W4995 Applied Machine Learning

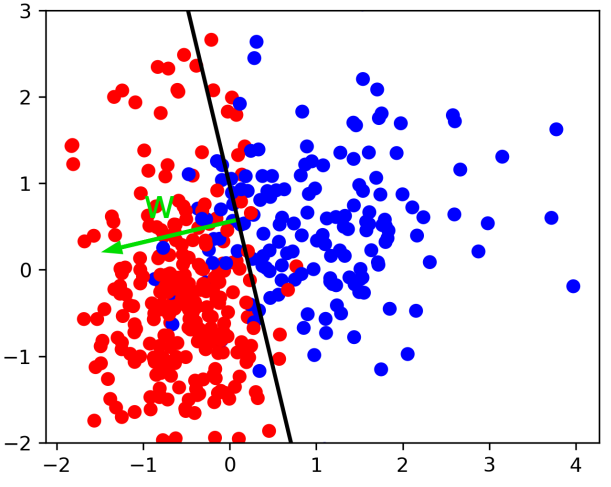
Linear Models for Classification, SVMs

02/12/19

Andreas C. Müller

(Adapted and modified for CC 6021236 @ PCC/Ciencias/UCV by Eugenio Scalise, September 2019)

Linear models for **binary** classification



$$\hat{y} = \text{sign}(w^T \mathbf{x} + b) = \text{sign}\left(\sum_i w_i x_i + b\right)$$

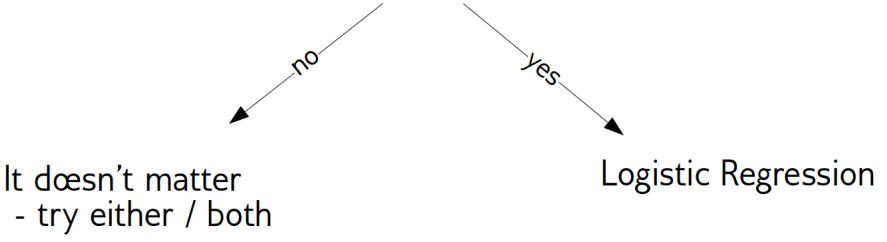
Linear models for **binary** classification

- The two most common linear classification algorithms are logistic regression and linear support vector machines (linear SVMs)
- In scikit-learn: linear_model.LogisticRegression and svm.LinearSVC
- Despite its name, LogisticRegression is a classification algorithm and not a regression algorithm, and it should not be confused with LinearRegression.

Note: Read about regularization (C)

SVM or LogReg?

Do you need probability estimates?



Multiclass classification

Reduction to Binary Classification

One vs Rest

One vs One

One Vs Rest

For 4 classes:

1v{2,3,4}, 2v{1,3,4}, 3v{1,2,4}, 4v{1,2,3}

In general:

n binary classifiers - each on all data

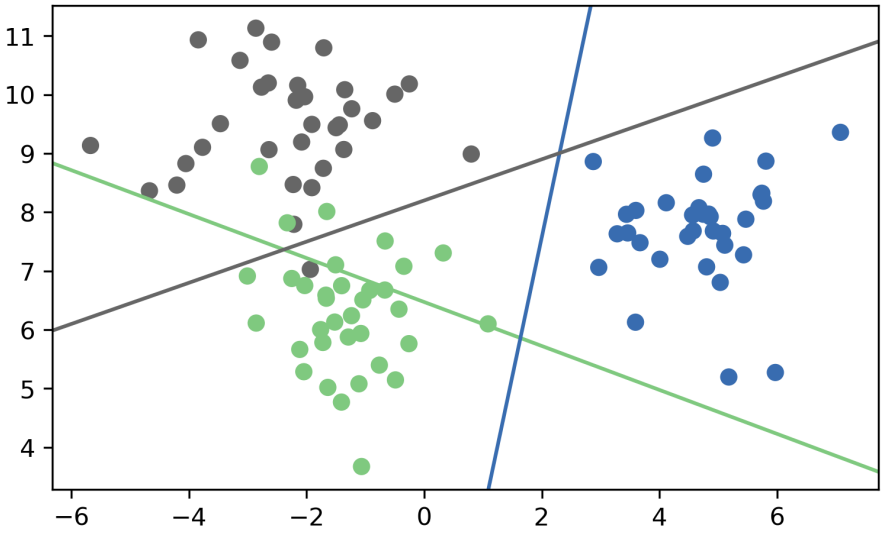
Prediction with One Vs Rest

"Class with highest score"

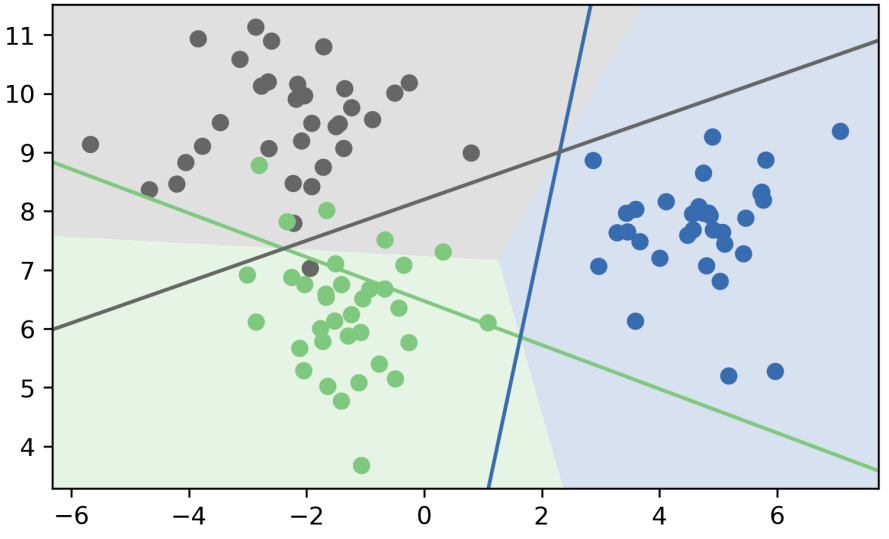
$$\hat{y} = \arg\max_{i \in Y} \mathbf{w}_i \mathbf{x} + b_i$$

To make a prediction, we compute the decision function of all classifiers on a new data point. The one with the highest score for the positive class wins, and that class is predicted.

One vs Rest Prediction



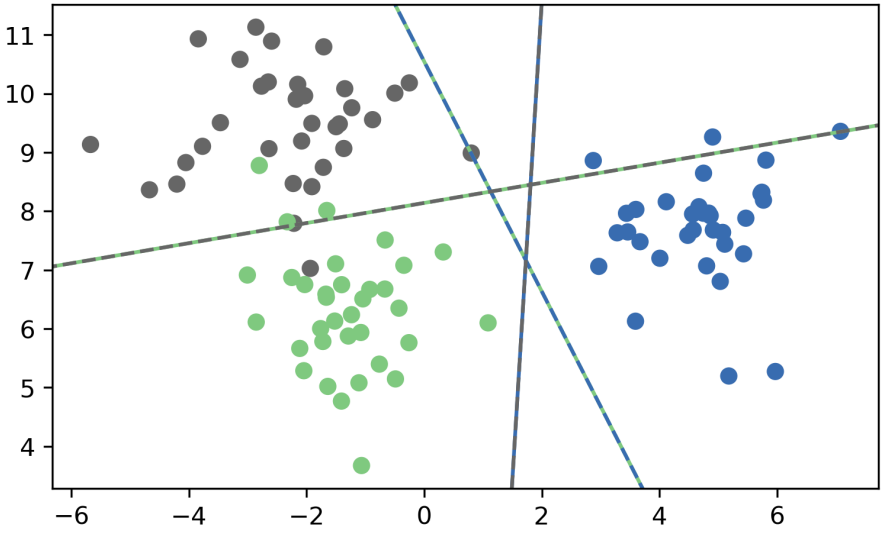
One vs Rest Prediction



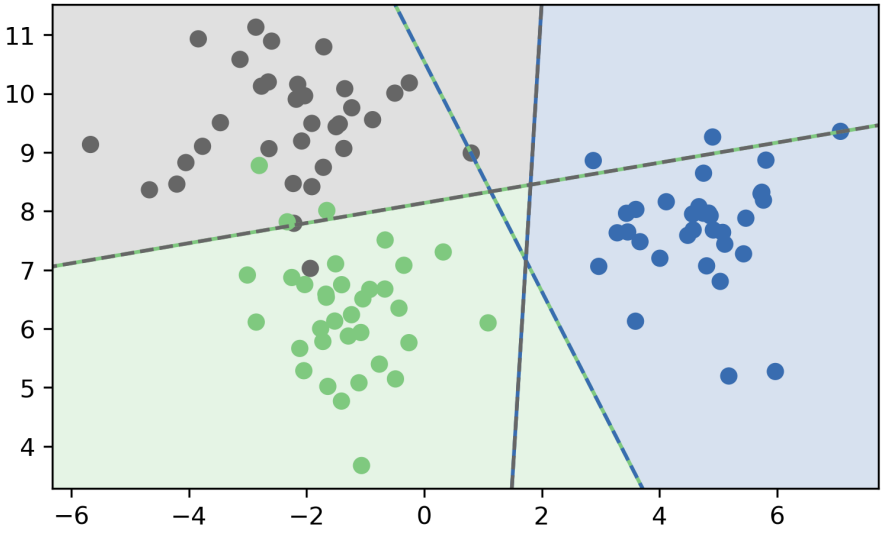
One Vs One

- 1v2, 1v3, 1v4, 2v3, 2v4, 3v4
- n * (n-1) / 2 binary classifiers
- Each classifier is trained only on the subset of the data that belongs to these classes.
- To make a prediction, we apply all of the classifiers. For each class we count how often one of the classifiers predicted that class, and we predict the class with the most votes.

One vs One Prediction

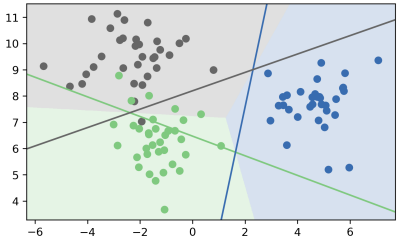


One vs One Prediction



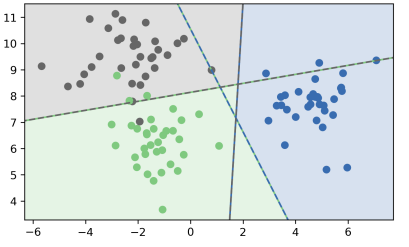
One vs Rest

- n_classes classifiers
- trained on imbalanced datasets of original size



One vs One

- n_classes * (n_classes - 1)/2 classifiers
- trained on balanced subsets



Kernel SVMs

Motivation

- Go from linear models to more powerful nonlinear ones.
- Keep convexity (ease of optimization).
- The optimization problem we have to solve from a kernel SVM is about as hard as a linear SVM.

Reminder on Linear SVM

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

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

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)

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

$$\alpha_i <= C$$

Introducing Kernels

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

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

k positive definite, symmetric ⇒ there exists a ϕ! (possilby ∞-dim)

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

Polynomial Kernel vs Features

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

Primal vs Dual Optimization

Explicit polynomials → compute on `n_samples * n_features ** d`
Kernel trick → compute on kernel matrix of shape `n_samples * n_samples`

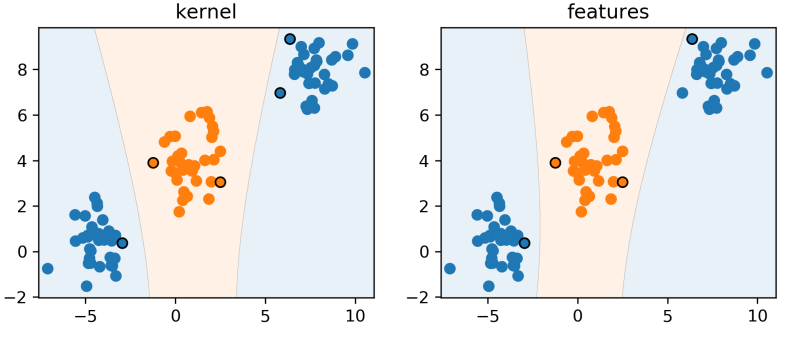
For a single feature:

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

Poly kernels vs explicit features

```
poly = PolynomialFeatures(include_bias=False)
X_poly = poly.fit_transform(X)
print(X.shape, X_poly.shape)
print(poly.get_feature_names())
```

```
((100, 2), (100, 5))
['x0', 'x1', 'x0^2', 'x0 x1', 'x1^2']
```



Understanding Dual Coefficients

```
linear_svm.coef_
```

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

$$y = \text{sign}(0.139x_0 + 0.06x_1 - 0.201x_0^2 + 0.048x_0x_1 + 0.019x_1^2)$$

```
linear_svm.dual_coef_
#array([[-0.03, -0.003, 0.003, 0.03]])
linear_svm.support_
#array([1,26,42,62], dtype=int32)
```

$$y = \text{sign}(-0.03\phi(\mathbf{x}_1)^T\phi(x) - 0.003\phi(\mathbf{x}_{26})^T\phi(\mathbf{x}) + 0.003\phi(\mathbf{x}_{42})^T\phi(\mathbf{x}) + 0.03\phi(\mathbf{x}_{62})^T\phi(\mathbf{x}))$$

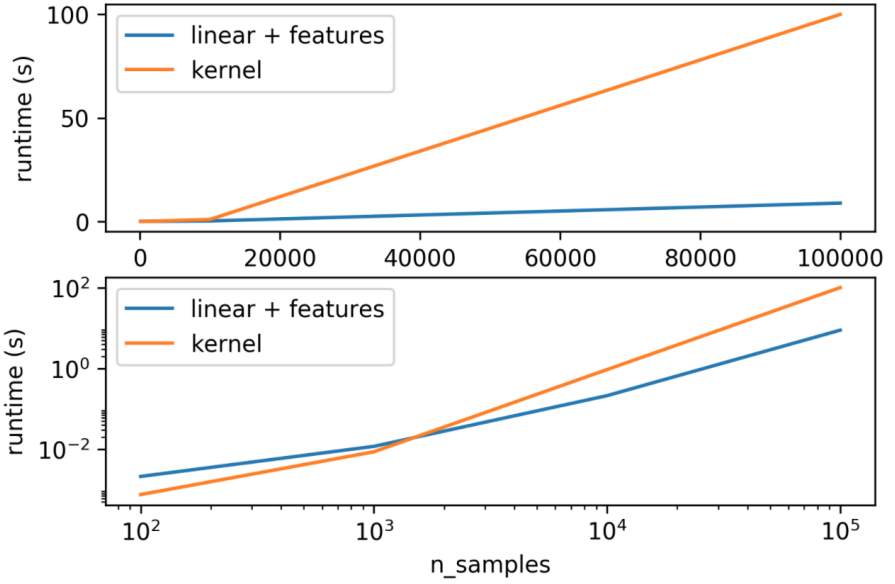
With Kernel

$$y = \text{sign}\left(\sum_{i}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x})\right)$$

```
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)
```

$$y = \text{sign}(-0.057(\mathbf{x}_1^T\mathbf{x} + 1)^2 - 0.012(\mathbf{x}_{41}^T\mathbf{x} + 1)^2$$
$$+ 0.008(\mathbf{x}_{42}^T\mathbf{x} + 1)^2 + 0.062 * (\mathbf{x}_{62}, \mathbf{x} + 1)^2)$$

Runtime Considerations



Kernels in Practice

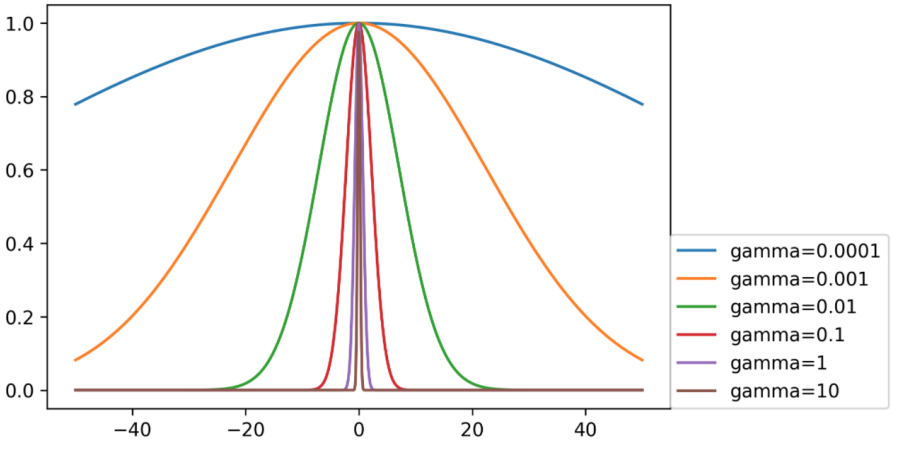
- Dual coefficients less interpretable
- Long runtime for “large” datasets (100k samples)
- Real power in infinite-dimensional spaces: rbf!
- Rbf is “universal kernel” - can learn (aka overfit) anything.

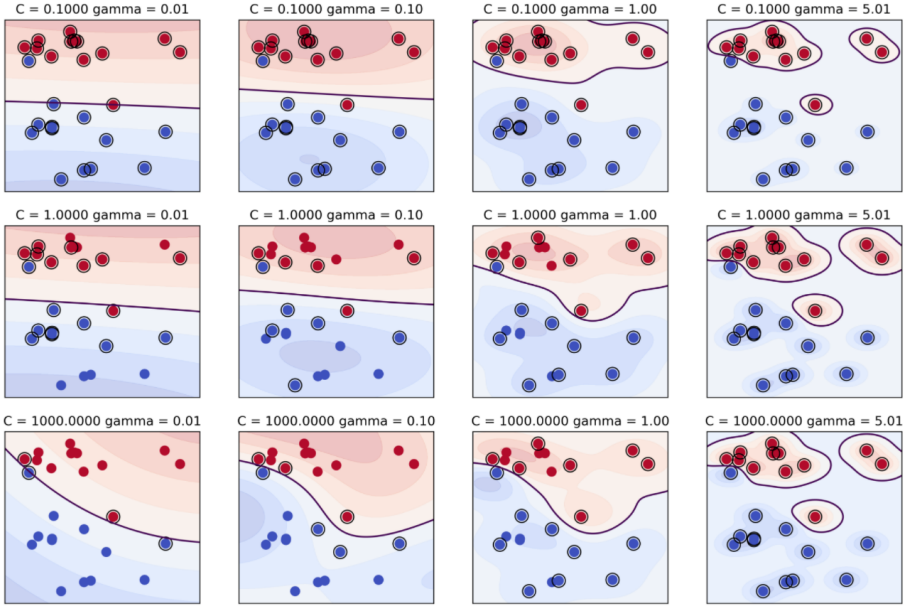
Preprocessing

- Kernel use inner products or distances.
- StandardScaler or MinMaxScaler ftw
- Gamma parameter in RBF directly relates to scaling of data and n_features – new default is `1/(X.std() * n_features)` but should be `1/X.var() * n_features)` 😞

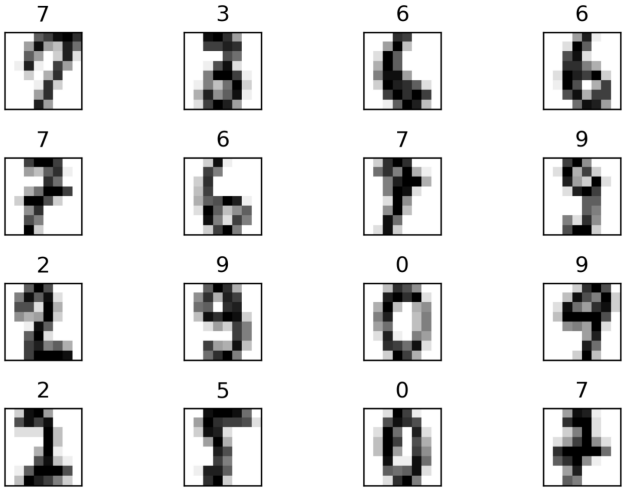
Parameters for RBF Kernels

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




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()
```



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
```

```
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)))
```

```
0.578
0.978
```

```
gamma = (1. / (X_train.shape[1] * X_train.var()))
print(np.mean(cross_val_score(SVC(gamma=gamma), X_train, y_train, cv=10)))
```

```
0.987
```

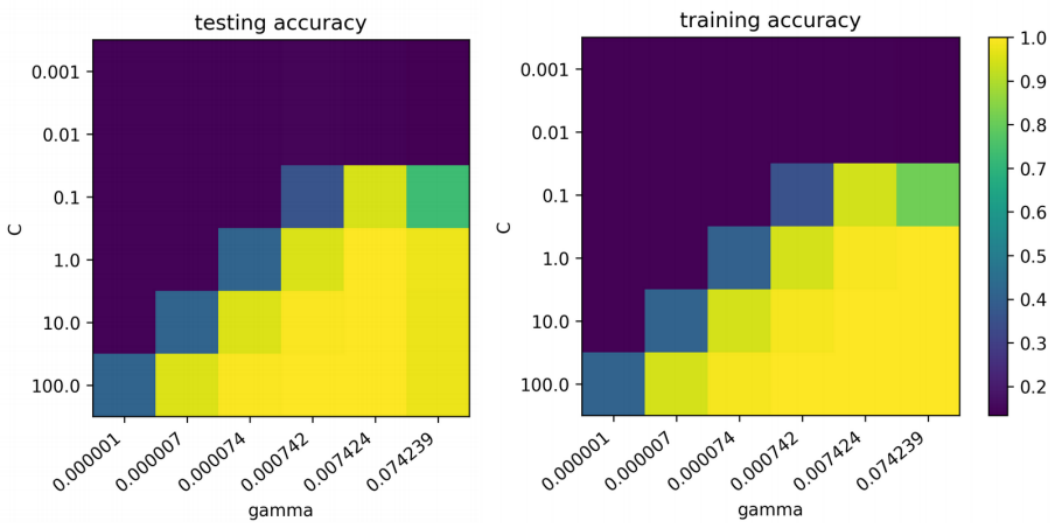
Grid-Searching Parameters

```
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



Questions ?