Appunti della tesi di PhD di Da San Martino

A.A. 2023/2024

Rosso Carlo

Abstract

For dealing with structured data, kernel methods seems to have strong theoretical background. They do not require explicit vectorial representation, instead it is defined a kernel function that measures the similarity between the two objects. This approach has two problems:

- kernel for trees should not be sparse: we use ML to group data, if they are sparse, we aren't able to group them;
- it is slow: the kernel function is used both in the prediction and in the training phase, so it needs to be fast.

This paper presents three alternative methods:

1. **kernel composition**: having representations too spare to be useful, a kernel function is applied to the tree structure, so the data are projected onto a lower dimensional space with the property that similar structures are mapped similarly. Then it is applied a second kernel function to the projected data, which is the "logic" function;
2. **convolutional kernel**: the kernel function which measures the similarity between two trees uses convolution;
3. **DAG**: the trees are exploded into a directed acyclic graph, instead of a forest of trees, thus the rapresentation is more dense.

Contents

1 Introduction

1.1 Kernel methods

Here we introduce the kernel methods. By definition, kernel methods look for linear relations in the feature space.

Basically, input items are compared via dot products of their representation in the feature space (the most similar items have the highest dot product). Actually, the dot product is replaced by a kernel function, which needs to be symmetric positive and semidefinite, so that it is applied directly in the original space (without the need to compute the feature space). So the computational complexity is not dependent on the size of the feature space but on the complexity of the kernel function (the generalization capability of a kernel method depends on the number of misclassified examples in the learning phase).

Briefly, the classification of a new example is computed via weighted sum of kernel evaluations between the example and a subset of the training instances (note that it is very similar to the concept of algebric basis of a vector space). In this way the technique is able to directly handle structured data (if the kernel function is appropriate enough).

Kernel functions have two interesting properties:

- they are closed under **addition** and **linear combination**. This makes different inputs easy to combine;
- they scale on the number of different examples, not on the number of features, so they do not scale on the dimensionality of the data, but on the number of the data. This is very interesting if we consider words as single data.

Finally, kernel function are super cool, but it is hard to find fast to compute and expressive ones.

1.2 Thesis motivation

Completely expressive kernels for graphs are NP-hard to compute (so not feasible).

If the data are too sparse, the kernel function is going to give sparse results as well (it's not going to be accurate enough). So we think about a way to make data more dense (using other kernels).

Another interesting thing might be keeping the information about the position of the substructures in the trees for a better classification. Kinda making it more convolutional and considering more the context of each node.

Finally, since the kernel function works on raw data, the model needs to store those raw data, which can be a problem if the data are too big. Indeed, if the kernel function doesn't have enough raw data to compare the new example with, it looses accuracy; in fact, the accuracy of the classifier improves with the size of the training set.

2 Notation

Since I've already studied most of the notation, I am going to write only the new stuff.

2.1 Statistical Learning Theory

The VC (Vapnik-Chervonenkis) dimension of a family of functions H is defined as the cardinality of the largest subset of points of the domain that can be labelled arbitrarily by choosing a function $h \in H$. For example for $H = ax, VC(H) = 2$ (I think).
There is a cool theorem which uses the VC dimension: let v be the VC dimension of the family of functions H. Then $\forall \delta > 0, h \in H$ dependent from a set of parameters Θ, the upper bound

$$R(h(\Theta)) \leq R_e(h(\Theta)) + \Omega\left(\frac{VC(h)}{n}\right) \tag{2.1}$$

where R_e is the empirical risk (space under the loss curve) and n is the size of the training set, holds with probability of at least $1 - \delta$ for $n > VC(h)$. Note that $\Omega\left(\frac{VC(h)}{n}\right)$ is a monotonic increasing function and it is called the confidence interval.

Given the reported theorem, we can see that if the VC dimension increases (and n remain constant), then the expected risk gets a bigger upper bound, which means that H might generalize poorly. On the other hand, the bigger n the lower the upper bound gets.
When a function is able to correctly classify the training set but has a large error on the rest of the distribution, then the function is told to overfit the data ($VC >> n$).

2.2 Self Organizing Maps

The aim of the Self Organizing Maps (SOM) learning arlgorithm is to learn a feature map

$$\mathcal{M} : \mathcal{I} \to \mathcal{A} \tag{2.2}$$

This is obtained by associating each point in $\mathcal{A}$ to a different neuron. High dimentional input vectors are projected into the two (actually $n \in \mathbb{N}$) dimensional coordinates of the lattice, with the aim of preserving, as much as possible, the topological relationships among the input vectors (which means that close input vectors are associated to neurons which are close on the lattice).

An example

When the input space is a structured domain with labels in $\mathcal{U}$, we redefine Equation 2.2 to be:

$$\mathcal{M}^{\#} : \mathcal{U}^{\#[i,o]} \to \mathcal{A} \tag{2.3}$$

Where we define $\mathcal{M}^{\#}$ recursively as:

$$\mathcal{M}^{\#}(G) = \begin{cases} nil_{\mathcal{A}} & \text{if } G = \xi \\ \mathcal{M}_{node}(u_s, \mathcal{M}^{\#}(G^{(1)}), \ldots, \mathcal{M}^{\#}(G^{(o)})) & otherwise \end{cases} \tag{2.4}$$

Where $s = source(G), G^{(1)}, \ldots, G^{(o)}$ are the subgraphs pointed by the outgoing edges leaving from s, $nil_{\mathcal{A}}$ represents the 0 and u_s is the label of the s node; finally, $\mathcal{M}_{node}$ is a SOM, defined on a generic node, which takes in input the label of the node and the "encoding" of the subgraphs $(G^{(1)}, \ldots, G^{(o)})$ according to the $M^{\#}$ map. So

$$\mathcal{M}_{node} : \mathcal{U} \times \mathcal{A} \times \cdots \times \mathcal{A} \to \mathcal{A} \tag{2.5}$$

Training lagorithm

The weights associated with each neuron in the q dimensional lattice $\mathcal{M}_{node}$ can be trained as follow:

Step 1: **Competitive step**: the winning neuron, at iteration t, with the closest weight vector to the input vector is selected:

$$y_{i^*}(t) = \arg\min_{c_i} ||\Lambda(x_v(t) - m_{c_i}(t))|| \tag{2.6}$$

where Λ is used to balance the importance of labels;

Step 2: **Comparative step**: the weight vector $m_{y_{i^*}}$, as well as the weight vector of neurons in the topological neighborhood of the winning neuron, are moved closer to the input vector:

$$m_{c_r}(t+1) = m_{c_r}(t) + \eta(t) f(\Delta_{i^* r})(x_v(t) - m_{c_r}(t)) \tag{2.7}$$

where $\Delta_{i^* r}$ is the topological distance between c_r and c_{i^*} in the lattice. Usually $f(\cdot)$ takes the form of a Gaussian function to update very near neurons and almost ignore far away ones. For example:

$$f(x) = \exp\left(-\frac{x^2}{2\sigma(t)^2}\right) \tag{2.8}$$

Usually the neighborhood radius $\sigma(t)$ decreases to zero along with the training.

The described model (SOM-SD) allows the processing of undirected graphs, and non-positional graphs where the order of edges is not relevant. The heuristic nature of the model can not formally guarantee to preserve the topology of the items in the input space.

2.3 Kernel Methods

The class of kernel methods comprises all those algorithms that do not require an explicit representation of the examples but only information about the similarities among them. Any kernel method can be decomposed into two modules:

- a problem specific kernel function (to get the differences between the different inputs);

- a general purpose learning algorithm.

The modularity of the approach allows to study representation and optimization independently. Wahba's representer theorem states that the solution of certain optimization problems involving an empirical risk term and a quadratic regularizer can be written in terms of an expansion of the training examples. Thus, given a dataset $S = \{(x_i, y_i) : i = 1, \ldots, n\}$ and a kernel function K, the solution w of the problem can be expressed as:

$$w = \sum_{i}^{n} \alpha_i y_i \phi(x_i) \tag{2.9}$$

Now let's introduce the score function:

$$S = x) = \sum_{i}^{n} \alpha_i y_i \phi(x_i)\phi(x) = \sum_{i}^{n} \alpha_i y_i K(x_i, x) \tag{2.10}$$

Note that the score function can be expressed as a weighted linear combination of kernel function evaluations between examples in the dataset and x. Here follows the classification function:

$$c(x) = sign(S(x)) = sign\left(\sum_{i}^{n} \alpha_i y_i K(x_i, x)\right) \tag{2.11}$$

Perceptron

The perceptron is meant to classify data encoded by real vectors with a linear decision function (a hyperplane). Every element of the dataset is represented by a feature vector. A prototype vector w is randomly initialized. Then the classification of each example x_i is compared to the one made by the prototype, computed according to the following formula:

$$f(x) = sign(w \cdot x_i + b) \tag{2.12}$$

If the perceptron in classifying uncorrectly te example, then a new prototype w' is computed as:

$$w' = w + \alpha y_i x_i \tag{2.13}$$

where α is a constant ($\alpha > 0$) and $y_i \in \{-1, 1\}$ is the correct classification of the example.
Using the kernel trick it is possible to extend the perceptron to generate a non-linear decision funciton and/or treat structured data by using kernels.
The on-line kernel-perceptron algorithm, adapted to tree-kernels, requires to keep in memory the set of the already seen examples for which the perceptron prediction was erroneous. Thus we can consider the set of examples $M = \{(x_i, y_i) \in S : \alpha_i \in \{-1, 1\}\}$ (I think that α_i is some sort of weight which is going to be stored with the example, so that it can be used by the kernel function (K)) as the model of the perceptron and slightly redefine the kernel-perceptron algorithm as in the following. Let $M = \emptyset$ be an initial empty model, a new example x_i is added to the model M whenever its score

$$S(x_i) = \sum_{(x_j, y_j) \in M} y_j K(x_i, x_j) \tag{2.14}$$

has different sign from its classification y_i. For many applications, the cardinality of M grows linearly with the number of tree presentations. Moreover, the efficiency in the evaluation of the function $S(x)$ decreases super-linearly. Clearly, this seems not satisfactory for on-line applications.

Support Vector Machines

Support Vector Machines (SVM) are based on the Structural Risk Minimization principle for which bounds on the generalization error have been proven. SVM is a binary classifier which projects the examples in feature space and then looks for an hyperplane separating positive nad negative examples. In particular, it is chosen the hyperplane which maximizes the margin between the two classes.
Given a linearly separable dataset, the separating hyperplane maximizing the margin corresponds to the solution of the following optimization problem:

$$\arg\min_{w,b} \frac{||w||^2}{2} \tag{2.15}$$

$$\text{subject to } \forall (x_i, y_i) \in S,\ y_i(w \cdot \phi(x_i) + b) \geq 1 \tag{2.16}$$

where w and b define the hyperplane in the feature space. The margin is inversely proportional to the norm of the weight vector w ($\gamma = 1/||w||$), so minimizing the norm of w is equivalent to maximizing the margin.
The representer theorem states that the solution f of the problem can be expressed as:

$$\forall x \in \mathcal{X}, f(x) = \sum_{i=1}^{n} \alpha_i K(x_i, x) \tag{2.17}$$

The examples for which the corresponding α is not 0 are called support vectors (note that this is also very similar to the concept of algebric basis of a vector space, where the basis is the set of support vectors).

Kernel Functions

The representation in feature space is obtained by the application of an appropriate function $\phi, x \rightarrow \phi(x) = \{\phi_i(x) | i \geq 1\}$. The elements $\phi_i(x)$ are called features of x.
A kernel function is a function measuring the similarity of any pair of objects of a domain, $K : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$.
The Gram matrix G^K related to a kernel K with respect to a set S of examples is defined as:

$$G^K_{ij} = K(x_i, x_j) \tag{2.18}$$

Given two examples x_i and x_j, the relationship between the distance $d(x_i, x_j)$ in the feature space and the kernel $K(x_i, x_j)$ is

$$d(x_i, x_j) = ||\phi(x_i) - \phi(x_j)|| = \sqrt{K(x_i, x_i) + K(x_j, x_j) - 2K(x_i, x_j)} \tag{2.19}$$

When two examples are mostly dissimilar, the application of a kernel K to them returns 0. When the kernel is normalized the maximum value of K is 1.
Here follows very cool properties of kernel functions. Let $K_1, K_2 : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ be two kernel functions, $X = \{x_1, \ldots, x_n\}$ a set of examples from the domain $\mathcal{X}$ then:

1. $K(x, x') = K_1(x, x') + K_2(x, x')$ is a valid kernel (additive property);

2. $K(x, x') = K_1(x, x') \cdot K_2(x, x')$ is a valid kernel (multiplicative property);

3. $K(x, x') = f(x)f(x')$, where f is any function defined on the domain $\mathcal{X}$, is a valid kernel;

4. $K(x, x') = K_4(\phi(x), \phi(x'))$ is a valid kernel;

5. $K(x, x') = K_1 \oplus K_3((x, u), (x', u))$, where $K_3 : U \times U \to \mathbb{R}$ is a valid kernel defined on domain U, is a valid kernel;

6. $K(x, x') = K_1 \otimes K_3((x, u), (x', u))$, where $K_3 : U \times U \to \mathbb{R}$ is a valid kernel defined on domain U, is a valid kernel.

Here we show how to normalize a kernel function:

$$K'(x, x') = \frac{K(x, x')}{\sqrt{K(x, x)K(x', x')}} \tag{2.20}$$

In the counting of the number of features the normalization can be useful, because two very tiny trees can be considered as similar as two very big trees, even if their distance calculated with the kernel function is very different.

Standard Kernels

Linear kernel

$$K(x, x') = \langle x, x' \rangle \tag{2.21}$$

The feature space is the same as the input space.

Polynomial kernel

$$K(x, x') = (\langle x, x' \rangle + c)^d, \ c \in \mathbb{R}, d \in \mathbb{N} \tag{2.22}$$

The feature space associated with the polynomial kernel is composed by products of elements of the original vectors. This operation allows to create new features as combinations of the original ones.

Gaussian kernel

$$K(x, x') = \exp\left(-\frac{||x - x'||^2}{2\sigma^2}\right), \ \sigma \in \mathbb{R} \tag{2.23}$$

The use of a kernel function is the only way for computing the corresponding dot product in feature space. The kernel value is maximum when $x = x', K(x, x') = 1$, and it is monotonic decreasing when the distance between x and x' increases. σ affects the resulting feature space as follows:

- $\sigma \to \infty$: all examples are identical;

- $\sigma \to 0$: all examples are orthogonal;

Later on we will see the kernel used for tree structured data.

Kernel Functions Evaluation

What is a good kernel function? Being valid is a necessary but not sufficient requirement for a kernel function to be good.

Sparsity index

$$Sparsity(K, S) = \frac{|\{(i,j) \in S | K(i,j) = 0\}|}{|S|^2} \tag{2.24}$$

Kernel Alignment

$$A(K_1, K_2, S) = \frac{\langle G^{K_1}, G^{K_2} \rangle_F}{\sqrt{\langle G^{K_1}, G^{K_1} \rangle_F \langle G^{K_2}, G^{K_2} \rangle_F}} \tag{2.25}$$

where $\langle G^{K_1}, G^{K_2} \rangle_F = \sum_{i,j} G_{ij}^{K_1} G_{ij}^{K_2}$. Note how similar the formula is to a normalized dot product, indeed $A \in [-1, 1]$. The value A can be used to measure how appropriate a kernel K is for a given two-class classification task by aligning K with a matrix Y defined as: $Y_{ij} = y_i y_j$ (it kinda measure the accuracy of the kernel).

Completeness, Correctness and Appropriateness Let $c : \mathcal{X} \to \Omega$ be a function that assigns to every example of a domain its class. Functions c are grouped into a concept class C.

- **Complete**: a kernel K is complete if: $\forall c \in C,\ K(x_i, \cdot) = K(x_j, \cdot) \Rightarrow c(x_i) = c(x_j)$ (kinda injectivity, though using c instead of identity, which is good enough for the purpose of classification);
- **Correct**: a kernel K is said to be correct with respect to a concept class C and an hypothesis space H, if for every concept can be found an hypothesis that correctly classifies all examples;
- **Appropriateness**: a kernel K is appropriate for learning concepts in a given concept class by a learning algorithm if polynomial bounds on its generalization error can be derived for some algorithms using this kernel (basically, conservation of the topology of the data).

A complete and correct kernel separates the concept well (it is able to achive high accuracy on the given data). An appropriate kernel is able to generalize well to unseen data.

Finally, in practical applications, the right trade-off between expressive power and computational complexity should be selected according to the current task.

3 State of the Art on Tree Kernel Functions

Kernel methods make use of kernel functions to measure the similarity of the items in feature space. Kernel functions are the only type of information specific to the task that this class of learning algoritms may use.
The methodologies for designing kernels for trees include:

- **Convolutional kernel framework**: a convolutional kernel measures the similarity of two objects in terms of the similarities of their subparts;
- **Extraction of features**: builds a vectorial representation of the data, and then applies kernels for vectors;
- **Generative models' kernels**: these kernels measure the similarity of two items as a function of the parameters and states reached by a generative model for the data.

3.1 Convolution Kernels

Convolution kernels express a kernel on a discrete object by a sum of kernels of their constituent pars.

Def. 3.1 (Convolution Kernel) *Let $\mathcal{X}, \mathcal{X}_1, \ldots, \mathcal{X}_D$ be $D+1$ non empty separable metric spaces, $x \in \mathcal{X}$ a structure and $\vec{x} = (x^1, x^2, \ldots, x^D)$ the parts of x. A relation $R : \mathcal{X}_1 \times \cdots \times \mathcal{X}_D \times \mathcal{X}$ where $R(\vec{x}, x)$ is true if and only if $x^1, \ldots, x^D$ are the parts of x. Moreover, let $R^*(x)$ be the set of all the subparts of x.*

Then the convolution kernel can be expressed as:

$$k(x_i, x_j) = \sum_{\vec{x}_i \in R^*(x_i)} \sum_{\vec{x}_j \in R^*(x_j)} \prod_{d=1}^{D} k_d(x_i^d, x_j^d) \tag{3.1}$$

where the k_d are kernels on the substructres (and are called local kernels).

Def. 3.2 (Mapping Kernel) *Let each $x \in \mathcal{X}$ be associated with a finite subset $\mathcal{X}'_x$, where $\mathcal{X}'_x$ is the set of substructures associated with x. Let $k : \mathcal{X}'_x \times \mathcal{X}'_x \to \mathbb{R}$ be a kernel. Then the mapping kernel is defined as:*

$$K(x_i, x_j) = \sum_{x_i;, x'_j \in \mathcal{M}_{x_i, x_j}} k(x'_i, x'_j) \tag{3.2}$$

where $\mathcal{M}$ is part of a mapping system $\mathbb{M}$ defined as follow:

$$\mathbb{M} = \Big(\mathcal{X}, \{\mathcal{X}'_x | x \in \mathcal{X}\}, \{\mathcal{M}_{x_i, x_j} \subseteq \mathcal{X}'_{x_i} \times \mathcal{X}'_{x_j} | (x_i, x_j) \in \mathcal{X} \times \mathcal{X}\}\Big) \tag{3.3}$$

$\mathbb{M}$ is a triplet composed by the domain of the examples, the space of the substructes of the examples, and a function $\mathcal{M}$ specifying for which pairs of substructures the local kernel has to be computed. $\mathcal{M}$ is assumed to be finite and symmetric. The kernel K is positive semidefinite if and only if $\mathcal{M}$ is transitive. A mapping system is transitive if and only if $\forall x_1, x_2, x_3 \in \mathcal{X}.\ (x'_1, x'_2) \in \mathcal{M}_{x_1, x_2} \wedge (x'_2, x'_3) \in \mathcal{M}_{x_2, x_3} \Rightarrow (x'_1, x'_3) \in \mathcal{M}_{x_1, x_3}$.

The following sections are devoted to review convolution kernels for tree structured data.

3.2 Convolutional Tree Kernels

Subtree Kernel

The features of the kernel when applied to trees are the proper subtrees of the input tree. The kernel returns a weighted sum of the number of common proper subtrees. The kernel for strings is defined as follows:

$$K(x_i, x_j) = \sum_{s \in x_i} \sum_{u \in x_j} [\![s = u]\!] w_s = \sum_{s \in \mathcal{A}^*} h_s(x) h_s(y) w_s \tag{3.4}$$

where s, u are substring of the strings x_i, x_j, w_s is the weight associated to the substring s, $\mathcal{A}$ is the set of non empty strings of the alphabet $\mathcal{A}$, $h_s(x)$ counts the frequency of the substring s in x, and $[\![condition]\!]$ is a function returning 1 whether $condition$ is true, i.e. $s = u$, 0 otherwise.
It is pointe out that not all subset trees can be generated if the trees are represented as strings (e.g. $[a[b][g]]$ is not a substring of $[a[b[c][e]][g]]$).

Subset Tree Kernel

This kernel solve the problem of the Subtree Kernel by considering all the subtrees of the input trees.
Let's consider a finite set of trees in which m different subset trees are present. Each subset tree can be indexed by an integer between 1 and m. Then $h_s(T)$ is the number of times the subset tree indexed with s occurs in tree T. We represent each tree T as a feature vector $\phi(T) = [h_1(T), \ldots, h_m(T)]$. We define the innder product between two trees under the reported representation as:

$$K(T_1, T_2) = \phi(T_1) \cdot \phi(T_2) = \sum_{s=1}^{m} h_s(T_1) h_s(T_2) \tag{3.5}$$

Thus the subset tree kernel defines a similarity measure between trees which is proportional to the number of shared subset trees. Since in the thesis it is explained how to reach the last formula I will not report it here, I will just report the final formula for the kernel:

$$K(T_1, T_2) = \sum_{t_1 \in N_{T_1}} \sum_{t_2 \in N_{T_2}} C(t_1, t_2) \tag{3.6}$$

where N_{T_1} is the set of all the subset trees of T_1. Let a *production at* node t be the subset tree constituted by t and only its direct children. Then C is defined as follow:

- if the produciton at t_1 and t_2 are different then $C(t_1, t_2) = 0$;
- if the productions at t_1 and t_2 are the same and they have only leaf children then:

$$C(t_1, t_2) = \lambda \tag{3.7}$$

- if the productions at t_1 and t_2 are the same (and previous condition not satisfied) then:

$$C(t_1, t_2) = \prod_{j=1}^{nc(t_1)} (\lambda + C(ch_j[t_1], ch_j[t_2])) \tag{3.8}$$

where $nc(t)$ is the number of children of t and $ch_j[t]$ is the j-th child of t (note that $nc(t_1) = nc(t_2)$, if their productions are the same).

Where $0 < \lambda \leq 1$ is a weight parameter: since subset tree kernel values depend on the number of nodes in the trees, λ is used to regularize the kernel.

The worst case time computational complexity of the kernel is $O(|N_{T_1}| \times |N_{T_2}|)$. Which makes the algorithm infeasible for large trees. Another problem, is that this kernel can't be used with continuous labels, because the resultant feature space would be too sparse.

The next sections will either reduce the computational complexity or they will add expressiveness to the kernel.

Approximate Kernels for Trees

In this section we describe an approximated tree kernel with worst case linear time complexity. The speed up is obtained by selecting a restricted set of sparse and discriminative features:

$$K(x_i, x_j) = \sum_s \gamma(s) \sum_{t_i \in x_i} \sum_{t_j \in x_j} C(t_i, t_j) \tag{3.9}$$

where t_i is a node of x_i and $\gamma(s) = \{0, 1\}$ is a function telling whether the current subtree has to be considered. The goodness of a kernel can be measured by its alignment to the matrix yy^T (see Equation 2.25):

$$\langle yy^T, K \rangle_{\mathcal{F}} = \sum_{y_i = y_j} K(x_i, x_j) - \sum_{y_i \neq y_j} K(x_i, x_j)$$

So the final problem to be solved can be formulated as follows:

$$max_{\gamma \in \{0,1\}^{|S|}} \sum_{i,j=1}^{n} \sum_{s \in S} \gamma(s) \sum_{t_i \in x_i} \sum_{t_j \in x_j} C(t_i, t_j),$$
$$\text{s.t.} \sum_{s \in S} \gamma(s) = N$$

where S is the feature space, n is the size of the training set and N is the number of different substructures we want to consider.

If this isn't enough, you can bound the expected runtime to a variable b:

$$\sum_{s \in S} \gamma(s)\sigma(s) \leq b$$

where $\sigma(s)$ is the number of times the substructure s appears in the training set.

Partial Tree Kernel

One way for enlarging the feature space consists on counting the partial matching between subtrees (the definition of partial tree correponds to subtree's definition). The partial tree kernel can be evaluated in $O(\rho^3 |N_{T_1}||N_{T_2}|)$, where ρ is the maximum out-degree of the two trees. To see how the kernel is defined, please refer to the thesis.

Elastic Tree Kernel

An elastic tree is a subset of nodes for which the relative positions in the original tree are preserved. Let two trees T_1 and T_2 and two subtrees, t_1 and t_2, rooted at $v_1 \in T_1$ and $v_2 \in T_2$, respectively. This kernel extends the subset tree kernel section 3.2 in the following way:

- t_1 and t_2 may match even if they don't have the same number of children, the only contraint is to preserve left-to-right order of the children, therefore the C function becomes:

$$C(v_1, v_2) = S_{v_1,v_2}(nc(v_1), nc(v_2)) \tag{3.10}$$

and S can be defined as:

$$\begin{aligned} S_{v_1,v_2}(i,j) =& S_{v_1,v_2}(i-1,j) \\ &+ S_{v_1,v_2}(i,j-1) \\ &- S_{v_1,v_2}(i-1,j-1) \\ &+ S_{v_1,v_2}(i-1,j-1)C(ch_i[v_1], ch_j[v_2]) \end{aligned}$$

and $S_{v_1,v_2}(i,0) = S_{v_1,v_2}(0,j) = 1$. Basically the idea is that the number of matchings considering l children can be expressed in terms of the number of matchings considering $l-1$ children, indeed the formulation can be seen as a dynamic programming algorithm;

- t_1 and t_2 may match even if their labels are not identical. This is obtained by multiplying each match by a value determined by a function $P_{mut}(l_1, l_2) \in [0, 1]$, where $P_{mut}(l_1, l_2)$ is the cost for transforming label l_1 into label l_2. The C function then becomes:

$$C(v_1, v_2) = \sum_{a \in \mathcal{A}} P_{mut}(l_1, a) P_{mut}(l_2, a) S_{v_1,v_2}(nc(v_1), nc(v_2)) \tag{3.11}$$

where $\mathcal{A}$ is the space of labels. Thus it takes into account all possible mutations of the labels of the nodes being computing the C value. No actual P_{mut} function is provided neither in the thesis or in its references;

- t_1 and t_2 can be elastic trees. Since all descendants of a node can be part of an elasti subtree, all of them have to be considered. This leads to C being defined as:

$$C(v_1, v_2) = \sum_{v_a \in D(v_1)} \sum_{v_b \in D(v_2)} C(v_a, v_b) \tag{3.12}$$

where $D(v)$ is the set of descendants of v, including v itself. Note, that the dynamic programming algorithm already stores the values for the descendants of a node, so the computation of the C value is not problematic.

Semantic Syntactic Tree Kernels

This kernel is designed for text categorization tasks. The kernel introduces two ideas:

- embedded semantic term kernel and a leaf weighting component;

- partial matches between tree fragments, where a partial match between two subtrees occurs when they differ only by their terminal symbols.

The tree fragment kernel is defined as:

$$K(f_1, f_2) = comp(f_1, f_2) \prod_{i=1}^{nt(f_1)} k_S(f_1(i), f_2(i)) \tag{3.13}$$

where the function $comp(f_1, f_2)$ equals 1 whether the f_1 and f_2 differs only in the terminal nodes, 0 otherwise; $nt(f_1)$ is the number of terminal nodes of the two tree fragments. The semantic syntactic tree kernel is obrained by modifying Equation 3.7 as follows:

$$C(v_1, v_2) = \lambda k_S(v_1, v_2) \tag{3.14}$$

Note that k_s might be:

- taxonomy for computing the term similarity (I don't know what it is);
- latent semantic indexing, which computes the similarity by means of co-occurence analysis of terms in documents and vice versa (which neither I know about, but I have some intuition about and I like it!).

3.3 Non Convolutional Kernels

Spectrum Tree Kernel

The spectrum tree kernel counts common tree q-grams. Tree q-grams are subtrees isomorphic to paths with with q nodes.
A subtree P_i matches a tree T at a node n if there exists a one-to-one mapping f from the nodes of P into the nodes of T satysfying the following constraints:

- f maps the root of P to n;
- The ordering o fhte children is preserved by the mapping;
- The labels of the nodes are preserved by the mapping ($\forall x \in P,\ l(x) = l(f(x))$).

Being $G_q(T)$ the vector collecting information about all $q - grams$ in T, the kernle is defined as follows:

$$K(T_1, T_2) = \langle G_q(T_1), G_q(T_2) \rangle \tag{3.15}$$

Tree Fisher Kernel

The Fisher Kernel is derived from a generative model. It uses gradient of the log likelihood of the data with respect to the model parameters. The Fisher kernel assumes that the data is generated from a parametric probability distribution: $P(x|\vec{\theta})$, where $\vec{\theta}$ is the parameter vector. The idea is to form a representation of the data in terms of those parameters which are sufficient to describe x. This is achieved by means of the Fisher Schore U_x:

$$U_x = \nabla_{\vec{\theta}} \log P(x|\vec{\theta}) \tag{3.16}$$

The Fisher kernel is defined as:

$$K(x_i, x_j) = U_{x_i}^T I^{-1} U_{x_j} \tag{3.17}$$

I didn't quite get how I is defined, but it looks like sometimes the identity matrix is used in its place (note that it is called Fisher Information Matrix).
It is worth noting that when deriving a kernel from a generative model, the value of kernel between two objects depends also on the set used to train the generative model.