Chapter 1 - Load Data From CSV

You must know how to load data before you can use it to train a machine learning model. When starting out, it is a good idea to stick with small in-memory datasets using standard file formats like comma separated value (.csv). In this tutorial you will discover how to load your data in Perl from scratch, including:

- How to load a CSV file.
- How to convert strings from a file to floating point numbers.
- How to convert class values from a file to integers.

Let's get started.

1.1 Description

1.1.1 Comma Separated Values

The standard file format for small datasets is Comma Separated Values or CSV. In its simplest form, CSV files are comprised of rows of data. Each row is divided into columns using a comma (,). In this tutorial, we are going to practice loading two different, standard machine learning datasets in CSV format.

1.1.2 Pima Indians Diabetes Dataset

In this tutorial we will use the Pima Indians Diabetes Dataset. This dataset involves the prediction of the onset of diabetes within 5 years. The baseline performance on the problem is approximately 65%. You can learn more about it in Appendix A, Section A.4. Download the dataset and save it into your current working directory with the filename pima-indians-diabetes.csv.

1.1.3 Iris Flower Species Dataset

In this tutorial we will also use the Iris Flower Species Dataset. This dataset involves the prediction of iris flower species. The baseline performance on the problem is approximately 26%. You can learn more about it in Appendix A, Section A.7. Download the dataset and save it into your current working directory with the filename iris.csv

1.2 Tutorial

This tutorial is divided into 3 parts:

1. Utility functions.
2. Load a file.
3. Load a file and convert Strings to Floats.
4. Load a file and convert Strings to Integers.

These steps will provide the foundations you need to handle loading your own data.

1.2.0. Utility functions

We need a few utilities to simplify object-oriented programming in Jupyter notebooks. One of the challenges is that class definitions tend to be fairly long blocks of code. Notebook readability demands short code fragments, interspersed with explanations, a requirement incompatible with the style of programming common for Perl libraries. The first utility function allows us to register functions as methods in a class after the class has been created. In fact, we can do so even after we have created instances of the class! It allows us to split the implementation of a class into multiple code blocks.

In [83]:
```
package sml{
  use strict;
  use warnings;
  use Data::Dump qw(dump);
  use List::Util qw(zip min max sum uniq all any shuffle);
  use Tie::IxHash;

  # https://stackoverflow.com/questions/28373405/add-new-method-to-existing-object-in-perl
  sub add_to_class{ #@save
    # Register functions as methods in created class.
    my($class, $method_name, $code_ref) = @_;

    {
      # We need to use symbolic references.
      no strict 'refs';
      # Shove the code reference into the class' symbol table.
      *{$class.'::'.$method_name} = $code_ref;
    }
  }

  1;
}
```

Out[83]: 1

```
Subroutine add_to_class redefined at reply input line 9.
```

We implement **add_to_class()** function inside a package named sml which stands for Statistical Machine Learning. This class will be expanded stepwisely throughout this course. The next step is to save the package sml into **sml.pm** in your Hard Drive in a visible Perl library path, such as /usr/local/share/perl5/5.34/x86_64-linux-thread-multi/, so that it can be loaded in every code. Find the path in your system and place the file sml.pm there. The function add_to_class() will allow registration of the functions needed by machine learning algorithms as methods of the class **sml**.

1.2.1 Load CSV File

The first step is to load the CSV file. We will use the csv module that is a part of the standard library. The reader() function in the csv module takes a file as an argument. We will create a function called load csv() to wrap this behavior that will take a filename and return our dataset. We will represent the loaded dataset as a list of lists. The first list is a list of observations or rows, and the second list is the list of column values for a given row. Below is the complete function for loading a CSV file.

In [94]:
```
# Load libraries
use strict;
use warnings;
use Data::Dump qw(dump);
use List::Util qw(uniq);
use AI::MXNet qw(mx);
```

In [63]:
```
# Defined in Section 1.2.1 Load CSV File
# Function for loading a CSV
# Load a CSV file
sub load_csv{
  my ($self, $file_path, %args) = (splice(@_, 0, 2), delimiter => '[,;\t]', @_);

  open (FILE, "<", $file_path) or die "Cannot open file $file_path: $!";
  my $header = <FILE>;
  chomp($header);
  my @dataset = ();
  while (<FILE>){
    my $row = $_;
    $row =~ s/[\r\n]+$//g; # Regular expression that deletes characters such as \r \n from a row
    next if (!defined $row || $row =~ /^\s*$/);
    push @dataset, [split /$args{delimiter}/, $row];
  }
  close FILE;

  return wantarray ? (\@dataset, $header) : \@dataset;
}
```

```
Subroutine load_csv redefined at reply input line 4.
```

Every time we defined a function for a machine learning algorithm, we need to include the variable $self as the first variable. It will contain the name of the class **sml**, followed by the variables that match with the obligatory parameters. The obligatory parameter variables values are directly received by their relative positions. Whereas the optional parameters are received as a dictionary, always including their names and respective values. To register the newly defined function **load_csv()** in the class sml, we just invoke **sml->add_to_class(funtion_name, & {'funtion_name'});** From that moment on, the function becomes a method of the sml class in memory. This allows for testing a newly implemented function. Once the function passes all the tests, it must be phisically embedded to the sml class. For that you must include the full function implementation as a method of the sml class, inside the file **sml.pm** in your local hard drive. This will allow you to construct your own statistical machine learning library for posterior use without the need of repeating the same codes everywhere.

In [64]:
```
sml->add_to_class('load_csv', \&{'load_csv'});
```

Out[64]:
```
*sml::load_csv
Subroutine sml::load_csv redefined at reply input line 17.
```

Now we can test this function by loading the Pima Indians dataset. Taking a peek at the first 5 rows of the raw data file we can see the following:

Pregnancies	Glucose	Blood Pressure	Skin Thickness	Insulin	BMI	Diabetes Pedigree	Age	Outcome
6	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
8	183	64	0	0	23.3	0.672	32	1
1	89	66	23	94	28.1	0.167	21	0
0	137	40	35	168	43.1	2.288	33	1

Peek at Pima Indians Diabetes dataset.

The data is numeric and phisically separated by commas in the file. Let's use the new function and load the dataset. Once loaded we can report some simple details such as the number of rows and columns loaded. Putting all of this together, we get the following:

In [91]:
```
# Load pima-indians-diabetes dataset
my $filename = '../data/pima-indians-diabetes.csv';
my ($dataset, $header) = sml->load_csv($filename);
printf "Loaded data file %s with %d rows and %d columns.\n",
$filename, scalar (@$dataset), scalar (@{$dataset->[0]});
```

```
Loaded data file ../data/pima-indians-diabetes.csv with 768 rows and 9 columns.
```

Out[91]:
```
1
```

In [85]:
```
printf "%s\n%s\n", $header, dump @$dataset[0 .. 4];
```

```
Pregnancies,Glucose,BloodPressure,SkinThickness,Insulin,BMI,DiabetesPedigreeFunction,Age,Outcome
(
  [6, 148, 72, 35, 0, 33.6, 0.627, 50, 1],
  [1, 85, 66, 29, 0, 26.6, 0.351, 31, 0],
  [8, 183, 64, 0, 0, 23.3, 0.672, 32, 1],
  [1, 89, 66, 23, 94, 28.1, 0.167, 21, 0],
  [0, 137, 40, 35, 168, 43.1, 2.288, 33, 1],
)
```

Out[85]: 1

In [87]:
```
# Alternative way to print a data sample:
print "@$_\n" for @$dataset[0 .. 4];
```

```
6 148 72 35 0 33.6 0.627 50 1
1 85 66 29 0 26.6 0.351 31 0
8 183 64 0 0 23.3 0.672 32 1
1 89 66 23 94 28.1 0.167 21 0
0 137 40 35 168 43.1 2.288 33 1
```

In [88]:
```
# We can also get the just the data without its header, thanks to 'wantarray()' command,
# which differentiates between scalar and array contexts.
my $new_dataset = sml->load_csv($filename);
printf "%s\n", dump @$new_dataset[0 .. 4];
```

```
(
  [6, 148, 72, 35, 0, 33.6, 0.627, 50, 1],
  [1, 85, 66, 29, 0, 26.6, 0.351, 31, 0],
  [8, 183, 64, 0, 0, 23.3, 0.672, 32, 1],
  [1, 89, 66, 23, 94, 28.1, 0.167, 21, 0],
  [0, 137, 40, 35, 168, 43.1, 2.288, 33, 1],
)
```

Out[88]: 1

1.2.2 Convert String into Floats

Most, if not all machine learning algorithms prefer to work with numbers. Specifically, floating point numbers are preferred. Our code for loading a CSV file returns a dataset as a list of lists, but each value is a string. We can see this if we print out one record from the dataset:

In [124…
```
printf "row[0]: %s\n", dump $dataset->[0];
```

```
row[0]: [5.1, 3.5, 1.4, 0.2, 0]
```

Out[124]: 1

We can write a small function to convert specific columns of our loaded dataset to floating point values. Below is this function called str column to float(). It will convert a given column in the dataset to floating point values, careful to strip any whitespace from the value before making the

conversion.

In [89]:
```
# Defined in Section 1.2.2 Convert String to Floats
# Function For Converting String Data To Floats.
# Convert string columns to float
sub str_column_to_float{
  my ($self, $dataset, $column, %args) = (splice (@_, 0, 3), precision=>1, @_);

  return if ($dataset->[0][$column] !~ /^\d+/);

  $args{precision} = '%.' . $args{precision} . 'f';
  for my $row (@$dataset){
    $row->[$column] = sprintf ($args{precision}, $row->[$column]);
  }
}

sml->add_to_class('str_column_to_float', \&{'str_column_to_float'});
```

Out[89]:
```
*sml::str_column_to_float
```

```
Subroutine str_column_to_float redefined at reply input line 4.

Subroutine sml::str_column_to_float redefined at reply input line 17.
```

We can test this function by combining it with our load CSV function above, and convert all of the numeric data in the Pima Indians dataset to floating point values. The complete example is below.

In [100…
```
($dataset, $header) = sml->load_csv($filename);
printf "Loaded data file %s with %d rows and %d columns.\n",
               $filename, scalar (@$dataset), scalar (@{$dataset->[0]});

printf "Header: %s\n", $header;
printf "Strings: %s\n", dump $dataset->[0];

# convert string columns to float
for my $i (0 .. $#{$dataset->[0]} -1){
  sml->str_column_to_float($dataset, $i);
}
printf "Floats: %s", dump $dataset->[0];
```

```
Loaded data file ../data/pima-indians-diabetes.csv with 768 rows and 9 columns.
Header: Pregnancies,Glucose,BloodPressure,SkinThickness,Insulin,BMI,DiabetesPedigreeFunction,Age,Outcome
Strings: [6, 148, 72, 35, 0, 33.6, 0.627, 50, 1]
Floats: ["6.0", "148.0", "72.0", "35.0", "0.0", 33.6, 0.6, "50.0", 1]
```

Out[100]:
```
1
```

1.2.3 Convert String into Integers

The iris flowers dataset is like the Pima Indians dataset, in that the columns contain numeric data. The difference is the final column, traditionally used to hold the outcome or value to be predicted for a given row. The final column in the iris flowers data is the iris flower species as a string. For example, below are the first 5 rows of the raw dataset.

SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3.0	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5.0	3.6	1.4	0.2	Iris-setosa

Peek at Iris Flower Species dataset.

Some machine learning algorithms prefer all values to be numeric, including the outcome or predicted value. We can convert the class value in the iris flowers dataset to an integer by creating a map.

1. First, we locate all of the unique class values, which happen to be: Iris-setosa, Iris-versicolor and Iris-virginica.
2. Next, we assign an integer value to each, such as: 0, 1 and 2.
3. Finally, we replace all occurrences of class string values with their corresponding integer values. Below is a function to do just that called str_column_to_int(). Like the previously introduced str_column_to_float() it operates on a single column in the dataset.

In [81]:
```
# Defined in Section 1.2.3 Convert String to Integers
# Function To Integer Encode String Class Values.
# Convert string column to integer
sub str_column_to_int{
  my ($self, $dataset, $column) = @_;

  my $class_values = [map {$_->[$column]} @$dataset];
  my @unique = uniq @$class_values;
  my %lookup = ();
  while (my ($i, $value) = each @unique) {
    $lookup{$value} = $i;
  }
  for my $row (@$dataset){
    $row->[$column] = $lookup{$row->[$column]};
  }

  return \%lookup;
```

```
}

sml->add_to_class('str_column_to_int', \&{'str_column_to_int'});
```

Out[81]: *sml::str_column_to_int

We can test this new function in addition to the previous two functions for loading a CSV file and converting columns to floating point values. It also returns the dictionary mapping of class values to integer values, in case any users downstream want to convert predictions back to string values again. The example below loads the iris dataset then converts the first 3 columns to floats and the final column to integer values.

In [123…

```
# Load iris dataset
$filename = '../data/iris.csv';
($dataset, $header) = sml->load_csv($filename);
print sprintf "Loaded data file %s with %d rows and %d columns.\n\n",
               $filename, scalar(@$dataset), scalar(@{$dataset->[0]});

printf "Before label conversion into integers:\n\n%s\n%s\n\n", $header, dump @$dataset[0 .. 4];

# convert string columns to float
for my $i (0 .. $#{$dataset->[0]} -1){
  sml->str_column_to_float($dataset, $i);
}

# convert class column to int
my $lookup = sml->str_column_to_int($dataset, -1);
printf "After label conversion into integers:\n\n%s\n%s\n\n", $header, dump @$dataset[0 .. 4];
printf "Conversion dictionary: %s\n", dump $lookup;
```

```
Loaded data file ../data/iris.csv with 150 rows and 5 columns.

Before label conversion into integers:

SepalLengthCm,SepalWidthCm,PetalLengthCm,PetalWidthCm,Species
(
  [5.1, 3.5, 1.4, 0.2, "Iris-setosa"],
  [4.9, "3.0", 1.4, 0.2, "Iris-setosa"],
  [4.7, 3.2, 1.3, 0.2, "Iris-setosa"],
  [4.6, 3.1, 1.5, 0.2, "Iris-setosa"],
  ["5.0", 3.6, 1.4, 0.2, "Iris-setosa"],
)

After label conversion into integers:

SepalLengthCm,SepalWidthCm,PetalLengthCm,PetalWidthCm,Species
(
  [5.1, 3.5, 1.4, 0.2, 0],
  [4.9, "3.0", 1.4, 0.2, 0],
  [4.7, 3.2, 1.3, 0.2, 0],
  [4.6, 3.1, 1.5, 0.2, 0],
  ["5.0", 3.6, 1.4, 0.2, 0],
)

Conversion dictionary: { "Iris-setosa" => 0, "Iris-versicolor" => 1, "Iris-virginica" => 2 }
```

Out[123]: 1

1.2.4 Convert Arrays into Tensors

Once the arrays are fully preprocessed as numeric and all columns and rows are consistently filled, the conversion into tensor is straightforward.

In [98]:
```
my $tensor = mx->nd->array($dataset);
printf "Tensor:  %s\n", $tensor;
printf "Tensor:  %s\n", $tensor->slice_axis(axis=>0, begin=>0, end=>5)->aspdl;
```

```
Tensor:  <AI::MXNet::NDArray 768x9 @cpu(0)>
Tensor:
[
 [      6     148      72      35       0    33.6     0.6      50       1]
 [      1      85      66      29       0    26.6     0.4      31       0]
 [      8     183      64       0       0    23.3     0.7      32       1]
 [      1      89      66      23      94    28.1     0.2      21       0]
 [      0     137      40      35     168    43.1     2.3      33       1]
]
```

Out[98]: 1

The next step is to obtain train and test partitions out of the dataset.

1.3 Extensions

You learned how to load CSV files and perform basic data conversions. Data loading can be a difficult task given the variety of data cleaning and conversion that may be required from problem to problem. There are many extensions that you could make to make these examples more robust to new and different data files. Below are just a few ideas you can consider researching and implementing yourself:

- Detect and remove empty lines at the top or bottom of the file.
- Detect and handle missing values in a column.
- Detect and handle rows that do not match expectations for the rest of the file.

1.4 Review

In this tutorial, you discovered how you can load your machine learning data from scratch in Perl. Specifically, you learned:

- How to load a CSV file into memory.
- How to convert string values to floating point values.
- How to convert a string class value into an integer encoding.