Machine Learning Methods on Simulated Experiment Data

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science in
Physics from the College of William and Mary in Virginia,

by

Bowen Zhang

Advisor: Prof. Keith Griffioen

Williamsburg, Virginia
May 2018

Contents

List of Figures

List of Tables

Abstract

As machine learning becomes a trend, physicists are exploring how to use it in scientific research. Physics is no exception. In this paper, I will investigate the potential utilization of two machine learning techniques on two different sets of simulation physics experiment data. Specifically, I utilized k-nearest neighbors algorithms and decision tree methods to classify particles in a trident event and particles that are accelerated into a uniformed magnetic field. The result shows that both of these two models, under certain circumstances, show some potential ability to classify data with high accuracy.

Chapter 1

Introduction

Traditionally, Physics experimental data processing is a quite heavy labor field. It required a very deep understanding of the experiment itself just to process the data correctly. It is even harder for the researcher to identify factors that affect the data accuracy, especially the biases embedded in the experimental system itself. For the last decades, as more and more powerful computers are created, people started to explore the machine learning field, where they hope to offer these computers the ability to learn and perform certain task, by using statistical techniques. In recent years, high performance central processing units with multiple cores has become very accessible. So, I am wondering if I can explore the possibility to utilize machine learning techniques to help processing data collected in Physics experiment. These techniques should not require a deep understanding of the theory behind experiments, and should be able to self-adjusted to noise, even if we do not know the cause of it. Apparently, this kind of method will not be accepted by today's researcher as a rigorous data processing method, due to the unproven uncertainties behind most of machine learning technique. But it does not stop people from analyzing such possibility. In this research, I will utilize two machine learning methods, the K-nearest Neighbors Algorithm and Decision Tree, on two general Physics experiment simulation settings, and assess these methods in term of their accuracy and efficiency.

Chapter 2

Theory

2.1 K-nearest Neighbors Algorithm

K-nearest Neighbors Algorithm, or the kNN Algorithm, is a simple machine learning method that can be used to classify unlabeled observations by comparing the label of the k most similar examples. Here, the most similar example is the nearest sample, and the distance between one sample and another is calculated using Euclidean distance in the kNN method, as shown in Equation 2.1.

$$D(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2} \tag{2.1}$$

Where $D(p,q)$ stand for the distance between observation p and q. $p_1, p_2, ..., p_n$ stand for n variables of observation p. $q_1, q_2, ..., q_n$ stand for n variables of observation q. After comparing and find the distance between the unknown observation and each of all the labeled observation, the class of this unknown observation is decided by the majority vote of the k nearest labeled observation, where k is a real positive number chosen by the user. In this paper, I will use the 'class' package in R to utilize the kNN method in several different ways.

2.2 Decision Tree

The decision tree algorithm is a kind of supervised learning, which can be used to solve classification problems. The overall goal is to classify the target variables based on several input variables of such target. This algorithm uses a decision tree to represent the solution to a certain problem, where every leaf node contains a class label while the value of each input variables is judged on the internal nodes. In order to identify the class value of targets, each level of the tree needs to identify an attribution of the model. There are many popular attribution selections like information gain method. However, in our discussion, I use the Gini Index as the attribution selection method.

Gini Index, also known as Gini score, was first introduced in 1912 by Corrado Gini in order to measure the income or wealth distribution of a nation's residents. Later on, it was adapted to many uses. In our case, it is used to evaluate splits in a model by measuring how often a randomly chosen element would be incorrectly identified. In this sense, a perfect split with no error will have a Gini score of 0, and the lower Gini score a split get, the better this split is. The formula I use to calculate the Gini Index in this decision tree model can be written as in Equation 2.1, where p_j stand for the probability of class j within each attribute split. In this research, I will use the 'rpart' package in R to apply a classification decision tree to the training dataset and use the resulting model to classify the testing dataset.

$$GiniIndex = 1 - \sum_j p_j^2 \tag{2.2}$$

The decision tree function as following. First, a training dataset is split into two lists, according to some input attribute. Then, using the Gini Index formula, each split that utilizing different attribute value is evaluated. Then the best split can be found for some specific input attribute. By iterating through this step over and over,

a tree can be constructed using the acquired best split values, and this tree can be used to predict the class of target variables.

Chapter 3

Simple Particles Four-vectors Model

3.1 Overview

This simulation data is meant to mimic a trident event where an electron is accelerated and collide with a tungsten target. This collision will emit an electron-positron pair. So, in total, two electrons and a positron will be detected. Each particle will be described by a four-vector that records its energy and three momentums under a three-dimensional Cartesian coordinate system. Our goal is to use k-nearest neighbors algorithm and decision tree classification to identify each particle based on its four-momentum vectors. This dataset is provided by Professor Keith Griffioen and contains 7714 groups of data. Each group contains 3 particles, with one scattered electron and a pair of electron and positron.

3.1.1 Data With No Charge Indicator

In this first part, I deal with the raw dataset that do not contain any information regarding the charge of each particles.

kNN-Classification

Using k-nearest neighbors algorithm (kNN) method, I take 5784 groups as the training set and the remaining 1929 group as the test dataset. The testing accuracy with different choices of neighborhood number n is described as follows. The code used for this Chapter is attached to Appendix A.

Applying the kNN algorithm with different levels of K-number on the original data with no additional noise, I get the accuracy shown in table 3.1. The first column indicate the choices of K number cover a fairly large range and thus can provide us with a better understanding of its effects on the accuracy. The second column shows the accuracy for predicting which one of the three particles is the scattered electron. The third and fourth columns shows the accuracy for predicting the paired positron and electron. As I can find from the table that there is no good indicator to tell the two paired particles apart and thus this kNN model cannot distinguish these two particles without further information. The accuracy for predicting the scattered electron is very promising, especially for small k numbers.

Since the original database is an ideal database, which is unrealistic. To mimic the actual experimental data, I need to add some noise to simulate the data. For that reason, I then generated independent random noise for each measurement based on a Gaussian distribution centered at mean of $\mu = 0$ and standard deviation of σ. Then these independent noises are add to every single measurement in the four-vector for each particle. To test how this model performs under different noise, I chose four different standard deviations $\sigma \in \{0.05, 0.1, 0.3, 0.5\}$. The resulting accuracy is shown in Tables 3.2 - 3.5.

As I can see from the tables that once the noise level σ reaches $\sigma = 0.3$, the accuracy of the model decreases rapidly. This may be caused by the nature of this dataset. Over two-thirds of the measurements contained in this dataset have a scale

Table 3.1: Accuracy of the kNN Model on a Simple Particle Dataset Without Noise

k	*Scattered* (%)	*Paired* e^+ (%)	*Paired* e^- (%)	*Total* (%)
1	97.04	51.21	50.33	66.2
3	97.09	50.75	49.66	65.83
5	97.14	51.11	48.72	65.66
10	96.94	50.18	48.88	65.33
20	96.52	50.23	48.31	65.02

Table 3.2: Accuracy of the kNN Model on a Simple Particle Dataset With Noise Level of $\sigma = 0.05$

k	*Scattered* (%)	*Paired* e^+ (%)	*Paired* e^- (%)	*Total* (%)
1	92.32	49.24	49.3	63.62
3	93.88	49.19	47.84	63.64
5	94.66	48.52	49.14	64.10
10	95.23	50.44	48.57	64.74
20	95.17	48.26	48.21	63.88

Table 3.3: Accuracy of the kNN Model on a Simple Particle Dataset With Noise Level of $\sigma = 0.1$

k	*Scattered* (%)	*Paired* e^+ (%)	*Paired* e^- (%)	*Total* (%)
1	90.51	47.22	47.95	61.89
3	92.53	47.22	48.1	62.62
5	92.53	48.52	48.67	63.24
10	93.83	50.07	47.17	63.69
20	94.03	49.19	48.1	63.78

Table 3.4: Accuracy of the kNN Model on a Simple Particle Dataset With Noise Level of $\sigma = 0.3$

k	*Scattered* (%)	*Paired* e^+ (%)	*Paired* e^- (%)	*Total* (%)
1	66.2	42.4	44.11	50.9
3	73.97	42.82	42.5	53.1
5	76.87	44.16	44.94	55.33
10	78.79	43.85	44.32	55.65
20	80.92	42.45	44.06	55.81

Table 3.5: Accuracy of the kNN Model on a Simple Particle Dataset With Noise Level of $\sigma = 0.5$

k	*Scattered* (%)	*Paired* e^+ (%)	*Paired* e^- (%)	*Total* (%)
1	50.12	38.77	36.85	41.92
3	55.78	38.56	36.91	43.75
5	57.8	38.41	37.53	44.58
10	62	37.42	38.25	45.89
20	64.74	39.03	38.93	47.57

that is smaller than 0.3 and thus a random noise with standard deviation of $\sigma \geq 0.3$ will have a quite huge impact on the dataset itself. Another interesting phenomenon I can find is that in the previous case with no noise, the smaller k-number I choose, the more accurate this model is. However, after adding noise to the dataset, even when the noise level remains small, I find that the accuracy of this model increases as the K-number increases. This may be due to the fact that our noise has a mean of $\mu = 0$ and thus the more training points the model can take into account, the more likely the noise will be averaged out in the classification process. According to these result, I conclude that kNN classification is quite able to find the scattered electron in this settings, even with some noise.

Decision Tree Model

Switching to the decision tree model, I also took 5784 groups as the training set and the remaining 1929 group as the test dataset. The prediction result on raw data with different levels of noise is shown in Table 3.6 and Figure 3.1. In each figure, 'pz' represent the momentum of the particle in the z direction, which is the direction of momentum of the input electron. The 'E' term represent the overall energy of the particle. 'p1', 'p2', 'p3', each successively represent the scattered electron, paired positron and paired electron. From this case, I find that decision tree only utilized momentum in one direction and the overall energy as the judgment variables. Fur-

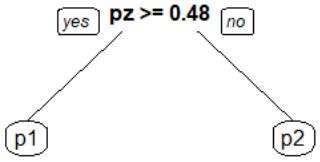
Table 3.6: Accuracy of Decision Tree Model on Simple Particle Dataset on Different Noise Level of σ

σ	*Scattered* (%)	*Paired* e^+ (%)	*Paired* e^- (%)	*Total* (%)
0	94.71	98.7	0	64.47
0.05	94.81	0	97.61	64.14
0.1	91.34	97.97	0	63.1
0.3	82.01	86.57	6.79	58.45
0.5	78.74	0	73.3	50.68

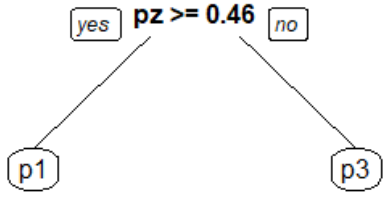
thermore, this model generated many cases with zero accuracies for one of the paired particles. These may be caused by the fact that the most distinguishing variables that differ from the scattered electron and paired particles are the momentum in the z direction and the overall energy. Also, since two paired particles have similar measurements in this setting, the decision tree model would just drop one of the paired particles and classify them both as either positrons or electrons. Another thing worth mentioning is that when I increased the standard deviation for the artificial noise, the tree structure is not affected until $\sigma \geq 0.3$. This is consistent with the result I see from the kNN classification model and thus indicates that around $\sigma = 0.3$, the noise would potentially render the data meaningless.

3.1.2 Data With Charge Indicator

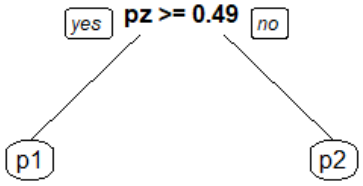
As I find in the previous section that it is impossible to tell apart the paired particle without knowing their charges and in reality, I have an instrument that can indicate the change of each particle. Thus, in this section, I added a charge indicator to each particle. Due to the possibility of misjudgment in actual experiment by the detractor, I need to add some artificial noise to this indicator. First, I assigned every electron with a charge indicator value of 1 and every positron with a value of 0. Then, for each indicator, I add a random noise that follows a normal distribution centered at $\mu = 0$ with a standard deviation of σ. Then I reassigned every particle with an indicator



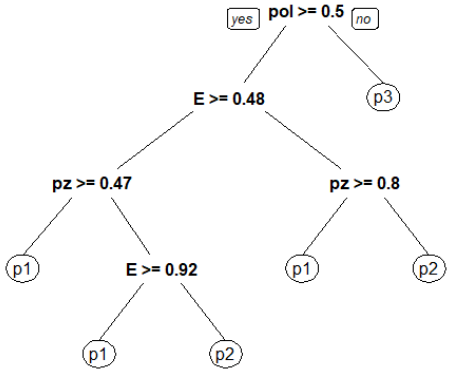
(a) Decision Tree For Simple Particle Dataset Without Noise



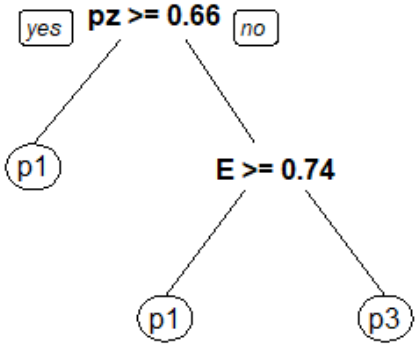
(b) Decision Tree For Simple Particle Dataset With Noise $\sigma = 0.05$



(c) Decision Tree For Simple Particle Dataset With Noise $\sigma = 0.1$



(d) Decision Tree For Simple Particle Dataset With Noise $\sigma = 0.3$



(e) Decision Tree For Simple Particle Dataset With Noise $\sigma = 0.5$

Figure 3.1: Decision Tree Generated For The Simple Particle Dataset

value bigger than 0.5 to 1 and others to 0. In this setting, for example, if I set $\sigma = 0.5$, the theoretical possibility of misjudgment is 15.86%. Then I utilized the previous two models with this new dataset.

kNN-Classification

Again, I took 5784 groups as the training set and the remaining 1929 group as the test dataset. The testing accuracy for the kNN model with a different choice of neighborhood number n is shown in Table 3.7. Note that 'S' stands for scattered electron, '$P\ e^-$' stands for the paired electron, the '$P\ e^+$' stands for the paired positron, and 'T' stands for the total accuracy. Predictably, without noise, the accuracy for classifying positron is one hundred percent, due to its unique charge indicator. This not only helps to boost the total accuracy, it also reduces other types of wrong judgment which results in an increase of the accuracy for classifying other particles. As I increase the noise level, I find that this model is very sensitive to the change in charge indicator. An increase in standard deviation σ as small as 0.05 leads to a huge drop in the positron prediction accuracy, especially on low-level k-number. However, as I increase the k-number, the positron prediction accuracy increases regardless of the noise level.

Decision Tree Model

Using the same model as above, I train the decision tree model with 5784 data from this new dataset. The generated trees and the prediction accuracy result are shown in Figure 3.2 and Table 3.8. Note that '*pol*' represents the charge indicator. As expected, it utilized the charge indicator as intended. If it is bigger than 0.5, the particles will be classified as a positron. Otherwise, the model uses both the total energy and the momentum in the z direction to distinguish the paired electron and the scattered electron. The prediction accuracy looks quite promising as well. Though

Table 3.7: Accuracy of kNN Model on Particle Dataset With Charge Indicator On Different Noise Level of σ

k	S (%)	e^- (%)	e^+ (%)	T (%)
1	97.72	97.46	100.00	98.39
3	97.72	98.34	100.00	98.69
5	97.62	98.13	100.00	98.58
10	97.15	97.51	100.00	98.22
20	96.32	97.20	100.00	97.84

(a) $\sigma = 0$

k	S (%)	e^- (%)	e^+ (%)	T (%)
1	93.47	70.55	69.31	77.78
3	94.66	76.36	76.31	82.44
5	94.66	79.16	77.92	83.91
10	95.33	81.03	80.25	85.54
20	95.28	81.60	80.72	85.86

(b) $\sigma = 0.05$

k	S (%)	e^- (%)	e^+ (%)	T (%)
1	91.39	67.50	69.88	76.26
3	93.52	75.58	76.67	81.93
5	94.19	78.43	79.00	83.88
10	94.45	80.40	81.29	85.38
20	94.30	81.29	81.86	85.81

(c) $\sigma = 0.1$

k	S (%)	e^- (%)	e^+ (%)	T (%)
1	80.15	61.95	68.38	70.16
3	82.37	66.98	76.00	75.12
5	83.51	70.81	79.00	77.78
10	83.46	74.70	81.39	79.85
20	84.81	75.69	82.12	80.87

(d) $\sigma = 0.3$

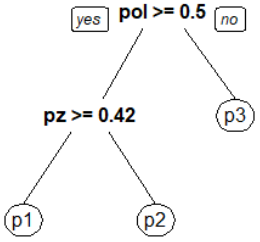
k	S (%)	e^- (%)	e^+ (%)	T (%)
1	65.73	57.18	64.18	62.36
3	68.53	61.38	71.07	67.00
5	68.84	65.01	75.01	69.62
10	70.76	67.24	78.02	72.01
20	70.50	68.22	78.90	72.54

(e) $\sigma = 0.5$

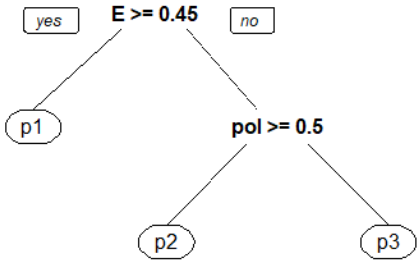
Table 3.8: Accuracy of Decision Tree Model on Simple Particle Dataset With Charge Indicator On Different Noise Level of σ

σ	*Scattered* (%)	*Paired* e^+ (%)	*Paired* e^- (%)	*Total* (%)
0	95.69	97.87	100	97.85
0.05	94.66	82.68	82.89	86.74
0.1	92.89	82.21	80.45	85.19
0.3	80.35	77.09	82.48	79.97
0.5	62.72	68.42	83.61	71.59

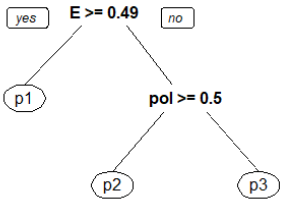
the accuracy drops over15% when I add a small level of noise, it holds quite well as I increase the noise standard deviation. Even when σ is as large as 0.3, the overall accuracy can still maintain around 80%.



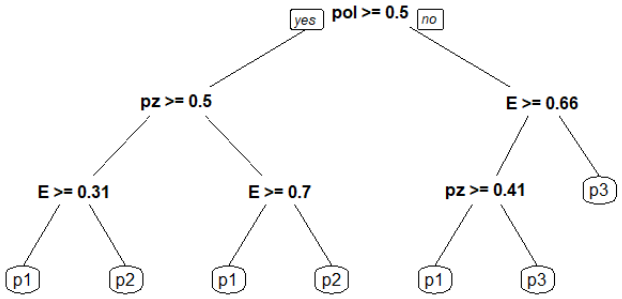
(a) Decision Tree For Particle Dataset With Charge Indicator But Without Noise



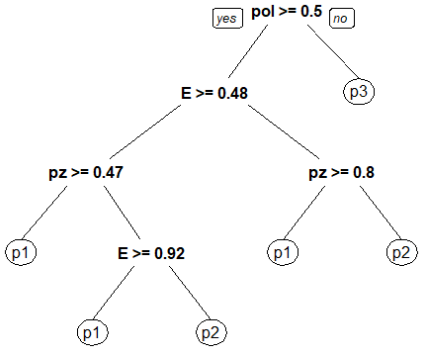
(b) Decision Tree For Simple Particle Dataset With Charge Indicator and Noise $\sigma = 0.05$



(c) Decision Tree For Simple Particle Dataset With Charge Indicator and Noise $\sigma = 0.1$



(d) Decision Tree For Simple Particle Dataset With Charge Indicator and Noise $\sigma = 0.3$



(e) Decision Tree For Simple Particle Dataset With Charge Indicator and Noise $\sigma = 0.5$

Figure 3.2: Decision Tree Generated For Particle Dataset With Charge Indicator

Chapter 4

Particle Track-Fitting Model

4.1 Overview

After testing both the kNN model and the Decision Tree model on the relatively simple particle four-vector database, I move on a more complicated case. In this section, I try to simulate an accelerated electron or positron being fired into two dimensional XY plane with a uniformed magnetic field pointed in the z direction. The plane is equipped with directors that will be triggered if the particle passes through it. The ultimate goal for this setting is to explore the potential ability of a trained program to read the triggered detectors patterns and then automatically classify the particles and predict the input vector of the particle. In other words, I want to tell the entry angle and velocity of the particles, along with their charge.

4.2 Data Simulation Setting

The first question is how to repeatedly simulate this setting using programming languages like Python and R. Our approach is to create an ordinary differential equation that describes such events. Then I can solve and plot the particles' trajectories using these languages with relative ease. Since the particle will travel in the XY plane that is perpendicular to the uniform field direction Z, I expected the particles to rotate in

the XY plane. The particle's motion can be described using Equation 4.1.

$$\frac{d\vec{v}}{dt} = \frac{q}{m}(\vec{v} \times \vec{B}) \tag{4.1}$$

Using such an equation, I set the field strength $|B| = 1$, set particle charge to the unite charge and set the mass to one unit.Then I can utilize the ODE module in Python to generate the particle's trajectory over time for a given random entry angle and velocity. The specific code written for this purpose is attached to the Appendix A. After generating the trajectory, I then need to decide the resolution of the detector. We expected that the finer resolution is, the more accurate the model will be. However, it will require more computing power and more time to be trained. So the rule of thumb here is to use a lest amount of detectors to get the most accurate result. We decided to pick the detector resolution of 100×100 as an ideal starting point. At this stage, to keep things simple and easy to handle, I assume every particle enters the field at the same point. Then I can convert the particles' trajectories into triggered detector patterns. Figure 4.1 shows three examples of such patterns.

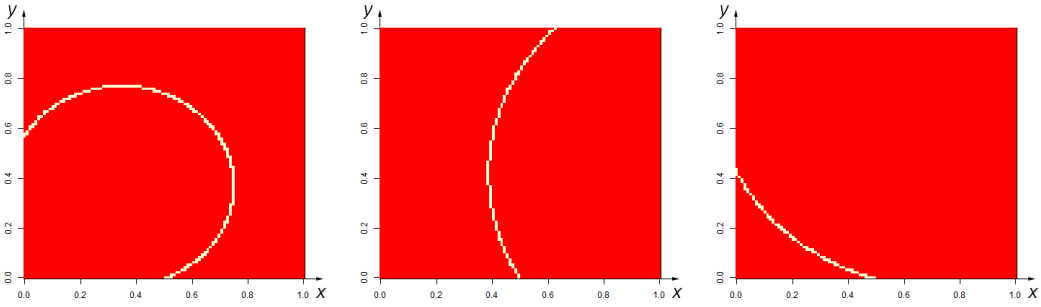


Figure 4.1: Sampled Triggered Detectors Pattens With A Resolution Of 100×100

4.3 kNN-Classification

Due to the nature of this simulation setting, the decision tree method may not be a good choice in term of predicting the charge and momentum of each particle. If I treat

each detector as a variable, then, each observation will have 10000 scattered variables, which will require tremendous calculation power. Also, if such method is utilized, the relative position between detectors will be lost. In general, the kNN method is a much better algorithm for this specific simulation setting, for it can treat the whole image as a single vector and thus preserve some relative position information even without more advanced smoothing. The reason I chose not to do smoothing on this simulation data is that I ultimately want to use the technique to process data live. So, the trained model should be able to handle at list one observation per second. This means that techniques like smoothing that require time to process the raw observation data before classification are not suitable for this case. So, I simply stretch the detector image to a single one vector line by line and then apply the kNN algorithm to predict both the charge and momentum of each particle. With the complexity of this setting, it becomes infeasible to use a normal computer to train the model. Especially when the training dataset becomes extremely large, a more powerful computer is required. So, I utilized the High Performance Computing unit at William and Mary along with parallel computing code to speed up the process.

The charge of each particle can be predicted in a similar manner as in the previous case. However, since momentum is continuous rather than discrete, I need to convert it into small blocks that can be classified by the kNN method. We predict the momentum on the x and y directions separately and divide them into 20 small blocks from the lower to the upper limit, with each block occupying 0.1 unit. This is a fairly large resolution. But for our purpose, it should do just fine. Then, I start to train the model with 850 labeled observations, and increase the training data size to 4250 sets and eventually 8500 set. For each dataset, I record the total time for training, accuracy in term of blocks and the actual difference between the predicted momentum and actual momentum. The result is shown below. Figure 4.2, Figure 4.3 and Figure 4.4 shows

the histograms and corresponding probability density lines of the difference between predicted and actual momentum in both the x and y directions, with $k = \{1, 3, 5, 10\}$. Table 4.1 shows a more detailed accuracy of each different setting along with the time required for training and predicting each dataset. In this table, 'T' stands for time, 'k' stands for k-number, and 'D' stands for the direction of momentum. The accuracy is presented as the percentage of predicted momentum that falls within 0.1, 0.2, 0.3, or 0.4 unit of the actual momentum. The high-performance computer cluster utilized in this research contains 210-core Intel Xeon E5-2640 v4 processors with a clock speed of 2.4 GHz. In this section, I used 12 threads for each model with different training size. Under this condition, it is reasonable to expect that same task would take ten times longer to calculate on a normal computer without parallel computing. So, for this specific simulation, the training and predicting of 10000 sets of data would cost around 19 hours to finish on a typical personal computer. So, this model required more advanced computing power.

Figure 4.2 - Figure 4.4 provide a very intuitive perspective of the result. If we just look at Figure 4.2 alone, we can find that if we choose a smaller k-number, the histogram would look more concentrated to the middle, which is the zero point. This indicates that a bigger k-number seems to decrease the prediction accuracy of the model. If we compared Figure 4.2 with Figure 4.3 and Figure 4.4, it is very clear that data size has a very positive impact on the model accuracy. the bigger the model is, the more concentrated are these histograms. Compare all the different setting, it seems that a data size of 10000 with $k = 1$ offer a most concentrated histogram and thus is more accurate. This can be confirmed by Table 4.1. Under this setting, the model is been able to predict over 95 percent of the momentum to be within ± 0.4 of the actual momentum and predict over 72 percent of them within ± 0.1 range of the actual momentum.

There are other questions need to be considered. What can we expect if we further increase the data size? How much time will it cost to run an even bigger dataset? We then can plot the biggest average accuracy (within ± 0.1) of both x and y momentum in each setting with the corresponding data size and fit the plot with a logarithmic function as shown in Equation 4.2, where y is accuracy and x represent data size.

$$y = a + b \times \ln(x) \tag{4.2}$$

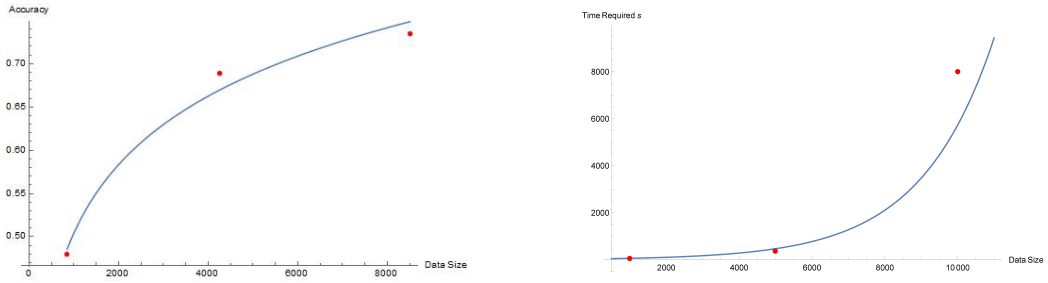
$$y = -0.28318 + 0.114029 \times \ln(x) \tag{4.3}$$

The result of this fit is shown in Equation 4.3. Then, we can do a simple prediction. To get an accuracy of 95 percent in ± 0.1 of the actual momentum, the expected data size need to be 49742, which is 5 times bigger than the biggest data size we have tried. Using similar method, we can fit an exponential function to describe the relationship between data size and required calculation time. The functional form is described in Equation 4.4, where y is required time and x represent data size.

$$y = a \times e^{bx} \tag{4.4}$$

$$y = 38.581 \times e^{0.0005x} \tag{4.5}$$

The result of this fit is shown in Equation 4.5. Then, we can estimate that with a data size of 50000, the time required to train the model would be 2.77802×10^{12} seconds, which is 88090 years. Apparently, we cannot wait that long. However, if we increase the thread count used in the parallel computing or even utilized GPU parallel computing, we can still deal with a smaller size within a relatively short time. If we chose a data size of 30000, the expected accuracy would be around 90 percent, and the calculation time could be decreased to 72 days with 240 computing threads.

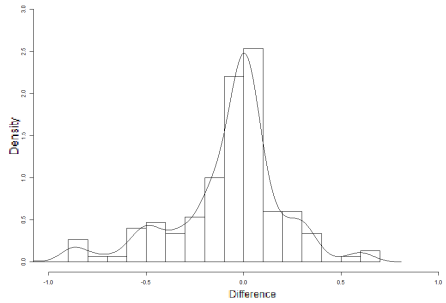


(a) Model Accuracy vs. Data Size

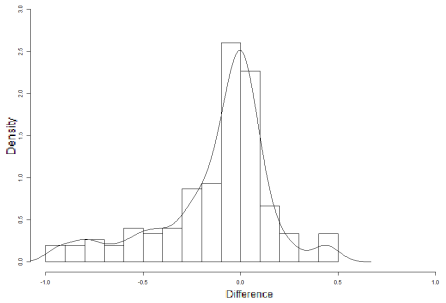
(b) Required Time vs. Data Size

Figure 4.2: Fitted curve for relationship between accuracy, data size and required time

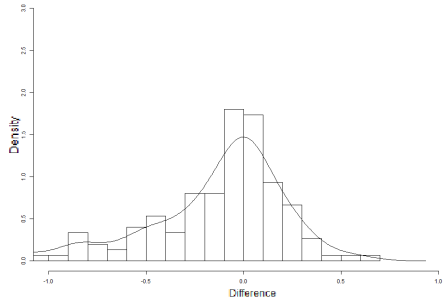
In the future, with powerful computers or advanced machine learning methods, the process of such task would only become faster and easier.



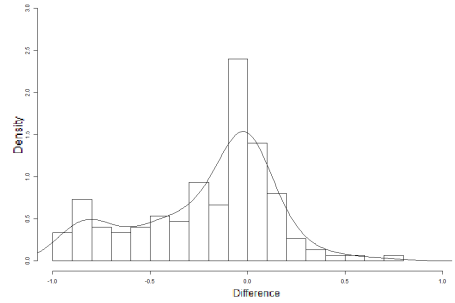
(a) In x direction, choosing $k = 1$



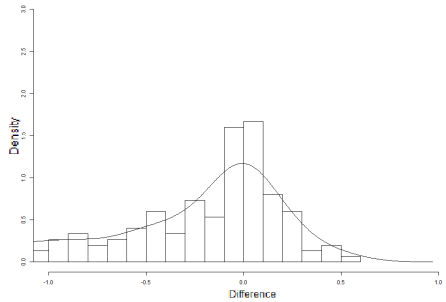
(b) In y direction, choosing $k = 1$



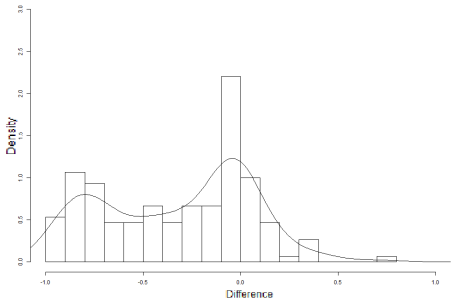
(c) In x direction, choosing $k = 3$



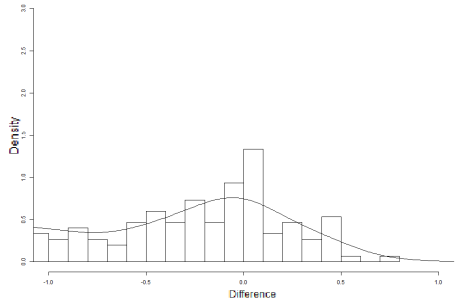
(d) In y direction, choosing $k = 3$



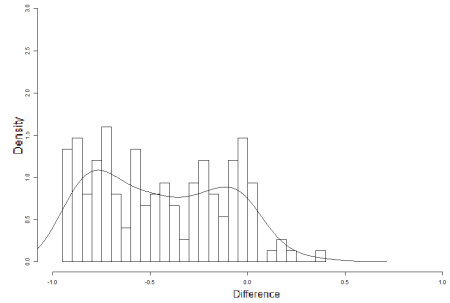
(e) In x direction, choosing $k = 5$



(f) In y direction, choosing $k = 5$

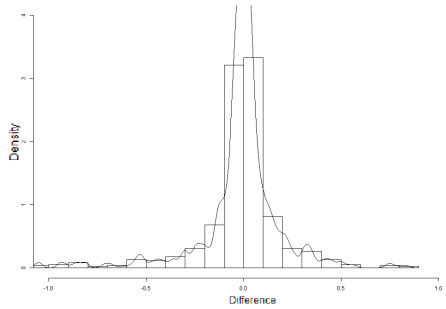


(g) In x direction, choosing $k = 10$

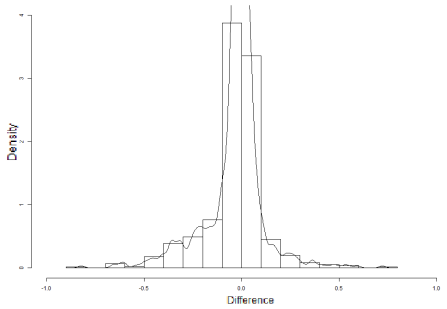


(h) In y direction, choosing $k = 10$

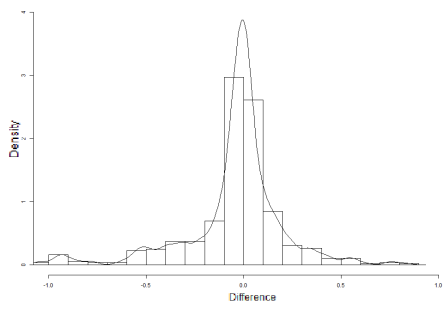
Figure 4.3: Histograms and corresponding probability density lines of the difference between predicted and actual momentum using kNN models with 1000 sets of data



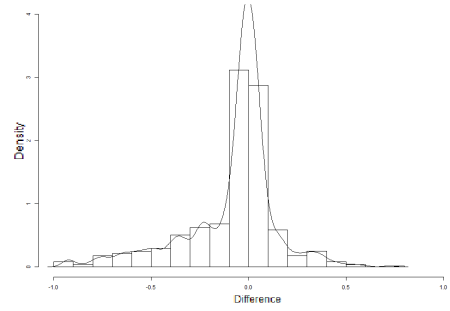
(a) In x direction, choosing $k = 1$



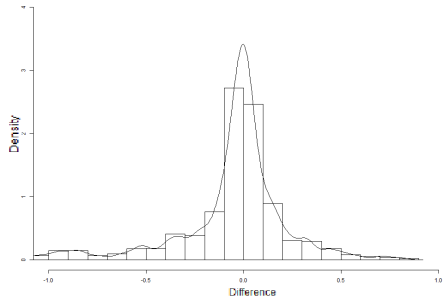
(b) In y direction, choosing $k = 1$



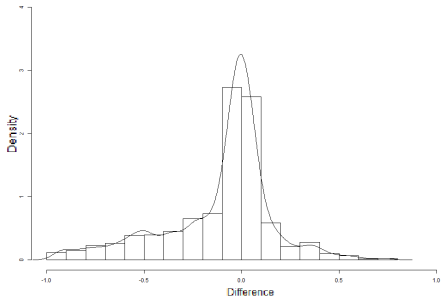
(c) In x direction, choosing $k = 3$



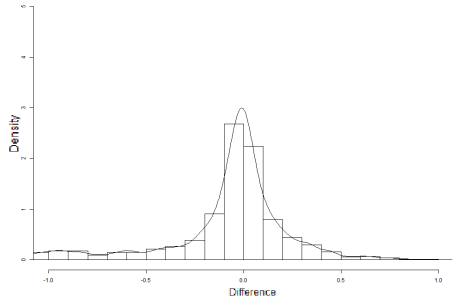
(d) In y direction, choosing $k = 3$



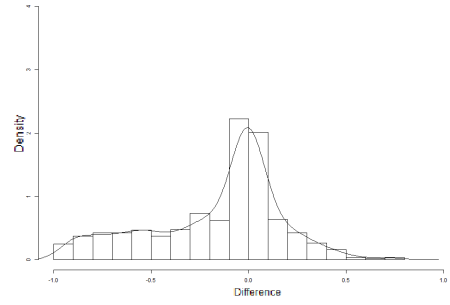
(e) In x direction, choosing $k = 5$



(f) In y direction, choosing $k = 5$

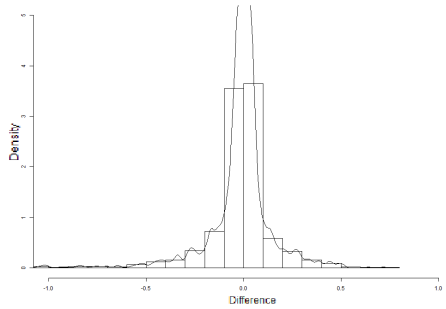


(g) In x direction, choosing $k = 10$

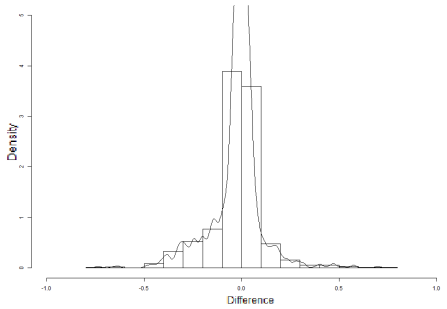


(h) In y direction, choosing $k = 10$

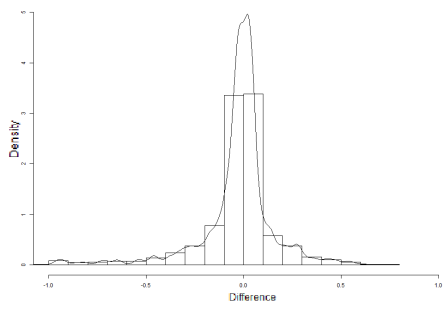
Figure 4.4: Histograms and corresponding probability density lines of the difference between predicted and actual momentum using kNN models with 5000 sets of data



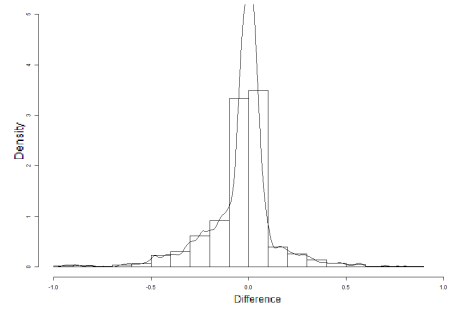
(a) In x direction, choosing $k = 1$



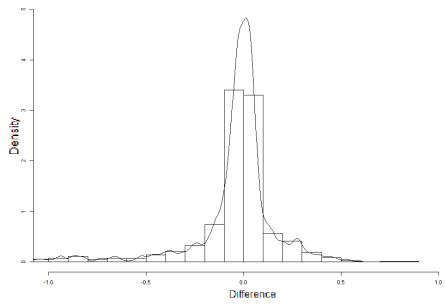
(b) In y direction, choosing $k = 1$



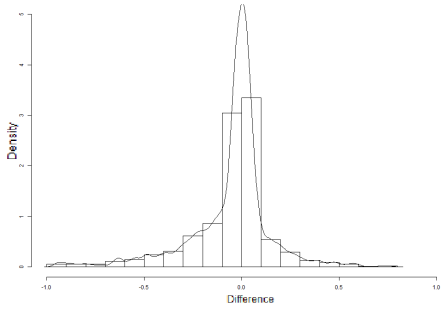
(c) In x direction, choosing $k = 3$



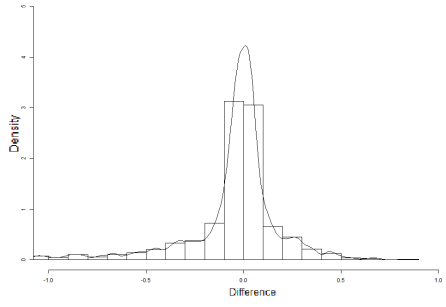
(d) In y direction, choosing $k = 3$



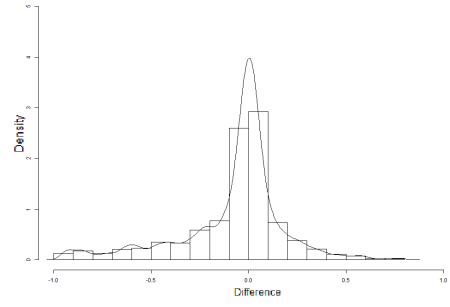
(e) In x direction, choosing $k = 5$



(f) In y direction, choosing $k = 5$



(g) In x direction, choosing $k = 10$



(h) In y direction, choosing $k = 10$

Figure 4.5: Histograms and corresponding probability density lines of the difference between predicted and actual momentum using kNN models with 10000 sets of data

Table 4.1: Accuracy of kNN model with different training size and k numbers in predicting particle momentum in both x and y directions

Size	T(s)	T(mins)	k	D	% ±0.1	Within ±0.2	Range ±0.3	of ±0.4
1000	57.42	0.96	1	x	0.473	0.633	0.747	0.813
				y	0.487	0.647	0.767	0.807
			3	x	0.353	0.527	0.673	0.733
				y	0.380	0.527	0.647	0.707
			5	x	0.327	0.460	0.593	0.640
				y	0.320	0.433	0.507	0.580
			10	x	0.227	0.307	0.427	0.500
				y	0.180	0.267	0.380	0.433
5000	768.38	12.81	1	x	0.655	0.804	0.865	0.909
				y	0.724	0.845	0.915	0.961
			3	x	0.559	0.713	0.781	0.845
				y	0.600	0.727	0.807	0.883
			5	x	0.519	0.684	0.753	0.824
				y	0.532	0.664	0.751	0.824
			10	x	0.492	0.663	0.745	0.801
				y	0.424	0.551	0.667	0.741
10000	7992.77	133.21	1	x	0.721	0.851	0.919	0.951
				y	0.749	0.873	0.941	0.979
			3	x	0.675	0.811	0.887	0.927
				y	0.683	0.814	0.901	0.946
			5	x	0.671	0.802	0.875	0.915
				y	0.639	0.779	0.870	0.915
			10	x	0.619	0.757	0.839	0.893
				y	0.553	0.703	0.800	0.855

Chapter 5

Conclusions And Future Plans

5.1 Conclusions

Through the two simulation settings, I find that both the decision tree algorithm and the kNN method have promising potential in certain type Physics experimental data processing areas. The decision tree method works great with tasks like classifying particle charges, especially when there are multiple independent variables inputs for each observation. With adequate training data, decision tree method already reaches over 97 percent accuracy in the simple particle setting. I believe with even a bigger training size, and longer training hours, it can reach an even higher accuracy. However, it does have certain limitations, as we see in the second simulation setting. kNN, on the other hand, is more versatile compared to the decision tree. In the simple particles setting, it generates a similar accuracy compared to the decision tree method and seems to be slightly more robust to noise when the k number is increased. Another aspect that needs to be discussed here is that machine learning has a very special calculation power and time requirement. As we see from the particle Track-Fitting setting to acquired a good accuracy, an enormous amount of calculation is required, compared to conventionally curve fitting method. However, all the extra calculation are used for training the model, which is detachable from the actual Physics experi-

ment. So, this calculation can be done before the experiment using simulation data, without occupying actual time in the experiment process. Once the experiment is started, the trained model can easily be fast enough to predict observations at the scene. I would conclude that as for now, both methods are not ready to be used as the sole data processing method in any experiment. However, for now, they can be applied to facilitate and reconfirm the experimental data process. Machine learning, due to its unique advantage, definitely has the potential to become the main power in Physics experimental data processing and even fully take over the experimental data processing in the future.

5.2 Future Plans

As for near future, it would be nice if a even bigger training dataset can be used to train the kNN model for the Particle Track-Fitting setting. I would like to know that for say, if 100 hours is spent on training, will the accuracy reached my expectation? Once a desired accuracy is achieved, a downgrade on detector resolution will be necessary to explore the character of how machine learning handles such changes. Other necessary steps in future study include exploring more advanced methods other than decision tree and the kNN algorithms, for they are still relativity simple ways to do machine learning. A more complicated method may be able to offer a better accuracy while requiring less calculation compared to my method, under similar conditions. With the development of more and more powerful computers and computing methods, machine learning is a very encouraging direction in term of Physics experimental data processing.

Appendix A

Selected Code Written For This Research

A.1 Code For Simulating Particle 2-D Movement In Magnetic Field

The following is the Python code that simulate a electron or a positron traveling in a plane that is perpendicular to a uniformed magnetic field.

```
import numpy as np
import random
import csv
import matplotlib.pyplot as plt
from scipy.integrate import ode
from scipy.stats import bernoulli as bern
info = []

def newton(t, Y, q, m, B):
    x, y = Y[0], Y[1]
    u, v = Y[2], Y[3]
    alpha = q / m * B
    return np.array([u, v, -alpha * v, alpha * u])

resolution = 100
num = 10000
inter = 1/resolution
mass = 1
Bfield = 1
charge = bern.rvs(0.5,loc=0, size=num)

vx = np.random.uniform(-1,1,num)
vy = np.random.uniform(0,1,num)
x0 = np.array([0.5, 0])

```

```
for ri in range(num):
    v0 = np.array([vx[ri], vy[ri]])
    singlecharge = charge[ri]
    vran = list(v0)
    velo = np.sqrt((vran[0])**2+(vran[1])**2)
    angle = np.arctan((vran[1]/vran[0]))/2/np.pi*360
    r = ode(newton).set_integrator('dopri5')
    t0 = 0

    initial_conditions = np.concatenate((x0, v0))
    r.set_initial_value(initial_conditions, t0).set_f_params\
  (singlecharge, mass, Bfield)
    positions = []
    t1 = 10
    dt = 0.01

    while r.successful() and r.t < t1:
        r.integrate(r.t+dt)
        positions.append(r.y[:2])
    positions = np.array(positions)
    pointinrangelist=[]

    for point in positions:
        if 0 <= point[0] <= 1 and 0 <= point[1] <= 1:
            pointinrangelist.append(list(point))
        else:
            break

    actived = []
    detline = [0]*resolution
    alldetect = []

    for i in range (resolution):
        alldetect.append(detline[:])

    for point in pointinrangelist:
        xnum = int(point[0]//inter)
        ynum = int(point[1]//inter)
        actived.append([ynum,xnum])

    for dete in actived:
        alldetect[dete[0]][dete[1]] = 1
    alldet=[]

    for line in alldetect:
        alldet += line
    info.append([vx[ri], int(vx[ri]//0.1+11), vy[ri], \
  int(vy[ri]//0.1+11), singlecharge]+alldet)

with open('simulation.csv','w',newline='') as simucsv:
    writer = csv.writer(simucsv)
```

```
77          writer.writerows(info)
78
```

A.2 Code For Using kNN Algorithm To Classify Particles In The Accelerator Model

The following is the R code that use the kNN algorithm to classify particles in th Track-Fitting model.

```
1  rm(list = ls())
2  library(class)
3
4  dataraw = matrix(unlist(read.csv('simulation.csv', header = FALSE)),
     ncol=10005)
5
6  nr=dim(dataraw)[1]
7  nc=dim(dataraw)[2]
8
9  train = dataraw[1:(0.98*nr),6:10005]
10 tranwithcharge = dataraw[1:(0.98*nr),5:10005]
11 test = dataraw[(0.98*nr+1):nr,6:10005]
12 testwithcharge = dataraw[(0.98*nr+1):nr,5:10005]
13
14 cl = dataraw[1:(0.98*nr),5]
15 testcl = dataraw[(0.98*nr+1):nr,5]
16
17
18 acvx = c()
19 vxp =
20 clvx = dataraw[1:(0.98*nr),2]
21 testclvx = dataraw[(0.98*nr+1):nr,2]
22 klist = c(1)
23 for (i in klist){
24 knn = knn(train=tranwithcharge, test=testwithcharge, cl=clvx, k=i)
25 acvx= c(acvx,100*(sum(testclvx== knn)+sum(testclvx+1== knn)+sum(
     testclvx-1== knn))/(0.02*nr))
26 vxp=c(vxp,c(knn))
27 }
28
29 acvy = c()
30 vyp=c()
31 clvy = dataraw[1:(0.98*nr),2]
32 testclvy = dataraw[(0.98*nr+1):nr,2]
33 klist = c(1)
34 for (i in klist){
35 knn = knn(train=tranwithcharge, test=testwithcharge, cl=clvy, k=i)
36 acvy= c(acvy,100*(sum(testclvy== knn)+sum(testclvy+1== knn)+sum(
     testclvy-1== knn))/(0.02*nr))
```

```
vyp=c(vyp,c(knn))
}

acutot = cbind( acvx, acvy)
acutot
result=matrix(c(testcl,chargep,testclvx,vxp,testclvy,vyp),nrow=200)

write.csv(result, 'resultwithcharge.csv')
write.csv(acutot, 'acuwithcharge.csv')

```

A.3 Code For Using kNN Algorithm To Predict Particles Charge And Momentum Utilizing Parallel Computing In The Accelerator Model

The following is the R code that use kNN method while utilizing parallel computing to classify particles in the Track-Fitting model. This is meant to be used under a Linux kernel environment, like the HPC in William and Mary.

```

library(foreach)
library(doMC)
library(class)
registerDoMC(12)

ptm <- proc.time()
dataraw = matrix(unlist(read.csv('s10000.csv', header = FALSE)),ncol
  =10005)

nr=dim(dataraw)[1]
nc=dim(dataraw)[2]

train = dataraw[1:(0.85*nr),6:10005]
tranwithcharge = dataraw[1:(0.85*nr),5:10005]
test = dataraw[(0.85*nr+1):nr,6:10005]
testwithcharge = dataraw[(0.85*nr+1):nr,5:10005]

cl = dataraw[1:(0.85*nr),5]
testcl = dataraw[(0.85*nr+1):nr,5]

accharge = c()
chargep=c()
klist = c(1,3,5,10,20)
chargep <- foreach (i = klist) %dopar% {
  knn = knn(train=train, test=test, cl=cl, k=i)
  accharge= c(accharge,100*sum(testcl== knn)/(0.15*nr))
```

```
    accharge
  }

  acvx = c()
  vxp = c()
  clvx = dataraw[1:(0.85*nr),2]
  testclvx = dataraw[(0.85*nr+1):nr,2]
  tlen=length(testclvx)
  testmx= dataraw[(0.85*nr+1):nr,1]
  vxp <- foreach (i = klist) %dopar% {
    knn = knn(train=tranwithcharge, test=testwithcharge, cl=clvx, k=i)
    acvx= c(acvx,100*(sum(testclvx== knn)+sum(testclvx+1== knn)+sum(
   testclvx-1== knn))/(0.15*nr))
    kp=(c(knn)-11)*0.1+0.05
    c(acvx,kp-testmx)
  }

  acvy = c()
  vyp=c()
  clvy = dataraw[1:(0.85*nr),4]
  testclvy = dataraw[(0.85*nr+1):nr,4]
  testmy= dataraw[(0.85*nr+1):nr,3]
  vyp <- foreach (i = klist) %dopar% {
    knn = knn(train=tranwithcharge, test=testwithcharge, cl=clvy, k=i)
    acvy= c(acvy,100*(sum(testclvy== knn)+sum(testclvy+1== knn)+sum(
   testclvy-1== knn))/(0.15*nr))
    kp=(c(knn)-11)*0.1+0.05
    c(acvy,kp-testmy)
  }

  acutot = cbind(chargep, vxp[c(1,tlen+2,2*tlen+3,3*tlen+4)], vyp[c(1,
   tlen+2,2*tlen+3,3*tlen+4)])
  time = proc.time() - ptm
  write.csv(acutot, 'acu10000par.csv')
  write.csv(c(time),'time10.csv')
  diff=cbind(vxp,vyp)
  write.csv(diff,'diff10000.csv')


```

References

[1] Biau, Devroye, Devroye, Luc, and SpringerLink. Lectures on the Nearest Neighbor Method by Grard Biau, Luc Devroye. Springer Series in the Data Sciences. 2015

[2] Cover, T., and P. Hart. "Nearest Neighbor Pattern Classification." Information Theory, IEEE Transactions on 13, no. 1 (1967): 21-27.

[3] R Core Team (2013). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

[4] Steele, Brian, John Chandler, and Swarna Reddy. Algorithms for Data Science / Brian Steele, John Chandler, Swarna Reddy. 2016.

[5] Su, Feng ; Xue, Like. Graph Learning on K Nearest Neighbours for Automatic Image Annotation. Proceedings of the 5th ACM on International Conference on Multimedia Retrieval, 2015, 403-10.

[6] Venables, W. N. & Ripley, B. D. (2002) Modern Applied Statistics with S. Fourth Edition. Springer, New York. ISBN 0-387-95457-0.

[7] Wang, Zhai, and Zhai, Junhai. Learning with Uncertainty / Xizhao Wang, Junhai Zhai. 2017.