Predicting Rideshare Prices with the Uber & Lyft Cab Prices Dataset

Krishna Kalakkad, Laura McGann, Ethan Waite, Ethan Zimbelman

Using the Uber & Lyft Cab prices | Kaggle dataset, we created a Scala program that predicts the price of a ride given conditions such as time of day, locations, and weather, using the K-nearest neighbors algorithm.

Source code for this program can be found on Github here.

Our Methodology

Preparing the Dataset

The data is split between two tables – rides and weather. The attributes of each data set are outlined below.

Cab rides

- Service (Uber or Lyft) and cab type
- Price and surge multiplier
- Source, destination, and distance
- Day of week, hour, half hour (1,2)

Weather

- Temperature, pressure, humidity, clouds, rain, and wind
- Location
- Day of week, hour, half hour (1,2)

Ride entries with missing values (e.g. rides that do not contain a price) were removed from our working dataset to ensure we are not evaluating incomplete data. Weather data with missing numerical values were given a value of 0.0, with the assumption that there was no measurement to record.

This data was joined on the day of the week (MONDAY - SUNDAY), the hour (0 - 23), which half of the hour the record was in (first (1) or second (2) half), and the location (only one location given for weather data; source location chosen for ride data since that may have more immediate effect on the rider's decision to call a ride).

The joined data is stored in a custom class object called LabeledRecord. A LabeledRecord contains the unique ride ID, the label (price), and another custom class object, Record. The Record stores all of the joined data itself - (distance, cabType, day, hour, destination, source, surgeMultiplier, rideName, temp, clouds, pressure, rain, humidity) - and is what is used in the distance calculations (discussed below). The LabeledRecord stores the wrapping information as

well so it is still associated with the data Record - for joining and label prediction and evaluation - but not included in the distance calculations.

With these records, we split the data 80/20 into training and testing subsets by way of random sample and then RDD subtraction.

Data Standardization

In order to standardize data, we need to compute the z-score for each value. The formula for z-score is $Z = \frac{x - \mu}{\sigma}$ where x is the current value, μ is the mean of all values, and σ is the standard deviation. Calculating the mean is pretty straight-forward, but to calculate standard deviation, we must square each value's deviation from the mean, sum up all the squares, divide that sum by the count - 1, and take the square root of that quotient. In other terms, the standard deviation $\sigma = \sqrt{\frac{\sum_{1}^{n}(x_i - \mu)^2}{n-1}}$.

KNN Predictions

The basic premise of KNN is that the label of a new data point is predicted based on the labels of its k nearest neighbors. "Nearness" is determined by creating a function to compute the "distance," or difference between, two data points. K is given as a parameter to the algorithm, determining how many of the nearest neighbors to look at and to include in the aggregation of their labels for prediction of the new data point's label. A k that is too small or too large can lead to inaccurate results due to failure to search a large enough neighborhood or due to overfitting.

Calculating Distance

The first step is to calculate the distance between each test point and every train point. This is done by calling the distance function on every combination (Cartesian product) of the train and test sets. Only the minimal amount of information is kept in this RDD: the test record's id (for a later join) and label (for later evaluation), the train record's label (for later prediction), and the computed distance between the two records.

The distance calculation itself must be split into two parts: distance between numerical attributes, and distance between categorical attributes.

1. For the numerical attributes, you can simply take the standard euclidean distance:
   $numerDist = \sqrt{sum\left((r1Numerical(i) - r2Numerical(i))^2\right)}$
   This calculation relies on the data being standardized so no attribute outweighs the others simply due to a larger scale or range of values.

2. For the categorical attributes, something known as the anti-dice distance can be found:
   $catDist = (total\ \#\ mismatches) / (total\ \#\ categorical\ attributes\ in\ one\ record)$
   This distance simply uses the binary question “Yes/no, does this categorical attribute value match for both records?” for each categorical attribute, then finds the proportion of categorical attributes that don’t match.
3. These two distances - for any two-record comparison - are merged by weighting each distance component by the proportion of attributes of that type (numerical vs categorical). For example:
   $dist = (\#\ numer) / (total\ \#\ attrbs) * numerDist + (\#\ cat) / (total\ \#\ attrbs) * catDist$

Finding the K-Nearest Neighbors

With all of the distances computed, we then determine which k data points in the training set are closest to each record in the testing set. We actually do this indirectly by first finding only the distance of the furthest neighbor within that k-nearest neighborhood. This “maxDist” is then used to join the test records on only train records whose distance from the train record is <= maxDist. The indirect method prevents the accumulation of nearest neighbors in a list, instead allowing for continued use of RDDs.

The end result of this step is k records for each initial test record. Each of these new joined records again contains only the minimal needed information for the next step: the test record’s id (for a later join) and label (for later evaluation), and the within-k-neighborhood train record’s label (for later prediction).

Predicting Price

The final KNN step is to actually aggregate the labels of the k-nearest neighbors to predict the label of the corresponding test record. Since the label for this dataset is a continuous number (price), the aggregation is simply the average of all neighbor values. We use a combineByKey() to perform this aggregation, since the data is structured such that the key is the id of the test record (this step is why the id was kept!). The output of this step is an RDD with records ((rTest_id, rTest_price), predicted_avgTrainPrice).

Prediction Evaluation

The main evaluation metric we can use is error between predicted and true labels (prices). Since we kept both the true test price in addition to calculating the predicted price, it is easy to simply map that RDD to a per-test-record error RDD - (rTest_id, error) records - which can then be aggregated to find the average error across the entire test set.

Results

This section shows both the prediction results themselves and their evaluation, as well as the results of some experiments done to optimize our code.

Hyperparameterization of the KNN Algorithm

Before analyzing the errors in our estimations, we can modify parameters of the KNN algorithm in an attempt to reduce the average error found. The parameters being modified include the train/test ratio and the K value, with results of these trials are outlined below.

	50% train/ 50% test	**60% train/ 40% test**	**70% train/ 30% test**	**80% train/ 20% test**	**90% train/ 10% test**
K = 5	$6.59	$6.24	$5.87	$5.82	**$5.37**
K = 10	$6.90	$6.65	$6.56	$6.69	$6.53
K = 50	$7.17	$6.93	$6.96	$7.01	$6.97
K = 100	$7.30	$7.06	$7.07	$7.15	$7.06

The average errors of different parameters of the KNN algorithm on the same dataset.

The largest training to testing ratio, coupled with the smallest amount of neighbors resulted in the lowest error rate in our model's prediction. Such a result seems reasonable, as training on more data means testing records are more likely to have a nearby neighbor (making predictions more accurate). Furthermore, using a small amount of neighbors ensures the prediction is not skewed by distant neighbors, neighbors that may exist if the data space is sparse.

Timing Improvements and Other Optimizations

Limiting the Search Space

Our initial dataset contains 619,531 records, which became a computational burden rather quickly. Computing the distance matrix requires a Cartesian product between the testing and training datasets, which (in the worst case, where train/data split is 50/50) requires 95,954,664,990 total comparisons. This "big data" also becomes problematic for large K values, as this increases the amount of comparisons required when finding nearby neighbors.

For the sake of completing computations in a timely manner, we reduced our working set to 1% of the records. Such a drastic reduction still requires intense computation, with around 10 million comparisons still being required to compute the distance matrix in the worst case. The

exponential impact of increasing the working set size on distance matrix computations (with an 80/20 split) is revealed in the chart below.

Amount of used records (%)	Time to compute distance matrix (ms)
(4956 train, 1239 test); 1% of all records	74,157ms
(9912 train, 2478 test); 2% of all records	296,577ms
(14868 train, 3717 test); 3% of all records	*ERROR (shuffle failure)*
(19824 train, 4957 test); 4% of all records	*ERROR (shuffle failure)*

Time to compute the distance matrix by record sizes.

As the amount of used records doubles, the time to compute the distance matrix increases four-fold. Such an increase is to be expected, as we are doubling the sizes of test and train data, then performing a cartesian product on these larger sets. The errors generated by trying to compute with 3% and 4% of the records was due to the ambari server's inability to handle this many records.

Scala optimizations

A direct implementation of the necessary K-nearest neighbor algorithms was used originally, however Scala provides built-in functionality that offer potential optimizations. Combinations of using the topByKey() method, partitions, and persits were tried on various steps of the KNN algorithm in an attempt to reduce run time. Results of these combinations are shown in the chart below.

Optimizations done	Calculate dists.	K-th max dist.	K-nearest dists.	Prediction	Total
topByKey(), partitioning (train, distMatrix, and kNearestNeighbors)	*76420ms*	*1232ms*	*10820ms*	*107ms*	*88.6s*
topByKey(), partitioning (train, kthDists, distMatrix, and kNearestNeighbors)	*75719ms*	*1306ms*	*11560ms*	*93ms*	*88.7s*
topByKey(), partitioning (train and distMatrix)	*73304ms*	*1189ms*	*3703ms*	*11549ms*	*88.8s*
topByKey(),	*81167ms*	*1297ms*	*11037ms*	*94ms*	*93.6s*

partitioning (kthDists, distMatrix, and kNearestNeighbors)					
topByKey(), partitioning distMatrix	*79308ms*	*1306ms*	*3763ms*	*10588ms*	*95.0s*
topByKey(), partitioning distMatrix, persisting test/train	*84908ms*	*1460ms*	*3813ms*	*11186ms*	*101.4s*
topByKey()	*296ms*	*79448ms*	*83526ms*	*6886ms*	*170.2s*
None	*295ms*	*79813ms*	*83149ms*	*11926ms*	*175.2s*

Optimization results using K=10, train record count=4956, test records=1239.

This chart shows that the speed of our KNN algorithm was reduced by almost 2x with only built in functions. The most notable reduction in computation time was found by introducing partitions to different RDDs. This reduction was expected, since we perform many key-based operations such as join() and reduceByKey(). Interestingly, introducing partitions clearly revealed where lazy evaluations were happening for certain RDDs (e.g. partitioning kNearestNeighbors moved evaluation from the “prediction” stage to the “k-nearest distances” stage).

Another significant and unexpected performance improvement was found by removing persist() calls from all RDDs. This was unexpected as we believed persisting reduces computation time, however persisting a single-use RDD seems to negatively impact performance as unnecessary writes to memory must be made by the process.

Further Optimizations

After focusing on decreasing computation time at a distributed level, we may consider reducing the amount of data required by these computations. Such reductions may be made to data or to the KNN algorithm.

- Converting categorical variables to integer enumerations reduces the amount of bytes required to represent an entry. As we are working with big data, minor reductions may make a large impact when this data is transferred between computers billions of times.
- Combining or reducing attributes used when computing distances may be used to optimize the KNN algorithm, potentially at the cost of accurate predictions. For instance, generalizing weather from numeric to type (“pleasant” or “brisk”) may bring cheaper distance calculations while still providing accurate predictions.