D3.1

Computational pipeline to extract prior network information at the proteomic level

Project number:	668858
Project acronym:	PrECISE
Project title:	PrECISE: Personalized Engine for Cancer Integrative Study and Evaluation
Start date of the project:	1st January, 2016
Duration:	36 months
Programme:	H2020-PHC-02-2015

Deliverable type:	Other
Deliverable reference number:	PHC-668858 / D3.1/ V2.0
Work package contributing to the deliverable:	WP 3
Due date:	AUG 2016 – M08
Actual submission date:	10th November 2017

Responsible organisation:	UKAACHEN
Editor:	Luis Tobalina
Dissemination level:	PU
Revision:	V2.0

Abstract:	This deliverable provides an overview of a computational tool to extract and aggregate prior information from different databases. It focuses on how Omnipath and its accompanying Python module Pypath can be used to extract prior knowledge information regarding protein interactions.
Keywords:	Omnipath, Pypath, signalling network databases, protein interaction networks

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 668858.

This work was supported (in part) by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0324-2. The opinions expressed and arguments employed therein do not necessarily reflect the official views of the Swiss Government.

Editor

Luis Tobalina (UKAACHEN)

Contributors

Julio Saez-Rodriguez (UKAACHEN)

María Rodríguez Martínez (IBM)

Laurence Calzone (CI)

Executive Summary

Literature curated information on protein interactions is an extremely valuable resource for researchers studying different biological questions. These resources can help in the generation of hypothesis to explain experimental data and they provide support for the building of computational models. However, the existing curated information is currently distributed throughout a large number of online resources, making gathering and retrieving this information in a consistent manner a non-trivial endeavour.

This document gives an overview of the Omnipath database and its accompanying Python module Pypath. Omnipath gathers 55 resources, including 27 high-confidence literature curated signaling resources, providing and easy, unified and convenient entry point to much of the protein interaction knowledge available.

Overall, Omnipath and Pypath greatly facilitate the integration and extraction of biological prior knowledge for analysis and model building, and they can be incorporated into wider data processing pipelines. Pypath and Omnipath are available at http://omnipathdb.org.

As requested during the revision of the deliverable, we have revised the document attending to the resources mentioned in Task 3.1. The introduction has been updated and a new table listing the resources accessible using Omnipath has been included.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Knowledge about protein interactions, backed up by experimental evidence, is a precious resource in biological research. Researchers can come up with new plausible hypothesis that can explain observed biological phenomena starting from this type of knowledge. In addition, they can build sensible computational models grounded on evidence based interaction networks, which can later give rise to new testable predictions. It also constitutes a benchmark against which interaction prediction algorithms can be tested. This type of data can be found currently in different online resources, however, gathering and retrieving information from all of them in a consistent manner is anything but trivial.

With the aim to facilitate the access to this information, UKAACHEN started the development of the Omnipath database (Türei et al. (2016)). During PrECISE, development in this resource continued, polishing the integration of the different resources (i.e. detecting and solving errors and adding new functionalities) and developing documentation showing some of its possibilities. UKAACHEN introduced this resource to other partners and publicized it in conferences. Interaction with other partners, like TUDA, IBM and CI, informed on some of the desired functionalities and helped to test the software.

The Omnipath database gathers 55 different resources, including 27 high-confidence literature curated signaling resources. In addition, it also provides access to other resources containing annotation and other kinds of information. These include post-translational modifications (PTMs), protein complexes, expression data, drug-target relationships or Gene Ontology annotations. Among these resources we find STRING, Human Protein Reference Database (HPRD), PhosphoSitePlus and ACSN. A list of resources is given in Table 1.

Furthermore, Omnipath comes with an accompanying Python module called Pypath. This software enhances the Omnipath database with several expansion and analysis functionalities. New interactions can be loaded and merged with the existing ones and multiple graph analysis methods are available right away. These analysis methods range from different network statistics calculations to shortest path searches. All of them facilitate the search and extraction of subnetworks of interest by filtering nodes and edges according to different criteria, speeding up this way the process of building and curating signaling pathway models.

Although Omnipath and Pypath by themselves do not constitute a completely automated way of getting a final biologically meaningful network starting from a list of proteins, they provide the basis for a computational pipeline capable of doing so. In fact, there might be several different ways of achieving that goal, and Omnipath and Pypath can give support to all of them. Moreover, they also fit into a more traditional manual curation pipeline, easing the process of information search and integration.

This document focuses on different practical use cases of Omnipath and Pypath. By using prostate cancer as an example, we show how Omnipath and Pypath can help to answer common questions around protein interactions and signaling networks. Overall, we establish a computational pipeline for the extraction of prior knowledge information around protein interactions.

Database	Category	Subcategory
ACSN	Literature curated	Reaction
AlzPathway	Literature curated	Pathway
ARN	Literature curated	Pathway
Ataxia	High-throughput	Interaction
Awan 2007	Literature curated	Pathway
BioCarta	Literature curated	Pathway
BioGRID	High-throughput	Interaction
Ma'ayan 2005	Literature curated	Pathway
CancerCellMap	Literature curated	Interaction
CARFMAP	Literature curated	Pathway
ConsensusPathDB	Literature curated	Pathway
CORUM	Literature curated	Complexes
CST Pathways	Literature curated	Pathway
Cui 2007	Literature curated	Pathway
dbPTM	Literature curated	Ptm
DeathDomain	Literature curated	Pathway
DEPOD	Literature curated	Post-translational modification
DIP	Literature curated	Interaction
DOMINO	Literature curated	Ptm
ELM	Literature curated	Post-translational modification
Guide to Pharmacology	Literature curated	Pathway
HPRD	Literature curated	Post-translational modification
HumanSignalingNetwork	Literature curated	Pathway

Database	Category	Subcategory
HuPho	High throughput and literature curated	Post-translational modification
InnateDB	Literature curated	Interaction
IntAct	Literature curated and high-throughput	Interaction
KEGG	Literature curated	Reaction network
Laudanna	Combined	Mixed
Li 2012	High-throughput	Yeast 2 hybrid
Lit-BM-13	High-throughput	Yeast 2 hybrid
LMPID	Literature curated	Post-translational modification
Macrophage	Literature curated	Pathway
MatrixDB	Literature curated	Interaction
MINT	Literature curated and high-throughput	Interaction
MPPI	Literature curated	Interaction
NCI-PID	Literature curated	Reaction network
Negatome	Literature curated	Negative
NetPath	Literature curated	Reaction network
NRF2ome	Literature curated	Pathway
PANTHER	Literature curated	Reaction network
PathwayCommons	Combined	Interaction
PDZBase	Literature curated	Pathway
phospho.ELM	Literature curated	Ptm
PhosphoPoint	Literature curated and prediction	Post-translational modification
PhosphoSite	Literature curated and high-throughput	Post-translational modification
Reactome	Literature curated	Reaction network
SignaLink	Literature curated	Pathway

Database	Category	Subcategory
Signor	Literature curated	Pathway
SPIKE	Literature curated	Pathway
STRING	High-throughput and prediction	Interaction
TLR	Literature curated	Model
TRIP	Literature curated	Pathway
Vidal HI-III	High-throughput	Yeast 2 hybrid
WikiPathways	Literature curated	Reaction network
Zaman 2013	Literature curated	Pathway

Table 1. Databases integrated in Omnipath. For a description of each resource, see http://omnipathdb.org/info.

Chapter 2 Preliminaries

Omnipath and Pypath are available at http://omnipathdb.org. The Omnipath database can be accessed directly from the website through a REST style API, but it is also supported by the Python package Pypath, which provides useful features for expansion and analysis of the networks. This document will make extensive use of the Pypath Python module.

Omnipath can be installed with *pip* (a Python module that eases the process of installing other packages) directly from the GitHub repository with the following command:

```
pip install git+git://github.com/saezlab/pypath.git
```

On some systems, however, it may be challenging to get it working due to the somewhat complicated installation of other programs on which the Pypath module depends (for example, the Cairo library on OS X systems). Omnipath's website provides instructions on how to install the software on different systems. However, unexpected issues may arise. In those cases, users are encouraged to contact *omnipath@googlegroups.com*.

Omnipath and Pypath will be updated regularly. Previous releases of the package can be found at *http://pypath.omnipathdb.org/releases/archive/*. Although in the future some functions and features (or their syntax) may change, the general concepts and ideas introduced in this document will remain valid. This document was developed with Pypath version 0.3.12. To install a previous version of the software, download the *tar.gz* file from the link mentioned above (e.g. pypath-0.3.12.tar.gz) and run the following command line from the folder containing the file:

```
pip install pypath-0.3.12.tar.gz
```

Chapter 3 Omnipath overview

The starting point for retrieving prior knowledge interaction information is usually a list of proteins we are interested in. In the frame of the PrECISE project, there are 5 genes we are specially interested in given that they are frequently altered in prostate cancer:

- PTEN
- FOXA1
- TP53
- SPOP
- AR

We are going to use these 5 genes to show how can we retrieve information around them using Omnipath and Pypath.

3.1 ID conversion

Pypath comes with a mapping module that allows translating gene symbols to Uniprot identifiers, among others. For example, to get the Uniprot IDs linked to our 5 genes of interest we could use the following code:

```
from pypath import mapping

m = mapping.Mapper()

query_genes = set(['PTEN', 'FOXA1', 'TP53', 'SPOP', 'AR'])

for i_gene in query_genes:
    prot_id = m.map_name(i_gene, 'genesymbol', 'uniprot')[0]
    print("{}: {}".format(i_gene, prot_id))
```

The output that we obtain is as follows:

```
FOXA1: P55317
PTEN: P60484
SPOP: O43791
AR: P10275
TP53: P04637
```

The `map_name()` function returns a list just in case the identifier provided matches to more than one queried identifier type. In most cases, the matching is one to one, like in this example, but some genes may map to more than one protein or some protein may be coded by more than one gene. Being Omnipath a protein interaction network, the main ID is the Uniprot ID. When compiling the network, if we have information about interactions based on gene identifiers and some of them map to more than one protein, interactions will be assigned to all of them.

3.2 Initializing the network

The Omnipath database can be loaded in Python using the Pypath module.

```
# load packages
```

```
import pypath

pa = pypath.main.PyPath()

pa.init_network()
```

By default, 27 curated resources will be loaded, but we can select specific resources to be loaded or incorporate our own interaction lists.

We can execute the following command to remove interactions that are only supported by papers reporting many interactions (which may be less curated than papers reporting just a low number of interactions).

```
# remove links reported in papers with more than 50 interactions (by
default)

pa.remove_htp()
```

The directed network can be constructed with the `get_directed()` function. This function takes advantage of some references explicitly supporting one specific direction. For those interactions without explicit directionality support, the interaction may be dropped (default behavior), assigned an arbitrary direction or added with a pair of opposite directed edges.

```
pa.get_directed()
```

3.3 Node neighbourhood exploration

One of the first things we might want to know is what are the other proteins our proteins of interest interact with.

```
query_nodes = set(['PTEN', 'FOXA1', 'TP53', 'SPOP', 'AR'])

for igene in query_nodes:
    # to query a node based on the value of an attribute we can use
the igraph find() method
    #prot = pa.graph.vs.find(label=i)['name']
    # if the attribute is the vertex label (genesymbol) we can use
pypath's genesymbol() function
    prot = pa.genesymbol(igene)['name']
    #neighbours_of_prot = pa.first_neighbours(prot)
    neighbours_of_prot = list(pa.gs_neighbors(igene).gs())
    print('{} ({}) has {} neighbours:'.format(igene, prot,
len(neighbours_of_prot)))
    if len(neighbours_of_prot)<10:
        print(neighbours_of_prot)
    else:
        print('(showing only 10 proteins)')
        print(neighbours_of_prot[0:10])
    print('---')
```

```
FOXA1 (P55317) has 4 neighbours:
['AR', 'TLE1', 'NFIX', 'NFIB']
---
PTEN (P60484) has 50 neighbours:
(showing only 10 proteins)
```

```
['AKT1', 'RELA', 'CDC42', 'EGR1', 'PPP1CA', 'CREB1', 'RAC1', 'ROCK1',
'CTNNB1', 'AR']
---
SPOP (O43791) has 4 neighbours:
['TRAF6', 'H2AFY', 'DAXX', 'CUL3']
---
AR (P10275) has 245 neighbours:
(showing only 10 proteins)
['AKT1', 'KDM3A', 'GTF2H3', 'GTF2H2', 'B3KNJ3', 'BAG1', 'AHR',
'CDK5', 'SVIL', 'IARS']
---
TP53 (P04637) has 271 neighbours:
(showing only 10 proteins)
['HDAC2', 'MAML1', 'CDK5', 'KDM1A', 'XPO1', 'BAD', 'DDX5', 'SIRT1',
'CCNA2', 'RELA']
---
```

Because these nodes are not necessarily connected between each other, we can find a way to connect them supported by literature evidence in order to get a unique network. One of the possibilities is to use the shortest paths between our genes of interest. For example, we may try to connect SPOP to FOXA1, two of the nodes with the least number of neighbours:

```
# find shortest path between SPOP and FOXA1
path = pa.graph.get_shortest_paths(pa.genesymbol('SPOP')['name'],
to=pa.genesymbol('FOXA1')['name'])
# the result is returned as a list with a single element
path = path[0]

path_SPOP_to_FOXA1_length = len(path)-1
print('The path from SPOP to FOXA1 has {}
steps:'.format(path_SPOP_to_FOXA1_length))
print('\t' + ' --> '.join(pa.graph.vs[i]['label'] for i in path))
```

```
The path from SPOP to FOXA1 has 4 steps:
      SPOP --> TRAF6 --> AKT1 --> AR --> FOXA1
```

Of course, the shortest path that connects two nodes might not be unique.

```
# find all paths between SPOP and FOXA1 (of length equal to the
shortest path length)

# to find the index based on the value of an attribute we can use
igraph's select() function
#node_start = pa.graph.vs.select(label='SPOP').indices[0]
#node_end = pa.graph.vs.select(label='FOXA1').indices[0]
# or, if the attribute is the gene symbol, we can use pypath's
genesymbol() function
node_start = pa.genesymbol('SPOP').index
node_end = pa.genesymbol('FOXA1').index

paths = pa.find_all_paths(start=node_start, end=node_end,
maxlen=path_SPOP_to_FOXA1_length)
print('Number of paths: {}'.format(len(paths)))
```

```
      :: Looking up all paths up to length 4: finished, 100.0%
Number of paths: 20


p = [item for sublist in paths for item in sublist]
p = set(p)
print('Number of nodes: {}'.format(len(p)))


Number of nodes: 25
```

In this case, we have 20 paths of length 4 that can connect SPOP and FOXA1. These paths involve 25 nodes. We could decide to prioritize these paths based on the references supporting each link involved in them, or by including experimental information (e.g. proteomics or transcriptomics data) to hypothesize what paths are more likely than others, for example.

We can extract and plot the network spanned by these nodes

```
import igraph  # import igraph to use the plot function
# Load the ipython display and image module
from IPython.display import Image
from IPython.display import display


# extract graph expanded by the nodes included in the path
connection_graph = pa.graph.induced_subgraph(p)

plot2 = igraph.plot(connection_graph,
layout=connection_graph.layout_auto(), **visual_style)
plot2.save('connection_SPOP_FOXA1.png')
display(Image('connection_SPOP_FOXA1.png'))
```

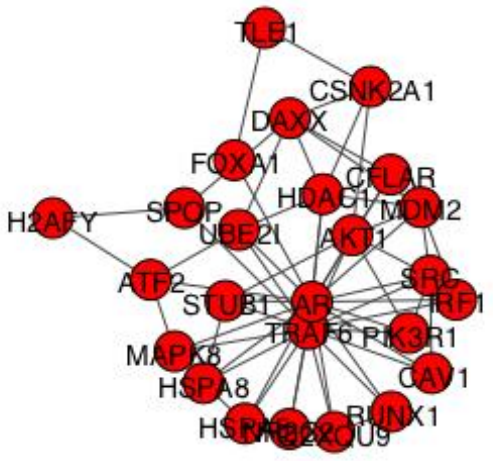

Figure 1. Network composed by all the nodes that participate in all the paths of length 4 between SPOP and FOXA1.

In summary, we can get a prior knowledge network that connects our nodes of interest following this strategy and use other methods to refine it.

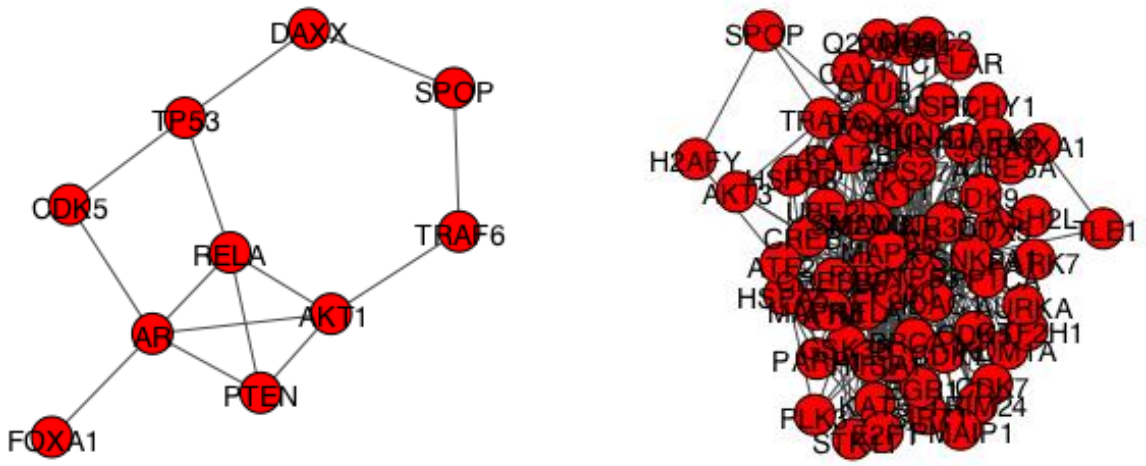

Figure 2. Network obtained by connecting all our genes of interest with one shortest path between pairs of nodes (left) or with all possible shortest paths (right).

The accompanying document **node_neighbourhood.html** is a rendered IPython Notebook that contains code and output corresponding to this analysis with some additional observations.

3.4 Pathway extraction

Omnipath includes pathway annotations from different databases (Kegg, Signalink, Signor, Netpath). We can have a look into what pathways are our genes of interest annotated to and see if there is one connecting all of them.

```
pa.load_all_pathways()
```

```
for igene in query_nodes:
    print(igene)
    print(pa.gs(igene)['kegg_pathways'])
    print(pa.gs(igene)['signalink_pathways'])
    print(pa.gs(igene)['signor_pathways'])
    print(pa.gs(igene)['netpath_pathways'])
    print('---')
```

```
FOXA1
set([])
set([])
set([])
set([u'Androgen receptor (AR)'])
---
PTEN
set([u'Phosphatidylinositol signaling system', u'Sphingolipid
signaling pathway', u'Focal adhesion', u'Tight junction', u'Hepatitis
B', u'FoxO signaling pathway', u'p53 signaling pathway'])
set(['HH', 'TNF/Apoptosis', 'WNT', 'RTK', 'IIP'])
set([u'MTOR Signaling', u'Insulin Receptor'])
set([u'Androgen receptor (AR)'])
---
SPOP
```

```
set([])
set(['HH', 'JAK/STAT'])
set([])
set([])
---
AR
set([u'ErbB signaling pathway', u'Pathways in cancer', u'Hippo
signaling pathway'])
set(['IIP', 'autophagy'])
set([])
set([u'Androgen receptor (AR)', u'Interleukin-6 (IL-6)', u'
Transforming growth factor beta (TGF-beta) receptor ', u'Alpha6 Beta4
Integrin'])
---
TP53
set([u'HTLV-I infection', u'Melanoma', u'Bladder cancer',
u'Sphingolipid signaling pathway', u'Chronic myeloid leukemia',
u'Proteoglycans in cancer', u'Apoptosis', u'Neurotrophin signaling
pathway', u'Cell cycle', u'p53 signaling pathway', u'Viral
carcinogenesis', u'PI3K-Akt signaling pathway', u'Pathways in
cancer', u'Hepatitis B', u'Hepatitis C', u'Longevity regulating
pathway - mammal', u'Wnt signaling pathway', u'Prostate cancer',
u'Thyroid hormone signaling pathway', u'Glioma', u'MAPK signaling
pathway', u'Epstein-Barr virus infection', u'Measles'])
set(['Notch', 'TNF/Apoptosis', 'IIP', 'RTK', 'autophagy'])
set([u'Mitochondrial Control of Apoptosis', u'P38 Signaling',
u'AcuteMyeloidLeukemia'])
set([u' Transforming growth factor beta (TGF-beta) receptor '])

---
```

Taking a look at the list of pathways each node participates in, we can find a set that connect all of them. For example, 'Androgen receptor (AR)' of 'netpath', 'HH' of 'signalink' and 'Transforming growth factor beta (TGF-beta) receptor' of 'netpath'. We can use this information to extract another prior knowledge interaction network based on these pathways.

```
connector_pathways = {'signalink_pathways': set(['HH']),
                      'netpath_pathways': set([u'Androgen receptor
(AR)',
                                               u' Transforming growth
factor beta (TGF-beta) receptor '])}

filter_func = []
filter_func.append(lambda vertex: 'HH' in
vertex['signalink_pathways'])
filter_func.append(lambda vertex: u'Androgen receptor (AR)' in
vertex['netpath_pathways'])
filter_func.append(lambda vertex: u' Transforming growth factor beta
(TGF-beta) receptor ' in vertex['netpath_pathways'])

connector_node_list = pa.graph.vs.select(lambda vertex: any(i(vertex)
for i in filter_func))

connector_subgraph = pa.graph.induced_subgraph(connector_node_list)
print('Number of nodes: {}'.format(connector_subgraph.vcount()))
print('Number of edges: {}'.format(connector_subgraph.ecount()))
```

```
Number of nodes: 358
Number of edges: 1562
```

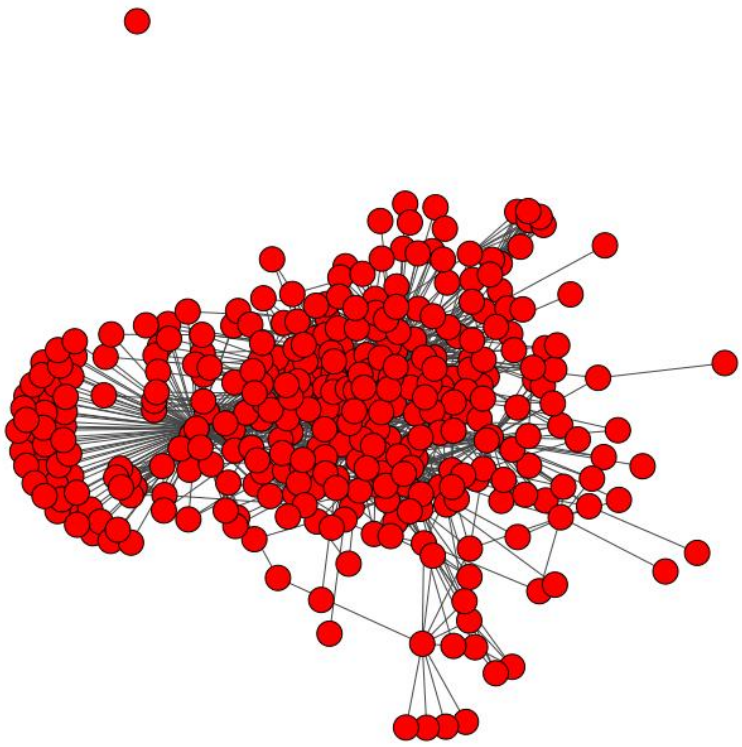

Figure 3. Network obtained by extracting all the interactions between the nodes annotated in the 'Androgen receptor (AR)' of 'netpath', 'HH' of 'signalink' and 'Transforming growth factor beta (TGF-beta) receptor' of 'netpath'.

The accompanying document **Pathway_extraction.html** is a rendered IPython Notebook that contains code and output corresponding to this section.

3.5 Protein complex extraction

Omnipath also provides access to databases reporting protein complexes, such as CORUM (http://mips.helmholtz-muenchen.de/genre/proj/corum). Again, if we want to know if our genes of interest are annotated to participate in some protein complex, we just need to load the database and make some queries.

```
# load CORUM database (http://mips.helmholtz-
muenchen.de/genre/proj/corum)

pa.load_corum()

for i_node in query_nodes:
    i_complex_list = pa.gs(i_node)['complexes']['corum'].keys()
    if len(i_complex_list)>0:
        print("Gene {} is reported in the following complexes in the
CORUM database:".format(i_node))
        for i in i_complex_list:
            print("\t{}".format(i))
```

```
    else:
        print("Gene {} is not reported to belong to any protein
complex in the CORUM database".format(i_node))
    print("---")
```

```
Gene FOXA1 is not reported to belong to any protein complex in the
CORUM database
---
Gene PTEN is not reported to belong to any protein complex in the
CORUM database
---
Gene SPOP is reported in the following complexes in the CORUM
database:
      Ubiquitin E3 ligase
---
Gene AR is reported in the following complexes in the CORUM database:
      AR-AKT-APPL complex
      AOF2-AR complex
---
Gene TP53 is reported in the following complexes in the CORUM
database:
      CNS-P53 complex
      p300-MDM2-p53 protein complex
      FHL2-p53-HIPK2 complex
      Axin-p53-HIPK2 complex
      P53-BARD1-Ku70 complex
      DAXX-Axin-p53-HIPK2 complex
      hSIR2-p53 complex
      FOXO3-TP53 complex, oxidative stress stimulated
      YY1-MDM2-p53 complex
      Daxx-Axin-p53 complex
      p53-BCL2 complex
      p53-SP1 complex
      NUMB-TP53-MDM2 complex
      MSH2/6-BLM-p53-RAD51 complex
      p53 homotetramer complex
      Er-alpha-p53-hdm2 complex
---
```

We can check the details of these complexes, for instance, the *Ubiquitin E3 ligase* complex in which *SPOP* participates.

```
prot_complex_details = pa.gs("SPOP")['complexes']['corum']['Ubiquitin
E3 ligase']
prot_complex_details
```

```
{'all_members': [u'Q13618', u'Q9UER7', u'O43791'],
 'all_members_original': [u'Q13618', u'Q9UER7', u'O43791'],
 'diseases': u'',
 'full_name': u'Ubiquitin E3 ligase (SPOP, DAXX, CUL3)',
 'functions': u'Ubiquitin E3 ligases covalently attach ubiquitin to a
lysine residue on a target protein. Polyubiquitination marks proteins
```

```
for degradation by the proteasome. SPOP serves as an adaptor of Daxx
for the ubiquitination by Cul3-based ubiquitin ligase and subsequent
degradation by the proteasome. Experiments suggest that SPOP/Cul3-
ubiquitin ligase plays an essential role in the control of Daxx level
and, thus, in the regulation of Daxx-mediated cellular processes,
including transcriptional regulation and apoptosis.',
 'references': [u'16524876']}
```

The *references* field contains the *pmid* of the article supporting this complex. We can query some details of it with the following code:

```
ref_pmid = prot_complex_details['references'][0]
pypath.main.Reference(ref_pmid).info()
# we can also open the webpage with the article abstract in a
separate window with the following command
# pypath.main.Reference(ref_pmid).open()
```

The accompanying document **protein_complex_extraction.html** is a rendered IPython Notebook that contains code and output corresponding to this section.

3.6 Transcription Factors and surface receptors

Apart from pathway annotations, Omnipath also supports Gene Ontology annotations. We can use them for several purposes, for example, locating transcription factors and surface receptors.

```
# load go annotations:
pa.load_go()

# get the GO annotation:
pa.go_dict()
```

```
Some GO terms that may be useful:
(C) transcription factor complex
(C) transcriptional repressor complex
(P) cell surface receptor signaling pathway
( ) plasma membrane receptor complex
(C) plasma membrane
(C) cell surface
```

We can use one term to filter the nodes.

```
tf = pa.dgraph.vs.select(lambda vertex:
pa.go[9606].get_term('transcription factor complex') in
vertex['go']['C'])
tfr = pa.dgraph.vs.select(lambda vertex:
pa.go[9606].get_term('transcriptional repressor complex') in
vertex['go']['C'])
print('Number of nodes annotated as \'transcription factor complex\':
{}'.format(len(tf)))
```

```
print('Number of nodes annotated as \'transcriptional repressor
complex\': {}'.format(len(tfr)))
# Note: some nodes may be annotated with both GO terms
print('Number of nodes annotated with any of the two terms above:
{}'.format(len(set(tf['label']+tfr['label']))))
```

```
Number of nodes annotated as 'transcription factor complex': 124
Number of nodes annotated as 'transcriptional repressor complex': 37
Number of nodes annotated with any of the two terms above: 155
```

Or we can filter the nodes according to several terms as well. For example, we can try to locate all the nodes corresponding to cell membrane proteins located in its surface.

```
filter_func = lambda vertex: pa.go[9606].get_term('cell surface') in
vertex['go']['C'] and pa.go[9606].get_term('plasma membrane') in
vertex['go']['C']
pm = pa.dgraph.vs.select(filter_func)
print('Number of nodes annotated with \'cell surface\' and \'plasma
membrane\': {}'.format(len(pm['label'])))
```

```
Number of nodes annotated with 'cell surface' and 'plasma membrane':
215
```

However, Pypath also provides some specific methods to locate transcription factors and receptors.

```
pa.set_transcription_factors()
pa_tf = pa.transcription_factors()
pa_tf = pa.graph.vs.select(lambda vertex: vertex['tf'] is True)
```

```
pa.set_receptors()
pa_rec = pa.graph.vs.select(lambda vertex: vertex['rec'] is True)
```

The accompanying document **TF_location.html** is a rendered IPython Notebook that contains code and output corresponding to this section.

3.7 Negatome database

The Negatome database (http://mips.helmholtz-muenchen.de/proj/ppi/negatome/) contains information on experimentally supported non-interacting protein pairs. In other words, the interactions loaded with this database represent interactions that do not occur according to lab experiments. This can be very useful information in different scenarios. For loading it we can use Pypath's `load_resources()` function.

```
pa.load_resources(lst=pypath.data_formats.negative)
```

If we load this resource after loading Omnipath, the edges in the graph will consist of both, interactions supported by the literature and non-existent interactions according to Negatome. We can get the list of edges coming from the Negatome database with the following line of code:

```
negatome_edge_list = pa.graph.es.select(lambda edge: 'Negatome' in
edge['sources'])
```

An interesting question is whether there are edges with contradictory evidence, i.e. with support in the literature and also listed in Negatome.

```
n_sources = np.array([len(i) for i in negatome_edge_list['sources']])
print('Number of edges with contradictory evidence:
{}'.format(np.sum(n_sources>1)))
```

```
Number of edges with contradictory evidence: 127
```

We can have a closer look at one of this contradictions and look into the references reporting it to decide whether the interaction can really take place or not.

```
contradictory_edges = negatome_edge_list.select(lambda edge:
len(edge['sources'])>1)

i_contradictory_edge = contradictory_edges[0]
i_source_label = pa.graph.vs[i_contradictory_edge.source]['label']
i_target_label = pa.graph.vs[i_contradictory_edge.target]['label']
print(' == '.join([i_source_label, i_target_label]))

print('Sources: ' + ', '.join(i_contradictory_edge['sources']))
```

```
AKT1 == TSC1

Sources: Negatome, SPIKE
```

We can open the paper supporting this non-interaction for a more in-depth evaluation.

```
contradictory_ref =
i_contradictory_edge['refs_by_source']['SPIKE'][0]
contradictory_ref_info =
contradictory_ref.info()[contradictory_ref.pmid]

contradictory_ref.open()  # opens reference in pubmed
```

The accompanying document **Negatome_loading.html** is a rendered IPython Notebook that contains code and output corresponding to this section.

Chapter 4 Prior-Knowledge Network building

In some cases, we might want to have a prior-knowledge network with the aim of simulating some experimental data. For example, Lescarbeau et al. (2014) provide a phosphoproteomic dataset in prostate cancer cell lines under different perturbations and time points.

Table 2 provides a list of phosphoproteins considered of interest in Lescarbeau et al. (2014). Here we have to take one thing into account: the names assigned to phosphoproteins might not be standard names, and thus it may be tricky to find the correct match in terms of the Uniprot ID or the gene symbol, which are the main entry point to Omnipath and Pypath. At the moment, knowledge about the experiment and the system being studied is probably the best way to find the correct matches.

We can use several of the techniques discussed in the previous chapter to extract our prior-knowledge network. The first thing we can do is, having our list of proteins stored in the variable `query_set_d`, retrieve all the links reported in the directed network.

```
query_set_dnetwork = pa.dgraph.induced_subgraph(query_set_d)

print('Number of edges: {}'.format(query_set_dnetwork.ecount()))
print('Number of nodes: {}'.format(query_set_dnetwork.vcount()))
```

```
Number of edges: 104

Number of nodes: 34
```

And we can check if there is any node or group of nodes not connected to the rest by querying the connected components of the network.

```
# extract connected components using igraph's clusters() function
# here mode='weak' instead of the default value 'strong'
query_set_dclusters = query_set_dnetwork.clusters(mode='weak')
n_dclusters = len(query_set_dclusters)
print('Number of connected components: {}'.format(n_dclusters))
for i in xrange(n_dclusters):
    print('\tComponent {} size: {}'.format(i,
len(query_set_dclusters[i])))
```

```
Number of connected components: 5
      Component 0 size: 29
      Component 1 size: 2
      Component 2 size: 1
      Component 3 size: 1
      Component 4 size: 1
```

```
for i in range(1, n_dclusters):
    print('Genes in component {}:'.format(i))
    print(query_set_dnetwork.vs[query_set_dclusters[i]]['label'])
```

```
Genes in component 1:
['IL6R', 'IL6']
```

```
Genes in component 2:
['TUBB']
Genes in component 3:
['MAPK10']
Genes in component 4:
['MAPK13']
```

Phosphoprotein	Gene	Measured	Stimulated
Erk1	MAPK3	1	0
Erk2	MAPK1	1	0
Akt1	AKT1	1	0
Akt2	AKT2	1	0
Akt3	AKT3	1	0
RPS6	RPS6	1	0
GSK3a	GSK3A	1	0
GSk3b	GSK3B	1	0
p38d	MAPK13	1	-1
JNK1	MAPK8	1	0
JNK2	MAPK9	1	0
JNK3	MAPK10	1	0
HSP27	HSPB1	1	0
Stat3	STAT3	1	0
IGF-1	IGF1	0	1
IL6	IL6	0	1
EGF	EGF	0	1
TNFa	TNF	0	1
Docetaxel	-	0	1
DHT	-	0	1
PI3K	PIK3CA	0	-1
mTOR	MTOR	0	-1
MEK	MAP2K1	0	-1
IKKa	CHUK	0	-1
IKKb	IKBKB	0	-1
b-Tubulin	TUBB	0	0
AR	AR	0	0
IGF1-R	IGF1R	0	0
b-Catenin	CTNNB1	0	0
IL6R	IL6R	0	0
Jak	JAK1	0	0
EGFR	EGFR	0	0
RAS	KRAS	0	0
Stress	-	0	0
Rac	RAC1	0	0
TNFR	TNFRSF1A	0	0
NF-kB	NFKB1	0	0

Table 2. List of phosphoproteins measured in Lescarbeau et al. (2014) and additional nodes considered of interest. The column stimulation takes the value 1, -1 or 0 depending on whether the node was stimulated with an activating agent, inhibited or not perturbed, respectively.

In the case of the unconnected node IL6R, if we look in the undirected network instead of in the directed one, we can see that it is connected to JAK1. The reason for not being connected in the directed network is that when getting a directed network, by default, only edges with an explicit reference to its directionality are kept. We can thus add this link manually to our network, as we are trying to get a single network that connects all the nodes and we have a reference for it (although without an explicit mention to its directionality).

```
# add directed edge from IL6R to JAK1
label_source = 'IL6R'
label_target = 'JAK1'

iprot_source = query_set_dnetwork.vs.find(label=label_source).index
iprot_target = query_set_dnetwork.vs.find(label=label_target).index
query_set_dnetwork.add_edge(iprot_source, iprot_target)

# copy attribute values present in the undirected network
original_attributes = pa.graph.es(pa.get_edge([pa.gs(label_source),
pa.gs(label_target)]))[0].attributes()

new_edge_id = query_set_dnetwork.get_eid(iprot_source, iprot_target)
new_edge_pointer = query_set_dnetwork.es(new_edge_id)[0]

for (ikey, ivalue) in original_attributes.iteritems():
    new_edge_pointer[ikey] = ivalue
```

For connecting the rest of the nodes, we might search paths to connect them to the nodes in the big connected component as suggested in the previous chapter. If we do it, we will observe that we are able to connect those isolated nodes to many other nodes in the big connected component just using one intermediary node not present in the list of queried nodes.

Because here we are interested in extracting a network that could be used for simulation, we want to keep the network small, so we decide not to include all those possibilities. Furthermore, in Lescarbeau et al. (2014), they provide a diagram of the signaling pathway under study, so we can decide to include all these possible edges or only some of them based on that. For example, *β*-Tubulin (TUBB) is not connected to other nodes in the diagram. Also, JNK3 (MAPK10) may already be considered to be represented by JNK1 and JNK2 (MAPK8 and MAPK9 respectively). On the other hand, p38 (MAPK13) is connected to Rac (RAC1) and HSP27 (HSPB1) in the diagram. We can check one of the shortest paths that connects RAC1 to MAPK13 and another that connects MAPK13 to HSPB1.

```
RAC1 --> PAK1 --> MAP2K3 --> MAPK13

MAPK13 --> PRKD1 --> HSPB1
```

To go from RAC1 to MAPK13 we need at least 3 steps in the directed network. We may decide, however, to add this connection nonetheless. In other words, we can decide to complete the network with the diagram provided in Lescarbeau et al. (2014) in mind. If the small network obtained this way does not explain the observations we can go back and consider other alternative wiring. We will leave TUBB and MAPK10 isolated for the moment too.

Additionally, we might be interested in limiting the indegree of the nodes. The reason is that this may help in the process of fitting the model to the data in another later stage. We can always come back later and include alternative edges if the model we obtain does not fit the data appropriately. In this case, we have decided that we would like to limit the maximum indegree to 5. For achieving this, we need to discard some edges for those nodes with an

indegree higher than 5. To choose the edges to be discarded, one option is to check the number of references supporting each edge and discard those with the lowest support.

```
# Add a new attribute to each vertex with its indegree
pkn_network.vs['indegree'] = [i for i in pkn_network.indegree()]

max_indegree = 5
ids_edges_to_remove = []
for i_vertex in pkn_network.vs:
    if i_vertex['indegree']>max_indegree:
        edges_in = pkn_network.es.select(_target=i_vertex.index)
        print(i_vertex['label'])
        print("nrefs: " + ", ".join([str(i) for i in
edges_in['nrefs']]))
        #print("degree of source: " + ",
".join([str(pkn_network.vs[i.source].degree()) for i in edges_in]))

        # get the threshold of nrefs that leaves max_indegree edges
or less
        nrefs_list = np.array([e['nrefs'] for e in edges_in])
        nrefs_list.sort()
        nrefs_threshold = nrefs_list[-(max_indegree+1)]
        # get indices of edges to be removed
        ids_edges_to_remove.extend([e.index for e in
edges_in.select(nrefs_le=nrefs_threshold)])
        print("Suggested deletions: ")
        for i_edge_to_remove in
edges_in.select(nrefs_le=nrefs_threshold):
            i_source_label =
pkn_network.vs[i_edge_to_remove.source]['label']
            i_target_label =
pkn_network.vs[i_edge_to_remove.target]['label']
            print('\t' + ' --> '.join([i_source_label,
i_target_label])  + "\t({} refs)".format(i_edge_to_remove["nrefs"]))
        print('---')


AKT1
nrefs: 8, 22, 1, 26, 11, 2, 4
Suggested deletions:
      TNF --> AKT1      (1 refs)
      HSPB1 --> AKT1    (2 refs)
---
CHUK
nrefs: 8, 5, 7, 1, 2, 32, 2, 1, 1
Suggested deletions:
      TNF --> CHUK      (1 refs)
      AKT2 --> CHUK     (2 refs)
      MTOR --> CHUK     (2 refs)
      HSPB1 --> CHUK    (1 refs)
      AKT3 --> CHUK     (1 refs)
---
STAT3
nrefs: 21, 18, 1, 9, 18, 17, 4
Suggested deletions:
      RAC1 --> STAT3    (1 refs)
      MTOR --> STAT3    (4 refs)
---
```

```
EGFR
nrefs: 2, 1, 15, 9, 15, 3
Suggested deletions:
      JAK1 --> EGFR     (1 refs)
---
CTNNB1
nrefs: 2, 7, 8, 18, 1, 1, 1, 32, 2, 3
Suggested deletions:
      AKT1 --> CTNNB1   (2 refs)
      NFKB1 --> CTNNB1  (1 refs)
      AKT2 --> CTNNB1   (1 refs)
      IKBKB --> CTNNB1  (1 refs)
      GSK3A --> CTNNB1  (2 refs)
---
NFKB1
nrefs: 8, 1, 4, 2, 1, 8
Suggested deletions:
      CTNNB1 --> NFKB1  (1 refs)
      AKT2 --> NFKB1    (1 refs)
---
```

We should take into account that removing edges may have undesired consequences, such as disconnecting a node from the rest of the network. If this happens, we might decide not to delete some of these edges. In this case, we have no new disconnected nodes.

```
pkn_network_v2 = pkn_network.copy()
pkn_network_v2.delete_edges(ids_edges_to_remove)
```

This last network, which could be further refined, contains 34 and 90 edges (Figure 4).

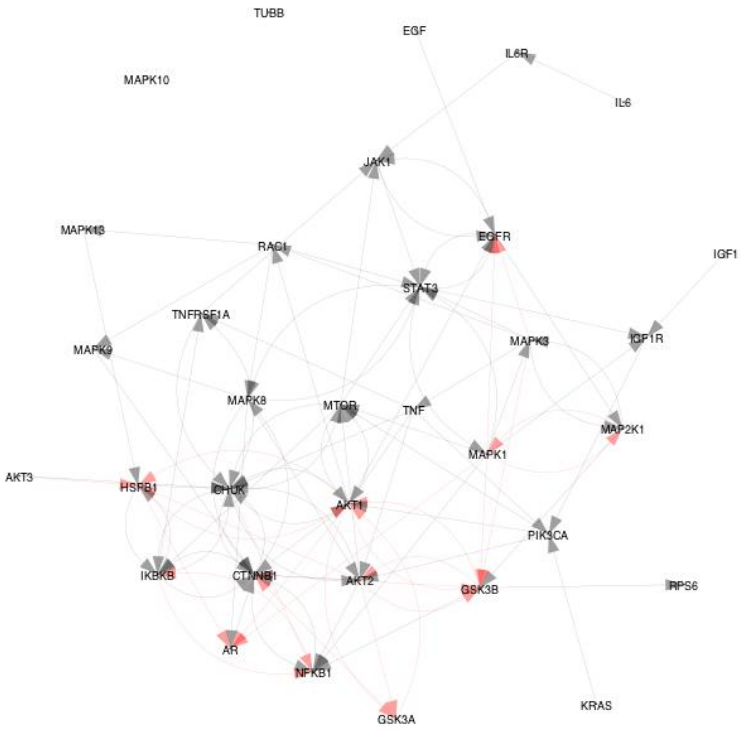

Figure 4. Extracted directed prior knowledge network for nodes of interest in Lescarbeau et al 2014.

The accompanying document **lescarbeau_et_al_2014_neighbourhood.html** is a rendered IPython Notebook that contains code and output corresponding to this chapter with some additional observations.

Chapter 5 Summary and Conclusion

This document has given an introduction to Omnipath and Pypath, and how they can be used to address common tasks in prior knowledge retrieval. The different use cases presented on it show the various ways in which Omnipath and Pypath can help to build a computational pipeline for the extraction of prior knowledge protein interaction information.

Following the pipeline presented in Chapter 4, members of the consortium are using Omnipath not only to extract the links reported in some of the 27 high confidence curated databases when a new protein is added to the network of prostate cancer (WP5), but also to update the network regularly with new links from recent publications.

In summary, Omnipath and Pypath greatly facilitate the integration and extraction of biological prior knowledge for analysis and model building, and they can easily be incorporated into wider data processing pipelines. Omnipath and Pypath are available at http://omnipathdb.org.

Chapter 6 Bibliography

[1] Türei D, Korcsmáros T, Saez-Rodriguez J (2016) Benchmark of literature curated signalling pathway resources (under review)

[2] Lescarbeau RM, Kaplan DL (2014) Quantitative analysis of castration resistant prostate cancer progression through phosphoproteome signalling. BMC Cancer, 14, 325