Pipeline Technical Documentation

Joy (Sizhe) Chen, Kenny Chiu, William Lu, Nelly (Nilgoon) Zarei

August 31st, 2018

1 Database operations

1.1 Obtaining a database connection

First, import the `Database` class:

```
from io_.db import Database
```

The `Database` class follows the singleton pattern to ensure that only one database connection is opened throughout the execution of a script. Thus, never directly call the `Database` constructor. Instead, call the static `get_instance` method, which returns a `Database` object representing the database connection:

```
db = Database.get_instance()
```

1.2 Loading data from the database

To load data from the database into a Pandas dataframe, write a SQL script to specify the tables to load from and the rows and columns to load, and save the script as a `.sql` file. For example, the following script loads all the *result_full_ descriptions* from the database:

```
SELECT test_key, result_key, result_full_description
FROM lab.dim_test_result
```

Next, call the `Database` object's `extract` method, passing the absolute path to the SQL script as a parameter:

```
df = db.extract("absolute_path_to_sql_script.sql")
```

The `extract` method returns a Pandas dataframe whose columns are the columns specified in the SQL script's `SELECT` statement:

test_key	result_key	result_full_description
5	-1	*Missing
6	3075034	HbA1C therapeutic goal $<$ 5y $<$ or =9% | HbA1C therapeutic goal ...
...	...	...

If a `NULL` is extracted from the database, it is stored as the Python `None` singleton in the dataframe.

If the SQL script is invalid (for example, a column name or table name does not exist,) a `sqlalchemy.exc.ProgrammingError` is raised.

1.3 Saving data to the database

Call the `Database` object's `insert` method, passing the dataframe to insert, the name of the table to insert to, and the schema name as parameters. The below example inserts a dataframe named `df` into the `lab.dim_test_result` table:

```
db.insert(df, "dim_test_result", "lab")
```

The `insert` method has **undefined behaviour** if:

- the table name does not exist
- the schema name does not exist
- the dataframe's columns do not match the table's columns
- inserting any row in the dataframe would cause a key conflict

2 Using MetaMap to annotate data

Our pipeline implements an interface to run MetaMap annotation on *result_full_description* strings. Some classification algorithms in our pipeline require MetaMap annotations to be given as input along with the *result_full_description*s.

To use the MetaMap interface, run *driver/metamap.py*. Set the constants at the top of the file to their desired values:

- **`SQL_FILEPATH`** - the absolute path to the SQL script for extracting the data to annotate
- **`TABLE`** - the name of the table to write the annotations to
- **`SCHEMA`** - the schema name of the table to write the annotations to
- **`OBSERVATIONS`** - `True` to run MetaMap at the observation level, `False` to run MetaMap at the test level

The data extracted from the database must have 3 columns: *test_key*, *result_key*, and *result_full_description*. If `OBSERVATIONS` is `True`, there must be an additional *obs_seq_nbr* column.

The table which the MetaMap annotations are written to must have 4 columns: *test_key*, *result_key*, *tags*, and *candidates*. If `OBSERVATIONS` is `True`, there must be an additional *obs_seq_nbr* column.

2.1 MetaMap tags format

Refer to Section 1.1 of "MetaMapBuild Source Code Documentation".

2.2 MetaMap candidates format

A *candidates* string from MetaMap contains the preferred names of all the organisms MetaMap found in the *result_full_description*. The *candidates* string is a serialized JSON object.

The JSON object contains an arbitrary number of key-value pairs. Each key is a preferred name. For example:

```
{
    "Genus Mycobacterium": { ... },
    "Mycobacterium avium complex": { ... }
}
```

Each value is another JSON object, containing the keys "CUI", "matched", and "position". For example, the value corresponding to "Genus Mycobacterium" is:

```
{
    "CUI": "C0026192",
    "matched": ["mycobacteria"],
    "position": [10]
}
```

- The "CUI" value is the Concept Unique Identifier that MetaMap uses internally to uniqely identify the preferred name.
- The "matched" value is an array of strings. Each string is a substring of the *result_full_ description* that MetaMap mapped to the preferred name.
- The "position" value is an array of integers. The ith integer is the ith matched substring's starting index in the *result_full_ description*.

In this example, "mycobacteria" (a substring starting at index 10 in the *result_full_ description*) was mapped by MetaMap to the "Genus Mycobacterium" preferred name.

Also refer to Section 1.2 of "MetaMapBuild Source Code Documentation".

2.3 Design tradeoffs

Running MetaMap is computationally expensive; it takes approximately 50 to 60 hours to run MetaMap on 365 thousand average-length *result_full_ descriptions*. It is intractable to re-annotate old training data on every run of the pipeline, so it is necessary to persist MetaMap annotations to the database. To simplify our pipeline implementation, we use blocking I/O operations when waiting for responses from the MetaMap API server.

As a result of these considerations, and to enforce best practices, we do not provide an interface for runing MetaMap on an in-memory dataframe or returning MetaMap annotations to an in-memory dataframe. All data to be annotated by MetaMap must be fetched from the database, and all MetaMap annotations must be written to the database.

We also do not support running MetaMap from any file other than *driver/metamap.py*.

2.4 Server connection bug and workaround

Our pipeline uses the Py4J library to connect to the MetaMap API server from Python. The connection code contains a non-deterministic bug: occasionally, a `py4j.protocol.Py4JNetworkError` is raised.

As a workaround, our pipeline catches this error, if it is raised, and attempts to retry the connection up to five times. If an attempt succeeds, our pipeline prints the attempt number that succeeded (for example, "Connected to Java server on attempt 2".) If all five attempts fail, our pipeline raises an `Exception`.

3 Using the classification modules

The `TestPerformedModule`, `TestOutcomeModule`, and `Level1MLModule` classes implement machine learning algorithms for classifying the *Test Performed*, *Test Outcome*, and *Level 1* labels, respectively.

The `Level1SymbolicModule` and `Level2Module` classes implement symbolic algorithms for classifying the *Level 1* and *Level 2* labels, respectively.

We use machine learning approaches to classify *Test Performed* and *Test Outcome* because these approaches do not require hard-coding any special cases manually into the source code. Machine learning algorithms are capable of dynamically adapting to new patterns in new training data, and are thus not dependent on the usage of grammatically correct English in the text to classify.

We use a symbolic approach to classify *Level 2* because of the high number of classes: there are more than 600 different *Level 2* classes in the database. This, combined with the low number of labelled data rows for each class, renders machine learning approaches ineffective. Symbolic approaches are also more transparent and interpretable, and are capable of finding new organisms that do not already exist in the database.

We provide both machine learning and symbolic approaches for classifying *Level 1*. In our testing, the machine learning approach achieved high (>95%) accuracy, while the symbolic approach achieved medium (>85%) accuracy.

3.0.1 Vectorizers for machine learning modules

All three machine learning modules use bag-of-words count vectorizers.

The `TestPerformedModule` uses unigrams, bigrams, and trigrams as features. Trigrams are used because "Test not performed" is a very important feature. To remove irrelevant features, a minimum document frequency of 10 is used. Features with variance less than 0.001 are also removed.

The `TestOutcomeModule` uses only unigrams as features, with a minimum document frequency of 5.

The `Level1MLModule` uses unigrams, bigrams, and trigrams as features. Trigrams are used because some organism names are up to three words long. Only the top 200 features, as selected by chi-squared feature selection, are used. No minimum document frequency is used because some features relevant to predicting an organism name may appear only in the few rows labelled as that organism name.

3.1 Instantiation

First, import the classes:

```python
from modules.test_performed_module import TestPerformedModule
from modules.test_outcome_module import TestOutcomeModule
from modules.level_1_ml_module import Level1MLModule
from modules.level_1_symbolic_module import Level1SymbolicModule
from modules.level_2_module import Level2Module
```

To instantiate a `TestPerformedModule`, `TestOutcomeModule`, or `Level1MLModule`, simply call the respective constructor, passing no arguments:

```python
tp_module = TestPerformedModule()
to_module = TestOutcomeModule()
l1ml_module = Level1MLModule()
```

3.1.1 Helper modules

The `Level1SymbolicModule` may **optionally** refer to a trained `TestOutcomeModule` to further improve the former module's predictive power. To use this functionality, pass a **trained** instance of `TestOutcomeModule` to the `Level1SymbolicModule` constructor:

```
# Assume: to_module is a trained TestOutcomeModule instance
l1s_module = Level1SymbolicModule(to_module)
# Now, l1s_module is a newly instantiated Level1SymbolicModule instance
```

Or to disable this functionality, call the `Level1SymbolicModule` constructor with no arguments:

```
l1s_module = Level1SymbolicModule()
# Now, l1s_module is a newly instantiated Level1SymbolicModule instance
```

The `Level2Module` **must** refer to a **trained** *Level 1* module, because *Level 2* is logically a subtype of *Level 1*:

```
# Assume: l1ml_module is a trained Level1MLModule instance
l2_module = Level2Module(l1ml_module)
# Now, l2_module is a newly instantiated Level2Module instance
```

```
# Assume: l1s_module is a trained Level1SymbolicModule instance
l2_module = Level2Module(l1s_module)
# Now, l2_module is a newly instantiated Level2Module instance
```

The `TestOutcomeModule` that the `Level1SymbolicModule` refers to, and the *Level 1* module that the `Level2Module` refers to, are called **helper modules** throughout our technical documentation.

3.1.2 Replacing organism names with a special token

To prevent the `TestPerformedModule` and `TestOutcomeModule` classifiers from overfitting to specific organism names, our pipeline implements an algorithm for replacing all organism names in a classifier's feature space with the "_ORGANISM_" feature.

This functionality is on by default. To turn it off, pass the `organisms=False` flag to the `TestPerformedModule` or `TestOutcomeModule` constructor:

```
tp_module = TestPerformedModule(organisms=False)
to_module = TestOutcomeModule(organisms=False)
```

This functionality is only applicable to `TestPerformedModule` and `TestOutcomeModule`. The `Level1MLModule`, `Level1SymbolicModule`, and `Level2Module` classes do not use this functionality because they classify the *Organism Name* label, so fitting to specific organism names is always desirable.

3.2 Training on existing labelled data

To train a classification module, call its `retrain` method, passing the dataframe containing the training data as a parameter:

```
# Assume: tp_module is a TestPerformedModule instance
# Assume: tp_df is a dataframe containing labelled test_performed rows
tp_module.retrain(tp_df)

# This syntax also works with instances of TestOutcomeModule,
# Level1MLModule, Level1SymbolicModule, and Level2Module
```

The training dataframe must contain 4 columns: *test_key*, *result_key*, *result_full_description*, and a column containing the true labels for the rows. The name of the column containing the labels varies depending on the classification module to train:

Classification module	Label column name
`TestPerformedModule`	*test_performed*
`TestOutcomeModule`	*test_outcome*
`Level1MLModule`, `Level1SymbolicModule`	*level_1*
`Level2Module`	*level_2*

Training dataframes for `Level2Module` must always have an extra *level_1* column.

If a `TestPerformedModule` or `TestOutcomeModule` has the functionality described in 3.1.2 enabled, its training dataframe must contain an additional *candidates* column with the MetaMap *candidate* strings (as described in 2.2.)

3.2.1 Training process for machine learning modules

When a `TestPerformedModule`, `TestOutcomeModule`, or `Level1MLModule` is trained, 5-fold cross-validation is used to select the best machine learning classifier out of a list of candidate classifiers.

For `TestPerformedModule`, the candidate machine learning classifiers are:

- **Logistic Regression** with L2-regularization
- **Logistic Regression** with L1-regularization
- **Random Forest** with 100 trees (and all-core parallel processing)
- **Support Vector Machine** with linear kernel and L2-regularization
- **Support Vector Machine** with linear kernel and L1-regularization

For `TestOutcomeModule`, the candidate machine learning classifiers are:

- **Logistic Regression** with L2-regularization and balanced class weights
- **Random Forest** with 100 trees and balanced class weights (and all-core parallel processing)
- **AdaBoost** with 100 decision stumps
- **Support Vector Machine** with linear kernel, L2-regularization, and balanced class weights

For `Level1MLModule`, the candidate machine learning classifiers are:

- **Logistic Regression** with L2-regularization
- **Random Forest** with 100 trees (and all-core parallel processing)
- **AdaBoost** with 100 decision stumps
- **Support Vector Machine** with linear kernel and L2-regularization

3.2.2 Training process for **Level1SymbolicModule**

When a `Level1SymbolicModule` is trained, a set of all the *level_1* labels in the training dataframe is created. For example, the set may be:

```
{"*not found", "bordetella", "campylobacter", "influzena",
 "parainfluenza or adenovirus", "vibrio", "yersinia"}
```

Next, `"*not found"` is removed from the set. Then, `"influzena"` is replaced with `"influenza"`. Finally, all strings containing `"or"` are split into the constituent organism names.

After these edits, the set looks like this:

```
{"adenovirus", "bordetella", "campylobacter", "influenza",
 "parainfluenza", "vibrio", "yersinia"}
```

This set is called the `Level1SymbolicModule`'s **dictionary**.

3.2.3 Training process for **Level2Module**

When a `Level2Module` is trained, a mapping from *level_1* labels to *level_2* labels is created from the training dataframe.

Each *level_1* label in the training dataframe is mapped to a set of *level_2* labels that have appeared with the *level_1* label in the training dataframe. For example, if the training dataframe is:

...	level_1	level_2
...	yersinia	yersinia frederiksenii
...	clostridium	clostridium difficile
...	vibrio	vibrio vulnificus
...	yersinia	yersinia pestis
...	vibrio	vibrio vulnificus
...	parainfluenza or adenovirus	parainfluenza
...	*not found	*not found

Then the mapping will be:

```
{
    "yersinia": {"yersinia frederiksenii", "yersinia pestis"},
    "clostridium": {"clostridium difficile"},
    "vibrio": {"vibrio vulnificus"},
    "parainfluenza or adenovirus": {"parainfluenza"}
    "*not found": {"*not found"}
}
```

Then, for all entries of the form `"A or B": S` in the mapping, the set `S` is added to the sets for `A` and `B`:

```
{
    "yersinia": {"yersinia frederiksenii", "yersinia pestis"},
    "clostridium": {"clostridium difficile"},
    "vibrio": {"vibrio vulnificus"},
    "parainfluenza or adenovirus": {"parainfluenza"}
    "*not found": {"*not found"},
    "parainfluenza": {"parainfluenza"},
    "adenovirus": {"adenovirus"}
}
```

This mapping is called the `Level2Module`'s **dictionary**.

3.3 Classifying new unlabelled data

All five classification modules must be trained before they are used. Attempting to use an untrained module instance to classify new data will result in a `ValueError` being raised.

To classify new unlabelled data, pass the dataframe containing the new data to the classification module's `classify` method:

```
# Assume: tp_module is a TestPerformedModule instance
# Assume: tp_df is a dataframe containing unlabelled test_performed
# rows
tp_module.classify(tp_df)

# This syntax also works with instances of TestOutcomeModule,
# Level1MLModule, Level1SymbolicModule, and Level2Module
```

3.3.1 Input dataframe format

The dataframe to classify must have 3 columns: *test_key*, *result_key*, and *result_full_description*.

In addition, any dataframe to be classified by `Level1SymbolicModule` or `Level2Module` must contain an additional *candidates* column with the MetaMap *candidate* strings (as described in 2.2.) The *candidates* column is also required in any dataframe to be classified by a `TestPerformedModule` or `TestOutcomeModule` that is set to replace organism names as described in 3.1.2.

If the data is given at the observation level in the dataframe, pass `observations=True` to the `classify` method. This works for all 5 classification modules. The dataframe passed to `classify` must then have an extra *obs_seq_nbr* column.

3.3.2 Output dataframe format

The returned dataframe has columns *test_key* and *result_key*, plus the following module-specific columns:

- **`TestPerformedModule`**: *test_performed_pred*, *test_performed_classifier*, *test_performed_confidence*, *test_performed_confidence_type*
- **`TestOutcomeModule`**: *test_outcome_pred*, *test_outcome_classifier*, *test_outcome_confidence*, *test_outcome_confidence_type*
- **`Level1MLModule`**: *level_1_ml_pred*, *level_1_ml_classifier*, *level_1_ml_confidence*, *level_1_ml_confidence_type*
- **`Level1SymbolicModule`**: *level_1_symbolic_pred*
- **`Level2Module`**: *level_2_pred*

If `observations=True`, the returned dataframe contains an extra *obs_seq_nbr* column.

By default, the `Level1SymbolicModule` and `Level2Module` return the most likely organism name among all the candidate organism names obtained from MetaMap. To return the list of **all** the candidate organism names (serialized as a JSON string,) pass `return_all=True` to the `classify` method.

3.3.3 Classification process for machine learning modules

The trained machine learning classifier is used to output the classifications.

3.3.4 Classification process for `Level1SymbolicModule`

A `Level1SymbolicModule` uses the following algorithm to return the *level_1* prediction for a data row.

If the `Level1SymbolicModule` holds a reference to a trained helper `TestOutcomeModule` as described in 3.1.1, the helper module is used to predict if the new data row has a *negative* test outcome. If so, the `Level1SymbolicModule` returns **not found*. This step is skipped if the `Level1SymbolicModule` does not hold a reference to a helper module.

At this point, if `return_all=True`, all of the preferred names in the MetaMap *candidates* list are returned as an array in a serialized JSON string.

Otherwise, the algorithm iterates over all preferred organism names in the candidate list. For each organism name, if a prefix of the name is in the `Level1SymbolicModule`'s dictionary, the prefix is returned. If no prefix of any organism name is in the dictionary, an arbitrary organism name in the candidates list is returned.

Intuitively, the algorithm prefers to return organism names that already exist in the database, but can also return a new organism name if none of the MetaMap candidates exist in the database.

3.3.5 Classification process for `Level2Module`

A `Level2Module` uses the following algorithm to return the *level_2* prediction for a data row.

First, the helper *Level 1* module is used to produce a *level_1* prediction for the data row:

- If the *level_1* prediction is **not found*, then **not found* is returned as the *level_2* prediction.
- Otherwise, if the *level_1* prediction is not in the dictionary, then **no further diff* is returned.

At this point, if `return_all=True`, all of the preferred names in the MetaMap *candidates* list are returned as an array in a serialized JSON string.

Otherwise, the algorithm iterates over all preferred organism names in the candidate list. For each organism name, if a prefix of the name is in the `Level2Module`'s dictionary and is in the set corresponding to the *level_1* prediction, the prefix is returned as the *level_2* prediction. If no prefix of any organism name is in the dictionary, an arbitrary organism name in the candidates list is returned.

3.4 Saving to disk

Due to the long training time of our classification modules, and for reproducibility reasons, we recommend training the classification modules periodically (e.g.: once per month,) and then saving the trained modules to disk and loading them back into memory every time new data needs to be classified.

To save a classification module, call its `save_to_file` method, passing the absolute path to the *.pkl* file to save to:

```
# Assume: tp_module is a TestPerformedModule instance
tp_module.save_to_file(from_root("pkl\\test_performed_module.pkl"))

# This syntax also works with instances of TestOutcomeModule,
# Level1MLModule, Level1SymbolicModule, and Level2Module.
```

As discussed in 3.3, `Level1SymbolicModule` and `Level2Module` instances may hold references to `TestOutcomeModule` and *Level 1* helper modules, respectively. When a `Level1SymbolicModule` or `Level2Module` is saved to disk, its helper module is **not** saved. It is the pipeline user's responsibility to manually save the helper module.

3.5 Loading from disk

To load a saved `TestPerformedModule`, `TestOutcomeModule`, or `Level1MLModule` instance, call the static `load_from_file` method, passing the absolute path to the *.pkl* file to load from:

```
tp_module = TestPerformedModule.load_from_file(
    from_root("pkl\\test_performed_module.pkl"))

# This syntax also works with TestOutcomeModule and Level1MLModule
```

To load a saved `Level1SymbolicModule` or `Level2Module` instance, call the constructor, passing the helper module instance; then chain a call to the `load_from_file` method, passing the absolute path to the *.pkl* file to load from:

```
l1s_module = Level1SymbolicModule(to_module).load_from_file(
    from_root("pkl\\level_1_symbolic_module.pkl"))
# l1s_module is a loaded Level1SymbolicModule that refers to to_module

l1s_module = Level1SymbolicModule().load_from_file(
    from_root("pkl\\level_1_symbolic_module.pkl"))
# l1s_module is a loaded Level1SymbolicModule with no helper module

l2_module = Level2Module(l1s_module).load_from_file(
    from_root("pkl\\level_2_module.pkl"))
# l2_module is a loaded Level2Module that refers to l1s_module
```

4 Diagnostics

4.1 Benchmarking the time complexity of training

Our pipeline can benchmark the runtime of the training process on varying amounts of training data. The pipeline saves a plot of the runtimes, so it is easy to visualize the relationship between training set size and runtime.

To run the time complexity benchmark, run *driver/complexity.py* after setting the constants at the top of the script to the desired values:

- **`TP_SQL`**, **`TO_SQL`**, **`L1_SQL`**, and **`L2_SQL`** - absolute paths to SQL scripts for extracting training data for the *Test Performed*, *Test Outcome*, *Level 1*, and *Level 2* labels, respectively
- **`SIZES`** - a list of positive integers representing sample sizes (number of rows in the training set) to benchmark training runtime for
- **`ORGANISMS`** - `True` to replace all organism names in the `TestPerformedModule` and `TestOutcomeModule` feature spaces with the "_ORGANISM_" feature as described in 3.1.2; `False` to disable this functionality
- **`SAVE_TO`** - the absolute path to the image file to save the plot to

The SQL scripts must return the columns necessary to run the `retrain` method on each classification module, and must not return empty training sets.

4.1.1 Design considerations

The benchmark's sampling logic implements **random sampling with replacement** to allow greater sample sizes than the number of rows extracted by the SQL script. As a result, there may be duplicate rows in a selected sample, **even when** the sample size is less than the total number of extracted rows.

These duplicates do not affect the training time, and thus do not affect the benchmark results; so the random sampling with replacement algorithm was chosen for ease of implementation.

Using duplicate rows would result in an inaccurate classifier, but this is a non-issue for the purposes of measuring training runtime.

4.2 Benchmarking the accuracy of classification

Our pipeline implements 5-fold cross-validation for computing the expected accuracy of the classification modules. The input to this process is the training set S of labelled *result_full_ descriptions*. In this process, the training set is split into 5 disjoint folds $f_1, \ldots, f_5$. Five passes are made; in the ith pass, a new instance of the classification module is trained on $S - f_i$ and the accuracy A_i of classifying f_i is recorded. The expected accuracy is the arithmetic mean of $A_1, \ldots, A_5$.

To run the benchmark, run *driver/verify.py.* The **`TP_SQL`**, **`TO_SQL`**, **`L1_SQL`**, **`L2_SQL`**, and **`ORGANISMS`** constants at the top of the script have the same meaning as they do in *driver/complexity.py.* The **`SAVE_TO`** constant is the absolute path to the folder to save the results into.

4.2.1 Saved result format

In the `SAVE_TO` folder, the results are stored as 6 text files: one file per fold of the cross-validation, plus one summary file.

A file for one fold contains the accuracy, class-wise precision, class-wise recall, class-wise F1 scores, and Cohen Kappa score achieved on that fold. It also contains the confusion matrix and the list of labels. The ith element of the class-wise precision, recall, and F1 score lists, and the ith row and column of the confusion matrix, refer to the class with the ith label in the list of labels.

The summary file contains the list of accuracies achieved in the 5 folds, their mean, and their standard deviation.

4.2.2 Design considerations

Our benchmark does not support classification at the observation level (set by the `observations =True` flag in the `classify` method for all five classification modules.) There is no labelled data at the observation level, so it is impossible to automatically measure the accuracy of the classification process at the observation level.

Our benchmark does not support returning all candidate organisms (set by the `return_all=True` flag in the `Level1SymbolicModule.classify` and `Level2Module.classify` methods.) Returning all candidates would overcomplicate the process for computing accuracy.

5 Filepaths

To convert a path relative to the project root to an absolute path, use the `from_root` function. Import it as follows:

```
from root import from_root
```

Use it as follows:

```
rel_path = "sql\\train\\level_1.sql"
abs_path = from_root(rel_path)
# Assume the Pipeline project is stored at "U:\\dssg_bccdc\\Pipeline",
# then abs_path is "U:\\dssg_bccdc\\Pipeline\\sql\\train\\level_1.sql"
```

6 Error handling

All of the pipeline entry points in the *driver* folder implement an error handling mechanism, allowing errors to be gracefully logged when the pipeline is run as part of a larger automation process.

When a script in the *driver* folder is run, a log file with the same name as the script is created in the *log* folder in the project root if such a file does not already exist.

If no errors are raised during the execution of the script, no lines are appended to the log file and the script exits with exit code 0.

If an error is raised during the execution of the script, it is gracefully caught and its stack trace, along with the current timestamp, is appended to the log file. The script then immediately exits with exit code 1, indicating an error occurred.

7 Folder structure

Our pipeline's folder structure is:

- *docs* - folder containing this technical documentation, in LaTeX and PDF formats
- *driver* - Python scripts that are entry points to the pipeline
- *io_* - Python functions for database and filesystem operations. This folder is named *io_* because *io* is a built-in Python module.
- *libs* - *.jar* files containing compiled Java code for interacting with the MetaMap Java API server
- *log* - folder for log files to be written to. Log files are generated during pipeline execution as described in 6.
- *modules* - Python classes implementing the classification modules for the *Test Performed*, *Test Outcome*, *Level 1*, and *Level 2* labels
- *pkl* - Python *pickle* files storing sample, pre-trained instances of the classification modules
- *results* - folder for the diagnostics results (time complexity plot and cross-validation details) to be written to
- *sql* - sample SQL scripts for extracting training and test data sets from the database
- *util* - Python functions used as private helper functions by code in other folders

8 Dependencies

Our pipeline depends on these external libraries:

8.1 Python dependencies

- **pyodbc** - a driver for interfacing with the Microsoft SQL Server database
- **SQLAlchemy** - an ORM to simplify database operations

- **pandas** - dataframes for in-memory storage and wrangling of tabular data
- **scikit-learn** - implementations of machine learning algorithms
- **numpy** - implementations of multidimensional arrays and matrices
- **scipy** - implementations of sparse matrices and scientific computing algorithms
- **matplotlib** - a plotting library for drawing the time complexity benchmark results
- **Py4J** - an interface for calling Java code from Python code

8.2 Java dependencies

- **MetaMap Java API** - an interface for running the MetaMap annotator on string data
- **org.json** - a JSON parser for converting JSON-formatted strings to Java objects
- **Py4J** - an interface for calling Java code from Python code