data_reader Documentation

Release 1.2

Will Alexander

Mar 22, 2018

CONTENTS:

CHAPTER

ONE

PYTHON DATA READER

Author William Alexander

worksprogress1@gmail.com

1.1 Introduction

Before model building can take place, there must be a dataset. Preparing the data for model building is both the least interesting but perhaps most crucial step in the process of building a model. Common tasks involved in preparing data for modeling are:

1. Pulling the data.

   Sometimes the data is stored in a lovely, well-maintained database. Often the data arrives in character based files–either delimited files or flat files. A deflimited file is one in which the fields are separated by a specific character, often a comma or pipe (|). A flat file is one in which each field occupies a specific set of columns within the file.

   For the purposes of speed, repeatability, documentation (and sanity!) it is very useful to take a programmatic approach to specifying the format of the data and processing it.

2. Cleaning the data.

   The data might be ready to go after reading it in. More often, though, some or all of the fields will have issues that must be dealt with. Common issues includes missing, bad or wild values. One may wish to treat these values in a variety of manners–flagging them, filling in a value, dropping them or even halting the process.

3. Derived variables.

   Often, there are other quantities to be calculated from those being read in. Some of these may be calculated from the data that's read in. Others may be mapped from auxilliary datasets. For example, it some domains it is very commong to map a zip code to a CBSA (core-based statistical area) code.

4. Sampling the data.

   If the file is large, then it can be useful or necessary to sample the data.

5. Output.

   Once the data has been read, there are a variety of formats that might be used to store the data: pandas DataFrame, numpy matrix, Python list, or write to a file (delimited or TensorFlow TFRecords format).

The *data_reader* module is designed to facilitate these tasks. *data_reader* provides the following functionality:

- Reading Common file types.

  data_reader reads both delimited and flat files.

- Support for multiple data types.

  data_reader supports the following data types: float, int, str, bytes, date, zip (U.S. zip codes), state (U.S. States), stateterr (U.S States and Territories).

- Data validation.

  data_reader will validate

  - the data type.
  - the data is within a specified range.
  - the data is in a list of allowed values.

- Data error handling options.

  If the data fails a validation rule, the following options are available:

  - replace the value with a user-specified value
  - drop the row from the output
  - halt the processing

- Supports a user-supplied function during file read.

  The user can supply a function that is called as each row is processed. The function can modify the values and create new fields.

- Supports a user-supplied class during file read.

  The user can supply a class. The class is initialized once before the data is read. A user-specified method from the class is called as each observation is read. A class allows for, e.g., reading in a data map so that a field in the file can be mapped to a new field (simple example: creating a State field from a zip code).

- Multi-processing support.

  data_reader supports parallel processing. This can significantly reduce processing time.

- Multiple output options.

  data_reader supports the following output formats: pandas DataFrame, numpy matrix, Python list, or delimited file.

- Read any portion of a file.

  The user can specify the start and end row to read.

- Random sampling.

  The user can specify a sampling rate.

The code is available here.

It may be installed via pip3. For example, on Ubuntu:

sudo pip3 install git+https://git@github.com/worksprogress1/DataReader

1.2 Approach

Any module designed to read a general file must provide the user a lot of flexibility. However, a reader that is constantly parsing a set of rules for each row is likely to be slow. Conversely, code that is specially written to read a specific file will likely run faster but won't have any flexibility.

The approach of *data_reader* is to aim at the best of both. The *data_reader* process is to:

- Build a data dictionary specific to the data to be read. Here is where all the data types and validation rules are specified.
- Create a *reader* module specific to this data dictionary. The rules and data types are 'hard coded' into the module to maximize execution speed.

1.3 List of Functions and Classes

The data_reader module contains the following classes and functions:

- class *BuildDataDictionary*

  This class builds up the data dictionary, field by field.

- function *create_reader*

  This function creates the *reader* module based on the dictionary created by *BuildDataDictionary*.

- function *multi_process*

  Executes the *reader* module in multi-process mode.

- class *PopulateCBSAData*

  This class adds the CBSA FIPS code and, optionally, CBSA name to the data. It can also check for the agreement between the zip code and the state postal code.

The *reader* module created by *create_reader* has one function to call: *reader*. The parameter to *reader* is a dictionary. The elements of the dictionary are:

- *data_file* (str). Name of the file to read.
- *module_path* (str). The path to the *reader* module. If this omitted, then it is assumed that the module is in the reader subdirectory of the *data_reader* module. The *reader* needs this path so that it can access legal values from the *data* subdirectory within the *reader* directory.
- *output_type* (str). How to output the data. Choices are:
  - 'list'. A list of lists where each sublist is a row of data.
  - 'numpy'. A numpy matrix.
  - 'pandas'. A pandas DataFrame. This is the default value.
  - 'delim'. A delimited file.
  - 'TFRecords'. A TensorFlow TFRecords format file.

    If the user selects output_type = 'delim' or 'TFRecords', additional parameters are used:

    * *output_file* (str). The name of the output file.
    * *output_delim* (str) ('delim' only). The delimiter to use with *output_file*. The default value is ','.
    * *output_headers* (bool) ('delim' only). If *True*, output a header row. The default is *True*.
  - *gzip* (bool). (optional) If *True* the output is gzipped. Note: gzip executable must be in the path for this to work.
  - *split_file* (int). (optional). If this is defined, the output is split into separate files of *split_file* rows.
  - *partition* (str). (optional). The name of a field in the data to partition on. If this is defined, then separate files are created for each value of *partition* within a subdirectory whose name is "<partition var>=<value>". The *partition* field is dropped from the output files.

  - *module_name*. (optional). The defualt is 'reader'. This is the name of the module that is created. If, in one run, multiple readers are created, they must have distinct module names.

  The value must be at least 10.

  Note that the file is written line by line so the entire dataset is never in memory.

- *headers* (bool). True means the input file has headers. The default value is *False*.
- *sample_rate* (float). The rate at which to sample the file. The default value is 1.
- *first_row* (int). The first row of data to read.
- *last_row* (int). The last row of the data to read.

  Note that *first_row* and *last_row* are ignored by function *multi_process*.

- *user_function* (function). A user-supplied function that is called as each row is processed. It can take only one argument, a dictionary. The dictionary keys are the names of the fields. The function can modify or add entries to the dictionary. The function is called only after the row has undergone data validation (field type, legal values, maximum and minimum values). The function must return a type *bool*. If the return is *True*, the row is kept.
- For user-supplied classes, the following entries are required:
  - *user_class* (class). The class for *reader* to use.
  - ***user_class_init* (dict). A dictionary supplying any initialization parameters required by the class.** The keys are the names of the parameters. If the initialization requires no parameters, supply an empty dictionary.
  - ***user_method* (str). The name of the method to run, as a string. The method can take only a single argument:** a dictionary. The dictionary contents are the current row being read. The dictionary keys are the names of the fields. The method can modify or add entries to the dictionary. *user_method* is called only after the row has undergone data validation (field type, legal values, maximum and minimum values). The method must return a type *bool*. If the return is true, the row is kept.

  Note: the *user_function* is called before the *user_method*.

- *window* (int). An optional window for *mmap*. If there are memory issues (which there should not be on 64 bit implementations), this is the size of the window into the file used by *mmap*. If *None*, there is no window. The default is *None*.
- *start_byte* (int). The byte at which to start reading the file. The default value is 0. If the value is greater than 0, then reading begins at the next line ("\n") after *start_byte*.
- *end_byte* (int). The byte at which to stop reading the file. The default value is *None* (read to the end of the file). If a non-*None* value is specified, reading stops at the first line whose first byte is greater than end_byte.

  The above two will generally only be used for reading the file in multiprocessing mode.

If the *output_type* is 'list', 'numpy' or 'pandas' then *reader* returns the data in that format. If the *output_type* is *delim*, then there is no return from *reader*.

1.4 Data Types

data_reader supports the following data types:

- float. Real value.
- int. Integer value.

- date. Date value.

  The following date formats are accepted:

  - CCYYMMDD
  - CCYYMM
  - YYMM
  - MM/DD/CCYY
  - MM/DD/YY
  - MMDDCCYY
  - MM/CCYY
  - CCYY/MM/DD

  If an E is appended to the format, the date is moved to the end of the month. If a B is appended, the date is moved to the first of the month.

- str. String value.
- bytes. Array of bytes.
- zip. US zip code. Values are automatically validated.
- state. US state postal code. Values are automatically validated.
- stateterr. US states and territories. Values are automatically validated.

1.5 TensorFlow Support

data_reader provides support for TensorFlow in the following manners:

1. Creation of TFRecords files.

   TensorFlow includes support for reading CSV files and a binary format known as TFRecords. TensorFlow includes several APIs for reading and writing this format. TFRecords files support lists of three basic data types:

   - int
   - float
   - byte

   Anything that is put into a TFRecord file must be one of these. data_reader makes the following conversions:

   - Strings. Strings are converted to a list of bytes.
   - Dates. Dates are converted to a length-3 list of integers: [year, month, day].

2. Support functions.

   There are two common functions used in TensorFlow that require the field names and types of the input data. These are:

   - *input_fn*. The input_fn is called by TensorFlow to supply its routines with data. The input_fn as attributes such as the size of each batch to draw, whether the data should be shuffled, the observations at which to start and end reading. The function can also include the creation of new features. The input function needs to explicitly define each feature and its type.

- *model_columns*. The model columns defines the features used in the model. The function not only specifies the features in the model but also their form. For instance, features of type *float* can enter as their raw value or as buckets. Features of type string can enter with a list of values the feature can take on or TensorFlow can build the list using a hash table. (*Note*: if the data_reader is given a list of legal values, the former is used.) Further, these categorical variables can enter using a one-hot (indicator) format or an embedding layer (for neural nets).

data_reader creates these functions by writing them to a Python module (file) specified by the user.

The *input_fn* created by data_reader reconstructs strings from byte arrays. TensorFlow itself has no date type. The user specifies how to handle dates when data_reader builds the input_fn. All the choices result in an output of type int. The choices are:

- CCYYMMDD
- CCYYMM
- CCYY
- MM
- DD

1.5.1 Using input_fn

The *input_fn* function takes several parameters that control the way the data is accessed. These are:

- files. The file name or list of file names to read.
- batch_size. The number of observations to include in each batch read from the file.
- shuffle. If 0, the data is not shuffled. If > 0, then this many observations are shuffled before reading.
- skip. The number of observations to skip before reading. This allows accessing records further into the file.
- num_epochs. The number of time to repeat reading the data before an EOF flag is thrown.
- parallel_calls. The number of threads to use when reading the file.
- **include_columns. This is the same dictionary that is input to the *model_columns* function (below). For *input_fn*, it** directs how features of type INT/DATE should be handled. The keys to *input_columns* are the features to be used in the model. The entries are themselves dictionary ("feature dictionary"). For features of type INT/DATE, the feature dictionary may have a key named "type". If "type" has value STR, then the output feature is converted to type STR. If "type" has value "FLOAT", the return is type FLOAT. If *include_columns* is not passed to *input_fn* or there is no key for an INT/DATE feature, then the return is type FLOAT.

1.5.2 Using model_columns

The *model_columns* function takes a dictionary, *include_columns*, as input and returns a dictionary of features which specifies the model structure to TensorFlow. The keys to *input_columns* are the names of the features to include in the model. The dictionary entry is also a dictionary ("feature dictionary"). The feature dictionary directs how the feature is treated in the model. Possible entries vary by the type of the tensor:

- **INT/DATE**
  - *type*. The type entry directs whether to treat the tensor as numeric (FLOAT) or a string (STR). The remainder of the entries conform to the class chosen.
- **STR, BYTES (and DATE/INT when key "type" is STR):**

  - if the data dictionary specifies legal values, then category_column_with_vocabulary_list is used.
  - **if the dict has an entry *vocab_list* then that vocabularly list is used. Note: the *vocab_list*** must be of type str.
  - if the dict has an entry *hash_size* then a hash list of that size is used
  - if there are none of the above, then a hash list of 1000 is used
  - if the dict has an entry *n_oov* then there are this many out-of-value catch-all buckets
  - if the dict has an entry *embed_size* then an embedded layer of that size is used
- **FLOAT (and DATE/INT when key "type" is FLOAT):**
  - if nothing is specified, then it is treated as numeric_column
  - if the dict has a key *boundaries* with a list of boundary values, then bucketized_column is used

1.6 Examples

Even though there are not many classes and functions within *data_reader*, there is a lot of flexibility. Examples will be helpful.

1.6.1 Example 1: Reading a delimited file with no headers.

This example reads the file that maps zip codes to CBSA codes that is in the data directory of this data_reader distribution. The file has the following fields:

- zip. The 5-digit zip code.
- CBSA code. If the zip is in a CBSA, this is the 5-digit CBSA code. If it is not, it is the state postal abbreviation.
- state postal abbreviation. The 2-character state abbreviation.
- level. The level of the CBSA code: metropolitan or micropolitan.
- CBSA name. The name of the CBSA.

The data dictionary for this file can be built as follows:

```
import data_reader.data_reader as d

d0 = d.BuildDataDictionary()
d0.add_field('zip', 'zip', illegal_replacement_value='"00000"', action='fix')
d0.add_field('cbsa_code', 'str')
d0.add_field('state', 'stateterr')
d0.add_field('level', 'str')
d0.add_field('cbsa_name', 'str')
d0.print()
```

The dictionary uses two special data types: *zip* and *stateterr* (U.S.states and territories). The *reader* code produced by *data_reader* will check that the values in the *zip* field are valid zip codes. Similarly, it will check that the *state* field contains valid state and territory values. If a *zip* is found to be invalid, we've directed that the value be replaced by '00000'.

Notes:

- String value passed to the *add_field* method for parameters *illegal_replacement_value*, *maximum_replacement_value* and *minimum_replacement_value* must be enclosed in double quotes inside single quotes as in '"00000"'. Bytes fields would include the *b* between the single and double quotes such as 'b"00000"'.
- If the file does not have headers, the fields must be put into the dictionary in the same order that they are in the file to be read.
- The *action* parameter value of 'fix' directs that invalid values are to be replaced by the illegal_replacement_value of '00000'. Other options for *action* are: 'drop' and 'fatal' which cause *reader* to drop the row and abend, respectively.

The code:

```
d.create_reader(d0.dictionary, file_format='delim', delimiter='|')
```

creates the *reader* module in the distribution directory of *data_reader*. It can be imported as:

```
import data_reader.reader.reader as r
```

The *reader* takes its parameters as a dictionary:

```
import pkg_resources
cbsa_file = pkg_resources.resource_filename('data_reader', 'data/') + 'zipCBSA.dat'

parameters0 = {}
parameters0['data_file'] = cbsa_path
parameters0['output_type'] = 'pandas'
```

These are the minimal parameters required by the *reader*: the file to read and the output type. This code reads the file:

```
import data_reader.reader.reader as r
d0_data = r.reader(parameters0)
```

The result is a pandas DataFrame with columns *zip*, *cbsa_code*, *state*, *level*, and *cbsa_name*.

1.6.2 Example 2. Reading a file with headers.

The file 'zipCBSA_headers.dat' in the *test_data* directory of the *data_reader* module contains the same data as the file in Example 1 but with headers. The code from Example 1 will work with the addition of:

```
parameters0['headers'] = True
```

Notes:

- The field names in the dictionary must match the corresponding names in the header row. The match is case sensitive.
- Since the file has headers, the entries may be placed into the dictionary in any order. Also, not all the fields need to be read. For example, if the dictionary is specified as:

  ```
  d0 = d.BuildDataDictionary()
  d0.add_field('zip', 'zip', illegal_replacement_value='"00000"')
  d0.add_field('state', 'stateterr')
  ```

  then only the *zip* and *state* fields will be read.

1.6.3 Example 3. Subsetting a file based on values in the file.

The 'drop' option of the *action* parameter can be used to subset the file as it is read. Returning to Example 1, suppose we were only interested in the zip codes in California. If we alter the *state* entry in the dictionary to:

```
d0.add_field('state', 'str', legal_values = np.array(['CA']), action='drop')
```

then any time a row has a value other than 'CA' for state, it will be dropped. Note that instead of specifying a type of 'stateterr' for state, we've used 'str' and supplied the one value we're interested in.

1.6.4 Example 4. A more complex subsetting example.

In this example, there are two files to read. The first file ('customers') contains information about customers that doesn't change month-to-month. There is one record per customer. The second file contains monthly information about the customer ('monthly'). We want to sample the 'customers' file and pull all the records for these customers from the 'monthly' file.

The code to sample 'customers' is:

```
# Create the dictionary.
ds = d.BuildDataDictionary()
ds.add_field('account_number','str')
ds.add_field('r','float')
ds.add_field('income','float')
ds.add_field('age','int')
ds.add_field('open_date','date','mm/dd/ccyy')
ds.add_field('close_date','date','ccyymmdd')
ds.add_field('trans_code','str')
ds.add_field('state','state')
ds.add_field('rate','float')
ds.add_field('gender','str')
ds.add_field('marketing_flag','str')

# create the *reader* module.
d.create_reader(ds.dictionary, file_format='delim', delimiter=',')

# parameters for *reader*
import pkg_resources
params = {}
params['data_file'] =  pkg_resources.resource_filename('data_reader', 'test_data/') +
↪'customers.csv'
params['output_type'] = 'pandas'
params['headers'] = True
params['sample_rate'] = 0.1

# read the file
import data_reader.reader.reader as r
ds_data = r.reader(params)
```

Note that the this example shows two of the different available date formats. The *open_date* field uses the format 'mm/dd/ccyy', such as '3/31/2017'. The *close_date* field uses the 'ccyymmdd' format, such as '20170331'. A 10% sample is read from the file. The key between the two files is *account_number*. The approach is to specify these account numbers as the legal value when constructing the data dictionary for the 'monthly' file and to direct it to drop any illegal values.

The code to do this looks like this:

```
# legal values are specified as a sorted numpy array
ok_id = np.asarray(np.squeeze(ds_data['account_number']))
ok_id.sort()

# build the data dictionary for the monthly file
dm = d.BuildDataDictionary()
dm.add_field('account_number','str', legal_values=ok_id, action='drop')
dm.add_field('r','float')
dm.add_field('cutoff_date','float')
dm.add_field('balance','float')

# create the reader
d.create_reader(dm.dictionary, file_format='delim', delimiter=',')

import importlib
# since we have already imported reader, this will force Python to re-load it.
importlib.reload(r)

# parameters for *reader*
param_m = {}
param_m['data_file'] =  pkg_resources.resource_filename('data_reader', 'test_data/')␣
↪+ 'monthly.csv'
param_m['output_type'] = 'pandas'
param_m['headers'] = True

# read the monthly records for our random sample.
dm_data = d.multi_process(r.reader, param_m, 2)
```

This example introduces another feature of *data_reader*: the *multi_process* function. The function takes as arguments the *reader* function, its parameters and the number of processes to spawn. Multiprocessesing can substantially reduce run time. An experimet was performed on a larger version of this file (5 MM customers, 450MM monthly records). On an i7 laptop, two processes reduce run time by about 30%;on a six-core Xeon workstation, six processes reduces the run time by about 80%.

If the *output_type* is 'list', 'numpy' or 'pandas' the return from *multi_processor* is the entire dataset in that format. If the *output_type* is 'delim' then a separate file is created by each process. The file name is the user-supplied file with a number appended. These can be combined by the bash *cat* command.

1.6.5 Example 5. User-supplied functions.

The user can supply a function to the *reader*. The function is executed as each row is read. This function can do such things as:

- Calculate new fields to include in the output.
- Perform complex QA on the fields.

The function takes a dictionary as its sole argument. The elements in the dictionary are the field values for the row, the keys are the field names. Since dictionaries are mutable, the function can modify–or add–values to the dictionary.

Notes:

- The function must return a type *bool*. If the function returns *True*, the row is kept; if *False* it is discarded.
- The function is called only after all the fields have been through the standard QA (checking type, illegal values, maximums, minimums) and if the row is to be otherwise kept (*e.g.* to be included in the sample).

Returning to Example 4, suppose we wish to calculate the number of months between the *open_date* and *close_date*. To add this field to the DataFrame while the file is being read, we first define a function to do the calculation and then

add it to the call to the *reader*:

```
def months(row):
    rx = row['close_date'].month - row['open_date'].month
    rx += 12 * (row['close_date'].year - row['open_date'].year)
    row['time_on_file'] = rx
    return True

params['user_function'] = months
```

The function takes the parameter *row* which is a dictionary of the fields being read. It calculates *time_on_file* and then adds it to the dictionary. Finally, it returns *True* so that the row is kept.

1.6.6 Example 6. A user-supplied class to include lagged values.

The *reader* also takes an optional user-supplied class. A class provides additional functionality beyond a function. For instance, it can retain information from row to row. Again returning to Example 4, suppose we wish to include the balance from the prior month in the DataFrame for monthly values. The lagged value only makes sense for values within an account. Suppose, also, we wish to drop the first month in which the lag does not exist. First, define the class we will need:

```
class lagger(object):
  def __init__(self):
      self.lagged_account_number = None
      self.lagged_balance = None

  def lag(self, fx):
      if self.lagged_account_number is not None:
          if fx['account_number'] == self.lagged_account_number:
              fx['lag_balance'] = self.lagged_balance
              self.lagged_account_number = fx['account_number']
              self.lagged_balance = fx['balance']
              return True
      self.lagged_account_number = fx['account_number']
      self.lagged_balance = fx['balance']
      return False
```

We need to keep track of both the *account_number* and *balance* from the prior row. These are stored in the *class* variables *self.lagged_account_number* and *self.lagged_balance*. The method returns *False* when the first month for an account is seen, otherwise it returns *True*. The following entries are added to the parameter dictionary for *reader*:

```
param_m['user_class'] = lagger
param_m['user_class_init'] = {}
param_m['user_method'] = 'lag'
```

When run, the return DataFrame, *dm_data* contains a new field: *lag_balance*. Of course, this depends on records within an *account_number* being sorted ascending in the 'monthly' file. The *lagger* class could be modified to check for this.

1.6.7 Example 7. Reading a flat file.

The file test2.dat in the *test_data* directory is an example flat file. The file has six fields:

- obs. The integer row number. Start column: 1, width: 3.
- float1. A real number. Start column: 4, width: 6.

- letters. A string. Start column: 10, width: 5.
- state. A U.S. state abbreviation. Start column: 15, width: 2.
- date1. A date using CCYYMMDD format. Start column: 17, width: 8.
- date2. A date using mm/dd/ccyy format. Start column: 25, width: 10.

The dictionary for this file can be specified as:

```
d8 = d.BuildDataDictionary()
d8.add_field('obs', 'int', field_start=1, field_width=3)
d8.add_field('float1', 'float', field_start=4, field_width=6)
d8.add_field('letters', 'str', field_start=10, field_width=5)
d8.add_field('state', 'state', field_start=15, field_width=2)
d8.add_field('date1', 'date', field_start=17, field_width=8, field_format='CCYYMMDDE')
d8.add_field('date2', 'date', field_start=25, field_width=10, field_format='MM/DD/
↪CCYYB')
```

The 'E' on the date format 'CCYYMMDDE' tells the *reader* to move the date to the last day of the month. The 'B' on the format 'MM/DD/CCYYB' tells the *reader* to move the date to the first day of the month. The code to create the *reader* is:

```
d.create_reader(d8.dictionary, file_format='flat', lrecl=36)
```

The *reader* needs to know the total width of each row. In this example, the data occupies 35 columns. The new line '\n' character occupies the 36th. In DOS format, the *lrecel* would be 37 since DOS uses both '\n' and '\r' at the end of each line.

1.6.8 Example 8. Reading a file with delimited strings.

When reading a delimited file, sometimes the file to be read contains strings which include the delimiter. In this case, it is customary to enclose the strings within another delimiter, often a double quote. *data_reader* can handle this situation. The *create_reader* function takes the optional argument *string_delim*. The CSV file originatorsFull.csv in the test_data directory is an example of this. It has two fields: a name and an integer key. Since the name may include a comma, the string is enclosed in double quotes.

The code to create the *reader* looks like this:

```
da = d.BuildDataDictionary()
da.add_field('originator', field_type='str')
da.add_field('key',field_type='int')
d.create_reader(da.dictionary, file_format='delim', delimiter=',', string_delim='"')
```

1.6.9 Example 9. Outputting to a file.

The *reader* can also write its output to a file.

Notes about writing the output to files:

- In this mode, the *reader* can work on arbitrarily large files as the files are read and written a line at a time.
- When used with *multi_process*, an output file is created by each process. A number is appended to the user-supplied file name. These can be concatenated afterword via *cat*.

Returning to Example 1, if the *reader* parameters are changed to:

```
import pkg_resources
cbsa_file = pkg_resources.resource_filename('data_reader', 'data/') + 'zipCBSA.dat'

parameters0 = {}
parameters0['data_file'] = cbsa_path
parameters0['output_type'] = 'delim'
parameters0['output_file'] = '~/test/zipCBSA.csv'
parameters0['output_delim'] = ','
parameters0['output_headers'] = False
```

Then the reader will produce a comma-delimited version of zipCBSA.dat.

1.6.10 Example 10. TensorFlow.

The test_data directory of the data_reader distribution contains a TensorFlow file with the following fields:

- x1. Type float.
- x2. Type float.
- x3. Type float.
- x4. Type float.
- x5. Type date tensor (length 3 of type int)
- xd. Type bytes list.
- xe. Type int.
- bad. Type int.

The file was created by the function below:

```
def write_tfr(tfrecord_filename, n):
    writer = tf.python_io.TFRecordWriter(tfrecord_filename)

    for rows in range(0, n):
        x1 = np.random.normal(0, 1, 1)
        x2 = np.random.normal(0, 1, 1)
        x3 = np.random.normal(0, 1, 1)
        x4 = np.random.uniform(0, 1, 1)
        yr = int(np.random.uniform(0, 1, 1) * 20) + 1999
        mo = int(np.random.uniform(0, 1, 1) * 12) + 1
        day = 1
        x5 = [yr, mo, day]
         pl = -1 + x1 * 1 - x2 + 0.5 * x3 + x3 * x3 + math.sin(math.pi * x2) + x2 *␣
↪x2 + x2 * x3

         xd = 'now now'  # values:  'now now', 'then then', 'now then', 'then now'
        xe = 0
        if x4 < 0.75 and x4 >= 0.5:
            xd = 'then then'
            xe = 1
            pl -= 1.0
        if x4 < 0.5 and x4 >= 0.25:
            xd = 'now then'
            xe = 2
            pl += 2.0
        if x4 < 0.25:
```

```
            xd = 'then now'
            xe = 3
            pl -= 2.0
        prob = math.exp(pl) / (1.0 + math.exp(pl))

        if np.random.uniform(0, 1, 1) < prob:
            bad = [1]
        else:
            bad = [0]
        xf = [a.encode() for a in xd]
        xe = [xe]

        f1 = tf.train.Feature(float_list=tf.train.FloatList(value=x1))
        f2 = tf.train.Feature(float_list=tf.train.FloatList(value=x2))
        f3 = tf.train.Feature(float_list=tf.train.FloatList(value=x3))
        f5 = tf.train.Feature(int64_list=tf.train.Int64List(value=xe))
        f6 = tf.train.Feature(int64_list=tf.train.Int64List(value=x5))
        f4 = tf.train.Feature(bytes_list=tf.train.BytesList(value=xf))
        i1 = tf.train.Feature(int64_list=tf.train.Int64List(value=bad))
        feature = {'bad': i1, 'x1': f1, 'x2': f2, 'x3': f3, 'xd': f4, 'xe': f5, 'x5':␣
↪f6}

        features = tf.train.Features(feature=feature)
        example = tf.train.Example(features=features)
        writer.write(example.SerializeToString())

    writer.close()
```

If you look at this function, you will see that

1. 'bad' is a binary outcome related to the the x's (*i.e.* it's a logistic model)
2. The x's interact.
3. The relationship of the x's to 'bad' is non-linear even in log odds space.
4. The date tensor is not related to 'bad'.
5. xe and xd encode the same information.

Since the file is already in TFRecord format, there is no need to create a reader. However, it would be very helpful to have *input_fn* and *model_columns* functions.

To do so, we first must create a data dictionary for the file:

```
import data_reader as d
from data_reader import make_input_fn
from data_reader import make_model_columns


da = d.BuildDataDictionary()
da.add_field('x1', field_type='float')
da.add_field('x2', field_type='float')
da.add_field('x3', field_type='float')
da.add_field('xd', field_type='str', legal_values=['now now', 'then then', 'now then',
↪ 'then now'],
             illegal_replacement_value='X')
da.add_field('xe','int',minimum_value=0, maximum_value=3, minimum_replacement_value=-
↪1, maximum_replacement_value=-1)
da.add_field('x5', field_type='date', field_format='CCYYMMDD')
```

```
da.add_field('bad', field_type='int')
```

Then the *input_fn* can be created:

```
make_input_fn(da.dictionary, '/home/will/tmp/inp.py', dep_var='bad', dates='ccyymmdd')
```

The call specifies that 'bad' is the dependent variable for the analyis. The *input_fn* will return this tensor separately. It also specifies that all dates should be converted to *int* in the format CCYYMMDD.

Here is the *input_fn* created b *make_input_fn*:

```
import tensorflow as tf
def input_fn(files, batch_size=1, shuffle=0, skip=0, take=0, num_epochs=1, parallel_
↪calls=4):
    def parse_input(proto):
        shape_np = 1
        keys_to_features = {}
        keys_to_features['x1'] = tf.FixedLenFeature((shape_np), tf.float32)
        keys_to_features['x2'] = tf.FixedLenFeature((shape_np), tf.float32)
        keys_to_features['x3'] = tf.FixedLenFeature((shape_np), tf.float32)
        keys_to_features['xd'] = tf.VarLenFeature(tf.string)
        keys_to_features['x5'] = tf.FixedLenFeature((shape_np, 3), tf.int64)
        keys_to_features['bad'] = tf.FixedLenFeature((shape_np), tf.int64)
        keys_to_features['xe'] = tf.FixedLenFeature((shape_np), tf.int64)
        parsed_features = tf.parse_single_example(proto, keys_to_features)
        dep_var = parsed_features.pop('bad')
        return parsed_features, dep_var

    ds = tf.data.TFRecordDataset(files).skip(skip)
    if take > 0:
        ds = ds.take(take)
    if shuffle > 0:
        ds = ds.suffle(buffer_size=shuffle)
    ds = ds.map(parse_input, num_parallel_calls=parallel_calls)
    if num_epochs > 1:
        ds = ds.repeat(num_epochs)
    ds = ds.batch(batch_size)
    iter = ds.make_one_shot_iterator()
    (features, dep_var) = iter.get_next()
    e = features.pop('xd')
    cols = e.dense_shape[1]
    d = tf.sparse_to_dense(e.indices, (batch_size, cols), e.values, '')
    h = tf.reduce_join(d,1)
    features['xd'] = h
    a = features['x5']
    yr = tf.slice(a, [0, 0, 0], [batch_size, 1, 1])
    mon = tf.slice(a, [0, 0, 1], [batch_size, 1, 1])
    day = tf.slice(a, [0, 0, 2], [batch_size, 1, 1])
    yrmonday = tf.add(tf.add(tf.multiply(yr,10000),tf.multiply(mon,100)),day)
    features['x5'] = yrmonday
    return features, dep_var
```

As read, the tensor *xd* is a bytes list. It is removed from the features dictionary, converted to strings, and returned to the dictionary.

The call below creates the *model_columns* function:

```
make_model_columns(da.dictionary, '/home/will/tmp/col.py')
```

The following function is produced by this call:

```
import tensorflow as tf
def model_columns(include_columns):
    """
    :param include_columns: features in the model
    :type include_columns: dict

    include_columns has keys equal to the feature name
    the entry is how to handle it. Options are
    - hash_size
    - embed_size
    """
    columns = []
    if 'x1' in include_columns.keys():
        try:
            boundaries = include_columns['x1']['boundaries']
        except:
            boundaries = None
        if boundaries is not None:
            tmp_field = tf.feature_column.numeric_column('x1')
            x1 = tf.feature_column.bucketized_column(tmp_field, boundaries)
        else:
            x1 = tf.feature_column.numeric_column('x1')
        columns += [x1]
    if 'x2' in include_columns.keys():
        try:
            boundaries = include_columns['x2']['boundaries']
        except:
            boundaries = None
        if boundaries is not None:
            tmp_field = tf.feature_column.numeric_column('x2')
            x2 = tf.feature_column.bucketized_column(tmp_field, boundaries)
        else:
            x2 = tf.feature_column.numeric_column('x2')
        columns += [x2]
    if 'x3' in include_columns.keys():
        try:
            boundaries = include_columns['x3']['boundaries']
        except:
            boundaries = None
        if boundaries is not None:
            tmp_field = tf.feature_column.numeric_column('x3')
            x3 = tf.feature_column.bucketized_column(tmp_field, boundaries)
        else:
            x3 = tf.feature_column.numeric_column('x3')
        columns += [x3]
    if 'xd' in include_columns.keys():
        vocab = []
        vocab += ['now now']
        vocab += ['now then']
        vocab += ['then now']
        vocab += ['then then']
        vocab += ['X']
        n_oov = 0
        tmp_field = tf.feature_column.categorical_column_with_vocabulary_list('xd',␣
↪vocab, num_oov_buckets=n_oov)
```

```
        try:
            embed_size = include_columns['xd']['embed_size']
        except:
            embed_size = 0
        if embed_size > 0:
            xd = tf.feature_column.embedding_column(tmp_field, embed_size)
        else:
            xd = tf.feature_column.indicator_column(tmp_field)
        columns += [xd]
    if 'x5' in include_columns.keys():
        try:
            hash_size = include_columns['x5']['hash_size']
        except:
            hash_size = 1000000
        tmp_field = tf.feature_column.categorical_column_with_hash_bucket('x5', hash_
↪bucket_size=hash_size, dtype=tf.int64)
        try:
            embed_size = include_columns['x5']['embed_size']
        except:
            embed_size = 0
        if embed_size > 0:
            x5 = tf.feature_column.embedding_column(tmp_field, embed_size)
        else:
            x5 = tf.feature_column.indicator_column(tmp_field)
        columns += [x5]
    if 'bad' in include_columns.keys():
        try:
            hash_size = include_columns['bad']['hash_size']
        except:
            hash_size = 1000000
        tmp_field = tf.feature_column.categorical_column_with_hash_bucket('bad', hash_
↪bucket_size=hash_size, dtype=tf.int64)
        try:
            embed_size = include_columns['bad']['embed_size']
        except:
            embed_size = 0
        if embed_size > 0:
            bad = tf.feature_column.embedding_column(tmp_field, embed_size)
        else:
            bad = tf.feature_column.indicator_column(tmp_field)
        columns += [bad]
    if 'xe' in include_columns.keys():
        try:
            hash_size = include_columns['xe']['hash_size']
        except:
            hash_size = 1000000
        tmp_field = tf.feature_column.categorical_column_with_hash_bucket('xe', hash_
↪bucket_size=hash_size, dtype=tf.int64)
        try:
            embed_size = include_columns['xe']['embed_size']
        except:
            embed_size = 0
        if embed_size > 0:
            xe = tf.feature_column.embedding_column(tmp_field, embed_size)
        else:
            xe = tf.feature_column.indicator_column(tmp_field)
        columns += [xe]
    return columns
```

The user passes a dictionary, *include_columns*. The keys to the dictionary are the features to use in the model. The dictionary entries themselves are dictionaries of options ('**feature dictionaries**').

For features of type *float*:

- To include the feature as-is, there only needs to be entry with the feature name as key.
- To bucketize the feature, the feature dictionary needs an entry with a key 'boundaries'. The value of the key is a list of boundary values for the buckets.

For features of type *str*:

You cannot include a *tf.string* directly into a DNN model (note: you can in a linear model). There are two options:

- indicator columns. The values of the string are mapped to a one-hot tensor whose length is the number of different values of the string. This is the default behavior if there is no key called *embed_size* in the feature dictionary. If the data_reader entry for the field includes the list of legal values, then these are built in to the feature. If the user has specified an 'illegal replacement' value for the feature in the data_reader dictionary, this is also included. If the user has not done this, then a special 'out of value' bucket is created for the feature. If the user has not specified legal values, then there is no vocabulary list to create. In this case, the hash index method is used. The user may specify the size of the hash array using the feature key 'hash_size'.
- embedded layer. If the feature dictionary contains a key 'embed_size' then an embedding layer is created with the number of elements specified by 'embed_size'. In this case, the model maps each unique value of the string to a tensor of size 'embed_size'. The entries of each tensor are treated as parameters to the model and are tuned during the model fit.

The options for features columns within TensorFlow are documented here.

The code below shows how these two functions are used:

```
from modeling_tools.functions import ks_calculate, decile_plot
import shutil
import numpy as np
import time
import pkg_resources

# specifies the model and optimization options.  The model_columns function is used␣
↪here.
def build_estimator(model_dir, include_columns):
    columns = model_columns(include_columns)

    run_config = tf.estimator.RunConfig().replace(
        session_config=tf.ConfigProto(device_count={'GPU': 0}))

    hidden_units = [50, 25]

    model = tf.estimator.DNNClassifier(
        model_dir=model_dir,
        feature_columns=columns,
        hidden_units=hidden_units,
        optimizer=tf.train.FtrlOptimizer(
            learning_rate=0.1,
            l2_regularization_strength=1.0,
            l1_regularization_strength=1.0
        )
    )
    return model

# builds the model, evaluates it, makes a KS and decile plot.
def bev(model_dir, data_file, include_columns, train_epochs=10, batch_size=100, build_
↪take=1000, val_skip=0,
```

```
        val_take=1000, steps=10):
    # Clean up the model directory if present
    shutil.rmtree(model_dir, ignore_errors=True)
    model = build_estimator(model_dir, include_columns)

    epochs_per_eval = 1
    start_time = time.time()
    for n in range(train_epochs // epochs_per_eval):
        print('training step ' + str(n))
        # steps is the number of batches to draw for the training epoch.
        # It corresponds to the 'steps' axis on TensorBoard.
        model.train(input_fn=lambda: input_fn(data_file, batch_size=batch_size,␣
↪take=build_take), steps=steps)
        model.evaluate(input_fn=lambda: input_fn(data_file, batch_size=batch_size,␣
↪take=build_take), steps=steps)

    # prints out the paramaters to the model just fit
    for name in model.get_variable_names():
        # skip Follow The Regularized Leader values
        if name.upper().find('FTRL') < 0:
            print(name)
            print(model.get_variable_value(name))

    elapsed_time = time.time() - start_time
    print('elapsed time: {:.3f} seconds'.format(elapsed_time))

    # pull a sample not used in model build
    val_batch = val_take - val_skip
    yh = list(model.predict(input_fn=lambda: input_fn(data_file, batch_size=val_batch,
↪ skip=val_skip, take=val_take)))
    print(len(yh))
    pr = np.array([x['probabilities'][1] for x in yh])
    print('train')
    with tf.Session() as sess:
        val_data_iter = input_fn(data_file, batch_size=val_batch, skip=val_skip,␣
↪take=val_take)
        val_data = sess.run(
            val_data_iter)  # cd is a tuple. First entry is a dict of features,␣
↪second a numpy array of labels
        y = val_data[1]

    # a couple sanity checks
    print(pr.shape)
    print(pr.mean())

    # p tensorboard.main --logdir='/tmp/model'
    # to load labels for the projector for embeddings, create a tab separated file␣
↪like:
    # index\t name
    # 0\t hi

    pr = pr.squeeze()
    y = y.squeeze()
    ks_calculate(pr, y, True, False)
    decile_plot(pr, y, wait=False)

##################################
# build and evaluate model
```

```
###################################
#
# all the model info is output here. Can use it later and also look at it on␣
↪TensorBoard.
model_dir = '/tmp/model'
#
tfrecord_filename = test_data_path = pkg_resources.resource_filename('data_reader',
↪'test_data/') + 'testxy.tfr'

import sys
# this is where data_reader was instructed to put the input_fn and model_columns␣
↪functions
sys.path.append('/home/will/tmp')
from inp import input_fn
from col import model_columns

# specify the model features
include_columns = {}
include_columns['x1'] = 'yes'
include_columns['x2'] = 'yes'
include_columns['x3'] = 'yes'
# add check for date part
include_columns['x5'] = {'embed_size': 0, 'hash_size': 100}
# use an embedding layer. If set to '0' or do not include the 'embed_size' key, it␣
↪will use one-hot tensors (indicator variables).
include_columns['xd'] = {'embed_size': 1}
bev(model_dir, tfrecord_filename, include_columns, train_epochs=10, batch_size=100,␣
↪build_take=75000, val_skip=75000, val_take=100000,
    steps=1000)
```

CHAPTER

TWO

DATA READER

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

D