Machine Learning for Predicting Flight Ticket Prices

Supervisor: Boi Faltings, Igor Kulev
Student: Jun Lu
School of Computer and Communication Sciences, EPFL
1005, Lausanne, Switzerland
Email: jun.lu@epfl.ch

***Abstract*—Airline companies use many different variables to determine the flight ticket prices: indicator whether the travel is during the holidays, the number of free seats in the plane etc. Some of the variables are observed, but some of them are hidden.**

This report gives the results of how machine learning methods can be applied in the domain of airplane ticket prediction. And based on the data over a 103 day period, we trained our models, getting the model which is AdaBoost-Decision Tree Classification. This algorithm has best performance over the observed 8 routes which has 61.35% better performance than the random purchase strategy, and relatively small variance over these routes.

And we also considered the situation that we cannot get too much historical datas for some routes(for example the route is new and does not have historical data) or we do not want to train historical data to predict to buy or wait quickly, in which problem, we used HMM Sequence Classification based AdaBoost-Decision Tree Classification to perform our prediction on 12 new routes. Finally, we got 31.71% better performance than the random purchase strategy.

I. Introduction

For purchasing a airplane ticket, the traditional purchase strategy is that it is generally best to buy a ticket far in advance of the flight's departure date to avoid the risk that the price may increase. However, this is usually not always this case, airplane companies can decrease the prices if they want to increase the sales. Airline companies use many different variables to determine the flight ticket prices: indicator whether the travel is during the holidays, the number of free seats in the plane etc, or even in which month it is. Some of the variables are observed, but some of them are hidden. In this context, buyers are trying to find the right day to buy the ticket, and on the contrary, the airplane companies are trying to keep the overall revenue as high as possible. The goal of this project is to use machine learning techniques to model the behavior of flight ticket prices over the time.

Airline companies have the freedom to change the flight ticket prices at any moment. Travellers can save money if they choose to buy a ticket when its price is the lowest. The problem is how to determine when is the best time to buy flight ticket for the desired destination and period. In other word, when given the historical price and the current price of a flight for a specific departure date, our algorithms need to determine whether it is suitable to buy or wait. The goal of this project is to use machine learning techniques to model the behavior of flight ticket prices over the time. In order to build and evaluate the model, we used data that contains historical flight ticket prices for particular routes.

June 09, 2016

II. Related work

Some work has been done for determining optimal purchase timing for airline tickets. Our work is especially inspired by [Etzioni et al., 2003]. Described in the paper, it achieves 61.8% of optimal. This result is very close to our result. However, our project goes beyond their work in several ways: the observation period is over a 103 day period(instead of a 41 day period); we extracted 8 routes for the prediction(rather than 2 routes in the existing work). This is a more difficult problem because over a longer period, the airplane companies tend to vary the price algorithm behind the company and they may have different price strategies for different routes.

Moreover, our novelty is that we extended the problem into regression and classification problems by some **model constructions**. Finally, given the historical datas and current data of the ticket, our system can predict to buy or to wait. In the end, we also implemented the Q-Learning method mentioned in [Etzioni et al., 2003] to compare the results.

Another novelty is that, we also considered such a situation that some routes do not have any historical datas, in which case we cannot perform the learning algorithms at all. This situation is very common, because there are always some new routes to be added by the airplane company or the company may conceal the historical datas for some reasons. And also, this model can reduce computation time when we want to predict to buy or wait quickly because we do not need to train a large amount of data. We call this problem as **generalized problem**. And on the contrary, for the previous problem, we call it as **specific problem**.

III. Problem Statement

As described above, we have two problems to solve. One is the **specific problem**, and the other one is the **generalized problem**. The Fig 1 gives an overall work flow of what we should do in the two problems. We also defined the terms of **specific routes**, which were given the historical records to train, and **generalized routes**, which were not given the historical records.

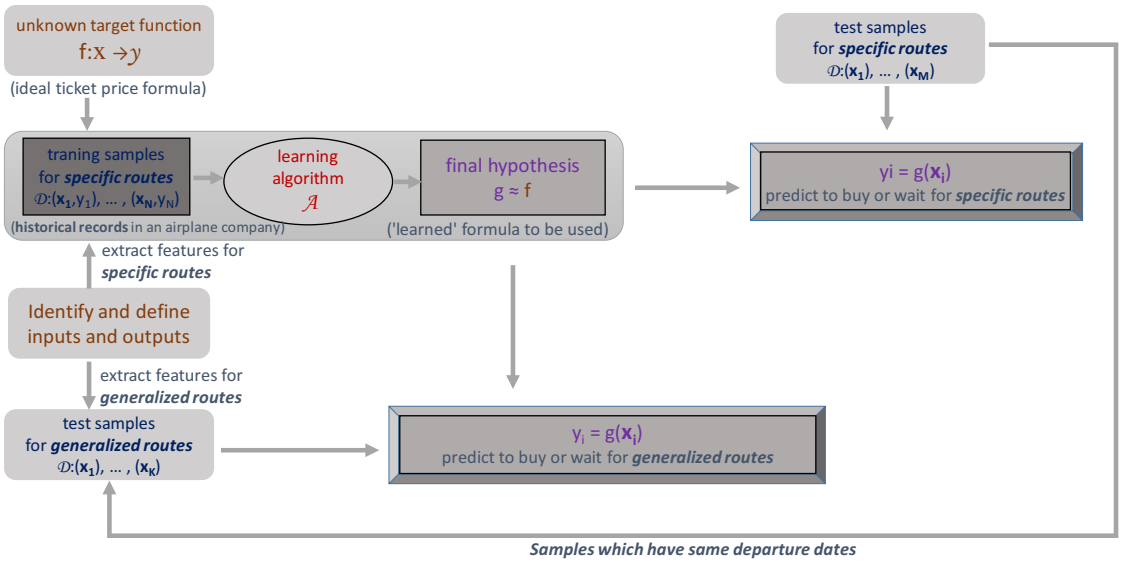

Fig. 1. Work flow for the specific problem and generalized problem.

In the specific problem, given the historical price data for some specific routes, we are forged to learn the formula from these historical datas and predict whether to buy or wait for the following days. The inputs are features extracted from the ticket datas, and although the outputs for regression or classification are different, in the sense of the system design, the outputs are to buy or to wait for each queried date for a specific route.

In the generalized problem, we were not given any historical data for these routes. But we were given the formula learned from specific problem and corresponding datas which have same departure data in specific problems(which will be described more clearly in the section of Data Description and Interpretation). So we need to extract input features based on the test samples of generalized routes and test samples of specific routes. This model may have many benefits, especially when the historical datas are not given or when we want to save time to quickly determine whether we should buy or wait for the new routes and not to spend too much time on model building for the new routes, and some other benefits such as decreasing data storage and so on.

IV. Data Collection

The data for our analysis was collected as daily price quotes from a major airplane search web site between Nov. 9, 2015 and Feb. 20, 2016 (103 observation days). A web crawler was used to query for each route and departure date pair, and the crawling was done every day at 10:00 AM.

For the purpose of our pilot study, we restricted the collecting data on non-stop, single-trip flights for 8 routes: 1 for Barcelona, Spain (BCN) to Budapest, Hungary (BUD); 2 for Budapest, Hungary to Barcelona, Spain; 3 for Brussels, Belgium (CRL) to Bucharest, Romania (OTP); 4 for Mulhouse, France (MLH) to Skopje, Macedonia (SKP); 5 for Malmo, Sweden (MMX) to Skopje, Macedonia; 6 for Bucharest, Romania to Brussels, Belgium; 7 for Skopje, Macedonia to Mulhouse, France; 8 for Skopje, Macedonia to Malmo, Sweden. Overall, we collected 36,575 observations(i.e. the queried price of each day for different departure dates and for the 8 different routes). In our observed airplane website, we did not find any tickets that were sold out in the specific queried days.

For the generalized problem, we also collected another 12 routes(i.e. BGY→OTP, BUD→VKO, CRL→OTP, CRL→WAW, LTN→OTP, LTN→PRG, OTP→BGY, OTP→CRL, OTP→LTN, OTP→LTN, PRG→LTN, VKO→BUD, WAW→CRL), which contains 14,160 observations. And you can notice that two routes have already been observed in the specific problem, which are CRL→OTP and OTP→CRL. We keep these two routes to see how the generalized model can influence the performance. We will compare the results of the specific problem and generalized problem for these two routes especially.

A. Pricing Behavior in the Collected Data

We found that the ticket price for flights can vary significantly over time. Table I shows the minimum price, maximum price, and the maximum different in prices that can occur for flights for the 8 specific routes. In this table, although it's the maximum price and minimum price for all the departure dates for each route, we can have an overall glance at how we can achieve to decrease the ticket purchasing price.

Fig. 2, 3 and 4 show how pricing strategies differ from flights.

Route	*Min Price*	*Max Price*	*Max Price Change*
BCN→BUD	29.99 €	279.99 €	250.0 €
BUD→BCN	28.768 €	335.968 €	307.2 €
CRL→OTP	9.99 €	239.99 €	230.0 €
MLH→SKP	14.99 €	259.99 €	245.0 €
MMX→SKP	15.48 €	265.08 €	249.6 €
OTP→CRL	9.75 €	269.75 €	260.0 €
SKP→MLH	16.182 €	332.982 €	316.8 €
SKP→MMX	16.182 €	332.982 €	316.8 €

TABLE I

Minimum price, maximum price, and maximum change in ticket price for 8 specific routes. The decimals of the prices are from the currency change.

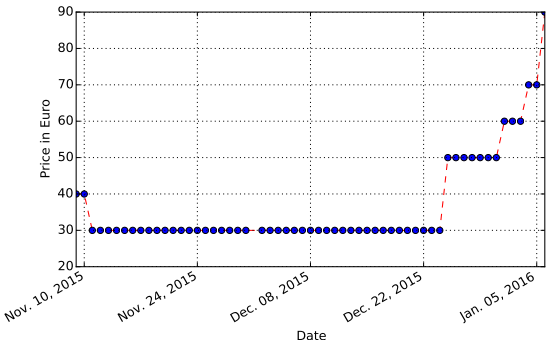

Fig. 2. Price change over time for flight BCN→BUD, departing on Jan. 6 2016. This figure shows an example of price rises continuously before departure date and low price fluctuation.

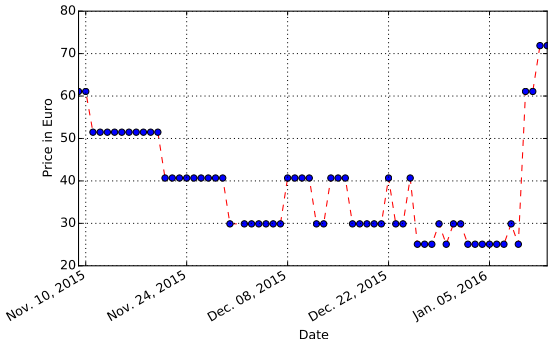

Fig. 3. Price change over time for flight MLH→SKP, departing on Jan. 13 2016. This figure shows an example of price drops to the minimum a slightly before departure date and have more price fluctuation.

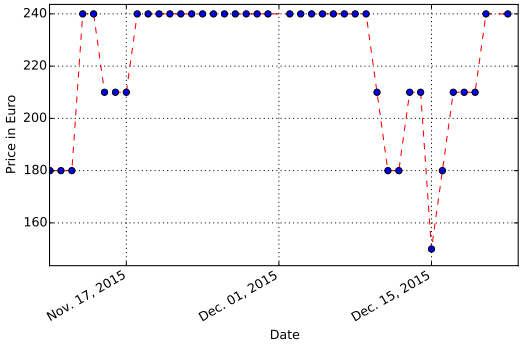

Fig. 4. Price change over time for flight CRL→OTP, departing on Dec. 22 2015. This figure shows an example of price drops to the minimum a slightly before departure date which may benefit the consumers and the high prices dominate.

V. Machine Learning Approach

A. Feature Extraction

The features extracted for training and testing are aggregated variables computed from the list of quotes observed on individual query days. For each query day, there are possibly eight airlines quoting flights for a specific origin-destination and departure date combination. For each query data, 5 features are computed, the *flight number*(encoded by dummy variables), the *minimum price so far*, the *maximum price so far*, the *query-to-departure* (number of days between the first query date(09.11.2015 in our case) and departure date), the *days-to-departure* (number of days between the query and departure date), and *current price*. Specifically, the inputs have the following format shown in Fig 5 (of course, the flight number should be encoded with dummy variables).

For the output for regression problem, we set it to be the minimum price for each departure date and each flight; and as for the output for classification problem, we set the data entry of which the price is the minimum price from the query date to the departure date to 1(namely Class 1 - to buy), otherwise, we set it to be 0(namely Class 2 - to wait).

B. Data Description and Interpretation

Our data set consists of one set of input feature vectors, each one of them will be split into training dataset and testing dataset. In our case, we split the datas that correspond to flights with departure date in the interval between Nov. 9, 2015 and Jan. 15, 2016 as the training dataset; and the datas that correspond to flights with departure date in the interval betweeen Jan. 16, 2016 and Feb. 20, 2016 as the testing dataset; as for the generalized problem, we got the same time period as the test dataset, which is from Jan. 16, 2016 and Feb. 20, 2016.

Finally, the training dataset consists of $\mathbf{N_{tr}}$=16,208 data samples of one output variable $\mathbf{y}$ and input variable $\mathbf{X}$. The input variable $\mathbf{X}$ depends on the feature provided, as detailed shown above.

The testing dataset consists of $\mathbf{N_{te}}$=20,367, for which the output is unknown, and where we forged our predictions.

The generalized problem testing dataset consists of $\mathbf{N_{ge}}$=14,160, for which the output is unknown as well, and we need to predict.

C. Model Construction

For regression, as the output is the minimum price for each departure date and each flight. Our regression method is to predict the expected minimum price for a given departure date and given flight with the input features. As a result, if the *current price* is less than the *expected minimum price*, we predict to buy; otherwise, we predict to wait. However,

$$\begin{bmatrix} x_{1\,\text{flight-no}} & x_{1\,\text{minimum-price}} & x_{1\text{maximum-price}} & x_{1\text{query-to-departure}} & x_{1\text{days-to-departure}} & x_{1\text{current-price}} \\ x_{2\,\text{flight-no}} & x_{2\,\text{minimum-price}} & x_{2\text{maximum-price}} & x_{2\text{query-to-departure}} & x_{2\text{days-to-departure}} & x_{2\text{current-price}} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{N\,\text{flight-no}} & x_{N\,\text{minimum-price}} & x_{N\text{maximum-price}} & x_{N\text{query-to-departure}} & x_{N\text{days-to-departure}} & x_{N\text{current-price}} \end{bmatrix}$$

Fig. 5. Input Features

although it is very rare to happen, sometimes we may predict the *expected minimum price* of every entry to be smaller than the *current price*. Then, the last date should be seen as to buy. After we studied the data in depth, we were able to see that the last date always has a very high price. Thus we make the last buy date to be 7 days before departure date. The following buy rule explicitly explains how to predict to buy or wait in the model of regression:

Algorithm 1 Buy Rule for Regression

for *days_to_departure = N, N-1, ..., 7* **do**
  IF ExpectedMinimumPrice(this entry) $>$ current price
    BUY
    Return "Success"
  ELSE
    WAIT
end for
IF *days_to_departure* == 7
    BUY it
    Return "Success"

For classification, as our classification method is to predict to buy or to wait with the input features. As a result, if the prediction is to buy, we buy the ticket, and we only buy the earliest. The following buy rule explicitly explain how to predict to buy or wait in the model of classification:

Algorithm 2 Buy Rule for Classification

for *days_to_departure = N, N-1, ..., 7* **do**
  IF Output(this entry) $==$ 1
    BUY
    Return "Success"
  ELSE
    WAIT
end for
IF *days_to_departure* == 7
    BUY it
    Return "Success"

D. Methodology

1) **Least Squares for Regression:** It fits a linear model with coefficients $w = (w_1, ..., w_p)$ to minimize the residual sum of squares between the observed "true" responses in the dataset, and the responses predicted by the linear approximation. Mathematically, it has the following formula to minimize:

$$min_w \mathcal{L}(w) = min_w \frac{1}{2N} \sum_{n=1}^{N} (y_n - \tilde{x}_n^T w)^2 \tag{1}$$

where $\tilde{x}_n$ is the input features added by a row of constant 1.

When $\tilde{X}^T\tilde{X}$(where $\tilde{X}$ takes every sample feature as a row) is invertible, we have a closed-form expression for the minimum: $w^* = (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T y$. When it is not invertible, we can use pseudo-inverse to get the approximated solution. Roughly speaking, the pseudo-inverse of $\tilde{X}$ is $(\tilde{X}^T\tilde{X} + \varepsilon\mathcal{I})^{-1}\tilde{X}^T$, where the limits are taken with $\varepsilon > 0$.

2) **Penalized Logistic Regression for Classification:** A linear model for classification, such as Penalized Logistic Regression, was believed to be the simplest choice for a model. This model was usually trained using 5-Fold Cross Validation, regardless of the feature engineering conducted over the input variables. In this method, two hyperparameters should be considered, say, the penalization term(i.e. which norm to penalize) and the tradeoff between the objective function and the penalization term. Mathematically, it has the following formula to minimize:

$$min_{\omega,b}||\omega||_p + C\sum_i log(exp(-y_i(\tilde{X}_i^T\omega + b)) + 1) \tag{2}$$

where p=1 or 2, means the L_1 norm or L_2 norm, and C is the tradeoff term.

3) **Neural Networks:** Neural Networks are modeled as collections of neurons that are connected in an acyclic graph. In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network. Instead of an amorphous blobs of connected neurons, Neural Network models are often organized into distinct layers of neurons. For regular neural networks, the most common layer type is the fully-connected layer in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections.

As for the related hyperparameters, transformation function, parameter update method, and regularization should be considered. In the following sections, we focused on tuning these hyperparameters.

4) **Decision Tree:** The core idea of Decision Tree is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. It learns from data to approximate a decision with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.

Another property I need to mention is the impurity function, the popular choices is Gini index error for classification, i.e. $impurity(\mathcal{D}) = 1 - \sum_{k=1}^{K}(\frac{\sum_{n=1}^{N}\mathbb{1}[y_n=k]}{N})^2$, and residual sum of squares for regression, i.e. $impurity(\mathcal{D}) = \frac{1}{N}\sum_{n=1}^{N}(y_n - \overline{y})^2$, with $\overline{y}$ = average of y_n. Algorithm 3 is the mathematical formula for this algorithm:

Decision Tree is simple to understand and to interpret and requires little data preparation. However, Decision-tree

Algorithm 3 Decision Tree

function **DecisionTree**(data $\mathcal{D}$=$\{(\mathbf{x}_n, y_n)\}_{n=1}^N$)
if cannot branch anymore(e.g. gets to the maximum depth)
    return base hypothesis $g_t(\mathbf{x})$
else
    1. learn branching criteria
    b(x)=*arg min*$_{\text{decision stumps h(x)}} \sum_{c=1}^{2} |\mathcal{D}_c \text{ with } h| \cdot impurity(\mathcal{D}_c \text{ with } h)$
    2. split $\mathcal{D}$ to 2 parts $\mathcal{D}_c = \{(\mathbf{x}_n, y_n) : b(\mathbf{x}_n) = c\}$
    3. build sub-tree $G_c \leftarrow$ **DecisionTree**($\mathcal{D}_c$)
    4. return $G(\mathbf{x}) = \sum_{c=1}^{2} \mathbb{1}[b(\mathbf{x}) = c] \cdot G_c(\mathbf{x})$
where $g_t(\mathbf{x})$ in classification: majority of $\{y_n\}$
    $g_t(\mathbf{x})$ in regression is average of $\{y_n\}$

learners can create over-complex trees that do not generalize the data well. This is called overfitting. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem. In our implementation, I mainly use the maximum depth restriction to overcome overfitting.

5) **AdaBoost-Decision Tree:** The core principle of AdaBoost is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction. The data modifications at each so-called boosting iteration consist of applying weights u_1, u_2, ..., u_N to each of the training samples. Initially, those weights are all set to $u_i = \frac{1}{N}$, so that the first step simply trains a weak learner on the original data. For each successive iteration, the sample weights are individually modified and the learning algorithm is reapplied to the reweighted data. At a given step, those training examples that were incorrectly predicted by the boosted model induced at the previous step have their weights increased, whereas the weights are decreased for those that were predicted correctly. As iterations proceed, examples that are difficult to predict receive ever-increasing influence. That is where the name boosting from. Each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence. Algorithm 4 and 5 is the mathematical formula for this algorithm, step 4 and step 5 in these two algorithms are the boosting process respectively.

6) **Random Forest:** In random forests, each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set. In addition, when splitting a node during the construction of the tree, the split that is chosen is no longer the best split among all features. Instead, the split that is picked is the best split among a random subset of the features. As a result of this randomness, the bias of the forest usually slightly increases (with respect to the bias of a single non-random tree). But, due to averaging, its variance also decreases, usually more than compensating for the increase in bias, hence yielding an overall better model.

Algorithm 4 AdaBoost Classification

$u^{(1)}$ = $[\frac{1}{N}, \frac{1}{N}, ..., \frac{1}{N}]$
for `t = 1, 2, ..., T` **do**
    1. obtain g_t by $A(D, u^{(t)})$,
    where A tries to minimize $u^{(t)}$-weighted 0/1 error
    2. update $u^{(t)}$ to $u^{(t+1)}$ by
    $\mathbb{1}[y_n \neq g_t(x_n)]$: $u^{(t+1)} \leftarrow u^{(t)} \cdot \Delta_t$
    $\mathbb{1}[y_n = g_t(x_n)]$: $u^{(t+1)} \leftarrow u^{(t)} / \Delta_t$
    where $\Delta_t = \sqrt{\frac{1-\epsilon_t}{\epsilon_t}}$ and $\epsilon_t = \frac{\sum_n u_n^{(t)} \mathbb{1}[y_n \neq g_t(x_n)]}{\sum_n u_n^{(t)}}$
    3. compute $\alpha_t = ln(\Delta_t)$
    return $G(x) = sign(\sum_t \alpha_t g_t(x))$
end for

Algorithm 5 AdaBoost Regression

$u^{(1)}$ = $[\frac{1}{N}, \frac{1}{N}, ..., \frac{1}{N}]$
for `t = 1, 2, ..., T` **do**
    1. obtain g_t by $A(D, u^{(t)})$,
    where A tries to minimize $u^{(t)}$-weighted MSE
    2. Calculate the adjusted error e_n^t for each instance:
    let M_t = $max_n |y_n - g_t(x_n)|$
    then normalized e_n^t = $|y_n - g_t(x_n)|/M_t$
    2. update $u^{(t)}$ to $u^{(t+1)}$ by
    $\mathbb{1}[y_n \neq g_t(x_n)]$: $u^{(t+1)} \leftarrow u^{(t)} \cdot \Delta_t$
    $\mathbb{1}[y_n = g_t(x_n)]$: $u^{(t+1)} \leftarrow u^{(t)} / \Delta_t$
    where $\Delta_t = \sqrt{\frac{1-\epsilon_t}{\epsilon_t}}$ and $\epsilon_t = \frac{\sum_n u_n^{(t)} \mathbb{1}[e_n^t > 0.5]}{\sum_n u_n^{(t)}}$
    3. compute $\alpha_t = ln(\Delta_t)$
    return $G(x) = sign(\sum_t \alpha_t g_t(x))$
end for

The main parameters to adjust when using this methods is

1. the number of trees in the forest(named $n\_estimators$): the larger the better, but also the longer it will take to compute;

2. the size of the random subsets of features to consider when splitting a node(named $max\_features$): the lower the greater the reduction of variance, but also the greater the increase in bias;

3. another parameter should be considered is the maximum depth of the tree(named max_depth).

Algorithm 6 is the mathematical formula for this algorithm:

Algorithm 6 Random Forest

function **RandomForest**(data $\mathcal{D}$=$\{(\mathbf{x}_n, y_n)\}_{n=1}^N$)
for `t = 1, 2, ...,` $n\_estimators$ **do**
    1. request size-N data $\mathcal{D}_t$ by bootstrapping with $\mathcal{D}$, where N is the same as the original sample size.
    2. obtain tree g_t by **DecisionTree**($\mathcal{D}_t$), where DecisionTree uses only $max\_features$ to do the decision and the maximum depth of DecisionTree is max_depth.
end for
return G = Uniform($\{g_t\}$)

From the algorithm above, we can see the final hypothesis is the uniform(i.e. averaging) of all the estimators(i.e. predictions). Let $g_1, g_2, ..., g_T$ be the T different models we trained. If we take the g_i to be identically distributed, with Variance(g_i) = σ^2, Covariance(g_i, g_j) = $\rho\sigma^2$ if $i \neq j$. Then the averaged prediction $z_T = \frac{1}{T}\sum_{i=1}^{T} g_i$, and the Variance($z_T$) = $\frac{1}{T}\sigma^2 + \rho\frac{T-1}{T}\sigma^2$. Then:

Variance reduction ratio: $\frac{Var(g_i)}{Var(z_T)} = \frac{T}{1+\rho(T-1)}$.

Therefore, if we use model averaging, we want large number of estimators to reduce the variance.

7) **K Nearest Neighbors:** The principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number of samples can be a user-defined constant (k-nearest neighbor learning), or vary based on the local density of points (radius-based neighbor learning). Here, I only use the K-nearest neighbors to measure our problem.

Neighbors-based methods are known as non-generalizing machine learning methods, since they simply remember all of its training data. Despite its simplicity, nearest neighbors has been successful in a large number of classification and regression problems, including handwritten digits or satellite image scenes. Being a non-parametric method, it is often successful in classification situations where the decision boundary is very irregular.

Neighbors-based classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point. Mathematically, it has the following prediction formula:

$$g_k(\mathbf{x}) = \mathbb{1}[\frac{1}{k}\sum_{\mathbf{x}_n \in nbh_k(\mathbf{x})} y_n > 0.5] \tag{3}$$

where $nbh_k(\mathbf{x})$ is the neighborhood of $\mathbf{x}$ defined by k closest points in the training data.

Neighbors-based regression can be used in cases where the data labels are continuous rather than discrete variables. The label assigned to a query point is computed based the mean of the labels of its nearest neighbors. Mathematically, it has the following prediction formula:

$$g_k(\mathbf{x}) = \frac{1}{k}\sum_{\mathbf{x}_n \in nbh_k(\mathbf{x})} y_n \tag{4}$$

The K-nearest neighbors regressor or classifier requires a setting for K, but what number works best? That's the parameter we need to train carefully. Additionally, we saw that there are many different distance functions we could have used: L1 norm, L2 norm. As we know, the basic nearest neighbors algorithm uses **uniform weights**: that is, each point in the local neighborhood contributes uniformly to the result of a query point. Under some circumstances, it can be advantageous to weight points such that nearby points contribute more to the result than far away points. In this case, we can assign weights proportional to the **inverse of the distance** from the query points and we call this weights as **distance weights**.

8) **Uniform Blending:** If we have diverse hypotheses, i.e. many different algorithms of regression or classification, then even simple uniform blending of these hypotheses can be better than any single hypothesis.

For classification, roughly speaking, if we have very different hypothesis, then the majority these hypothesis can correct minority, thus have better performance. The mathematical algorithm can be explained as following:

$$G(\mathbf{x}) = \mathbb{1}[\sum_{t=1}^{T} g_t(\mathbf{x}) > 0.5] \tag{5}$$

where T algorithms $g_1, ..., g_T$ predict the same problem.

For regression, very different g_t: some $g_t(\mathbf{x}) > f(\mathbf{x})$, some $g_t(\mathbf{x}) < f(\mathbf{x})$(where $f(x)$ be the true hypothesis), then average could be more accurate than individual. The mathematical algorithm can be explained as following:

$$G(x) = \frac{1}{T}\sum_{t=1}^{T} g_t(x) \tag{6}$$

Theoretical Analysis of Uniform Blending:

$$\begin{aligned} avg((g_t(x) - f(x))^2) &= avg(g_t^2 - 2g_t f + f^2) \\ &= avg(g_t^2) - 2Gf + f^2 \\ &= avg(g_t^2) - G^2 + (G - f)^2 \\ &= avg(g_t^2 - 2g_t G + G^2) + (G - f)^2 \\ &= avg((g_t - G)^2) + (G - f)^2 \end{aligned} \tag{7}$$

which means:

$$\begin{aligned} avg(E_{out}(g_t)) &= avg(\mathbf{E}(g_t - G)^2) + E_{out}(G) \\ &\geq E_{out}(G) \end{aligned} \tag{8}$$

where $E_{out}(g)$ is the out-of-sample error.

From the analysis above, we can see that the uniform blending reduces variance for more stable performance. This algorithms is very similar to random forest. But it is from the aspect of diverse hypotheses rather than the sample space or feature space.

9) **Q Learning:** Other than Regression and Classification methods, we also considered Q-learning, a specie of reinforcement learning. The standard Q-learning formula is:

$$Q(a', s') = R(s', a') + \gamma max_{a'}(Q(a', s')) \tag{9}$$

Here, $R(s, a)$ is the instant reward, γ is the discount factor for future rewards, and s' is the state resulting from taking action a in state s. There are two possible actions in each state: b for 'buy' and w for 'wait'. The Q-Learning algorithm goes as Algorithm 7.

The algorithm is used by the agent to learn from experience. Each episode is equivalent to one training session. In each training session, the agent explores the environment (represented by matrix R), receives the reward (if any) until it reaches the goal state. The purpose of the training is to enhance the 'brain' of our agent, represented by matrix Q. More training results in a more optimized matrix Q. In this

Algorithm 7 Q Learning

1: Set the γ parameter, and environment rewards in matrix R.
2: Initialize matrix Q to zero.
3: **for** `each episode` **do**
Select a random initial state.
4: **while** the goal state hasn't been reached **do**
- Select one among all possible actions for the current state.
- Using this possible action, consider going to the next state.
- Get maximum Q value for this next state based on all possible actions.
- Compute: Q(state, action) = R(state, action) + γ * Max[Q(next state, all actions)]
- Set the next state as the current state.

5: **end while**
6: **end for**

case, if the matrix Q has been enhanced, instead of exploring around, and going back and forth to the same states, the agent will find the fastest route to the goal state.

The γ parameter has a range of 0 to 1 ($0 <= \gamma <= 1$). If γ is closer to zero, the agent will tend to consider only immediate rewards. If γ is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.

To use the matrix Q, the agent simply traces the sequence of states, from the initial state to goal state. The algorithm finds the actions with the highest reward values recorded in matrix Q for current state:

Algorithm to utilize the Q matrix is as Algorithm 8. This algorithm will return the sequence of states from the initial state to the goal state.

Algorithm 8 Use Q matrix

1: Set current state = initial state.
2: From current state, find the action with the highest Q value.
3: Set current state = next state.
4: Repeat Steps 2 and 3 until current state = goal state.

E. Performance Benchmarks

The naive purchase algorithm, called the *Random Purchase*, is to purchase a ticket randomly before the departure date. To be more concrete, for every departure date, for example departure date B, then from the first historical data for this departure date(namely date A), we pick several tickets randomly in this interval to simulate the clients buying ticket. And the average price would be computed as the *Random Purchase Price*. Another purchase strategy benchmark introduced in [Groves, et al., 2013] is called *earliest purchase*, in which it splits the time interval into many day periods and it purchases one ticket in each day period. These two performance benchmarks are similar. However, we thought that the *Random Purchase* strategy conforms more closely to reality because in reality some clients may choose the same time period to buy. For example, imagine there are 100 clients want to buy the tickets from date A to the departure date B. The random purchase strategy will purchase randomly in this period. But the *earliest purchase* strategy may split the time period of date A and date B into 50 periods(i.e. 1st time period, 2nd time period, ..., 50th time period). And in each time period, it will buy two tickets in the first day of each time period. But this is usually not real. Because 3 or more clients may buy in the 1st time period. And maybe no client will buy ticket in the 50th time period. So this is the main idea of random purchase strategy and it is much easier to implement so that we chose *Random Purchase* strategy as our benchmark.

The lowest achievable cost is called the *Optimal Price* and it is the lowest price between the first query date and the departure date.

F. Performance Metric

As long as we get the *Randome Purchase Price*, *Optimal Price*, and the *Predicted Price*, we can use the following performance metric to evaluate our results:

$$Performance = \frac{Random\ Purchase\ Price - Predicted\ Price}{Random\ Purchase\ Price}\% \tag{10}$$

$$Optimal\ Perfor = \frac{Random\ Purchase\ Price - Optimal\ Price}{Random\ Purchase\ Price}\% \tag{11}$$

$$Normalized\ Performance = \frac{Performance}{Optimal\ Performance}\% \tag{12}$$

Having these metrics in mind, we used the Normalized Performance to evaluate our results, because it normalizes every route and it ranges from 0% to 100%, in which case, the higher the better, so that it gives more intuition about how well or bad is the result performance.

VI. Regression Results

A. Hyperparameter Tuning

1) **Least Squares:** In our problem, although the Least Squares did not work very well, but it is useful for the Uniform Blending Model described in the following section.

2) **Neural Networks:**

a) **Hidden Layers:** Regarding the neural network architecture, we explored several configurations in terms of number of nodes and number of hidden layers. We reached the conclusion that the neural network architecture that performed the best consisting of a single hidden-layer neural network of 6 neurons. Even though the increasing number of hidden layers enables to express the output as any function of the input, it also conveys the risk of the model overfit. Hence, a single hidden-layer model can reasonably well explain the input output relationship. Regarding the number of neurons of this layer, it is known that a low number of neurons in the

hidden layers implies loss of important information, while a higher number of neurons can prevent convergence. We ran our models under 100 epochs, which conveys convergence while preventing to high computational cost.

b) **Activation Function:** Finally, as shown above, we defined three layers, namely the input layer, the hidden layer and the output layer. We set the output's activation function to be sigmoid function. So the output of the data is the probability of buying the ticket on that query date. The nonlinearity function used by other layers is $rectifier$, which is simply $max(0, x)$. It is the most popular choice of activation function these days. The neural net's weights are initialized from a uniform distribution with a cleverly chosen interval. That is, Lasagne(a python package) figures out this interval for us, using "Glorot-style" initialization.

c) **Optimization Methods:** The update function will update the weights of our network after each batch. The most naive method is stochastic gradient decent (SGD): the parameter moves in the opposite direction of gradient: $\triangle x = -\eta \frac{\partial f}{\partial x}$, where η is the learning rate. The drawback of SGD is that it is easy to get stuck in local minimum.

A revised version is **momentum**, this algorithm remembers the latest update and adds it to present update by multiplying a parameter ρ called momentum: $\triangle x_t = \rho \triangle x_{t-1} - \eta \frac{\partial f(x_{t-1})}{\partial x}$. When $\triangle x_t$ and $\triangle x_{t-1}$ are of the same direction, the momentum accelerates the update step; while they are of the opposite direction, the algorithm tends to update in the former direction if x has been updated in this direction for many iterations. This mechanism helps x to go over the local minimum but makes the model more difficult to converge especially when ρ is big.

Nesterov momentum is a slightly different version of the momentum update has recently been gaining popularity. The core idea behind Nesterov momentum is that when the current parameter vector is at some position x, then looking at the momentum update above, we know that the momentum term alone (i.e. ignoring the second term with the gradient) is about to nudge the parameter vector by $\rho \triangle x_{t-1}$. Therefore, if we are about to compute the gradient, we can treat the future approximate position $x_{t-1} + \rho \triangle x_{t-1}$ as a lookahead - this is a point in the vicinity of where we are soon going to end up. Hence, it makes sense to compute the gradient at $x_{t-1} + \rho \triangle x_{t-1}$ instead of at the old position x. Finally, the step has this form: $\rho \triangle x_{t-1} - \eta \frac{\partial f(x_{t-1} + \rho \triangle x_{t-1})}{\partial x}$. Fig 6, 7 shows the difference between momentum and Nesterov momentum. To see the effect of different update methods, I will show this in the Neural Networks Classification section.

We'll use the $nesterov\_momentum$ gradient descent optimization method to do the job, because it enjoys stronger theoretical converge guarantees for convex functions and in practice it also consistenly works slightly better than standard momentum. There's a number of other methods that Lasagne implements, such as $adagrad$ and $rmsprop$. We chose $nesterov\_momentum$ because it has proven to work very well for a large number of problems.

Because it is a regression problem. So we specified the mean squared error(MSE) as an objective function to minimize.

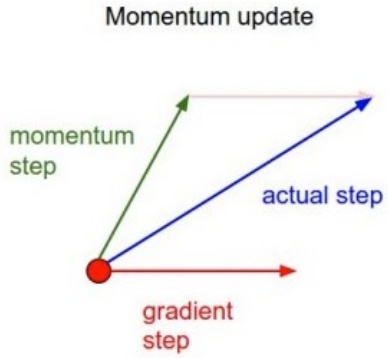

Fig. 6. Momentum

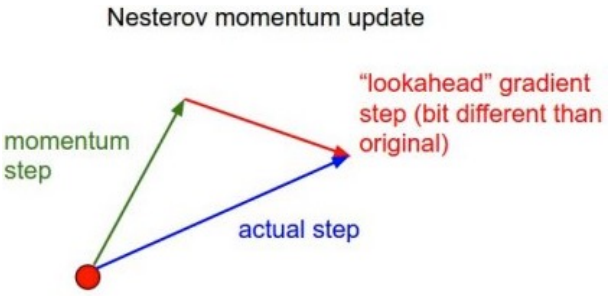

Fig. 7. Nesterov momentum, instead of evaluating gradient at the current position (red circle), we know that our momentum is about to carry us to the tip of the green arrow. With Nesterov momentum we therefore instead evaluate the gradient at this "looked-ahead" position.

Then the data provided in X were split into a training and a validation set, using 20% of the samples for validation.

3) **Decision Tree:** In Fig 8, the x axis represents the parameter max_depth of Decision Tree, while the y axis stands for Cross Validation MSE. It can be observed that along with growth of max_depth, the MSE of our refressor firstly decrease step by step then increases, and finally it remains a stable state which has a small variance. This result meets our expectation, as increase the number of max_depth will increase the power of Decision Tree, which will have overfitting. We then set max_depth to 7, and similarly, we set the number of features to be considered when looking for the best split to be all the features.

4) **AdaBoost-Decision Tree:** Similarly, we can get all the hyperparameters for AdaBoost-Decision Tree, as shown in the Table III. The validation curve is similar to Decision Tree so I do not show it here.

5) **Random Forest:** In Fig 9, the x axis represents the parameter max_depth of Decision Tree in Random Forest while the y axis stands for Cross Validation MSE. It can be observed that along with growth of max_depth, the MSE of our refressor firstly decrease step by step, and finally it remains a stable state which has a small variance. This result meets our expectation, as increase the number of max_depth will increase the power of Decision Tree. We then set max_depth to 39.

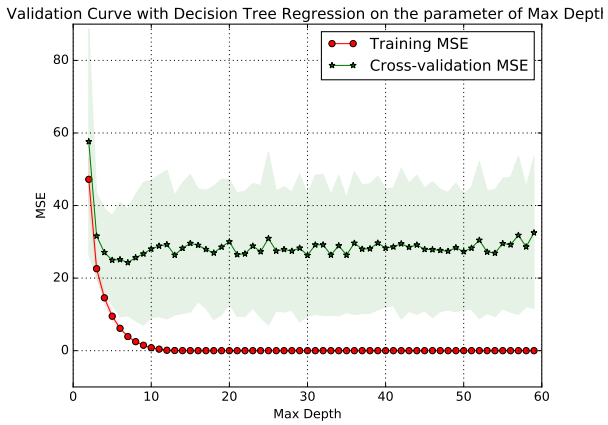

Fig. 8. Effects of tuning the parameter max_depth. Numbers are arithmetic mean of results from 5-fold cross validation and in the case of all features are to be considered when looking for the best split. And the shadow part of the image is the **variance** of the cross validation precision or error for all the folds, so as the following validation curves.

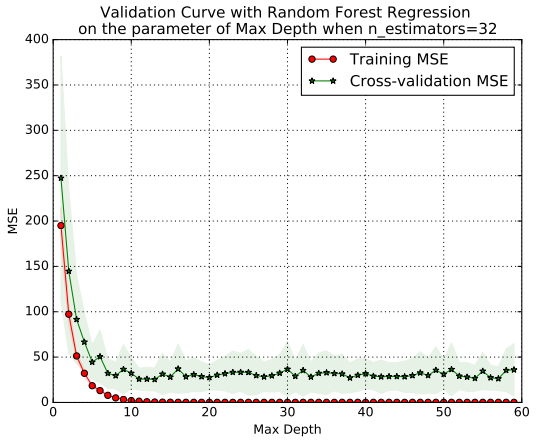

Fig. 9. Effects of tuning the parameter max_depth. Numbers are arithmetic mean of results from 5-fold cross validation and in the case of estimators is equal to 32.

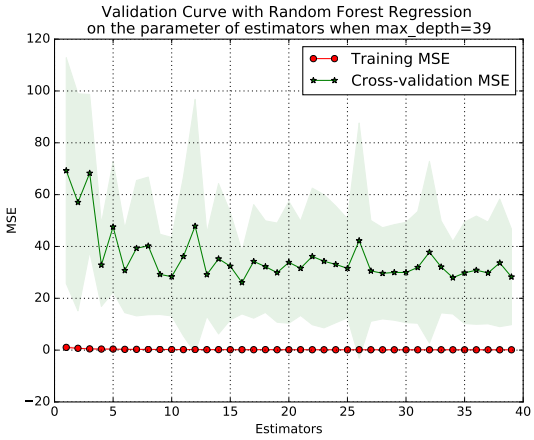

Fig. 10. Effects of tuning the parameter $n\_estimators$. Numbers are arithmetic mean of results from 5-fold cross validation and in the case of max depth is equal to 39.

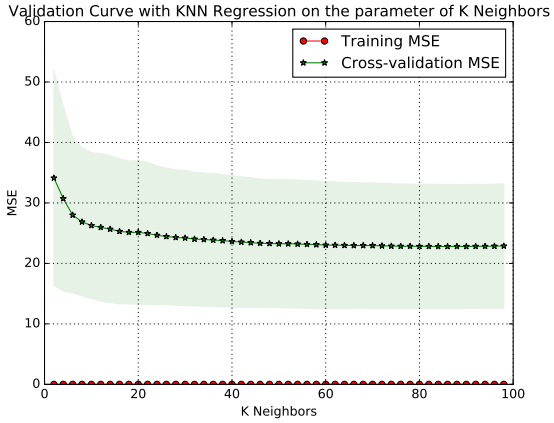

Fig. 11. Effects of tuning the parameter K nearest neighbors. Numbers are arithmetic mean of results from 5-fold cross validation and in the case of distance weight is to be considered when looking for the best split.

And theoretically, the larger the number of estimators, the better. From Fig 10, when the number of estimators larger than 32, the effects of it become negligible, we then set it to be 32, which is not too large and gets the relative small MSE from 5-fold cross validation. As we can see from the figure, when the number of estimators is smaller than 32, it has a large effect on the performance, the performance bounds up and down randomly. This parameter has different validation curve from others. That is the bad performance of small number of estimators.

And similarly, we set the number of features to be considered when looking for the best split to be all the features.

6) **K Nearest Neighbors:** In Fig 11, the x axis represents the parameter K nearest neighbors while the y axis stands for Cross Validation MSE. It can be observed that along with growth of K, the MSE of our regressor firstly decreases step by step and then remains 22.5, and finally increases slightly. We then set K to 86 and the distance used is distance weights.

7) **Uniform Blending:** The uniform blending is the average by the previous 6 models. As we saw in the result Table II, the uniform blending got the relatively small variance compared to other models, although it did not get the smallest variance.

B. Regression Performance Results

Table II shows the results of regression methods. Random Forest Regression gets the best performance in regression method. However, it's variance is not small enough, which means it is sensitive to different routes. In this case, although for some routes, it gets good performance, for other routes, it gets bad performance. From the aspect of the clients, it is not fair for the some clients to buy tickets for which the system may predict badly. The preferred method in regression is AdaBoost-Decision Tree Regression method, which has smallest variance and a relative high performance.

Performance(%)	Model								
Routes \ Method	Optimal	Random Purch.	Linear Regression	NN	Decison Tree	KNN	AdaBoost	Random Forest	Uniform Blending
BCN→BUD	100.0	0.00	-42.17	67.03	42.31	38.19	48.49	54.67	44.37
BUD→BCN	100.0	0.00	1.91	57.12	72.66	91.96	71.59	96.78	80.17
CRL→OTP	100.0	0.00	10.34	18.60	30.40	37.48	41.01	56.35	25.68
MLH→SKP	100.0	0.00	-21.02	-10.74	22.37	36.07	31.50	77.17	29.22
MMX→SKP	100.0	0.00	-118.53	53.25	16.46	67.91	53.80	-0.69	65.70
OTP→CRL	100.0	0.00	-19.47	57.30	63.85	69.09	64.11	67.51	63.85
SKP→MLH	100.0	0.00	47.72	35.80	50.11	52.10	51.30	45.34	45.34
SKP→MMX	100.0	0.00	-96.14	64.63	43.33	30.07	43.33	46.34	49.76
Mean Performance	100.0	0.00	-29.67	42.87	42.68	52.86	50.64	**55.43**	50.51
Variance	0.00	0.00	2654.78	636.21	**37.04**	409.11	**143.49**	707.99	302.90

TABLE II
Regression Normalized Performance Comparison of 8 routes.

Model	Hyperparameters
Neural Networks	hidden layers=1 hidden units=6 optimization method=nesterov_momentum update_learning_rate=0.002 update_momentum=0.4
Decision Tree	max_depth=7 max_features=All
AdaBoost-Decision Tree	max_depth of DT = 10 max_features of DT=All num_estimators=400
Random Forest	max_depth=39 num_estimators=32 max_features=sqrt(All features)
KNN	num_neighbors=86 weights=distance distance metric=L2 Norm

TABLE III
Hyperparameter Tuning in Regression.

VII. Classification Results

A. Solving Imbalanced Data Set

Concerning a classification problem, an imbalanced data set leads to biased decisions towards the majority class and therefore an increase in the generalization error. As referred in the section of Data Description and Interpretation of the classification problem, the number of samples per class in our problem is not equally distributed, only few entries are to buy, most of the entries should be to wait. To address this problem, we considered three approaches.

- **Random Under Sampling** - Randomly select a subset the majority classes' data points so that the number of data points of each class is equal to the number of points of the minority class - Class 1 (to buy).
- **Random Over Sampling** - Randomly add redundancy to the data set by duplicating data points of the minority classes so that the number of data points of each class is equal to the number of points of the majority class - Class 2 (to wait).
- **Algorithmic Over Sampling** - Add redundancy to the data set by simulating the distribution of each minority class and consequently creating new data points so that the number of data points of each class is equal to the number of points of the majority class - Class 2.

Having these methods in mind, firstly, the *Random Under Sampling* would not be useful in our problem, because in our problem, the data set is very unbalanced, i.e. the buy entries is very sparse. If we use *Random Under Sampling*, we will lose many information. Secondly, if we use *Algorithmic Over Sampling*, it will add many noises into the data, because we do not know the hidden relationship between the features and the output. As a result, we preferred the second method, which is *Random Over Sampling*

B. Identification of Outliers

We addressed the question of outlier removal making use of unsupervised learning methods, in particular through the implementation of K-Means and EM algorithm. Our approach was based on the fact that each characteristic class (either class 1 or 2) should in theory be restricted to a certain volume of the input space. Bearing this fact in mind, each of these two classes can be thought as a cluster. Consequently, we consider a sample to be an outlier if it does not belong to its labeled class cluster.

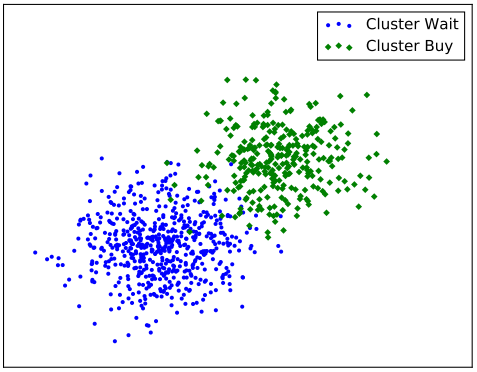

Fig. 12. Outlier Removal.

Having said that, we applied K-Means and EM to a training dataset. In the case of K-Means, 5896 samples were tagged as outliers. In the case of EM algorithm, 7318 samples were tagged as outliers. The choice of the initial conditions was

crucial in terms of the algorithms convergence and in terms of obtaining a reliable result.

Regarding the initial conditions, for our propose of use, we computed the initial center to be such that:

$\mu_{\mathbf{k}}^{\mathbf{initial}} = \frac{\mathbf{1}}{\mathbf{N_k}} \sum_{\mathbf{x_n} \in \mathbf{C_k}} \mathbf{x_n}$, where N_k is the number of points belongs to Class k.

Fig 12 shows how does the outlier can be found due to clustering methods. The cluster buy entries evolve in the cluster wait should be considered to be outlier, and vice-versa.

We tested our classification methods on both the datas with and without outlier removal. Then we chose the best result to show.

C. Hyperparameter Tuning

1) **Logistic Regression:** In Fig 13, the x axis represents the parameter C which is the tradeoff between objective function and penalization term, while the y axis stands for Cross Validation Precision. It can be observed that along with growth of C, it will focus more on the objective function, and the Precision of our classifier firstly increase step by step, then it drops a little. And finally it remains a stable state which has a precision of 67.8%. This result meets our expectation, as increase the number of C will penalize less. We then set C to 0.0010985 and penalization term to be L1 norm which is also tuned from 5-fold cross validation. In other words, finally, the mathematical formula was chosen to be in the following form:

$$min_{\omega,b}||\omega||_1 + C\sum_i log(exp(-y_i(X_i^T\omega + b)) + 1) \quad (13)$$

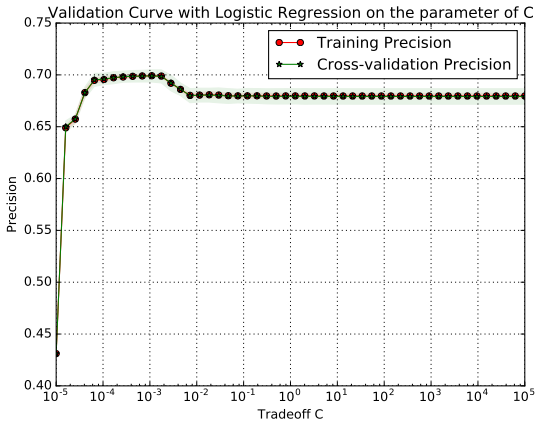

Fig. 13. Effects of tuning the parameter C in logistic regression. Numbers are arithmetic mean of results from 5-fold cross validation and in the case of L1 norm is to be considered when looking for the best parameter. And the shadow part of the image is the **variance** of the cross validation precision or error for all the folds, so as the following validation curves.

2) **Neural Networks:** Finally, we defined three layers, namely the input layer, the hidden layer with 7 neurons and the output layer. We set the output's activation function to be sigmoid function. So the output of the data is the probability of buying the ticket on that query date. The nonlinearity function used by other layer is $rectifier$, which is simply $max(0, x)$. The neural net's weights are initialized from a uniform distribution with a cleverly chosen interval.

Although it is a classification problem, due to its probability property, it can also be seen as a regression problem. So we specified the mean squared error(MSE) as an objective function to minimize. Why not try to maximize that number directly, rather than minimizing a proxy measure like the quadratic cost? The problem with that is that the number of entries correctly classified is **not a smooth function** of the weights and biases in the network. For the most part, making small changes to the weights and biases won't cause any change at all in the number of training datas classified correctly. That makes it difficult to figure out how to change the weights and biases to get improved performance. If we instead use a smooth cost function like the quadratic cost it turns out to be easy to figure out how to **make small changes** in the weights and biases so as to get an improvement in the cost. That's why we focus on minimizing the quadratic cost - MSE, and only after that will we examine the classification accuracy.

Then the data provided in X were spit into a training and a validation set, using 20% of the samples for validation.

a) ***Effect of Update Methods****:* The update function will update the weights of our network after each batch. We used the $nesterov\_momentum$ gradient descent optimization method to do the job. To see the effect of different update methods, Fig 14 shows the effects of different update methods for different epochs. It shows the convergence of three methods for the same learning rate=0.1. Model using Nesterov momentum update policy achieves the highest accuracy on validation set but converges slower than SGD. This is partly because its mechanism makes it more easier to miss the minimum and thus slower the convergence. On the contrary, SGD has the worst performance in respect of accuracy. This is partly because its mechanism makes it more easier to get stuck in local minimum.

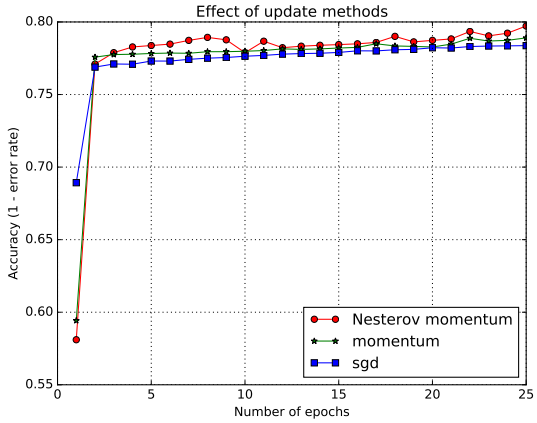

Fig. 14. Effect of update methods on validation set.

3) **Decision Tree:** In the classification Decision Tree method, we set $Gini\ impurity$ as the criterion to measure the

split of each step. In Fig 15, the x axis represents the parameter max_depth of Decision Tree while the y axis stands for Cross Validation Precision. It can be observed that along with growth of max_depth, the Precision of our classifier firstly increase step by step, and finally it remains a stable state which has a small variance. This result meets our expectation, as increase the number of max_depth will increase the power of Decision Tree, which will have slight overfitting. We then set max_depth to 45 and the number of features to consider when looking for the best split should be log(total features). However, when we tested the performance of the test set, there was always one or more routes get negative performance over the test dataset. But if we set the maximum depth to be 45, we almost got 0 error rate for the cross validation, i.e. 100% normalized performance, which had overfitting. Although we got overfitting in the method of Decision Tree Classification, we thought it was still a good news. Because we saw the power of Decision Tree Classification. We can use Decision Tree Classification to **form a stronger leaner**. See the section of AdaBoost-Decision Tree.

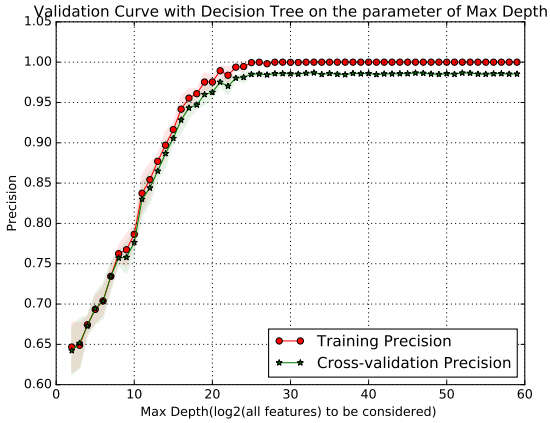

Fig. 15. Effects of tuning the parameter max_depth. Numbers are arithmetic mean of results from 5-fold cross validation and in the case of log2(all features) are to be considered when looking for the best split.

4) **AdaBoost-Decision Tree:** In this method, as we saw that Decision Tree Classification with $Maximum\,Depth = 45$ got overfitting, we set the maximum depth of Decision Tree to be 10, and we set the number of estimators at which boosting is terminated to be $T = 5$ due to 5-fold cross validation as shown in Table IV. The validation curve is similar to Decision Tree.

5) **Random Forest:** Similarly from Decision Tree and AdaBoost-Decision Tree, we can get the hyperparamters for this algorithm due to 5-fold cross validation. The result is shown in Table IV.

In Fig 16, the x axis represents the parameter max_depth of Decision Tree in Random Forest while the y axis stands for Cross Validation Precision. It can be observed that along with growth of max_depth, the precision of our classifier firstly decrease step by step then increases, and finally it remains a stable state which has a small variance. This result meets our expectation, as increase the number of max_depth will increase the power of Decision Tree. We then set max_depth to 30.

Model	Hyperparameters
Logistic regression	tradeoff term C=0.0010985
Neural Networks	hidden layers=1 hidden units=6 optimization method=nesterov_momentum update_learning_rate=0.002 update_momentum=0.9
Decision Tree	max_depth=45 max_features=log(All features)
AdaBoost-Decision Tree	max_depth of DT = 10 max_features of DT=log(All features) num_estimators=5
Random Forest	max_depth=30 num_estimators=20 max_features=log(All features)
KNN	num_neighbors=2 weights=uniform distance metric=L2 Norm

TABLE IV
Hyperparameter Tuning in Classification.

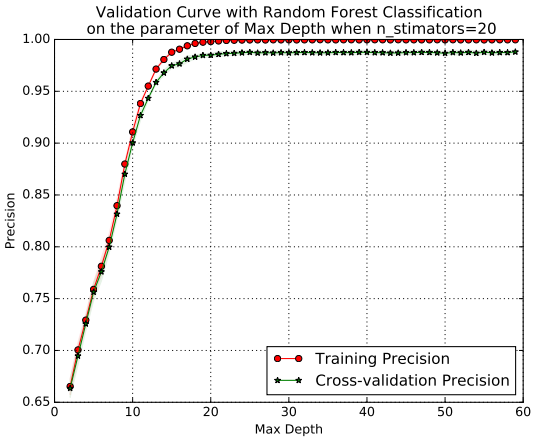

Fig. 16. Effects of tuning the parameter max_depth. Numbers are arithmetic mean of results from 5-fold cross validation and in the case of estimators is equal to 32.

And theoretically, the larger the number of estimators, the better. From Fig 17, when the number of estimators larger than 20, the effects of it becomes negligible, we then set it to be 20. As we can see from the figure, when the number of estimators is smaller than 20, its has a large effect on the performance, the performance bounds up and down randomly. That is the bad performance of small number of estimators as already described in Random Forest Regression.

And similarly, we set the number of features to be considered when looking for the best split to be all the features.

6) **K Nearest Neighbors:** In Fig 18, the x axis represents the parameter K nearest neighbors while the y axis stands for Cross Validation Precision. It can be observed that along with growth of K, the Precision of our classifier firstly increase a little and then decrease step by step, and finally it remains about 0.8 precision. This result meets our expectation, as increase the number of K will take irrelevant neighbors into

Performance(%)	Model								
Routes \ Method	Optimal	Random Purch.	Logistic Reg	NN	Decison Tree	KNN	AdaBoost	Random Forest	Uniform Blending
BCN→BUD	100.0	0.00	36.13	36.13	54.67	17.58	75.27	29.95	48.49
BUD→BCN	100.0	0.00	39.97	39.98	11.02	64.62	60.34	86.60	30.32
CRL→OTP	100.0	0.00	18.60	18.60	70.51	83.48	71.69	52.81	45.73
MLH→SKP	100.0	0.00	-22.16	-22.15	57.76	38.35	32.64	0.67	46.34
MMX→SKP	100.0	0.00	53.25	53.26	-26.42	34.44	50.76	-34.16	65.15
OTP→CRL	100.0	0.00	57.30	57.30	80.35	87.69	85.33	19.83	71.71
SKP→MLH	100.0	0.00	31.03	31.03	-28.2	55.28	65.02	33.61	49.12
SKP→MMX	100.0	0.00	64.63	64.63	23.63	14.99	49.76	9.57	57.8
Mean Performance	100.0	0.00	34.84	34.84	30.42	49.55	**61.35**	24.86	51.83
Variance	100.0	0.00	660.74	660.74	1592.45	679.48	151.25	1148.16	**144.56**

TABLE V
Classification Normalized Performance Comparison of 8 routes.

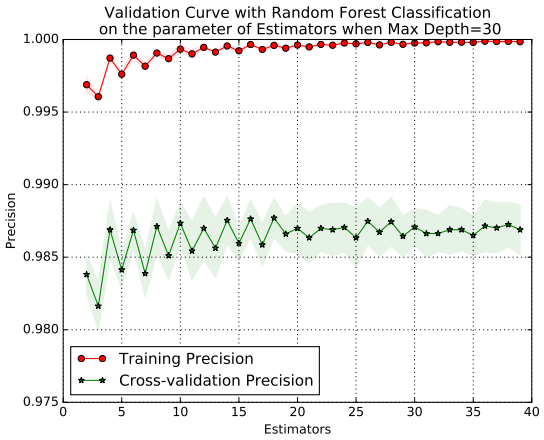

Fig. 17. Effects of tuning the parameter $n\_estimators$. Numbers are arithmetic mean of results from 5-fold cross validation and in the case of max depth is equal to 39.

account, which will have negative influence. We then set K to 2 and the distance used is uniform weights.

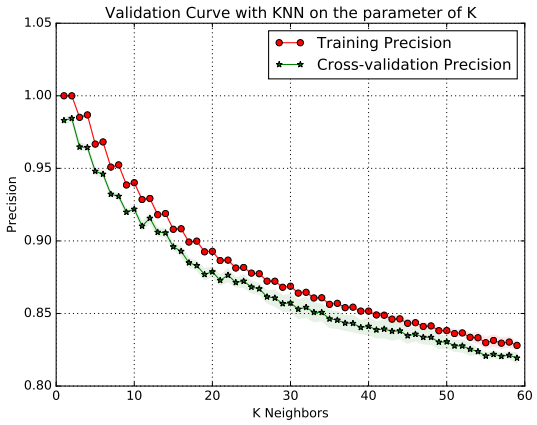

Fig. 18. Effects of tuning the parameter K nearest neighbors. Numbers are arithmetic mean of results from 5-fold cross validation and in the case of uniform weight is to be considered when looking for the best split.

a) **Interpretation of KNN classification:** From the parameter tuning result, we set the K to be 2, and $K = 1$ also gets good result. So only one or two nearest neighbors should be considered to predict the result. From the features we extracted: *flight number*, *minimum price so far*, *maximum price so far*, *query-to-departure*, *days-to-departure*, and *current price*. The *flight number* is encoded as dummy variables, so it has little influence on the euclidean distance of two input vectors. We conclude that the other 4 features have larger contribution to decide the classification result.

7) **Uniform Blending:** The uniform blending is the majority voting by the previous 6 models. As we saw in the result Table V, the uniform blending got the smallest variance compared to other models.

D. Classification Performance Results

Table V shows the results of classification methods. As we see, AdaBoost-DecisionTree, KNN, and Uniform Blending get positive performance for all the 8 routes and have smaller variance over these routes compared to other classification algorithms. The AdaBoost-DecisionTree method gets the best performance and a relative low variance over 8 routes. And as expected, the uniform blending method has the lowest variance just like the theory of uniform blending describes.

VIII. Q Learning

A. Parameter Assignment

In our case, the reward associated with b (buy) is the negative of the ticket price at that state, and the state resulting from b is a terminal state so there is no future reward. The instant reward associated with w (wait) is zero. And we set γ = 1, so we did not discount future reward.

In short, we define the Q function as following:

$$\begin{aligned} Q(b,s) &= -price(s) \\ Q(w,s) &= max(Q(b,s'), Q(w,s')) \end{aligned} \tag{14}$$

B. Averaging step by Equivalence class

We defined an **equivalence class** over states. Our equivalence class is the set of states with the same flight number and the same days before takeoff, but different departure dates. We denote that s and s^* are in the same equivalence class. Thus, our revised Q-learning formula is as following:

$$Q(a',s') = Avg_{s^* \sim s}(R(s^*,a') + \gamma max_{a'}(Q(a',s'))) \tag{15}$$

The output of Q-learning is the learned policy, which determines whether to buy or wait in unseen states by mapping them to the appropriate equivalence class and choosing the action with the lowest learned cost.

Performance(%)	Model		
Routes \ Method	Optimal	Random Purch.	Q Learning
BCN→BUD	100.0	0.00	68.76
BUD→BCN	100.0	0.00	61.81
CRL→OTP	100.0	0.00	51.13
MLH→SKP	100.0	0.00	54.01
MMX→SKP	100.0	0.00	61.27
OTP→CRL	100.0	0.00	72.08
SKP→MLH	100.0	0.00	65.29
SKP→MMX	100.0	0.00	6.50
Mean Performance	100.0	0.00	55.11
Variance	0.00	0.00	380.06

TABLE VI
Q Learning Performance.

C. Q Learning Performance Result

Table VI shows the result of Q-Learning. As we see, the Q-Learning method described in [Etzioni et al., 2003] has an acceptable performance and the variance is not large as well. The performance of it is very close to AdaBoost-DecisionTree Classification and Uniform blending Classification algorithms.

IX. Generalized Model

A. Uniform Blending

Our first idea came to mind is uniform blending again. From our specific problem, we got 8 patterns(i.e. 8 flight numbers). As long as we trained the specific problem, we could get 8 models (or learners) for 8 flight numbers separately. After that we can use the 8 models to predict for our new route, then we let these 8 models vote to buy or wait for every ticket.

B. HMM Sequence Classification

The second idea came to us is to allocate every data entry a "flight number" from the 8 routes, rather than average the 8 models. We then used sequence classification to allocate the "flight number" to every data entry.

1) **HMM Basics:**

- A *Markov chain* or *process* is a sequence of events, usually called *states*, the probability of each of which is dependent only on the event immediately preceding it.
- A *Hidden Markov Model* (HMM) represents stochastic sequences as Markov chains where the states are not directly observed, but are associated with a probability density function (pdf). The generation of a random sequence is then the result of a random walk in the chain (i.e. the browsing of a random sequence of states $Q = \{q_1, \cdots q_K\}$) and of a draw (called an *emission*) at each visit of a state.
  The sequence of states, which is the quantity of interest in speech recognition and in most of the other pattern recognition problems, can be observed only *through* the stochastic processes defined into each state (i.e. you must know the parameters of the pdfs of each state before being able to associate a sequence of states $Q = \{q_1, \cdots q_K\}$ to a sequence of observations $X = \{x_1, \cdots x_K\}$). The true sequence of states is therefore *hidden* by a first layer of stochastic processes.
  HMMs are *dynamic models*, in the sense that they are specifically designed to account for some macroscopic structure of the random sequences.

In our problem, given a new observation sequence and a set of models, we want to explore which model explains best the sequence, or in other terms which model gives the highest likelihood to the data so that to extract corresponding features for the entry. To solve this problem, it is necessary to compute $p(X|\Theta)$, i.e. the likelihood of an observation sequence given a known HMM model.

2) **Likelihood of a sequence given a HMM:** I will derive the likelihood of a sequence given a HMM model by the following steps:

- *Probability of a state sequence* : the probability of a state sequence $Q = \{q_1, \cdots, q_T\}$ coming from a HMM with parameters Θ corresponds to the product of the transition probabilities from one state to the following :
$$P(Q|\Theta) = \prod_{t=1}^{T-1} a_{t,t+1} = a_{1,2} \cdot a_{2,3} \cdots a_{T-1,T}$$
where $a_{t,t+1}$ is the transition probability from state q_t to state q_{t+1}.
- *Likelihood of an observation sequence given a state sequence*, or *likelihood of an observation sequence along a single path* : given an observation sequence $X = \{x_1, x_2, \cdots, x_T\}$ and a state sequence $Q = \{q_1, \cdots, q_T\}$ (of the same length) determined from a HMM with parameters Θ, the likelihood of X along the path Q is equal to :
$$p(X|Q, \Theta) = \prod_{i=1}^{T} p(x_i|q_i, \Theta) = b_1(x_1) \cdot b_2(x_2) \cdots b_T(x_T)$$
i.e. it is the product of the emission probabilities computed along the considered path. Where $b_i(x_i)$ is the emission probability of x_i at state q_i under the model Θ.
- *Joint likelihood of an observation sequence X and a path Q* : it consists in the probability that X and Q occur simultaneously, $p(X, Q|\Theta)$, and decomposes into a product of the two quantities defined previously :
$$p(X, Q|\Theta) = p(X|Q, \Theta)P(Q|\Theta) \qquad \text{(Bayes)}$$
- *Likelihood of a sequence with respect to a HMM* : the likelihood of an observation sequence $X = \{x_1, x_2, \cdots, x_T\}$ with respect to a Hidden Markov Model with parameters Θ expands as follows :
$$p(X|\Theta) = \sum_{\text{every possible } Q} p(X, Q|\Theta)$$
i.e. it is the sum of the joint likelihoods of the sequence over all possible state sequence allowed by the model.

3) HMM sequence classification: Also called Maximum Likelihood classification. In practice, it is very often assumed that all the model priors are equal (i.e. that the new route to be predicted have equal probabilities of having same pattern in the observed 8 specific routes). Hence, this task consists mostly in performing the Maximum Likelihood classification of feature sequences. For that purpose, we must have of a set of HMMs that model the feature sequences. These models can be considered as **"stochastic templates"**. Then, we associate a new sequence to the most likely generative model. This part is called the *decoding* of the feature sequences.

4) **Use HMM to solve our problem:** We defined an **equivalence sequence** over different routes. Our equivalence sequence is the set of states with the same departure date, the same days before takeoff(i.e. current date or current entry to predict), and the same first observed date. Fig 19 shows how these 8 stochastic templates can be used to predict and intuitive meaning of **equivalence sequence**. Our goal is to allocate one pattern to each entry.

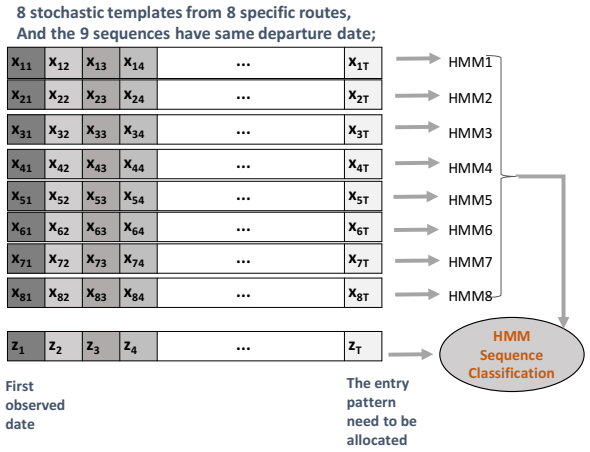

Fig. 19. How to use 8 stochastic templates in HMM sequence classification.

In our case, as you know, we already have 8 patterns from the 8 specific routes. When we have to find out which pattern should be classified to the entries for a new route, we do a HMM Sequence Classification on the new sequence. In other word, we trained the 8 referenced sequence by HMM to get each HMM model parameters Θ_i(i=1, 2, ..., 8), i.e. getting the transition matrix and emitting probability of each model. Then we compared it to the new sequence to get the *likelihood of the new sequence with respect to each HMM model*(i.e. $p(X|\Theta_i)$). Finally, allocate the pattern to the entry which makes it have the largest probability. This model is widely used in Automatic Speech Processing. To be more concrete, if an data entry is allocated to the second pattern of the 8 patterns. Then the dummy variable of flight number allocated to this entry should be: $F = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$. And the final feature allocated should be:

$$\begin{bmatrix} F \\ \textit{minimum price so far} \\ \textit{maximum price so far} \\ \textit{query-to-departure} \\ \textit{days-to-departure} \\ \textit{current price} \end{bmatrix}$$

We did not put much effort to explore different model types. However, the following model type is quite useful:

Parameter	Type
Number of states	2
Feature used	price feature

TABLE VII
Model type used in HMM.

C. Generalized Problem Performance Result

Table VIII shows the result of generalized problem. As we can see, the uniform blending does not get any improvement. But the HMM Sequence Classification algorithm makes 9 routes get improvement, 3 routes have negative performance. Although the average performance is 31.71%, which is lower than that of the specific problem, it makes sense that we did not use any historical data of these routes to predict (actually, there are two routes already appear in the specific problem, which are CRL→OTP and OTP→CRL). In specific problem, when using the AdaBoost-DecisionTree Classification, the performances for these two routes(i.e. CRL→OTP and OTP→CRL) are 71.69% and 85.33% respectively. However, in generalized problem, using same classification method, the performances are 63.11% and 0.54%, which has poorer performance than the specific problem. This is tolerable because we only used the formula trained in specific problem to predict.

Performance(%)	Model			
Routes \ Method	Optimal	Random Purch.	Uniform	HMM
BGY→OTP	100.0	0.00	80.52	87.09
BUD→VKO	100.0	0.00	-75.1	-48.6
CRL→OTP	100.0	0.00	51.06	63.11
CRL→WAW	100.0	0.00	-16.57	17.32
LTN→OTP	100.0	0.00	9.47	53.22
LTN→PRG	100.0	0.00	21.14	45.83
OTP→BGY	100.0	0.00	24.95	62.33
OTP→CRL	100.0	0.00	-56.99	0.54
OTP→LTN	100.0	0.00	-105.07	-84.45
PRG→LTN	100.0	0.00	-0.18	38.16
VKO→BUD	100.0	0.00	-48.61	-15.97
WAW→CRL	100.0	0.00	-3.3	35.12
Mean Performance	100.0	0.00	-14.84	31.71
Variance	0.00	0.00	2637.84	2313.25

TABLE VIII
Generalized Model Performance.

X. Future Work

In this project, we only used several features: *flight number*, *minimum price so far*, *maximum price so far*, *query-to-departure*, *days-to-departure*, and *current price*. However, we thought that some other features can be taken into consideration as well. For example, whether it is a

national holiday or not, whether it is weekend or not. We just put efforts in how feature extraction can be used in the field of airplane ticket prediction.

And as for the generalized problem, we only used two methods to predict for generalized routes. In the future, we may find more algorithms to see how can we extend the ticket prediction to generalized routes. Because this model may have many benefits, such as reducing computation time.

And another situation we need to mention is that the observed data is over a 103 day period. We may guess that the airplane companies must vary their ticket price algorithm over time to time to get highest profits. So our model need to be time-variant as well. In other sense, we cannot use all the historical datas to predict the current price. But what is the time threshold is another aspect of the project.

XI. Conclusion

In this project, we used the airplane ticket datas over a 103 day period for 8 routes to perform out models. **Removing outlier** through K-Means Algorithm and EM Algorithm implied that our training algorithms were not influenced by non-representative class members; **tackling the fact that our dataset was imbalanced**, through Random Over Sampling, meant our algorithms were not biased towards the majority classes. For the classification and regression methods, the best values for hyperparameters were found through 5-fold grid search.

As shown by the results, for the **specific problem** and from the aspect of performance, **AdaBoost-Decision Tree Classification** is suggested to be the best model, which has **61.35**% better performance over random purchase strategy and has relatively small performance variance for the 8 different routes. From the aspect of performance variance for different routes, **Uniform Blending Classification** is chosen as the best model with relatively high performance. On the other hand, the Q-Learning method got a relatively high performance as well. Compare the validation curve of classification methods to that of regression methods, we could find that the cross validation error or precision in classification has far smaller variance than that of regression (the shadow parts of the figures are the variances of the cross validation error or precision for all the 5 folds). We then considered that the classification model construction is more suitable in this problem.

Under the preferred AdaBoost-Decision Tree Classification, if the person wants to buy a ticket, its better to use our models and predictions because if he is using he would save 15.28€ compared to the *Random Purchase Strategy* for every client on the average.

For the **generalized problem**(i.e. predict without the historical data of routes that we want to predict), we did not test many models. However, the **HMM Sequence Classification based AdaBoost-Decision Tree Classification** model got a good performance over 12 new routes, which has **31.71**% better performance than the random purchase strategy.

Additionally, this project may contribute to other price strategy problems, for example, hotel and traveling agent price predicting. This project is just a beginning of the price strategy problems. We think that it will be interesting to see the correlation of different price strategy problems, in other sense, the hotel price predicting and hotel price predicting are very correlated.

Acknowledgment

I would like to thank Professor Boi and Igor, for offering their help and suggestions during this project. All rights of this report and published codes are reserved.

References

[1] O. Etzioni, R. Tuchinda, C. A. Knoblock, and A. Yates, *To buy or not to buy: mining airfare data to minimize ticket purchase price*, in Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2003, pp. 119128.

[2] Chawla, Sanjay, and Aristides Gionis. *k-means-: A Unified Approach to Clustering and Outlier Detection*, SDM. 2013

[3] Liu, Wei, Gang Hua, and John R. Smith, *Unsupervised one-class learning for automatic outlier removal*, Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on. IEEE, 2014.

[4] William Groves, and Maria Gini, *Optimal Airline Ticket Purchasing Using Automated User-Guided Feature Selection*, IJCAI. 2013.

[5] Class Imbalance Problem: http://www.chioka.in/class-imbalance-problem/

[6] Stewart, G. W. *On the perturbation of pseudo-inverses, projections and linear least squares problems.* SIAM review 19.4 (1977): 634-662.

[7] Saul, Lawrence K., and Mazin G. Rahim. *Maximum likelihood and minimum classification error factor analysis for automatic speech recognition.* Speech and Audio Processing, IEEE Transactions on 8.2 (2000): 115-125.

[8] Gopinath, Ramesh A. *Maximum likelihood modeling with Gaussian distributions for classification.* Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on. Vol. 2. IEEE, 1998.

[9] Watkins, Christopher JCH, and Peter Dayan. *Q-learning.* Machine learning 8.3-4 (1992): 279-292.

[10] Pedregosa, Fabian, et al. *Scikit-learn: Machine learning in Python.* The Journal of Machine Learning Research 12 (2011): 2825-2830.