



AI創薬のための ケモインフォマティクス 入門

@fmkz__, @iwatobipen

Version 0.1(Draft) 2019/02/23

目次

はじめに	1
RDKitとは	1
対象読者	2
おまけ	3
ケモインフォマティクスのための環境を整えよう	4
Anacondaとは	4
Anacondaのインストール方法	4
Packageのインストール	4
インストールしたパッケージの説明	5
Condaについてもう少し詳しく	5
Pythonプログラミングの基礎	7
Pythonの基礎	7
Jupyter notebookで便利に使おう	8
Pythonで機械学習をするために	8
ケモインフォマティクスのための公開データベース	9
ChEMBL	9
PubChem	9
どちらを使うべき?	9
その他有用なデータベース	10
RDKitで構造情報を取り扱う	11
SMILESとは	11
構造を描画してみよう	11
複数の化合物を一度に取り扱うには?	12
化合物の類似性を評価してみる	15
化合物が似ているとはどういうことか?	15
類似度を計算する	15
(おまけ) もう少しドラッグライクな分子の類似性を評価する	16
グラフ構造を利用した類似性の評価	17
主要な骨格による分類(MCS)	17
Matched Molecular Pairによる化合物ネットワーク	19
Cytoscapeを使ってMMPネットワークを可視化する	20
沢山の化合物を一度にみたい	25
Chemical Spaceとは	25
ユークリッド距離を用いたマッピング	26
tSNEをつかったマッピング	28
構造活性相関 (QSAR) の基礎	30
効果ありなしの原因を考えてみる (分類問題)	30
薬の効き目を予測しよう (回帰問題)	32
R分解とFree-Wilson Analysis	33
モデルの適用範囲(applicability domain)	33
ディープラーニング入門	34
ディープラーニングに関して	34

TensorFlowとKerasについて	34
Google colabについて	35
インストールしてみよう	35
ディープラーニングを利用した構造活性相関	36
DNNを利用した予測モデル構築	36
記述子を工夫してみる(neural fingerprint)	40
準備	41
実例	42
REINVENTについて簡単に説明	43

はじめに

ケモインフォマティクス(Chemoinformatics)とは、主に化学に関するデータをコンピュータを用いて解析し、様々な課題を解くために用いる方法論のことです。ケモインフォマティクスという言葉は1990年終わりから2000年はじめに定義され、現在では製薬業界でも薬学系のアカデミアで一般的に利用されている技術となっています。

ケモインフォマティクスは薬剤の効果と化合物の形状との関連性を解析したり、大量の化合物情報の可視化、化合物の類似性に基づいたクラスタリングなどの多岐にわたるプロセスで活用されていますが、近年ではディープラーニング(Deep Learning)の創薬応用がライフサイエンス分野でのトレンドとなっています。

特に創薬化学の領域では、活性や物性を予測するQSAR（構造活性相関）への適用だけではなく、新規デザインへの応用や合成経路提案への適用といった従来のケモインフォマティクスではあまり扱わなかった領域においても応用研究が盛んに行われています。

そもそもどのような化合物を作るべきか、どのように化合物を合成するか、を考えるプロセスは背景知識と想像力が求められる領域であるため、従来は人以外が担うのは難しい領域であると認識していましたが、そういう領域に対してもAIと呼ばれるものが進出がここ数年(2017-2019)で急速に進みました。

ケモインフォマティクスはすでに様々な場面で利用されていますが、これまで活用するための情報があり見つかりませんでした。この理由として考えられることはいくつかあります。ケモインフォマティクス自体の進化の速度が早いことも一因ですが、やはりオープンソースのツールが存在しなかったというのが最も大きな原因と考えています。しかし、RDKitというオープンソースのケモインフォマティクスツールキットの登場により、この点は急速に解消しつつあります。近年では、ケモインフォマティクスもバイオインフォマティクス同様、だれでも自由に使えるデータベースが充実してきています。このため現在ではウェブで検索すれば多くの情報がすぐに得られ、自己学習することは十分可能となっています。

ですが、最初の一歩を踏み出すためのまとまった情報も必要だと考え、ケモインフォマティクスに関する基礎を学び、それらを応用できるようになるようなコンテンツを用意することにしました。最後のほうの章では「AI創薬」の文脈で用いられるディープラーニングを利用した化合物提案の方法を説明しています。

RDKitとは

warning

ここは@iwatobipenがRDKitについて熱く語るサブセクションです。下書き段階での「申し上げる」とか「踏まえて」等の言い回しをそのまま生かし、自称を「拙者」にして「ござる」調が@iwatobipen風文体です。

拙者、本書の一部を執筆する@iwatobipenと申す。ここではRDKitに関して熱く語ってみようと思うでござる。

RDKitのRDはなんなのか？実は**Rational Discovery**の略省であり現在のオープンソースの前身となるフレームワークは2000年に開発されたでござる。随分と古いでござるね。その後、2006年にコードがオープンソースになりsourceforgeから公開されたでござる。PythonのケモインフォマティクスツールキットはRDKit以外にOpenBabelもあるぞと思われる読者もござろう。OpenBabelは2005年に最初のリリースがなされておる。いずれも、もう10年以上の歴史があるツールキットでござる。拙者がこの辺りに興味を持ち始めた2012年ころはどちらかというとOpenBabelの方がメジャーだったように記憶しておる。当時、日

本語の記事はほぼ皆無であり、拙者は本書共著者であり業界のパイオニアでもある@fmkz__殿の[ケモインフォックブック](#)などを参考にRDKitのコードを書いて試行錯誤していたでござるよ。なお、ケモインフォ関連のヒストリを追いたい御仁はこちらの[記事](#)を一読されるとよかろう。

おっと話が横道に逸れてしまった。本題に戻ろう。

開発者のGreg Landorum氏いわく

RDKitはケモインフォマティクスにおけるSwiss Army Knifeであり、様々な機能ピースの集合体である

— Greg Landorum

これはまさに目的を得た表現でござる。[公式ドキュメント](#)を見ればわかるでござろうが、既に色々な機能が用意されておるのだ。 化合物情報の読み込み、書き込みに始まり、構造の描画、3次元構造配座発生、Rグループ分解、記述子、フィンガープリント計算、ファーマコフォア算出などなど、挙げればきりがないほどの機能が実装されておる。解析から可視化まで幅広い範囲をカバーできるのだ。 さらにContributerらがRDKitを利用して開発したツール群がその熱い想いとともに[Contrib](#)フォルダーに詰められておるのだ。どうじゃ使ってみたくならんか？。拙者はもう書きながらも早くRDKitに触りたくなってきたでござる。

RDKitは開発やユーザーコミュニティの活動も活発で、どんどん機能追加がされておる。世界中の有能な研究者が全体で盛り上げ開発していくスタイルはオープンソースの強みであり、魅力であろう。もしチャンスがあれば毎年開催されるRDKit User Group Meetingへの参加を検討するのもよかろう。Face2Faceでユーザー同士議論ができるのは何事にも代え難いものがあるでござる。 また、先ほど拙者が使い始めた当時は日本語の情報ほぼ皆無であったと申したが、近年は非常に良質な日本語記事もたくさん増えておる。下記に何個か例を挙げたでござる。Qiitaにも多くの記事が掲載されているでござるよ。

また、有志による[RDKit-users-jp](#)も立ち上がっておる。英語での質問がちょっと、、、と思われる御にはこちらに質問を投げかけるとよかろう。また、最新版のRDKitのリポジトリには日本語のドキュメントもマージされておる。こちらも参考になるであろう。 本書ではRDKitの一部の機能しか使わん。それでも非常に多くのことができると感じていただけるはずじゃ。興味持ちはじめの一歩を踏み出したら後はどんどん自分の興味、意欲のままに足を進めていけばよかろう。何かわからないことがあれば上記のコミュニティに問い合わせ、本書のリポジトリへIssueとして投稿してみるのもよかろう。 さあそれでは始めよう！

主な日本語解説サイト

- [rdkit-users.jp](#)
- [RDKitドキュメンテーション非公式日本語版サイト](#)
- [化学の新しいカタチ](#)

対象読者

- 医学薬学系の大学院生及び薬学系のデータ解析を行いたいポスドク
- 製薬企業の薬理研究者で自分のデータを自分で解析したい人
- 創薬化学者でケモインフォマティクスの必要性を感じている方
- 突然企業でケモインフォマティクス要員にアサインされた方

- AI創薬に興味があるがなにからはじめたらいいかわからない人

おまけ

Chemoinformatics or Cheminformatics?

もともとはBioに対してChemoと語感を合わせて登場してきたように記憶しているが、[Journal of Cheminformatics](#)の創刊により一時期Chemに大きく離されていた。

最近の[Google trend](#)によるとどちらでもいいようですが個人的にはRhymeは重視したほうが良いと思うので本書ではChemoの方を使うことにします。

ケモインフォマティクスのための環境を整えよう

本書を読み進めるにあたり必要な環境構築を行いましょう。 本書の実行環境はPython3.6を想定しています。

Anacondaとは

Anacondaとは、機械学習を行うための準備を限りなく楽にするためのパッケージで、これを利用することで、機械学習をすぐに始められるようになります。

Q&A

なぜAnacondaを利用するのか？

機械学習で広く用いられているプログラミング言語Pythonは多くの有用なライブラリが用意されており、言語自体も非常にわかりやすく学習コストが低いために、ケモインフォマティクスに限らず多くの場面で使われています。しかしながら、それらのライブラリの多くは標準のPythonに付属しておらず自分でインストールする必要があります。プログラミングに精通している方にとっては大した問題ではありませんが、初学者や単純に機械学習をするためPythonを導入したい方には環境構築は大変苦痛な作業です。Anacondaを利用することで、このプログラミング環境の構築作業の手間を大幅に削減することができます。

Pythonには大きく2.x系のバージョンと3.x系のバージョンとがありますが?

2.x系はサポートが2020年までとなっているので新規にPythonを学ぶ方は2.x系を使う必要はありません。

Anacondaのインストール方法

では早速Anacondaをインストールしましょう。[アナコンダの公式サイト](#)にアクセスし、ご自身の環境にあったインストーラー(Python3.7version(201901現在))をダウンロードします。 ついでLinuxであれば、ターミナルからインストーラーを実行します。

```
$ bash ~/Downloads/Anaconda3-4.1.0-Linux-x86_64.sh
```

インストーラーが起動しいくつかの質問をされますが、基本的にエンターまたはYesで進めてください。

Anacondaのインストールが完了すると、コマンドプロンプトまたはターミナルから'conda'コマンドが使えるようになるはずです。

Packageのインストール

今回はPython3.6の環境を構築し作業を進めていくので次の手順で開発環境を構築します。コマンドの-nの後ろは"py4chemoinformatics"としていますが皆さんの好きな名前でも構いません。 環境構築後、本章以降で利用するパッケージをインストールします。

```
$ conda create -n py4chemoinformatics python3.6
$ source activate py4chemoinformatics

# install packages
$ conda install -c conda-forge rdkit
$ conda install -c conda-forge seaborn
$ conda install -c conda-forge ggplot
$ conda install -c conda-forge git
```

インストールしたパッケージの説明

RDKit

RDKitはケモインフォマティクスの分野で最近よく用いられるツールキットの一つです。オープンソースソフトウェア(OSS)と呼ばれるものの一つで、無償で利用することができます。詳しくは[はじめに](#)を参照してください。

seaborn

統計データの視覚化のためのパッケージの一つです。

ggplot

グラフ描画パッケージの一つで一貫性のある文法で合理的に描けることが特徴です。もともとはRという統計解析言語のために開発されました。yhatという会社により[Pythonに移植](#)されました。

git

バージョン管理システムです。本書ではgitについては説明しませんのでもしGitについて全然知らないという方は[サルでもわかるGit入門](#)でも読みましょう。

Condaについてもう少し詳しく

先に説明したAnacondaでインストールされるデフォルトのPythonのバージョンは3.7ですが、本書執筆時点ではconda-forgeで配布されているRDKitのパッケージが要求するPythonのバージョンが3.6となっています。このためcondaで仮想環境を構築しました。

なぜ仮想環境を作るのでしょうか

いくつかのシステムでは様々な機能を提供するために内部的にPythonを利用しているため、特定のパッケージのためにPythonのバージョンを変更してしまうと問題が起こることがあります。仮想環境はこのような問題を解決します。もし、パッケージが異なるライブラリのバージョンを要求しても仮想的なPython環境を準備して試行錯誤できます。不要になれば仮想環境を簡単に削除でき、もとの環境にトラブルを持ちこむこともありません。このように、ひとつのシステム内にそれぞれ個別の開発環境を作成できるようにすることで開発時によく起こるライブラリの依存問題やPythonのバージョンの違いに悩まされることがなくなります。

よく利用するcondaのサブコマンドを挙げておきます。

```
# install package
$ conda install <package name>

# 仮想環境の作成。
$ conda create -n 仮想環境の名前 python=/バージョン

# 作った仮想環境一覧の表示
$ conda info -e

# 仮想環境の削除
$ conda remove -n 仮想環境の名前

# 仮想環境を使う
# mac/linuxの場合
$ source activate 仮想環境の名前

# 仮想環境を使う
# windowsの場合
$ activate 仮想環境の名前

# 仮想環境から出る
$ source deactivate

# 今使っている仮想環境にインストールされているライブラリの一覧を表示
$ conda list
```

Pythonプログラミングの基礎

Pythonの基礎

この章ではPythonに触れたことのない読者のために効率的に勉強するためのサイトや本などを紹介します。もしこれ以降の章でわからないことなどがあったら、この章のサイトや本を参考に学んでみてください。

Pythonを本で学びたい

Pythonスタートブック増補改訂版

プログラミング自体が初心者であればこの本が良いでしょう。

みんなのPython 第4版

Javascript,Javaなどのなかでプログラミングを少しかじっていて、これからPythonを覚えたいのであればこちらの本をおすすめします。

Pythonを本以外で学びたい

Python Boot Camp(初心者向けPythonチュートリアル)

一般社団法人PyCon JPが開催している初心者向けPythonチュートリアルイベントです。全国各地で行われているので近くで開催される場合には参加するとよいでしょう

その他ローカルコミュニティなど

あちこちで入門者向けからガチのヒト向けまでの勉強会やコミュニティのあつまりなどもあるので、そういうのに参加してモチベーションを高めるのもよい方法です。

udemy/python

オンライン学習サービスを利用するのも効果的な手段のひとつですが、筆者は試したことがないのでわかりません。周りの評判を効いてみても良いでしょう。YouTubeを探すのもあります。

本書でわからないことがあったら

py4chemoinformaticsのissues

py4chemoinformaticsのissuesに質問していただければお答えします。わかりにくい場合だったら修正しますので、よりよくなつてみんなハッピー。

Qiita

Qiitaで探せば大抵答えが見つかるとはずです。

stackoverflow

それでも答えが見つからなかつたらsofで探すか質問しましょう

Mishima.syk

本書を書いている人たちが集まるコミュニティです。特に話題をPythonに限定していませんが、Pythonを使ったネタが多めです。かなりガチですが、初心者対応も万全でハンズオンに定評があります。質問されれば大体答えられます。

Jupyter notebookで便利に使おう

Jupyter notebookを利用すると、コードを書いて結果を確認するということがとても簡単にできるようになります。

Jupyter notebookはWebブラウザベースのツールで、コードだけではなくリッチテキスト、数式、なども同時にノートブックに埋め込みます。また結果を非常に綺麗な図として可視化することも容易にできます。つまり、化学構造やグラフも描画できるため、ケモインフォマティクスのためのプラットフォームとして使いやすいです。さらに、プログラミングの生産性を上げるような、ブラウザ上でコードを書くとシンタックスハイライトや、インデント挿入を自動で行ってくれたりという便利な機能もついているので、特に初学者は積極的に使うべきでしょう。

使い方

terminal(Windowsではanaconda prompt)から

```
$ jupyter notebook
```

と打てばjupyter notebookが立ち上がります。本書ではこれ以降特に断らない限りjupyter notebookでのコードを実行することとします。

Pythonで機械学習をするために

chemoinformaticsに限らず、インフォマティクスを学ぶにあたり、機械学習は外せません。本書でもある程度の機械学習の知識があることを前提に進めていきます。Pythonで機械学習をするにはScikit-learnというライブラリを利用するのが定番であり、本書でも特に説明せずに利用していきますが、初学者のために参考となる書籍などをすすめておきます。

Pythonではじめる機械学習 —scikit-learnで学ぶ特徴量エンジニアリングと機械学習の基礎

Pythonで機械学習をやるために基礎を学べます。数学的な表現があまりないので読みやすいです。

sklearn-tutorial

y-samaによるsklearnのチュートリアルハンズオンのjupyter notebookです。

ケモインフォマティクスのための公開データベース

この章ではケモインフォマティクスでよく使うデータベースを紹介します。

化合物関連データベースの歴史

そもそも公共の化合物関連のデータベースというものは歴史が浅いです。では化合物関連データベース自体の歴史が浅いのかと言われれば全然そんなことはなく、昔から色々なデータベースが存在しました。問題はほぼ全てが商用のサービスであったということで、オープンな化合物データベースが登場したのは実はごく最近のことです。アミノ酸、塩基配列、蛋白質結晶構造などのデータベースがかなり以前から無償で提供されていたのに比べると、公開データベースがないという状況はオープンネスを失わせる要員の一つであったと言えるでしょう。

ChEMBL

ChEMBLはEBIのChEMBLチームにより維持管理されている医薬品及び開発化合物の結合データ、薬物動態、薬理活性を収録したデータベースです。データは主にメディシナルケミストリ関連のジャーナルから手動で抽出されており、大体3,4ヶ月に一度データの更新があります。

メディシナルケミストリ関連のジャーナルからデータを収集しているため、QSARに関する情報や背景知識を論文そのものに求めることができます。創薬研究をする際には有用です。

ChEMBLの秘密

ChEMBLはもともとはstARLITeという商用のデータベースだったはずです。（要出典）

PubChem

PubChemはNCBIにより維持管理されている低分子化合物とその生物学的活性データを収録している公開リポジトリです。5000万件以上の化合物情報と、100万件を超えるアッセイデータを含みそのデータ量の多さが特徴とも言えます。もうひとつの特徴はデータをアカデミアからの化合物登録や圧制結果の登録により成長することであり、ここが先のChEMBLとの大きな違いであるといえます。

特にPubChemは初期スクリーニングのデータが多いため、そのようなデータに対しなんらかのマイニングや分析を行いたい場合は有用だと考えられる。

どちらを使うべき？

QSARをやりたい場合にはやはりChEMBLのデータを利用することが多いです。IC50のようなデータが得られていることが多いですし、モデルの解釈に元論文をあたることができるというのが大きな理由です。

その他有用なデータベース

のちほど追記していきます。IssueやPRでも受け付けてます。

RDKitで構造情報を取り扱う

この章ではRDKitを使って分子の読み込みの基本を覚えます。本章のjupyter notebook

SMILESとは

Simplified molecular input line entry system(SMILES)とは化学構造を文字列で表現するための表記方法です。 詳しくは[SMILES Tutorial](#)で説明されていますが、例えばc1ccccc1は6つの芳香族炭素が最初と最後をつないでループになっている構造、つまりベンゼンを表現していることになります。

構造を描画してみよう

SMILESで分子を表現することがわかったので、SMILESを読み込んで分子を描画させてみましょう。まずはRDKitのライブラリからChemクラスを読み込みます。二行目はJupyter Notebook上で構造を描画するための設定です。

```
from rdkit import Chem  
from rdkit.Chem.Draw import IPythonConsole
```

RDKitにはSMILES文字列を読み込むためにMolFromSmilesというメソッドが用意されていますので、これを使い分子を読み込みます。

```
mol = Chem.MolFromSmiles("c1ccccc1")
```

続いて構造を描画しますが、単純にmolを評価するだけで構造が表示されます。

```
mol
```

図のように構造が表示されているはずです。

```
In [31]: from rdkit import Chem  
from rdkit.Chem.Draw import IPythonConsole
```

```
In [32]: mol = Chem.MolFromSmiles("c1ccccc1")
```

```
In [35]: mol
```

Out[35]:



複数の化合物を一度に取り扱うには？

複数の化合物を一つのファイルに格納する方法にはいくつかありますが、sdfというファイル形式を利用するのが一般的です。

sdfフォーマットとは？

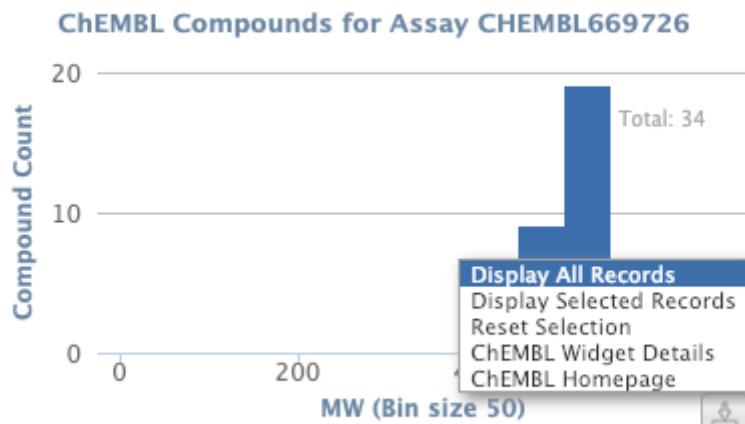
MDL社で開発された分子表現のためのフォーマットにMOL形式というものがあります。このMOL形式を拡張したものがSDF形式です。具体的にはMOL形式で表現されたものを""という行で区切ることにより、複数の分子を取り扱えるようにしてあります。

sdfファイルをChEMBLからダウンロードする

ここではトポイソメラーゼII阻害試験の結果をsdfファイル形式でダウンロードします。

ChEMBLの検索結果のページのcompound summaryからDisplay All Recordsを選択します。

Compound Summaries



さらに、化合物表示画面からdownload sdfでsdfファイルをjupyter notebookを起動したディレクトリにダウンロードします。

Max Phase	Parent Mol Weight	ALogP	PSA	HBA	HBD	#RO5	Vio.	#Rotatable Bonds
0	611.49	6.26	95.48	8	2	2		5

Please select....
FOR ALL COMPOUNDS ON DISPLAY:
1. Download (SDF)
2. Download (Tab-delimited + includes SMILES)
3. Download Compound IDs

FOR CHECKBOX SELECTED COMPOUNDS:
1. Download (SDF)
2. Download (Tab-delimited + includes SMILES)
3. Download Compound IDs
4. Filter Bioactivities
5. Display Bioactivities

便宜上ch05_compounds.sdfという名前で保存しましたが、好きな名前で保存してください。

RDKitでsdfを取り扱う

RDKitでsdfファイルを読み込むにはSDMolSupplierというメソッドを利用します。複数の化合物を取り扱うことになるのでmolではなくmolsという変数に格納していることに注意

```
mols = Chem.SDMolSupplier("ch05_compounds.sdf")
```

何件の分子が読み込まれたのか確認します。数を数えるにはlenを使います。

```
len(mols)
```

34件でした。

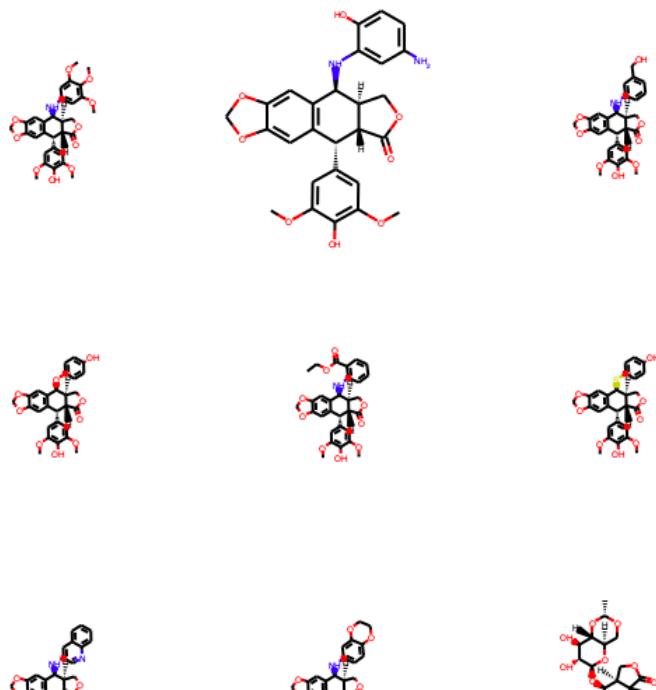
分子の構造を描画する

forループを使って、ひとつずつ分子を描画してもいいですが、RDKitには複数の分子を一度に並べて描画するメソッドが用意されているので、今回はそちらのMolsToGridImageメソッドを使います。

```
Draw.MolsToGridImage(mols)
```

In [49]: Draw.MolsToGridImage(mols)

Out[49]:



(おまけ)参考までにループを回すやりかたも載せておきます。

```
from IPython.core.display import display
for mol in mols:
    display(mol)
```

Q&A

変数名を分子の数でmolやmolsに変えるのはなぜですか？

どういう変数を使うかの決まりはありませんが、見てわかりやすい変数名をつけることで余計なミスを減らし、生産性が上がります

MolsToGridImageで一行に並べる分子の数を変更することはできますか？

できます。molsPerRowというオプションが用意されています。Draw.MolsToGridImage(mols, molsPerRow=10)のように使います

化合物の類似性を評価してみる

化合物が似ているとはどういうことか？

2つの化合物が似ているとはどういうことでしょうか？なんとなく形が似ている？という表現は科学的ではありません。ケモインフォマティクスでは類似度(一般的に0-100の値を取ります)や非類似度（距離）といった定量的な尺度により似ているか どうかを評価しますがここでは主に2つの代表的な測り方を紹介します。

記述子

分子の全体的な特徴をあらわすもの、分子量や極性表面性（PSA）、分配係数(logP)などを分子記述子または単に記述子と呼び、現在までにかなりの数が提案されています。これらの記述子の類似性を評価することで2つの分子がどのくらい似ているかを表現することが可能ですが、いくつかの記述子に関しては市販ソフトでないと計算できない場合があります。また分子全体の特徴を一つの数字で表現するために局所的な特徴を表現できないといったデメリットもあります。

フィンガープリント

もう一つの方法としてフィンガープリントを利用することもできます。フィンガープリントとは分子の部分構造を0,1のバイナリーで表現したもので部分構造の有無とビットのon(1),off(0)を対応させたものです。フィンガープリントには固定長FPと可変長FPの二種類が存在し、古くはMACSKeyという固定長FP(予め部分構造とインデックスが決められているFP)が使われていましたが、現在ではECFP4(Morgan2)という可変長FPが利用されるのが普通です。

RDKitのフィンガープリントに関しては[Gregさんのスライド](#)が詳しいので熟読してください。

今回はこのECFP4(Morgan2)を利用して類似性評価をしてみましょう。

類似度を計算する

まずは手始めに簡単な分子としてトルエンとクロロベンゼンの類似性を評価してみましょう。

```
from rdkit import Chem, DataStructs
from rdkit.Chem import AllChem, Draw
from rdkit.Chem.Draw import IPythonConsole
```

smilesで分子を読み込みます。

```
mol1 = Chem.MolFromSmiles("Cc1ccccc1")
mol2 = Chem.MolFromSmiles("Clc1ccccc1")
```

一応目視で確認しておきます。

```
Draw.MolsToGridImage([mol1, mol2])
```

類似度の評価にはタニモト係数を使います。

```
DataStructs.TanimotoSimilarity(fp1, fp2)
# 0.5384615384615384
```

(おまけ) もう少しドラッグライクな分子の類似性を評価する

ここでは抗凝固薬として上市されているapixaban, rivaroxabanの類似性を評価します。構造を見るとわかりますが、なんとなく似ていますが、どの部分とどの部分が対応するか 想像つくでしょうか？実はこの2つの化合物は両方共FXaというセリンプロテアーゼの同じポケットに 同じような結合モードで結合することでプロテアーゼの働きを阻害することが知られています。興味があれば 実際にPDBから複合体の結晶構造を探して眺めてみるといいかもしれません。（pymol入門まで拡張するか？要検討）

SMILESはChEMBLのものを利用しています（要リンク？）。

```
apx = Chem.MolFromSmiles("C0c1ccc(cc1)n2nc(C(=O)N)c3CCN(C(=O)c23)c4ccc(cc4)N5CCCCC5=O")
rvx = Chem.MolFromSmiles("Clc1ccc(s1)C(=O)NC[C@H]2CN(C(=O)O2)c3ccc(cc3)N4CCOCC4=O")
```

構造を眺めてみます。メトキシフェニルとクロロチオールは同じような結合様式をとるんでしょうか？このような結合の成分をきちんと評価する方法もあるのですが、本書の内容を超えるので説明はしません。もし興味があればFragment Molecular Orbital Methodで調べてみてください

```
Draw.MolsToGridImage([apx, rvx], legends=["apixaban", "rivaroxaban"])
```

```
apx_fp = AllChem.GetMorganFingerprint(apx, 2, useFeatures=True)
rvx_fp = AllChem.GetMorganFingerprint(rvx, 2, useFeatures=True)
```

```
DataStructs.TanimotoSimilarity(apx_fp, rvx_fp)
# 0.40625
```

40%くらいの類似度ということになりました。

グラフ構造を利用した類似性の評価

グラフとはノード（頂点）群とノード間の連結関係を示すエッジ（枝）群で構成されるデータのことを指します。化学構造はこのグラフで表現できます。つまり原子をノード、結合をエッジとしたグラフ構造で表せます。

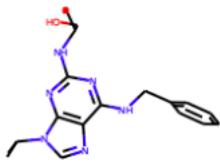
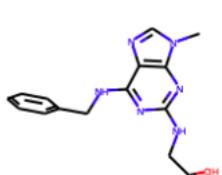
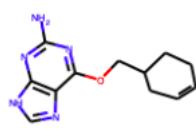
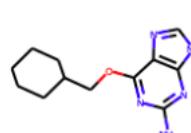
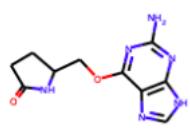
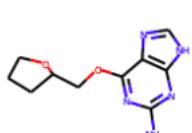
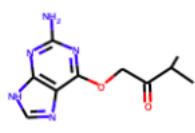
通常、06章で紹介したようなフィンガープリントを使い分子同士の類似性を評価することが多いですが、グラフ構造を利用して類似性を評価する手法もあります。次に紹介するMCS（Maximum Common Substructure）は対象となる分子集合の共通部分構造のことを指します。共通部分構造が多いほどそれらの分子はより似ていると考えます。

主要な骨格による分類(MCS)

最大共通部分構造Maximum Common Substructure(MCS)とは与えられた化学構造群において共通する最大の部分構造のことです。RDKitではMCS探索のためにrdFMCSというモジュールが用意されています。

今回はMCS探索のサンプルデータとしてrdkitに用意されているcdk2.sdfというファイルを利用します。RDConfig.RDDocsDirが、サンプルデータのディレクトリを表す変数で、そのディレクトリ以下のBooks/data/にcdk2.sdfというファイルが存在するので、os.path.joinメソッドでファイルパスを設定します。尚、os.path.joinはosのパスの違いを吸収するためのpythonの組み込みモジュールです。

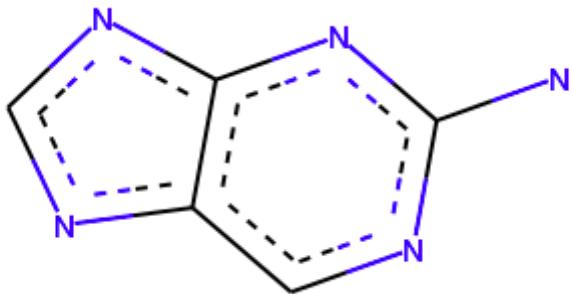
```
import os
from rdkit import Chem
from rdkit.Chem import RDConfig
from rdkit.Chem import rdFMCS
from rdkit.Chem.Draw import IPythonConsole
from rdkit.Chem import Draw
filepath = os.path.join(RDConfig.RDDocsDir, 'Book', 'data', 'cdk2.sdf')
mols = [mol for mol in Chem.SDMolSupplier(filepath)]
# 構造を確認します
Draw.MolsToGridImage(mols[:7], molsPerRow=5)
```



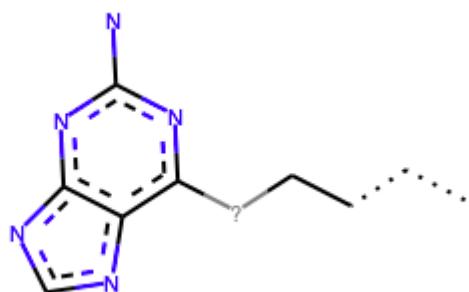
読み込んだ分子を使ってMCSを取得します。RDKitではMCSの取得方法に複数のオプションが指定できます。以下にそれぞれのオプションでの例を示します。

1. デフォルト
2. 原子がなんであっても良い（構造とボンドの次数があつていれば良い）
3. 結合次数がなんでも良い（例えば、ベンゼンとシクロヘキサンは同じMCSとなる）

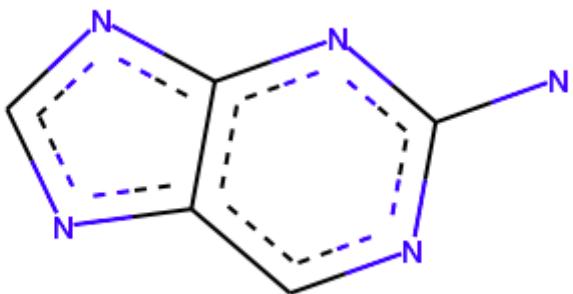
```
result1 = rdFMCS.FindMCS(mols[:7])
mcs1 = Chem.MolFromSmarts(result1.smartsString)
mcs1
print(result1.smartsString)
#[#6]1:[#7]:[#6](:[#7]:[#6]2:[#6]:1:[#7]:[#6]:[#7]:2)-[#7]
```



```
result2 = rdFMCS.FindMCS(mols[:7], atomCompare=rdFMCS.AtomCompare.CompareAny)
mcs2 = Chem.MolFromSmarts(result2.smartsString)
mcs2
print(result2.smartsString)
#[#6]-,:[#6]-,:[#6]-[#6]-[#8,#7]-[#6]1:[#7]:[#6](:[#7]:[#6]2:[#6]:1:[#7]:[#6]:[#7]:2)-[#7]
```

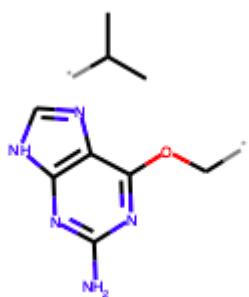


```
result3 = rdFMCS.FindMCS(mols[:7], bondCompare=rdFMCS.BondCompare.CompareAny)
mcs3 = Chem.MolFromSmarts(result3.smartsString)
mcs3
print(result3.smartsString)
#[#6]1:[#7]:[#6](:[#7]:[#6]2:[#6]:1:[#7]:[#6]:[#7]:2)-[#7]
```



RDKitではMCSに基づく類似性を数値化するアルゴリズムのひとつにFraggle Similarityが実装されています。これを利用することでクラスタリングや、類似性に基づいた解析が行なえます。

```
from rdkit.Chem.Fraggle import FraggleSim
sim, match = FraggleSim.GetFraggleSimilarity(mols[0], mols[1])
print(sim, match)
#0.925764192139738 *C(C)C.*C0c1nc(N)nc2[nH]cnc12
match_st = Chem.MolFromSmiles(match)
match_st
```



このようにFraggleSimilarityは類似性及びマッチした部分構造を返します。ECFPを利用した類似性よりもケミストの感覚に近いことが多いです。詳しくは参考リンクを参照してください。

参考リンク

- [Efficient Heuristics for Maximum Common Substructure Search](#)
- [Fraggle – A new similarity searching algorithm](#)

Matched Molecular Pairによる化合物ネットワーク

創薬研究の構造最適化ステージにおいて、起点となる化合物（リード化合物）をどのように構造を変換していくかは非常に重要な問題です。加えてステージが進んだ場合どの構造変換が活性や物性に影響を及ぼしたかというレトロスペクティブな解析することも大切です。

TIP

興味があればhttps://sar.pharm.or.jp/wp-content/uploads/2018/09/SARNews_19.pdfを読むとよいです。

このような解析を行うためのアプローチの一つがMatched Molecular Pair Analysis (MMPA)です。MMPAでは、二つの分子の活性、物性の変化と部分構造の変化を比較し解析します。例えばフェニル基上に置換基、Cl基→F基に変換した場合、活性、物性にどのような変化があるかを調べます。もし変換

の前後で活性は変化せず物性パラメータが大きく変化したらそれは生物等価体と見なせます。MMPは基本的には変換した部位に着目する解析手法であり、大規模なデータを利用することでパラメータの変動のトレンドの把握ができます。

ここではRDKitのContribに提供されている[RDKit/Contrib/MMPA\[mmpa\]](#)を使ってMMP解析を行います。

作業ディレクトリをRDKitインストール先の下にあるContrib/mmpaに移し、pythonスクリプトを順次実行します。

```
python rfrag.py <MMPAを実施したいSmilesFileの名前 >フラグメント化したデータの保存ファイル名
```

```
# 例えば
```

```
# python rfrag.py <data/sample.smi >data/sample_fragmented.txt
```

```
python indexing.py <先のコマンドでできたフラグメントのファイル >MMP_アウトプットファイル.CSV
```

```
# 例えば
```

```
# python index.py <data/sample_fragmented.txt >data/mmp.csv
```

以上のコマンドを実行するとmmp.csvに分子A,分子B,分子AのID,分子BのID,変換された構造のSMIRKS,共通部分構造（context）が出力されます。ペアのIDが出力されていますのでこれに活性や物性などの評価値を紐つけることで構造変化と評価結果の変動の解析を実施することができます。 MMPは変換前、変換後の情報をノード、変換ルールをエッジと考えるとグラフ構造です。Cytoscapeなどのネットワーク可視化ツールを利用するとMMPAの結果をより直感的に把握することができます。

興味のある方はさらに読みすすめてください。

Cytoscapeを使ってMMPネットワークを可視化する

WARNING この内容は入門の内容を超えるので興味がなければ飛ばしてください

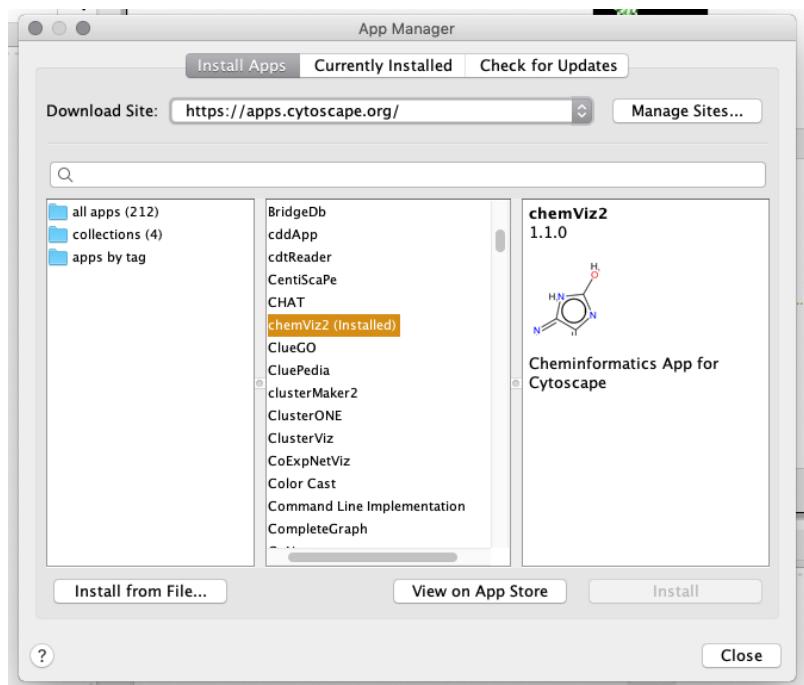
RDKitには先に紹介したMMPAの他に[mmpdb](#)という別プロジェクトがあります。 こちらはコマンドラインのツール群とデータベースシステムとして提供されているため、長期的な管理がしやすいという特徴があります。本セクションではこのmmpdbとCytoscapeを利用したMMPの可視化を紹介します。

Cytoscapeのインストール

[Cytoscape](#)はオープンソースのネットワーク可視化ソフトで色々なシーンで広く使われています。化合物の構造表示用プラグインを使うことで構造のネットワークを表示することができます。

インストールは簡単で[ダウンロードサイト](#)から対応するOSのインストーラをダウンロードして指示のとおりにインストールするだけです。

インストールが完了したらCytoscapeを起動して化合物構造描画用のChemviz2プラグインをインストールします。手順は簡単でApps→App Managerからchemviz2を選択してインストールします。



mmpdbからgmlファイルを作成する

今回利用するデータは<Inhibition of recombinant GSK3-beta> J. Med. Chem. (2008) 51:2062-2077の151化合物です。MMPAを行うにはHTSのような探索データではなくて構造最適化のようにスキヤフオールドが決まっているものを使うのが原則です。

コマンドの流れを載せておきます。smilesのtextと活性や物性値のデータは別々にデータベースに登録する必要があります。

```
$ mmpdb fragment smiles.txt -o CHEMBL930273.fragments      # fragmentation
$ mmpdb index CHEMBL930273.fragments -o CHEMBL930273.db    # make db
$ mmpdb loadprops -p act.txt CHEMBL930273.db                 # load properties
```

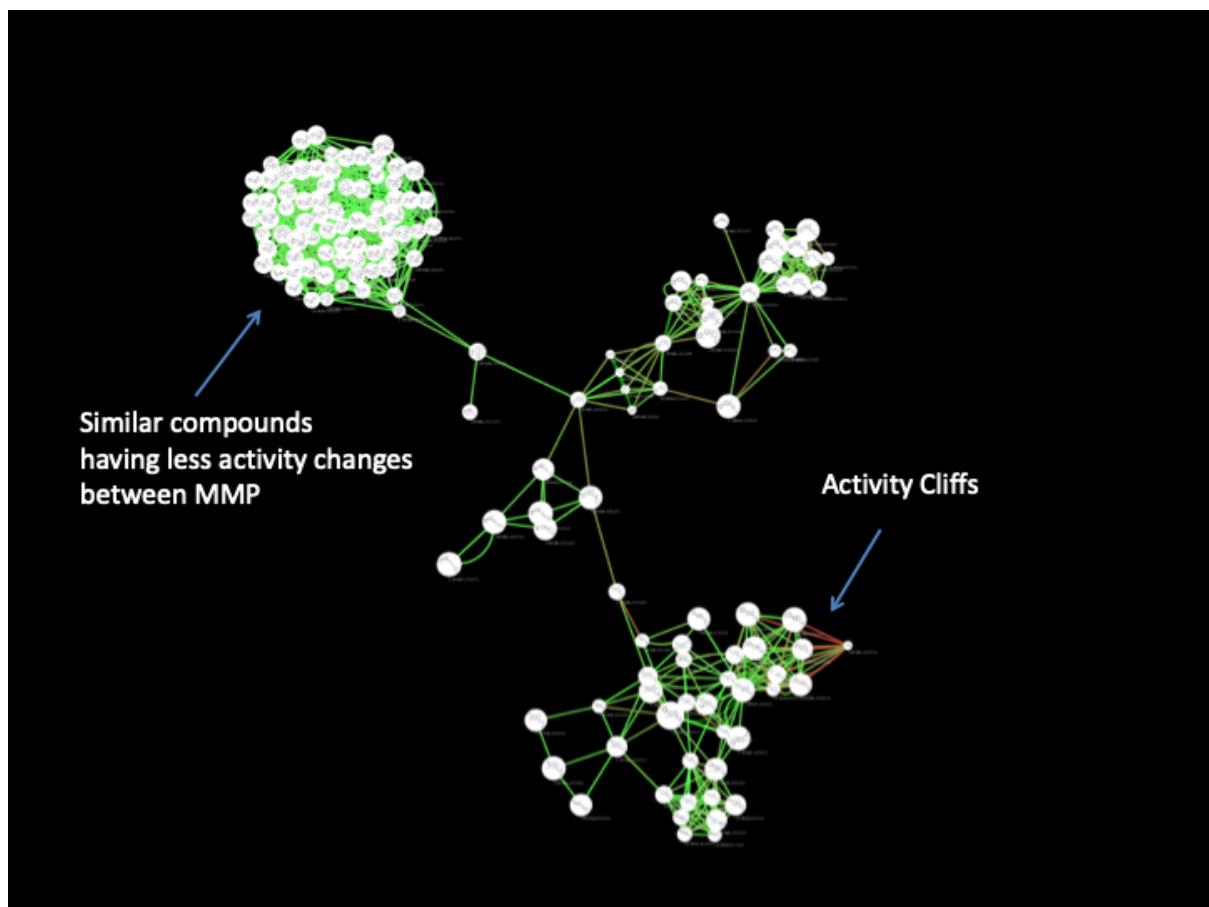
その後Cytoscapeで読み込むためのgmlファイルを作成しますが、これは本書の範囲を超えるので割愛します。もし興味があるのであれば[コード](#)を直接読んでもらうといいのですが流れは以下のとおりです。

1. mmpdbからpython-igraphをつかてgmlファイルを作る
2. gmlファイルをCytoscapeで読み込む
3. Cytoscapeで属性をいい感じにいじる
 - a. ノードの大きさを物性値に対応
 - b. エッジの色を活性差に対応
 - c. chemviz2 pluginで構造を描画してノードに貼り付ける

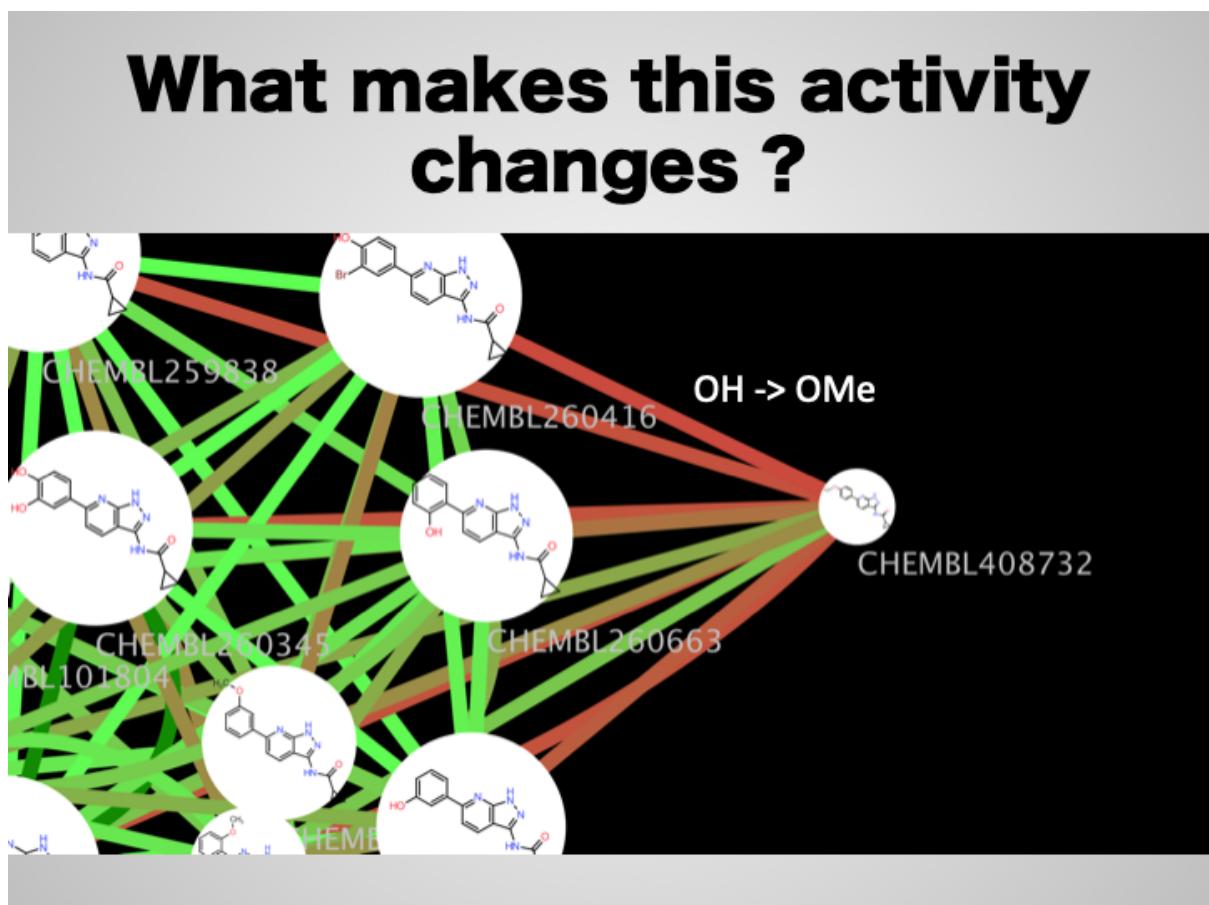
解釈する

さてMMPネットワークを見てみましょう。あまり活性差のないMMPが左上の方に固まっています。右下の方にはエッジが赤い（活性差が大きい）ものが観測されます。このような小さな置換基変化が大きな活性差を生むもMMPをActivity Cliffと呼びます。一般的にActivity Cliffは創薬プロジェクトにおいてプレー

クスルーとなることが多いのでこういう活性変化を見逃さないことが大切です。



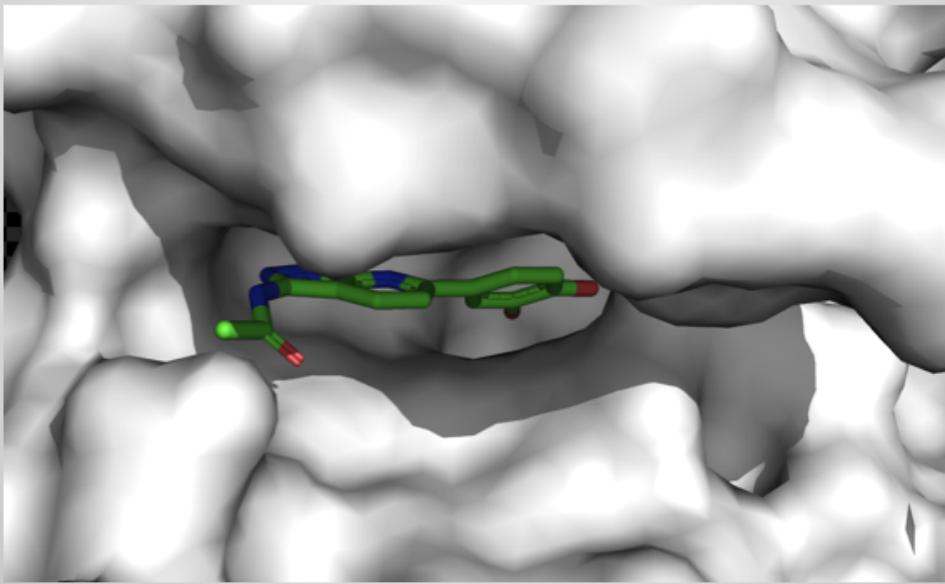
実際にどういう置換が行われたのかを確認すると、OH基をMeO基に置換することで活性が消失しています。



MMPだけではこのように単純に事実しかわかりませんが、今回はもう少し深く考察するために類似体の複合体結晶構造を探してみました。するとPDBID:5OY4というGSK3 β と類似化合物の複合体が見つかりました。

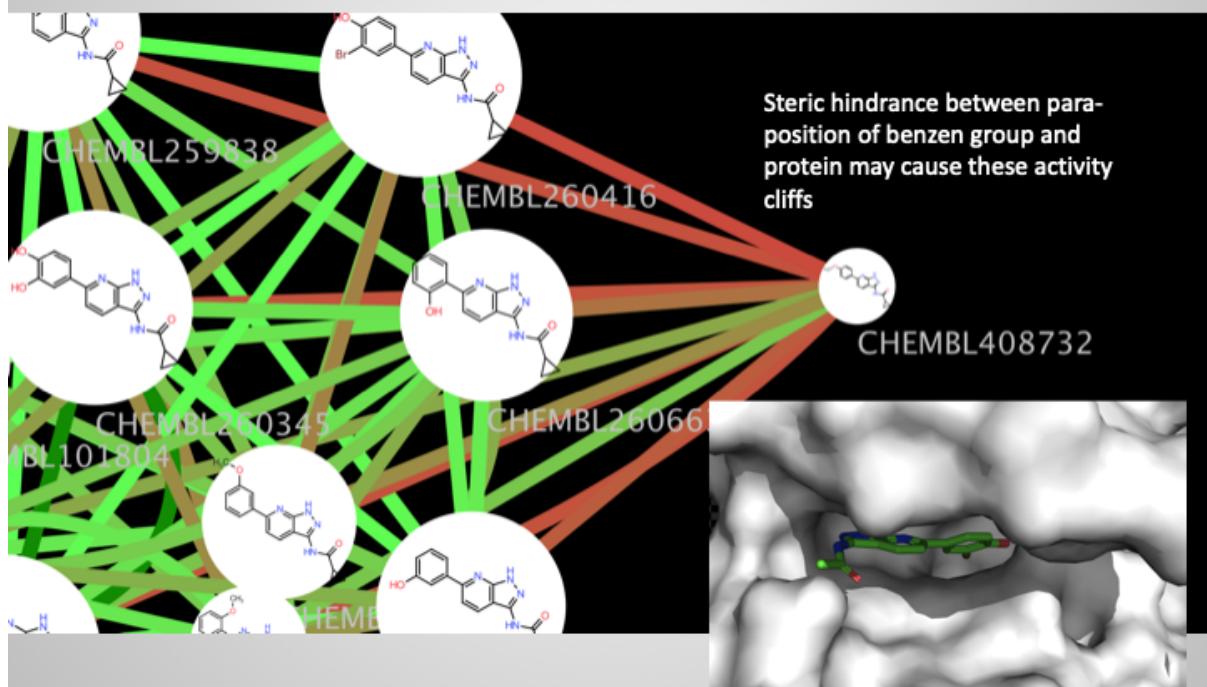
PDB-ID:5OY4

- GSK3beta complex with N-(6-(3,4-dihydroxyphenyl)-1H-pyrazolo[3,4-b]pyridin-3-yl)acetamide



OH基をMeO基に置換するとポケットの壁にぶつかりそうですね。したがってこのActivity Cliffはリガンドと蛋白質の立体障害により引き起こされたと考察されます。

Interpretation of AC



MMPを視覚化して解釈する例を紹介しました。

参考資料

- MMP visualization with Cytoscape
- Cytoscapeでchemoinformatics
- mmpdb: An Open Source Matched Molecular Pair Platform for Large Multi-Property Datasets

沢山の化合物を一度にみたい

沢山のデータがどのように分布しているのかを見るには適当な空間にマッピングするのが一般的です。特にケモインフォマティクスではケミカルスペースという言葉が使われます。

Chemical Spaceとは

ケミカルスペースとは化合物を何らかの尺度でn次元の空間に配置したものを指します。一般に、2次元または3次元が使われることが多いです（人間の理解のため）。尺度つまり類似性に関しては色々な手法が提案されていますが、うまく化合物の特徴を表すような距離が定義されるように決められることが多いです。

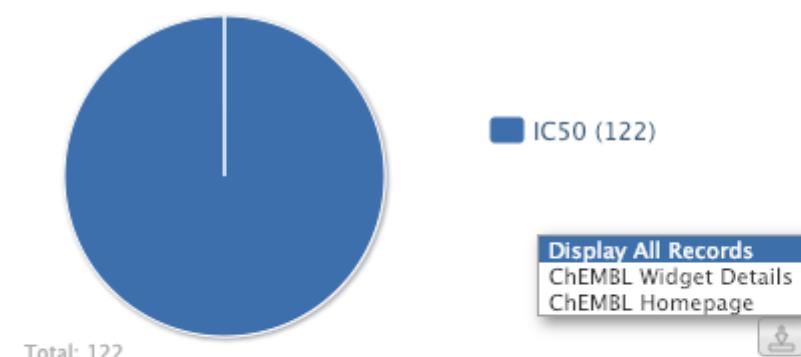
ここではいくつかの尺度を用いてケミカルスペースを構築してみましょう。使うデータはChEMBLのFXa酵素阻害アッセイに含まれる122化合物です。

アッセイから全化合物表示をします。

Target	Target Type
Coagulation factor X	SINGLE PROTEIN

Bioactivity Summary

ChEMBL Activity Types for Assay CHEMBL3705525



Compound Summaries

続いてタブ区切りテキストでダウンロードします。



Cells Tissues Exact Match [Activity Source Filter](#)

Please select....
[Download All Bioactivity Data \(Tab-delimited\)](#)
[Download All Bioactivity Data \(XLS\)](#)

[Show / hide columns](#)

Assay Src Description	Assay Organism	Target Type	Target Name	Target Organism	Reference
BindingDB Database	Homo sapiens	SINGLE PROTEIN	Coagulation factor X	Homo sapiens	CHEMBL3638623

この中にSMILES形式で構造情報が入っています。

ユーリックリッド距離を用いたマッピング

描画ライブラリにはggplotを使います。まだインストールされていない場合はconda install ggplotでインストールしてください。主成分分析(PCA)を利用して、化合物が似ているものは近くになるように分布させて可視化します。

```
from rdkit import Chem, DataStructs
from rdkit.Chem import AllChem, Draw
import numpy as np
import pandas as pd
from ggplot import *
from sklearn.decomposition import PCA
```

タブ区切りテキストを読み込み、SMILESの領域を取り出してMolFromSmilesメソッドでmolオブジェクトに変換します。

```

mols = []
with open("ch08_compounds.txt") as f:
    header = f.readline()
    smiles_index = -1
    for i, title in enumerate(header.split("\t")):
        if title == "CANONICAL_SMILES":
            smiles_index = i
    for l in f:
        smi = l.split("\t")[smiles_index]
        mol = Chem.MolFromSmiles(smi)
        mols.append(mol)

```

molオブジェクトからフィンガープリントを構築しますが、sklearnで取り扱えるようにnumpyアレイに変換します。 rdkitのフィンガープリントをsklearnで取り扱う場合にはこの変換操作が必須なのでそういうものだとして覚えててしまうとよいです。

```

fps = []
for mol in mols:
    fp = AllChem.GetMorganFingerprintAsBitVect(mol, 2)
    arr = np.zeros((1,))
    DataStructs.ConvertToNumpyArray(fp, arr)
    fps.append(arr)
fps = np.array(fps)

```

これで主成分分析の用意が整ったので早速やりましょう。主成分の数はn_componentsで指定できます。

```

pca = PCA(n_components=2)
x = pca.fit_transform(fps)

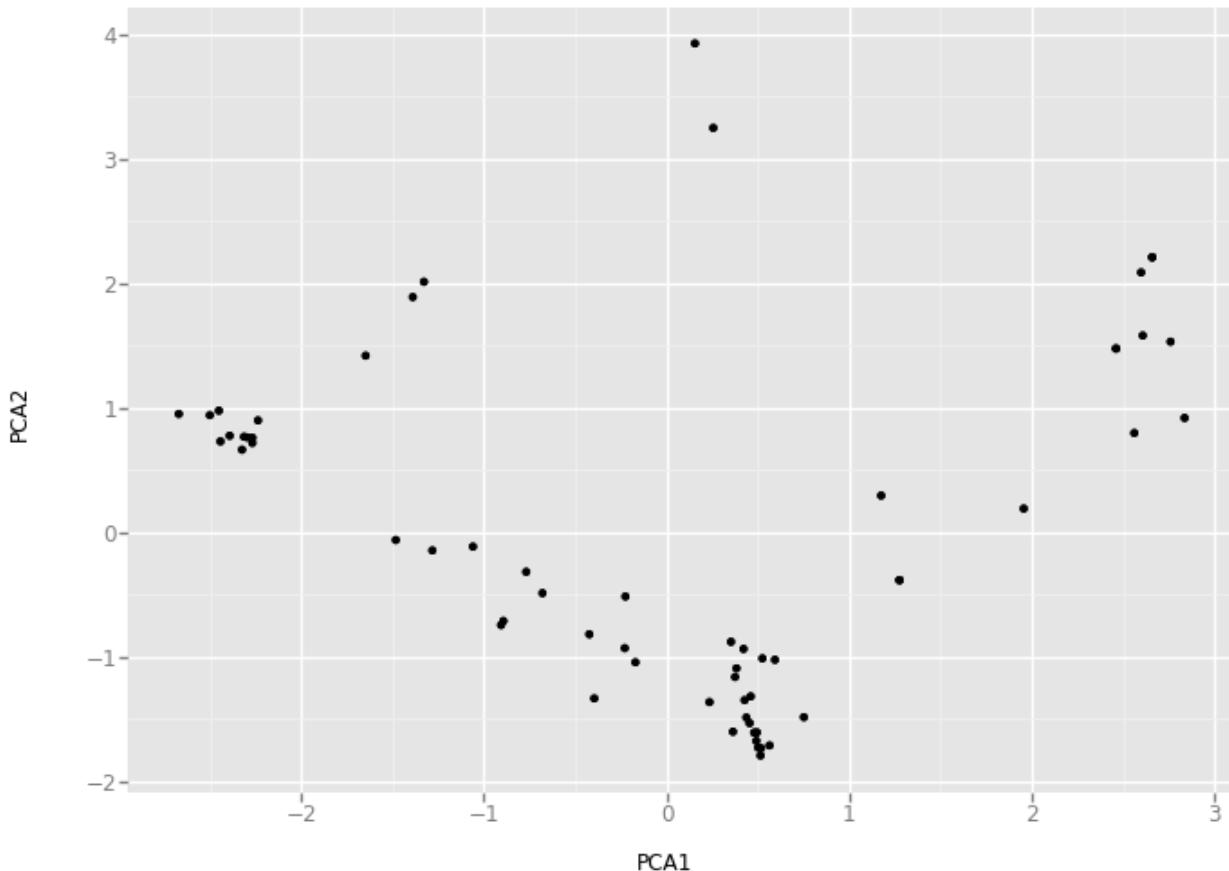
```

描画します。

```

d = pd.DataFrame(x)
d.columns = ["PCA1", "PCA2"]
g = ggplot(aes(x="PCA1", y="PCA2"), data=d) + geom_point() + xlab("PCA1") + ylab("PCA2")

```



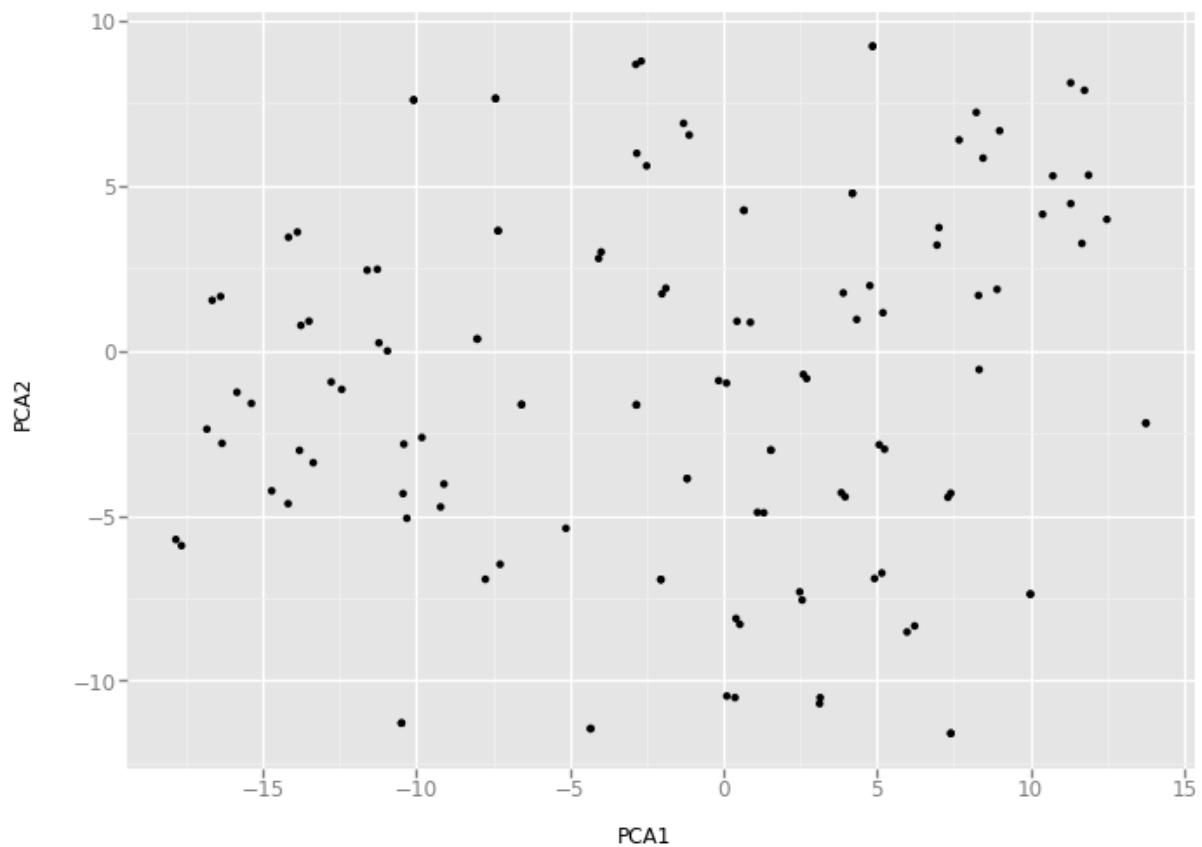
tSNEをつかったマッピング

PCAよりもtSNEのほうが分離能がよく、メディシナルケミストの感覚により近いと言われています。

```
from sklearn.manifold import TSNE
tsne = TSNE(n_components=2, random_state=0)
tx = tsne.fit_transform(fps)
```

描画します。

```
d = pd.DataFrame(tx)
d.columns = ["PCA1", "PCA2"]
g = ggplot(aes(x="PCA1", y="PCA2"), data=d) + geom_point() + xlab("PCA1") + ylab("PCA2")
```



今回紹介したPCA,tSNEの他にも色々な描画方法があるので調べてみるとよいでしょう。

構造活性相関（QSAR）の基礎

化学構造と生物学的活性における相関関係をStructure Activity Relationship(SAR)またはQuantitative SAR(QSAR)と呼びます。一般的には似たような化合物は似たような生物学的活性を示すことが知られており、この相関関係を理解しドラッグデザインに活かすことが創薬研究において大変重要です。

また、このような問題には細胞の生死、毒性の有無といった化合物がどのクラスに入るのかを推定する分類問題と阻害率（%inhibition）といった連続値を推定する回帰問題の2つがあります。

効果ありなしの原因を考えてみる（分類問題）

ChEMBから[hERG阻害アッセイ](#)の73データを用いてIC50が1uM未満のものをhERG阻害あり、それ以外をhERG阻害なしとラベルします。

まずは必要なライブラリをインポートします。

```
from rdkit import Chem, DataStructs
from rdkit.Chem import AllChem, Draw
from rdkit.Chem.Draw import IPythonConsole
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, f1_score
from sklearn.ensemble import RandomForestClassifier
```

ChEMBLでダウンロードしたタブ区切りテキストの処理は8章とほぼ同じですが、今回は活性データが欲しいのでSTANDARD_VALUEという列を探して数値を取り出します。この値が1000nM未満であればPOSというラベルを、そうでなければNEGというラベルを振ります。最後にラベルをnumpy arrayにしておきます。

```

mols = []
labels = []
with open("ch09_compounds.txt") as f:
    header = f.readline()
    smiles_index = -1
    for i, title in enumerate(header.split("\t")):
        if title == "CANONICAL_SMILES":
            smiles_index = i
        elif title == "STANDARD_VALUE":
            value_index = i
    for l in f:
        ls = l.split("\t")
        mol = Chem.MolFromSmiles(ls[smiles_index])
        mols.append(mol)
        val = float(ls[value_index])
        if val < 1000:
            labels.append("POS")
        else:
            labels.append("NEG")

labels = np.array(labels)

```

続いてmolオブジェクトをフィンガープリントに変換します。このフィンガープリントからhERG阻害の有無を予測するモデルを作成します。

```

fps = []
for mol in mols:
    fp = AllChem.GetMorganFingerprintAsBitVect(mol, 2)
    arr = np.zeros((1,))
    DataStructs.ConvertToNumpyArray(fp, arr)
    fps.append(arr)
fps = np.array(fps)

```

データセットを訓練セットテストセットの2つに分けます。テストセットは作成した予測モデルの精度を評価するためにあとで使います。

```
x_train, x_test, y_train, y_test = train_test_split(fps, labels)
```

予測モデルを作成するにはインスタンスを作成してfitメソッドで訓練させるだけです

```

rf = RandomForestClassifier()
rf.fit(x_train, y_train)

```

先程分割しておいたテストセットを予測します。

```
y_pred = rf.predict(x_test)
```

confusion matrixを作成します。

```
confusion_matrix(y_test, y_pred)
```

f1スコアを見てみましょう。

```
f1_score(y_test, y_pred, pos_label="POS")
```

あまりよくないですね。

薬の効き目を予測しよう（回帰問題）

回帰モデルは最初に説明したとおり、連続値を予測するモデルとなります。今回はRandomForestの回帰モデルを作成して、その精度をR2で評価します。データは分類問題で使ったhERGのアッセイデータを利用することにしましょう。最初に必要なライブラリをインポートします。

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score
from math import log10
```

分類問題のときにはラベル化しましたが、今度は連続値を予測したいのでpIC50に変換します。(なぜpIC50にすると都合が良いのかはそのうち補足する)

```
pIC50s = []
with open("ch09_compounds.txt") as f:
    header = f.readline()
    for i, title in enumerate(header.split("\t")):
        if title == "STANDARD_VALUE":
            value_index = i
    for l in f:
        ls = l.split("\t")
        val = float(ls[value_index])
        pIC50 = 9 - log10(val)
        pIC50s.append(pIC50)

pIC50s = np.array(pIC50s)
```

データセットをトレーニングセットとテストセットの2つに分割します。フィンガープリントは分類モデルのときに作成したものを使います。

```
x_train, x_test, y_train, y_test = train_test_split(fps, pIC50s)
```

訓練します。Scikit-learnの場合はこの手順はどの手法でもほぼ同じメソッドでfitしてpredictです。

```
rf = RandomForestRegressor()  
rf.fit(x_train, y_train)
```

予測しましょう。

```
y_pred = rf.predict(x_test)
```

予測精度をR2で出してみます。

```
r2_score(y_test, y_pred)
```

まずはといったところでしょうか。

R分解とFree-Wilson Analysis

モデルの適用範囲(applicability domain)

今回紹介した手法は似たような化合物は似たような生物学的活性を示すという仮設に基づいて生成されるモデルです。もしトレーニングセットに似ている化合物が含まれなかった場合の予測精度はどうなるのでしょうか？

当然その場合は予測された値は信頼できませんよね。つまり、予測値にはその予測が確からしいか？という信頼度が常にについてまわります。そのようなモデルが信頼できる、または適用できる範囲をapplicability domainと呼びます。これに関しては明治大学金子先生の[モデルの適用範囲・モデルの適用領域](#)が詳しいです。

(おまけコラム)applicability domainは信頼できるのか？

applicability domainはトレーニングセットの類似性からその予測が信頼できるかという確度を測る手法です。ここで類似性が誰のための類似性なのかという問題が出てきます。我々がこの化合物とこの化合物は似ているよねと思うのは我々の勝手ですが、似ているか似ていないかは最終的には蛋白質が判断します。このあたりはいわゆるインピーダンスマッチだと考えています。MMPの文脈で説明されるActivity Cliffなどインピーダンスマッチに気取った名前をつけただけではないでしょうか？

筆者はHugo Kubinyi先生の似ている化合物は果たして似た活性を示すのか？という疑問を、estradiolのOH基がMetxy基に変換すると活性が消失する例を上げて説明されているコメントに感銘を受けたのを覚えています。J.M.Cだと思うんですが、探しても見つからないのもしご存知でしたら教えてください。

ディープラーニング入門

9章でRandomForestを用いたQSAR解析をしましたが、QSARに用いられるアルゴリズムはこれ以外にSupportVectorMachine(SVM), LogisticRegression, ArtificialNeuralNetwork(ANN)などといったものが利用されています。

本章では、近年注目を浴びているANNの一種であるディープラーニングの概要に関して説明し、ディープラーニングフレームワークであるTensorflow/Kerasを離床してみます。

ディープラーニングについて

生物の脳には神経細胞が存在し、それらがネットワークを形成することで情報を伝達したり、記憶や学習しています。このネットワーク構造を数理モデル化したものがArtificial Neural Network(ANN)です。

ANNは、学習のための情報を入力層、入力情報のパターンを元に反応するかしないか（神経シナプスの発火に対応）を学習する中間層（または隠れ層）、最後の出力層の三層から構成されていますが、ディープラーニングはこの隠れ層の部分が分岐などを含む多層構造にすることで高精度な予測を可能としています。

ディープラーニングは近年盛んに研究されており、上記のような多層構造を持つという単純なモデルだけではなく様々な構造のモデルが日々提案されています。それらすべてのフォローは本書の範囲を超えるのでこれ以上は説明しません。

TensorFlowとKerasについて

Pythonでディープラーニングを行う場合、複数のフレームワークがあります。主なものを挙げると

- [Theano](#)
- [Tensorflow](#)
- [Keras](#)
- [MXNet](#)
- [Chainer](#)
- [PyTorch](#)

この中で本書ではGoogle社が開発しているフレームワークのTensorflow/Kerasを使います。

Tensorflowは最近1.xから2.xにメジャーアップデートをしていますが、2.x版はまだ登場したばかりで参考情報も少ないので、1.x系でまずは試しましょう。また同じ1.xでもバージョンによってAPIが少し変更されていますので、今後動かしたいコードがあった時にどのバージョンで書かれているかに気をつける必要があります。

KerasはTensorflowなどの低レベルフレームワークをバックエンドに動く高レベルAPIで、Kerasを利用することでテンポよくコードが書けます。KerasはもともとTensorflowとは独立して開発されてきましたが、最近のTensorflowはKerasを取り込んでいます。したがってTensorflow側からKerasを呼び出し、ネットワークを構築することができます。Kerasそのものを使うのが良いか、Tensorflow側のKerasを使うのが良いのか、悩ましいところです。Kerasで作ってきたコードをTensorflowに統合されたKerasに移行

するのはそれほど大変ではありません。また、今後Tensorflow側のモジュールを使いたくなることもあるかもしれませんので本書ではTensorflowに統合されたKerasを利用します。

Google colabについて

Google colaboratoryはクラウド上で実行できるJupyter notebook環境です。Theano, Tensorflow, Keras, Pytorchなどのディープラーニング用のフレームワークがインストール済みなのと時間の制限はありますがGPUが使えるため、手元にGPUマシンがなくてもディープラーニングを利用できる点が非常に魅力的です。

利用にはGoogleのアカウントを作成する必要があるので、もしGoogleアカウントを持っていなければこの機会にアカウントを取得し利用してみましょう。

インストールしてみよう

Tensorflow とKerasをインストールしてみましょう。anacondaでインストールする場合、GPU対応バージョンを使うか、CPUバージョンを使うかでインストールするパッケージが少し異なります。

```
# CPU版  
$ conda install -c conda-forge tensorflow  
# GPU版  
$ conda install -c anaconda tensorflow-gpu
```

```
$ conda install -c conda-forge keras
```

この例ではcondaコマンドを利用してインストールしていますが、pipコマンドを利用してインストールすることもできます。その場合は[公式ドキュメント](#)を参照してください。しかし基本的にはCondaで環境を作ったらCondaでパッケージを入れることが望ましいでしょう。

参考リンク

- <https://keras.io/#installation>
- <https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-pkgs.html>

ディープラーニングを利用した構造活性相関

9章で構造活性相関の基礎を学びました。10章でディープラーニングを学びました。本章では、早速DNNを利用して構造活性相関解析をします。

DNNを利用した予測モデル構築

はじめにDNNを利用したシンプルな予測モデルを構築してみます。ここでは9章と同じデータを使います。最初に分類モデルを作成し、Positiveのラベルを[0, 1], Negativeのラベルを[1, 0]の二次元のOneHotベクトルで表します。KerasのModelオブジェクトを利用してモデルを作成した場合、上記の二次元のそれぞれの期待値が得られます。どちらのクラスに属する可能性が高いかを知るにはNumpyのArgmax関数を使えばよいです。

TIP このアプローチは次元が増えても同じで、10クラス分類であれば10次元で各クラスの期待値が返ってくるので同様にArgmaxを使うことで最も期待値が大きいクラスのインデックスを取得できます。

必要なライブラリをインポートします。

```
from rdkit import Chem, DataStructs
from rdkit.Chem import AllChem, Draw
from rdkit.Chem.Draw import IPythonConsole
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, f1_score
# DNN用のライブラリを読み込みます。
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Activation
from keras.models import Model
```

次にデータを読み込みます。9章では”POS”/”NEG”をlabelsというリストに入れたので一次元表現でしたが、今回はここが二次元になっています。

```

mols = []
labels = []
with open("ch09_compounds.txt") as f:
    header = f.readline()
    smiles_index = -1
    for i, title in enumerate(header.split("\t")):
        if title == "CANONICAL_SMILES":
            smiles_index = i
        elif title == "STANDARD_VALUE":
            value_index = i
    for l in f:
        ls = l.split("\t")
        mol = Chem.MolFromSmiles(ls[smiles_index])
        mols.append(mol)
        val = float(ls[value_index])
        if val < 1000:
            labels.append([0,1]) # Positive
        else:
            labels.append([1,0]) # Negative
labels = np.array(labels)

```

続いて分類モデルと回帰モデルを順次作成します。

まずは回帰モデルで、入力は9章と同じECFPを利用しています。DNNの構築には入力データの次元を明示的に指定する必要があるためnBitsという変数を定義しています。

TIP train_test_splitにrandom_stateで適当な整数を指定すると毎回同じデータが得られるので検証の際に有用です。

```

nBits = 2048
fps = []
for mol in mols:
    fp = AllChem.GetMorganFingerprintAsBitVect(mol, 2, nBits=nBits)
    arr = np.zeros((1,))
    DataStructs.ConvertToNumpyArray(fp, arr)
    fps.append(arr)
fps = np.array(fps)

x_train1, x_test1, y_train1, y_test1 = train_test_split(fps, labels, random_state=794)

```

入力が2048次元、300ニューロンの全結合層が三層、最後の出力層が2となるニューラルネットワークを作成します。活性化関数にはReLU、出力層には二次元の多クラス分類のためにSoftmaxを用いました。

Dropout層はランダムにニューロンを欠損させることにより過学習を防ぐ役割を果たします。

Kerasではモデルを定義した後compile関数を呼ぶことでモデルを構築します。optimizer, lossは目的に応じて変更する必要がありますが、今回は'categorical_crossentropy'を使いました。

TIP ReLUはSigmoid関数の勾配消失の課題を克服できるためよく利用されます。

TIP optimizerはadam以外にも多くあるのでどれが適切かは実際は試行錯誤が必要となるでしょう。

```
# Define DNN classifier model
epochs = 10
inputlayer1 = Input(shape=(nBits, ))
x1 = Dense(300, activation='relu')(inputlayer1)
x1 = Dropout(0.2)(x1)
x1 = Dense(300, activation='relu')(x1)
x1 = Dropout(0.2)(x1)
x1 = Dense(300, activation='relu')(x1)
output1 = Dense(2, activation='softmax')(x1)
model1 = Model(inputs=[inputlayer1], outputs=[output1])

model1.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

NOTE KerasにはSequentialモデルが用意おり、これを使うことで上記の例（Functional API）よりもシンプルにネットワークを記述できます。今回Functional APIでモデルを定義したのは、こちらに慣れておくと入力が複数の場合やより複雑なモデルの構築にも対応しやすいからです。もしSequentialの書き方に興味がある方は公式サイトやQiitaを調べてください。

NOTE DNNは初期のランダムに発生させた重みに基づいて予測した予測値と実際の値を比較し、その差（LOSS）を最小化するように重みを更新するBackprobagationという手順を繰り返しながらモデルを最適化します。この繰り返しの回数を指定するのがEpochsです。Epochsを増やすとどんどん賢くなるように思われるかもしれません、計算コストがかかることと、過学習のリスクがあるので長ければ良いというものでもありません。Loss/Accuracyなどを観測しつつ適切なEpoch数を考えましょう。

モデルを構築したら後はScikit-learnと同じ感覚でfit/predictが行えます。

```
hist1 = model1.fit(x_train1, y_train1, epochs=epochs)
```

最後に結果を可視化してみます。

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(range(epochs), hist1.history['acc'], label='acc')
plt.legend()
plt.plot(range(epochs), hist1.history['loss'], label='loss')
plt.legend()
```

今回の例ではだいたい6Epochくらいでモデルが良い制度になりました。

次にテストデータで検証します。

```
y_pred1 = model1.predict(x_test1)
y_pred_cls1 = np.argmax(y_pred1, axis=1)
y_test_cls1 = np.argmax(y_test1, axis=1)
confusion_matrix(y_test_cls1, y_pred_cls1)
```

ちょっと微妙でしょうか、、、

回帰モデルも基本的には先ほどの分類問題と同じです。今度は回帰なので最後の出力層は値そのもの、つまり一次元になります。また活性化関数はSigmoidなどでは0-1になってしまってLinearとしています。学習データは9章のコードを流用しています。

```
from math import log10
from sklearn.metrics import r2_score
pIC50s = []
with open("ch09_compounds.txt") as f:
    header = f.readline()
    for i, title in enumerate(header.split("\t")):
        if title == "STANDARD_VALUE":
            value_index = i
    for l in f:
        ls = l.split("\t")
        val = float(ls[value_index])
        pIC50 = 9 - log10(val)
        pIC50s.append(pIC50)

pIC50s = np.array(pIC50s)
x_train2, x_test2, y_train2, y_test2 = train_test_split(fps, pIC50s, random_state=794)
```

次にモデルを定義します。Lossの部分が先ほどの分類モデルとは異なり、MSEになっていることに注意して下さい。

```
epochs = 50
inputlayer2 = Input(shape=(nBits, ))
x2 = Dense(300, activation='relu')(inputlayer2)
x2 = Dropout(0.2)(x2)
x2 = Dense(300, activation='relu')(x2)
x2 = Dropout(0.2)(x2)
x2 = Dense(300, activation='relu')(x2)
output2 = Dense(1, activation='linear')(x2)
model2 = Model(inputs=[inputlayer2], outputs=[output2])
model2.compile(optimizer='adam', loss='mean_squared_error')
```

ここまでできたら後は同じです。

```

hist = model2.fit(x_train2, y_train2, epochs=epochs)
y_pred2 = model2.predict(x_test2)
r2_score(y_test2, y_pred2)
plt.scatter(y_test2, y_pred2)
plt.xlabel('exp')
plt.ylabel('pred')
plt.plot(np.arange(np.min(y_test2)-0.5, np.max(y_test2)+0.5), np.arange(np.min(y_test2)-0.5, np.max(y_test2)+0.5))

```

いかがでしょうか。予測モデルはちょっとUnderEstimate気味ですかね。DNNは重ねるレイヤーの数、ドロップアウトの割合、隠れ層のニューロンの数、活性化関数の種類など数多くのパラメータをチューニングする必要があります。今回の例は決め打ちでしたが、色々パラメータを変えてモデルの性能を比較してみるのも面白いです。

記述子を工夫してみる(neural fingerprint)

さて、ここまで分子のフィンガープリントを入力としてRandomForestやDNNのモデルを作成してきました。DNNが大きく注目を浴びた理由の一つに人が特徴量を抽出しなくてもモデルが特徴量を認識してくれるということが挙げられます。

例えば画像の分類においては、からSIFTという特徴量を人が定義し、これを入力としたモデルが作られていましたが、現在のDNNにおいては基本的に画像のピクセル情報そのものを利用しています。

ケモインフォマティクスに置き換えてみると、SIFTは分子のフィンガープリントに相当します。ですのでここ(入力)をもっとPrimitiveな表現に変えることでDNNの性能が上がるのではないか?と考えるのは至極当然の流れです。2015年、Harvard大学の、Alan Aspuru-Guzikらのグループは一つのチャレンジとしてNeural Finger print/NFPというものを提唱しました。

今まで利用してきたECFPとNFPとの違いを、彼らの論文中の図を引用して示します。

Algorithm 1 Circular fingerprints

- 1: **Input:** molecule, radius R , fingerprint length S
- 2: **Initialize:** fingerprint vector $\mathbf{f} \leftarrow \mathbf{0}_S$
- 3: **for** each atom a in molecule
- 4: $\mathbf{r}_a \leftarrow g(a)$ ▷ lookup atom features
- 5: **for** $L = 1$ to R ▷ for each layer
- 6: **for** each atom a in molecule
- 7: $\mathbf{r}_1 \dots \mathbf{r}_N = \text{neighbors}(a)$
- 8: $\mathbf{v} \leftarrow [\mathbf{r}_a, \mathbf{r}_1, \dots, \mathbf{r}_N]$ ▷ concatenate
- 9: $\mathbf{r}_a \leftarrow \text{hash}(\mathbf{v})$ ▷ hash function
- 10: $i \leftarrow \text{mod}(r_a, S)$ ▷ convert to index
- 11: $\mathbf{f}_i \leftarrow 1$ ▷ Write 1 at index
- 12: **Return:** binary vector \mathbf{f}

Algorithm 2 Neural graph fingerprints

- 1: **Input:** molecule, radius R , **hidden weights** $H_1^1 \dots H_R^5$, **output weights** $W_1 \dots W_R$
- 2: **Initialize:** fingerprint vector $\mathbf{f} \leftarrow \mathbf{0}_S$
- 3: **for** each atom a in molecule
- 4: $\mathbf{r}_a \leftarrow g(a)$ ▷ lookup atom features
- 5: **for** $L = 1$ to R ▷ for each layer
- 6: **for** each atom a in molecule
- 7: $\mathbf{r}_1 \dots \mathbf{r}_N = \text{neighbors}(a)$
- 8: $\mathbf{v} \leftarrow \mathbf{r}_a + \sum_{i=1}^N \mathbf{r}_i$ ▷ sum
- 9: $\mathbf{r}_a \leftarrow \sigma(\mathbf{v} H_L^N)$ ▷ smooth function
- 10: $\mathbf{i} \leftarrow \text{softmax}(\mathbf{r}_a W_L)$ ▷ sparsify
- 11: $\mathbf{f} \leftarrow \mathbf{f} + \mathbf{i}$ ▷ add to fingerprint
- 12: **Return:** **real-valued** vector \mathbf{f}

Figure 2: Pseudocode of circular fingerprints (*left*) and neural graph fingerprints (*right*). Differences are highlighted in blue. Every non-differentiable operation is replaced with a differentiable analog.

ECFP(Circular Fingerprints)は入力の分子それぞれの原子からN近傍 (Nは任意) までの原子までの情報

をHash関数（この例ではMod）任意の値に変換、で固定長のベクトルに直すといったものでした。ざっくりいうと部分構造の有無を0/1のビット情報に直したものを利用するといったイメージです。一方、今回紹介するNFPはECFPにコンセプトは似ているのですが、Hash関数の部分がSigmoidに、Modで離散化する部分がSoftmaxになっています。従って入力されるデータセットによりECFPよりも柔軟に分子のフィンガープリントを生成することが期待されます。

この論文が発表されて以降、数多くの実装がGithubに公開されていますが、各実装ごとにKerasでもBackendがTheanoであったり、Keras/Tensorflowであっても、Keras1.xじゃないと動作しなかったりと意外と環境依存のものが多く扱いにくい状況になっています。残念なことに今回構築した環境で動作するものが公開されていませんのでKeras2.x/Python3.6で動作するものをこちらの[コード](#)をベースに作成しました。

```
git clone https://github.com/iwatobipen/keras-neural-graph-fingerprint.git
```

example.pyというファイルのコードを眺めるとなんとなく雰囲気がつかめると思います。分子の表現は、これまでの例はフィンガープリントをRDKitを使い生成していましたが、今回はこのフィンガープリントそのものをDNNが学習します。

ということで、分子をグラフとして表現したものが入力になります。Atom_matrixとして(max_atoms, num_atom_features)をEdge_matrixとして(max_atoms, max_degree)をbond_tensorとして(max_atoms, max_degree, num_bond_features)という三つの行列を使います。分子はそれぞれ原子数が異なるためmax_atomsで最大原子数を定義しています。こうすることで分子ごとに同一の行列サイズの入力となりバッチ学習が可能となります。

Exampleを実行するのであれば下記のコマンドを入力してください。

```
python example.py
```

参考リンク - [NGF-paper](#) - [DeepChem-paper](#) - [keiserlab](#) - [HIPS NFP](#) - [Theano base](#) - [for keras1.x](#) - [ericmj/graph_fp](#) - [DeepChem](#) == コンピューターに化学構造を考えさせる :imagesdir: images

Deep Learningがメディカルケミストリに大きなインパクトをもたらしたものの一つに生成モデルがあげられます。特にこの数年での生成モデルの進化は素晴らしいです。ここでは[Marcus Olivecrona](#)により開発された[REINVENT](#)を使って新規な合成案を提案させてみましょう。

準備

pytorchというディープラーニングのライブラリをcondaでインストールします。新しいバージョンでは動かないでバージョンを指定してインストールします。

pytorchとは？

keras同様TensorFlowをより便利に使うためのライブラリです。

```
$ conda install pytorch=0.3.1 -c pytorch
```

続いてREINVENT本体をGitHubからクローンします。

```
$ cd <path to your working directory>
$ git clone https://github.com/MarcusOlivecrona/REINVENT.git
```

続いて、ChEMBLの110万件くらいのデータセットで予め訓練済みのモデルをダウンロードしてきて元のデータと置き換えます。このデータはGTX 1080TiGPUマシンを利用して5,6時間かかっていますのでもしトレーニングを自分で行うのであればGPUマシンは必須です。

```
$ wget https://github.com/Mishima-syk/13/raw/master/generator_handson/data.zip
$ unzip data.zip
$ mv data ./REINVENT/
```

これで準備が整いました。

実例

ここではJanuviaとして知られる抗糖尿病薬sitagliptinの類似体を生成するようなモデルを作成してみます。

まずはtanimoto係数をスコアとして類似度の高い構造を生成するようにモデルを訓練します。今回は3000ステップ訓練しますが、大体ちょっと前のMacbook Airで7,8時間かかるので気長に待ちましょう。待てない場合は[ここ](#)のデータを使ってください。

```
./main.py --scoring-function tanimoto --scoring-function-kwags query_structure
'N[C@H](CC(=O)N1CCn2c(C1)nnc2C(F)(F)F)Cc3cc(F)c(F)cc3F' --num-steps 3000 --sigma 80
```

ここからはjupyter notebookを立ち上げます。

必要なライブラリを読みこみます。sys.path.appendはREINVENTのディレクトリを指定してください。

```
%matplotlib inline
import sys
sys.path.append("[Your REINVENT DIR]")
from rdkit import Chem
from rdkit.Chem import AllChem, DataStructs, Draw
import torch
from model import RNN
from data_structs import Vocabulary
from utils import seq_to_smiles
```

続いて、トレーニングしたモデルから50化合物サンプリングします。

```

voc = Vocabulary(init_from_file="/Users/kzfm/mishima_syk/REINVENT/data/Voc")
Agent = RNN(voc)
Agent.rnn.load_state_dict(torch.load("sitagliptin_agent_3000/Agent.ckpt"))
seqs, agent_likelihood, entropy = Agent.sample(50)
smiles = seq_to_smiles(seqs, voc)

```

実際にどんな構造が生成されたのか見てみましょう。

```

mols = []
for smi in smiles:
    mol = Chem.MolFromSmiles(smi)
    if mol is not None:
        mols.append(mol)

Draw.MolsToGridImage(mols, molsPerRow=3, subImgSize=(500,400))

```

まずはといったところでどうか？

```

In [3]: mols = []
for smi in smiles:
    mol = Chem.MolFromSmiles(smi)
    if mol is not None:
        mols.append(mol)

Draw.MolsToGridImage(mols, molsPerRow=3, subImgSize=(500,400))

```

Out[3]:



REINVENTについて簡単に説明

元論文(Molecular De Novo Design through Deep Reinforcement Learning)を読みましょう。

(要説明)