Kaggle ML Challenge 1

2023-02-03

Introduction

The Getting Started Notebook was a great help for us to jump-start our Kaggle challenge. Similar to the notebook, our team devised a simple K-Nearest Neighbor(KNN) algorithm to solve the challenge. KNN is a supervised classification algorithm and a more appropriate tool for building the recommendation engine.

For the initial implementation, we built a recommendation system with 1-Nearest Neighbor using `python` as below:

```python
# Import data
training_set = pd.read_csv("data/training_set.csv", index_col=0)
test_set = pd.read_csv("data/test_set.csv", index_col=0)

def generate_recommendations(train: pd.DataFrame, test: pd.Series) -> list:
    distances = [
        (lambda x1,x2: sum([abs(item[0] - item[1]) for item in zip(x1, x2)]) )(val, test)
        for col, val in train.items()
    ]

    training_nn = train.iloc[:, distances.index(min(distances))].copy()
    training_nn.loc[test != 0] = 0

    recommendations = [0] * len(test)
    for item in list((-training_nn).argsort() )[0:5]:
        recommendations[item] = 1

    return recommendations

recommendations = [
    generate_recommendations(train=training_set.copy(), test=test_set[column])
    for column in test_set
]
```

However, the above implementation took approximately 2 hours to complete execution and sometimes more. To boost the run time, we did several optimizations, such as using the Pandas `apply()`, `itertuples()`, and vectorization method, but the code was still running significantly slower compared to a similar R solution. We found that loop operations in Panda's dataframe and the `apply` method are slower than R. So, we started from scratch on R as R's `dataframe` and `apply` function was much more efficient than running loops in Python. We also want to rewrite our Python code so that we can benchmark similar to R performance in the near future.

Import dataset

Firstly, we read the training and test csv file and converted them into matrix using the `as.matrix` function.

```r
training_set <- as.matrix(read.csv("training_set.csv", row.names = 1))
test_set <- as.matrix(read.csv("test_set.csv", row.names = 1))
```

Normalization

We then worked on normalizing our dataset before calculating the distance between the samples. We first tried log normalization, which didn't improve the error rate of **0.11244**. We also tried implementing z-score normalization. However, that also didn't help us improve our error rate. Lastly, we tried min-max log normalization, which improved our score.

Before implementing the min-max log normalization for each column, we calculated the training and test set's minimum and maximum values of each column using the apply function. We then performed min-max log normalization for each column. Later, we refactored this operation into a single reusable function

```r
# Function to calculate the min-max log normalization
normalize_data <- function(data) {
  return ((log(data + 1) - min(data)) / (max(data) - min(data)))
}

# Function to calculate the z transfrom normalization
normalize_z_transform <- function(data) {
  return ((log(data + 1) - mean(data)) / sd(data))
}

# Normalize the training data
normal_training_set <- apply(training_set, 2, normalize_data)

# Normalize the test data
normal_test_set <- apply(test_set, 2, normalize_data)
```

Distance measurement

For calculating the distance between two vectors, we first used the Euclidean distance, through which we got **a 0.6513** score. We then used the Manhattan Distance to calculate the distance between two vectors whose score was comparatively better than Euclidean Distance. We also tried cosine distance, but our overall score was reduced. We found that **the Manhattan distance** works much better than the Euclidean and cosine distances for the given dataset.

Below are the distance function we created:

```r
# Function to calculate the Manhattan distance between two vectors x, and y
get_manhatten_distance <- function(x,y) {
  return(sum(abs(x-y)))
}

# Functions to calculate the Euclidean distance between the two vectors x, and y
get_eculidean_distance <- function(x, y) {
  return(sqrt(sum((x-y)^2)))
}

# Function to calculate the cosine distance between the two vectors x, and y
get_cosine_distance <- function(x, y) {
  return(sum(x * y) / (sqrt(sum(x)) * sqrt(sum(y))))
}
```

Approach

We used the K-nearest neighbor for the recommendation because it recommends looking at similar characteristics using the distance formula. The K-means initially chooses the centroid of the clusters randomly, which might not provide accurate information than the KNN.

Our get_recommendations function performs the Manhattan distance between each column of the test dataset with the training dataset and goes on for the loop until it provides a matrix of **6009 * 14832**. After the Manhattan distances are calculated, we identify k nearest neighbors by selecting the top k neighbors with the least distance from our test set instance. After getting K neighbors, we performed a row-wise sum of neighbors and generated the top five recommendations based on the order of the neighbor column values. We ran the recommendations generation through a different number of k that helped to increase our score significantly.

```r
# Function to generate top 5 recommendations for the given test column
get_recommendations <- function(train, test_col, k = 3) {
  # Calculate Manhattan Distance for each training column from test column
  manhatten_distances <-
    apply(train, 2, function(x)
      get_manhatten_distance(x, test_col))

  # Get top K nearest neighbors
  k_neighbor <-
    train[, which(manhatten_distances %in% sort(manhatten_distances)[1:k])]

  # Sum K neighbors into single column
  neighbor <- rowSums(k_neighbor)

  # Don't recommend any values which are non-zero in the test sample
  neighbor[which(test_col != 0)] <- 0

  # Generate recommendations
  res <- rep(0, length(test_col))

  # Get top 5 recommendations
  res[order(neighbor, decreasing = TRUE)[1:5]] <- 1

  return(res)
}

# Generate recommendations for each column
recommendations <-
  pbapply::pbapply(normal_test_set, 2, function(x)
    get_recommendations(normal_training_set, x, 77))
```

To verify that we are submitting exactly 5 recommendations per column for each of the 6009 columns in the test set:

```
table(colSums(recommendations != 0))
```

```
##
##    5
## 6009
```

We can also verify that all values are either 0 or 1 through the below code:

```
table(recommendations)
```

```
## recommendations
##      0      1
## 480720  30045
```

Submitting our recommendations

To convert the result from the wide to tall format, we perform below code.

```
rownames(recommendations) <- rownames(test_set)
submission <- reshape2::melt(recommendations)
head(submission)
```

```
##                  Var1  Var2 value
## 1             Apapane test1     0
## 2      Hawaii_Elepaio test1     0
## 3      Kalij_Pheasant test1     0
## 4   Northern_Cardinal test1     0
## 5                Omao test1     0
## 6 Warbling_White_eye test1     1
```

```
sample_submission <- read.csv("sample_submission.csv", row.names = 1)
submission_ids <- read.csv("submission_ids.csv", row.names = 1)
head(sample_submission)
```

```
##      Id Expected
## 1 row_1        0
## 2 row_2        1
## 3 row_3        0
## 4 row_4        0
## 5 row_5        0
## 6 row_6        0
```

```
head(submission_ids)
```

```
##      Id           row_name col_name
## 1 row_1            Apapane    test1
## 2 row_2     Hawaii_Elepaio    test1
## 3 row_3     Kalij_Pheasant    test1
## 4 row_4  Northern_Cardinal    test1
## 5 row_5               Omao    test1
## 6 row_6 Warbling_White_eye    test1
```

We can verify that our results line up with `submission_ids`:

```r
all.equal(submission_ids$row_name, as.character(submission$Var1))
```

```
## [1] TRUE
```

```r
all.equal(submission_ids$col_name, as.character(submission$Var2))
```

```
## [1] TRUE
```

```r
sample_submission$Expected <- submission$value
```

To save this as a CSV file, we perform below code:

```r
write.csv(sample_submission, "my_submission.csv", row.names = FALSE)
```

Reference

- Zachary DeBruine(2023, January). ML Challenge #1: Getting Started. Version: 6 from https://www.kaggle.com/code/zdebruine/getting-started