ゼミ
ディレクトリ構成のすすめ

人工知能研究室 M1
奥出 拓生

今回の話題について

研究の内容は話しません！

- 説明できるほどの進捗成果がないので……

……なので、進捗の中身は一旦おいといて、

コードのディレクトリ構成について、前から考えていることを発表します

目次

1. 他人に見せられるコードが書けない！

2. 現在の運用

3. ぼくがかんがえたさいきょーのディレクトリ構成

1. 他人に見せられるコードが書けない！

コードを書いてて陥りがちなこと

- (僕) “処理”は書いてるんだけど、増えるとゴチャゴチャしてくる……
- (K氏) O氏が「処理を分けろ」って言ってたけど、どう分けたら良いのか全くわからない
- (H氏) Python わからない
- (N氏) ディープラーニングやりたい

—— “処理を分ける”とは？

アンチパターン (1)

```
import ...

if __name__ == '__main__' :

    # 以下 数百行以上 処理が続く
```

みたいなコード

- 実験的に数十行程度から処理を書き始めるのは全然ありだと思います、手をすぐに動かせるのは良き
- ……それ、だんだん長くなってませんか？

(個人的には一画面に収まる程度が許容限界かと)

アンチパターン (2)

```
import ...

def method_1(...):
    ...
    return ...

def method_2():

# 以下 1ファイル内に 関数がたくさん

def method_99():

if __name__ == '__main__' :
    ...
```

みたいなコード

- 関数(やクラス)を使って処理・関心の対象を分割するのは良き
- でも、1ファイルの中にたくさん増えてくると……
- 全体で1000行とかあると長くないですか？

アンチパターンのようなコードを書いている人へ

- こうやって書いている人を批難するつもりは全くないです！
  - 自分自身はどこに何を書いてあるか(意外と?)把握していられるんです
  - 皆さん、研究室に配属される・大学院に来るくらいには頭が良いので！

※ 自分がよく触れている間はね。

“あとで見返すと意味不明”

つまり…？

- 昨日のコードは読めますか？
- 先週のコードは読めますか？
- 1ヶ月前のコードは…
- 数ヶ月前の…
- 半年…
- 1年…

- 「あとから見返しても」≒「他人が見ても」わかるコードを書いていますか (いつから)？
- 僕は実験やプログラミング演習でも、あとから見返して理解できるように努力してました！(Javaの言語仕様等忘れかけてますが)

休憩1: NECのインターンシップに行ってきました

深層学習を用いたNECの製品で、次のバージョンに使いたい機能のテストと評価を行いました。

見せてもらったコードは2種類

- 製品版
- プロトタイプ版 (性能評価用に個人が使っていたコード、規模は数千行程度。こちらに一部実装していました)

備考: RAPID機械学習 (https://jpn.nec.com/rapid/)

休憩2: VBAを触ったことがありますか？

- 僕はあります: 4年ほど前に音ゲーのスコア解析してました
- 行数カウントしたら、空白行・コメント行を含んで全体で6,827行ありました (コード行は4〜5,000行程度？)
- 1ファイルで最大2200行くらいのやつがありました……

ちなみに、**言語仕様が超古い！！！！！**言語です。

- 変数のスコープしっかりしてない → グローバル変数
- 継承できないクラス → コピペ (!?!?)

で対応していました (そりゃあ行数増えるよね)

数千行の規模のコードに対するイメージ

コードをしっかり切り分けなくても、なんとか自分がやってる間だけなら把握していられるんだろうなあ……

- NECのインターンで見たコード→「僕が見ても理解するのに時間がかかりそう」
- 自分が書いたVBAのコード→「あれはもう絶対に読みたくない」

“未来の自分が詰む”

問題点: まとめ

- 最初はちょろっとメインの処理だけ書いている → 関数定義やクラスが増えてくると、どこになにがあるのか把握できなくなる
- 書いてる間は覚えていられても、忘れた頃の将来の自分、すなわち他人が理解できないコードになる

↓

わかりやすいディレクトリ構成にしたい！

機械学習モデルの開発における問題点(3/3)

今回扱う課題は、以下の3つとなる。

- 開発時: 原因の特定が難しい
  - 学習がうまくいかないなどの問題が発生した際に、その原因を特定するのが難しい
- 学習時: コード上で管理できない依存が発生する
  - 学習させる時だけ指定するパラメーターが数多くあり、それらが(モデルの再現に必要な情報にもかかわらず)どこにも残らない。
- 配置時: 処理に必要な静的ファイルが多い
  - 予測処理を行う際、機械学習モデルだけでなく前処理のパラメーターを保存したファイルや入力/出力を変換するための辞書など、多くのファイルに依存する。

Copyright © TIS Inc. All rights reserved.

機械学習で泣かないためのコード設計 2018 (https://www.slideshare.net/takahirokubo7792/2018-97367311) より

原因の特定が難しい(2/6)

機械学習モデルの開発では、うまく動かない際にその原因として考えられる要素が数多くある。

- モデルに原因がある
  - 選択したモデルが問題に適していない
  - モデルの構成(レイヤ構成など)に問題がある
- データに原因がある
  - データの量、あるいは質に問題がある
  - 前処理に問題がある
- 学習方法に原因がある
  - 目的関数、初期化方法、学習率、オプティマイザとその設定etc
  - 辛抱が足りない(明日になれば学習が進んでいるかも)
  - 蜃気楼の学習結果(あの時はたまたま収束した)
- その他
  - 実装に使っているフレームワークの差異

Copyright © TIS Inc. All rights reserved.

機械学習で泣かないためのコード設計 2018 (https://www.slideshare.net/takahirokubo7792/2018-97367311) より

原因の特定が難しい: 実例(5/6)

実装に使っているフレームワークの差異の例(Keras vs TensorFlow)

正規乱数で初期化を行う際の、平均/分散のデフォルト値がフレームワーク間で異なる。

```
class RandomNormal(Initializer):
    """Initializer that generates tensors with a normal

    # Arguments
        mean: a python scalar or a scalar tensor. Mean of the random values
          to generate.
        stddev: a python scalar or a scalar tensor. Standard deviation of the
          random values to genera
        seed: A Python integer. U
    """

    def __init__(self, mean=0., stddev=0.05, seed=None):
        self.mean = mean
        self.stddev = stddev
        self.seed = seed
```

平均0/分散0.05

```
class RandomNormal(Initializer):
  """Initializer that generates tensors with a normal distr

  Args:
    mean: a python scalar or a scalar tensor. Mean of the random values
      to generate.
    stddev: a python scalar or a scalar tensor. Standard deviation of the
      random values to generate.
    seed: A Python integer. Used to create random seeds. See
      @{tf.set_random_seed}
      for behavior.
    dtype: The data type. O
  """

  def __init__(self, mean=0.0, stddev=1.0, seed=None, dtype=dtypes.float32):
    self.mean = mean
    self.stddev = stddev
    self.seed = seed
    self.dtype = _assert_float_dtype(dtypes.as_dtype(dtype))
```

平均0/分散1

kerasで使う時とtf.kerasで使う時で挙動が異なる罠

あれ？論文通り/公開実装と同等に実装をしたのに？と思った際はこうしたところが原因の可能性もある(他にもPyTorchのLSTMにおけるforget biasの初期値などいろいろある)。

Copyright © TIS Inc. All rights reserved.

機械学習で泣かないためのコード設計 2018 (https://www.slideshare.net/takahirokubo7792/2018-97367311) より

コード上で管理できない依存が発生する(2/3)

学習の結果生成される「機械学習モデル」は、ソースコード以外の要素への依存を内包することになる。

- 学習に使ったデータ
- 学習時のハイパーパラメーター

```
> model.train --dataset=./data/faces/20180101 --epoch 25 --
lr=0.0002 --batch_size=64
```

つまり、コードがあるからといって機械学習モデルが再現できるとは限らない。同じコード(モデル)でも、実行条件により精度は大きく異なる。

Copyright © TIS Inc. All rights reserved.

機械学習で泣かないためのコード設計 2018 (https://www.slideshare.net/takahirokubo7792/2018-97367311) より

処理に必要な静的ファイルが多い(2/2)

機械学習モデルを利用して予測を行う際は、多くの静的ファイルに依存する。

- モデルファイル
  - 学習の結果生成されたモデルファイル
- 前処理のパラメーター(データの平均/分散など)
  - 学習時に前処理をしていれば、予測時も前処理が必要。
- 入力/予測結果を変換するための辞書
  - モデルへの入力/モデルからの出力は数値(0,1,2…)なので、数値が表す意味を変換する辞書(0=>猫、1=>犬など)が必要。

これらの静的ファイルはサイズが非常に大きい場合もあり(数Gなど)、ソースコードと同じリポジトリに含めて管理することが難しい。

Copyright © TIS Inc. All rights reserved.

機械学習で泣かないためのコード設計 2018 (https://www.slideshare.net/takahirokubo7792/2018-97367311) より

機械学習モデルの開発における問題点

機械学習は「機械」だけで完結しない

機械=モデルを定義したソースコード、モデルファイル

機械学習モデルの開発における問題点を解決するには、「学習」はもちろんそれに使用されるデータやパラメーターなど、機械学習を取り巻く要素を含めて設計を行う必要がある。

Copyright © TIS Inc. All rights reserved.

機械学習で泣かないためのコード設計 2018 (https://www.slideshare.net/takahirokubo7792/2018-97367311) より

じゃあどうする？

学習や予測のコードの分離だけでなく、使用するデータ、ハイパーパラメーター、(その他にも生成されるモデル、結果のグラフや表、ログ) などなど

をうまく扱えるようなディレクトリ構造、一度考えてみませんか…… ？

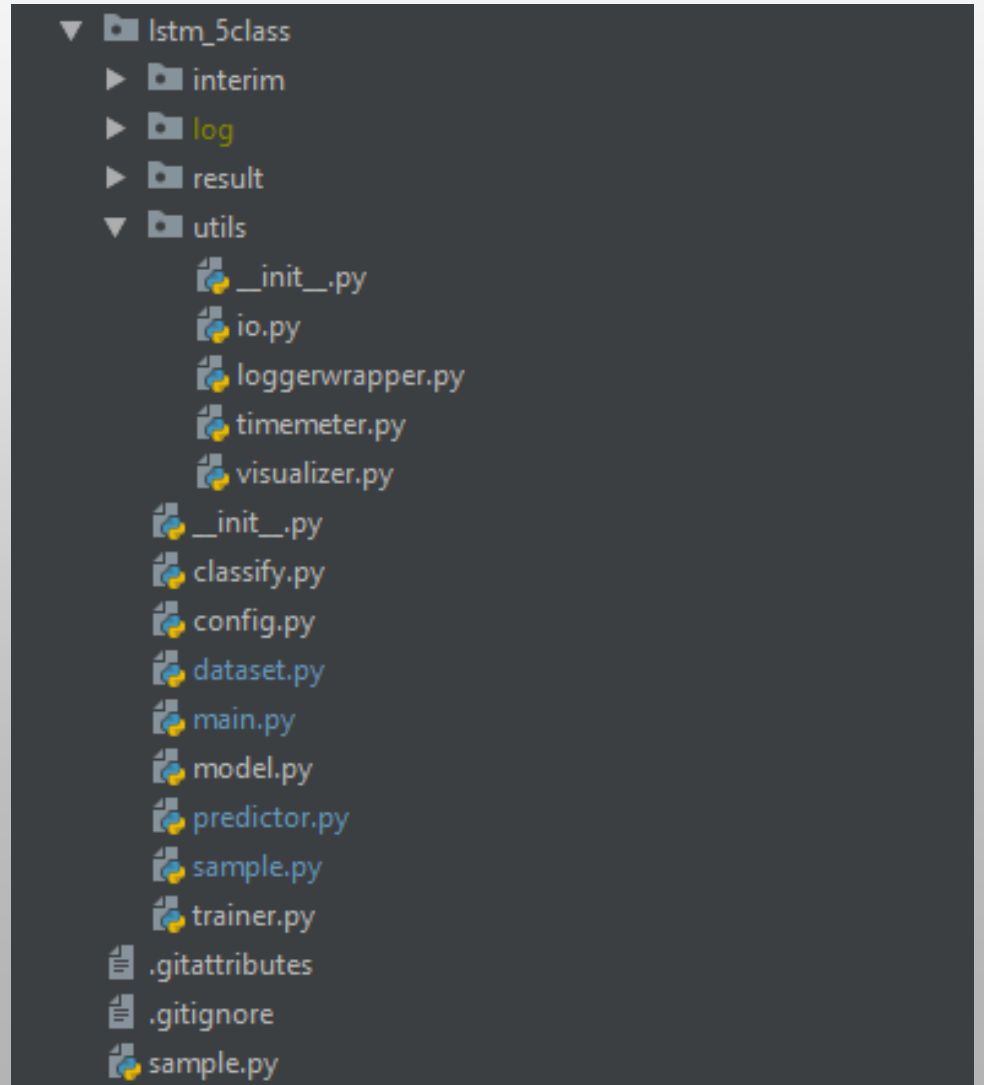
- 現在、和音認識に向けての下処理のコードを書いているのですが、ちゃんとディレクトリ構造考えないと死にそうです

  音程を表現するクラス、WAVファイル読み書き用ラッパー などなど

2. 現在の運用

いまのディレクトリ構成 (異常検知のやつ)

```
(root)/
    ├ interim/
    ├ log/
    ├ result/
    ├ utils/
    │   ├ __init__.py
    │   ├ io.py
    │   ├ ⋮
    │   └ visualizer.py
    ├ classify.py
    ├ config.py
    ├ dataset.py
    ├ main.py
    ├ model.py
    ├ predictor.py
    └ trainer.py
```



いまのディレクトリ構成 (異常検知のやつ)

```
(root)/
    ├ interim/          (学習途中のモデル)
    ├ log/              (ログ)
    ├ result/           (結果: 学習済モデルやグラフ、csvデータなど)
    ├ utils/            (よく使う処理)
    │   ├ __init__.py
    │   ├ io.py
    │   ├ ⋮
    │   └ visualizer.py
    ├ classify.py       (異常検知のクラス分類を行うコード)
    ├ config.py         (設定)
    ├ dataset.py        (データセット生成: 今回は連続する数値データと正解の配列)
    ├ main.py           (trainer + predictor + classify)
    ├ model.py          (モデルの定義、生成 or 読込: 今回はLSTMなので短い)
    ├ predictor.py      (予測を行うコード)
    └ trainer.py        (学習を行うコード)
```

各ディレクトリ・ファイルの詳細 (1)

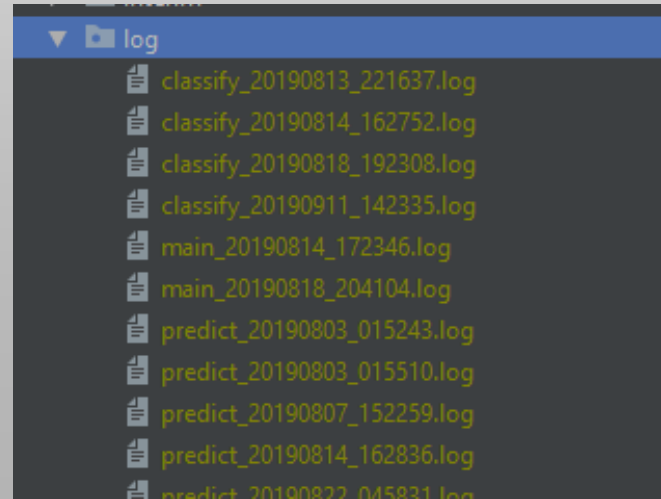
```
(root)/
    ├ interim/
    ├ log/
    ├ result/
    ├ utils/
    │   ├ __init__.py
    │   ├ io.py
    │   ├ ⋮
    │   └ visualizer.py
    ├ classify.py
    ├ config.py
    ├ dataset.py
    ├ main.py
    ├ model.py
    ├ predictor.py
    └ trainer.py
```

interim/

- 一定エポックごとに学習途中のモデルをログとして保存
- 意図せず終了しても (Windows Update とか)、ここから読み込んで途中から再開できる

log/

- 実行時のログを保存 →

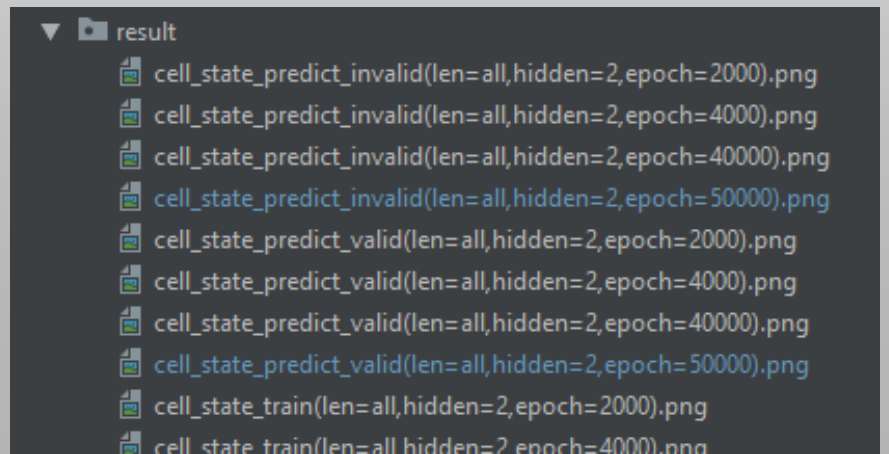


各ディレクトリ・ファイルの詳細 (2)

```
(root)/
    ├ interim/
    ├ log/
    ├ result/
    ├ utils/
    │    ├ __init__.py
    │    ├ io.py
    │    ├ ⋮
    │    └ visualizer.py
    ├ classify.py
    ├ config.py
    ├ dataset.py
    ├ main.py
    ├ model.py
    ├ predictor.py
    └ trainer.py
```

result/

- 学習済モデルやグラフ、csvデータなどの結果をすべて保存
  - 図は matplotlib.pyplot で作成
  - lossは np.savetxt
  - モデルは torch.save



各ディレクトリ・ファイルの詳細 (3)

```
(root)/
    ├ interim/
    ├ log/
    ├ result/
    ├ utils/
    │   ├ __init__.py
    │   ├ io.py
    │   ├ ⋮
    │   └ visualizer.py
    ├ classify.py
    ├ config.py
    ├ dataset.py
    ├ main.py
    ├ model.py
    ├ predictor.py
    └ trainer.py
```

utils/

- よく使う処理、機械学習の本筋とは異なる処理についてのファイル
  - io.py: ファイル・ディレクトリ操作
  - loggerwrapper.py: ログ出力 *+ 保存*
  - timemeter.py: 時間測定
  - visualizer.py: グラフ作成 *+ 出力*

各ディレクトリ・ファイルの詳細 (4)

```
(root)/
    ├ interim/
    ├ log/
    ├ result/
    ├ utils/
    │   ├ __init__.py
    │   ├ io.py
    │   ├ ⋮
    │   └ visualizer.py
    ├ classify.py
    ├ config.py
    ├ dataset.py
    ├ main.py
    ├ model.py
    ├ predictor.py
    └ trainer.py
```

utils/ (続き)

- 1ファイル中は基本1クラスのみ or 関数のみ
- __init__.py を下のように書くと参照で楽できます

  例えば io.py の `save_plt` 関数の参照時に `utils.io.save_plt` と書くべきところを `utils.save_plt` と書ける…のですが、説明は省略

```
from .io import *
from .loggerwrapper import *
from .timemeter import *
from .visualizer import *
```

各ディレクトリ・ファイルの詳細 (5)

```
(root)/
    ├ interim/
    ├ log/
    ├ result/
    ├ utils/
    │   ├ __init__.py
    │   ├ io.py
    │   ├ ⋮
    │   └ visualizer.py
    ├ classify.py
    ├ config.py
    ├ dataset.py
    ├ main.py
    ├ model.py
    ├ predictor.py
    └ trainer.py
```

config.py

- モデルのパラメータ(入力・出力の次元数)やハイパーパラメーターなどを設定するクラスを格納

```python
class ModelConfig:

    def __init__(self, resource: ResourceConfig):
        self.input_dim: int = 1
        self.hidden_dim: int = 2
        self.output_dim: int = resource.states
        self.epoch: int = 50000
        self.pre_epoch: int = 0
        self.progress_epoch: int = 20
        self.save_epoch: int = 500
        self.n_components: int = 5
```

各ディレクトリ・ファイルの詳細 (6)

```
(root)/
    ├ interim/
    ├ log/
    ├ result/
    ├ utils/
    │   ├ __init__.py
    │   ├ io.py
    │   ├ ⋮
    │   └ visualizer.py
    ├ classify.py
    ├ config.py
    ├ dataset.py
    ├ main.py
    ├ model.py
    ├ predictor.py
    └ trainer.py
```

dataset.py

- データセット (今回の場合は連続する数字の羅列) を生成する関数を格納

(関係ない話なのですが、静的型付け大好きなので、Pythonでも引数や返り値の型を明示しましょう派です)

```python
def make_data_all(res: ResourceConfig, *,
                  logger: LoggerWrapper = None) -> (torch.Tensor, torch.Tensor, np.ndarray, np.ndarray):
    index_dict = {}
    for l in range(res.min * res.states, res.max * res.states + 1):
        index_list = []
        for i in range(0, l):
            index_list.append([[(n % l) for n in range(i, i + res.seqlen)], (i + res.seqlen) % l])
        index_dict[l] = index_list
    …
```

各ディレクトリ・ファイルの詳細 (7)

```
(root)/
    ├ interim/
    ├ log/
    ├ result/
    ├ utils/
    │   ├ __init__.py
    │   ├ io.py
    │   ├ ⋮
    │   └ visualizer.py
    ├ classify.py
    ├ config.py
    ├ dataset.py
    ├ main.py
    ├ model.py
    ├ predictor.py
    └ trainer.py
```

trainer.py

- モデルの学習部分のコードを記述

predictor.py

- モデルの予測部分のコードを記述 (visualizer.py の関数を呼んでグラフの出力したり)

各ディレクトリ・ファイルの詳細 (8)

```
(root)/
    ├ interim/
    ├ log/
    ├ result/
    ├ utils/
    │   ├ __init__.py
    │   ├ io.py
    │   ├ ⋮
    │   └ visualizer.py
    ├ classify.py
    ├ config.py
    ├ dataset.py
    ├ main.py
    ├ model.py
    ├ predictor.py
    └ trainer.py
```

~~classify.py~~ → classifier.py

- 学習したモデルにおけるLSTMの内部状態から、GMMで分類を行う処理を分離して記述

main.py

- このプロジェクトでやりたいこと全体を実行 (train.py, classify.py, predictor.py を呼んでいるだけ)

運用してみてわかってきた問題点

```
(root)/
    ├ interim/
    ├ log/
    ├ result/
    ├ utils/
    │    ├ __init__.py
    │    ├ io.py
    │    ├ ⋮
    │    └ visualizer.py
    ├ classify.py
    ├ config.py
    ├ dataset.py
    ├ main.py
    ├ model.py
    ├ predictor.py
    └ trainer.py
```

- どの設定でのリザルトかわからなくなる (ファイル名に設定を書く今のやり方はナンセンス)
- モデルもグラフも同じ場所に保存してて中身がゴチャゴチャ

- いろいろな条件で実験したいのに設定を保存しておけない

(今回はデータを全部自分で生成しているので)
データセットの置き場所を考えていない

3. ぼくのかんがえたさい きょーのディレクトリ構成

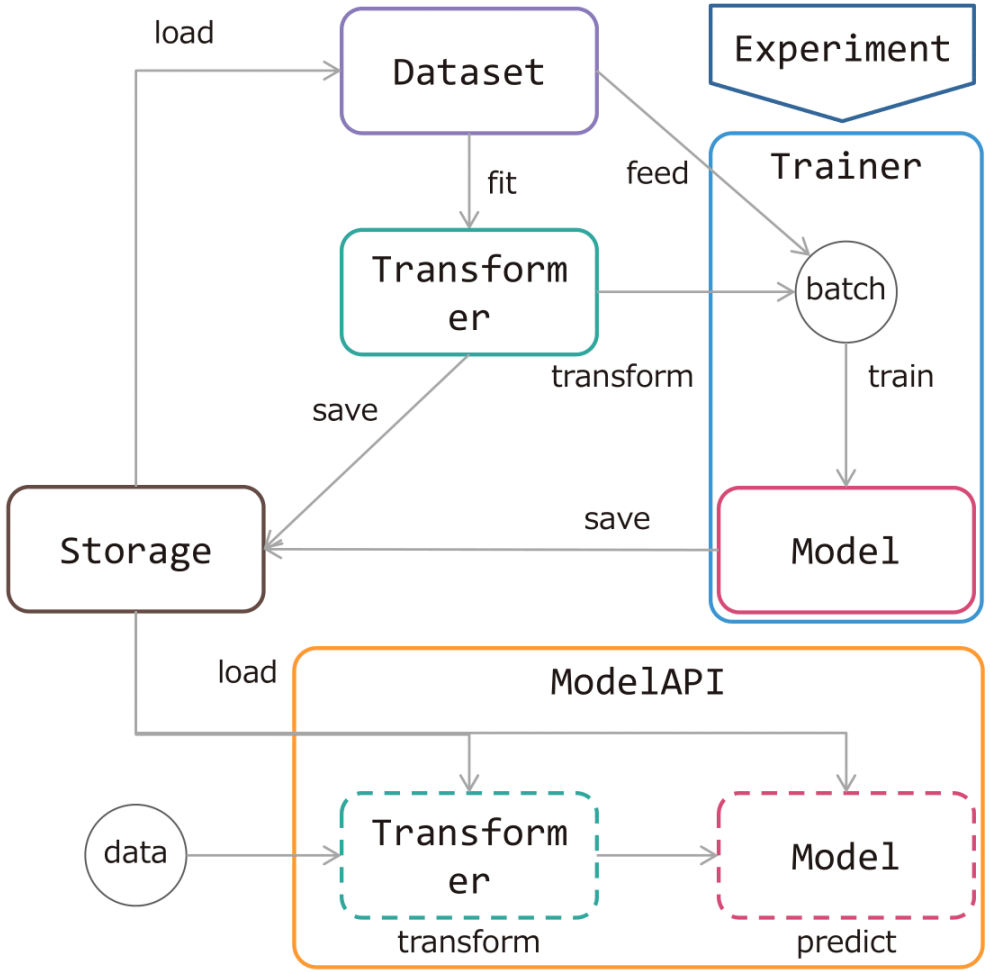
参考にしたサイト

- ① 機械学習で泣かないためのコード設計 2018
  (https://www.slideshare.net/takahirokubo7792/2018-97367311)
- ② データ分析をするときのフォルダ構成をどうするのか問題について
  (https://www.st-hakky-blog.com/entry/2017/03/24/140738)
- ③ Pythonの機械学習プロジェクトにおけるプログラミング設計
  (https://qiita.com/mokemokechicken/items/c9c0b042f3b1a7c383db)

とりあえず見てみましょう…

①

構成(2018年版)



- Dataset
  データセットの取得を行う
- Transformer
  前処理/後処理を行う
- Trainer
  モデルの学習を行う
- Model
  機械学習モデルを定義する
- ModelAPI
  モデルによる予測を行う
- Experiment
  学習条件を記述する
- Storage
  ファイルの配置を管理する

Copyright © TIS Inc. All rights reserved.

機械学習で泣かないためのコード設計 2018 (https://www.slideshare.net/takahirokubo7792/2018-97367311) より

構成のポイント

紹介した構成は、基本的にはソフトウェア設計の基本原則である「単一責任原則」に則り設計されている。

　機械学習モデルの開発もソースコードを通じて行われるものであり、既存のソフトウェア設計論は十二分に通用する。

むしろ、「機械学習だから」という形で特別扱いしないことが重要。

- 機械学習だから1ファイルに全部処理をまとめていい
- 機械学習だからグローバル変数を気軽に使っていい
- 機械学習だから単体テストできない

…ということはない。システム開発者として「おかしい」と感じたら、それは本当におかしいので、対策を考える。

Copyright © TIS Inc. All rights reserved.

機械学習で泣かないためのコード設計 2018 (https://www.slideshare.net/takahirokubo7792/2018-97367311) より

②

調べてみた結果

ということで、調べてみました。その結果は以下。

- もう、悩まなくていい
- pipで一発インストール。
- テンプレートをコマンド一発でジェネレートできる

最高ですね。

調べて行き着いたところ

見つけました。もう少しなんとかしたいと思った点を克服しているサイトが！！

テンプレートを使って爆速で機械学習プロジェクトを作成する - Qiita
あなたは機械学習のプロジェクトを毎回違う構成で作っていませんか？ 何をどこに配置するかで悩んで時間がかかっていませんか？
<!-- <img width="630" alt="スクリーンショ...
qiita.com 90 users

qiita.com

このサイトに書いてある、

> あなたは機械学習のプロジェクトを毎回違う構成で作っていませんか？
> 何をどこに配置するかで悩んで時間がかかっていませんか？

すげー共感した。そして、「コマンド一発」で書けるなんて、、、以下のサイトがデータ分析で使用するテンプレートとして用意されているもの。

drivendata/cookiecutter-data-science
A logical, reasonably standardized, but flexible project structure for doing and sharing data science work. - drivendata/cookiecutter-data-science
github.com 5 users

github.com

フォルダの構成はご覧の通り。

データ分析をするときのフォルダ構成をどうするのか問題について
(https://www.st-hakky-blog.com/entry/2017/03/24/140738) より

フォルダの構成はご覧の通り。

```
├── LICENSE
├── Makefile           <- Makefile with commands like `make data`
├── README.md          <- The top-level README for developers usi
├── data
│   ├── external       <- Data from third party sources.
│   ├── interim        <- Intermediate data that has been transfo
│   ├── processed      <- The final, canonical data sets for mode
│   └── raw            <- The original, immutable data dump.
│
├── docs               <- A default Sphinx project; see sphinx-do
│
├── models             <- Trained and serialized models, model pr
│
├── notebooks          <- Jupyter notebooks. Naming convention is
│                         the creator's initials, and a short `-`
│                         `1.0-jqp-initial-data-exploration`.
│
├── references         <- Data dictionaries, manuals, and all oth
│
├── reports            <- Generated analysis as HTML, PDF, LaTeX,
│   └── figures        <- Generated graphics and figures to be us
│
├── requirements.txt   <- The requirements file for reproducing t
│                         generated with `pip freeze > requiremen
│
├── src                <- Source code for use in this project.
│   ├── __init__.py    <- Makes src a Python module
│   |
│   ├── data           <- Scripts to download or generate data
│   |   └── make_dataset.py
│   |
│   ├── features       <- Scripts to turn raw data into features
│   |   └── build_features.py
│   |
│   ├── models         <- Scripts to train models and then use tr
```

```
│   ├── models         <- Scripts to train models and then use tr
│   |                     predictions
│   ├── predict_model.py
│   └── train_model.py
│   |
│   └── visualization  <- Scripts to create exploratory and resul
│       └── visualize.py
|
└── tox.ini            <- tox file with settings for running tox;
```

私のよりも見通しが良くて、どこに何を書けばいいかわかりやすい、、、

しかも使用者が気をつければ、上で紹介した機械学習のコードで泣くポイントをほとんど回避できる気がしている。。。

というか、このプロジェクトの大本のサイトにあるこの記事で書かれている通り、フォルダ構成がほぼすべてを表現できるように設計されている。。。

このテンプレートを使用する方法も、コマンド一発、、、しかもパッケージもpipでインストールできるし最強かよ、、、

```
pip install cookiecutter
```

わーい！

後は、自分の使いたいテンプレート(上で紹介したテンプレート)を指定して、使用するだけという。

```
cookiecutter https://github.com/drivendata/cookiecutter-data-scie
```

後は指示に従って頑張るだけ。

データ分析をするときのフォルダ構成をどうするのか問題について
(https://www.st-hakky-blog.com/entry/2017/03/24/140738) より

③

モデルの切り替え

例えば、同じタスクでも、RNNを使うモデルやCNNを使うモデルがあります。
それらを切り替えたい場合に、設定だけで切り替えると、Modelの中にIF文が大量発生して、可読性が悪くなります。
また、Training時の工夫もモデルの構造で変わってくることが多いので、大変です。

モデルを切り替える場合、(Model, Trainer, ModelAPI)の３つを１つのセットとして増やしていくと、たぶん破綻せずにすみます。
例えば、

```
config.py
data_processor.py

model_cat/model.py
model_cat/trainer.py
model_cat/model_api.py

model_dog/model.py
model_dog/trainer.py
model_dog/model_api.py
```

というような構成にします。
※ model/trainer/model_api の基底クラスを作っておいても良いかもしれません。

こうしておけば、設定ファイルの `model_cat` を使うと書けばmodel_catを使うようにしておくことで、簡単にモデルを切り替えることができるようになります。

このような構成で1年以上運用していて、何パターンも新しいモデルを追加していますが、今のところ破綻せずに済んでいます。

Pythonの機械学習プロジェクトにおけるプログラミング設計
(https://qiita.com/mokemokechicken/items/c9c0b042f3b1a7c383db) より

プログラムの起動周り

ここは色々な方法がありそうですが、私は以下のような方法に決めています。

run.py

まず、プログラムの起動は `run.py` で始めるようにします。
だいたい以下のようなコードになります。

run.py

```
import os
import sys
from dotenv import load_dotenv, find_dotenv

if find_dotenv():
    load_dotenv(find_dotenv())

_PATH_ = os.path.dirname(os.path.dirname(__file__))  # これはソースコードのTop Directoryを指すようにする

if _PATH_ not in sys.path:
    sys.path.append(_PATH_)


if __name__ == "__main__":
    from myproject import manager
    sys.exit(manager.start())
```

以下の処理をしています。

- python-dotenv を呼び出して、環境変数の設定
- PYTHONPATHにプロジェクトコードのTop Directoryを注入
- 真の mainルーチンの呼び出し

この manager.py を別ファイルにしておくと、相対PATHによるimportができるので少し便利です。

- 真の mainルーチンの呼び出し

この manager.py を別ファイルにしておくと、相対PATHによるimportができるので少し便利です。
※ Pythonだと起動時に呼び出したファイルから `from .hoge import fuga` という相対指定のimportができない。

また、毎回毎回実行時に環境変数PYTHONPATHを指定したりしなくて済みます。

manager.py

例えば、以下のようなコードになります。
https://github.com/mokemokechicken/reversi-alpha-zero/blob/2a48aeccfc79038bc153518f1da36ec3ec7e50ec/src/reversi_zero/manager.py

- ArgumentParser で 引数の処理
- Config インスタンスの作成
- ログのConfiguration
- 起動時にやっておくと良い色々な処理
- 実際に行いたいコマンドに応じて、Configを引数とした関数の呼び出し。

などを行います。

他のプログラムから呼び出しやすい？

一応、YAMLファイルを一時ディレクトリに書き出して、引数で与えることでプログラムの振る舞いを変えることができます。
入力や出力も、その一時ディレクトリに書き出すようにそのYAMLファイルで指示しておけば、
並列に独立に実行することが一応できます。

Dockerなどのコンテナに入れて実行する場合も大きな問題はないです（そりゃまあ、そうか）。

Pythonの機械学習プロジェクトにおけるプログラミング設計
(https://qiita.com/mokemokechicken/items/c9c0b042f3b1a7c383db) より

調べてみて思ったこと

- 僕、案外間違ったことしてないじゃん……！(ちょっと安心しました笑)
- 僕が求めている必要最低限ぴったりを満たすディレクトリ構成はなかったが、僕の想定以上にいろいろなことを考えている構成はあった → それを使うのがラク
  (素直に `pip install cookiecutter` していいのでは。`anaconda search` でも見つかったのでいけます)

それはそれとして、自分でももう少し考えてみました

ぼくのかんがえたさいきょーのディレクトリ構成: 全体図

```
(root)/
    ├ config/
    │    ├ config1.py
    │    └ ⋮
    ├ data/
    │    ├ external/
    │    ├ interim/
    │    ├ processed/
    │    └ raw/
    ├ interim/
    ├ log/
    ├ result/
    │    ├ config1/
    │    └ ⋮
    ├ src/
    │    ├ model1/
    │    │    ├ layers/
    │    │    │    └ __init__.py
    │    │    │    └ ⋮
    │    │    ├ __init__.py
    │    │    ├ config.py
    │    │    ├ dataset.py
    │    │    ├ model.py
    │    │    ├ predictor.py
    │    │    └ trainer.py
    │    ├ model2/
    │    ├ ⋮
    │    └ utils/
    │         ├ __init__.py
    │         └ ⋮
    ├ sample.py
    └ main.py
```

現在の運用と異なる部分

```
(root)/
├ config/
│   ├ config1.py
│   └ ⋮
├ data/
│   ├ external/
│   ├ interim/
│   ├ processed/
│   └ raw/
├ interim/
├ log/
├ result/
│   ├ config1/
│   └ ⋮
├ src/
│   ├ model1/
│   │   ├ layers/
│   │   │   └ __init__.py
│   │   │   └ ⋮ (モデル内の各層の.py)
│   │   ├ __init__.py
│   │   ├ config.py (デフォルト設定用)
│   │   ├ dataset.py
│   │   ├ model.py
│   │   ├ predictor.py
│   │   └ trainer.py
│   ├ model2/
│   ├ ⋮
│   └ utils/
│       ├ __init__.py
│       └ ⋮
├ sample.py (試しに何か動かす用)
└ main.py
```

- 設定ファイルごとに保存
- データの置き場所を確保
- 設定ファイルごとに同名ディレクトリに結果を格納
- モデルごとにディレクトリを分割して処理を格納
- (層が複雑になるときは)各層ごとにファイルを分割して格納

どこから何を持ってきたか

```
(root)/
    ├ config/
    │    ├ config1.py
    │    └ ⋮
    ├ data/
    │    ├ external/
    │    ├ interim/
    │    ├ processed/
    │    └ raw/
    ├ interim/
    ├ log/
    ├ result/
    │    ├ config1/
    │    └ ⋮
```

← ②から

```
├ src/
│    ├ model1/
│    │    ├ layers/
│    │    │    └ __init__.py
│    │    │    └ ⋮
│    │    ├ __init__.py
│    │    ├ config.py
│    │    ├ dataset.py
│    │    ├ model.py
│    │    ├ predictor.py
│    │    └ trainer.py
│    ├ model2/
│    ├ ⋮
│    └ utils/
│         ├ __init__.py
│         └ ⋮
├ sample.py
└ main.py
```

← ②から
← ③から

← ②から

← ①, ③から
ModelAPI は predictor
Transformer は未採用(?)

この構成で相対インポートすると？

- trainer.py や predictor.py から utils 内
  例) `from ..utils import *`

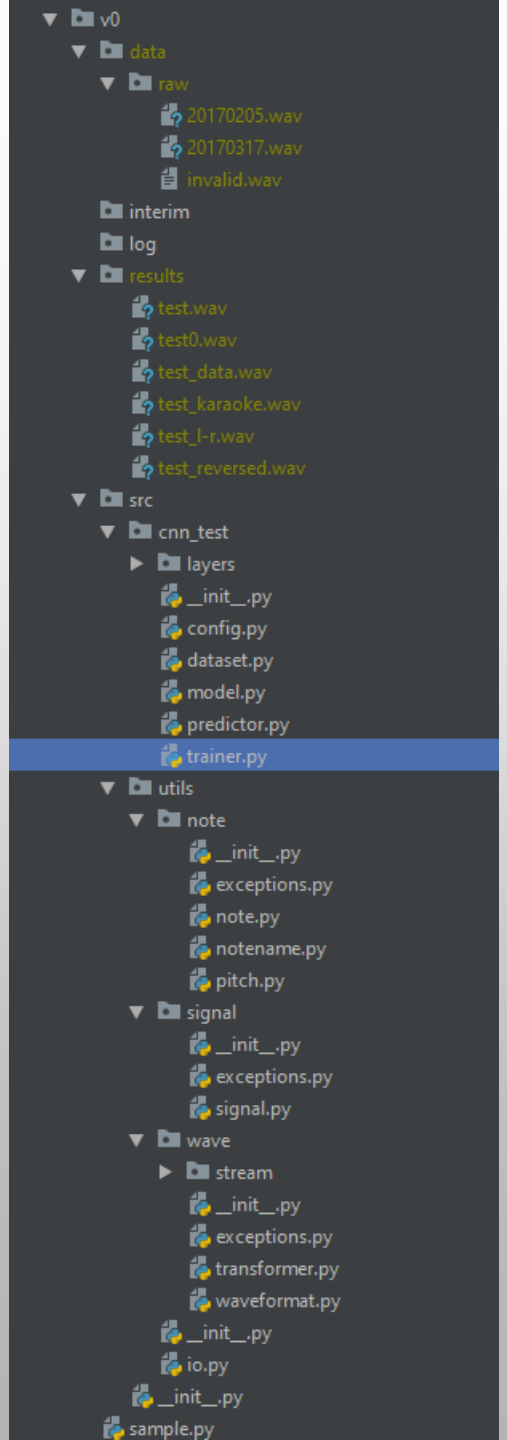
- config/config1.py から 各モデルのデフォルト Config
  例) `from ..src/model1 import Config`

- model.py から各層のクラス
  例) `from .layers import Layer1`

実際にやってみています

- 和音認識のプロジェクトのディレクトリ構成です
- モデル名は cnn_test ですが、中身はまだすべて空です(!) (なので進捗がありません！)
- src/utils/ 内の実装をしています (あんまり進んでないです、がんばります)

utils 内が巨大になりそう



この構成の不安要素について

- テストなんて知らない (アプリケーションを作るわけじゃないのでまあいらないでしょう)
- utilsによく使うクラスや関数をまとめるのって良い慣習なのでしょうか…？(何も考えずにテキトーに放り込めるので使ってしまっています)
- config/ ……、result/ ……、(ここらへんは触り始めたときにすぐ「やっぱやめた！」となる可能性がありそう)

とりあえず手を動かします！

さいごに

- 個人的な指針にしたい構成をお話ししました
- こういう部分、他のM1, M2, 卒業した先輩方がみんなどうしてるのか、もっと知りたいです
- 半分くらいは以前から考えていたことですが、半分くらいは一昨日と昨日考えたことです！(詰めが甘い)
- 「ここをこうした方がいいよ」などありましたらツッコミお待ちしています
- もっと他に考えなきゃいけないことある気がする

補足1) 以前に僕が書いたVBAのコード

遊び感覚だったのですがなんと
「<u>変数名を日本語にする</u>」
という縛りでやってました。
わかりやすいようでわかりにくいです。

- If (日本語) = (日本語) Then とか読みにくい
- 変数名の横幅が長くなりがち(僕の変数名の付け方も悪いですが)

```
Dim 切替可能性 As Boolean
Dim 行 As Long
Dim マーク As Long
For 行 = 開始行 To 終了行
    Def.譜面テーブル.再計算 行, , 自動再計算

    ' 現在の比較対象の行スコアより一定以上スコアが落ちた場合は終了
    Dim 現在スコア As Long
    現在スコア = 現在行スコア取得(行)
    If 現在ルート行スコア(行) > 0 And 現在スコア > 0 Then
        If 現在スコア < 現在ルート行スコア(行) + Def.スコアタスキップ点 Then
            Exit For
        End If
    End If

    '完奏モードでない場合、ライフが0なら終了
    If (Not 完奏モード) And Def.譜面テーブル.ライフ列(行) = 0 Then
        Exit For
    End If

    '50コンボ以上でライフが最大なら、以前に設定された現在ルート行スコアと同じスコア上昇ペースになるので終了
    If Not (Def.スコアタテーブル.Is簡易版) And 影響開始行 > 0 Then
        If Def.譜面テーブル.コンボ列(行) >= 50 And Def.譜面テーブル.ライフ列(行) = Def.最大ライフ量 Then
            Exit For
        End If
    End If

    ' ルート開始位置になれるかどうかを判定
    isルート開始行 = False
    For マーク = 1 To Def.マーク数
        If Def.譜面テーブル.ノーツ列(マーク, 行) = Def.HOLD文字 Then
            isホールド開始(マーク) = True
            isルート開始行 = True
        Else
            isホールド開始(マーク) = False
        End If
    Next マーク

    If isルート開始行 Then
        ' ルート開始位置になれる場合の処理

        ' 開始行の切替設定(切替は強制変更)
        切替可能性 = False
        For マーク = 1 To Def.マーク数
            If Def.譜面テーブル.ホールド個別開始フレーム列(マーク, 行 - 1) > 0 Then
                切替可能性 = True
            End If
        Next
        If 切替可能性 And Not (Def.譜面テーブル.ホールドボーナス列(行) > 0) Then
            Def.譜面テーブル.切替判定列(行) = True
        End If

        ' 直前ブロックの切替と早遅の再設定
        切替再計算開始行 = 指定行直前までの切替再計算(開始行, 行, 画面更新, 自動再計算)

        ' 開始行となる現在の行のフレーム前ずれ許容フレームを設定して、早遅フレームと評価をリセット
        ' ※直前がC-SdでMAXのルートなど、最早COOLでMAXにならない場合はリセットしない
        開始早ずれ許容フレーム = Def.譜面テーブル.デフォルト早ずれ許容フレーム
        If Def.譜面テーブル.ホールドボーナス列(行) > 0 Then
            ' MAXが入る場合のみデフォルトから変更
            If Def.譜面テーブル.ホールドフレーム列(行) > Def.未MAXフレーム最大値 Then
                開始早ずれ許容フレーム = Application.WorksheetFunction.Max(開始早ずれ許容フレーム,
                    Def.未MAXフレーム最大値 - Def.譜面テーブル.ホールドフレーム列(行) + Def.譜面テーブル.早遅フレーム列(行) + 1)
            End If
        End If

        早遅フレームと評価の解除 行, , Not (開始早ずれ許容フレーム > Def.譜面テーブル.最早COOLフレーム)
        Def.譜面テーブル.再計算 行, , 自動再計算

        '現在行の次の行からのルート探索
        If 影響開始行 = 0 Then
            指定行からのルート探索 行, 終了行, isホールド開始, 対象ルート行スコア.行スコア, 開始早ずれ許容フレーム, , 完奏モード, 画面更新, 自動再計算, 切替再計算開始行
        Else
            指定行からのルート探索 行, 終了行, isホールド開始, 対象ルート行スコア.行スコア, 開始早ずれ許容フレーム, , 完奏モード, 画面更新, 自動再計算, 影響開始行
        End If

        '切替と早遅を元に戻す
        Def.早遅切替一覧テーブル.早遅切替情報書き出し Def.譜面テーブル, 切替再計算開始行, 行, False
        Def.譜面テーブル.再計算 切替再計算開始行, 行, 自動再計算

    End If

Next 行

' 戻り値を設定
指定範囲のルート判定 = 対象ルート行スコア

' 早遅リセットはなくても大丈夫?

#If スコアタ詳細ログ Then
    詳細ログ出力 "指定範囲のルート判定" & vbTab & 開始行 & vbTab & 終了行 & vbTab & "終了", True
```

補足2) 別の視点から解決してみる

- そもそもコードをたくさん書かない！
  - ちょっと試すときとか簡単に書ける Keras 楽ですよね
  - もしくは関心の対象を狭める (モデルの学習部分のコードしか書かないぞ！…？)

- GitHubからとってきたコードをベースにする or 先輩からコードをもらう
  - しっかりした構成のものならそれをベースにして、自分に合うようにアレンジしていく！ラク！…？