Unit 7
Maximum-Margin Classification and the Support Vector Machine

Prof. Phil Schniter

THE OHIO STATE UNIVERSITY

ECE 5307: Introduction to Machine Learning, Sp23

Learning objectives

- Gain experience with linear classification of images
  - representing images, displaying images, classifying images
- Understand the geometry of the linear classification boundary:
  - orthogonality to $\boldsymbol{w}$, offset b, margin $1/\|\boldsymbol{w}\|$
- Understand the margin-maximizing classifier
- Understand the support vector classifier (SVC)
- Understand the support vector machine (SVM)
- Understand how to implement the SVC and SVM with sklearn

Outline

- Motivating Example: Recognizing Handwritten Digits
- Maximum-Margin Classification
- The Support Vector Classifier
- The Support Vector Machine

MNIST digit classification

- Goal: Handwriting recognition
- Dataset: MNIST
  - Only digits 0–9
  - 28x28 images (grayscale)
  - 70,000 examples
  - Collected by American Census Bureau in 1980s
- Why use it?
  - Simple
  - Widely accessible
  - Many benchmarks

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1, cache=True)
mnist.data.shape
```

```
(70000, 784)
```

MNIST benchmarks

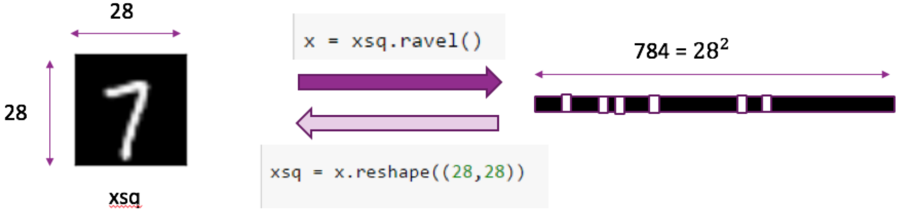
- In this unit, we focus on the support vector machine (SVM).
- Not state-of-the-art, but quite good, and widely used in machine learning

Type	Classifier	Distortion	Preprocessing	Error rate (%)
Convolutional neural network	Committee of 20 CNNS with Squeeze-and-Excitation Networks[35]	None	Data augmentation	0.17[36]
Random Multimodel Deep Learning (RMDL)	10 NN-10 RNN - 10 CNN	None	None	0.18[25]
Convolutional neural network	Committee of 5 CNNs, 6-layer 784-50-100-500-1000-10-10	None	Expansion of the training data	0.21[22][23]
Convolutional neural network	Committee of 35 CNNs, 1-20-P-40-P-150-10	Elastic distortions	Width normalizations	0.23[15]
Convolutional neural network (CNN)	13-layer 64-128(5x)-256(3x)-512-2048-256-256-10	None	None	0.25[20]
Convolutional neural network	6-layer 784-50-100-500-1000-10-10	None	Expansion of the training data	0.27[34]
Convolutional neural network (CNN)	6-layer 784-40-80-500-1000-2000-10	None	Expansion of the training data	0.31[33]
Deep neural network	6-layer 784-2500-2000-1500-1000-500-10	Elastic distortions	None	0.35[32]
K-Nearest Neighbors	K-NN with non-linear deformation (P2DHMDM)	None	Shiftable edges	0.52[26]
Support-vector machine (SVM)	Virtual SVM, deg-9 poly, 2-pixel jittered	None	Deskewing	0.56[30]
Deep neural network	2-layer 784-800-10	Elastic distortions	None	0.7[31]
Boosted Stumps	Product of stumps on Haar features	None	Haar features	0.87[27]
Deep neural network (DNN)	2-layer 784-800-10	None	None	1.6[31]
Random Forest	Fast Unified Random Forests for Survival, Regression, and Classification (RF-SRC)[28]	None	Simple statistical pixel importance	2.8[29]
Non-linear classifier	40 PCA + quadratic classifier	None	None	3.3[10]
Linear classifier	Pairwise linear classifier	None	Deskewing	7.6[10]

https://en.wikipedia.org/wiki/MNIST_database

Representing images in Numpy

- Each pixel is a value between 0 (black) and 255 (white)
- Images can be represented as matrices or vectors



- In Numpy, vectorization is done by stacking rows, but in most papers/books it is done by stacking columns:

$$\boldsymbol{X} = \begin{bmatrix} x_1 & x_{29} & \cdots & x_{757} \\ x_2 & x_{30} & \cdots & x_{758} \\ \vdots & \vdots & & \vdots \\ x_{28} & x_{56} & \cdots & x_{784} \end{bmatrix} = \mathrm{mat}(\boldsymbol{x}), \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{784} \end{bmatrix} = \mathrm{vec}(\boldsymbol{X})$$

  - Basically, Numpy works with $\boldsymbol{X}^\mathsf{T}$ and $\boldsymbol{x}^\mathsf{T}$ for the $\boldsymbol{X}$ and $\boldsymbol{x}$ defined above

Displaying images with matplotlib

- plt.imshow is the main plot command
- We randomly permute the sample indices using `Iperm`
  - Else training set might include different digits than testing set
  - Do this for any dataset!
- A random sample of 4 MNIST digits:

  - A human could easily classify these

```
def plt_digit(x):
    nrow = 28
    ncol = 28
    xsq = x.reshape((nrow,ncol))
    plt.imshow(xsq,  cmap='Greys_r')
    plt.xticks([])
    plt.yticks([])

# Convert data to a matrix
X = mnist.data
y = mnist.target.astype(np.int8) # fetch_ope

# Select random digits
nplt = 4
nsamp = X.shape[0]
Iperm = np.random.permutation(nsamp)

# Plot the images using the subplot command
for i in range(nplt):
    ind = Iperm[i]
    plt.subplot(1,nplt,i+1)
    plt_digit(X[i,:])
```

Try multinomial logistic regression (MLR)

- Train and test on 10000 samples
  - Using more samples leads to better results but takes more time
- Select a fast solver (lbfgs)
  - Even this may take several minutes

```
ntr = 10000
nts = 10000
Xs = X/255.0*2 - 1 # convert to range [-1,1]
# careful: 2*X/255.0 will fail if X is uint8
Xtr = Xs[Iperm[:ntr],:]
ytr = y[Iperm[:ntr]]
Xts = Xs[Iperm[ntr:ntr+nts],:]
yts = y[Iperm[ntr:ntr+nts]]
```

```
logreg = linear_model.LogisticRegression(verbose=1, multi_class='multinomial',
                                         solver='lbfgs', max_iter=500, C=1e-2)
logreg.fit(Xtr,ytr)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    4.5s finished
```

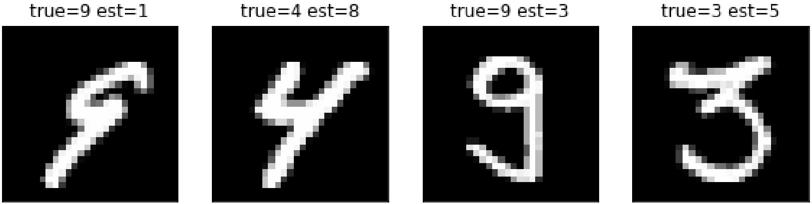
```
LogisticRegression(C=0.01, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=500,
                   multi_class='multinomial', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=1,
                   warm_start=False)
```

Performance of MLR

- Accuracy = 90% (not very good)
- Most of the errors MLR made would be obvious to a human, for example:

```
yhat = logreg.predict(Xts)
acc = np.mean(yhat == yts)
print('Accuracy = {0:f}'.format(acc))
```

```
Accuracy = 0.902300
```

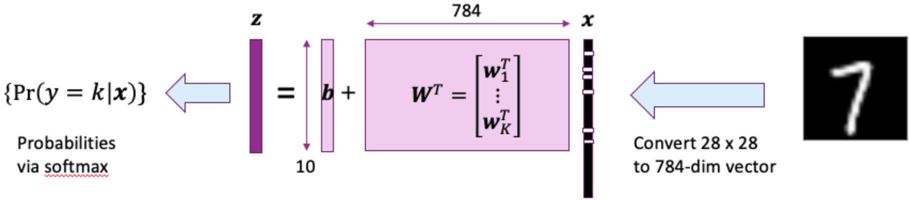


- What went wrong?
- Can we do better?

Review of MLR

- To classify a test vector $\boldsymbol{x}$ (e.g., an MNIST image), we do the following:
- For each class hypothesis $k = 1, \dots, K$,
  - design weights b_k and $\boldsymbol{w}_k = [w_{k1}, \dots, w_{kd}]^\mathsf{T}$ that classify $\boldsymbol{x}$ as in-k or not-in-k
  - compute a linear score $z_k = b_k + \boldsymbol{x}^\mathsf{T}\boldsymbol{w}_k$

then classify as $\widehat{y} = \arg\max_k \underbrace{\Pr\{y = k \,|\, \boldsymbol{x}\}}_{\substack{\text{probability of} \\ \boldsymbol{x} \text{ in class } k}} = \arg\max_k \underbrace{\frac{e^{z_k}}{\sum_{l=1}^K e^{z_l}}}_{\text{softmax}} = \arg\max_k z_k$

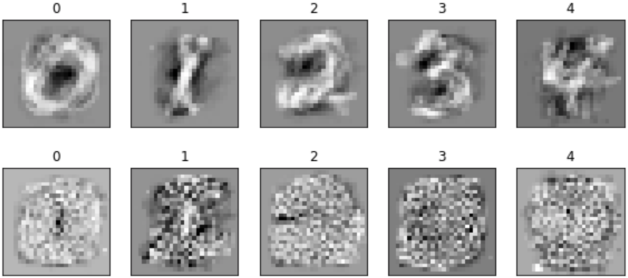


What weights does MLR learn?

- Intuitively, we want $z_k = b_k + \boldsymbol{x}^\mathsf{T}\boldsymbol{w}_k$ to be large when $\boldsymbol{x}$ is from class k, and small otherwise. This should happen when $\boldsymbol{w}_k$ looks like the kth digit
- Let's visualize the learned $\boldsymbol{w}_k$ for $k = 0, \ldots, 4$ by plotting them as images:

Learned from 100 training samples:

Learned from 10000 training samples:



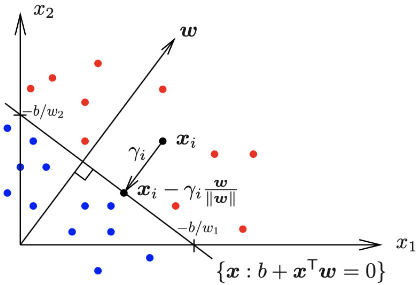
- MLR struggles to match *all* the training examples from each class due to variations across that class, especially when number of training samples is large!

Outline

- Motivating Example: Recognizing Handwritten Digits
- Maximum-Margin Classification
- The Support Vector Classifier
- The Support Vector Machine

Hyperplane geometry

- In *binary* linear classification, the decision boundary is $z = 0$ or
$$\{\boldsymbol{x} : b + \boldsymbol{x}^\mathsf{T}\boldsymbol{w} = 0\}$$
which is a hyperplane



- This hyperplane is orthogonal to $\boldsymbol{w}$:
  - For any $\{\boldsymbol{x}_1, \boldsymbol{x}_2\}$ on the hyperplane, $b + \boldsymbol{x}_i^\mathsf{T}\boldsymbol{w} = 0$ for $i = 1, 2$.
  - Thus $(\boldsymbol{x}_1 - \boldsymbol{x}_2)^\mathsf{T}\boldsymbol{w} = 0$

- From the figure, the signed distance from a point $\boldsymbol{x}_i$ to the hyperplane is γ_i:

$$0 = b + \left(\boldsymbol{x}_i - \gamma_i \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}\right)^\mathsf{T}\boldsymbol{w} = b + \boldsymbol{x}_i^\mathsf{T}\boldsymbol{w} - \gamma_i\|\boldsymbol{w}\| \quad \Rightarrow \quad \boxed{\gamma_i = \frac{b + \boldsymbol{x}_i^\mathsf{T}\boldsymbol{w}}{\|\boldsymbol{w}\|}}$$

- The classifier $(\alpha b, \alpha\boldsymbol{w})$ is equivalent to the classifier $(b, \boldsymbol{w})$ for all $\alpha > 0$

Review of linear separability

- Assume binary ± 1 training data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$
- The data is "linearly separable" if there exists $(b, \boldsymbol{w})$ and $\epsilon > 0$ such that

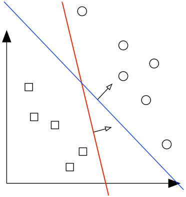
$$b + \boldsymbol{x}_i^\mathsf{T} \boldsymbol{w} \geq \epsilon \qquad \text{whenever } y_i = 1$$

$$b + \boldsymbol{x}_i^\mathsf{T} \boldsymbol{w} \leq -\epsilon \qquad \text{whenever } y_i = -1$$

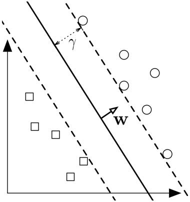
  or, more compactly, $y_i(b + \boldsymbol{x}_i^\mathsf{T} \boldsymbol{w}) \geq \epsilon \;\; \forall i$

- Because $(b, \boldsymbol{w})$ equivalent to $(\frac{b}{\epsilon}, \frac{\boldsymbol{w}}{\epsilon})$, linearly separable means $\boxed{\exists (b, \boldsymbol{w}) : \; y_i(b + \boldsymbol{x}_i^\mathsf{T} \boldsymbol{w}) \geq 1 \; \forall i}$
- If separable, the distance γ_i from the boundary to datapoint $\boldsymbol{x}_i$ obeys $\boxed{y_i \gamma_i \geq 1/\|\boldsymbol{w}\| \triangleq \gamma, \;\; \forall i.}$
- The "margin" γ is a lower bound on the minimum distance from any training $\boldsymbol{x}_i$ to the boundary

linear separability:



margin maximizing $\boldsymbol{w}$:

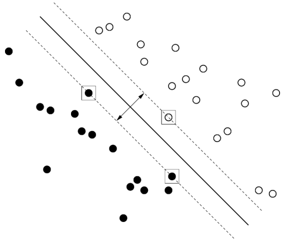


Maximum-margin classification

- To make the classifier robust, we should *maximize the margin!*
  - We can do this by minimizing $\|\boldsymbol{w}\|$ under the linear-separability constraint
- The hard-margin classifier is defined as the $(b, \boldsymbol{w})$ that solves

$$\min_{b,\boldsymbol{w}} \frac{1}{2}\|\boldsymbol{w}\|^2 \text{ s.t. } y_i(b + \boldsymbol{x}_i^\mathsf{T}\boldsymbol{w}) \geq 1 \;\; \forall i = 1, \dots, n$$

  - Of all the $\{b, \boldsymbol{w}\}$ satisfying $y_i(b + \boldsymbol{x}_i^\mathsf{T}\boldsymbol{w}) \geq 1 \; \forall i$, the one yielding the widest "street" is the one with the smallest $\|\boldsymbol{w}\|^2$



- The orientation of its boundary depends only on a few samples $\boldsymbol{x}_i$ near the boundary:
  - Called the support vectors
  - More on this later ...
- Problem: The hard-margin classifier requires the data to be linearly separable!

Outline

The support vector classifier

- To handle data that is *not* linearly separable, we replace the hard constraint

$$y_i(b + \boldsymbol{x}_i^\mathsf{T}\boldsymbol{w}) \geq 1 \ \forall i \quad \Leftrightarrow \quad 0 \geq 1 - y_i(b + \boldsymbol{x}_i^\mathsf{T}\boldsymbol{w}) \ \forall i$$

with the cost term
$$\underbrace{\sum_{i=1}^{n} \max\left\{0, 1 - y_i(b + \boldsymbol{x}_i^\mathsf{T}\boldsymbol{w})\right\}}_{=0 \text{ if linearly separable, } >0 \text{ otherwise}}$$

- This results in the soft-margin classifier or support-vector classifier (SVC):

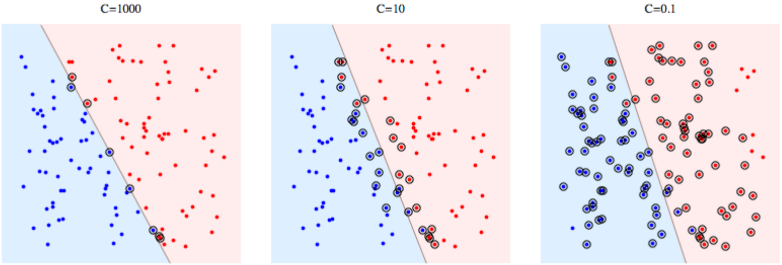
$$\min_{b,\boldsymbol{w}} \left\{ C \sum_{i=1}^{n} \max\left\{0, 1 - y_i(b + \boldsymbol{x}_i^\mathsf{T}\boldsymbol{w})\right\} + \frac{1}{2}\|\boldsymbol{w}\|^2 \right\}$$ with tunable parameter $C > 0$

- C trades off between margin size $1/\|\boldsymbol{w}\|$ and number of margin violations
  - Larger C penalizes violations more and thus yields a smaller margin $1/\|\boldsymbol{w}\|$
  - For linearly separable data and sufficiently large C, soft-margin $\Leftrightarrow$ hard-margin

Effect of SVC penalty C

C controls the margin $\frac{1}{\|\boldsymbol{w}\|}$, and thus the number of support vectors

- Larger C: smaller margin, fewer support vectors
  - More sensitive to datapoints close to hyperplane: Lower bias, higher variance
- Smaller C: larger margin, more support vectors
  - Less sensitive to data: Higher bias, lower variance



Support vectors are circled

Support vectors are the $\boldsymbol{x}_i$ with distance $y_i\gamma_i$ that is $\leq \frac{1}{\|\boldsymbol{w}\|}$ from boundary

- Note: Increasing C beyond a certain value will have no effect: You will reach the minimum # of support vectors and the boundary won't change.

The SVC and hinge loss

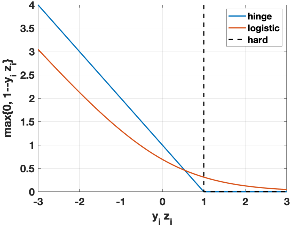
- Recalling L2-regularized logistic regression:

$$\min_{\boldsymbol{w},b}\left\{\sum_{i=1}^{n}\underbrace{\left(\ln[1+e^{z_i}]-\tfrac{y_i+1}{2}z_i\right)}_{\text{"logistic loss"}}+\frac{1}{2C}\|\boldsymbol{w}\|^2\right\} \quad \text{for} \quad z_i=b+\boldsymbol{x}_i^\mathsf{T}\boldsymbol{w}$$

we see that the SVC looks similar, but with a different loss:

$$\min_{\boldsymbol{w},b}\left\{\sum_{i=1}^{n}\underbrace{\max\left\{0,1-y_iz_i\right\}}_{\text{"hinge loss"}}+\frac{1}{2C}\|\boldsymbol{w}\|^2\right\} \quad \text{for} \quad z_i=b+\boldsymbol{x}_i^\mathsf{T}\boldsymbol{w}$$

- Comparison of loss functions:
  - hinge loss places *no penalty* on samples obeying the margin (i.e., $y_iz_i>1$)
  - thus, the SVC boundary determined only by $\boldsymbol{x}_i$ that cause $y_iz_i\leq 1$, called "support vectors"
  - logistic loss penalizes *all* samples



A closer look at the SVC

- We just saw that the SVC solution $(\boldsymbol{w}_*, b_*)$ minimizes the cost

$$J_{\text{svc}}(\boldsymbol{w}, b) \triangleq C \sum_{i=1}^{n} \max\left\{0, 1 - y_i \underbrace{\left(b + \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_i\right)}_{z_i}\right\} + \frac{1}{2}\|\boldsymbol{w}\|^2$$

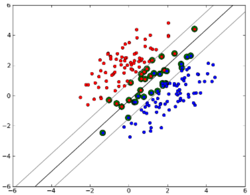
- The gradient (where defined) must equal zero at $(\boldsymbol{w}_*, b_*)$:

$$0 = \frac{\partial}{\partial b} J(\boldsymbol{w}_*, b_*) = -C \sum_{i=1}^{n} \alpha_i y_i \quad \text{for} \quad \alpha_i = \begin{cases} 1 & y_i(b_* + \boldsymbol{w}_*^\mathsf{T}\boldsymbol{x}_i) < 1 \\ ? & y_i(b_* + \boldsymbol{w}_*^\mathsf{T}\boldsymbol{x}_i) = 1 \\ 0 & y_i(b_* + \boldsymbol{w}_*^\mathsf{T}\boldsymbol{x}_i) > 1 \end{cases}$$

$$\mathbf{0} = \nabla_{\boldsymbol{w}} J(\boldsymbol{w}_*, b_*) = -C \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i + \boldsymbol{w}_*$$

which implies $\boldsymbol{w}_* = C \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i$ and $\sum_{i=1}^{n} \alpha_i y_i = 0$

- The equation for $\boldsymbol{w}_*$ shows that it depends only on the support vectors $\{\boldsymbol{x}_i : y_i(b_* + \boldsymbol{x}_i^\mathsf{T}\boldsymbol{w}_*) \leq 1\}$

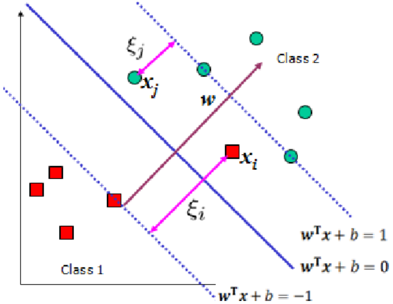


Alternate form of the SVC

- The SVC is often written using "slack variables" $\xi_i \triangleq \max\{0, 1 - y_i z_i\}$:

$$\min_{b,\boldsymbol{w}} \left\{ C \sum_{i=1}^{n} \xi_i + \frac{1}{2}\|\boldsymbol{w}\|^2 \right\} \text{ s.t. } \xi_i \geq 0 \text{ and } \xi_i \geq 1 - y_i(b + \boldsymbol{x}_i^\mathsf{T}\boldsymbol{w}) \text{ for all } i$$

- ξ_i measures violation of the margin
  - $\xi_i = 0$: sample outside "street" (on correct side of hyperplane)
  - $\xi_i \in (0, 2)$: sample in "street"
  - $\xi_i > 1$: sample misclassified (wrong side of hyperplane)
- Any $\xi_i > 0$ is considered a violation



Applying the SVC to test data

- Given a test vector $\boldsymbol{x}$, the SVC computes the score

$$z = b_* + \boldsymbol{w}_*^{\mathsf{T}}\boldsymbol{x} \quad \text{with} \quad \boldsymbol{w}_* = C\sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i$$

  where $\alpha_i = 0$ for all $\boldsymbol{x}_i$ that are not support vectors

- The resulting score

$$z = b_* + C\sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i^{\mathsf{T}}\boldsymbol{x} \quad \text{used for} \quad \widehat{y} = \operatorname{sgn}(z)$$

  is a weighted linear combination of the inner products $\boldsymbol{x}_i^{\mathsf{T}}\boldsymbol{x}$ between $\boldsymbol{x}$ and each support vector $\boldsymbol{x}_i$.
  - When classifying $\boldsymbol{x}$, the SVC uses only $\{\boldsymbol{x}_i\}$ that are support vectors, whereas logistic regression uses *all* the training samples $\{\boldsymbol{x}_i\}$
  - The SVC tries to focus on what is most important when making a decision

Multiclass SVC

- In multiclass linear classification with K classes, we compute the score *vector*

$$\boldsymbol{z}_i = \begin{bmatrix} z_{i1} \\ \vdots \\ z_{iK} \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_K \end{bmatrix} + \begin{bmatrix} \boldsymbol{w}_1^\mathsf{T} \\ \vdots \\ \boldsymbol{w}_K^\mathsf{T} \end{bmatrix} \boldsymbol{x}_i = \boldsymbol{b} + \boldsymbol{W}^\mathsf{T} \boldsymbol{x}_i \in \mathbb{R}^K$$

- One option is to *separately* train one-vs-rest binary SVCs $\{(b_k, \boldsymbol{w}_k)\}_{k=1}^K$
- Another is to *jointly* train $(\boldsymbol{b}, \boldsymbol{W})$ via the Crammer-Singer approach

$$\min_{\boldsymbol{b},\boldsymbol{W}} \left\{ \sum_{i=1}^n \sum_{j \neq y_i} \max\left\{0, 1 + z_{ij} - z_{i,y_i}\right\} + \frac{1}{2C} \|\boldsymbol{W}\|_F^2 \right\}$$

  - Note that, when $K = 2$, we get (using $j, y_i \in \{-1, 1\}$ for convenience)

$$\begin{aligned} \sum_{j \neq y_i} \max\left\{0, 1 + z_{ij} - z_{i,y_i}\right\} &= \max\left\{0, 1 + z_{i,-y_i} - z_{i,y_i}\right\} \\ &= \max\left\{0, 1 - y_i(z_{i,1} - z_{i,-1})\right\} \\ &= \max\left\{0, 1 - y_i(b + \boldsymbol{w}^\mathsf{T} \boldsymbol{x}_i)\right\} \end{aligned}$$

  for $b \triangleq b_1 - b_{-1}$ and $\boldsymbol{w} \triangleq \boldsymbol{w}_1 - \boldsymbol{w}_{-1}$, which matches our earlier binary SVC

Implementing the SVC in sklearn

- The SVC is easy to implement in sklearn using svm.LinearSVC
  - The parameter C penalizes the slack variables
  - As we discussed, larger C means fewer support vectors (up to a point)

```
from sklearn import svm
svc = svm.LinearSVC(loss='hinge', C=1e-2, multi_class='crammer_singer'
svc.fit(Xtr,ytr)
yhat_ts = svc.predict(Xts)
acc = np.mean(yhat_ts == yts)
print('Accuracy = {0:f}'.format(acc))
```

```
[LibLinear]Accuracy = 0.907300
```

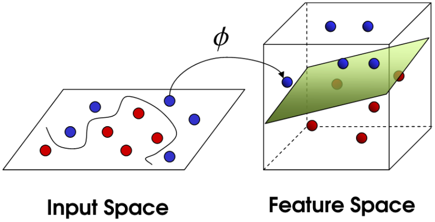
- The SVC outperforms logistic regression by 0.5% on this MNIST dataset
  - good, but not a huge improvement
- Note that the SVC can also be implemented using svm.SVC with `kernel='linear'`, but it's slower, especially with many training samples

Outline

- Motivating Example: Recognizing Handwritten Digits
- Maximum-Margin Classification
- The Support Vector Classifier
- The Support Vector Machine

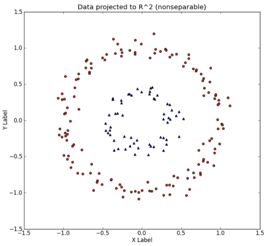
Support vector machine

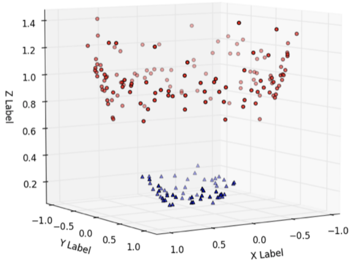
- Support vector classifier (SVC)
  - works very well when the features are nearly linearly separable
- Support vector machine (SVM)
  - SVC plus ...
    1) a feature transformation $\boldsymbol{x} \mapsto \boldsymbol{\phi}(\boldsymbol{x})$ to enhance linear separability
    2) the "kernel trick" for fast implementation



Feature transformations

- We want a feature transformation $\boldsymbol{x} \mapsto \boldsymbol{\phi}(\boldsymbol{x})$ that leads to linear separability
- Usually we do this by mapping to an appropriate higher-dimensional space
- Example: For the $\boldsymbol{x} = [x_1, x_2]^\mathsf{T}$ data below, which is not linearly separable, we can use $\boldsymbol{\phi}(\boldsymbol{x}) = [x_1, x_2, x_1^2 + x_2^2]^\mathsf{T}$ to get linear separability!





The problem of dimensionality

- In principle, given transformed features $\phi(\boldsymbol{x})$, we could train via

$$(\boldsymbol{w}_*, b_*) = \min_{\boldsymbol{w},b} \left\{ C \sum_{i=1}^{n} \max\{0, 1 - y_i z_i\} + \frac{1}{2}\|\boldsymbol{w}\|^2 \right\} \text{ for } z_i = b + \boldsymbol{w}^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{x}_i)$$

  and classify a test vector $\boldsymbol{x}$ via

$$\widehat{y} = \operatorname{sgn}(z) = \operatorname{sgn}\left[b_* + C \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{x}) \right]$$

- But what if the dimension of $\phi(\boldsymbol{x})$ is very high?
  - We would need a very high dimensional $\boldsymbol{w}$
  - Leads to computational issues and overfitting!
- And what if $\phi(\boldsymbol{x})$ is infinite-dimensional?
  - This might sound crazy, but—as we'll see—it's true for popular $\phi(\cdot)$
  - We're stuck; we can't handle infinite dimensional $\boldsymbol{w}$

The Lagrangian dual formulation of the SVC

- We can avoid the high-dimensional $\boldsymbol{w}_*$ space using the "Lagrangian dual" form of the SVC. First, find the Lagrange multipliers $\{\lambda_i\}$ that solve

$$\min_{\boldsymbol{\lambda}} \left\{ \sum_{i=1}^n \lambda_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^\mathsf{T} \boldsymbol{x}_j \right\} \text{ s.t. } \lambda_i \in [0, C] \text{ and } \sum_{i=1}^n \lambda_i y_i = 0,$$

after which one could optionally compute

$$\boldsymbol{w}_* = \sum_{i=1}^n \lambda_i y_i \boldsymbol{x}_i \quad \text{and} \quad b_* = \boldsymbol{w}_*^\mathsf{T} \boldsymbol{x}_{i_*} - y_{i_*} \text{ for any } i_* \text{ s.t. } \lambda_{i_*} > 0$$

  - The derivation is outside the scope of this course

    See Sec. 12.2 of Hastie, Tibshirani, Friedman, *The Elements of Statistical Learning, 2nd Ed.*, 2009

  - But note the similarity to our gradient analysis on page 20 (using $\lambda_i = C\alpha_i$)

- Given $\boldsymbol{\lambda}$, we can directly compute the decision $\widehat{y}$ as follows (without $\boldsymbol{w}_*$):

$$\widehat{y} = \operatorname{sgn}\left(b_* + \sum_{i=1}^n \lambda_i y_i \boldsymbol{x}_i^\mathsf{T} \boldsymbol{x} \right) \text{ where } b_* = \sum_{i=1}^n \lambda_i y_i \boldsymbol{x}_i^\mathsf{T} \boldsymbol{x}_{i_*} - y_{i_*}$$

The kernel trick

- Now consider the dual-form SVC with $\boldsymbol{x}$ replaced by the transformation $\boldsymbol{\phi}(\boldsymbol{x})$:

$$\widehat{y} = \operatorname{sgn}\left(b_* + \sum_{i=1}^{n} \lambda_i y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathsf{T}} \boldsymbol{\phi}(\boldsymbol{x})\right) \text{ where } b_* = \sum_{i=1}^{n} \lambda_i y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathsf{T}} \boldsymbol{\phi}(\boldsymbol{x}_{i_*}) - y_{i_*}$$

  where i_* is any support index, and where $\{\lambda_i\}$ solve

$$\min_{\boldsymbol{\lambda}} \left\{ \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathsf{T}} \boldsymbol{\phi}(\boldsymbol{x}_j) \right\} \text{ s.t. } \begin{cases} \lambda_i \in [0, C] \\ \sum_{i=1}^{n} \lambda_i y_i = 0 \end{cases}$$

- The transformed features appear only in the form of a "kernel" function

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') \triangleq \boldsymbol{\phi}(\boldsymbol{x})^{\mathsf{T}} \boldsymbol{\phi}(\boldsymbol{x}')$$

  - Often, directly computing $\kappa(\boldsymbol{x}, \boldsymbol{x}')$ is easier than computing $\boldsymbol{\phi}(\boldsymbol{x})$ and $\boldsymbol{\phi}(\boldsymbol{x}')$
  - For both learning and inference, <u>we only need to compute the kernel</u> (not $\boldsymbol{\phi}(\boldsymbol{x})$)!
  - Called the "kernel trick"

Popular kernels

- Popular kernels include:
  - radial basis function (RBF): $\kappa(\boldsymbol{x},\boldsymbol{x}') = \exp(-\gamma\|\boldsymbol{x}-\boldsymbol{x}'\|^2)$ for "width" $\gamma > 0$
  - polynomial: $\kappa(\boldsymbol{x},\boldsymbol{x}') = (\gamma\boldsymbol{x}^\mathsf{T}\boldsymbol{x}' + r)^d$ where $\gamma > 0$, $r \geq 0$, and often $d = 2$
  - sigmoidal: $\kappa(\boldsymbol{x},\boldsymbol{x}') = \tanh(\gamma\boldsymbol{x}^\mathsf{T}\boldsymbol{x}' + r)$ for some $\gamma > 0$ and $r > 0$
  - linear: $\kappa(\boldsymbol{x},\boldsymbol{x}') = \boldsymbol{x}^\mathsf{T}\boldsymbol{x}'$ ...corresponds to trivial transformation $\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{x}$
- Each kernel measures a "similarity" between $\boldsymbol{x}$ and $\boldsymbol{x}'$
- Note: The feature space of the RBF kernel is infinite dimensional!

$$\exp(-\tfrac{1}{2}\|\boldsymbol{x}-\boldsymbol{x}'\|^2) = \exp(-\tfrac{1}{2}\|\boldsymbol{x}\|^2)\exp(-\tfrac{1}{2}\|\boldsymbol{x}'\|^2)\sum_{j=0}^{\infty}\frac{(\boldsymbol{x}^\mathsf{T}\boldsymbol{x}')^j}{j!}$$

  - Can't implement the RBF kernel via $\boldsymbol{\phi}(\boldsymbol{x})$ since infinite dimensional
  - But can implement RBF efficiently using the kernel trick!

Implementing the SVM in sklearn

- The SVM is easy to implement in sklearn using svm.SVC
  - Choose from several kernels: rbf (default), poly, sigmoid, linear, or custom
  - Specify the kernel width $\gamma > 0$
  - Control number of support vectors via $C > 0$
  - Ideally, you should optimize all parameters via cross-validation
    - See example in sklearn documentation
  - Using MNIST settings from `https://martin-thoma.com/svm-with-sklearn/`:

```
from sklearn import svm
svm = svm.SVC(probability=False, kernel="rbf", C=2.8, gamma=.0073, verbose=1)
svm.fit(Xtr,ytr)
yhat_ts = svm.predict(Xts)
acc = np.mean(yhat_ts == yts)
print('Accuracy = {0:f}'.format(acc))
```

```
[LibSVM]Accuracy = 0.971900
```

- The SVM significantly outperforms both MLR and SVC!
  - Could do even better with more training samples and tweaked parameters

Multiclass SVM

There are several ways to train an SVM for $K > 2$ classes:

- One-versus-rest:
  - For each class k, separately train a binary 1-vs-rest classifier $(b_k, \boldsymbol{w}_k)$
  - This is implemented in svm.LinearSVC, but only for the linear kernel (i.e., SVC)
- Crammer-Singer:
  - Jointly train a K-ary classifier $(\boldsymbol{b}, \boldsymbol{W})$
  - This is implemented in svm.LinearSVC, but only for the linear kernel (i.e., SVC)
  - It gives slightly better accuracy than the above OVR, but is slower
- One-versus-one:
  - For each unique class pair (k, k') such that $k \neq k'$, separately train a binary classifier $(b_{k,k'}, \boldsymbol{w}_{k,k'})$. There are $K(K-1)/2$ such pairs
  - This is implemented in svm.SVC for a variety of kernels (i.e., SVM), but it's slower than svm.LinearSVC
  - svm.NuSVC gives an equivalent formulation, but instead of using C it uses $\nu \in (0, 1]$, which is a lower bound on the fraction of support vectors

Learning objectives

- Gain experience with linear classification of images
  - representing images, displaying images, classifying images
- Understand the geometry of the linear classification boundary:
  - orthogonality to $\boldsymbol{w}$, offset b, margin $1/\|\boldsymbol{w}\|$
- Understand the margin-maximizing classifier
- Understand the support vector classifier (SVC)
- Understand the support vector machine (SVM)
- Understand how to implement the SVC and SVM with sklearn