Linear Classification Models & SVMs

Presented by Yasin Ceran

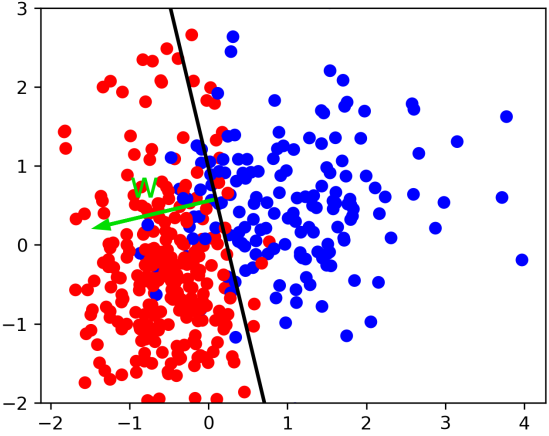
Table of Contents

1 Linear Models for Binary Classification

2 Multi Class classification

3 Kernel SVMs

Linear Binary Classification

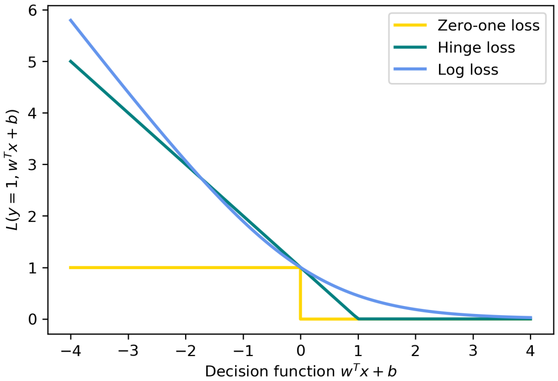


$$\hat{y} = \text{sign}(w^T \mathbf{x} + b) = \text{sign}\left(\sum_i w_i x_i + b\right)$$

Picking A Loss

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

$$\min_{w \in \mathbb{R}^p, b \in \mathbb{R}} \sum_{i=1}^{n} 1_{y_i \neq \text{sign}(w^T \mathbf{x} + b)}$$



Logistic Regression

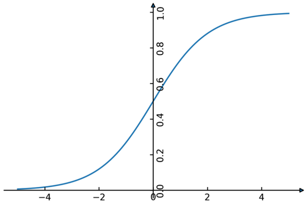
$$\log\left(\frac{p(y=1|x)}{p(y=-1|x)}\right) = w^T\mathbf{x} + b$$

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

$$\hat{y} = \begin{cases} 1, & \text{if } \hat{p} \geq 0.5 \\ -1, & \text{if } \hat{p} < 0.5 \end{cases}$$

$$p(y|\mathbf{x}) = \sigma(w^T x + b) = \frac{1}{1 + e^{-w^T\mathbf{x} - b}}$$

$$C(\mathbf{w}) = \begin{cases} -log(\hat{p}), & \text{if } y = 1 \\ -log(1-\hat{p}), & \text{if } y = -1 \end{cases}$$

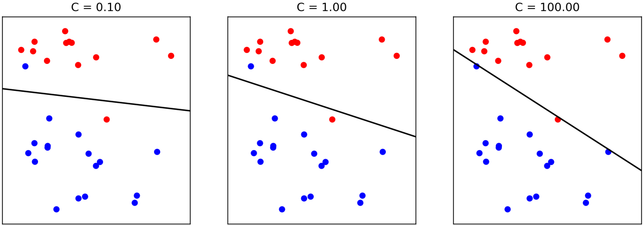


$$J(\mathbf{w}) = -\frac{1}{2n}\sum_{i=1}^{n}\left[(1 + y^{(i)})\log(\hat{p}^{(i)}) + (1 - y^{(i)})\log(1 - \hat{p}^{(i)})\right]$$

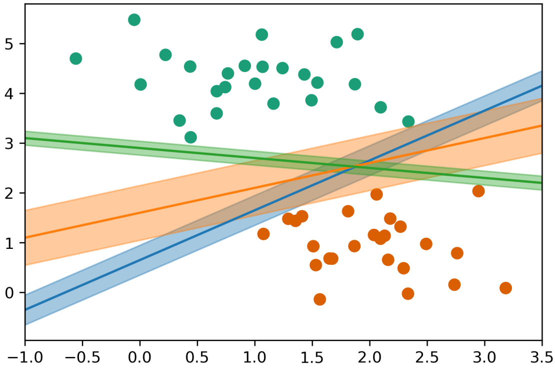
$$J(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}\log(\exp(-y_i(w^T\mathbf{x}_i + b)) + 1)$$

Penalized Logistic Regression

- $\min_{w \in ^p, b \in \mathbb{R}} C \sum_{i=1}^{n} \log(\exp(-y_i(w^T \mathbf{x}_i + b)) + 1) + ||w||_2^2$
- $\min_{w \in ^p, b \in \mathbb{R}} C \sum_{i=1}^{n} \log(\exp(-y_i(w^T \mathbf{x}_i + b)) + 1) + ||w||_1$
- C is inverse to alpha (or $alpha/n_samples$)
- Small C (a lot of regularization) limits the influence of individual points!

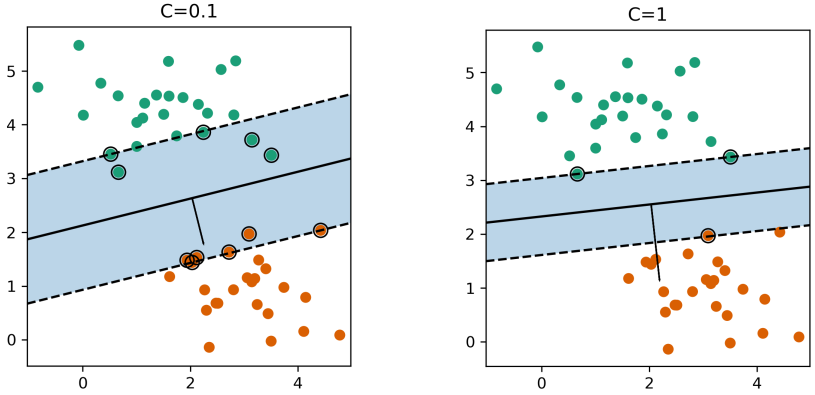


Max-Margin and Support Vectors (1)

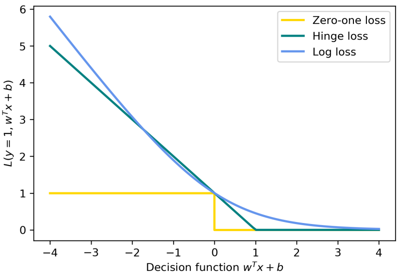


- $\min_{w \in \mathbb{R}^p, b \in \mathbb{R}} C \sum_{i=1}^{n} \max(0, 1 - y_i(w^T \mathbf{x} + b)) + ||w||_2^2$
- Within margin $\Leftrightarrow y_i(w^T x + b) < 1$
- Smaller $w \Rightarrow$ larger margin

Max-Margin and Support Vectors (2)



Logistic Regression vs SVM



- $\min_{w \in ^p, b \in \mathbb{R}} C \sum_{i=1}^{n} \log(\exp(-y_i(w^T \mathbf{x}_i + b)) + 1) + ||w||_2^2$
- $\min_{w \in ^p, b \in \mathbb{R}} C \sum_{i=1}^{n} \max(0, 1 - y_i(w^T \mathbf{x}_i + b)) + ||w||_2^2$
- Do you need probability estimates?
    - If yes, use Logistic Regression
    - If it doesn't matter, try either/both
- Need compact model or believe solution is sparse, use L_1.

Table of Contents

1 Linear Models for Binary Classification

2 Multi Class classification

3 Kernel SVMs

MultiClass Classification

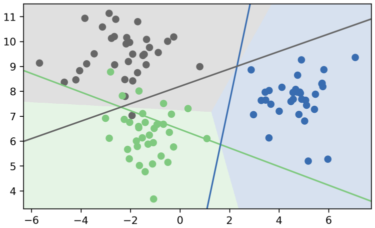
Reduction to Binary Clasification

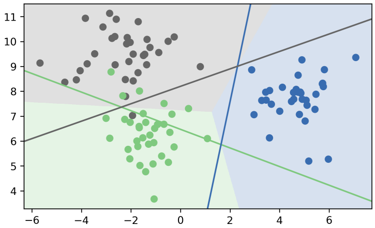
One Vs Rest

- For 4 classes:
  1v2,3,4, 2v1,3,4,
  3v1,2,4, 4v1,2,3
- In general:
  n binary classifiers
  each on all data

One vs One

- 1v2, 1v3, 1v4, 2v3, 2v4, 3v4
  n * (n-1) / 2 binary classifiers
  each on a fraction of the data
- “Vote for highest positives”
- Return most commonly predicted class.





In Scikit Learn

- OvO: only SVC
- OvR: default for all linear models except for logistic regression
- `LogisticRegression(multi_class='auto')`
- `clf.decision_function` $= w^T x + b$
- `logreg.predict_proba`
- `SVC(probability=True)` not great

MultiClass in Practice

OvR and multinomial LogReg produce one coef per class:

```
from sklearn.datasets import load_iris
iris = load_iris()
X, y = iris.data, iris.target
print(X.shape)
print(np.bincount(y))
```

```
(150, 4)
[50 50 50]
```

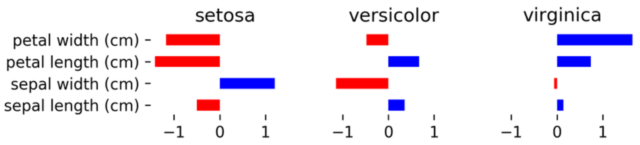
```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC

logreg = LogisticRegression(multi_class="multinomial", solver="lbfgs").fit(X, y)
linearsvm = LinearSVC().fit(X, y)
print(logreg.coef_.shape)
print(linearsvm.coef_.shape)
```

```
(3, 4)
(3, 4)
```

MultiClass in Practice

```
logreg.coef_
```

```
array([[-0.42339232,  0.96169329, -2.51946669, -1.0860205 ],
       [ 0.53411332, -0.31794321, -0.20537377, -0.93961515],
       [-0.11072101, -0.64375008,  2.72484045,  2.02563566]])
```



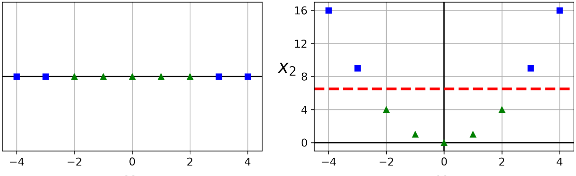
(after centering data, without intercept)

Table of Contents

1. Linear Models for Binary Classification
2. Multi Class classification
3. Kernel SVMs

Motivation

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

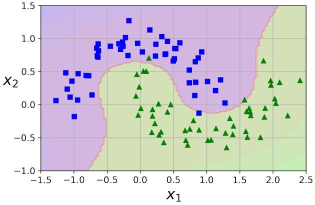


Reminder on Linear SVM

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

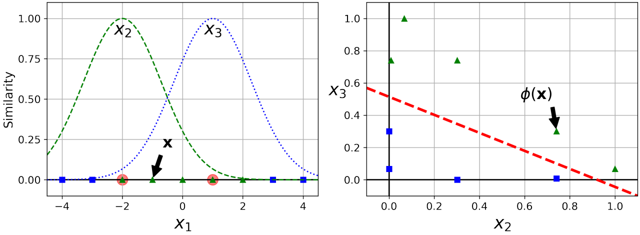
$$\hat{y} = \text{sign}(w^T \mathbf{x} + b)$$

```python
#####POLYNOMIAL FEATURES######
from sklearn.datasets import
    make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import
    PolynomialFeatures,
    StandardScaler
from sklearn.svm import LinearSVC

polynomial_svm_clf = Pipeline([
        ("poly_features",
    PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10,
    loss="hinge", random_state=42))
    ])

polynomial_svm_clf.fit(X, y)
```



Similarity Features

- You can use a *similarity function* to measure how much each instance resembles a particular *landmark*
- *Gaussian Radial Basis Function (RBF)*: $\phi_\gamma(\mathbf{x}, l) = exp\left(\gamma||\mathbf{x} - l||^2\right)$



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

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

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

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 $k + k', kk', ck', \ldots$

Polynomial Kernel vs Features

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

Complexity:

Explicit polynomials $\rightarrow n\_samples * n\_features^d$

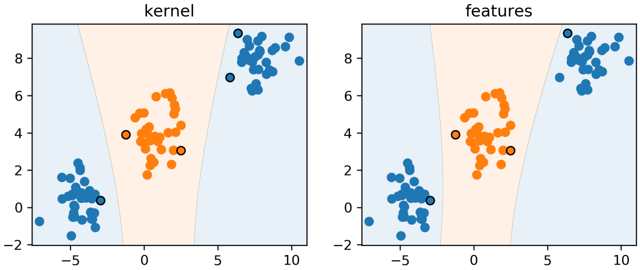
Kernel trick $\rightarrow n\_samples * n\_samples * n\_features$

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

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

```
linear_svm.coef_

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

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

```
linear_svm.dual_coef_

array([[-0.03, -0.003, 0.003, 0.03]])

linear_svm.support_

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

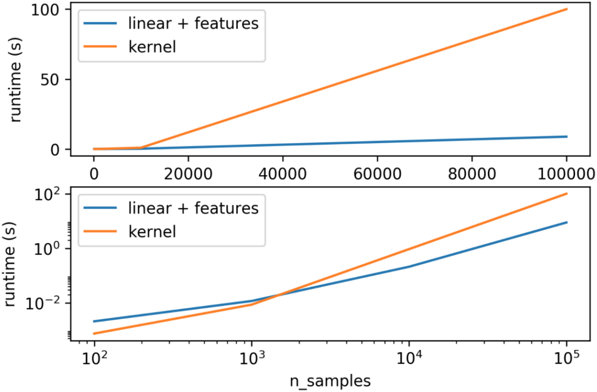
With Kernel

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

```
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)
```

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

Runtime Considerations



Kernels in Practice

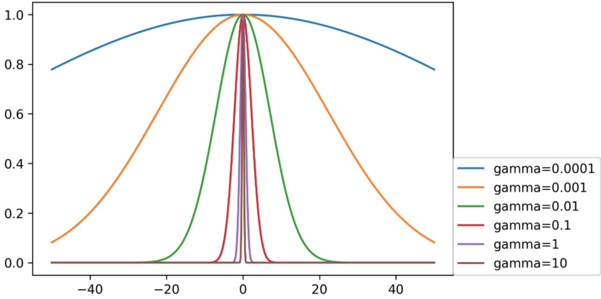
- Dual coefficients less interpretable
- Long runtime for "large" datasets (100k samples)
- As a rule of thumb, try linear kernel first (remember 'LinearSVC' is much faster than 'SVC(kernel=linear)'
- If the training data is not so large, try Gaussian RBF kernel

Preprocessing

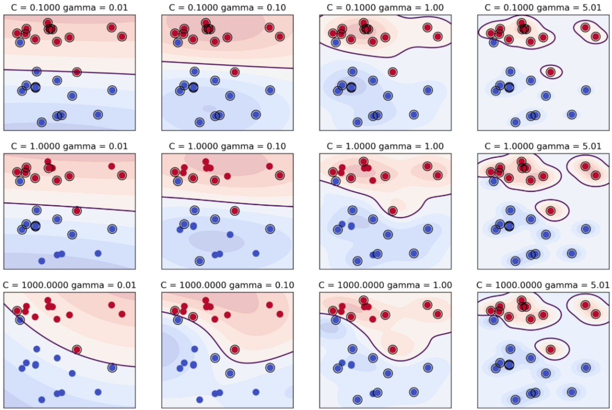
- Kernel use inner products or distances.
- StandardScaler or MinMaxScaler
- Gamma parameter in RBF directly relates to scaling of data and n_features – the default is `1/(X.var() * n_features)`

Parameters for RBF Kernels

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

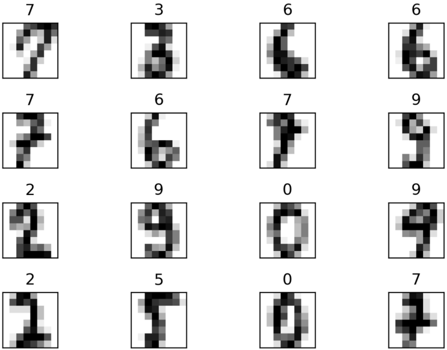


Parameters for RBF Kernels



MNIST Example

```python

from sklearn.datasets import load_digits

digits = load_digits()
```

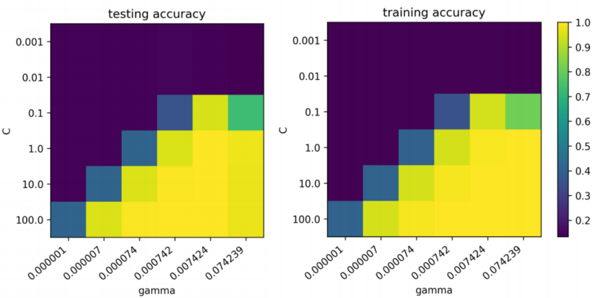


Scaling and Default Params

```
#  gamma : {'scale', 'auto'} or float, optional (default='scale')
#  Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.
#  if gamma='scale' (default) is passed then it uses 1 / (n_features * X.var())
#  as value of gamma
#  if 'auto', uses 1 / n_features.

print('auto', np.mean(cross_val_score(SVC(gamma='auto'), X_train, y_train, cv=10)))
print('scale', np.mean(cross_val_score(SVC(gamma='scale'), X_train, y_train, cv=10)))
scaled_svc = make_pipeline(StandardScaler(), SVC())
print('pipe', np.mean(cross_val_score(scaled_svc, X_train, y_train, cv=10)))


auto 0.563
scale 0.987
pipe 0.977

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)
```



Summary

- Logistic Regression and Linear SVM differ from each other by their loss functions