W4995 Applied Machine Learning

NMF; Outlier detection

04/01/19

Andreas C. Müller

Check out dabl!

https://amueller.github.io/dabl/user_guide.html

For now: visualization and preprocessing

try:

```
import dapl
dapl.plot(data_df, target_col='some_target')
```

and:

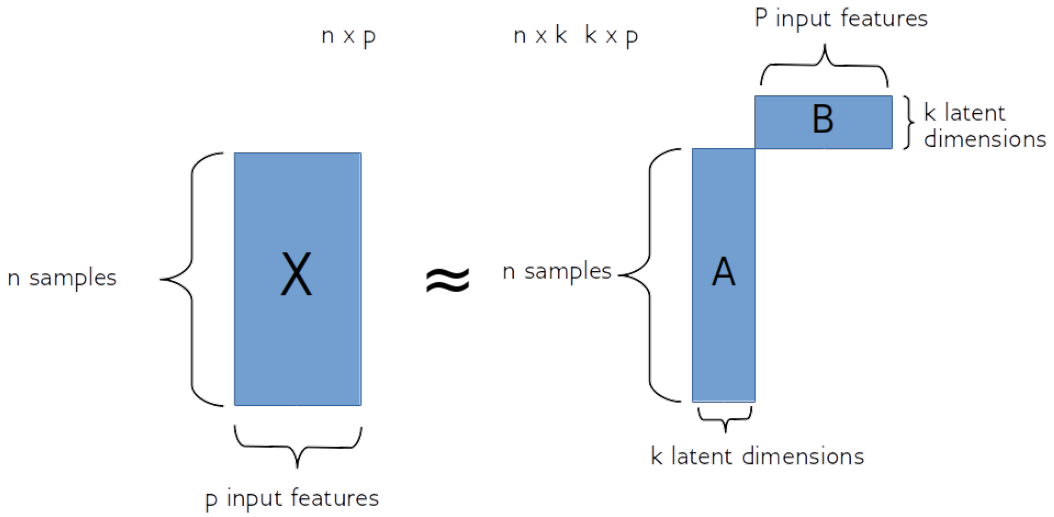
```
import dapl
data_clean = dapl.clean(data)
```

Non-Negative Matrix Factorization

Matrix Factorization

$$X \approx AB$$

n x p n x k k x p



Matrix Factorization

$$X \approx AB$$

R input features

PCA

Other Matrix Factorizations

- PCA: principal components orthogonal, minimize squared loss
- Sparse PCA: components orthogonal and sparse
- ICA: independent components
- Non-negative matrix factorization (NMF): latent representation and latent features are nonnegative.

NMF

NMF Loss and algorithm

Frobenius loss / squared loss ($Y = HW$):

$$l_{\text{frob}}(X, Y) = \sum_{i,j}(X_{ij} - Y_{ij})^2$$

Kulback-Leibler (KL) divergence:

$$l_{\text{KL}}(X, Y) = \sum_{i,j} X_{ij} \log\left(\frac{X_{ij}}{Y_{ij}}\right) - X_{ij} + Y_{ij}$$

NMF Loss and algorithm

Frobenius loss / squared loss ($Y = HW$):

$$l_{\text{frob}}(X, Y) = \sum_{i,j}(X_{ij} - Y_{ij})^2$$

Kulback-Leibler (KL) divergence:

$$l_{\text{KL}}(X, Y) = \sum_{i,j} X_{ij} \log\left(\frac{X_{ij}}{Y_{ij}}\right) - X_{ij} + Y_{ij}$$

optimization :

- Convex in either W or H, not both.
- Randomly initialize (PCA?)
- Iteratively update W and H (block coordinate descent)

NMF Loss and algorithm

Frobenius loss / squared loss ($Y = HW$):

$$l_{\text{frob}}(X, Y) = \sum_{i,j}(X_{ij} - Y_{ij})^2$$

Kulback-Leibler (KL) divergence:

$$l_{\text{KL}}(X, Y) = \sum_{i,j} X_{ij} \log\left(\frac{X_{ij}}{Y_{ij}}\right) - X_{ij} + Y_{ij}$$

optimization :

- Convex in either W or H, not both.
- Randomly initialize (PCA?)
- Iteratively update W and H (block coordinate descent)

Transforming data:

- Given new data `X_test`, fixed `W`, computing `H` requires optimization.

Why NMF?

Why NMF?

- Meaningful signs

Why NMF?

- Meaningful signs
- Positive weights

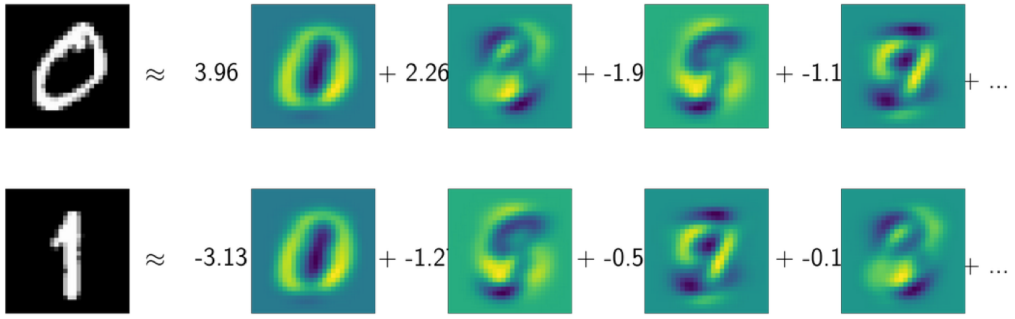
Why NMF?

- Meaningful signs
- Positive weights
- No “cancellation” like in PCA

Why NMF?

- Meaningful signs
- Positive weights
- No “cancellation” like in PCA
- Can learn over-complete representation

PCA (ordered by projection, not eigenvalue)



NMF (ordered by hidden representation)

Downsides of NMF

- Can only be applied to non-negative data

Downsides of NMF

- Can only be applied to non-negative data
- Interpretability is hit or miss

Downsides of NMF

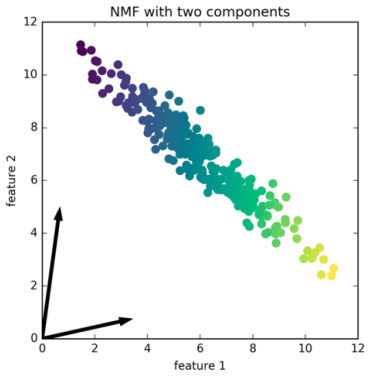
- Can only be applied to non-negative data
- Interpretability is hit or miss
- Non-convex optimization, requires initialization

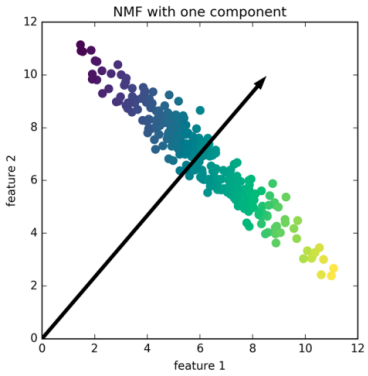
Downsides of NMF

- Can only be applied to non-negative data
- Interpretability is hit or miss
- Non-convex optimization, requires initialization
- Not orthogonal

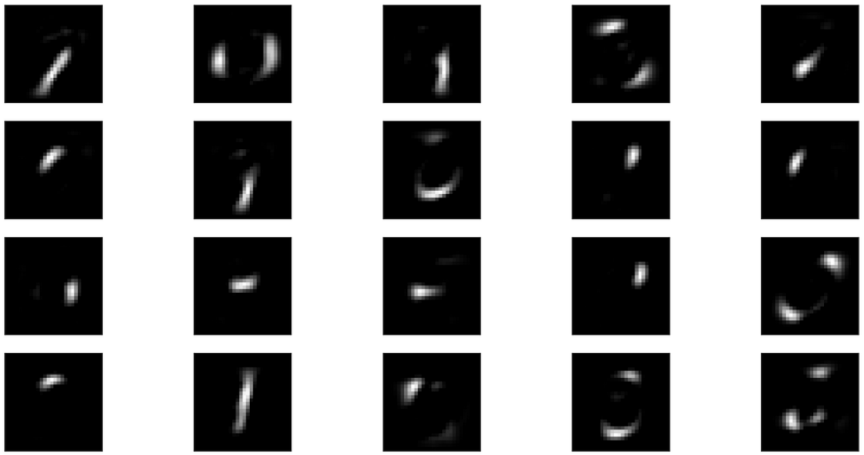
Downsides of NMF

- Can only be applied to non-negative data
- Interpretability is hit or miss
- Non-convex optimization, requires initialization
- Not orthogonal





NMF with 20 components on MNIST



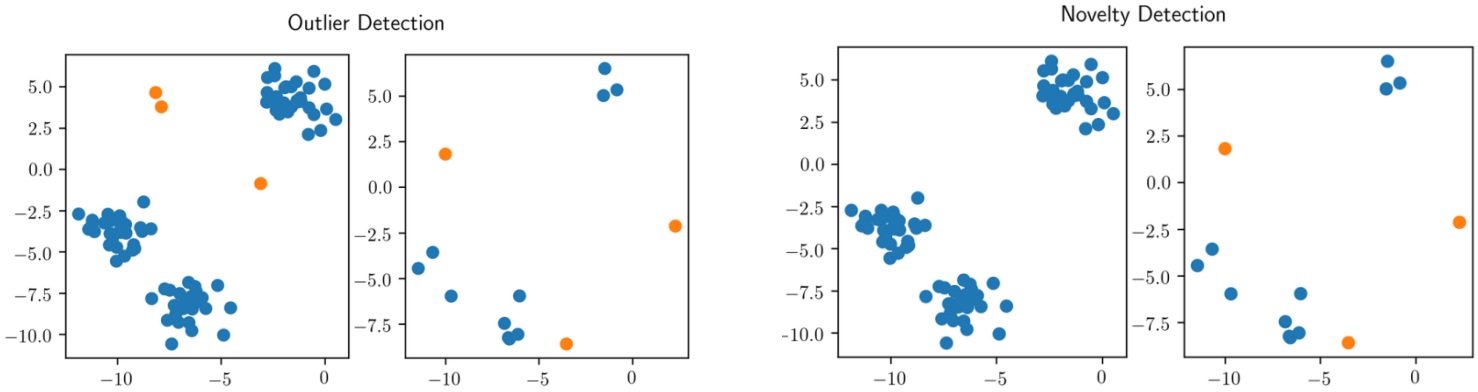
NMF with 5 components on MNIST

Applications of NMF

- Text analysis (next week)
- Signal processing
- Speech and Audio (see librosa)
- Source separation
- Gene expression analysis

Outlier Detection

Motivation



Applications

- Fraud detection (credit cards, click fraud, ...)
- Network failure detection
- Intrusion detection in networks
- Defect detection (engineering etc...)
- News? Intelligence?

Basic idea

- Model data distribution $p(X)$

Basic idea

- Model data distribution $p(X)$
- Outlier: $p(X) < \varepsilon$

Basic idea

- Model data distribution $p(X)$
- Outlier: $p(X) < \varepsilon$
- For outlier detection: be robust in modelling $p(X)$

Elliptic Envelope

$$p(X) = \mathcal{N}(\mu, \Sigma)$$

Elliptic Envelope

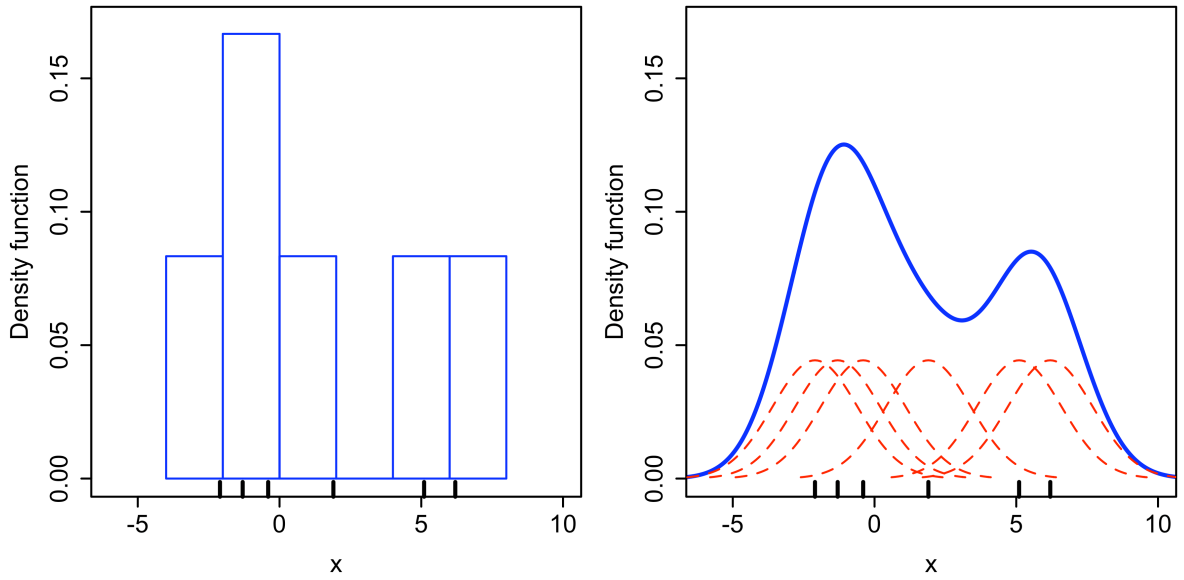
- Preprocessing with PCA might help sometimes.

```
from sklearn.covariance import EllipticEnvelope
ee = EllipticEnvelope(contamination=.1).fit(X)
pred = ee.predict(X)
print(pred)
print(np.mean(pred == -1))
```

```
[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
 -1  1  1  1 -1 -1  1 -1  1  1 -1  1 -1 -1 -1  1  1 -1 -1 -1 -1  1 -1  1  1]
0.104
```

Failure-case: Non-Gaussian Data

Kernel Density



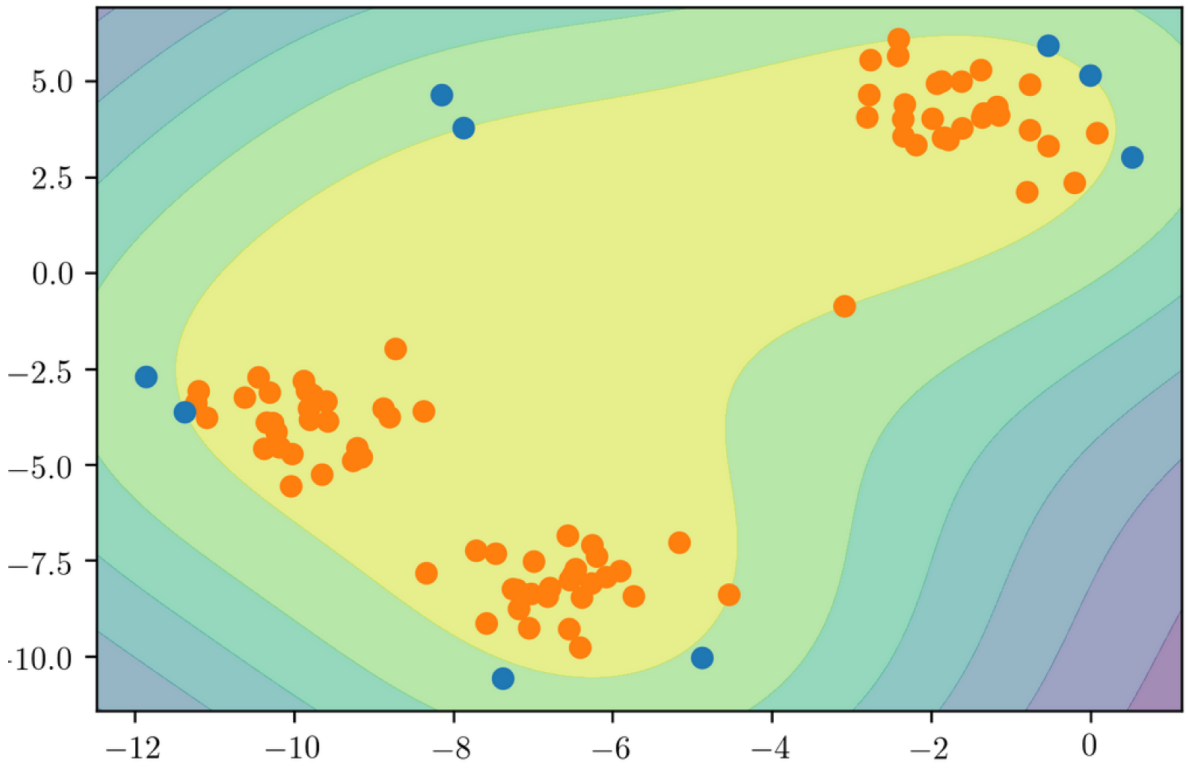
Kernel Bandwidth

```
kde = KernelDensity(bandwidth=3)
kde.fit(X_train_noise)
pred = kde.score_samples(X_train_noise)
pred = (pred > np.percentile(pred, 10)).astype(int)
```

One Class SVM

- Also uses Gaussian kernel to cover data
- Only select support vectors (not all points)
- Specify outlier ratio (contamination) via nu

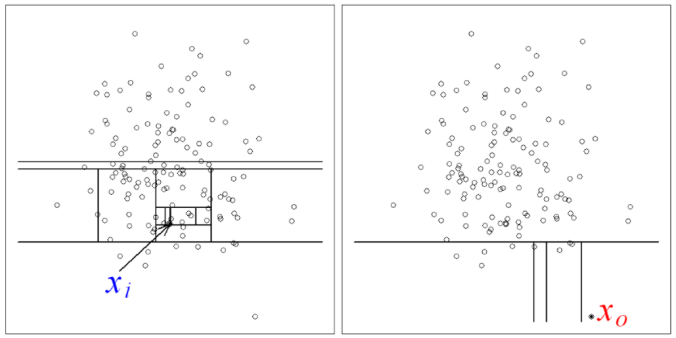
```
from sklearn.svm import OneClassSVM
scaler = StandardScaler()
X_train_noise_scaled = scaler.fit_transform(X_train_noise)
oneclass = OneClassSVM(nu=.1).fit(X_train_noise_scaled)
pred = oneclass.predict(X_train_noise_scaled)
```


5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12
−10
−8
−6
−4
−2
0

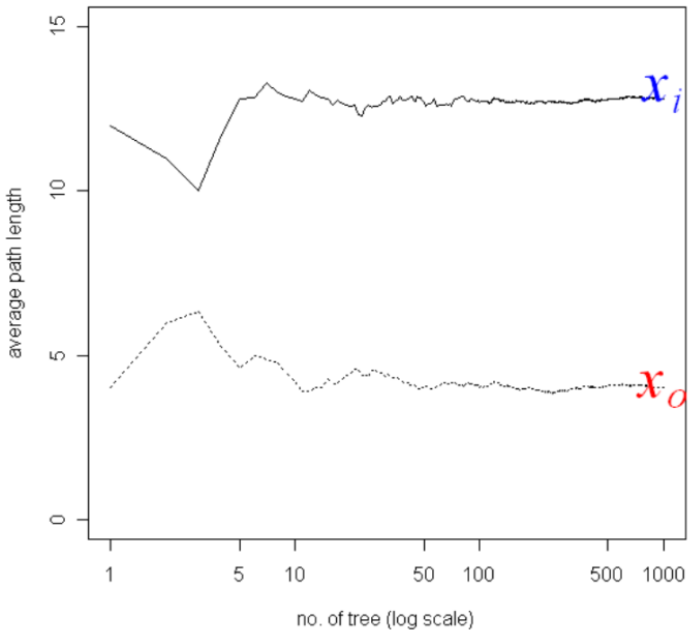
Isolation Forests

Idea

- Outliers are easy to isolate from the rest



(a) Isolating x_i (b) Isolating x_o



Normalizing the Path Length

Average path length of unsuccessful search in Binary Search Tree:

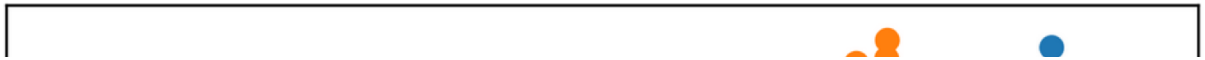
$$c(n) = 2H(n-1) - \left(\frac{2(n-1)}{n}\right) \text{ (H = Harmonic number)}$$

$$s(x,n) = 2^{-\frac{E(h(x))}{c(n)}} \text{ (h = depth in tree)}$$

- s < 0.5 : definite inlier
- s close to 1: outlier

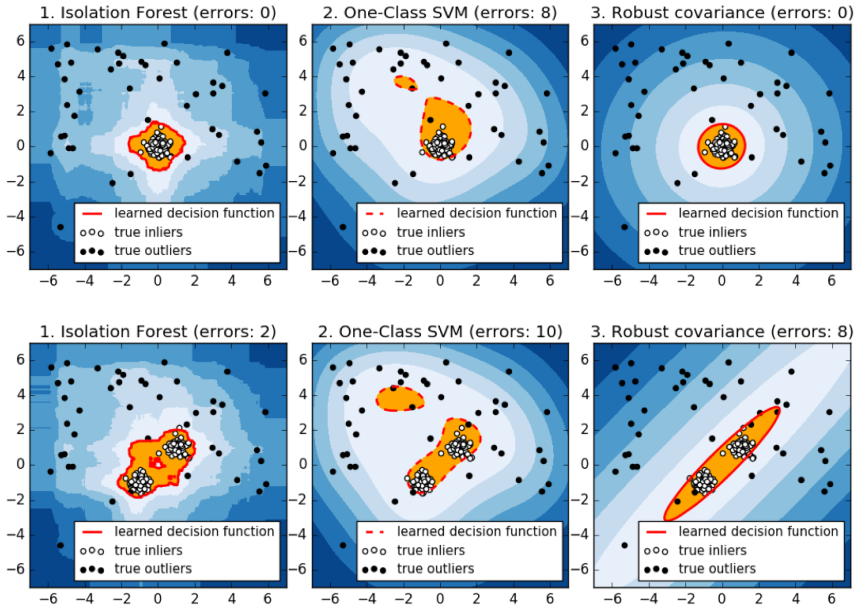
Building the forest

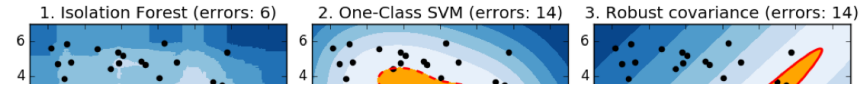
- Subsample dataset for each tree
- Default sample size of 256 works surprisingly well
- Stop growing tree at depth log_2(sample size) – so 8
- No bootstrapping
- More trees are better – default 100
- Need to specify contamination rate



Other density based-models

- PCA
- GMMs
- Robust PCA (not in sklearn :-()
- Any other probabilistic model - “robust” is better.


1. Isolation Forest (errors: 0)
2. One-Class SVM (errors: 8)
3. Robust covariance (errors: 0)
1. Isolation Forest (errors: 2)
2. One-Class SVM (errors: 10)
3. Robust covariance (errors: 8)
learned decision function
true inliers
true outliers


1. Isolation Forest (errors: 6)
2. One-Class SVM (errors: 14)
3. Robust covariance (errors: 14)

Summary

- Isolation Forest works great!
- Density models are great if they are correct.
- Estimating bandwidth can be tricky in the unsupervised setting.
- Validation of results often requires manual inspection.

Questions ?