Hands on Machine Learning with Scikit-Learn &TensolFlow

Chapter 7

Ensemble Learning and Random Forests

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 7

Ensemble Learning and Random Forests

ポイント

- 弱い学習器と強い学習器
- アンサンブル学習 = 弱学習器をたくさん使って学習・予測する
  - bagging, boosting, stacking
- Random Forests
  - Decision Tree とは何が違うのか？
- 「wisdom of the crowd」=「群衆の知恵、みんなの意見」

1. Voting Classifiers

- weak learner（弱学習器）
  - ランダムよりは良いという意味が込められてる
- strong learner（強学習器）
  - 精度高い
- Hard voting → 各々の学習器の予測結果を多数決

大数の法則

[大数の法則 - Wikipedia]

期待値 μ であるような可積分独立同時分布から得た確率変数列 $X_1, X_2, \cdots$ の算術平均

$$[X_n] = \frac{X_1 + X_2 + \cdots + X_n}{n}$$

の取る値が、ほぼ確実に

$$\lim_{n \to \infty} [X_n] = \mu$$

となること。無限個データがあったら本来の期待値が得られるよね、ということ

- 表が出る確率が 51%（かろうじて表の方が出やすい）、裏が出る確率が 40% のコインを 1000 回投げると、表の出た回数の方が裏が出た回数より多い確率は **75%** になる（ただしこれは各サンプルが独立であることが前提）。
- この通り一つ一つは精度が高くない学習器でも大量に用意すると精度が高くなる
- Voting の Hard と Soft の違い
  - Hard : 各学習器で予測結果を出し多数決で決定
  - Soft : 各学習器で予測確率を出し、平均をとって最も高いクラスを出力
  - 例: 0,1 クラス分類で A(0) = 55%, B(0) = 65%, C(0) = 0% としたときの Voting 結果の違い

コード説明

```
heads_proba = 0.51  # 表が出る確率
a = np.random.rand(10000, 10)  # 10000×10 の乱数(0 から 1)
# a と比較して表が出る確率が大きい部分を True として int に変換
# True は int に変換すると 1 になる（復習）
coin_tosses = (a < heads_proba).astype(np.int32)
# 縦方向に、累計和を求める
c = np.cumsum(coin_tosses, axis=0)
# 1 から 10000 までの整数を (1, 1) 型に変換
# 元は 1次元リストなので
d = np.arange(1, 10001).reshape(-1, 1)
# ある回数コインを投げたときの表が出た回数の比率を表す(10通り)
cumulative_heads_ratio = c / d
```

- `np.cumcum()` : 累計和を求める
  - 例: `np.cumsum([1, 2, 3]) -> array([1, 3, 6])`

コード説明

これでハードな投票分類器が作れる

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression(random_state=42)
rnd_clf = RandomForestClassifier(random_state=42)
svm_clf = SVC(random_state=42)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc',
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Bagging and Pasting

- 学習する際に、同じ学習方法（分類器も同じ？）のままデータセットを変える
  - bagging (bootstrap aggregating)
    - 各訓練データを元データから **復元抽出** したサブデータセットを使う
  - pasting
    - 各訓練データを元データから **非復元抽出** したサブデータセットを使う

予測時の合意は statistical mode (多数決)で行われる
各分類器は独立しているので並列訓練可能

Bagging and Pasting in Scikit-Learn

通常の決定木

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
y_pred_tree = tree_clf.predict(X_test)
```

Bagging with 決定木

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state=42),
    n_estimators=500, max_samples=100, bootstrap=True,
    n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

Out-of-Bag Evaluation

- `BaggingClassifier` 使用時に、復元抽出を行う (`bootstrap=True`) と、 **約63%** のデータはどの学習器用のサブデータセットに含まれる。残りの **約37%** は学習器毎に違う組み合わせとなる。
- 残りの部分を out-of-bag(oob) と呼ぶ
- `oob_score=True` として、 `.oob_score_` で学習時のoobデータに対する精度を確認することができる → テストデータの結果に似ていた

Random Patches and Random Subspaces

- ここまでは**データ**に対してサンプリングを行ってきた（どのデータを訓練データとするか）
- `BaggingClassifier` は、以下のパラメータを与えることで **特徴** に対してサンプリングを行えるようになる（この学習器ではどの特徴を用いるか）
  - `max_features` : int or float, optional (default=1.0)
    - 特徴の個数（次元数）
  - `bootstrap_features` : boolean, optional (default=False)
    - 復元抽出を行うか否か

- これは画像の様な、入力ベクトルが高次元となる場合に効果がある。
- トレーニングデータと特徴の両方に対してサンプリングを行う方法を **Random Patches** 法と呼ぶ
- トレーニングデータはそのままにして、特徴だけサンプリングする方法は **Random Subspace** 法と呼ぶ。特徴空間を分割しているからだと思われる

これらを使うことでより多彩な分類器が作れる
→　表現力が上がる

Random Forests

- RandomForest は決定木のアンサンブル学習
- Scikit-laern での `BaggingClassifier` に対し決定木を適用したもの。`RandomForestClassifier` で使える

以下の2つの分類器は大体同じ

```
BaggingClassifier(
    DecisionTreeClassifier(splitter="random",
                           max_leaf_nodes=16,
                           random_state=42),
    n_estimators=500, max_samples=1.0,
    bootstrap=True, n_jobs=-1, random_state=42)

RandomForestClassifier(
    n_estimators=500, max_leaf_nodes=16,
    n_jobs=-1, random_state=42)
```

Random Forests の学習の仕方

- 決定木ではノードを分割する時により良く分割できるような特徴を網羅的に探す
- RF ではランダムに分割した特徴の部分集合から、最も良い特徴を探す

Extra-Trees

Random Forest とその派生アルゴリズム

- RandomForest の派生形の一つ
- RFでは特徴の選択やその特徴の閾値は、良くなるように考慮されていた
- ET ではそれらもランダムに選択する
- 精度は下がるが学習が速く過学習しにくい
- RandomForestClassifier と ExtraTreesClassifier の優劣をつけるのは難しい。データに対し Cross-validation かけて確認して判断する

Feature Importance

- 決定木 : 重要な特徴ほど根本で判断してる → 下（葉）に近づくほど重要でない特徴を使うようになる
- Scikit-learn では `feature_importances_` で各特徴の重要度が確認できる
- とりあえず重要な特徴が知りたい場合は RF でモデルを作ってみるのはあり

Boosting

- Boosting(元は hypothesis boosting)は弱学習器を強い学習器にする方法
- アイデア : 予測器を順次学習させ、前の学習器を修正しようとすること？
- **AdaBoost** と **Gradient Boosting** が有名

AdaBoost

- 以前の学習器を修正する方法の１つ：予測を間違えたデータにより気をつけて学習する　→　重みをつける(boost)
- AdaBoost の学習ステップ
  - ① ベースとなる分類器を作成し予測する
  - ② 作った分類器で予測し、間違えたデータの重みを増やす
  - ③ 重みを元に分類器を構築する
  - ④ ②と③を繰り返す
- 上記は並列化できないためスケーリングに注意

AdaBoost 補足

j 番目の学習器のエラー率

$$r_j = \frac{\sum_{i=1, \hat{y}_j{}^{(i)} \neq y^{(i)}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$

- m : 訓練データの数
- j : 学習器の番号（j番目）
- $\hat{y}_j{}^{(i)}$: j 番目の学習器の i 番目のデータに対する予測値

エラー重みから各学習器の重み①を計算

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

各データ i に対して重みを更新（間違えたら増えるイメージ）

$$\text{if } \hat{y_j}^{(i)} = y^{(i)} \text{ , } w^{(i)} \leftarrow w^{(i)}$$

$$\text{if } \hat{y_j}^{(i)} \neq y^{(i)} \text{ , } w^{(i)} \leftarrow w^{(i)} \exp(\alpha_j)$$

その後全ての重みは $\sum_{i-1}^{m} w^{(i)}$ で割って正規化

Adaboost での予測

$$\hat{y}(\mathbf{x}) = \arg\max_{k} \sum_{j=1, \hat{y}_j(\mathbf{x})=k}^{N} \alpha_j$$

- N : 予測器の数
- 予測が k だと言っている予測器の α（重み）を足し合わせ、最も大きくなるような k が予測値

Gradient Boosting

- Boosting の仲間、進化系の XGBoost は 2015 年の Kaggle で猛威を振るった
- Boosting 全体の特徴 : すでに学習した弱分類器のパラメータを変化させるのではなく、新しい弱分類器を加えることで全体の出力を変化させられる
- AdaBoost → 前の分類器で誤分類された値の重みを大きくするように更新する
- Gradient Boosting → 勾配情報を使った Boosting らしい。全分類器の和を 1 つの関数と見てる？
- GBDT(Gradient Boosting Decision Tree) : 弱分類器に決定木を使った Gradient Boosting

Gradient Boosting（ベーシック）の擬似コード

- $f_0(x) \leftarrow \arg\min_\rho \sum_{i=1}^{N} L(y_i, \rho)$;
- for $t = 1$ to T do
  - $$\tilde{y}_i \leftarrow -\frac{\partial L(y_i, F_{t-1}(x_i))}{\partial F_{t-1}(x_i)} \text{ for all } i$$
  - fit f_t to minimize $\sum_{i=1}^{N} (\tilde{y}_i - f_t(x_i))^2$
  - $\alpha_t \leftarrow \arg\min_\alpha \phi(F_{t-1} + \alpha f_t)$
  - $F_t = F_{t-1} + \alpha_t f_t$
- return F_T;

Stacking

- アンサンブル法の１つ。 stacked generalization の略
- 簡単な流れ
  1. 訓練用データセットを分けて図のように複数の学習器で学習する（Figure 7-13）
  2. 学習した学習器を用いて、訓練に使ったデータセットで予測して値を取得し、各学習器から取得した値を特徴量として使って学習器をされに作る
- Scikit-learn には無いけど、gitでの実装 はあるので試せる

Exercises

間に合わなかったので省略

Reference

- Gradient Boosting と XGBoost
- アンサンブル学習について勉強したのでまとめました：Bagging(Random Forest / Decision Jungles / Deep Forest) / Boosting(AdaBoost / Gradient Boosting / XGBoost) / Stacking