metric-learn-etics

September 26, 2019

1 ETICS 2019 - Practical session on metric learning

You will need to install `metric-learn` with `pip` (in command-line or Anaconda prompt):

```
pip install metric-learn
```

[ ]:
```
from metric_learn import LMNN, NCA, MMC_Supervised, ITML_Supervised
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline, Pipeline
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import cross_validate, cross_val_score, GridSearchCV
```

[ ]:
```
def plot_2d(X, y, colormap=plt.cm.Paired):
    """ Plot in 2D the dataset X with colors according to the
    class given by the vector y """
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=colormap)
    plt.show()
```

Useful links:
metric-learn documentation: http://contrib.scikit-learn.org/metric-learn/
scikit-learn documentation: https://scikit-learn.org/stable/documentation.html (there is a search field in top right corner)

1.1 Supervised metric learning

We will start with the classic supervised learning scenario. We use the toy dataset digits.

[ ]:
```
# Load digits dataset
digits = datasets.load_digits()
X = digits.data
y = digits.target

# We keep only 7 classes for easier visualization
```

```
X = X[y < 7]
y = y[y < 7]
```

We can print the shape of the data matrix `X` to see the number of data points and the dimension.

[ ]:
```
print(X.shape)
```

We can also print the set of classes using the `unique` method of numpy.

[ ]:
```
print(np.unique(y))
```

The first step is to split the dataset into a training set and a test set using sklearn's `train_test_split`, and to standardize the features.

[ ]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.25,␣
 ↪random_state=0)
print(X_train.shape, X_test.shape)

# We standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

1.1.1 Dimensionality reduction and visualization

We would like to reduce the dimension of the data to 2D. A standard way to do this is PCA. We can easily use scikit-learn to fit on the training set and then transform the training and test sets (to set the dimension to 2, use `n_components=2`). We can then visualize the data using the function `plot_2d` defined at the top of the notebook.

[ ]:

Let's use metric learning instead to transform the data in 2D. You can try for instance LMNN and NCA: you can check the doc to see the usage, the available methods, default parameters, etc. They essentially follow the same API as a Transformer in scikit-learn, with `fit`, `transform`, and `fit_transform`. The parameter to choose the dimension of the transformed data is `n_components`, as in PCA.

Note: You may get some `ChangedBehaviorWarning`, this to warn users about the changes in the last release. If you want to ignore warnings, you can use:

```
import warnings
warnings.filterwarnings('ignore')
```

[ ]:

You can then plot the transformed data and compare visually to PCA.

[ ]:

From the trained metric learner, you can easily get a copy of the Mahalanobis matrix that was learned (method `get_mahalanobis_matrix`), and a pairwise function that corresponds to the learned metric (method `get_metric`) and can be easily plugged into other machine learning algorithms in scikit-learn.

[ ]:

1.1.2 Combination of metric learning with k-nearest neighbors classification

Scikit-learn (and by extension metric-learn) allows to use pipelines to combine transformers with classifiers or other machine learning models. Using `make_pipeline` or `Pipeline`, create a combination of PCA with `KNeighborsClassifier` (you can use `n_neighbors=1`) and use `fit` on the training set and `score` on the test set to obtain the prediction accuracy.

[ ]:
```
#äpca_knn = make_pipeline()
```

You can do the same with LDA (`LinearDiscriminantAnalysis`) which is a supervised dimensionality reduction algorithm and therefore a stronger baseline than PCA.

[ ]:

Now do the same for the metric learners you have used above, and compare the test accuracy.

[ ]:

Combination of metric learning with KMeans clustering

We can also combine metric learning with K-Means clustering (`KMeans`), again using a pipeline. For this you may try weakly supervised algorithms such as ITML or MMC: all weakly-supervised algorithms have a supervised version which creates pairs from labels. Check for instance `ITML_Supervised` and `MMC_Supervised`. MMC training is a bit unstable, sometimes leading to suboptimal performance. You may increase the number of pairwise constraints and maybe fit only a diagonal metric.

After you have `fit`, you can get the cluster labels given to the data by K-Means by calling the `predict` method, and use `adjusted_rand_score` between the predicted labels and the ground truth labels to measure the clustering performance. You can compare the score to K-Means on the original representation (without metric learning).

For all K-Means models we can simply set the number of clusters (`n_clusters`) to the number of classes in the dataset.

[ ]:

1.1.3 Cross validation, grid search, etc

In machine learning, it is very important to tune hyperparameters and more generally to do model selection. With metric-learn it is easy to run cross validations and grid searches to tune hyperparameters in the same way as normally done in scikit-learn.

First, you can use `cross_val_score` or `cross_validate` to quickly have cross validation scores for any metric learner (or pipeline). Check the scikit-learn documentation and give it a try. The number of CV folds can be set using the argument `cv` (e.g., `cv=3` for 3-fold cross validation). With `cross_validate` you can use the option `return_train_score=True` to show the training accuracy, which is convenient to check for possible overfitting.

[ ]:

You can also use `GridSearchCV` to tune hyperparameters using cross validation. Check the documentation of scikit-learn and use it to tune the parameters of one of your pipelines. For instance, you could use the pipeline LMNN+KNN and tune the parameter `k` of LMNN (between 1 and 3 for instance) and the parameter `n_neighbors` of KNN (in the same range). It may be slow as one needs to train as many models as the number of CV folds times the number of possible combinations of hyperparameter values.

Note: to be able to specify the parameters to tune in the dictionary, you need to give a name to each component of the pipeline with `Pipeline`. Example:

```
lmnn_knn = Pipeline(steps=[('lmnn', LMNN(n_components=2)), ('knn', KNeighborsClassifier())])
```

Then you can refer to parameters of each using the name as prefix, for instance `'lmnn__k'`.

[ ]:

We can then show the best combination of hyperparameters found by cross validation by printing the attribute `best_params_` of the `GridSearchCV` object.

[ ]:

You can then easily compute the test error of the tuned combined model by calling the `score` method of the `GridSearchCV` object.

[ ]:

2 Weakly-supervised metric learning

If you have time, you may experiment with other datasets, such as "Labeled Faces in the Wild". One version of this dataset only give weak supervision in the form of pairs (pairs of face images of same/different persons). You can load it with the following code. It may take some time as the images need to be downloaded first.

[ ]:
```
pairs, y_pairs = [datasets.fetch_lfw_pairs()[key] for key in ['pairs', 'target']]
pairs = pairs.reshape(2200, 2, 2914)
y_pairs = 2 * y_pairs - 1
```

[ ]:
```
print(pairs.shape, np.unique(y_pairs))
```

You can see that there is 2200 pairs with binary labels (similar / dissimilar), and each data point is in dimension 2914. To reduce the computational load as well as overfitting it may be a good idea to reduce the data dimension first, using PCA for instance... This is not ideal but easier to work with. There is a bit of nasty reshaping job to do, so I'll give the code for applying PCA (25 dimensions, you can easily change this)

[ ]:
```
pairs = PCA(n_components=25).fit_transform(pairs.reshape(4400, -1)).reshape(-1,␣
↪2, 25)
print(pairs.shape)
```

You can take a look at the scoring functions used for the weakly supervised setting (in particular the default, which is the Area under the ROC Curve, or AUC), and explore the other algorithms on this dataset or others. Have fun!

[ ]: