Module interface

The module interface connects all code for a module in terms of parsing, calculating metrics, and plotting. The module interfaces are located here:

`ProteoBench/proteobench/modules/[module]`

Creating a new interface is a good place to start when implementing a new module. First think about potentially shared code with other modules, usually if there is a large portion of shared code consider making a new "[module]_base" where shared code can be implemented and inherited to the specific modules.

The following functions are likely needed by code in other parts of the package:

1. benchmarking (entry point for running the module)
2. generate_intermediate (parsing related)
3. generate_datapoint (scoring related)
4. add_current_data_point (resulting mostly used for plotting; add the point for plotting)

Also see the following code for an abstract class:

ProteoBench\proteobench/modules/interfaces.py

For an implementation of a "[module]_base" interface see:

ProteoBench\proteobench\modules\dda_quant_base/module.py

For an implementation of a "[module]" interface see:

ProteoBench\proteobench\modules\dda_quant_ion/module.py

As can be seen in the examples above, the benchmarking function needs to be provided with the output from a proteomics workflow, the name of the workflow, and the current datapoints that were previously added. In the example the function returns an intermediate format (format that is equal for all workflows and is required to calculate the metrics). All the datapoints – i.e., calculated metrics – and the input dataframe after renaming columns.

Let's start implementing the DDA peptidoform module. The imports are taken from the ion module, and are changed to reflect the new module (see the bold and underlined parts):

```
from __future__ import annotations

import pandas as pd
from pandas import DataFrame

from proteobench.exceptions import (
    ConvertStandardFormatError,
    DatapointAppendError,
    DatapointGenerationError,
    IntermediateFormatGenerationError,
    ParseError,
    ParseSettingsError,
    QuantificationError,
```

```
)
from proteobench.io.parsing.parse_peptidoform import load_input_file
from proteobench.io.parsing.parse_settings_peptidoform import ParseSettingsBuilder
from proteobench.modules.dda_quant_base.module import Module
from proteobench.score.quant.quantscores import QuantScores
from proteobench.utils.quant_datapoint import Datapoint
```

This also means that in io.parsing.* we need to add the appropriate peptidoforms parsers. See the parsing section for more information how to add these parsers. We are using the same quant metrics and how the datapoints are initialized, so these do not need to be changed.

After the imports the class is defined as:

```
class PeptidoformModule(Module):
    """Object is used as a main interface with the Proteobench library within
the module."""

    def __init__(self, token):
        super().__init__(token)
        self.precursor_name = "peptidoform"

    def is_implemented(self) -> bool:
        """Returns whether the module is fully implemented."""
        return False
```

The name of the class is changed to reflect the new module. The name used for the modality quantified is defined in the init, this is change to peptidoform. For now “is_implemented” is set to False, once finished we can set this to True.

The benchmarking function is not changed as the only thing that is really different is the parsing of the peptidoforms compared to the peptide ions.

Parsing

First step in the parsing process is to read the input files as a pandas dataframe (see ProteoBench\proteobench\io\parsing/parse_ion.py):

```
def load_input_file(input_csv: str, input_format: str) -> pd.DataFrame:
    """Method loads dataframe from a csv depending on its format."""
    input_data_frame: pd.DataFrame


    if input_format == "MaxQuant":
        input_data_frame = pd.read_csv(input_csv, sep="\t", low_memory=False)
    elif input_format == "AlphaPept":
        input_data_frame = pd.read_csv(input_csv, low_memory=False)
    elif input_format == "Sage":
        input_data_frame = pd.read_csv(input_csv, sep="\t", low_memory=False)
    elif input_format == "FragPipe":
```

```
        input_data_frame = pd.read_csv(input_csv, low_memory=False, sep="\t")
        input_data_frame["Protein"] = input_data_frame["Protein"] + "," + input_data_frame["Mapped
Proteins"].fillna("")
    elif input_format == "WOMBAT":
        input_data_frame = pd.read_csv(input_csv, low_memory=False, sep=",")
        input_data_frame["proforma"] = input_data_frame["modified_peptide"]
    elif input_format == "Proline":
        input_data_frame = pd.read_excel(
            input_csv,
            sheet_name="Quantified peptide ions",
            header=0,
            index_col=None,
        )

        # TODO this should be generalized further, maybe even moved to parsing param in toml
        input_data_frame["modifications"] = input_data_frame["modifications"].fillna("")
        # input_data_frame.fillna({"modifications": ""}, inplace=True)
        input_data_frame["proforma"] = input_data_frame.apply(
            lambda x: aggregate_modification_column(x.sequence, x.modifications),
            axis=1,
        )
    elif input_format == "i2MassChroQ":
        input_data_frame = pd.read_csv(input_csv, low_memory=False, sep="\t")
        input_data_frame["proforma"] = input_data_frame["ProForma"]
    elif input_format == "Custom":
        input_data_frame = pd.read_csv(input_csv, low_memory=False, sep="\t")
        input_data_frame["proforma"] = input_data_frame["Modified sequence"]

    return input_data_frame
```

Any specific parsing that cannot be generalized easily is also done here. That means if there are columns separated that should be merged here. Also, if the modified sequence column already follows the proforma format we make that column here, no further changes are needed. We will start with the support in this new module with just two workflows:

```
def load_input_file(input_csv: str, input_format: str) -> pd.DataFrame:
    """Method loads dataframe from a csv depending on its format."""
    input_data_frame: pd.DataFrame
    if input_format == "WOMBAT":
        input_data_frame = pd.read_csv(input_csv, low_memory=False, sep=",")
        input_data_frame["proforma"] = input_data_frame["modified_peptide"]
    elif input_format == "Custom":
        input_data_frame = pd.read_csv(input_csv, low_memory=False, sep="\t")
        input_data_frame["proforma"] = input_data_frame["Modified sequence"]

    return input_data_frame
```

The remaining functions stay the same. The parse_settings_peptidoform.py in the same folder are further adjusted:

```
class ParseSettingsBuilder:
    def __init__(self, parse_settings_dir=None):
        if parse_settings_dir is None:
            parse_settings_dir = os.path.join(os.path.dirname(__file__), "io_parse_settings")

        self.PARSE_SETTINGS_FILES = {
            "WOMBAT": os.path.join(parse_settings_dir, "parse_settings_wombat.toml"),
            "Custom": os.path.join(parse_settings_dir, "parse_settings_custom_DDA_quand_peptidoform.toml"),
        }
        self.PARSE_SETTINGS_FILES_MODULE = os.path.join(parse_settings_dir, "module_settings.toml")
        self.INPUT_FORMATS = list(self.PARSE_SETTINGS_FILES.keys())
        # Check if all files are present
        cwd = os.getcwd()
        missing_files = [file for file in self.PARSE_SETTINGS_FILES.values() if not os.path.isfile(file)]
        if not os.path.isfile(self.PARSE_SETTINGS_FILES_MODULE):
            missing_files.append(self.PARSE_SETTINGS_FILES_MODULE)

        if missing_files:
            raise FileNotFoundError(f"The following parse settings files are missing: {missing_files}")

    def build_parser(self, input_format: str) -> ParseSettings:
        toml_file = self.PARSE_SETTINGS_FILES[input_format]
        parse_settings = toml.load(toml_file)
        parse_settings_module = toml.load(self.PARSE_SETTINGS_FILES_MODULE)

        parser = ParseSettings(parse_settings, parse_settings_module)
        if "modifications_parser" in parse_settings.keys():
            parser = ParseModificationSettings(parser, parse_settings)
        return parser
```

Only non-supported search engines are removed, no further changes are made. The most important function to change is the “convert_to_standard_format”, that converts the dataframe in such a way that the column names are the same, so further parsing the same column names with the same kind of information can be expected. This is mostly done with the information found in the .toml files in “proteobench/io/parsing/io_parse_settings”.

Calculating metrics

The first step towards calculating the metrics is to generate an intermediate file that contains only the data needed to calculate the metrics. For example, in this case we need to calculate

ratios between the same peptidoforms of the same species found in different ratios in different conditions.

For an example see:

ProteoBench\proteobench\score\quant\ quantscores.py

Here we should adjust the function:

QuantScores.generate_intermediate()

Then for the calculation of the final metric (using the intermediate file) we initialize a datapoint through:

Datapoint.generate_datapoint()

This datapoint should contain all relevant information for plotting and calculating the metrics. A series is returned that can be combined to create a dataframe where each row is a workflow and associated performance.

Plotting

Plotting is usually called from the webinterface and the plot_metric function is used. Again, in the current peptidoforms module there is no need to change the code. If you need to make a new plot, make sure the “plot_metric” function returns a plotly.graph_objects.

Webinterface

See an example of the implementation here:

Proteobench/webinterface/pages/DDA_quant_ion.py

One of the important parts here is to initialize the webpage with all correct and relevant variables:

```
class StreamlitUI:
    def __init__(self):
        self.variables_dda_quant: VariablesDDAQuant = VariablesDDAQuant()
        self.texts: Type[WebpageTexts] = WebpageTexts
        self.texts.ShortMessages.title = "DDA peptidoform quantification"

        self.user_input: Dict[str, Any] = dict()

        pbb.proteobench_page_config()
        pbb.proteobench_sidebar()

        if self.variables_dda_quant.submit not in st.session_state:
            st.session_state[self.variables_dda_quant.submit] = False
        try:
            token = st.secrets["gh"]["token"]
        except KeyError:
            token = ""
```

```
        self.ionmodule: IonModule = IonModule(token=token)
        self.parsesettingsbuilder = ParseSettingsBuilder()

        self.quant_uiobjects = QuantUIObjects(self.variables_dda_quant, self.ionmodule,
self.parsesettingsbuilder)

        self._main_page()
```

These variables are configured in:

/Proteobench/webinterface/pages/pages_variable/dda_quant_variables.py

It is important to change all the variable names too, as these will be kept in the session state. As a result if variables are shared between pages they will keep the same values too.

The following function of the page takes care of creating the webpage:

```
def _main_page(self) -> None:
        """
        Sets up the main page layout for the Streamlit application.
        This includes the title, module descriptions, input forms, and
configuration settings.
        """
        self.quant_uiobjects.create_text_header()
        self.quant_uiobjects.create_main_submission_form()
        self.quant_uiobjects.init_slider()

        if self.quant_uiobjects.variables_quant.fig_logfc in st.session_state:
            self.quant_uiobjects.populate_results()

        self.quant_uiobjects.create_results()
```

Here the quant_uiobjects is part of the base_pages that help with making parts of the objects. When the new module is very close in terms of what it needs to show and functionality you can keep using this object. However, in the case of new functionality that is very different, it is probably better if a completely new page is created.

Github

You will need two new repositories, follow these steps:

1. Make a new repository in the Proteobench organisation and give it a sensible name
2. See Proteobench/Results_quant_ion_DDA: Results_Module2_quant_DDA (github.com) as an example for files that are likely required
3. Login to proteobot (ask for the login details to relevant people)
4. Make a fork of the new repository