Hands on Machine Learning with Scikit-Learn &TensolFlow Chapter 8

Dimensionality Reduction

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 8

Dimensionality Reduction

ポイント

- データの特徴次元（特徴数）が大きいと、学習が遅いだけでなく良い結果が得られない
  - 次元の呪い
- 次元削減すれば対処できる場合がある
- PCA は分散が大きい順に軸を取り出す(線形な座標変換をする)
- 第 i 軸は第 $i-1$ 軸と直交と定義すると逐次的に計算できる

The Curse of Dimensionality

- 日常生活は 3 次元だから高次元イメージしづらい
- 1×1 の四角形(2次元)を考える
  - ランダムな点を指定して、点が四角形の枠から 0.001 以内にある可能性は、
  - $1 - (1 - 0.001 \times 2)^2 = 0.003999 \cdots$、約 0.4%
- $10,000$ 次元の四角形だと、
  - $1 - (1 - 0.001 \times 2)^{10000} = 0.9999 \cdots$、99.999999% 以上になる ⇒ ほとんどのデータは枠の付近にある

- これらの可能性から、高次元にあるデータは**スパース**であるリスクがある
- どのデータ間でも距離が大きすぎる
- Overfitting しやすい
  - 解決法の一つ: Training Data 増やす
  - 十分な密度になるくらいデータを増やす
  - 100 次元の空間だと、もし各データの（最も近いデータ？との）距離が平均して0.1 以内にあるためには、観測可能な宇宙内の原子の数よりもデータが必要らしい？　⇒　たくさんデータが必要

Main Approaches for Dimensionality Reduction

- Projection (低次元への写像)
  - ある低次元の空間（2次元だと面、1次元だと線）に、各データを写す
- Manifold Learning (多様体学習)
  - d 次元多様体: 元は d 次元の空間（線とか面）を、より大きな$n(n > d)$次元上で曲げたりひねったりしたもの
  - 例: Swiss roll は、2次元の空間（面）を、3次元でくるくる巻いた形になっている

Approach 2 : Manifold Learning

- Swiss roll データの場合、局所的にみると2次元の面に似ている
- MNIST データの場合、各文字画像は、線が繋がってできている、境界線は白、大体中央に文字がある、などの共通点がある ⇒ 「文字画像を生成する自由度 << 全くランダムな画像を作る自由度 」
- これらの制約はデータセットをより低次元に圧縮する傾向がある

- 多様体である仮定を置くこと ⇒ 低次元で表現するとより簡単になることを仮定している
  - いつもそうとは限らない（テキスト図参照）
  - ⇒ モデルのトレーニング前に次元削減を行うことで、**学習は速くなるが、より良い精度やよりシンプルに解けるようになるとは限らない**

PCA

- PCA: Principal Component Analysis (主成分分析)
- 最初にデータの最も近い超平面を特定し、その超平面にデータを写像する

PCA: Preserving the Variance

- 二次元データの場合: どの直線（方向）に写像すると、最も分散が大きくなるのかをテキストの図で確認する
- ⇒ 各データの写像先データの距離（超平面との距離みたいなもの）の平均二乗誤差が最も小さくなる超平面を探していることと等価

PCA: Principal Components

- orthogonal: 直交
- ある軸（主成分）を探す際に、それまでに探した主成分すべてと直交するような軸を探す

PCA: Projecting Down to d Dimensions

- SVD(特異値分解) してる　→　行列の分解
- PCA では各点が原点の中心にあると仮定している ⇒ 各データについて、そのデータの平均値が引いてあることを確認する ⇒ Scikit-laern は勝手に引いているから気にしなくてよい

PCA: Using Scikit-Learn

- メモ: Scikit-learn のPCA は内部で SVD 使ってる

```
# scikit-learn の PCA
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
# SVD 使う場合
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
S[:n, :n] = np.diag(s)  # 対角行列に変更する
# 2つの値が近いと True 返す。（元の行列を近似できていることを確認）
np.allclose(X_centered, U.dot(S).dot(Vt))
W2 = Vt.T[:, :2]
X2D_using_svd = X_centered.dot(W2)
```

PCA: Explainged Variance Ratio

- ある主成分が、データ全体の分散のうちどれくらいを占めるのかを表す
- 主成分をどこまで残すのかを検討する際に参考にする
- 95% などが 1 つの目安

PCA: Choosing the Right Number of Dimensions

```
pca = PCA()
pca.fit(X)
cumsum = np.cumsum(pca.explainled_variance_ratio_)
# 累計和が 0.95 以上の最初のインデックス
# cumsum >= 0.95 は、True(1) か False(0) のみの配列
d = np.argmax(cumsum >= 0.95) + 1
```

以下のように設定すれば勝手にできる

```
# n_components が1以上の整数だと個数、0から1の実数だと比率になる
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X)  # 低次元の写像したベクトル
```

PCA: PCA for Compression

- PCA はでは以下のようにして d 次元上の空間にベクトル $\mathbf{X}$ を写像している
  - $\mathbf{X}_{d-proj} = \mathbf{X} \cdot \mathbf{W}_d$
- 逆に以下のようにして d 次元上のベクトルを元の次元に復元できる
  - $\mathbf{X}_{recovered} = \mathbf{X}_{d-proj} \cdot \mathbf{W}_d^T$
- ただし完全にもとと同じように復元されない → 圧縮された状態

PCA: Incremental PCA(IPCA)

- 元のPCAの課題
  - 全データを使ってSVDを適用しないといけない
  - メモリが足りない（乗り切らない）
- IPCA では、逐次実行できるようになっている

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    print(".", end="") # not shown in the book
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

PCA: Randomized PCA(RPCA)

- 確率的勾配降下法を使って求める方法
- 普通のPCAの計算量が $O(m \times n^2) + O(n^3)$ に対し、RPCAだと $O(m \times d^2) + O(d^3)$ になる
  - m: データの数
  - n: 元のデータの次元
  - d: 変換後の次元、主成分の数
- よって d が n よりかなり小さいとすごく速くなる
- PCAに、 `svd_solver="randomized"` をつけるだけ

Kernel PCA

- 非線形な超平面(軸)を考えるためのもの
- SVMと同じ発想
- カーネル空間（さらに高次元）に変換して考える

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2,
                    kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

Kernel PCA: Selecting a Kernel and Tuning Hyperparameters

下記のように grid search で最適なカーネルやパラメータを探す。例ではロジスティック回帰の精度を基準にしている

```
clf = Pipeline([
        ("kpca", KernelPCA(n_components=2)),
        ("log_reg", LogisticRegression()) ])

param_grid = [{
        "kpca__gamma": np.linspace(0.03, 0.05, 10),
        "kpca__kernel": ["rbf", "sigmoid"] }]
# 3-クロスバリデーションしている
grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

LLE

- 多様体学習の一つ
- 多様体: 元は線形である空間（面とか線とか）を曲げたりねじったりして高次元上に表される空間
- 局所的な位置関係を保存して低次元に写像しようとしている
-

LLEの詳細

1. あるデータに $\mathbf{x}$ に対して K 最近傍(最も近い K 個のデータのこと)を選出する
   - 最近傍で得られたデータ集合を κ とする
2. κ に属するデータ集合 $\mathbf{x}_j \in \kappa$ の線形結合を考える。これが小さくなれば、近傍データで $\mathbf{x}$ を表現できていることになる

$$\left| \mathbf{x} - \sum_{\mathbf{x}_j \in \kappa} w_j \mathbf{x}_j \right|^2$$

3. 全てのデータ $\mathbf{x}_i$ に対して上記の誤差を最小化する $\mathbf{W}$ を求める(ローカルな位置関係を取得)

$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} \sum_{i=1}^{m} \left| \mathbf{x}_i - \sum_{j=1}^{m} w_{i,j} \mathbf{x}_j \right|^2$$

$$\text{subject to} \begin{cases} w_{i,j} = 0 \text{ if } \mathbf{x}_j \text{ is not } k \text{ c.n. of } \mathbf{x}_i \\ \sum_{j=1}^{m} w_{i,j} = 1 \text{ for } i = 1, 2, \cdots, m \end{cases}$$

4. 位置関係を保ったまま次元を削減（低次元表現の獲得）

$$\hat{\mathbf{Z}} = \arg\min_{\mathbf{Z}} \sum_{i=1}^{m} \left| \mathbf{z}_i - \sum_{j=1}^{m} \hat{w}_{i,j} \mathbf{z}_j \right|^2$$

- Computional complexity(LLE)、LLEの計算量
  - k 近傍の探索: $O(m\log(m)n\log(k))$
  - $\mathbf{W}$ の最適化: $O(mnk^3)$
  - 低次元表現の獲得: $O(dm^2)$ ← 重すぎる
- あまり量が大きいデータは適さない

Other Dimensionality Reduction Techniques

- Multidimensional Scaling (MDS)
- Isomap
- t-distributed
- Linear Discriminant Analysys (LDA)

Isomap

- 非線形次元削減手法の一つ
- K近傍グラフを用いて多様体上の測地線距離を（近似的に）求め、多次元尺度構成法を用いて近時的にユークリッドな低次元空間に射影する

補足: 共分散

ある n 次元のデータ x と y の間の共分散

$$s_{xy} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

- n : 各データの次元
- $\bar{x}$: x の平均
- $\bar{y}$: y の平均

補足: 共分散

python だと下記のように求められる

```
x = np.array([1, 2, 4, 5])
y = np.array([5, 6, 8, 9])
x_bar = x.mean()
y_bar = y.mean()
s = (x - x_bar) @ (y - y_bar)
s / x.shape[0]
```

Exercises

- 省略。やらないとまずい

参考サイト

- PCAの最終形態GPLVMの解説
- Rでisomap（多様体学習のはなし）
- 【多様体学習】LLEとちょっとT-SNE - HELLO CYBERNETICS