CS7641 Machine Learning Fall 2021
Project 1 : Supervised Learning

Ruhan Li
(rli445)

Abstract

Five algorithms (Decision Tree, Neural Network (ANN), Boosting, Support Vector Machine (SVM), and K-Nearest Neighbor (KNN)) were applied to two data sets (Customer Satisfaction Questionnaire and Phishing Websites). This report mainly focuses on the hyper-parameter tuning and model analysis of all five algorithms. Their performances have been compared as well. The implement of algorithms are based on python and with public libraries, including Scikit-Learn pandas, and numpy.
Link of code: https://github.com/Hanlarious/ML

Introduction of data sets

Basic information

I have chosen two data sets from different fields. The first one is a customer satisfaction questionnaire from an airline. The other one is about phishing websites.

Interesting Data Sets

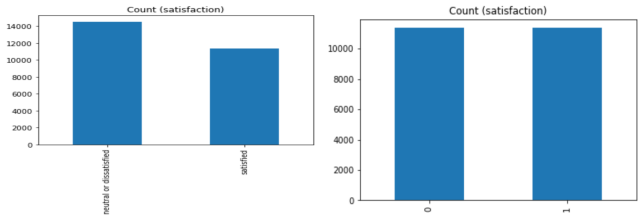
When selecting data sets, I fitted those data sets into different algorithms. I found that these two data sets had different performance with different algorithms. And some of the curves that I drew looked "strange", which made me think what was the reason behind them.
Further more, these two data sets have different reality demands (one needs more accuracy, while the other one needs to limit cost). Therefore, it is worth discussing and comparing when selecting the final models.

Data pre-processing

Data set 1: Customer Satisfaction Questionnaire

This data set has 25976 entries with 25 columns. I did the following preparation work:

- **Remove the entries with missing data:** There were 83 missing data in the data set. Considering the size of the data set, I decided to remove these entries with missing data.
- **Make data set balanced using *Under-sampling* technique:** The data set is slightly unbalanced with 14528 entries in the "neutral or dissatisfied" class, and 11365 entries in the "satisfied" class. Given the size of the data set, I decided to randomly select 11365 samples from the "neutral or dissatisfied" class. To make sure the same samples were chosen when every time I re-run the code, I used the parameter *random_state()* to fix the seed. Further, in the balanced data set, the labels of satisfaction have been changed to "0": neutral or dissatisfied, and "1": satisfied. (Figure 1)



(a) unbalanced data set (b) balanced data set

Figure 1: Data Balance Bar

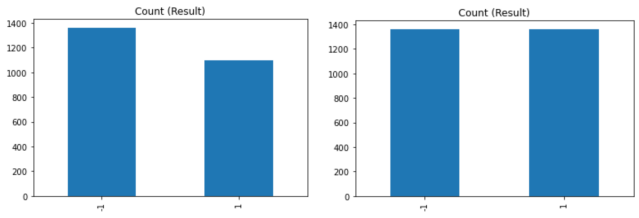
- **Drop irrelevant columns:** This data set includes some useless columns that can have a negative effect on the results, including serial numbers and user IDs. Therefore, I need to remove these columns before processing data. After completing the above operations, I have a balanced data set with 25893 entries and 23 columns (including the column to be predicted), in which 18 of them are numerical features and 4 of them are categorical features.
- **Assign numbers to categorical features:** Because all the categorical features have no more than 3 unique values, I decided to manually change their label names to numbers like 0,1,2.
- **Split data into *training set* and *test set*:** For this data set, I used 70% data for training, and 30% for test. To make the model selecting process unbiased, the test set was not touched until the end step – getting test scores.
- **Scale data:** The features in this data set include large numbers like flight distance, as well as small numbers like gender (0 for female and 1 for male). Without data scaling, the features with larger range of numbers will have more weight compared with the features with smaller range of numbers, which will likely effect the performance of models. Therefore, I used *StandardScaler()*

function to scale the data. I have already separated the "features"(x) and the column that needs to be predicted (y), the scaling was applied only to the "features".

Data set 2: Phishing Websites

This data set has 2456 entries with 31 columns, and no missing data. I did the following preparation work:

- **Make data set balanced using *Over-sampling* technique:** Because the size of this data set is relatively small, I decided to randomly select entries in the minority class multiple times until the amount reaches the number of entries in the majority class. (Figure 2)



(a) unbalanced data set (b) balanced data set

Figure 2: Data Balance Bar

  After this, I have a balanced data set with 2724 entries and 30 features, excluding the column that needs to be predicted.

- **Split data into *training set* and *test set*:** For this data set, I used 80% data for training, and 20% for test. To make the model selecting process unbiased, the test set was not touched until the end step – getting test scores.
- **Scale data:** Even though this data set does not have a huge range of features, I still scaled the data.

General Implement

Tuning Process

For both data sets and five algorithms, I followed the following progress:

1. **Fit training set with default values:** For the first four algorithms, the only hyper parameter that I used at this point was *random_state()*. The KNN algorithm does not need to fix the seed. After this step, I would get the initial Cross Validation scores (f1 and precision), training accuracy score, training f1 score, training balanced accuracy score,training recall score, training precision score, and the confusion matrix.
2. **Run grid search:** Use *GridSearchCV()* to find the best values of hyper parameters with given ranges. I ran this function multiple times with different ranges and selections of hyper parameters.
3. **Find the best hyper parameter:** Fit model with hyper parameter combinations given by *GridSearchCV()* in turn and record their Cross Validation scores (f1 and precision) respectively. Pick the combination with the highest Cross Validation score. In my case, The model with higher f1 Cross Validation score normally had higher precision Cross Validation score too. If they did not, I would take a look at their confusion matrices and choose the one with fewer False Positive cases. Because I feel like the False Positive cases are more harmful than False Negative cases, thinking about a bank grants loans to people who supposed to be classified as "not qualified" (False Positive) versus a bank does not grant loans to people who supposed to be classified as "qualified" (False Negative). Therefore, my purpose was to keep as less False Positive cases as possible.
4. **Use of *learning curve* and *validation curves*:** Draw a *learning curve* and *validation curves* based on the current best model. I focused more on *validation curves* at this point. Because sometimes I gave discrete ranges *(i.e. 'learning_rate' : [0.001, 0.05, 0.1])* , which made me potentially miss a better value.
   Another situation is that the *GridSearchCV()* may have found a local maxima for me. Therefore, I would observe the peak and the shape of *validation curves*. If I found a point with higher y-value than my current best hyper parameter value, or a potential trend in some way, I would expand my range towards that direction and run *GridSearchCV()* again.
   Then I would repeat STEP 3 to fit training set to models with new best hyper parameter values, and compare the Cross Validation scores between the old and new best models.
   Furthermore, the observation of *learning curve* was also important. It helped to measure the quality of models. I would analyze the behavior of the *learning curve* for each model, to decide if re-run *GridSearchCV()* was necessary.
5. **Select the best model:** At this point, I already had the Cross Validation scores of all models, I would select the one with the highest Cross Validation score as my final model. Sometimes I needed to check their confusion confusion matrices too, because of the same reason that I have mentioned in STEP 3.
6. ***Iteration Curve:*** For the ANN and SVM algorithms, I also made *Iteration Curves* based on the selected best model. As I understand, the purpose of the *Iteration Curve* is to evaluate when to stop iterating. Ideally, we would like the iteration to stop when the error rate is at the lowest point. Some studies suggested to make loss curves, however, I was not able to make the loss curves work for ANN and SVM. Instead, I made the validation curves for the *'max_iter'*. So the curve shows the change of accuracy of the model with the increase of iterations. When the accuracy reaches a high and stable value, I interpret it as the error rate has reached a low and stable level. From a time-saving perspective, this is the good timing for stopping iteration.
   While tuning the hyper parameters, I already have a general ideas about how many iterations it took the model to reach a relatively stable performance. Therefore, I would first give a bigger range to try to confirm my guessing, then adjust the iteration range to make the curve more precise.

Techniques

- **Cross Validation** I had tried to run some models on data sets when selecting them, and I found that most of the data sets have the problem of over-fitting – returned unbelievably high accuracy scores. But when I did the k-fold Cross Validation to the data set, the accuracy score dropped, which to me provided more accurate evaluations. Therefore, I did cross validation at every step when pruning the hyper parameters.
- **Evaluation Metrics:** Because I made both of my data sets balanced before tuning the hyper parameters, the accuracy scores, balanced accuracy scores and the f1 scores did not have obvious differences, therefore, I picked the *accuracy score* as one of my evaluation metrics. On top of that, I think the value of *True Positives / Total number of predicted Positives* is also important.
  Specifically, for the *Phishing Websites* data set, a secure website being classified as "unsafe" (False Negative)is more acceptable than undetected phishing websites (False Positive), because entering a phishing website sometimes causes serious damages. On the other hand, for the *Customer Satisfaction* data set, a dissatisfied customer being classified as "satisfied" (False Positive) could bring a worse impact on the company's customer service improvement, compared with a misclassified happy customer.
  As a result, I picked the *precision score* as my second evaluation metrics.
  To have a more comprehensive analysis of the model performance, I made two sets of *learning curves* ,*validation curves* and *iteration curves* (applied for ANN and SVM only) using both evaluation metrics for each algorithm and each data set. Most of the time, both evaluation metrics had similar behaviors, but sometimes the curves using *precision score* varied more drastically.
- **Apples to Apples** I used the following techniques to control variables: *1)* Use the same k value for the k-fold cross validation. Considering the sizes of these two data sets, I decided to set k to 5 for the *Customer Satisfaction* data set, and 10 for the *Phishing Websites*. *2)* I used the parameter *random_state* to fix the seed while making the data sets balanced, as well as fitting them into different models. *3)* I pre-processed the data set, and kept the training and test data sets separate and fixed, and used the same set of training and test set for every algorithm. *4)* When comparing between two evaluation metrics, I kept all parameters the same, including selected models and ranges, and changed only the scoring parameter (evaluation metrics).
- ***Validation Graphs:*** For the categorical hyper parameters, including *criterion* in Decision Tree, *activation function* in ANN, *kernel* in SVM, and *weights* in KNN, I made *Validation Graphs* (bar) instead of *Validation Curves*, because the bar graphs are clearer to me when comparing the scores.

Challenges

The first challenge was finding data sets. I have used *UCI Machine Learning Repository* and *Kaggle ML Data Sets* to search for interesting data sets. The problem was that you would not know if the data set was suitable until you fit them into some models.
The second challenge was the improvement of performance. I have tried more than 10 data sets with difference sizes, number of attributes, and types of features. However, all of them had really high initial accuracy scores when fitting in models with default hyper parameter values. This made finding good pairs of hyper parameter values to improve performance extremely difficult. After designing the tuning process, I spent most of the time running *GridSearchCV()* with different pairs of hyper parameter values, wishing to improve the performance more, with little success. I think the problem is that the data sets already had good performances with default hyper parameter values. I would try data sets with more complicated structures in the future, so that I have more room for improvement.

Algorithms

Decision Tree

- **Tuned hyper parameters:**
  - ***max depth (pruning):*** The major problem of Decision Tree algorithm that I need to handle is over-fitting. Specifically, the baseline model of my first data set had an initial accuracy score of 1, and a cross validation score of 0.91, which was obviously an over-fitting scenario. Therefore, pruning was necessary. The hyper parameter that I chose for pruning was *'max depth'*. By limiting the maximum depth, the training stops when the limit has been reached. Therefore, this hyper parameter should be tuned to help creating a good model.
  - ***min samples leaf*** The other hyper parameter that I chose to tune is the minimum leaf size. This is also a good method to avoid over-fitting, because technically the prediction accuracy can reach 100% if we allow every node to be a leaf, which over-fits the training data set, and will very likely to perform bad in the test data set. Therefore, the limit of minimum leaf size will avoid this situation effectively.
  - ***criterion:*** This parameter determines how the impurity of a split will be measured. I selected it as an additional possibility to improve the performance of the model.
- **Analyses: Data set 1 - *Customer Satisfaction***

  summary
  Training time: around 0.074s
  Validation time: around 0.003s
  Test accuracy score: 0.9403
  Performance improvement (CV scores): 3%
  The best combination of hyper parameter values returned by *GridSearchCV()* was:
  max depth :18, *min samples leaf*: 10, *criterion*: entropy.
  The CV score (f1/precision): 0.9412 /0.945.

  First of all, traning *CV scores* and the *test accuracy score* are very close, which means the problem of over-fitting is likely to be solved.

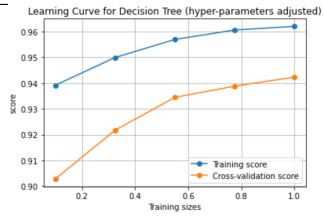
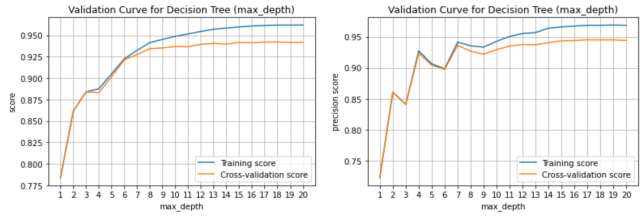


Figure 3: Decision Tree Learning Curve (1)

However, the *learning curve* (Figure 3) does not support my conclusion. The curves have high variances, law bias, and do not seem to converge. This means that the model can fit the known data well, but has bad stability, and may still have the problem of over-fitting. To further improve the performance, I will need to reduce the complexity of the model. Specifically, for Decision Tree algorithms, reducing the maximum depth or increasing the minimum leaf size should work.
The *validation curves* (Figure 4) show the performance of maximum depth with different numbers. Both curves suggest that the performance become relatively stable after reaching depth 15, and it is reasonable to pick 18 as the best value. The interesting point is that the curves of training and CV are almost the same before depth 7 in both graphs, especially with the evaluation metric of *precision score* (Figure 4(b)), despite the fact that the curves go up and down dramatically. It could be an coincident, but still interesting.



(a) Metric: accuracy scores (b) Metric: precision scores

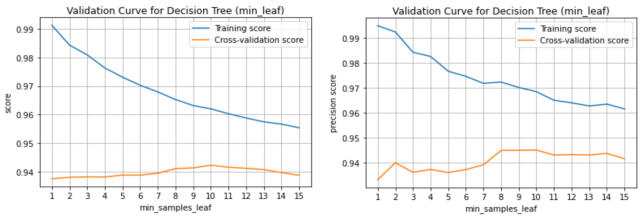
Figure 4: Validation Curves (max depth)

The *validation curves* (Figure 5) show the performance of different numbers of minimum leaf size. The curves based on the *accuracy score* is more gradual, whereas the cures based on the *precision score* vary more, but it is obvious in both graphs that the highest score is reached when the value is 10.

- **Analyses: Data set 2 - *Phishing Websites***

  summary
  Training time: around 0.004s
  Validation time: 0 (too small to be shown)
  Test accuracy score: 0.9669
  Performance improvement (CV scores): 3.5%

  After deciding which hyper parameters to tune, I did an experimental manual best-value-search for *max*



(a) Metric: accuracy scores (b) Metric: precision scores

Figure 5: Validation Curves (min samples leaf)

depth by plotting the Decision Tree Model Complexity based on *max depth* (Figure 6)

Figure 6: Decision Tree Model Complexity

From the graph, I figured that 6 could be the value with best performance for *max depth* .
On top of that, I applied the *GridSearchCV()* to search for the best value of minimum samples of leaf. I set the range to 0.5% to 5% of the training data set. And it returned me the value 11. so I built the model with maximum depth of 6 and minimum leaf size of 11, and fitted in with my training data set. Then I got a CV score of 0.9471, which was lower than the CV scores of the models with these two hyper parameters being changed separately.
Then I realized that scores are the results of a combination of different hyper parameters. Therefore, I decided not to tune single hyper parameters anymore. Instead, I would try different combinations of hyper parameters using *GridSearchCV()*.
The best parameter combination that I found was: *max depth* :12, *min samples leaf*: 1, *criterion*: gini.
Its CV score (f1/precision): 0.977/0.9731.

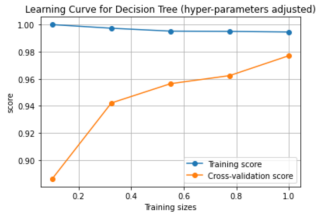
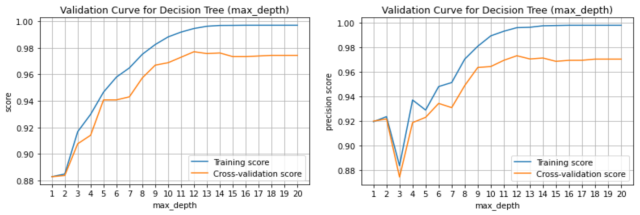


Figure 7: Decision Tree Learning Curve (2)

The *learning curve* (Figure 7) of final model shows a good bias-variance trade-off. In fact it is probably the best bias-variance trade-off that I have for this project. Even though

it is not converged yet, but it shows obvious trend to converge. Plus, the model has a good accuracy. To further improve the performance, I may need a larger size of data set.

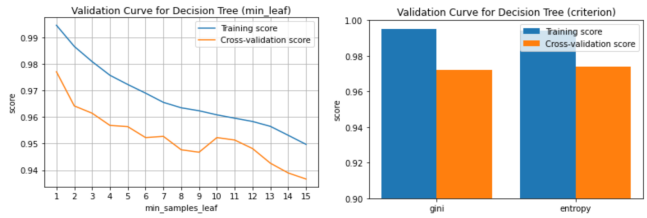
The *Validation Curves* (Figure 8) shows the maximum depth curves of the Decision Tree. Both graphs suggest that the best value of *max depth* should be 12, which matches the selected model. What is interesting is the dramatic drop in Figure 8(b) at value 3. This suggests that the model has more False Positive cases when the maximum depth is small, especially at 3.



(a) Metric: accuracy scores (b) Metric: precision scores

Figure 8: Validation Curves (max depth)

The *Validation Curves* (Figure 9) are showing the curves of minimum leaf size and criterion. (The curves with the metric of precision scores have very similar shape, so I feel like Figure 9 can represent them. In the rest of the report, I will only show both graphs based on different evaluation metrics when they are very different.)
It is obvious to conclude from the *Validation Curve* (Figure 9(a)) that the accuracy decreases while the minimum samples leaf increases. So the best value should be 1. For the criterion (Figure 9(b)), both *gini* and *entropy* have similar performance, so I will say it would not really matter which criterion to choose.



(a) HP: min samples leaf (b) HP: criterion

Figure 9: Validation Curves

Neural Network

- **Tuned hyper parameters:**
  - ***hidden layers:*** It is the core of Neural Network. More accurate predictions are expect with a good pair of layer amount and neuron amount at each layer.
  - ***learning rate:*** The model could take forever to converge with a small learning rate, and could swing around and keep missing the global maxima with a big learning rate. Therefore, finding a proper learning rate is crucial for building a good model.
  - ***max iter:*** This is mainly for the sake of time cost. If the *max iter* is too small, the training would stop before convergence happens, which brings in more error, but if it is too big, the training would not stop after convergence happens, which will increase training time without improving performance. Therefore, a proper max iter is also important to a successful model.
  - ***activation function:*** This is also an important element of Neural Network. The activation function should be chosen based upon the complexity of the data set. Whether a proper activation function has been used will greatly affect model performance.
- **Analyses: Data set 1 - *Customer Satisfaction***

summary
Training time: around 35.57s
Validation time: around 0.022s
Test accuracy score: 0.9469
Performance improvement (CV scores): 1.2%
The best combination of hyper parameter values returned by *GridSearchCV()* was:
hidden_layer_sizes: logistic, *min_samples_leaf*: (180,), *learning rate*: 0.001, *max_iter*: 300.
The CV score (f1/precision): 0.9468 /0.9491.

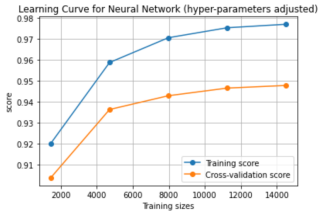
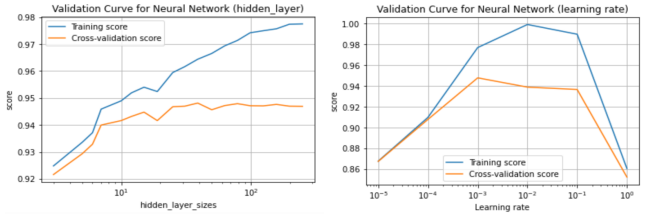


Figure 10: Neural Network Learning Curve (1)

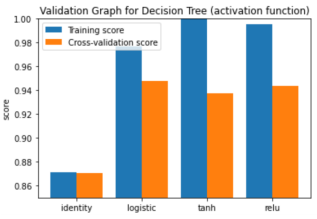
This is a *learning curve* (Figure 10) with high bias and high variance without a trend to converge. Although the accuracy is not that bad, the big difference between the *Training score* and the *Cross validation score* suggests the probability of over-fitting, as well as a poor stability.

For the *hidden layer sizes* validation curve (Figure 11(a)), the CV score becomes stable after the neutron number of 80 at a single layer, and there is a little peak at around 200. I feel like the value at 200 is similar to the value at 80, so 80 could be an alternative.
In fact, *GridSearchCV()* used to return (80,) as the best hyper parameter values, but that model had a lower CV score compared with the model with value (180,). I have also tried multiple hidden layers, including (20,20), (10,10,10), and (20,20,20). However, all these models did not have a better performance. To further improve the model performance, I may need to increase the number of neurons at each layer. I did not have time to try more larger numbers since it takes super long to do the grid search for Neural Network.
For the *learning rate* validation curve (Figure 11(b)), the

CV curve shows that the high scores is at the learning rate of 0.001 and the curve is relatively flat between 0.001 and 0.1, then drops significantly after that.
For the *activation function* validation curve (Figure 11(c)), it is obvious that the *identity function* performs the worst, whereas the other functions have similar performance in the validation set. Among the other three functions, *logistic function* has slightly better performance, so it was picked as the best activation function.



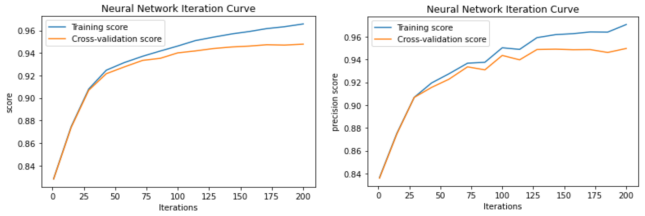
(a) HP: hidden_layer_sizes (b) HP: learning rate



(c) HP: activation function

Figure 11: Validation Curves

The *iteration curves* in Figure 12 shows the change of scores as the iteration increases. The curve based on *accuracy scores* (Figure 12(a)) shows that the accuracy reached a relatively high and stable level at around 100 iterations, whereas the curve based on *precision scores* (Figure 12(b)) suggests that the precision reached a high and stable level at around 125 iterations. There is not a huge difference, so both 100 and 125 are acceptable numbers to stop iterating.



(a) Metric: accuracy scores (b) Metric: precision scores

Figure 12: Iteration Curves

- **Analyses: Data set 2 - *Phishing Websites***

summary
Training time: around 0.748s
Validation time: around 0.001s
Test accuracy score: 0.9596
Performance improvement (CV scores): 0.4%
The best combination of hyper parameter values returned by *GridSearchCV()* was:
hidden_layer_sizes: logistic, *min_samples_leaf*: (110,), *learning rate*: 0.1, *max_iter*: 500.
The CV score (f1/precision): 0.9774 /0.9713.

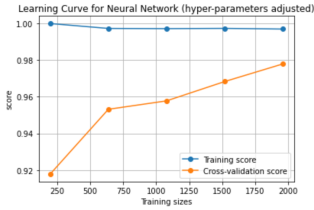
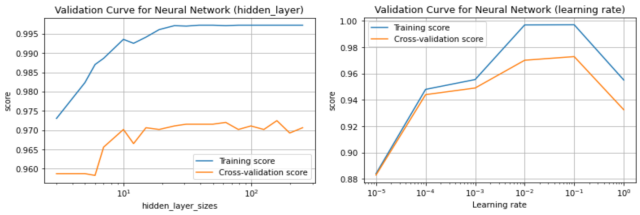
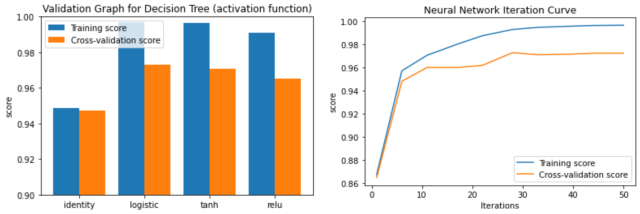


Figure 13: Neural Network Learning Curve (2)

The *learning curve* (Figure 13) suggests that the model has high variance and low bias. There is a trend of convergence, therefore, increasing the size of data set should help improving the performance.



(a) HP: hidden layer sizes (b) HP: learning rate



(c) HP: activation function (d) HP: iteration

Figure 14: Validation Curves

For the *hidden layer sizes* validation curve (Figure 14(a)), the CV score actually become gentle after 12, and reaches a little peak at around 120. I have tried different layers with different numbers of neutrons, but one layer with 110 neutrons was the pair with highest CV score among all the pairs that grid search returned.
For the *learning rate* (Figure 14(b)), it is clear that the highest scores is reached between the learning rate of 0.01 and 0.1, and the curve drops immediately after 0.1. The CV curve suggests that 0.1 has a slightly higher score, therefore, it should be the best hyper parameter value of learning rate in the model.
For the *activation function* validation curve (Figure 14(c)), it is obvious that the *identity function* performs

the worst, whereas the other functions have similar performance in the validation set. Among the other three functions, *logistic function* has slightly better performance, so that it was picked as the best activation function.
For the *iteration curve* (Figure 14(d)), the model reaches a relatively steady scores after around 30 iterations. I think any number after 30 would perform similarly, but the bigger the number is, the longer the training time will be.

Boosting

Boosting is considered an advanced algorithm of Decision Tree. Ideally, this algorithm should be applied based on the tuned Decision Tree model, to further improve the performance. However, this method did not work for both of my data sets. I've got really low accuracy and CV scores using this method (around 0.8), no matter how hard I tried to tune the hyper parameters. Therefore, I had to try boosting without the tuned Decision Tree as the base model. With *GradientBoostingClassifier()*, I managed to reach high accuracy and CV scores.

- **Tuned hyper parameters:**
  - ***number of weak learners:*** This is the key of Boosting algorithm. A proper number of weak learners could help improving model performance and avoid the problem of over-fitting.
  - ***max depth:*** This can be considered a pruning method as it works in the Decision Tree algorithm.
  - ***leaning rate:*** Tuning this hyper parameter could avoid potential missing of global maxima or long wait time. (As explained in the Neural Network part)
- **Analyses: Data set 1 - *Customer Satisfaction***

  summary
  Training time: around 15.29s
  Validation time: around 0.086s
  Test accuracy score: 0.9589
  Performance improvement (CV scores): 2.1%
  The best combination of hyper parameter values returned by *GridSearchCV()* was:
  learning rate: 0.1, *n_estimators*: 200, *max depth*: 8.
  The CV score (f1/precision): 0.9566 /0.9622.

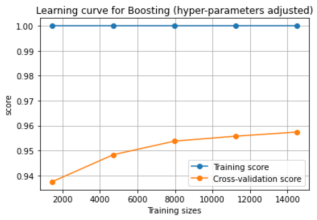


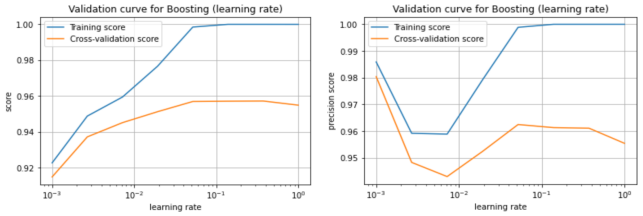
Figure 15: Boosting Learning Curve (1)

The *learning curve* (Figure 15) shows that the performance of the model is not very good. At least it has high variance, and the *Training score* is always 1, which is a typical over-fitting case. According to the shape of the curves, I do not think that increasing data set size could help with the performance. However, we could try to reduce the complexity of the model by reducing the depth and increasing the leaf size, as well as dropping features.

(a) HP: weak learners (b) HP: max depth

Figure 16: Validation Curves (weak learners)

For the *number of weak learners* validation curve (Figure 16(a)), the score reaches the peak when the number of weak learner is 125 and drops a bit at 150 then comes back to peak at 175 and 200. Actually, I feel like 125 and 175 could be good alternatives of the number of weak learners.
For the *max depth* validation curve(Figure 16(b)), the validation reaches the peak when the max depth is 8, and drops after 16. Therefore, 8 was selected as the best value for maximum depth. However, the training curve shows over-fitting after 7 depth. I think this may be the reason why the learning curve of the model does not converge well, nor have a good stability.



(a) Metric: accuracy scores (b) Metric: precision scores

Figure 17: Validation Curves (learning rate)

The *validation curves* (Figure 17) are the performances of *learning rate* based on different evaluation metrics.
The behavior of these two sets of curves are different at lower values, but both of them have their inflection points at a similar value before 0.1. The curves using precision scores (Figure 17(b)) suggest that the model would have more False Positive cases, which we should avoid, when the learning rate is around 0.01, but it will face the issue of over-fitting after the inflection points. This is a bad situation. I think the solution could be choosing the value at the peak of *validation score* (inflection points), combining with tuning of other hyper parameters.

- **Analyses: Data set 2 - *Phishing Websites***

  summary
  Training time: around 0.5s
  Validation time: around 0.003s

Test accuracy score: 0.9651
Performance improvement (CV scores): 0.4%
The best combination of hyper parameter values returned by *GridSearchCV()* was:
learning rate: 0.1, *n_estimators*: 132, *max depth*: 5.
The CV score (f1/precision): 0.9802 /0.9765.

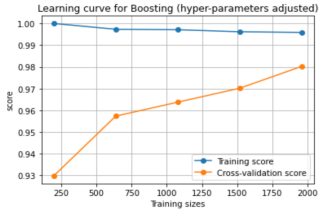
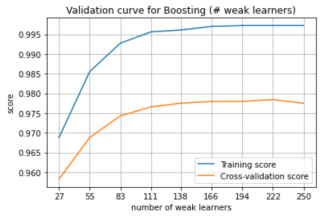
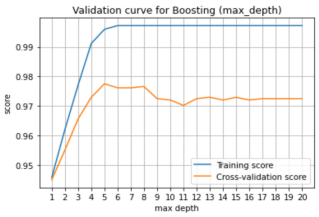


Figure 18: Boosting Learning Curve (2)

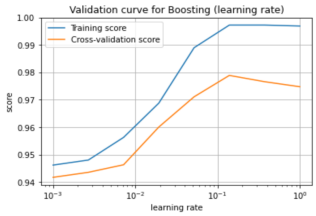
This *learning curve* (Figure 18) looks like the learning curve of Neural Network (Figure 13) a lot. It also has a slightly high variance and low bias. The trend of convergence in this graph is more obvious, so the same strategy of increasing the size of data set should help improving the performance.



(a) HP: weak learners



(b) HP: max depth



(c) HP: learning rate

Figure 19: Validation Curves

For the *number of weak learners* validation curve (Figure 19(a)), the best range is from 138 to 222, but bigger number of weaker learners may cause over-fitting problem. Therefore, a smaller number may be a better choice.
For the *max depth* validation curve (Figure 19(b)), 5 is a great choice since it is the peak and the model tends to be over-fitting after 5.
For the *learning rate* validation curve (Figure 19(c)), 0.1 is good, but we can improve it by finding the peak of the CV curve.

Support Vector Machine

- **Tuned hyper parameters:**
  - ***kernel:*** kernel is important in SVM since it reduces the complexity of calculation. A good choice of kernel could definitely help improving the model quality.
  - ***gamma:*** This parameter defines how far the influence of a single training example reaches. It has affects on the speed of training. So I think while tuning the kernel, I should also tune the value of gamma.
  - ***C:*** This is the cost of mis-classification. In my data sets, I want to avoid mis-classification, so I may want to set C to a high value.
  - ***max_iter:*** same reason as in Neural Network part.
- **Analyses: Data set 1 - *Customer Satisfaction***
summary
Training time: around 8s
Validation time: around 3.5s
Test accuracy score: 0.9469
Performance improvement (CV scores): 0.6%
The best combination of hyper parameter values returned by *GridSearchCV()* was:
kernel: rbf, *gamma*: 0.045, *C*: 5, *max_iter*: The model with highest score did not tune this hyper parameter.
The CV score (f1/precision): 0.9463 /0.9508.

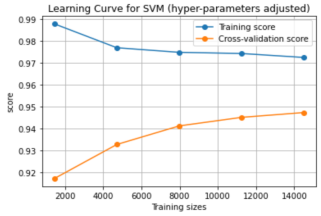
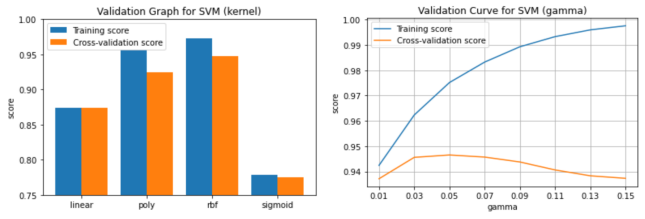
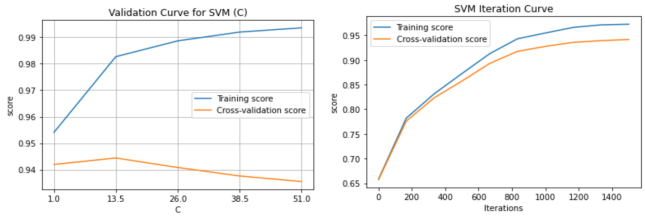


Figure 20: SVM Learning Curve (1)

The *learning curve* (Figure 20) shows another high variance case without the trend to converge, whereas the bias is not high. To improve the performance, I think I should not have focused only on the scores. Instead, I should sacrifice a little bit of accuracy in exchange for more stability.
For the *kernel* validation graph (Figure 21(a)), It is clear that the 'sigmoid' kernel had the lowest score while the 'poly' and 'rbf' kernel had similar performance. The 'rbf'kernel had the best performance on validation data set, therefore was selected.
For the *gamma* validation curve (Figure 21(b)), It seems that the performance was the best at around 0.05, and slowly fell after that.
For the *C* validation curve (Figure 21(c)), the performance was slightly improved when the C value increased from 1 to 13.5, and the score decreased after the C value of 13.5. In my opinion, any number that is smaller than 13.5 would perform similarly, among which 13.5 might be the value performs the best.
For the *iteration curve* (Figure 21(d)), I have tried the number of iterations from 400 to 1500. it seems that the model finally reached a relatively high and steady score after 1200 iterations. I believe the performance after 1200 iterations will not vary much. Therefore, the training should be stopped after 1200 iterations.

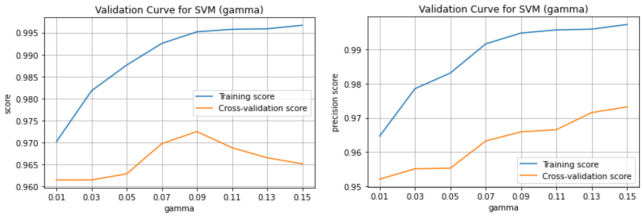


(a) HP: kernel (b) HP: gamma

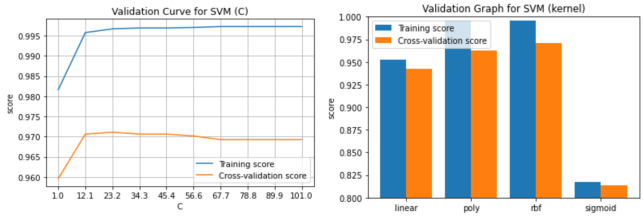


(c) HP: C (d) HP: iteration

Figure 21: Validation Curves



(a) Metric: accuracy (b) Metric: precision



(c) HP: C (d) HP: kernel

Figure 23: Validation Curves

- **Analyses: Data set 2 - *Phishing Websites***
  summary
  Training time: around 0.143s
  Validation time: around 0.054s
  Test accuracy score: 0.9614
  Performance improvement (CV scores): 1.6%
  The best combination of hyper parameter values returned by *GridSearchCV()* was:
  kernel: rbf, *gamma*: 0.1, *C*: 10, *max_iter*: The model with highest score did not tune this hyper parameter.
  The CV score (f1/precision): 0.9742 /0.9677.

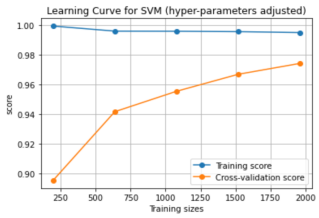


Figure 22: SVM Learning Curve (2)

The *learning curve* (Figure 22) shows a low bias converging shape with a slightly high variance. In this case, increase the data size may help it to narrow down the variance.

For the *gamma* validation curves (Figure 23(a)(b)) based on different evaluation metrics, the performances were different. If the *accuracy score* was selected as the metric (Figure 23(a)), 0.09 would be the first choice, whereas if the *precision score* (Figure 23(b)) was selected, the score kept rising until the biggest value of 0.15 was reached. If I care more about avoiding False Positive cases, I would enlarge the range to find a better gamma value, because I believe that the precision score of validation curve would keep rising after the gamma value of 0.15.

For the *C* validation curve (Figure 23(c)), it is obvious that the performance stays steady when C value is bigger than 12.1. I feel like both 12.1 and 23.2 would be good values of the hyper parameter.
For the *kernel* validation graph (Figure 23(d)), the 'sigmoid' has a significantly lower accuracy score compared with the other three kernels. 'poly' and 'rbf' kernels have similar performance on training set, but the 'rbf' kernel has a better performance on CV set. Therefore'rbf' should be chosen.

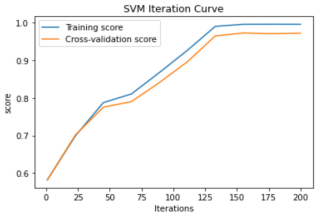


Figure 24: Iteration curve

For the *iteration curve* (Figure 24), The performance gradually improves while the the number of iterations increases, until around 150 iterations. And the performance after 150 iterations has micro changes, therefore, the iteration should be stopped after 150 iterations.

K-Nearest Neighbors

- **Tuned hyper parameters:**
  - ***k:*** This is the core of the KNN algorithm. We need to decide how many nearest neighbors we would like to use.
  - ***weights:*** The default value of 'weights' is uniform, meaning that all neighbors are equally weighted, whereas 'distance' will make the weights change based

upon the distances between the node and the neighbor. I suppose the change of this hyper parameter would improve the performance, since the closer neighbors should have more weights.

- **Analyses: Data set 1 - *Customer Satisfaction***
  summary
  Training time: around 0.002s
  Validation time: around 3.425s
  Test accuracy score: 0.9163
  Performance improvement (CV scores): 0.3%
  The best combination of hyper parameter values returned by *GridSearchCV()* was:
  k: 6, *weights*: distance.
  The CV score (f1/precision): 0.9132 /0.9364.

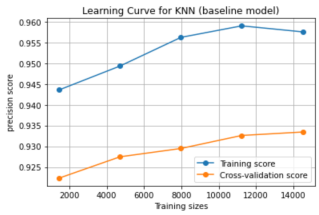
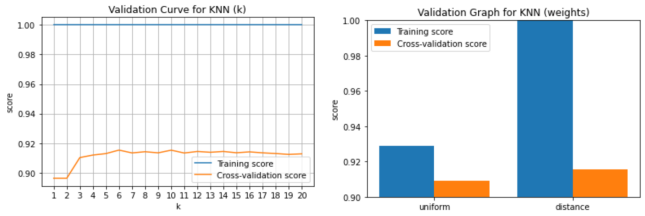


Figure 25: KNN Learning Curve (1)

The *learning curve* (Figure 25) is with high variance, high bias, and is not converging. I have tried different hyper parameters and different values, but none of them had really worked well. To improve the performance, I may need to try tuning more hyper parameters, including p, which changes the distance calculation method.



(a) HP: k (b) HP: weights

Figure 26: Validation Curves

For the *k* validation graph (Figure 26(a)), the curve reaches a little peak when k value is 6. I also tried other peaks including 10 and 16, but the performances were slightly worse.
For the *weights* validation curve (Figure 26(b)), the *distance* option has an overwhelming good performance on the training set, but only a slightly better performance on the validation set, which is very likely to be over-fitting. The *distance* option did improve the model quality a bit, but not as much as I expected, which was surprising.

- **Analyses: Data set 2 - *Phishing Websites***
  summary
  Training time: around 0.001s
  Validation time: around 0.03s
  Test accuracy score: 0.9541
  Performance improvement (CV scores): 3.6%
  The best combination of hyper parameter values returned by *GridSearchCV()* was:
  k: 13, *weights*: distance.
  The CV score (f1/precision): 0.9673 /0.9535.

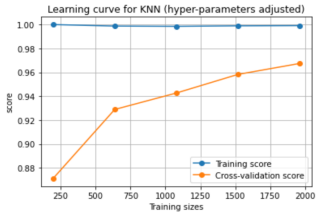
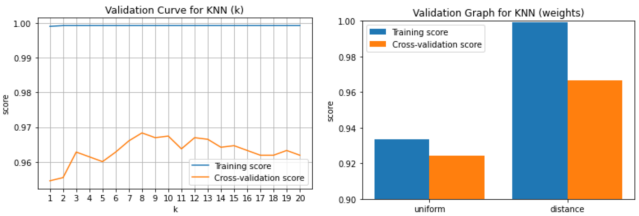


Figure 27: KNN Learning Curve (2)

Compared with the *learning curve* of the first data set (Figure 25), this *learning curve* (Figure 27) has a better performance with relatively low bias and high variance, but it shows the trend to converge. Therefore, I feel like the performance can be further improved with a larger size of data set.



(a) HP: k (b) HP: weights

Figure 28: Validation Curves

For the *k* validation graph (Figure 28(a)), the CV is sort of like a "camel" shape, in which the peak is reached when the k value is 8.
It is also interesting to compare the performances of the same algorithm on two different data sets. For the *weights* hyper parameter, the performances on training data set are very similar in both Data Set 1 (Figure 26(b)) and Data Set 2 (Figure 28(b)), but very different on variation data sets. Overall, Data Set 2 has a better performance on the variation data set. Specifically, the *distance* option has a significant advantage. I think this might be one of the reasons why Data Set 2 has a better performance improvement, as well as higher CV and accuracy scores compared with data set 1.

Comparison between Algorithms

Time

The training and validation time vary when the data set complexities and the data sizes are different. However, we could still try to sum up some general rules.

Algo	DT	ANN	Boosting	SVM	KNN
Tra-Time(1)	0.074	35.57	15.29	8	0.002
Val-Time(1)	0.003	0.022	0.086	3.5	3.425
Tra-Time(2)	0.004	0.748	0.5	0.143	0.001
Val-Time(2)	0	0.001	0.003	0.054	0.03

Table 1: Summary of Time

The above table of training and validation time with 2 different data sets using 5 different algorithms shows that 1) the ANN algorithm takes the longest time to train in both data sets; 2) KNN algorithm takes the shortest time to train but longer time for validation; 3) Decision Tree takes the shortest time to validate; 4) Even though the training time for ANN algorithm is several times or even dozens of times longer than other algorithms, it only takes a considerably small amount of time to validate.

- **Data set 1 - *Customer Satisfaction***

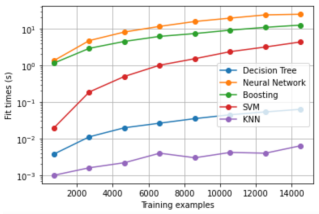


Figure 29: Time curve (1)

It is clearer to observe from the *Time curve* (Figure 29) that the fastest algorithm is KNN, which would not increase fitting time dramatically as the number of samples increases. The ANN algorithm takes the longest time to fit while Boosting takes slightly shorter time. The Decision Tree, Boosting, and ANN would increase fitting time linearly as sample size grows. Whereas SVM will take more time when handling larger size of data.

- **Data set 2 - *Phishing Websites***

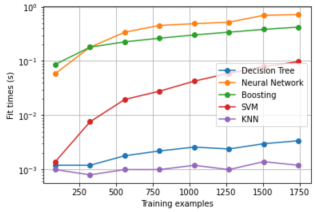


Figure 30: Time curve (2)

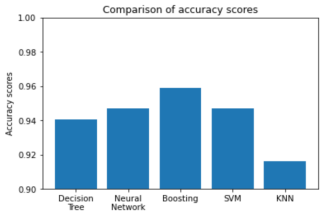
The conclusions of *Time curve* (Figure 30) of Data Set 2 is similar to Data Set 1, except that 1) the fitting time is almost the same for KNN, Decision Tree, and SVM when the sample size is small (probably smaller than 100); 2) ANN takes less time than Boosting when the sample size is less than 300; 3) compared with Data Set 1, the differences between ANN and Boosting, as well as KNN and Decision Tree are smaller on Data Set 2; 4) the rate of svm slowing down as the number of samples increases increases.

Performance

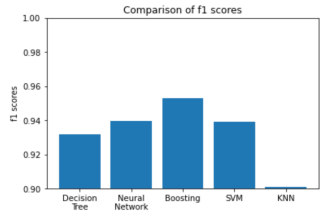
I would like the evaluate the performance based on three scores: accuracy score, f1 score, and precision scores.

- **Data set 1 - *Customer Satisfaction***

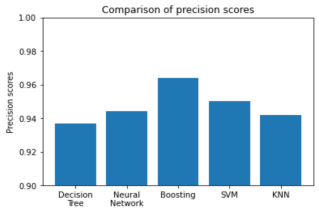
  From the *accuracy* and *f1* point of view, KNN algorithm is the first one to be sifted out. SVM and ANN have similar performance with accuracy and f1 scores at around 0.94. The winner is the Boosting algorithm with the highest accuracy and f1 score. From the *precision* point of view, Decision Tree and KNN algorithms have similar performance, which are slightly worse than the ANN and SVM algorithms. But still, the algorithm with the highest precision score is also Boosting algorithm.



(a) Comparison: Accuracy



(b) Comparison: f1



(c) Comparison: precision

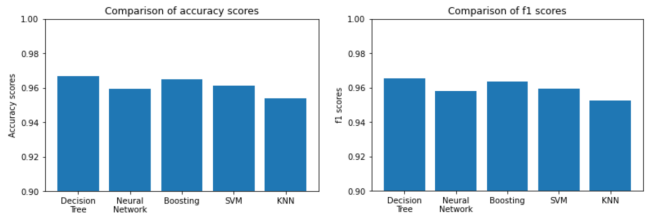
Figure 31: Comparison of scores (1)

- **Data set 2 - *Phishing Websites***

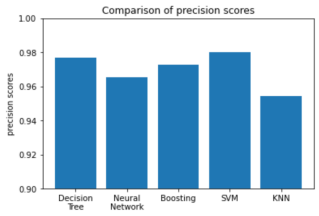
  Similar to the Data Set 1, the comparison results are similar between the *accuracy* scores and *f1* scores. Honestly there is no algorithm with significant advantages. All five algorithms have performed well on Data Set 2, whereas Decision Tree has performed slightly better than other algorithms. From the perspective of *precision* scores, the KNN algorithm has relatively worse performance, whereas SVM and Decision Tree algorithms are equally good.

Conclusion

In my opinion, there are more than one perspective to consider when picking a proper algorithm. The time cost, the accuracy, some other evaluation metric, and any special requirements should all be taken into consideration. It is more like a game of trade-off. In my two data sets, as I explained at the beginning, on top of the *accuracy* scores, I will also



(a) Comparison: Accuracy (b) Comparison: f1



(c) Comparison: precision

Figure 32: Comparison of scores (2)

consider the *precision* scores, since the cost of a False Positive case is more unaffordable.

- **Data set 1 - *Customer Satisfaction***
  For this data set, KNN algorithm is the fastest, but its accuracy is too low, so I would not pick this algorithm.
  ANN and SVM algorithms take long time to fit, but their performances are not good enough to make me feel like they are worth the time. Therefore, I will pick from the Boosting and Decision Tree algorithms.
  Boosting algorithm has the highest scores, but also takes a long time to fit, so if I have no limit on time, but need a considerably high accuracy, I will select this algorithm.
  On the other hand, the Decision Tree algorithm scores about 2% less than the Boosting algorithm, but it only takes about 0.5% of the time of Boosting algorithm to fit. Therefore, if the requirement of accuracy is not that high, I will choose the Decision Tree algorithm for the sake of time cost.
  In the *Customer Satisfaction* case, I think the requirement for accuracy is not as high, but a company definitely wants to reduce cost, therefore, I may end up choosing the Decision Tree algorithm.
- **Data set 2 - *Phishing Websites***
  For the situation of *Phishing Websites*, I think the essential thing is to avoid undetected phishing websites. That is to say, I would pick the algorithm with excellent performance of precision, which is either Decision Tree or SVM algorithms.
  Roughly speaking, the fitting time of the SVM algorithm is about 50 times that of the Decision Tree algorithm. Furthermore, as I concluded from the Time curve (2) (Figure 30), as the number of samples increases, the fitting time of SVM algorithm will increase exponentially.
  In reality, there would be a lot more samples than what I had for the project, therefore, SVM will be too slow for handling this problem. Even though it performs slightly better than the Decision Tree algorithm, I will still select Decision Tree algorithm to deal with this situation.

Appendix

This is the chart I used to record and compare the scores. The original chart is stored in the git repo.

Data Set			DecisionTree	NeuralNetwork	Boosting	SVM
	Hyperparameters		default	default	default	default
satisfaction (baseline model)	f1_CV	Training	0.9243	0.9416	0.9354	0.9409
	precision_CV	Training	0.9125	0.9373	0.9457	0.9452
	Accuracy	Training	1	0.979	1	0.9547
	recall score	Training	1	0.9719	1	0.9368
	precision	Training	1	0.9801	1	0.9589
	Hyperparameters		criterion + max_depth + min_leaf	hidden_layers + max_iter + learning rate + activation	#weak learners + max_depth + leaning rate	kernel + gamma + C + max_iter
satisfaction (Hper-params adjusted)	best_f1_CV	Training	0.9412	0.9468	0.9566	0.9463
	best_pre_CV	Training	0.945	0.9491	0.9622	0.9508
	Accuracy	Test	0.9403	0.9469	0.9589	0.9469
	recall score	Test	0.9269	0.9347	0.9421	0.9283
	precision	Test	0.937	0.9441	0.9638	0.95
	Hyperparameters		default	default	default	default
Phishing (baseline model)	f1_CV	Training	0.9742	0.9738	0.977	0.9582
	precision_CV	Training	0.9703	0.9695	0.9722	0.9708
	Accuracy	Training	0.9967	0.9921	0.9967	0.9687
	recall score	Training	0.9954	0.9927	0.9963	0.9772
	precision	Training	0.9981	0.9918	0.9972	0.9614
	Hyperparameters		criterion + max_depth + min_leaf	hidden_layers + max_iter + learning rate + activation	#weak learners + max_depth + leaning rate	kernel + gamma + C + max_iter
Phishing (Hper-params adjusted)	best_f1_CV	Training	0.977	0.9774	0.9802	0.9742
	best_pre_CV	Training	0.9731	0.9713	0.9765	0.9677
	Accuracy	Test	0.9669	0.9596	0.9651	0.9614
	recall score	Test	0.9545	0.9507	0.9545	0.9393
	precision	Test	0.9767	0.9653	0.9729	0.9802

Figure 33: Complete record of model fitting scores

References

algorithm documents (scikit-learn)

1. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
2. https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
3. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html
4. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
5. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

other documents

6. https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62
7. https://www.geeksforgeeks.org/validation-curve/#:~:text=A%20Validation%20Curve%20is%20an,some%20parameter%20of%20the%20model.&text=A%20validation%20curve%20is%20used,used%20to%20tune%20a%20model.
8. https://scikit-learn.org/stable/modules/neural_networks_supervised.html
9. https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html
10. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.bar.html