CS 453X: Class 13

Jacob Whitehill

Detour

L_2-regularized regression

- Recall the definition of L_2-regularized regression:

$$
\begin{aligned}
f_{\mathrm{MSE}}(\mathbf{w}) &= \frac{1}{2n}\left(\mathbf{X}^\top\mathbf{w}-\mathbf{y}\right)^\top\left(\mathbf{X}^\top\mathbf{w}-\mathbf{y}\right)+\frac{\alpha}{2n}\mathbf{w}^\top\mathbf{w} \\
\nabla_{\mathbf{w}} f_{\mathrm{MSE}} &= \frac{1}{n}\mathbf{X}\left(\mathbf{X}^\top\mathbf{w}-\mathbf{y}\right)+\frac{\alpha}{n}\mathbf{w}=0
\end{aligned}
$$

L_2-regularized regression

- Recall the definition of L_2-regularized regression:

$$
\begin{aligned}
f_{\mathrm{MSE}}(\mathbf{w}) &= \frac{1}{2n}\left(\mathbf{X}^\top \mathbf{w} - \mathbf{y}\right)^\top \left(\mathbf{X}^\top \mathbf{w} - \mathbf{y}\right) + \frac{\alpha}{2n}\mathbf{w}^\top \mathbf{w} \\
\nabla_{\mathbf{w}} f_{\mathrm{MSE}} &= \frac{1}{n}\mathbf{X}\left(\mathbf{X}^\top \mathbf{w} - \mathbf{y}\right) + \frac{\alpha}{n}\mathbf{w} = 0 \\
\mathbf{X}\mathbf{X}^\top \mathbf{w} + \alpha \mathbf{w} &= \mathbf{X}\mathbf{y}
\end{aligned}
$$

L_2-regularized regression

- Recall the definition of L_2-regularized regression:

$$
\begin{aligned}
f_{\mathrm{MSE}}(\mathbf{w}) &= \frac{1}{2n}\left(\mathbf{X}^\top\mathbf{w}-\mathbf{y}\right)^\top\left(\mathbf{X}^\top\mathbf{w}-\mathbf{y}\right)+\frac{\alpha}{2n}\mathbf{w}^\top\mathbf{w} \\
\nabla_{\mathbf{w}} f_{\mathrm{MSE}} &= \frac{1}{n}\mathbf{X}\left(\mathbf{X}^\top\mathbf{w}-\mathbf{y}\right)+\frac{\alpha}{n}\mathbf{w}=0 \\
\mathbf{X}\mathbf{X}^\top\mathbf{w}+\alpha\mathbf{w} &= \mathbf{X}\mathbf{y} \\
\left(\mathbf{X}\mathbf{X}^\top+\alpha\mathbf{I}\right)\mathbf{w} &= \mathbf{X}\mathbf{y}
\end{aligned}
$$

L_2-regularized regression

- Recall the definition of L_2-regularized regression:

$$\begin{aligned}
f_{\mathrm{MSE}}(\mathbf{w}) &= \frac{1}{2n}\left(\mathbf{X}^\top\mathbf{w}-\mathbf{y}\right)^\top\left(\mathbf{X}^\top\mathbf{w}-\mathbf{y}\right)+\frac{\alpha}{2n}\mathbf{w}^\top\mathbf{w} \\
\nabla_{\mathbf{w}} f_{\mathrm{MSE}} &= \frac{1}{n}\mathbf{X}\left(\mathbf{X}^\top\mathbf{w}-\mathbf{y}\right)+\frac{\alpha}{n}\mathbf{w}=0 \\
\mathbf{X}\mathbf{X}^\top\mathbf{w}+\alpha\mathbf{w} &= \mathbf{X}\mathbf{y} \\
\left(\mathbf{X}\mathbf{X}^\top+\alpha\mathbf{I}\right)\mathbf{w} &= \mathbf{X}\mathbf{y} \\
\mathbf{w} &= \underbrace{\left(\mathbf{X}\mathbf{X}^\top+\alpha\mathbf{I}\right)}_{m \times m}^{-1}\mathbf{X}\mathbf{y}
\end{aligned}$$

Matrix inversion lemma (special case)

- For any $\alpha > 0$ and m x n matrix $\mathbf{X}$:

$$(\mathbf{X}\mathbf{X}^\top + \alpha \mathbf{I})^{-1} = \frac{1}{\alpha}\mathbf{I} - \frac{1}{\alpha}\mathbf{X}(\mathbf{X}^\top\mathbf{X} + \alpha\mathbf{I})^{-1}\mathbf{X}^\top$$

Matrix inversion lemma (special case)

- For any $\alpha > 0$ and m x n matrix $\mathbf{X}$:

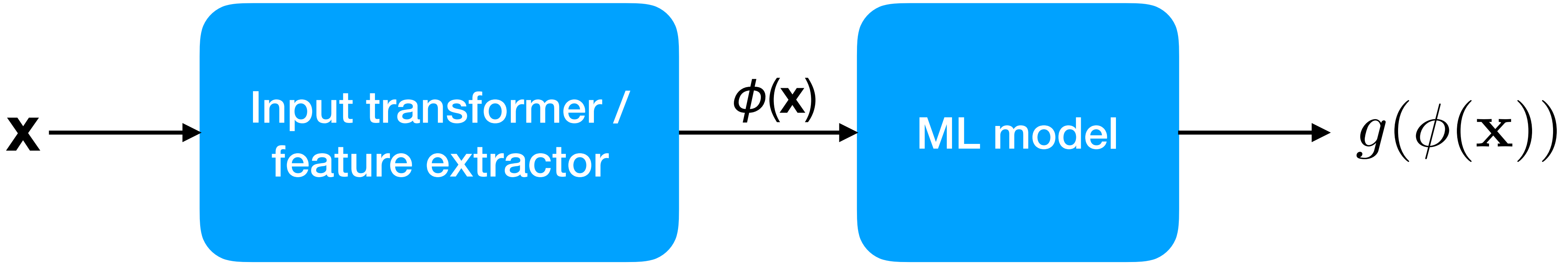
$$(\underbrace{\mathbf{X}\mathbf{X}^\top + \alpha\mathbf{I}}_{m \times m})^{-1} = \frac{1}{\alpha}\mathbf{I} - \frac{1}{\alpha}\mathbf{X}(\underbrace{\mathbf{X}^\top\mathbf{X} + \alpha\mathbf{I}}_{n \times n})^{-1}\mathbf{X}^\top$$

- 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 $\phi(\mathbf{x})$.
  - Train on the transformed data $\phi(\mathbf{x}^{(1)}), \ldots, \phi(\mathbf{x}^{(n)})$
- At test time:
  - Transform the test point **x** to $\phi(\mathbf{x})$; then classify $\phi(\mathbf{x})$.
- This can be done for *any* ML model.



Feature transformations

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

$$\tilde{\mathbf{X}} = \begin{bmatrix} | & & | \\ \phi\left(\mathbf{x}^{(1)}\right) & \dots & \phi\left(\mathbf{x}^{(n)}\right) \\ | & & | \end{bmatrix}$$

m* x *n

- We can then pass $\tilde{\mathbf{X}}$ to the SVM solver:

```
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 **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 **w**:

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

Dual form

- This also suggests a different way of optimizing an SVM:
  - Instead of optimizing over $\mathbf{w} \in \mathbb{R}^m$, 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.

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

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: $\mathbf{w} \in \mathbb{R}^m$ or $\alpha \in \mathbb{R}^n$?

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

Show support_vectors.py demo

Dual form

- Optimizing over *α* instead of **w** is called the **dual form** of the constraint optimization.

- Optimizing **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:

$$
\begin{aligned}
L(\mathbf{w}, b, \alpha) &= \frac{1}{2}\mathbf{w}^\top\mathbf{w} - \sum_{i=1}^{n} \alpha^{(i)} \left(y^{(i)} \left({\mathbf{x}^{(i)}}^\top \mathbf{w} + b - 1\right)\right) \\
\frac{\partial L}{\partial b} &= -\sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \\
\implies \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} &= 0
\end{aligned}
$$

Kernelization

- We can now substitute for **w** into *L* and simplify:

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

Kernelization

- We can now substitute for **w** into *L* and simplify:

$$
\begin{aligned}
L(\mathbf{w}, b, \alpha) &= \frac{1}{2}\mathbf{w}^\top \mathbf{w} - \sum_{i=1}^{n} \alpha^{(i)} \left(y^{(i)} \left({\mathbf{x}^{(i)}}^\top \mathbf{w} + b - 1 \right) \right) \\
&= \frac{1}{2}\left| \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)} \right|^2 - \sum_{i=1}^{n} \alpha^{(i)} \left(y^{(i)} \left({\mathbf{x}^{(i)}}^\top \left(\sum_{i'=1}^{n} \alpha^{(i')} y^{(i')} \mathbf{x}^{(i')} \right) + b - 1 \right) \right) \\
\implies L(\alpha) &= \sum_{i=1}^{n} \alpha^{(i)} - \frac{1}{2} \sum_{i=1}^{n} \sum_{i'=1}^{n} \alpha^{(i)} \alpha^{(i')} y^{(i)} y^{(i')} {\mathbf{x}^{(i)}}^\top \mathbf{x}^{(i')}
\end{aligned}
$$

Only a function of *α* 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 **x** and **w**:

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

Kernelization

- At test time, we compute the inner product between **x** and **w**:

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

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

Kernelization

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

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

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

Kernelization

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

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

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

Kernelization

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

$$\tilde{\mathbf{X}} = \left[\begin{array}{ccc} | & & | \\ \phi\left(\mathbf{x}^{(1)}\right) & \dots & \phi\left(\mathbf{x}^{(n)}\right) \\ | & & | \end{array} \right]$$

m* x *n

Kernelization

- …instead compute the **kernel matrix** containing all pairs of inner products:

$$\mathbf{K} = \begin{bmatrix} \phi(\mathbf{x}^{(1)})^\top \phi(\mathbf{x}^{(1)}) & \ldots & \phi(\mathbf{x}^{(1)})^\top \phi(\mathbf{x}^{(n)}) \\ & \ddots & \\ \phi(\mathbf{x}^{(n)})^\top \phi(\mathbf{x}^{(1)}) & \ldots & \phi(\mathbf{x}^{(n)})^\top \phi(\mathbf{x}^{(n)}) \end{bmatrix}$$

n* x *n

Kernelization

- Then we just need to pass **K** to the SVM solver:

```
svm = sklearn.svm.SVC(kernel='precomputed')
K = Xtilde.T.dot(Xtilde)    # K = X̃ᵀX̃
svm.fit(K, y)
```

$\mathbf{K} = \tilde{\mathbf{X}}^\top \tilde{\mathbf{X}}$

Kernelization

- **K** is an n x n matrix, where n is # training examples.
- Suppose n=1000, m=10000 (e.g., 100x100 pixels).
- Storing each $\phi(\mathbf{x}^{(i)})$ 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 $n \ll m$).