STRING KERNEL BASED BINARY TEXT SOURCE CLASSIFICATION

Pradeep Bihani

A thesis submitted in partial fulfillment of
the requirements for the degree of
Master of Arts

Department of Mathematics

Central Michigan University
Mount Pleasant, Michigan
November 2020

ACKNOWLEDGEMENTS

First, I would like to express my sincere gratitude to my advisor Prof. Debraj Chakrabarti for continuous support, for his patience, motivation and immense knowledge. I would also like to thank Prof. Jordan Watts and Prof. Sivaram Narayan for being so helpful and supportive mentors. I am fortunate to have them in my committee. I thank Twitter, Inc. for allowing me to use their data for my project.

To my family.

ABSTRACT

STRING KERNEL BASED BINARY TEXT SOURCE CLASSIFICATION

by Pradeep Bihani

We try to understand how simple mathematical principles from linear algebra and optimization theory can be applied to machine learning. As a starting problem, we considered the task of classifying text coming from two sources with different but a priori unknown characteristics (e.g. in different languages). We used standard techniques such as support vector machines and kernelization to try to solve this problem. The kernel trick establishes a relation of this problem with Reproducing Kernel Hilbert Spaces of functional analysis, which is quite surprising. We used the subsequence string kernel for our problem, which matches substrings of length n in two given strings to compute the kernel function. These substrings need not be contiguous, and computing this kernel efficiently is quite difficult because of the huge number of computations involved. To deal with this we use dynamic programming, and to further enhance our computation performance we employ parallelization and concurrent computing techniques. We tested our technique by trying to classify strings arising from twitter accounts of world-leaders, and famous literary texts. The results were in accordance with expectations based on the similarity of the sources.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

CHAPTER I

INTRODUCTION

I.1. Statement of the Problem

In this project, we consider a simple problem of machine learning and try to solve it using an elementary method based on linear algebra.

Though the main concepts and methods of machine learning and artificial intelligence go back decades, the field has seen a lot of interest recently. The main reasons are first, the easy availability of fast and cheap computing power, and second, the availability of huge amounts of data for study and use. This project is mainly exploratory and our goal is to understand some of the very basic ideas behind the field from a mathematical point of view.

The problem we want to consider is as follows. We assume that we are given two sources of text strings S_1 and S_2. We want to build a machine learning model which will learn to distinguish between S_1 and S_2, and will then be able to classify further text strings coming from these two sources. A typical example of such sources is a twitter account or a literary text. We assume that the sources produce a sequence of strings in the same alphabet (which for our practical implementation will be the extended ASCII characters used by modern computers).

It is very important to note that we do not assume any knowledge of the features of the two sources of text, e.g. what is the language of the texts, or what special words it may use. Thus, our problem is different from similar problems where human programmers have prior knowledge of the characteristics of the sources S_1 and S_2. For example, if the sources are in two languages (say S_1 is in Spanish and S_2 is in Hungarian), and this is known to us beforehand, we can build a simple system to detect whether a given string is generated by S_1 or S_2 by simply comparing with dictionaries in Spanish and Hungarian. Of course, the two languages may have some words in common, but a simple count of the number of Spanish and Hungarian words would give a good indication of the language. We emphasize that our goal is different, in that we do not allow ourselves to use any other information except that the strings are generated by S_1 and S_2.

We approach this problem using the *kernel method* and the standard technique of *support vector machines*. This method, originally developed by Vapnik (see [9]), involves the mapping of the training data (i.e. in our case a collection of strings, with their source as S_1 or S_2) into a high dimensional Euclidean space and constructing a separating hyperplane that divides the two clouds of points corresponding to the two text sources. We test our method to classify several pairs of sources. Depending on the similarity of the sources, we are able to separate them to different extents. We try to understand what characteristics of the sources lead to efficient classification.

This project is different from a typical mathematics research project, in that we do not prove any theorems. Rather, the mathematics plays a crucial role in the design of the algorithm used, and our main outcome is the program itself and its performance. The code written in the Python language can be found in the Appendix to this thesis, and in some ways represents the bulk of our efforts. In this thesis, we try to explain the theoretical background of our work, the practical details of its implementation, and the results obtained in testing the system with real-world data.

I.2. Support Vector Machine (SVM)

In this section we will describe the *support vector machine*, a standard scheme for binary classification of points in $\mathbb{R}^n$ or more generally in a real Hilbert space $\mathcal{H}$ (the vector spaces of this section are finite dimensional and real). The problem can be stated as follows: we are given two collections of points, let us say labeled by $+1$ and -1. This is the *training data*, and each collection will be referred to as a *cloud* (so there is a +1 cloud and a -1 cloud). We want to build a binary classification machine that will learn to distinguish between the two clouds, and when presented with a new point in $\mathcal{H}$, will be able to assign it to the correct cloud. We want to achieve this goal in the simplest way possible: we will try to construct a hyperplane in $\mathcal{H}$ such that the two clouds are on opposite sides of this hyperplane. This hyperplane is therefore called a *separating hyperplane* for the problem. Notice that in general, it may not be possible to find a hyperplane that separates the two clouds, so in practice we will content ourselves with a hyperplane which separates *most*

of the training data. Notice that this already introduces some error into the classification, since now when presented with some of the training data, the classification does not work as expected.

The given data for the problem can the thought of as a subset $\mathscr{X} \subset \mathcal{H} \times \{\pm 1\}$. Notice that each point of $\mathscr{X}$ is of the form $(x, \pm 1)$, where $x \in \mathcal{H}$. Points of the form $(x, 1)$ belong to the +1 cloud, and points of the form $(x, -1)$ belong to the -1 cloud. We assume that both clouds are nonempty, since otherwise training is impossible.

Recall that each hyperplane in $\mathcal{H}$ is of the form

$$\{x \in \mathcal{H} | \langle w, x \rangle + b = 0\} \tag{I.1}$$

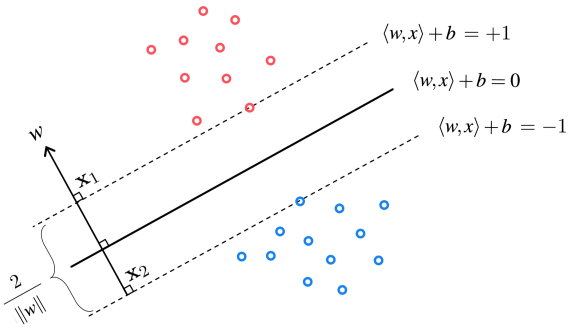
for some $w \in \mathcal{H} \setminus \{0\}, b \in \mathbb{R}$. Clearly w is a vector orthogonal to the hyperplane and the distance of the hyperplane from the origin is $\frac{|b|}{\|w\|}$. Since $\langle w, x \rangle$ is the length of x along the direction of w, geometrically, the hyperplane (I.1) consists of vectors that all have the same length along w.

Let X be the projection of the training data $\mathscr{X}$ onto $\mathcal{H}$, i.e., X is the collection of training points without the labels ± 1. In order to find an optimal representationn of the separating hyperplane, we introduce the following definition:

Definition I.1 (Canonical Hyperplane)**.** The pair $(w, b) \in \mathcal{H} \times \mathbb{R}$ is called the *canonical* form of the hyperplane (I.1) with respect to $X \subset \mathcal{H}$ if it is scaled such that

$$\min_{x \in X} |\langle w, x \rangle + b| = 1 \tag{I.2}$$

Geometrically this means that the points of X which are closest to the canonical hyperplane have a distance of $\frac{1}{\|w\|}$ from it. In the figure below, the points $\mathbf{x}_1$ and $\mathbf{x}_2$ of the training data are closest to the separating hyperplane.



It is clear that the canonically represented hyperplane (I.2) will produce the best results if the distance $\dfrac{1}{\|w\|}$ of the nearest point of X to it is the largest. This will ensure so-called *maximal margin* i.e., the largest distance between the clouds and the hyperplane. Therefore we seek to construct a separating hyperplane in canonical form (I.2) so that *the value of* $\|w\|$ *is the smallest possible.*

It is natural to introduce the *decision function* $f_{w,b} : \mathcal{H} \to \{\pm 1\}$

$$f_{w,b}(x) = \operatorname{sgn}(\langle w,x\rangle + b) \tag{I.3}$$

which takes values $+1$ and -1 on opposite sides of the hyperplane. Our aim is to find the decision function (I.3) $f_{w,b}(x) = \operatorname{sgn}\left(\langle w,x\rangle + b\right)$ satisfying the constraint

$$f_{w,b}(x) = y \quad \text{for all } (x,y) \in \mathscr{X}, \tag{I.4}$$

i.e., f assumes the values ± 1 on the respective clouds. If such a function exists, then canonicality (I.2) implies

$$y(\langle w,x\rangle + b) \geq 1 \text{ for } (x,y) \in \mathscr{X} \subset \mathcal{H} \times \{\pm 1\}. \tag{I.5}$$

Notice that out of two canonical forms of the same hyperplane given by parameters (w,b) and $(-w,-b)$, only one will satisfy the equations (I.4) and (I.5).

Therefore, the problem of finding the maximal margin separating hyperplane is equivalent to minimizing $\|w\|$ under the constraint (I.5), i.e. we want to solve the following optimization

problem:

$$\begin{gathered}\underset{w\in\mathcal{H},b\in\mathbb{R}}{\text{minimize}}\ \frac{1}{2}\|w\|^2 \\ \text{subject to } y(\langle w,x\rangle+b))\geq 1 \text{ for } (x,y)\in\mathscr{X}\end{gathered} \tag{I.6}$$

A standard technique in optimization theory to solve such constrained optimization problems is to convert the given problem (called a *primal* problem) to a new problem (called the *dual* of the given problem), [7, Chapter 6]. The first step in formulating the dual problem is to introduce the *Lagrangian*

$$\mathcal{L}(w,b,\alpha)=\frac{1}{2}\|w\|^2-\sum_{(x,y)\in\mathscr{X}}\alpha_x(y(\langle w,x\rangle+b)-1),$$

where $\alpha_x\geq 0$ are the *Lagrange* multipliers. It is not difficult to see (see, e.g. [7, Theorem 6.21]) that a solution of the unconstrained optimization problem

$$\min_{w,b}\max_{\alpha}\ \mathcal{L}(w,b,\alpha) \tag{I.7}$$

is also a solution of (I.6), and the optimal point $(\overline{w},\overline{b},\overline{\alpha})$ satisfies the Karush-Kuhn-Tucker (KKT) conditions [7, Theorem 6.21]:

$$\overline{\alpha_x}[y(\langle x,\overline{w}\rangle+\overline{b})-1]=0,\ \text{ for } (x,y)\in\mathscr{X} \tag{I.8}$$

A duality principle allows us to convert (I.7) to an associated maximization problem. The solution to the dual problem provides a lower bound to the solution of the primal (minimization) problem. In our case, we are trying to solve a convex optimization problem, satisfying Slater's condition so Slater's theorem [8] tells us that strong duality holds (i.e solution of the primal and the dual problem are equal). To formulate the dual problem, we first find the minimum of (I.7) with respect to w and b, while α is thought of as a vector-valued parameter:

$$\frac{\partial\mathcal{L}}{\partial b}(w,b,\alpha)=0,\ \frac{\partial\mathcal{L}}{\partial w}(w,b,\alpha)=0$$

which lead to

$$\sum_{(x,y)\in\mathscr{X}} \alpha_x y = 0 \tag{I.9}$$

$$w = \sum_{(x,y)\in\mathscr{X}} \alpha_x y x, \tag{I.10}$$

Substituting (I.9) and (I.10) in (I.7), we get

$$\mathcal{L}(\alpha) = \sum_{(x,y)\in\mathscr{X}} \alpha_x - \frac{1}{2} \sum_{(x,y),(\xi,\eta)\in\mathscr{X}} \alpha_x \alpha_\xi y \eta \left\langle x, \xi \right\rangle . \tag{I.11}$$

This $\mathcal{L}(\alpha)$ is the *Wolfe dual Lagrangian function*. The optimization problem (I.7) is therefore equivalent to the *Wolfe dual problem*:

$$\max_{\alpha} \mathcal{L}(\alpha)$$

$$\text{subject to } \alpha_x \geq 0 \text{ and } \sum_{(x,y)\in\mathscr{X}} \alpha_x y = 0 \text{ for } (x,y) \in \mathscr{X}$$

From (I.8), we see that, for solutions of the above problem, non-zero values $\alpha_x > 0$ are only achieved in the cases where

$$y(\langle x, w\rangle + b) = 1 \text{ for } (x,y) \in \mathscr{X} \tag{I.12}$$

Those vectors x which satisfy above equality are called *Support Vectors*, and we denote the set of support vectors by $SV \subset X$. Hence, if $x \notin SV$, we have $\alpha_x = 0$. Now, substituting the value of w found in (I.10) in the decision function (I.3) we get

$$f(x) = \text{sgn} \left(\sum_{(\xi,\eta)\in\mathscr{X}} \alpha_\xi \eta \left\langle x, \xi \right\rangle + b \right), \tag{I.13}$$

but by the above observation about nonzero α_x's this becomes:

$$f(x) = \text{sgn} \left(\sum_{(\xi,\eta)\in SV} \alpha_\xi \eta \left\langle x, \xi \right\rangle + b \right), \tag{I.14}$$

where $SV \subset \mathscr{X}$ corresponds to the support vectors.

I.2.1. Soft Margin Hyperplane

In practice, it may not be possible to find a hyperplane that separates the two clouds of training data. Therefore, we introduce so-called *slack variables* which allows us to obtain an approximate separating hyperplane which may have some points on the "wrong side".

We introduce non-negative "slack" variables $\zeta_x \geq 0$, for $x \in X$, so that the error can be given by,

$$\phi(\zeta) = \sum_{x \in X} \zeta_x.$$

To determine the "soft margin hyperplane" we incorporate this error as a penalty into our optimization function and as a result obtain the following optimization problem generalizing the maximal margin problem described in the previous section, where C is a constant which determines the amount of penalty:

$$\underset{w \in \mathcal{H}, b \in \mathbb{R}}{\text{minimize}} \ \frac{1}{2}\|w\|^2 + C \sum_{x \in X} \zeta_x$$

subject to constraints

$$y(\langle x, w \rangle + b) \geq 1 - \zeta_x \text{ for } (x, y) \in \mathscr{X}$$

$$\zeta_x \geq 0$$

The Lagrangian associated to this problem is

$$\mathcal{L}(w, \zeta, b, \alpha, \gamma) = \frac{1}{2}\|w\|^2 + C \sum_{x \in X} \zeta_x - \sum_{(x,y) \in \mathscr{X}} \alpha_x (y(\langle w, x \rangle + b) - 1 + \zeta_x) - \sum_{x \in X} \gamma_x \zeta_x$$

where α and γ are (vector-valued) Lagrange multipliers. As shown by Vapnik ([9]), using the same technique as for the separable case above, we find that we need to maximize the same Wolfe dual (I.11) as before, under a slightly different constraint:

$$\underset{\alpha}{\text{maximize}} \ \mathcal{L}(\alpha) \tag{I.15}$$

$$\text{subject to } 0 \leq \alpha_x \leq C, \text{ for } x \in X$$

$$\sum_{(x,y)\in\mathscr{X}} \alpha_x y = 0$$

Quadratic programming

In order to complete the construction of the separating hyperplane we still need to solve the optimization problem (I.15). There are standard algorithms for this, the most popular of which is the *interior point method* (see [7, section 6.4]). There are standard Python packages such as `CVCOPT` available implementing this algorithm.

I.3. Kernel Trick

The SVM method described in the previous section assumes that the data is a subset of the linear space $\mathcal{H}$. Only then can we talk about hyperplanes separating the clouds. In our problem, where we deal with classification of strings, the strings are not naturally points of a vector space. The strings form a set S with no linear structure, though there are other algebraic operations on the strings, such as concatenation and extracting substrings.

There is a standard technique to deal with this situation. We use a map

$$\phi : S \to \mathcal{H}$$

to map the data into a Hilbert space. The map ϕ is chosen so that it preserves to some extent the structure of S, i.e., strings which are similar are mapped to nearby points of the Hilbert space $\mathcal{H}$. Such a map is called a *feature map* and the target space $\mathcal{H}$ is called a *feature space*.

With the application of the feature map, the problem is reduced to the one considered in the previous section, and the SVM method can be used.

However, there is a serious computational downside to this process, since the feature space $\mathcal{H}$ usually has very large dimension. In our problem, we will have to use a feature space of ap-

proximately 10^{12} dimensions. It is practically impossible for computers to perform computations with vectors of 10^{12} entries or matrices of 10^{24} entries.

However, at this point, there is an interesting observation which allows us to do the computations needed for SVM's without using very high dimensional spaces. Introduce the function

$$k : S \times S \to \mathbb{R}$$

by the definition

$$k(s,t) = \langle \phi(s), \phi(t) \rangle , \tag{I.16}$$

where the inner product is taken in the space $\mathcal{H}$. We claim that the geometry of the clouds as a subset of $\mathcal{H}$ is completely determined by the function k. Indeed, the geometry of the cloud is completely known as soon as we know the distance between any pair of points. But we have

$$\begin{aligned} \|\phi(s) - \phi(t)\|^2 &= \langle \phi(s) - \phi(t), \phi(s) - \phi(t) \rangle \\ &= \langle \phi(s), \phi(s) \rangle - 2 \langle \phi(s), \phi(t) \rangle + \langle \phi(t), \phi(t) \rangle \\ &= k(s,s) - 2k(s,t) + k(t,t). \end{aligned}$$

It follows that one can perform the construction of the separating hyperplane (i.e. determine the parameters w, b) using only the function k, and without performing any computations in $\mathcal{H}$. If computing k turns out to be a reasonable task, we have an example of what is known as the *kernel trick* of machine learning.

The function k is called a *reproducing kernel*, and these are of great importance in functional analysis and operator theory. In section I.5 below we explain this surprising link.

Once the separating hyperplane is found, i.e. the parameters $w \in \mathcal{H}$ and $b \in \mathbb{R}$ are known in (I.1), we can rewrite the decision function (I.14) in terms of elements of S alone, by composing with the feature map. Let $T \subset S$ be the set of those strings which are mapped to support vectors

by ϕ, then we have

$$g(s) = f(\phi(s)) \tag{I.17}$$

$$= \operatorname{sgn}\left(\sum_{(\xi,\eta)\in SV} \alpha_\xi \eta \langle \phi(s), \xi \rangle + b\right) \tag{I.18}$$

$$= \operatorname{sgn}\left(\sum_{t\in T} \alpha_{\phi(t)} \eta(t) \langle \phi(s), \phi(t) \rangle + b\right) \tag{I.19}$$

$$= \operatorname{sgn}\left(\sum_{t\in T} \alpha_{\phi(t)} \eta(t) k(s,t) + b\right), \tag{I.20}$$

where $\eta(t) \in \{\pm 1\}$ denotes the label associated to the string t. Therefore, the classification of a new string s into the two classes $\{\pm 1\}$ can be performed using the kernel k alone, and no computations in $\mathcal{H}$ are required.

Notice that the function g gives us our method for deciding to which class a new string should be assigned. The parameters defining the function g will be called the *SVM model* of the classification problem. Therefore, an SVM model consists of

1. The set of strings T mapped to support vectors by ϕ. By abuse of language, we call the strings in T *support vectors* as well.
2. for each $t \in T$, its label $\eta(t) \in \{\pm 1\}$.
3. also, for each $t \in T$, the corresponding Lagrange multiplier $\alpha_{\phi(t)}$.
4. the intercept $b \in \mathbb{R}$.
5. the kernel function $k(\cdot,\cdot)$.

I.4. String Subsequence Kernel

As explained in the previous section, we are going to use the kernel trick to apply an SVM model for binary classification of the strings. We will use a kernel introduced by Lodhi-Saunders-Shawe-Taylor-Cristianini-Watkins in [4], called the *string subsequence kernel*. The important feature of a kernel that leads to effective separation is that it should map "similar" data (in this case

strings) to nearby points in the feature space. Experience has shown that the string subsequence kernel has this property.

Let Σ be a finite alphabet. A *string* is a finite sequence of characters from Σ, including the empty sequence. We denote by Σ^n the set of all strings of length n, and by Σ^* the set of all strings. For a string $s \in \Sigma^m$, we denote by s_j the j-th character of s, so that $s = s_1 s_2 \dots s_m$. We will first define a feature map ϕ from Σ^* to a feature space. The map ϕ will depend on two parameters: a positive integer n and a number $\lambda \in (0,1)$. Having chosen these parameters, the feature space will be $\mathbb{R}^{\Sigma^n}$ with the standard inner product, i.e., $\mathbb{R}^{\Sigma^n}$ is the collection of real-valued functions on Σ^n with pointwise addition. Notice that the feature space has dimension $|\Sigma|^n$. The map

$$\phi : \Sigma^* \to \mathbb{R}^{\Sigma^n}$$

can be given by specifying for each $u \in \Sigma^n$ the component ϕ_u, which is defined to be

$$\phi_u(s) = \sum_{\substack{i \in \mathbb{N}^n \\ s[i]=u}} \lambda^{l(i)}, \tag{I.21}$$

where the sum is taken over all multi-indices $i = (i_1, \cdots, i_n) \in \mathbb{N}^n$, and for such an i, the notation $s[i]$ denotes the substring $u_1 \dots u_n$ of length n in s, where $u_j = s_{i_j}$, and $l(i) = i_n - i_1 + 1$ is the *length* of the multi-index i. In other words, for each such multi-index i, we see if the substring $s[i]$ of s determined by i is equal to u, and if yes, we add $\lambda^{l(i)}$ to the sum. The feature ϕ_u measures the number of occurrences of the substring u in the string s weighting different occurrences of u as a substring according to their lengths. Hence, the full feature map $\phi : \Sigma^* \to \mathbb{R}^{\Sigma^n}$ compares strings $s \in \Sigma^*$ by means of the subsequences of length n they contain, the more subsequences in common, the greater the similarity but rather than just weighting all occurrences equally, the degree of contiguity of the subsequence in the input string s determines how much it will contribute to the comparison. As a result, "similar" strings (which contain the same subsequences of length n many times) are mapped to nearby points of $\mathbb{R}^{\Sigma^n}$. This feature map gives less weight to the non-contiguous words by using the decay factor $\lambda \in (0,1)$.

For example: the string "con" occurs as a subsequence of the strings "cone","counter" and "complement", but we consider the first occurrence as more important since it is contiguous, while the final occurrence is the weakest of all three.

Therefore, in the corresponding kernel (I.16), the inner product of the feature vectors of two strings, gives a sum over all common subsequences weighted according to their frequency of occurrence and lengths:

$$\begin{aligned} K(s,t) &= \sum_{u\in\Sigma^n} \langle \phi_u(s), \phi_u(t) \rangle \\ &= \sum_{u\in\Sigma^n} \sum_{\substack{i\in\mathbb{N}^n \\ s[i]=u}} \lambda^{l(i)} \sum_{\substack{j\in\mathbb{N}^n \\ t[j]=u}} \lambda^{l(j)} \\ &= \sum_{u\in\Sigma^n} \sum_{\substack{i\in\mathbb{N}^n \\ s[i]=u}} \sum_{\substack{j\in\mathbb{N}^n \\ t[j]=u}} \lambda^{l(i)+l(j)}. \end{aligned} \tag{I.22}$$

We call $K(s,t)$ the *subsequence string kernel* with parameters n and λ.

Example

Let $\Sigma = \{a,b,c,r,t\}$. Consider - as simple documents - the words cat, car, bat, bar. If we consider only $n = 2$, we obtain a feature space of dimension $|\Sigma|^n = 5^2 = 25$. However, this reduces to an 8-dimensional feature space, because 17 elements of Σ^2 never occur in the documents being studied. The words are mapped as follows:

Table 1. SSK Example

u =	c-a	c-t	a-t	b-a	b-t	c-r	a-r	b-r
$\phi_u(cat)$	λ^2	λ^3	λ^2	0	0	0	0	0
$\phi_u(car)$	λ^2	0	0	0	0	λ^3	λ^2	0
$\phi_u(bat)$	0	0	λ^2	λ^2	λ^3	0	0	0
$\phi_u(bar)$	0	0	0	λ^2	0	0	λ^2	λ^3

Hence, $K(car,cat) = \lambda^4$, $K(car,car) = K(cat,cat) = 2\lambda^4 + \lambda^6$, and $K(bat,car) = 0$.

In practice, we will use a modified version of the string kernel. Instead of the feature map ϕ_u of (I.21), we will use the normalized map

$$\overline{\phi}_u(t) = \frac{\phi_u(t)}{\|\phi_u(t)\|}.$$

The advantage of this normalization is that image of the feature map is now contained in the unit sphere $\{\|x\| = 1\}$ of $\mathbb{R}^{\Sigma^n}$, so the length of the string does not influence $\overline{\phi}_u$ as much as it does the original subsequence string kernel ϕ_u. Since ϕ_u maps similar strings to nearby points of $\mathbb{R}^{\Sigma^n}$, the same is true for $\overline{\phi}_u$. Thus we get the *normalized string subsequence kernel*:

$$\begin{aligned} \overline{K}(s,t) &= \left\langle \overline{\phi}(s), \overline{\phi}(t) \right\rangle = \left\langle \frac{\phi(s)}{\|\phi(s)\|}, \frac{\phi(t)}{\|\phi(t)\|} \right\rangle \\ &= \frac{\langle \phi(s), \phi(t) \rangle}{\|\phi(s)\| \, \|\phi(t)\|} = \frac{K(s,t)}{\sqrt{K(s,s)}\sqrt{K(t,t)}} \end{aligned} \tag{I.23}$$

I.5. Reproducing Kernel Hilbert Spaces

The kernels introduced above as inner products of feature maps are special cases of a very interesting construction in functional analysis, and in this section we point out this surprising connection. In this section, we allow our Hilbert spaces to be infinite dimensional and over complex scalars. The case of real scalars is similar and can be easily deduced.

Let E be a set, and let $\mathcal{H}$ be a Hilbert space of complex valued functions on E. We say that $\mathcal{H}$ is a *Reproducing Kernel Hilbert space* (RKHS) if for each point $p \in E$, the map

$$f \mapsto f(p)$$

is a continuous linear functional on $\mathcal{H}$. Then, by the Riesz representation theorem, there is for each $p \in E$, an element $k_p \in \mathcal{H}$ such that $f(p) = \langle f, k_p \rangle$. We call k_p the *reproducing kernel of* $\mathcal{H}$ *at the point* p. The function $K : E \times E \to \mathbb{C}$ given by

$$K(x,y) = \langle k_y, k_x \rangle = k_y(x).$$

is called the *Reproducing kernel* of the space $\mathcal{H}$. A fundamental theorem in the theory of RKHSs is the following:

Theorem 1 (Moore-Aronszajn[1]). *Let E be a set, and $K : E \times E \to \mathbb{C}$ be a function such that*

1. *(Hermitian symmetry) $K(y,x) = \overline{K(x,y)}$, and*
2. *(Positive definiteness) For each n-tuple of points $(x_1, \cdot, x_n)$ with $x_i \in E$ the $n \times n$ Gram matrix*

$$\big(K(x_i, x_j)\big)$$

is positive semidefinite, for each positive integer n.

Then there exists a vector space $\mathcal{H}_K$ of complex-valued functions on E, and an inner product $\langle \cdot, \cdot \rangle$ on $\mathcal{H}_K$ such that $\mathcal{H}_K$ is an RKHS with reproducing kernel equal to K.

Important examples of reproducing kernels in complex analysis include the Szegő kernel (Hardy space) and the Bergman kernel (Bergman space). There are many other examples of infinite dimensional RKHS's coming from other parts of analysis as well.

We now note that the kernels (I.16) arising from the classification problem are also reproducing kernels of certain (finite dimensional and real) Hilbert spaces. Thanks to Theoerem 1, we only need to verify that the conditions 1 and 2 are met. The (real version) of condition (1) is obvious. To see that the positive definiteness holds, let $t = (t_1, \ldots, t_n) \in \mathbb{R}^n$

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{n} t_i k(s_i, s_j) t_j &= \sum_{i=1}^{n}\sum_{j=1}^{n} t_i \left\langle \phi(s_i), \phi(s_j) \right\rangle t_j \\
&= \left\langle \sum_{i=1}^{n} t_i \phi(s_i), \sum_{j=1}^{n} t_j \phi s_j \right\rangle \\
&= \left\| \sum_{i=1}^{n} t_i \phi(s_i) \right\|^2 \geq 0
\end{aligned}$$

which shows that the Gramian $(k(s_i, s_j))$ is positive definite.

For more information on RKHSs and their applications to machine learning, see [6].

I.6. Decision Score

In order to analyze the accuracy of the testing data or in other words how well the testing data got separated by classifying model, we will define the following.

Definition I.2 (Decision Score)**.** We call

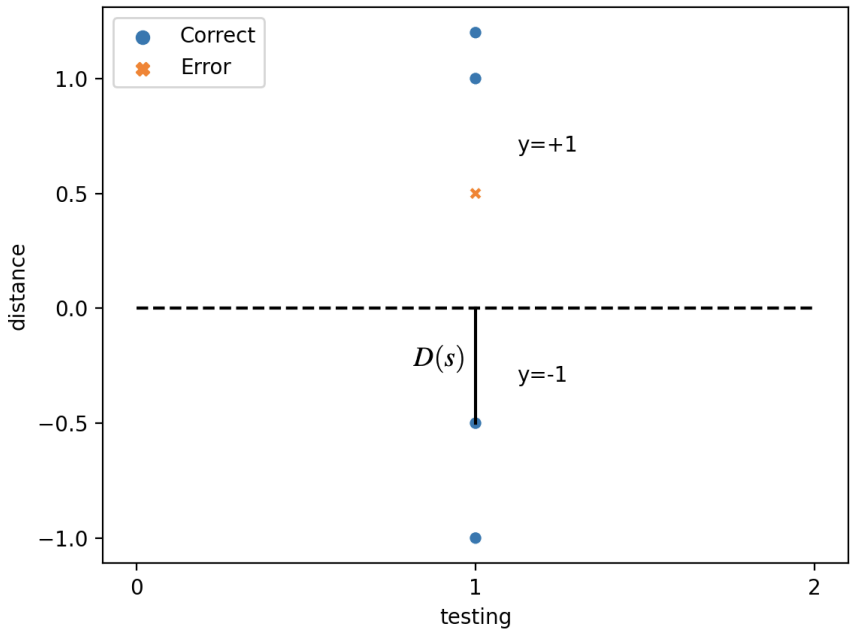
$$d_{(w,b)}(x) := (\langle w, x\rangle + b), \tag{I.24}$$

where $(x,y) \in \mathcal{H} \times \{\pm 1\}$ the *decision score* where (I.1) represents the separating hyperplane of the string classification problem. This is clearly proportional to the signed distance of the point x from the hyperplane.

A useful expression for the decision score can be obtained by substituting w from (I.10). Let $T \subset S$ be the set of those strings which are mapped to support vectors by ϕ, then

$$\begin{aligned} D(s) &= d(\phi(s)) && \text{(I.25)}\\ &= \sum_{(\xi,\eta)\in SV} \alpha_\xi \eta \,\langle \phi(s), \xi\rangle + b && \text{(I.26)}\\ &= \sum_{t\in T} \alpha_{\phi(t)} \eta(t) \,\langle \phi(s), \phi(t)\rangle + b && \text{(I.27)}\\ &= \sum_{t\in T} \alpha_{\phi(t)} \eta(t) k(s,t) + b, && \text{(I.28)} \end{aligned}$$

where $\eta(t) \in \{\pm 1\}$ denotes the label associated to the string t. A higher numerical value of $D(s)$ (negative or positive) means the string s is well separated from the hyperplane with good margin, while a smaller numerical value means that there is a good chance that the string might have been wrongly classified. One can visualize the position of a data point (or entire clouds) by plotting $D(s)$ along a line as shown in the figure below. These diagrams will be extensively used in Chapter III below. The quantity $D(s)$ represented in the the figure is proportional to the distance from the hyperplane. The blue dots represent data which are classified by the model correctly, while the orange cross signs represent the wrong predictions by the model.



I.7. Measures of Effectiveness and the f1 Score

In this section, we introduce the measures of effectiveness of binary classification which are commonly used in the machine-learning literature. These are *precision, recall* and the *f1 score*.

I.7.1. Precision

Precision is defined as the fraction of relevant instances among all retrieved instances. In other words, it is number of correctly identified positive results divided by the number of all positive results including those not identified correctly. So, Precision (P) is number of true positives (T_p) over the number of true positives plus the number of false positives F_p

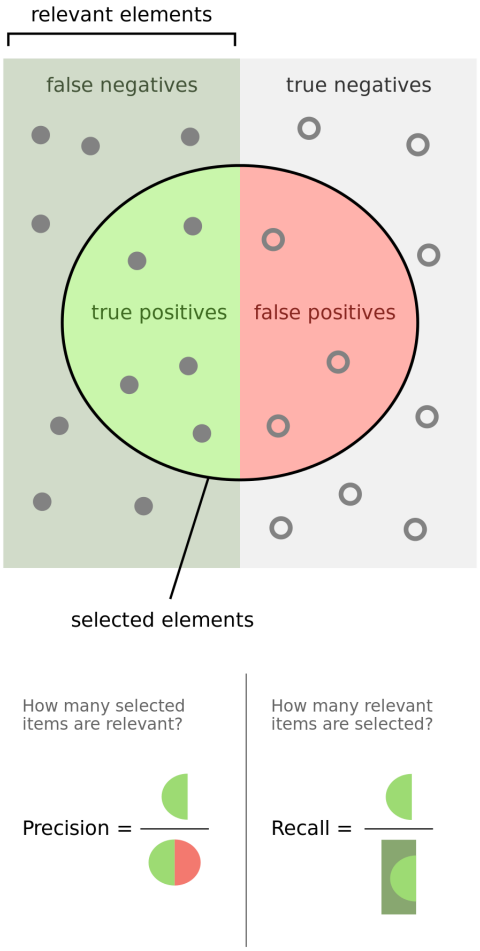
$$P = \frac{T_p}{T_P + F_p}$$

I.7.2. Recall

Recall is fraction of retrieved instances among all relevant instances. In other words, the number of correctly identified positive results divided by the number of all samples that should

have been identified as positive. So, Recall (R) is number of true positives (T_p) over the number of true positives plus the number of false negatives (F_n).

$$R = \frac{T_p}{T_P + F_n}$$



I.7.3. $f1$ Score

As we can see both precision and recall play important role in determining the efficiency of the model. In order to give equal importance to both precision and recall, it is customary to take the harmonic mean of the two which is called by $f1$ score. The $f1$ score is therefore given by $\frac{2PR}{P+R}$, where P is the precision and P is the recall.

CHAPTER II

IMPLEMENTATION

In this chapter we will describe how we implemented the theoretical considerations of the previous chapter. Among other things, we will describe the algorithms used for computation, the data to which it was applied and other implementation details.

II.1. Code Implementation

For this project we chose Python as the coding language to implement the algorithms and techniques which we have discussed in our previous chapter. Python is an interpreted, high-level and general-purpose programming language which is widely used by the open-source and machine-learning communities. There are many open-source libraries of subroutines available in this language, developed by machine-learning enthusiasts. We will use some of those libraries (described below) in our implementation.

II.1.1. `NumPy: https://numpy.org/`

`NumPy` is a library in Python, which provides a lot of support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays like normalization, matrix multiplication, dot product etc. Hence, it is very useful in the high-performance linear algebra and array operations which were essential requirements for this project. We extensively used `NumPy` throughout the project.

II.1.2. `Panda`: `https://pandas.pydata.org/`

`Panda` is another Python library which is used to manipulate array-type data structures. It converts arrays to "Dataframe" structures, which can be seamlessly integrated with Python and `NumPy`. We used it extensively, since we needed to read strings from csv (comma separated value) files and use the data to compute kernels.

II.1.3. `Pickle`: `https://docs.python.org/3/library/pickle.html`

"Pickling" is a process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. It is thus useful in storing and retrieving data with complicated internal structure. In our case, we used it to save the giant data structures that were needed in this project, such as SVM models, subsequence Gramian matrix etc.

II.1.4. `Matplotlib` and `Seaborn`: `https://matplotlib.org/`

`Matplotlib` is a comprehensive library for creating static, animated, and interactive visualizations in Python. `Seaborn` can be thought of as a bigger "wrapper" around the `Matplotlib` library which provides additional functionality for visualization of data. We used these in the visualization of the decision score data.

II.1.5. `Tweepy`: `https://www.tweepy.org/`

`Tweepy` is a library which provides a wrapper around the twitter Application Programming Interface (API) which also takes care of complex authentication of twitter. Apart from that it provides different functionalities like retreiving tweets with respect to different parameters such as counts, time, location, etc. For our project, we are particularly interested on getting tweets of different individuals posted in given time-frames.

II.1.6. `scikit`: `https://scikit-learn.org/`

`scikit-learn` is a free and open source machine learning library for Python. It is one of the popular machine learning libraries and has lot of support from machine learning enthusiasts. It has diverse functionalities, like different classification, regression, and clustering algorithms. For our project we are particularly interested in the support vector machine algorithm. We will give more details regarding it in our section of SVM implementation (section II.3 below).

II.2. Data Collection

For this project, we need to get text from different sources in order to apply our method of classification. The text should come in short strings of roughly equal sizes. An ideal source is therefore an individual twitter account, where there is a limit of 280 characters. We chose the twitter accounts of famous world leaders such as Donald Trump, Joe Biden, Emmanuel Macron, Recep Erdoğan and Boris Johnson. These leaders not only tweet often (thus generating copious amounts of data), but as we will see in the next chapter, the properties of the tweet streams as text sources are sufficiently different so that efficient classification is possible.

In the following table we describe the details of the individual twitter accounts which we used for this project.

Table 2. Twitter Data

Name	Twitter Name	From	To	Count
Donald Trump	realDonaldTrump	01 Jan 2020	04 Oct 2020	4187
Joe Biden	JoeBiden	01 May 2020	14 Oct 2020	1510
Emmanuel Macron	EmmanuelMacro	01 Jan 2020	11 Oct 2020	746
Recep Tayyip Erdoğan	RTErdogan	01 July 2020	20 Oct 2020	226
Boris Johnson	BorisJohnson	01 June 2019	29 Oct 2020	1488

We collected the tweets from the above-mentioned accounts using the open source library `tweepy`. We obtained the permission from Twitter, Inc. to crawl the data from twitter.com. For this purpose we need to set up a developer account at `developer.twitter.com`.

The other two data sources which we used for our project are the literary sources King James Version (KJV) of the Bible and the writings of Shakespeare. These are 400 year-old texts in the public domain. We used the project Gutenberg text files of KJV and Shakespeare, which are freely available on the internet.

Table 3. Literary Sources

Source	Source Name	Link
Bible	The KJV of the Bible	http://www.gutenberg.org/ebooks/10
Shakespeare	Hamlet	http://www.gutenberg.org/ebooks/1787
	Julius Caesar	http://www.gutenberg.org/ebooks/1785
	Macbeth	http://www.gutenberg.org/ebooks/2264
	Henry IV part I	http://www.gutenberg.org/ebooks/1780
	Merchant of Venice	http://www.gutenberg.org/ebooks/1515
	Midsummer Night's Dream	http://www.gutenberg.org/ebooks/1514

II.3. SVM Implementation

To efficiently utilize limited time available for our project we used the already available SVM functionality from the library `scikit-learn` described below.

II.3.1. Scikit-learn SVM

Scikit-learn library provides a complete and efficient SVM functionality. As we saw in our previous chapter if our SVM section (I.2) the problem of constructing the separating hyperplane boils down to solving quadratic programming problem (I.15). The scikit-learn package does this efficiently for us. Another advantage of the scikit-learn package is that it can use user-defined kernels (such as our normalized string subsequence kernel), which is provided to the routine as a precomputed Gramian matrix. The functionality returns the SVM model parameters described in Page 10. Below we describe in more detail the functionalities present in the svm implementaion of scikit-learn. Recall the decision function

$$f(x) = \sum_{(\xi,\eta)\in SV} \alpha_\xi \eta \langle x, \xi \rangle + b$$

introduced in (I.14) which represents the SVM model. The parameters of this model can be accessed using the following functions:

Table 4. Scikit Learn SVM Functionality

Model Parameter	Details
`decision_function`	$f(x)$
`n_support_vectors_`	Number of support vectors in each class
`support_vectors_`	Enumeration of training data representing SVs
`dual_coef_`	coefficients $\eta\alpha_x$
`intercept_`	b

In the soft-margin hyperplane construction (I.2.1), we also need to specify the parameter C which determines the amount of penalty for wrong classification, when the data is not separable by hyperplanes. For our project we chose the value of the constant $C = 1$, which is standard in machine-learning practice. Ideally, we need to find the optimal value of C using grid-search by cross validation method, but with the computing power and time at our disposal this is not practically feasible.

II.4. Kernel Implementation

In this section, we describe some practical aspects of the computation of the kernel K of (I.22). Notice that once we have the kernel K, it is easy to compute the normalized kernel $\overline{K}$ using (I.23).

Notice first, that $K(s,t)$ actually depends on the parameters n and λ. We will denote $K = K_n$ in this section to make precise the dependence on n. We describe now a recursive algorithm suggested in [4] which allows an efficient computation of K_n. From (I.22) we get,

$$K_n(s,t) = \sum_{u\in\Sigma^n} \sum_{\substack{i\in\mathbb{N}^n \\ s[i]=u}} \sum_{\substack{j\in\mathbb{N}^n \\ t[j]=u}} \lambda^{l(i)+l(j)}. \tag{II.1}$$

A direct implementation of this formula to compute $K_n(s,t)$ would be $O(|\Sigma|^n |s| |t|)$ in time, which is very costly. Therefore, we need better ways to compute K_n.

We observe that we do not really need to compute the sum over all substrings of length n, since if for a $u \in \Sigma^n$, the prefix substring $u_1 u_2 \ldots u_k$ obtained by deleting the last $n-k$ characters of u does not occur in s, then u a fortiori does not occur in s. In order to use this insight to simplify our computation, we introduce the quantity K'_k, for each k, with $1 \leq k \leq n-1$, given by

$$K'_k(s,t) = \sum_{u \in \Sigma^n} \sum_{\substack{i \in \mathbb{N}^n \\ s[i]=u}} \sum_{\substack{j \in \mathbb{N}^n \\ t[j]=u}} \lambda^{l(i)+l(j)-i_1-j_1+2} \tag{II.2}$$

For strings s,t, we denote by $|s|$ the length of the string $s = s_1 \ldots s_{|s|}$, and by st the string obtained by concatenating the strings s and t. The string $s[i:j]$ is the substring $s_i \ldots s_j$ of s. It is easy to see that the "kernel" K'_k satisfies the following recursive relations:

$$K'_0(s,t) = 1 \qquad \text{for all } s,t$$

$$K'_k(s,t) = 0 \qquad \text{if } \min(|s|,|t|) < k$$

$$K_k(s,t) = 0 \qquad \text{if } \min(|s|,|t|) < k$$

$$K'_k(sx,t) = \lambda K'_k(s,t) + \sum_{j:t_j=x} K'_{k-1}(s,t[1:j-1])\lambda^{|t|-j+2} \qquad \text{for all } x \in \Sigma$$

$$K_n(sx,t) = \lambda K_n(s,t) + \sum_{j:t_j=x} K'_{n-1}(s,t[1:j-1])\lambda^2 \qquad \text{for all } x \in \Sigma.$$

Using the above recursive relations, we now present an efficient recursive computation of the string subsequence kernel K_n that reduces the time complexity of the computation to $O(n|s||t|)$, by first evaluating

$$K''_k(sx,t) = \sum_{j:t_j=x} \lambda K'_{k-1}(s,t[1:j-1])\lambda^{|t|-j+2}$$

and observing that we can then evaluate $K'_i(s,t)$ with the $O(|s||t|)$ recursion,

$$K'_k(sx,t) = \lambda(K'_n(s,t) + \lambda K'_{k-1}(sx,t))$$

Now, observe that $K''(sx,tu) = \lambda^{|u|} K''(sx,t)$, provided x does not occur in u, while

$$K''_k(sx,tx) = \lambda(K''_k(sx,t) + \lambda K'_{k-1}(s,t)$$

These observations together give an $O(|s||t|)$ recursion for computing $K''_i(s,t)$. Hence, we can evaluate the overall kernel in $O(n|s||t|)$ time. This algorithm is described in pseudocode below.

Algorithm 1 Subsequence Kernel Algorithm

```
1:  procedure KERNEL(s, t, n, λ)
2:      length_s ← length of s
3:      length_t ← length of t
4:      K_p ← 3-dimensional array[length_s][length_t][n+1]
5:      K_p[0][i][j] ← 1.0 i = 1,···,length_s j = 1,···,length_t
6:      for k = 1,···,n do
7:          for i = 1,···,length_s − 1 do
8:              K_pp ← 1.0
9:              for j = 1,···,length_t − 1 do
10:                 K_pp ← λ ∗ (K_pp + λ ∗ (s[i] == t[j]) ∗ K_p[k][i][j])
11:                 K_p[k+1][i+1][j+1] ← λ ∗ K_p[k+1][i][j+1] + K_pp
12:     K ← 0
13:     for k = 1,···,n do
14:         for i = 1,···,length_s do
15:             for j = 1,···,length_t do
16:                 K ← K + λ ∗ λ ∗ (s[i] == t[j]) ∗ K_p[k][i][j]
        return K
```

II.5. Choice of Parameters

The kernel K used in our technique depends on the parameters n and λ. Ideally, these parameters would be selected for each model generated from data. However, that would need more time and computing power than we could afford. Therefore, we chose one set of data (Trump vs Biden) and we did a grid search with respect to the $n = \{1,2,3,4,5,6,7,8,9\}$ and $\lambda = \{0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9\}$. After different combination of experiment we got optimal value of $n = 5$ and $\lambda \approx 0.85$ with respect to the f1 score. We used these values of the parameters in all our experiments.

It is interesting to conjecture why $n = 5$ may be the best choice of this parameter. Notice that in the definition of the feature map (I.21), n represents the length of the substrings being compared. Since the average length of an English word in typical prose is about 5.5 (as can be readily checked by word counting programs), it seems that the fact that $n = 5$ is the optimal value of the paramter is related to this.

We also tried to understand why $\lambda = 0.85$ is the best value for this parameter, but were not successful.

We should also note here that in some of the experiments, the data was so well-separated that other values of n and λ also work. These will be discussed in the last chapter.

II.6. Parallelization

The computations we needed to perform for this project were extremely time and space consuming. This is because the amount of data, as well as the number of dimensions in which linear algebra needed to be done, were both huge. For example, to compute the Gramian matrix for the kernel for each experiment to be described in the next chapter, we computed $K(s,t)$ for s,t each assuming 500 values. As discussed above, the computation of each value $K(s,t)$ took $O(n|s||t|)$ time. We can cut down the computation in half, by noticing that $K(s,t) = K(t,s)$, but nevertheless, we still need $O(nm^2|s||t|)$ time, where $m = 500$ is the number of training data in each cloud. Since the computations of different values of the kernel can be done parallely, and we were using an 8 core machine, we took advantage of the multi-core functionality of our machine by parallelizing the computation of the various values of the kernel. Effectively, 8 computers worked parallely on the job of creating the SVM models. Even then, each computation took about 4 hours.

CHAPTER III

EXPERIMENT RESULTS

In this last chapter, we will report the results of several experiments to check the performance of the subsequence-kernel based binary text-source classification method described in the two previous chapters. One of the objectives of the experiments was to understand the internal features of text that lead to successful classification of sources by our method. In other words, not all pairs of sources are (easily) distinguishable, either by humans or machines. For example, two sources may produce similar (or even the same) output, in which case we cannot tell them apart. Therefore, we concentrate on finding situations where the sources are sufficiently different to be distinguishable. Mathematically, this means that the clouds in feature space corresponding to the two sources under the mapping ϕ of (I.21) can be effectively separated by a hyperplane computed by the SVM method.

We chose the following sources for our experiments:

1. Tweets of the Turkish president Recep Tayyip Erdoğan.
2. Tweets of US president Donald J Trump.
3. Tweets of French president Emmanuel Macron.
4. Tweets of US president-elect Joe Biden.
5. Tweets of UK prime minister Boris Johnson.
6. Writings of William Shakespeare (1564–1616).
7. The Old Testament in the King James version of the Bible (1611).

Notice that there are obvious differences in the texts produced by these sources. For example, the Turkish president's tweets are mostly in Turkish, those of the French president are in French, whereas the US president tweet's almost exclusively in English. However, we want to emphasize that our method **does not** use these differences in any way. In fact, as we will see in our results, it can learn to distinguish between two languages successfully based on the training data.

The differences between the several varieties of English text (Shakespeare, Bible, Trump, Biden and Johnson) are more subtle and ill-defined, so we would expect that our system is less effective in distinguishing these. This intuition is borne out by the experiments, though there are some surprises. For example, our method successfully distinguishes between Shakespeare and the Bible, and between Trump and Biden. We will discuss below in more detail what the intrinsic differences between these sources may be.

We performed seven experiments trying to distinguish pairs of sources from the list given above. Except for experiment 1 (Erdogan vs. Trump), we used training sets of 1000 data points in each experiment, and performed 5 tests with 100 test datasets. The F1 scores were collected for each test and give an indication of the effectiveness of source separation.

III.1. Experiment 1: Trump Tweets and Erdogan Tweets

This is the simplest of our problems, since the internal characteristics of the two types of text are very different, the tweets of Erdoğan being mostly in Turkish, whereas those of Trump being in English. We used a total of 100 training data points in this experiment. The following table shows the actual data used, and the performance of the model when applied to the training data set.

Table 5. Experiment 1 Training Sets

Source (count)	Data	precision	recall	$f1$	Support vector
Trump Tweets (50)	06 Sept – 03 Oct	0.98	1.00	0.99	40
Erdoğan Tweets (50)	16 Oct – 19 Oct	1.00	0.98	0.99	37

Notice that the SVM model has a total of 77 support vectors and is able to classify the training set almost perfectly. In fact, the one tweet that was misclassified was a URL shared by Erdoğan without any accompanying text, so one can in fact say that the classification was perfect when applied to the training set.

We then used the SVM model to classify 500 additional tweets (divided into 5 lots of 100 tweets each). The following table summarizes the outcomes:

Table 6. Experiment 1 Testing Sets

Testing	Source (count)	Data	Precision	Recall	$f1$
1	Trump Tweets (50)	03 Sept - 06 Sept	0.98	1.0	0.99
	Erdoğan Tweets (50)	10 Oct - 16 Oct	1.0	0.98	0.99
2	Trump Tweets (50)	31 Aug - 03 Sept	0.98	1.0	0.99
	Erdoğan Tweets (50)	01 Oct - 10 Oct	1.0	0.98	0.99
3	Trump Tweets (50)	29 Aug - 31 Aug	0.98	1.0	0.99
	Erdoğan Tweets (50)	04 Sept - 01 Oct	1.0	0.98	0.99
4	Trump Tweets (50)	23 Aug - 29 Aug	0.94	0.98	0.96
	Erdoğan Tweets (50)	23 May - 04 Sept	0.98	0.94	0.96
5	Trump Tweets (50)	19 Aug - 23 Aug	0.94	1.0	0.97
	Erdoğan Tweets (50)	15 Jul - 23 May	1.0	0.94	0.97

Observe that the separation was very effective and almost all the tweets were correctly classified. In fact, the only errors were made when Erdoğan tweeted in English, e.g, when the Trumps contracted covid-19, Erdoğan tweeted

Recep Tayyip Erdoğan @RTErdogan · Oct 2

I wish a speedy recovery to U.S. President @realDonaldTrump and @FLOTUS Melania Trump, who tested positive for COVID-19.

I sincerely hope that they will overcome the quarantine period without problems and regain their health as soon as possible.

Some other tweets of Erdoğan, which were either in Arabic, or were simple URLs without text were also classified as being from Trump.

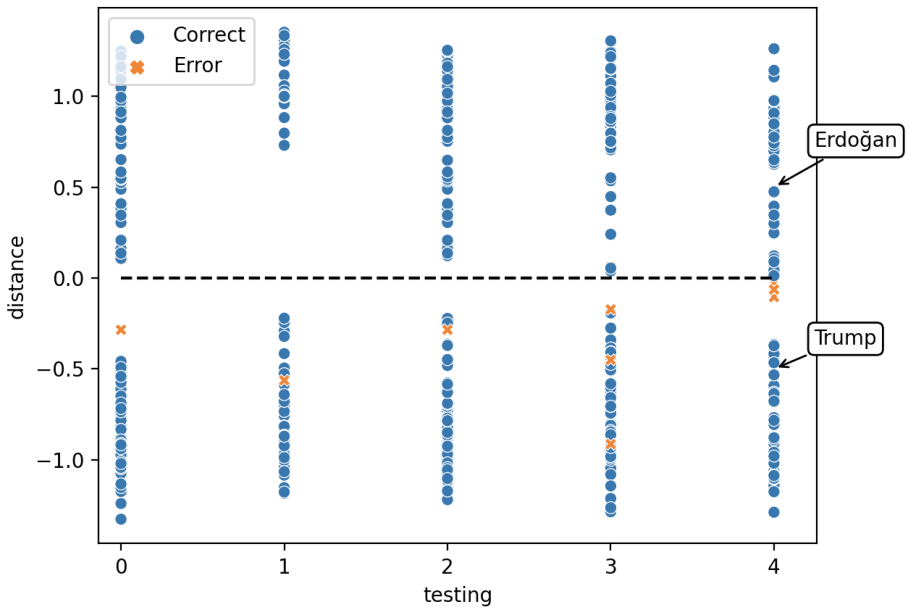
Therefore it seems that the model was in fact distinguishing between **Turkish and English** rather than Erdoğan and Trump. To verify this, the model was tested with Turkish and English text (not necessarily from tweets), and it was able to distinguish them perfectly.

The question arises that what is the intrinsic difference between Turkish and English text that the SVM model detects. Two obvious differences can be observed:

1. Turkish uses a number of accented characters not used in English, such as: ç, ğ, Ö, Ş The SVM model clearly sees the presence of these characters and uses them for separation.

2. Further, for unaccented characters, the frequency with which the characters appear in text of two languages is very different, for example 12.702% of letters in typical English text is ”e”, whereas the same number for Turkish is 9.39%. To check whether the frequency plays a role in the separation, we stripped off Turkish text of the accents, and also changed the parameter k of the feature map to 1, still obtaining good separation. This verifies that the system detects the differences of frequencies.

The following is the decision graph for this experiment. Each vertical line corresponds to one of the five tests performed. Each dot corresponds to the value of the decision function of a tweet. The distance of the dot from the horizontal line is proportional to the distance from the separating hyperplane given by the SVM. One clearly notices that there is a gap separating the two clouds, which corresponds to the fact that the Trump and Erdoğan clouds are well-separated in feature space.



III.2. Experiment 2: Trump Tweets and Macron Tweets

The second experiment is similar to the first one, in which there is an obvious difference between the two sources, which are tweets of Donald Trump and those of Emmanuel Macron, the president of France. Again, Trump tweets almost exclusively in English, whereas Macron tweets mostly in French but sometimes also in other languages, including English, Arabic etc. The following table shows the actual data used, and the performance of the model when applied to the training data set.

Table 7. Experiment 2 Training Sets

Source (count)	Data	precision	recall	$f1$	Support vector
Trump Tweets (500)	06 Sept – 03 Oct	0.96	1.00	0.98	227
Macron Tweets (500)	16 Apr – 9 Oct	1.00	0.96	0.98	153

With 380 (=227+153) support vectors were successfully able to classify tweets almost completely between Trump and Macron. In fact, the system successfully learned the difference

between French and English. All misclassified training tweets are from Macron and are in a language different from French. This is the same behaviour which we saw in Experiment 1.

We then used the SVM model to classify 500 additional tweets (divided into 5 lots of 100 tweets each). The following table summarizes the outcomes:

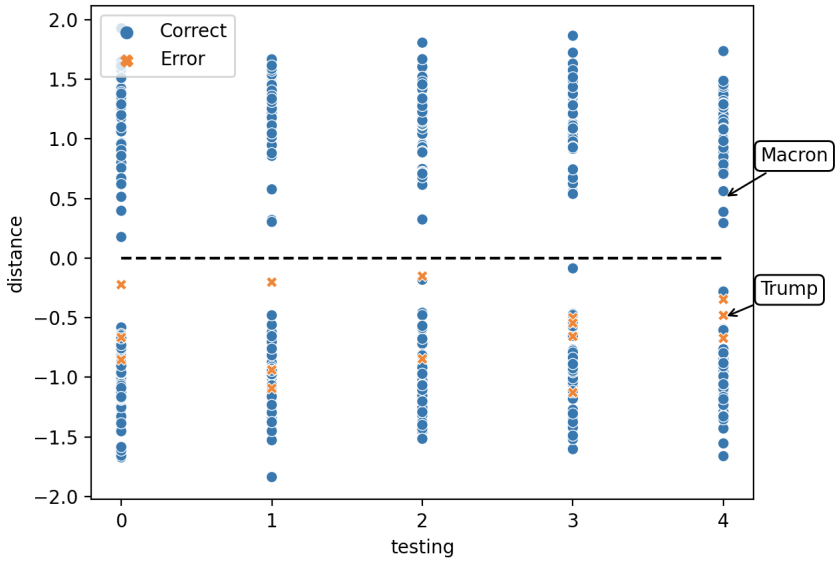
Table 8. Experiment 2 Testing Sets

Testing	Source (count)	Data	Precision	Recall	$f1$
1	Trump Tweets (50)	03 Sept - 06 Sept	0.94	1.0	0.97
	Macron Tweets (50)	10 Mar - 19 Mar	1.0	0.94	0.97
2	Trump Tweets (50)	31 Aug - 03 Sept	0.94	1.0	0.97
	Macron Tweets (50)	31 Jan - 10 Mar	1.0	0.94	0.97
3	Trump Tweets (50)	29 Aug - 31 Aug	0.96	1.0	0.98
	Macron Tweets (50)	07 Jan - 31 Jan	1.0	0.96	0.98
4	Trump Tweets (50)	23 Aug - 29 Aug	0.93	1.0	0.96
	Macron Tweets (50)	04 Jan - 07 Jan	1.0	0.92	0.96
5	Trump Tweets (50)	19 Aug - 23 Aug	0.94	1.0	0.97
	Macron Tweets (50)	01 Jan - 04 Jan	1.0	0.94	0.97

To verify our conjecture that the model actually separates English and French, it was tested with French and English text (not necessarily from tweets), and it was able to distinguish them perfectly.

One can ask the same question as we asked in Experiment 1, i.e. what are the distinguishing factors between French and English text. The reasons are exactly the same, (i) accented characters such as ”é”, ”ö” in French, which do not occur in English, and (ii) the difference in frequencies of the letters. It should be noted that as expected, the similarity between these two sources are much more than those between Turkish and English, so we do need a model with larger training set for successful separation.

From decision score graph below, it is quite evident that the text from the Trump and Macron was nicely separated as different cloud in higher dimensional space and in the graph it is clearly separated with the good margin from each side. Hence, we can conclude that French and English has quite distinct characteristics which was learned by the model and giving expected result.



III.3. Experiment 3: Trump Tweets and Shakespeare

In our third experiment we try to see whether our model is able to differentiate between two different English literary styles. We choose our data sources as the plays of William Shakespeare (1564–1616) and tweets of Donald Trump (1946–). These two sources are very different in almost every way possible: they belong to different eras, different spiritual and moral universes. Further, Shakespeare's plays are written primarily in verse, mostly iambic pentameter, i.e. lines of 10 syllables each, which is approximately 40 characters per line. A tweet is at most 280 characters, usually less, and on an average 170 characters for Trump's tweets. To maintain a balance in the data, four of Shakespeare's lines were concatenated to make one training string. The following table shows the actual data used, and the performance of the model when applied to the training data set.

Table 9. Experiment 3 Training Sets

Source (count)	Data	precision	recall	$f1$	Support vector
Trump Tweets (500)	06 Sept – 03 Oct	1.00	0.99	0.99	207
Shakespeare[1] (500)	Hamlet 1.1.1 – 3.2.6	0.99	1.00	1.00	231

As we can notice the model has classified the training data with good accuracy. We expect that there will be some mistakes, since now the data are much more similar as compared to experiments 1 and 2. We also notice that there is increment in number of support vectors (438) in comparison to the last two experiments. This means that the data is more difficult to separate. We tested our model using 500 strings from each source (divided into 5 lots of 100 tweets each). The following table summarizes the outcomes:

Table 10. Experiment 3 Testing Sets

Training	Source (count)	Data	Precision	Recall	$f1$
1	Trump Tweets (50)	03 Sept - 06 Sept	0.87	0.9	0.88
	Shakespeare[2] (50)	Julius Caesar 1.1.1 - 4.2.5	0.9	0.86	0.88
2	Trump Tweets (50)	31 Aug - 03 Sept	0.85	0.9	0.87
	Shakespeare[2] (50)	Macbeth 1.2.1 - 1.3.1	0.89	0.84	0.87
3	Trump Tweets (50)	29 Aug - 31 Aug	0.94	0.88	0.91
	Shakespeare[2] (50)	Henry IV part I 1.1.1 - 1.2.7	0.89	0.94	0.91
4	Trump Tweets (50)	23 Aug - 29 Aug	0.92	0.98	0.95
	Shakespeare[2] (50)	The Merchant of Venice 1.1.1 - 1.2.2	0.98	0.92	0.95
5	Trump Tweets (50)	19 Aug - 23 Aug	0.87	0.96	0.91
	Shakespeare[2] (50)	A Midsummer Night's Dream 1.1.1 - 3.2.14	0.96	0.86	0.91

[1]The 500 input strings of Shakespeare correspond to 2000 actual verse lines of *Hamlet*

[2]The 50 input strings of Shakespeare correspond to 200 actual verse lines

The experiments achieve reasonable accuracy, but with some errors, which are expected since now the sources are more similar that the two previous experiments. A typical error is

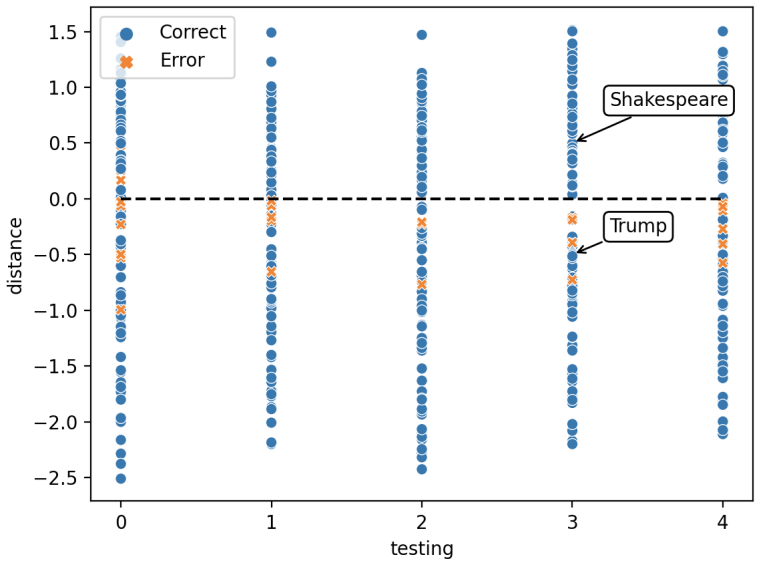
> Or else to wed Demetrius, as he would, Or on Diana's altar to protest

is from *Midsummer Night's Dream*, but is misclassified by our system as being from Trump. Similarly Trump's tweet from 20 August:

> In 47 years, Joe did none of the things of which he now speaks. He will never change, just words

is misclassified as being by Shakespeare. It is not easy to see the basis on which the correct decisions were made, and how the above two examples were not classified correctly. One possibility, which is worth investigating, is the difference in vocabulary in the two sources. Some conjectures can be made: the word *protest* does not occur in the first 2000 verses of Hamlet (our training data), but is used by Trump in his tweets (8 times in our training data). If we test our model with the single word "protest" as input, the system detects it as being from Trump. One important structural difference (which must have been used in the classification by our system) is that Shakespeare uses much more punctuation than Trump: e.g., in the training data, Shakespeare uses a comma 1630 times, whereas Trump uses only 734 commas.

From the decision score graph, we can clearly note that graph is continuous not well separated as we saw in last two experiments. Orange points in the graph are misclassified data which are seen to lie almost on the boundary between the two regions, and near the separating hyperplane. This suggests that a larger model, with more training data would be more successful in classifying these two types of text.



III.4. Experiment 4: Shakespeare and Bible

In our fourth experiment, we try to classify text-sources from the same era. We choose our data sources to be two of the monuments of English literature, the King James version of Bible and Shakespeare's plays. The KJV bible was compiled in 1611, so these represent English of approximately the same era. The KJV Bible is in prose, divided into short sections corresponding to verses in the Hebrew original. On an average, a verse of Genesis has 101 characters, so three verses were concatenated to produce strings of approximately the same size as the strings from Shakespeare (4 lines of iambic pentameter).The tables below give more information about the data sources we used and accuracy we got in classifying the training sets.

Table 11. Experiment 4 Training Sets

Source (count)	Data	precision	recall	$f1$	Support vector
Shakespeare[3] (500)	Hamlet 1.1.1 – 3.2.6	1.00	0.99	0.99	214
Bible[4] (500)	Genesis 1:1 – 25:13	0.99	1.00	0.99	212

[3]The 500 input strings of Shakespeare correspond to 2000 actual verse lines of *Hamlet*
[4]The 500 input strings of Bible correspond to approximately 1500 actual verse lines of *Genesis*

As we can see the model is able to classify the training data quite efficiently. One text which got classified wrongly was

> I will not again curse the ground any more for man's sake; for the imagination of man's heart is evil from his youth; neither will I again smite any more every thing living, as I have done

It is difficult to decide on the basis of language alone whether this is from the Bible or from Shakespeare (it is actually from the Bible). Our model also had the same difficulty, and was not able to classify this training data correctly, i.e., this training point is on the wrong side of the separating hyperplane. The table below gives details of five experiments in classifying Shakesperean and Biblical text, along with the f1 scores:

Table 12. Experiment 4 Testing Sets

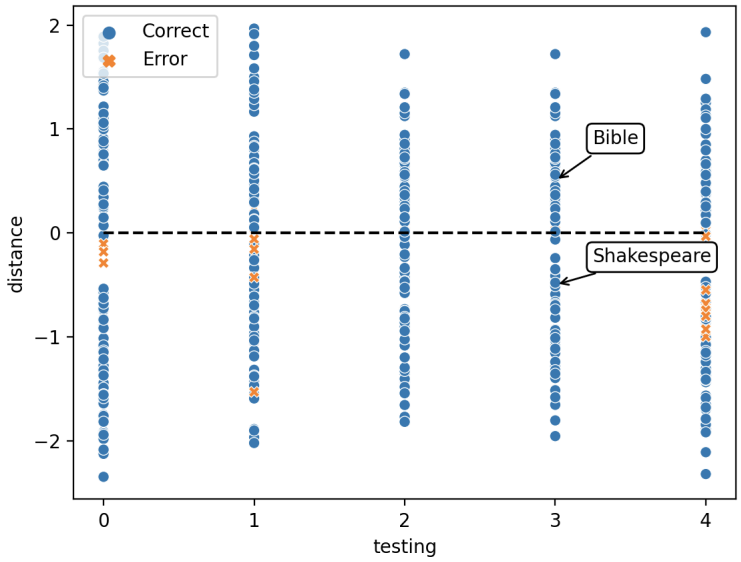
Training	Source (count)	Data	Precision	Recall	$f1$
1	Shakespeare[5] (50)	Julius Caesar 1.1.1 - 4.2.5	0.94	1.0	0.97
	Bible[6] (50)	Exodus 1:1 – 40:32	1.0	0.94	0.97
2	Shakespeare[5] (50)	Macbeth 1.2.1 - 1.3.1	0.92	0.92	0.92
	Bible[6] (50)	Leviticus 1:5 – 4:10	0.92	0.92	0.92
3	Shakespeare[5] (50)	Henry IV part I 1.1.1 - 1.2.7	1.0	0.92	0.96
	Bible[6] (50)	Numbers 1:1 – 2:10	0.93	1.0	0.96
4	Shakespeare[5] (50)	The Merchant of Venice 1.1.1 - 1.2.2	1.0	0.96	0.98
	Bible[6] (50)	Joshua 1:1 – 3:20	0.96	1.0	0.98
5	Shakespeare[5] (50)	A Midsummer Night's Dream 1.1.1 - 3.2.14	0.75	0.96	0.84
	Bible[6] (50)	Deuteronomy 1:1 – 2:14	0.94	0.68	0.79

[5]The 50 input strings of Shakespeare correspond to 200 actual verse lines

[6]The 50 input strings of Bible correspond to approximately 150 actual verse lines

As we can see from the testing results the model can classify the sources quite efficiently, with one exception. It seems that the book of Deuteronomy is somehow rather different from the training data (the first 1500 verses of Genesis). It would be interesting to understand this difference and its causes. In the decision graph these errors correspond to the orange crosses.

We tried to understand the differences between these two sources by analyzing some samples. One interesting fact that emerged is that the Bible and Shakespeare are punctuated very differently. Shakespeare uses a much higher amount and variety of punctuation than as compared to the KJV. For example, in the training data from Hamlet, there were 486 uses of the apostrophe, whereas in the training data from Genesis, there are 75 uses, though the two training data are roughly comparable in size. Part of the reason is the use of contracted forms like *o'er, e'er* by Shakespeare, which never occurs in the Bible. Also, Shakespeare likes to use dashes and hyphens more often than the translators of the KJV.



III.5. Experiment 5: Trump Tweet and Biden Tweet

The fifth experiment is arguably the most important experiment of this project. The two sources for this experiment are the tweets of Trump and Biden which share a lot in common (since, even when expressing opposing viewpoints, they have to use a large shared vocabulary).

It is therefore interesting to see whether our model is able to find the differences between the data sources and classify the tweets. The table below gives more details about the data sources and the model classification efficiency over training sets.

Table 13. Experiment 5 Training Sets

Source (count)	Data	precision	recall	$f1$	Support vector
Trump Tweet (500)	06 Sept – 19 Oct	0.99	0.98	0.99	307
Biden Tweet (500)	18 Aug – 30 Sept	0.98	0.99	0.99	319

As we can see we need 626 (=307+319) support vectors to define the separating hyperplane for 1000 data points which is the highest among all the experiments which we performed. This suggests that the clouds represented by the Trump and Biden tweets in feature space are not very well separated hence need more support vectors for good classification. The model gives a good f1 score for classification of the training sets.

Table 14. Experiment 5 Testing Sets

Training	Source (count)	Data	Precision	Recall	$f1$
1	Trump Tweet (50)	03 Sept - 06 Sept	0.96	0.86	0.91
	Biden Tweet (50)	14 Aug - 18 Aug	0.87	0.96	0.91
2	Trump Tweet (50)	31 Aug - 03 Sept	1.0	0.92	0.96
	Biden Tweet (50)	08 Aug - 13 Aug	0.93	1.0	0.96
3	Trump Tweet (50)	29 Aug - 31 Aug	1.0	0.92	0.96
	Biden Tweet (50)	03 Aug - 08 Aug	0.93	1.0	0.96
4	Trump Tweet (50)	23 Aug - 29 Aug	0.96	0.9	0.93
	Biden Tweet (50)	28 July - 02 Aug	0.91	0.96	0.93
5	Trump Tweet (50)	19 Aug - 23 Aug	0.94	0.92	0.93
	Biden Tweet (50)	22 July - 28 July	0.92	0.94	0.93

Even though the tweets of the two leaders share a lot of area of common interest, our model is still able to classify them quite efficiently. This is surprising, so we further tried to analyze the tweets to understand what characteristics of the tweets might have played a role in the classification by our model. Some of the observations we made while analyzing the texts are collected below.

1. We noticed that in the training data, the word *pandemic* was used by Joe Biden 13 times and never by Donald Trump. Hence, whenever the model classified tweets containing the word "pandemic", it was inclined to classify them as being from Biden. Perhaps there are other similar features of the vocabulary of the two leaders that our model has detected.

2. Words like *MAGA*, *Make America great again*, *Sleepy Joe Biden* are signatures of Trump in his tweets. Tweets containing these words were classified as being from Trump correctly.

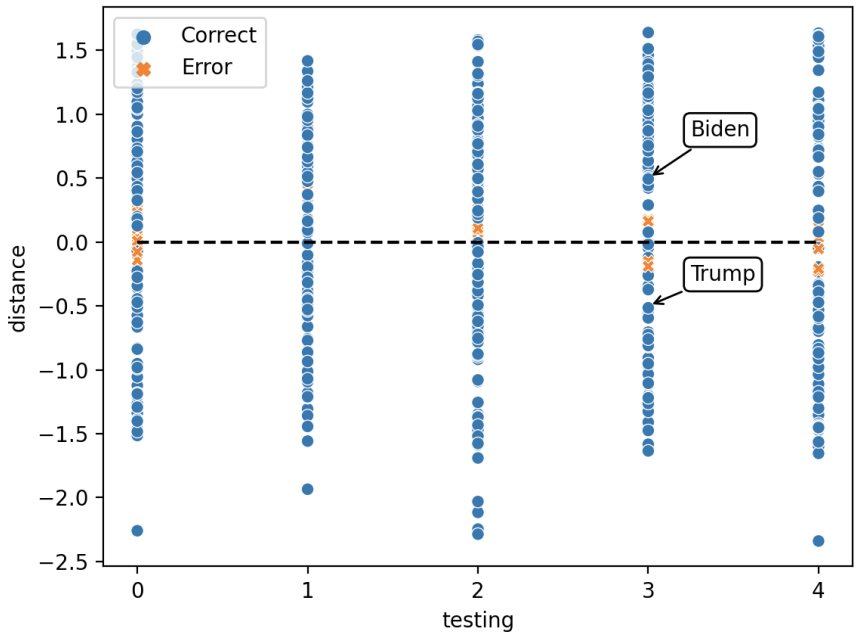
3. Trump shares links of videos and websites more often than Biden. So tweets with such links were classified as being from Trump (not always correctly).

In spite of the success of the model, it still remains a mystery what exactly are the characteristics it detects. It would be very interesting to gain further insight into this.

We also also see that the orange points (misclassified points) are always near the separating hyperplane which indicates that these points share the characteristics of both data sources, and cannot be easily classified (even by knowledgeable humans). An example of a tweet which was misclassified is

Thank you to @CardinalDolan for this evenings opening prayer.

This was classified as of Joe Biden by the model but actually it was tweeted by Trump on 24th Aug. From the content, even humans are unable to decide who it is from, unless they know more about the context.



III.6. Experiment 6: Biden Tweet and Bible

The sixth experiment is similar to the third experiment (Trump vs. Shakespeare). Here we consider the tweets of Biden in 21st century English compared to the verses of the KJV (17th century). Below are the given details of the data sources of the training datasets and the efficiency of the classification of the training data.

Table 15. Experiment 6 Training Sets

Source (count)	Data	precision	recall	$f1$	Support vector
Biden Tweet (500)	18 Aug – 30 Sept	1.00	1.00	1.00	119
Bible (500)	Genesis 1:1 – 25:13	1.00	1.00	1.00	138

The model is able to differentiate the training sets perfectly. And we can see with just 257(=119+138) support vectors, we are able to define the separating hyperplane, which suggests that the clouds formed by Bible and Joe Biden texts in feature space are easily separable. To un-

derstand more about the model, we further analyzed efficiency with 5 different sets of experiments whose details are given below.

Table 16. Experiment 6 Testing Sets

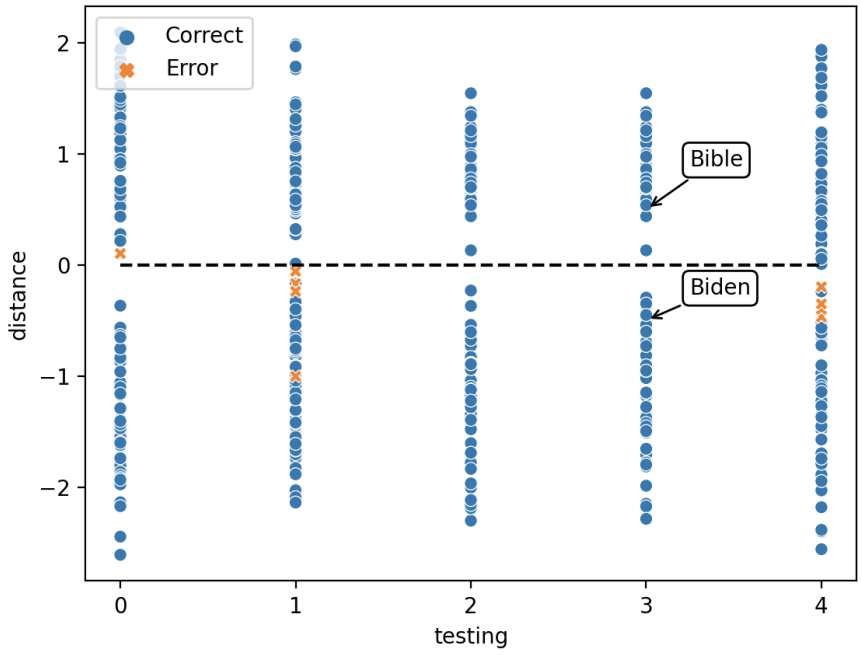
Training	Source (count)	Data	Precision	Recall	$f1$
1	Biden Tweet (50)	14 Aug - 18 Aug	1.0	0.98	0.99
	Bible (50)	Exodus 1:1 – 40:32	0.98	1.0	0.99
2	Biden Tweet (50)	31 Aug - 03 Sept	0.93	1.0	0.96
	Bible (50)	Leviticus 1:5 – 4:10	1.0	0.92	0.96
3	Biden Tweet (50)	29 Aug - 31 Aug	1.0	1.0	1.0
	Bible (50)	Numbers 1:1 – 2:10	1.0	1.0	1.0
4	Biden Tweet (50)	23 Aug - 29 Aug	1.0	1.0	1.0
	Bible (50)	Joshua 1:1 – 3:20	1.0	1.0	1.0
5	Biden Tweet (50)	19 Aug - 23 Aug	0.93	1.0	0.96
	Bible (50)	Deuteronomy 1:1 – 2:14	1.0	0.92	0.96

In all our experiments we are able to differentiate the two sources with good accuracy. As we can notice from the graph below the clouds are quite well separated having some margin of distance from the hyperplane. The fact that the two souces are well-classified by our model is not surprising, given that the vocabularies contain elements which occur in one and not the other, e.g., words like “Trump”, “immigration”, etc. were used by Biden but not the Bible, and the Bible used words like “Hittites”, “Amorites”, “Pharaoh” which were never used by Biden.

Observe that the Orange cross signs (misclassified data points) are near the boundary. The following is an interesting example of a misclassified string:

> I myself alone bear your cumbrance, and your burden, and your strife? Take you wise men, and understanding, and known among your tribes, and I will make them rulers over you.

This is from the Bible (Deuteronomy 1:12) but was classified as from Biden. It would be interesting to see if a larger model gives better results.



III.7. Experiment 7: Trump Tweet and Johnson Tweetgam

In the last experiment we try to classify tweets of Donald Trump and Boris Johnson (Prime Minister of the UK). Given below are the details of the data sources and model efficiency when tested against the training data.

Table 17. Experiment 7 Training Sets

Source (count)	Data	precision	recall	$f1$	Support vector
Trump Tweets (500)	05 Sept – 03 Oct	1.00	0.94	0.97	274
Johnson Tweets (500)	17 May – 28 Oct	0.94	1.00	0.97	300

We notice that we need 574(=274+300) support vectors. This indicates difficulty in separating Trump and Johnson, though this problem is less difficult than Trump and Biden, which needs 626 support vectors. An interesting example of a misclassified tweet is

> We thank God for the blessings we share as citizens of the Greatest Country on earth – and we hope, pray, and work for the day when the people of Cuba can finally reclaim their glorious destiny!

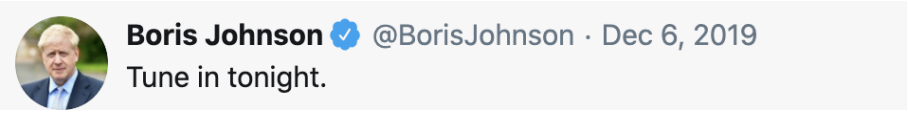
This is obviously (to a human knowing the context) a tweet of Trump (which he posted on September 23). However, our model classified it wrongly.

As with the other modes, we performed 5 different tests:

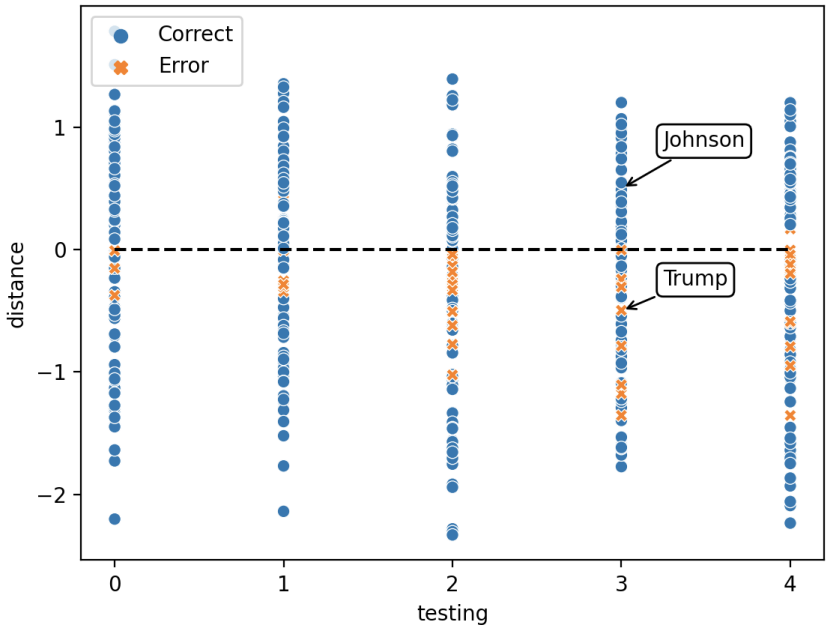
Table 18. Experiment 7 Testing Sets

Training	Source (count)	Data	Precision	Recall	$f1$
1	Trump Tweet (500)	03 Sept - 06 Sept	0.92	0.9	0.91
	Johnson Tweet (500)	29 Jan - 17 Mar	0.9	0.92	0.91
2	Trump Tweet (500)	31 Aug - 03 Sept Aug	0.88	0.88	0.88
	Johnson Tweet (500)	17 Dec 19- 22 Jan	0.88	0.88	0.88
3	Trump Tweet (500)	29 Aug - 31 Aug	0.76	0.88	0.81
	Johnson Tweet (500)	11 Dec 19 - 17 Dec 19	0.86	0.72	0.78
4	Trump Tweet (500)	23 Aug - 29 Aug	0.82	0.84	0.83
	Johnson Tweet (500)	09 Dec 19 - 11 Dec 19	0.84	0.82	0.83
5	Trump Tweet (500)	9 Aug - 23 Aug	0.8	0.94	0.86
	Johnson Tweet (500)	06 Dec 19 - 09 Dec 19	0.93	0.76	0.84

As we can see model is not quite efficient in classifying the testing data. This means that perhaps as leaders of two English-speaking countries with similar jobs, cultural similarities and common concerns, perhaps Trump and Johnson do sound very similar. Also, it is not clear what role the differences between British and American varieties of English have played in the classification. More careful study of the data is needed to understand this.



Our model wrongly classified this tweet as being from Trump rather than Johnson, which is quite understandable, since nothing in the tweet itself gives an indication of the authorship.



CHAPTER IV

CONCLUSION AND FURTHER WORK

The main goal of this project was to understand some of the mathematical underpinnings of modern machine learning. It turns out, that very good results can be obtained with very little mathematics, but using large amount of computing power. The technique employed by us did not use much more than linear algebra, and a few ideas of optimization. We were able to use a ready-made SVM model builder, we could concentrate on constructing the kernel and doing a series of experiments to check the working of the SVM model. The experiments revealed interesting facts about the degree of similarity of different sources of text.

This project can be thought of as a very preliminary step. Machine learning today is a huge field with many techniques and methods, such as neural networks, deep learning etc. There is a lot to do and learn in this area, and the links with mathematics are very deep and strong.

APPENDICES

Appendix A

Csv Reader: Read data from a CSV file and return the object as an array

```
import numpy as np
import Constant as C


class CsvReader:
    def __init__(self, filename):
        self.fileName = filename

    def get_lines(self):
        file_read = open(self.fileName, 'r')
        return file_read.readlines()

    def get_text_input_data(self):
        x = []
        y = []
        for line in self.get_lines():
            array_line = line.rstrip().split(',')
            x.append("".join(array_line[1:]))
            y.append(int(array_line[0]))
        return np.array(x).reshape((len(x), 1)), np.array(y)

    def get_text_data(self):
        x = self.get_lines()
        return np.array(x).reshape((len(x), 1))
```

Csv Writer: Write the array into a CSV file

```
import sys

sys.path.append('../')
import Constant as constant


class CsvWriter:
    def __init__(self, file_name):
        self.file_name = file_name

    def write_to_file(self, data):
        with open(constant.data_folder + self.file_name, 'w') as fd:
            fd.write(data)

    def append_to_file(self, data):
        with open(self.file_name, 'a+') as fd:
            fd.write(data)
```

Tweet Process: Retrieve tweets from Twitter

```
import tweepy
import csv

access_key = '##################'
access_secret = '##################'
consumer_key = '##################'
consumer_secret = '##################'

TOKEN="##################"
def search_tweets(text_query, number_of_tweets):
    # initialize tweepy
    auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
    auth.set_access_token(access_key, access_secret)
    api = tweepy.API(auth)
```

```
    # initialize a list to hold all the tweets
    all_tweets = []

    # make initial request for most recent tweets (200 is the maximum allowed count)
    new_tweets = api.search(q=text_query, count=300, tweet_mode="extended")
    # print(new_tweets)
    all_tweets.extend(new_tweets)
    while len(new_tweets) > 0:
        if len(all_tweets) >= number_of_tweets - 100:
            break
        new_tweets = api.search(q=text_query, count=200, tweet_mode="extended")
        all_tweets.extend(new_tweets)
        print("...%s tweets extracted so far" % (len(all_tweets)))

    out_tweets = [[tweet.full_text.strip()] for tweet in all_tweets]

    # write the csv file
    with open('%s.csv' % text_query, 'w') as f:
        writer = csv.writer(f)
        writer.writerows(out_tweets)
    pass


if __name__ == '__main__':
    # X is approximate number of tweets to be retrieved.
    search_tweets('from:realDonaldTrump', 300)
```

Constant: Constant parameters in the code

```
base_folder = "/Users/pradeep/Files/ML/StringKernel/"
model_folder = base_folder + "Model/"
data_folder = base_folder + "data/"
input_folder = data_folder + "input/"
result_folder = data_folder + "result/"
test_folder = data_folder + "test/"
inputFile = "/input.csv"
inputFile_0 = "/input_0.csv"
inputFile_1 = "/input_1.csv"
testFile_0 = "/test_0.csv"
testFile_1 = "/test_1.csv"
testFile = "/test.csv"

REPUBLICAN = "0"
DEMOCRATS = "1"
m_lambda = 0.85
N = 5
m_lambdas = [0.85]
NValues = [5]

cases = {
    "case1": {
        0: "Trump",
        1: "Erdogan"
    },
    "case2": {
        0: "Trump",
        1: "Shakespeare"
    },
    "case3": {
        0: "None",
        1: "None"
    },
    "case4": {
        0: "Trump",
        1: "Biden"
    },
```

```
    "case5": {
        0: "Trump",
        1: "Macron"
    },
    "case6": {
        0: "Shakespeare",
        1: "Bible"
    },
    "case7": {
        0: "Democrats",
        1: "Republican"
    },
    "case8": {
        0: "Biden",
        1: "Bible"
    },
    "case9": {
        0: "Trump",
        1: "Johnson"
    }
}
```

Util: Utility file

```
import Constant as C
from TextProcessing.CsvWriter import CsvWriter
import pickle
from sklearn import metrics
import numpy as np
from TextProcessing.CsvReader import CsvReader


def get_case_file_name(case_name, m_lambda, n):
    return "_".join([case_name, str(int(m_lambda * 100)), str(n)])


def get_model_file_name(prefix, case_name, m_lambda, n):
    return C.model_folder + case_name + "/" \
           + prefix + get_case_file_name(case_name, m_lambda, n)


def get_result_file_name(case_name, m_lambda, suffix):
    output_file = get_case_file_name(case_name, m_lambda, suffix)
    return get_result_folder(case_name) + output_file


def get_result_folder(case_name):
    return C.result_folder + case_name + "/"


def get_result_header(case_name, m_lambda, n):
    return " ".join([case_name, "lambda:"
                     + str(m_lambda), "N:" + str(n)]) + "\n"


def print_result(case_name, m_lambda, n, reports, y_predict, suffix):
    csv_writer = CsvWriter(
        get_result_file_name(case_name, m_lambda, suffix))
    csv_writer.append_to_file(
        get_result_header(case_name, m_lambda, n))
    csv_writer.append_to_file(reports)
    # csv_writer.append_to_file(y_predict.tostring())


def read_model(case_name, m_lambda, n):
    with open(get_model_file_name(
```

```
            "pickle_test_", case_name, m_lambda, n), 'rb') as file:
        return pickle.load(file)


def print_model(case_name, m_lambda, n, clf):
    with open(get_model_file_name(
            "pickle_test_", case_name, m_lambda, n), 'wb') as file:
        pickle.dump(clf, file)


def print_array(case_name, m_lambda, n, kernel, purpose):
    with open(get_model_file_name(
            "kernel_" + purpose + "_", case_name, m_lambda, n), 'wb') as file:
        return np.save(file, kernel)


def print_predict(case_name, x_test, y_predict):
    status = get_status(case_name, y_predict)
    for i in range(len(x_test)):
        print(x_test[i], " ", status[i], "\n")


def get_status(case_name, y_predict):
    status = []
    for i in range(len(y_predict)):
        status.append(C.cases[case_name][0 if y_predict[i] == -1 else 1])
    return status


def read_array(case_name, m_lambda, n, purpose):
    with open(get_model_file_name(
            "kernel_" + purpose + "_", case_name, m_lambda, n), 'rb') as file:
        kernel_array = np.load(file)
        return kernel_array, normalized_array(kernel_array)


def read_array_test(case_name, m_lambda, n, purpose):
    with open(get_model_file_name(
            "kernel_" + purpose + "_", case_name, m_lambda, n), 'rb') as file:
        kernel_array = np.load(file)
        return kernel_array, np.array([0])


def normalized_array(mat_build):
    mat_x = np.diag(
        mat_build).reshape((mat_build[0].shape[0], 1))
    return np.divide(mat_build, np.sqrt(mat_x.T * mat_x))


def get_kernels(case_name):
    return iterate_cases(case_name, read_array)


def get_models(case_name):
    return iterate_cases(case_name, read_model)


def iterate_cases(case_name, func):
    result = []
    for m_lambda in C.m_lambdas:
        for n in C.NValues:
            result.append(func(case_name, m_lambda, n))
    return result


def different_parameters(case_name, func, purpose):
```

```
    for m_lambda in C.m_lambdas:
        for n in C.NValues:
            func(case_name, m_lambda, n, purpose)


def print_reports(case_name, y_actual, y_predict, m_lambda, n, suffix):
    reports = metrics.classification_report(
        y_actual, y_predict, target_names=C.cases[case_name].values())
    print_result(case_name, m_lambda, n, reports, y_predict, suffix)


def print_reports_send(
        case_name, y_actual, y_predict, m_lambda, n, suffix):
    return metrics.classification_report(
        y_actual, y_predict, target_names=C.cases[case_name].values())


def get_file(case_name, purpose):
    if purpose == "input":
        file_name1 = C.input_folder + case_name + C.inputFile_0
        file_name2 = C.input_folder + case_name + C.inputFile_1
    else:
        file_name1 = C.test_folder + case_name + C.testFile_0
        file_name2 = C.test_folder + case_name + C.testFile_1
    return file_name1, file_name2


def get_file_x(case_name, purpose):
    if purpose == "input":
        file_name = C.input_folder + case_name + C.inputFile
    else:
        file_name = C.test_folder + C.testFile
    return file_name


def get_data(case_name, purpose):
    file_name1, file_name2 = get_file(case_name, purpose)

    csv_reader = CsvReader(file_name1)
    x = csv_reader.get_text_data()
    y = np.array([-1 for _ in range(x.shape[0])])

    csv_reader = CsvReader(file_name2)
    x1 = csv_reader.get_text_data()
    x = np.append(x, x1)
    y = np.append(y, np.array([1 for _ in range(x1.shape[0])]))
    length = x.shape[0]
    return x.reshape(length, 1), y

def get_data_test_new(case_name, purpose):
    file_name1, file_name2 = get_file(case_name, purpose)

    csv_reader = CsvReader(file_name1)
    x = csv_reader.get_text_data()
    y = np.array([-1 for _ in range(x.shape[0])])

    csv_reader = CsvReader(file_name2)
    x1 = csv_reader.get_text_data()
    x = np.append(x, x1)
    y = np.append(y, np.array([1 for _ in range(x1.shape[0])]))
    length = x.shape[0]
    return x.reshape(length, 1), y


def get_data_test(case_name, purpose, i):
    file_name1, file_name2 = get_file(case_name, purpose)
```

```
    csv_reader = CsvReader(file_name1)
    x = csv_reader.get_text_data()[i * 50:(i + 1) * 50]
    y = np.array([-1 for _ in range(x.shape[0])])

    csv_reader = CsvReader(file_name2)
    x1 = csv_reader.get_text_data()[i * 50:(i + 1) * 50]
    x = np.append(x, x1)
    y = np.append(y, np.array([1 for _ in range(x1.shape[0])]))
    length = x.shape[0]
    return x.reshape(length, 1), y


def get_data_x(case_name, purpose):
    file_name = get_file_x(case_name, purpose)
    csv_reader = CsvReader(file_name)
    x = csv_reader.get_text_data()
    length = x.shape[0]
    return x.reshape(length, 1)


def read_argument(argv):
    case_name = argv[0]
    m_lambda = float(argv[1])
    n = int(argv[2])
    return case_name, m_lambda, n
```

TextUtil: Text processing utility

```
from TextProcessing.CsvReader import CsvReader
from TextProcessing.CsvWriter import CsvWriter
from Constant import base_folder

csvReader = CsvReader(base_folder + "text")

text_line = []
final_line = []
k = 0
for line in csvReader.get_lines():
    k += 1
    text_line.append(line.strip())
    if k % 4 == 0:
        k = 0
        final_line.append(" ".join(text_line))
        text_line=[]

csvWriter = CsvWriter(base_folder + "text_modify")
csvWriter.append_to_file("\n".join(final_line))
print(final_line)
```

MetricsUtil: Metrics utility

```
from Util import Util as util
import glob
from TextProcessing.CsvReader import CsvReader
import numpy as np
from Model.SVMModel import SVMModel
from Util.KernelUtil import KernelUtil
import Constant as C


def get_input_result_files(case_name):
    return glob.glob(util.get_result_folder(case_name) + "**input")


def get_test_result_files(case_name):
```

```
    return glob.glob(util.get_result_folder(case_name) + "**test_?")


def get_header_metrics(texts):
    ns = []
    m_lambda = texts[0].split(" ")[1].split(":")[1]
    for text in texts:
        header_array = text.split(" ")
        n = header_array[2].split(":")[1]
        ns.append(int(n.strip()))
    return m_lambda, ns


def get_precision_row(texts):
    precisions = []
    recalls = []
    f1s = []
    support = []
    for text in texts:
        precision_array = text.strip().split(" ")
        precisions.append(float(precision_array[1].strip()))
        recalls.append(float(precision_array[2].strip()))
        f1s.append(float(precision_array[3].strip()))
        support.append(int(precision_array[4].strip()))
    return precisions[0], recalls[0], f1s[0], support[0]


def gt_f1_row(texts):
    f1s = []
    for text in texts:
        precision_array = text.split("  ")
        f1s.append(float(precision_array[4].strip()))
    return f1s


def get_metrics_data(case_name, purpose):
    if purpose == "input":
        files_input = get_input_result_files(case_name)
    else:
        files_input = [util.get_result_file_name(case_name, C.m_lambda, purpose)]
    metrics = []
    for f in files_input:
        reader = CsvReader(f)
        texts = reader.get_lines()
        m_lambda, ns = get_header_metrics(texts[::9])

        precision1, recall1, f1_1, support1 = get_precision_row(texts[3::9])
        precision2, recall2, f1_2, support2 = get_precision_row(texts[4::9])
        svm = SVMModel(case_name)
        support_count = svm.get_support_vector_count()
        metric = {
            "m_lambda": m_lambda,
            "ns": ns,
            "Data1": support1,
            "Data2": support2,
            "precision1": precision1,
            "precision2": precision2,
            "recall1": recall1,
            "recall2": recall2,
            "f1_1": f1_1,
            "f1_2": f1_2,
            "support1": support_count[0],
            "support2": support_count[1]
        }
        metrics.append(metric)
    return sorted(metrics, key=lambda k: k['m_lambda'])
```

```
def get_distance_f1_score(case_name, purpose):
    x, y_actual = util.get_data(case_name, purpose)
    k = get_kernel(case_name, purpose)
    distance, predict = get_distance_predict(
        case_name, purpose)
    positive = distance[y_actual > 0]
    negative = distance[y_actual < 0]

    true_positive = np.sum(positive[positive > 0])
    false_negative = -np.sum(positive[positive < 0])
    false_positive = np.sum(negative[negative > 0])
    f1_1 = (2 * true_positive) / (
           2 * true_positive + false_positive + false_negative)

    true_positive = -np.sum(negative[negative < 0])
    false_negative = np.sum(negative[negative > 0])
    false_positive = -np.sum(positive[positive < 0])
    f1_2 = (2 * true_positive) / (
           2 * true_positive + false_positive + false_negative)
    return f1_1, f1_2


def get_mean(case_name, purpose):
    distance, predict = get_distance_predict(case_name, purpose)
    x, y_actual = util.get_data(case_name, purpose)
    positive = distance[y_actual > 0]
    negative = distance[y_actual < 0]
    return np.mean(positive[positive > 0]
                ), -np.mean(negative[negative < 0])


def get_mean_ratio(case_name, purpose):
    x, y_actual = util.get_data(case_name, purpose)
    positive, negative = get_classes_distance(y_actual, case_name, purpose)
    m1, m2 = np.mean(positive), -np.mean(negative)
    distance, prediction = get_distance_predict(case_name, purpose)
    return np.divide(distance[y_actual > 0], m1
                  ), np.divide(distance[y_actual < 0], m2)


def get_classes_distance(y_actual, case_name, purpose):
    distance, predict = get_distance_predict(case_name, purpose)
    positive = distance[y_actual > 0]
    negative = distance[y_actual < 0]
    return positive[positive > 0], negative[negative < 0]


def get_distance_predict(case_name, purpose):
    k = get_kernel(case_name, purpose)
    svm = SVMModel(case_name)
    return svm.decision_function_local(k), svm.prediction_local(k)


def get_kernel(case_name, purpose):
    kernel_util = KernelUtil(case_name, purpose)
    kernel = kernel_util.get_kernel()
    return kernel
```

KernelUtil: Kernel utility

```
import Util.Util as Util
import Constant as C
```

```
class KernelUtil:
    def __init__(self, case_name, purpose):
        self.case_name = case_name
        if purpose == "input":
            clf = Util.read_model(case_name, C.m_lambda, C.N)
            self.kernel = Util.read_array(
                case_name, C.m_lambda, C.N, purpose)[1][clf.support_]
        else:
            self.kernel = Util.read_array_test(
                case_name, C.m_lambda, C.N, purpose)[0]

    def get_kernel(self):
        return self.kernel
```

Sub SequenceString Kernel: Computation of Gramian matrix of the subnsequence kernel

```
# coding: utf-8
import numpy as np
import multiprocessing as mp
import Util.Util as Util


class SubsequenceStringKernel:

    def __init__(self, N, m_lambda, X, Y):
        self.X = X
        self.Y = Y
        self.m_lambda = m_lambda
        self.N = N
        self.mat_build = np.zeros((X.shape[0], Y.shape[0]))

    def compute(self, s, t, s1=0, t1=0):
        N = self.N
        s = s.strip()
        t = t.strip()
        lengthOfs = len(s)
        lengthOft = len(t)

        Kp = [[[0 for col in range(lengthOft)]
               for row in range(lengthOfs)]
              for x in range(N + 1)]

        for i in range(lengthOfs):
            for j in range(lengthOft):
                Kp[0][i][j] = 1.0

        lmbda = self.m_lambda
        S = s
        T = t
        for n in range(N):
            for i in range(lengthOfs - 1):
                Kpp_n = 0.0
                for j in range(lengthOft - 1):
                    Kpp_n = lmbda * (Kpp_n + lmbda * (S[i] == T[j]) * Kp[n][i][j])
                    Kp[n + 1][i + 1][j + 1] = lmbda * Kp[n + 1][i][j + 1] + Kpp_n
        K = 0.0
        for n in range(N):
            for i in range(lengthOfs):
                for j in range(lengthOft):
                    K += lmbda * lmbda * (S[i] == T[j]) * Kp[n][i][j]
        return K

    def get_kernel_mat(self, func):
        X = self.X
        Y = self.Y
        lenX, lenY = X.shape[0], Y.shape[0]
```

```
    pool = mp.Pool(mp.cpu_count())
    results = pool.starmap(self.compute, [(X[i, 0], Y[j, 0])
                                          for i in range(lenX) for j in range(lenY)])
    pool.close()
    pool.join()
    k = 0
    return np.array(results).reshape((lenX, lenY))

def get_kernel(self, case_name, support_):
    X, Y = self.X, self.Y
    lenX, lenY = X.shape[0], Y.shape[0]
    mat = self.get_kernel_mat()
    pool = mp.Pool(mp.cpu_count())
    mat_Y = pool.starmap(self.compute, [(Y[i, 0], Y[i, 0]) for i in range(lenY)])
    pool.close()
    pool.join()

    mat_X = self.get_diagonal(case_name)
    mat_X = mat_X[support_].reshape((lenX, 1))
    mat_Y = np.array(mat_Y).reshape((lenY, 1))
    return np.divide(mat, np.sqrt(mat_Y.T * mat_X))

def get_diagonal(self, case_name):
    mat_build, normalized_mat_build = \
        Util.read_array(case_name, self.m_lambda, self.N, "input")
    return np.diag(mat_build)

def getKernel(self):
    X, Y = self.X, self.Y
    lenX, lenY = X.shape[0], Y.shape[0]

    mat = np.zeros((lenX, lenY))

    for i in range(lenX):
        for j in range(lenY):
            val = self.compute(X[i, 0], Y[j, 0])
            mat[i, j] = val

    mat_X = np.zeros((lenX, 1))
    mat_Y = np.zeros((lenY, 1))

    for i in range(lenX):
        mat_X[i] = self.compute(X[i, 0], X[i, 0])
    for j in range(lenY):
        mat_Y[j] = self.compute(Y[j, 0], Y[j, 0])
    return np.divide(mat, np.sqrt(mat_Y.T * mat_X))

def build_kernel_mat(self, case_name):
    x = self.X
    len_x = x.shape[0]
    pool = mp.Pool(mp.cpu_count())
    results = pool.starmap(self.compute, [(x[i, 0], x[j, 0], i, j)
                                          for i in range(len_x) for j in range(len_x)[i:]])
    pool.close()
    pool.join()
    k = 0
    for i in range(len_x):
        for j in range(len_x)[i:]:
            self.mat_build[i][j] = results[k]
            self.mat_build[j][i] = results[k]
            k += 1
    Util.print_array(case_name, self.m_lambda, self.N, self.mat_build, "input")

def build_kernel(self, case_name):
    self.build_kernel_mat(case_name)
```

```
        return Util.normalized_array(self.mat_build)
```

ModelBuilder: Build SVM model from the Gramian of kernel

```
# coding: utf-8
from Kernel.SubsequenceStringKernel import SubsequenceStringKernel
from sklearn import svm
import sys
import Util.Util as Util
import numpy as np


def build_model(case_name, m_lambda, n, purpose):
    print("started")
    x, y = Util.get_data(case_name, purpose)
    ssk = SubsequenceStringKernel(n, m_lambda, x, x)
    clf = svm.SVC(kernel='precomputed')
    k = ssk.build_kernel(case_name)
    clf.fit(k, y)
    y_predict = clf.predict(k)
    print(y_predict)
    Util.print_model(case_name, m_lambda, n, clf)
    Util.print_reports(
        case_name, y, y_predict, m_lambda, n, purpose)


if __name__ == "__main__":
    Util.different_parameters(sys.argv[1], build_model, "input")
```

TestModel: Test SVM model and analyze the performance

```
from Kernel.SubsequenceStringKernel import SubsequenceStringKernel
import Util.Util as Util
import sys
import Constant as C
from Model.SVMModel import SVMModel


def test_model(case_name, m_lambda, n, purpose, flag, i):
    clf = Util.read_model(case_name, m_lambda, n)

    if flag:
        x_test = Util.get_data_x(case_name, purpose)
    else:
        x_test, y = Util.get_data_test(case_name, purpose, i)

    x, y1 = Util.get_data(case_name, "input")
    ssk = SubsequenceStringKernel(
        n, m_lambda, x[clf.support_], x_test)
    k = ssk.get_kernel(case_name, clf.support_)

    svm = SVMModel(case_name)

    y_predict = svm.prediction_local(k)
    Util.print_predict(case_name, x_test, y_predict)
    if not flag:
        Util.print_array(
            case_name, m_lambda, n, k, purpose + "_" + str(i))
        Util.print_reports(
            case_name, y, y_predict, m_lambda, n, purpose + "_" + str(i))


if __name__ == "__main__":
    case_name = sys.argv[1]
    flag = len(sys.argv) > 2
    for i in range(1):
```

```
    for m_lambda in C.m_lambdas:
        for n in C.NValues:
            test_model(case_name, m_lambda, n, "test", flag, i)
```

REFERENCES

[1] N. Aronszajn, *Theory of reproducing kernels*, Transactions of the American Mathematical Society **68** (1950), no. 3, 337–404.

[2] Christopher M. Bishop, *Pattern recognition and machine learning (information science and statistics)*, 1 ed., Springer, 2007.

[3] William Karush, *Minima of functions of several variables with inequalities as side conditions*, Master's thesis, Department of Mathematics, University of Chicago, Chicago, IL, USA, 1939.

[4] Lodhi, Saunders, Shawe-Taylor, Cristianini, and Watkins, *Text classification using string kernels*, J. Mach. Learn. Res. **2** (2002), 419–444.

[5] E. H. Moore, *General analysis*, 2, vol. 1, The MIT Press, 1939.

[6] Vern I. Paulsen and Mrinal Raghupathi, *An introduction to the theory of reproducing kernel Hilbert spaces*, Cambridge Studies in Advanced Mathematics, vol. 152, Cambridge University Press, Cambridge, 2016.

[7] Bernhard Schölkopf and Alexander J. Smola, *Learning with kernels : support vector machines, regularization, optimization, and beyond*, Adaptive computation and machine learning, MIT Press, 2002.

[8] Morton Slater, *Lagrange multipliers revisited*, pp. 293–306, Springer Basel, Basel, 2014.

[9] Vladimir N. Vapnik, *Statistical learning theory*, Wiley-Interscience, 1998.