Detour
\( L_2 \)-regularized regression

- Recall the definition of \( L_2 \)-regularized regression:

\[
\begin{align*}
\hat{f}_{\text{MSE}}(w) &= \frac{1}{2n} (X^T w - y)^\top (X^T w - y) + \frac{\alpha}{2n} w^\top w \\
\nabla_w f_{\text{MSE}} &= \frac{1}{n} X (X^T w - y) + \frac{\alpha}{n} w = 0
\end{align*}
\]
**$L_2$-regularized regression**

- Recall the definition of $L_2$-regularized regression:

$$f_{\text{MSE}}(w) = \frac{1}{2n} (X^\top w - y)^\top (X^\top w - y) + \frac{\alpha}{2n} w^\top w$$

$$\nabla_w f_{\text{MSE}} = \frac{1}{n} X (X^\top w - y) + \frac{\alpha}{n} w = 0$$

$$XX^\top w + \alpha w = Xy$$
$L_2$-regularized regression

- Recall the definition of $L_2$-regularized regression:

$$f_{MSE}(w) = \frac{1}{2n} (X^T w - y)^\top (X^T w - y) + \frac{\alpha}{2n} w^\top w$$

$$\nabla_w f_{MSE} = \frac{1}{n} X (X^T w - y) + \frac{\alpha}{n} w = 0$$

$$XX^\top w + \alpha w = Xy$$

$$(XX^\top + \alpha I) w = Xy$$
**L₂-regularized regression**

- Recall the definition of $L₂$-regularized regression:

$$f_{\text{MSE}}(w) = \frac{1}{2n} (\mathbf{X}^\top \mathbf{w} - \mathbf{y})^\top (\mathbf{X}^\top \mathbf{w} - \mathbf{y}) + \frac{\alpha}{2n} \mathbf{w}^\top \mathbf{w}$$

$$\nabla_w f_{\text{MSE}} = \frac{1}{n} \mathbf{X} (\mathbf{X}^\top \mathbf{w} - \mathbf{y}) + \frac{\alpha}{n} \mathbf{w} = 0$$

$$\mathbf{X}\mathbf{X}^\top \mathbf{w} + \alpha \mathbf{w} = \mathbf{X}\mathbf{y}$$

$$\left(\mathbf{X}\mathbf{X}^\top + \alpha \mathbf{I}\right) \mathbf{w} = \mathbf{X}\mathbf{y}$$

$$\mathbf{w} = \left(\mathbf{X}\mathbf{X}^\top + \alpha \mathbf{I}\right)^{-1} \mathbf{X}\mathbf{y} \quad \text{for } m \times m$$
Matrix inversion lemma
(special case)

- For any $\alpha > 0$ and $m \times n$ matrix $X$:
  $$(XX^\top + \alpha I)^{-1} = \frac{1}{\alpha} I - \frac{1}{\alpha} X(X^\top X + \alpha I)^{-1} X^\top$$
Matrix inversion lemma (special case)

- For any $\alpha > 0$ and $m \times n$ matrix $X$:
  \[
  (XX^\top + \alpha I)_{m \times m}^{-1} = \frac{1}{\alpha}I - \frac{1}{\alpha}X(X^\top X + \alpha I)_{n \times n}^{-1}X^\top
  \]

- The RHS method is much faster when $n < m$!
Kernel trick
Feature transformations

• The conceptually simplest approach to training a classifier using transformed features is:
  • Transform each example \( x \) into \( \phi(x) \).
  • Train on the transformed data \( \phi(x^{(1)}), \ldots, \phi(x^{(n)}) \)

• At test time:
  • Transform the test point \( x \) to \( \phi(x) \); then classify \( \phi(x) \).

• This can be done for any ML model.

\[
\begin{align*}
    \text{x} & \xrightarrow{\text{Input transformer / feature extractor}} \phi(x) \xrightarrow{\text{ML model}} g(\phi(x))
\end{align*}
\]
Feature transformations

- To train a model in this way, we could easily construct the design matrix of transformed examples:

\[
\tilde{X} = \begin{bmatrix}
\phi(x^{(1)}) & \ldots & \phi(x^{(n)}) \\
\end{bmatrix} \\
m \times n
\]

- We can then pass \( \tilde{X} \) to the SVM solver:

```python
svm = sklearn.svm.SVC(kernel='linear')
svm.fit(Xtilde, y)
```
Feature transformations

• While this works fine in principle, for certain kinds of models — those that can be **kernelized** — the process can be made:
  • More efficient.
  • More powerful.
• SVMs are probably the most prominent kernelizable ML model...
Kernelization

• Recall that, in an SVM, the optimal $\mathbf{w}$ will always be a **linear combination** of the data points $\mathbf{x}^{(i)}$, weighted by the $\alpha^{(i)}$.

• Only the support vectors — those examples $\mathbf{x}^{(i)}$ such that $\alpha^{(i)} > 0$ — will contribute to $\mathbf{w}$:

\[
L(\mathbf{w}, b, \alpha) = \frac{1}{2} \mathbf{w}^\top \mathbf{w} - \sum_{i=1}^{n} \alpha^{(i)} \left( y^{(i)} \left( \mathbf{x}^{(i)} \top \mathbf{w} + b - 1 \right) \right)
\]

\[
\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)}
\]

\[
\implies \mathbf{w} = \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)}
\]
Dual form

This also suggests a different way of optimizing an SVM:

Instead of optimizing over \( w \in \mathbb{R}^m \), where \( m \) is size of the feature vector (e.g., number of image pixels), we can optimize over \( \alpha \in \mathbb{R}^n \), where \( n \) is the number of training examples.

\[
L(w, b, \alpha) = \frac{1}{2} w^T w - \sum_{i=1}^{n} \alpha^{(i)} \left( y^{(i)} \left( x^{(i)^T} w + b - 1 \right) \right)
\]

\[
\frac{\partial L}{\partial w} = w - \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} x^{(i)}
\]

\[
\implies w = \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} x^{(i)}
\]
Dual form

• Suppose we are training a smile detector, where the number of features $m = 10,000$ and $n=1000$ (examples).

• Which would you rather optimize: $w \in \mathbb{R}^m$ or $\alpha \in \mathbb{R}^n$?

\[
L(w, b, \alpha) = \frac{1}{2} w^\top w - \sum_{i=1}^{n} \alpha^{(i)} \left( y^{(i)} \left( x^{(i)} \top w + b - 1 \right) \right)
\]

\[
\frac{\partial L}{\partial w} = w - \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} x^{(i)}
\]

\[
\implies w = \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} x^{(i)}
\]

Show support_vectors.py demo
Dual form

- Optimizing over $\alpha$ instead of $\mathbf{w}$ is called the dual form of the constraint optimization.

- Optimizing $\mathbf{w}$ directly is called the primal form.

- Both approaches give the same solution.

- Training the SVM in dual form requires that we manipulate the function $L$ algebraically a bit first...
Kernelization

- By setting $\frac{\partial L}{\partial b}$ to 0 and solving, we can deduce:

\[
L(w, b, \alpha) = \frac{1}{2} w^\top w - \sum_{i=1}^{n} \alpha^{(i)} \left( y^{(i)} \left( x^{(i)} \top w + b - 1 \right) \right)
\]

\[
\frac{\partial L}{\partial b} = - \sum_{i=1}^{n} \alpha^{(i)} y^{(i)}
\]

\[\Rightarrow \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} = 0\]
Kernelization

- We can now substitute for $\mathbf{w}$ into $L$ and simplify:

$$L(\mathbf{w}, b, \alpha) = \frac{1}{2} \mathbf{w}^\top \mathbf{w} - \sum_{i=1}^{n} \alpha^{(i)} \left( y^{(i)} \left( \mathbf{x}^{(i)} \mathbf{w} + b - 1 \right) \right)$$

$$= \frac{1}{2} \left| \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)} \right|^2 - \sum_{i=1}^{n} \alpha^{(i)} \left( y^{(i)} \left( \mathbf{x}^{(i)} \mathbf{w} + b - 1 \right) \right)$$
Kernelization

• We can now substitute for $w$ into $L$ and simplify:

\[
L(w, b, \alpha) = \frac{1}{2} w^\top w - \sum_{i=1}^{n} \alpha^{(i)} \left( y^{(i)} \left( x^{(i)} \top w + b - 1 \right) \right) \\
= \frac{1}{2} \left| \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} x^{(i)} \right|^2 - \sum_{i=1}^{n} \alpha^{(i)} \left( y^{(i)} \left( x^{(i)} \top \left( \sum_{i'=1}^{n} \alpha^{(i')} y^{(i')} x^{(i')} \right) + b - 1 \right) \right) \\
\implies L(\alpha) = \sum_{i=1}^{n} \alpha^{(i)} - \frac{1}{2} \sum_{i=1}^{n} \sum_{i'=1}^{n} \alpha^{(i)} \alpha^{(i')} y^{(i)} y^{(i')} x^{(i)} \top x^{(i')} 
\]

Only a function of $\alpha$ now. The training data occur only as inner products in the function $L$ that we optimize.
Kernelization

- At test time, we compute the inner product between \( \mathbf{x} \) and \( \mathbf{w} \):

\[
\mathbf{x}^\top \mathbf{w} + b = \mathbf{x}^\top \left( \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)} \right) + b
\]
Kernelization

• At test time, we compute the inner product between $\mathbf{x}$ and $\mathbf{w}$:

$$\mathbf{x}^\top \mathbf{w} + b = \mathbf{x}^\top \left( \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)} \right) + b$$

$$= \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \mathbf{x}^\top \mathbf{x}^{(i)} + b$$

• The result depends only on the inner products between the test point $\mathbf{x}$ and each of the support vectors $\mathbf{x}^{(i)}$. 
Kernelization

• Both during training and testing, we only use each training point $x^{(i)}$ as part of an inner product — we don’t need the raw values themselves.

• Therefore, even if we want to transform each input using $\phi$, we only really need to know the inner products between each $\phi(x)$ and $\phi(x^{(i)})$ (for training):

$$L(\alpha) = \sum_{i=1}^{n} \alpha^{(i)} - \frac{1}{2} \sum_{i=1}^{n} \sum_{i'=1}^{n} \alpha^{(i)} \alpha^{(i')} y^{(i)} y^{(i')} \phi(x^{(i)})^\top \phi(x^{(i')})$$
Kernelization

• Both during training and testing, we only use each training point \( x^{(i)} \) as part of an inner product — we don’t need the raw values themselves.

• Therefore, even if we want to transform each input using \( \phi \), we only really need to know the inner products between each \( \phi(x) \) and \( \phi(x^{(i)}) \) (for testing):

\[
x^\top w + b = \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \phi(x)^\top \phi(x^{(i)}) + b
\]
Kernelization

- For training, rather than compute $\phi(x^{(i)})$ for every training example $x^{(i)}$: ...

$$\tilde{X} = \begin{bmatrix} \phi(x^{(1)}) & \ldots & \phi(x^{(n)}) \end{bmatrix}$$

$m \times n$
Kernelization

• ...instead compute the kernel matrix containing all pairs of inner products:

\[ K = \begin{bmatrix} 
\phi(x^{(1)})^\top \phi(x^{(1)}) & \ldots & \phi(x^{(1)})^\top \phi(x^{(n)}) \\
\vdots & \ddots & \vdots \\
\phi(x^{(n)})^\top \phi(x^{(1)}) & \ldots & \phi(x^{(n)})^\top \phi(x^{(n)}) 
\end{bmatrix}_{n \times n} \]
Kernelization

- Then we just need to pass $\mathbf{K}$ to the SVM solver:

```python
svm = sklearn.svm.SVC(kernel='precomputed')
K = Xtilde.T.dot(Xtilde)  # $\mathbf{K} = \tilde{\mathbf{X}}^\top \tilde{\mathbf{X}}$
svm.fit(K, y)
```
Kernelization

• $K$ is an $n \times n$ matrix, where $n$ is # training examples.

• Suppose $n=1000$, $m=10000$ (e.g., 100x100 pixels).

• Storing each $\phi(x^{(i)})$ explicitly would take $O(10,000,000)$ bytes.

• Storing just $K$ will take $O(1,000,000)$ bytes — 10x less!

• Training the SVM in dual form can also be much faster (for $n \ll m$).