W4995 Applied Machine Learning

Support Vector Machines

02/15/17

Andreas Müller

Motivation

- Go from linear models to more powerful non-linear ones.
- Keep convexity (ease of optimization).
- Generalize the concept of feature engineering.

Reformulate linear models

$$\min_{w \in \mathbb{R}^p} C \sum_{i=1}^{n} \max(0, 1 - y_i w^T \mathbf{x}) + ||w||_2^2$$

$$\hat{y} = \text{sign}(w^T \mathbf{x})$$

FIXME support vector image here!

Reformulate linear models

Optimization Theory:

$$w = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i$$

(alpha are dual coefficients,
Non-zero for support
vectors only)

$$\hat{y} = \text{sign}(w^T \mathbf{x}) \Rightarrow \hat{y} = \text{sign}\left(\sum_{i}^{n} \alpha_i (\mathbf{x}_i^T \mathbf{x})\right)$$

Can formulate learning as optimization over alpha, only involves x via dot-products (x_i^T, x_j)

Regularization parameter C is limit on alphas!

Introducing Kernels

$$\hat{y} = \text{sign}\left(\sum_{i}^{n} \alpha_i (\mathbf{x}_i^T \mathbf{x})\right) \Rightarrow \hat{y} = \text{sign}\left(\sum_{i}^{n} \alpha_i (\phi(\mathbf{x}_i)^T \phi(\mathbf{x}))\right)$$

Dimensionality of \phi “doesn’t matter”. We only ever need to compute dot-product!

$$\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \Rightarrow k(\mathbf{x}_i, \mathbf{x}_j)$$

As long as k is positive definite, symmetric, there exists a phi! (might be infinite-dimensional)

Can now design k instead of \phi – which might be easier, more flexible!

Can be done for any linear model – not only SVM! (svm has some alpha zero).
Kernel Logistic Regression, Kernel PCA, Kernel Kmeans….

Examples of Kernels

$$k_{\text{linear}}(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}'$$

$$k_{\text{poly}}(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T \mathbf{x}' + c)^d$$

$$k_{\text{rbf}}(\mathbf{x}, \mathbf{x}') = \exp(\gamma ||\mathbf{x} - \mathbf{x}'||^2)$$

$$k_{\text{sigmoid}}(\mathbf{x}, \mathbf{x}') = \tanh\left(\gamma \mathbf{x}^T \mathbf{x}' + r\right)$$

$$k_{\cap}(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{p} \min(x_i, x_i')$$

If k, k' are kernels, so are $k + k', kk', ck', ...$

Polynomial Kernel vs Polynomial Features

SVM with poly kernel equivalent to linear SVM with polynomial features!

$$k_{\text{poly}}(\mathbf{x}, \mathbf{x}') = \underbrace{(\mathbf{x}^T\mathbf{x}' + c)^d}_{\text{Single Number! No matter what degree!}}$$

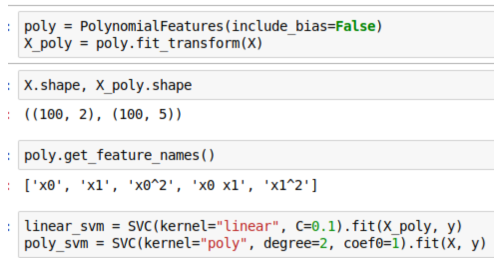
Primal vs dual optimization:

Explicit polynomials => compute on n_features ^ d * n_samples
Kernel trick => compute on kernel matrix of shape n_samples * n_samples

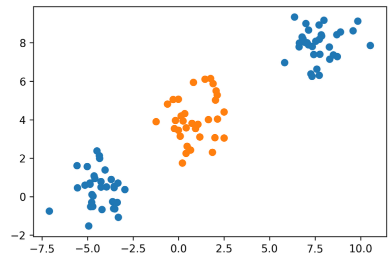
For a single feature, easy to see:

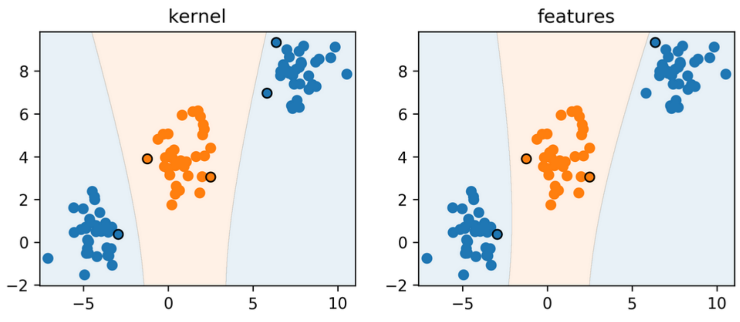
$$(x^2, \sqrt{2}x, 1)^T(x'^2, \sqrt{2}x', 1) = x^2x'^2 + 2xx' + 1 = (xx' + 1)^2$$

Poly kernels with sklearn

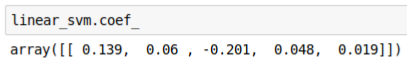


```
: poly = PolynomialFeatures(include_bias=False)
  X_poly = poly.fit_transform(X)

: X.shape, X_poly.shape

: ((100, 2), (100, 5))

: poly.get_feature_names()

: ['x0', 'x1', 'x0^2', 'x0 x1', 'x1^2']

: linear_svm = SVC(kernel="linear", C=0.1).fit(X_poly, y)
  poly_svm = SVC(kernel="poly", degree=2, coef0=1).fit(X, y)
```



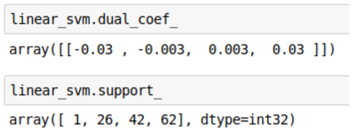


Understanding Dual Coefficients



```
linear_svm.coef_
array([[ 0.139,  0.06 , -0.201,  0.048,  0.019]])
```

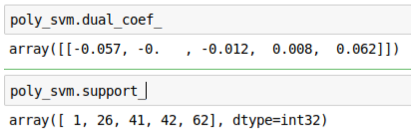
y = sign(0.139 * x[0] + 0.06 * x[1] – 0.201 * x[0] ** 2 + 0.048 * x[0] * x[1] + 0.019 x[1] ** 2)



```
linear_svm.dual_coef_
array([[-0.03 , -0.003,  0.003,  0.03 ]])
```

```
linear_svm.support_
array([ 1, 26, 42, 62], dtype=int32)
```

y = sign(-0.03 * np.inner(poly(X[1]), poly(x)) – 0.003 * np.inner(poly(X[26]), poly(x))
+0.003 * np.inner(poly(X[42]), poly(x)) + 0.03 * np.inner(poly(X[63]), poly(x))

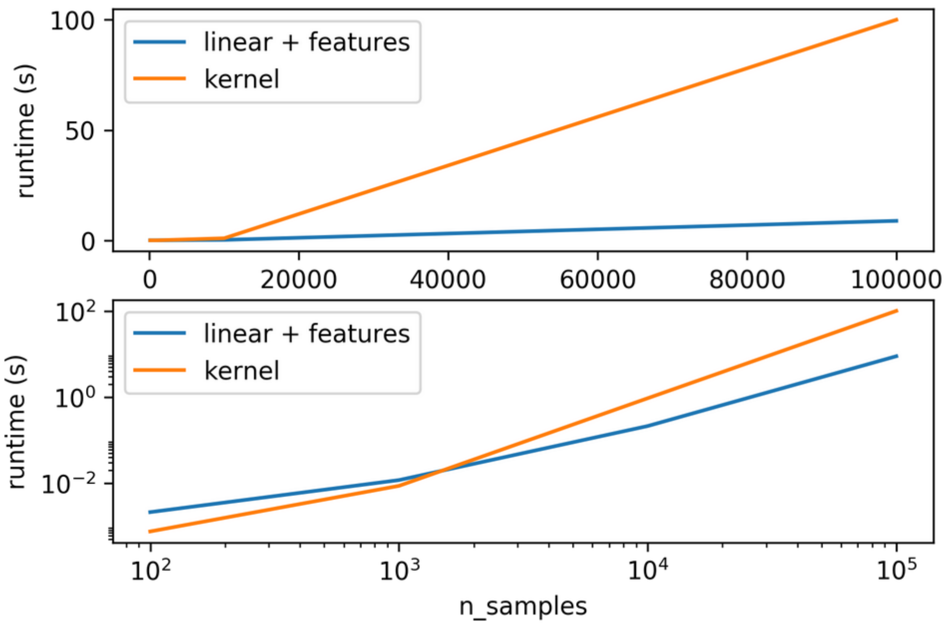


```
poly_svm.dual_coef_
array([[-0.057, -0.   , -0.012,  0.008,  0.062]])
```

```
poly_svm.support_
array([ 1, 26, 41, 42, 62], dtype=int32)
```

y = sign(-0.057 * (np.inner(X[1], x) + 1) ** 2 – 0.012 * (np.inner(X[41], x) + 1) ** 2
+0.008 * (np.inner(X[42], x) + 1) ** 2 + 0.062 * (np.inner(X[63], x) + 1) ** 2

Runtime Considerations



Kernels in Practice

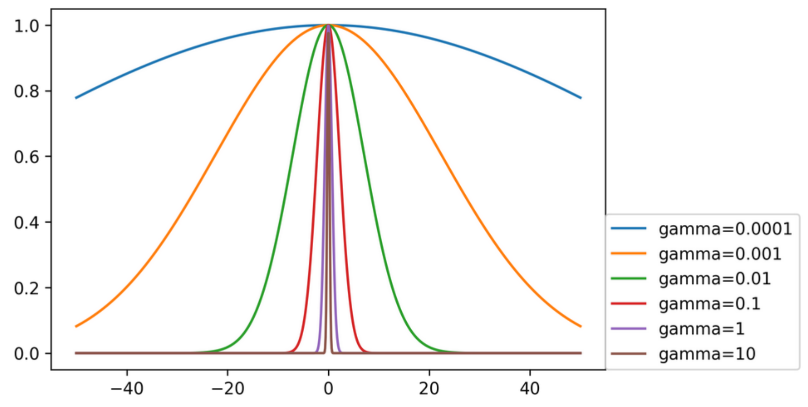
- Dual coefficients less interpretable
- Long runtime for “large” datasets (100k samples)
- Real power in infinite-dimensional spaces: rbf!
- Rbf is “universal kernel” - can learn (aka overfit) anything.

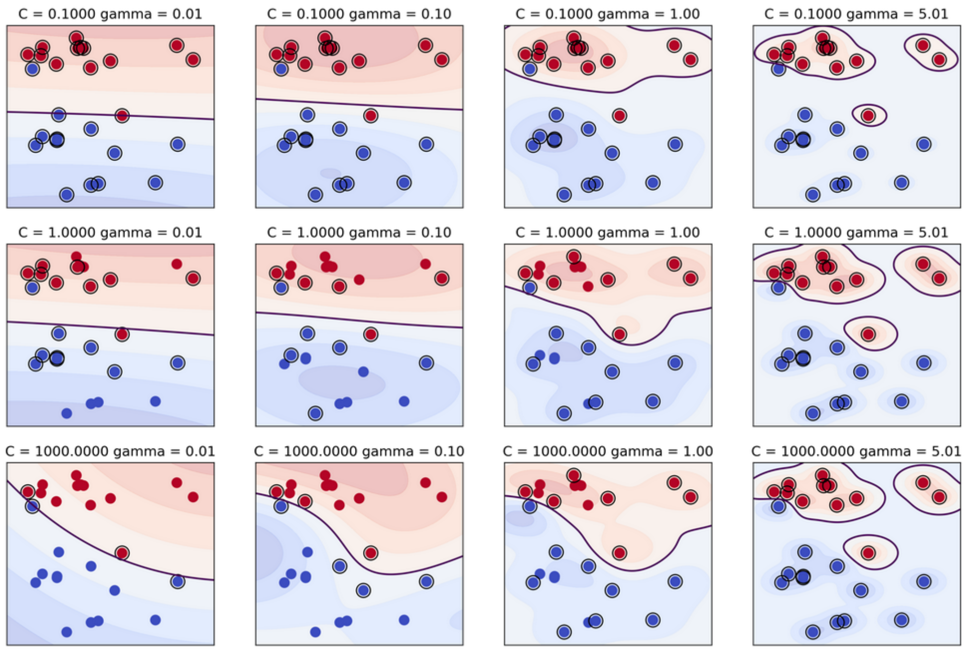
Preprocessing

- Kernel use inner products or distances.
- StandardScaler or MinMaxScaler ftw
- Gamma parameter in RBF directly relates to scaling of data – default only works with zero-mean, unit variance.

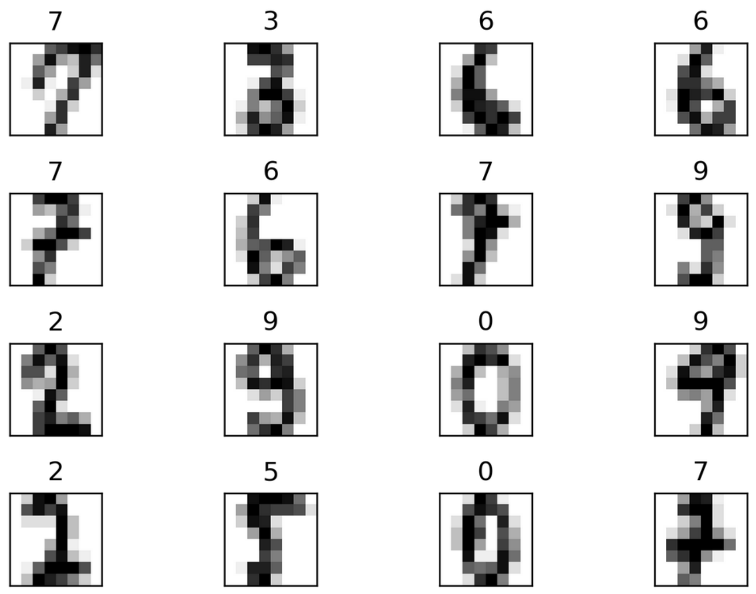
Parameters for RBF Kernels

- Regularization parameter C is limit on alphas (for any kernel)
- Gamma is bandwidth: $k_{\mathrm{rbf}}(\mathbf{x}, \mathbf{x}') = \exp(\gamma ||\mathbf{x} - \mathbf{x}'||^2)$

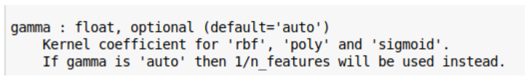



C = 0.1000 gamma = 0.01
C = 0.1000 gamma = 0.10
C = 0.1000 gamma = 1.00
C = 0.1000 gamma = 5.01
C = 1.0000 gamma = 0.01
C = 1.0000 gamma = 0.10
C = 1.0000 gamma = 1.00
C = 1.0000 gamma = 5.01
C = 1000.0000 gamma = 0.01
C = 1000.0000 gamma = 0.10
C = 1000.0000 gamma = 1.00
C = 1000.0000 gamma = 5.01

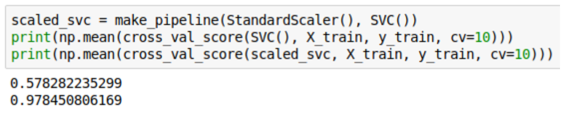
```
from sklearn.datasets import load_digits

digits = load_digits()
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target, stratify=digits.target, random_state=0)
```



Scaling and Default Params

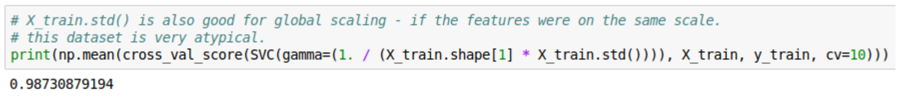


```
gamma : float, optional (default='auto')
    Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.
    If gamma is 'auto' then 1/n_features will be used instead.
```



```
scaled_svc = make_pipeline(StandardScaler(), SVC())
print(np.mean(cross_val_score(SVC(), X_train, y_train, cv=10)))
print(np.mean(cross_val_score(scaled_svc, X_train, y_train, cv=10)))
```

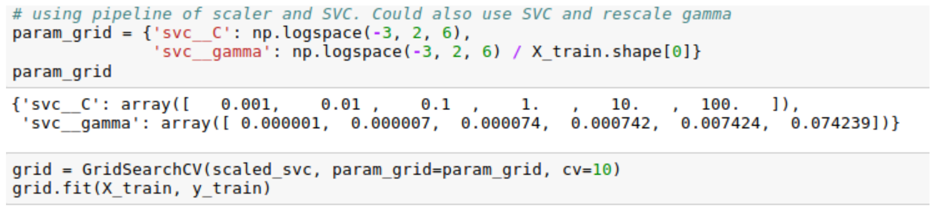
```
0.578282235299
0.978450806169
```



```
# X_train.std() is also good for global scaling - if the features were on the same scale.
# this dataset is very atypical.
print(np.mean(cross_val_score(SVC(gamma=(1. / (X_train.shape[1] * X_train.std()))), X_train, y_train, cv=10)))
```

```
0.98730879194
```

Grid-Searching Parameters

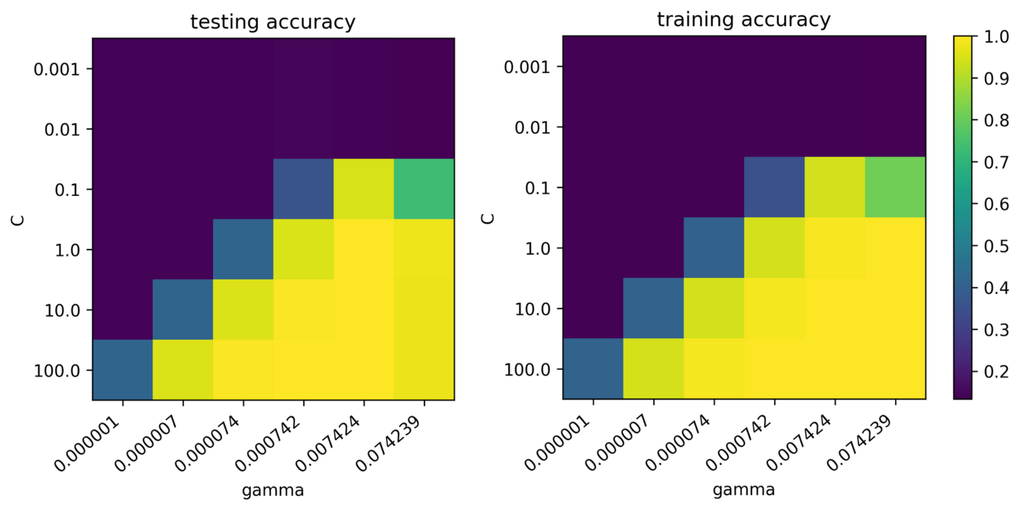


```
# using pipeline of scaler and SVC. Could also use SVC and rescale gamma
param_grid = {'svc__C': np.logspace(-3, 2, 6),
              'svc__gamma': np.logspace(-3, 2, 6) / X_train.shape[0]}
param_grid
```

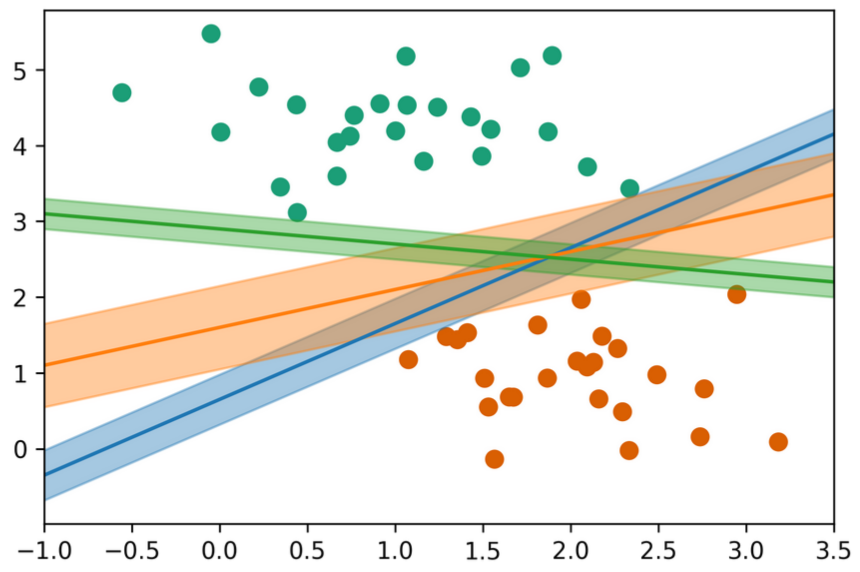
```
{'svc__C': array([   0.001,    0.01 ,    0.1  ,    1.   ,   10.   ,  100.   ]),
 'svc__gamma': array([ 0.000001,  0.000007,  0.000074,  0.000742,  0.007424,  0.074239])}
```

```
grid = GridSearchCV(scaled_svc, param_grid=param_grid, cv=10)
grid.fit(X_train, y_train)
```

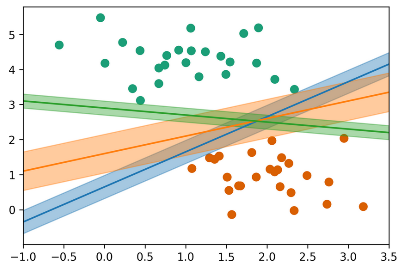
Grid-Searching Parameters

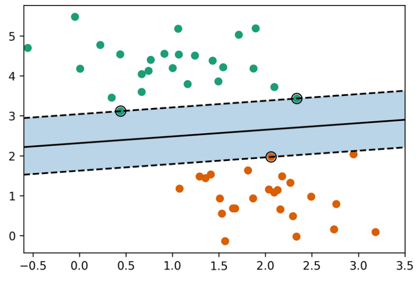


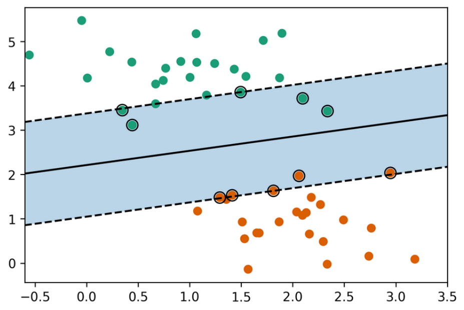
Max-Margin and Support Vectors



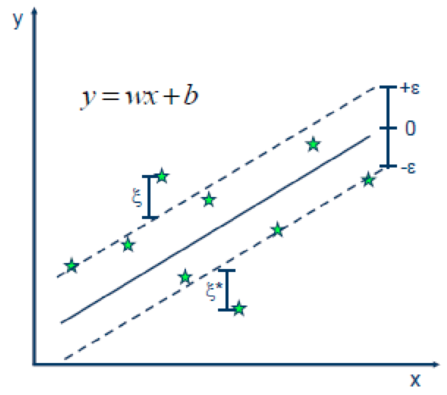
Max-Margin and Support Vectors







Support Vector Regression



- Minimize:

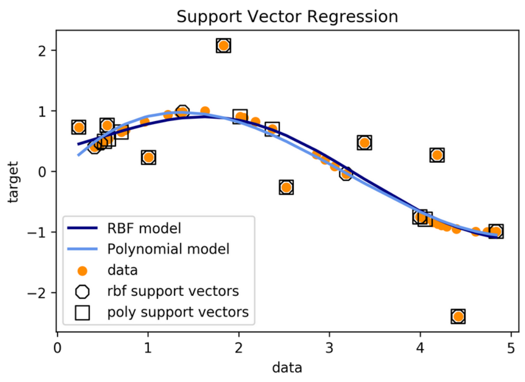
$$\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\left(\xi_i + \xi_i^*\right)$$

- Constraints:

$$y_i - wx_i - b \le \varepsilon + \xi_i$$
$$wx_i + b - y_i \le \varepsilon + \xi_i^*$$
$$\xi_i, \xi_i^* \ge 0$$

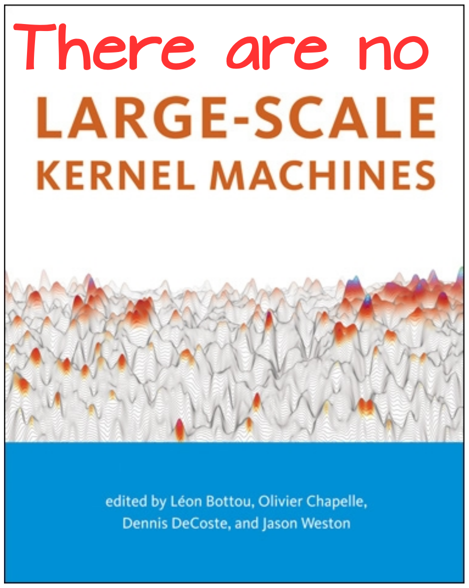
Using SVR

- Fix epsilon based on application /outliers
- Linear kernel → robust linear regression
- Ploy / rbf kernel → robust non-linear regression



Undoing the Kernel-Trick

Why undo the kernel trick?



There are no
LARGE-SCALE
KERNEL MACHINES

edited by Léon Bottou, Olivier Chapelle,
Dennis DeCoste, and Jason Weston

RKHS vs RKS

(Reproducing Kernel Hilbert-Spaces vs Random Kitchen Sinks)

- Idea: ditch kernel, approximate (infinite-dimensional) feature map

$$\phi(x)^T \phi(x') = k(x, x') \approx \hat{\phi}(x)^T \hat{\phi}(x')$$

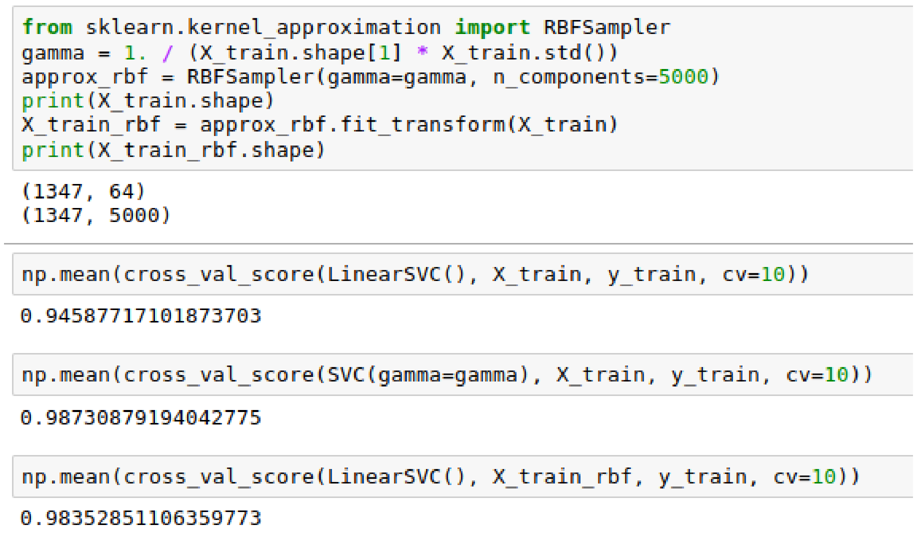
- For rbf-kernel

  random projection followed by sin/cos

  higher n_features is better

https://www.microsoft.com/en-us/research/video/random-kitchen-sinks-replacing-optimization-with-randomization-in-learning/

Kernel Approximation in sklearn



```python
from sklearn.kernel_approximation import RBFSampler
gamma = 1. / (X_train.shape[1] * X_train.std())
approx_rbf = RBFSampler(gamma=gamma, n_components=5000)
print(X_train.shape)
X_train_rbf = approx_rbf.fit_transform(X_train)
print(X_train_rbf.shape)
```

```
(1347, 64)
(1347, 5000)
```

```python
np.mean(cross_val_score(LinearSVC(), X_train, y_train, cv=10))
```

```
0.94587717101873703
```

```python
np.mean(cross_val_score(SVC(gamma=gamma), X_train, y_train, cv=10))
```

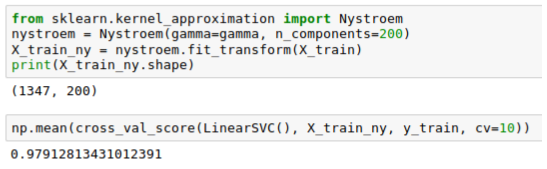
```
0.98730879194042775
```

```python
np.mean(cross_val_score(LinearSVC(), X_train_rbf, y_train, cv=10))
```

```
0.98352851106359773
```

Nyström Approximation

- Use low-rank approximation of kernel matrix
- Select some samples, compute kernel with only those, embed all the points.
- Using all points = full rank = exact
- For same number of components more expensive than RBFSampler, but needs less!



```
from sklearn.kernel_approximation import Nystroem
nystroem = Nystroem(gamma=gamma, n_components=200)
X_train_ny = nystroem.fit_transform(X_train)
print(X_train_ny.shape)
```

```
(1347, 200)
```

```
np.mean(cross_val_score(LinearSVC(), X_train_ny, y_train, cv=10))
```

```
0.97912813431012391
```

Fast Food etc

- Many newer / faster algorithms out there
- Not in sklearn so far
- FastFood one of the most prominent ones
- Current research on selecting good points for Nystroem.

Relation to Random Neural Nets

- Why approximate kernels?
- Just go random!

```
rng = np.random.RandomState(0)
w = rng.normal(size=(X_train.shape[1], 100))
X_train_wat = np.tanh(scale(np.dot(X_train, w)))
print(X_train_wat.shape)
```

```
(1347, 100)
```

```
np.mean(cross_val_score(LinearSVC(), X_train_wat, y_train, cv=10))
```

```
0.96354231101095089
```

Extreme Learning Machine Hoax

- AKA random neural networks
- Same result published in the 90s
- Bogus math

Kernel Approximation in Practice

- SVM: only when 100000 >> n_samples,

  but works for n_features large
- SVMSampler, Nystroem can allow making anything kernelized!
- Some kernels (like chi2 and intersection) have really fast approximation.

Summary

- Kernels are cool!
- Kernels work best for “small” n_samples
- Approximate kernels or random features for many samples
- Could do even SGD / streaming with kernel approximations!
- Could use Logistic Regression (hurray probabilities)