# Comparison of Different Machine Learning Methods for Wine Quality Classification Problem

**Final Report of ESE 419 SP2020**

**Machine Learning and Pattern Classification**

Name: Xiaozhi Shen

Date: 04/29/20

## 1 Introduction

Machine learning has become more and more popular in recent decades. In many real-world applications, we can build machine learning models to find patterns behind existing statistics and help to make predictions of unseen data. Machine learning problems can be divided into three categories, supervised learning, unsupervised learning and reinforcement learning. In supervised learning, instances comprise of features along with target values. On the contrary, unsupervised learning problems provide only features without target values. Supervised learning can be further divided into two categories, one is regression and the other is classification. The target values of classification problems are discrete class labels while target values in regression problems are continuous variables. For example, email filtering is a typical classification problem and price prediction is a regression problem.

In this paper, I will solve a machine learning problem with different models. The dataset comes from the "winequality-white" dataset from the UCI repository [1]. This dataset contains 4898 instances and 12 attributes. The first 11 attributes came from physicochemical tests and serve as features, and the last attribute (target value) represents the quality (0-10) of the wine scored by sensory assessors.

My goal is to use machine learning algorithms to find the underlying relationship between physicochemical tests and sensory scores. I build different models based on the given dataset and evaluate them with different performance metrices.

This wine quality dataset has discrete and limited target values, so it is a classification problem. Some commonly used classification algorithms are Artificial Neural Networks (ANN), Logistic Regression, Support Vector Machine (SVM), K Nearest Neighbors (KNN), Naïve Bayes, Decision Trees, Random Forest, etc. These methods have advantages and disadvantages when solving different problems. For example, SVM works effectively in high dimensional spaces. Random Forest reduces variance to avoid over-fitting and is more accurate than decision trees in most cases. Considering our problem is a multi-class problem and the dataset is imbalanced, I tried ANN, SVM and Random Forest algorithms, trained the models with the same training set and test on the same testing set. For each model, I used grid search and cross validation methods to tune hyperparameters. Detailed work will be presented in Section 2.

In Section 3 and 4, I evaluated the performance of the models with confusion matrices and compared running time of training different models. Among the three models, Random Forest model returns highest test accuracy of 68.16%, SVM comes next with 65.60% and ANN with 64.46%. However, even though the test accuracies of the models are all around 2/3, the recalls of minority classes are very low. This can be caused by the imbalanced training data. Performance will be improved if more data are collected or we apply some oversampling methods to increase the sample size of minority classes. As for running time, Random Forest takes the least time to train the model, SVM takes about 1.6 times of that amount of time and ANN takes about 35 times. Taking both performance and computational cost into consideration, Random Forest is the optimal model for this classification problem.

## 2 Methods

A basic training procedure of training a machine learning model contains three steps: preprocessing data, defining a model and tuning hyperparameters.

*2.1 Preprocessing Data*

Before starting to work on training a particular model, I need to preprocess the raw data. The reason is that after analyzing the raw data, I found there are several outliers and the distribution of the features are not normalized, which may influence the performance of the trained models.

To detect outliers, I used method demonstrated in article [2], calculated the interquartile range ($IQR$), which is the difference between 75th and 25th percentiles, or between upper and lower quartiles, i.e. $IQR = Q_3 - Q_1$. Then I assigned those values below $Q_1 - 3 \times IQR$ or above $Q_3 + 3 \times IQR$ as outliers and remove them from the dataset. Fig. 1 and Fig. 2 show the distributions of each attribute before and after outlier removal.

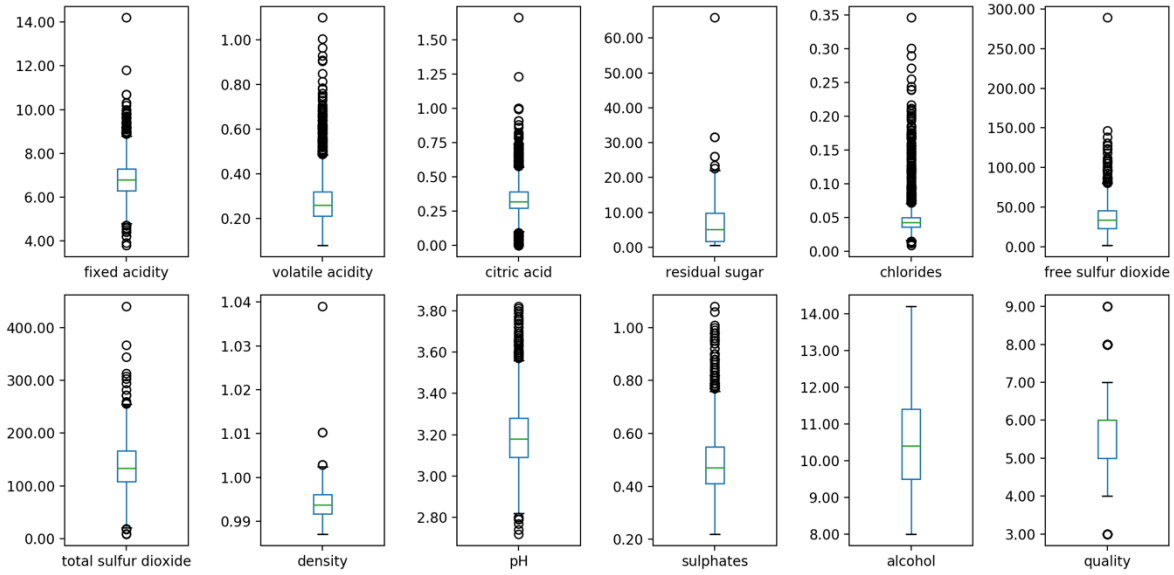


**Fig. 1** Distributions of attributes from raw dataset

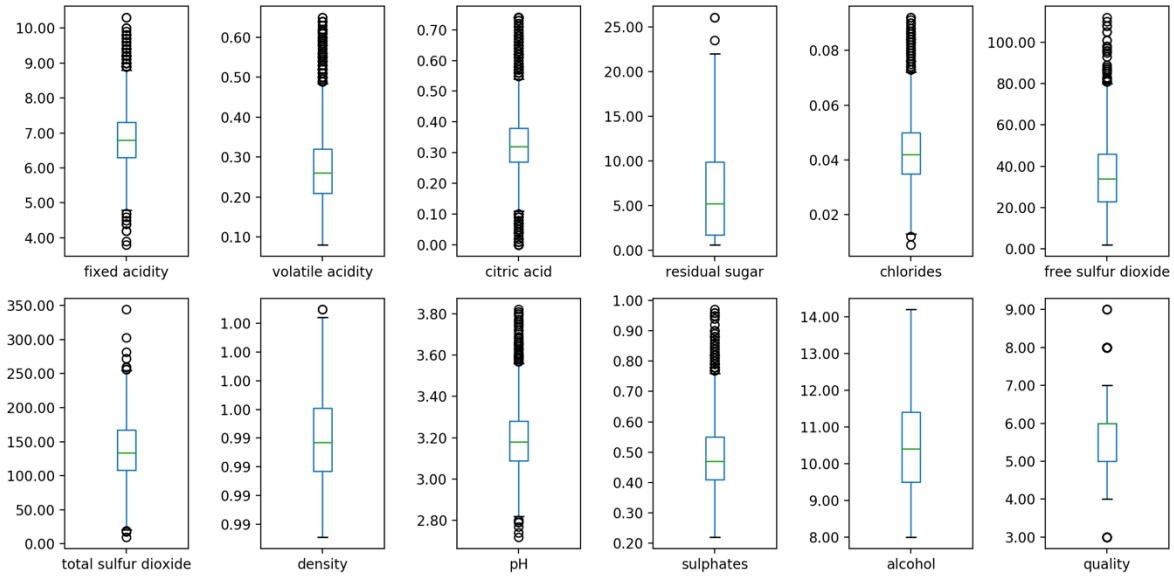


**Fig. 2** Distributions of attributes after removing outliers

After removing the outliers, I split the dataset into training set and test set by 7:3. Training set is used to train the model and test set is served as unseen dataset to evaluate the performance of the trained models.

Since the values of different features are not in similar scale, then I standardized the dataset into zero mean and unit variance, fitted the scaler with training set and then applied the same scaler to the test set. The python code of data preprocessing is in Appendix A.1.

## *2.2 Models*

### *2.2.1 Artificial Neural Networks (ANN)*

For linear classification problems, we can use a classification method called Perceptron. As shown in Fig. 3, Perceptron method takes feature vector X as inputs, computes the weighted sum of its inputs to form a scalar called net activation, then applies some activation function on the net activation. Sign function serves as activation function in Perceptron, that means it assigns the instance to class 1 if the weighted sum is positive and assigns the instance to class 2 if the weighted sum is negative. For non-linear cases, we can use multi-layer perceptron to train an ANN. Between the input and output layer, we can have multiple hidden layers. A simple three-layer ANN model is shown in Fig. 4. The relations between any adjacent two layers are the same as a single perceptron.

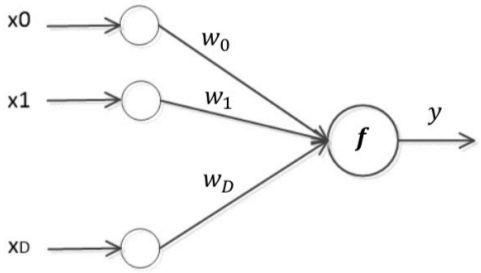


**Fig. 3** Perceptron model

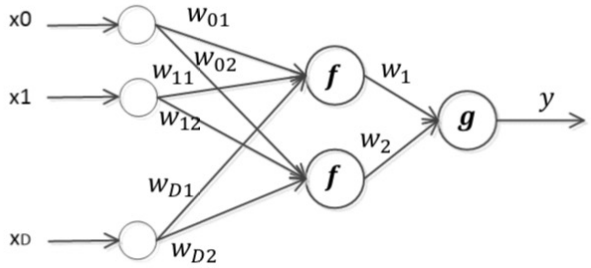


**Fig. 4** Three-layer ANN model

To train an ANN model, we need to find the weights to minimize the error function (Eq. 1). According to first order necessary condition, we can let the gradient of error function over weights to be zero and find the local minimizer(s). In practice, we can apply optimization methods, e.g. stochastic gradient descent (Eq. 2) or quasi-Newton methods (Eq. 3), to find the minimizer(s).

$$E(\boldsymbol{w}) = \sum_{n=1}^{N} E_n(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}\|y(\boldsymbol{x}_n, \boldsymbol{w}) - y_n\|^2 = \frac{1}{2}\sum_{n=1}^{N}\|\widehat{y_n} - y_n\|^2 \quad \text{(Eq. 1)}$$

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta_t \nabla E(\boldsymbol{w}^t) \quad \text{(Eq. 2)}$$

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta_t [HE(\boldsymbol{w}^t)]^{-1} \nabla E(\boldsymbol{w}^t) \quad \text{(Eq. 3)}$$

Actually, ANN with a single hidden layer is sufficient to represent any function, so it is a universal approximator. If we continue increasing the size of the hidden layer, the model will try to fit all the noise and training accuracy will approach 100%. However, it will cause overtraining problem and have poor generalization to unseen data. My solution is to add an L2 penalty to the error function and also to tune the optimal hidden layer size.

*2.2.2 Support Vector Machines (SVM)*

First, consider a linearly separable binary classification problem. We have learned from the Perceptron method that the decision boundary is the hyperplane $H: \{\boldsymbol{x}: \boldsymbol{w}^T \Phi(\boldsymbol{x}) + b = 0\}$. Since rescaling $\boldsymbol{w}$ and $b$ does not change the hyperplane, we can try to make all the training instances satisfy $y_i(\boldsymbol{w}^T \Phi(\boldsymbol{x_i}) + b) \geq 1$. Hence, we can guarantee the margin of $H$ will be at least $1/\|\boldsymbol{w}\|$. Only those vectors with $y_i(\boldsymbol{w}^T \Phi(\boldsymbol{x_i}) + b) = 1$ matter in this method, so they are called support vectors. Support vectors are those circled points lying on $y_i(\boldsymbol{w}^T \Phi(\boldsymbol{x_i}) + b) = 1$ in Fig. 5.

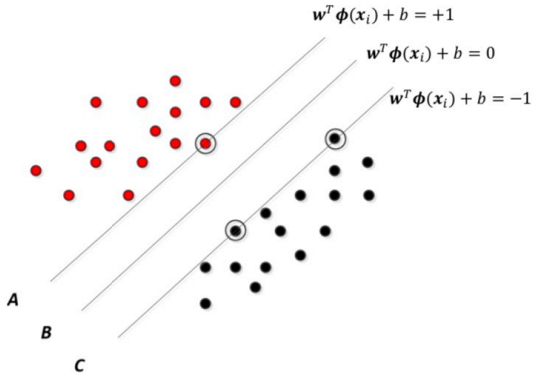


**Fig. 5** Support vectors in SVM

When solving the constrained optimization problem, we can take derivatives of $\boldsymbol{w}$ and $b$ the primal problem (Eq. 4) and turn it into its dual problem (Eq. 5).

$$L_P(\boldsymbol{w}, b, \boldsymbol{a}) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{i=1}^{n} a_i[y_i(\boldsymbol{w}^T \Phi(\boldsymbol{x_i}) + b) - 1] \quad \text{(Eq. 4)}$$

$$L_D(\boldsymbol{a}) = \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j y_i y_j \Phi(\boldsymbol{x})^T \Phi(\boldsymbol{x'}) \quad \text{(Eq. 5)}$$

Typically, in order to separate non-linear classes, we need to apply feature mapping $\Phi: \mathbb{R}^n \rightarrow \mathbb{R}^m$ to the original feature vectors. However, if we choose some kernel function, we can achieve that $k(\boldsymbol{x}, \boldsymbol{x'}) = \Phi(\boldsymbol{x})^T \Phi(\boldsymbol{x'})$. By choosing a proper kernel function, we can learn in high dimension feature space without explicit mapping into that space, thus saving computational cost. Some popular kernels are Linear kernel, Polynomial kernel, Gaussian Radial Basis Function (RBF) and Sigmoid kernel. In this problem, I chose RBF kernel.

To extend binary-class SVM algorithm to training a multi-class model, we can use "one-vs-one" or "one-vs-rest" methods. "One-vs-one" is to train classifiers between any class pairs. Suppose we have $n$ classes, then it requires to build $n(n-1)/2$ classifiers, so the runtime will be very high. Here I chose "one-vs-rest" method. When training a classifier models for each class, we treat the other classes as a combined class and turn it into binary classification problem. In total, we need to train $n$ classifiers, so the runtime is significantly less than "one-vs-one" method.

In SVM algorithm, I tried to tune two hyperparameters. I already picked kernel function to be RBF, then I tried different kernel coefficients. Also, I used L2 regularization to avoid $\boldsymbol{w}$ to grow too large and avoid the margin of $H$ approaching zero. So, I tuned kernel coefficient gamma and squared L2 penalty coefficient C during the training process.

*2.2.3 Random Forest*

Decision tree is another useful classification model. To classify a new instance, we start at a root node, test the feature specified at that node and then move down the tree branch to a subtree. Continue this process until it reaches a leave node and gets the target value. Fig. 6 shows a simple decision tree to classify a 2-D instance. To learn a decision tree, we need to find the attribute that best classifies the training instances at the root of each subtree. The attribute with the highest information gain is chosen as the decision attribute for the root node.

Random Forest is an ensemble model consists of many decision trees. Each learner in Fig.7 is a decision tree. The prediction of each learner will be averaged to form the prediction of the ensemble model. So Random Forest will alleviate overtraining problem caused by a single decision tree. It uses two key concepts to enhance the randomness of the member decision trees:

- Bootstrap sampling of training data when building decision trees (bagging)
- Random subset of features (attributes) considered when splitting nodes

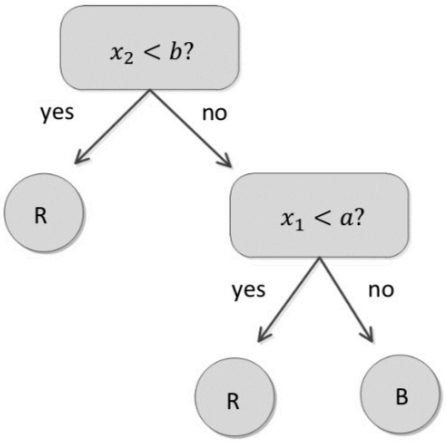


**Fig. 6** Decision Trees model

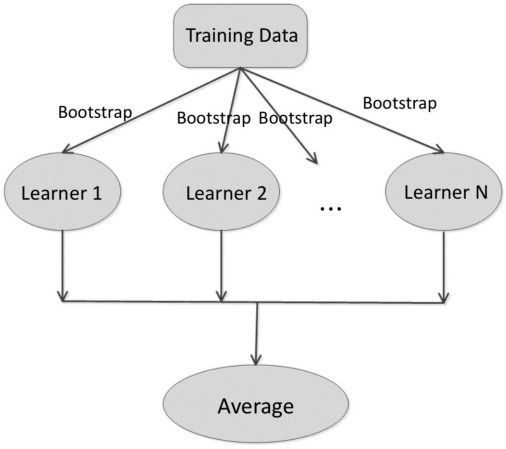


**Fig. 7** Random Forest model

If a decision tree is fully grown, it may try to learn the noise in the training set and lose some generalization capability. One solution is to limit the depth of the tree.

In general, performance of a random forest model will be enhanced with the number of trees in a forest increasing. However, some researchers have found that a larger number of trees in a forest only increases its computational cost and has no significant performance gain [3]. According to their suggest, I tried to build random forests with a range between 50 and 300 trees.

*2.3 Tuning hyperparameters*

As I mentioned in part 2.2, different classification models have different hyperparameters to tune. In ANN, I chose to tune hidden layer size and L2 penalty parameter alpha to avoid overtraining problem. In SVM, I chose to tune kernel coefficient gamma and squared L2 penalty parameter C. In Random forest algorithm, I chose to tune the number of trees in the forest and the max depth of the trees.

I followed the general procedure to tune hyperparameters:

a) define the range of possible values of hyperparameters

b) define a method of sampling the hyperparameters

c) define evaluation criteria to judge the model

d) define a cross-validation method

I set the range of hidden layer size in ANN from 50 to 500, the range of L2 regularization parameter alpha from $10^{-3}$ to 1. For SVM, I tuned the kernel coefficient gamma from $10^{-4}$ to 10, squared L2 penalty parameter C from 1 to 500. For random forest, I tuned the number of trees from 50 to 300, and the max depth from 3 to no limit. Then I used grid search method, which is an exhaustive sampling of the potential hyperparameters, to train all candidate models and evaluate different models base on its accuracy on the validation set.

Generally, we need a separate validation set to evaluate the performance of the model trained by training set. However, since the dataset is not very large, I used cross-validation method to tune the hyperparameters. To learn each model of a particular permutation of hyperparameters, we need first to shuffle the training data and split it into $k$ equally sized folders. Then we consider one out of $k$ folders to be validation set and the others as training set, as shown in Fig. 8. Each time of learning, we train the model with the training set, then evaluate the performance on the validation set. Since we have $k$ folders, we can train $k$ models and have $k$ validation performances. The averaged performance is used to evaluate the model with the chosen hyperparameters and we pick the model with best performance as the optimal model.

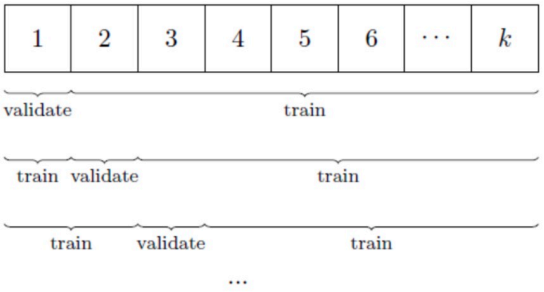


**Fig. 8** k folder cross-validation

The python code of training models and tuning hyperparameters are in Appendix A. 2, A. 3 and A. 4. After running grid search, I found the best hyperparameters of ANN are {alpha = 1, hidden layer size = 400}. The best hyperparameters of SVM are {gamma = 1, C = 50}. As for Random Forest model, I found best hyperparameters are {number of trees = 200, max depth = None}.

## 3 Results and Analysis

After training classification models in Section 2, I tested them with the same test set. Confusion matrix is the most common performance metric for classification problems. Table 1, 2 and 3 show confusion matrices of three classifiers. Then we can calculate the test accuracy of three models to be 64.46% (ANN), 65.60% (SVM) and 68.16% (Random Forest), as shown in Table 4.

However, although the overall accuracies are around 2/3 for all the models, the recalls of minority classes are very low. None of the three models can classify any of the actual Class 3 and 9 instance right. In addition, the probabilities to classify actual Class 4 and 8 right are all below 50%. The trained models tend to classify instances of minority classes to majority classes. It is because our dataset is heavily imbalanced and we do not have enough training data to build classifiers for minority classes. We can also find the precisions of ANN models are significantly lower than SVM and Random Forest especially of Class 4 and 8. That means more instances are misclassified into the minority classes in ANN model compared with SVM and Random Forest.

**Table 1** Confusion matrix of ANN Classifier

| ANN | Actual Quality | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Prediction Quality | | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Precision |
| | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N/A |
| | 4 | 1 | 9 | 10 | 5 | 1 | 0 | 0 | 34.62% |
| | 5 | 1 | 21 | 283 | 81 | 9 | 2 | 0 | 71.28% |
| | 6 | 0 | 8 | 129 | 440 | 88 | 11 | 1 | 64.99% |
| | 7 | 0 | 0 | 14 | 73 | 156 | 17 | 1 | 59.77% |
| | 8 | 0 | 0 | 0 | 16 | 11 | 19 | 0 | 41.30% |
| | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N/A |
| | Recall | 0 | 23.68% | 64.91% | 71.54% | 60.94% | 38.78% | 0 | |

**Table 2** Confusion matrix of SVM Classifier

| SVM | Actual Quality | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Prediction Quality | | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Precision |
| | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N/A |
| | 4 | 0 | 3 | 1 | 1 | 0 | 0 | 0 | 60.00% |
| | 5 | 0 | 6 | 241 | 55 | 3 | 0 | 0 | 79.02% |
| | 6 | 2 | 29 | 189 | 527 | 126 | 24 | 2 | 58.62% |
| | 7 | 0 | 0 | 5 | 30 | 133 | 6 | 0 | 76.44% |
| | 8 | 0 | 0 | 0 | 2 | 3 | 19 | 0 | 79.17% |
| | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N/A |
| | Recall | 0 | 7.89% | 55.28% | 85.69% | 50.19% | 38.78% | 0 | |

**Table 3** Confusion matrix of Random Forest Classifier

| Random Forest | Actual Quality | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Prediction Quality | | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Precision |
| | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N/A |
| | 4 | 0 | 5 | 1 | 0 | 0 | 0 | 0 | 83.33% |
| | 5 | 2 | 21 | 291 | 68 | 4 | 0 | 0 | 75.39% |
| | 6 | 0 | 12 | 140 | 500 | 113 | 17 | 1 | 63.86% |
| | 7 | 0 | 0 | 4 | 46 | 145 | 14 | 1 | 69.05% |
| | 8 | 0 | 0 | 0 | 1 | 3 | 18 | 0 | 81.82% |
| | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N/A |
| | Recall | 0 | 13.16% | 66.44% | 81.30% | 54.72% | 36.73% | 0 | |

Besides evaluating the performance of the model, I also recorded the running time of training the models as computation cost. From Table 4, we can see the running time of Random Forest is the least, so it is the most efficient algorithm for this dataset. SVM takes about 1.6 times of the time to training Random Forest and ANN takes about 35 times of that amount.

**Table 4** Test accuracy and running time of different models

| Algorithm | ANN | SVM | Random Forest |
|---|---|---|---|
| Test accuracy | 64.46% | 65.60% | 68.16% |
| Running time (s) | 8463.31 | 389.30 | 236.34 |

## 4 Conclusions

From the results and analysis in Section 3, we can conclude that random forest algorithm has highest test accuracy 68.16% and is most computational efficient, so it is the best classification algorithm for this dataset. SVM is also a good classifier for this dataset regarding to computational cost, but its accuracy is 2.56% lower than Random Forest. Compared with Random Forest and SVM, ANN has low accuracy and significant low precision and takes much more time to train the model, so I do not recommend ANN method for this dataset.

Although the overall test accuracy is not bad, I noticed the recalls of minority classes are very low. It is because the dataset is heavily imbalanced and there are not enough training instances for minority classes. Actually, I tried randomly resampling the minority classes with the existing minority instances to increase the training set size, but it had little improvement to the performance of the models. So, I suggest collecting more data with the minority classes or using other machine learning package like “imblearn” to apply more complex oversampling algorithm like SMOTE to improve the performance of the models in the future.

## References

[1] P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553, 2009.

[2] Natasha Sharma. Ways to Detect and Remove the Outliers. https://towardsdatascience.com/ways-to-detect-and-remove-the-outliers-404d16608dba

[3] Oshiro, Thais & Perez, Pedro & Baranauskas, José. (2012). How Many Trees in a Random Forest?. Lecture notes in computer science. 7376. 10.1007/978-3-642-31537-4_13.

## Appendix

*A.1 Python code of preprocessing data*

```
import numpy as np
import pandas as pd
from pandas import read_csv
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
np.set_printoptions(threshold=np.inf)
pd.set_option('display.max_columns',None)

# load wine quality data set
url =
"https://raw.githubusercontent.com/sxz294/WineQualityClassifier/master/winequality-
white.csv"
df = read_csv(url,sep=';')

# data visualization
fmt = '%.2f'
yticks = mtick.FormatStrFormatter(fmt)
print(df.head())
print(df.describe())
print(df.groupby('quality').size())
print(df.corr())
ax1=df.plot(kind='box', subplots=True, layout=(2,6), sharex=False, sharey=False,
figsize=(12,6))
for ax in ax1:
    ax.yaxis.set_major_formatter(yticks)
plt.tight_layout()

# detect and remove outliers
Q1 = df.quantile(0.25)
Q3 = df.quantile(0.75)
IQR = Q3 - Q1
df_out = df[~((df < (Q1 - 3 * IQR)) |(df > (Q3 + 3 * IQR))).any(axis=1)]
print(((df < (Q1 - 3 * IQR)) |(df > (Q3 + 3 * IQR))).any(axis=1))
df_out.shape
print(df_out.groupby('quality').size())
ax2=df_out.plot(kind='box', subplots=True, layout=(2,6), sharex=False, sharey=False,
figsize=(12,6))
for ax in ax2:
    ax.yaxis.set_major_formatter(yticks)
plt.tight_layout()
plt.show()

data=df_out.values
col=df_out.columns.size
X=data[:,:col-1]
y=data[:,col-1]

# split data and standardize data
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3,
random_state=1)
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

*A.2 Python code of training ANN model and test on test set*

```
import time
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import make_pipeline
from sklearn import metrics
from sklearn.model_selection import GridSearchCV

# train the model and tune hyperparameters
start_time=time.time()
pipeline = make_pipeline(MLPClassifier(solver='lbfgs', max_iter=5000, random_state=1))
hyperparameters = { 'mlpclassifier__hidden_layer_sizes' :
[(50,),(100,),(200,),(250,),(300,),(350,),(400,),(450,),(500,)],
                    'mlpclassifier__alpha':[1e-3,1e-2,1e-1,1]}
clf = GridSearchCV(pipeline, hyperparameters,cv=3)
clf.fit(X_train, y_train)
end_time=time.time()

# make prediction on test set
y_pred = clf.predict(X_test)

# print performance of the model
acc=metrics.accuracy_score(y_pred,y_test)*100
print("Test accuracy is: %0.2f" % acc+"%.")
print("Confusion matrix is:")
print(metrics.confusion_matrix(y_pred, y_test, labels=[3,4,5,6,7,8,9]))
print("Best parameters are:")
print(clf.best_params_)
print("Running time is %0.2f seconds." % (end_time - start_time))
```

Printouts:
Test accuracy is: 64.46%.
Confusion matrix is:
[[ 0  0  0  0  0  0  0]
 [ 1  9 10  5  1  0  0]
 [ 1 21 283 81  9  2  0]
 [ 0  8 129 440 88 11  1]
 [ 0  0 14 73 156 17  1]
 [ 0  0  0 16 11 19  0]
 [ 0  0  0  0  0  0  0]]
Best parameters are:
{'mlpclassifier__alpha': 1, 'mlpclassifier__hidden_layer_sizes': (400,)}
Running time is 8463.31 seconds.

*A.3 Python code of training SVM model and test on test set*

```
import time
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline
from sklearn import metrics
from sklearn.model_selection import GridSearchCV

# train model
start_time=time.time()
pipeline =
make_pipeline(SVC(kernel='rbf',random_state=0,decision_function_shape='ovr'))
hyperparameters = { 'svc__gamma': [1e-4,1e-3,1e-2,1e-2,1,10],
                    'svc__C':[1,50,100,200,300,500]}
clf = GridSearchCV(pipeline, hyperparameters,cv=8)
clf.fit(X_train, y_train)
end_time=time.time()

# make prediction on test set
y_pred = clf.predict(X_test)

# print performance of the model
acc=metrics.accuracy_score(y_pred,y_test)*100
print("Test accuracy is: %0.2f" % acc +"%.")
print("Confusion matrix is:")
print(metrics.confusion_matrix(y_pred, y_test, labels=[3,4,5,6,7,8,9]))
print("Best parameters are:")
print(clf.best_params_)
print("Running time is %0.2f seconds." % (end_time - start_time))
```

Printouts:
Test accuracy is: 65.60%.
Confusion matrix is:
[[ 0  0  0  0  0  0  0]
 [ 0  3  1  1  0  0  0]
 [ 0  6 241  55  3  0  0]
 [ 2  29 189 527 126  24  2]
 [ 0  0  5  30 133  6  0]
 [ 0  0  0  2  3  19  0]
 [ 0  0  0  0  0  0  0]]
Best parameters are:
{'svc__C': 50, 'svc__gamma': 1}
Running time is 389.30 seconds.

*A.4 Python code of training Random Forest model and test on test set*

```
import time
from sklearn.pipeline import make_pipeline
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from sklearn.model_selection import GridSearchCV

# train model
start_time=time.time()
pipeline = make_pipeline(RandomForestClassifier(random_state=0))
hyperparameters = { 'randomforestclassifier__n_estimators' : [50,100,150,200,250,300],
                    'randomforestclassifier__max_depth': [3,5,7,9,11,None]}
clf = GridSearchCV(pipeline, hyperparameters,cv=8)
clf.fit(X_train, y_train)
end_time=time.time()

# make prediction on test set
y_pred = clf.predict(X_test)

# print performance of the model
acc=metrics.accuracy_score(y_pred,y_test)*100
print("Test accuracy is: %0.2f." % acc+"%.")
print("Confusion matrix is:")
print(metrics.confusion_matrix(y_pred, y_test, labels=[3,4,5,6,7,8,9]))
print("Best parameters are:")
print(clf.best_params_)
print("Running time is %0.2f seconds." % (end_time - start_time))
```

Printouts:
Test accuracy is: 68.16%.
Confusion matrix is:
[[ 0  0  0  0  0  0  0]
 [ 0  5  1  0  0  0  0]
 [ 2 21 291 68  4  0  0]
 [ 0 12 140 500 113 17  1]
 [ 0  0  4 46 145 14  1]
 [ 0  0  0  1  3 18  0]
 [ 0  0  0  0  0  0  0]]
Best parameters are:
{'randomforestclassifier__max_depth': None, 'randomforestclassifier__n_estimators': 200}
Running time is 236.34 seconds.