**Extended Essay on Computer Science:**

Artificial Intelligence (AI), Machine Learning (ML), Tuning a Support Vector Machine (SVM)

**Research Question:**

What are the best kernels and the optimal hardness of an SVM for target functions of different complexities and with varying levels of noise and number of training examples?

**Table of Contents**

1. Introduction………………………………………………………………………………………………………………… 3

2. Theoretical Investigation……………………………………………………………………………………………… 3

3. Research Question………………………………………………………………………………………………………. 7

4. Hypothesis…………………………………………………………………………………………………………………… 8

5. Experiment Description……………………………………………………………………………………………… 8

6. Recording the Results……………………………………….…………………………………………………………. 12

7. Results and Analysis of Results………………………………………………………………………………………14

8. Summing up the Results……………………………………………………………………………………………… 26

9. Conclusion…………………………………………………………………………………………………………………… 26

10.Further Research………………………………………………………………………………………………………… 27

References……………………………………………………………………………………………………………………… 28

Appendix A - The main program of the experiment…………………………………………………………. 28

Appendix B - Code to visualize the results……………………………………………………………………… 34

1. **Introduction**

My father walked up to me looking rather furious and confused. “I have been trying to get an email from a patient for an hour without success; he has been sending and resending it to me, but it never reaches me. What is wrong?”. Nothing was wrong, the client’s emails were automatically placed in my father’s spam folder. Once I explained that to him, his bewilderment was not eased. I showed him all his spam emails and told him that his host decided automatically which emails should be placed in his inbox and which ones should be considered spam. “And how does it know which emails I want to see?”. That’s a bit harder to answer dad, let me try to find an answer and come back to you…

**Support Vector Machines**

After hours of searching, I came across Support Vector Machines. These are a cutting-edge algorithm in Machine Learning. They are arguably the most widely used and best performing classifiers. Since their performance heavily depends on their hardness and the Kernels used, the optimum of which varies for different data sets, it is crucial to understand how these should be chosen. This essay deals with this issue and relates to the 4th topic of the Computer Science Guide, namely “Computational thinking, problem-solving and programming”.

1. **Theoretical Investigation**

According to Google Developers[[1]](#footnote-1), machine learning is a field of artificial intelligence that studies programs or systems which use input data to create a model that can “make useful predictions on new data drawn from the same distribution”1 as the training data. For a model to be trained, it is essential that the input data form a pattern, whether easily visible or not, meaning that they are not randomized.

Supervised machine learning, which is studied in this essay, refers to, according to the same source, “training a model from input data and its corresponding labels”1, where labels means the result of each data input, whether this is which category it belongs in, or a real value that should be produced as a result of the specific input.

Binary classification, which is the task accomplished by support vector machines, is a type of machine learning model for distinguishing between strictly two discrete and mutually exclusive classes 1.

The emailing example may help clarify what falls under the category of supervised machine learning and binary classification. Assume that we have data on a million different emails sent in the past, such as the sender’s host, the number of recipients, the number of appearances of some keywords etc., as well as whether they were considered by the user as spam or not. Now, if we want to make predictions on whether future emails are spam, a binary classifier comes in handy, since this is a binary event, and for every past email we have the corresponding label, -1 if it wasn’t considered spam or 1 if it was.

Most classifiers would deal with this task taking the following approach:

1. Create N d-dimensional vectors x1,x2,…,xN, one for each example, where N is the number of examples (here 1 million) and d the number of features available for each example, and N scalars y1,y2,…,yN, with value either +1 or -1 corresponding to the labels of each example.

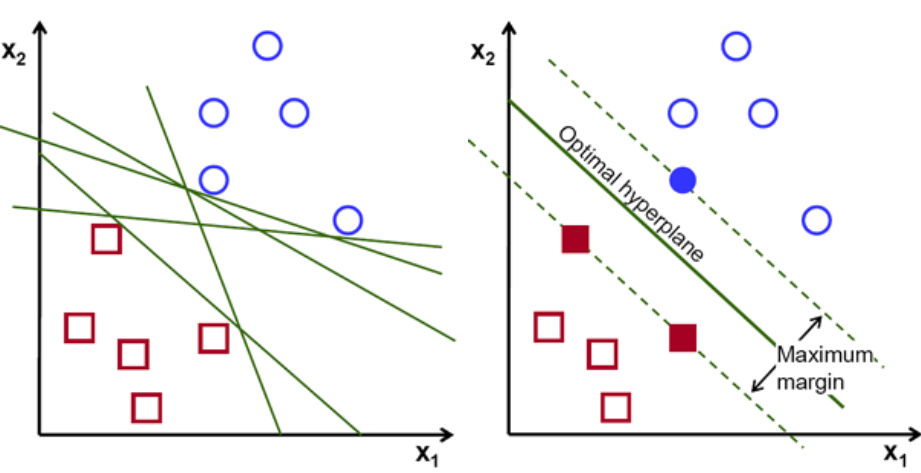
2. Use an algorithm to find a d-dimensional vector of weights. This vector w should satisfy sign(w∙xi)=yi for every i ranging from 1 to N.

3. To predict the result of new data simply take the sign of the dot product w∙xnew.[[2]](#footnote-2)

This, however, is only possible if the vectors are linearly separable in the d-dimensional space. Otherwise, any non-linear transformation Φ(x) can be used to transform all vectors into new ones, in a higher dimensional space, z, where the vectors do become linearly separable. Then, the learning algorithm can produce a vector w in the z space, that satisfies sign(w∙Φ(xi))=yi for every i ranging from 1 to N, and the prediction for new data can be acquired by taking sign(w∙Φ(xnew)).

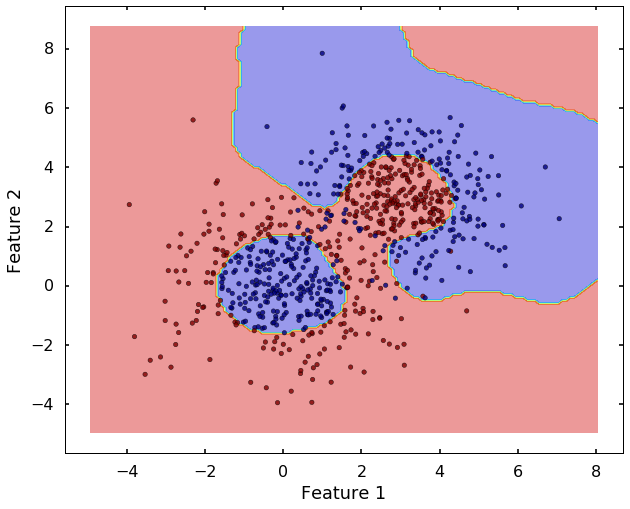
Going into a higher dimensional space, allows for separating all the data points input, producing zero-error in the training data, also called in-sample error, Ein. However, it harms the model’s generalization theory2, meaning that the model might perform worse when dealing with new data. This is said to be a higher out-of-sample error, Eout. Thus, sometimes it is preferable to allow for some in-sample error so as to avoid visiting the z space and to minimize the out-of-sample error, which is the goal of training. After all, we want the machinery to perform well on never-before-seen data, not on the examples it trained with.

Support Vector Machines, however, introduced a novelty: they don’t only try to separate the data, but the separating line, plane or hyperplane, depending on how large d is, should have the maximum possible margin from the training points, i.e. the maximum distance to the closest points, as depicted in Figure 1. The points that achieve exactly the margin are called support vectors. Thus, it can mathematically be proved, that the generalization of SVMs does not depend on the dimensionality to which they map the points, but it is only affected by the number of support vectors. Then, Eout is statistically less than (number of support vectors/N) with a great degree of certainty. The space z to which the points are mapped can be very large and we don’t even need to know its dimensionality. This allows for what is known as the Kernel trick. Instead of transforming each vector to z, we can use a set of functions known as Kernels, that given two vectors x1 and x2, they return a single number which is guaranteed to be the dot product of the two vectors in some space z. This is all that SVMs need to separate the points and maximize the margin in the z-space. How this might look going back to the d-dimensional space can be seen in Figure 2.

**Figure 1: Possible outcomes of a simple classifier vs outcome of SVM in linearly separable case[[3]](#footnote-3)**

The final notion that I would like to introduce, is noise. Noise refers to any distortion in the given data. It can occur for a variety of reasons, the most common of which is inaccuracy when measuring the data. In the example given above, noise could be produced for example by the fact that the sender of an email might hide the number of recipients or send the email at more recipients at different times, thus not making them visible. Rarely can we find completely noiseless data sets.

Noise and extreme outliers, which are data points off pattern, might bring about the need for some error allowance in SVMs as well. Intuitively, this means that in trying to correctly classify such points, the SVM might pass closer to some data points that used to be further away from the separating plane, leading to more support vectors and worse generalization. The smaller the tuning parameter C is, the greater the error allowance and thus the softness of the SVM.

**Figure 2: SVM non-separable case, return from z-space with error-allowance***[[4]](#footnote-4)*

1. **Research Question**

If we use target functions of varying complexities, with different levels of noise and with a different number of training examples, the kernel and the parameter C that will produce the minimum Eout will not be the same in all cases. The resulting research question is “What are the best kernels and the optimal hardness of an SVM for target functions of different complexities and with varying levels of noise and number of training examples?”

1. **Hypothesis**

Based on the theoretical investigation and the intuition that I have developed around the topic, I assume that Eout is minimum when the kernel used maps the points to a dimension as close as possible to the target’s function complexity, or maybe even a higher dimension if there is enough training data. The reason behind this thought, is that kernels of lower complexity than the target function will always underfit the data; they will not be able to catch the twists and turns of the target function. On the other hand, kernels of too high a complexity may overfit the data, trying to reach the outliers, especially if there are not enough examples to prove that such points are indeed outliers.

A larger amount of training examples should generally improve performance in all SVMs, except maybe the ones using naïve kernels which after a point can’t get better no matter what the number of training examples. This is because more examples means elimination of noise to a degree.

Moreover, it follows to logic that higher noise levels in the training data means higher Eout since the points will be more distorted and thus the algorithm won’t be achieving the same results.

As far as optimal C is concerned, I believe it to be solely dependent on the noise levels of a data set, and I believe the relationship between the two to be a negatively sloped relationship, maybe even an inverse-relationship.

1. **Experiment Description**

The experiment aims to find how SVMs of different hardness levels and kernels, which dictate the model’s complexity, perform with different data sets, noise levels and number of training examples. In order to directly evaluate Eout rather than using generalization theory, as well as to dictate the target function, i.e. the one that yields Eout=0, and to know its complexity, I will be creating my own target functions, polynomial functions of up to 3rd degree, combinations of multiple polynomial functions, either different functions in different ranges, or logical combinations of the output of multiple polynomial functions, and a very complex trigonometric function. All the training examples that will be generated will have two features x1 and x2. This will keep the time needed to run the experiment low, without affecting the nature of the observations which hold true for vectors of any dimension.

At first a variable NOISE will be initialized to 0. It will dictate the maximum percentage of noise that will be added on the training points during each run.

A variable N, which will dictate the number of training examples, will be set to 10.

A low order polynomial function t(x1, x2) will be used as the target function to be defined by the SVM. The more turns the function t has, the greater the complexity of the target function. Then, N data points x1,x2,…,xN will be generated with random coefficients (xi1,xi2) within the range [-0.5,0.5], since skicit[[5]](#footnote-5) anyway advises applying a function both to the training points and the new data to restrict their range within [-1,1](added noise, the new max range will be the aforementioned one). Each point xi will be plugged into t and compared to 0. If it is greater, then the corresponding label yi will be given the value +1. Otherwise, yi will take the value -1. After the point’s labelling, two random numbers r1 and r2 will be generated in the range [-NOISE, -NOISE/2]∪[NOISE/2, NOISE]. The point’s first coordinate will be multiplied by 1+r1, while the second one by 1+r2. This way, noise proportional to the desired noise level will be added to every point. At the same time, the expected value of the generated noise will be 0 since the range used is symmetric around 0 and the probability of any value in this range being chosen is equal. In other words, if the process is repeated a great number of times, we expect the total noise of all points to have 0 as its average, making the SVM’s work easier and more reasonable, since even if there are fluctuations in the coefficients, they will not be towards a certain direction that would deliberately trick the algorithm.

Having generated N points with their labels and added noise, we can now check the performance of different SVMs in trying to predict the target function. The SVMs used will have one of the following kernels:

* Linear Kernel, which given two vectors x, x’ returns their inner product x∙x’.
* Polynomial Kernel, which takes the form (x∙x’+c) d, where d determines the dimension reached and c will be fixed at 1. The values used for d will be {2,3,5,10,100}, since the first three will show us the results of three similar but not identical, low complexity kernels, the value of 10 serves as an intermediate complexity kernel, and the value 100 will return from an immensely large z space.

The RBF kernel, which returns exp(-γ||x-x’||2), where γ is a positive real-valued number, is the most famous kernel used in SVMs since it maps points to an infinite dimension. However, due to time limitations of the experiment, it is not used, because it requires optimizing for both parameters C and γ. The polynomial kernel with d=100 however maps the points in a very high dimension itself and there is no need in this experiment to go beyond this.

Ten-fold cross validation will be used in every case to determine the optimum C based in the training set out of {10-4, 5\*10-4, 10-3,…,102, 5\*102}, which exponentially spans a very large range for C, from the practical minimum in most cases to the practical maximum.

Each SVM’s Eout will be calculated by summing the number of misclassified points over a total of more than 2500 points distributed in the space and dividing by the number of points. If the SVM fails to complete within a certain time limit, it will be considered failed and Eout will be set to 1. This is a necessity considering the vast time the algorithm needs to run (see Table 2).

The exact same experiment will be repeated with new data points generated from the same target function 5 times.

N will be increased by 20 and the same experiment will be repeated with the same target function, until N becomes 1000.

Then N will be set back to 10 and NOISE will be increased by 0.1. The same experiment will be repeated with the same target function, until NOISE becomes 1.

Then NOISE will be set back to 0. A new target function will be used, and the experiment will be repeated until all predefined target functions have been tested.

The 5 repetitions, the increments of 0.1 for noise and 20 for N, are a compromise between time efficiency and sufficient representation of results.

Maximum noise is set to 1, since at that point, each data point will have its value possibly utterly distorted.

Maximum N is set at 1000, because out of around 2500 points that will be evaluated in the end, a thousand constitutes a large enough to be completely representative sample.

All the results will be printed in an excel file.

I personally wrote the code, which can be found in Appendix A, for the above algorithm, the variables of which are clearly pointed out in Table 1, in python, using the sklearn module developed by scikit (0 is used as a label instead of -1 in this library) and other external libraries found in Table 2.

**Table 1: The experiment’s variables**

|  |  |
| --- | --- |
| **Independent Variables** | Target function |
| NOISE |
| N (number of examples) |
| **Dependent Variables** | Eout |
| Optimum Kernel (affects SVM’s complexity) |
| Optimum parameters C |

|  |  |
| --- | --- |
| **CPU** | AMD Ryzen 5 1600X Six-Core processor, 3.6GHz, 6 Cores, 12 logical processors |
| **GPU** | Nvidia GeForce GT 730 |
| **Operating System** | Windows 10 |
| **Installed, required software** | Microsoft Office Excel, python interpreter |
| **Libraries needed** | Random[[6]](#footnote-6), skicit-learn[[7]](#footnote-7), numpy[[8]](#footnote-8), matplotlib[[9]](#footnote-9), sys[[10]](#footnote-10), os[[11]](#footnote-11), openpyxl[[12]](#footnote-12), math[[13]](#footnote-13), warnings[[14]](#footnote-14), pandas[[15]](#footnote-15), seaborn[[16]](#footnote-16) |
| **Experiment’s run time** | Approximately 17d in total |
| **Run time of code to visualize results** | Dependent on user interaction |

**Table 2: Specifications of the computer the code was run on and code-relevant information**

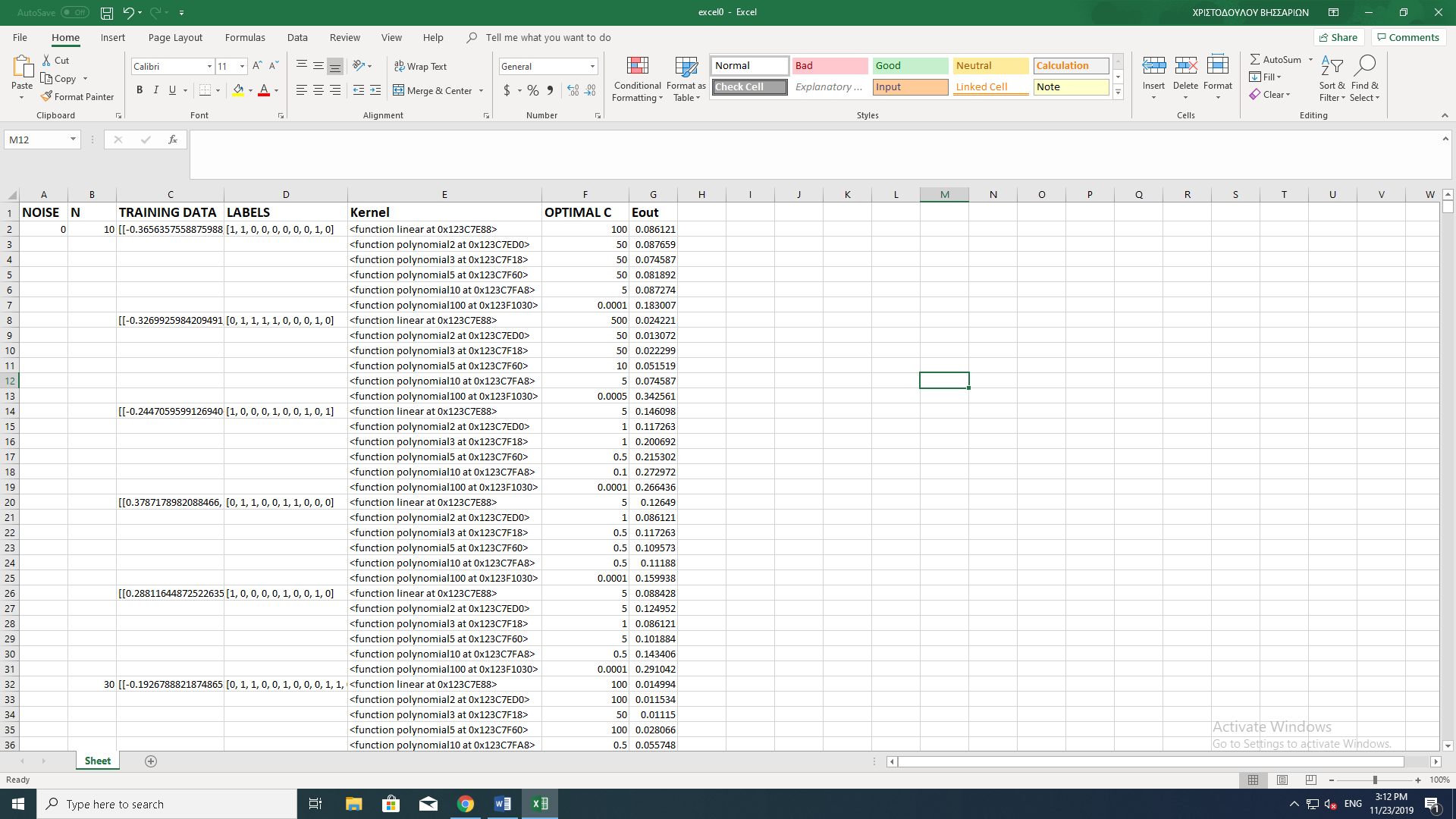
**6. Recording the Results**

The results of the experiment are saved in seven different excel workbooks, named excel0 to excel6, each one representing one of the seven target functions. Each excel workbook has (11\*50\*5\*6)=16500 rows:

* 11 is the number of different noise levels tested, in the range [0,1] with step size equal to 0.1
* 50 is the amount of all the different numbers of examples used, ie different Ns.
* 5 is the times the experiment will be repeated with the same N and noise levels
* 6 is the numbers of kernels tested, for each one of which the optimum C and the out-of-sample error are going to be recorded

As for columns, each workbook contains 7, representing NOISE, N, the generated training points, the corresponding labels, the kernel tested, the optimal C and the out-of-sample error. The first few lines of such an excel file can be seen right below (column titles added to make data comprehensive):

**Figure 3 (close-up): Raw representation of results as they are automatically saved in an excel file**



As it is obvious, the experiment’s raw data is meaningless, for analyzing thousands of rows and columns by hand is practically impossible. Therefore, I wrote a second, smaller piece of program, available in Appendix B, to make analysis and interpretation of the results plausible. Essentially, what it does is, given an excel workbook as the one above, it calculates the average optimum C and the average out-of-sample error achieved with each kernel, firstly for different Ns regardless of noise levels and then for different noise levels regardless of N, and plots them. It does not take into consideration values of Eout equal to 1, i.e. if the time limit was exceeded for some SVM results could have been drawn should there be more time available.

**7. Results and Analysis of Results**

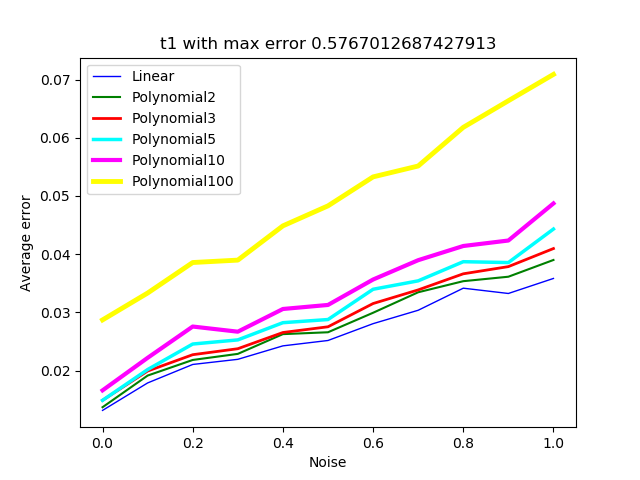
In the following graphs, t1 to t7 refer to the seven different target functions used in order of increasing complexity. Specifically, t1 is a constant function, t2 is a linear function, t3 is a quadratic function, t4 is a cubic function, t5 is a combination of t2, t3 and t4 in different ranges, t6 is a logical combination of t4 and a circle and t7 is a very complex trigonometric function.

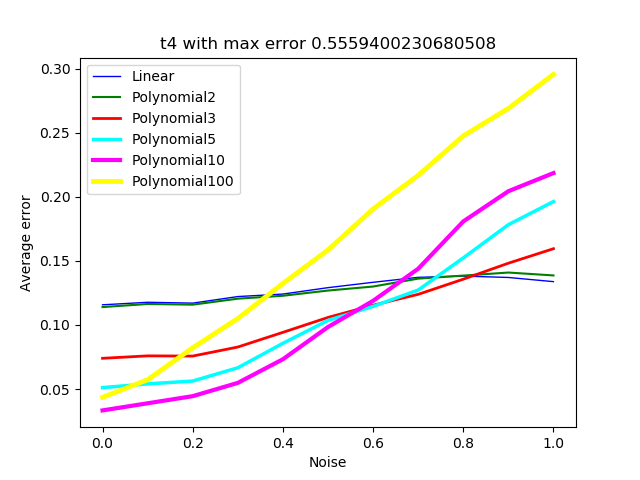
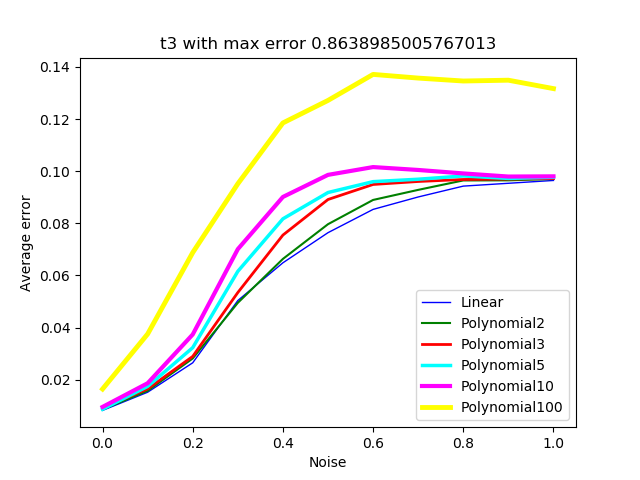
Moreover, the maximum error of the target functions is mentioned in the titles of the graphs of the first two subchapters. I mean this to be the percentage of the most frequent binary result over all points checked. For example, if a target function classified 60% of points as 1 and 40% of points as 0, the maximum out-of-sample error allowed would be 0.6, if the classifier always returned 1. It is worth noting that t7 is such a complex function that more than 4% of points are roots to its equation. This explains the maximum error being less than 0.5.

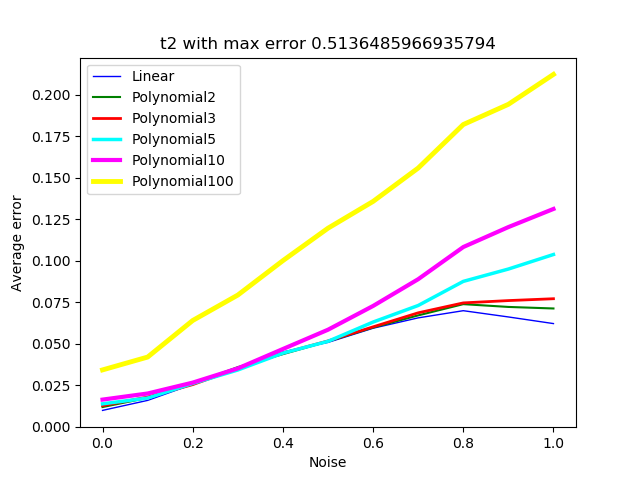
In all graphs, the relation between the quantities of interest are different for different kernels, thus all the curves correspoding to the 6 kernels are depicted in each graph, with different colors and line widths.

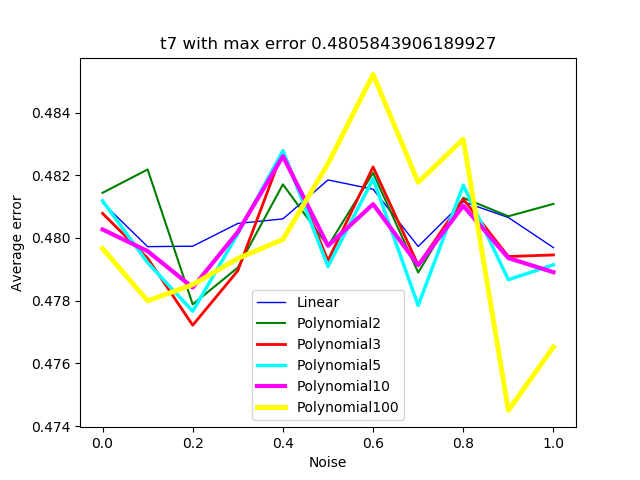
**Eout, kernels and noise levels**

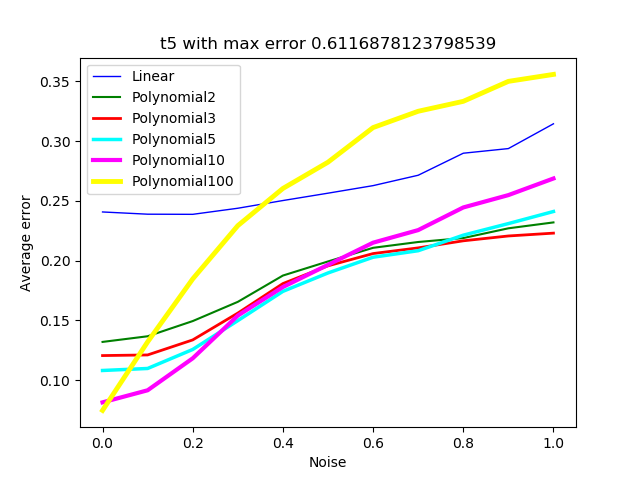
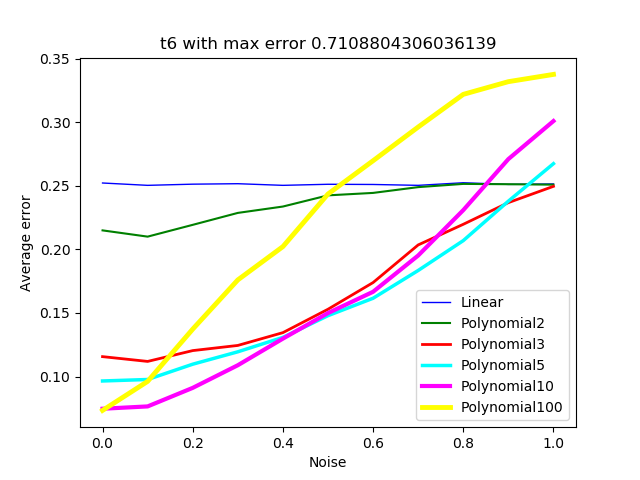
The following 7 graphs plot the average out-of-sample error over noise levels for all kernels. Each one refers to a different target function.

****

****

****

****

****

The first point to make is that all graphs depicting the relationship between the out-of-sample error and noise are positively sloped. This means that, as expected, the higher the noise levels, the more distorted the data and the more difficult it is for the SVM to produce the right results.

It is interesting that in target functions of low max error the relationship between the two quantities seems to be almost linear, completely proportional, while in cases where the target function has a greater max error the curve tends to be logarithmic. This happens because in functions with a bigger max error, the points need much less noise to change labels, essentially to move from the smallest semi-plane defined by the function to the bigger one, as they are much closer to the separating line. Thus, even small distortions can produce devastating results, whereas the difference between medium noise levels and higher ones are not that drastic.

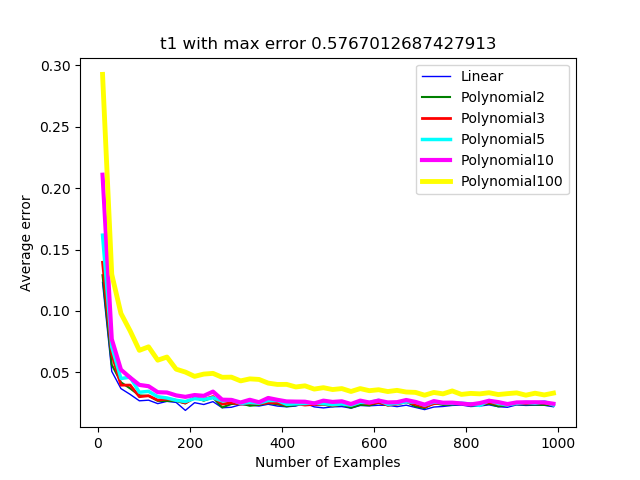
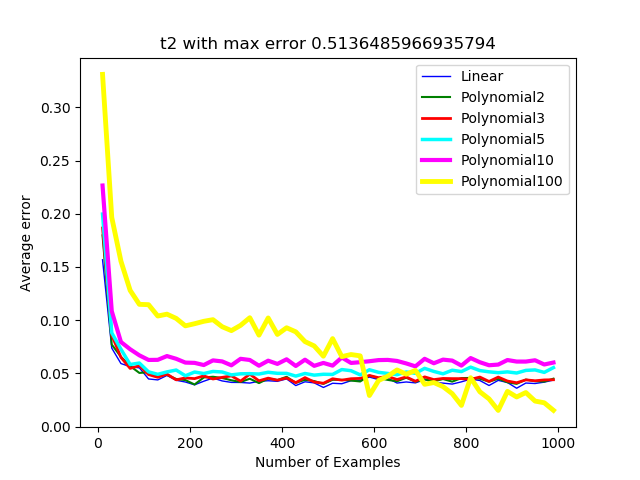
t7 is the only target function to which the positive relationship between noise levels and out-of-sample error does not apply. The reason is that this function is so complex that points move sides extremely quickly, even with very low levels of noise. Except for looking correct intuitively, this argument is enhanced by the fact that almost 5% of the points in the ([-1,1]x[-1,1]) region lie exactly on the separating curve. As such, higher levels of noise do not inflict any more damage at all than lower levels. Thus, just in this case, there is no dependence between noise levels and Eout.

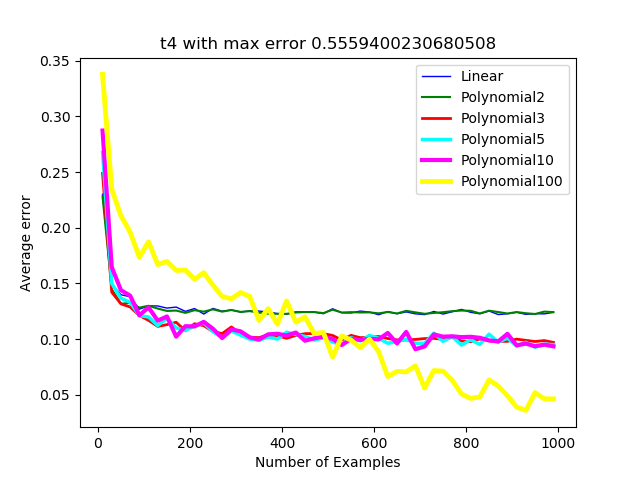
Another common element among the graphs Eout-Noise levels, is that the polynomial with d=100 performs significantly worse than the other kernels as noise levels increase. The reason why this happens, is that the very complex polynomial kernel overfits the examples, by trying to correctly classify the noisy data. On the contrary, the other kernels leave the very noisy data misclassified, since they do not have the ability to reach out for them, due to their lower complexity.

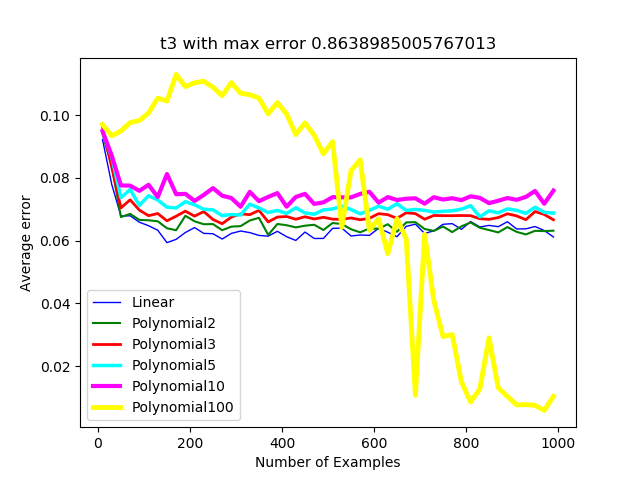
Finally, comparing these graphs for different target functions, shows that Eout is much greater in cases of higher target complexity functions.

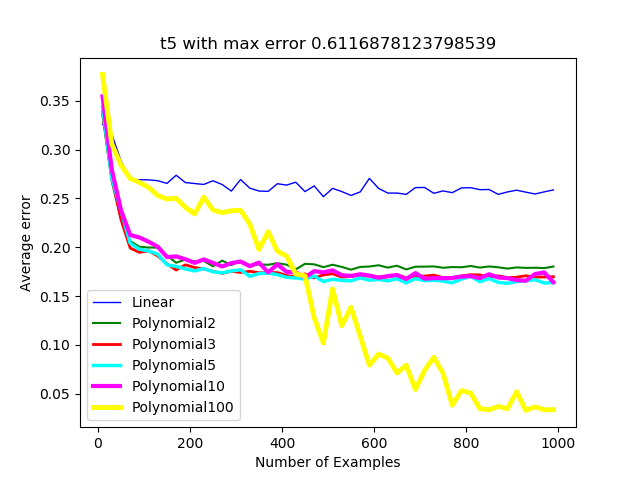
**Eout, kernels and number of examples**

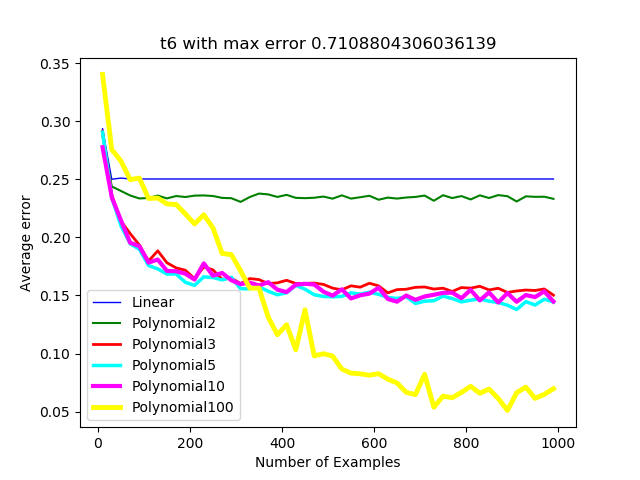
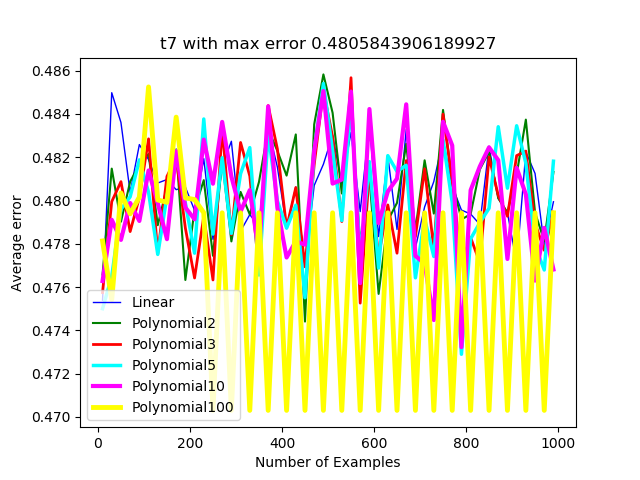
Similarly, the following graphs plot average-error over number of examples.

****

****

****

****

****

It is also clear, again as expected, that a higher number of examples leads to less Eout in every case. This, as was explained in the hypothesis, occurs because a greater N entails more and thus better training for the SVM.

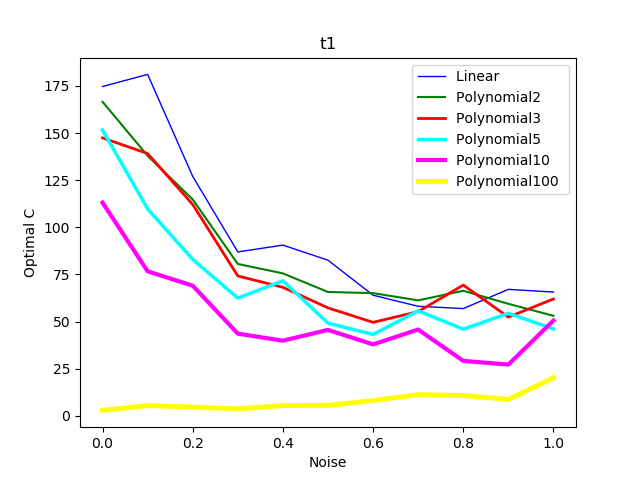
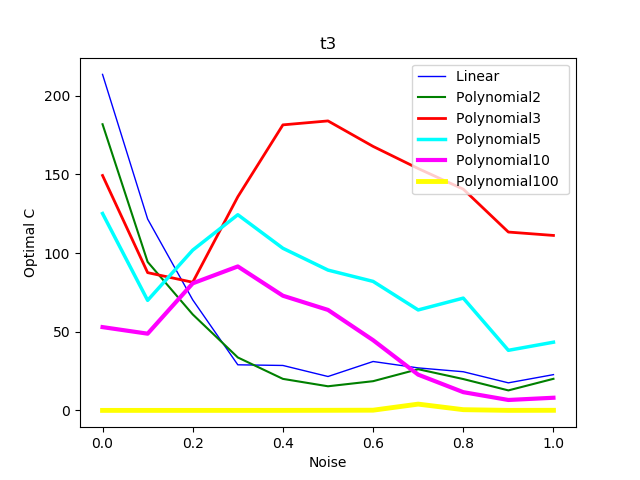
As was also assumed and mentioned in the hypothesis, the kernels that most drastically improve at bigger N’s are the more complex ones, for they turn from overfitting the small amounts of data into correctly classifying the larger ones. Moreover, when the target functions were more complex than the ability of some kernels, these naïve kernels could not fit the data better, no matter what the number of training data. This is particularly evident in functions t3 to t6, where the linear kernel and the polynomial with d=2 failed to improve throughout the course, whereas polynomials with values of 5, 10, and especially 100 minimized their out-of-sample error.

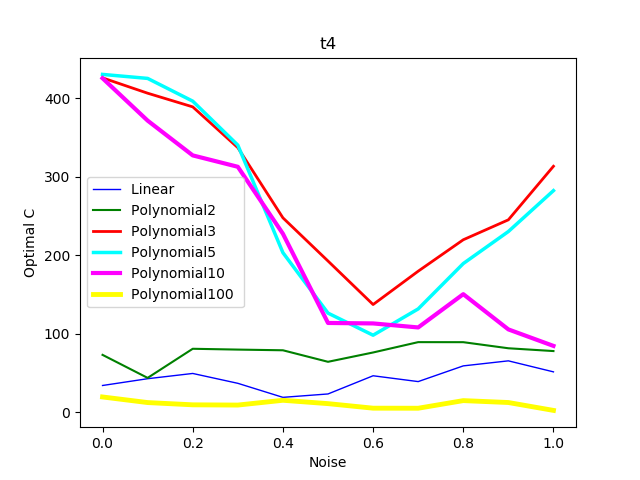
As for t7, even the polynomial100 was too naïve to handle such a case.

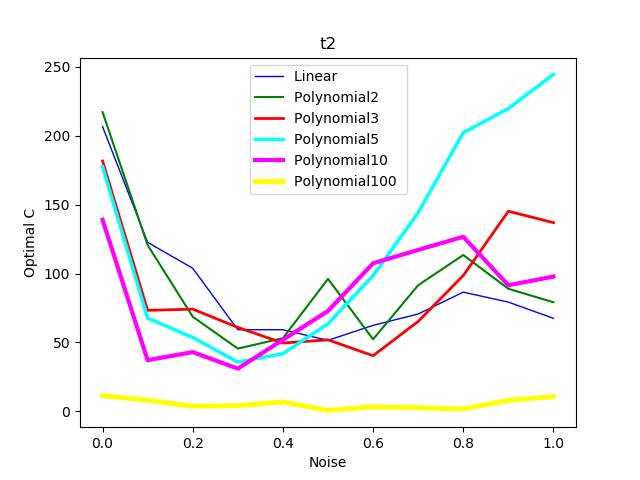
Finally, in this case too, comparing the graphs for different target functions, shows that Eout is much greater in cases of higher target complexity functions.

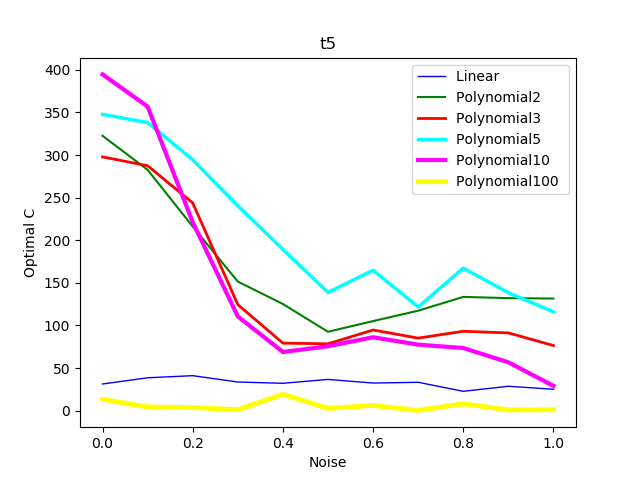
**C, kernels and noise**

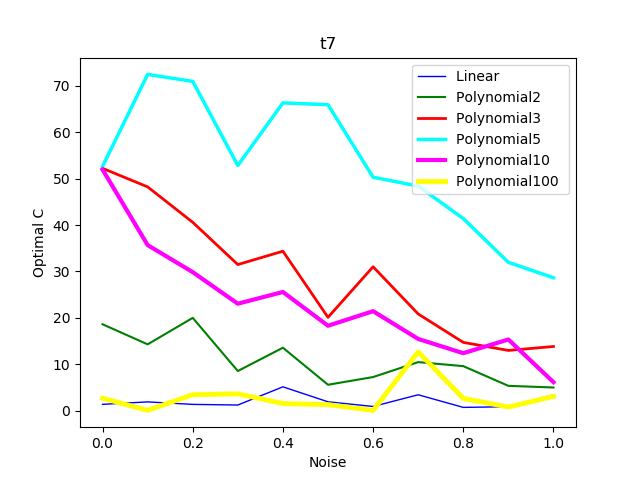
The following graphs plot the average optimal C as it was found through ten-fold cross validation for all kernels for each target function.

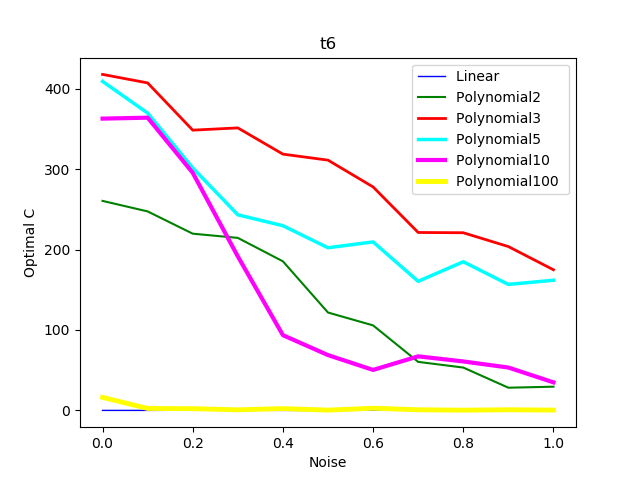
****

****

****

****

****

****

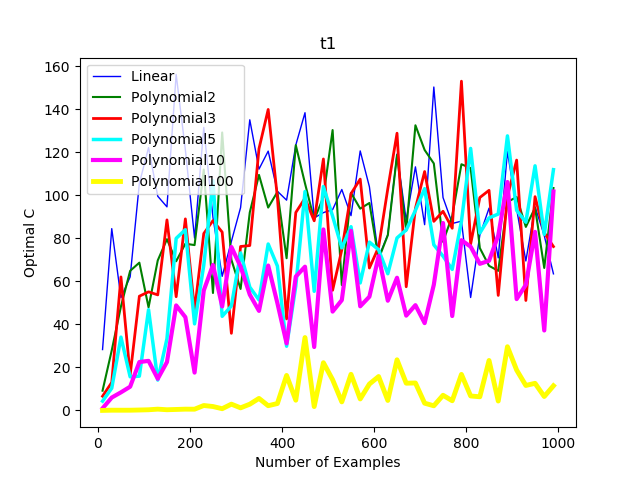
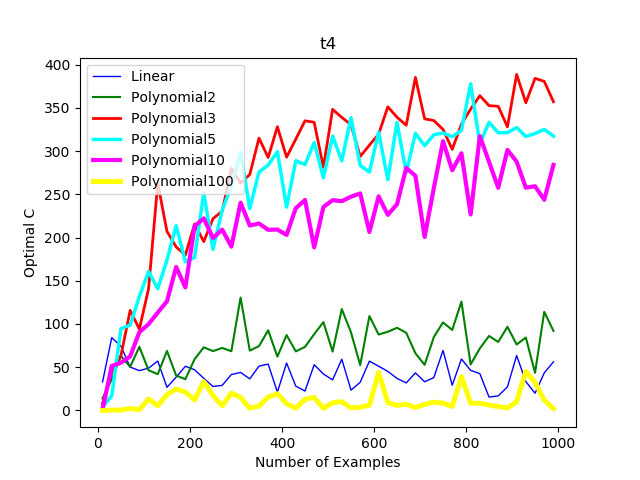
As far as the relationship between C and noise is concerned, it is not as clear as was assumed. There is definitely a tendency for C to decrease when noise increases, evident in the optimal C – noise levels graphs of t1, t5, t6 and t7. However, the three remaining graphs depict a relationship that is either parabolic or almost constant.

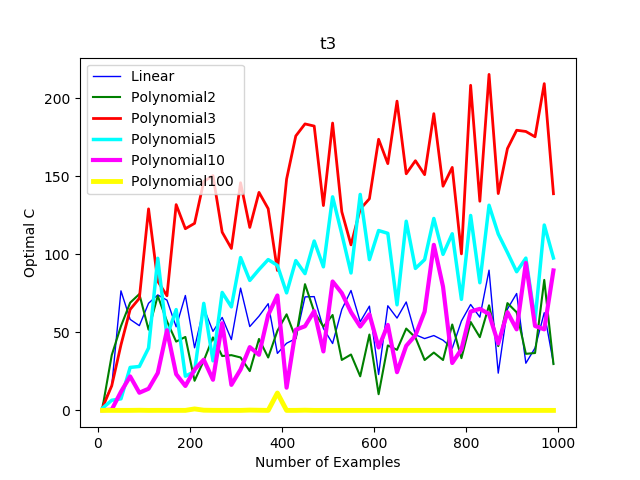
This is indeed counterintuitive. A possible explanation may be that at higher noise levels, the optimum C would indeed be lower, but the SVM just fails to understand it due to the vast distortion. Thus, it considers a different C than the real optimum as its optimum.

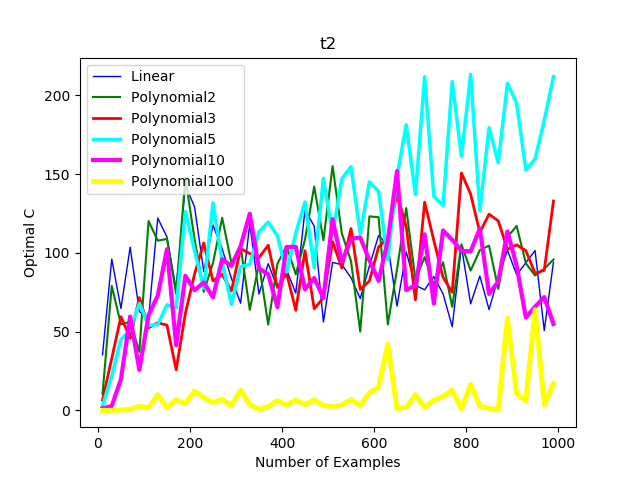
Finally, more complex kernels compared to the target function’s complexity, had consistently the lowest C, as an indication that they should “degrade”, let some of their complexity go to fit the data better. Thus, although it was not considered true before the experiment, it follows to logic that C does not depend on noise levels only, but also on the kernel’s complexity.

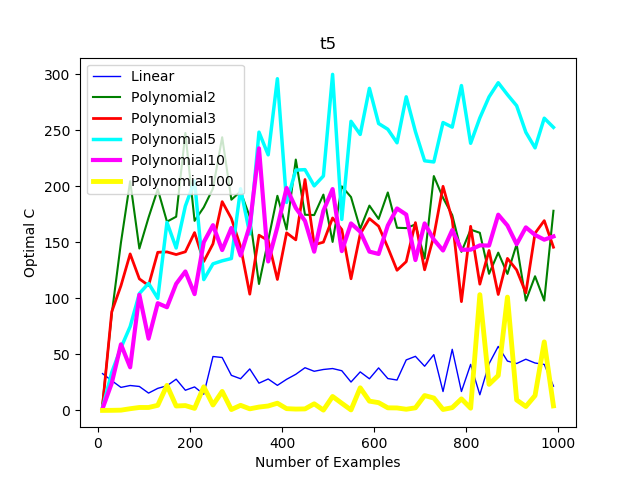
**C and number of examples**

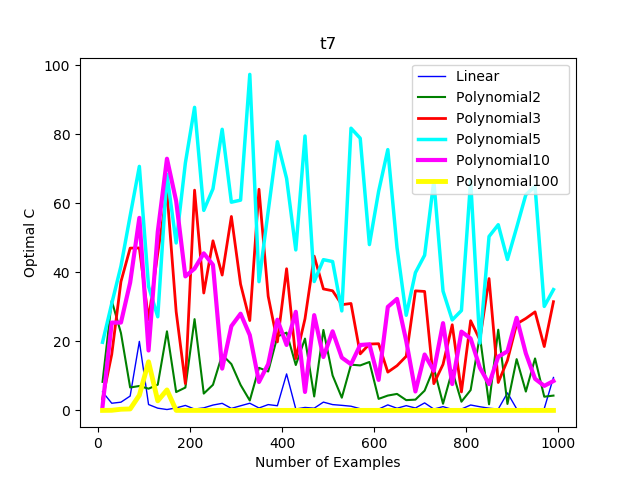
The last group of graphs plots the average optimal C over the number of examples.

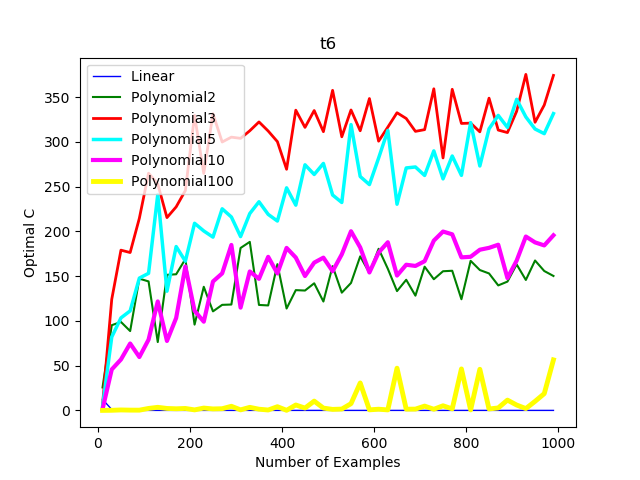
****

****

****

****

****

****

We might have found it that there’s more behind the selection of the optimal C than just noise, for example the kernels used, but the number of examples is, as was predicted, definitely not affecting the choice of C. All the graphs showing a relationship between the two, depict random curves with a lot of fluctuations.

**8.Summing up the results**

All in all, a quite simplistic representation, yet representative of the relationship between the independent and dependent variables, can be found below. An upward arrow means a positive relationship while a downward arrow a negative. Two arrows entail a stronger relationship. A dash means that the two variables are independent.

**Table 3: Summing up the results**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Optimal C | Optimal kernel complexity | Eout |
| Target function complexity |  |  |  |
| Noise levels |  |  |  |
| Number of examples |  |  |  |

**9.Conclusion**

Generalizing the above results, we can argue that the conclusion reached partially agrees and partially disagrees with the initial hypothesis. On the one hand, the negatively-sloped relationship between the number of training examples and the out-of-sample error is clearly shown. Furthermore, the negative relationship between C and noise levels is also evident. Finally, the perception that kernels of similar complexities to the complexity of the target function perform better was also evident, as well as that complex kernels improved more when trained with many examples.

On the other hand, the preferred hardness proved to be dependent on factors other than the levels of noise, too. The kernel’s complexity was one of them. Others could also be playing a role. Failure to identify the best C was also found to be possible. Moreover, it was established that the max error of the target function affects the effect of the noise levels of the sample.

**10.Further Research**

Due to limitations in computational power, time and word count, a number of additions and improvements that could be introduced to the experiment were not carried out.

The most important weakness of the current experiment was the absence of the rbf kernel. Further research could be carried out to determine whether the conclusions reached before hold true for this kernel as well. Optimization in relation to gamma, parameter of rbf, would also be noteworthy.

Finally, a different experiment could be carried out, to determine what all the factors affecting C are, and how to avoid the fallacy of choosing an unsuitable value.

**WORD COUNT: 3968**

**References**

Caltech. *YouTube,* YouTube, 28 Aug. 2012,

https://m.youtube.com/watch?v=mbyG85GZ0PI&list=PLD63A284B7615313A&index=2&t=0s.

Drakos, Georgios. “Support Vector Machine vs Logistic Regression.” Medium, Towards Data Science, 5 Oct. 2018, https://towardsdatascience.com/support-vector-machine-vs-logistic-regression-94cc2975433f.

“Installing¶.” *Installing - Matplotlib 3.1.1 Documentation*, https://matplotlib.org/3.1.1/users/installing.html.

Kong, Qingkai. “Machine Learning 8 - Support Vector Machine - Part 2 - Sklearn Classification Example.” Machine Learning 8 - Support Vector Machine - Part 2 - Sklearn Classification Example, 1 Jan. 1970, http://qingkaikong.blogspot.com/2016/12/machine-learning-8-support-vector.html.

“Learn.” *Scikit,* https://scikit-learn.org/stable/.

“Machine Learning Glossary | Google Developers.” *Google,* Google, https://developers.google.com/machine-learning/glossary/.

“Math - Mathematical Functions¶.” *Math - Mathematical Functions - Python 3.8.0 Documentation*, https://docs.python.org/3/library/math.html.

“NumPy¶.” *NumPy*, https://numpy.org/.

“Openpyxl.” *PyPI*, https://pypi.org/project/openpyxl/.

“Os-Sys.” *PyPI*, https://pypi.org/project/os-sys/.

“Pandas.” *PyPI*, https://pypi.org/project/pandas/.

“Random - Generate Pseudo-Random Numbers¶.” *Random - Generate Pseudo-Random Numbers - Python 3.8.0 Documentation*, https://docs.python.org/3/library/random.html.

“Scikit-Learn.” *PyPI*, https://pypi.org/project/scikit-learn/.

“Seaborn.” *PyPI*, https://pypi.org/project/seaborn/.

“Warnings - Warning Control¶.” *Warnings - Warning Control - Python 3.8.0 Documentation*, https://docs.python.org/3/library/warnings.html.

**Appendix A - The main program of the experiment**

from random import seed

from random import random

from sklearn import svm

import numpy as np

import sys

import openpyxl

import os

import math

import warnings

os.chdir('D:\\Desktop')

wb=[openpyxl.Workbook()]

acIND=0

for i in range(6):

wb.append(openpyxl.Workbook())

def t1(x1,x2):

return x2-0.1 #0 degree polynomial

def t2(x1,x2):

return x2-1.5\*x1+0.05 #first degree polynomial

def t3(x1,x2):

return x2-18\*(x1\*\*2)\*+1.3\*x1+1.4 #second degree polynomial

def t4(x1,x2):

return x2-172.8\*(x1\*\*3)+40\*(x1\*\*2)+x1-0.25 #third degree polynomial

def t5(x1,x2):

if x1<-0.2:

return t2(x1,x2)

elif x1<0.2:

return t4(x1,x2)

else:

return -(t3(x1,x2))#different functions in different ranges

def t6(x1,x2):

if (x1-0.2)\*\*2+(x2+0.3)\*\*2<0.1\*\*2:

return 1

elif (x1+0.4)\*\*2-(x2-0.1)\*\*3 < 0.1:

return -1

else:

return t4(x1,x2) #logical combinations of different function

def t7(x1,x2):

if 0.01\*(x2\*\*2) > 100\*((x1+2.5)\*\*2)\*math.sin(100\*((x1+2.5)\*\*2)+0.01\*x1\*\*2):

return 1

else:

return -1 #complex trig function

def linear(x1,x2):

return np.dot(x1,x2.T)

def polynomial2(x1,x2):

return (linear(x1,x2)+1)\*\*2

def polynomial3(x1,x2):

return (linear(x1,x2)+1)\*\*3

def polynomial5(x1,x2):

return (linear(x1,x2)+1)\*\*5

def polynomial10(x1,x2):

return (linear(x1,x2)+1)\*\*10

def polynomial100(x1,x2):

return (linear(x1,x2)+1)\*\*100

targetFunctions = [t1, t2, t3, t4, t5, t6, t7]

functionError=[]

for i in range(7):

functionError.append([0,0])

for t in range(0,7):

INDEX1=-0.5

while INDEX1<=0.5:

INDEX2=-0.5

while INDEX2<=0.5:

if (targetFunctions[t])(INDEX1, INDEX2) >= 0:

functionError[t][0] += 1

else:

functionError[t][1] +=1

INDEX2+=0.02

INDEX1+=0.02

functionError[t][0]/=2601

functionError[t][1]/=2601

kernels=[linear, polynomial2, polynomial3, polynomial5, polynomial10,polynomial100]

possC=[]

for i in functionError:

print(str(i[0]) + " " + str(i[1]))

for i in range(-4,3):

possC.append(10\*\*i)

possC.append(5 \* (10\*\*i))

seed(1)

genError=0

for t in range(0,len(targetFunctions)):

noiseRow=1

NRow=1

exRow=1

kernelRow=1

ac=wb[acIND]

acIND+=1

sh=ac.active

NOISE=0

while NOISE<=1:

sh.cell(row=noiseRow, column=1).value=NOISE

noiseRow+=1500

N=10

while N<=1000:

sh.cell(row=NRow, column=2).value=N

NRow+=30

for temp in range(5):

y=[]

ex=[]

for tempp in range(N):

x1=random()-0.5

x2=random()-0.5

if targetFunctions[t](x1,x2) >= 0:

y.append(1)

else:

y.append(0)

r1=(random()\*NOISE/2)+(NOISE/2)

if random() < 0.5:

r1 = -r1

r2=(random()\*NOISE/2)+(NOISE/2)

if random() < 0.5:

r2-=r2

ex.append([x1\*(1+r1), x2\*(1+r2)])

sh.cell(row=exRow, column=3).value=str(ex)

sh.cell(row=exRow, column=4).value=str(y)

exRow+=6

uniqueClass=True

for tempp in y:

if tempp != y[0]:

uniqueClass=False

break

if uniqueClass:

if y[0]==1:

genError=functionError[t][1]

else:

genError=functionError[t][0]

for ker in kernels:

sh.cell(row=kernelRow, column=5).value=str(ker)

sh.cell(row=kernelRow, column=6).value="undefined"

sh.cell(row=kernelRow, column=7).value=genError

kernelRow+=1

continue

for ker in kernels:

sh.cell(row=kernelRow, column=5).value=str(ker)

minError=-1

perfectC=-1

for PC in possC:

error=0

for COUNT in range(10):

val=ex[COUNT\*N//10 : (COUNT+1)\*N//10]

tr = ex[0:COUNT\*N//10]

tr += ex[(COUNT+1)\*N//10 : N]

valY=y[COUNT\*N//10 : (COUNT+1)\*N//10]

trY = y[0:COUNT\*N//10]

trY += y[(COUNT+1)\*N//10 : N]

unCl=True

for temporary in trY:

if temporary!=trY[0]:

unCl=False

break

if unCl:

for dis in valY:

if dis!=y[0]:

error+=1

continue

with warnings.catch\_warnings(record=True) as w:

warnings.simplefilter("error")

try:

clf = svm.SVC(kernel=ker, C=PC, max\_iter=1000000)

clf.fit(tr, trY)

except:

error+=N//10

continue

for cur in range(N//10):

if clf.predict([val[cur]]) != valY[cur]:

error+=1

if ((error < minError) or (minError<0)):

minError=error

perfectC=PC

with warnings.catch\_warnings(record=True) as w:

warnings.simplefilter("error")

sh.cell(row=kernelRow, column=6).value = perfectC

try:

clf=svm.SVC(kernel=ker, C=perfectC, max\_iter=1000000)

clf.fit(ex,y)

except:

sh.cell(row=kernelRow, column=7).value=1

kernelRow+=1

continue

misclassified=0

INDEX1=-0.5

while INDEX1<=0.5:

INDEX2=-0.5

while INDEX2<=0.5:

expectation=0

if targetFunctions[t](INDEX1,INDEX2) >= 0:

expectation=1

if clf.predict([[INDEX1, INDEX2]]) != expectation:

misclassified += 1

INDEX2 += 0.02

INDEX1 += 0.02

sh.cell(row=kernelRow,column=7).value=misclassified/2601

kernelRow+=1

N+=20

NOISE+=0.1

ac.save("excel" + str(acIND-1) + ".xlsx")

*Note: Results might be slightly different if the code given is run, since the experiment was stopped and started from different points with different seed values twice, due to power losses while running.*

**Appendix B – Code to visualize the results**

import matplotlib.pyplot as plt

import numpy as np

import seaborn as sns

import pandas as pd

import sys

import openpyxl

import os

import math

os.chdir('D:\\Desktop')

def t1(x1,x2):

return x2-0.1 #0 degree polynomial

def t2(x1,x2):

return x2-1.5\*x1+0.05 #first degree polynomial

def t3(x1,x2):

return x2-18\*(x1\*\*2)\*+1.3\*x1+1.4 #second degree polynomial

def t4(x1,x2):

return x2-172.8\*(x1\*\*3)+40\*(x1\*\*2)+x1-0.25 #third degree polynomial

def t5(x1,x2):

if x1<-0.2:

return t2(x1,x2)

elif x1<0.2:

return t4(x1,x2)

else:

return -(t3(x1,x2))#different functions in different ranges

def t6(x1,x2):

if (x1-0.2)\*\*2+(x2+0.3)\*\*2<0.1\*\*2:

return 1

elif (x1+0.4)\*\*2-(x2-0.1)\*\*3 < 0.1:

return -1

else:

return t4(x1,x2) #logical combinations of different function

def t7(x1,x2):

if 0.01\*(x2\*\*2) > 100\*((x1+2.5)\*\*2)\*math.sin(100\*((x1+2.5)\*\*2)+0.01\*x1\*\*2):

return 1

else:

return -1 #complex trig function

targetFunctions=[t1,t2,t3,t4,t5,t6,t7]

maxError=[]

for t in range(0,7):

err1=0

err2=0

INDEX1=-0.5

while INDEX1<=0.5:

INDEX2=-0.5

while INDEX2<=0.5:

if(targetFunctions[t])(INDEX1, INDEX2) >= 0:

err1 += 1

else:

err2 += 1

INDEX2+=0.02

INDEX1+=0.02

me=err1/2601

if err2>err1:

me=err2/2601

maxError.extend([me])

def noiseLevelDiagrams(num):

wb = openpyxl.load\_workbook("excel" + str(num) + ".xlsx")

sheet = wb.active

error=[]

noiseLevels=[]

funcError=[ [], [], [], [], [], [] ]

optC=[ [], [], [], [], [], [] ]

for i in range(1, 16500, 1500):

error.extend([maxError[0]])

noiseLevels.extend([float(sheet.cell(row=i, column=1).value)])

sum1=[0,0,0,0,0,0,0]

sum2=[0,0,0,0,0,0,0]

rem=[0,0,0,0,0,0,0]

for j in range(0,1500,6):

for k in range(0,6):

if((float(sheet.cell(row=i+j+k, column=7).value) != 1) and (sheet.cell(row=i+j+k, column=6).value != "undefined")):

sum1[k]+=float(sheet.cell(row=i+j+k, column=7).value)

sum2[k]+=float(sheet.cell(row=i+j+k, column=6).value)

else:

rem[k]+=1

for j in range(0,6):

'''if(rem[j] == 250):

func.Error[j].extend([2\*funcError[j-1]-funcError[j-2]])

optC.extend([2\*optC[j-1]-optC[j-2]])

continue'''

sum1[j] /= (250-rem[j])

sum2[j] /= (250-rem[j])

funcError[j].extend([sum1[j]])

optC[j].extend([sum2[j]])

df=pd.DataFrame({'Noise levels':noiseLevels, 'Linear': funcError[0],

'Polynomial2': funcError[1], 'Polynomial3': funcError[2],

'Polynomial5': funcError[3], 'Polynomial10': funcError[4],

'Polynomial100': funcError[5], 'Linear ': optC[0],

'Polynomial2 ': optC[1], 'Polynomial3 ': optC[2],

'Polynomial5 ': optC[3], 'Polynomial10 ': optC[4],

'Polynomial100 ': optC[5]})

plt.figure(0)

plt.plot('Noise levels', 'Linear', color='blue', linewidth=1, data=df)

plt.plot('Noise levels', 'Polynomial2', color='green', linewidth=1.5, data=df)

plt.plot('Noise levels', 'Polynomial3', color='red', linewidth=2, data=df)

plt.plot('Noise levels', 'Polynomial5', color='cyan', linewidth=2.5, data=df)

plt.plot('Noise levels', 'Polynomial10', color='magenta', linewidth=3, data=df)

plt.plot('Noise levels', 'Polynomial100', color='yellow', linewidth=3.5, data=df)

"""plt.plot('Noise levels', 'Max Error', color='black', linewidth=1, linestyle='dashed', data=df)"""

plt.legend()

plt.title("t" + str(num+1) + " with max error " + str(maxError[num]))

plt.xlabel("Noise")

plt.ylabel("Average error")

plt.show()

plt.figure(1)

plt.plot('Noise levels', 'Linear ', color='blue', linewidth=1, data=df)

plt.plot('Noise levels', 'Polynomial2 ', color='green', linewidth=1.5, data=df)

plt.plot('Noise levels', 'Polynomial3 ', color='red', linewidth=2, data=df)

plt.plot('Noise levels', 'Polynomial5 ', color='cyan', linewidth=2.5, data=df)

plt.plot('Noise levels', 'Polynomial10 ', color='magenta', linewidth=3, data=df)

plt.plot('Noise levels', 'Polynomial100 ', color='yellow', linewidth=3.5, data=df)

plt.legend()

plt.title("t"+str(num+1))

plt.xlabel("Noise")

plt.ylabel("Optimal C")

plt.show()

noiseLevelDiagrams(0)

noiseLevelDiagrams(1)

noiseLevelDiagrams(2)

noiseLevelDiagrams(3)

noiseLevelDiagrams(4)

noiseLevelDiagrams(5)

noiseLevelDiagrams(6)

def numberOfExampleDiagrams(num):

wb=openpyxl.load\_workbook("excel" + str(num) + ".xlsx")

sheet=wb.active

error=[]

numberOfExamples=[]

funcError=[ [], [], [], [], [], [] ]

optC=[ [], [], [], [], [], [] ]

for f in range(1,1500,30):

numberOfExamples.extend([float(sheet.cell(row=f, column=2).value)])

sum1=[0,0,0,0,0,0,0]

sum2=[0,0,0,0,0,0,0]

rem=[0,0,0,0,0,0,0]

for i in range(0, 16500, 1500):

for j in range(0,30,6):

for k in range(0,6):

if((float(sheet.cell(row=f+i+j+k, column=7).value != 1)) and (sheet.cell(row=f+i+j+k, column=6).value != "undefined")):

sum1[k]+=float(sheet.cell(row=f+i+j+k, column=7).value)

sum2[k]+=float(sheet.cell(row=f+i+j+k, column=6).value)

else:

rem[k]+=1

for i in range(0,6):

if rem[i]==55:

'''funcError[i].extend([2\*funcError[i][len(funcError[i])-1]-funcError[i][len(funcError[i])-2]])

optC[i].extend([2\*optC[i][len(optC[i])-1]-optC[i][len(optC[i])-2]])'''

funcError[i].extend([funcError[i][len(funcError[i])-2]])

optC[i].extend([optC[i][len(optC[i])-2]])

continue

sum1[i] /= (55-rem[i])

sum2[i] /= (55-rem[i])

funcError[i].extend([sum1[i]])

optC[i].extend([sum2[i]])

df=pd.DataFrame({'Number of Examples':numberOfExamples, 'Linear': funcError[0],

'Polynomial2': funcError[1], 'Polynomial3': funcError[2],

'Polynomial5': funcError[3], 'Polynomial10': funcError[4],

'Polynomial100': funcError[5], 'Linear ': optC[0],

'Polynomial2 ': optC[1], 'Polynomial3 ': optC[2],

'Polynomial5 ': optC[3], 'Polynomial10 ': optC[4],

'Polynomial100 ': optC[5]})

plt.figure(0)

plt.plot('Number of Examples', 'Linear', color='blue', linewidth=1, data=df)

plt.plot('Number of Examples', 'Polynomial2', color='green', linewidth=1.5, data=df)

plt.plot('Number of Examples', 'Polynomial3', color='red', linewidth=2, data=df)

plt.plot('Number of Examples', 'Polynomial5', color='cyan', linewidth=2.5, data=df)

plt.plot('Number of Examples', 'Polynomial10', color='magenta', linewidth=3, data=df)

plt.plot('Number of Examples', 'Polynomial100', color='yellow', linewidth=3.5, data=df)

"""plt.plot('Noise levels', 'Max Error', color='black', linewidth=1, linestyle='dashed', data=df)"""

plt.legend()

plt.title("t" + str(num+1) + " with max error " + str(maxError[num]))

plt.xlabel("Number of Examples")

plt.ylabel("Average error")

plt.show()

plt.figure(1)

plt.plot('Number of Examples', 'Linear ', color='blue', linewidth=1, data=df)

plt.plot('Number of Examples', 'Polynomial2 ', color='green', linewidth=1.5, data=df)

plt.plot('Number of Examples', 'Polynomial3 ', color='red', linewidth=2, data=df)

plt.plot('Number of Examples', 'Polynomial5 ', color='cyan', linewidth=2.5, data=df)

plt.plot('Number of Examples', 'Polynomial10 ', color='magenta', linewidth=3, data=df)

plt.plot('Number of Examples', 'Polynomial100 ', color='yellow', linewidth=3.5, data=df)

plt.legend()

plt.title("t"+str(num+1))

plt.xlabel("Number of Examples")

plt.ylabel("Optimal C")

plt.show()

numberOfExampleDiagrams(0)

numberOfExampleDiagrams(1)

numberOfExampleDiagrams(2)

numberOfExampleDiagrams(3)

numberOfExampleDiagrams(4)

numberOfExampleDiagrams(5)

numberOfExampleDiagrams(6)

1. https://developers.google.com/machine-learning/glossary/. [↑](#footnote-ref-1)
2. https://m.youtube.com/watch?v=mbyG85GZ0PI&list=PLD63A284B7615313A&index=2&t=0s. [↑](#footnote-ref-2)
3. *Copied,* https://towardsdatascience.com/support-vector-machine-vs-logistic-regression-94cc2975433f [↑](#footnote-ref-3)
4. *Copied,* http://qingkaikong.blogspot.com/2016/12/machine-learning-8-support-vector.html [↑](#footnote-ref-4)
5. https://scikit-learn.org/stable/. [↑](#footnote-ref-5)
6. https://docs.python.org/3/library/random.html [↑](#footnote-ref-6)
7. https://pypi.org/project/scikit-learn/ [↑](#footnote-ref-7)
8. https://numpy.org/ [↑](#footnote-ref-8)
9. https://matplotlib.org/3.1.1/users/installing.html [↑](#footnote-ref-9)
10. https://pypi.org/project/os-sys/ [↑](#footnote-ref-10)
11. https://pypi.org/project/os-sys/ [↑](#footnote-ref-11)
12. https://pypi.org/project/openpyxl/ [↑](#footnote-ref-12)
13. https://docs.python.org/3/library/math.html [↑](#footnote-ref-13)
14. https://docs.python.org/3/library/warnings.html [↑](#footnote-ref-14)
15. https://pypi.org/project/pandas/ [↑](#footnote-ref-15)
16. https://pypi.org/project/seaborn/ [↑](#footnote-ref-16)