

CS-IS-2010-1: Midsemester Project Report

Santrippta Sharma

April 4, 2024

1 Problem Statement & Requirements

1.1 Problem Statement

Develop a classification model over the Iris Dataset and store the model. Then write a Haskell code to restore the model, input new data (based on the four features of the iris) and generate a prediction in real time.

1. Input: The Iris Dataset.
2. Output: Real-time prediction for the class of the flower of a new data row containing the sepal width, petal width, sepal length and petal length.
3. Method: You can use Python to train and save a classification model (SVM or NN). However, restoring the model and the real-time prediction of a new data row has to be written only in Haskell.

1.2 Requirements

Based on this problem statement, the project's requirements can be formulated as follows:

- The system should be able to load a trained classification model.
- Given any new data point of the same format as the iris dataset by the user, the system should be able to use the loaded model to make a prediction on this data and report it to the user.
- The inference part of the system should be as performant as possible, since the problem description suggests that the application domain is a real-time system.

2 Specification

Generalising from the requirements, we offer the following featureset:

1. Parsing capabilities to restore the shape, weights, and biases of **an arbitrary** numerical FFN-based classifier from a file.
2. Interactive & batched modes of performing inference.

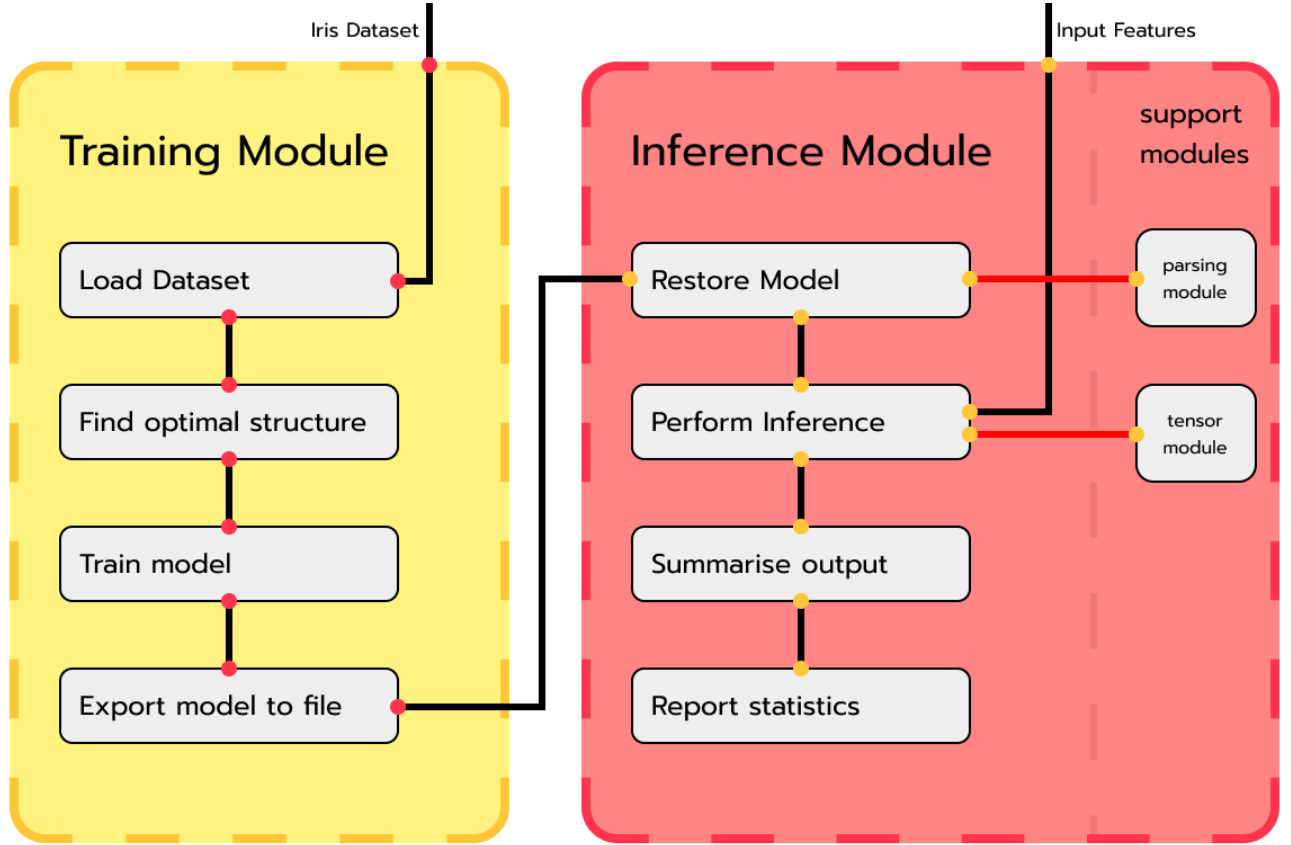


Figure 1: the high-level architecture

- **Interactive:** The user enters a single new datapoint and gets predictions for it in real-time. Structured as a Read-Eval-Print-Loop (REPL).
- **Batched:** The user provides many datapoints in a csv file, and receives predictions for each point. Additionally, the system reports aggregate statistics.

To briefly expand on point 1, since the type of model (FFN) is known, we can generalise the system to work with any classifier following that architecture, since this is independent of the particular dataset used, as long as that dataset only contains numeric feature. This allows the system to be reused for similar tasks beyond the Iris dataset.

3 Design & Architecture

Since we are only concerned with running the inference for a new row for this dataset, we train our classification models separately (for testing purposes) before runtime. At runtime, we restore the structure, weights, and biases of the model, and push the given feature vector through the model in order to arrive at the inferred output.

Then, the system can broadly be divided into two modules, the training module & the inference module. These operate independently from each other, with the only interface between them being the inference module taking the training module's output (model structure, weights, biases) as its input, which is then used to reconstruct the model. Figure 1 illustrates the general flow of this design.

First, we look at the common thread between the two modules, the file format for sharing the trained model parameters.

3.1 Trained Model Format

For the prototyping stage, we opt to use a plaintext format for the trained model parameters, since this allows easier debugging and editing.

We know that a feedforward neural network is comprised of several layers of neurons, with each layer (besides the input layer) having a matrix of weights & a vector of biases associated with itself. Further, using the propagation formula $Wx + b$, we can infer the incoming and outgoing dimension of each stage from these matrices alone.

Then, our format is simply a text file structured as follows:

```
for each layer in the network:
    weights matrix
    biases vector
```

For example, this is the model file for a network with a single hidden layer of size 5:

```
[[ 0.23400233  0.46448114 -0.18169046 -1.02006724]
 [ 0.91292487 -0.83703552  0.46820428  1.07709024]
 [ 0.77124784  0.60032112 -1.04011005 -1.06176582]
 [ 0.04108179  0.0236362  -0.6077499  -0.57499983]
 [ 0.41324034  0.71752045 -1.10962444  0.24183919]]
[ 1.05104845 -1.15333131  1.42181299  0.46979225 -0.31311588]
[[ 0.54906348 -1.2267044  1.117347  -0.7446324  1.17420772]
 [-0.24563245 -0.00604219  1.02716034  0.51893559 -0.44818289]
 [-0.79535198  0.72517445 -1.57061672  0.55235219  0.11317021]]
[ 0.18887961  0.43315065 -0.37568123]
```

From the first layer's weights having four columns, we can infer that there are four input features, and from the last layer's weights having three rows, we can infer that there are three output classes. This ability to infer input & output shape at parse-time is what allows this structure to be independent of the dataset the model is trained for.

It is also worth stating that the system currently makes the assumption that the network uses a ReLU activation at every layer except the last, where softmax is used (as is customary for multiclass classification). Therefore, in truth, the space of models the system accepts is a little more restricted than claimed in the specification (since in a classification model, the ReLU here could very well be replaced by a variety of alternative activation functions).

We can now dissect the modules separately.

3.2 Training Module

The purpose of the training module is to provide us with a set of models we can use to test our inference module. As such, we don't strictly require a high performance model. However, we still perform some basic hyperparameter selection after loading the dataset in using a 5-fold cross validation approach to determine the shape of the network (number & size of hidden layers).

The model architecture used is the one mentioned above, a simple feedforward network, a sequence of linear layers using the ReLU activation function between the hidden layers, and the softmax activation at the end. Using a grid search, we find the top 5 performing shapes, and also one poorly performing shape, out of a generated set of possible shapes for the network.

Before fitting 6 models using these shapes on the data, the data is transformed by converting the output classes column from textual labels to numerical labels, to avoid any ordering ambiguities arising during the training process. This transformed data is then split into training & test sets, and fit the models on the training set.

Finally, we write the networks' shapes, weights, and biases into model files, which can then be used by the inference module.

3.3 Inference Module

The inference module constitutes the bulk of the codebase. It produces an executable with the following signature:

```
inference-exe <path_to_model_file> [<path_to_batch_csv>]
```

If the second, optional argument is omitted, it starts in interactive mode. Otherwise, it starts in batched mode.

In either case, its first task is to restore the model from the given model file. Here, we utilise the parsing support module, which will be discussed in the Tooling section.

Through this parsing process, the model shape, and its parameters are loaded into a data structure which takes the form of a list of layers, where each layer stores its input & output shapes, and the weights & biases matrices.

Next, based on which mode it was started in, the system either enters a REPL or parses the input batch csv file it was provided (again, with the help of the parsing support module). In either mode, once the feature vector(s) are acquired, the tensor support module is used to perform the sequence of computations that lead to the final prediction.

The final output (for each feature vector) is the predicted class and the probabilities associated with each class.

4 Tooling

4.1 Training Module

The training module is written in python, using numpy, pandas, and sklearn to perform the data processing & model training/selection.

4.2 Inference Module

The inference module is written in Haskell. We use the Haskell Tool Stack (or just Stack) as our build tool. The following dependencies are used:

- **matrix & vector**

As the name suggests, these two packages provide a Matrix & Vector type respectively, also providing the definitions for common mathematical operations on them, including matrix multiplication, converting between the two types, etc.

Together, they form the tensor support module, which is used to finally perform the inference once the inputs have been parsed.

- **parsec**

Parsec is Haskell’s standard parser combinator library, allowing us to write atomic parsers and compose them together. Parsec forms the foundations of the parsing support module, which is used for parsing both the model files & csv files for batched input mode.

Discussion of the specifics of the parser implementations is deferred to the Prototype Details section.

4.3 Testing/Miscellaneous

A mix of bash & python scripts is used to implement the end-to-end tests and certain convenient utilities (eg. infer.py for loading and making predictions using sklearn, used to verify infer module).

5 Test Plan

Due to the large number of disjoint moving parts in the project, the plan is to implement a test suite consisting of unit & end-to-end tests.

5.1 Unit Testing

First, we note that the training module does not require testing, due to its simplicity and it almost exclusively using sklearn library functions. For the inference module, all the parser functions, and many of the inference functions are complex enough to warrant unit testing. We plan to implement these using HSpec, which is the testing framework Stack provides.

In order to cover the large space of inputs for these functions, we plan to heavily depend on generated test suites, as opposed to hand-crafted ones.

Consider two examples:

- **For the matrix parser:** We generate random real-valued matrices, stringify them, run them through the parser, and verify that the same matrix is recovered.
- **For batch softmax:** Use single-vector inference softmax as oracle and verify batch softmax.

5.2 End-to-End Testing

We use an end-to-end test to verify the correctness of our inference module. Here, we treat our sklearn model as an oracle, randomly generating many batches of feature vectors (in addition to the existing test split), running them through both our oracle & the inference module, and comparing the output activations.

6 Prototype Details

7 Plan for Completion

A high-level roadmap for the project's completion is as follows:

- Perform runtime checks on input dimension during inference.
- Runtime checks for interactive mode (for graceful fails).
- Implement tests.
- Allow alternative activation functions for non-output layers, as opposed to assuming ReLU.
- Add testing mode, given batch and expected outputs, report model accuracy.
- Add aggregation mode, use multiple trained models with voting (ensemble method).