CS51 Final Project: Final Specification

Project Name

FX Predictor - an application of machine learning on financial markets

Group Members

Masakazu Tanami <tanami@g.harvard.edu>

Ben Guild <ben@benguild.com>

Ethan Brooks <ethanabrooks@gmail.com>

Kaleem Abdullah <kaleem.abdullah@gmail.com>

Brief Overview

Many people analyze historical financial charts to find patterns and predict future markets. The underlying assumption is that human beings act predictably, not randomly. As we say, "History repeats itself."

Our product will use the concepts of machine learning, using large quantities of categorized data to train our algorithm to recognize and categorize new data that is fed to it. Our training data will consist of 1-minute candles (tuples consisting of Open, High, Low, and Close prices) stretching as far back as we can acquire data. 10 years comprise 5 million candles, so we expect to handle large numbers and will build our product to scale.

The data to be classified will be candle "paths," sequences of candle data of arbitrary length (within performance limits). We will classify paths based on their attributes. At minimum, these attributes will derive from the shape of the path (this is explained in the **Mathematical Foundations** section). However, we hope to experiment with other attributes to determine which have the greatest predictive power.

Our algorithm will use the Naïve Bayes equation to classify a given path as "Up," "Down," or "Equal" – that is, a path that is about to increase its currency value, decrease it, or stay the same. An example output would be "USD/JPY in 10min: UP (78%)". This kind of prediction applies especially well to investments in binary options.

Feature List

Our final product will be an executable that takes two commands: "train" and "predict." The first command takes a path to training data as an argument and writes a data structure of computed values to disk. The second command takes a path to candle data as an argument and prints the probability of different classifications (Up, Down, or Equal) to the console.

Core features

- **Read data**: convert training data in the form of a csv file into a tuple list.
- **Classify paths**: classify each path in training data as “Up”, “Down”, or “Equal.”
- **Get path attributes**: identify attributes of a given path in training data. At minimum, this will convert a sequence of candles into an ordered sequence of “Up” and “Down” values. An example attribute would be having a first candle that is “Down.”
- **Build data structures**: construct data structures from training data. The data structure itself will perform many functions, with the objective of minimizing runtime computation for the main algorithm. Details below.
- **Write data structure**: in order to avoid retraining every time the program runs, previously trained data structures will have to write themselves to the disk.
- **Read data structure**: previously written data structures will then have to be read from the disk during runtime.
- **Predict probabilities**: Given an input path, compute probabilities for all possible classifications (“Up”, “Down”, or “Equal”).

Cool Features

- Interface with a real time data feed to continuously return predictions.
- Pipe this real time data feed into the data structure in order to periodically retrain it.
- Computationally evaluate the effectiveness of our predictions by testing the accuracy of our predictions against known outcomes.
- Test different attribute sets to determine which attributes are predictive and which are noise.
- Include as many different kinds of attributes as possible. Ideas include: time of year, volume, movements of other currencies, overall market conditions, etc.
- Handle a variety of currency types (the prototype will only predict USD/JPY).

Technical Specification

The modules of our project correspond with the core feature list. We will encapsulate the code for classification, attribution, and prediction as modules while the data structure will be an object. We will use abstraction to ensure that all processes are entirely independent from the details of specific attributes or categories. One advantage of this approach is that it will enable us to easily add new attributes and categories or remove old ones.

Details of functionality for each module are available in the **Interfaces** section below. Here I will give a general sketch of the functionality of the program as a whole.

Naturally, training must precede prediction. In response to the train command, the program reads in the file given as an argument and converts it to a tuple list. The data structure builder scans through the tuple list and tallies attributes and categories using the Attributes and Categories modules. For example, suppose we have the categories Up and Down and the attributes A, B, and C. The data structure builder will build a structure equivalent to this:

	A	B	C
Up	12	5	8
Down	2	8	23

In this example, there were 12 paths in the Up category with attribute A and 8 paths in the Down category with attribute B, etc. We will use abstraction to hide the details of these attributes and categories from the operations of the data structure. At this point, the program no longer needs individual candle values and can conserve memory by disposing them.

Next the data structure builder creates a structure, this time consisting of computed values motivated by the Naive Bayes equation. The objective of this array is to minimize the runtime computation. Details regarding these computed values are provided in the **Interfaces** section and in the **Mathematical Foundations** section.

At the conclusion of the training stage, the first data structure is discarded and the second is written to disk. In response to the predict command, the program reads the written file and uses its contents to compute probabilities using the NB Module (for Naive Bayes). Finally the program prints these results to the console.

Mathematical Foundations

Naïve Bayes takes data with attributes and computes the conditional probabilities of those attributes given a data category. The formula is:

$$\mathrm{P}(C|d) \propto \mathrm{P}(C)\prod_{i=1}^{n}\mathrm{P}(C|X_i)$$

C is the category in which we want to classify the data – in our case it is either UP, DN, or EQ.

d is the data which we want to classify – the file that the "predict" command takes as an argument.

$\mathrm{P}(C|d)$ is the conditional probability of a category given a data set. This is what we are trying to compute.

$\mathrm{P}(C|d)$ is the probability of d belonging to category C based on the training (historical) data.

$P(C|X_i)$ is the probability of a data value possessing attribute X_i given category C. It is not immediately obvious how to ascribe attributes to currency data. We will demonstrate on the following 4-candle path using trends (up, down, or equal) between one-minute intervals:

Time	Open	High	Low	Close	Attribute
4 min	120	122	119	121	UP
3 min	121	122	118	122	UP
2 min	122	122	118	120	DN
1 min	120	121	119	121	UP
0 min	121	122	120	122	UP

We ascribe the following attributes and category:

1_Min_Trend	2_Min_Trend	3_Min_Trend	4_Min_Trend	Category
UP	DN	UP	UP	UP (Based on the 0_min_trend)

A training-data matrix for ten paths of five candles each, sorted by category, might look like this:

1_Min_Trend	2_Min_Trend	3_Min_Trend	4_Min_Trend	Category
UP	UP	DN	UP	UP
UP	DN	DN	UP	UP
DN	UP	UP	UP	UP
DN	DN	UP	UP	UP
UP	UP	UP	DN	UP
DN	DN	UP	UP	UP
DN	DN	UP	DN	DN
UP	UP	DN	DN	DN
UP	DN	UP	DN	DN
DN	UP	DN	UP	DN

This data yields the following values:

$\mathrm{P}(C = \mathrm{UP}) = 6/10 = 60\%$

$\mathrm{P}(C = \mathrm{DN}) = 4/10 = 40\%$

Now take a new four-candle path that we wish to classify:

Time	Open	High	Low	Close	Attribute
4 min	116	116	113	115	DN
3 min	115	117	114	116	UP
2 min	116	117	116	117	UP
1 min	117	118	116	118	UP

$\mathrm{P}(C|X_i)$ is the percentage of paths in category C with attribute X_i. This is how we would apply this equation to each of the above four attributes using the training-data matrix above:

For Category = UP:		For Category = DN	
Equation	Value	Equation	Value
P(1_MIN_TREND=UP | C=UP)	3/6	P(1_MIN_TREND=UP | C=DN)	2/4
P(2_MIN_TREND=UP | C=UP)	3/6	P(2_MIN_TREND=UP | C=DN)	2/4
P(3_MIN_TREND=UP | C=UP)	4/6	P(3_MIN_TREND=UP | C=DN)	2/4
P(4_MIN_TREND=DN | C=UP)	5/6	P(4_MIN_TREND=DN | C=DN)	1/4

We can now calculate $\mathrm{P}(\mathrm{UP}|d)$ and $\mathrm{P}(\mathrm{DN}|d)$. These are the formulas:

$$\begin{aligned}&\mathrm{P}(\mathrm{UP}|d)\\&= \mathrm{P}(\mathrm{UP})\times\mathrm{P}(\mathrm{1\_MIN\_TREND} = \mathrm{UP}|C = \mathrm{UP})\\&\times\mathrm{P}(\mathrm{2\_MIN\_TREND} = \mathrm{UP}|C = \mathrm{UP})\\&\times\mathrm{P}(\mathrm{3\_MIN\_TREND} = \mathrm{UP}|C = \mathrm{UP})\\&\times\mathrm{P}(\mathrm{4\_MIN\_TREND} = \mathrm{DN}|C = \mathrm{UP})\end{aligned}$$

$$\begin{aligned}&\mathrm{P}(\mathrm{DN}|d)\\&= \mathrm{P}(\mathrm{DN})\times\mathrm{P}(\mathrm{1\_MIN\_TREND} = \mathrm{UP}|C = \mathrm{DN})\\&\times\mathrm{P}(\mathrm{2\_MIN\_TREND} = \mathrm{UP}|C = \mathrm{DN})\\&\times\mathrm{P}(\mathrm{3\_MIN\_TREND} = \mathrm{UP}|C = \mathrm{DN})\\&\times\mathrm{P}(\mathrm{4\_MIN\_TREND} = \mathrm{DN}|C = \mathrm{DN})\end{aligned}$$

Sometimes multiplying probability values can result in floating point underflow. To avoid this, instead of multiply probabilities in the above equations, our program will take the logarithm of the probabilities and add them. The equation becomes:

$$\mathrm{P}(C|d) \propto \mathrm{P}(C)\sum_{i=1}^{n}\log\mathrm{P}(C|X_i)$$

Next Steps

Development will be broken down into three phases: basic algorithm implementation, testing and refinement, and cool features implementation.

[Phase 1] Develop prototype code. This phase is complete when we have code that compiles and interacts with training data and user input data without crashing. This phase will be complete be complete by April 23.

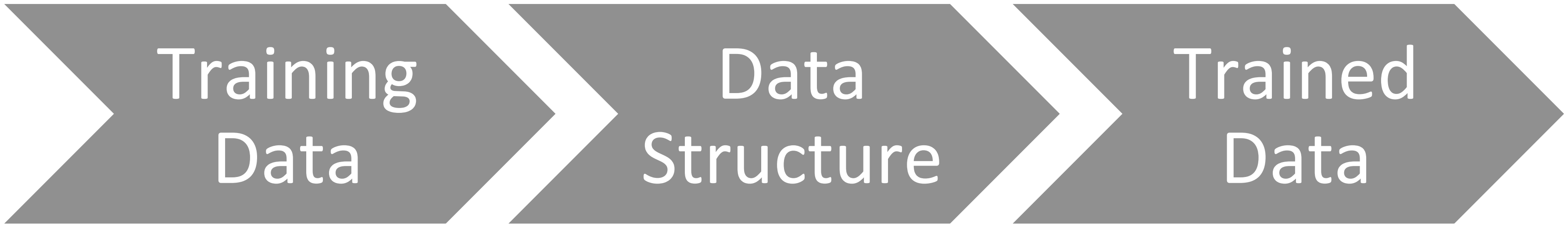
[Phase 2] The goal of this phase is to debug and optimize performance. Where possible, assertion statements will be introduced to test for invariantes. Research on google suggest, Facebook suggest, etc. may be a source of ideas for improvement. The goal is robustness and scalability. This phase will be complete by April 27.

[Phase 3] The first step will be to prioritize additional features. Alternately, everyone could just choose his favorite and work somewhat independently. This phase will by complete by the time of submission, May 1.

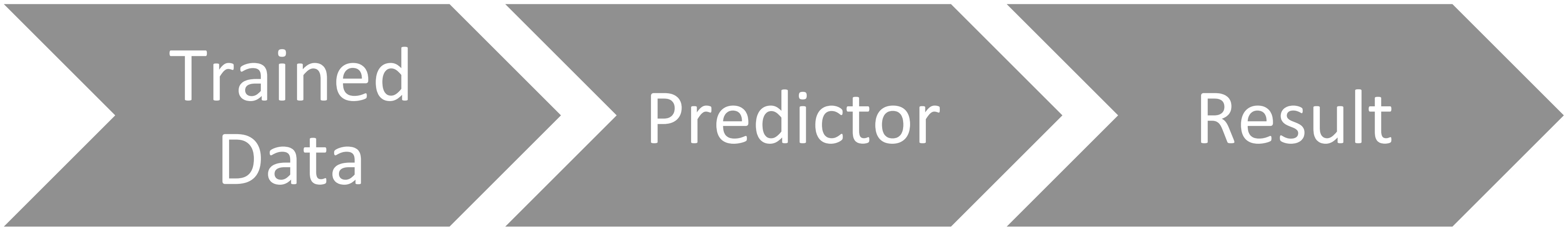
```
(* Description of Signatures and Interfaces *)

(* We will primarily use one-minute candles representing
 * changes in currency values over one minute.
 * o is opening price; h is the high price;
 * l is the low price; and c is the closing price. *)
type candle = {o: float; h: float; l: float; c: float}

(* Paths are a way to quantize candle data for the purposes
 * of classification. Length should be arbitrary within
 * perfomance limits *)
type path = candle list

(* Ids are unique identifiers assigned to each attribute for
 * the purposes of abstraction. Once the program knows an
 * attribute's id, it doesn't need to know anything about
 * the details of that attribute. *)
type id = int

(* Categories correspond to the thing we are trying to predict.
 * For example, if our categories are Up and Down, then we are
 * trying to predict whether currency will go Up or Down
 * according to some meaningful criterion. Categories are
 * completely encapsulated so that the rest of the program
 * doesn't need to know details about the categories themselves.
 * The only requirement is that the get function can map any
 * given path to the category space. *)
module type CATEGORIES =
  sig

    type category

    (* A list of all categories as defined above *)
    val all : category list

    (* Categorizes given path *)
    val get : path -> category

  end

(* Attributes are qualities that paths can possess (e.g.
 * whether the first candle closes higher than it opened).
 * The details are completely encapsulated, so long as the
 * get function can map any given path to an id list (and
 * that list may be empty). Modules of this type will be
 * defined as functors that take modules of type TESTS as
 * arguments. *)
module type ATTRIBUTES =
  sig

    (* Returns list of unique ids corresponding to each
     * attribute that given path possesses *)
    val get : path -> id list

  end

(* This class is the data structure to hold the trained NB model with
 * computed values ready-made for computation. It provides method to add
 * a data record, which updates the model with attribute values and category
 * of this data record. It provides methods to transform the model to
 * and from a text representation for offline storage and retrieval. *)
class type nb_model_i =
object

  (* Add a data record to the model; *)
  method addDataRecord : data_record -> unit

  (* Rebuild the model from a string *)
  method load_from_text : string -> unit

  (* Convert data model to a string form for storage *)
  method get_text_representation : string
  (* Log P(C) in the Naive Bayes equation; equivalently, the
   * percent of training paths in category c *)
  method log_cat_prior_prob : CATEGORIES.category -> float

  (* mathematically equal to P(X) where X is the
   * vector of attributes belonging to the input path;
   * normalizes the result of P(C) ¿ (i=1..n) P(Xi|C) *)
  method prob_x : float

  (* given a list of attribute ids i, returns
   * [Log P(X1|C),Log P(X2|C),...,Log P(Xi|C),...] *)
  method log_cat_cond_prob :
    id list -> string -> float list

end

(* This module is designed to be passed in as an argument in
 * functors defining modules of type ATTRIBUTES. This is for
 * the purpose of code reuse, since the mechanism of testing
 * for each attribute and putting the associated ids in a list
 * will always be the same. Generally TESTS will actually define
 * the meat of a given set of attributes by providing the
 * functions which map given paths to the attribute space. *)
module type TESTS =
  sig

    (* List of tests to determine attributes of a given path;
     * returns a list of unique ids associated with attributes
     * that the path possesses *)
    val tests : (path -> id option) list

  end

(* Responsible for actual computation of probabilities. As much
 * as possible, computations are already done by the data
 * structure to minimize runtime computation. *)
module type NB =
  sig

    (* The Naive Bayes equation P(C) ¿ (i=1..n) P(Xi|C) *)
    val event : path -> CATEGORIES.category -> data_structure -> float

    (* Probability P(C|d); normalizes the result of the previous function *)
    val probability : CATEGORIES.category -> path -> data_structure -> float

    (* Returns the highest probability of all categories;
     * designed for arbitrary positive number of categories *)
    val prediction : path -> data_structure -> CATEGORIES.category

  end
```

Process Overview

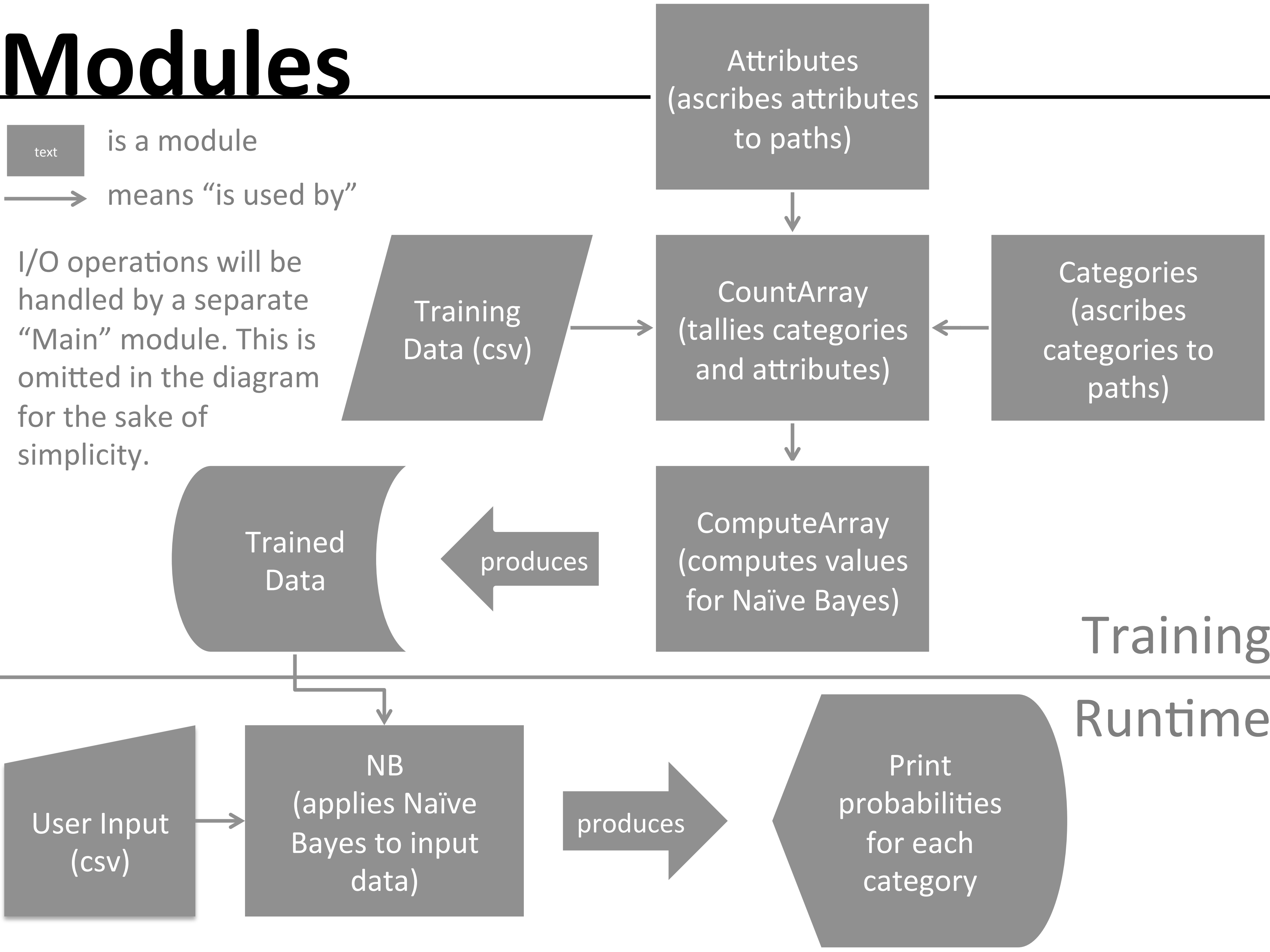
Training



Runtime



Modules


text
is a module
means "is used by"
Attributes (ascribes attributes to paths)
Training Data (csv)
CountArray (tallies categories and attributes)
Categories (ascribes categories to paths)
Trained Data
produces
ComputeArray (computes values for Naïve Bayes)
Training
Runtime
User Input (csv)
NB (applies Naïve Bayes to input data)
produces
Print probabilities for each category

I/O operations will be handled by a separate "Main" module. This is omitted in the diagram for the sake of simplicity.