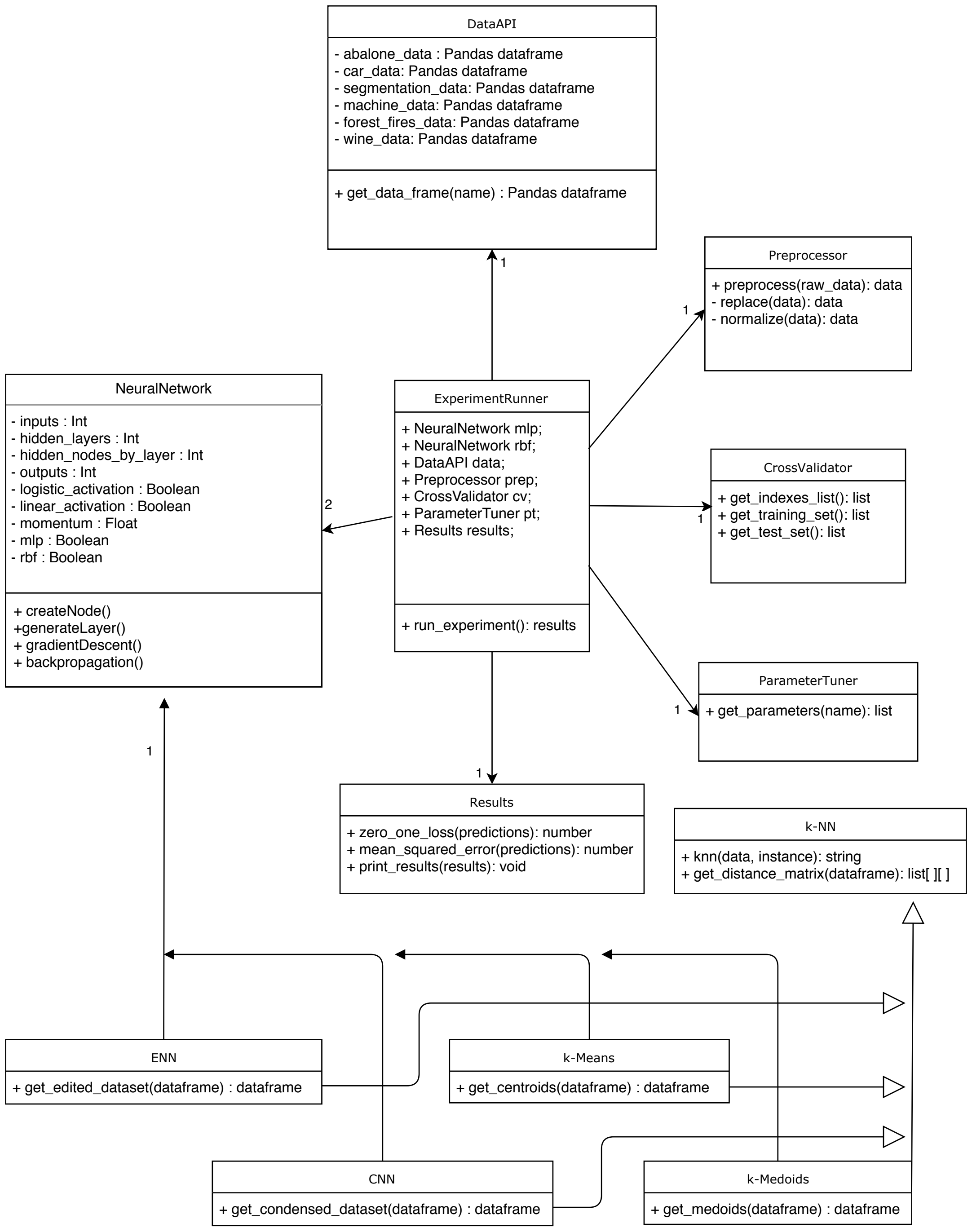
As mentioned in previous sections, our performance analysis is twofold. First, we gauge a form of correctness through 0-1 loss (accuracy) based on results from classification tests. A higher percentage correct indicates a better performance. Second, we test convergence rate. We compare this based on runtime of each algorithm. A higher runtime is a worse convergence rate, and we will compare times directly.

References

Laurent Boué. Deep learning for pedestrians: backpropagation in cnns, 2018.

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

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan 2015. ISSN 0893-6080. doi: 10.1016/j.neunet.2014.09.003. URL `http://dx.doi.org/10.1016/j.neunet.2014.09.003`.


DataAPI
- abalone_data : Pandas dataframe
- car_data: Pandas dataframe
- segmentation_data: Pandas dataframe
- machine_data: Pandas dataframe
- forest_fires_data: Pandas dataframe
- wine_data: Pandas dataframe
+ get_data_frame(name) : Pandas dataframe
1
Preprocessor
+ preprocess(raw_data): data
- replace(data): data
- normalize(data): data
1
NeuralNetwork
- inputs : Int
- hidden_layers : Int
- hidden_nodes_by_layer : Int
- outputs : Int
- logistic_activation : Boolean
- linear_activation : Boolean
- momentum : Float
- mlp : Boolean
- rbf : Boolean
+ createNode()
+generateLayer()
+ gradientDescent()
+ backpropagation()
2
ExperimentRunner
+ NeuralNetwork mlp;
+ NeuralNetwork rbf;
+ DataAPI data;
+ Preprocessor prep;
+ CrossValidator cv;
+ ParameterTuner pt;
+ Results results;
+ run_experiment(): results
CrossValidator
+ get_indexes_list(): list
+ get_training_set(): list
+ get_test_set(): list
1
ParameterTuner
+ get_parameters(name): list
1
1
Results
+ zero_one_loss(predictions): number
+ mean_squared_error(predictions): number
+ print_results(results): void
1
k-NN
+ knn(data, instance): string
+ get_distance_matrix(dataframe): list[ ][ ]
ENN
+ get_edited_dataset(dataframe) : dataframe
k-Means
+ get_centroids(dataframe) : dataframe
CNN
+ get_condensed_dataset(dataframe) : dataframe
k-Medoids
+ get_medoids(dataframe) : dataframe

(1).pdf

Figure 2: UML Implementation of Neural Network and k-Nearest Neighbor Suite v