MTAT.03.227 Machine Learning (Spring 2015)
Exercise session XV: Kernel Methods

Konstantin Tretyakov

May 26, 2015

The aim of this exercise session is to get acquainted with the idea of kernel methods. As usual, for all exercises you need to write a brief explanation and, for most of them, also a short piece of code demonstrating the result. You can submit your whole solution as a single *decently commented* R file, provided it is sequentially readable and executable.

We shall use the `kernlab` R package. Install it using `install.packages`. *kernlab*

```
install.packages("kernlab")
library(kernlab)
```

We shall be working with the collection of Reuters articles from 1987 – one of the famous benchmark datasets in the text mining field. Load it[1], using the `load.data` function in the provided base code `kernels_base.R`. *Reuters dataset*

```
reuters = load.data()
```

The resulting data frame contains 500 news items on two topics – *crude oil* and *grain*[2]. The news item text is in the column `Content` and its category is in the column `Topic`. In addition, column `y` is equal to $+1$ whenever `Topic == "grain"` and -1 otherwise. Let us split the dataset into training and test folds of size 300 and 200:

```
reuters_train = reuters[1:300,]
reuters_test = reuters[301:500,]
```

Our first goal will be to train and evaluate a classifier for the two types of articles.

String Kernels

To deal with textual data we need to use a *string kernel*. Several such kernels are implemented in the `stringdot` method of the `kernlab` package. We shall *String kernel*

[1] The invocation of `load.data` downloads the data, so you need to be online. The data file is 7.4M, so it might take a while.

[2] I actually provide you with a larger dataset to play with, if you wish. The data is getting trimmed to just two topics and 500 items in the `load.data` function

use the simplest one – the *p-spectrum* kernel. The feature mapping for this kernel represents the string a multiset of its substrings of length p. E.g. for $p = 2$ we have

$$\phi(\text{“ababc”}) \to \{\text{“ab”} \to 2, \text{“ba”} \to 1, \text{“bc”} \to 1, \text{other} \to 0\}.$$

Using the `kernlab` package, a p-spectrum kernel is created as follows:

```
k = stringdot("spectrum", length=2, normalized=F)
```

It can then be applied to pairs of strings:

```
k("first string", "second string")
```

We can also compute the complete *kernel matrix* for a list of strings:

```
K = kernelMatrix(k, list_of_strings)
```

Exercise 1* (1pt). Study the implementation of the p-spectrum kernel in `kernlab`. Choose a small p (e.g. 2) and try applying k to strings like ‘aa’, ‘aaa’, etc. Does the result correspond to your expectations? If not, guess the reasons for the observed differences and explain. *p-spectrum*

Solution. It is easy to see that the output does not exactly correspond to what we should expect:

```
> k = stringdot("spectrum", length=2, normalized=F)
> k("aa", "aa") # Expecting output 1
[1] 2
> k("aaa", "aaa") # Expecting output 4
[1] 5
> k("aa", "baa") # Expecting output 1
[1] 2
> k("a", "aa") # Expecting output 0
[1] 1
```

The first guess might be, that the kernel function somewhy adds 1 to the correct value. This is not true, though, because:

```
> k("aa", "aab") # Expecting output 1
[1] 1
```

Some more experimentation should convince you that what is happening is the following: the string kernel function appends an implicit “end-of-line” character to the end of each string and uses it when computing the kernel. Thus, `"aa"` is internally represented as `"aa$"`, where `$` is an end of line character. The corresponding two-character substrings are then `aa` and `a$`.

For most text classification tasks a *normalized* kernel performs better. A normalized version k_{norm} of any kernel k can be obtained as

$$k_{\text{norm}}(x, y) = \frac{k(x, y)}{\sqrt{k(x, x)k(y, y)}},$$

but using `kernlab` it is sufficient to specify[3] `normalized=T` in `stringdot`.

Kernel matrices

Exercise 2 (0.5pt). In the following, we shall need two matrices:

- The 300×300 *training* kernel matrix $\mathbf{K}$, such that $\mathbf{K}_{(i,j)} = k(\mathbf{x}_i, \mathbf{x}_j)$, where $\mathbf{x}_i$ and $\mathbf{x}_j$ are i-th and j-th elements of `reuters_train`.
- The 300×200 *train-vs-test* kernel matrix $\mathbf{K}_{\text{test}}$ such that $\mathbf{K}_{\text{test}(i,j)} = k(\mathbf{x}_i, \mathbf{x}'_j)$, where $\mathbf{x}'_j$ is the j-th element of `reuters_test`.

Use `stringdot` and `kernelMatrix` to compute those two matrices. Use a normalized 5-spectrum string kernel. Name the corresponding variables `K` and `K_test`.

Solution. The necessary computation is simply:

```
k = stringdot("spectrum", length=5)
K = kernelMatrix(k, reuters_train$Content)
K_test = kernelMatrix(k, reuters_train$Content,
                         reuters_test$Content)
```

It takes some time to compute.

Kernel Classification

A kernel classifier is a function of the form:

$$f(\mathbf{x}) = \sum_i \alpha_i k(\mathbf{x}_i, \mathbf{x}) + b \tag{1}$$

where α_i is the *dual representation* of the classifier normal $\mathbf{w}$ in the (possibly high-dimensional) feature space.

Dual confusion

Exercise 3 (0.5pt). Suppose you have trained an SVM classifier and obtained the SVM parameters α'_i from the SVM dual optimization. Are those α'_i also the dual representation of $\mathbf{w}$ as defined in the paragraph above?

[3] In fact, it is the default option.

Solution. No, not exactly. In the dual form of the SVM optimization problem, the parameter vector $\boldsymbol{\alpha}'$ is used to reconstruct the weight vector $\mathbf{w}$ as follows:

$$\mathbf{w} = \sum_i \alpha'_i y_i \mathbf{x}_i.$$

whereas in the dual representation, the relationship between $\boldsymbol{\alpha}$ and $\mathbf{w}$ should be:

$$\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i.$$

Consequently, in order to obtain the dual representation from the SVM solution $\boldsymbol{\alpha}'$, we need to multiply it elementwise by $\mathbf{y}$:

$$\boldsymbol{\alpha} = \boldsymbol{\alpha}' \circ \mathbf{y}.$$

This can be confusing, given the use of the term "dual" for both cases.

Kernel classifier

Exercise 4 (2pt). Suppose we trained a kernel classifier on our `reuters_train` data. Suppose $b = 0$, $\alpha_1 = 1, \alpha_2 = -1$ and all the remaining α_i values are 0.

1. Evaluate the prediction of such classifier on two strings: "eat more corn" and "petroleum".
2. Evaluate the prediction of this classifier on all training examples and compute its prediction accuracy on the training set. Hint: make use of the matrix $\mathbf{K}$ you computed recently, don't recompute the kernel values!
3. Evaluate the prediction of this classifier on all test examples and compute its prediction accuracy on the test set. Hint: make use of the matrix $\mathbf{K}_{\text{test}}$ you computed recently.
4. In general, suppose you have a "training" kernel $\mathbf{K}$ and a kernel-based classifier with parameters $(\boldsymbol{\alpha}, b)$, that was trained on this data. Write a matrix expression which computes the predictions of the classifier for *all* training examples as a single vector.
5. Similarly, write a single matrix expression, which computes predictions of the classifier $(\boldsymbol{\alpha}, b)$ for all test examples as a single vector.

Solution.

1. Substituting $b = 0$ and the two nonzero α_i values into Equation (1), we obtain

   $$f(\mathbf{x}) = \alpha_1 \cdot k(\mathbf{x}_1, \mathbf{x}) + \alpha_2 \cdot k(\mathbf{x}_2, \mathbf{x}) + b = k(\mathbf{x}_1, \mathbf{x}) - k(\mathbf{x}_2, \mathbf{x}).$$

   Evaluating this on our data:

```
f = function(x) {
   k(reuters_train$Content[1],x)
     - k(reuters_train$Content[2],x)
}
f("eat more corn") # 0.03582489, i.e. correctly classified
                   # as "grain"
f("petroleum")     # -0.08271528, i.e. correctly classified
                   # as "crude oil"
```

2. Observe that prediction of our classifier functional f for the training example $\mathbf{x}_j$ is:

$$\hat{y}_j = k(\mathbf{x}_1, \mathbf{x}_j) - k(\mathbf{x}_2, \mathbf{x}_j) = \mathbf{K}_{1,j} - \mathbf{K}_{2,j},$$

i.e. the difference of the first and second elements of the j-th column of the kernel matrix $\mathbf{K}$. Thus, predictions for all training elements can be obtained by operating on the whole rows of $\mathbf{K}$:

```
yhat = K[1,] - K[2,]
sum(sign(yhat) == reuters_train$y)/300 # 0.7333333 accuracy
```

3. Similarly, the predictions for the test instances can be computed by operating on the rows of the matrix $\mathbf{K}_{\text{text}}$:

```
yhat = K_test[1,] - K_test[2,]
sum(sign(yhat) == reuters_test$y)/300 # 0.745 accuracy
```

Note that the results are actually unexpectedly good, given the unsophisticated nature of the classifier we are using.

4. In general, observe that prediction of the classifier functional f for the training example $\mathbf{x}_j$ is:

$$\hat{y}_j = \sum_i \alpha_i k(\mathbf{x}_i, \mathbf{x}_j) + b = \sum_i \alpha_i \mathbf{K}_{ij} + b,$$

which, in matrix form for all j is simply:

$$\hat{\mathbf{y}} = \mathbf{K}^T \boldsymbol{\alpha} + b.$$

5. Similarly, the prediction for all test instances can be computed as:

$$\hat{\mathbf{y}}_{\text{test}} = \mathbf{K}_{\text{test}}^T \boldsymbol{\alpha} + b.$$

Exercise 5 (1pt). The base code `kernels_base.R` provides you with a bare implementation of the kernelized perceptron algorithm. Some statements are missing, however. Fix the implementation to have a working algorithm. Then train the algorithm on the training set and evaluate its performance on the test set.

Kernel perceptron

Solution. The complete algorithm is presented below, with the fixed lines highlighted:

```
kernel_perceptron = function(K, y) {
  alpha = rep(0, nrow(K))
  b = 0
  while (TRUE) {
    # Make predictions for all points in the training sample
    predictions = t(K) %*% alpha + b

    # Find misclassified instances
    misclassified = which(sign(predictions) != y)

    # If no misclassified, we're done!
    if (length(misclassified) == 0) { break; }

    # If something is misclassified, pick the first element
    i = misclassified[1]

    # Update parameters
    alpha[i] = alpha[i] + y[i]
    b = b + y[i]
  }

  # Return result
  result = list(alpha, b)
  names(result) = c("alpha", "b")
  result
}
```

Training and testing the algorithm:

```
kp = kernel_perceptron(K, reuters_train$y)
yhat = t(K_test) %*% kp$alpha + kp$b
sum(sign(yhat) == reuters_test$y)/200 # 0.96
```

Note that the algorithm converged very quickly on a well-performing and sparse solution (examine the solution vector `kp$alpha` and the highly-weighed support vectors). For more complicated problems, however, relying on a non-regularized learning procedure such as the perceptron, may not always be the best idea to use with kernels.

---

ksvm

Exercise 6 (1pt). The `kernlab` package has its own implementation of most popular kernel methods, and SVM in particular.

1. Use the `ksvm` function with the kernel matrix $\mathbf{K}$ to train an SVM classifier using the chosen kernel.

2. Evaluate the performance of the resulting algorithm on the test set. Hint: chances are high you will have troubles if you try to use the `predict` method to apply the model on the test set. Instead just take the raw alpha values (`model@alpha`, `model@alphaindex`) and compute the model predictions manually.

Solution. The use of the `ksvm` method is fairly straightforward:

```
model = ksvm(K, reuters_train$y, C=1)
yhat = t(K_test[model@alphaindex,]) %*% model@alpha - model@b
sum(sign(yhat) == reuters_test$y)/200
```

We achieved perfect test performance even without the need to tune the C parameter. But again, note that things are not always as simple.

Regularization

Exercise 7* (1pt). An important issue in kernel methods is overfitting. Indeed, when the feature space is infinite-dimensional, it is easy to achieve perfect performance on *any* training set which often leads to severe overfitting. This is best seen on a kernelized formulation of ordinary least squares regression.

Derive the kernelized version of ordinary least squares and show that if the kernel is positive definite, it is indeed possible to classify any training set perfectly. An important consequence of this is that you should generally avoid using kernel methods without regularization[4]

Hint. The coefficient vector $\mathbf{w}$ for the linear regression problem is the least-squares solution to $\mathbf{X}\mathbf{w} = \mathbf{y}$. Rewrite this condition in dual form by replacing $\mathbf{X}$ with the higher-dimensional data matrix $\boldsymbol{\Xi}$ and substituting the dual representation $\mathbf{w} = \boldsymbol{\Xi}^T\boldsymbol{\alpha}$ for the weight vector. The positive definiteness of the kernel matrix is equivalent to its invertibility.

Solution. As suggested in the hint, let us first re-denote $\mathbf{X}$ as $\boldsymbol{\Xi}$. Note that this is purely for convenience, to signify that the actual "data vectors" are not the original $\mathbf{x}_i$ any more, but the feature vectors $\phi(\mathbf{x}_i)$. The linear regression equation now takes the form:

$$\boldsymbol{\Xi}\mathbf{w} = \mathbf{y}.$$

Now, substituting the dual representation $\mathbf{w} = \boldsymbol{\Xi}^T\boldsymbol{\alpha}$ we get:

$$\boldsymbol{\Xi}\boldsymbol{\Xi}^T\boldsymbol{\alpha} = \mathbf{y},$$

which is simply

$$\mathbf{K}\boldsymbol{\alpha} = \mathbf{y}.$$

[4]However, as we have just seen in the perceptron example, they can sometimes work well none the less despite the lack of regularization. It is also customary to close eyes on the regularization issue when performing unsupervised learning, as we do in the following section.

Now, $\mathbf{K}$ is a square matrix. Moreover, if it is positive definite (which will be the case for most common nonlinear kernels, given there are no repeating points in the training set), it will also be invertible. Consequently, the solution vector

$$\hat{\boldsymbol{\alpha}} = \mathbf{K}^{-1}\mathbf{y}$$

results in a regression function that predicts *every* training point's response y_i with zero error. This usually indicates severe overfitting.

Kernel PCA

Principal Components Analysis (PCA) is a method for finding linear "components" in the data. It typically proceeds as follows:

1. First, center[5] the data matrix $\mathbf{X}$.
2. Solve $\mathbf{X}^T\mathbf{X}\mathbf{v} = \lambda\mathbf{v}$ for $\mathbf{v}$ (by finding the eigenvalues of $\mathbf{X}^T\mathbf{X}$).
3. Project the data onto the largest eigenvalue(s): $\mathbf{p} = \mathbf{X}\mathbf{v}$.

As all linear methods, PCA may be performed in the higher-dimensional feature space using only the dual representation. This allows to extract non-obvious non-linear patterns from the data as well provide nice visualizations.

To kernelize PCA[6] we denote the hypothetical high-dimensional data matrix $\phi(\mathbf{X})$ by $\boldsymbol{\Xi}$. We shall be looking for linear component $\mathbf{v}$ in that space by solving

$$\boldsymbol{\Xi}^T\boldsymbol{\Xi}\mathbf{v} = \lambda\mathbf{v}.$$

However, instead of working directly with $\mathbf{v}$ we substitute its dual representation $\mathbf{v} = \boldsymbol{\Xi}^T\boldsymbol{\alpha}$. Then,

$$\begin{aligned}
\boldsymbol{\Xi}^T\boldsymbol{\Xi}\mathbf{v} &= \lambda\mathbf{v},\\
\boldsymbol{\Xi}^T\boldsymbol{\Xi}\boldsymbol{\Xi}^T\boldsymbol{\alpha} &= \lambda\boldsymbol{\Xi}^T\boldsymbol{\alpha},\\
\boldsymbol{\Xi}\boldsymbol{\Xi}^T\boldsymbol{\Xi}\boldsymbol{\Xi}^T\boldsymbol{\alpha} &= \lambda\boldsymbol{\Xi}\boldsymbol{\Xi}^T\boldsymbol{\alpha},\\
\mathbf{K}\mathbf{K}\boldsymbol{\alpha} &= \lambda\mathbf{K}\boldsymbol{\alpha},\\
\mathbf{K}\boldsymbol{\alpha} &= \lambda\boldsymbol{\alpha}.
\end{aligned}$$

The last operation is obviously legal if $\mathbf{K}$ is positive definite (and thus invertible). Using some careful math it is possible to show that it works out for positive semidefinite $\mathbf{K}$ as well.

Consequently, dual representations of pricipal components are simply the eigenvectors of the kernel matrix.

The projection of the training data to the corresponding eigenvalue is:

$$\mathbf{p} = \boldsymbol{\Xi}\mathbf{v} = \boldsymbol{\Xi}\boldsymbol{\Xi}^T\boldsymbol{\alpha} = \mathbf{K}\boldsymbol{\alpha} = \lambda\boldsymbol{\alpha}.$$

Thus, the Kernel PCA algorithm is:

[5] By the way, PCA often works fine even without centering.

[6] This is not the most formally rigorous derivation, but it's correct none the less.

1. Center the kernel $\mathbf{K}$.
2. Solve $\mathbf{K}\boldsymbol{\alpha} = \lambda\boldsymbol{\alpha}$ (by finding the eigenvalues of $\mathbf{K}$).
3. Project the training data onto the largest eigenvalue(s): $\mathbf{p} = \lambda\boldsymbol{\alpha}$.

Exercise 8 (2pt). Implement Kernel PCA on the Reuters data and visualize the two largest principal components. *Kernel PCA*

Hints.

1. Kernel centering:

$$\mathbf{K}_{\text{cent}} = \mathbf{K} - \frac{1}{n}\mathbf{1}\mathbf{1}^T\mathbf{K} - \frac{1}{n}\mathbf{K}\mathbf{1}\mathbf{1}^T + \frac{1}{n^2}\mathbf{1}\mathbf{1}^T\mathbf{K}\mathbf{1}\mathbf{1}^T.$$

2. Eigenvalue decomposition:

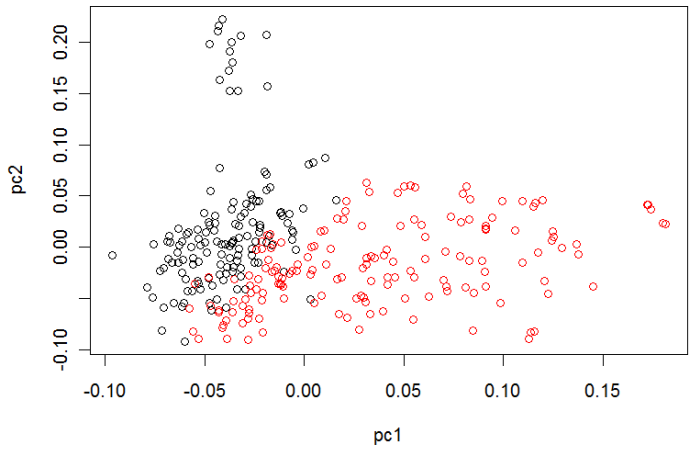
```
e = eigen(K) # The largest eigenvector is in e$vectors[,1]
```

3. For the purposes of visualization you can ignore the multiplication by λ.
4. For a nicer picture, don't forget to color the points according to their class:

```
plot(<pc1>, <pc2>, col=reuters_train$Topic)
```

Solution. The solution is way shorter than the problem statement:

```
n = nrow(K)
ones = matrix(1, n, n)
K_cent = K - ones %*% K/n - K %*% ones/n + ones %*% K %*% ones/n^2
e = eigen(K_cent)
pc1 = e$vectors[,1]
pc2 = e$vectors[,2]
plot(pc1, pc2, col=reuters_train$Topic)
```



The resulting plot demonstrates that in the feature space of our kernel there indeed is a fairly nice separation between the two classes (something we could already observe in our classification results). In addition, it lets us immediately see that there is a separate cluster of "crude oil" articles that stands out. Taking a look at the corresponding elements:

```
i = which(pc2 > 0.1)
reuters_train$Content[i]
```

It is easy to see that those are exactly the announcements on oil price changes.

Exercise 9 (1pt). Naturally, Kernel PCA is implemented in `kernlab`. Use the `kpca` function and visualize the two largest principal components. Do you get the same picture? *kpca*

Solution.

```
kp = kpca(K, features=2)
plot(kp@rotated, col=reuters_train$Topic)
```

The picture is the same, except for a rescaling of the axes.

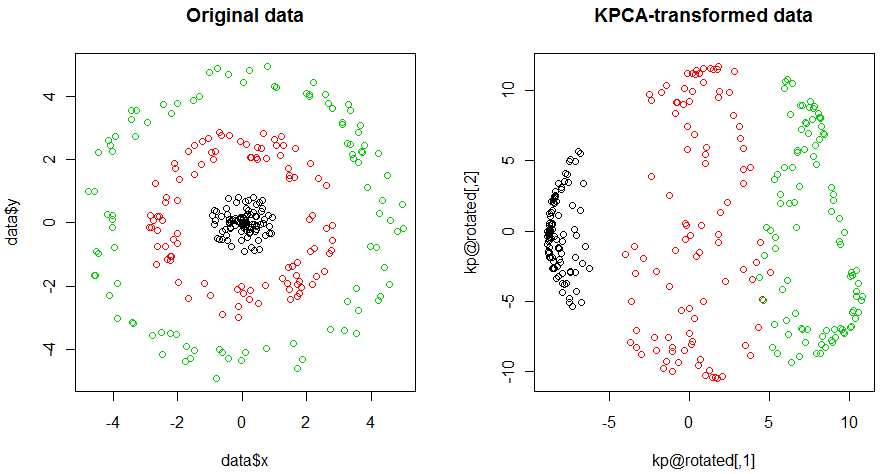
Exercise 10 (1pt). We have just seen the use of Kernel PCA to visualize non-numeric data, but this is not the only exciting feature of KPCA. More importantly, KPCA is capable of discerning *nonlinear* components. To see that, load the dataset produced by the `circle.data` function in the base code: *Nonlinear PCA*

```
data = circle.data()
plot(data$x, data$y, col=data$c)
```

Then apply KPCA to this dataset (use the default `rbfdot` kernel) and visualize the transformed points.

Solution. Let us plot the original data alongside the KPCA-transformed points:

```
data = circle.data()
kp = kpca(~x+y, data=data, features=2)
par(mfrow=c(1,2))
plot(data$x, data$y, col=data$c, main="Original data")
plot(kp@rotated, col=data$c, main="KPCA-transformed data")
```



Note that clusters in the data which were inherently nonlinear as a result of KPCA turned into something linearly discernible.

Exercise 11* (4pt). Derive and implement the kernelized version of the *K-means* algorithm. Run it on the Reuters dataset. Can the clustering separate the two topics? Compare the results to the implementation in `kernlab`.

Kernel K-means

Solution. Let us start by writing the K-means algorithm in its primal form. Let k be the number of clusters desired.

1. Start with k randomly picked elements from the training set. Set them as the current cluster centers:

$$\mathbf{c}_j := \mathbf{x}_{\text{rand}}, \qquad j \in \{1, \ldots, k\}.$$

2. For each element $\mathbf{x}_i$ in the training set, measure its squared distance to each cluster center $\mathbf{c}_j$:
$$d_{ij} = \|\mathbf{x}_i - \mathbf{c}_j\|^2.$$

3. Assign each element to the closest cluster center:
$$a_i = \operatorname{argmin}_j d_{ij},$$
where a_i denotes the cluster id assigned to point $\mathbf{x}_i$.

4. Recompute cluster centers:
$$\mathbf{c}_j = \frac{1}{|\{i,\, a_i = j\}|} \sum_{a_i = j} \mathbf{x}_i.$$

5. If no elements were reassigned in Step 3, stop. Otherwise go to Step 2.

In order to kernelize the algorithm we have to get rid of all explicit uses of the training examples $\mathbf{x}_i$ (those are now feature-mapped examples $\phi(\mathbf{x}_i)$), cluster centers $\mathbf{c}_j$, and replace them with dual representations. Let us consider each step of the algorithm:

1. First, note that the dual representation of each training example $\phi(\mathbf{x}_i)$ is the vector $\mathbf{e}_i$, which has all coordinates set to 0 except for i-th:
$$\mathbf{e}_i = (0, 0, \ldots, \underbrace{1}_{i}, \ldots, 0, 0)^T.$$
Now, let $\boldsymbol{\gamma}_j$ denote the dual representation of $\mathbf{c}_j$. That is,
$$\mathbf{c}_j = \sum_i \gamma_{ji} \mathbf{x}_i.$$
In this case, initialization of $\boldsymbol{\gamma}_j$ is equivalent to picking a random $\mathbf{e}_i$:
$$\boldsymbol{\gamma}_j := \mathbf{e}_{\text{rand}}.$$

2. Distance computation in the feature space can be kernelized, as shown in the lecture:
$$d_{ij} = \|\phi(\mathbf{x}_i) - \mathbf{c}_j\|^2 = (\mathbf{e}_i - \boldsymbol{\gamma}_j)^T \mathbf{K} (\mathbf{e}_i - \boldsymbol{\gamma}_j).$$

3. This step needs no modification.

4. As linear operations in primal representation correspond to the same operations in dual, we have:
$$\boldsymbol{\gamma}_j = \frac{1}{|\{i,\, a_i = j\}|} \sum_{a_i = j} \mathbf{e}_i.$$

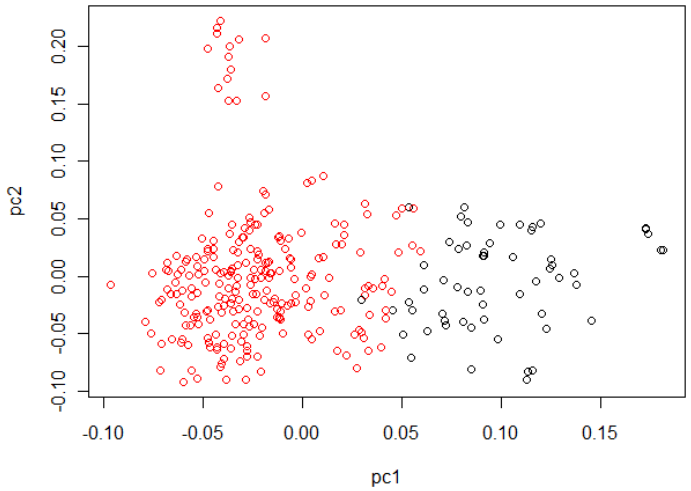
5. This step needs no modification.

Now all what's left is implementing the algorithm. Here's an example implementation:

```
kernel_kmeans = function(K, k) {
  # Variables used:
  #   k - number of desired clusters (given)
  #   K - n x n kernel matrix (given)
  #   n - number of training points
  #   G - n x k matrix of cluster center duals
  #   D - n x k matrix of point-cluster distances
  #   A - n x k assignment matrx:
  #             A[i,j] = 1 if x_i as assigned to c_j
  #   a - n x 1 vector of cluster assignments (returned)
  #   new_a - new assignments computed on this iteration

  n = nrow(K)
  a = matrix(0, n, 1)

  # Step 1
  G = matrix(0, n, k)
  for (i in 1:k) G[sample(1:n, 1), i] = 1

  while(TRUE) {
    # Step 2
    dist = function(i, j) {
      gamma = G[,j]
      gamma[i] = gamma[i] - 1
      t(gamma) %*% K %*% gamma
    }
    D = outer(1:n, 1:k, Vectorize(dist))

    # Step 3
    Dmin = apply(D, 1, min)
    A = (Dmin == D)+0
    new_a = max.col(-D)

    # Step 4 (Note that G is simply column-normalized A)
    G = scale(A, center=FALSE, scale=colSums(A))

    # Step 5
    if (all(new_a == a)) break
    else a = new_a
  }
  a # Return value
}
```

Let us apply this on our data and see how it works:

```
set.seed(1)
clusters = kernel_kmeans(K, 2)
plot(pc1, pc2, col=clusters)
```



The clustering makes sense, but does not correspond to the separation by topic.

Finally, we could have just used the `kkmeans` method from `kernlab`, which is as simple as:

```
clusters = kkmeans(K, 2)
```

It works considerably faster (at least in comparison to the the sample implementation above, which is far from being efficient), and produces a similar result in general.
