An Introduction to Classification

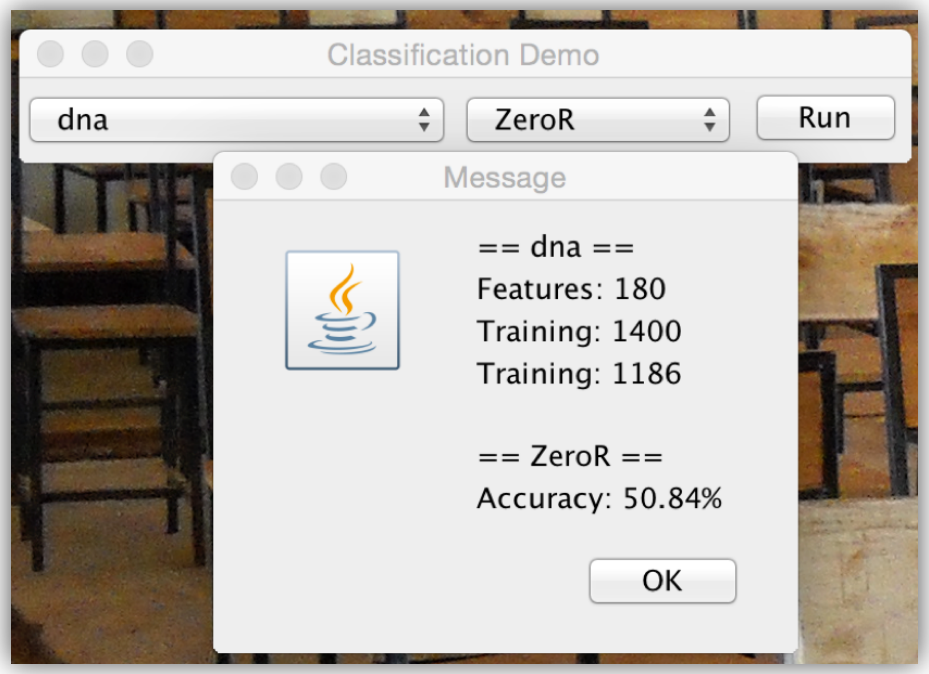
A CS2 Object-Oriented Programming Project

Motivations

- In-depth OOP end-of-course project
  - "Put all the pieces together"
- Marketability ala Machine Learning
  - Students learn basic terms, representations, pipeline
- Develop interest in CS, AI/ML, research

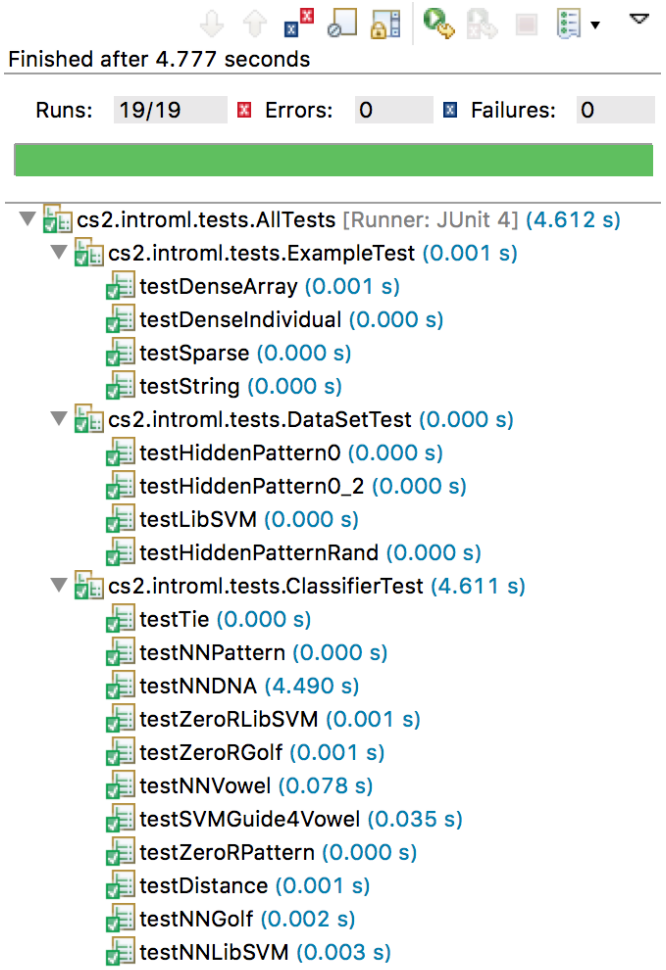
Assignment Outcomes

- Real-world application
  - "Mix and match" datasets and algorithms
- Material
  - OOP (16 classes)
  - Supervised learning
  - KR, reservoir sampling



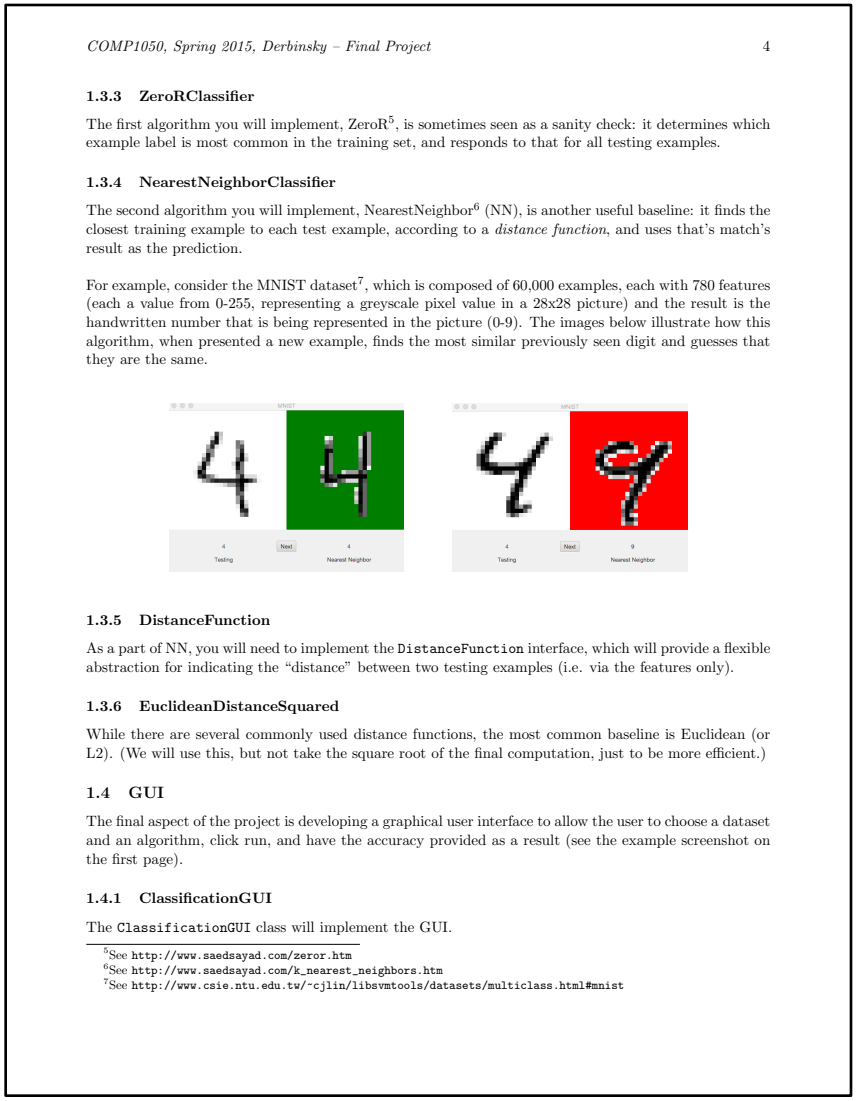
Assignment Scaffolding

- Logically grouped, independent sub-problems
- Unit testing (JUnit)
  - Real and synthetic datasets
  - Controlled via RNG
- Documentation
  - Writeup
  - JavaDoc



Part 0: Project Overview

- Big picture
- ML overview
- Classes to be implemented
  - Extra credit ideas!
- Runnable binaries



COMP1050, Spring 2015, Derbinsky – Final Project 4

1.3.3 ZeroRClassifier

The first algorithm you will implement, ZeroR[5], is sometimes seen as a sanity check: it determines which example label is most common in the training set, and responds to that for all testing examples.

1.3.4 NearestNeighborClassifier

The second algorithm you will implement, NearestNeighbor[6] (NN), is another useful baseline: it finds the closest training example to each test example, according to a *distance function*, and uses that's match's result as the prediction.

For example, consider the MNIST dataset[7], which is composed of 60,000 examples, each with 780 features (each a value from 0-255, representing a greyscale pixel value in a 28x28 picture) and the result is the handwritten number that is being represented in the picture (0-9). The images below illustrate how this algorithm, when presented a new example, finds the most similar previously seen digit and guesses that they are the same.

1.3.5 DistanceFunction

As a part of NN, you will need to implement the `DistanceFunction` interface, which will provide a flexible abstraction for indicating the "distance" between two testing examples (i.e. via the features only).

1.3.6 EuclideanDistanceSquared

While there are several commonly used distance functions, the most common baseline is Euclidean (or L2). (We will use this, but not take the square root of the final computation, just to be more efficient.)

1.4 GUI

The final aspect of the project is developing a graphical user interface to allow the user to choose a dataset and an algorithm, click run, and have the accuracy provided as a result (see the example screenshot on the first page).

1.4.1 ClassificationGUI

The `ClassificationGUI` class will implement the GUI.

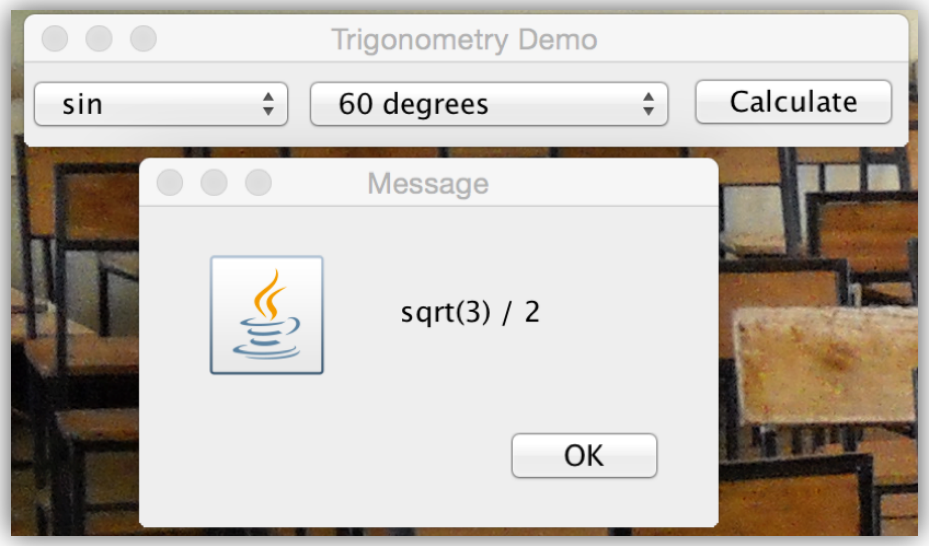
[5] See `http://www.saedsayad.com/zeror.htm`
[6] See `http://www.saedsayad.com/k_nearest_neighbors.htm`
[7] See `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#mnist`

Part 1: Algorithms

- **`TieBreaker<T>`**
  - Generic mechanism by which to fairly choose the "winner" (i.e. lowest cost) in a stream of objects of unknown size
  - **O(1)** time/space requirement (i.e. store best, cost, counter)
  - Reproducibility via **`Random`** instance
- **`EuclideanDistanceSquared`**
  - Given **`DistanceFunction`**/**`TestingExample`** interfaces
  - Find $d^2(\mathrm{e1}, \mathrm{e2})$
    - Arbitrary dimensionality = iteration
- Classifiers
  - Given abstract **`Classifier`** base class (**`train`**, **`test`**)
    - **`ZeroR`** (find most common class @ train)
    - **`NearestNeighbor`** (find closest example @ test, uses **`DistanceFunction`**)
  - Requires **`TieBreaker`**!

Part 2: GUI

- GUI elements populated with objects
  - Operation simply composes inputs
  - Unified output formatting



- Provides analogous structure for final GUI

Part 3: Data Structures

- **`DataSet`**
  - Abstract class to support iteration over examples
- **`HiddenPatternDataSet`**
  - Provides methods to construct feature-level patterns
    - {Feature=Constant} in **`LinkedHashMap`**
    - Other features are either 0 or $\mathcal{N}(\mu, \sigma^2)$ via overloaded constructor
  - Generates examples on iteration (i.e. $\mathcal{O}(d)$ space complexity)
    - Differentially generates **`Sparse`** vs. **`Dense`** example objects (given)
- **`ListDataSet`**
  - Facilitates storing examples in **`ArrayList`**
- **`LibSVMDataSet`**
  - Extends **`ListDataSet`**
  - Parses LibSVM format via **`InputStream`**, adds each to list

Part 4: Infrastructure

- Interfaces
  - **`FeatureEntry`**: (number, value)
  - **`TestingExample`**: hides classification

- **`Example`**
  - Abstract base class
    - Subclasses require implementing custom iterators (sparse/dense versions – gets at issues of representation vs. implementation)
  - **`Dense`**: array backed
  - **`Sparse`**: required to be linear in non-zero values

Final Integration

- Individual sub-parts do not depend upon each other

- However, after all four are complete the students must integrate their work, and implement 3 additional classes
  - **`DistanceFunction`** interface
  - **`Classifier`** base class
    - **`double evaluate(DataSet`** tr, **`DataSet`** t);
  - **`ClassificationGUI`** application
    - Variant of Part 3

Strengths

- Complex application
  - Possible via scaffolding
  - Extensible
    - Data formats, algorithms, regression
  - Exposure to GUI, JUnit

- OOP + AI/ML
  - Motivating
  - Works with LibSVM datasets

Weaknesses

- Complex application
  - ML content
  - Level of comfort required with a variety of aspects of Java/OOP
- Lacking design component
  - Students implement via API + unit tests
- Used only with a 3-student cohort
  - 1 loved, 2 survived
- Java-specific
  - Quite involved to port

Opportunities

- More [visual] application components
  - More satisfying, motivating
  - MNIST currently used only as a demo
    - Maybe provide source, require NN
    - Extend to ”drawn” input
- Performance measures
  - Currently simple accuracy
  - Cross-validation, confusion matrix, …

Thank You :)

Questions?