Predictive Analytics for connected car dataset

Summary

The aim of this project is to perform an analysis of a real world dataset generated by cars transmitting a number of events associated to their tracking, suggest an strategy to address trip destination prediction and implement the solution.

Dataset summary

The dataset is comprised of 600000 rows with the following structure:

- **deviceId:** This is a unique identifier for the car transmitting the data.
- **eventId:** This variable is not required for the analysis below.
- **eventTime:** This is the the unix timestamp for when the event was trans- mitted.
- **lat:** This is the latitude of the vehicle at the time of transmission.
- **lon:** This is the longitude of the vehicle at the time of transmission.
- **eventType**: Categorical values which can take the following values:
  - trip-start: Transmitted when the car starts a trip.
  - trip-end: Transmitted when the car ends a trip.
  - location
  - status-update
  - mil-update

Predictive analytics proposed

With the dataset as provided, one possible predictive analytics would be to estimate the destination of a car for a date and starting point given.

Solution implementation

The implementation of a solution which could answer the question of where a car is heading depending on the starting point could be addressed in many different ways, but the election of one or another strategy will be strongly dependent on the quality and significance of the data contained in the data set. In the same manner, the accuracy and generalization of the model will be strongly affected by the right feature selection, amount and spread of the data.

To be able to make these decisions, we need to get a more concise idea of the data. To achieve this, the first part of our pipeline will be dataset cleansing. After that, we will be in conditions of doing further exploration of the data to evaluate some of the aspects commented below, in order to decide which approach we will take.

The code explained by the present document can be found in the jupyter notebooks of the project. All the insights and solution implementation are contained there. Although reading this document is a good practice for a better understanding of the project and the analytical process followed to come to a solution, the notebooks are self-explanatory enough if followed in the order below:

1. data-cleansing.ipynb
2. features-preparation.ipynb

3. random-forest-model.ipynb
4. k-nearest-model.ipynb

The solution consists of the following deliverables:

- Documentation with analysis of the problem and explanation of the implementation and analytical process.
    - *predictive-analytics-connected-car.pdf*
- Jupyter notebooks with the code for cleaning, analyzing the data and generating the predictive models.
- Two files containing predictive models:
    - *random_forest_model.pkl*
    - *k_nearest_model.pkl*
- Script in python to evaluate time and start position input and predict end position using the generated models:
    - *predict-destination.py*

Dataset cleansing

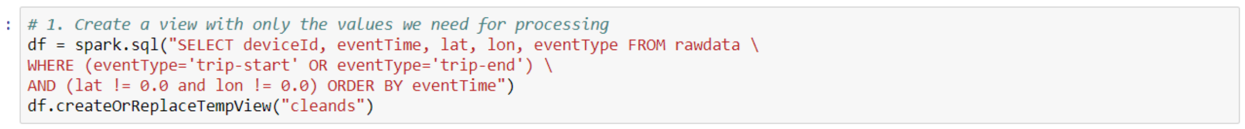
This stage consists mainly of getting rid of non needed data and perform all the heavy transformations we will need to feed our models later. Also, we will get a better idea how does our dataset look like, and take the first intuitions about the semantic of the data.

The operations of this stage can be found in the Jupyter notebook ***data-cleansing.ipynb*** of the project.

Identify the data we need

Since we want to predict the destination position from an origin position, the semantic entity which describes this in our problem domain would be a trip. So the dependent variables will be only those contained in the trip-end and trip-start events. Fors this reason, first step will be to delete unnecessary events.

Remove unnecessary events:



```
# 1. Create a view with only the values we need for processing
df = spark.sql("SELECT deviceId, eventTime, lat, lon, eventType FROM rawdata \
WHERE (eventType='trip-start' OR eventType='trip-end') \
AND (lat != 0.0 and lon != 0.0) ORDER BY eventTime")
df.createOrReplaceTempView("cleands")
```

This operation leaves us with 1344 rows. So with just a simple operation we dropped 97,5 % of the data.

Remove trip-start and trip-end non valid rows:

As we commented before, a trip is defined by a trip-start event followed by a trip-end event. However, we know that those events could be incomplete. That is, cars can be transmitting trip-start events without its corresponding trip-end afterwards. We need to get rid of them and keep only complete trip-start / trip-end events.

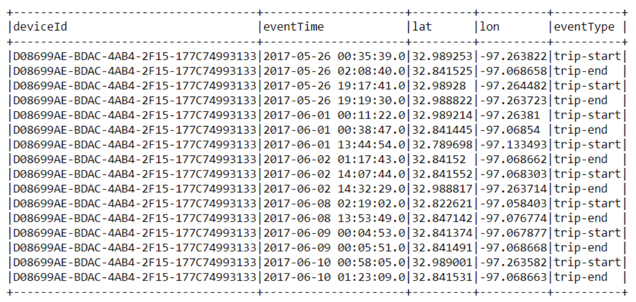
However, transformations to achieve this are tricky. We will address it by generating two new columns: *previousEvent* and *nextEvent*. We must take into account that trips are associated to a certain device, so we must apply the transformation **partitioning by *deviceId***:

```
# Shift values of eventType to add previous and next event
allDevices = spark.sql("SELECT *, LAG(eventType, 1) OVER (PARTITION BY deviceId ORDER BY eventTime) AS previousEvent, \
LEAD(eventType, 1) OVER (PARTITION BY deviceId ORDER BY eventTime) AS nextEvent FROM cleands")
allDevices.createOrReplaceTempView("alldevices")
allDevices.show(20)
```

Then, by looking at the actual event, the previous and the next one, we can decide wether a row must be or not deleted. trip-start with no trip-end after it, or those trip-end without a preceding trip-start are removed from the dataset.

```
# Remove non valid trip-start and trip-end (those without its corresponding pair)
allDevices = spark.sql("SELECT deviceId, eventTime, lat, lon, eventType FROM alldevices \
WHERE NOT ((eventType='trip-end' AND previousEvent='trip-end') \
OR (eventType='trip-start' AND nextEvent='trip-start'))")
allDevices.createOrReplaceTempView("alldevices")
allDevices.show(20, False)
```

That leaves us with only trip-start followed by trip-event rows:



```
+------------------------------------+---------------------+---------+----------+----------+
|deviceId                            |eventTime            |lat      |lon       |eventType |
+------------------------------------+---------------------+---------+----------+----------+
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-05-26 00:35:39.0|32.989253|-97.263822|trip-start|
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-05-26 02:08:40.0|32.841525|-97.068658|trip-end  |
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-05-26 19:17:41.0|32.98928 |-97.264482|trip-start|
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-05-26 19:19:30.0|32.988822|-97.263723|trip-end  |
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-06-01 00:11:22.0|32.989214|-97.26381 |trip-start|
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-06-01 00:38:47.0|32.841445|-97.06854 |trip-end  |
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-06-01 13:44:54.0|32.789698|-97.133493|trip-start|
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-06-02 01:17:43.0|32.84152 |-97.068662|trip-end  |
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-06-02 14:07:44.0|32.841552|-97.068303|trip-start|
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-06-02 14:32:29.0|32.988817|-97.263714|trip-end  |
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-06-08 02:19:02.0|32.822621|-97.058403|trip-start|
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-06-08 13:53:49.0|32.847142|-97.076774|trip-end  |
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-06-09 00:04:53.0|32.841374|-97.067877|trip-start|
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-06-09 00:05:51.0|32.841491|-97.068668|trip-end  |
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-06-10 00:58:05.0|32.989001|-97.263582|trip-start|
|D08699AE-BDAC-4AB4-2F15-177C74993133|2017-06-10 01:23:09.0|32.841531|-97.068663|trip-end  |
+------------------------------------+---------------------+---------+----------+----------+
```

Trip features merged in a row

For our later analysis and data processing we need the dependent and independent variables required for the model to be put together in a row, instead of spreaded in two (trip-start / trip-end).

To achieve this again we make use of SQL window functions (supported by Spark 2) and partition by deviceId. After that, we can get rid of the deviceId column, since we won't use it any more:

```
# Move start and end values of a trip to a single row. We won't need deviceId any more
allDevices = spark.sql("SELECT eventTime AS eventTimeStart, \
LEAD(eventTime, 1) OVER (PARTITION BY deviceId ORDER BY eventTime) AS eventTimeEnd, lat AS latStart, lon AS lonStart, \
LEAD(lat, 1) OVER (PARTITION BY deviceId ORDER BY eventTime) AS latEnd, \
LEAD(lon, 1) OVER (PARTITION BY deviceId ORDER BY eventTime) AS lonEnd, \
eventType FROM alldevices")
allDevices.createOrReplaceTempView("alldevices")
allDevices = spark.sql("SELECT eventTimeStart, eventTimeEnd, latStart, lonStart, latEnd, lonEnd \
FROM alldevices WHERE eventType='trip-start'")
allDevices.createOrReplaceTempView("alldevices")
allDevices.show(20, False)
```

That produces a result of 3154 rows, with the following structure

```
+--------------------+--------------------+----------+---------+--------+---------+
|eventTimeStart      |eventTimeEnd        |latStart  |lonStart |latEnd  |lonEnd   |
+--------------------+--------------------+----------+---------+--------+---------+
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
|2017-03-22 16:50:25.0|2017-03-22 17:14:26.0|38.7735368|-9.168737|38.76522|-9.098054|
+--------------------+--------------------+----------+---------+--------+---------+
```

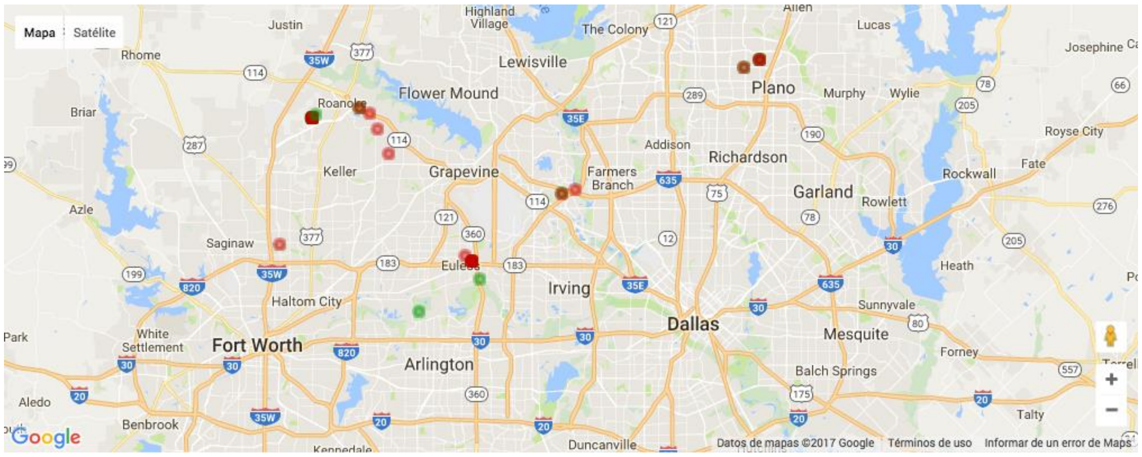
However, we observe that there are many duplicate rows, so we drop them. After removal, we have 1537 rows left.

Visualization

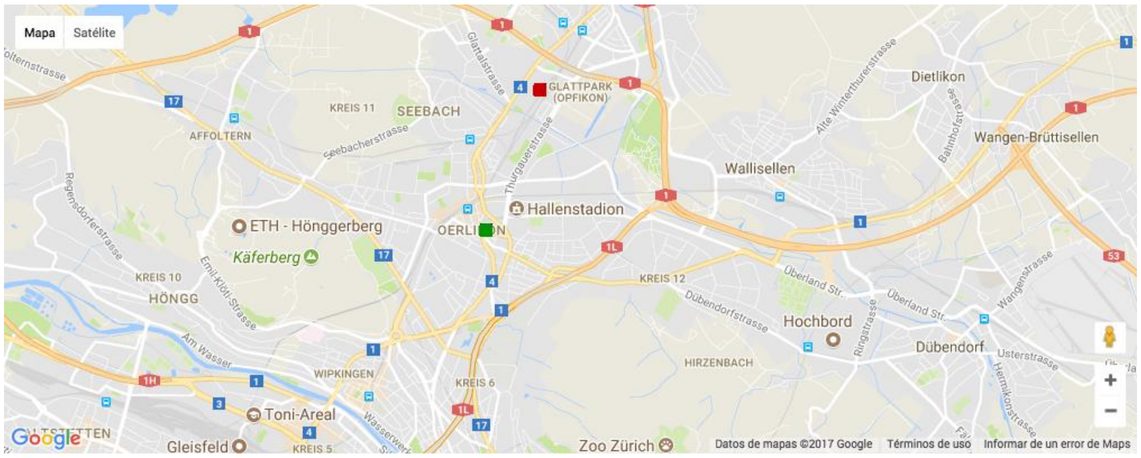
When working with geographic information data, is always a good practice to visualize the data over a map.

By doing this, we realize that our data is concentrated over 3 different areas of interest:

1. Dallas area.



2. Zurich area.



3. Lisbon area.

Green dots are trip-start points, meanwhile red dots represent trip-end.

There are some interesting intuitions that emerge from the visualization of the maps. As is to be expected, these are round trips, and in most of the cases the cars do the same trip for a given starting point. This can be observed specially in the area of Zurich, where all the trip-start are concentrated in the same area, as well as the trip-end. The same thing happens in Lisbon, but the number of samples from that area is almost negligible.

Features preparation

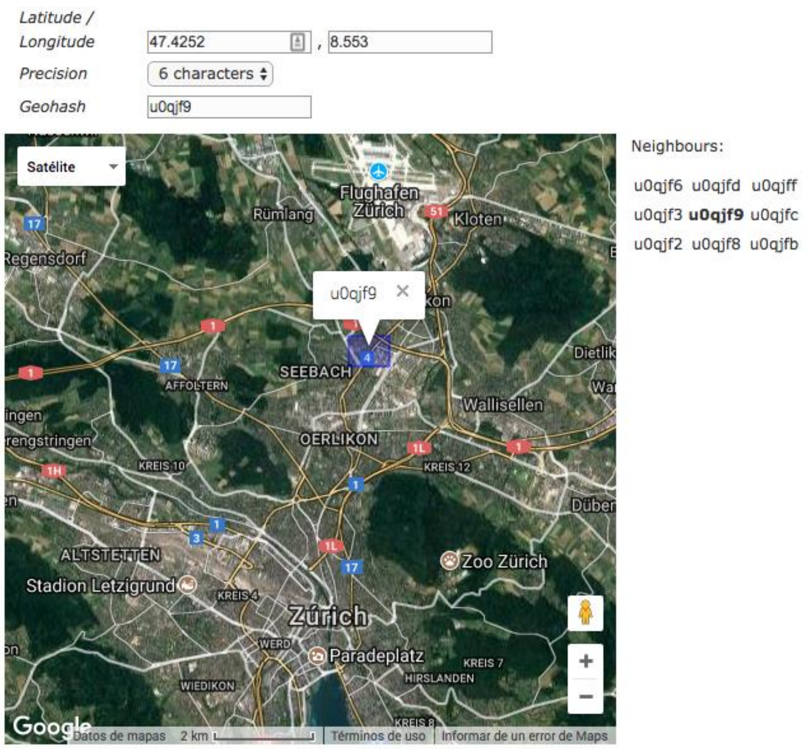
Prior to dump the data into our model, we still need to do some transformations. In this stage is were we typically perform normalization of the data, merge some features, or grow dimensionality.

The process of feature selection has many back and forths, so we will normalize and expand our features, and later in the model tuning, we will check which of those perform better. The code for this process can be found

Given, a certain granularity in location (geohash length g), granularity in time (bins per day b) and a chosen wideness (w) of the neighbourhood we want to look at, the aggregated data in the end should have the following columns:

"geohash"

Geohash with length g (categorical feature). This value can be used to bin latitude and longitude values. By enconding and decoding after, the positions get normalized to a centroid, which radius will depend on the precision we chose.



Latitude / Longitude 47.4252 , 8.553

Precision 6 characters

Geohash u0qjf9

Neighbours:

u0qjf6	u0qjfd	u0qjff
u0qjf3	**u0qjf9**	u0qjfc
u0qjf2	u0qjf8	u0qjfb

Depending on the length of the geohash, the cell will cover a bigger or smaller area. Cell dimensions vary with latitude. For example, at equator an 8 characters geohash covers an area of 38.2m x 9m, and a geohash of 6 will be 1.2km x 609.4m. Adjusting this length allows us to play with positions density if the model was too biased by location features.

"time_cat"

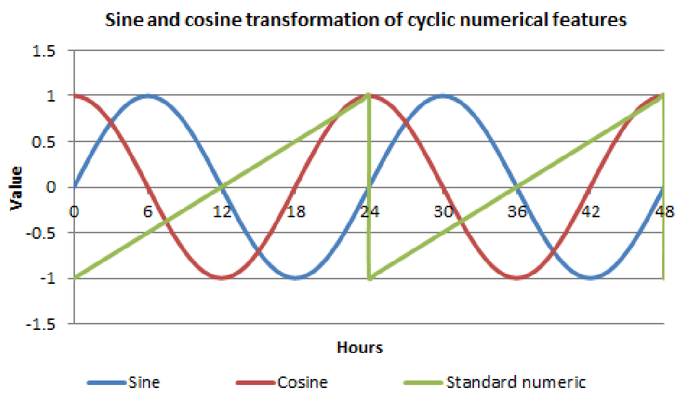
Time of the day as a categorical feature. If b=24b=24 (one bin for every hour), then "time_cat" for a pickup at 14:20:00 should be the string "14:00". If b=96b=96 (one bin for every quarter of an hour), then "time_cat" for a pickup at 14:20:00 should be the string '14:15'.

"time_num"

Time of the day as a (binned!) floating point number between 0 and 1, where the center of the bin is converted to a floating point number between 0 and 1. So if b=24b=24 , then "time_num" for a pickup at 14:20:00 should be 14.5/24=0.604214.5/24=0.6042 . If b=96b=96 , it should translate to 14.375/24=0.599014.375/24=0.5990 .

"time_cos"

The binned "time_num" variable converted to a cosine version so that time nicely 'loops' rather than going saw-like when it traverses midnight. See the figure below. This transformation doesn't have any magic powers, but it can make it easier for a model to find the right patterns. "time_cos" = cos(time_num·2π)cos⁡(time_num·2π) . So for 24 bins, 14:20:00 would translate to cos(0.6042·2π)=−0.7932cos⁡(0.6042·2π)=−0.7932 .



"time_sin"

Same thing as 4) but then with sine. So, "time_sin" = sin(time_num·2π)sin⁡(time_num·2π) . For 24 bins per day, 14:20:00 would translate to sin(0.6042·2π)=−0.6089sin⁡(0.6042·2π)=−0.6089 .

"day_cat"

Day of the week as a categorical feature: "Monday", "Tuesday", etc.

"day_num"

Day of the week as a numerical feature going from 0 (Monday morning, start of the week) to 1 (Sunday night), European style. With 24 bins, Tuesday afternoon 14:20:00 would translate to (1+14.524)/7=0.2292(1+14.524)/7=0.2292 .

"day_cos"

Binned "day_num" variable converted to a cosine version. "day_cos" = cos(day_num·2π)cos⁡(day_num·2π)

"day_sin"

Binned "day_num"variable converted to a sine version. "day_sin" = sin(day_num·2π)sin⁡(day_num·2π)

"weekend"

0 if weekday, 1 if weekend (Saturday/Sunday)

Location features

Latitude and longitude of the center of the geohash the record was bucketed in.

“x_start”, “y_start”, “z_start”

Lat long coordinates have the problem that they are 2 features that represent a three dimensional space. This means that the long coordinate goes all around, which means the two most extreme values are actually very close together. The solution is to project them to an Euclidean 3d space: x, y and z coordinates. This means close points in these 3 dimensions are also close in reality. The transformation is done this way:

x = cos(lat) * cos(lon)

y = cos(lat) * sin(lon),

z = sin(lat)

“location_start, location_end”

Latitude and longitude of the center of the geohash the record was bucketed in.

Predictive models

Machine learning can be summarized as learning a function (f) that maps input variables (X) to output variables (Y).

Y = f(x)

An algorithm learns this target mapping function from training data.

The form of the function is unknown, so our job as machine learning practitioners is to evaluate different machine learning algorithms and see which is better at approximating the underlying function.

Different algorithms make different assumptions or biases about the form of the function and how it can be learned.

The first thing we must consider is whether we can make some assumptions in our data or not. If we detect some linearity in our data or we can fit it into a well known function or distribution, then we will take a parametric algorithm. Some example of this would be:

- Logistic Regression
- Linear Discriminant Analysis
- Perceptron
- Naive Bayes
- Simple Neural Networks

However, sometimes we can’t make any of this asumptions, then we use nonparametric machine learning algorithms. These algorithms do not make assupmtions about the form of the mapping function. Examples of these are:

- K-Nearest Neighbors
- Decision Trees / Random Forest
- Support Vector Machine

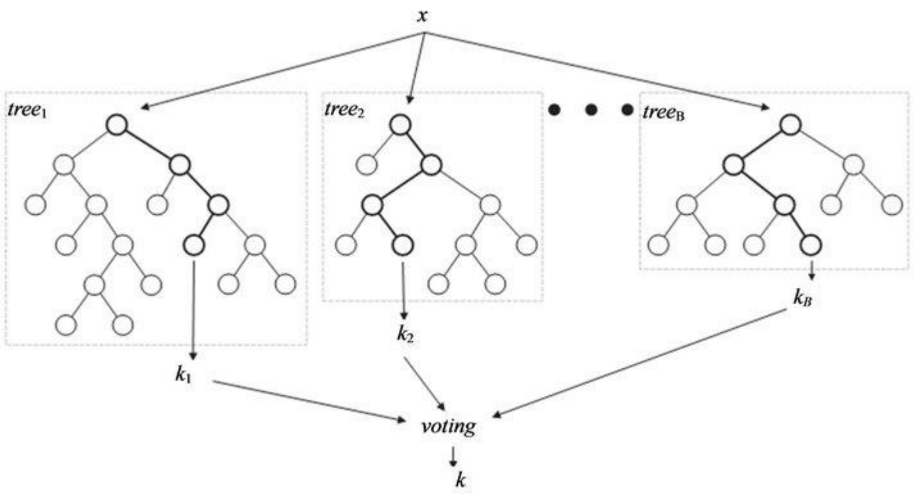
k-nearest neighbors algorithm makes predictions based on the k most similar training patterns for a new data instance. The method does not assume anything about the form of the mapping function other than patterns that are close are likely have a similar output variable.

In our specific problem, we bust bear with variables which don't belong to a linear space, such as latitude and longitude, so it makes more sense to use nonparametric algorithms, such as those used for recommender system (neigborhood based algorithms). Since we don't have widely spread data around the geographical areas, we must assume that our algorithms won't generalise well.

Random Forest

The random forest is an ensemble approach that can also be thought of as a form of nearest neighbor predictor.

Ensembles are a divide-and-conquer approach used to improve performance. The main principle behind ensemble methods is that a group of "weak learners" can come together to form a "strong learner".



Pros:

Random forest runtimes are quite fast, and they are able to deal with unbalanced and missing data. They are able to predict data with nonlinearity.

Cons:

Random Forest weaknesses are that when used for regression they cannot predict beyond the range in the training data, and that they may over-fit data sets that are particularly noisy.

Implementation:

The implementation of the model is in the notebook ***"random-forest-model.ipynb"***

From the previous steps, we got a dataset with 19 features. However, many of them are directly correlated, we'll just take some of them. One of the trickiest parts of model tuning is a good feature selection.

Although we have many features, they can be separated into two main groups, the ones related to the time, and the one related to the positioning.

The approach to address model tuning will be:

1. Pick some features, ones related to time and ones to positioning.

2. Fit the model.
3. Check accuracy.
4. Check feature importance.
5. Play with features density and selection and repeat the process.

To tune regressor hpyerparameters, we use Grid Search Cross Validation.

Grid Search CV implements an exhaustive search over specified parameter values for an estimator. Important members are fit, predict.

It iterates through a dictionary of hyper parameters and choose the combination that better fits the model.

Then we find that the best candidate settings are:

RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=1, max_features='auto', max_leaf_nodes=None, min_impurity_split=1e-07, min_samples_leaf=1, min_samples_split=5, min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1, oob_score=False, random_state=None, verbose=0, warm_start=False)

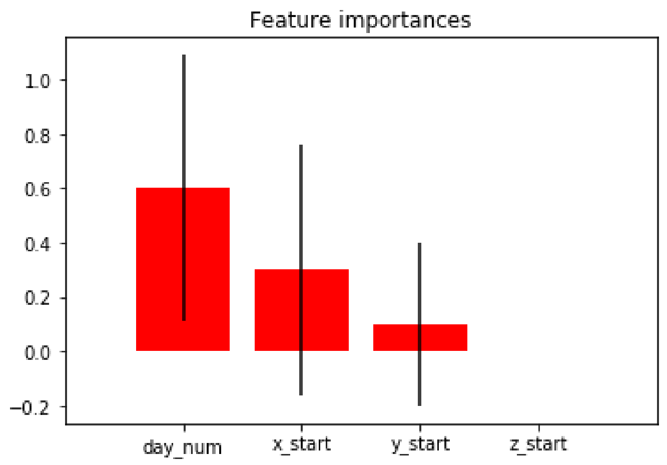
Results

We fit the model to the train set, and check scores. For this we use R-Squared and Root-mean-squared deviation. R-squared is a statistical measure of how close the data are to the fitted regression line. It ranges from 0 to 1, being 1 the best coefficient. RMSE is the square root of the mean square error. In other words the distance, on average, of a data point from the fitted line, measured along a vertical line.

We find that our model gets the following scores:

- R-Squared 0.999995
- RMSE: 0.023013

So we can tell that our model does it very good at fitting the data. However, we know that our training dataset is not well spread around positions, and as a consequence the model presents a big bias towards the time features. To fix this, we reduced the bins per day to 12, and used the x, y, z coordinates, which proved to perform better. Thus, the bias was reduced at it's best as shown this chart:



K-Nearest Neighbor

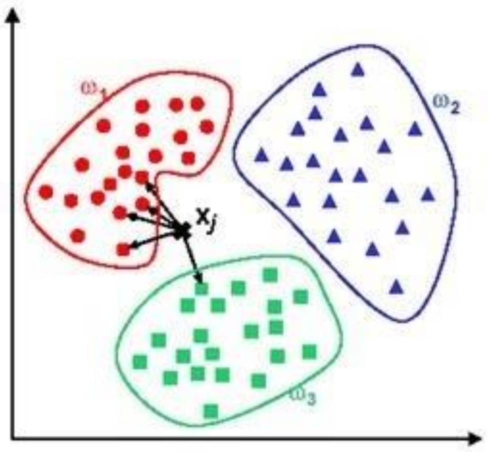
KNN is an non parametric lazy learning algorithm. As explained before, this means that it does not make any assumptions on the underlying data distribution (eg gaussian mixtures, linearly separable etc).

Before using KNN, let us revisit some of the assumptions in KNN.

KNN assumes that the data is in a feature space. More exactly, the data points are in a metric space. The data can be scalars or possibly even multidimensional vectors. Since the points are in feature space, they have a notion of distance – This need not necessarily be Euclidean distance although it is the one commonly used. In our case, we transformed latitude and longitude into 3D coordinates, so we are ready to meet that requirement.

Each of the training data consists of a set of vectors and class label associated with each vector. In the simplest case , it will be either + or – (for positive or negative classes). But KNN , can work equally well with arbitrary number of classes.

We are also given a single number "k" . This number decides how many neighbors (where neighbors is defined based on the distance metric) influence the classification. This is usually a odd number if the number of classes is 2. If k=1 , then the algorithm is simply called the nearest neighbor algorithm.



KNN are typically used for recommender systems, and allow us to make predictions based on the distance of their neighbors.

Pros:

- Very easy to understand and implement.
- Does not assume any probability distributions on the input data. This can come in handy for inputs where the probability distribution is unknown and is therefore robust.
- Can quickly respond to changes in input. k-NN employs lazy learning, which generalizes during testing--this allows it to change during real-time use.

Cons:

- Sensitive to localized data. Since k-NN gets all of its information from the input's neighbors, localized anomalies affect outcomes significantly, rather than for an algorithm that uses a generalized view of the data.
- Computation time. Lazy learning requires that most of k-NN's computation be done during testing, rather than during training. This can be an issue for large datasets.
- Dimensions. In the case of many dimensions, inputs can commonly be "close" to many data points. This reduces the effectiveness of k-NN, since the algorithm relies on a correlation between closeness and similarity. One workaround for this issue is dimension reduction, which reduces the number of working variable dimensions (but can lose variable trends in the process).

Implementation:

The implementation of the model is in the notebook ***"k-nearest-model.ipynb"***

Our main concern here will be to use a reduce dimension dataset and to make sure that the data points are in a metric space. Since we already took care of this in the feature selection step of the random forest solution, we will use the same training set. Also, this will help us to better compare the models.

So for this model our dependent variables will be again *day_num*, *x_start*, *y_start* and *z_start*.

And, as in the Random Forest, we will tune hpyerparameters using Grid Search Cross Validation.

Results

We find that our model gets the following scores:

- R-Squared 0.99999609
- RMSE: 0.020617

If we compare them with the scores we get with random forest, they are almost the same. However, a fast check against ground truth values will show as that the model does not behave exactly the same.

Conclusions

We were able to produce two models with a high score of accuracy for the model both with a Random Forest approach and a K-Nearest Neighbors.

These two algorithms stand out for being able to make predictions over a data set with no assumptions in their distribution, as it is our case. For a given time and start position input we are able to predict the destination very precisely. However, in both cases we lack of a good generalization, in part caused by the algorithms nature and in part by our dataset characteristics.

Both algorithms base their prediction in an inferred distance function (or graph) to feature similar inputs. For this reason, in our case we are able to provide good results if the latitude and longitude are very close to previously observed start positions, and will produce meaningless results as we move away from them.

The effect explained before is specially strong in Random Forest model. This happens because our positions samples come from three widely separated areas, which affect the overall computation. So, having a lot of positions in Zurich will introduce noise to our predictions in Dallas. On the other hand, K-Nearest Neighbors makes it better to divide the decisions in clusters. And so, even when we also observe a bad generalization and poor performance as we move away from the clusters, this doesn't introduce that noise to each other (i.e. Zurich data doesn't affect that much to Dallas).

Ways to improve

One of the most eye-catching things after addressing the problem, is that our data may not be always as good as we think. And depending on the exact information we need, a promising dataset of 600000 records may be reduced to less than 2000. Also, density balance between features is important if we want to avoid biasing.

So one way of improving would be getting more data and try to complete the data in every feature space. This means that if we are tracking hours of the day, it would be useful to try to cover all the hours. And if we want to introduce the month as a dependant variable, we need to get data from the 12 months of the year.

Also, we have seen that our problem would be better addressed by areas, since there is no way to find a correlation between the predicted destinations for locations in Zurich, and those in other geographical areas.