Hands on Machine Learning with Scikit-Learn &TensolFlow

Chapter 2

End-to-end Machine Learning project

Created by Yusuke FUJIMOTO

はじめに

- この資料は「Hands-On Machine Learning with Scikit-Learn and TensorFlow - O'Reilly Media」を読んだ際の（主にソースコードに関する）簡単な解説を残したものです。
- 全部を解説したわけではないので注意
- 余裕があればソースコード周りの背景知識もまとめたい
- 何かあったら yukkyo12221222@gmail.com まで

Chapter 2

End-toend Machine Learning project

Get the data

```
import os
import tarfile
from six.moves import urllib

# ダウンロード元
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/"
                        + "ageron/handson-ml/master/"
# os 依存したパス構造を考慮
HOUSING_PATH = os.path.join("datasets", "housing")
# 実際にダウンロードするファイル名
HOUSING_URL = DOWNLOAD_ROOT +
                  "datasets/housing/housing.tgz"
```

Get the data（続き）

```
def fetch_housing_data(housing_url=HOUSING_URL,
                       housing_path=HOUSING_PATH):
    # ディレクトリが無かったら作る
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    # housing_url にアクセスして取ってきて tgz_path に配置
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    # tgz ファイルの展開
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

Split train data and test data

```
# For illustration only. Sklearn has train_test_split()
def split_train_test(data, test_ratio):
    # 0 から len(data) までの値をランダムに並べたもの
    shuffled_indx = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indx = shuffled_indices[:test_set_size]
    train_indx = shuffled_indices[test_set_size:]
    # python は一度に複数の値を返せる
    return data.iloc[train_indx], data.iloc[test_indx]
```

Split train data and test data

```
import hashlib

# この id のデータはテストデータか否かを返す
# 変数名が良くない -> is_test_data とかの方が良い
def test_set_check(identifier, test_ratio, hash):
    ident_int = np.int64(identifier) #  64bit整数に変換

    # 与えられた関数 hash を適用してダイジェスト（結果）表示
    hashed_obj = hash(ident_int).digest()

    # 最後の一文字を数値にしたもの。0から255までのばらけた数
    hashed_num = hashed_obj[-1]

    # 右辺の数より小さいか否か（True, False）を返す
    return hased_num < 256 * test_ratio
```

Split train data and test data

```
def split_train_test_by_id(data, test_ratio,
                           id_column, hash=hashlib.md5):
    ids = data[id_column] #  id とする列を指定
    # ids 内の各id に対して、テストデータか否かを求めた結果
    in_test = ids.apply(
                  lambda id_: test_set_check(
                                                id_,
                                                test_ratio,
                                                hash
                                    )
              )
    # トレーニングデータとテストデータを返す
    return data.loc[~in_test], data.loc[in_test]
```

Split train data and test data

- apply : それぞれに関数を適用する関数
- lambda : `lambda a: aの計算結果` は `a` を引数に `aの計算結果` を返す名前のない関数を表す
  - apply の中に入れる時などは便利
- ~ : ビット反転。True なら False になる
  - `~np.array([True, False, False])` の結果は `np.array([False, True, True])` となる

Split train data and test data

```
from sklearn.model_selection \\
     import StratifiedShuffleSplit

# クロスバリデーション用オブジェクト
splitter = StratifiedShuffleSplit(
            n_splits=1,      #  何回再シャッフルするか
            test_size=0.2,   #  テストデータ比率
            random_state=42 #  乱数シード
        )
# splitter.split（データ、ラベル）で分割結果を返す
split_result = splitter.split(housing,
                              housing["income_cat"])

# 各ラベルについて、なるべく test と train 両方に割り振る
for train_index, test_index in split_result:
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

Split train data and test data

言いたいこと：自作の train_test_split 関数（ランダムに振り分ける）だと内訳の比率が良くないよね

```
def income_cat_proportions(data):
    # 各内訳の比率を返す
    return data["income_cat"].value_counts() / len(data)

train, test = train_test_split(housing, test_size=0.2,
                               random_state=42)

# cp = compare_props
cp = pd.DataFrame({
    "Overall": income_cat_proportions(housing),
    "Stratified": income_cat_proportions(strat_test_set),
    "Random": income_cat_proportions(test),
}).sort_index() #  index 順に並べ替え
```

Prepare the data for ML algorithms

Imputer（欠損値の扱い方を決める）

```
from sklearn.preprocessing import Imputer
impute = Imputer(strategy="median") #  mean,most_frequent
```

LabelEncoder（数値に変換）

```
import numpy as np
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
sample = np.array(["a", "b", "c", "b", "c"])
sample2 = encoder.fit_transform(sample)
print(sample2)
# => array([0, 1, 2, 1, 2])
```

Prepare the data for ML algorithms

OneHotEncoder（OneHot ベクトルに変換）

```
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder()
sample3 = encoder.fit_transform(sample2.reshape(-1, 1))
print(sample3.toarray())
# [[ 1.  0.  0.]
#  [ 0.  1.  0.]
#  [ 0.  0.  1.]
#  [ 0.  1.  0.]
#  [ 0.  0.  1.]]
```

LabelBinarizer（0 か 1 の OntHot 表現に変換）

```
from sklearn.preprocessing import LabelBinarizer
LabelBinarizer().fit_transform(sample) #  上と同じ書き方もok
```

Prepare the data for ML algorithms

平均家族数等を計算して付け足す

```
from sklearn.base import BaseEstimator, TransformerMixin

# column index
rooms_ix, bedrooms_ix, population_ix, household_ix = \\
                                                    3, 4, 5, 6
```

- python では複数の変数に同時に代入できる

続き

```
## 一部省略（クラス定義部分とか）
def transform(self, X, y=None):
    # 世帯あたりの部屋数、家族人数の計算
    rooms_per_household = \\
        X[:, rooms_ix] / X[:, household_ix]
    population_per_household = \\
        X[:, population_ix] / X[:, household_ix]

    # フラグに応じて追加したデータを返す
    if self.add_bedrooms_per_room:
        bedrooms_per_room = \\
            X[:, bedrooms_ix] / X[:, rooms_ix]
        return np.c_[X, rooms_per_household,
                     population_per_household,
                     bedrooms_per_room]
    else:
        return np.c_[X, rooms_per_household,
                     population_per_household]
```

Prepare the data for ML algorithms

複数の（sklearn の）手続きを一連の流れにまとめる

[【翻訳】scikit-learn 0.18 User Guide 4.1. パイプラインとFeatureUnion：推定器の組み合わせ - Qiita]

```
# 複数の処理や分類をまとめる
from sklearn.pipeline import Pipeline
# 標準化（だいたい平均 0 分散 1 が多い）するため
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])
housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Prepare the data for ML algorithms

数値かカテゴリ列を選択するためのクラス

```
from sklearn.base import BaseEstimator, TransformerMixin

# Create a class to select
# numerical or categorical columns
# since Scikit-Learn doesn't handle DataFrames yet

# クラス定義のカッコ内は、継承元を表す
# うまく継承すると 先ほどの Pipeline 等に組み込める
class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values
```

Prepare the data for ML algorithms

数値かカテゴリ列を選択するためのクラス

```
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

# 実際に Pipeline に組み込んでいる例
num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

cat_pipeline = Pipeline([
        ('selector', DataFrameSelector(cat_attribs)),
        ('label_binarizer', LabelBinarizer()),
    ])
```

補足: パイプラインについて

- **利便性** : 推定器のシーケンス全体に合わせるためには、fit を呼び出してデータを一度 predict する
- **ジョイントパラメータの選択** : パイプライン内のすべての推定器のパラメータを一度にグリッドで検索できる

最後のパイプライン以外はすべて変換器でないといけない（transform メソッドが必要 → 継承しなきゃ）。最後の推定器は、任意のタイプ（変換器、分類器）であってよい

Prepare the data for ML algorithms

複数の変換器の出力を組み合わせる

```
from sklearn.pipeline import FeatureUnion

# それぞれの変換を横にして繋げている（1行を対象としている）
full_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", num_pipeline),
        ("cat_pipeline", cat_pipeline),
    ])
```

Select and train a model

```
# 線形回帰
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)

# let's try the full pipeline on a few training instances
some_data = housing.iloc[:5]          #  テストデータ
some_labels = housing_labels.iloc[:5] #  そのラベル
some_data_prepared = full_pipeline.transform(some_data)

print("Predicts:", lin_reg.predict(some_data_prepared))

# 平均二乗誤差 と 平均絶対値誤差
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
```

Select and train a model

```
# 決定木を用いた学習
from sklearn.tree import DecisionTreeRegressor

# モデル定義と学習
tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(housing_prepared, housing_labels)

# 予測
housing_predicts = tree_reg.predict(housing_prepared)

# 平均二乗誤差
tree_mse = mean_squared_error(
                housing_labels, housing_predicts)

tree_rmse = np.sqrt(tree_mse) #  平方根
```

Fine-tune your model

```
# クロスバリデーション
from sklearn.model_selection import cross_val_score

scores = cross_val_score(
            tree_reg,             #  学習モデル
            housing_prepared, #  データ
            housing_labels,    #  データのラベル
            # この score は何故 negative ??? -> 要確認
            scoring="neg_mean_squared_error",
            cv=10                 #  分割数
        )
# 平方根
tree_rmse_scores = np.sqrt(-scores)
```

Fine-tune your model

ランダムフォレスト回帰とサポートベクター回帰

```
# ランダムフォレスト回帰（回帰とか分類の違いは知っている前提）
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor(random_state=42)
forest_reg.fit(housing_prepared, housing_labels) #  学習
predicts = forest_reg.predict(housing_prepared)  #  予測

# サポートベクター回帰
from sklearn.svm import SVR

svm_reg = SVR(kernel="linear") #  SV 系は要カーネル指定
svm_reg.fit(housing_prepared, housing_labels) #  学習
predicts = svm_reg.predict(housing_prepared)  #  予測
```

Fine-tune your model

良いパラメータを探す

```
from sklearn.model_selection import GridSearchCV

# どこら辺を探索するか定義する
# それぞれのパラメータの意味はそのうち調べる
param_grid = [
    # try 12 (3×4) combinations of hyperparameters
    {'n_estimators': [3, 10, 30],
     'max_features': [2, 4, 6, 8]},
    # then try 6 combinations with bootstrap set as False
    {'bootstrap': [False], #  重複を許して新しいサンプルを作る
     'n_estimators': [3, 10],
     'max_features': [2, 3, 4]},
  ]
```

Fine-tune your model

良いパラメータを探す（続き）

```
forest_reg = RandomForestRegressor(random_state=42)
# train across 5 folds,
# that's a total of (12+6)*5=90 rounds of training

# 探索モデル作成
grid_search = GridSearchCV(
                    forest_reg, #  モデル
                    param_grid, #  探索するパラメーター一覧
                    cv=5,       #  クロスバリデーション分割数
                    scoring='neg_mean_squared_error'
                )

# 探索実行
grid_search.fit(housing_prepared, housing_labels)
```

Fine-tune your model

良いパラメータを探す（ランダム探索）

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint
# パラメータの分布を定義する。分布に沿った乱数で探索
param_distribs = {
        'n_estimators': randint(low=1, high=200),
        'max_features': randint(low=1, high=8),
    }
forest_reg = RandomForestRegressor(random_state=42)
rnd_search = RandomizedSearchCV(
                forest_reg,
                param_distributions=param_distribs,
                n_iter=10, cv=5,
                scoring='neg_mean_squared_error',
                random_state=42)
rnd_search.fit(housing_prepared, housing_labels)
```

Extra: Label Binarizer hack

- **やりたいこと**：前処理と学習も一つの pipeline でまとめたい
- これまでの処理は以下の通り

```
# 前処理（ラベル以外の部分）
housing_prepared = full_pipeline.fit_transform(housing)
# ラベルの部分の定義（元が実数なので特に加工していない）
housing_labels = strat_train_set[
                        "median_house_value"].copy()
# 学習する部分（線形回帰の場合）
forest_reg.fit(housing_prepared, housing_labels)
# 予測部分
housing_predictis = lin_reg.predict(test_data)
```

Extra: Label Binarizer hack

- もとの `LabelBinarizer` の `fit_transform()` は一つのパラメータしか受け付けない。

```
# 0 か 1 からなる One-hot vector に変換する
from sklearn.preprocessing import LabelBinarizer
LabelBinarizer().fit_transform(sample) #  引数がひとつだけ
```

- よってそのままとめるとエラーになる。（余裕があれば実行結果載せる）

```
full_pipeline_with_predictor = Pipeline([
        ("preparation", full_pipeline),
        ("linear", LinearRegression())
    ])
full_pipeline_with_predictor.fit(housing, housing_labels)
```

Extra: Label Binarizer hack

- 補足 : 最後以外のパイプラインの要素には `transform()` が必要。実際には `fit_transform()` が全体パイプラインの `fit()` 実行時に呼び出されている模様？
- つまりどういうことだってばよ？
  - → `LabelBinarizer` を継承して `fit_transform()` をうまく書き換えたクラスを作ってパイプラインにつっこめばOK

Extra: Label Binarizer hack

- クラス継承らへんは[Python]クラス継承(super) - Qiita などを見て勉強してください。
- 以下のように拡張したクラスを定義する。

```
# SFLB : SupervisionFriendlyLabelBinarizer の略
# クラス定義の引数部分は継承元
class SFLB(LabelBinarizer):
    # 引数が二つあっても良いように定義している
    # デフォルト引数を定義しているので、引数一つにも対応している
    def fit_transform(self, X, y=None):
        # super() は自身の親クラスを呼び出している
        # よって親クラス（LabelBinarizer）の fit_transform
        # を呼び出している
        # Python3 からは super() のみで良い模様
        return super(SFLB, self).fit_transform(X)
```

Extra: Label Binarizer hack

```
# Replace the Labelbinarizer with a SFLB
cat_pipeline.steps[1] = ("label_binarizer", SFLB())

# これで全体パイプラインが作成できる
# cat_pipeline は full_pipeline 内にはいっていることに注意
# hint: 参照渡し
full_pipeline_with_predictor = Pipeline([
        ("preparation", full_pipeline),
        ("linear", LinearRegression())
    ])

# データの前処理と学習
full_pipeline_with_predictor.fit(housing, housing_labels)

# 予測
full_pipeline_with_predictor.predict(some_data)
```

Extra: Model persistence using joblib

```
# モデル（学習した後等の）は下記のように保存・読み込みができる
from sklearn.externals import joblib
joblib.dump(my_model, "my_model.pkl")          # 保存
my_model_loaded = joblib.load("my_model.pkl") # 読み込み
```

Extra: Example SciPy distributions for RandomizedSearchCV

```
from scipy.stats import geom, expon
# geometric : 幾何分布。
# ベルヌーイ試行で初めて成功させるまでの回数
geom_distrib=geom(0.5).rvs(10000, random_state=42)
# exponential : 指数分布
expon_distrib=expon(scale=1).rvs(10000, random_state=42)
```

Exercise solutions: 1

- `問題1` : Support Vector Machine regressor （サポートベクター回帰）を用いて最もパフォーマンスが良いものを探せ。カーネルは `linear` と `rbf` の二種類選べる。
- rbf カーネルとか指定するとどうなる？
  - 各点を高次元に写像することで、これまでできなかった分類ができるようになったりする
  - ただし linear よりは計算量が多くなる

Exercise solutions: 1

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'kernel': ['linear'],
     'C': [10., 30., 100., 300., 1000.,
           3000., 10000., 30000.0]},
    {'kernel': ['rbf'],
     'C': [1.0, 3.0, 10., 30., 100., 300., 1000.0],
     # rbf の方がパラメータが増えている
     'gamma': [0.01, 0.03, 0.1, 0.3, 1.0, 3.0]}]

svm_reg = SVR()
grid_search = GridSearchCV(
                  svm_reg, param_grid, cv=5,
                  scoring='neg_mean_squared_error',
                  verbose=2, n_jobs=4)
grid_search.fit(housing_prepared, housing_labels) #  探索
```

Exercise solutions: 2

- 問題2 : GridSearch の代わりに RandomizedSearch を使ってみる

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import expon, reciprocal
# Note: gamma is ignored when kernel is "linear"
param_distribs = {
        'kernel': ['linear', 'rbf'],
        'C': reciprocal(20, 200000),
        'gamma': expon(scale=1.0)}
svm_reg = SVR()
rnd_search = RandomizedSearchCV(
                svm_reg, n_iter=50, cv=5,
                param_distributions=param_distribs,
                scoring='neg_mean_squared_error',
                verbose=2, n_jobs=4, random_state=42)
rnd_search.fit(housing_prepared, housing_labels)
```