The intent of this graduate project is to provide software that can help students using data science for categorization using basic statistical model such as *k-nearest neighbor* and *linear regression*.

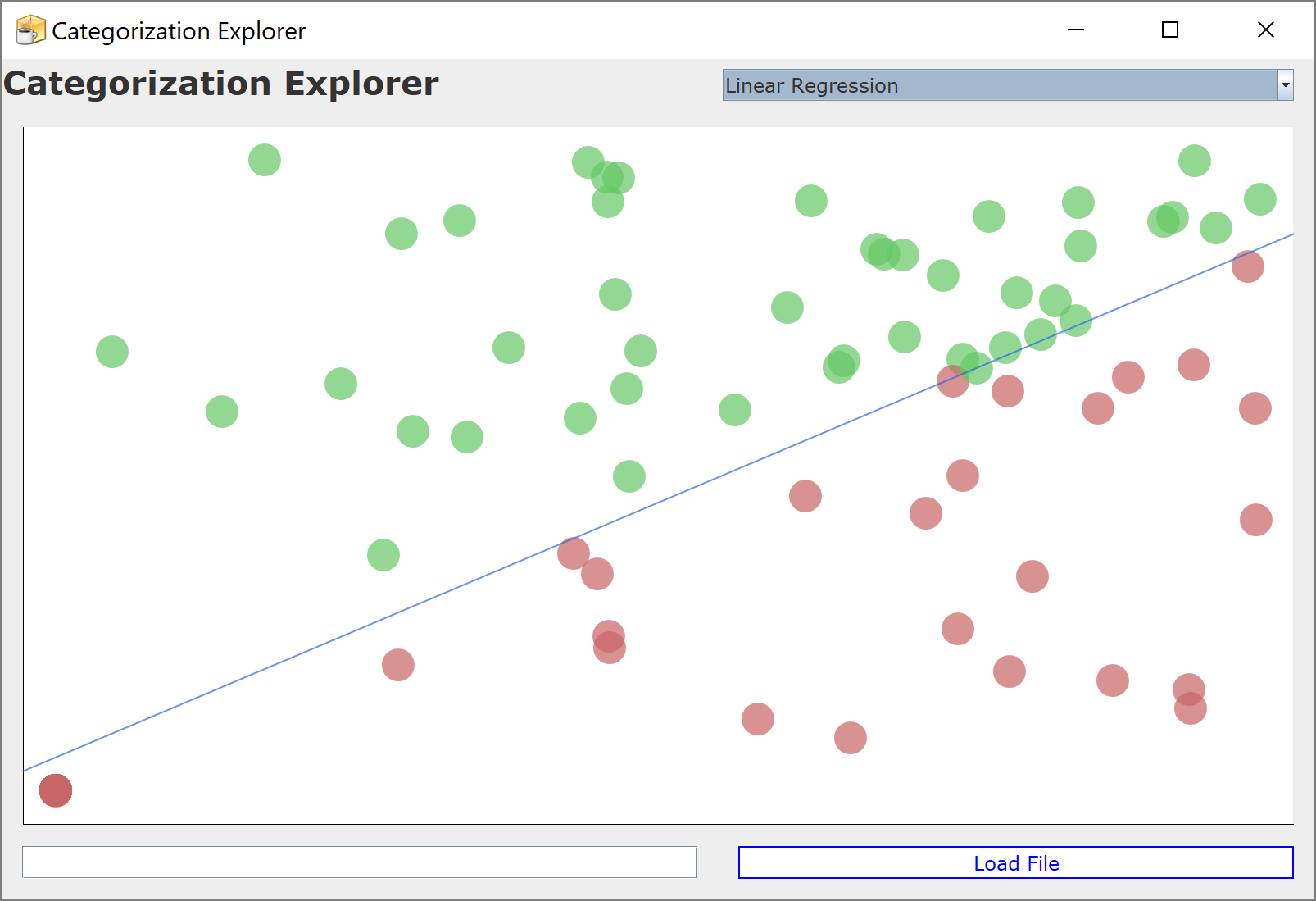
Categorization Explorer

CSCI E 10b Graduate Project

Instructor: Professor Leitner

Robert Collins

Introduction

The *Categorization Explorer* software graphs a two-dimensional dataset and a line of categorization when available (this is the line of best fit). It provides the user the opportunity to categorize the data using one of the algorithms provided. It allows input of additional custom datapoints to test. It allows loading datapoints from a file.

*screenshot at initialization*

It standardizes the data. It allows for both numeric and categorical data. It fits a model to the data, producing a categorization of the data. A color for each category is assigned. Points are colored according to category. Details about individual data points are visible on hover.

In general, anyone beginning with data science and also using Java could benefit from writing an app like this, while anyone just learning about data science would do better using standard tools like R. Nonetheless, students exploring data science benefit from a system that allows them to explore these models.

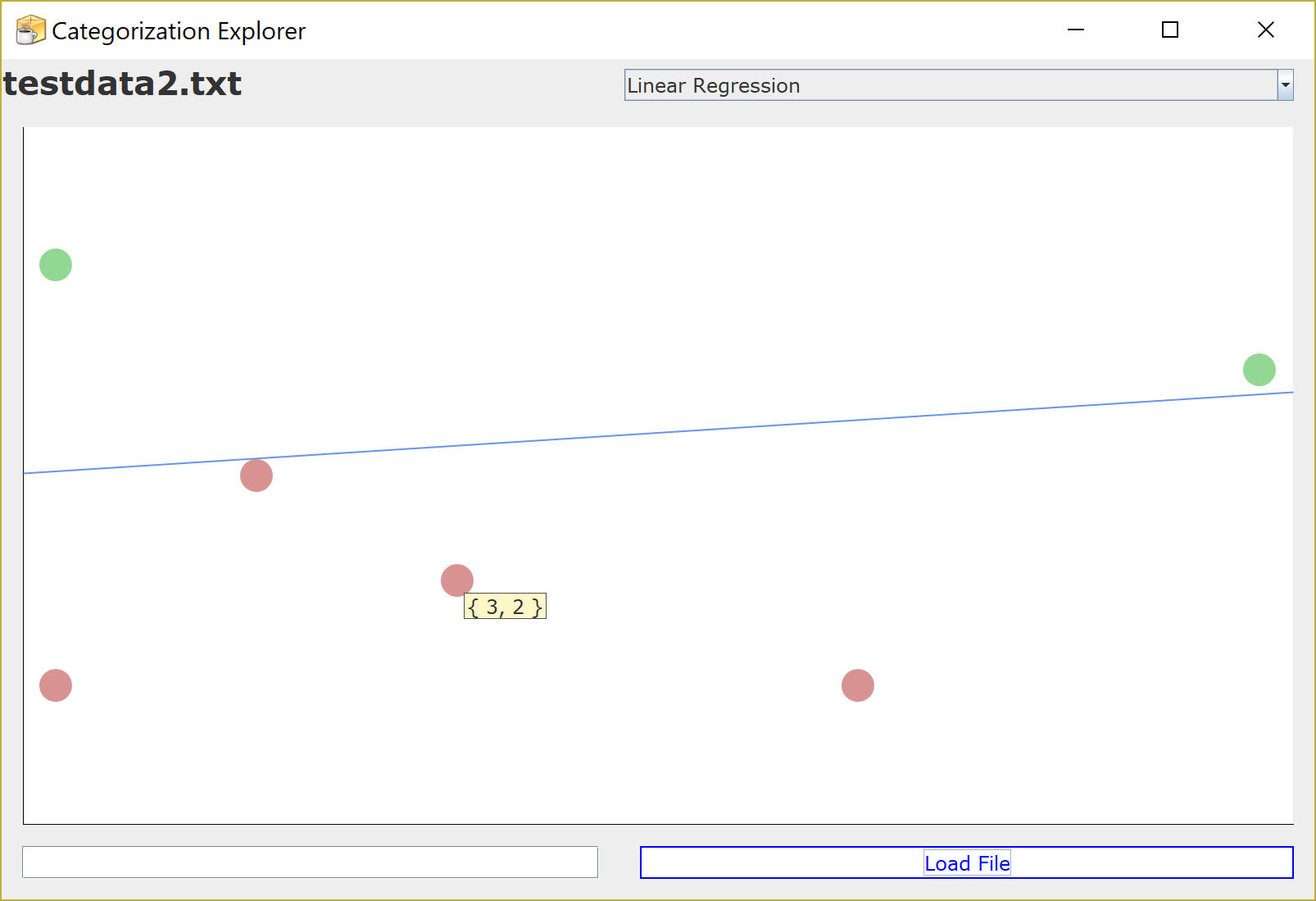
OPERATION

On startup, appropriate random data is generated. This data is standardized for analysis. The default categorization method is applied, producing a model from the provided data as the training set. The results are graphed (see *Modeling and Rendering Data*). all results are modelled under a binary categorization, such as *linear regression*. The user is then given control of the application during essentially a *wait moment* where it waits for user input.

From the *wait moment,* several options are available to the user. Each are described in a section under that title in the following pages. The user may hover over points on the data, triggering the *display a tooltip* option. The user may enter a data point, triggering the *enter test data* option. The user may opt to load a file, triggering the *load a file* option. The user may exit, triggering the *exit the program* option.

DISPLAY A TOOLTIP

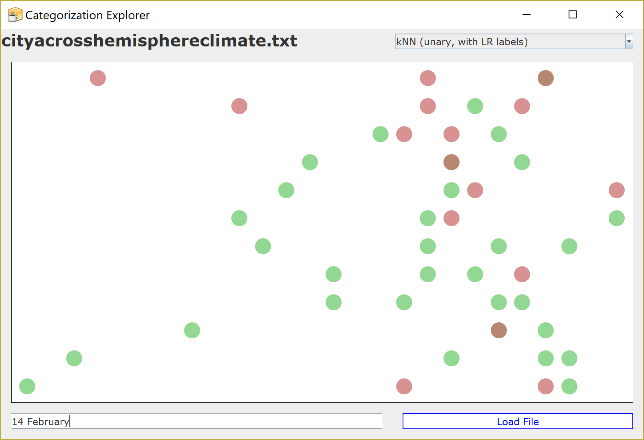
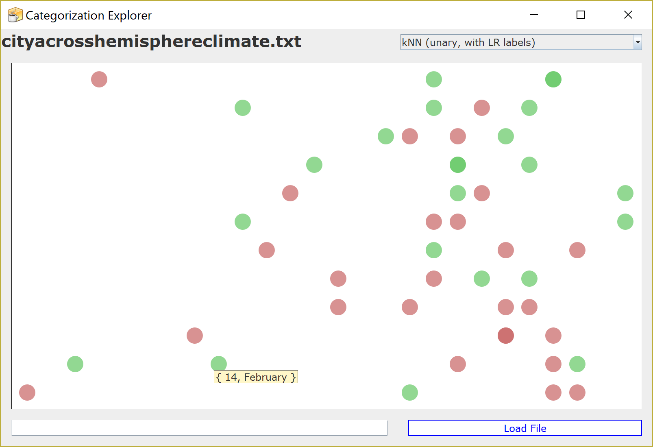
When the user hovers a data point, a tooltip is rendered:



*screenshot with mouse (not part of capture) hovering a datapoint*

ENTER TEST DATA

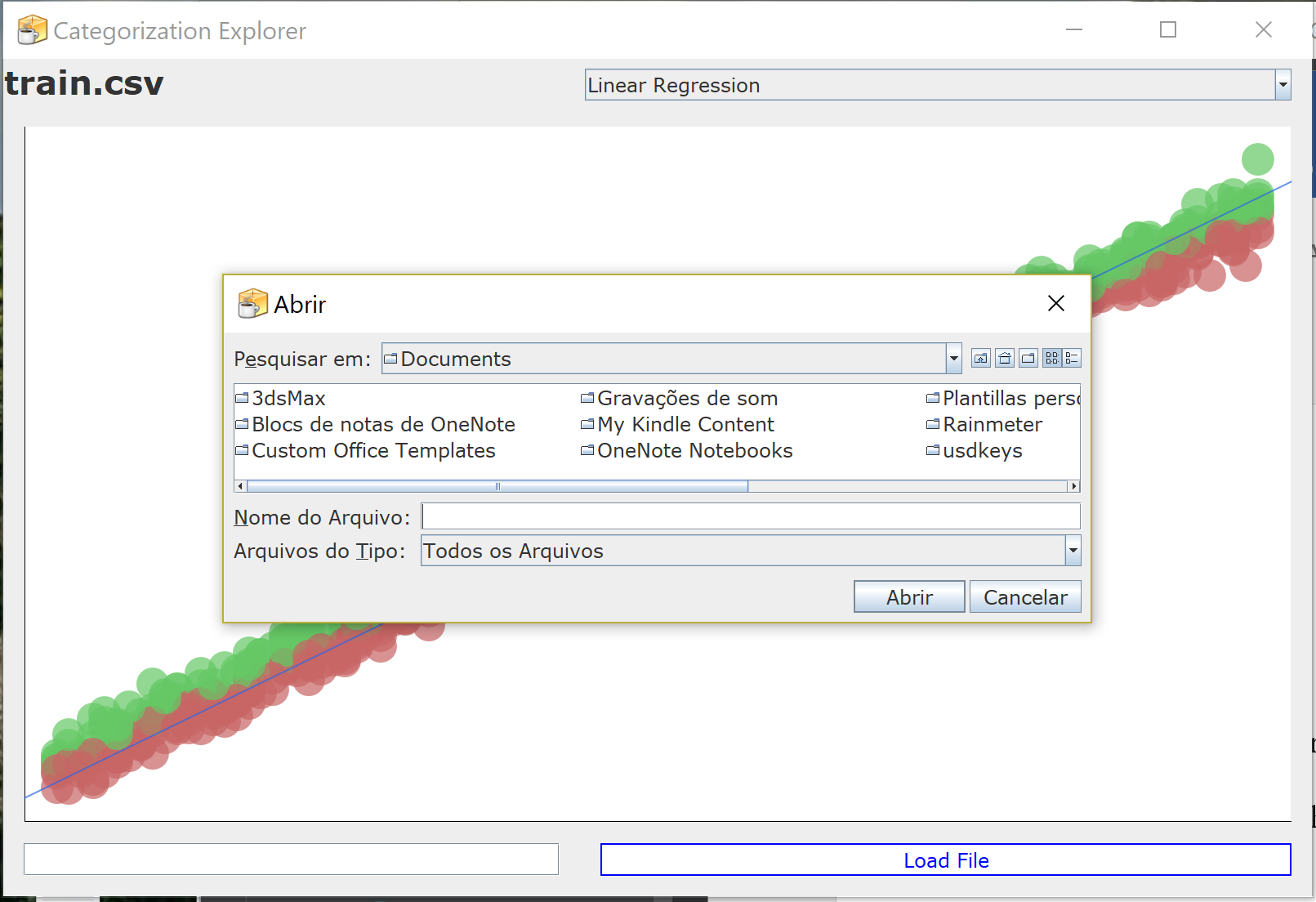
At the bottom right of the screen is a text input. This can be used to enter test data. That data will be automatically normalized and added to the data set, and the results rendered to the graph. The entered data should match the data already in the set. Here is an example of a set of average monthly temperatures at different cities in the north and south hemispheres, with the user entering a test point:



*screenshots of the process of entering test data*

LOAD A FILE

Should the user press the button labelled “Load file”, a *JFileChooser* dialog is opened, allowing them to locate and open a file:

*screenshot of the file chooser*

EXIT THE PROGRAM

When closing the window, the program appropriately terminates.

MODELING AND RENDERING DATA

When data is entered, either on start up with random data, or by user input, it is processed. This process first receives each posited data point as a String.

for(String point: items) {  
 if (Utils.*isValidPoint*(point)) {  
 *addTestPoint*(point);  
 } else {  
 System.*out*.println("invalid point" + point);  
 }  
}

*CartesianExplorer.java, lines 191-197*

The String is validated. String validation is done by comparing the string to the cartesian input specification (see *Input Specification*). If the data matches the expected input pattern, it is added as a test point to the data. This simply adds the String to a List, for later use (see *Display a Tooltip*). After that List has been changed, the list is refined to raw data in *onStringyDataChanged* (*CartesianExplorer.java*, lines 256-282), and further processing and graphing occurs. This begins by resetting the variables in the *ResultsGraph* graph component.

Continuing, a normalization of the points is rendered by the *Standardizer* class. The currently selected categorizer (see *Categorizer*)is the fed the list of data points and asked to produce a corresponding list of categories. The stringy list, the normalized list, and the categorizations list are mated together in the *buildGraphablePoints* method. The graph is then fed these graphable points (see *Graphable Points*). If a categorization line is available, the categorizer is called to return this line. The line is also passed to the graph.

GRAPHING

The graph component is a custom class extending *JPanel* and implementing *MouseMotionListener* to enable tooltips (see *Display a Tooltip*). It is a heavily customized variation of work by Rodrigo Castro and an anonymous user on *stackoverflow* (Castro, R., Anonymous, 2013). It keeps track of four collections of data:

scores = new ArrayList<>();  
tooltipData = new ArrayList<>();  
categories = new ArrayList<>();  
graphPoints = new ArrayList<>();

*ResultsGraph.java, lines 71-74*

The *scores* are the normalized representation of the datapoints. The *tooltipData* are the stringy form of the same. The *categories* are the category data from the points. The *graphPoints* are the coordinates for the graphed data adjusted to the UI *x* and *y* values of the *ResultsGraph*.

When the graph is given data, each point is processed into the first three of these lists. When the graph is rendered by the Swing UI rendering, it is done in the *paintComponent* method (lines 77-186). This method is overridden to calculate the *graphPoints* and draw the *line* if non-null and the *graphPoints*. Each point of the line and dataset are normalized against the center of their range, then renormalized to the center of the visible graph. In this way, as the graph’s size and shape changes on *resize* events, the relative positioning of the elements remains the same.

As the mouse moves over the component a check is made to see if any rendered *graphPoint* is under the current position of the mouse. If it is, its corresponding stringy representation is displayed in a tooltip (see *Display a Tooltip*).

STANDARDIZATION

The *Standardizer* class is a port of C# software by Dr. James McCaffrey, presented in *Visual Studio Magazine* (McCaffrey, J., 2014). A significant modification was made to this software: by specification we are always working on *unary* variable sets: there is a requirement of one dependent and one independent variable (see *Input Specification*). The original library produced a unique binary column for each distinct label in a categorical column. For example, given data of the form {1, hen}, {2, duck}, along with the numerical column, the original algorithm created two additional columns for the categorical data:

|  |  |  |
| --- | --- | --- |
| NUMERICALX | CATEGORICAL\_HEN | CATEGORICAL\_DUCK |
| 1 | 1 | 0 |
| 2 | 0 | 1 |

Although more valuable in general, in this simple case we need a single column, with data like the following:

|  |  |
| --- | --- |
| NUMERICALX | CATEGORICALY |
| 1 | 1 |
| 2 | 2 |

This was accomplished with a modification to the code:

private static final boolean *COLLAPSED* = true; // toggle multivariate categorization

} else if (this.subTypes[j] == "categoricalX") { // ex: x-data is 'democrat' 'republican' 'independent' 'other'  
 String v = tuple[j];  
 int index = distinctValues[j].indexOf(v); // 0, 1, 2, 3  
 if(*COLLAPSED*) {  
 result[p++] = index;  
 } //…  
} else if (this.subTypes[j] == "categoricalY") {  
 String v = tuple[j];  
 int index = distinctValues[j].indexOf(v); // 0, 1, 2, 3  
 if(*COLLAPSED*) {  
 result[p++] = index;  
 } //…  
}

*Standardizer.java, lines 14 and 155-190*

CATEGORIZATION

In support of categorization, an abstract class *Categorizer* was written that could be used to instantiate a generic categorization algorithm for use with this software given the constraints of the data specification. Each categorizer must identify itself with a title, in order to populate the *JComboBox* selector, and it must provide the following methods:

*/\*\*  
 \* load - it must be able to load data  
 \** ***@param*** *data  
 \*/*public abstract void load (List<Pair<Double>> data);  
  
*/\*\*  
 \* learn - it must be able to learn from the data  
 \*/*public abstract void learn ();  
  
*/\*\*  
 \* reset - it must be able to be reset  
 \*/*public abstract void reset ();  
  
*/\*\*  
 \* getLine -it must be able to return the categorization line it has found  
 \** ***@return*** *\*/*public abstract Line getLine ();  
  
*/\*\*  
 \* getCategorizations - it must be able to return a list of cateogrizations based on the list of data it learned  
 \** ***@return*** *\*/*public abstract List<Boolean> getCategorizations ();

*Categorizer, lines 19-45*

The software comes with two categorizers: *LinearRegression.java* and *KNN*. Each are fitted with the abstract categorizer in their own class. The implementation of linear regression was a modified port from JavaScript from the work of Assistant Arts Professor Daniel Shiffman at New York University (Shiffman, D., 2017). The primary modification was the adaptation of dynamic tuning in Frankl’s *Machine Learning in JavaScript* (Frankl, D., 2014) to implement an extremely simple convergence in the algorithm. The kNN implementation is entirely naïve, and my own work. A caveat of this exercise is that the kNN results seem dubious and further research into its proper implementation appears appropriate. I found Julong Wu’s elegant unary NN solution (Wu, J., 2016), but as it was in JAVA 8 I only used the distance methods there; these were added to *Utils*.*java*.

GENERATOR

The software initializes with *pseudo-random* data, drawn from the *Generator* class. This was entirely my own work. Two loci are situated at random across a plane of semi-random size, each are positioned in separate quadrants. A limit number of samples relative to the size of the plane is exhausted, generating data points distributed at random around each loci. A *variance* factor is ascribed to loci to help distribute to each set uniformly. This work is a sketch: it is not rigorous in generation nor in the exploration of the edges of the categorization algorithms.

INPUT SPECIFICATION

Data points are considered valid if they match a cartesian input specification.

public static final String *NUMERIC* = "[-+]?\\d+(?:\\.\\d+)?";  
public static final String *CARTESIAN\_SPLIT* = "(?:(?:, ?)| )";  
public static final String *CARTESIAN\_FORMAT* = "^(?:\\w+|"+*NUMERIC*+")"+*CARTESIAN\_SPLIT*+"(?:\\w+|"+*NUMERIC*+")$";

*/\*\*  
 \* isValidPoint - check if the string representation of a point is valid for consideration  
 \** ***@param*** *point the string representation of a point  
 \** ***@return*** *boolean  
 \*/*static boolean isValidPoint(String point) {  
 return point.matches(*CARTESIAN\_FORMAT*);  
}

*Utils.java, lines 10-12, 52-59*

A file is considered valid if each line is composed entirely of one valid point (at *CategorizationExplorer.java, processFile* method, lines 202-236).

APPENDIX

train.csv – a sample training set with a large body of correlated data.

cityacrosshemisphereclimate.txt – a small sample of weather for select western cities.

CategorizationExplorer.java – the main program to compile and run

Categorizer.java – the abstract categorizer class

Generator.java – the generator or random data

GraphablePoint.java – the UI POJO

KNN.java – the *k-nearest-neighbor* implementation

KNNCategorizer.java – the wrapper class for the kNN implementation

KNNode.java – a kNN POJO for data and label

Line.java – a primitive line class

LinearRegression.java – the *linear regression* implementation

LinearRegressionCategorizer.java – the wrapper class for the LR implementation

Pair.java – a primitive pair class

ResultsGraph.java – the graph class

Standardizer.java – the class that normalizes the data for analysis

Utils.java – the kitchen sink.

REFERENCES

Castro, R., Anonymous [<https://stackoverflow.com/users/1058210/user1058210>]. (2013, August 24). *Drawing a simple line graph in Java.* Retrieved from <https://stackoverflow.com/questions/8693342/drawing-a-simple-line-graph-in-java>

Frankl, D. (2014, December 10). *Machine Learning In JavaScript* [Scholarly project]. Retrieved from [http://cs229.stanford.edu/proj2014/David Frankl,Machine Learning In JavaScript.pdf](http://cs229.stanford.edu/proj2014/David%20Frankl,Machine%20Learning%20In%20JavaScript.pdf)

McCaffrey, J. (2014). *How To Standardize Data for Neural Networks*. Retrieved from <https://visualstudiomagazine.com/Articles/2014/01/01/How-To-Standardize-Data-for-Neural-Networks.aspx>

Shiffman, D. (2017, April). [Github repository] *Shiffman/NOC-S17-2-Intelligence-Learning*. Retrieved May 1, 2018, from <https://github.com/shiffman/NOC-S17-2-Intelligence-Learning/blob/master/week3-classification-regression/06_linear_regression_interactive/sketch.js>

Wu, J. (2016, August). DistanceUtil.java [JAVA]. Retrieved from <https://github.com/mybreeze77/Simple-KNN-with-Java8/blob/master/src/main/java/com/sample/DistanceUtil.java>