How DeepDive applications are typically developed

All successful DeepDive applications that achieve high quality are never developed in a single shot. It is common that users start with a basic first version that usually gives relatively poor results. Then, the quality is **improved iteratively** through *error analysis* and debugging. DeepDive provides a suite of tools that aim to keep each development iteration as quick and easy as possible.

1. Write

First of all, the user has to write in DDlog a schema that describes the input data and the data to be produced, along with how data should be processed and transformed. Data transofmation rules as well as *user-defined functions* (UDFs) written in Python or any other language can be used for defining the data processing operations. Then, using the processed data, a statistical inference model describing a set of random variables and their correlations can be defined—also in DDlog—to specify what kind of predictions are to be made by the system.

How to write each of these parts in a DeepDive application is described in the following pages:

- Defining data flow in DDlog
- Writing user-defined functions in Python
- Specifying a statistical model in DDlog

2. Run

As soon as a new piece of the DeepDive application is written, the user can compile and run it incrementally. For example, after declaring the schema for the input text corpus, the actual data can be loaded into the database and queried. New tranformation rules added by the user can be executed incrememtally. The statistical inference model can be constructed from the data according to the DDlog specification. The model's parameters can be learned or reused to make predictions about the marginal probabilities of new data points.

DeepDive provides a suite of commands and conventions to carry out these operations. These are documented in the following pages:

- Compiling a DeepDive application
- Managing input data and data products

- Controlling execution of data processing
- Learning and inference with the statistical model

3. Evaluate / Debug

Based on our observation of several successful DeepDive applications, we can say that the more rapid the user moves through the development cycle, the more quick she achieves high-quality. Users can identify the most common mode of errors after each iteration by evaluating the predictions made by the system. This is done via formal error analysis supported by interactive tools. This enables the user to focus on fixing the mistakes that caused the observed errors, instead of spending her time poorly on some corners that might only give marginal improvement.

DeepDive provides a suite of tools and guides to accelerate the development process, which are documented in the following pages:

- Debugging UDFs
- Labeling data products
- Browsing data
- Monitoring descriptive statistics of data
- Calibration
- Generating negative examples

DeepDive application structure

A DeepDive application is a directory that contains the following files and directories:

app.ddlog

This file can be thought as the blueprint of a DeepDive application. DDlog declarations and rules written in this file tell DeepDive:

- What each relation looks like and how one is derived from others.
- What user-defined functions are there, what input/output schema they expect, and their implementation details.
- What random variables are to be modeled, how they are correlated.

deepdive.conf

Extra configuration not expressed in the DDlog program is in this file. Extractors, and inference rules can also be written in HOCON syntax in this file, although DDlog is the recommended way. See the Configuration Reference for full details.

db.url

A URL representing the database configuration is supposed to be stored in this file. Environment variable `$DEEPDIVE_DB_URL` overrides the URL stored this file. For example, the following URL can be the line stored in it:

```
postgresql://user:password@localhost:5432/database_name
```

PostgreSQL SSL

SSL connections for PostgreSQL can be enabled by setting parameter `ssl` as true in the URL, e.g.:

```
postgresql://user:password@localhost:5432/database_name?ssl=true
```

If you use a self-signed certificate, you may want to disable validation with an extra `sslfactory` parameter:

```
postgresql://user:password@localhost:5432/database_name?ssl=true&sslfactory=org.postgresql.ssl.NonValidatingFactory
```

input/

Any data to be processed by this application is suggested to be kept under this directory.

udf/

Any user-defined function (UDF) code is suggested to be kept under this directory. They can be referenced from deepdive.conf with path names relative to the application root.

run/

Each run/execution of the DeepDive application has a corresponding subdirectory under this directory whose name contains the timestamp when the run was started, e.g., `run/20150618/223344.567890/`. All output and log files that belong to the run are kept under that subdirectory. There are a few symbolic links with mnemonic names to the most recently started run (`run/LATEST`), last successful run (`run/FINISHED`), last failed run (`run/ABORTED`) for handy access.

Other files

schema.json and schema.sql

Data-Definition Language (DDL) statements for setting up the underlying database tables should be kept in this file. This may be omitted when the application is written in DDlog.

input/init.sh

In addition to the data files, there should be an executable script that knows how to load the data here to the database once its tables are created.

Writing a DeepDive Application

Defining data flow in DDlog

DDlog is a higher-level language for writing DeepDive applications in a succinct, Datalog-like syntax. Here we focus on describing the general language features for defining the data flow in a DeepDive application. The DDlog language constructs for using user-defined functions and specifying a statistical inference model are described in another page. A complete end-to-end example written in DDlog is described in the tutorial in greater detail.

DDlog program

A DDlog program is a collection of declarations and rules. Each declaration and rule ends with a period (`.`). Comments in DDlog begin with a hash (`#`) character. The order of the rules have no meaning. A DDlog program consists of the following parts:

1. Schema declarations for relations
2. Normal derivation rules
    1. Relation derived (head atom)
    2. Relations used (body atoms)
    3. Conditions
3. User defined functions (UDFs)

    1. Function declarations
    2. Function call rules
4. Inference rules

All DDlog code should be placed in a file named `app.ddlog` under the DeepDive application.

Schema declarations

First of all, the schema of the *relations* defined and used throughout the program should be declared. These relations are mapped to tables in the database configured for the DeepDive application. The documentation for managing the database introduces various ways provided by DeepDive to interact with it. The order of declarations doesn't matter, but it's a good idea to place them at the beginning because that makes it easier to understand the rest of the rules that refer to the relations.

```
relation_name(
  column1_name  column1_type,
  column2_name  column2_type,
  ...
).
```

Similar to a SQL table definition, a schema declaration is just the name of a relation followed by a comma separated list of the column names and their types. Currently, DDlog maps the types directly to SQL, so any type supported by the underlying database can be used, e.g., PostgreSQL's types.

Below is a realistic example.

```
article(
  id     int,
  length int,
  author text,
  words  text[]
).
```

Here we are defining a relation named "article" with four columns, named "id", "length", "author" and "words" respectively. Each column has it's own type, here utilizing `int`, `text` and `text[]`.

Normal derivation rules

Typical Datalog-like rules are used for defining how a relation is derived from other relations. For example, the following rule states that the tuples of relation `Q` are derived from relations `R` and `S`.

```
Q(x, y) :- R(x, y), S(y).
```

Here, `Q(x, y)` is the *head atom*, and `R(x, y)` and `S(y)` are *body atoms*. `x` and `y` are *variables* in the rule. The head is the relation being defined, and the body is a conjunction of predicates or conditions that bound the variables used for the definition, separated by commas (`,`). In this example, the second column of `R` is *unified* with the first column of `S`, i.e., the body is an *equi-join* between relations `R` and `S`.

Expressions

In the atom columns, both variables and expressions can be used. DDlog's semantics is based on unification of variables and expressions, i.e., variables with the same name unify the columns, and expressions in the body unify the values with the corresponding columns. More concretely, expressions in the head atom decide values for the columns of the head relation, while expressions in the body atom constrain the values of the body columns.

Examples

A few examples of expressions are shown below.

```
a(k int).
b(k int, p text, q text, r int).
c(s text, n int, t text).

Q("test", f(123), id) :- a(id), b(id, x,y,z), c(x || y,10,"foo
").

R(TRUE, power(abs(x-z), 2)) :- a(x), b(y, _, _, _).

S(if l > 10 then TRUE
else if l < 10 then FALSE
else NULL end) :- R ( _, l).
```

Here tuples of string literal "text", a function `f` applied to 123, and `id` bound by the body of the rule are added to relation `Q`. Then, we add to the relation `R` the boolean `TRUE` and operations from variables in `a` and `b`. Finally, we fill out the relation `S` by a condition value according to the second variable in `R`.

We observe that sophisticated queries are realizable in ddlog. The following paragraphs explain in details the different possible operations in ddlog.

Constant literals

Numbers, text strings, boolean values, null value are expressed as follows:

- Numeric literals, e.g.: `123`, `-456.78`.
- String literals are surrounded by double quotes (`"`), e.g.: `"Hello, World!"`.

  Preceding backslash (`\`) is used as the escape sequence for including double quote itself as well as backslash:

  ```
  "For example, \"There're backslashes (\\) before the surro
  unding double quotes.\""
  ```

  is how the following text is written as string literal:

  ```
  For example, "There're backslashes (\) before the surround
  ing double quotes."
  ```

- Boolean literals: `TRUE`, `FALSE`.
- Null literal: `NULL`.

Operators

Derived values are expressed as *`expr op expr`* or *`uop expr`* where *op* is a binary operator and *uop* is a unary operator among the following ones:

Binary operators

- Numerical arithmetic `+`, `-`, `*`, `/`
- String concatenation `||`

Unary operators

- Negative sign `-`
- Negation of boolean values `!`

Expressions can be arbitrarily nested, and parentheses can surround them *`(expr)`* to make the operator association clear.

Comparisons

- Equality `=`
- Distinction `!=`
- Inequality `<`, `<=`, `>`, `>=`
- Null *`expr`* `IS NULL`, *`expr`* `IS NOT NULL`
- String pattern *`exprText`* `LIKE` *`exprPattern`*

Conditional values

It is possible to express values determined by conditional cases with a syntax of `if` *`expr`* `then` *`expr`* `else` *`expr`* `end`. For example, the following expresses a value 1 when variable `x` is positive, -1 when it's negative, otherwise `NULL`.

```
if x > 0 then 1
else if x < 0 then -1
else NULL
end
```

Function applications

More complex values can be expressed as applying predefined functions to a tuple of expressions with the following syntax: *`name(expr,expr, ...)`*. Currently, DDlog directly maps the function *name* directly to SQL, so any function defined by the underlying database can be used, e.g., PostgreSQL's functions. For example, the following takes the absolute value using the `abs` function and the `power` function to square the value.

```
power(abs(x - y), 2)
```

Type casts

When a value needs to be turn into a different type, e.g., integer into a floating point, or vice versa, then a type casting syntax can follow the expression, i.e., *`expr :: type`*. For example, the following turns the variable in the denominator into a floating point number, so the ratio can be correctly computed:

```
num_occurs / total :: FLOAT
```

Placeholder

A placeholder `_` can be used in the columns of body atoms, indicating no variable is relevant to that column. This is useful if only a subset of columns are needed for the derivation. For example, the following rule uses only two columns of `R` to define `Q`, and the rest of the columns are not bound to any variable:

```
Q(x, y, x + y) :- R(x, _, _, y, _, _, _).
```

Conditions

Conditions are put at the same level of body atoms, and collectively with body atoms bound the variables used in the head. Conjunctive conditions are separated by commas (`,`), and disjunctive conditions are separated by semicolon (`;`). Exclamation (`!`) can precede a condition to negate it. Conditions can be arbitrarily nested by putting inside square brackets (`[cond, cond, ...; cond; ...]`). For example, the following rule takes the values for variable `x` from the first column of relation `b` that satisfy a certain condition with the fourth column of `b`.

```
Q(x) :- b(x,_,_,w), [x + w = 100; [!x > 50, w < 10]].
```

Disjunctions

Disjunctive cases can be expressed as multiple rules defining the same head, or as one rule having multiple bodies separated by semicolons (`;`). Take the following two rules as an example.

```
Q(x, y) :- R(x, y), S(y).
Q(x, y) :- T(x, y).
```

They can be written as a single rule:

```
Q(x, y) :- R(x, y), S(y); T(x, y).
```

Aggregation

Instead of deriving one head tuple for every combination of variables satisfying the body, a group of head tuples can be reduced into one by taking aggregate values for some of the columns. By surrounding some columns of the head with an aggregate function, the aggregates are computed from each group of column values determined by the values of other columns not under aggregation. DDlog recognizes the following aggregate functions:

- `SUM`
- `MAX`

- `MIN`
- `ARRAY_AGG`
- `COUNT`

For example, the following rule groups all values of the third column of `R` by the first and second columns, then take the maximum value for deriving a tuple for `Q`.

```
Q(a,b,MAX(c)) :- R(a,b,c).
```

Select distinct

In order to select only distinct elements from other relations, the operator `*:-` can be used. For instance, let's consider the following rule:

```
Q(x,y) *:- R(x, y).
```

In this rule, only disctinct couple of variables `(x,y)` from the relation `R` will be inserted in the head `Q`.

Quantifiers

Existential/Universal

Another way to derive a reduced number of tuples for the head is to use *quantified bodies*. Body atoms and conditions can be put under *quantifiers* to express existential (`EXISTS`) or universal (`ALL`) constraints over certain variables with the following syntax:

```
body, ..., EXISTS[body, body, ...], body, ...
body, ...,    ALL[body, body, ...], body, ...
```

For the variables used both inside and outside of the quantification, the relations inside the quantifier express constraints placed on the variables already bound by some relation outside the quantifier. `EXISTS` expresses the constraint for such variables that there must exist a combination of values satisfying the quantified body, while `ALL` means for each combination of values for the variables all possible bindings for other variables within the quantified body must be satisfied. This makes it possible to express complex conditions using derived values in relations. Here are two example rules using quantifiers:

```
P(a) :- R(a, _), EXISTS[S(a, _)].
Q(a) :- R(a, b),    ALL[S(a, c), c > b].
```

The first rule means `P` contains all the first columns of relation `R` that also appears at least once in the first column of relation `S`. The second rule means `Q` contains all the first columns of

relation `R` if tuples of relation `S` with the same value on the first column always have a second column that is greater than the value for the second column of `R`.

Optional

`OPTIONAL` quantifier can be used when certain body atoms are optional. It means that a tuple for the head relation should still be derived even if there may not be tuples in certain body relations. The variables appear only in the optional bodies are bound to `NULL` when they cannot be satisfied. This quantifier roughly corresponds to *outer joins* in SQL.

Here is an example rule using an optional quantifier:

```
Q(a, c) :- R(a, b), OPTIONAL[S(a, c), c > b].
```

It means `Q` contains every value of the first column of relation `R` paired with all values for the second column of relation `S` tuples that have the same value on the first column and the second column is greater than that of `R`, or `NULL` if no such value exists.

Writing user-defined functions in Python

DeepDive supports *user-defined functions* (UDFs) for data processing, in addition to the normal derivation rules in DDlog. UDF can be any program that takes TAB-separated JSONs (TSJ) format or TAB-separated values (TSV or PostgreSQL's text format) from stdin and prints the same format to stdout. TSJ puts a fixed number of JSON values in a fixed order in each line, separated by a TAB. TSJ can be thought as a more efficient encoding than simply putting a JSON object per line, which has to repeat the field names on every line, especially when the fixed data schema for every line is known and fixed ahead of time.

The following sections describe DeepDive's recommended way of writing UDFs in Python and how they are used in DDlog programs.

Using UDFs in DDlog

To use user-defined functions in DDlog, they must be declared first then called using special syntax.

First, let's define the schema of the two relations for our running example.

```
article(
```

```
    id     int,
    url    text,
    title  text,
    author text,
    words  text[]
).

classification(
    article_id int,
    topic      text
).
```

In this example, let's suppose we want to write a simple UDF to classify each `article` into different topics by adding tuples to the relation `classification`. The two following sections detail how to declare such a function and how to call it in DDlog.

Function declarations

A function declaration includes input/output schema as well as a pointer to its implementation.

```
function <function_name> over (<input_var_name> <input_var_type>,...)
    returns [(<output_var_name> <output_var_type>,...) | rows like <relation_name>]
    implementation "<executable_path>" handles tsj lines.
```

In our example, suppose we will use only the `author` and `words` of each `article` to determine the topic identified by its `id`, and the implementation will be kept in an executable file with relative path `udf/classify.py`. The exact declaration for such function is shown below.

```
function classify_articles over (id int, author text, words text[])
    returns (article_id int, topic text)
    implementation "udf/classify.py" handles tsj lines.
```

Notice that the column definitions of relation `classification` are repeated in the `returns` clause. This can be omitted by using the `rows like` syntax as shown below.

```
function classify_articles over (id int, author text, words text[])
```

```
    return rows like classification
    implementation "udf/classify.py" handles tsj lines.
```

Also note that the function input is similar to the `articles` relation, but some columns are missing. This is because the function will not use the rest of the columns as mentioned before, and it is a good idea to drop unnecessary values for efficiency. Next section shows how such input tuples can be derived and fed into a function.

Function call rules

The function declared above can be called to derive tuples for another relation of the output type. The input tuples for the function call are derived using a syntax similar to a normal derivation rule. For example, the rule shown below calls the `classify_articles` function to fill the `classification` relation using a subset of columns from the `articles` relation.

```
classification += classify_articles(id, author, words) :-
    article(id, _, _, author, words).
```

Function call rules can be viewed as special cases of normal derivation rules with different head syntax, where `+=` and a function name is appended to the head relation name.

Writing UDFs in Python

DeepDive provides a templated way to write user-defined functions in Python. It provides several Python function decorators to simplify parsing and formatting input and output respectively. The Python generator to be called upon every input row should be decorated with `@tsj_extractor`, i.e., before the `def` line `@tsj_extractor` should be placed. (A Python generator is a Python function that uses `yield` instead of `return` to produce an iterable of multiple results per call) The input and output column types expected by the generator can be declared using the argument default values and `@returns` decorator, respectively. They tell how the input parser and output formatter should behave.

Let's look at a realistic example to describe how exactly they should be used in the code. Below is a near-complete code for the `udf/classify.py` declared as the implementation for the DDlog function `classify_articles`.

```
#!/usr/bin/env python
from deepdive import *  # Required for @tsj_extractor and @returns


compsci_authors = [...]
bio_authors     = [...]
```

```
bio_words       = [...]


@tsj_extractor  # Declares the generator below as the main func
tion to call
@returns(lambda # Declares the types of output columns as decla
red in DDlog
        article_id = "int",
        topic      = "text",
    :[])
def classify(   # The input types can be declared directly on ea
ch parameter as its default value
        article_id = "int",
        author     = "text",
        words      = "text[]",
    ):
    """
    Classify articles by assigning topics.
    """
    num_topics = 0

    if author in compsci_authors:
        num_topics += 1
        yield [article_id, "cs"]

    if author in bio_authors:
        num_topics += 1
        yield [article_id, "bio"]
    elif any (word for word in bio_words if word in words):
        num_topics += 1
        yield [article_id, "bio"]
```

```
    if num_topics == 0:
        yield [article_id, None]
```

This simple UDF assigns a topic to articles based on their authors' membership in known categories. If the author is not recognized, we try to look for words that appear in a predefined set. Finally, if nothing matches, we simply put it into another catch-all topic. Note that the topics themselves here are completely user defined.

Notice that to use these Python decorators you'll need to have `from deepdive import *`. Also notice that the types of input columns can be declared as default values for the generator parameters in the same way as `@returns`.

@tsj_extractor decorators

The `@tsj_extractor` decorator should be placed as the first decorator for the main generator that will take one input row at a time and `yield` zero or more output rows as list of values. This basically lets DeepDive know which function to call when running the Python program. (For TSV, there's `@tsv_extractor` decorator, but TSJ is strongly recommended.)

Caveats

Generally, this generator should be placed at the bottom of the program unless there are some cleanup or tear-down tasks to do after processing all the input rows. Any function or variable used by the decorated generator should be appear before it as the `@tsj_extractor` decorator will immediately start parsing input and calling the generator. The generator should not `print` or `sys.stdout.write` anything as that will corrupt the standard output. Instead, `print >>sys.stderr` or `sys.stderr.write` can be used for logging useful information. More information can be found in the debugging-udf section

Parameter default values and @returns decorator

To parse the input TSJ lines correctly into Python values and format the values generated by the `@tsj_extractor` correctly in TSJ the column types need to be written down in the Python program. They should be consistent with the function declaration in DDlog. The types for input columns can be declared directly in the `@tsj_extractor` generator's signature as parameter default values as shown in the example above. Arguments to the `@returns` decorator can be either a list of name and type pairs or a function with all parameters having their default values set as its type. The use of `lambda` is preferred because the list of pairs require more symbols that clutter the declaration. For example, compare above with `@returns([("article_id", "int"), ("topic", "text")])`. The reason `dict(column="type", ... )` or `{ "column": "type", ... }` do not work is because Python forgets the order of the columns with those syntax, which is crucial for the TSJ parser and formatter. The passed function is never called so

the body can be left as any value, such as empty list (`[]`). DeepDive also provides an `@over` decorator for input columns symmetric to `@returns`, but use of this is not recommended as it incurs redundant declarations.

Running and debugging UDFs

Once a first cut of the UDF is written, it can be run using the `deepdive do` and `deepdive redo` commands. For example, the `classify_articles` function in our running example to derive the `classification` relation can be run with the following command:

```
deepdive redo classification
```

This will invoke the Python program `udf/classify.py`, giving as input the TSJ rows holding three columns of the `article` table, and taking its output to add rows to the `classification` table in the database.

There are dedicated pages describing more details about running these UDFs and debugging these UDFs.

Specifying a statistical model in DDlog

Every DeepDive application can be viewed as defining a statistical inference problem using input data and data derived by a series of data processing steps. This document describes (a) how to declare random variables for a DeepDive application's statistical model, (b) how to define their scope as well as supervision labels, and (c) how to write inference rules for specifying features and correlations.

Variable declarations

DeepDive requires the user to specify the name and type of the *variable relations* that hold random variables used during probabilistic inference. Currently DeepDive supports Boolean (i.e., Bernoulli) variables and Categorical variables. Variable relations are declared in `app.ddlog` with a small twist to the syntax used for declaring normal relations.

Boolean variables

A question mark after the relation name indicates that it is a variable relation containing random variables rather than a normal relation used for loading or processing data to be later used by the model. The columns of the variable relation serve as a key. The following is an example declaration of a relation of Boolean variables.

```
has_spouse?(p1_id text, p2_id text).
```

This declares a variable relation named `has_spouse` where each unique pair of `(p1_id, p2_id)` represents a different random variable in the model.

Categorical variables

DeepDive supports categorical variables, which take integer values ranging from 0 to a user-specified upper bound. The variable relation is declared similarly as a Boolean variable except that the declaration is followed by a `Categorical(N)` where `N` is the number of categories the variables can take, defining the size of the domain. Each variable can take values from 0, 1, ..., `N`-1. For instance, in the chunking example, a categorical variable of 13 possible categories is declared as follows:

```
tag?(word_id bigint) Categorical(13).
```

Scoping and supervision rules

After declaring a variable relation, its *scope* needs to be defined along with the *supervision labels*. That means, (a) all possible values for the variable relation's columns must be defined by deriving them from other relations, and (b) whether a random variable in the relation is true or false (Boolean), or which value it takes from its domain of categories (Categorical) must be defined using a special syntax. For these scoping and supervision rules, a syntax very similar to normal derivation rules is used for defining the variable relation, except that the head is followed by an `=` sign and an expression that corresponds to the random variable's label. For instance, in the spouse example, we scope the `has_spouse` variable by the following rule:

```
has_spouse(p1_id, p2_id) = NULL :-
    spouse_candidate(p1_id, _, p2_id, _).
```

This means all distinct `p1_id` and `p2_id` pairs found in the `spouse_candidate` relation is considered the scope of all `has_spouse` random variables in the model. By using a `NULL` expression on the right hand side, they are considered as unsupervised variables.

On the other hand, the following similar looking rule provides the supervision labels using a sophisticated expression.

```
has_spouse(p1_id, p2_id) = if l > 0 then TRUE
                      else if l < 0 then FALSE
                      else NULL end :- spouse_label_resolved(p1_id, p2_id, l).
```

This rule is basically doing a *majority vote*, turning aggregate numbers computed from `spouse_label_resolved` relation into Boolean labels.

Inference rules

Inference rules specify features for a variable and/or the correlations between variables. They are basically the templates for the factors in the factor graph, telling DeepDive how to *ground* them based on what input and derived data. Again, these rules extend the syntax of normal derivation rules and allow the type of the factor to be specified in the rule's head, preceded by a `@weight` declaration as shown below.

```
@weight(...)
FACTOR_HEAD :- RULE_BODY.
```

Here, *RULE_BODY* denotes a typical conjunctive query also used for normal derivation rules in DDlog. *FACTOR_HEAD* denotes the part where more than one variable relations can appear with a special syntax. Let's first look at the simplest case of describing one variable in the head of an inference rule.

Specifying features

In common cases, one wants to model the probability of a Boolean variable being true using a set of features. Expressing this kind of binary classification problem is very simple in DDlog. By writing a rule with just one variable relation in the head, DeepDive creates in the model a *unary factor* that connects to it. The weight of this unary factor is determined by a user-defined feature. For instance, in the spouse example, there is an inference rule specifying features for the `has_spouse` variables written as:

```
@weight(f)
has_spouse(p1_id, p2_id) :-
    spouse_candidate(p1_id, _, p2_id, _),
    spouse_feature(p1_id, p2_id, f).
```

This rule means that:

- A factor should be created for each pair of person mentions found in the `spouse_candidate` relation and each of the corresponding features found in `spouse_feature` relation.
- Each of those factors connects to a single `has_spouse` variable identified by a pair of mentions `(p1_id, p2_id)` originating from the `spouse_candidate` relation.
- The feature `f` for a factor determines a *weight* (to be learned for this particular rule) that translates into the factor's potential, which in turn influences the probability of the connected `has_spouse` variable being true or not.

Specifying correlations

Now, in almost every problem, the variables are correlated with each other in a special way, and it is desirable to enrich the model with this domain knowledge. Such correlations can be modeled by creating certain types of factors that connect multiple correlated variables together. This is where a richer syntax in the *FACTOR_HEAD* comes into play. DDlog borrows a lot of syntax from Markov Logic Networks, and hence, first-order logic.

For example, the following rule in the smoke example correlates two variable relations.

```
@weight(3)
smoke(x) => cancer(x) :-
   person(x).
```

This rule expresses that if a person smokes, there is an *implication* that he/she will have cancer. Here, a constant `3` is used in the `@weight` to express some level of confidence in this rule, instead of learning the weight from the data (explained later).

Implication

Logical implication or consequence of two or more variables can be expressed using the following syntax.

```
@weight(...)  P(x) => Q(y)              :- RULE_BODY.
@weight(...)  P(x), Q(y) => R(z)        :- RULE_BODY.
@weight(...)  P(x), Q(y), R(z) => S(k)  :- RULE_BODY.
```

Disjunction

Logical disjunction of two or more variables can be expressed using the following syntax.

```
@weight(...)  P(x) v Q(y)          :- RULE_BODY.
@weight(...)  P(x) v Q(y) v R(z)   :- RULE_BODY.
```

Conjunction

Logical conjunction of two or more variables can be expressed using the following syntax.

```
@weight(...)  P(x) ^ Q(y)          :- RULE_BODY.
```

```
@weight(...)  P(x) ^ Q(y) ^ R(z)  :- RULE_BODY.
```

Equality

Logical equality of two variables can be expressed using the following syntax.

```
@weight(...)  P(x) = Q(y)  :- RULE_BODY.
```

Negation

Whenever a rule has to refer to a case when the Boolean variable is false (also referred to as a *negative literal*), then it can be negated using a preceding `!` as shown below.

```
@weight(...)    P(x) => ! Q(x)  :- RULE_BODY.
@weight(...)  ! P(x) v  ! Q(x)  :- RULE_BODY.
```

Multinomial factors (STALE. NEEDS REVAMP.)

DeepDive has limited support for expressing correlations of categorical variables. The introduced syntax above can be used only for expressing correlations between Boolean variables. For categorical variables, DeepDive only allows the conjunction of the variables each taking a certain category value being true to be expressed using a special syntax shown below:

```
@weight(...)  Multinomial( P(x), Q(y) )        :- RULE_BODY.
@weight(...)  Multinomial( P(x), Q(y), R(z) )  :- RULE_BODY.
```

`Multinomial` takes only categorical variables as arguments*, and it can be thought as a compact representation of an equivalent model with Boolean variables corresponding to each category connected by a conjunction factor for every combination of category assignments.

For example, suppose `a` is a variable taking values 0, 1, 2, and `b` is a variable taking values 0, 1. Then, `Multinomial(a, b)` is equivalent to having factors between `a` and `b` that correspond to the following indicator functions.

- I{a = 0, b = 0}
- I{a = 0, b = 1}
- I{a = 1, b = 0}
- I{a = 1, b = 1}
- I{a = 2, b = 0}
- I{a = 2, b = 1}

Note that each of the factors above has a distinct weight, i.e., one weight for each possible assignment of variables in the `Multinomial` factor. For more detail on how to specify Conditional Random Fields and perform Multi-class Logistic Regression using categorical factor, see the chunking example.

* Because of this limitation, categorical variables and categorical factor support is likely to go away in a near future release, in favor of a more flexible way to express multi-class predictions and mutual exclusions. Emulate categorical variables with functional dependencies in the Boolean variable columns, i.e., `tag?(@key word_id BIGINT, pos TEXT)` rather than `tag?(word_id BIGINT) Categorical(N)`, and replace `Multinomial` factor with conjunction of those Boolean variables with columns having functional dependencies.

Specifying weights

Each factor is assigned a *weight*, which represents the confidence in the correlation it expresses in the model. During statistical inference, these weights translate into their potentials that influence the probabilities of the connected variables. Factor weights are real numbers and only the relative magnitude to each other matters. Factors with larger weights have a greater impact on the connected variables than factors with smaller weights. Weights can be fixed to a constant manually, or they can be learned by DeepDive from the supervision labels at different granularity. Weights can be parameterized by some data originating from the data (in most time referred to as *features*), in which case factors with different parameter values will use different weights. In order to learn weights automatically, there must be enough training data available.

DDlog syntax for specifying weights for three different cases are shown below.

```
Q?(x TEXT).

data(x TEXT, y TEXT).


# Fixed weight (10 can be treated as positive infinite)

@weight(10)   Q(x) :- data(x, y).


# Unknown weight, to be learned from the data, but not dependin
g on any variable.

# All factors created by this rule will have the same weight.

@weight("?")  Q(x) :- data(x, y).


# Unknown weight, each to be learned from the data per differen
t values of y.

@weight(y)    Q(x) :- data(x, y).
```

Running a DeepDive Application

Compiling a DeepDive application

The first step to run a DeepDive application is to compile it. We describe here the use of the different `deepdive` commands, why and where it compiles and which files can be interesting to look at.

How to compile

To compile a DeepDive application, simply run the following command within the DeepDive app.

```
deepdive compile
```

All subsequent `deepdive do` commands will run what has been compiled, so an app has to be compiled at least once initially before executing any part of it.

What is compiled

All compiled output is created under the `run/` directory of the app:

- `run/dataflow.svg`

  A data flow diagram showing the dependencies between processes. This file contains a graph of the dataflow and helps better understand the dependencies among relations, rules and user-defined functions used to produce them. It can be opened in a web browser (Chrome or Safari work particularly well for it).

- `run/Makefile`

  A Makefile that mirrors the dependencies, used for generating execution plans for running certain parts of the data flow.

- `run/process/**/run.sh`

  Shell scripts that contain what actually should be run for every process. These scripts will be run by certain `deepdive do` commands.

- `run/LATEST.COMPILE/` and `run/compiled/`

  Each compilation step creates a unique timestamped directory under `run/` to store all the intermediate representation used for compilation (`*.json`). In particular, the latest compiled version can be found in `run/LATEST.COMPILE/`. For instance,

the `deepdive.conf` used for the latest compilation can be found at `run/LATEST.COMPILE/deepdive.conf`.

- `config.json`

  The compiled final configuration object.

- `code-*.json`

  The compiled object for generating actual code.

- `compile.log`

  The log file left by the latest compilation step.

If you keep track of your DeepDive application with Git, it may be important to add a `/run` line in `.gitignore`.

When to compile

The following files in a DeepDive application are considered as the input to `deepdive compile`:

- `app.ddlog`
- `deepdive.conf`
- `schema.json`

`deepdive compile` should be rerun whenever changes to these files are to be reflected on the execution done by `deepdive do`. Otherwise, the modifications in `app.ddlog` won't be considered, and simply the last compiled code will execute.

Why compile

Compiling a DeepDive application presents two main advantages. First, it helps to run the application faster by extracting the correct dependencies between the different operations. Second, it provides many checks for the application and helps detecting errors even before running DeepDive.

Compile-time checks

All the extractors and tables declarations can be written in any order in `app.ddlog`: the whole dataflow is extracted during the compilation and many checks are made. All these checks can also be performed individually by the command `deepdive check`. In particular, the following checks are made:

- `input_extractors_well_defined`: checks sanity of all defined extractors.
- `input_schema_wellformed`: checks if schema variables are well formed.

- `compiled_base_relations_have_input_data`: all the tables declared that are not filled by an extractor must have input data in `input/` that can be loaded automatically.
- `compiled_dependencies_correct`: check that the dependencies are correct. In particular, in the application includes extractors in deepdive.conf, each output relation must be output by exactly one extractor.
- `compiled_input_output_well_defined`: checks if all inputs and outputs of compiled processes are well-defined.
- `compiled_output_uniquely_defined`: checks if all outputs in the compiled documents are defined by one process.

Managing input data and data products

DeepDive provides a handful of commands and conventions to manage input data to an application as well as the data produced by it. The actual processing of the data is discussed in a page about executing the DeepDive application.

Inspecting schema for relations in the DeepDive app

There are a few handy ways to inspect the relational schema declared for a DeepDive app. Once the application is compiled, the following commands allow you to access the schema.

Listing relations

```
deepdive relation list
articles
has_spouse
num_people
person_mention
sentences
spouse_candidate
spouse_feature
```

```
spouse_label
spouse_label__0
spouse_label_resolved
spouses_dbpedia
```

Listing columns of a relation

```
deepdive relation columns articles
id:text
content:text
```

Listing variable relations

```
deepdive relation variables
has_spouse
```

Preparing the database for the DeepDive app

The database for the DeepDive application is configured through the `db.url` file or can be overridden with the `DEEPDIVE_DB_URL` environment variable.

Initializing the database

To initialize the database in a clean state, also dropping it if necessary, run:

```
deepdive db init
```

Creating tables in the database

To make sure an empty table named *foo* is created in the database as declared in `app.ddlog`, run:

```
deepdive create table foo
```

It is also possible to create any table with a particular column-type definitions, after another existing table such as *bar*, or using the result of a SQL query. A view can be created given a SQL query. This command ensures any existing table or view with the same name is dropped, and a new empty one is created and exists.

```
deepdive create table foo x:INT y:TEXT ...
deepdive create table foo like bar
deepdive create table foo as 'SELECT ...'
deepdive create view bar as 'SELECT ...'
```

To create a table only if it does not exist, the following command can be used instead:

```
deepdive create table-if-not-exists foo ...
```

Organizing input data

All input data for a DeepDive application should be kept under the `input/` directory. DeepDive relies on a naming convention and assumes data for a relation *`foo`* declared in `app.ddlog` (and not output by an extractor) exists at path `input/`*`foo.extension`* where *`extension`* can be one of `tsv`, `csv`, `sql` `tsv.bz2`, `csv.bz2`, `tsv.gz`, `csv.gz`, `tsv.sh`, `csv.sh`. This indicates in what format it is serialized as well as how it is compressed, or whether it's a shell script that emits such data or a file containing the data itself. For example, in the spouse example, the `input/articles.tsv.sh` is a shell script that produces lines with tab-separated values for the "articles" relation.

Moving data in and out of the database

Loading data to the database

To load such input data for a relation *foo*, run:

```
deepdive load foo
```

To load data from a particular source such as `source.tsv` or multiple sources `/tmp/source-1.tsv`, `/data/source-2.tsv.bz2` instead, they can be passed over as extra arguments:

```
deepdive load foo  source.tsv
deepdive load foo  /tmp/source-1.tsv /data/source-2.tsv.bz2
```

When a data source provides only particular columns, they can be specified as follows:

```
deepdive load 'foo(x,y)'  only-x-y.csv
```

If the destination to load is a variable relation and no columns are explicitly specified, then the sources are expected to provide an extra column at the right end that corresponds to each row's supervision label.

Unloading data products from the database

To unload data produced for a relation to files, such as *bar* into two sinks `dump-1.tsv` and `/data/dump-2.csv.bz2`, assuming you have the table populated in the database by executing some data processing, run:

```
deepdive unload bar  bar-1.tsv /data/bar-2.csv.bz2
```

This will unload partitions of the rows of relation *bar* to the given sinks in parallel.

Running queries against the database

Running queries in DDlog

It is possible to write simple queries against the database in DDlog and run them using the `deepdive query` command. A DDlog query begins with an optional list of expressions, followed by a separator `?-`, then a typical body of a conjunctive query in DDlog. Following are examples of actual queries that can be used against the data produced by DeepDive for the spouse example after running `deepdive run` command at least once.

Browsing values

To browse values in a relation, variables can be placed at the columns of interest, such as `name1` and `name2` for the second and fourth columns of the `spouse_candidate` relation, and the wildcard `_` can be used for the rest to ignore them as follows:

```
deepdive query '?- spouse_candidate(_, name1, _, name2).'
```

Joins, selection, and projections

Finding values across multiple relations that satisfy certain conditions can be expressed in a succinct way. For example, finding the names of candidate pairs of spouse mentions in a document that contains a certain keyword, such as "President" can be written as:

```
deepdive query '
    name1, name2 ?-
        spouse_candidate(p,name1,_,name2),
        person_mention(p,_,doc,_,_,_),
        articles(doc, content),
```

```
        content LIKE "%President%".
    '
```

Aggregation

Computing aggregate values such as counts and sums can be done using aggregate functions. Counting the number of tuples is as easy as just adding a `COUNT(1)` expression before the separator. For example, to count the number of documents that contain the word "President" is written as:

```
deepdive query 'COUNT(1) ?- articles(_, content), content LIKE "%President%".'
```

Grouping

If expressions using aggregate functions are mixed with other values, the aggregates are grouped by the other ones. This makes it easy to break down a single aggregate value into parts, such as number of candidates per document as shown below.

```
deepdive query '
    doc, COUNT(1) ?-
        spouse_candidate(p,_,_,_),
        person_mention(p,_,doc,_,_,_),
        articles(doc, content).
    '
```

Ordering

Ordering the results by certain ways is also easily expressed by adding `@order_by` annotations before the value to use for ordering. For example, the following modified query shows the documents with the most number of candidates first:

```
deepdive query '
    doc, @order_by("DESC") COUNT(1) ?-
        spouse_candidate(p,_,_,_),
        person_mention(p,_,doc,_,_,_),
        articles(doc, content).
```

```
    '
```

`@order_by` takes two arguments: 1) `dir` parameter taking either the "ASC" or `"DESC"` for ascending (default) or descending order and 2) `priority` parameter taking a number for deciding the priority for ordering when there are multiple `@order_by` expressions (smaller the higher priority). For example, `@order_by("DESC", -1)`, `@order_by("DESC")`, `@order_by(priority=-1)` are all recognized.

Limiting (top *k*)

By putting a number after the expressions separated by a pipe character, e.g., `| 10`, the number of tuples can be limited. For example, the following modified query shows the top 10 documents containing the most candidates:

```
deepdive query '
    doc, @order_by("DESC") COUNT(1) | 10 ?-
        spouse_candidate(p,_,_,_),
        person_mention(p,_,doc,_,_,_),
        articles(doc, content).
    '
```

Using more than one rule

Sometimes one rule is not enough to express, so a query may define multiple temporary relations first. For example, to produce a histogram of the number of candidates per document, the counts must be counted, so two rules are necessary as shown below.

```
deepdive query '
    num_candidates_by_doc(doc, COUNT(1)) :-
        spouse_candidate(p,_,_,_),
        person_mention(p,_,doc,_,_,_),
        articles(doc, content).

    @order_by num_candidates, COUNT(1) ?- num_candidates_by_doc
(doc, num_candidates).
    '
```

Saving results

By providing the extra `format=tsv` or `format=csv` argument, the resulting tuples can be easily saved into a file. For example, the following command saves the names of candidates with their document id as a comma-separated file named `candidates-docs.csv`.

```
deepdive query '
    doc,name1,name2 ?-
        spouse_candidate(p1,name1,_,name2),
        person_mention(p1,_,doc,_,_,_).
    ' format=csv  >candidates-docs.csv
```

Viewing the SQL

Using the `-n` flag will display the SQL query to be executed instead of showing the results from executing it.

```
deepdive query -n '?- ...'
```

Running queries in SQL

```
deepdive sql
```

This command opens a SQL prompt for the underlying database configured for the application.

Optionally, the SQL query can be passed as a command line argument to run and print its result to standard output. For example, the following command prints the number of sentences per document:

```
deepdive sql "SELECT doc_id, COUNT(*) FROM sentences GROUP BY doc_id"
```

To get the result as tab-separated values (TSV), or comma-separated values (CSV), use the following commands:

```
deepdive sql eval "SELECT doc_id, COUNT(*) FROM sentences GROUP BY doc_id" format=tsv
```

```
deepdive sql eval "SELECT doc_id, COUNT(*) FROM sentences GROUP BY doc_id" format=csv header=1
```

Executing processes in the data flow

After compiling a DeepDive application, each compiled process in the data flow defined by the application can be executed with great flexibility. DeepDive provides a core set of commands that supports precise control of the execution. The following tasks are a few examples that are easily doable in DeepDive's vocabulary:

- Executing the complete data flow.
- Executing a fragment of the complete data flow.
- Stopping execution at any point and resuming from where it stopped.
- Repeating certain processes with different parameters.
- Skipping expensive processes by loading their output from external data sources.

End-to-end execution

To simply run the complete data flow in an application from the beginning to the end, use the following command under any subdirectory of the application:

```
deepdive run
```

This command will:

1. Initialize the application as well as its configured database.
2. Run all processes that correspond to the normal derivation rules as well as the rules with user-defined functions.
3. Ground the factor graph according to the inference rules defining the model for statistical inference.
4. Perform weight learning and inference to compute marginal probability of every variable.
5. Generate calibration plots and data for debugging.

Granular execution scenarios

There are several execution scenarios that frequently arise while developing a DeepDive application. Any of the commands shown in this page can be run under any subdirectory of a DeepDive application. To see all options for each command, the online help message can be seen with the `deepdive help` command. For example, the following shows detailed usage of `deepdive do` command:

```
deepdive help do
```

Partial execution

Running only a small part of the data flow defined in an application is the most common scenario. The following command allows the execution to stop at given TARGETs instead of continuing to the end.

```
deepdive do TARGET...
```

It will present a shell script in a text editor that enumerates the processes to be run for the given TARGETs. By saving a final plan and quiting the editor, the actual execution starts.

Valid TARGET names are shown when no argument is given.

```
deepdive do
```

Refer to the next section for more detail about these TARGET names.

Stopping and resuming

The execution started can be interrupted at any time (with Ctrl-C or ^C) or aborted by an error, then resumed later.

```
deepdive do TARGET...
...
^C
...
deepdive do TARGET...
```

DeepDive will resume the execution from the last unfinished process, skipping what has already been done.

Repeating

Repeating certain parts of the data flow is another common scenario. DeepDive provides a way to mark certain processes as not done so they can be repeated. For example, the sequence of two commands below is basically what `deepdive run` does, i.e., repeats the end-to-end execution.

```
deepdive mark todo init/app calibration weights
deepdive do       init/app calibration weights
```

DeepDive provides a shorthand `deepdive redo` for easier repeating:

```
deepdive redo init/app calibration weights
```

Skipping

Skipping certain parts of the data flow and starting from data manually loaded is also easily doable. This can be useful to skip certain processes that take very long. For example, suppose relation `bar_derived_from_foo` is derived from relation `foo`, and `foo` takes excessive amount of time to compute and therefore has been saved at `/some/data/source.tsv`. Then the following sequence of commands skips all processes that `foo` depends on, assumes it is new, and executes the processes that derive `bar_derived_from_foo` from `foo`.

```
deepdive create table foo
deepdive load foo /some/data/source.tsv
deepdive mark new foo
deepdive do bar_derived_from_foo
```

Compiled processes and data flow

For execution, DeepDive produces a compiled data flow graph that consists of primarily *data*, *model*, and *process* nodes and edges representing dependencies between them.

- Data nodes correspond to relations or tables in the database.
- Model nodes denote artifacts for statistical learning and inference.
- Process nodes represent a unit of computation.
- An edge between a process and data/model nodes mean the process takes as input or produces such node.
- An edge between two processes denotes one is dependent on another while hiding the details about the intermediary nodes.

DeepDive compiles a few built-in processes into the data flow graph that are necessary for initialization, statistical learning and inference, and calibration. The rest of the processes correspond to the rules for deriving relations according to the DDlog program and the extractors in deepdive.conf. Below is a data flow graph compiled for the tutorial example, which can be found at `run/dataflow.svg` for any application.

Execution plan

DeepDive uses the compiled data flow graph to find the correct order of processes to execute. Any node or set of nodes in the data flow graph can be set as the target for execution, and DeepDive enumerates all necessary processes as an *execution plan*.

deepdive plan

This execution plan can be seen using the `deepdive plan` command. For example, when a plan for `data/sentences` is asked using the following command:

```
deepdive plan data/sentences
```

DeepDive gives an output that looks like:

```
# execution plan for sentences


: ## process/init/app ########################################
##################

: # Done: 2016-02-01T20:56:50-0800 (2d 23h 55m 9s ago)

: process/init/app/run.sh

: mark_done process/init/app

: ############################################################
##################

:

: ## process/init/relation/articles ############################
##################

: # Done: 2016-02-01T20:56:55-0800 (2d 23h 55m 4s ago)

: process/init/relation/articles/run.sh

: mark_done process/init/relation/articles

: ############################################################
##################

:

: ## data/articles ##############################################
##################

: # Done: 2016-02-01T20:56:55-0800 (2d 23h 55m 4s ago)

: # no-op

: mark_done data/articles

: ############################################################
##################
```

```
## process/ext_sentences_by_nlp_markup ########################################

# Done: N/A

process/ext_sentences_by_nlp_markup/run.sh

mark_done process/ext_sentences_by_nlp_markup

################################################################################


## data/sentences ##############################################################

# Done: N/A

# no-op

mark_done data/sentences

################################################################################
```

An execution plan is basically a shell script that invokes the actual run.sh compiled for each process. Processes that have been already marked as done in the past are commented out (with `:`), and exactly when they were done is displayed in the comments.

deepdive do

DeepDive provides a `deepdive do` command that takes as input the target nodes in the data flow graph, presents the execution plan in an editor, then executes the final plan. Because of the provided chance to modify the generated execution plan, the user has complete control over what is executed, and override or skip certain processes if needed.

```
deepdive do data/sentences
```

Execution timestamps

After a process finishes its execution, the execution plan includes a `mark_done` command to mark the executed process as done. The command touches a *`process/name`*`.done` file under the `run/` directory to record a timestamp. These timestamps are then used for determining which processes have finished and which others have not.

deepdive mark

DeepDive provides a `deepdive mark` command to manipulate such timestamps to repeat or skip certain parts of the data flow. It allows a given node to be marked as:

- `done`

  So all processes depending on the process including itself can be skipped.

- `todo-from-scratch`

  So all processes from the first process in the execution plan can be repeated.

- `todo`

  So all processes that depend on the node including itself can be repeated.

- `new`

  So all processes that depend on the node can be repeated (not including itself).

- `all-new`

  So all processes that depend on the node or any of its ancestor can be repeated (not including itself).

For example, if we mark a process as to be repeated using the following command:

```
deepdive mark todo process/init/relation/articles
```

Then `deepdive plan` will give an output like below to repeat the processes already marked as done in the past:

```
# execution plan for sentences


: ## process/init/app ##########################################
##################

: # Done: 2016-02-01T20:56:50-0800 (2d 23h 55m 39s ago)

: process/init/app/run.sh

: mark_done process/init/app

: ##############################################################
##################


## process/init/relation/articles ##############################
################

# Done: 2016-02-01T20:56:55-0800 (2d 23h 55m 34s ago)
```

```
process/init/relation/articles/run.sh

mark_done process/init/relation/articles

################################################################
################

## data/articles ###############################################
###############

# Done: 2016-02-01T20:56:55-0800 (2d 23h 55m 34s ago)

# no-op

mark_done data/articles

################################################################
################

## process/ext_sentences_by_nlp_markup #########################
###############

# Done: N/A

process/ext_sentences_by_nlp_markup/run.sh

mark_done process/ext_sentences_by_nlp_markup

################################################################
################

## data/sentences ##############################################
###############

# Done: N/A

# no-op

mark_done data/sentences

################################################################
################
```

Built-in data flow nodes

DeepDive adds several built-in processes to the compiled data flow to ensure necessary steps are performed before and after the user-defined processes.

- `process/init/app`

  This process initializes the application's database and executes the `input/init.sh` script if available, which can be used for ensuring input data is downloaded and libraries and code required by UDFs are set up correctly.

  All processes that are not dependent on any other automatically becomes dependent on this process to ensure every process is executed after the application is initialized.

- `process/init/relation/R`

  Such process creates the table *R* in the database and loads data from `input/R.*`. These processes are automatically created and added to the data flow for every relation that is not derived from any other relation nor output by any process.

- `process/grounding/*`

  All processes for grounding the factor graph are put under this namespace.

- `process/model/*`

  All processes for learning and inference are put under this namespace.

- `model/*`

  All artifacts related to the statistical inference model that do not belong to the database are put under this namespace, e.g.:

  - `model/factorgraph`

    Denotes factor graph binary files under `run/model/factorgraph/`.

  - `model/weights`

    Denotes text files that contain learned weights of the model under `run/model/weights/`.

  - `model/probabilities`

    Denotes files holding the computed marginal probabilities under `run/model/probabilities/`

  - `model/calibration-plots`

    Denotes calibration plot images and data files under `run/model/calibration-plots/`.

- `data/model/*`

All data relevant to statistical inference that go into the database are put under this namespace. For example, `data/model/weights`and `data/model/probabilities` correspond to tables and views that keep the learning and inference results.

Environment variables

There are several environment variables that can be tweaked to influence the execution of processes for UDFs.

- `DEEPDIVE_NUM_PROCESSES`

  Controls the number of UDF processes to run in parallel. This defaults to one less than the number of processors, minimum one.

- `DEEPDIVE_NUM_PARALLEL_UNLOADS` and `DEEPDIVE_NUM_PARALLEL_LOADS`

  Controls the number of processes to run in parallel for unloading from and loading to the database. These default to one.

- `DEEPDIVE_AUTOCOMPILE`

  Controls whether `deepdive do` should automatically compile the app whenever a source file changes, such as `app.ddlog` or `deepdive.conf`. By default, it is `DEEPDIVE_AUTOCOMPILE=true`, i.e., automatically compiling to reduce the friction of manually running `deepdive compile`. However, it can be set to `DEEPDIVE_AUTOCOMPILE=false` to prevent unexpected recompilations.

- `DEEPDIVE_INTERACTIVE`

  Controls whether `deepdive do` should be interactive or not, asking to review the execution plan and allow editing before actual execution. By default, it is `DEEPDIVE_INTERACTIVE=false`, not asking whether to recompile the app upon source change, nor providing a chance to review or edit the execution plan.

- `DEEPDIVE_PLAN_EDIT`

  Controls whether a chance to edit the execution plan is provided (when set to `true`) or not (when `false`) in interactive mode.

- `VISUAL` and `EDITOR`

  Decides which editor to use. Defaults to `vi`.

Learning and inference with the statistical model

For every DeepDive application, executing any data processing it defines is ultimately to supply with necessary bits in the construction of the statistical model declared in DDlog for *joint inference*. DeepDive provides several commands to streamline operations on the statistical model, including its creation (*grounding*), parameter estimation (*learning*), and computation of probabilities (*inference*) as well as keeping and reusing the parameters of the model (*weights*).

Getting the inference result

To simply get the inference results, i.e., the marginal probabilities of the random variables defined in DDlog, use the following command:

```
deepdive do probabilities
```

This takes care of executing all necessary data processing, then creates a statistical to perform learning and inference, and loads all probabilities of every variable into the database.

Inspecting the inference result

For viewing the inference result, DeepDive creates a database view that corresponds to each variable relation (using a `_inference` suffix). For example, the following SQL query can be used for inspecting the probabilities of the variables in relation `has_spouse`:

```
deepdive sql "SELECT * FROM has_spouse_inference"
```

It shows a table that looks like below where the `expectation` column holds the inferred marginal probability for each variable:

```
                    p1_id                     |
    p2_id                     | expectation
----------------------------------------------+-----------
---------------------------------------+-------------
 7b29861d-746b-450e-b9e5-52db4d17b15e_4_5_5       | 7b29861d-74
6b-450e-b9e5-52db4d17b15e_4_0_0       |       0.988
 ca1debc9-1685-4555-8eaf-1a74e8d10fcc_7_25_25     | ca1debc9-16
85-4555-8eaf-1a74e8d10fcc_7_30_31     |       0.972
 34fdb082-a6ef-4b54-bd17-6f8f68acb4a4_15_28_28    | 34fdb082-a6
ef-4b54-bd17-6f8f68acb4a4_15_23_23    |       0.968
 7b29861d-746b-450e-b9e5-52db4d17b15e_4_0_0       | 7b29861d-74
6b-450e-b9e5-52db4d17b15e_4_5_5       |       0.957
 a482785f-7930-427a-931f-851936cd9bb1_2_34_35     | a482785f-79
30-427a-931f-851936cd9bb1_2_18_19     |       0.955
```

```
 a482785f-7930-427a-931f-851936cd9bb1_2_18_19     | a482785f-79
30-427a-931f-851936cd9bb1_2_34_35     |       0.955

 93d8795b-3dc6-43b9-b728-a1d27bd577af_5_7_7       | 93d8795b-3d
c6-43b9-b728-a1d27bd577af_5_11_13     |       0.949

 e6530c2c-4a58-4076-93bd-71b64169dad1_2_11_11     | e6530c2c-4a
58-4076-93bd-71b64169dad1_2_5_6       |       0.946

 5beb863f-26b1-4c2f-ba64-0c3e93e72162_17_35_35    | 5beb863f-26
b1-4c2f-ba64-0c3e93e72162_17_29_30    |       0.944

 93d8795b-3dc6-43b9-b728-a1d27bd577af_3_5_5       | 93d8795b-3d
c6-43b9-b728-a1d27bd577af_3_0_0       |        0.94

 216c89a9-2088-4a78-903d-6daa32b1bf41_13_42_43    | 216c89a9-20
88-4a78-903d-6daa32b1bf41_13_59_59    |       0.939

 c3eafd8d-76fd-4083-be47-ef5d893aeb9c_2_13_14     | c3eafd8d-76
fd-4083-be47-ef5d893aeb9c_2_22_22     |       0.938

 70584b94-57f1-4c8c-8dd7-6ed2afb83031_20_6_6      | 70584b94-57
f1-4c8c-8dd7-6ed2afb83031_20_1_2      |       0.938

 ac937bee-ab90-415b-b917-0442b88a9b87_5_7_7       | ac937bee-ab
90-415b-b917-0442b88a9b87_5_10_10     |       0.934

 942c1581-bbc0-48ac-bbef-3f0318b95d28_2_35_36     | 942c1581-bb
c0-48ac-bbef-3f0318b95d28_2_18_19     |       0.934

 ec0dfe82-30b0-4017-8c33-258e2b2d7e35_36_29_29    | ec0dfe82-30
b0-4017-8c33-258e2b2d7e35_36_33_34    |       0.933

 74586dd9-55af-4bb4-9a95-485d5cef20d7_34_8_8      | 74586dd9-55
af-4bb4-9a95-485d5cef20d7_34_3_4      |       0.933

 70bebfae-c258-4e9b-8271-90e373cc317e_4_14_14     | 70bebfae-c2
58-4e9b-8271-90e373cc317e_4_5_5       |       0.933

 ca1debc9-1685-4555-8eaf-1a74e8d10fcc_7_30_31     | ca1debc9-16
85-4555-8eaf-1a74e8d10fcc_7_25_25     |       0.928

 ec0dfe82-30b0-4017-8c33-258e2b2d7e35_36_15_15    | ec0dfe82-30
b0-4017-8c33-258e2b2d7e35_36_33_34    |       0.927

 f49af9ca-609a-4bdf-baf8-d8ddd6dd4628_4_20_21     | f49af9ca-60
9a-4bdf-baf8-d8ddd6dd4628_4_15_16     |       0.923

 ec0dfe82-30b0-4017-8c33-258e2b2d7e35_16_9_9      | ec0dfe82-30
b0-4017-8c33-258e2b2d7e35_16_4_5      |       0.923

 93d8795b-3dc6-43b9-b728-a1d27bd577af_3_23_23     | 93d8795b-3d
c6-43b9-b728-a1d27bd577af_3_0_0       |       0.921
```

```
 5530e6a9-2f90-4f5b-bd1b-2d921ef694ef_2_18_18     | 5530e6a9-2f
90-4f5b-bd1b-2d921ef694ef_2_10_11     |       0.918

[...]
```

To better understand the inference result for debugging, please refer to the pages about calibration, Dashboard, labeling, and browsing data.

The next several sections describe further detail about the different operations on the statistical model supported by DeepDive.

Grounding the factor graph

The inference rules written in DDlog give rise to a data structure called *factor graph* DeepDive uses to perform statistical inference. *Grounding* is the process of materializing the factor graph as a set of files by laying down all of its variables and factors in a particular format. This process can be performed using the following command:

```
deepdive model ground
```

The above can be viewed as a shorthand for executing the following built-in processes:

```
deepdive redo process/grounding/variable_assign_id process/grou
nding/combine_factorgraph
```

Grounding generates a set of files for each variable and factor under `run/model/grounding/`. They are then combined into a unified factor graph under `run/model/factorgraph/` to be easily consumed by the DimmWitted inference engine for learning and inference. For example, below shows a typical list of files holding a grounded factor graph:

```
find run/model/grounding -type f

run/model/grounding/factor/inf_imply_has_spouse_has_spouse/fac
tors.part-1.bin.bz2

run/model/grounding/factor/inf_imply_has_spouse_has_spouse/ned
ges.part-1

run/model/grounding/factor/inf_imply_has_spouse_has_spouse/nfa
ctors.part-1

run/model/grounding/factor/inf_imply_has_spouse_has_spouse/wei
ghts.part-1.bin.bz2

run/model/grounding/factor/inf_imply_has_spouse_has_spouse/wei
ghts_count

run/model/grounding/factor/inf_imply_has_spouse_has_spouse/wei
ghts_id_begin
```

```
run/model/grounding/factor/inf_imply_has_spouse_has_spouse/weights_id_exclude_end
run/model/grounding/factor/inf_istrue_has_spouse/factors.part-1.bin.bz2
run/model/grounding/factor/inf_istrue_has_spouse/nedges.part-1
run/model/grounding/factor/inf_istrue_has_spouse/nfactors.part-1
run/model/grounding/factor/inf_istrue_has_spouse/weights.part-1.bin.bz2
run/model/grounding/factor/inf_istrue_has_spouse/weights_count
run/model/grounding/factor/inf_istrue_has_spouse/weights_id_begin
run/model/grounding/factor/inf_istrue_has_spouse/weights_id_exclude_end
run/model/grounding/factor/weights_count
run/model/grounding/variable/has_spouse/count
run/model/grounding/variable/has_spouse/id_begin
run/model/grounding/variable/has_spouse/id_exclude_end
run/model/grounding/variable/has_spouse/variables.part-1.bin.bz2
run/model/grounding/variable_count
```

Learning the weights

DeepDive learns the weights of the grounded factor graph, i.e., estimates the maximum likelihood parameters of the statistical model from the variables that were assigned labels via distant supervision rules written in DDlog. DimmWitted inference engine uses *Gibbs sampling*with *stochastic gradient descent* to learn the weights.

The following command performs learning using the grounded factor graph (or grounds a new factor graph if needed):

```
deepdive model learn
```

This is equivalent to executing the following targets:

```
deepdive redo process/model/learning data/model/weights
```

DimmWitted outputs the learned weights as a text file under `run/model/weights/`. For convenience, DeepDive loads the learned weights into the database and creates several views for the following target:

```
deepdive do data/model/weights
```

This will create a comprehensive view of the weights named `dd_inference_result_weights_mapping`. The weights corresponding to each inference rule and by their parameter value can be easily accessed using it. Below shows a few example of learned weights:

```
deepdive sql "SELECT * FROM dd_inference_result_weights_mapping
"
    weight    |                      description
--------------+-----------------------------------------------
-----------------
     1.80754 | inf_istrue_has_spouse--INV_NGRAM_1_[wife]
     1.45959 | inf_istrue_has_spouse--NGRAM_1_[wife]
    -1.33618 | inf_istrue_has_spouse--STARTS_WITH_CAPITAL_[Tru
e_True]
     1.30884 | inf_istrue_has_spouse--INV_NGRAM_1_[husband]
     1.22097 | inf_istrue_has_spouse--NGRAM_1_[husband]
    -1.00449 | inf_istrue_has_spouse--W_NER_L_1_R_1_[O]_[O]
    -1.00062 | inf_istrue_has_spouse--NGRAM_1_[,]
          -1 | inf_imply_has_spouse_has_spouse-
    -0.94185 | inf_istrue_has_spouse--IS_INVERTED
    -0.91561 | inf_istrue_has_spouse--INV_STARTS_WITH_CAPITAL_
[True_True]
    0.896492 | inf_istrue_has_spouse--NGRAM_2_[he wife]
    0.835013 | inf_istrue_has_spouse--INV_NGRAM_1_[he]
   -0.825314 | inf_istrue_has_spouse--NGRAM_1_[and]
    0.805815 | inf_istrue_has_spouse--INV_NGRAM_2_[he wife]
   -0.781846 | inf_istrue_has_spouse--INV_W_NER_L_1_R_1_[O]_
[O]
     0.75984 | inf_istrue_has_spouse--NGRAM_1_[he]
    -0.74405 | inf_istrue_has_spouse--INV_NGRAM_1_[and]
```

```
    0.701149 | inf_istrue_has_spouse--INV_NGRAM_1_[she]
   -0.645765 | inf_istrue_has_spouse--INV_NGRAM_1_[,]
      0.6105 | inf_istrue_has_spouse--INV_NGRAM_2_[husband ,]
    0.585621 | inf_istrue_has_spouse--INV_NGRAM_2_[she husband]
    0.583075 | inf_istrue_has_spouse--INV_NGRAM_2_[and he]
    0.581042 | inf_istrue_has_spouse--NGRAM_1_[she]
    0.540534 | inf_istrue_has_spouse--NGRAM_2_[husband ,]
[...]
```

Inference

After learning the weights, DeepDive uses them with the grounded factor graph to compute the marginal probability of every variable. DimmWitted's high-speed implementation of Gibbs sampling is used for performing a marginal inference by approximately computing the probablities of different values each variable can take over all possible worlds.

```
deepdive model infer
```

This is equivalent to executing the following nodes in the data flow:

```
deepdive redo process/model/inference data/model/probabilities
```

In fact, because performing inference as a separate process from learning incurs unnecessary overhead of reloading the factor graph into memory again, DimmWitted also performs inference immediately after learning the weights. Therefore unless previously learned weights are being reused, hence skipping the learning part, the following command that performs just the inference has no effect:

DimmWitted outputs the inferred probabilities as a text file under `run/model/probabilities/`. As shown in the first section, DeepDive loads the computed probabilities into the database and creates views for convenience.

Reusing weights

A common use case is to learn the weights from one dataset then performing inference on another, i.e., train model on one dataset and test it on new datasets.

1. Learn the weights from a small dataset.
2. Keep the learned weights.
3. Reuse the kept weights for inference on a larger dataset.

DeepDive provides several commands to support the management and reuse of such learned weights.

Keeping learned weights

To keep the currently learned weights for future reuse, say under a name `FOO`, use the following command:

```
deepdive model weights keep FOO
```

This dumps the weights from the database into files at `snapshot/model/weights/FOO/` so they can be reused later. The name `FOO` is optional, and a generated timestamp is used instead when no name is specified.

Reusing learned weights

To reuse a previously kept weights, under a name `FOO`, use the following command:

```
deepdive model weights reuse FOO
```

This loads the weights at `snapshot/model/weights/FOO/` back to the database, then repeats necessary grounding processes for including the weights into the grounded factor graph. The name `FOO` is optional, and the most recently kept weights are used when no name is specified.

A subsequent command for performing inference reuses these weights without learning.

```
deepdive model infer
```

Managing kept weights

DeepDive provides several more commands to manage the kept weights.

To list the names of kept weights, use:

```
deepdive model weights list
```

To drop a particular weights, use:

```
deepdive model weights drop FOO
```

To clear any previously loaded weights to learn new ones, use:

```
deepdive model weights init
```

Debugging a DeepDive Application

Debugging user-defined functions

Many things can go wrong in user-defined functions (UDFs), so debugging support is important for the user to write the code and easily verify that it works as expected. UDFs can be implemented in any programming language as long as they take the form of an executable that reads from *standard input* and writes to *standard output*. Here are some general tips for printing information to the log and running the UDFs in limited ways to help work through issues without needing to run the entire data flow of the DeepDive application.

Printing to the log

Remember that the *standard output* of a UDF is already reserved for TSJ or TSV formatted data that gets loaded into the database. Therefore when a typical print statement is used for debugging, it won't appear anywhere in the log but just mangle the TSJ output stream and ultimately fail the UDF execution or corrupt its output. The correct way to print log statements is to print to the *standard error*. Below is an example in Python.

```
#!/usr/bin/env python
from deepdive import *
import sys

@tsj_extractor
@returns( ... )
def extract( ... ):
    ...
    print >>sys.stderr, 'This prints some_object to logs :', some_object
    ...
```

During execution of the script, anything written to *standard error* appears in the console as well as in the file named `run.log` under the `run/LATEST/` directory.

Executing UDFs within DeepDive's environment

To assist with debugging issues in UDFs, DeepDive provides a wrapper command to directly execute it within the same environment it uses for the actual execution.

Suppose a Python UDF at `udf/fn.py` imports `deepdive` and `ddlib` as suggested in the guide for writing UDFs. When run normally as a Python script, it will give an error that looks like this:

```
python udf/fn.py
Traceback (most recent call last):
  File "udf/fn.py", line 2, in <module>
    from deepdive import *
ImportError: No module named deepdive
```

Instead, by prefixing the command with `deepdive env`, they can be executed as if they were executed in the middle of DeepDive's data flow.

```
deepdive env python udf/fn.py
```

This will take TSJ rows from standard input and print TSJ rows to standard output as well as debug logs to standard error. It can therefore be debugged just like a normal Python program.

Calibration

One of the most important aspects of DeepDive is its iterative workflow. After performing probabilistic inference using DeepDive, it is crucial to evaluate the results and act on the feedback that the system provides to improve the accuracy. DeepDive produces *calibration plots* to help the user with this task.

Defining a holdout set

To get the most out of calibration, the user should specify either a *holdout fraction* or a custom *holdout query* to define a subset of evidence as training data. DeepDive uses the holdout variables to evaluate its accuracy, and no holdout is used by default.

Holdout fraction

When the `holdout_fraction` is set as below in the application's `deepdive.conf` file, DeepDive randomly selects the specified fraction of evidence variables, i.e., ones that are labeled can be selected and are *held out* from training.

```
deepdive.calibration.holdout_fraction: 0.25
```

Custom holdout query

DeepDive also supports a SQL query to define the holdout set. A custom holdout query must insert the internal `dd_id` column of all variables that are to be held out into the `dd_graph_variables_holdout` table through an arbitrary SQL.

For example, a custom holdout query can be specified as follows in `deepdive.conf`:

```
deepdive.calibration.holdout_query: """
    INSERT INTO dd_graph_variables_holdout(variable_id)
    SELECT dd_id
    FROM mytable
    WHERE predicate
"""
```

When a custom holdout query is defined as `holdout_query`, the `holdout_fraction` setting is ignored.

Inspecting probabilities and weights

To improve the prediction accuracy, it is useful to inspect the probabilities for each variable and the learned factor weights. DeepDive creates a view called `dd_inference_result_weights_mapping` which contains the factor names and the learned weights sorted by their absolute values.
The `dd_inference_result_weights_mapping` view has the following schema:

```
View "public.dd_inference_result_weights_mapping"
   Column    |       Type       | Modifiers
-------------+------------------+-----------
 id          | bigint           |
 isfixed     | integer          |
```

```
initvalue   | real             |
cardinality | text             |
description | text             |
weight      | double precision |
```

Specification for these fields:

- **id**: the unique identifier for the weight
- **initial_value**: the initial value for the weight
- **is_fixed**: whether the weight is fixed (cannot be changed during learning)
- **cardinality**: the cardinality of this factor. Meaningful for categorical factors.
- **description**: description of the weight, composed by [the name of inference rule]-[the specified value of "weight" in inference rule]
- **weight**: the learned weight value

Calibration data and plots

DeepDive generates a calibration data file for each variable defined in the schema when the following command is run:

```
deepdive do model/calibration-plots
```

They are generated at the path below where each file contains ten lines with the following five columns:

```
run/model/calibration-plots/[variable_name].tsv
```

```
[bucket_from] [bucket_to] [num_predictions] [num_true] [num_false]
```

DeepDive places the inference results into ten buckets. Each bucket is associated to a probability interval from 0.0 to 1.0. The meaning of the last three columns is the:

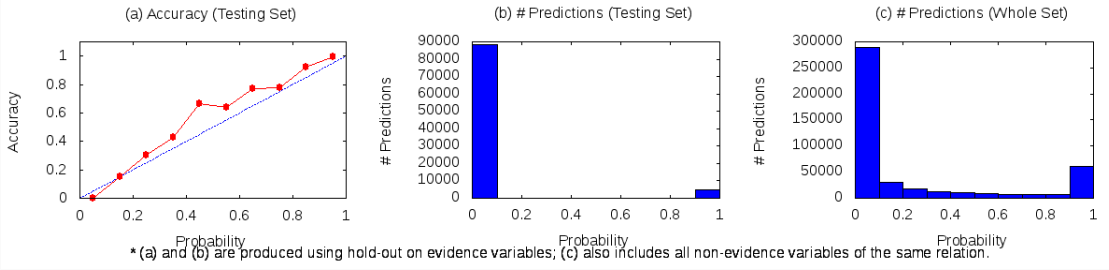
- `num_predictions` is the number of variables in the probability bucket, including both holdout and query variables (variables without any label).
  Basically, `num_predictions` = `num_holdout` + `num_unknown_var`.
- `num_true` is the number of holdout variables in the probability bucket with the value of true. The number should be high for buckets with large probabilities and small for buckets with small probabilities since the actual value of these variables are true and with high probability they should be predicted as true. Note that in this case only the holdout data is used.
- `num_false` is the number of holdout variables in the probability bucket with the value of false. The number should be small for buckets with large probabilities and large for buckets with small probabilities since the actual value of these variables are false and

with low probability they should be predicted as true. Note that in this case only the holdout data is used.

DeepDive also generates an image file called calibration plot for each of the variables defined in schema. The image file is generated next to the calibration data:

```
run/model/calibration-plots/[variable_name].png
```

A typical calibration plot looks as follows:



* (a) and (b) are produced using hold-out on evidence variables; (c) also includes all non-evidence variables of the same relation.

Interpreting calibration plots

The accuracy plot (a) shows the ratio of correct positive predictions for each probability bucket. Ideally, the red line should follow the blue line, representing that the system finds high number of evidence positive predictions for higher probability buckets and for lower probability buckets the system finds less number of evidence positive predictions linearly. Which means for probability bucket of 0 there should be no positive prediction, and for 100% bucket all the predictions should be positive. The accuracy is defined as `num_holdout_true` / `num_holdout_total`.

Plots (b) and (c) show the number of total prediction on the test and the training set, respectively. Ideally these plots should follow a U-curve. That is, the system makes many predictions with probability 0 (events that are likely to be false), and many predictions with probability > 0.9 (events that are likely to be true). Predictions in the range of 0.4 - 0.6 mean that the system is not sure, which may indicate that it needs more features to make predictions for such events.

Note that plots (a) and (b) can only be generated if a holdout fraction was specified in the configuration.

Acting on calibration data

There could many factors that lead to suboptimal results. Common ones are:

- **Not enough features:** This is particularly common when a lot of probability mass falls in the middle buckets (0.4 - 0.6). The system may be unable to make predictions about events because the available features are not specific-enough for the events. Take a look at variables that were assigned a probability in the 0.4 to 0.6 range, inspect them, and

come up with specific features that would push these variables towards a positive or negative probability.

- **Not enough positive evidence:** Without sufficient positive evidence the system will be unable to learn weights that push variables towards a high probability (or a low probability if the variables are negated). Having little probability mass on the right side of the graph is often an indicator for not having enough positive evidence, or not using features that use the positive evidence effectively.
- **Not enough negative evidence:** Without sufficient negative evidence, or with negative evidence that is biased, the system will not be able to distinguish true events from false events. That is, it will generate many *false positives*. In the graph this is often indicated by having little probability mass on the left (no U-shape) in plots (b) and (c), and/or by having a low accuracy for high probabilities in plot (a). Generating negative evidence can be somewhat of an art.
- **Weight learning does not converge:** When DeepDive is unable to learn weights for the inference rules the predicated data will be invalid. Check the DeepDive log file for the gradient value at the end of the learning phrase. If the value is very large (1000 or more), then it is possible that weight learning was not successful. In this case, one may try to increase the number of learning iterations, decrease the learning rate, or use a faster decay.
- **Weight learning converges to a local optimum:** The user can try increasing the learning rate, or using a slower decay. Check the DimmWitted sampler documentation for more details.

Recall errors

Recall is the fraction of relevant events that are extracted. In information extraction applications there are generally two sources of recall error:

- **Event candidates are not recognized in the text.** In this case, no variables are created are recall errors and these events do not show up in the calibration plots. For example, the system may fail to identify "micro soft" as a company name if it is lowercase and misspelled. Such errors are difficult to debug, unless there is a complete database to test against, or the user makes a closed-world assumption on the test set.
- **Events fall below a confidence cutoff.** Assuming that one is only interested in events that have a high probability, then events in the mid-range of the calibration plots can be seen as recall errors. For example, if one is interested only in company names that are > 90% confident are correct, then the company names in the buckets below 0.9 are recall errors. Recall can be improved using some of the aforementioned suggestions.