demo_kernel_approximation

March 4, 2021

Luis Antonio Ortega Andrés
Antonio Coín Castro

1 Approximation of the kernel matrix

Author: alberto.suarez@uam.es

In this notebook we illustrate the quality of the approximation to the kernel matrix using random features of different types (RBF, Matérn) and Nyström features.

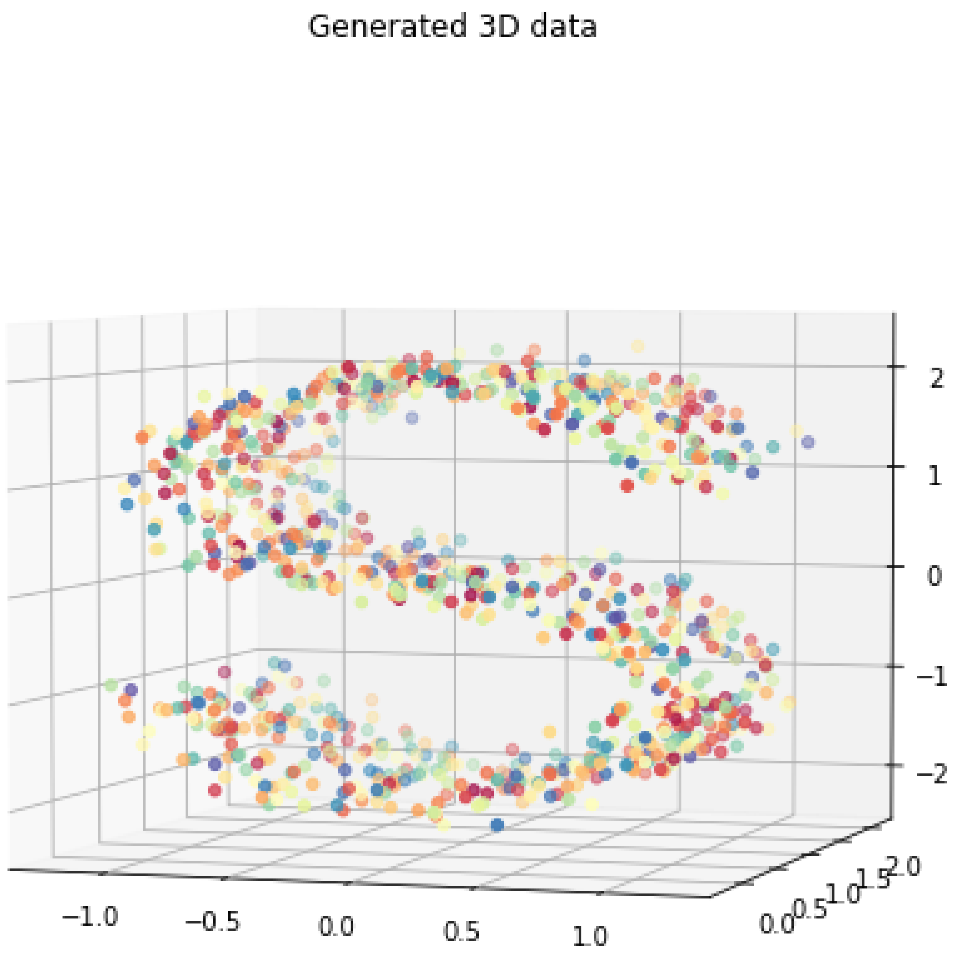
Note that even the best results for classification need not be obtained by the method that gives the best approximation to the kernel matrix. The reason is that the approximation can have a regularization effect that may improve the accuracy of the predictions on the test set.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.metrics.pairwise import rbf_kernel
from sklearn import datasets
from sklearn.kernel_approximation import RBFSampler
from sklearn.gaussian_process.kernels import Matern

import kernel_approximation as ka

%load_ext autoreload
%autoreload 2

seed = 0
np.random.seed(seed)  # for reproducible results
```

1.1 Generate dataset

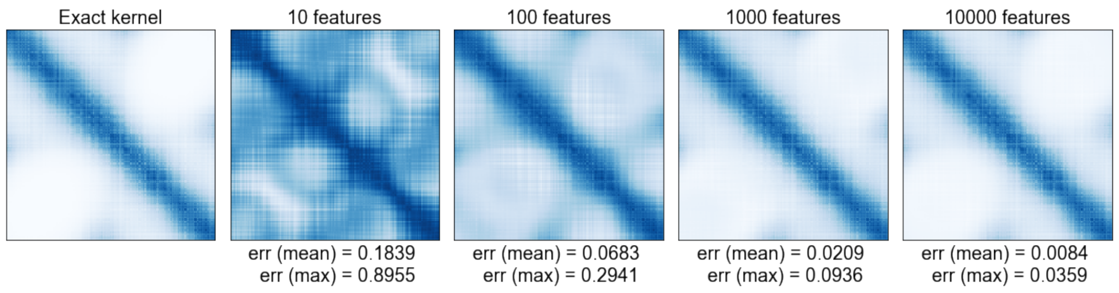
[2]:
```
# Generate 3D data
n_instances = 1000
X, color = datasets.make_s_curve(n_instances,
                                 noise=0.1,
                                 random_state=seed)
X = X[np.argsort(color)]
```

```python
# Visualize dataset
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)
ax.view_init(4, -72)
ax.set_title("Generated 3D data")
plt.show()
```



1.2 Random features with RBF kernel

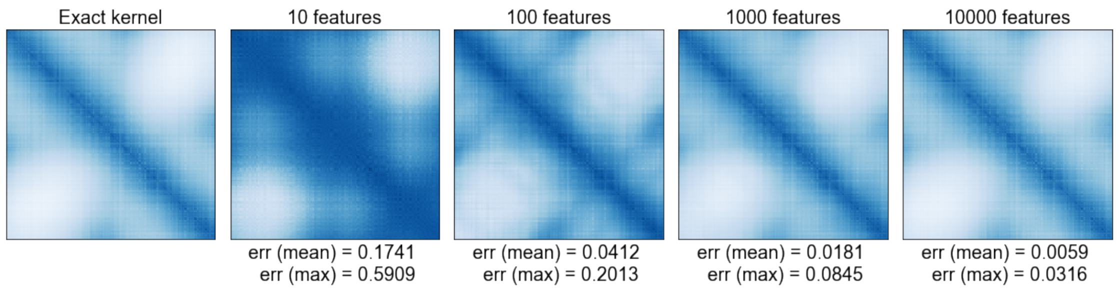
[3]:
```
# RBF kernel parameters
sigma = 1.0
gamma = 1.0/(2.0*sigma**2)

# Kernel function
def kernel(X, Y):
    return rbf_kernel(X, Y, gamma=gamma)


# Create an instance of the random features object
rbf_sampler = ka.RandomFeaturesSamplerRBF(
    sigma=sigma,
    random_state=seed
)

# Plot the approximation to the kernel matrix

n_random_features = [10, 100, 1000, 10000]

ka.demo_kernel_approximation_features(
    X,
    kernel,
    rbf_sampler,
    n_random_features
)
```



Looking at the results, we observe that incresing the number of features sampled decreases the resulting error (in mean and maximum value) up to a sufficiently good value. Using only 100 features (of 1 million available) reduces the mean error up to 0.06, and using 1% of the total features (i.e. 10000) further reduces it to 0.008, which as we can see in the last graph, is a very good approximation. This is to be expected, as we know theoretically that the approximated kernel coincides with the original kernel in the limit of `n_features` $\to \infty$.

1.3 Random features with Matérn kernel

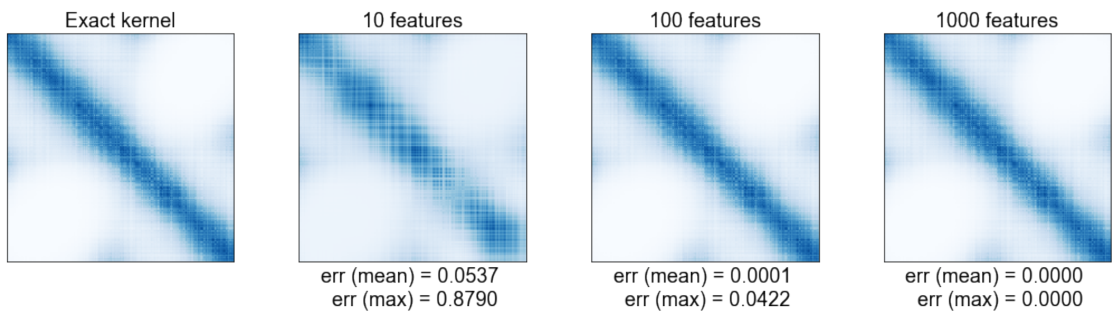
[4]:
```
# Matérn kernel parameters
length_scale = 3.0
nu = 0.5

# Kernel function
kernel_matern = Matern(length_scale=length_scale, nu=nu)

def kernel(X, Y):
    return kernel_matern(X, Y)


# Create an instance of the random features object
matern_sampler = ka.RandomFeaturesSamplerMatern(
    scale=length_scale,
    nu=nu,
    random_state=seed,
)

# Plot the approximation to the kernel matrix

n_random_features = [10, 100, 1000, 10000]

ka.demo_kernel_approximation_features(
    X,
    kernel,
    matern_sampler,
    n_random_features
)
```



In this case, the usage of the Matérn kernel enhances the performance of the kernel method, leading to better results compared to the RBF kernel for every value of `n_features` tried. As it happened before, the approximation error decreases when whe increase the number of random features, but in this case the error seems to be decreasing a bit faster.

1.4 Nyström features with RBF kernel

[5]:
```python
# Kernel parameters
sigma = 1.0
gamma = 1.0/(2.0*sigma**2)

# Kernel function
def kernel(X, Y):
    return rbf_kernel(X, Y, gamma=gamma)


# Create an instance of the Nyström features object
nystroem_sampler = ka.NystroemFeaturesSampler(
    kernel=kernel,
    random_state=seed,
)

# Plot the approximation to the kernel matrix

n_nystroem_features = [10, 100, 1000]

ka.demo_kernel_approximation_features(
    X,
    kernel,
    nystroem_sampler,
    n_nystroem_features
)
```



As we can see, in this case the error also decreases as we increase the number of features, which is limited by the total number of features of the data. In order to approximately compare the performance of this method given a fixed number of features, we look at the square of that number of features on the previous methods. For example, using 10 features leads to a mean error of 0.053, which is of the same order of magnitude of the results obtained with the previous methods using 100 features.

As a final observation, using 1000 features leads to a perfectly reconstructed kernel, which is obvious, since we are using all the features of the training set, and we are essentially not doing any approximation at all.

1.4.1 Approximation of unseen points

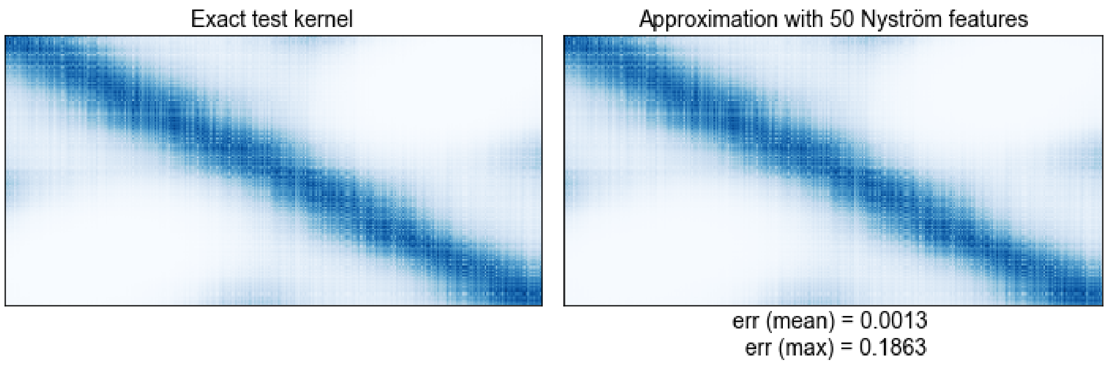
We compute an approximation of the kernel matrix $k(X_{test}, X)$ using the fitted Nyström features from the data matrix X. Specifically, we have:

$$\hat{k}(X_{test}, X) = \Phi(X_{test})\Phi(X)^T = (k(X_{test}, X_J)(W^+)^{1/2})((W^+)^{1/2}k(X_J, X)),$$

where X_J is the subset of X that contains only the J randomly sampled columns.

[6]:
```python
# Generate test data
n_instances = 500
Xt, color_t = datasets.make_s_curve(n_instances,
                                    noise=0.05,
                                    random_state=seed)
Xt = Xt[np.argsort(color_t)]

# Compute genuine kernel (test) matrix
Kt = kernel(Xt, X)

# Compute kernel (test) matrix approximation
n_features = 50
Kt_hat = nystroem_sampler.set_params(
    n_components=n_features
).approximate_kernel_matrix(X, Xt)

# Set plot options
fig, axes = plt.subplots(1, 2)
fig.set_size_inches(10, 7)
font = {'fontname': 'arial', 'fontsize': 14}

# Plot original kernel
kernel_matrix = kernel(X, X)
axes[0].imshow(Kt, cmap=plt.cm.Blues)
axes[0].set_title('Exact test kernel', **font)
axes[0].set_xticks([])
axes[0].set_yticks([])

# Plot approximated kernel and print absolute errors
axes[1].imshow(Kt_hat, cmap=plt.cm.Blues)
axes[1].set_title(
    'Approximation with {} Nyström features'.format(
        n_features
    ), **font)
axes[1].set_xticks([])
axes[1].set_yticks([])
err_approx = Kt - Kt_hat
```

```
err_mean = np.mean(np.abs(err_approx))
err_max = np.max(np.abs(err_approx))
axes[1].set_xlabel(
    'err (mean) = {:.4f} \n err (max) = {:.4f}'.format(
        err_mean,
        err_max
    ), **font)

plt.tight_layout()
plt.show()
```

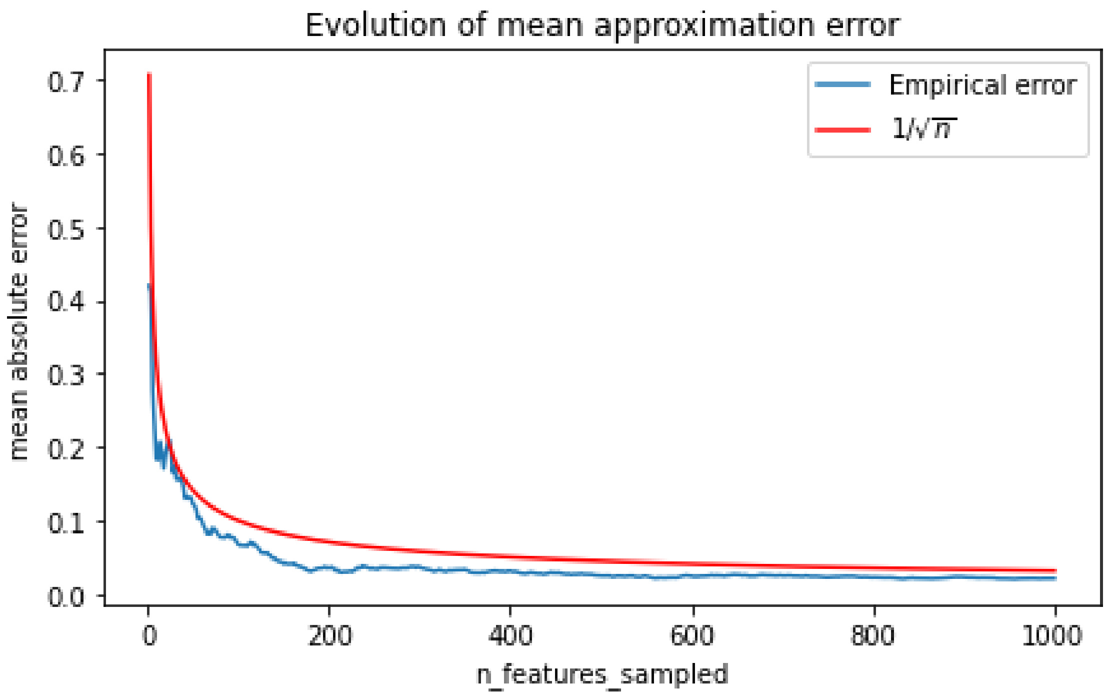


1.4.2 Error curve analysis

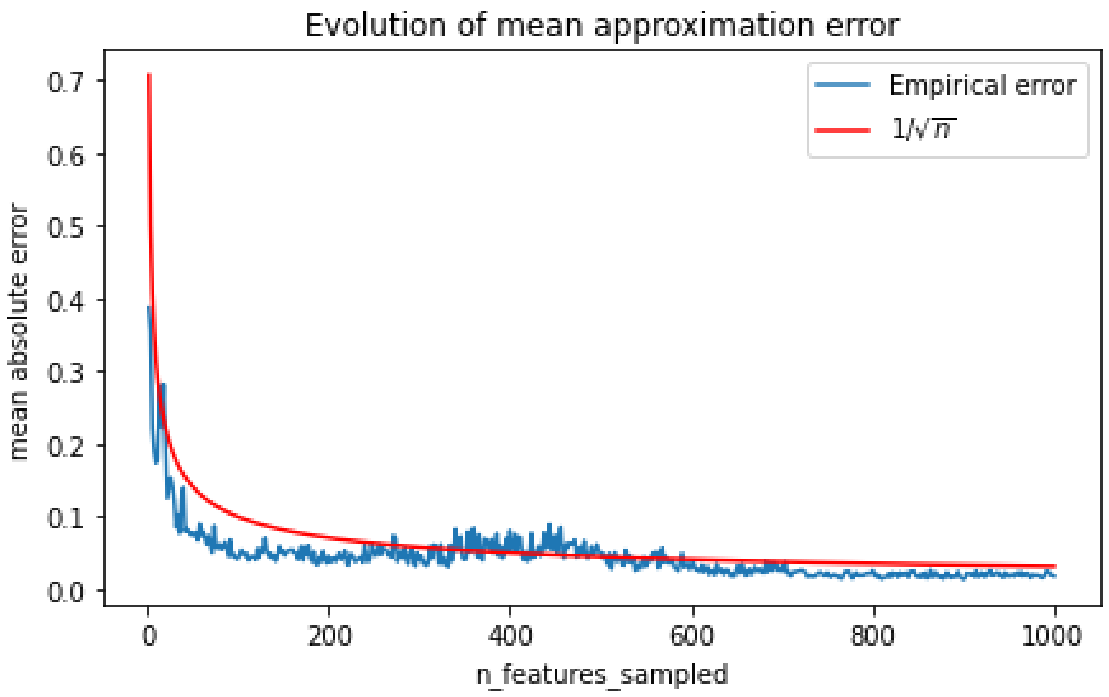
We study the dependence of the mean error with respect to the number of features sampled for the different random feature models.

[7]:
```
# Random Features RBF error analysis

sigma = 1.0
gamma = 1.0/(2.0*sigma**2)

def kernel(X, Y):
    return rbf_kernel(X, Y, gamma=gamma)


features_range = ka.plot_mean_approx_err(
    X,
    kernel,
    rbf_sampler,
    1000
)

plt.plot(features_range, 1.0/np.sqrt(features_range),
         color='red', label=r"$1/\sqrt{n}$")
```

```
plt.legend()
plt.show()
```

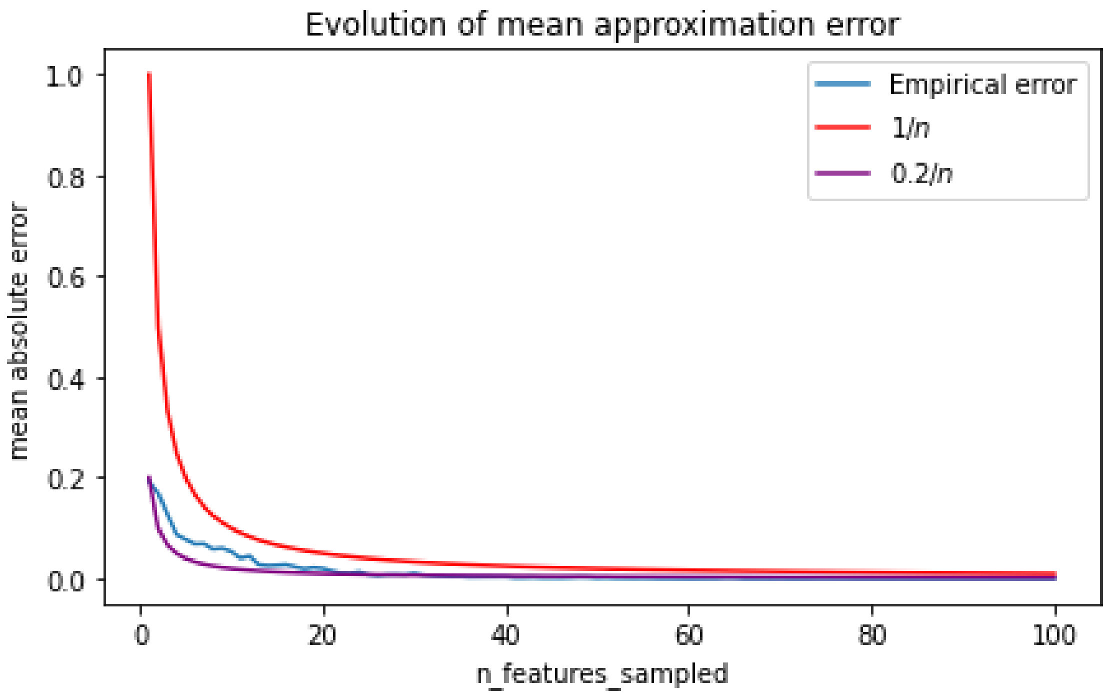


[8]:
```
# Random Features Matérn error analysis

length_scale = 3.0
nu = 0.5
kernel_matern = Matern(length_scale=length_scale, nu=nu)

def kernel(X, Y):
    return kernel_matern(X, Y)


features_range = ka.plot_mean_approx_err(
    X,
    kernel,
    matern_sampler,
    1000
)

plt.plot(features_range, 1.0/np.sqrt(features_range),
         color='red', label=r"$1/\sqrt{n}$")
plt.legend()
plt.show()
```



In these two cases (Random Features), the empirical error curve is compared with $1/\sqrt{n}$, as we know that this is the expected error rate of the subjacent Monte Carlo method, which is the primary source of randomness (and hence, of error). As we can see, in both cases the curve is pretty well adjusted, even though there is some noise.

[9]:
```
# Nyström features RBF error analysis

sigma = 1.0
gamma = 1.0/(2.0*sigma**2)

def kernel(X, Y):
    return rbf_kernel(X, Y, gamma=gamma)


features_range = ka.plot_mean_approx_err(
    X,
    kernel,
    nystroem_sampler,
    100,
    start=1,
    step=1
)

plt.plot(features_range, 1/features_range,
         color='red', label=r"$1/n$")
plt.plot(features_range, 0.2/features_range,
         color='purple', label=r"$0.2/n$")
plt.legend()
plt.show()
```



In this case, no Monte Carlo approximation is used, so the error rate is not necessarily $1/\sqrt{n}$, but instead it must be realted to the error incurred when sampling a subset of columns uniformily at random. We have found empirically that the error decreases as $1/n$, but we have found no theoretical results that support this claim in the little time we had to research. Nonetheless, it seems to fit the error curve reasonably well. We can even make a more precise statement, as we have seen that the proportionality constant is close to 0.2, **independent of the number of samples considered**.