Big Data Analytics Programming
Assignment 1: *Incremental classifiers*

Pieter Robberechts
`pieter.robberechts@kuleuven.be`

Collaboration policy
Projects are independent: no working together! You must come up with how to solve the problem independently. Do not discuss specifics of how you structure your solution, etc. You cannot share solution ideas, pseudocode, code, reports, etc. You cannot use code that is available online. You cannot look up answers to the problems online. If you are unsure about the policy, ask the professor in charge or the TAs.

1 Incremental classifiers [30 pts]

The goal of this assignment is to build and evaluate two incremental classifier learning algorithms. In contrast to standard classifier learning techniques that typically learn by iterating multiple times over a finite (generally small) number of training examples, incremental classifiers build a model from a continuous stream of training examples. The learned model can be queried at any time during training to make predictions. However, the predictive accuracy is expected to increase with the number of training examples that have been processed. Incremental classifiers are well suited to problems with a very large or even infinite number of training examples. In this assignment, we will consider **binary classification problems** where the class label is either 0 or 1.

You will implement two incremental classifiers in `Java`: Perceptron and Very Fast Decision Trees (VFDT).

1.1 Perceptron [10 pts]

The perceptron classifier dates from 1958 and is one of the first artificial neural networks, laying the foundations of the current deep neural networks. For this assignment, you will implement incremental learning with the delta rule and stochastic gradient descent. This classifier should be able to handle both binary and continuous features. All the continuous features are normalized to have values between 0 and 1.

Class stubs for the implementation are provided. Section 1.4 gives more information about them.

Attention! The perceptron expects the classes to be 1 and -1, while the implementation provides them as 1 and 0. Keep this in mind when writing the perceptron code.

1.2 Very fast decision tree [16 pts]

P. Domingos and G. Hulten have proposed a incremental decision tree learning algorithm called *VFDT* for *Very Fast Decision Tree Learner*. The algorithm to build the decision tree is described in [**?**] for multi-valued features.

The procedure in [**?**] (Table 1) describes how to initialize and update a decision tree. Section 3 describes some refinements to the algorithm. You should at least implement *ties* and *G computation*.

Class stubs for the implementation are provided. Section 1.4 gives more information about them.

1.3 Data

The data that you will use to test your classifiers is synthetically generated by randomly sampling a model. There are 200 features and 200,400,000 examples. The features can be binary or continuous in the range [0,1]. The basic version of VFDT, as described by the algorithm in the paper, cannot handle continuous variables. Therefore a separate dataset is provided where the continuous variables are discretized to 5 values.

The data was generated by sampling a randomly generated decision tree. This is similar to what is done in the VFDT paper. There are two versions of the data: clean and noisy. The clean data is the data which was generated by sampling a model. The noisy data is the same data where some values are randomly changed. To simulate concept drift, the model to generate examples is switched in the middle of the data generation.

The full datasets are together 322GB, so you should not copy it to your machine. You can access it directly on the departmental machines in the directory `/cw/bdap/assignment1/data/`

1.4 Implementation

`code/` contains class stubs of the incremental classifiers, complete implementations of some helper classes, and sanity checks. Read them and understand them carefully.

Class stubs:

- `Perceptron.java` is the class stub for the perceptron model
- `Vfdt.java` is the class stub for the VFDT algorithm.
- `VfdtNode.java` is the data structure of a decision tree that is used by VFDT.

These classes are incomplete. In order to make a functional algorithm, you must at least complete the methods that are tagged with “THIS METHOD IS REQUIRED”. You should not change methods that are tagged with “DO NOT CHANGE THIS METHOD”. The headers of all the methods, including the constructor, should not be changed. The rest of the code may be changed as long as all the required methods work as expected.

Helper classes:

- `Data.java` and `Example.java` deal with reading in the data.

- `IncrementalLearner.java` is the superclass of `Vfdt` and `Perceptron`

These classes are complete and are not to be changed.

Sanity checks:

- `Perceptron.java` is the class stub for the perceptron model.
- `Vfdt.java` is the class stub for the VFDT algorithm.

Sanity checks serve to check the interface and minimal functionality of your code. Submitted code that does not pass all the sanity checks will not be graded. You can run the tests with `make check_pc` and `make check_vfdt`.

1.5 Reading and Writing models

Your code needs to be able to write out (partially) learned models to a file, to read models from a file, and to continue learning with them. This is good practice in machine learning. Most importantly, it prevents having to learn a model every time that you want to use it. Furthermore, learning can be continued after a crash, if you periodically back up the partially learned model. It is also useful for testing; a manually created corner case can just be read in. This will be used for grading, so it is very important that you get the reading and writing methods right.

The following paragraphs describe the file formats used to store the models. Some examples can be found in `code/models/`.

Perceptron file format

The weights (and bias) are the only parameters used by the perceptron. The model is described by one line that lists these parameters (w_0 to w_d), separated by a space. For example, the model with $f(x) = 0.5 + 1.5x_1 - 2.0x_2 + 0.1x_3 - 1.8x_4$ has the following description:

```
 0.5 1.5 -2.0 0.1 -1.8
```

VFDT file format

The VFDT file needs to describe the tree. The first line of the file gives the size of the tree (the number of nodes). All the other lines are nodes. Every node has an identifier, which is the line that it is on (not counting the first line with the size for the tree). These identifiers are used to point to the children of a node. The nodes appear bottom-up (children before parents) in the file. Leafs and decision nodes are each described in a specific way.

A leaf node has to specify the possible features that it can split on and the instance counts n_{ijk} of those features. It uses a dense matrix for the possible split features and a sparse matrix for the instance counts. The instance counts matrix stores n_{ijk}: the number of examples where feature i has value j and the class is k for all possible values of i,j and k, with i restricted to the possible split features. The sparse matrix represents all non-zero counts n as $i : j : k : n$. Consider data with 3 binary features. A leaf node with identifier 5 that contains 4 examples: $\{[1,0,1]:1,[1,0,1]:1,[1,0,0]:0,[1,0,0]:1,\}$ that can still split on feature 2 looks like this:

```
 5 L pf:[2,] nijk:[2:1:1:2,2:0:0:1,2:0:1:1,]
```

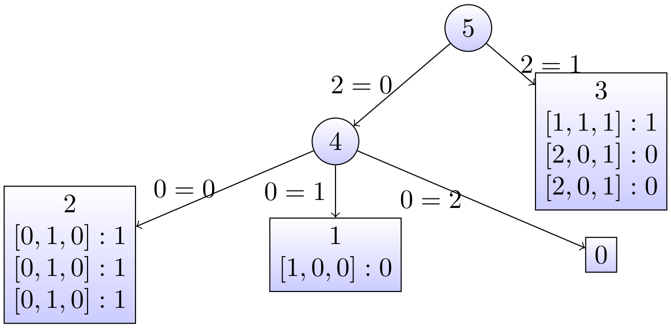
A decision node has to specify on which feature it splits and its children. The children are represented by an array of node identifiers, where the index of the array is the value of the split feature. A decision node with identifier 7 that splits on feature 2 and has 2 children: the node with identifier 5 for $2 = 0$ and the node with identifier 3 for $2 = 1$, looks like this:

```
 7 D f:2 ch:[5,3,]
```

This is an example of a complete file:

```
6
0 L pf:[1,] nijk:[]
1 L pf:[1,] nijk:[1:0:0:1,]
2 L pf:[1,] nijk:[1:1:1:3,]
3 L pf:[0,1,] nijk:[0:1:1:1,0:2:0:2,1:1:1:1,1:0:0:2,]
4 D f:0 ch:[2,1,0,]
5 D f:2 ch:[4,3,]
```

This represents the following tree, which has the nodes labeled with its identifiers



1.6 Compiling and Running code

A Makefile is provided that can be used for compiling, testing and running the code. Make sure that you read and understand this file. You can adjust it for running experiments if you wish.

You can compile the code with `make Perceptron.class` and `make Vfdt.class`. You can test the code locally on a subset of the data with `make pc_small` and `make vfdt_small`. From the departmental machines, you can run the code with default parameters on all the clean data with `make pc_clean` and `make vfdt_clean` and on all the noisy data with `make pc_noise` and `make vfdt_noise`.

1.7 Report [4 pts]

Part of this course's goal is to gain insights into the interplay between large amounts of data and algorithms. To this end, you should conduct several experiments to explore issues such as efficiency, scalability, effects of parameter values, etc., to help gain insight into how the algorithms you implemented work.

Please focus on posing an experimental question (e.g., What is the effect of varying parameter X), designing an experiment to test that question, and then providing a summary of the finding (e.g., increasing the value of parameter X causes ...) Any valuable insight on your work will be graded accordingly. You should write a small report of at most two pages (excluding Tables or Figures) explaining your experiments and findings.

One experiment that you must perform and include in your report is generating a *learning curve*, which visualizes how a classifier's accuracy varies as a function of the number of training examples. Figure 1 gives an example of a learning curve for a logistic regression classifier on the UCI `zoo` dataset. The learning curve in your report should show the performance of the Perceptron and VFDT classifiers. Failing to use the full dataset to generate these learning curves will incur a substantial penalty.

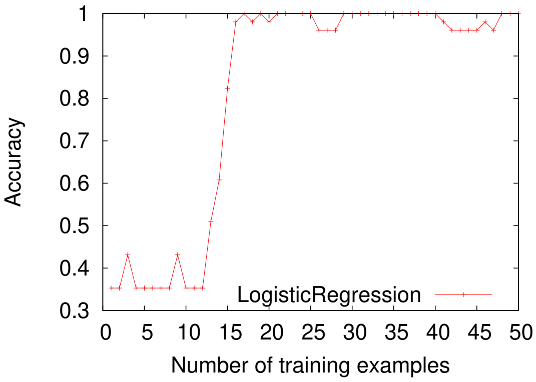


Figure 1: Example of a learning curve on `zoo`.

In addition, the report can include a maximum of half a page extra describing any problems (that is, bugs) in your code and what may have caused them as well as any special points about your implementation.

You should also submit the raw output ("out.clean.pc.acc", "out.noise.pc.acc", "out.clean.vfdt.acc" and "out.noise.vfdt.acc") of learning with the default parameters.

References

[DH00] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80. ACM, 2000.

2 Important remarks about grading and submitting the assignment

Deadline The assignment should be handed in on Toledo before **December 22nd, 9am**. There will be a 10% penalty per day, starting from the due day.

Deliverables You must upload an archive (`.zip` or `.tar.gz`) containing the report (as `.pdf`) and a folder with all the source files using the file hierarchy below:

```
assignment1/
├── report.pdf
├── src/
│   ├── Perceptron.java
│   ├── Vfdt.java
│   └── VfdtNode.java
└── out/
    ├── out.clean.pc.acc
    ├── out.noise.pc.acc
    ├── out.clean.vfdt.acc
    └── out.noise.vfdt.acc
```

The `.java` files should contain complete source code for the perceptron and the VFDT algorithms. Files included in the template that were not modified do not have to be included in your submission. The `out.*.*.acc` should contain the data for the learning curve that was generated using the learners with the default parameters from the Makefile.

Automated grading Automated tests are used for grading. Therefore you must follow these instructions:

- Use the exact filenames as specified in the assignment.
- Do not change the provided interface/skeletons, this includes the constructor.
- Your code must run on the departmental machines, this can be done over ssh (see documents of exercise session 1).
- Never use absolute paths.

If you fail to follow these instructions, you will lose points.

Running experiments This is a class about big data, so running experiments could take hours or even multiple days. You can do this easily on the departmental machines: it takes several minutes to start the experiments and then you check back later to see the results. Failure to run and report results on the full data set will result in a substantial point reduction.

Good luck!