STAT 440 Final Report

Hujie Han, Henglin Huang, Fang Yi Wang, Jing Xie

Abstract

This project consists of implementing the algorithm while enhancing three classification models, the logistics, the support vector machine and the gradient boosting machine. The enhancements consist of trying to perform an autotune and cross validation-like procedure on the logistics; introducing new kernel on the SVM; and a faster and more accurate algorithm on the GBM. The new models are then tested on a real dataset “Rain in Australia” in order to validate their prediction power. Finally, we found that the new kernel introduced in SVM does not always outperform the pre-existing ones. However, the GBM enhancement (namely the Quick Nodes Morph Mend) has successfully reduced the run time while increasing the accuracy at the same time.

Introduction

A well performing prediction model predicts future events based on today’s available data with high effectiveness and accuracy. The type of problem could range from the smallest molecular reactions to global macroeconomic financial behaviors. The dataset “Rain in Australia” contains current geographical conditions such as Location, Weather conditions, etc. This information is helpful to make a highly probable and almost conclusive prediction to tomorrow’s weather, when applying an appropriate statistical model. By performing this, one can better understand the implication and importance of the current condition to the next day’s weather, rainy or not.

There is a multitude of prediction models, each one is powerful in its own way. Finding the most suitable for a dataset would be a challenging process. Plus, the existing approaches involve intensive calculations that are very repetitive and time consuming and have sometimes a limited functionality. This is why the following points would be worth the investigation for a more efficient algorithm: - Expand an existing classification method’s power in R - Provide a more friendly way to tune model parameters - Enhance the algorithm to reduce time efficiency and/or boost the predicting accuracy

The final product is a R package including implemented useful prediction tools with additional features that make them more powerful than the current available ones in R. These functions are then tested against simulated data while referencing to the existing R packages. Finally, the models will be applied to the real dataset “Rain in Australia (https://www.kaggle.com/jsphyg/weather-dataset-rattle-package)” and their prediction power will be compared.

To limit the length of this report, three models are chosen to be implemented and explained in details. The mathematical background of each model as well as how the models were enhanced are included. The chosen models are: Logistics (Poupart, 2019; Swaminathan, 2018), Support Vector Machine (SVM) (Gandhi, 2018; Chen, 2018) and Gradient Boosting Machine (GBM) (Friedman, 2002; Ridgeway, 2019).

Then, this report explains how we altered the models in attempt to enhance the prediction accuracy and/or efficiency. The new models are then tested against the real dataset “Rain in Australia” and their results are compared.

The dataset “Rain in Australia” is pre-processed with imputed missing values and feature engineered. The dataset will be split in the same 80/20 train/test ratio. Thus, all models would be trained with same 80% of the data. **More details and code about data analysis and pre-processing is included in Appendix.**

Models used

The three prediction models tackled in this project consist of very popular models nowadays. Below is a deeper presentation of each of them including their mathematical logic and application on the real dataset.

Logistics

General idea

Logistic Regression is a form of binary classification in machine learning. While using the logistic (sigmoid) funtion, the posterior $Pr(y|x)$ generated can take any real value between 0 and 1. In other words, it is a continuous function. Even though in reality, the cluster labels could be any two values, but without loss of generality it is assumed here that the response variable y can only take values 0 or 1.

Model

The Logistic regression is of the following form:

$$log(\frac{p}{1-p}) = w_0 + w_1x_1 + \cdots + w_px_p$$

For an entire dataset (X, y) , the goal is to maximize the likelihood in order to find the best fit of the model. The weight w that maximize the posterior can be derived from the following expression. The estimated weights can then be applied to predict the classification:

$$w^* = argmax_w \prod_n \sigma(w^T\bar{x}_n)^{y_n}(1-\sigma(w^T\bar{x}_n))^{1-y_n}$$

Learning the model

If we simply convex the loss by setting derivative to 0, it gives out the form:

$$0 = \frac{\partial L}{\partial w} = -\sum_n y_n \frac{\sigma(w^T\bar{x}_n)(1-\sigma(w^T\bar{x}_n))\bar{x}_n}{\sigma(w^T\bar{x}_n)} - \sum_n (1-y_n)\frac{\sigma(w^T\bar{x}_n)(1-\sigma(w^T\bar{x}_n))(1-\bar{x}_n)}{(1-\sigma(w^T\bar{x}_n))}$$

$$\Rightarrow 0 = \sum_n [\sigma(w^T\bar{x}_n) - y_n]\bar{x}_n$$

The sigmoid function σ prevents us from isolating w , so we will use Newton's method to iteratively reweight least square:

$$w \leftarrow w - H^{-1}\triangledown L(w)$$

where $\triangledown L$ is the gradient and H is the Hessian matrix

$$\begin{aligned} H &= \triangledown(\triangledown L(w)) \\ &= \sum_{n=1}^{N} \sigma(w^T\bar{x}_n)(1-\sigma(w^T\bar{x}_n))\bar{x}_n\bar{x}_n^T \\ &= \overline{X}R\overline{X}^T \end{aligned}$$

where

$$R = \begin{bmatrix} \sigma_1(1-\sigma 1) & & \\ & \ddots & \\ & & \sigma_N(1-\sigma_N) \end{bmatrix}$$

and $\sigma_1 = \sigma(w^T\bar{x}_1), \sigma_N = \sigma(w^T\bar{x}_N)$

Regularization

To make sure that the coefficient don't take extreme values, we introduce a L2 penalty term $\frac{1}{2}\lambda||w||_2^2$ to penalize large weight. λ will be our tuning parameter.

$$min_w L(w) + \frac{1}{2}\lambda||w||_2^2$$

Then the Hessian Matrix , with λI ensuring the H is not singular, becomes:

$$H = \overline{X}R\overline{X} + \lambda I$$

This is the final formula used for training the model

Classification

The prediction is either 0 or 1. After the weight has been optimized, the predict value is defined by applying the sigmoid function on the new input x . Here the prediction is represented by the term y^* :

Fitting the model

$$y^* = \begin{cases} 1 & \sigma(w^T x) \geq 0.5 \\ 0 & otherwise \end{cases}$$

The logistic function we implemented takes in both training data and validation data, which are all numerical. Thus, data processing such as encoding categorical data should be done outside of the function. In the case of the project, see Appendix for data pre-processing code. The optimal tuning parameter λ (L2 regularization parameter) is automatically select among values from 1 to 210 with spacing 10 (11 values in total) which maximizes the prediction accuracy on the validation data. In this case, we can have a general knowledge about the complexity of the model.

Random extract 3000 data points for training purpose and 300 data points for validation purpose.

```
knitr::opts_chunk$set(echo = TRUE)

library(mypkg)
library(dplyr)
rain = read.csv("train_data_coded.csv", header = TRUE)
train = rain[sample(nrow(rain), 3000),]
train_out = train$RainTomorrow
train_in = select(train, -RainTomorrow)

test = rain[sample(nrow(rain), 300),]
test_out = test$RainTomorrow
test_in = select(test, -RainTomorrow)

logistic_train(train_in, train_out, test_in, test_out)
```

Result of above code:

```
with penalty term = 1: model accuracy is 0.8433333333333334
with penalty term = 21: model accuracy is 0.8433333333333334
with penalty term = 41: model accuracy is 0.84
with penalty term = 61: model accuracy is 0.84
with penalty term = 81: model accuracy is 0.84
with penalty term = 101: model accuracy is 0.8433333333333334
with penalty term = 121: model accuracy is 0.8466666666666667
with penalty term = 141: model accuracy is 0.8466666666666667
with penalty term = 161: model accuracy is 0.8466666666666667
with penalty term = 181: model accuracy is 0.8466666666666667
with penalty term = 201: model accuracy is 0.8533333333333334
```

Training result and conclusion

From the result shown above, the accuracy doesn't vary much with different lambda. Hence, this dataset doesn't contain much extreme values. The average accuracy is above 80%, which indicates logistic model is a good fit for this dataset.

Support Vector Machine

General principles

The SVM algorithm is a machine learning algorithm which is widely used to solve classification objectives. It uses a flexible representation of the class boundaries and implements automatic complexity control to reduce overfitting. Its objective function has a single global minimum which can be found in polynomial time.

SVM is a very popular machine learning algorithm, because it has a good generalization performance. SVM solves a variety of problems with little tuning required and it acheives significant accuracy with less computational power.

The objectives of the SVM algorithm is to find a hyperplane that separates the two classes of data. There are many possible hyperplanes that could be chosen, but choosing the ones having a maximum marginal distance in between, classification of future data points would more accurate.

The position and orientation of the hyperplane is influenced by many support vectors, which are data points that are the closest to the seperating hyperplanes.

Support vector machine computes a linear classifier of the form:

$$f(x) = w^T x + b$$

Since we want to apply this to a binary classification problem, we will ultimately predict:

$$\text{y} = 1 \qquad \text{if } f(x) \geq 0$$

$$\text{y} = \text{-}1 \qquad \text{if } f(x) < 0$$

And express using inner products as:

$$f(x) = \sum_{i=1}^{m} \alpha_i y_i \langle x_i, x \rangle + b$$

We can substitute a kernel $K(x_i, x)$ in place of the inner product if we desire.

The loss function that helps maximize the margin is hinge loss to training classifiers :
The cost is 0 if the predicted value and the actual value are of the same sign.
If $(1 - y * f(x)) >= 1$ then:

$$c(x, y, f(x))) = 0$$

Otherwise we calculate the cost using function below:

$$c(x, y, f(x))) = 1 - y * f(x)$$

Regularization parameter is to balance the margin maximization and loss, which is the objective of SVM. After adding the regularization parameter, we have the cost function as follow:

$$min_w \lambda \|w\|^2 + \sum_{i=1}^{n} (1 - y_i \langle x_i, w \rangle)$$

Take partial derivatives with respect to the weights to find the gradients. By using the gradients, we then update our weights.

$$\frac{\delta}{\delta w_k} \lambda \|w\|^2 = 2\lambda w_k$$

Now we have the function as:

$$(1 - y_i \langle x_i, w \rangle) = \begin{cases} 0, & \text{if } y_i \langle x_i, w \rangle \geq 1 \\ y_i x_{ik}, & o/w \end{cases}$$

If there is no misclassification then update gradient from the regularization parameter given:

$$w = w - \alpha * (2\lambda w)$$

If there is a misclassification, include the loss along with the regularization parameter to perform gradient update:

$$w = w + \alpha * (y_i * x_i - 2\lambda w)$$

The Simplified SMO Algorithm

The Sequential minimal optimization (SMO) algorithm is an algorithm for solving the quadratic programming (QP) problem that arises during the training of support-vector machines (SVM). It is widely used as it gives an efficient way of solving the dual problem of the support vector machine (SVM) optimization problem.

The SMO algorithm selects two α parameters, α_i and α_j and optimizes the objective with both these α s. Then it adjusts the b parameter based on the new α s. This process is repeated until the α s converge.

So, we are interested in solving:

$$max \quad W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} y_i y_j \alpha_i \alpha_j \langle x_i, x_j \rangle$$
$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, i = 1, \ldots, m$$

$$\sum_{i=1}^{m} \alpha_i y_i = 0$$

where C is a regularization parameter.
In order to check for convergence to the optimal point, the SMO algorithm iterates until all these conditions are satisfied and for any α_i s that satisfy these properties for all i will be an optimal solution to the optimization problem given below.

$$\alpha_i = 0 \quad \Rightarrow \quad y_i \left(w^T x_i + b \right) \geq 1$$

$$\alpha_i = C \quad \Rightarrow \quad y_i \left(w^T x_i + b \right) \leq 1$$

$$0 < \alpha_i < C \quad \Rightarrow \quad y_i \left(w^T x_i + b \right) = 1$$

The simplified Sequential minimal optimization(SMO) stops after all conditions above are satisfied.

Selecting Alpha Parameters

In order to find the alpha parameters, iterate over all α_i , for $i = 1 \ldots m$. If α_i does not fulfill the conditions to within some numerical tolerance, we select α_j at random from the remaining m-1 α s and attempt to optimize α_i and α_j .

Finding b threshold

After optimizing α_i and α_j , we need to find a threshold b such that the conditions are satisfied for the ith and jth.

If $0 < \alpha_i < C$, then the following threshold b_1 is valid

$$b_1 = b - E_i - y_i \left(\alpha_i - \alpha_i^{old} \right) \langle x_i, x_i \rangle - y_j \left(\alpha_j - \alpha_j^{old} \right) \langle x_i, x_j \rangle$$

And following threshold b_2 is valid if $0 < \alpha_j < C$:

$$b_2 = b - E_j - y_i \left(\alpha_i - \alpha_i^{old} \right) \langle x_i, x_j \rangle - y_j \left(\alpha_j - \alpha_j^{old} \right) \langle x_j, x_j \rangle$$

Now the b threshold is equal:

$$b = \begin{cases} b_1 & \text{if} \quad 0 < \alpha_i < C \\ b_2 & \text{if} \quad 0 < \alpha_j < C \\ (b_1 + b_2/2) & otherwise \end{cases}$$

Details of SVM Package Implementation

In the R package mypkg, SVM is implemented with SMO algorithm using Python. There are five different options for Kernel function such as:
- Linear
- polynomial
- hyperbolic tangent
- gaussian
- $log\left(\frac{\|U-V\|^2}{-2*k^2}\right)$

For U and V being any two data points, and k being a tuning parameter, this is a new kernel we would like to try out throughout this project.

For function SVM(X, Y, C, max_passes, tol, kernal_type, use_kernel=TRUE) in SVM.py:

Input:
C: regularization parameter
tol: numerical tolerance
max passes: max number of times to iterate over α s without changing
$(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$: as the training data format

Output:
α : Lagrange multipliers for solution
b: threshold for solution

Result

```
library("mypkg")
library(reticulate)
py_install("matplotlib")
py_install("cvxopt")
py_install("sklearn",pip = TRUE)

train_date = read.csv("train_date.csv", header = TRUE)
X = c(train_date$Sunshine, train_date$MinTemp, train_date$Rainfall,
      train_date$WindGustSpeed, train_date$WindSpeed3pm, train_date$Humidity9am,
      train_date$Humidity3pm, train_date$Cloud3pm)
y = c(train_date$RainTomorrow)
X = X[0:100]
y = y[0:100]
result_linear <- svmWrapper(X,y, 'linear', 0.2)
result_gaussian <- svmWrapper(X,y, 'gausian', 0.2)
result_polynomial <- svmWrapper(X,y, 'polynomial', 0.2)
result_hyperbolic_tangent <- svmWrapper(X,y, 'hyperbolic_tangent', 0.2)
result <- svmWrapper(X,y, 'kernel', 0.2)
```

The accuracy obtained are as follows:

```
result_linear: 0.9
result_gaussian: 0.76
result_polynomial: 0.8
result_hyperbolic_tangent: 0.6
result: 0.75
```

As the result shown above, the accuracy is stable at a range around 70%-90% with different kernels. This indicates that with the SMO algorithm, the Support-Vector Machine (SVM) still has a high accuracy in prediction outcome. Hence, with the average accuracy above 70%, we can conclude that Support-Vector Machine (SVM) is a good fit for this dataset.

Gradient Boosting Machine

General principles

As a basic principle of machine learning, a Gradient Boosting Machine (GBM) is trying to find a regression function $\hat{f}(x)$, which is able to minimize the expectation of a loss function, $\Phi(y, f)$. The way to optimize and approximate the best function is Friedman's gradient boosting machine, which is using the negative gradient with gradient descent step by step to fit the function. The weak classifier we use is decision trees, which is tree-like models creating splits by the greedy algorithm to minimize the loss function.

In Gradient Boosting Machines, we want to find $f(x)$ such that

$$\hat{f}(x) = arg\ min_{f(x)} E_{y|x}[\Phi(y, f(x))|x]$$

A Parametric regression model assumes that there exists a function $f(x)$ with a finite number of parameters, β , and estimates them by selecting the values that minimize a loss function over a training sample of N observations on (y,x) pair.

$f(x)$, the decision tree model, is a non-parametric model. To find the $\hat{f}(x)$, we are modifying the current model by adding decision trees with a greedy algorithm. we can write this step as function, $f_i = f(x_i)$,

$$\begin{aligned} J(f) &= \Sigma_{i=1}^{N} \Phi(y_i, f(x_i)) \\ &= \Sigma_{i=1}^{N} \Phi(y_i, F_i) \end{aligned}$$

$J(f)$'s negative gradient will lead the direction of the greatest locally decrease in $J(f)$. Now, we can use the Gradient descent algorithm to modify f as

$$\hat{f} \leftarrow \hat{f} - \lambda \bigtriangledown J(F)$$

We treat λ as the study rate in the GBM.

Applying to data

Back to the data, our data is following a Bernoulli distribution. We choose its deviance as loss function such as:

$$-2\frac{1}{w_i}\Sigma w_i(y_i f(xi) - log(1 + exp(f(x_i))))$$

w_i is the weight of each observation during the train. Usually the model will set $w_i = 1$. We will discuss further in the section “Weight boost”. Before the fit, we have to select the number of trees, depth of each tree, and most importantly, the study rate.

1. Initialize $\hat{f}(x)$ according to the following function:

$$log\frac{\Sigma w_i y_i}{\Sigma w_i(1-y_i)}$$

Which is the log of the odds between the negative and positive of y_i , which implies the probility of y_i is observed positive unconditionally.

2. Lead and compute the negative gradient as working response.

$$\begin{aligned} z_i &= -\frac{\partial}{\partial f(x_i)}\Phi(y_i, F_i)|_{f(x_i)=\hat{f}(x_i)} \\ &= y_i - \frac{1}{1+exp(-f(x_i))} \end{aligned}$$

3.Randomly select half of whole dataset, and fit a regression tree with K terminal nodes. K is a parameter that determines the depth of each tree, $g(x) = E(z|x)$.

$$\hat{j}_j^2 = \Sigma_{splits\ on\ x_j} I_t^2$$

I_t^2 is the empirical improvement by spliting on x_j at that point.

4.Compute the terminal node estimates with function:

$$\rho = \frac{\Sigma w_i(y_i - p_i)}{\Sigma w_i p_i(1-p_i)}$$

where p_i is the expit function of $f(x_i)$

We will get the optimal terminal node prediction,

$$\rho_k = arg\ min_\rho \Sigma_{x_i \in s_k} \Phi(y_i, \hat{f}(x_i) + \rho)$$

where S_k is the set of x_s that define terminal node k.

5. Update the estimate of $f(x)$ as

$$\hat{f}(x) \leftarrow \hat{f}(x) - \lambda\rho_k(x)$$

6. Repeat the step 2-5, until the iterator has reached the number of trees previously specified.

Quick Nodes Morph Mend

In this section, we are discussing the relationship between the competing factors: the accuracy and time efficiency, and the parameters they depend on: the size of dataset N, tree depth K, number of decision trees T and the learning rate λ .

Performance analysis

Due to the feature of the GBM, a higher performance requires more training time. An example is λ (represented as “shrinkage” in the code), a smaller λ makes a better fit to training data. However, the marginal increment in goodness of fit decreases as λ increases. To overcome the loss of accuracy due to small λ , we have to increase the number of trees T. This implies that more time is required to train for smaller λ . Good news for us is, the relation between the accuracy, T, and λ is not a linear relationship. The equation of distance between the best performance and current performance is,

$$d = |\alpha - \Sigma_{j=1}^{T} \quad (\lambda)|$$

α is the best accuracy we can achieve in theory. Function $\prod_j(\lambda)$ is the marginal utility of adding a decision tree. which is

$$\prod_j$$

$$\begin{aligned}\Pi_j(\lambda) &= \lambda * \theta(K,N) * \phi(j,\lambda,|\alpha - \Sigma_{i=1}^{j}\Pi_i(\lambda)|) \\ &= \lambda * \Theta * \Phi(j,\lambda)\end{aligned}$$

$\theta(i)$ is utility function of trees, its outcome depends on the depth of tree and the size of dataset, therefore it has a nearly fixed outcome.let $\Theta = \theta(K,N)$

$\phi(i)$ is utility function of trees. Its outcome is a positive number when d is large enough. It oscillates between the negative and positive, when the learning rate is too large to make a better fit on that model. Let $\phi(j,\lambda,|\alpha - \Sigma_{i=1}^{j}\Pi_i(\lambda)|) = \Phi(j,\lambda)$.

Time cost analysis

From the introduction of section “Applying to data”, we get an equation of it

$$\begin{aligned}FLOPS &= Step1 + \Sigma^{T}(Step2 + step3 + step4 + step5) \\ &= NM + N(A+M) + T(N(2A+M+E) + N + N/2 + K*N(M+S)) + (N(M+A) + M + N(2M+A)*K + K) \\ &= O(n) + O(T(O(N) + O(N) + O(N*K) + O(K))) \\ &= O(TKN)\end{aligned}$$

A,M,E is the time cost of addition, mutiplication, and exponentiation, which usually are the constants.

Therefore, size of dataset, number of trees and depth of trees have the same level of influence on the time cost.

Model enhancement

To reduce the operation time, we have to reduce one of the parameters among the size of dataset N, the number of trees T, and the depth of the trees, K. The value N and the value K’s marginal utility is slowly decreasing as these two values increasing until the model hit the theoretically best fit. Therefore, reducing The value N and the value K will always harm the accuracy, before their marginal utility decreases to zero. Otherwise, keep inceasing value K and value N to make their marginal utility reduce to zero is not a good idea. We should balance the cost between three values and reach the best performance in a unit time.

A good idea is to reduce the operation time by cutting the number of trees. We will divide decision trees into several groups. The first groups has the highest learning rate, second group’s learning rate is slightly lower than the first group but higher then the third group. This method makes sure that $\Phi(j,\lambda)$ outputs a positive number, and also raises the marginal utility of adding a decision trees $\prod_j(\lambda)$. The performance equation will be

$$d = |\alpha - \Sigma_{i=1}^{G}\Sigma_{j=1}^{T}\Pi_j(\lambda_i)|$$

G is the number of tree groups, and $\lambda_i = (\lambda_{i-1} - \lambda_G)/r + \lambda_G$ where r is the decreasing rate between two λ_{i-1} in one iteration.

Weight boost

In a general Gradient Boosted Machine, each observations has an equal weight. We are lucky to study with a such large and balance dataset. We also normalized all variables. That is we do not have to reweight for any observation. In this section, we are thinking how does the weight affect the model, if the train data is already balanced.

Approach of reweight

We give the weight of each observation as the product of all their variables frequency and inverse them.

We split the data into two kinds of variables: numeric and factors. From the data processing we know that all numeric data is fitting a normal distribution. Therefore, the subweight of each numeric variable is the probability distribution with that variable’s mean and standard deviation.

The factors variable, we calculate its subweight is simply inverse their appeareth probability.

there are three methods to produce these subweight into a weight, such as mutiply, adds, and mutiply theier Logarithms

We can calculate the weight of an observation by adds such as

$$\Sigma_{x_i \in num} f(x_i \mid_i \mu_{,i} \sigma 2) + \Sigma_{x_i \in fac} P(x_i)$$

We can calculate the weight of an observation by mutiply such as

$$\Pi_{x_i \in num} f(x_i \mid_i \mu_{,i} \sigma 2) + \Pi_{x_i \in fac} P(x_i)$$

We can calculate the weight of an observation by mutiply Logarithms such as

$$\Pi_{x_i \in num} log(f(x_i \mid_i \mu_{,i} \sigma 2)) + \Pi_{x_i \in fac} log(P(x_i))$$

From the result of the code in Appendix we find that. Reweight to a balanced data set is might be a bed idea. it slight effect the accuracy and operation times, when the model is close enough to the best fit. In the rest of time, It will decreasing the accuracy, and lead more bias into model. special mutiply methods and mutiply Logarithms method.They are hardly damaging the accuracy of the model.

Conclusion

By observation of the results in the Appendix, we are concluding that, the Quick Nodes Morph Mend is truly helpful. Under the same time conditions and weight condition, the Morph Model always have a better performance. Also, the shorter the time, the more obvious the advantage.

Appendix

Data pre-processing code:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import category_encoders as ce


# train_date, val_data = train_test_split(dataframe,test_size=0.2,random_state=128364578)
# val_data = df_to_dataset(val_data, batch_size=batch_size)


old = pd.read_csv("weatherAUS.csv",low_memory=False)

col_names = old.columns
df = old.copy()

df.drop(['RISK_MM'], axis=1, inplace=True)

df['Weeknum'] = pd.to_datetime(df['Date']).dt.week
df.drop('Date', axis=1, inplace = True)

df.drop('Temp3pm', axis=1, inplace = True)
df.drop('Temp9am', axis=1, inplace = True)
df.drop('Pressure9am', axis=1, inplace = True)

categorical = [col for col in df.columns if df[col].dtypes == 'O']
numerical = [col for col in df.columns if df[col].dtypes != 'O']

#df.describe()

for col in numerical:
   col_median=df[col].median()
   df[col].fillna(col_median, inplace=True)

tp = categorical.copy()
tp.pop(0) # drop the location
for col in tp:
    df[col].fillna(df[col].mode()[0], inplace=True)

def max_value(df3, variable, top):
    return np.where(df3[variable]>top, top, df3[variable])


df['Rainfall'] = max_value(df, 'Rainfall', 3.2)
df['Evaporation'] = max_value(df, 'Evaporation', 21.8)
df['WindSpeed9am'] = max_value(df, 'WindSpeed9am', 55)
df['WindSpeed3pm'] = max_value(df, 'WindSpeed3pm', 57)
df['WindGustSpeed'] = max_value(df, 'WindGustSpeed', 91)

#['Location', 'WindGustDir', 'WindDir9am', 'WindDir3pm', 'RainToday', 'RainTomorrow']

scaler = MinMaxScaler()
df[numerical] = scaler.fit_transform(df[numerical])

train_date, val_data = train_test_split(df,test_size=0.2,random_state=128364578)

df_coded = pd.concat([df[numerical],pd.get_dummies(df.Location),
                      pd.get_dummies(df.WindGustDir),
                      pd.get_dummies(df.WindDir9am),
                      pd.get_dummies(df.WindDir3pm),
                      pd.get_dummies(df.RainToday),
                      df['RainTomorrow']], axis=1)
```

```
scaler = MinMaxScaler()
df[numerical] = scaler.fit_transform(df[numerical])

df['RainTomorrow']=df['RainTomorrow'].map({'Yes': 1, 'No': 0})
df['RainToday']= df['RainToday'].map({'Yes': 1, 'No': 0})

#['Location', 'WindGustDir', 'WindDir9am', 'WindDir3pm', 'RainToday', 'RainTomorrow']

df_encode = pd.concat([df[numerical],
                       pd.get_dummies(df.Location),
                       pd.get_dummies(df.WindGustDir),
                       pd.get_dummies(df.WindDir9am),
                       pd.get_dummies(df.WindDir3pm),
                       df['RainToday'],
                       df['RainTomorrow']], axis=1)
useless_cols = ['RainToday','WindSpeed9am','Cloud9am','Evaporation']

df.drop(useless_cols, axis=1, inplace = True)
df_encode.drop(useless_cols, axis=1, inplace = True)


train_date, val_data = train_test_split(df,test_size=0.2,random_state=128364578)

train_date.to_csv("train_date.csv")
val_data.to_csv("val_data.csv")

train_date, val_data = train_test_split(df_encode,test_size=0.2,random_state=128364578)

train_date.to_csv("train_date_coded.csv")
val_data.to_csv("val_data_coded.csv")
```

GBM Testing code

```
library('mypkg')
library('pROC')
library('tictoc')
setwd("E:/440proj")


train_data = read.csv("train_date.csv")
val_date =  read.csv("val_data.csv")

train_data = train_data[,-which(names(train_data) %in% 'X')]
val_date = val_date[,-which(names(val_date) %in% 'X')]

ntree = 200
shrinkage = 0.015
interaction.depth = 12 #
n.minobsinnode = 13
tic("gmb ")
a = mygbm(RainTomorrow~.,data = train_data,n.trees = ntree,shrinkage = shrinkage,interaction.depth =interaction.de
pth ,bag.fraction = 2,n.minobsinnode = n.minobsinnode,keep.data = TRUE)
toc()
best_tree = which.min(a[["train.error"]])
Predicted = predict(a, val_date, n.trees=best_tree)


roc(val_date$RainTomorrow,Predicted, plot = FALSE)

weights = weightCal(train_data,c("X","RainTomorrow"),"Rainfall",patten = "a")
tic("gmb +a")
a = mygbm(RainTomorrow~.,data = train_data,weights = weights,n.trees = ntree,shrinkage = shrinkage,interaction.dep
th =interaction.depth ,bag.fraction = 2,n.minobsinnode = n.minobsinnode,keep.data = TRUE)
toc()
best_tree = which.min(a[["train.error"]])
Predicted = predict(a, val_date, n.trees=best_tree)


roc(val_date$RainTomorrow,Predicted, plot = FALSE)


weights = weightCal(train_data,c("X","RainTomorrow"),"Rainfall",patten = "m")
tic("gmb +m")
a = mygbm(RainTomorrow~.,data = train_data,weights = weights,n.trees = ntree,shrinkage = shrinkage,interaction.dep
th =interaction.depth ,bag.fraction = 2,n.minobsinnode = n.minobsinnode,keep.data = TRUE)
toc()
best_tree = which.min(a[["train.error"]])
Predicted = predict(a, val_date, n.trees=best_tree)


roc(val_date$RainTomorrow,Predicted, plot = FALSE)

weights = weightCal(train_data,c("X","RainTomorrow"),"Rainfall",patten = "l")
tic("gmb +l")
a = mygbm(RainTomorrow~.,data = train_data,weights = weights,n.trees = ntree,shrinkage = shrinkage,interaction.dep
th =interaction.depth ,bag.fraction = 2,n.minobsinnode = n.minobsinnode,keep.data = TRUE)
toc()
best_tree = which.min(a[["train.error"]])
Predicted = predict(a, val_date, n.trees=best_tree)


roc(val_date$RainTomorrow,Predicted, plot = FALSE)

tic("qnmm")
qnm = Qnmbrm(RainTomorrow~.,
             data = train_data,stage.level=3,stage.rate=2, n.trees = 120,
```

```
             interaction.depth = 13, n.minobsinnode = 10, shrinkage = 0.1,shrinkage.top=0.1)
toc()
best_tree = which.min(qnm[["train.error"]])
Predicted = predict(qnm, val_date, n.trees=best_tree)
roc(val_date$RainTomorrow,Predicted, plot = FALSE)

weights = weightCal(train_data,c("X","RainTomorrow"),"Rainfall",patten = "a")
tic("qnmm + a")
qnm = Qnmbrm(RainTomorrow~.,
             data = train_data,stage.level=3,stage.rate=2, n.trees = 120, weights=weights ,
             interaction.depth = 13, n.minobsinnode = 10, shrinkage = 0.1,shrinkage.top=0.1)
toc()
best_tree = which.min(qnm[["train.error"]])
Predicted = predict(qnm, val_date, n.trees=best_tree)
roc(val_date$RainTomorrow,Predicted, plot = FALSE)


weights = weightCal(train_data,c("X","RainTomorrow"),"Rainfall",patten = "m")

tic("qnmm+m")
qnm = Qnmbrm(RainTomorrow~.,
             data = train_data,stage.level=3,stage.rate=2, n.trees = 120, weights=weights ,
             interaction.depth = 13, n.minobsinnode = 10, shrinkage = 0.1,shrinkage.top=0.1)
toc()
best_tree = which.min(qnm[["train.error"]])
Predicted = predict(qnm, val_date, n.trees=best_tree)
roc(val_date$RainTomorrow,Predicted, plot = FALSE)


weights = weightCal(train_data,c("X","RainTomorrow"),"Rainfall",patten = "l")

tic("qnmm+l")
qnm = Qnmbrm(RainTomorrow~.,
             data = train_data,stage.level=3,stage.rate=2, n.trees = 120, weights=weights ,
             interaction.depth = 13, n.minobsinnode = 10, shrinkage = 0.1,shrinkage.top=0.1)
toc()
best_tree = which.min(qnm[["train.error"]])
Predicted = predict(qnm, val_date, n.trees=best_tree)
roc(val_date$RainTomorrow,Predicted, plot = FALSE)
```

Result of GBM testing code

```
gmb : 66.78 sec elapsed
Area under the curve: 0.8773
gmb +a: 68.36 sec elapsed
Area under the curve: 0.8768
gmb +m: 72.64 sec elapsed
Area under the curve: 0.7027
gmb +l: 67.4 sec elapsed
Area under the curve: 0.7181
qnmm: 58.44 sec elapsed
Area under the curve: 0.892
qnmm + a: 57.8 sec elapsed
Area under the curve: 0.8923
qnmm+m: 60.33 sec elapsed
Area under the curve: 0.893
qnmm+l: 60 sec elapsed
Area under the curve: 0.8929
```

Reference

Rohith, Gandhi. Support Vector Machine — Introduction to Machine Learning Algorithms. Towards Data Science. URL

(https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47), (2018).

Lujing, Chen. Support Vector Machine - Explained (Soft Margin/Kernel Tricks). Bite-Sized Machine Learning. URL (https://medium.com/bite-sized-machine-learning/support-vector-machine-explained-soft-margin-kernel-tricks-3728dfb92cee), (2018).

Saishruthi, Swaminathan. Logistic Regression — Detailed Overview. Towards Data Science. URL (https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc), (2018)

Pascal, Poupart. CS480/680 Lecture 8. University of Waterloo. URL (https://cs.uwaterloo.ca/~ppoupart/teaching/cs480-spring19/slides/cs480-lecture8.pdf%20https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc), (2019).

J. H. Friedman. Stochastic gradient boosting. Computational Statistics and Data Analysis, 38(4):367–378, 2002.

Ridgeway, Greg. Generalized Boosted Models: A Guide to the Gbm Package. 2019.