Hopworks-Integration

Documentation

This project has been built to be called using the CLI commands. This application provides the abstraction with the Hopsworks Feature Store and lets the user build and retrieve the required Hopsworks feature groups and feature view.

Environment Setup:

This application is coded in python language using the PyCharm IDE. There are certain libraries that are required to be installed before running this application. These libraries are also mentioned in the requirements.txt file in the project directory and are also listed below:

- hopsworks==3.0.3
- hsfs==3.0.2
- pandas==1.5.0
- SQLAlchemy==1.4.41
- pytest==7.1.3

Other than the libraries, there is a DB post script which is necessary to be run before running the application since it does some changes in the 'NBA' database and adds a column of Poss_PT. The details of this script can be found here.

Calling Modes:

This application has two execution modes:

1. It can either be called directly using the CLI commands, in that case you need to pull this Github repo and execute the ***cmd.py python*** file present in the ***src*** directory. ***To be able to run this command, make sure to switch the directory to src.***
2. It can also be called using import command by installing this library from the pypi and using it in your existing project. The library can be installed from pypi using the following command:

```
pip install Hopsworks-Integration==0.0.2
```

The details of both the calling modes are described below. The user can choose either one of them.

CLI Calling Mode:

Installation:

This mode does not need any installation, instead you need to clone the Github repo from the following URL: https://github.com/muneebsmh/hopsworks-integrations.git
After cloning the code, you can directly use the code present in ***hopsworks-integration/src/*** directory and run the commands given in the example here.

Runtime Arguments:

The application can be called using the following CLI command or by setting these arguments in the “Run/Debug Configurations” in any editor:

```
python.exe cmd.py --conf /path/to/config/conf.json --run_mode get_group
--group_name gp_box_score --group_version 1
```

Explanation of the Runtime Arguments:

1. *--conf:* This tells the path to the config file containing database and hopsworks credentials.
2. *--build_conf:* This tells the path to the groups.json or views.json config file where you will define the source and destination and the schema of the feature group or view.
3. *--run_mode:* This tells the application what type of operation to perform.
4. *--group_name:* This is the required feature group name that is needed to be created or retrieved.
5. *--group_version:* This the required version of the feature group.
6. *--view_name:* This is the required feature view name that is needed to be created or retrieved.
7. *--view_version:* This the required version of the feature view.
8. *--file_path:* This is the directory path of the output files when you are exporting feature group or view to the .csv file.
9. *--file_name:* This is the file name of the output files when you are exporting feature group or view to the .csv file.

Mandatory arguments:

conf, run_mode

Optional arguments:

build_conf, group_name, group_version, view_name, view_version, file_path, and file_name are optional and will be used based on the operation type. For e.g. if you are running only get_group then you don't need to specify build_conf argument, but you when you are running build_group, then you need to specify build_conf argument but you don't need to specify group version in that. See the examples below for clarification.

Example Runs:

```
`python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode
get_group --group_name gp_game_base --group_version 1`
`python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode
get_group_features --group_name gp_box_score --group_version 1 --features
adv_idx_uuid,box_type,mp`
`python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/groups.json" --run_mode build_group_from_db --group_name
gp_game_base`
`python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/groups.json" --run_mode build_group_from_view
--group_name test_group_from_view`
`python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/groups.json" --run_mode build_group_from_file
--group_name test_group_from_file`
`python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode
export_group_to_file --group_name gp_game_base --group_version 1
--file_path "/path/to/export/folder/" --file_name
gp_game_base_export.csv`
`python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode
drop_group --group_name gp_game_base --group_version 1`
`python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode get_view
--view_name t1vst2 --view_version 1`
`python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode
get_view_features --view_name t1vst2 --view_version 1  --features
uuid,box_type,mp`
`python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/views.json" --run_mode build_view --view_name t1vst2`
`python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode
export_view_to_file --view_name t1vst2 --view_version 1 --file_path
"/path/to/export/folder/" --file_name t1vst2_view_export.csv`
`python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode
drop_view --view_name t1vst2 --view_version 1`
```

Library Calling Mode (Recommended Way):

Installation:

The run modes, and functions are the same as described in the *CLI Calling Mode*. The only difference is that, in this mode you can install this project as a library in your existing python project and use like any other libraries.

The steps to install and use this as a standalone library are:

1. To install this library, you first need to install pip on your respective OS. The installation guides for each OS can be found below:
    a. Windows: https://www.geeksforgeeks.org/how-to-install-pip-on-windows/
    b. MacOS: https://www.geeksforgeeks.org/how-to-install-pip-in-macos/
    c. Linux: https://www.geeksforgeeks.org/how-to-install-pip-in-linux/

2. Create an empty python project or open up your existing python project in which you want to use any of the 12 functions written in this library.
3. Install this library using **`pip install Hopsworks-Integration==0.0.2`**
4. This command will install all the required dependencies in your project.
5. Once the installation is done, the library can be used by:

```
import src.Cmd as s

# For operations which return something like get_group, get_view, get_group_features,
get_view_features, the returned value should be saved in a dataframe.
arguments = []
df = s.main(arguments)


# For operations which do not return anything like build_group_from_db, drop_group, etc.
the returned value does not need to be saved in a variable
arguments = []
s.main(arguments)

# arguments is a list, which is composed of all the CLI arguments that you write in the CLI
calling mode.
# For e.g.:
# To get group features in CLI, you will write:
# python.exe cmd.py --conf /path/to/config/conf.json --run_mode get_group_features
--group_name gp_box_score --group_version 1 --features adv_idx_uuid,box_type,mp

# Similarly, to get group features in Library Mode, you will write:
# import src.Cmd as s
# arguments =
["--conf","/path/to/config/conf.json","--run_mode","get_group_features","--group_name","gp_
box_score","--group_version","1","--features","adv_idx_uuid,box_type,mp"]
# df = s.main(arguments)
```

Similarly, to build group, the arguments can be provided like:

```
import src.Cmd as s

arguments =
["--conf","/path/to/config/conf.json","--build_conf","/path/to/config/groups.json","--run_m
ode","build_group_from_db","--group_name","gp_game_base"]
s.main(arguments)
```

Similarly, to get group handle, the arguments can be provided like:

```
import src.Cmd as s

arguments =
["--conf","/path/to/config/conf.json","--run_mode","get_group","--group_name","gp_game_bas
e","--group_version","1"]
```

```
df = s.main(arguments)
```

Configurations:

This application is totally configurable and it picks runtime configuration from the config files present in the project directory or they can be present elsewhere but their path would needed to be specified while calling the application.

There are two types of configurations that are used in this application, the primary configuration which is config.json file and the build_conf.json which is used to create feature group or view.

1) **Conf.json:** This is the primary config file and is mandatory to run this application. This file holds details such as the source database credentials, the hopsworks account credentials and key, and the logs path where the application logs will be dumped on each run. The format of the file is given below:

```
{
  "host": "localhost",
  "database": "nba",
  "username": "root",
  "password": "root",
  "hopsworks_host": "c.app.hopsworks.ai",
  "hopsworks_port": "443",
  "hopsworks_project": "basketball_feature_tool",
  "hopsworks_api_key_value": "oBo1bB2QzeD6AIzT.WXujCzTCPaHI2yKe0GnMHu7IIe0dRPKBLKaDz0tbekGlfFoLbBIDMwfGYMyQPElv",
  "log_path": "D:\\Documents\\Office Documents\\Toptal\\TSGNC\\basketball-feature-tool\\logs"
}
```

Before running the application, the user has to provide the configuration in the above-mentioned format and it is necessary to use the same naming convention. For e.g. if the user renames the “hostname” instead of “host” then the application will crash. Exact names should be used. The user should only change the value part of the config file and should never change the name of keys. The sample configuration file is already present in the ***basketball-feature-tool/config*** directory in the project. The user can either change this config file or create another file and use that one. The important thing is that wherever the user creates this configuration file, they have to insert the full path of the config file in the CLI command while calling the application using the ***–conf*** command line argument.

Example: `python.exe cmd.py **--conf "/path/to/config/conf.json"** --run_mode get_group --group_name gp_game_base --group_version 1`

2) **Build_Conf.json:** There is another configuration type called Build_conf.json file. This config is optional to mention in the CLI command, and should be used only while running build commands for creating ***Feature Group or Feature View.*** Other than that, this config will not be needed for running any other type of commands like retrieving a feature group or deleting a feature view, etc. The reason this config is only required in build commands is that, it holds the metadata or the required elements that are essential to create a feature group or view.

   Creating a feature group requires many essential elements such as group name, version, the source table and the columns, primary key, partition key and so on. Similarly, there are certain required elements for feature view as well. Therefore, this argument holds the path of the groups.json or view.json files which hold the required elements needed to built either of the entities.

   The syntax of this argument while creating the feature group is:

```
`python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/groups.json" --run_mode build_group_from_db --group_name
gp_game_base`
```

   The syntax of this argument while creating the feature view is:

```
`python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/views.json" --run_mode build_view --view_name t1vst2`
```

Specifications for Build_Conf:

As mentioned above, build_conf argument takes the path of the json file that contains the metadate for creating either groups or views on the hopsworks project. There should be separate files for groups and views for the sake of clarity and it can be present anywhere in the system as long as the full path is provided in the ***–build_conf*** argument.

There are sample ***groups.json*** and ***views.json*** files that are present in the ***basketball-feature-tool/config*** directory in the project.

Explanation of the content in the groups.json file:

The groups.json file contains the metadata for all the groups that are needed to be created on hopsworks. You don’t need to create separate file for each group as the groups are separated using key-value pairs. For e.g. if you provide the the build_group_from_db command for gp_game_base group, the command would like as follows:

```
`python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/groups.json" --run_mode build_group_from_db --group_name
gp_game_base`
```

And if the groups.json file has keys for three groups gp_game_base, gp_box_score, and gp_adv_stats as follows:

```
{
gp_game_base: {
      <metadata for game_base>
},
gp_box_score: {
      <metadata for box_score>
},
gp_adv_stats: {
      <metadata for adv_stats>
}
}
```

Then the metadata for gp_game_base will be picked up, and all the required fields will be taken for this group only.

If you want to add another group, then just copy paste the entire metadata for any group and change the name of the group like below:

```
{
gp_game_base: {
      <metadata for game_base>
},
gp_box_score: {
      <metadata for box_score>
},
gp_adv_stats: {
      <metadata for adv_stats>
},
gp_new_group: {
      <metadata for new group>
}
}
```

Do note that this is not the hopsworks group name that would be visible on the hopsworks. This is just to choose the required metadata of the feature group that you need to create. Therefore, make sure that this name should always be the same as the name you pass in the CLI command along with group_name argument.

Build_Group_From_DB:

Let's talk about the elements or metadata inside the block of a particular feature group.

- **table_name:** This is the source DB table name which would be used to fetch data from.
- **hopsworks_group_name:** This is the actual group name that would be visible on the hopsworks project UI.

- **hopsworks_group_version:** The version of the group to be made (default is 1). If you run the build_group_from_db command of the same version of the group which is already present on the hopsworks then it would only update the data, will not recreate the group.
- **description:** The textual description of the group.
- **primary_key:** The list of columns that would be used as primary key. Primary key is mandatory.
- **partition_key:** The list of columns that would be used as partitions.
- **features:** The list of database columns that you want to include in the feature group. You can also give alias using 'AS' keywork.
- **derived_features:** This list of derived/calculated columns from the existing database columns. For e.g. 'NBA' AS League

example:

```
{
   "gp_game_base": {
      "table_name": "adv_idx",
      "hopsworks_group_name": "gp_game_base",
      "hopsworks_group_version": 1,
      "description": "to be filled",
      "primary_key": [
        "uuid"
      ],
      "partition_key": [
      ],
      "columns": [
        "uuid",
        "Season_Year AS Season",
        "G_Type",
        "Date",
        "Team1",
        "Team2",
        "Year",
        "Month",
        "Day",
        "T1_Loc",
        "T1_sea_g_played",
        "T1_sea_g_played_h",
        "T1_sea_g_played_a",
        "T1_rest_day",
        "T1_B2B",
        "T2_Loc",
        "T2_sea_g_played",
        "T2_sea_g_played_h",
        "T2_sea_g_played_a",
        "T2_rest_day",
        "T2_B2B"
      ],
      "derived_columns": [
        "'NBA' AS League"
      ]
   },
```

Build_Group_From_View:

Let's talk about the elements or metadata inside the block of a particular feature group.

- **hopsworks_view_name:** This is the source view name present in the hopsworks project which would be used to fetch data from.
- **Hopsworks_view_version**: This is the source view version present in the hopsworks project which would be used to fetch data from.

- **hopsworks_group_name:** This is the actual group name that would be visible on the hopsworks project UI.
- **hopsworks_group_version:** The version of the group to be made (default is 1). If you run the build_group_from_db command of the same version of the group which is already present on the hopsworks then it would only update the data, will not recreate the group.
- **description:** The textual description of the group.
- **primary_key:** The list of columns that would be used as primary key. Primary key is mandatory.
- **partition_key:** The list of columns that would be used as partitions.
- **features:** The list of features from view that you want to include in the feature group.

example:

```
"test_group_from_view":
{
  "hopsworks_view_name": "tlvst2",
  "hopsworks_view_version": 1,
  "hopsworks_group_name": "test_group_from_view",
  "hopsworks_group_version": 1,
  "description": "to be filled",
  "primary_key": [
    "adv_idx_uuid",
    "Box_Type"
  ],
  "partition_key": [
    ""
  ],
  "features": [
    "uuid",
    "Box_Type",
    "MP",
    "T1_PTS",
    "T1_Result",
    "T1_FGM",
    "T1_FGA",
    "T1_FG_Pct",
    "T1_3PM",
    "T1_3PA",
    "T1_3P_Pct"
  ]
},
```

Build_Group_From_File:

Let's talk about the elements or metadata inside the block of a particular feature group.

- **file_path:** The complete path of the source .csv file
- **file_name**: The name of the source .csv file.
- **sep:** column delimiter of the .csv file.
- **index_col:** Either true of false.
- **header_row:** Index of the header row. 0 means that the first row in the .csv file is the header row.
- **quote_char:** quote character. Like: “”\”
- **escape_char:** escape character. Like: “”\”
- **hopsworks_group_name:** This is the actual group name that would be visible on the hopsworks project UI.

- **hopsworks_group_version:** The version of the group to be made (default is 1). If you run the build_group_from_db command of the same version of the group which is already present on the hopsworks then it would only update the data, will not recreate the group.
- **description:** The textual description of the group.
- **primary_key:** The list of columns that would be used as primary key. Primary key is mandatory.
- **partition_key:** The list of columns that would be used as partitions.
- **features:** The list of features from view that you want to include in the feature group.

example:

```
"test_group_from_file":
{
  "file_path": "D:\\Documents\\Office Documents\\Toptal\\TSGNC\\basketball-feature-tool\\files\\",
  "file_name": "import.csv",
  "sep": "|",
  "index_col": "None",
  "header_row": 0,
  "quote_char": "\"",
  "escape_char": "\"",
  "hopsworks_group_name": "test_group_from_file",
  "hopsworks_group_version": 1,
  "description": "to be filled",
  "primary_key": [
    "adv_idx_uuid",
    "Box_Type"
  ],
  "partition_key": [
    ""
  ],
  "features": [
    "adv_idx_uuid",
    "Box_Type",
    "MP",
    "T1_PTS",
    "T1_Result",
    "T1_FGM",
    "T1_FGA",
    "T1_FG_Pct",
    "T1_3PM",
    "T1_3PA",
    "T1_3P_Pct"
  ]
}
```

Explanation of the content in the views.json file:

The views.json file contains the metadata for all the views that are needed to be created on hopsworks. You don't need to create separate file for each view as the views are separated using key-value pairs just like the groups.json. For e.g. if you provide the the build_view command for t1vst2 view, the command would like as follows:

```
`python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/views.json" --run_mode build_view --view_name t1vst2`
```

And if the views.json file has keys for three views t1vst2, view2, and view3 as follows:

```
{
```

```
t1vst2: {
      <metadata for t1vst2>
},
view2: {
      <metadata for view2>
},
view3: {
      <metadata for view3>
}
}
```

Then the metadata for t1vst2 will be picked up, and all the required fields will be taken for this view only.

If you want to add another view, then just copy paste the entire metadata for any view and change the name of the view like below:

```
{
t1vst2: {
      <metadata for t1vst2>
},
view2: {
      <metadata for view2>
},
view3: {
      <metadata for view3>
},
new_view: {
      <metadata for new view>
}
}
```

Do note that this is not the hopsworks view name that would be visible on the hopsworks. This is just to choose the required metadata of the feature view that you need to create. Therefore, make sure that this name should always be the same as the name you pass in the CLI command along with view_name argument.

Let's talk about the elements or metadata inside the block of a particular feature view.

- **hopsworks_view_name:** This is the actual view name that would be visible on the hopsworks project UI.
- **hopsworks_view_version:** The version of the view to be made (default is 1). If you run the build_view command of the same version of the view which is already present on the hopsworks then it would fail the execution.
- **description:** The textual description of the view.
- **labels:** The list of columns that would be treated as labels.

- **features:** This is a nested dictionary. Like database views, hopsworks views are also a conceptual table but in the backend they are composed of the columns/features from different feature groups. Therefore, this element tells the build_view statement that how many features are to be chosen from different feature groups, how to join different the feature groups, put necessary filters, and the join type and sequence. The elements of the features dictionary are:
    - **name:** name of the feature group.
    - **features:** features to be fetched from this feature group
    - **version:** version of the feature group
    - **joining_sequence:** the first group would be 1, the second group be 2, and so on. This joining sequence basically tells that which group is the main group. If there are three feature groups A, B, and C and we assign A=1, B=2, C=3, then the hopworks would choose A as the main group and will first join A with B and then A with C. Nested joins are not supported in hopsworks so A joins B, and B joins C is not possible. Write you joining logically keeping this limitation in mind.
    - **joining_columns:** This would be empty for the first (main) group since it is the only group, but for the other joining groups it should include the list of columns required to perform join.
    - **join_type:** The required join type, inner, left, etc.
    - **filters:** If you need to put any filter or restrict some data to be the part of the feature view so you can specify the filters in this field like:
        - "filters": "Feature('t2_poss_cnt')==108"
        - "filters": "(Feature('g_type') == 2) & (Feature('team1') == 'BOS')"
        - "filters": "(Feature('box_type')=='FT') | (Feature('mp')==48)"

example:

Operation Types:

get_group:

This operation is used to get the group handle from the hopworks project. You can then play around with this group handle like exporting it to the dataframe.

This operation takes group name and group version as the command line arguments and returns the respective feature group handle.

example:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode get_group
--group_name gp_game_base --group_version 1
```

get_group_features:

This operation is used to fetch certain features from the feature group. The list of features are provided in a comma separated string to the CLI command.

This operation takes group name, group version, and the comma-separated list of features as the command line arguments and returns the respective feature dataframe.

example:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode
get_group_features --group_name gp_box_score --group_version 1 --features
adv_idx_uuid,box_type,mp
```

build_group_from_db:

This operation builds a feature group from the DB table. It is recommended to read the detailed documentation first to understand the command and its different elements which can be found here.

To run this command, following are the steps:

1. In the config directory, find or create the groups.json file or with any other name but it should be a json file and with a proper format as mentioned in the sample groups.json file.
2. groups.json file contains the list of feature groups that can be created. Each feature group has its own key and the respective value tells the details like the name, version, primary key, partitions, and description of the hopsworks feature group to be made, the database table and required columns which will be used in creating that feature group. Make sure to use the same name of the feature group in the CLI argument which is mentioned as the key in the groups.json file.

3. Once you have created or configured the json file for groups, you need to tell the path of this file as the command line argument using --build_conf argument while running the application.
4. Once done, you just need to open a command prompt or terminal and write the build_group_from_db command mentioning the correct path for conf.json, and groups.json files respectively. For e.g. assuming you have made the changes in the groups.json and conf.json file, and the json key name in the groups.json file is "gp_game_base", then your command should look like:
```
python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/groups.json" --run_mode build_group_from_db
--group_name gp_game_base
```
5. Sample **conf.json** and **groups.json** files can be found at *config/conf.json* and *config/groups.json* respectively.

example:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/groups.json" --run_mode build_group_from_db --group_name
gp_game_base
```

build_group_from_view:

This operation builds a feature group from the feature view. It is recommended to read the detailed documentation first to understand the command and its different elements which can be found here.

Just like the build_group_from_db command, this command also takes required arguments from the groups.json file (or whichever file you have set). To run this command, following are the steps:

1. In the config directory, find or create the groups.json file or with any other name but it should be a json file and with a proper format as mentioned in the sample groups.json file.
2. groups.json file contains the list of feature groups that can be created. Each feature group has its own key and the respective value tells the details like the name, version, primary key, partitions, and description of the hopsworks feature group to be made, the feature view and version which will be used in creating that feature group. Make sure to use the same name of the feature group in the CLI argument which is mentioned as the key in the groups.json file.
3. Once you have created or configured the json file for groups, you need to tell the path of this file as the command line argument using --build_conf argument while running the application.
4. Once both these configurations files are ready, you just need to open a command prompt or terminal and write the build_group_from_view command mentioning the correct path for conf.json, and groups.json files respectively. For e.g. assuming you have made the changes in the groups.json and conf.json file, and the json key name in the groups.json file is "test_group_from_view", then your command should look like:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/groups.json" --run_mode build_group_from_view
--group_name test_group_from_view
```

5. Sample **conf.json** and **groups.json** files can be found at *config/conf.json* and *config/groups.json* respectively.

example:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/groups.json" --run_mode build_group_from_view --group_name
test_group_from_view
```

build_group_from_file:

This operation builds a feature group from the .csv file. It is recommended to read the detailed documentation first to understand the command and its different elements which can be found here.

Just like the build_group_from_db command, this command also takes required arguments from the groups.json file(or whichever file you have set). To run this command, following are the steps:

1. In the config directory, find or create the groups.json file or with any other name but it should be a json file and with a proper format as mentioned in the sample groups.json file.
2. groups.json file contains the list of feature groups that can be created. Each feature group has its own key and the respective value tells the details like the name, version, primary key, partitions, and description of the hopsworks feature group to be made, the feature view and version which will be used in creating that feature group. Make sure to use the same name of the feature group in the CLI argument which is mentioned as the key in the groups.json file.
3. Once you have created or configured the json file for groups, you need to tell the path of this file as the command line argument using --build_conf argument while running the application.
4. Once both these configurations files are ready, you just need to open a command prompt or terminal and write the build_group_from_view command mentioning the correct path for conf.json, and groups.json files respectively. For e.g. assuming you have made the changes in the groups.json and conf.json file, and the json key name in the groups.json file is "test_group_from_file", then your command should look like:
```
python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/groups.json" --run_mode build_group_from_file
--group_name test_group_from_file
```

5. Sample **conf.json** and **groups.json** files can be found at *config/conf.json* and *config/groups.json* respectively.

example:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/groups.json" --run_mode build_group_from_file --group_name
test_group_from_file
```

export_group_to_file:

This operation is used to export the feature group present on the hopsworks project to the .csv file. This is helpful if you want to export the data to some other project or do some analysis on external tools or excel sheet.

This operation takes group name, group version, destination directory, and file name as the command line arguments and outputs the feature group in the form of .csv file to the described destination directory. Be very sure in writing the complete desired destination path.

example:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode
export_group_to_file --group_name gp_game_base --group_version 1 --file_path
"/path/to/export/folder/" --file_name gp_game_base_export.csv
```

drop_group:

This operation takes group name and group version as the command line arguments and deletes the feature group from the hopsworks project.

example:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode drop_group
--group_name gp_game_base --group_version 1
```

get_view:

This operation is used to get the view handle from the hopworks project. You can then play around with this view handle like exporting it to the dataframe.

This operation takes view name and view version as the command line arguments and returns the respective feature view handle.

example:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode get_view
--view_name t1vst2 --view_version 1
```

get_view_features:

This operation is used to fetch certain features from the feature view. The list of features are provided in a comma separated string to the CLI command.

This operation takes view name, view version, and comma-separated features list as the command line arguments and returns the respective feature dataframe.

example:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode
get_view_features --view_name t1vst2 --view_version 1 --features
adv_idx_uuid,box_type,mp
```

build_view:

This operation builds a feature view from the mentioned feature groups. It is recommended to read the detailed documentation first to understand the command and its different elements which can be found here.

To run this command, following are the steps:

1. In the config directory, find or create the views.json file or with any other name but it should be a json file and with a proper format as mentioned in the sample views.json file.
2. views.json file contains the list of feature views that can be created. Each feature views has its own key and the respective value tells the details like the name, version, description of the hopsworks feature view to be made, source feature groups and their required features, filters, and the joining criteria which will be used in creating that feature view. Make sure to use the same name of the feature view in the CLI argument which is mentioned as the key in the views.json file.
3. Once you have created or configured the json file for views, you need to tell the path of this file as the command line argument using --build_conf argument while running the application.
4. Once done, you just need to open a command prompt or terminal and write the build_view command mentioning the correct path for conf.json, and views.json files respectively. For e.g. assuming you have made the changes in the views.json and conf.json file, and the json key name in the views.json file is "t1vst2", then your command should look like:
   ```
   python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
   "/path/to/config/views.json" --run_mode build_view --view_name t1vst2
   ```
5. Sample **conf.json** and **views.json** files can be found at *config/conf.json* and *config/views.json* respectively.

example:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --build_conf
"/path/to/config/views.json" --run_mode build_view --view_name t1vst2
```

export_view_to_file:

This operation is used to export the feature view present on the hopsworks project to the .csv file. This is helpful if you want to export the data to some other project or do some analysis on external tools or excel sheet.

This operation takes view name, view version, destination directory, and file name as the command line arguments and outputs the feature view in the form of .csv file to the described destination directory. Be very sure in providing the complete desired destination path.

example:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode
export_view_to_file --view_name t1vst2 --view_version 1 --file_path
"/path/to/export/folder/" --file_name t1vst2_export.csv
```

drop_view:

This operation takes view name and view version as the command line arguments and deletes the feature view from the hopsworks project.

example:

```
python.exe cmd.py --conf "/path/to/config/conf.json" --run_mode drop_view
--view_name t1vst2 --view_version 1
```

Logging:

This application provides two types of logging:

1) Runtime logging: These logs are shown on the terminal screen or in the IDE run screen and tells you the status of each operation and whether the job executed successfully or failed.
2) Filesystem logging: The application also generates log files for each run in a log directory. The path of this directory is specified in the conf.json file when the project is set up the first time. The example log path can be found below in the ***config/conf.json*** *file*:


```
{
  "host": "localhost",
  "database": "nba",
  "username": "root",
  "password": "root",
  "hopsworks_host": "c.app.hopsworks.ai",
  "hopsworks_port": "443",
  "hopsworks_project": "basketball_feature_tool",
  "hopsworks_api_key_value": "oBo1bB2QzeD6AIzT.WXujCzTCPaHI2yKe0GnMHu7IIe0dRPKBLKaDz0tbekGlfFoLbBIDMwfGYMyQPElv",
  "log_path": "D:\\Documents\\Office Documents\\Toptal\\TSGNC\\basketball-feature-tool\\logs"
}
```

The logs are maintained for each run and tells you the complete journey of the run. It also tells whether the run was successful or failed somewhere giving the accurate error. Hence, these logs are very helpful in debugging the errors.

Code Components:

This project has following modules:

1. Cmd.py (entry point of the application)
2. Service module
3. Database module
4. Utils module
5. Config module
6. Logs module
7. Tests module
8. Files module

Cmd.py:

This module is the entry point of the application and the entire flow of the application goes from this file. It has the argparser library that identifies which run_mode of the application is called and then it calls the respective modules.

Service Module:

This module has three clases:

1. **SimpleFeatureService.py:** This class acts like a business logic layer and fulfills all the required business and logical requirements. You can write the program flow, function calls, and interaction with different modules like the database module and the feature store modules.

2. **ComplexFeatureService.py:** This class also acts as a business logic layer but this class is specifically designed to write complex logics which cannot be made into generic statements. For e.g. we have written one single function for build_group operation and all the groups can be built using that module since they have a generic implementation. THerefore, build_group function is kept in the SimpleFeatureService.py class. On the other hand, if we talk about build_view function then it is much more complex and has different logics for each view type, t1vst2 has different requirements then team or league view. Hence, there cannot be one generic implementation for all three of them. That is why, the build_view function is defined in the complex feature service class. Similarly, all such complex functions can be defined in the same class.
   Another detailed explanation on how to write new functions in the ComplexFeatureService.py can be found here.

3. **FeatureStoreService.py:** This class acts as a mediator between business logic layer and the hopsworks project. This class has API calls, and function triggers that interact with the hopsworks project according to the business logic described in the other two above-mentioned classes.

Database Module:

This module has database related operations and interacts with the mysql database using sqlalchemy. For now, only establish connection, close connection, and select statements are being written in this module. But in future, if the requirement comes in to have more DML, DDL operations for database then those could be implemented in this module.

Utils Module:

This module is a helper module. It contains classes, functions, and mini codeblocks that are used for any type of operations but are not directly linked with the business logic. An example would be a custom built multi-melt function. This function is used in the team view logic and uses a custom code to use the pandas melt function multiple times as per the requirement and since this code is not provided in the pandas library hence a separate code block is written to fulfill this requirement.

Config Module:

This module contains all the json files be it conf.json or groups.json. The purpose of this module is to keep all configurations related details or files in this folder. Though the user can maintain separate directories as well other than the project directory but it is recommended to keep every thing related to the project in the same project directory. The configs are picked up at run-time based on the path the user provides in the CLI command.

Logs Module:

As mentioned in this module, this module holds the log files that are generated on each run of the application. This path is again as per the user choice, they can either put it in the same project folder or they can set the logs path to some other directory.

Tests Module:

This module contains the UnitTest.py class that holds the code for unit testing purpose. Currently there are only four unittests that are written in this file:

1. Test DB Connection
2. Test Hopsworks Connection
3. Test Hopsworks Group Retrieval
4. Test Hopsworks View Retrieval

Further test can be added in the same file as per the requirements.

The test case configurations can also be found within the same directory with the name of ***test_conf.json***. This file is also configurable.

Files Module:

This module is used to store the files generated by the ***export_group_to_file*** and ***export_view_to_file*** and also to hold .csv files which could be used to load into the feature group. In simple words, this directory or module acts like a bucket which could hold the flat files that could be used in this application.

Limitations of Hopsworks:

Hopsworks is no doubt a very powerful feature store and is very good in feature engineering and AI modeling. However, there are certain short-comings that has limited the development, at least for this particular project. I have tried to compile all the limitations that has been found during the development of this project and are listed below:

1. **No nested joins:** In database, nested joins are very common. There are cases where we want to join table A with table B and then table B with table C (transition). But in hopsworks, nested joins are not allowed. It would only allow one main feature group A to join with every other feature groups that are required to create a view. Hence, while creating a feature group from the DB, keep this in mind that you also export the respective PKs and FKs.
2. **Primary key is compulsory in Feature group:** Feature groups cannot be created without a primary key. You have to have at least 1 primary key column in the feature group. So keep this in mind while exporting a table to create a feature group.
3. **Strict schema representation of the Feature View:** You cannot modify the structure of the feature group while creating a feature view. For example if you want to transpose the rows and columns of the feature group, or you add some more columns in the dataframe created from the feature group and then use that dataframe to create a view then the query will fail. Feature view can only be created by the existing schema or metadata of the feature group.
4. **Cannot stop the ingestion job:** When you create a feature group from a database table then it first creates the schema of the feature group and then runs an ingestion job that takes data from the database and populates the feature group. Once the ingestion starts, there is no option to cancel it on the Hopworks project UI. You cannot kill the ingestion process.
5. **Limited Documentation and Support forums:** Hopsworks has very limited documentation and the forum is also not very big. So for a newbie, it could take some time to get hands-on with the Hopsworks API and functions. My suggestion is to start with the main documentation that can be found in their page and then create the basic functions like get_group, build_group etc. and then use Debug Mode of the IDE to explore what elements does each function return and what could be done with those elements. This is a hit and trial method but this is the most efficient one.

Other Resources:

There are two other types of resources along with the codebase, which are:

Database Post Script:

This script is written to add a column of PT_Poss column in the adv_stats table. The reason for adding this column is that this column is made up by joining the box_score and adv_stats table and hence the implementation of this column on the python's end was a bit complex. Therefore, I have put the implementation on the database end using a post script so that the python implementation could be made generic. This script must be executed before running the application since it is the prerequisite of that. The script can be found at ***misc/db_post_script.sql*** in the project directory.

Batch Files:

To automate the project demo and run each function, I have written two batch scripts to demo the 12 operations this application does and calling the developed unittests. The end user can also use these scripts, they just need to modify the config paths present in the script to their desired paths and run the batch script. Below is the screenshot of the path and an example command that is being use in the ***misc/run.bat*** file:

```
@echo off
set "config_path=D:\\Documents\\Office Documents\\Toptal\\TSGNC\\basketball-feature-tool\\config\\conf.json"
set "group_build_conf=D:\\Documents\\Office Documents\\Toptal\\TSGNC\\basketball-feature-tool\\config\\groups.json"
set "view_build_conf=D:\\Documents\\Office Documents\\Toptal\\TSGNC\\basketball-feature-tool\\config\\views.json"
set "script_path=D:\\Documents\\Office Documents\\Toptal\\TSGNC\\basketball-feature-tool\\cmd.py

echo:
echo ================================================================================
echo FUNCTION 1:
echo RUNNING DROP_VIEW()
echo dropping already created view t1vst2 from hopsworks project
echo ================================================================================
python.exe "%script_path%" --conf "%config_path%" --run_mode "drop_view" --view_name "t1vst2" --view_version 1

```

The other batch file is used to test the unittests written in the ***tests/UnitTest.py*** module. The name of that file is ***misc/run_test.bat.*** Again, you need to set the config parameters in the script before calling it. The content of the file is mentioned below:

```
@echo off
set "config_path=D:\\Documents\\Office Documents\\Toptal\\TSGNC\\basketball-feature-tool\\tests\\test_conf.json"
set "script_path=D:\\Documents\\Office Documents\\Toptal\\TSGNC\\basketball-feature-tool\\tests\\UnitTest.py"

echo:
echo ================================================================================
echo UNIT TESTS
echo RUNNING UnitTests
echo ================================================================================
python -m pytest -v "%script_path%"

pause
```

Implementation of other Complex Functions in the ComplexFeatureService.py:

Logic:

The logic is totally dependent on the developer. If the view is very simple like ***t1vst2*** in which only a group of feature groups are getting joined then this is very simple. The entire function of t1vst2 can be copied. The function body can be found in ***src/service/ComplexFeatureService.py*** by the name of ***build_view_t1vst2()***. For the understanding on what each element is working inside the function and what each entity means, please read the documentation here. Using the json elements present in the views.json config file, the list of groups will be created dynamically. If there are 2 feature groups present in the views.json for this particular then 2 feature groups will be created and joined together. If there are 3 or more features groups in the views.json for this particular view metadata then the respective iteration will happen inside the loop.
Remember, the feature view at least requires one feature group to be built. So if there is only one feature group then it would be used for view creation, if there are more than the first feature group in the joining sequence will be the primary one and the rest will be joined to it based on the join_type, joining_column, and join_sequence. (**Note:** nested joins are not allowed in Hopsworks)

How to connect with main:

In the ***src/Cmd.py*** there is an **if condition** that checks if the args.run_mode is **"build_view"** in this function, the code is written to either call the instance of the ***SimpleFeatureService.py*** or the ***ComplexFeatureService.py*** modules.
If your newly implemented function is needed to be kept in the ***SimpleFeatureService.py*** then write the whole function definition in the ***SimpleFeatureService.py***. And call this function in the elif condition of the ***build_view() function*** in the same class.
Example:

SimpleFeatureService.py:

```
def view_name1(self):
    pass

def build_view(self, view_name, view_version, description, view_json):
    try:
        sfs_obj = SimpleFeatureService(self.conf)

        if (view_name == 'view_name1'):
            ##TODO: Implement logic for view 1
            pass

        elif (view_name == 'view_name2'):
            ##TODO: Implement logic for view 2
            pass

        elif (view_name == 'view_name3'):
            ##TODO: Implement logic for view 3
            pass
        else:
            logger.error('Feature view function not found. Please implement the function in '
                         'ComplexFeatureService class.')
    except Exception as e:
        logger.error(e)

    finally:
        pass
```

But if your newly implemented function is needed to be kept in the ***ComplexFeatureService.py*** then write the whole function in the ***ComplexFeatureService.py*** And call this function in the elif condition of the ***build_view() function*** in the same class. And lastly, in the ***src/Cmd.py*** in the same **"args.run_mode==build_view"** if condition, there is a list of complex views names, insert the name of the complex view in that list. Find the example in the screenshot below:

ComplexFeatureService.py:

```
def build_view_league(self, main_feature_group, secondary_feature_groups, view_name, view_version, labels,
                      description):...

def build_view(self, view_name, view_version, description, view_json):
    try:
        cfs_obj = ComplexFeatureService(self.conf)
        if (view_name == 'team'):
            ##TODO: Implement logic for team view
            pass
            # query = cfs_obj.build_view_team()
            # fss_obj = fss.FeatureStoreService(self.conf)
            # fss_obj.build_view(view_name, view_version, description, "", query)

        elif (view_name == 'league'):
            ##TODO: Implement logic for league view
            pass
            # query = cfs_obj.build_view_league(view_name, view_version, description, view_json)
            # fss_obj = fss.FeatureStoreService(self.conf)
            # fss_obj.build_view(view_name, view_version, description, "", query)

        elif (view_name == 't1vst2'):
            query = cfs_obj.build_view_t1vst2(view_name, view_version, description, view_json)
            fss_obj = fss.FeatureStoreService(self.conf)
            fss_obj.build_view(view_name, view_version, description, "", query)
        else:
            logger.error('Feature view function not found. Please implement the function in '
                         'ComplexFeatureService class.')
```

Cmd.py

```
elif args.run_mode == 'build_view':
    with open(args.build_conf, "r") as jsonfile:
        views_info = json.load(jsonfile)
        if args.view_name not in views_info:
            logger.error('Feature group key is not found in the groups json file')

        for view_name in views_info:
            if view_name == args.view_name:
                if (args.view_name in ['t1vst2', 'team', 'league']):
                    view = views_info[view_name]
                    a = view['hopsworks_view_name']
                    cfs.build_view(view['hopsworks_view_name'],
                                   view['hopsworks_view_version'],
                                   view['description'],
                                   view['features'])
                else:
                    view = views_info[view_name]
                    a = view['hopsworks_view_name']
                    sfs.build_view(view['hopsworks_view_name'],
                                   view['hopsworks_view_version'],
                                   view['description'],
                                   view['features'])
```