A Practical Application at Binary-class Classification Techniques

MA4270 Term Project Report

Wang Jiangyi, Yao Yuhan
National University of Singapore

Contents

1 Introduction

Nowadays, it is common that we will take weather forecast into consideration when we have to make a day plan. However, how can we make such prediction based on the previous records?

Actually, such a task is a binary-classification problem. In this semester, we have learnt many algorithms which are aimed to handle these classification problems, like "Logistic Regression". In this report, we will analyze the dataset `'weatherAUS.csv'` from Kaggle which contains 145,461 pieces of Australian weather data in past ten years.

Our goal is to find the optimal machine learning method to predict next-day rain in Australia corresponding the optimal test accuracy to the dataset. In the following sections, we will employ the algorithms "Logistic Regression", "AdaBoost", "Gradient Boost", "eXtreme Boost".

All experiments in this report are done in PyCharm. The main packages we use are `scikit-learn`, `pandas`, `seaborn` and `plotly`. The former two can be used to train our models and the latter two can be used to visualize the outcomes.

2 Dataset

This dataset is attained from Kaggle, and our aim is to predict the next-day rain with the help of past-10-year weather records. However, in this dataset, there are MANY missing values among features. This requires us to do pre-processing carefully to our dataset. Additionally, it can be observed that there are several features which are HIGHLY dependent(e.g. `'MaxTemp'` and `'Temp3pm'` almost decribe the same thing). This implies that we should do feature selection to delete the highly dependent features.

2.1 Primal Dataset

The 145,461 points in the dataset each have 22 features as indicated in Table 1:

Feature	Feature Discription
Date	date of record
Location	location of record
MinTemp	minimum temperature in the 24 hours
MaxTemp	maximum temperature in the 24 hours
Rainfall	precipitation in the 24 hours(millimetres)
Evaporation	pan evaporation in the 24 hours(millimetres)
Sunshine	bright sunshine in the 24 hours(hours)
WindGustDir	the STRONGEST wind direction
WindGustSpeed	the STRONGEST wind speed
WindDir9am	wind direction averaged over 10 minutes prior to 9 am
WindDir3pm	wind direction averaged over 10 minutes prior to 3 pm
WindSpeed9am	wind speed averaged over 10 minutes prior to 9 am
WindSpeed3pm	wind speed averaged over 10 minutes prior to 3 pm
Humidity9am	relative humidity at 9 am(percentage)
Humidity3pm	relative humidity at 3 pm(percentage)
Pressure9am	atmospheric pressure reduced to mean sea level at 9 am
Pressure3pm	atmospheric pressure reduced to mean sea level at 3 pm
Cloud9am	fraction of sky obscured by cloud at 9 am(eighths)
Cloud3pm	fraction of sky obscured by cloud at 3 pm(eighths)
Temp9am	temperature at 9 am
Temp3pm	temperature at 3 pm
RainToday	whether rain or not today(binary)

Table 1: features and their descriptions of Kaggle Dataset

2.2 Pre-processing

At the beginning of this section, it is worthwile to point out that there are **Four** main challenging problems according to our dataset.

- Missing Values
- Feature Dependence
- Encoding of Categorical Data
- Label Imbalance

2.2.1 Missing Values

To be more precise, while checking the details of this dataset, it can be noticed that there are MANY missing values among features, which can be shown in Figure 1:

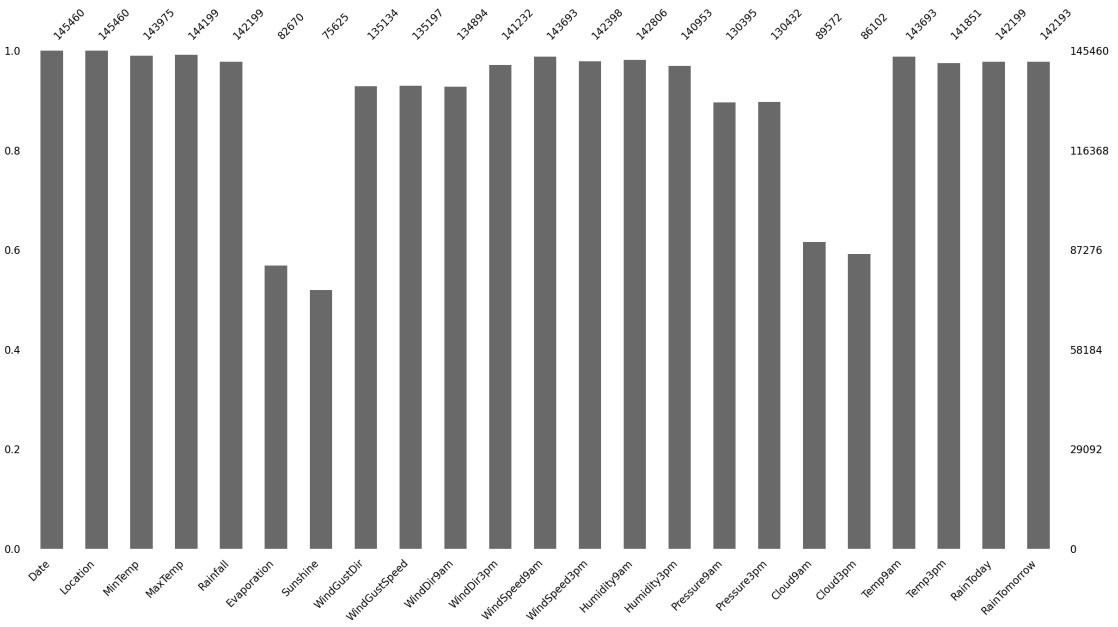


Figure 1: Missing Values of Features

From the bar, it is obvious that over one half of the dataset corresponding to features `'Evaporation'`, `'Sunshine'`, `'Cloud9am'` and `'Cloud3pm'` are missing.

In this report, we apply different fill-in methods to Labels and Features:

1. As for the features, considering that changes of them might be relatively consecutive in two adjacent days, we use the Median Method for filling in the missing values.

2. As for labels, we choose to drop out all missing values. On one hand, it can be seen from Figure 1 that there are **few** data points which have missing value in labels (the last column in Figure 1). On the other hand, some of the machine learning may be sensitive to the mislabels. Therefore, something bad may happen if we fill in the missing values incorrectly.

2.2.2 Feature Dependence

When we check the name of features, we find that there are some features have the similar meanings. For example, `'MaxTemp'` and `'Temp3pm'` will share the approximate value since it is a common sense that the

temperature is relatively high at 3pm.

Moreover, such dependent relationship between different numerical features can be easily seen from the Covariance Diagram below:

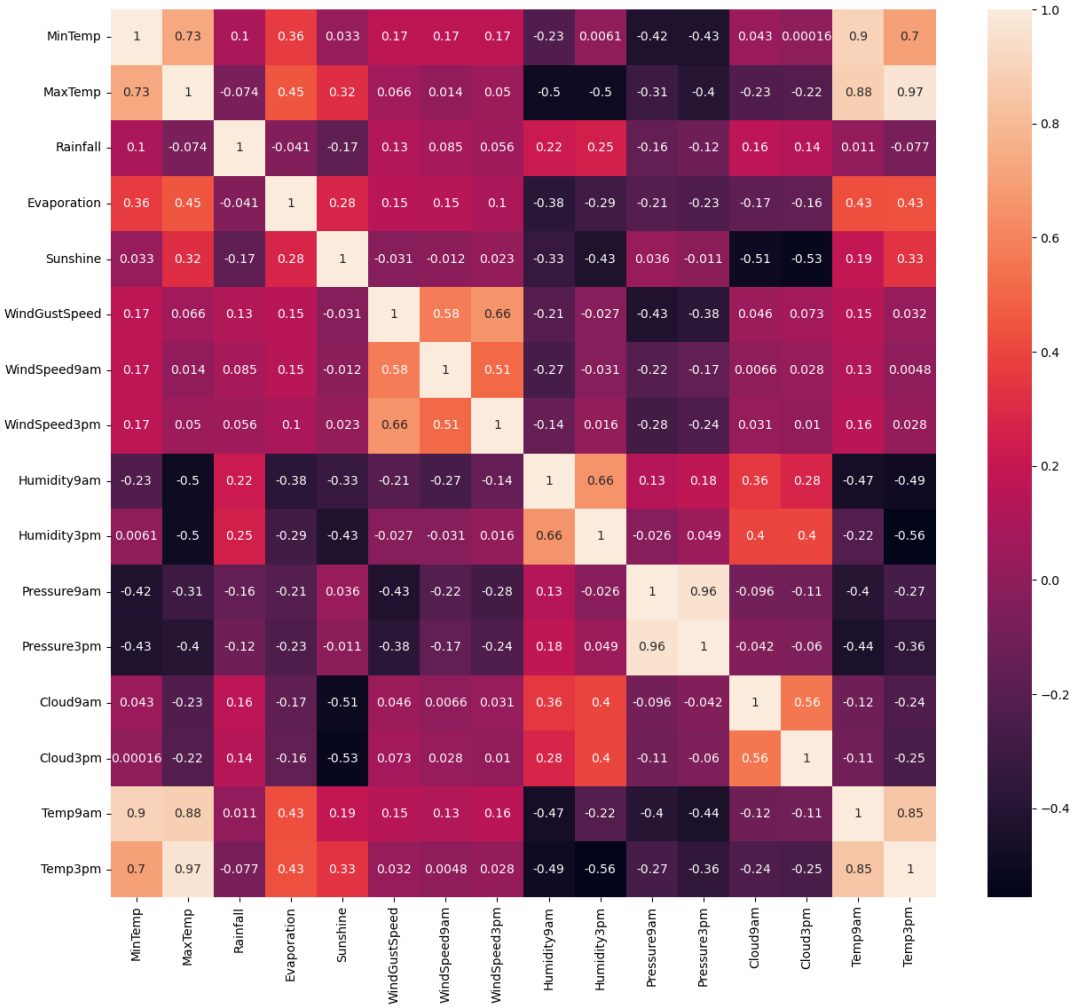


Figure 2: Covariance Diagram of Numerical Features

From Figure 2, it is apparent that in the left lower corner, the corvariance between features is extremely close to 1, which clarifies our previous intuition. Therefore, we drop the features `'Temp9am'` and `'Temp3pm'` since they are highly related to the feature `'MaxTemp'`. Furthermore, we drop the feature `'Date'` as well.

Up to now, the number of features in dataset is 19.

2.2.3 Encoding of Categorical Data

In our lectures, all the inputs are Numeric, whose quantities are meaningful. Just take our exam score as an example. If student A attains 90 marks and student B attains 60 marks in one exam, then we say this 'feature' is Numeric. This is because the value of this 'feature' is meaningful, which can be seen from the difference between the two marks. In other words, this quantity of this difference (i.e. the difference between two marks) can be viewed as a measure of the performance difference between two students.

However, when it comes to the Categorical features, things are totally different. Let me take types of toys as

an example. Suppose you have a car, a bike and a plane as your toys, then how can we mark them?

One simple way is to mark car, bike and plane with 1, 2 and 3 respectively. However, with such kind of marks, many machine learning algorithms will think the difference between **cars** and **bikes** is different from that between **cars** and **planes**. This is unreasonable when we consider that these three things are three equivalent categories, which means the difference between any two of them should be equal.

Based on this, we apply *one-hot encoding* over the whole dataset. Let me show the encoding process by means of feature `'WindGustDir'` in our dataset. Actually, this feature has 17 different values, each of which represents one certain directions. Therefore, we map the i-*th* direction to e_i, and e_i is the i-*th* standard basis in $\mathbb{R}^{17}$. With this mapping, each dimension (17 dimension in total) has its description, which represents the corresponding direction.

Note that there are all 5 categorical features in our dataset (`'Location'`, `'WindGustDir'`, `'WindDir9am'`, `'WindDir3pm'`, `'RainToday'`). We repeat the above process for all of them. After filtering the one-hot encoding, each categorical feature is now a length 99 binary feature vector, and the total number of all features is 113 now (14 more numeric features).

2.2.4 Label Imbalance

It is a common sense that for most countries, the days of rain and no rain are not always similar, and so is Australia. Since our dataset is attained by recording the meteorological conditions over the past ten years, it must lead to the label imbalance.

To see this more intuitively, we visualize this situation in Figure 3:

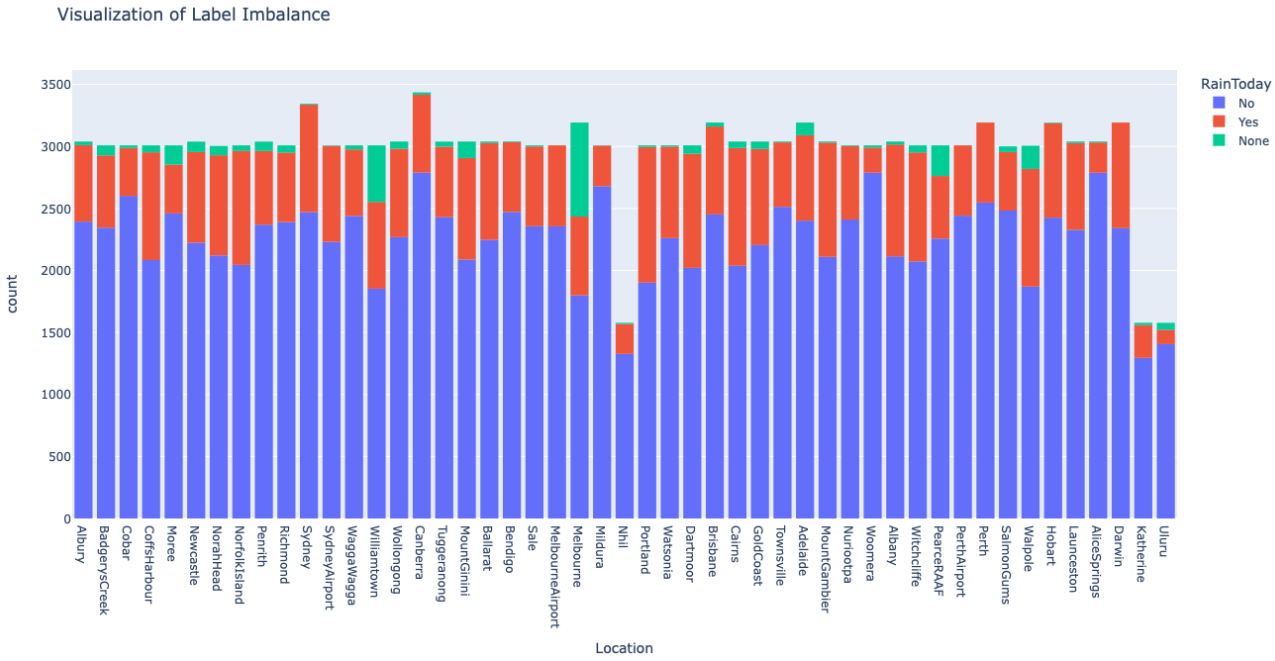


Figure 3: Visualization of Label Imbalance

In Figure 3, the red color represents feature `'RainToday'` is 'Yes', and the blue color represents the feature is 'No'. Therefore, in most countries (`'x'` axis), only about one quarter of the days are rainy. Therefore, in our dataset, we will encounter a big problem of label imbalance.

Because of label imbalance, something interesting happens and we will discuss more details about it later in **Section 4.2**. Here we give some motivation about it.

Suppose that we have a naive classifier, which predicts that next day will never rain. When we consider its performance, we can know that this naive classifier will achieve about 75% testing accuracy since from the previous recordings, we are convinced that only one-forth of the days will rain.

For one thing, compared with random guess, which is supposed to achieve 50% accuracy, the naive classifier is relatively good. Therefore, this gives us a lower bound of the training accuracy. If we learn some classifiers whose testing accuracy is far less than the naive one, then it is hard for us to convice ourselves that the classifiers we learnt are good. For this reason, we name the naive classifier (the simplest decision tree who has no branches) with 'Baseline', which can be used to judge the performance in some respects.

For another thing, although the naive classifier will achieve a relatively high testing accuracy, it does not mean it is good, at least in practice. The point is, we do classify every no-rainy day correctly, but at the same time, we classify every rainy day incorrectly. It is terrible in our daily life since we cannot predict the rainy days at all. Actually, we care about more about rainy days than no-rainy days since we may get cold without taking an umbrella with us in rainy days. This implies that we should change the criterion of model evaluation.

In our following experiments, we will use *Confusion Matrix* to evalute our models. For binary classification problem, Confusion Matrix is a 2×2 matrix defined as below:

		Actual Values	
		Positive(1)	Negative(0)
Predict Values	Positive[1]	TP	FP
	Negative[0]	FN	TN

Table 2: Confusion Matrix

In Table 2, 'TP' and 'TN' represents those testing points which are classified correctly in positive and nagative class repectively. Similarly, 'FP' and 'FN' represents those which are classified incorrectly in respective classes. With the help of confusion matrix, we are able to analyse the training accuracy in different classes, which is more reasonable compared with the overall training accuracy.

Moreover, in order to reduce the degree of label imbalance, we apply the SMOTE[3] in our dataset, which can be used to over-sample. That is, we can increase the number of data points in minority class by some clever strategies. The motivation is that it will guide our machine learning algorithms to make more predictions on the minority labels with more training data in minority class. In our report, we set the ratio of over-sampling as `'Rainy': 76190`, `'No-rainy': 76190` (In original dataset, the ratio is `'Rainy': 21798`, `'No-rainy': 76190`). More details of related results will be discussed in Section 4.

2.3 Training and testing sets

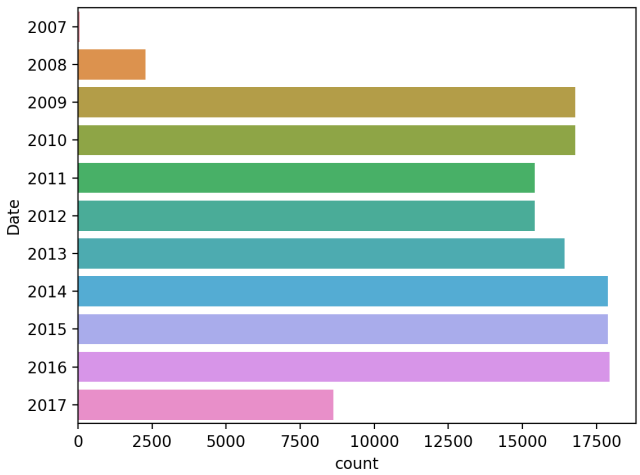


Figure 4: Different Number of Data Points in 10 Years

In Figure 4, in order to ensure fair evaluation, we can partition the dataset into 2 groups - those data

points which are before 2015 are in the training set (70%) and those after 2015 (including 2015) are in the testing set (30%). Furthermore, while selecting models, i.e., tweaking hyper-parameters, we will perform Cross Validation on the training set. Finally, after finding the best hyper-parameters, we use testing set to evaluate its performace.

We choose to split the dataset in this way because we want to preserve the time structure for our training and testing sets. Notice that our aim is to use the previous information to make predictions, and this also should be the aim for our choice of training and testing sets. That is why we construct the training and testing sets by splitting time axis.

3 Proposed Methods

After pre-processing, we are able to apply our machine learning algorithms to our modified dataset. Since our aim is to make prediction to a binary-classifcation problem, we choose the "Logistic Regression" and "AdaBoost" Algorithm, which are taught in lectures. Furthermore, considering that "AdaBoost" is the original one in Boosting Algorithm, we try some more advanced Boosting Algorithms like "Gradient Boost" and "eXtreme Boost" to see the improvement compared with AdaBoost. More details will be discussed in the following subsections.

In order to achieve good models, we apply a so-called grid search method over the relevant tuning parameter values for all the algorithms above. This step can be done by `'sklearn.GridSearchCV'` package in Python. In this method, we apply Cross Validation in training set to learn the best hyper-parameters. After that, we evaluate our models in testing set to approximate the genralization error.

3.1 Logistic Regression

Recall that Losgistic Regression is a Probablisitic and Discriminative Model[1]. It is worthwhile to emphasize that it can be viewed as one special case of Generalized Linear Model (GLM) for two-class classification. Therefore, there are many beautiful properties in Logistic Regression.

In Logistic Regression, we model the posterior distribution with logistic sigmoid function:

$$p(C_1|\phi) = \sigma(w^T\phi)$$

When we consider the normal Logistic Regression (without regularization) in affinely-separable dataset, the resulting probability values will be close to 1. This means we will overfit the dataset. To avoid this, in our report, we apply the Logistic Regression with regularization in the experiments. That is to say, we try to solve the optimization problem (1):

$$min : \frac{\lambda}{2}||\boldsymbol{\theta}||^2 + \sum_{t=1}^{n} log[1 + exp(-y_t(\langle \boldsymbol{x_t}, \boldsymbol{\theta}\rangle + \theta_0))] \tag{1}$$

Then we use grid search method to determine the best hyper-paremeter λ.

3.2 AdaBoost

AdaBoost is an algorithm which is invented by Freund and Schapire, and there are many interesting properties about it, including resistance to overfitting. Furthermore, to explain its resistance to overfitting, margin theory is introduced and an upper bound to its genralization error is given, which is not related to the number of weak learners[2]. This algorithm has been covered in MA4270 lecutres. Therefore, in order to see its performance, it is meaningful to test it in our dataset. However, as far as I am concerned, it seems that it is not so useful in application.

3.2.1 Algorithm

Recall the **algorithm** of AdaBoost, which can be concluded into three steps:

- Reassign the weights for all data points
- Find the best weak learner corresponding to the current weights

- Find the best vote corresponding to the current weak learner

We can also apply regularization to this Additive Model, that is, shrinking the vote α_t by a hyper-parameter v. In convention, this hyper-parameter v is so-called 'Learning rate'. Due to this, the Additive Model changes to (2):

$$h_m(\boldsymbol{x}) = h_{m-1}(\boldsymbol{x}) + v\alpha_m h(\boldsymbol{x}; \boldsymbol{\theta_m}) \tag{2}$$

3.2.2 Experiment Setting

In our experiment, we will use grid search method to determine the following hyper-parameters `'learning_rate'` and `'n_estimators'` from a large searching space and fix `'min_samples_leaf'`, `'min_samples_split'` and `'max_depth'`. This allows us to attain the optimal classifer in a certain searching space.

In lecture, we choose the weak learner as a decision stump. Note that this is equivalent to set `'max_depth'=1`, which is the simplest weak learner as for decision trees. However, in experiments, we want to attain better performance in our models. Therefore, we do need to make our weak learners a bit more complicated.

3.3 Gradient Boost

Gradient Boost (GBM[5]) is another famous algorithm in Boosting, which is not taught in lectures. It can be viewed as an extension of AdaBoost by enlarging the allowable loss function. In our report, we mainly consider one special type with Decision-Tree weak learner, namely Gradient Boost Decision Tree (GBDT). Note that if we choose the loss function as exponential loss, then GBDT degenerates into AdaBoost.

3.3.1 Motivation

Here we give the **motivation** of this algorithm as follows:

Our **ultimate aim** is to minimize the loss function $Loss(h_m(x))$. Assume that our model is additive, then we can rewrite it as $Loss(h_{m-1}(x) + \alpha_m h(x))$.

Recall that in AdaBoost, the loss function is fixed as the exponential loss, which has a good property of separation. In other words, we are able to separate the term $h_{m-1}(x)$ and $\alpha_m h(x)$. Then, by using greedy trick, we can attain the new vote and weak learner in a closed form.

Therefore, we want to apply the above framework to any loss function. The BIG question is, for arbitrary loss function, we are not able to achieve property of separation. Then, the big trick, Taylor Expansion, comes to do us a favor. That is because, with Taylor Expansion at the point $h_{m-1}(x)$, we can trivially approximate the primal loss function with a affine function of $\alpha_m h(x)$, which has the form (3):

$$Loss(h_{m-1}(x) + \alpha_m h(x)) \approx Loss(h_{m-1}(x)) + W_{m-1}(x)\alpha_m h(x) \tag{3}$$

where constant $W_{m-1}(x) = \dfrac{\partial Loss(h_{m-1}(x), y)}{\partial h_{m-1}(x)}$

All the rest we need to do is similar to AdaBoost since we separate the objective into two separate parts.

3.3.2 Algorithm

Based on the motivation, we can also conclude the GBDT **algorithm** into three steps:

- Reassign the weights $W_{m-1}(x_t) = \dfrac{\partial Loss(h_{m-1}(x_t), y)}{\partial h_{m-1}(x_t)}$ for all data points x_t
- Find the best weak learner corresponding to the current weights $W_{m-1}(x_t)$
- Find the best vote corresponding to the current weak learner

3.3.3 Experiment Setting

In our experiment, we will use grid search method to determine the following hyper-parameters, which can be also divided into two types, namely Boosting parameters and Tree-Special parameters. As for Boosting parameters, what we consider is similar to AdaBoost, except for an additional parameter `'subsample'`. As for Tree-Special paramters, we mainly deal with `'max_depth'`, `'min_samples_split'` and `'min_samples_leaf'`, which control the strategy of generating weak learners.

3.4 eXtreme Boost

eXtreme Boost (XGBoost[4]) Algorithm is a leap of the traditional GBDT, which is put forward by Tianqi Chen. It is the algorithm that combines efficiency with performance, which is widely in use on many occasions. Compared with GBDT, it has several advantages:

- take both the first and second order Taylor Expansion into consideration to achieve better approximation
- take regularization of the scale of trees into consideration
- use column subsampling, which is applied in Random Forest
- use greedy algorithm to construct the weak learner, i.e., the decision tree, which is similar to ID3 (entropy) and CART (gini) algorithm

3.4.1 Main ideas

Firstly, this time, we consider the regularized loss function $L^{(m)}(h)$, which is defined as (4):

$$L^{(m)}(h_m) = \sum_{t=1}^{n} Loss(h_{m-1}(x_t) + \alpha_m h(x)) + \Omega(h_{m-1}) + \Omega(h) \tag{4}$$

Then, we expand loss function to second order, which is different from GBDT. In XGBoost, we are able to compute the optimal parameters in closed form, which will give the exact optimal value of objective as (5):

$$\hat{L}^{(m)}(q) = -\frac{1}{2}\sum_{j=1}^{T}\frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma T \tag{5}$$

where g_i and h_i represent the first-order partial derivative and second-order partial derivative respectively.

We use (5) as a scoring function to measure the quality of a tree structure q, which is similar to the entropy in ID3 algorithm.

Finally, we apply greedy algorithm by means of scoring function to generate the weak learner and do pruning.

3.4.2 Algorithm

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i\in I} g_i$, $H \leftarrow \sum_{i\in I} h_i$
for $k = 1$ ***to*** m **do**
  $G_L \leftarrow 0$, $H_L \leftarrow 0$
  for j *in sorted(I, by* $\mathbf{x}_{jk}$*)* **do**
    $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L$, $H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split with max score

3.4.3 Experiment Setting

In XGBoost algorithm, we will still use grid search method to determine the hyper-parameters. To compare the performance with GBDT, we choose to determine the same hyper-parameters as we do in GBDT.

4 Experiment

The dataset **D** to be analysed is the one we derived in Section 2 which has 114 features. Experiments will be made upon dataset **D** with all the methods specified in Section 3.

Before discussing the experiments of above four algorithms in detail, it is important to determine the measure of a good model. We assume that there are two evaluation criteria, which are '**accuracy**' and '**recall**', and there are two stories for them.

As for 'accuracy', it measures the probability that our prediction is correct. As for 'recall', it can be regarded as the probability that we can correctly predict the positive class, i.e., rainy days.

In our following discussion, we will apply both of the two criteria to non-oversampling case by grid search method, which can give us the optimal hyper-parameters in searching space respectively. This can be regarded as a quantitative analysis. Furthermore, we will just choose the hyper-parameters by our hand in oversampling case. Although we do not give the optimal hyper-parameters in this case, it can illustrate a qualitative analysis, which is able to show some advantages of over-sampling in some sense (we will only show **part** of our results in this report for the limit of space, and the whole results will be attached to the Appendix).

4.1 Non-oversampling

4.1.1 Logistic Regression

The result from a grid search can be shown in Table 3, and in the grid search method, we choose two different evaluation criteria (`'Accuracy'` and `'Recall'`) to search for the optimal hyper-parameters. Note that our searching space is, `'C': [0.2, 0.5, 1, 2, 5, 10, 15, 20, 50, 100]`.

'C'	Cross-Validation Accuracy	Cross-Validation Recall
0.2	0.813161	0.502889
0.5	0.805231	0.503072
1	0.798996	0.501696
2	0.793996	0.501328
5	0.791424	0.500686
10	0.790475	0.509494
15	0.790393	0.509219
20	0.790005	0.509907
50	0.789720	0.512246
100	0.789373	0.512888

Table 3: results of grid search for Logistic Regression

From Table 3, we can conclude that if we prefer higher accuracy, then we set `C=0.2`, with the CV-accuracy is equal to **0.813**. However, if we prefer higher recall, then we set `C=100`, with CV-recall is equal to 0.513. Suppose that we are conservative, then we will choose `C=100` as the model we want. Therefore, we can apply this model to the testing set in order to see its performance. The confusion matrix of our prediction is shown in Figure 5:

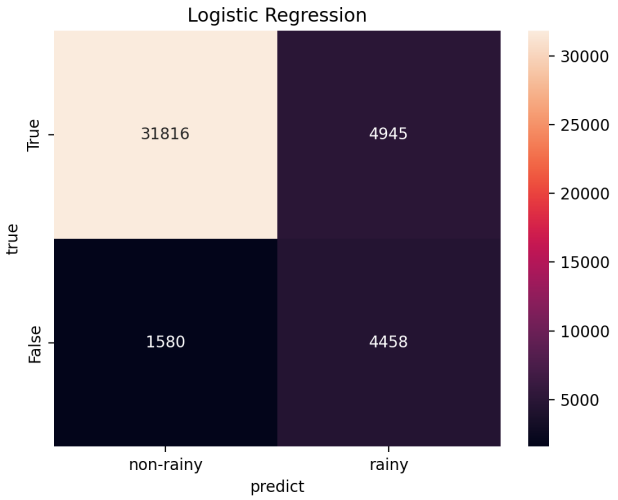


Figure 5: Logistic Regression Confusion Matrix

From Figure 5, we can conclude that our prediction is good since the diagonal entries are very large, which means the testing accuracy is high (**0.8475** approximately).

For simplicity, in the following discussion, we all seek for the optimal classifiers to attain the **best CV-recall** (the full results will be attached to Appendix).

4.1.2 AdaBoost

The result from a grid search on selecting hyper-parameters for AdaBoost Algorithm is shown in Table 4. Note that our searching space is, `'learning_rate': [0.3, 0.7, 1]` and `'n_estimators': [50, 150, 400]` with fixed `'min_samples_leaf'=4`, `'min_samples_split'=400` and `'max_depth'=6`.

`'learning_rate'`	`'n_estimators'`	Cross-Validation Recall
0.3	50	0.499677
0.7	50	0.511971
1	50	0.516422
0.3	150	0.514587
0.7	150	0.521743
1	150	0.518761
0.3	400	0.524084
0.7	400	0.529175
1	400	0.523899

Table 4: results of grid search for AdaBoost

From the grid search method, we obtain the best Cross-Validation Recall of **0.529175** for learning rate of 0.7 and for 400 estimators (DecisionTreeClassifier). Therefore, we set hyper-parameters to `'learning_rate'=0.7`, `'n_estimators'=400`, `'min_samples_leaf'=4`, `'min_samples_split'=400` and `'max_depth'=6`. Then we apply this model to the testing set and obtain the confusion matrix as Figure 6 shows:

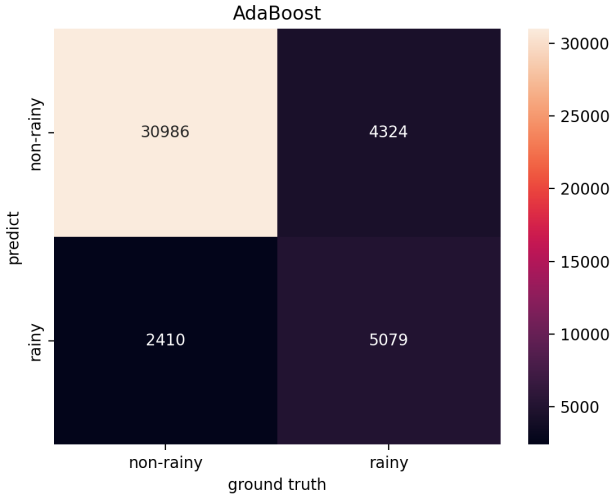


Figure 6: AdaBoost Confusion Matrix

Compared with Figure 5, it can be noticed that the testing accuracy is slightly lower (Logistic Regression: **0.8475**; AdaBoost: **0.8423**). More significantly, we actually achieve much better 'recall' in the testing set. That is to say, we can better predict rainy days at the cost of some tolerable decrease in predicting non-rainy days. In this sense, we do achieve a better model by AdaBoost Algorithm.

4.1.3 Gradient Boost

The result from a grid search on selecting hyper-parameters for GBDT Algorithm is shown in Table 5. Here, we search for the optimal parameters by two steps:

Firstly, we search for Boosting parameters `'learning_rate'`, `'subsample'`and `'n_estimators'`. After some simple experiments, we determine the searching space, that is, `'subsample'`=`[0.9]`, `'n_estimators'`: `[10, 20, 30]`, `'learning_rate'`: `[0.2, 0.3, 0.35, 0.37, 0.4]`.

Secondly, after fixing the `'learning_rate'`, `'subsample'`and `'n_estimators'`, we start to search for Tree-Special parameters, i.e., `'min_samples_split'`, `'min_samples_split'` and `'max_depth'` in searching space `'min_samples_split'`: `[50, 150, 300]`, `'min_samples_leaf'`: `[5, 10]`, `'max_depth'`: `[4, 6, 8, 10]`.

`'max_depth'`	`'min_samples_leaf'`	`'min_samples_split'`	Cross-Validation Recall
4	5	150	0.511698
4	5	300	0.513532
4	10	150	0.514450
4	10	300	0.511376
⋮	⋮	⋮	⋮
8	5	150	0.518854
8	5	300	0.516284
8	10	150	0.519313
8	10	300	0.516239
10	5	150	0.519588
10	5	300	0.520964
10	10	150	0.522799
10	10	300	0.523167

Table 5: results of grid search for GBDT

From Table 5, we obtain the best Cross-Validation Recall of 0.52316 for learning rate of 0.35 and for 30 estimators. Therefore, we set hyper-parameters as follows, `'learning_rate'=0.35`, `'n_estimators'=30`, `'subsample'=0.9`, `'min_samples_leaf'=10`, `'min_samples_split'=300` and `'max_depth'=10`. Then we apply this model to the testing set and have the confusion matrix as Figure 7 shows:

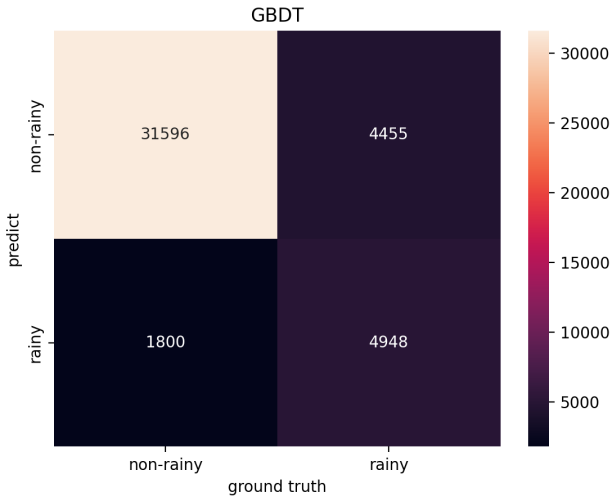


Figure 7: GBDT Confusion Matrix

From Figure 7, we notice that GBDT algorithm absorbs all the advantages in the above results of those two algorithms - Logistic Regression and AdaBoost. That is to say, as for 'accuracy', the testing accuracy of GBDT is **0.8522**, which is the best of the three algorithms. Moreover, as for the 'recall' of prediction in testing set, the figure shows its behavior is almost the same as AdaBoost, while GBDT predicts over 600 more testing samples correctly, which can be seen in the lower left corner of Figure 7. All in all, up to now, GBDT is actually the best algorithm of the three.

4.1.4 eXtreme Boost

Different from above experiments, here we will search for the hyper-paremeters more carefully. The main steps can be concluded as follows:

1. We want to determine an appropriate `'n_estimators'` by searching in the interval `[1,500]`. To achieve this, we fix other hyper-parameters as `'learning_rate'=0.2`, `'max_depth'=5`, `'min_child _weight'=1` and `'scale_pos_weight'=1`. In this step, we can also obtain the importance of each feature, which is attached to Appendix. Here we obtain optimal `'n_estimators'=384`.

2. Then we determine `'min_child_weight'` and `'max_depth'` by grid search method. The searching space is, `'min_child_weight': [1, 2, 3]` and `'max_depth': [6, 10]`. The results of grid searching is in Table 6:

`'max_depth'`	`'min_child_weight'`	Cross-Validation Recall
6	1	0.515918
6	2	0.521652
6	3	0.516744
10	1	0.525001
10	2	0.528717
10	3	0.526148

Table 6: results of grid search for XGBoost

In Table 6, we obtain the best Cross-Validation Recall of 0.52871 for `'max_depth'` of 10 and `'min_child_weight'` of 2. Therefore, we set hyper-parameters as `'learning_rate'=0.2`, `'max_depth'=10`, `'min_child _weight'=2`, `'n_estimators'=350` and `'scale_pos_weight'=1` and apply this model to the testing set. Then we have the confusion matrix as Figure 8 shows:

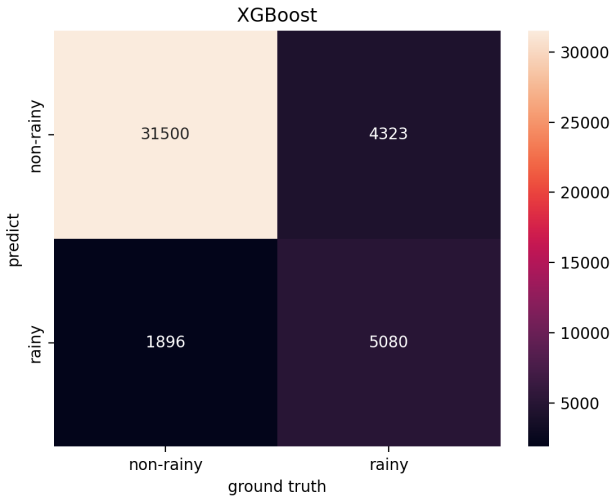


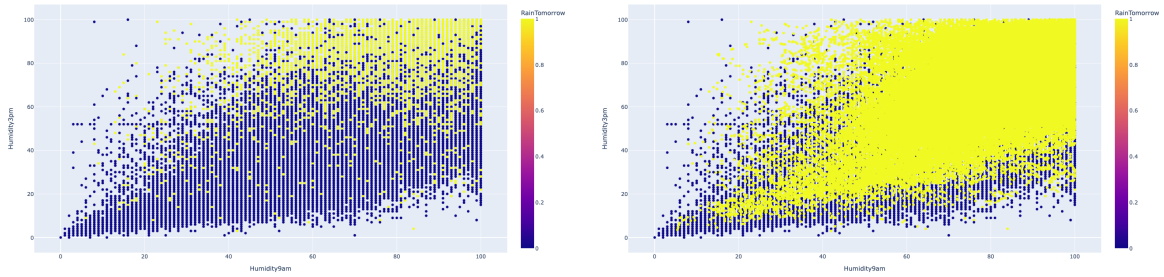
Figure 8: XGBoost Confusion Matrix

Compared with GBDT Algorithm, which correctly classifies 4948 rainy days right, XGBoost is able to predict over 100 more rainy days correctly without any decrease in testing accuracy (testing accuracy of XGBoost is **0.8547**, which is even higher). However, note that compared with the great leap from Logistic Regression to AdaBoost and GBDT, the improvement here is not so tremendous. This may because the imbalance of dataset, which we are going to discuss.

4.2 Oversampling

Here, we only apply the oversampling strategy to XGBoost since it is the greatest one among the four algorithms when we consider the primal dataset. The main motivation is not to find the optimal hyper-parameters. Instead, we hope to see the improvements of our results as we apply the oversampling strategy. Therefore, we choose the hyper-parameters which is the same as the previous one and see what will happen if we apply oversampling.

Before discussing the experiments, we visualize the oversampling dataset in Figure 8, compared with the non-oversampling one:



(a) Primal Dataset

(b) SMOTE Dataset

Figure 9: Non-oversampling vs Oversampling

In Figure 9, it is straightforward that in primal dataset, the majority is bule-colored points. Therefore, our algorithm may not catch the information in yellow-colored points. To see whether algorithm can show something different, we use oversampled training samples to train the XGBoost model. Then we use testing set to check its generalization. The results of confusion matrix is in Figure 10:

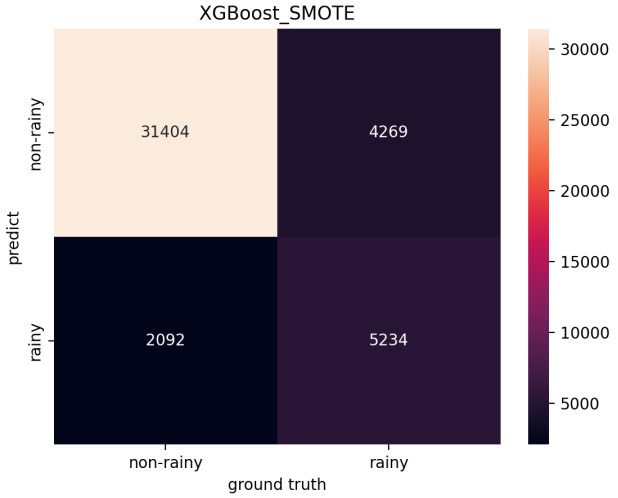


Figure 10: SMOTE XGBoost Confusion Matrix

In Figure 10, we find that although the testing accuracy of the new model just decreases slightly, its testing recall starts to increase again. This shows that, with SMOTE dateset coming in, we are able to improve the recall in a reasonable way. That is to say, we will receive the gain in recall rate, without losing too much accuracy (acutally the accuracy is almost the same after using SMOTE dataset). Thus, as for our problem, this is indeed an improvement of our XGBoost Model, which gives us more confidence in predicting rainy days. At the same time, we can still believe our prediction in general, since the testing accuracy is almost the same.

5 Conclusion

5.1 Non-oversampling

1. There is no ambiguity that XGBoost gives the best result. This is not surprising actually, since its technique behind the algorithm is the most advanced one. Furthermore, while doing the experiments, it shows that XGBoost will be much faster than any other algorithms.

2. From the results of our experiments, it shows that from Logistic Regression to AdaBoost and GBDT, great improvement has been made. However, from GBDT to XGBoost, it seems that the improvement is not so big. One possible reason is that, from Logistic Regression to AdaBoost, actually they are two different types of classifiers. The former one is a simple probablistic discriminative classifier, and the latter one is a Tree-Based Model. The difference between these two classifiers can be attributed to the diiference between **parametric model** and **non-parametric model**. As for parametric models, if we capture the key of the problem, then the parametric models will behave well. However, if the problem is complicated, like this dataset, the parametric models will lose and non-parametric models will win. Our experiment confirms the advantages of non-parametric models in some sense.

5.2 Oversampling

Techniques like oversampling are not always necessary. That is, we should choose such methods according to our aims and requirements. In our dataset, oversampling is meaningful since the minority-class data is extremely important to us. That is, we prefer to predict rainy days correctly more than non-rainy days. Therefore, using oversampling method can help us learn the dataset sufficiently. Just as we show in Figure 10 in Section 4.2, with the help of oversampling, we are able to improve the recall with tolerable changes in

accuracy.

Finally, we can conclude that our aim of predicting rainy days with the help of machine learning is obtained in success, with 85% accuracy and 60% recall rate approximately. Our recall rate, while it still has room for improvement, is better than that from purely guessing.

Note that this problem is non-trivial, since we lack non-rainy days information. There are some processes in this report which can be modified. For example, when we deal with the missing data, we actually have some cleverer ways which take the time structure into consideration. With this kind of fill-in strategy, the algorithms that we apply in the training process can be more powerful and give better results (actually the effect of the algorithm is determined by the quality of our dataset).

Finally the last, it is worth mentioning that XGBoost is the most powerful algorithm in our report.

6 Reference

References

[1] Yuichiro Anzai. *Pattern recognition and machine learning*. Elsevier, 2012.

[2] Peter Bartlett, Yoav Freund, Wee Sun Lee, and Robert E Schapire. Boosting the margin: A new explanation for the effectiveness of voting methods. *The annals of statistics*, 26(5):1651–1686, 1998.

[3] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[4] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, et al. Xgboost: extreme gradient boosting. *R package version 0.4-2*, 1(4):1–4, 2015.

[5] Alexey Natekin and Alois Knoll. Gradient boosting machines, a tutorial. *Frontiers in neurorobotics*, 7:21, 2013.

7 Appendix

7.1 Pre-processing

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from imblearn.over_sampling import SMOTE
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import MinMaxScaler
import plotly.express as px
import plotly.io as io
import missingno as msno
import seaborn as sns
from plotly import tools
# load the dataset
raw_df = pd.read_csv('/Users/wangjiangyi/Desktop/weatherAUS.csv')
raw_df.dropna(subset=['RainToday', 'RainTomorrow'], inplace=True)

#draw the label imblance diagram
#raw_df1=raw_df.fillna('None')
#fig=px.histogram(raw_df1,
#             x='Location',
#             color='RainToday',
#             title='Visualization of Label Imbalance',)
#io.write_image(fig, '/Users/wangjiangyi/Desktop/imb.png')

# setting target column
y = raw_df.RainTomorrow
# fill NA
X = raw_df.drop(['RainTomorrow'], axis=1)

# split the time information
year = pd.to_datetime(raw_df.Date).dt.year

#plot the datasets
#df1=pd.read_csv('/Users/wangjiangyi/Desktop/weatherAUS.csv', index_col=['Date'],
                                                parse_dates=True)
#sns.countplot(y=df1.index.year)
#plt.show()

#visulize the missing value
#msno.bar(raw_df, labels=True, figsize=(24,7),fontsize=8)
#plt.show()

#visualize some interesting property
#fig=px.histogram(X.fillna('None'), x='Location', title='Location vs. Rainy Days', color='
                                                RainToday')

#fig.show()

# split the dataset to three part

train_inputs = X[year < 2015]
val_inputs = X[year == 2015]
test_inputs = X[year > 2015]

train_targets = y[year < 2015]
val_targets = y[year == 2015]
test_targets = y[year > 2015]
print(y)
#split the dataset according to the data type for further fill NA
numeric_cols = X.select_dtypes(include = 'float64').columns.to_list()
cat_cols = X.select_dtypes(include = 'object').columns.to_list()[1:]

# fill NA for numerical type
imputer = SimpleImputer (strategy = 'median')
imputer.fit(X[numeric_cols])
'''
```

```
train_inputs=train_inputs.copy()
train_inputs.loc[:, numeric_cols]=imputer.transform(train_inputs[numeric_cols])
val_inputs=val_inputs.copy()
val_inputs[numeric_cols] = imputer.transform(val_inputs[numeric_cols])
test_inputs=test_inputs.copy()
test_inputs[numeric_cols] = imputer.transform(test_inputs[numeric_cols])
'''
#plt.figure(figsize=(15,13))
X[numeric_cols]=X[numeric_cols].copy()
X.loc[:, numeric_cols]=imputer.transform(X[numeric_cols])
#sns.heatmap(X[numeric_cols].corr(), annot=True)

#plt.show()

#as for categorical feature
# fill NA
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
X2 = X[cat_cols].fillna(method='ffill')
encoder.fit(X2)
#print(X['RainToday'].isna().sum())

encoded_cols = list(encoder.get_feature_names_out(cat_cols))
#print(len(encoded_cols))
'''
train_inputs=train_inputs.copy()
#onehot
train_inputs.loc[:,encoded_cols] = encoder.transform(train_inputs[cat_cols].fillna(method='
                                        ffill'))
#print(train_inputs.columns)
val_inputs=val_inputs.copy()
val_inputs.loc[:,encoded_cols] = encoder.transform(val_inputs[cat_cols].fillna(method='
                                        ffill'))
test_inputs=test_inputs.copy()
test_inputs.loc[:,encoded_cols] = encoder.transform(test_inputs[cat_cols].fillna(method='
                                        ffill'))
'''
X=X.copy()
X.loc[:, encoded_cols]=encoder.transform(X[cat_cols].fillna(method='ffill'))

X=X.copy()
X.loc[:,'RainTomorrow']=y
fig1=px.scatter(X, x='Humidity9am', y='Humidity3pm',
color='RainTomorrow')

#X.drop(cat_cols+['Temp9am','Temp3pm'],axis=1,inplace=True)

sm=SMOTE(sampling_strategy={0:109586,1:109586})  # should be open
#y_train.index=range(0,97988)
#X_train.index=range(0,97988)
X_sm, y_sm = sm.fit_resample(X[numeric_cols], y) # should be open
X_sm = X_sm.copy()
X_sm.loc[:,'RainTomorrow']=y_sm

fig2=px.scatter(X_sm, x='Humidity9am', y='Humidity3pm',color='RainTomorrow')

fig=tools.make_subplots(rows=2,cols=1)
fig.append_trace()

numeric_cols.pop()
numeric_cols.pop()

'''
#scaler
scaler = MinMaxScaler()
scaler.fit(X[numeric_cols])
X[numeric_cols] = scaler.transform(X[numeric_cols])

X.to_csv('/Users/wangjiangyi/Desktop/pre-processed.csv',index=False,sep=',')
'''
```

7.2 Model Training

```
import time
import pandas as pd
from sklearn import  metrics
import numpy as np
from sklearn.metrics import confusion_matrix
from imblearn.over_sampling import SMOTE
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import zero_one_loss
import xgboost as xgb
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier
import seaborn as sn
import matplotlib.pyplot as plt
from xgboost.sklearn import XGBClassifier
from sklearn.model_selection import GridSearchCV
import warnings


def tun_parameters(train_x, train_y):
xgb1 = XGBClassifier(learning_rate=0.2, n_estimators=350, max_depth=5, min_child_weight=1,
                                                gamma=0, subsample=0.8,
colsample_bytree=0.8, objective='binary:logistic', scale_pos_weight=1, seed=27)
modelfit(xgb1, train_x, train_y)


def modelfit(alg, X, y, useTrainCV=True, cv_folds=5, early_stopping_rounds=50):
if useTrainCV:
xgb_param = alg.get_xgb_params()
xgtrain = xgb.DMatrix(X, label=y)
cvresult = xgb.cv(xgb_param, xgtrain, num_boost_round=alg.get_params()['n_estimators'],
                                                nfold=cv_folds,
metrics='auc', early_stopping_rounds=early_stopping_rounds)
alg.set_params(n_estimators=cvresult.shape[0])

# Fit the algorithm on the data
alg.fit(X, y, eval_metric='auc')

# Predict training set:
dtrain_predictions = alg.predict(X)
dtrain_predprob = alg.predict_proba(X)[:, 1]

# Print model report:
print("\nModel Report")
print("Accuracy : %.4g" % metrics.accuracy_score(y, dtrain_predictions))
print("AUC Score (Train): %f" % metrics.roc_auc_score(y, dtrain_predprob))

feat_imp = pd.Series(alg.get_booster().get_fscore()).sort_values(ascending=False)
feat_imp.plot(kind='bar', title='Feature Importances')
plt.ylabel('Feature Importance Score')
plt.show()
print('n_estimators=', cvresult.shape[0])



def plot_matrix(cm,title_name):
ax = sn.heatmap(cm,annot=True,fmt='g',xticklabels=['non-rainy', 'rainy'],yticklabels=['non-
                                                rainy', 'rainy'])
ax.set_title(title_name)
ax.set_xlabel('ground truth')
ax.set_ylabel('predict')
#warnings.filterwarnings("ignore")  # ignore warning

# load the pre-processed dataset
raw_df= pd.read_csv('/Users/wangjiangyi/Desktop/pre-processed.csv')
temp_y= pd.read_csv('/Users/wangjiangyi/Desktop/weatherAUS.csv')
temp_y.dropna(subset=['RainToday', 'RainTomorrow'], inplace=True)
```

```
y = temp_y.RainTomorrow
X = raw_df

# there are 109586 0-class, 31201 1-class in total

# split for training and testing sets
year1 = pd.to_datetime(X.Date).dt.year
year2 = pd.to_datetime(temp_y.Date).dt.year
X.drop(['Date'],axis=1,inplace=True)

X_train = X[year1 < 2015]
X_test = X[year1 >= 2015]

y_train = y[year2 < 2015]
y_test = y[year2 >= 2015]

#there are 76190 0-class, 21798 1-class in training set



#-------------------------------------------------------------------------------------
#-------------------------------------------------------------------------------------
#-------------------------------------------------------------------------------------
#-------------------------------------------------------------------------------------
# Model training part
# training set: X_train, y_train <=======> X_train_sm, y_train_sm
# testing set: X_test, y_test

lr = LogisticRegression(C=.2, max_iter=5000) # C=.2 vs C=100
ada = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=9,
                                            min_samples_split=300, min_samples_leaf=3),
                                            learning_rate=0.7, n_estimators=400)
gbdt = GradientBoostingClassifier(learning_rate=0.35, loss= 'deviance', max_depth=10,
                                            min_samples_leaf=10, min_samples_split= 300,
                                            n_estimators= 30, subsample= 0.9)
xgbst = XGBClassifier(learning_rate =0.2, n_estimators=350, max_depth=10,
min_child_weight=2, gamma=0, subsample=0.8,colsample_bytree=0.8,
objective= 'binary:logistic', nthread=8,scale_pos_weight=1, seed=21)


'''
# Logistic Regression
models = {
  'LogisticRegression': {
    'model': LogisticRegression(max_iter=5000),
    'params': {
      'C': [.2, .5, 1, 2, 5, 10, 15, 20, 50, 100]
    }
  }
}
scores = []

for model_name, mp in models.items():
gridsearch = GridSearchCV(mp['model'], mp['params'], cv=5, return_train_score=False,
                                            scoring = 'accuracy')
grid_result = gridsearch.fit(X_train, y_train)
scores.append({
  'model': model_name,
  'best_score': gridsearch.best_score_,
  'best_params': gridsearch.best_params_
})

means = grid_result.cv_results_['mean_test_score']
params = grid_result.cv_results_['params']
for mean, param in zip(means, params):
print("%f  with:   %r" % (mean, param))

print(scores)
'''
'''
```

```
lr.fit(X_train, y_train)
lr_pred=lr.predict(X_test)
print(confusion_matrix(lr_pred, y_test))
print(lr.score(X_test, y_test))
'''

#-------------------------------------------------------------------------------------
#-------------------------------------------------------------------------------------
# Adaboost
'''
models = {
  'AdaBoostClassifier': {
    'model': AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=6,
                                                min_samples_split=400, min_samples_leaf=4)),
    'params': {
      "learning_rate": [0.3, 1],
      "n_estimators": [50,150,400],
    }
}}
scores_gradboost = []

for model_name, mp in models.items():
gridsearch = GridSearchCV(mp['model'], mp['params'], cv=5, return_train_score=False,
                                                scoring='recall')
grid_result=gridsearch.fit(X_train, y_train)
scores_gradboost.append({
  'model': model_name,
  'best_score': grid_result.best_score_,
  'best_params': grid_result.best_params_
})
means = grid_result.cv_results_['mean_test_score']
params = grid_result.cv_results_['params']
std = grid_result.cv_results_['std_test_score']
for std, mean, param in zip(std, means, params):
print("%f  %f  with:   %r" % (std, mean, param))

print(scores_gradboost)


ada.fit(X_train, y_train)
ada_pred=ada.predict(X_test)
print(confusion_matrix(ada_pred, y_test))
print(ada.score(X_test, y_test))

'''
#-------------------------------------------------------------------------------------
#-------------------------------------------------------------------------------------
# GBDT
'''
models = {
  'GradientBoostingClassifier': {
    'model': GradientBoostingClassifier(),
    'params': {
      "loss": ["deviance"],
      "learning_rate": [0.35],
      "min_samples_split": [150, 300],
      "min_samples_leaf": [5, 10],
      "max_depth": [4, 8, 10],
      "subsample": [.9],
      "n_estimators": [30]
    }
}}
scores_gradboost = []

for model_name, mp in models.items():
grid_gbdt = GridSearchCV(mp['model'], mp['params'], cv=5, return_train_score=False, scoring
                                                ='recall')
grid_gbdt.fit(X_train, y_train)
scores_gradboost.append({
  'model': model_name,
  'best_score': grid_gbdt.best_score_,
```

```
  'best_params': grid_gbdt.best_params_
})
means = grid_gbdt.cv_results_['mean_test_score']
params = grid_gbdt.cv_results_['params']
std = grid_gbdt.cv_results_['std_test_score']
for std, mean, param in zip(std, means, params):
print("%f  %f  with:   %r" % (std, mean, param))
print(scores_gradboost)
'''
'''
gbdt.fit(X_train, y_train)
gbdt_pred=gbdt.predict(X_test)
print(confusion_matrix(gbdt_pred, y_test))
print(gbdt.score(X_test, y_test))
'''

#------------------------------------------------------------------------------
#------------------------------------------------------------------------------
# XGBoost
'''
models = {
  'XGBClassifier': {
    'model': XGBClassifier(objective= 'binary:logistic'),
    'params': {
      'max_depth': [5],
      "min_child_weight": [1],
      "subsample": [.8],
      'n_estimators': [50],
      'scale_pos_weight' : [1],
      'leanning_rate' : [1000]
    }
}}
scores_xgb = []

for model_name, mp in models.items():
grid_xgb = GridSearchCV(mp['model'], mp['params'], cv=5, return_train_score=False, scoring
                                                ='recall')
grid_xgb.fit(X_train, y_train)
scores_xgb.append({
  'model': model_name,
  'best_score': grid_xgb.best_score_,
  'best_params': grid_xgb.best_params_
})
means = grid_xgb.cv_results_['mean_test_score']
params = grid_xgb.cv_results_['params']
std = grid_xgb.cv_results_['std_test_score']
for std, mean, param in zip(std, means, params):
print("%f  %f  with:   %r" % (std, mean, param))
print(scores_xgb)
'''
'''
xgbst.fit(X_train, y_train)
xgb_pred=xgbst.predict(X_test)
print(confusion_matrix(xgb_pred, y_test))
print(xgbst.score(X_test, y_test))
'''
# step 1
#tun_parameters(X_train, y_train)
# step 2
'''
param_test1 = {
  'max_depth':[6,10],
  'min_child_weight': [1,2,3]
}
gsearch1 = GridSearchCV(estimator = XGBClassifier(learning_rate =0.2, n_estimators=350,
                                                  max_depth=5,
min_child_weight=1, gamma=0, subsample=0.8,colsample_bytree=0.8,
objective= 'binary:logistic', nthread=8,scale_pos_weight=1, seed=27),
param_grid = param_test1,scoring='recall',n_jobs=-1, cv=5 )
gsearch1.fit(X_train,y_train)
print(gsearch1.best_score_, gsearch1.best_params_,     gsearch1.best_score_)
```

```
means = gsearch1.cv_results_['mean_test_score']
params = gsearch1.cv_results_['params']
std = gsearch1.cv_results_['std_test_score']
for std, mean, param in zip(std, means, params):
print("%f  %f  with:   %r" % (std, mean, param))
'''
#-------------------------------------------------------------------------------
#-------------------------------------------------------------------------------

sm=SMOTE(sampling_strategy={0:76190,1:76190})  # should be open
#y_train.index=range(0,97988)
#X_train.index=range(0,97988)
X_train_sm, y_train_sm = sm.fit_resample(X_train, y_train) # should be open

xgbst.fit(X_train_sm, y_train_sm)
xgb_pred=xgbst.predict(X_test)
print(confusion_matrix(xgb_pred, y_test))
print(xgbst.score(X_test, y_test))

plot_matrix(xgbst.score(X_test, y_test),"XGBoost_SMOTE")
plt.show()
```

7.3 Results of grid search

Take the results of XGBoost as an example, and full results will be attached by '.txt'.

```
First part:
Simple grid search
0.046042  0.503210  with:   {'learning_rate': 0.3, 'max_depth': 10, 'n_estimators': 3}
0.050264  0.510091  with:   {'learning_rate': 0.3, 'max_depth': 10, 'n_estimators': 5}
0.060062  0.508028  with:   {'learning_rate': 0.3, 'max_depth': 10, 'n_estimators': 10}
0.072637  0.502981  with:   {'learning_rate': 0.3, 'max_depth': 10, 'n_estimators': 15}
0.081772  0.503348  with:   {'learning_rate': 0.3, 'max_depth': 10, 'n_estimators': 20}
0.085129  0.504311  with:   {'learning_rate': 0.3, 'max_depth': 10, 'n_estimators': 25}
0.088215  0.505045  with:   {'learning_rate': 0.3, 'max_depth': 10, 'n_estimators': 30}
[{'model': 'XGBClassifier', 'best_score': 0.5100912064251755, 'best_params': {'
                                        learning_rate': 0.3, 'max_depth': 10, '
                                        n_estimators': 5}}]

Process finished with exit code 0

0.047295  0.501742  with:   {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 3}
0.050771  0.505642  with:   {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 5}
0.056831  0.509862  with:   {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 10}
0.060532  0.507935  with:   {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 15}
0.066616  0.505413  with:   {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 20}
0.078791  0.504816  with:   {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 25}
0.084003  0.505274  with:   {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 30}
0.088960  0.507292  with:   {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 50}
0.053604  0.505918  with:   {'learning_rate': 0.4, 'max_depth': 10, 'n_estimators': 3}
0.054667  0.509312  with:   {'learning_rate': 0.4, 'max_depth': 10, 'n_estimators': 5}
0.069830  0.504999  with:   {'learning_rate': 0.4, 'max_depth': 10, 'n_estimators': 10}
0.082632  0.504861  with:   {'learning_rate': 0.4, 'max_depth': 10, 'n_estimators': 15}
0.088842  0.507751  with:   {'learning_rate': 0.4, 'max_depth': 10, 'n_estimators': 20}
0.087518  0.507889  with:   {'learning_rate': 0.4, 'max_depth': 10, 'n_estimators': 25}
0.088198  0.509999  with:   {'learning_rate': 0.4, 'max_depth': 10, 'n_estimators': 30}
0.085173  0.517201  with:   {'learning_rate': 0.4, 'max_depth': 10, 'n_estimators': 50}
0.050871  0.508991  with:   {'learning_rate': 0.5, 'max_depth': 10, 'n_estimators': 3}
0.056514  0.506881  with:   {'learning_rate': 0.5, 'max_depth': 10, 'n_estimators': 5}
0.074822  0.505780  with:   {'learning_rate': 0.5, 'max_depth': 10, 'n_estimators': 10}
0.085896  0.507385  with:   {'learning_rate': 0.5, 'max_depth': 10, 'n_estimators': 15}
0.088049  0.510459  with:   {'learning_rate': 0.5, 'max_depth': 10, 'n_estimators': 20}
0.085297  0.509357  with:   {'learning_rate': 0.5, 'max_depth': 10, 'n_estimators': 25}
0.084550  0.512339  with:   {'learning_rate': 0.5, 'max_depth': 10, 'n_estimators': 30}
0.083474  0.513165  with:   {'learning_rate': 0.5, 'max_depth': 10, 'n_estimators': 50}
[{'model': 'XGBClassifier', 'best_score': 0.517201014036129, 'best_params': {'learning_rate
                                        ': 0.4, 'max_depth': 10, 'n_estimators': 50}}]

Process finished with exit code 0
0.086376  0.515366  with:   {'learning_rate': 0.4, 'max_depth': 10, 'n_estimators': 40}
0.085173  0.517201  with:   {'learning_rate': 0.4, 'max_depth': 10, 'n_estimators': 50}
0.083854  0.518990  with:   {'learning_rate': 0.4, 'max_depth': 10, 'n_estimators': 60}
[{'model': 'XGBClassifier', 'best_score': 0.51899048893884, 'best_params': {'learning_rate'
                                        : 0.4, 'max_depth': 10, 'n_estimators': 60}}]

--------------------------------------------------------------------------------
Second part:
1. Fix other hyper-parameters, pick the optimal n_estimators
Model Report
Accuracy : 0.902
AUC Score (Train): 0.948307
n_estimators= 384
2. Choose the optimal max_depth and min_child_weight
{'max_depth': 10, 'min_child_weight': 2} 0.5287173011232692
```

7.4 Feature Importance

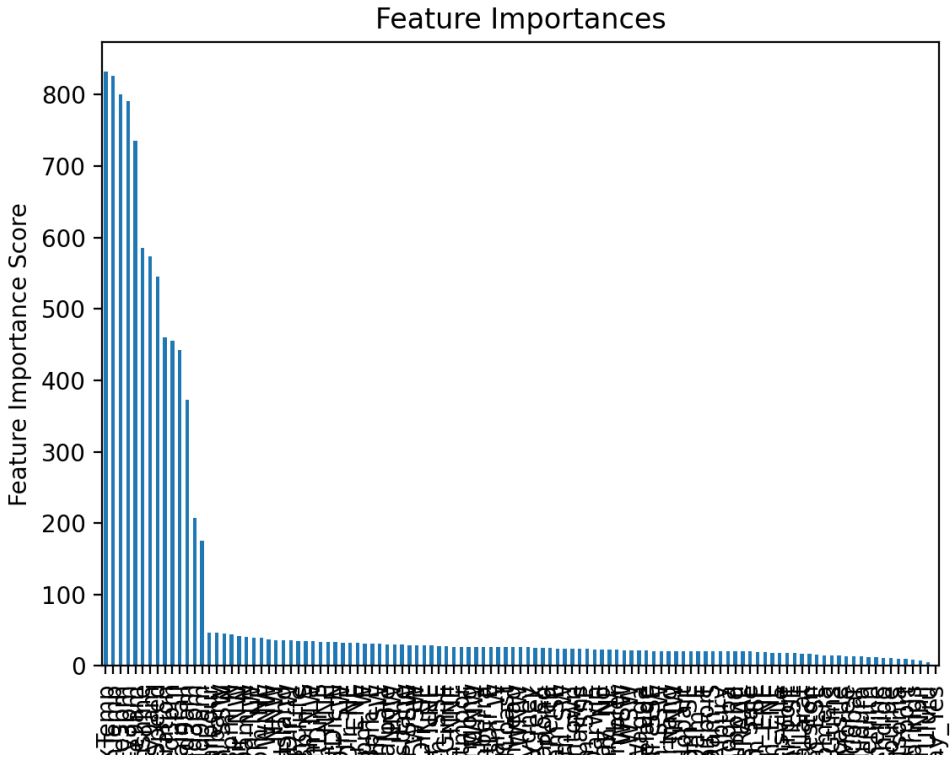


Figure 11: Feature Importance