Conservation analysis on PDB fragments binding mode

The following scripts underline the main analysis on the evaluation of binding mode conservation of PDB fragments. In a first step, the subset files are generated considering the initial filtering thresholds and the fragments subsets at the three possible conservation levels. Then a parallel pairwise comparison for the interactions fingerprints is carried out, to finalize with a global conservation score calculation.

Example code used in test case

Main script to generate the conservation subsets according to the initial filters and the different levels of conservation: fragments in all ligands binding all targets, fragments binding the same target, and fragments binding the same ligand. Independent files are generated after each step for further analysis and easy track back of data generation.

```
### Parameters to filter the dataset ###
MWmin = 40 #Defines the Threshold for fragments size. (less than 40 too small)
MWmax = 300 #According to definition of fragments
ligs = 4 #Number of min different ligands the fragment is found. (according to scatter plot and mean)
unireps = 9 #Number os min different targets the fragments is found. (according to scatter plot and mean)

def write_subset(dataset, subset_path, MWmin, MWmax, ligs, unireps):
    """Read fragments data and generates a subset according to the filter"""
    #parameters: fragmentation report, output folder, min molsize, max molsize, Nr ligs, Nr targets
    #Reads the fragmentation report
    report = pd.read_csv(dataset, sep = '\t', index_col=None, dtype=None)
    #Filters the data according to type of data and cutoff of MW
    reportf1 = report[(report['TYPE']=='FR') & (report['MW']>=MWmin) & (report['MW']<=MWmax)]
    #Group the data by fragments and gets count of unique ligands and unique proteins
    groups = reportf1.groupby('FRAGINCHI')['LIGINCHI','UNIREP'].nunique(dropna=False).reset_index()
    #Gets the subset of fragments according to cutoff of ligands and proteins
    fragments = groups[(groups['LIGINCHI']>ligs) & (groups['UNIREP']>unireps)]
    #Subsets the original report by fraginchis in fragments subset
    fil_subset = reportf1[reportf1['FRAGINCHI'].isin(fragments['FRAGINCHI'])]
    #writes the subset into a file
    fil_subset.to_csv(subset_path, sep='\t', index=False)


def generate_frag_files(subset_path, frags_subsets_path):
    """Reads the filtered subset and for each fragment gets a new subset"""
    #parameters: the path to the filtered subset and the path to folder that will contain the independent fragments
    subsets
    fil_subset = pd.read_csv(subset_path, sep = '\t', index_col=None, dtype = str)
    #Gets a list of unique fragments among the filtered subset
    fragments = fil_subset["FRAGINCHI"].unique().tolist()
    for frag in fragments:
        #for a given fragment inchikey gets the full subset
        frag_subset = fil_subset[fil_subset["FRAGINCHI"] == frag]
        #writes the fragments subset into an independent file
        frag_subset.to_csv(frags_subsets_path+frag+".csv", sep='\t', index=False)

def generate_same_tar_files(subset_path, frag-tar_subsets_path):
    """Reads the filtered subset and for each fragment and target combinations generates a new subset"""
    #parameters: the path to the filtered subset and the path to folder that will contain the independent fragments-
    targets subsets
    fil_subset = pd.read_csv(subset_path, sep = '\t', index_col=None, dtype = str)
    setcode = []
    for index, row in fil_subset.iterrows():
        #for each fragment-target combination generates a code
        frag = str(row["FRAGINCHI"])
        tar = str(row["UNIREP"])
        #appends the code to a list
        setcode.append('~'.join((frag,tar)))
    #Adds the combination code list as new column into the subset
    fil_subset["SETCODE"] = setcode
    #obtains the unique codes as a list
    setcodes = fil_subset["SETCODE"].unique().tolist()
```

```
    for setcode in setcodes:
        #for each combination code, extract the subset of complexes
        set_subset = fil_subset[fil_subset["SETCODE"] == setcode]
        #write the combination code subset into an independent file
        set_subset.to_csv(frag-tar_subsets_path+setcode+".csv", sep='\t', index=False)

def generate_same_lig_files(subset_path, frag-lig_subsets_path):
    """Reads the filtered subset and for each fragment and ligand combinations generates a new subset"""
    #parameters: the path to the filtered subset and the path to folder that will contain the independent fragments-ligands subsets
    fil_subset = pd.read_csv(subset_path, sep = '\t', index_col=None, dtype = str)
    setcode = []
    for index, row in fil_subset.iterrows():
        #for each fragment-ligand combination generates a code
        frag = str(row["FRAGINCHI"])
        lig = str(row["LIGINCHI"])
        #appends the code to a list
        setcode.append('~'.join((frag,lig)))
    #Adds the combination code list as new column into the subset
    fil_subset["SETCODE"] = setcode
    #obtains the unique codes as a list
    setcodes = fil_subset["SETCODE"].unique().tolist()
    for setcode in setcodes:
        #for each combination code, extract the subset of complexes
        set_subset = fil_subset[fil_subset["SETCODE"] == setcode]
        #write the combination code subset into an independent file
        set_subset.to_csv(frag-lig_subsets_path+setcode+".csv", sep='\t', index=False)
```

The class Combis takes care of the pairwise comparisons. Collects the fingerprints by UIDs and writes down all possible pairs of them. In order to achieve a fair comparison of binding modes, the number of UIDs by fragment subset was limited to 500. The multiple UIDs having the same ligand and target were reduced to one representative and only 500 samples were chosen.

```
class Combis():
    """Defines a Combis object containing the pairwise comparisons of a fragment subset"""
    def __init__(self, path_combis, path_FPs, frag_subset, frag):
        #parameters: output path for combis, output path for the fingerprints, the input fragment subset, the fragment inchikey
        self.path_combis = path_combis
        self.path_FPs = path_FPs
        self.subset =  pd.read_csv(frag_subset, sep = '\t', index_col=None, dtype = str)
        self.frag = frag
        self.frag_uids = defaultdict(list)
        self.uid_combis = defaultdict(list)

    def write_uid_FP_csv(self):
        """Read subset dataframe and writes csv file with the fingerprint data"""
        subset_fps = self.subset[['UID','FPSIMPLE']]
        subset_fps.to_csv(self.fol_FPs+self.frag+"_fps.csv", sep='\t', index=False) #Only FPsimple data

    def get_rep_complex(self):
        """For a given fragment, subset the dataset and group by uniprot-liginchi combinations.
       Then selects only one uid as representative complex of the group and return the full list of uids"""
        groups = self.subset.groupby(['UNIREP','LIGINCHI'])['UID'].apply(list).to_frame() #group by uniprot-liginchi and get list of uids
        uids = [uid_list[0] for uid_list in groups['UID']] #Select the first complex in group and add to list
        if len(uids) > 500:
            uids = uids[:500]
        return uids

    def get_uids_for_frag(self):
        """Creates a dictionary with FRAGINCHI as key and as value a list of all UIDs with such fragment.
       If the fragment has > 500 uids, the set of UIDs will be reduced"""
        self.frag_uids = self.subset["UID"].tolist()
        if len(self.frag_uids) > 500:
```

```
            uids = self.get_rep_complex()
            self.frag_uids = uids

    def write_uid_combis(self):
        """Writes a file containing the list of all uid combinations for each fragment and another file containing each uid combination"""
        uids = self.frag_uids
        if len(uids) > 1:
            rep = open(self.fol_combis+self.frag+".csv", "w")
            rep.write('UID1\tUID2\tFRAGINCHI\n')
            for combi in itertools.combinations(uids,2):
                rep.write('{0}\t{1}\t{2}\n'.format(combi[0], combi[1], self.frag))
            rep.close()
```

The class FPSimCalculator defines the calculation of the (cosine or tanimoto) similarities between pairwise fingerprints. For each UID pair, it calculates the fingerprints similarity and writes it down to a file.

```
class FPSimCalculator():
    def __init__(self, uid_pairs, fpsimple):
        self.uid_pairs = pd.read_csv(uid_pairs, sep = '\t', index_col=None, dtype = str)
        self.FPsimple_dic = self.get_data_dict(fpsimple, 'UID', 'FPSIMPLE')

    def get_data_dict(self, datafile, key1, key2):
        """Reads a file as dataframe and constructs a dictionary with column (key1) as dict key and column (key2) as value"""
        data_dic = {}
        data = pd.read_csv(datafile, sep = '\t', index_col=None, dtype = str)
        for index, row in data.iterrows():
            data_dic[row[key1]] = row[key2]
        return data_dic

    def get_cosine_sim(self,va,vb):
        """Takes two fingerprint vectors va and vb and calculates the cosine similarity."""
        va = list(map(int, va))
        vb = list(map(int, vb))
        csim = 1.0-distance.cosine(va, vb)
        if np.isnan(csim):
            csim = -1.0
        return csim

    def get_tanimoto_sim(self, va, vb):
        """Takes two fingerprint vectors va and vb and calculates the cosine tanimoto coefficient metric"""
        #common = [a for a, b in zip(va, vb) if a == b]
        #return float((len(common))/(len(va) + len(vb) - len(common)))
        common = [a for a, b in zip(va, vb) if a == b and a=='1']
        binsva = [a for a in va if a=='1']
        binsvb = [b for b in vb if b=='1']
        nom = len(common)
        denom = len(binsva)+len(binsvb)-len(common)
        if denom != 0:
            return float(nom/denom)
        else:
            return float(1.0)

    def get_fp_sim(self,fp_dic, id1,id2):
        """Checks if the tuple of ids is different, then calculates the FP similarity. If not, then sim is automatically 1"""
        if id1 != id2:
            id1_fp = fp_dic[id1]
            id2_fp = fp_dic[id2]
            fp_sim = self.get_tanimoto_sim(list(id1_fp), list(id2_fp))
        else:
            fp_sim = 1.0
        return fp_sim

    def write_fingerprints_sim(self,output):
        """Iterates over the ID pairs, calls the function to calculates similarity and write a new report"""
```

```
        report = open(output, 'w')
        report.write('UID1\tUID2\tFPSIMPLE_SIM\tGROUP\n')
        for index, row in self.uid_pairs.iterrows():
            id1 = row['UID1']
            id2 = row['UID2']
            group = row['FRAGINCHI'] #In Desaphy analysis is GROUP
            fpsimple_sim = self.get_fp_sim(self.FPsimple_dic, id1,id2)
            report.write('{0}\t{1}\t{2:.2f}\t{3}\n'.format(id1,id2, fpsimple_sim, group))
        report.close()
```

The class DataAnalyzer provides the final report with the global binding mode conservation (similarities) for each fragments subset or group.

```
class DataAnalyzer():
    def __init__(self, sim_data_path, frag):
        self.sim_data = pd.read_csv(sim_data_path, sep = '\t', index_col=None, dtype = str)
        self.frag = frag
        self.sim_fps = self.sim_data['FPSIMPLE_SIM'].tolist()

    def write_fp_sim(self, output):
        """Writes the MIN/MAX values of fingerprints into a report file"""
        report = open(output, 'w')
        report.write("FRAGMENT\tFPSMIN\tFPSMAX\tFPSAVG\n")
        minfps = min(self.sim_fps)
        maxfps = max(self.sim_fps)
        fps =  list(map(float, self.sim_fps))
        avgfps = sum(fps)/len(self.sim_fps)
        report.write('{0}\t{1}\t{2}\t{3:.2f}\n'.format(self.frag, minfps, maxfps, avgfps))
        report.close()
```