Support Vector Machine
Kernel Trick

M. R. Hasan
Readings

• Alpaydin: 10.3, 13.1, 13.2
• Murphy: 14.5.2.2
• Geron: chapter 5, appendix C
What We Will Cover

• Kernel Trick
• Gaussian Radial Basis Function (RBF) Kernel
Linearly Non-Separable Data

Feature Augmentation:
Map data into a higher-dimensional space

Increased dimension of the feature space increases the complexity!

As a solution use Kernel SVM

Bernhard Boser, Isabelle Guyon and Vladimir Vapnik, 1992

Solve the margin maximization problem on a high-dimensional feature space without actually adding the features (kernel trick).
Solve the margin maximization problem on a high-dimensional feature space *without actually adding* the features (*kernel trick*).

**Polynomial Kernel:** new features are the polynomial and interaction terms of the existing features.

**Gaussian Radial Basis Function (RBF) Kernel:** new features are created based on *similarity* with the existing features.

**Analogy-based learning**
SVM: Linearly Non-separable Data - Hard Margin Classifier

Linearly Non-separable Data

Kernel SVM

Bernhard Boser, Isabelle Guyon and Vladimir Vapnik, 1992
Hard Margin Linear SVM Classification

- Dual QP makes the kernel trick possible, while the primal does not.

Primal QP:

\[ L(w, b, \alpha) = \frac{1}{2} \|w\|^2 + \sum_{i=1}^{N} \alpha_i \left[ 1 - y_i(w^T \hat{x}_i + b) \right] \]

Dual QP:

\[ L(w^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \hat{x}_i^T \hat{x}_j \]
Hard Margin Linear SVM Classification

• Kernel trick makes it possible to get the **same result as if we added many polynomial features**, even with very high-degree polynomials, **without actually having to add them.**

• So there is **no combinatorial explosion** of the number of features since we don’t actually add any features.

• It’s like a **magic!**

$$\Phi(x_1) = (x_1, x_1^2)$$
Hard Margin Linear SVM Classification

• To understand the kernel trick, observe that in the dual QP, data \{x_i\} only appears in pairs as \(x_i^T x_j\).

• To solve a linearly non-separable problem, we increase dimensionality by feature “augmentation” using some basis function \(\Phi(x_i)\).

\[
\hat{x}_i \rightarrow \phi(\hat{x}_i)
\]

Kernel trick makes this replacement trivial without blowing the complexity.

\[
\mathcal{L}(\overrightarrow{w^*}, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \hat{x}_i^T \hat{x}_j
\]

\[
\mathcal{L}(\overrightarrow{w^*}, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \phi(\hat{x}_i^T)\phi(\hat{x}_j)
\]
• Example: Take two \textbf{2D vectors} \(a\) and \(b\) and apply the 2\textsuperscript{nd} degree polynomial mapping using a basis function \(\phi()\).

\[
\phi(a_1, a_2) = a_1^2 + \sqrt{2}a_1a_2 + a_2^2 \quad \phi(b_1, b_2) = b_1^2 + \sqrt{2}b_1b_2 + b_2^2
\]

Then compute the \textbf{dot product} of the \textbf{augmented} vectors.

\[
\phi(a)^T \cdot \phi(b) = \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2
\]

\[
= (a_1b_1 + a_2b_2)^2 = \left( \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (a^T \cdot b)^2
\]

The dot product of the transformed vectors is \textbf{equal to the square} of the dot product of the \textbf{original} vectors:

\[
\Phi(a)^T \cdot \Phi(b) = (a^T \cdot b)^2
\]
Kernel SVM

- What **insight** did we gain from this observation?
- If we apply the transformation \( \varphi \) to all training instances, then the **dual problem will contain the dot product** \( \varphi(x_i)^T \varphi(x_j) \).

\[
\mathcal{L}(\vec{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \langle \hat{x}_i^T \hat{x}_j \rangle
\]

But if \( \varphi \) is the 2\textsuperscript{nd} degree polynomial transformation, then we can **replace this dot product of transformed vectors** simply by \((x_i^T x_j)^2\)
So we don’t actually need to transform the training instances at all!
Just replace the dot product by its square.

\[
\phi(a)^T \cdot \phi(b) = \left(\sqrt{2}a_1a_2\right)^T \cdot \left(\sqrt{2}b_1b_2\right) = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2
\]
\[
= (a_1b_1 + a_2b_2)^2 = \left(\frac{a_1^T}{a_2} \cdot \frac{b_1}{b_2}\right)^2 = (a^T \cdot b)^2
\]

\[
\phi(x) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}
\]

\[
\mathcal{L}(\mathbf{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j
\]

\[
\mathcal{L}(\mathbf{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)
\]

\[
\mathcal{L}(\mathbf{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i^T \mathbf{x}_j)^2
\]
Kernel SVM

• The result will be strictly the same as if we went through the trouble of actually transforming the training set then fitting a linear SVM algorithm.

• But this trick makes the whole process much more computationally efficient.

• This is the essence of the kernel trick.

\[
\phi(a)^T \cdot \phi(b) = \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2 = (a_1b_1 + a_2b_2)^2 = \left( \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (a^T \cdot b)^2
\]

\[
\mathcal{L}(\hat{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \hat{x}_i^T \hat{x}_j
\]

\[
\mathcal{L}(\hat{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (\hat{x}_i^T \hat{x}_j)^2
\]
Kernel SVM

\[
L(\vec{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (\vec{x}_i^T \vec{x}_j)^2
\]

- We use a **function** that gives the product of the basis functions by computing the **power of the feature dot product**.
- For example, we used the function \(K(\vec{x}_i^T, \vec{x}_j) = (\vec{x}_i^T \vec{x}_j)^2\)

It’s called the **kernel function**.
Kernel SVM

\[ \mathcal{L}(\vec{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (\vec{x}_i^T \vec{x}_j)^2 \]

- The kernel function \( K(\vec{x}_i^T, \vec{x}_j) = (\vec{x}_i^T \vec{x}_j)^2 \) is actually computing the inner/dot product of points in \( \mathbb{R}^3 \).

It is implicitly mapping our points from \( \mathbb{R}^2 \) to \( \mathbb{R}^3 \).

It’s like in the dream we are visiting the \( \mathbb{R}^3 \) space!

\[ \mathcal{L}(\vec{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \phi(\vec{x}_i^T) \phi(\vec{x}_j) \]

\[
\phi(x) = \begin{pmatrix} x_1 \\
\sqrt{2} x_1 x_2 \\
x_2 
\end{pmatrix}
\]

\[ \mathcal{L}(\vec{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(\vec{x}_i^T, \vec{x}_j) \]
Kernel SVM

![Kernel SVM equation]

- The function $K(a, b) = (a^T b)^2$ is a 2\textsuperscript{nd} degree polynomial kernel.
- In Machine Learning, a kernel is a function capable of computing the dot product of the augmented features $\phi(a)^T \phi(b)$ based only on the original vectors $a$ and $b$, without having to compute (or even to know about) the transformation $\phi$. 
Kernel SVM

- Some of the **most commonly used kernels**.

<table>
<thead>
<tr>
<th>Kernel Type</th>
<th>Kernel Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>$K(a, b) = a^T \cdot b$</td>
</tr>
<tr>
<td>Polynomial</td>
<td>$K(a, b) = (\gamma a^T \cdot b + r)^d$</td>
</tr>
<tr>
<td>Gaussian RBF</td>
<td>$K(a, b) = \exp\left(-\gamma |a - b|^2\right)$</td>
</tr>
<tr>
<td>Sigmoid</td>
<td>$K(a, b) = \tanh\left(\gamma a^T \cdot b + r\right)$</td>
</tr>
</tbody>
</table>
Kernel SVM

• The **d-degree polynomial kernel** is used **slightly differently** than the one that we described.

\[ K(\vec{a}, \vec{b}) = (\gamma \vec{a}^T \vec{b} + r)^d \]

• The **parameter r** controls the **relative weighting** between the **lower order** and the **higher order terms**.

• Let’s illustrate this using the **2nd degree** polynomial kernel.

<table>
<thead>
<tr>
<th>Type</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>( K(a, b) = a^T \cdot b )</td>
</tr>
<tr>
<td>Polynomial</td>
<td>( K(a, b) = (\gamma a^T \cdot b + r)^d )</td>
</tr>
<tr>
<td>Gaussian RBF</td>
<td>( K(a, b) = \exp \left( -\gamma | a - b |^2 \right) )</td>
</tr>
<tr>
<td>Sigmoid</td>
<td>( K(a, b) = \tanh \left( \gamma a^T \cdot b + r \right) )</td>
</tr>
</tbody>
</table>
Kernel SVM

• Consider the following 2nd degree polynomial kernel.

\[ K(\vec{a}, \vec{b}) = (\gamma \vec{a}^T \vec{b} + r)^2 \]

\[ K(\vec{a}, \vec{b}) = (\vec{a}^T \vec{b} + r)^2 \] For convenience \( \gamma = 1 \)

\[ K(\vec{a}, \vec{b}) = (\vec{a}^T \vec{b})^2 + 2\vec{a}^T \vec{b}r + r^2 \]

For an \( m \)-dimensional feature space

\[ K(\vec{a}, \vec{b}) = (a_1 b_1 + a_2 b_2 + \ldots + a_m b_m)^2 + 2(a_1 b_1 + a_2 b_2 + \ldots + a_m b_m)r + r^2 \]
Kernel SVM

For an $m$-dimensional feature space

$$K(\vec{a}, \vec{b}) = (a_1 b_1 + a_2 b_2 + \ldots + a_m b_m)^2 + 2(a_1 b_1 + a_2 b_2 + \ldots + a_m b_m)r + r^2$$

For a 2D feature space ($m = 2$):

$$K(\vec{a}, \vec{b}) = (a_1 b_1 + a_2 b_2)^2 + 2ra_1 b_1 + 2ra_2 b_2 + r^2$$

$$K(\vec{a}, \vec{b}) = (a_1 b_1 + a_2 b_2)^2 + \sqrt{2}ra_1 \sqrt{2}rb_1 + \sqrt{2}ra_2 \sqrt{2}rb_2 + r^2$$

$$K(\vec{a}, \vec{b}) = a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2 + \sqrt{2}ra_1 \sqrt{2}rb_1 + \sqrt{2}ra_2 \sqrt{2}rb_2 + r^2$$
Kernel SVM

\[ K(\vec{a}, \vec{b}) = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2 + \sqrt{2}ra_1\sqrt{2}rb_1 + \sqrt{2}ra_2\sqrt{2}rb_2 + r^2 \]

We can show that \( K(\vec{a}, \vec{b}) \) computes the dot product between the augmented (2\textsuperscript{nd} degree polynomial) features \( \phi(\vec{a}) \) and \( \phi(\vec{b}) \).

Observe that the parameter \( r \) controls the relative weighting between the first order \((a_i \text{ and } b_i)\) and the second order terms.
Kernel SVM

\[ K(\vec{a}, \vec{b}) = (\vec{a}^T \vec{b} + r)^2 = \phi(\vec{a})^T \phi(\vec{b}) \]

The dot product of the 2nd degree polynomial augmented features \( \phi(\vec{a})^T \phi(\vec{b}) \) is computed based only on the original vectors \( \vec{a} \) and \( \vec{b} \).
Kernel SVM

• What if the degree of the polynomial is 1000?

The dot product between 1000-dimensional vectors:

\[ \phi(\vec{a})^T \phi(\vec{b}) = K(\vec{a}, \vec{b}) = (\vec{a}^T \vec{b} + r)^{1000} \]

The dot product of the 2\(^{nd}\) degree polynomial augmented features \(\phi(\vec{a})^T \phi(\vec{b})\) is computed based only on the original vectors \(\vec{a}\) and \(\vec{b}\).
We have seen that using the polynomial kernel we can map the features onto a higher dimensional space.

And this happens in our “dream” (we don’t actually map the features).

In dream anything is possible!

Can we dream that we have mapped the features onto an infinite dimensional space?

Polynomial Kernel: \( K(\vec{a}, \vec{b}) = (\gamma \vec{a}^T \vec{b} + r)^d \)
Kernel SVM

• This is exactly what we can accomplish with the **Gaussian RBF kernel**.

• It uses a **radial basis function (RBF)** for mapping features to an infinite dimensional space.

• RBF: \( \phi(\hat{x}, \hat{c}) = \phi(||\hat{x} - \hat{c}||) \)

• A RBF is a real-valued function \( \phi \) that transforms \( \hat{x} \) by using its **distance** from another point \( \hat{c} \) (the center).
Kernel SVM

• RBF: \( \phi(\hat{x}, \hat{c}) = \phi(||\hat{x} - \hat{c}||) \)

• It can be shown that using the **Gaussian RBF kernel** \( \phi \) actually maps each training instance to an **infinite-dimensional space**.

• Let’s illustrate this.

\[
\begin{align*}
\text{Linear:} & \quad K(a, b) = a^T \cdot b \\
\text{Polynomial:} & \quad K(a, b) = (\gamma a^T \cdot b + r)^d \\
\text{Gaussian RBF:} & \quad K(a, b) = \exp(-\gamma \|a - b\|^2) \\
\text{Sigmoid:} & \quad K(a, b) = \tanh(\gamma a^T \cdot b + r)
\end{align*}
\]
Kernel SVM

• Gaussian RBF kernel.

\[ K(\vec{a}, \vec{b}) = \exp\left(-\gamma \|\vec{a} - \vec{b}\|^2\right) \]

For convenience \( \gamma = 1 \)

For a 2D feature space:

\[ K(\vec{a}, \vec{b}) = \exp\left(-(a_1 - b_1)^2 - (a_2 - b_2)^2\right) \]

\[ K(\vec{a}, \vec{b}) = \exp\left(-a_1^2 + 2a_1b_1 - b_1^2 - a_2^2 + 2a_2b_2 - b_2^2\right) \]
Kernel SVM

- Gaussian RBF kernel.

$$K(\vec{a}, \vec{b}) = \exp(-a_1^2 + 2a_1b_1 - b_1^2 - a_2^2 + 2a_2b_2 - b_2^2)$$

$$K(\vec{a}, \vec{b}) = \exp(-(a_1^2 + a_2^2))\exp(-(b_1^2 + b_2^2))\exp(2a_1b_1 + 2a_2b_2)$$

$$K(\vec{a}, \vec{b}) = \exp(-\|\vec{a}\|^2)\exp\left(-\|\vec{b}\|^2\right)\exp(2\vec{a}^T\vec{b})$$

The above expression holds for \textit{d-dimensional} \(\vec{a}\) and \(\vec{b}\), where \(d\) can be any \textbf{arbitrary integer}. 
Gaussian RBF kernel.

\[ K(\vec{a}, \vec{b}) = \exp(-\|\vec{a}\|^2) \exp\left(-\|\vec{b}\|^2\right) \exp(2\vec{a}^T \vec{b}) \]

\[ K(\vec{a}, \vec{b}) = \exp(-\|\vec{a}\|^2) \exp\left(-\|\vec{b}\|^2\right) \sum_{n=0}^{\infty} \frac{(2\vec{a}^T \vec{b})^n}{n!} \]

Last term can be written as an infinite sum!

Using Maclaurin series expansion

\[ e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots. \]

The length of this vector is infinite!

Thus \( K(\vec{a}, \vec{b}) \) implicitly creates all possible higher-order terms of \( \vec{a} \) to find its “similarity” with \( \vec{b} \).
Kernel SVM

- The Gaussian RBF kernel corresponds to the **feature mapping** to an **infinite vector**.

**Polynomial Kernel:**

\[
K(\tilde{a}, \tilde{b}) = \sum_{i,j=1}^{m} (a_i a_j)(b_i b_j) + \sum_{i=1}^{m} (\sqrt{2} r a_i)(\sqrt{2} r b_i) + r^2
\]

\[
K(\tilde{a}, \tilde{b}) = \exp(-||\tilde{a}||^2)\exp(-||\tilde{b}||^2) \sum_{n=0}^{\infty} \frac{(2\tilde{a}^T \tilde{b})^n}{n!}
\]

We wanted to know the “similarity” or the distance of a sample \(\tilde{a}\) from another point \(\tilde{b}\).

Gaussian RBF “similarity” measure is an **infinite-dimensional** vector.

We use this vector as a mapping of \(\tilde{a}\).
Kernel SVM

- Gaussian RBF kernel **implicitly maps every point** to an infinite dimensional space.

\[
e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots
\]

\[
K(\vec{a}, \vec{b}) = \exp(-\|\vec{a}\|^2) \exp\left(-\|\vec{b}\|^2\right) \sum_{m=0}^{\infty} \frac{(2\vec{a}^T \vec{b})^m}{m!}
\]

The mapping to the infinite space is computed **based only on the original vectors**: sample \(\vec{a}\) (that we want to map) and a **fixed reference** point \(\vec{b}\).

\[
K(\vec{a}, \vec{b}) = \exp(-\|\vec{a}\|^2) \exp\left(-\|\vec{b}\|^2\right) \exp(2\vec{a}^T \vec{b})
\]
Kernel SVM

Let’s use the Gaussian RBF kernel function to **augment features** and compute the dual Lagrangian.

\[
\mathcal{L}(\boldsymbol{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \phi(\tilde{x}_i^T) \phi(\tilde{x}_j)
\]

\[
\phi(\tilde{x}_i) = K(\tilde{x}_i, \tilde{l}) = \exp \left( -\gamma \| \tilde{x}_i - \tilde{l} \|^2 \right)
\]

\[
\phi(\tilde{x}_j) = K(\tilde{x}_j, \tilde{l}) = \exp \left( -\gamma \| \tilde{x}_j - \tilde{l} \|^2 \right)
\]

Here \( \tilde{l} \) is another sample used to measure similarity, i.e., to augment \( \tilde{x}_i \) and \( \tilde{x}_j \)

\( \phi(\tilde{x}_i) \) and \( \phi(\tilde{x}_j) \) are two infinite dimensional vectors!
\[ \mathcal{L}(\vec{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \phi(\vec{x}_i^T) \phi(\vec{x}_j) \]

\( \phi(\vec{x}_i) \) and \( \phi(\vec{x}_j) \) are two infinite dimensional vectors!

\[ \phi(\vec{x}_i) = K(\vec{x}_i, \vec{l}) = \exp \left( -\gamma \| \vec{x}_i - \vec{l} \|^2 \right) \]

\[ \phi(\vec{x}_j) = K(\vec{x}_j, \vec{l}) = \exp \left( -\gamma \| \vec{x}_j - \vec{l} \|^2 \right) \]

\[ K(\vec{x}_i, \vec{l}) = \exp( -\| \vec{x}_i \|^2 ) \exp \left( -\| \vec{l} \|^2 \right) \exp( 2\vec{x}_i^T \vec{l} ) \]

\[ K(\vec{x}_j, \vec{l}) = \exp \left( -\| \vec{x}_j \|^2 \right) \exp \left( -\| \vec{l} \|^2 \right) \exp( 2\vec{x}_j^T \vec{l} ) \]

\( \phi(\vec{x}_i^T) \phi(\vec{x}_j) \): Product of two infinite dimensional features is computed using the original features.
Kernel SVM

• We conclude that the following kernels allow us to visit high-dimensional feature space without actually transforming the features.
• But how are these two kernels different?
• They visit the “dreamland” or the new feature space using two very different techniques.

**Polynomial Kernel:** \( K(\vec{a},\vec{b}) = (\gamma \vec{a}^T \vec{b} + r)^d \)

**Gaussian RBF Kernel:** \( K(\vec{a},\vec{b}) = \exp \left( -\gamma \| \vec{a} - \vec{b} \|^2 \right) \)
Kernel SVM

- These two kernels visit the “dreamland” or the new feature space using two very different techniques.

**Polynomial Kernel:** \( K(\hat{a}, \hat{b}) = (\gamma \hat{a}^T \hat{b} + r)^d \)

- New features are created by computing the higher order polynomials of the original features.

**Gaussian RBF Kernel:** \( K(\hat{a}, \hat{b}) = \exp \left( -\gamma \| \hat{a} - \hat{b} \|^2 \right) \)

- New features are created by **measuring the similarity** of the original features with fixed reference points (neighbors).
Linearly Non-Separable Data

Kernel SVM

Polynomial Kernel: new features are the polynomial and interaction terms of the existing features.

Gaussian Radial Basis Function (RBF) Kernel: new features are created based on similarity with the existing features.

Solve the margin maximization problem on a high-dimensional feature space **without actually adding** the features (**kernel trick**).

Let’s understand how the Gaussian RBF kernel creates the features based on similarity and **why it produces an amazing result!!**
Kernel SVM

• Our goal is to augment (map) features by computing the existing features’ similarity with some predefined instances.
• For this we need a similarity function.
• How do we create features that are computed using a similarity function?

For example, given a 1D feature, we can map it into a high dimensional (e.g., 2D) feature space.
Kernel SVM

- To define the **order of the higher dimension**, we use some fixed number of points.
- These points are called **landmarks** (red circles).
- Then, we measure how much each instance **resembles a particular landmark**.

The landmarks should be selected from the **neighbor points**.
Kernel SVM

- For example, let’s take a 1D dataset that is *not linearly separable* (green triangles & blue squares).
- We will project each feature $x_1$ into a 2D feature space $x_2$ and $x_3$.

Let’s add two landmarks to it at $x_1 = -2$ and $x_1 = 1$

These landmarks are chosen randomly from the two existing neighbor instances.
Kernel SVM

Gaussian RBF Kernel: \( K(\mathbf{a}, \mathbf{b}) = \exp\left(-\gamma \|\mathbf{a} - \mathbf{b}\|^2\right) \)

- Next, let’s define the similarity function to be the **Gaussian Radial Basis Function** (RBF) with \( \gamma = 0.3 \)

\[
\phi_\gamma(x, \ell) = \exp\left(-\gamma \| x - \ell \|^2\right)
\]

Gaussian RBF is a bell-shaped function **varying** from 0 (very far away from the landmark) to 1 (at the landmark)
Kernel SVM

• Let’s see how the **new features are computed** by the Gaussian RBF.
• For example, let’s look at the instance $x_1 = -1$:
• It is located at a distance of 1 from the 1\(^{\text{st}}\) landmark, and 2 from the 2\(^{\text{nd}}\) landmark.

Therefore, this instance’s (-1) **new features** are

- $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$ and
- $x_3 = \exp(-0.3 \times 2^2) \approx 0.30$.

$$\phi_y(x, \ell) = \exp \left( -\gamma \| x - \ell \|^2 \right)$$
Kernel SVM

- The plot on the right shows the transformed dataset (dropping the original features): $x \rightarrow \Phi(x)$
- See that it is now linearly separable.

$x_2 \approx 0.74$
$x_3 \approx 0.30$
Kernel SVM

- Let’s take a **closer look** at the Gaussian RBF kernel.
- It defines a spherical kernel.
- The **radius (bandwidth)** is denoted by $s$.

$$K(\hat{x}^t, \hat{x}) = \exp \left[ -\frac{\|\hat{x}^t - \hat{x}\|^2}{2s^2} \right]$$

$$K(\hat{x}^t, \hat{x}) = \exp \left[ -\gamma \|\hat{x}^t - \hat{x}\|^2 \right]$$
Kernel SVM

• Notice that $\gamma$ is inversely proportional to radius $s$.
• The test point $\tilde{x}^t$ (that we want to augment) is the center and it measures its similarity with other points $\tilde{x}$ that are within its radius.

$$K(\tilde{x}^t, \tilde{x}) = \exp \left[ -\frac{||\tilde{x}^t - \tilde{x}||^2}{2s^2} \right]$$

$$K(\tilde{x}^t, \tilde{x}) = \exp[-\gamma ||\tilde{x}^t - \tilde{x}||^2]$$
Kernel SVM

• Thus the radius $s$ (or $\gamma$) defines the **range of influence** of $\tilde{x}^t$.

\[
\gamma = \text{small value}: \text{indicates that radius of influence is large.}
\]

The data point measures its **similarity with all data points** and gives only an average measure of the decision boundary that results into **underfitting**.

\[
K(\tilde{x}^t, \tilde{x}) = \exp \left[ -\frac{||\tilde{x}^t - \tilde{x}||^2}{2s^2} \right]
\]

\[
K(\tilde{x}^t, \tilde{x}) = \exp[ -\gamma ||\tilde{x}^t - \tilde{x}||^2]
\]
Kernel SVM

• Thus the radius $s$ (or $\gamma$) defines the range of influence of $\hat{x}^t$.

\[ K(\hat{x}^t, \hat{x}) = \exp \left[ -\frac{||\hat{x}^t - \hat{x}||^2}{2s^2} \right] \]

\[ K(\hat{x}^t, \hat{x}) = \exp[\gamma ||\hat{x}^t - \hat{x}||^2] \]

$\gamma = \text{large value}$: indicates that radius of influence is small.

It will include only itself in its range of influence. As a consequence, the decision boundary ends up being more irregular, wiggling around individual instances. For a very large value, the model will overfit.
Kernel SVM

- The parameter $\gamma$ needs to be tuned carefully for optimal performance.
- See the notebook “Support Vector Machine: Investigation of Gaussian RBF Kernel SVM” for an empirical discussion on the tuning of $\gamma$.

$$K(\tilde{x}_t, \tilde{x}) = \exp \left[ - \frac{\|\tilde{x}_t - \tilde{x}\|^2}{2s^2} \right]$$

$$K(\tilde{x}_t, \tilde{x}) = \exp [-\gamma \|\tilde{x}_t - \tilde{x}\|^2]$$
Prediction in Kernel SVM
• The solution of the Kernel SVM is computed based on the augmented (high-dimensional) feature space.
• Thus, the dimension of the parameter vector $\vec{w}$ would be high (potentially infinite dimensional)!
• How do we use an infinite-dimensional $\vec{w}$ to make predictions on new samples?

$$L(\vec{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \vec{x}_i^T \vec{x}_j$$

$$\vec{w}^* = \sum_{i=1}^{N} \alpha_i^* y_i \vec{x}_i$$

$\vec{w}$ is infinite dimensional!

Prediction for a new sample $\vec{x}_n$

$$h_{\vec{w}^*, b^*}(\phi(\vec{x}_n)) = (\vec{w}^*)^T (\phi(\vec{x}_n)) + b^*$$

$$\vec{x}_i \rightarrow \phi(\vec{x}_i)$$
Kernel SVM

- Another benefit of the kernel trick is that it makes prediction computationally simple for a new data.

- We can plug in the formula for \( \mathbf{w} \) from into the decision function for a new instance \( \mathbf{x}_n \), and we get an equation with only dot products between input vectors.

\[
\begin{align*}
\hat{h}_{\mathbf{w}^*, b^*} (\phi(\mathbf{x}_n)) &= (\mathbf{w}^*)^T (\phi(\mathbf{x}_n)) + b^* \\
\mathbf{w}^* &= \sum_{i=1}^{N} \alpha_i^* y_i \phi(\mathbf{x}_i) \\
&= \sum_{i=1}^{N} \alpha_i^* y_i (\mathbf{x}_i)^T \phi(\mathbf{x}_n) + b^* \\
&= \sum_{i=1}^{N} \alpha_i^* y_i K(\mathbf{x}_i, \mathbf{x}_n) + b^*
\end{align*}
\]
Kernel SVM

- Note that since \( \alpha_i \neq 0 \) only for support vectors, making predictions involve computing the dot product of the new input vector \( x_n \) with only the support vectors \((x_i)\), not with all the training instances.

\[
h_{\mathbf{w}^*, b^*}(\phi(\mathbf{x}_n)) = (\mathbf{w}^*)^T(\phi(\mathbf{x}_n)) + b^*
\]

Prediction for a new sample \( \mathbf{x}_n \)

\[
\mathbf{w}^* = \sum_{i=1}^{N} \alpha_i^* y_i \phi(\mathbf{x}_i)
\]

\[
= \sum_{i=1}^{N} \alpha_i^* y_i (\phi(\mathbf{x}_i))^T \phi(\mathbf{x}_n) + b^*
\]

\[
= \sum_{i=1}^{N} \alpha_i^* y_i K(\mathbf{x}_i, \mathbf{x}_n) + b^*
\]
Kernel SVM

- The decision function requires us to compute $b^*$:

$$b^* = \frac{1}{N_{\text{support vectors}}} \sum_{i=1}^{N} \left[ 1 - y_i (\hat{w}^T \phi(\tilde{x}_i)) \right]$$

where $\alpha_i^* > 0$.

The kernel function makes this computation feasible in higher dimensional feature space!

$$h_{\hat{w}^*, b^*}(\phi(\tilde{x}_n)) = (\hat{w}^*)^T (\phi(\tilde{x}_n)) + b^*$$
Kernel Trick: Theoretical Justification
Kernel Trick: Theoretical Justification

• Given some function $K$, how can we tell if it is a valid kernel?
• For example, can we tell if there is some feature mapping $\phi$ so that $K(x, z) = \phi(x)^T\phi(z)$ for all $x, z$?
• It appears that there is a theoretical justification to the kernel trick.

• It’s called the Mercer’s theorem.
• Let’s illustrate this theorem.
Kernel Trick: Theoretical Justification

• **Mercer’s theorem:**
  • Suppose that \( K \) is a valid kernel corresponding to some feature mapping \( \varphi \).
  • Then, we can create a square matrix of kernel values \( K_{ij} = K(\tilde{x}_i, \tilde{x}_j) \)

\[
K(\tilde{x}_i, \tilde{x}_j) = \begin{bmatrix}
K(\tilde{x}_1, \tilde{x}_1) & K(\tilde{x}_1, \tilde{x}_2) & \cdots & K(\tilde{x}_1, \tilde{x}_N) \\
K(\tilde{x}_2, \tilde{x}_1) & K(\tilde{x}_2, \tilde{x}_2) & \cdots & K(\tilde{x}_2, \tilde{x}_N) \\
\vdots & \vdots & \ddots & \vdots \\
K(\tilde{x}_N, \tilde{x}_1) & K(\tilde{x}_N, \tilde{x}_2) & \cdots & K(\tilde{x}_N, \tilde{x}_N)
\end{bmatrix}
\]
Kernel Trick: Theoretical Justification

- In this $N \times N$ matrix we list the values of the Kernels on every pair of samples in the dataset.
- It’s called the **Gram matrix**.
- It computes the **dot product of the augmented features** for every pair of samples in the dataset.

\[
K(\tilde{x}_i, \tilde{x}_j) = \begin{bmatrix}
K(\tilde{x}_1, \tilde{x}_1) & K(\tilde{x}_1, \tilde{x}_2) & \cdots & K(\tilde{x}_1, \tilde{x}_N) \\
K(\tilde{x}_2, \tilde{x}_1) & K(\tilde{x}_2, \tilde{x}_2) & \cdots & K(\tilde{x}_2, \tilde{x}_N) \\
\vdots & \vdots & \ddots & \vdots \\
K(\tilde{x}_N, \tilde{x}_1) & K(\tilde{x}_N, \tilde{x}_2) & \cdots & K(\tilde{x}_N, \tilde{x}_N)
\end{bmatrix}
\]
Kernel Trick: Theoretical Justification

- Mercer’s theorem states that $K$ is a valid kernel if its Gram matrix is **symmetric** and **positive semidefinite**.
  - Symmetric: $x_i^T x_j = x_j^T x_i$
  - Positive semidefinite: $x_i^T x_j \geq 0$ (distances should be zero or positive)

\[
K(\tilde{x}_i, \tilde{x}_j) = \begin{bmatrix}
K(\tilde{x}_1, \tilde{x}_1) & K(\tilde{x}_1, \tilde{x}_2) & \cdots & K(\tilde{x}_1, \tilde{x}_N) \\
K(\tilde{x}_2, \tilde{x}_1) & K(\tilde{x}_2, \tilde{x}_2) & \cdots & K(\tilde{x}_2, \tilde{x}_N) \\
\vdots & \vdots & \ddots & \vdots \\
K(\tilde{x}_N, \tilde{x}_1) & K(\tilde{x}_N, \tilde{x}_2) & \cdots & K(\tilde{x}_N, \tilde{x}_N)
\end{bmatrix}
\]
Kernel Trick: Beyond SVM
Kernel Trick: Generalizability

• Can we apply the kernel trick in **other learning models**?

• To answer this question we need to recall **why kernel trick was possible in SVM**.

• In the dual optimization problem of SVM, the data \{x_i\} only appears in pairs as \(x_i^T x_j\).

• After augmenting the features, it also appears in pairs of inner products of the basis functions: \(\Phi(x_i)^T \Phi(x_j)\)

\[
\mathcal{L}(\vec{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \tilde{x}_i^T \tilde{x}_j
\]

\[
\mathcal{L}(\vec{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \phi(\tilde{x}_i^T)\phi(\tilde{x}_j)
\]
Kernel Trick: Generalizability

- Thus, this inner product of the features allows us to replace the the inner product of the basis functions by the kernel.
- Now say that we have a learning algorithm that we can write in terms of only inner products between input attribute vectors $x_1$ and $x_2$.
- Then by replacing this with the kernel $K(x_1, x_2)$, we can “magically” allow our algorithm to work efficiently in the high dimensional feature space corresponding to $K$.

$$
\mathcal{L}(\vec{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \phi(\vec{x}_i^T) \phi(\vec{x}_j)
$$

$$
\mathcal{L}(\vec{w}^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(\vec{x}_i^T, \vec{x}_j)
$$
Kernel Trick: Generalizability

- Thus the idea of kernels has significantly broader applicability beyond SVMs.
- For instance, this kernel trick can be applied with the perceptron to derive a kernel perceptron algorithm.
- Also kernel trick can be applied to the Principle Component Analysis (PCA) dimensionality reduction technique to create Kernel PCA.
Kernel SVM: Analogy Based Learning
Kernel SVM & K-NN

- Now that we have presented the Kernel SVM, observe its **closeness with the K-Nearest Neighbors** approach.
- In both models, we compute a **similarity measure** of the data points with other data points.
- Kernel SVM uses these similarity measures to **augment the feature space** and then do the classification.
- K-NN uses the similarity measures for **direct classification**.

It is interesting to note that both models use similarity or **analogy** for learning!
Learning by **Analogy**!

<table>
<thead>
<tr>
<th>Tribe</th>
<th>Strength</th>
<th>Tools</th>
<th>Inspiration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolists</td>
<td>Formal Systems</td>
<td>Decision Tree, Inductive Logic</td>
<td>Logic, Philosophy</td>
</tr>
<tr>
<td>Connectionists</td>
<td>Neural Networks</td>
<td>Backpropagation, Reinforcement Learning</td>
<td>Neuroscience</td>
</tr>
<tr>
<td>Bayesians</td>
<td>Probabilistic Inference</td>
<td>Hidden Markov Model (HMM), Graphical Models</td>
<td>Statistics</td>
</tr>
<tr>
<td>Evolutionaries</td>
<td>Structure Learning</td>
<td>Genetic Algorithms, Evolutionary Programming</td>
<td>Biology</td>
</tr>
<tr>
<td>Analogizers</td>
<td>Mapping to Novelty</td>
<td>K-NN, Support Vector Machine (SVM)</td>
<td>Psychology</td>
</tr>
</tbody>
</table>
Learning by Analogy

- We swim in a **vast ocean of analogies**.
- All intelligent behaviors **reduce to analogy**.

Douglas Hofstadter (World’s best known Analogizer): *Everything we learn is the result of analogy in action!*