Google Store Revenue Prediction

A comparison of several Regression models to find the optimal algorithm

Anisa Zhurda*
Rabbir Bin Rabbani*
a.zhurda@stud.uis.no
rb.rabbani@stud.uis.no
University of Stavanger, Norway

ABSTRACT

As the online marketplace is thriving nowadays, predicting the revenue from customers is a wise approach for any company so that they have a better understanding of the customer behaviour and plan a more efficient use of the marketing budget.

This project aims to find the best algorithm among several Machine Learning algorithms to better understand which method is best suited to this kind of dataset.

KEYWORDS

Regression, SKlearn, Tensorflow, Hyperparameter Tuning

In *Proceedings of* . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

According to the 80/20 rule, only a small percentage of customers produce most of the revenue. It is in best interest for marketing teams to make appropriate investments in promotional strategies for the target audience. To survive in the face of competition marketplaces need to adapt their marketing strategies and budget to meet the demands of their paying customers.

The massive amount of data generated collectively by the Google Store customers, or E-commerce users for that matter, can be used to predict the potential revenue from each customer in upcoming months. This incentivizes the use of several prediction models in order to get the one that minimizes the error and makes the most accurate predictions possible, at the expense of explainability .

The models we are developing learn from existing transaction information in Google Store and provide a prediction of how many dollars a costumer is expected to spend in a given time period. This,of course, is a typical supervised regression problem.

In this project we accomplish the following:

- We preproccess the data collected from Kaggle
- We demonstrate the implementation of several regression algorithms such as: Linear regression, Decision tree regression, Polynomial regression, Random Forest regression, Perceptron regression,Neural Network.
- We compare the performance by using learning curves and cross validation for each of the implementations

2 BACKGROUND

This section provides the necessary background needed for visualizing the marketing strategies by using machine learning techniques in advanced distributed systems to predict customer behavior.

2.1 Machine Learning and Predictive Analytics

Basically, machine learning is the ability of a computer to learn from experience (Mitchell, 1997). Experience is usually given in the form of input data and the computer can find dependencies in the data to make predictions that are too complex for a human ,which is also the main focus of the paper.

Predictive analytics is the act of predicting future events and behaviors present in previously unseen data, using a model built from similar past data (Nyce, 2007; Shmueli, 2011). It has a wide range of applications in different fields,including marketing which is our field of interest.However,the method is similar in all of them and it consists in using previously collected data and a machine learning algorithm to find the relations between different features of the data. The resulting model is then able to make a prediction on one of the features of future data.

2.2 Selected Regression Methods

In this paper we use 6 different Regression models and all essentially have the same task,which is predicting the revenue from customers in Google Store,but the difference is that each of them are based on different mathematical methods of computation.

2.2.1 Linear Regression. Linear Regression is a linear approach to modelling the relationship between dependent and independent variables and it creates a continuous function generalizing these relations.

The goal is to find the line, plane or hyperplane that best represents the dataset, considering that it does not have to cover the exact points in the previous dotted graph, since it is just making an approximation.

2.2.2 Polynomial Regression. Considered to be a special case of multiple linear regression,polynomial regression is a form of regression analysis in which the relationship between the independent

*Both authors contributed equally to this research.

, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

variable and the dependent variable is modelled as an nth degree polynomial in the independent variable.

2.2.3 Decision Tree Regressor. A decision tree splits the input features in several regions and assigns a prediction value to each region. The selection of the regions and the predicted value within a region are chosen in order to produce the prediction which best fits the data,that means minimizing the error of the prediction.

2.2.4 Random Forest Regressor. Random forest builds multiple decision trees and merges them together to get a more accurate and stable prediction.Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model.

2.2.5 Neural Network. The simplest definition of a neural network is provided by the inventor of one of the first neurocomputers, Dr. Robert Hecht-Nielsen. He defines a neural network as: "...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs."

Although neural networks are widely known for use in deep learning and modeling complex problems such as image recognition, they are easily adapted to regression problems.

2.2.6 Perceptron. Perceptron is a single layer neural network.In a similar way with a neuron, the Perceptron receives input signals from samples of training data that we weight and combined ,in a linear equation called the activation. By default SKLearn uses RELU activation function which is linear for all positive values, and zero for all negative values.

2.3 Evaluation Methods

2.3.1 Grid Search. In machine learning, a hyperparameter is a parameter whose value is set before the learning process begins. Grid search is the process of performing hyper parameter tuning in order to determine the optimal values for a given model.

2.3.2 Learning Curve. Graph that compares the performance of a model on training and testing data over a varying number of training instances.The key is minimizing bias and variance by finding the right level of model complexity.

2.3.3 Cross Validation Curve . A method to validate the stability of our machine learning models ,so we can understand how well "the learner" will generalize to an independent/ unseen data set.

2.4 Dataset

The dataset was originally used in a Kaggle competition for predictive analysis (Linked Here). The data is provided by Google in cooperation with RStudio as CSV files and is already separated into training and test sets. The training set is 23.67 GB and contains over 1.7 million rows, whereas the test set is 7.09 GB and contains a little over 400,000 rows. Each row represents a single visit to the google store.

3 PREPROCESSING

The first step in preprocessing data is to analyse the data and check its quality. Visualisation can assist a user in verification for existence of missing values, errors, and outliers. Since improving the accuracy of a prediction is very important for any machine learning task, we are reducing the number and complexity of features to get stability.

3.1 Loading the data

As mentioned above, our initial training dataset is about 24GB and the test dataset is 7GB, on the other hand we only had 8GB of RAM to handle these relatively large datasets. Fortunately for us, using the library 'pandas' in python, we can load this data in chunks.

```
import pandas as pd
train_chunks=pd.read_csv("train.csv", chunksize = 20000)
test_chunks=pd.read_csv("test.csv", chunksize = 20000)
```

Listing 1: Loading the dataset in chunks

This gives us a generator that, when looped, will give us the dataset in chunks of 20000 rows. We can use the first chunk from this to get an initial idea of what our dataset is like.

3.2 Initial analysis of the data

The first thing we noticed when seeing this data is that we have several columns with JSON data of varying sizes and depth, namely the columns 'customDimensions', 'hits', 'device', 'geoNetwork', 'trafficSource' and 'totals'.


```
{
    "browser": "Firefox",
    "browserVersion": "not available in demo dataset",
    "browserSize": "not available in demo dataset",
    "operatingSystem": "Windows",
    "operatingSystemVersion": "not available in demo dataset",
    "isMobile": false,
    "flashVersion": "not available in demo dataset",
    "language": "not available in demo dataset",
    "screenColors": "not available in demo dataset",
    "deviceCategory": "desktop"
}
```


Figure 1: A sample value of the 'device' column.

Looking deeper into these JSON columns we can also see that a lot of the data in these has been censored by google before providing this data and has the value "Not available in demo dataset". Two of these JSON columns, 'hits' and 'customDimensions' are completely censored. There is also a 'visitId' column that acts as the index of the dataset. We removed these columns before starting the preprocessing.

3.3 Initial preprocessing

It was mentioned in the Kaggle competition overview, that our target variable, 'totalTransactionRevenue', is inside the 'totals' JSON column. So we have to flatten these JSON values and add them to our dataset to get value out of them.

```
import pandas as pd
import json

json_columns = ['device','geoNetwork','totals','
    trafficSource']
dataset = pd.DataFrame()
for chunk in chunks:
    chunk.drop(columns = ['customDimensions','hits','
     visitId'], inplace=True)
    for col in json_columns:
        json_to_df = chunk[col].map(json.loads)
                               .apply(pd.Series)
        chunk = pd.concat([chunk, json_to_df], axis=1)

    chunk.drop(columns = json_columns, inplace=True)
    dataset = pd.concat([dataset, chunk], sort=True)
```

Listing 2: Flattening JSON and adding them to chunks

The code above does the following:

- Drops the columns 'hit' and 'customDimensions' which were censored and the 'visitId' column which acts as the index.
- For every chunk, creates new dataframes with the flattened JSON.
- Concatenates those columns to the chunk then removes the original JSON columns.
- Concatenates each of these to make a new dataframe.

We put this in its own separate function, so it is easier for us to perform these tasks for both the training and test set.

It should also be mentioned that we deal with a few exceptional cases which we found with trial and error, we have handled them accordingly and so, the code above represents the idea of what we are trying to do here, not the final code.

This is, by far, the most time consuming process in the preprocessing of our implementation. For the training set it takes 1 hour and 30 minutes and for the test set it takes a little over 16 minutes.

By the end of this process, we have made 2 new datasets where the JSON columns are flattened, this results in the datasets having a total of 57 columns.

3.4 Further analysis and managing empty columns

With these new datasets we combined them into a single dataset and analysed the different data available in each of the columns. We found that there are many values that can be considered to be empty values which are in the dataset and considered as values even though they will provide no insight to the prediction, we ultimately decided to replace these values with the nan value provided in the numpy package. The values we have been able to discover are - "not available in demo dataset", "unknown.unknown", "nan", "(none)", "(not set)", "/", "", "None".

We also noticed two more special cases -

- Most of our target value is empty so we filled them with zeroes since an empty transaction amount means that there was no transactions for that visit.
- The date column up until now was being considered categorical data. Since time is continuous and this dataset is daily visit data, we decided to convert the dates to number of days since the epoch. This turns the date column into a continuous field.

Then we removed all columns that are more than 50% of empty.

3.5 Data Imputation and Encoding

After all the previous works are done, we are left with 20 columns in the dataset including our target column. But we still have some columns with empty values. Since there was a very small percentage of empty data, we decided to do a simple imputation of the data and replace all empty values in numeric columns with the mean and replace all empty values in categorical columns with the most frequent value, in other words, the mode.

To make it simpler to compute during training, we decided to encode all the categorical columns in the datasets. We used SKlearns label encoder to encode each of the columns in our datasets.

We then save the training set, the test set and also the combined dataset to a folder so we can use them during the comparison.

4 HYPERPARAMETER TUNING WITH GRID SEARCH

In this section, we discuss how we used Gridsearch to find the best hyperparameter combination for each of the models. To do this part we decided to keep several things constant to make this a fair comparison across all the models and their parameters. They are as follows -

- All of them use negative mean square error for scoring during the GridSearch.
- All of them use 5-fold crossvalidation for comparison.
- If any model has some randomization to them, SKlearn provides a hyperparameter called 'random_state' to initialize the random seed. All such models use 'random_state = 0'.

For each model we first set several values for the selected Hyperparameters and do an exhaustive comparison using Gridsearch.

4.1 Linear Regression

For linear regression we only had 2 hyperparameters to use and both were boolean.

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import GridSearchCV

parameters = {
    'fit_intercept': [True, False],
    'normalize': [True, False]
}

gridSearchResult = GridSearchCV(LinearRegression(),
                    parameters,
                    cv = 5,
                    scoring='neg_mean_squared_error',
                    verbose=5
                   ).fit(all_data_X, all_data_y)
```

Listing 3: Grid search for Linear Regression

This takes about 30 seconds to compute. The results are given below:

fit_intercept	True
normalize	False
mean_fit_time(s)	1.006
std_fit_time(s)	0.0248439
mean_train_score	9.41896e+14
std_train_score	2.14111e+14
mean_test_score	1.01523e+15
std_test_score	8.71677e+14

4.2 Polynomial Regression

Polynomial Regression is similar to linear regression, we 2 hyperparameters to use.

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

from sklearn.model_selection import GridSearchCV

parameters = {
    'fit_intercept': [True, False],
    'normalize': [True, False]
}

poly = PolynomialFeatures(degree=2)
transformed_X = poly.fit_transform(all_data_X)

gridSearchResult = GridSearchCV(LinearRegression(),
                    parameters,
                    cv = 5,
                    scoring='neg_mean_squared_error',
                    verbose=5
                   ).fit(transformed_X, all_data_y)
```

Listing 4: Grid search for Linear Regression

This takes around 2 hours to compute. The results are given below:

fit_intercept	True
normalize	False
mean_fit_time(s)	202.249
std_fit_time(s)	30.5
mean_train_score	5.99278e+14
std_train_score	1.50912e+14
mean_test_score	8.41857e+14
std_test_score	6.85371e+14

4.3 Decision Tree Regressor

For Decision tree we chose 2 hyperparameters. The max depth and the minimum number of values a sample must have to split and make a subtree.

```
from sklearn.linear_model import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV

parameters = {
    'max_depth': [depth for depth in range(2, 20, 2)],
    'min_samples_split': [min_samples for min_samples in
     range(200, 2001, 200)]
}

gridSearchResult = GridSearchCV(
                    DecisionTreeRegressor(random_state =
    0),
                    parameters,
                    cv = 5,
                    scoring='neg_mean_squared_error',
                    verbose=5
                   ).fit(all_data_X, all_data_y)
```

Listing 5: Grid search for Decision Tree

This takes about a little over 1 hour to compute. The results are given as follows:

max_depth	6
min_samples_split	200
mean_fit_time(s)	7.746
std_fit_time(s)	0.139
mean_train_score	1.238378e+15
std_train_score	3.143901e+14
mean_test_score	1.613924e+15
std_test_score	8.7949090e+14

4.4 Random Forest Regressor

For Random Forest we chose 3 hyperparameters. The max depth, the minimum number of values a sample must have to make a subtree and also the number of estimators.

```
from sklearn.linear_model import RandomForestRegressor
from sklearn.model_selection import GridSearchCV

parameters = {
    'n_estimators': [est for est in range(2, 11, 2)],
    'max_depth': [depth for depth in range(2, 20, 2)],
    'min_samples_split': [min_samples for min_samples in
     range(200, 2001, 200)]
}

gridSearchResult = GridSearchCV(
                    RandomForestRegressor(random_state =
    0),
                    parameters,
                    cv = 5,
                    scoring='neg_mean_squared_error',
                    verbose=5
                   ).fit(all_data_X, all_data_y)
```

Listing 6: Grid search for Random Forest

max_depth	8
min_samples_split	400
n_estimators	10
mean_fit_time(s)	60.134
std_fit_time(s)	0.932
mean_train_score	9.652872e+14
std_train_score	1.734750e+14
mean_test_score	1.299687e+15
std_test_score	9.934932e+14

4.5 Perceptron

For Perceptron we chose 3 hyperparameters. The hidden layer sizes which accepts tuples with a number of values, where each value represents the number of nodes in that hidden layer, the weight optimization algorithm, and the learning rate.

```
from sklearn.linear_model import MLPRegressor
from sklearn.model_selection import GridSearchCV

parameters = {
```

```
    'hidden_layer_sizes': [(10,), (14, 7), (14, 7, 3)],
    'solver': ['adam', 'sgd'],
    'learning_rate' : ['constant', 'invscaling', '
     adaptive'],
}

gridSearchResult = GridSearchCV(
                    MLPRegressor(random_state = 0),
                    parameters,
                    cv = 5,
                    scoring='neg_mean_squared_error',
                    verbose=5
                   ).fit(all_data_X, all_data_y)
```

Listing 7: Grid search for Perceptron

This takes almost 5 hours and 40 minutes to compute and gives us 3 estimators with rank 1. The results are given below:

hidden_layer_sizes	(10,)	(10,)	(10,)
learning_rate	constant	invscaling	adaptive
solver	adam	adam	adam
mean_fit_time(s)	171.473	171.908	170.877
std_fit_time(s)	57.978	57.237	59.204
mean_train_score	9.37463e+14	9.37463e+14	9.37463e+14
std_train_score	2.06898e+14	2.06898e+14	2.06898e+14
mean_test_score	9.97678e+14	9.97678e+14	9.97678e+14
std_test_score	8.52126e+14	8.52126e+14	8.52126e+14

Notice that the only difference is the choice of the learning rate parameter. Since all of the scores and the fit times are almost the same, it can be assumed that any of the three parameter combinations will be optimal for our model.

4.6 Neural Network

For Neural Network we chose 3 hyperparameters. The hidden layer size and the activation function of each layer coupled, the number of epochs, and the optimizer.

We made our Neural network in tensorflow, and of course, it isn't compatible with the gridsearch function provided in SKlearn, so we had to make our own gridsearch code.

```
import tensorflow as tf

activation_functions = [None, tf.nn.relu, tf.nn.
    leaky_relu, tf.nn.log_softmax]
hidden_layer_sizes = [(10,), (14, 7), (14, 7, 3)]
epochs = [2, 5, 10]
optimizers = ['sgd', 'adam', 'adadelta']

parameters = []
for e in epochs:
    for o in optimizers:
        for hidden in hidden_layer_sizes:
            for f in activation_functions:
                h = [0]*len(hidden)
                for i in range(len(hidden)):
                    h[i] = (hidden[i], f)

            parameters.append((h, e, o))
```

Listing 8: Getting Parameter Combinations for Neural Network

The code above shows the different parameters and their respective values we wanted to check. In total we had 27 combinations of parameters and with our 5-fold cross-validation, we have a total of 135 iterations. In total the process takes about 27 hours. We decided to take the outputs with the lowest mean test score. The code for the gridsearch is too big to be in this report. But the result is given in the table below:

Hidden layer sizes	(10,)	(14, 7)
Activation functions	(RELU,)	(RELU, RELU)
Epochs	5	5
Optimizer	adam	adam
mean_fit_time(s)	52.096	71.336
std_fit_time(s)	1.540	1.406
mean_train_score	4.68157e+15	4.68157e+15
std_train_score	6.24669e+14	6.24669e+14
mean_test_score	4.68157e+15	4.68157e+15
std_test_score	2.49868e+15	2.49868e+15

Notice that they both have very identical test and training scores and since the 1st model with 1 hidden layer gives us a lower training time, we chose to use this.

5 CROSS VALIDATION

In this section, we take all of the best models we found in the previous section and compare how well they learn during training using the Learning Curve and also how well the data is able to predict using the cross-validation curve.

Just like before, we had decided to keep some variables constant to make this a fair comparison across all the models. They are as follows:

- All of them use negative mean square error for scoring during the GridSearch.
- All of them use 5-fold crossvalidation.

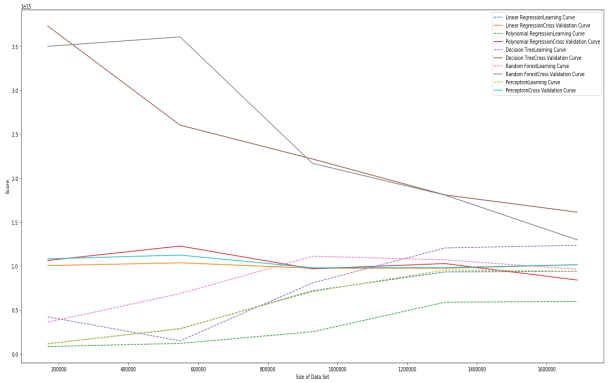

Figure 2: Learning curves and cross validation curves

As we can see from the result,the best model for our dataset in terms of error in learning and predicting is Polynomial Regressor. In both cases the error is increasing by a very low rate as the dataset capacity increases. However, the overall error in both learning curve and cross validation curve is shown to be noticeably smaller than in the other models. Although, judging by the performance in time, Polynomial Regression is the most time consuming model and definitely not an optimal choice.

If our goal is to use a simple regression model, that is not time demanding and shows a surprisingly good error rate then we should choose Linear Regression .

Both Decision Tree and Random Forest show a significant drop of error rate in predicting as the dataset capacity gets larger. We can imply that they are best used for large amount of data. Moreover,their learning errors are nearly the same as linear regression and they are both time-efficient.

6 CONCLUSION

The success of machine learning in predicting customer's revenue relies on the good use of the data and machine learning algorithms. Selecting the right machine learning method for the right problem is necessary to achieve the best results. However, the algorithm alone can not provide the best prediction results. Tuning hyperparameters and modifying data for machine learning, is also an important factor.

The aim of this project was to compare method selection by their ability to improve the prediction results in terms of error and time. Our dataset was preprocessed and analyzed with six different machine learning methods tuned in the right parameters by Grid Search to get the optimal version for each of them. Then using learning and cross validation curves we evaluated their performance in both learning and predicting. The results show that choosing a model depends on the user requirements and dataset characteristics. Finally, we have uploaded our codes to a GitHub repository (Linked Here).