Lab 10: Principal Component Analysis

ACTL3142 and ACTL5110

These questions were sourced from the excellent textbook Géron (2022), which is available through the UNSW Library's access to O'Reilly Media texts. The author provided a Google Colab notebook containing the coding solutions.

Questions

Conceptual Questions

1. (HandsOnML3, Q8.1) ⋆ What are the main motivations for reducing a dataset's dimensionality? What are the main drawbacks?
2. (HandsOnML3, Q8.3) ⋆ Once a dataset's dimensionality has been reduced, is it possible to reverse the operation? If so, how? If not, why?
3. (HandsOnML3, Q8.4) ⋆ Can PCA be used to reduce the dimensionality of a highly nonlinear dataset?
4. (HandsOnML3, Q8.5) ⋆ Suppose you perform PCA on a 1,000-dimensional dataset, setting the explained variance ratio to 95%. How many dimensions will the resulting dataset have?
5. (HandsOnML3, Q8.7) ⋆ How can you evaluate the performance of a dimensionality reduction algorithm on your dataset?

Applied Question

1. ⋆ Load the MNIST dataset (uploaded as two CSV files to Moodle) keeping the given smaller training set (only 10,000 of the 60,000 training instances) and test set (another 10,000 instances). Train a random forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set. Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95%. Train a new random forest classifier on the reduced dataset and see how long it takes. Was training much

faster? Next, evaluate the classifier on the test set. How does it compare to the previous classifier? Try again with a SGDClassifier (if using Python) or a kNN (if using R). How much does PCA help now?

Solutions

Conceptual Questions

1. The main motivations for dimensionality reduction are:

   - To speed up a subsequent training algorithm (in some cases it may even remove noise and redundant features, making the training algorithm perform better)
   - To visualize the data and gain insights on the most important features
   - To save space (compression)

   The main drawbacks are:

   - Some information is lost, possibly degrading the performance of subsequent training algorithms.
   - It can be computationally intensive.
   - It adds some complexity to your Machine Learning pipelines.
   - Transformed features are often hard to interpret.

2. Once a dataset's dimensionality has been reduced using one of the algorithms we discussed, it is almost always impossible to perfectly reverse the operation, because some information gets lost during dimensionality reduction. Moreover, while some algorithms (such as PCA) have a simple reverse transformation procedure that can reconstruct a dataset relatively similar to the original, other algorithms (such as t-SNE) do not.

3. PCA can be used to significantly reduce the dimensionality of most datasets, even if they are highly nonlinear, because it can at least get rid of useless dimensions. However, if there are no useless dimensions—as in the Swiss roll dataset—then reducing dimensionality with PCA will lose too much information. You want to unroll the Swiss roll, not squash it.

4. That's a trick question: it depends on the dataset. Let's look at two extreme examples. First, suppose the dataset is composed of points that are almost perfectly aligned. In this case, PCA can reduce the dataset down to just one dimension while still preserving 95% of the variance. Now imagine that the dataset is composed of perfectly random points, scattered all around the 1,000 dimensions. In this case roughly 950 dimensions are required to preserve 95% of the variance. So the answer is, it depends on the dataset, and it could be any number between 1 and 950. Plotting the explained variance as a function of the number of dimensions is one way to get a rough idea of the dataset's intrinsic dimensionality.

5. Intuitively, a dimensionality reduction algorithm performs well if it eliminates a lot of dimensions from the dataset without losing too much information. One way to measure this is to apply the reverse transformation and measure the reconstruction error. However, not all dimensionality reduction algorithms provide a reverse transformation. Alternatively, if you are using dimensionality reduction as a preprocessing step before another Machine Learning algorithm (e.g., a Random Forest classifier), then you can simply measure the performance of that second algorithm; if dimensionality reduction did not lose too much information, then the algorithm should perform just as well as when using the original dataset.

Applied Question

The solutions for Python & R are below. The Python solution (shown first) is taken from the textbook, and it was converted/adapted to the R solution (shown second).

Python

Exercise: Load the MNIST dataset

```
import pandas as pd

mnist_train = pd.read_csv("mnist_small_train.csv")
X_train = mnist_train.drop("label", axis=1)
y_train = mnist_train["label"]

mnist_test = pd.read_csv("mnist_test.csv")
X_test = mnist_test.drop("label", axis=1)
y_test = mnist_test["label"]
```

Exercise: Train a Random Forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set.

```
from sklearn.ensemble import RandomForestClassifier
from timeit import default_timer as timer

rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)

start = timer()
rnd_clf.fit(X_train, y_train);
print(timer() - start)
```

```
3.215074209000022
```

```
from sklearn.metrics import accuracy_score

y_pred = rnd_clf.predict(X_test)
accuracy_score(y_test, y_pred)
```

```
0.9505
```

Exercise: Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95%.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=0.95)
X_train_reduced = pca.fit_transform(X_train)
```

Exercise: Train a new Random Forest classifier on the reduced dataset and see how long it takes. Was training much faster?

```
rnd_clf_with_pca = RandomForestClassifier(n_estimators=100, random_state=42)

start = timer()
rnd_clf_with_pca.fit(X_train_reduced, y_train);
print(timer() - start)
```

```
8.52665279100006
```

Oh no! Training is actually about twice slower now! How can that be? Well, as we saw in this chapter, dimensionality reduction does not always lead to faster training time: it depends on the dataset, the model and the training algorithm. See Figure 8-6. If you try SGDClassifier instead of RandomForestClassifier, you will find that training time is reduced by a factor of 5 when using PCA. Actually, we will do this in a second, but first let's check the precision of the new random forest classifier.

Exercise: Next evaluate the classifier on the test set: how does it compare to the previous classifier?

```
X_test_reduced = pca.transform(X_test)

y_pred = rnd_clf_with_pca.predict(X_test_reduced)
accuracy_score(y_test, y_pred)
```

```
0.9129
```

Exercise: Try again with an SGDClassifier. How much does PCA help now?

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)

start = timer()
sgd_clf.fit(X_train, y_train);
print(timer() - start)
```

```
4.131636959000161
```

```
y_pred = sgd_clf.predict(X_test)
accuracy_score(y_test, y_pred)
```

```
0.8919
```

Okay, so the SGDClassifier takes much longer to train on this dataset than the RandomForestClassifier, plus it performs worse on the test set. But that's not what we are interested in right now, we want to see how much PCA can help SGDClassifier. Let's train it using the reduced dataset:

```
sgd_clf_with_pca = SGDClassifier(random_state=42)
start = timer()
sgd_clf_with_pca.fit(X_train_reduced, y_train);
print(timer() - start)
```

```
0.8716932079998969
```

Nice! Reducing dimensionality led to roughly 5× speedup. :) Let's check the model's accuracy:

```
y_pred = sgd_clf_with_pca.predict(X_test_reduced)
accuracy_score(y_test, y_pred)
```

```
0.8965
```

Great! PCA not only gave us a roughly 5x speed boost, it also improved performance slightly.

So there you have it: PCA can give you a formidable speedup, and if you're lucky a performance boost... but it's really not guaranteed: it depends on the model and the dataset!

R

```
library(randomForest)
library(rbenchmark)
library(caret)
library(kknn)
```

Exercise: Load the MNIST dataset

```
mnist_train <- read.csv("mnist_small_train.csv")
X_train <- mnist_train[, -1]
y_train <- as.factor(mnist_train[, "label"])

mnist_test <- read.csv("mnist_test.csv")
X_test <- mnist_test[, -1]
y_test <- as.factor(mnist_test[, "label"])

# Some of the columns have zero variance, so we remove them
nonzero_var_cols <- which(apply(X_train, 2, var) != 0)
X_train <- X_train[, nonzero_var_cols]
X_test <- X_test[, nonzero_var_cols]
```

Exercise: Train a Random Forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set.

```
benchmark(
  rnd_clf <- randomForest(X_train, y_train,
                          ntree=20, seed=42, maxnodes=100),
  replications=1)
```

test	replications	elapsed	relative	user.self	sys.self	user.child	sys.child
rnd_clf <- randomForest(X_train, y_train, ntree = 20, seed = 42, maxnodes = 100)	1	5.006	1	4.897	0.031	0	0

```
y_pred <- predict(rnd_clf, X_test)
accuracy_score <- sum(y_pred == y_test) / length(y_test)
accuracy_score
```

```
[1] 0.8879
```

Exercise: Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95%.

```
pca <- preProcess(X_train, method=c("pca"), thresh=0.95)
X_train_reduced <- predict(pca, X_train)
X_test_reduced <- predict(pca, X_test)
ncol(X_train_reduced)
```

```
[1] 284
```

Exercise: Train a new Random Forest classifier on the reduced dataset and see how long it takes. Was training much faster?

```
benchmark(
  rnd_clf_with_pca <- randomForest(X_train_reduced, y_train,
                                   ntree=20, seed=42, maxnodes=100),
  replications=1
)
```

test	replications	elapsed	relative	user.self	sys.self	user.child	sys.child
rnd_clf_with_pca <- randomForest(X_train_reduced, y_train, ntree = 20, seed = 42, maxnodes = 100)	1	2.126	1	2.112	0.007	0	0

Oh no! Training time is ~~actually about twice slower now!~~ not hugely different. How can that be? Well, as we saw in this chapter, dimensionality reduction does not always lead to faster training time: it depends on the dataset, the model and the training algorithm. See Figure 8-6. If you try KNN instead of RandomForest, you will find that training time is reduced by a factor of 2 when using PCA. Actually, we will do this in a second, but first let's check the precision of the new random forest classifier.

Exercise: Next evaluate the classifier on the test set: how does it compare to the previous classifier?

```
y_pred <- predict(rnd_clf_with_pca, X_test_reduced)
accuracy_score <- sum(y_pred == y_test) / length(y_test)
accuracy_score
```

```
[1] 0.7971
```

It is common for performance to drop slightly when reducing dimensionality, because we do lose some potentially useful signal in the process. However, the performance drop is rather severe in this case. So PCA really did not help: it slowed down training and reduced performance.

Exercise: Try again with an KNNClassifier. How much does PCA help now?

```r
# The train.kknn function takes a formula as input
df_train <- data.frame(y = y_train, X_train)[1:5000,]

# Fit the model
benchmark(
  knn_model <- train.kknn(y ~ ., data = df_train, ks=1:4),
  replications=1
)
```

test	replications	elapsed	relative	user.self	sys.self	user.child	sys.child
knn_model <- train.kknn(y ~ ., data = df_train, ks = 1:4)	1	10.546	1	10.485	0.037	0	0

```r
# You can then predict with this model using the test data like so:
df_test <- data.frame(X_test)
predictions <- predict(knn_model, newdata = df_test)
accuracy_score <- sum(predictions == y_test) / length(y_test)
accuracy_score
```

```
[1] 0.8952
```

Okay, so the KNNClassifier takes much longer to train on this dataset than the RandomForestClassifier, plus it performs worse on the test set. But that's not what we are interested in right now, we want to see how much PCA can help KNNClassifier.

Exercise: Let's train it using the reduced dataset:

```r
df_train <- data.frame(y = y_train, X_train_reduced)[1:5000,]

benchmark(
  knn_model <- train.kknn(y ~ ., data = df_train, ks=1:4),
  replications=1
)
```

test	replications	elapsed	relative	user.self	sys.self	user.child	sys.child
knn_model <- train.kknn(y ~ ., data = df_train, ks = 1:4)	1	5.534	1	5.493	0.022	0	0

```r
df_test <- data.frame(X_test_reduced)
predictions <- predict(knn_model, newdata = df_test)
accuracy_score <- sum(predictions == y_test) / length(y_test)
accuracy_score
```

```
[1] 0.7935
```

Great! PCA gave us speed boost, and performance was similar.

So there you have it: PCA can give you a formidable speedup, and if you're lucky a performance boost... but it's really not guaranteed: it depends on the model and the dataset!

References

Géron, A. (2022). *Hands-on machine learning with scikit-learn, keras, and TensorFlow* (3rd ed.). O'Reilly Media, Inc.